

Programación orientada a objetos y clases

Veremos cómo implementamos la realidad a nuestras aplicaciones, para ello utilizaremos la programación orientada a objetos, también llamada POO.

Índice

- 1 - ¿Qué es la programación orientada a objetos?
- 2 - Crear una clase
 - 2.1 - Crear una instancia de una clase
- 3 - Constructores
- 4 - Herencia

1 - ¿Qué es la programación orientada a objetos?

Es un paradigma que convierte objetos reales a entidades de código llamadas clases. Esto puede parecer muy complejo, pero no lo es para nada.

Imaginémonos un objeto del mundo real como puede ser una moto



Como podemos observar en la imagen, disponemos de dos entidades. `Piloto` y `Moto`.

Dependerá de la lógica de negocio saber si es la moto la que contiene al piloto, o es el piloto el que contiene la moto. Para nuestro ejemplo, la moto contendrá al piloto.

Cuando convertimos la moto a un objeto, debemos pensar en sus características, como marca, modelo, número de ruedas, cilindrada, etc. Estas características serán nuestras propiedades.

Además de eso, en la moto realizas acciones, como son arrancar, acelerar, frenar, cambiar de marcha, todas estas acciones serán los métodos.

Por lo tanto, una clase es un tipo específico que sirve para crear objetos.

Recordemos que existen los tipos primitivos como `int`, `decimal`, etc. Las clases son un tipo más, pero personalizadas a nuestro gusto.

2 - Crear una clase

Para crear una clase debemos indicar la palabra registrada `class` y posteriormente el nombre que queremos utilizar. Como en todos los bloques de código, definiremos cuando empieza y cuando acaba utilizando llaves `{ }`, dentro del bloque de código, introduciremos nuestras propiedades y nuestros métodos.

```
class Moto {  
  
    public decimal VelocidadMaxima { get; set; }  
  
    public int NumeroRuedas { get; set; }  
  
    public Motorista Piloto { get; set; }  
  
    public Acelerar(){ //Código aquí }  
  
    public Arrancar(){ //Código aquí }  
  
}
```

Como vemos además utilizamos la palabra reservada `public` la cual es un modificador de acceso, que veremos más adelante.

2.1 - Crear una instancia de una clase

Un objeto es la representación en memoria de una clase y puedes crear tantos objetos de una clase como quieras, para ello utilizaremos la palabra reservada `new`.

```
Moto aprilia = new Moto();
```

Posteriormente le podemos dar valores a sus propiedades, así como llamar a sus métodos. Por supuesto, al disponer de `Piloto` dentro del tipo `Moto`, también le podemos dar valores

```
aprilia.VelocidadMaxima = 320;
```

```
aprilia.NumeroRuedas = 2;
```

```
aprilia.Acelerar();
```

```
aprilia.Piloto = new Motorista();
```

```
aprilia.Piloto.Nacionalidad = "ESP";
```

3 - Constructores

Como hemos observado, cuando creamos la instancia de la clase utilizamos `new Moto()` bien, esta sentencia lo que hace es llamar al constructor por defecto dentro de `System.Object`, pero claro, que pasa si no queremos utilizar el constructor por defecto, ya que el constructor por defecto nos inicializa todas las variables a null.

Para ello podemos crear un constructor personalizado por nosotros mismos, donde podremos asignar valores manualmente, como podemos observar en el ejemplo. Ahora cuando realizamos `Moto aprilia = new Moto();` nos creará por defecto una `VelocidadMaxima` de 320 y un `NumeroRuedas` de 2;

```
class Moto {  
  
    public decimal VelocidadMaxima { get; set; }  
  
    public int NumeroRuedas { get; set; }  
  
    public Motorista Piloto { get; set; }  
  
    public string Marca { get; set; }  
}
```

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

    public Moto(){

        VelocidadMaxima = 320;

        NumeroRuedas = 2;

    }

    public Acelerar(){ //Código aquí }

    public Arrancar(){ //Código aquí }

}
```

Pero ¿qué pasa si tenemos múltiples motos?

En nuestro ejemplo al tener los valores asignados en el constructor por defecto se nos crearían todas las motos con los mismos valores, esto quiere decir que en las siguientes sentencias

```
Moto aprilia = new Moto();
```

```
Moto Ducati = new Moto();
```

Ambas motos tendrán una `VelocidadMaxima` de 320 y un valor de 2 en `NumeroRuedas`, por lo que los valores podrían no ser correctos.

Para arreglar este problema, lo que hacemos es añadir un constructor que acepte valores por parámetro. Utilizando Sobrecarga de operadores podemos crear tantos constructores como deseemos.

```
class Moto {

    public decimal VelocidadMaxima { get; set; }

    public int NumeroRuedas { get; set; }

    public Motorista Piloto { get; set; }

    public string Marca { get; set; }

}
```

```

public string Modelo { get; set; }

public Moto(decimal velocidadMaxima, int NumeroRuedas){

    VelocidadMaxima = velocidadMaxima;

    NumeroRuedas = NumeroRuedas;

}

public Moto(string marca, string modelo){

    Marca = marca;

    modelo = modelo;

}

public Moto(){

    VelocidadMaxima = 320;

    NumeroRuedas = 2;

}

public Acelerar(){ //Código aquí }

public Arrancar(){ //Código aquí }

}

```

Como podemos comprobar, ambos constructores funcionan a la perfección

```
Moto aprilia = new Moto("Aprilia", "RX");
```

```
Moto motoSinMarca = new Moto(210,2);
```

4 - Herencia en programación

Otra característica muy importante dentro de la programación orientada a objetos es la herencia. La cual nos permite heredar información de una clase padre a su hijo.

En el ejemplo anterior de la moto, supongamos que tenemos también un coche, ambos son vehículos, por lo que ambos tienen características comunes, como pueden ser marca, modelo, número de ruedas, y otras diferentes como pueden ser la cilindrada para las motos o la tracción para los coches. El uso de la Herencia nos permite ahorrar multitud de líneas de código y complejidad dentro de nuestros programas.

```
class Vehiculo {  
  
    public decimal VelocidadMaxima { get; set; }  
  
    public int NumeroRuedas { get; set; }  
  
    public string Marca { get; set; }  
  
    public string Modelo { get; set; }  
  
    public Vehiculo(decimal velocidadMaxima, int  
NumeroRuedas){  
  
        VelocidadMaxima = velocidadMaxima;  
  
        NumeroRuedas = NumeroRuedas;  
  
    }  
  
    public Vehiculo(string marca, string modelo){  
  
        Marca = marca;  
  
        Modelo = modelo  
  
    }  
  
    public Acelerar(){ //Código aquí }  
  
    public Arrancar(){ //Código aquí }
```

```

}

class Moto : Vehiculo {

    public int Cilindrada { get; set; }

    public Moto(decimal velocidadMaxima, int NumeroRuedas,
        int cilindrada) : base(velocidadMaxima, NumeroRuedas){

        Cilindrada = cilindrada;

    }

    public Moto(string marca, string modelo, int cilindrada)
        : base(marca, modelo){

        Cilindrada = cilindrada;

    }

    public void HacerCaballito(){ //codigo }

}

class Coche : Vehiculo {

    public string Traccion { get; set; }

    public Coche(string marca, string modelo, string
        Traccion) : base(marca, modelo){ }

    public bool CerrarPuertas(){ }

}

```

Como podemos observar tanto `Coche` como `Moto` heredan, utilizando `:`, de la clase `Vehiculo`, esto quiere decir que desde ambas clases podemos acceder a los métodos y propiedades de la clase Vehiculo, además de a las suyas propias.

Otro detalle interesante es que como podemos observar en el constructor, desde la clase hijo se llama al constructor de la clase padre utilizando la palabra reservada `: Base()` y mandando los parámetros que ese constructor necesita.

Modificadores de acceso

En este tutorial veremos todo lo referente a los modificadores de acceso.

Índice

- 1 - Qué es un modificador de acceso
- 2 - Modificadores de acceso
 - public
 - private
 - internal
 - protected
 - protected internal
 - private protected

1 - Qué es un modificador de acceso

Un modificador de acceso es aquella cláusula de código que nos indica si podemos acceder o no a un bloque de código específico desde otra parte del programa.

Hay una gran variedad de modificadores de acceso, y estos son aplicados tanto a métodos, propiedades, o clases.

2 - Modificadores de acceso

public

Acceso no restringido que permite acceder a sus miembros desde cualquier parte del código al que se le hace referencia.

```
public class EjemploPublic
{
    public string PruebaAcceso { get; set; }
}
```



```

class Program
{
    static void Main(string[] args)
    {
        EjemploPublic ejemplo = new EjemploPublic();

        Console.WriteLine(ejemplo.PruebaAcceso); //Funciona
        correctamente
    }
}

```

Private

Permite acceder a los miembros exclusivamente desde la clase o struct que los contiene.

```

public class EjemploPrivate
{
    private string PruebaAcceso { get; set; }

    public EjemploPrivate() {
        PruebaAcceso = "funciona";//Funciona
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        EjemploPrivate ejemplo = new EjemploPrivate();
        Console.WriteLine(ejemplo.PruebaAcceso); //Da un error
    }
}

```

internal

Permite acceder desde el mismo proyecto o assembly pero no desde uno externo.

Por ejemplo, si tenemos una librería, podremos acceder a los elementos internal desde la propia librería, pero si referenciamos a esa librería desde otro proyecto no podremos acceder a ellos.

```

//Libreria externa (Distinto proyecto|Assembly)

public class EjemploInternalLibreria
{
    internal string PruebaAcceso { get; set; }
}

class EjemploImprimirInternal{
    public void Imprimir()
    {
        EjemploInternalLibreria ejemplo = new
        EjemploInternalLibreria();
        Console.WriteLine(ejemplo.PruebaAcceso); //Funciona
    }
}

```

```
//proyecto principal

class Program

{

    void Imprimir()

    {

        EjemploInternalLibreria ejemplo = new
        EjemploInternalLibreria();

        Console.WriteLine(ejemplo.PruebaAcceso); // Error. Al
        estar en otro proyecto

    }

}
```

protected

Podremos acceder a los elementos desde la misma clase, o desde una que deriva de ella.

```
class EjemploProtected

{

    protected string PruebaAcceso { get; set; }

}
```

```
class claseHerencia : EjemploProtected //Herencia, osea clase
hija.
```

```
{
```

```
    void Imprimir()
```

```
{
```

```
        Console.WriteLine(PruebaAcceso); //Accedemos sin
problemas ya que es una propiedad de la clase padre.
```

```
    }
```

```
}
```

```
class Program
```

```
{
```

```
    void Print()
```

```
{
```

```
        EjemploProtected ejemplo = new EjemploProtected();
```

```
        Console.WriteLine(ejemplo.PruebaAcceso); // Error. no
podemos acceder ya que esta clase no hereda de
EjemploProtected
```

```
    }
```

```
}
```

protected internal

Combina tanto protected como internal permitiendo acceder desde el mismo proyecto o assembly o de los tipos que lo derivan.

private protected

Finalmente combinamos private y protected lo que nos permitirá acceder desde la clase actual o desde las que derivan de ella. Lo que permite referenciar métodos y propiedades en clases de las cuales heredamos.

Manejo de excepciones

En el tutorial de hoy veremos, qué es una excepción y como trabajar con ellas.

Índice

- 1 - Qué es una excepción?
- 2 - Bloque try-catch
- 3 - Especificar una excepción
- 4 - Conclusión

1 - ¿Qué es una excepción?

Una excepción es un problema o error que aparece de una manera repentina en nuestro código mientras se está ejecutando.

Debemos controlar las excepciones, ya que si no lo hacemos causará que el programa deje de trabajar, lo que implica que su funcionamiento se detiene.

La mejor forma de evitar excepciones es controlando que estas no puedan pasar. Por ejemplo, una excepción puede ocurrir cuando intentamos leer un fichero y este fichero no existe. Por lo que para evitar la excepción tendríamos que comprobar que el fichero existe, y si existe, leer, en caso contrario no leer el fichero.

Pero desafortunadamente no podemos poner un filtro o una comprobación en cada aspecto del programa. Así que lo que tenemos que hacer es un bloque `try-catch`.

2 - Bloque try-catch

C# nos permite controlar las excepciones utilizando un bloque `try-catch`. Su funcionamiento es muy básico, ponemos código dentro del bloque `try` y si salta una excepción se ejecutará el `catch`, el cual contiene otro bloque de código. Así evitamos que el programa deje de funcionar.

Por ejemplo, no podemos dividir un número por 0 lo que hace que salte una excepción.

```

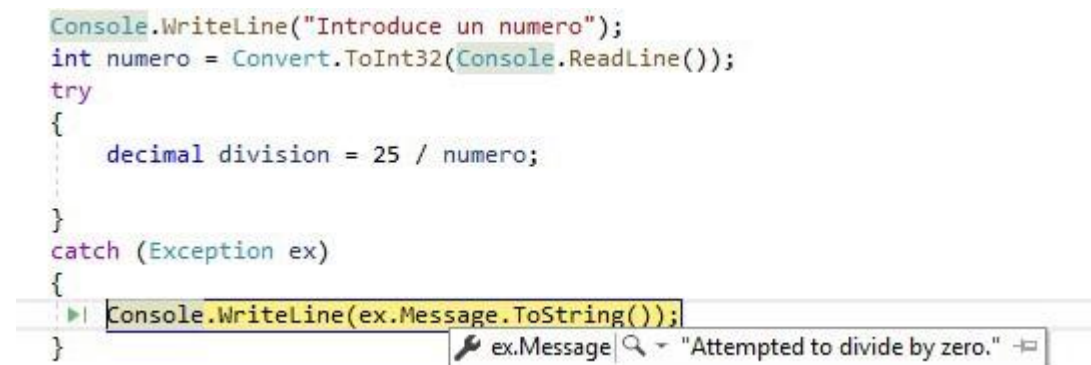
try
{
    int numero = Convert.ToInt32(Console.ReadLine());

    decimal division = 25 / numero;
}

catch (Exception ex)
{
    Console.WriteLine(ex.Message.ToString());
}

```

Como vemos el bloque `catch` nos da acceso a la variable `Exception`, la cual contiene toda la información sobre la excepción, como son el mensaje o el 'stacktrace' que nos indica donde ha ocurrido esa excepción.



```

Console.WriteLine("Introduce un numero");
int numero = Convert.ToInt32(Console.ReadLine());
try
{
    decimal division = 25 / numero;
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message.ToString());
}

```

ex.Message "Attempted to divide by zero."

Otra opción muy común dentro de un `catch` es utilizar la cláusula `throw` la cual enviará la excepción al bloque `catch` padre.

```

try
{
    decimal division = 25 / numero;
}

catch (Exception ex)
{
    throw;
}

```

Lo que quiere decir que si el método que hace saltar la excepción está controlado en niveles superiores ejecutara el catch de los niveles superiores. En caso de no estarlo, pararíamos la ejecución del programa.

3 - Especificar una excepción

Como hemos visto esa excepción es muy concreta, ya que no se puede dividir un número por cero.

Dentro de un `try-catch` podemos indicar tantos `catch` como queramos, siempre y cuando el tipo de dato (`Exception`) sea diferente, como vimos con los constructores, aquí también se utiliza sobrecarga de operadores.

Para nuestro ejemplo, otra posibilidad es que el número que introducimos sea una palabra y no un número, si queremos comprobar esa excepción en concreto, debemos especificarla en el catch como vemos a continuación.

```

Console.WriteLine("Introduce un numero");

try
{
    int numero = Convert.ToInt32(Console.ReadLine());

    decimal division = 25 / numero;
}

```

```
catch (FormatException ex)
{
    Console.WriteLine("El valor introducido no era un numero entero");
}

catch (DivideByZeroException ex)
{
    Console.WriteLine("No es posible dividir por 0");
}

catch (Exception ex)
{
    throw;
}
```

Como podemos observar si el valor introducido no es un número, la ejecución del programa continuará en el `catch(FormatException)`, si ese valor es 0 continuara en `catch(DivideByZeroException)` y si la excepción es cualquier otra saltara al `catch(Exception)` el cual es el bloque por defecto.

4 - Conclusión

El manejo de excepciones es algo primordial en el día a día laboral, por eso hay que tenerlo muy en cuenta cuando realizamos nuestros proyectos personales, o mientras estudiamos, ya que le dará una robustez al sistema ante fallos y problemas catastróficos.

Parámetros por valor y referencia

Índice

- 1 - Parámetros por valor
- 2 - Parámetros por referencia
 - 2.1 - Palabra reservada ref
 - 2.2 - Palabra reservada out
 - 2.3 - Palabra reservada In
- 3 - El caso especial de objetos
- 4 - Datos finales

1 - Parámetros por valor

Cuando tenemos un método le pasamos un parámetro, en verdad no estamos mandando ese parámetro, estamos mandando una copia de este, lo que significa, que fuera del método, el valor de la variable seguirá siendo el mismo. Incluso si modificamos su valor internamente.

```
static void Main(string[] args)
{
    int valorActual = 10;

    Actualizar(valorActual);

    Console.WriteLine($"el valor es: {valorActual}");//imprime
10

    Console.ReadKey();
}
```

```
//Comportamiento normal.  
  
public static void Actualizar(int valor)  
{  
  
    valor += 5;  
  
    Console.WriteLine($"el valor es: {valor}"); //imprime 15  
}
```

`valorActual` sigue valiendo 10 después de finalizar el método `Actualizar()` ya que el valor únicamente se modifica para ese método.

2 - Parámetros por referencia

Puede llegar el momento de que necesitemos actualizar el valor de los datos. Para ello pasaremos los parámetros por referencia, y lo haremos utilizando las palabras reservadas `ref` y `out`. Cuando utilizamos `ref` y `out` no almacenamos en la variable el valor de esta, sino que almacenamos la posición de memoria a la que apunta por lo que, si modificamos el valor, también cambiamos el valor de la original.

2.1 - Palabra reservada ref

Utilizar `ref` significa que vas a pasar el valor por referencia a un método, lo que implica que el valor va a cambiar dentro del método. Además, si deseamos utilizar `ref` la variable tiene que estar inicializada con anterioridad.

Finalmente debemos indicar tanto en el método como en la llamada al método que vamos a pasar un valor por referencia.

```
static void Main(string[] args)  
{  
  
    int valorActual = 10;  
  
    ActualizarRef(ref valorActual); //pasar por referencia
```

```

        Console.WriteLine($"el valor es: {valorActual}"); //
imprime 12

        Console.ReadKey();

    }

//Actualizar por referencia

public static void ActualizarRef(ref int valor)

{

    valor += 2;

}

```

2.2 - Palabra reservada out

Utilizar `out` significa que la asignación del valor de esa variable está dentro del método al que se llama. No es necesario inicializar el valor de la variable, aunque sí debemos instanciarla.

```

static void Main(string[] args)

{

    int valorActual;

    ActualizarOut(out valorActual);

        Console.WriteLine($"el valor es: {valorActual}"); //
imprime 13

    Console.ReadKey();

}

```

```
//Crear utilizando out

public static void ActualizarOut(out int valor)

{

    valor = 13;

}
```

2.3 - Palabra reservada In

La palabra clave `in` evita que podamos modificar el valor de la variable dentro del método, por lo que el valor siempre será el que hayamos pasado previamente.

```
public static void ActualizarIn(in int valor)

{

    valor += 5; //Da Error

    Console.WriteLine($"el valor es: {valor}");

}
```

3 - El caso especial de objetos

Cuando pasamos un Objeto/Tipo creado por nosotros mismos, este siempre se pasa por referencia. Con lo que, si lo actualizamos dentro de la función, se actualizará fuera de ella.

```
class Program
{
    static void Main(string[] args) {
        ObjEjemplo ejemploValor = new ObjEjemplo(10);
        ActualizarObj(ejemploValor);
        Console.WriteLine($"el valor es: {ejemploValor.Entero}");
        // imprime 25
        Console.ReadKey();
    }

    public static void ActualizarObj(ObjEjemplo obj)
    {
        obj.Entero = 25;
    }
}

public class ObjEjemplo
{
    public int Entero { get; set; }

    public ObjEjemplo(int entero)
    {
        Entero = entero;
    }
}
```

4 - Datos finales

- Debemos utilizar ref si queremos cambiar el valor de la variable, en cambio si queremos crear una nueva variable debemos utilizar out. In cuando queramos que ese valor no se pueda modificar.
- No podemos utilizar in, ref, out en métodos asíncronos o iteradores.

Polimorfismo en programación orientada a objetos

Índex

- 1- Qué es el polimorfismo en programación orientada a objetos
- 2 – Polimorfismo estático en C#
 - 2.1 – Utilizando la palabra clave new
 - 2.2- Utilizando las palabras clave virtual y override

1- Qué es el polimorfismo en programación orientada a objetos

Denominamos polimorfismo al mecanismo que nos permite tener un método en una clase padre como vimos en la herencia y sobrescribirlo en la clase hija.

Esto quiere decir que tendremos el mismo método en ambas clases, pero en la clase hija realizará diferentes acciones. Por lo que el polimorfismo es también denominado sobreescritura de métodos.

Hay dos formas de polimorfismo

- A. En tiempo de ejecución, que tiene que ver con las interfaces
- B. La segunda llamada polimorfismo estático, la cual determina que método se va a ejecutar durante la compilación.

2 – Polimorfismo estático en C#

2.1 – Utilizando la palabra clave new

Para indicar que queremos sobrescribir un método debemos hacerlo utilizando la palabra reservada `new` antes del nombre del método en la clase hijo. La cual nos crea una nueva versión del método, únicamente para esa clase.

Con ello lo que conseguimos es permitir que llamando “al mismo método” se realicen acciones diferentes.

```
class Vehiculo
{
    public decimal VelocidadMaxima { get; set; }

    public int NumeroRuedas { get; set; }

    public string Marca { get; set; }

    public string Modelo { get; set; }

    public Vehiculo(string marca, string modelo)
    {
        Marca = marca;

        Modelo = modelo;
    }

    public void Acelerar()
    {
        Console.WriteLine("Acelerar vehículo");

        //Pisar el acelerador
    }
}

class Moto : Vehiculo
{
    public int Cilindrada { get; set; }

    public Moto(string marca, string modelo, int cilindrada) :
base(marca, modelo)
```



```

    {

        Cilindrada = cilindrada;

    }

    public new void Acelerar()

    {

        //Girar el puño

        Console.WriteLine("Acelerar Moto");

    }

}

```

Como podemos observar el método `Acelerar` de la clase `Moto` contiene la palabra reservada `new`, ello causará que si llamamos a `Moto.Acelerar()` se ejecutará el nuevo código, Mientras que si tenemos otra clase, como `Vehiculo.Acelerar()`, o `Coche.Acelerar()` si utilizamos los ejemplos de cuando vimos Herencia, se ejecutará el de la clase `Vehiculo`, ya que `Coche` no posee un método que sobrescriba al de la clase padre.

2.2- Utilizando las palabras clave virtual y override

En el mundo laboral, utilizar la forma con la palabra clave `new` no es muy común, ya que da la sensación de que no se va a sobrescribir y puede llegar a ser muy confuso.

Para evitar esta sensación se puede indicar en el método padre que este va a ser sobrescrito utilizando la palabra clave `virtual`

```

public virtual void Acelerar()

{

    Console.WriteLine("Acelerar vehículo");

    //Pisar el acelerador

}

```

Y en el método de la clase hija, se indica con la palabra `override` que va a sobrescribir al método padre.

Es importante recordar que no podemos utilizar la palabra clave `override` si el método padre no es `virtual`.

```
public override void Acelerar()
{
    //Girar el puño

    Console.WriteLine("Acelerar Moto");
}
```

Aquí podemos ver el ejemplo completo:

```
class Vehiculo
{
    public decimal VelocidadMaxima { get; set; }
    public int NumeroRuedas { get; set; }
    public string Marca { get; set; }
    public string Modelo { get; set; }

    public Vehiculo(string marca, string modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }

    public virtual void Acelerar()
    {
        Console.WriteLine("Acelerar vehículo");

        //Pisar el acelerador
    }
}
```

```
    }

}

class Moto : Vehiculo
{
    public int Cilindrada { get; set; }

    public Moto(string marca, string modelo, int cilindrada) :
base(marca, modelo)
    {
        Cilindrada = cilindrada;
    }

    public override void Acelerar()
    {
        //Girar el puño

        Console.WriteLine("Acelerar Moto");
    }
}
```

Encapsulamiento en programación orientada a objetos

Veremos otros pilares de la programación orientada a objetos, el encapsulamiento y la abstracción.

Índice

- 1 - Qué es encapsulamiento en programación orientada a objetos
- 2 - Abstracción en programación orientada a objetos
- 3 - Diferencias entre encapsulamiento y abstracción

1 - Qué es encapsulamiento en programación orientada a objetos

Decimos que el encapsulamiento en la programación orientada a objetos es cuando limitamos el acceso o damos un acceso restringido de una propiedad a los elementos que necesita un miembro y no a ninguno más.

El elemento más común de encapsulamiento son las clases, donde encapsulamos y englobamos tanto métodos como propiedades.

Otro ejemplo muy común de encapsulamiento son los getters y setters de las propiedades dentro de una clase. Por defecto nos dan el valor "normal" pero podemos modificarlos para que cambie.

```
private decimal _velocidadActual { get; set; }

public decimal VelocidadActual
{
    get{
        return _velocidadActual + 2;
    }
    set{
        _velocidadActual = value;
    }
}
```

```
}  
  
}
```

En el ejemplo que acabamos de ver tenemos dos propiedades, ambas hacen referencia a la velocidad actual, pero hay ligeras diferencias.

Una es privada, por lo que desde fuera de la clase no podemos acceder a su valor.

La segunda es pública y accede a la privada anteriormente mencionada.

El ejemplo que hemos visto es un caso real, los coches suelen marcar un par de kilómetros por hora más de los reales. Por lo que encapsulamos esa lógica dentro del setter, el cual está oculto para el consumidor que vería en su cuenta kilómetros la velocidad con los dos kilómetros por hora extra.

Podemos decir que encapsulamiento es una forma de ocultación de información entre entidades, mostrándose entre ellas solo la información más necesaria.

1.1 – Modificadores de acceso

Como hemos podido comprobar, el encapsulamiento va ligado a los modificadores de acceso.

Conviene bien recordarlos:

- Public acceso total
- Private, acceso únicamente desde la clase o struct que los contiene
- Internal, acceso desde el mismo Proyecto
- Protected desde la misma clase o desde las clases que heredan
- Protected internal, combina protected e internal y nos permite acceder desde el mismo Proyecto o de clases que la derivan.
- Private protected, que permite acceder desde la clase actual o de la que derivan de ella.

2 - Abstracción en programación orientada a objetos

La abstracción es un concepto muy similar al de la encapsulación, con la diferencia principal de que la abstracción nos permite representar el mundo real de una forma más sencilla. Podríamos definirlo también como la forma de identificar funcionalidades necesarias sin entrar en detalle de lo que estamos haciendo.

El ejemplo más claro es cuando frenamos en el coche, para nosotros, el usuario, es únicamente pisar el freno, pero por detrás el coche realiza una gran cantidad de acciones. También imaginad que en todos los coches el conductor frena de la misma manera, independientemente de si los frenos son de disco o de tambor. Mientras que el coche realiza diferentes acciones.

Pisar el freno sería el nivel de abstracción.

3 – Diferencias entre encapsulamiento y abstracción

Abstracción	Encapsulamiento
Búscala solución en el diseño	Búscala solución en la implementación
Únicamente información relevante	Ocultación de código para protegerlo
Centrado en la ejecución	Centrado en ejecución

Interfaces en programación orientada a objetos

Veremos que es una interfaz el cual está directamente relacionado con la herencia, ya que como explicamos en el post de la herencia, una clase puede heredar o implementar tantas interfaces como necesite.

Índice

- 1 - Qué es una interfaz en programación orientada a objetos?
- 2 – implementación de una interfaz
- 3 – Múltiples clases heredan de una interfaz
- 4 –múltiples interfaces para en una sola clase

1 - Qué es una interfaz en programación orientada a objetos

Una interfaz es un contrato entre dos entidades, esto quiere decir que una interfaz provee un servicio a una clase consumidora. Por ende, la interfaz solo nos muestra la declaración de los métodos que esta posee, no su implementación, permitiendo así su encapsulamiento.

Aunque esta regla de utilizar únicamente la cabecera del método en la interfaz puede verse afectada en C#8, ya que va a permitir añadir cuerpo en la declaración en las interfaces.

Una interfaz se define utilizando la palabra reservada "interface". Y por norma general, indicamos en el nombre que es una interfaz haciendo empezar el nombre de la misma por la letra I mayúscula.

```
public interface IPieza  
  
{  
  
}
```

Las interfaces pueden contener los siguientes miembros:

- Métodos
- Propiedades
- Indexers
- Eventos

```
public interface IPieza  
  
{  
  
    decimal Area();  
  
    decimal Perimetro();  
  
}
```

Como vemos en el ejemplo, hemos declarado dos métodos.

2 – Implementación de una interfaz

Para implementar una interfaz, debemos declarar una clase o una estructura(struct) que herede de la interfaz, y entonces, implementar todos sus miembros.

Por ejemplo, creamos la clase cuadrado y heredamos de una interfaz, por lo que debemos implementar sus métodos. Por defecto lo podemos dejar en NotImplementedException()

```
public class Cuadrado : IPieza
{
    public decimal Lado { get; private set; }

    public Cuadrado(decimal lado)
    {
        Lado = lado;
    }

    public decimal Area()
    {
        return Lado * Lado;
    }

    public decimal Perimetro()
    {
        return Lado * 4;
    }
}
```

Cuando implementamos una interfaz debemos de asegurar un par de puntos:

- Los métodos y tipos que devuelven deben coincidir tanto en la interfaz como en la clase.
- Deben ser los mismos parámetros
- Los métodos de la interfaz deben ser públicos

- La utilización de interfaces mejora el código y el rendimiento de la aplicación.

3 – Múltiples clases heredan de una interfaz

Uno de los casos más comunes en programación orientada a objetos es que tenemos varias clases que heredan de una interfaz, lo que implica que en ambas debemos implementar el código. Para ello creamos en el código la clase triángulo (rectángulo) conjunto a la anterior de cuadrado, ambas son piezas. Por lo que ambas heredan de IPieza.

```
public class TrianguloRectangulo : IPieza
{
    public decimal LadoA { get; set; }

    public decimal LadoB { get; set; }

    public decimal Hipotenusa { get; set; }

    public TrianguloRectangulo(decimal ladoa, decimal ladob)
    {
        LadoA = ladoa;

        LadoB = ladob;

        Hipotenusa = CalculateHipotenusa(ladoa, ladob);
    }

    private decimal CalculateHipotenusa(decimal ladoa, decimal
ladob)
    {
        return Convert.ToDecimal(Math.Sqrt((double) (ladoa *
ladoa + ladob * ladob)));
    }
}
```

```

    public decimal Area()
    {
        return LadoA * LadoB / 2;
    }

    public decimal Perimetro()
    {
        return LadoA + LadoB + Hipotenusa;
    }
}

```

Como podemos observar en ambos casos tenemos el área y el perímetro.

Si recordamos de la herencia, las clases son también del tipo del que heredan, lo que quiere decir que ambas clases, tanto Cuadrado, como TrianguloRectangulo, las podemos convertir en IPieza:

```

IPieza cuadrado = new Cuadrado(5);

IPieza trianguloRectangulo = new TrianguloRectangulo(5, 3);

```

```

Console.WriteLine($"El área del cuadrado es {cuadrado.Area()}");

```

```

Console.WriteLine($"El perímetro del cuadrado es {cuadrado.Perimetro()}");

```

```

Console.WriteLine($"El área del triángulo es {trianguloRectangulo.Area()}");

```

```

Console.WriteLine($"El perímetro del triángulo es {trianguloRectangulo.Perimetro()}");

```

```
//Resultado :
```

```
El área del cuadrado es 25
```

```
El perímetro del cuadrado es 20
```

```
El área del triángulo es 7.5
```

```
El perímetro del triángulo es 13.8309518948453
```

Este ejemplo es muy sencillito, pero imaginarios una lista con más de un millón de piezas, y cada una con diferentes formas, al heredar todas de IPieza, nos permite tener un bucle con únicamente una línea, un bucle foreach que ejecute el método Area() si lo que queremos es mostrar el área.

El uso de interfaces facilita enormemente programar cuando pasamos entidades reales a código.

Por supuesto se pueden tener métodos adicionales, normalmente privados, dentro de la clase, pero ellos no serán accesibles a través de la interfaz.

4 – Múltiples interfaces para en una sola clase

Puede darse el caso de que necesitemos implementar más de una interfaz en nuestra clase, para ello no hay ningún problema simplemente las separamos con comas.

Pero esas interfaces pueden contener un método con el mismo nombre, el mismo valor de retorno y los mismos parámetros.

La forma que tendremos entonces para diferenciar uno del otro es, en la implementación dentro de la clase

```
public interface interfaz1
```

```
{
```

```
    void MetodoRepetido();
```

```
}
```

```
public interface interfaz2
```

```
{  
  
    void MetodoRepetido();  
  
}  
  
public class EjemploRepeticionMetodo : interfaz1, interfaz2  
{  
  
    public EjemploRepeticionMetodo()  
  
    {  
  
    }  
  
    void interfaz1.MetodoRepetido()  
  
    {  
  
        throw new NotImplementedException();  
  
    }  
  
    void interfaz2.MetodoRepetido()  
  
    {  
  
        throw new NotImplementedException();  
  
    }  
  
}
```

Como podemos observar no utilizamos un modificador de acceso en la implementación del método.

Modificador sealed en C#

Hablaremos de dos piezas claves en la programación orientada a objetos, aunque no se utilizan demasiado, siempre está bien tenerlas en cuenta.

Además, ambas tienen mucho que ver, ya que utilizar sealed en una clase previene que podamos implementarla, y al final evita que podamos sobrescribir métodos.

Índice

- 1 - Qué es el modificador sealed
- 2 - Modificador sealed en métodos

1 - Qué es el modificador sealed

Una sealed class o clase sellada es aquella de la cual no podemos heredar. Lo que quiere decir que no podemos implementarla en otras clases para tener acceso a sus miembros.

Lo más fácil es verlo con un ejemplo.

Si recordamos el ejemplo que vimos en la herencia con los vehículos, teníamos lo siguiente:

```
class Vehiculo
```

```
{
```

```
    //Código
```

```
}
```

```
class Coche : Vehiculo
```

```
{
```

```
    //Código
```

```
}
```

```
class Moto : Vehiculo
```

```
{  
  
//Código  
  
}
```

Dentro de coche, podríamos incluir tanto furgoneta, como furgón, turismo, etc.

```
class Furgoneta : Coche {}  
  
class Furgon : Coche {}  
  
class Turismo : Coche {}
```

Pero por la lógica de nuestro negocio podemos querer que esta funcionalidad no suceda. Para ello podemos añadir el modificador sealed, el cual evitará que podamos heredar de ella.

```
sealed class Coche : Vehiculo  
  
{  
  
    //Código  
  
}
```

```
sealed class Moto : Vehiculo  
  
{  
  
    //Código  
  
}
```

```
class Furgoneta : Coche { } //Error  
  
class Furgon : Coche { } //Error  
  
class Turismo : Coche { } //Error
```

Como vemos hemos añadido el modificador a la clase Coche, para que así, ninguna otra pueda heredar de ella.

Como vemos en el ejemplo, la implementación de las clases Furgoneta, Furgon y Turismo nos da un error de que no pueden derivar de una clase sealed.

2 - Modificador sealed en métodos

Como recordamos del polimorfismo en POO () podemos sobrescribir métodos en las clases que derivan de otra utilizando los modificadores virtual y override.

```
class Vehiculo
{
    public decimal VelocidadMaxima { get; set; }

    public int NumeroRuedas { get; set; }

    public string Marca { get; set; }

    public string Modelo { get; set; }

    public Vehiculo(string marca, string modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }

    public virtual void Acelerar()
    {
        Console.WriteLine("Acelerar vehículo");

        //Pisar el acelerador
    }
}
```

```

class Moto : Vehiculo
{
    public int Cilindrada { get; set; }

    public Moto(string marca, string modelo, int cilindrada) :
base(marca, modelo)
    {
        Cilindrada = cilindrada;
    }

    public override void Acelerar()
    {
        //Girar el puño

        Console.WriteLine("Acelerar Moto");
    }
}

```

Además de ello, podemos incluir lo visto anteriormente con las clases, pero para métodos individuales.

Cuando incluimos sealed en un método, quiere decir que ya no podemos seguir sobrescribiendo el método en las clases hijas. Con lo que cuando llamamos al método, será siempre el que hemos indicado como final.

```

class Moto : Vehiculo
{
    public int Cilindrada { get; set; }

```



```

    public Moto(string marca, string modelo, int cilindrada) :
base(marca, modelo)

    {

        Cilindrada = cilindrada;

    }


    public sealed override void Acelerar()

    {

        //Girar el puño

        Console.WriteLine("Acelerar Moto");

    }

}

class Triciclo : Moto

{

    public Triciclo(string marca, string modelo, int
cilindrada) : base(marca, modelo, cilindrada)

    {

    }


    public override void Acelerar() //Error

    {

        //Código

    }

}

```

Como vemos en el ejemplo hemos añadido el modificador sealed antes del override, lo que indica que desde ahí ya no se va a poder heredar ese método, y por ese

motivo, en la clase que hereda de moto si intentamos crear un método llamado Acelerar() nos da un error, ya que el método de la clase padre esta sellado.

Sobrecarga de métodos en programación

La sobrecarga de métodos podemos definirla como polimorfismo en tiempo de ejecución.

Índice

- 1 - Qué es la sobrecarga de métodos?
- 2 – Ejemplo sobrecarga de métodos en C#

1 - Qué es la sobrecarga de métodos?

Sobrecarga de métodos significa que tenemos múltiples métodos dentro de una clase los cuales contienen el mismo nombre, pero diferentes parámetros.

Estos parámetros pueden ser diferentes en múltiples aspectos:

- a. Cantidad de parámetros
- b. Tipo de los parámetros
- c. Orden de los parámetros

Hay que tener en cuenta que no podemos definir dos métodos iguales, ósea con el mismo nombre, y mismos parámetros y mismo orden ya que resultaría en un error de compilación.

2 – Ejemplo sobrecarga de métodos en C#

La forma más sencilla de ver esto es con un ejemplo.

Disponemos de una clase que realiza una suma a la cual le pasamos dos valores.

```
public int Suma(int item1, int item2)
{
```

```

    return item1 + item2;
}

```

Ahora lo que deseamos es mandar 3 parámetros en vez de 2 lo que haríamos en vez de cambiar los nombres que sería una mala práctica, creamos otro método, con el mismo nombre, pero le pasamos tres parámetros, en vez de dos.

```

public int Suma(int item1, int item2)
{
    return item1 + item2;
}

```

```

public int Suma(int item1, int item2, int item3)
{
    return item1 + item2 + item3;
}

```

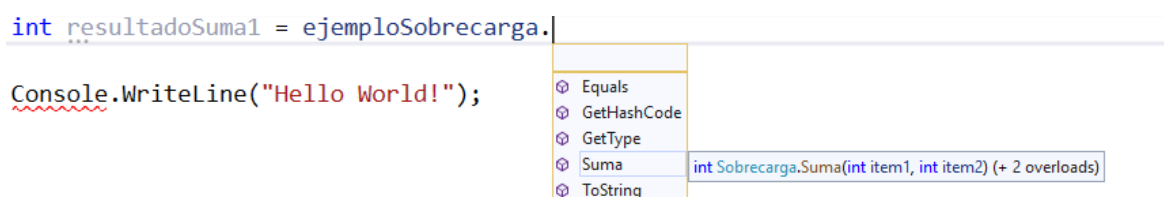
Y como he indicado previamente, también podemos tener la misma cantidad de parámetros, pero con distintos tipos, el compilador sabe que método es el que debe utilizar y no nos genera ningún problema.

```

public int Suma(int item1, int item2)
{
    return item1 + item2;
}
public string Suma(string item1, int item2)
{
    return $"{item2} sumado a {item1}";
}

```

Como vemos a la hora de ejecutar el código, funciona sin problemas. Además, cuando estamos escribiendo el método que queremos usar, visual studio detecta que es sobrecarga y no nos muestra líneas para elegir el método. Sino que nos muestra solo una, y nos indica que hay 2 métodos mas que hacen sobrecarga.



```

int resultadoSuma1 = ejemploSobrecarga.

```

The screenshot shows a code completion menu for the `Suma` method. The menu lists several methods: `Equals`, `GetHashCode`, `GetType`, `Suma`, and `ToString`. The `Suma` method is highlighted, and a tooltip shows the signature `int Sobrecarga.Suma(int item1, int item2) (+ 2 overloads)`.

Y una vez elegido el método, podemos iterar sobre las distintas sobrecargas del mismo con las teclas arriba y abajo.

```
int resultadoSuma1 = ejemploSobrecarga.Suma()  
▲ 1 of 3 ▼ int Sobrecarga.Suma(int item1, int item2)  
int resultadoSuma1 = ejemploSobrecarga.Suma()  
▲ 3 of 3 ▼ int Sobrecarga.Suma(int item1, int item2, int item3)
```

Y esto sería el ejemplo de llamar a ambos métodos y lo hace sin ningún problema

```
int resultadoSuma1 = ejemploSobrecarga.Suma(1, 2);  
int resultadoSuma2 = ejemploSobrecarga.Suma(1, 2, 3);
```

Así es como implementamos sobrecarga de métodos en C# definiendo múltiples métodos con el mismo nombre, pero con diferentes parámetros dependiendo de los requisitos que necesitemos.

Clase abstracta en C#

Índice

- 1 – El modificador abstract
- 2 – Clases y miembros Abstractos
- 3 - Uso de una clase abstracta en el mundo real
- 4 - Clase abstracta vs interface en C#

1 – El modificador abstract

Utilizamos el modificador `abstract` para definir clases o miembros de clases (métodos, propiedades, events, o indexers) para indicar que esos miembros deben ser implementados en las clases que derivan de ellas.

2 – Clases y miembros Abstractos

Cuando declaramos una clase como `abstract` estamos indicando que esa clase va a ser utilizada como clase base de otras clases, ya que ella misma no se puede instanciar.

Una clase abstracta puede contener miembros abstractos como no abstractos, y todos los miembros deben ser implementados en la clase que la implementa.

Para seguir con el ejemplo que vimos en las interfaces en POO utilizaremos el ejemplo de las piezas.

```
public abstract class Pieza
{
    public abstract decimal Area();
    protected abstract decimal Perimetro();
}
```

Como vemos a diferencia de las interfaces en las clases abstractas podemos incluir modificadores de acceso.

Y como hacíamos en las interfaces, en la clase que va a implementar la clase, indicamos dos puntos y la clase.

A diferencia de las interfaces, en las clases que implementan una clase abstracta debemos sobrescribir cada uno de los métodos. Para ello, como vimos en el post de polimorfismo utilizaremos la palabra clave `override` como vemos en el ejemplo, en las interfaces, simplemente implementamos el método, no lo sobrescribimos.

```
public class Cuadrado : Pieza
{
    readonly decimal Lado;
    public Cuadrado(decimal lado)
    {
        Lado = lado;
    }
    public override decimal Area()
    {
        return Lado * Lado;
    }

    public override decimal Perimetro()
    {
        return Lado * 4;
    }
}
```

Y como podemos observar, instanciamos el objeto sin ningún problema:

```
Cuadrado cuadrado = new Cuadrado(3);
```

Y finalmente observamos como podemos incluir miembros no abstractos dentro de la clase abstracta, lo cual definiremos como implementación por defecto:

```
public abstract class Pieza
{
    public abstract decimal Area();
    public abstract decimal Perimetro();
    public bool EjemploMetodo()
    {
        return false;
    }

    public int ValorNatural = 1;
}
```

Estos métodos, no estamos obligados a implementarlos en las clases que implementan la clase abstracta, pero podemos marcarlo como `virtual` para así poder sobrescribir el método.

Como nota especial diremos que el compilador no nos dejará incluir el modificador sealed en las clases abstractas ya que, al no poder heredar de ella no podríamos implementarla y dejaría de tener sentido.

3 - Uso de una clase abstracta en el mundo real

Como hemos explicado las clases abstractas se utilizan como clase base y es ahí donde radica todo su potencial. Ya que nos da la posibilidad de detectar el código común y extraerlo en ella.

Un ejemplo muy claro sería, por ejemplo, si tenemos eventos, distintos tipos de eventos, en los que debemos procesarlos, por ejemplo.

Alquiler de coches y alquiler de motos. Cuando vamos a procesar dichos eventos debemos estar seguros de que, por ejemplo

El coche no está alquilado, así como la moto no está alquilada, para ello debemos comprobar en sus respectivas bases de datos que no están alquilados. Combinando generics (que veremos lo que son más adelante), junto con las clases abstractas nos queda un código muy potente, pero por ahora veremos un ejemplo más sencillo.

Como en el ejemplo anterior trabajaremos con figuras geométricas ya que es un ejemplo muy fácil de entender.

Como sabemos, podemos calcular el área de un triángulo utilizando su hipotenusa y uno de sus lados. Y esto es así con todos los diferentes tipos de triángulo. Por lo que ese método es común y deberemos ponerlo dentro de la clase abstracta.

```
public abstract class TrianguloBase
{
    public abstract decimal Perimetro();

    public double CalcularAreaConHipotenusa(int lado, int
hipotenusa)
    {
        double ladob = Math.Sqrt(Math.Pow(hipotenusa, 2) -
Math.Pow(lado, 2));
        return lado * ladob / 2;
    }
}
```

Y cuando definimos cualquier otro triángulo no vamos a definir ese método en concreto.

```
public class Escaleno : TrianguloBase
{
    public override decimal Perimetro()
    {
        throw new NotImplementedException();
    }
}
public class Acutangulo : TrianguloBase
{
    public override decimal Perimetro()
    {
        throw new NotImplementedException();
    }
}
```

En cambio podemos acceder a él sin problema cuando hemos inicializado las variables

```
Escaleno escalenno = new Escaleno();  
Acutangulo acutangulo = new Acutangulo();  
  
escalenno.CalcularAreaConHipotenusa(1, 5);  
acutangulo.CalcularAreaConHipotenusa(1, 7);
```

4 - Clase abstracta vs interfaz en C#

Como has podido observar el uso de las clases abstractas es muy similar al de las interfaces pese a ello utilizamos unas u otras dependiendo del contexto, ya que su uso se puede solapar.

1. La principal diferencia que podemos observar es que en las clases abstractas podemos indicar el modificador de acceso, mientras que en las interfaces no podemos modificarlo.
2. Cuando utilizamos interfaces, podemos implementar múltiples interfaces. Mientras que solo podemos utilizar una clase abstracta como base.
3. En una clase abstracta, al poder incluir miembros no abstractos podemos incluir una implementación por defecto. Desde C#8 podemos crear elementos por defecto.
4. En una clase abstracta podemos incluir un constructor, mientras que en una interfaz no.

Static en C#

Índice

- 1 – Qué es una clase estática en C#
- 2 – Métodos estáticos en C#
- 3 – Cuando usar static

1 – Qué es una clase estática en C#

Una clase estática (`static class`) en términos generales es similar a una clase normal, pero tiene una diferencia crucial. Y es que no puede ser instanciada. Lo que quiere decir que para acceder a los miembros de una clase estática utilizando directamente el nombre de la clase estática.

Definimos una clase estática indicando la palabra `static`, y cuando lo hacemos debemos incluir todos sus miembros como estáticos.

```
public static class Calculadora
{
    public static int Suma(int x, int y)
    {
        return x + y;
    }
}
```

Y cuando queremos referenciarla lo hacemos directamente llamando a la clase y al método que también es estático.

```
int resultadoSuma = Calculadora.Sum(1, 2);
```

2 – Métodos estáticos en C#

Cuando creamos un método, este método lo incluimos dentro de una clase, y podemos utilizarlo cuando instanciamos la clase. Como está explicado en el ejemplo anterior, en el caso de ser un método estático no debemos instanciar la clase, ya que el método no pertenece a la instancia de la clase. Para ello debemos marcar el miembro como `static`.

Esta funcionalidad es activa independientemente de si la clase es estática o no, por ejemplo, en el siguiente código:

```
public class Calculadora
{
    public static int Suma(int x, int y)
    {
        return x + y;
    }

    public double Media(List<int> valores)
    {
        return valores.Average();
    }
}
```

Accedemos al método suma sin instanciar la clase, mientras que para utilizar el método media debemos instanciar la clase para poder acceder a él.

```
int resultadoSuma = Calculadora.Suma(1, 2);
Calculadora calc = new Calculadora();
double media = calc.Media(new List<int>());
```

3 – Cuando usar static

Para decidir si debemos crear una clase o un método como estática debemos preguntarnos si tiene sentido llamar al método o la clase independientemente del objeto.

- En el siguiente post veremos los “**extensión methods**” los cuales son estáticos.

Un ejemplo muy común, sobre todo cuando empezamos a programar, es tener una clase llamada “**Util**” en la que ponemos las cosas que necesitamos, pero no tenemos muy claro dónde, así que a para “útil” que van. Bien, los métodos ahí suelen ser, enviar mensajes, algún tipo de log, hacer un parseo. Métodos muy concretos que hacen una cosa muy concreta independientemente del contexto.

Una clase útil suele lucir como la siguiente:

```
public class Util
{
    public static void LogError(string mensaje)
    {
        //codigo
    }
    public static bool ComprobarCaracteresEspeciales(string
frase)
    {
        //Codigo
    }
}
```

Al ser un método estático, podemos acceder al método directamente en vez de instanciar la clase, ósea: **Util.LogError()** en cambio si los métodos no fueran

estáticos deberíamos instanciar la clase, y para acceder lo haríamos de la siguiente forma: `new Util().LogError()`.

- Debido a que es independiente del contexto, incluso si un método estático está dentro de una clase no estática, no podemos acceder a los miembros de la clase que no son estáticos. Esto quiere decir que, desde un miembro estático, sólo podemos acceder a otros estáticos.
- Una clase estática es `sealed` por lo que otra no puede heredar de ella, eso implica que tampoco podemos utilizar `polimorfismo`
- Si la clase es estática, contendrá únicamente miembros estáticos, mientras que, si no es estática, podrá contener tanto estáticos como no estáticos.
- No podemos incluir un constructor dentro de una clase estática.

Extension methods en C#

El post que vamos a ver hoy está directamente relacionado con el que vimos la semana pasada de los métodos estáticos y es que hoy vamos a ver que son los `"Extension methods"` los cuales nos permiten ampliar la funcionalidad de un objeto.

1 - Qué son los Extension Methods

Por ejemplo, como recordamos del post de en el que vimos las cadenas de texto el método `string` posee una serie de funciones o métodos por defecto, como pueden ser `.ToUpper()`, `.ToLower()` etc. .NET nos permite extender esa funcionalidad utilizando los `extension methods` o métodos extensibles.

2 - Cómo crear extension methods o métodos de extensión

Para este ejemplo vamos a crear un método que nos convierta la primera letra de una palabra o frase a mayúscula.

Para ello debemos crear un método que realice esta funcionalidad.

```
public static string PrimeraMaysucula(string fraseInicial)
{
    char primeraLetra = char.ToUpper(fraseInicial[0]);
    string RestoDeFrase = fraseInicial.Substring(1);

    return primeraLetra + RestoDeFrase;
}

Console.WriteLine(PrimeraMaysucula("hello world!"));
```

Pero como vemos en este método debemos mandar la palabra o frase como parámetro cada vez que queramos esta funcionalidad. Esta forma de programar es correcta, y válida, pero lo podemos hacer mejor. Y aquí es donde entran los `extension methods`

2.1 – Creación de un extension method

Como hemos indicado un `extension method` nos permite extender la funcionalidad de un objeto o tipo con métodos estáticos. Con las únicas dos condiciones que son, el método tiene que ser estático y en el primer parámetro, debemos indicar la palabra clave "this".

Utilizamos la palabra clave this en el primer parámetro para indicarle al compilador a que tipo va a extender. Por lo tanto, el método anterior, nos quedaría de la siguiente forma

```
public static class StringExtensions
{
    public static string PrimeraMaysucula(this string
fraseInicial)
    {
        char primeraLetra = char.ToUpper(fraseInicial[0]);
        string RestoDeFrase = fraseInicial.Substring(1);

        return primeraLetra + RestoDeFrase;
    }
}

//Llamada
Console.WriteLine("hello world!".PrimeraMaysucula());
```

Como nota adicional indicar que los `extension methods` se suelen colocar en clases estáticas por supuesto, pero que además su nombre hace referencia, por ejemplo, si vamos a extender el tipo `string`, lo indicaremos con el nombre `StringExtensions` si vamos a extender el tipo `int` la llamaremos `IntExtensions`

Finalmente indicar que los extension methods pueden ser implementados para cualquier tipo, inclusive los creados por nosotros mismos.

Tipos anónimos en C#

Índice

- 1 - Qué son los tipos anónimos en C#.
 - 1.1 - Cómo enviar un tipo anónimo como parámetro.

1 - Qué son los tipos anónimos en C#

Un tipo anónimo es una clase que no tiene nombre, lo cual quiere decir que no tenemos esa clase como tal en el código. La gran mayoría de las veces, utilizaremos tipos anónimos cuando realizamos queries.

Pero también podemos utilizarlos fuera de las queries, para crear un tipo anónimo, lo único que necesitamos es utilizar la palabra reservada `new`

```
var equipo = new { Nombre = "Real Betis", Ligas = 1 };
```

Como podemos observar el objeto que hemos creado es un objeto llamado `equipo` el cual tiene dos propiedades, Nombre y Ligas; Cuando asignamos un valor a estas propiedades, el compilador automáticamente detecta el tipo que van a ser basándose en el valor que asignamos a la propiedad. En este caso el nombre será un `string` y Ligas será un `int`.

Posteriormente si queremos acceder a sus propiedades únicamente debemos escribir el nombre de la variable y su propiedad utilizando el punto

```
string nombreEquipo = Equipo.Nombre;
```

1.1 - Cómo enviar un tipo anónimo como parámetro

Antes de continuar, decir que enviar tipos anónimos como parámetro no es una buena práctica que se deba hacer. Pese a ello hay una forma de realizar esta acción.

Los tipos anónimos no tienen tipo como tal, por lo que para enviarlo a un método diferente, debemos utilizar en ese método el tipo `dynamic` pero nos puede dar muchos errores en tiempo de ejecución, ya que el compilador no comprueba que el tipo que pasemos sea el correcto.

```
public void test(dynamic equipo)
{
    var t = equipo.Nombre;
}
```

Una vez hemos creado el método y pasado el tipo `dynamic`, podemos acceder a sus propiedades utilizando el punto, como en un objeto normal. Hay que tener en cuenta que a partir de ahora este objeto no será comprobado de que la propiedad exista, y si por ejemplo, escribimos mal la propiedad, indicamos una que no existe, nos dará un error.

Tipos nullables en C#

Índice

- 1 - Tipos nullables en C#
 - 1.1 - Propiedades de los tipos nullables

1 - Tipos nullables en C#

No podemos inicializar una variable a null, porque en sí es un valor null, osea que no sería del mismo tipo.

En cristiano, no podemos hacer lo siguiente

```
int enteroNulo = null;
```

nota: en nuestro ejemplo lo realizaremos con un entero, pero puede ser cualquier tipo de dato, como un decimal o un datetime

En cambio C# nos permite tratar esta encrucijada permitiéndonos modificar los tipos para que estos sean nullables. Esta acción se realiza indicando el carácter `?` después del tipo de dato que va a ser.

```
int? enteroNulo = null;
enteroNulo = 123;
```

Como vemos nos permite tener un entero `null`. Y le podemos asignar un valor sin ningun problema.

Pero si queremos asignar este valor a una variable que sí es un entero, nos dará un error.

```
int enteroNormal = 234;
int? enteroNulo = 123;
enteroNormal = enteroNulo;
```

esto tiene sentido, ya que no son del mismo tipo, `int?` puede contener un valor nulo, mientras que `int` no puede contenerlo. para realizar esa conversión debemos utilizar una de sus propiedades.

1.1 - Propiedades de los tipos nullables

Los tipos nulos nos dejan acceder a ciertas propiedades que nos permitirán de una manera sencilla identificar varias acciones que necesitamos, como puede ser la propiedad `HasValue`, la cual nos dice si contiene valor o si es un null. para finalmente acceder a la propiedad `Value`, que nos dará el valor de esa variable si no es un nulo.

Por lo que, basándonos en el ejemplo anterior, podríamos comprobar si la variable `enteroNulo` tiene un valor y si no lo tiene, asignarlo.

```
if(enteroNulo.HasValue())
{
    enteroNormal = enteronulo.Value;
}
```

Crear excepciones en C#

Como vimos anteriormente cuando sucede un error en nuestro sistema, tendremos lo que se define como excepción

En este post, veremos una extensión a este post. En el que crearemos nuestras propias excepciones.

Los grandes beneficios de tener excepciones personalizadas es que nos permite, obviamente controlar nuestro código, y principalmente se usa para monitorizar nuestra página web.

Para nuestro ejemplo, imaginemos que tenemos un sistema de facturas, y un usuario intenta acceder a una factura que no es suya.

Índice

- 1 – Crear una excepción personalizada
- 2 – Utilizar una excepción personalizada

1 – Crear una excepción personalizada

Para crear una excepción personalizada, únicamente debemos crear una clase que implemente el objeto Exception e indicar un constructor, como vemos en el ejemplo.

```
public class FacturaDiferenteClienteException : Exception
{
    public FacturaDiferenteClienteException(string message) :
base(message)
    {
        Console.WriteLine(message);
        Util.EnviarEmailAlerta("Intento de hackeo", message);
    }
}
```

Para simular un sistema he creado una clase `Util` que contiene un método `EnviarEmailAlerta`, ya que este ejemplo en concreto es o bien un bug muy gordo o un intento de hackeo.

2 – Utilizar una excepción personalizada

Para utilizar nuestra excepción que acabamos de crear, debemos instanciarla manualmente, para ello, y para agilizar el ejemplo he creado una clase llamada Repository al que pasamos dos ids, y simplemente devuelve true o false. Vamos al lío, hacemos la llamada, y si nos devuelve falso, es que, en este caso, el cliente y el dueño de la factura no coinciden, con lo que lanzaremos nuestra `excepción`, como vemos en el siguiente código:

```
if(!repo.ClienteYFacturaDuenoSonElMismo(clienteId, facturaId))
{
```



```
        throw new FacturaDiferenteClienteException($"El cliente {clienteId} esta intentando acceder a la factura {facturaId} que no le corresponde.");
    }
```

Operador ternario en C#

Índice

- 1 - Qué es el operador ternario
 - 1.1 - Convertir If en un operador ternario
- 2 - Operadores ternarios anidados

1 - Qué es el operador ternario

Como sabemos del post de la sentencia de toma de decisiones, C# nos permite tomar decisiones mediante condicionales, `if` por ejemplo.

C# Incluye un tipo de toma de decisiones especial, llamado `operador ternario`, el cual se representa con el carácter `?` y nos devolverá, siempre un `booleano`, por lo tanto `true` o `false`.

La sintaxis de la sentencia es así:

Expresión booleana `?` sentencia `1` : sentencia `2`;

Como puedes observar la sentencia contiene tres partes:

- A. La expresión booleana; La cual nos devolverá verdadero o falso. (antes del símbolo de cerrar la interrogación `?`)
- B. Sentencia 1; es la expresión que se va a devolver en caso de que la expresión booleana sea `true`. (antes de los dos puntos `:`)
- C. Sentencia 2; Es la expresión que se va a devolver en caso de que la expresión devuelva `false`;

Por lo tanto podemos ver la expresión ternaria como:

```
resultado = condicion ? true : false;
```

1.1 - Convertir If en un operador ternario

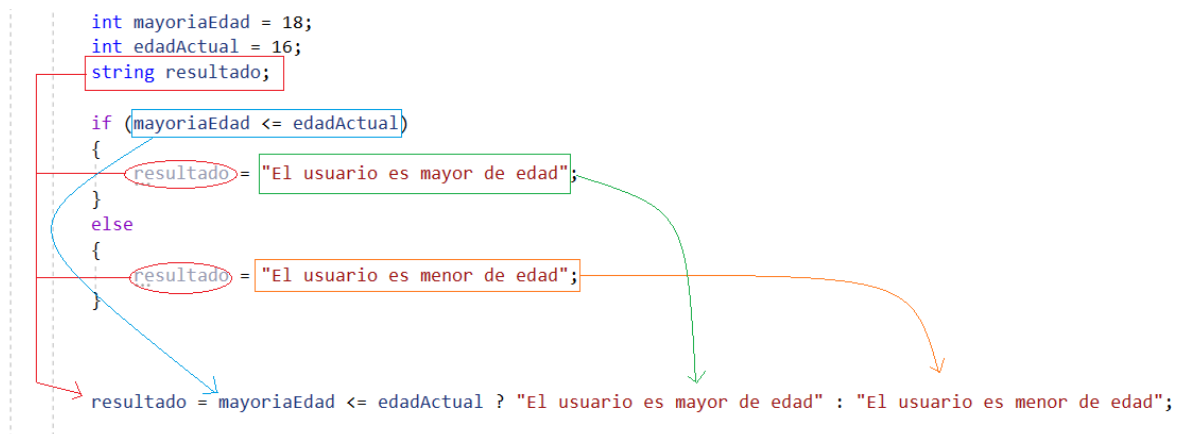
En este ejemplo veremos el resultado de comprobar si una persona es mayor de 18 años o no con un if normal.

```
if (mayoriaEdad <= edadActual)
{
    resultado = "El usuario es mayor de edad";
}
else
{
    resultado = "El usuario es menor de edad";
}
```

Y posteriormente con un operador ternario:

```
resultado = mayoriaEdad <= edadActual ? "El usuario es mayor de edad" : "El usuario es menor de edad";
```

Como podemos observar queda todo mucho mas limpio, pero, veamos en detalle como ha sido la transformación:



2 - Operadores ternarios anidados

Por supuesto cabe destacar que podemos ejecutar operadores ternarios anidados, osea uno dentro de otro. y estos se ejecutan de derecha a izquierda, no de izquierda a derecha.

- Supongamos el ejemplo que la mayoría de edad son 18
- La edad para ingerir bebidas alcohólicas son 21
- Y finalmente la edad para conducir son 25

Por lo tanto tendremos 3 condicionales y un total de 7 opciones en la ecuación.

```
int mayoría = 18, votar = 21, conducir = 25, edadactual=25;

resultado = conducir <= edadactual ? "puede conducir y votar"
: votar <= edadactual ?
    "puede votar" : mayoría <= edadactual ?
    "es mayor de edad" : "No puede hacer nada";
```

lo cual lo podemos traducir en lo siguiente `a ? b : c ? d : e ? f : g` que se evalúa como `a ? b : (c ? d : (e ? f : g))`

Por lo que si hacemos un pequeño cambio en el código anterior y cambiamos el orden de la mayoría de edad y conducir

```
resultado = mayoría <= edadactual ? "es mayor de edad" : votar
<= edadactual ?
    "puede votar" : conducir <= edadactual ?
    "puede conducir" : "No puede hacer nada";
```

Como podemos observar comúnmente utilizaremos expresiones ternarias para reemplazar un pequeño `if else`, y nos ayudara a tener un código más limpio.

Generics en C#

Índice

- 1 - Qué son Generics en C#
- 2 - Ejemplo de Generics en C#
 - 3 - Convertir nuestro Código a generic en C#
 - 3.1 - Múltiples tipos en una clase genérica en c#
- 4 - Uso de una clase genérica
- 5 - Métodos Genéricos en c#
- 6 - Consejos de Generics en C#
- 7 - Condiciones en los tipos
- Conclusión

1 - Qué son Generics en C#

Generics es una funcionalidad que nos da **.NET** la cual nos permite crear código reusable entre múltiples entidades.

Cuando creamos código genérico lo hacemos para que sea compatible con cualquier tipo de dato y por ello, “seguro” ante diferentes tipos.

2 - Ejemplo de Generics en C#

La forma más sencilla de demostrar generics es utilizando un código ejemplo.

Para ello utilizare código que presente en una prueba técnica, la cual cumple perfectamente con lo acometido.

En dicha entrevista tenía que realizar diferentes llamadas API y devolver el resultado si estaba bien etc, por ejemplo, habia una opcion que era `get/{id}` el cual devolvía un objeto `Persona`.

Para ello utilicé una clase llamada `OperationResult` la cual contiene 3 propiedades.

- `Bool Success`
- `List<string> messageList`
- `Persona Persona`

Englobar los objetos dentro de otro nos puede ayudar en caso de que tengamos una aplicación leyendo de nuestra API. Ya que de ese modo facilitaremos la vida al cliente.

```
public class OperationResult
{
    public bool Success => !MessageList.Any();
    public List<string> MessageList { get; private set; }
    public Persona Persona { get; set; }

    public OperationResult()
    {
        MessageList = new List<string>();
    }

    public void AddMessage(string message)
    {
        MessageList.Add(message);
    }

    public void SetSuccessResponse(Persona pers)
    {

```

```

        Persona = pers;
    }
}

```

A su vez, tenemos otro endpoint que debemos devolver el objeto `Coche`.

```

public class OperationResultCars
{
    public bool Success => !MessageList.Any();
    public List<string> MessageList { get; private set; }
    public Car Coche { get; set; }

    public OperationResult()
    {
        MessageList = new List<string>();
    }

    public void AddMessage(string message)
    {
        MessageList.Add(message);
    }

    public void SetSuccessResponse(Car coche)
    {
        Coche = coche;
    }
}

```

Como podemos observar ambos objetos son prácticamente iguales, por lo que estamos repitiendo mucho código. En nuestro ejemplo en concreto estamos repitiendo 21 de 23 líneas de código por lo que no estamos cumpliendo la norma de *"no repetir código"*.

3 - Convertir nuestro Código a generic en C#

Para convertir nuestro código a generic debemos reemplazar el código que es diferente por su parte "genérica". por lo que si vemos los dos ejemplos anteriores, únicamente debemos "juntar" los tipos `Persona` y `Car`.

Pero estos tipos deben de seguir existiendo en nuestro objeto, ya que los vamos a necesitar.

Para ello convertimos la clase anterior de la siguiente forma:

```
public class OperationResult <T>
{
    public bool Success => !MessageList.Any();
    public List<string> MessageList { get; private set; }
    public T Response { get; set; }

    public OperationResult()
    {
        MessageList = new List<string>();
    }

    public void AddMessage(string message)
    {
        MessageList.Add(message);
    }

    public void SetSuccessResponse(T obj)
    {
        Response = obj;
    }
}
```

Como podemos observar hay una sola diferencia, la principal es que hemos convertido los tipos Persona y Car en el tipo T justo después del nombre de la clase, con lo que nuestra clase ya es genérica.

```

public class OperationResult <T>
{
    public bool Success => !MessageList.Any();
    public List<string> MessageList { get; private set; }
    public T Response { get; set; }

    public OperationResult()
    {
        MessageList = new List<string>();
        Success = true;
    }

    public void AddMessage(string message)
    {
        MessageList.Add(message);
    }

    public void SetSuccessResponse(T obj)
    {
        Response = obj;
    }
}

```

Por convención, cualquier programador que trabaje con generics, utilizar la letra **T** para indicar que es el tipo.

Finalmente, indicar que el tipo **T** aceptará cualquier tipo, tanto uno creado por nosotros como puede ser la clase o tipo **Persona** o un tipo como puede ser el tipo **string**.

3.1 - Múltiples tipos en una clase genérica en c#

Por supuesto podemos implementar más de un tipo en nuestra **generic class**, para ello debemos pasar más de un tipo junto al nombre de la clase

```

public class OperationResult <T, U>
{
    //Code
}

```

De todas formas, por convención, cuando introducimos más de un tipo, debemos utilizar más de una letra, para hacer la vida más sencilla a los programadores que vengan detrás de nosotros.

```
public class OperationResult <TEntrada, TSalida>
{
    //Code
}
```

4 - Uso de una clase genérica

cuando implementamos una clase genérica implementamos un tipo, eso quiere decir que cuando instanciamos la clase genérica debemos indicar el tipo que vamos a utilizar, para ello instanciamos la variable de la siguiente manera:

```
OperationResult<Car> optResult = new OperationResult<Car>();
```

Recordar que **T** dentro de nuestra clase, acepta cualquier tipo. por lo que nos servirá tanto para la clase **Car**, como **Persona**, etc.

5 - Métodos Genéricos en c#

Las clases no son las únicas partes que pueden ser genéricas en nuestro código, también podemos hacer métodos genéricos. Para ello tenemos dos opciones

A - Método genérico en clase genérica

Primero podemos tener un método que recibe un parámetro T o devuelve un parámetro T lo cual es completamente válido, como hemos visto en el ejemplo anterior:

```
public void SetSuccessResponse(T obj)
{
    Response = obj;
}
```

B - Método genérico en clase NO genérica

Si disponemos de una clase no genérica, podemos crear un método que acepte tipos genéricos, quizá no son tan comunes, pero se suelen ver de vez en cuando, sobre todo en clases base.

Para tener un método genérico en una clase no genérica lo hacemos de la siguiente manera:

```
public class Ejemplo{
    public T GenericMethod<T>(T receivedGeneric)
    {
        return receivedGeneric;
    }
}
```

Indicamos después del nombre del método el tipo `<T>`.

Y como vemos podemos devolver también el propio tipo.

6 - Consejos de Generics en C#

- Debemos utilizar Generics para crear código reusable y clases o métodos que no están enlazadas a tipos. Pero debemos ser cuidadosos, ya que si una clase no va a ser reutilizada, NO debemos hacerla genérica, ya que añade una gran complejidad a nuestro código.
- Debemos utilizar la letra T cuando solo tenemos un tipo parámetro, pero si tenemos más de uno, debe ser un nombre más descriptivo.
- Pese a ello, debemos poner los nombres de los parámetros empezando por `T`, para indicar a quien lea nuestro código que es un Tipo

7 - Condiciones en los tipos

Cuando declaramos una clase o un método genérico, tenemos la capacidad de extender la condición de que tipo de objeto vamos a pasar como tipo.

para ello utilizaremos la cláusula `where T : {condición}`

```
public interface IExample {}
public class Example : IExample{}

public class EjemploGeneric <T>
where T : struct //tipo valor
```

```
where T : class // tipo referencia
Where T : new() //constructor sin parametros
Where T : IExample //Interfaz concreta
Where T : Example //una clase concreta
{

}
```

Podemos introducir condiciones para todos los tipos que utilizamos, además podemos incluir varias condiciones para un solo tipo, por ejemplo `IExample, new()` te obligará a indicar una clase, que implemente `IExample` además de tener un constructor sin parámetros.

Conclusión

- Utilizaremos Generics como forma o técnica de definir múltiples tipos de datos con una sola variable.
- Lo cual nos permite crear código reusable de una forma segura, que además funciona con cualquier tipo de dato.
- Altamente recomendable utilizar generics tanto como podamos, pero con cabeza porque a la vez que mejoramos la calidad de nuestro código podemos añadirle cierta dificultad.

Test Unitarios en C#

Índice

- 1 - Qué son los test
 - 1.1 - Diferentes tipos de test
- 2 - Por qué escribir test?
 - 3 - Cómo crear test unitarios en C#
 - 3.1 - Añadir la referencia del proyecto a testear
- 4 - Creación de un test unitario en C#
 - 4.1 - Comprobación de una excepción dentro de los test.
- 5 - Code coverage o cobertura de código
- Conclusión:

En este post daremos un pequeño vistazo a qué son los test, empezando en los test unitarios.

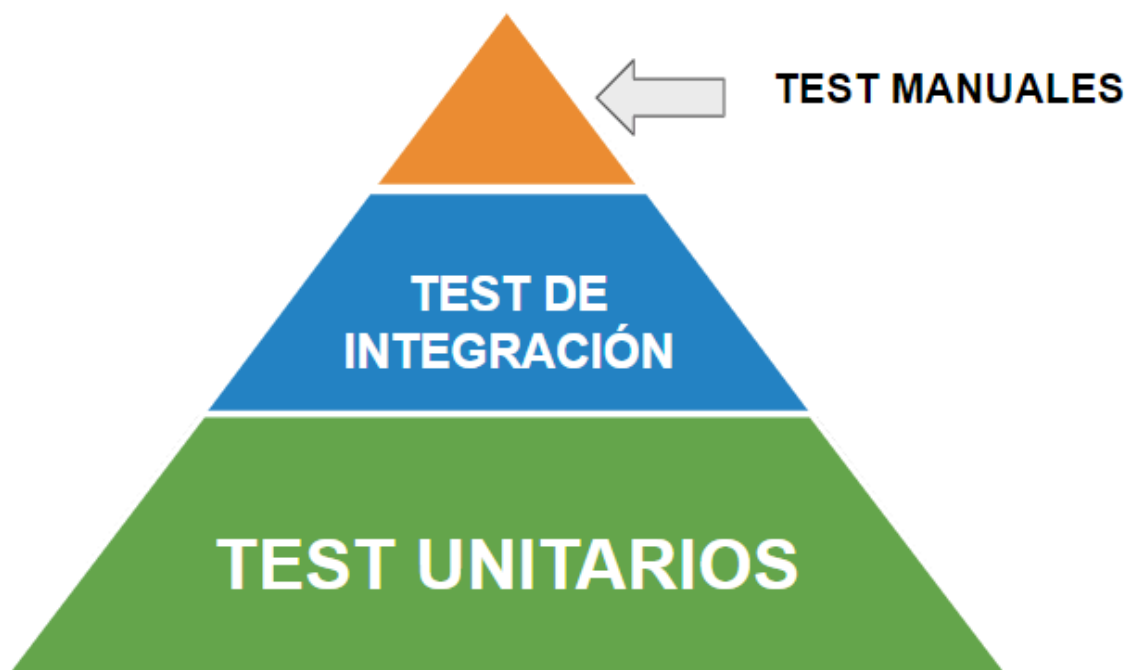
Pero antes de empezar por un tipo de test en concreto, debemos entender Qué son los test y para qué sirven.

1 - Qué son los test

Primero tenemos que tener clara la importancia que los test dan a nuestro software, ya que escribir tests, nos permitirá estar seguros de que nuestro código funciona, utilizando diferentes elementos de entrada y bajo diferentes circunstancias.

1.1 - Diferentes tipos de test

Cuando hablamos de test, se suelen representar con una imagen de una pirámide:



Esta pirámide puede cambiar, para simplificarlo, por ahora la he dividido en tres partes, que son las partes más comunes que se suelen testear

Primero, en la parte inferior, tenemos los:

- Test unitarios, los cuales van a testear métodos o servicios concretos de forma individual.
 - Este tipo de test es el que más vamos a escribir, y del tipo que trata este post.
- Test de integración, los cuales probamos diferentes integraciones del sistema, procesos “semicompletos”, pero siempre simulando llamadas a aplicaciones externas o bases de datos.
 - Este tipo de tests los veremos más adelante. Un ejemplo puede ser, el proceso de creación de un usuario en un servicio, pero sin guardar ese usuario en la base de datos.
- Test End to end, que hacen la prueba de un proceso completo, en el cual no simulamos llamadas a otros servicios o bases de datos.
 - Un ejemplo, como en el caso anterior, es la creación de un usuario, pero en este caso crearemos el usuario en la base de datos.

Se utiliza una pirámide indicando que en la parte inferior disponemos de un mayor número de test que en el bloque de la parte superior. Lo mismo sucede para encontrar y arreglar errores, si encontramos un error durante un test unitario, será más fácil de arreglar que si lo encontramos durante los test de integración o de end to end.

Idealmente deberíamos de tener todos estos test automatizados, y se puede hacer sin mucho problema, pero es común en las empresas hacer los test end to end de forma manual.

2 - Por qué escribir test?

Como he indicado en el punto anterior, hay que tener claro que escribir test nos va a traer varias ventajas, entre las que entran:

- saber que nuestro código funciona.
 - Debemos escribir tests para todos o la mayoría de casos posibles. para así asegurarnos que ese trocito de código no tiene bugs.

- Mejorar la calidad de nuestro código, al asegurarnos que, si tenemos un test que pasa, si cambiamos algo en el código y este deja de funcionar, este test va a fallar.
- Tener un mayor control de los errores, permitiéndonos ya sea filtrarlos antes para que no pasen o capturarlos y lanzar excepciones acordes, recuerda que hay un post de como capturar excepciones

3 - Cómo crear test unitarios en C#

Cabe destacar que podemos crear test unitarios en cualquier parte de nuestro código, pero, vamos a seguir el sentido común y para ello utilizaremos, dentro del proyecto en el que estamos trabajando, uno nuevo que sea exclusivamente para tests. Para ello en visual studio pulsamos en `archivo -> nuevo -> proyecto` y seleccionamos un test.

Como vemos hay tres tipos de tests para `.NET Core`

- MsTest
- NUnit
- xUnit

Los tres frameworks son muy similares, cambian algunas configuraciones, etc, que veremos en sus propios posts.

Seleccionamos `MsTest Test Project` y veremos como visual studio nos crea un nuevo proyecto, el cual contiene una sola clase, la cual contiene nuestro test.

Nota: Para el ejemplo de uso he creado una pequeña librería en C# que es una calculadora, únicamente, suma, resta, multiplica y divide.

Por lo tanto la clase que vamos a testear es la siguiente:

```
public static class CalculadoraEjemplo
{
    public static int Suma(int item1, int item2)
```

```

    {
        return item1 + item2;
    }

    public static int Resta (int item1, int item2)
    {
        return item1 - item2;
    }

    public static int Multipliacion (int item1, int item2)
    {
        return item1 * item2;
    }
    public static double Division (int item1, int item2)
    {
        return item1 / item2;
    }
}

```

Y la clase que contiene los test, la cual se ha creado automaticamente cuando hemos creado el proyecto, es la siguiente:

```

[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}

```

Como podemos observar por defecto nos crea una clase que va a ejecutar un test, y esto lo sabemos porque la clase contiene un atributo llamado `[TestClass]` y el método que va a ejecutar el test contiene un atributo llamado `[TestMethod]`

3.1 - Añadir la referencia del proyecto a testear

Cuando queremos testear un proyecto o una librería debemos añadir la referencia en el csproj de nuestro proyecto de test. en este caso las siguientes líneas:

```
<ItemGroup>
  <ProjectReference
Include="..\Calculadora\Calculadora.csproj" />
</ItemGroup>
```

Otra opción es seleccionar en el proyecto de test, sobre las dependencias **botón Secundario -> add reference**

4 - Creación de un test unitario en C#

Cuando creamos un test, debemos tener en cuenta que debemos seguir cierta estructura para que nuestros test sean fáciles de entender por otros desarrolladores.

Por convención existen unas “*best practices*” para los test que son un acrónimo llamado **AAA** cada una de las cuales significa lo siguiente:

- Arrange -> Inicializamos las variables
- Act -> Llamamos al método a testear
- Assert -> Verificamos el resultado.

Y cuando escribimos el test, podemos observar el ejemplo

```
[TestMethod]
public void Test_Calcular_Suma()
{
    //Arrange : Inicializar las variables
    int sumando1 = 2;
    int sumando2 = 3;

    //Act : llamar al metodo a testear
    int resultado = CalculadoraEjemplo.Sumar(sumando1,
sumando2);

    //Assert: comprobar el valor con el esperado.
```

```
Assert.Equals(5, resultado);  
}
```

Como vemos, primero asignamos los valores, posteriormente llamamos al método de sumar y finalmente, comprobamos que suma es correcta.

La funcionalidad de los test no termina comprobando números, los `Assert` se van a tragar cualquier tipo de datos que le metamos, y disponemos de gran cantidad de opciones, por ejemplo `.Null` para comprobar si un elementos es `null`, o `.True` para comprobar si es verdadero.

4.1 - Comprobación de una excepción dentro de los test.

También disponemos de la opción de comprobar excepciones, para ello es una forma un poco diferente, aunque varía si utilizamos `MsTest`, `Nunit` o `xUnit`.

En nuestro caso con `MsTest` debemos de utilizar el atributo `[ExpectedException]` sobre el método, el cual va a capturar y comprobar la excepción, y en este caso, no nos hará falta un `Assert`.

```
[TestMethod]  
[ExpectedException(typeof(DivideByZeroException))]  
public void Test_Calcular_Division()  
{  
    //Arrange : Inicializar las variables  
    int dividendo = 120;  
    int divisor = 0;  
  
    //Act : llamar al metodo a testear  
    double resultado = CalculadoraEjemplo.Division(dividendo,  
divisor);  
  
    //Assert: comprobar el valor con el esperado.  
    Assert.Equals(24, resultado);  
}
```


5 - Code coverage o cobertura de código

Como último punto, veremos que es el code coverage. ya que cuando programamos siempre nos estamos preguntando si hemos cubierto todo nuestro código con tests.

Ya que si, cubrimos nuestro código, sabemos al menos, que el curso normal de los acontecimientos pasará sin ningún error, aunque puede ser que salte alguna excepción, pero por eso son excepciones.

Visual Studio contiene una herramienta para hacer cobertura de código, lo malo es que es en la versión enterprise, tenemos otras alternativas, pero todas son de pago, así que por ahora no veremos los resultados.

Antes de terminar con el code coverage, no hay que obsesionarse con tener todo testeado al 100%, sobre todo más adelante en test de integración veremos que si la lógica es muy compleja puede llevar mucho tiempo cubrir el 100% del código, por lo que en estos casos es más recomendable, gastar 8h para cubrir el 70% que gastar 40h en cubrir el 100%, aunque siempre habría que intentar tener cubierto ese 30% restante con pequeños test unitarios.

Conclusión:

Crear tests es necesario en nuestro desarrollo, ya que no solo nos permite comprobar que el código que hemos escrito está bien, sino, que en caso de romper otra funcionalidad, esta se verá reflejada en los tests, y podremos ver que esa funcionalidad ya no funciona.

En proyectos grandes también sirven de gran ayuda, debido a que nos será más fácil hacer el desarrollo utilizando tests, que ir probando todo a mano, obviamente si es algo de interfaz de usuario no, pero si no lo es. Escribir el test que implementará nuestra funcionalidad es muy útil. A este tipo de programación se le llama, **Test Driven Design**.

Que una empresa utilice test, es signo de buen hacer, así que hay que fiarse de las empresas que tienen tests.

LINQ en C#

Índice

- 1 - Qué es LINQ?
- 2 - Cómo Utilizar y escribir LINQ
- 3 - La interfaz IEnumerable
 - 3.1 - Qué hace por detrás
- 4- Funcionalidades para construir queries en LINQ
 - 4.1 - Where en LINQ
 - 4.2 - Obtención de un único elemento con LINQ
 - 4.3 - Último elemento con LINQ
 - 4.4 - Ordenar elementos en LINQ
 - 4.5 - Agrupar elementos en LINQ
- 5 - Creación de un filtro para LINQ

En este post vamos a ver una pequeña introducción a LINQ y como realizar pequeñas consultas con el mismo, debido a que LINQ es muy amplio, he decidido realizar un post introductorio y en el futuro iré a cosas más específicas.

1 - Qué es LINQ?

Más o menos en 2007 los desarrolladores de C# vieron que tenían cierto patrón, o problema, donde querían acceder a datos haciendo consultas sobre los mismos y no podían hacerlo de una forma sencilla.

El motivo por el que no se podía hacer de una forma sencilla era porque, había datos en 3 fuentes diferentes, Como son.

- Las colecciones de datos en memoria: para las que necesitamos los diferentes métodos aportados por Generics o diferentes algoritmos para conseguir estos datos.
- Bases de datos: para el que necesitabamos conectores de ADO.NET y escribir nuestro propio SQL.

- Ficheros XML: Para poder iterar sobre los mismos, necesitábamos utilizar `XmlDocument` o `Xpath`.



Todas estas APIs/librerías tienen diferentes funcionalidades y sintaxis, entonces microsoft decidió introducir un lenguaje que ofreciera una sintaxis única para todas estas funcionalidades. Y aquí es donde microsoft introdujo *Language Integrated Query* o `LINQ`.



`LINQ` nos proporciona comprobaciones de tipo en consultas durante la compilación, pero estas consultas van a ser ejecutadas en tiempo de ejecución sobre datos en memoria, contra una base de datos o en xml. Pero además una vez comprendamos LINQ veremos que podemos utilizarlo contra por ejemplo el sistema de archivos de nuestro pc, una base de datos no relacional, ficheros CSV, y mucho más.

2 - Cómo Utilizar y escribir LINQ

Para realizar este ejemplo, utilizaremos LINQ sobre datos en memoria, en este caso un `Array[]` de la clase `Libro`. Podríamos utilizar una `Lista<T>` como vimos en el capítulo de arrays y listas pero, `List<T>` es específico del namespace `System.Linq` que fue específicamente creado para trabajar con él después de utilizar `IEnumerable`, el cual es el centro de este post.

```

public class Libro
{
    public int Id { get; set; }
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public Libro(int id, string titulo, string autor)
    {
        Id = id;
        Titulo = titulo;
        Autor = autor;
    }
}

Libro[] arrayLibros = new Libro[5];
arrayLibros[0] = new Libro(1, "Poeta en nueva york", "Federico
García Lorca");
arrayLibros[1] = new Libro(2, "Los asesinos del emperador",
"Santiago Posteguillo");
arrayLibros[2] = new Libro(3, "circo máximo", "Santiago
Posteguillo");
arrayLibros[3] = new Libro(4, "La noche en que Frankenstein
leyó el Quijote", "Santiago Posteguillo");
arrayLibros[4] = new Libro(5, "El origen perdido", "Matilde
Asensi");

```

Antes de empezar a utilizar LINQ hay que tener muy claro y comprender que son los extension methods y sobre todo las expresiones lambda ya que LINQ esta construido a traves de estas dos tecnologías principalemnete.

Además de hacer una sola API que pudiera consultar todas las fuentes de datos otro objetivo muy importante era hacer que estas consultas fueran fáciles de entender tanto para la persona que escribe la query, como para los que vienen después ya sea a modificarla como para hacer la review.

Lo que se consiguió fue un lenguaje muy sencillo de entender ya que utiliza *extension methods* que son declarados de una forma similar a las expresiones SQL.

Por ejemplo, la cláusula `.Where()` dentro de LINQ nos permite filtrar.

Para realizar esta acción, tenemos dos opciones, ambas nos otorgan una sintaxis entendible y sencilla y por supuesto comprobación de los tipos en tiempo de compilación.

Forma numero 1 de escribir consultas LINQ:

```
public static void LinqQueryFormal(Libro[] arrayLibros, string
autor)
{
    var libros = from libro in arrayLibros
                  where libro.Autor == autor
                  select libro ;
}
```

Como vemos disponemos de un método que recibe un `array` y un `string` autor, el cual lo utilizaremos para filtrar la lista. Si sabes SQL puedes comprobar que LINQ utiliza una sintaxis muy similar.

Al final de la consulta, indicamos una cláusula `select libro` la cual nos va a devolver un tipo `IEnumerable<T>` donde `T` es `Libro`.

Si en vez de devolver libro, quisiéramos devolver solo el título, `T` sería un `string`.

Forma numero 2 de escribir consultas LINQ

La segunda forma es utilizando *extension methods* llamándolos uno detrás de otro.

Replicar el ejemplo anterior es muy sencillo, únicamente tenemos que indicar el siguiente código:

```
public static void LinqQueryForma2(Libro[] arrayLibros, string
autor)
{
    var libros = arrayLibros.Where(a => a.Autor == autor);
}
```

Como vemos `.Where` es un *extension method*, el cual acepta un delegado `Func` por parametro. El cual filtrará cuando sea verdadero y falso, devolviendo la lista filtrada.

Como podemos observar ya no incluimos el `.Select` y es porque por defecto el código entiende que queremos hacer un select.

además de un `.Where` podemos concatenar más acciones, si por ejemplo queremos ordenar por el título podemos hacerlo de la siguiente manera:

```
public static void LinqQueryForma2(Libro[] arrayLibros, string
autor)
{
    var libros = arrayLibros.Where(a => a.Autor ==
autor).OrderBy(a=>a.Titulo);
}
```

```
}
```

Ambas opciones son igual de válidas, personalmente prefiero la segunda, pero ambas nos permiten realizar consultas que lucen como consultas reales y fáciles de entender.

3 - La interfaz IEnumerable

Como acabamos de ver, cuando hacemos una consulta LINQ el resultado nos viene en un tipo `IEnumerable`.

Esta interfaz es la más importante cuando estamos utilizando LINQ ya que la gran mayoría (el 98%) de *extension methods* que vamos a utilizar lo realizan sobre `IEnumerable<T>`.

Como es un tipo genérico, nos permite indicar qué tipo vamos a introducir en la colección.

3.1 - Qué hace por detrás

El motivo por el que podemos hacer un `foreach` sobre el resultado de la query o el motivo por el que podemos hacer un `foreach` sobre el array que hemos creado al principio, es debido a que ambos tipos implementan la interfaz `IEnumerable`, que tiene un método llamado `GetEnumerator()`.

El método `GetEnumerator()` de `IEnumerable` viene directamente de heredar `IEnumerator`. Y si convertimos cualquiera de nuestros elementos, en mi caso `arrayLibros` en `IEnumerable<Libros>` puedo acceder a `arrayLibros.GetEnumerator()` y una vez tienes este enumerador, puedes pedir al enumerador que se mueva al siguiente elemento utilizando `.MoveNext()` y podremos acceder a el con la propiedad `.Current` que contiene un puntero al elemento que estamos iterando, con lo que podemos acceder a su valor.

```
IEnumerable<Libro> arrayLibros = new Libro[] {  
    new Libro(1, "Poeta en nueva york", "Federico García  
Lorca"),  
    new Libro(2, "Los asesinos del emperador", "Santiago  
Posteguillo"),  
    new Libro(3, "circo máximo", "Santiago Posteguillo"),  
    new Libro(4, "La noche en que Frankenstein leyó el  
Quijote", "Santiago Posteguillo"),  
    new Libro(5, "El origen perdido", "Matilde Asensi")  
};
```

```
IEnumerator<Libro> enumeratorLibros =
arrayLibros.GetEnumerator();
while (enumeratorLibros.MoveNext())
{
    Console.WriteLine(enumeratorLibros.Current);
}
```

Lo bueno de utilizar `IEnumerable` es que es una interfaz que puede englobar, un array, una lista, o una consulta a una base de datos, por lo que queda totalmente transparente al usuario lo que significa que es muy fácil de mantener y trabajar con ello.

Antes de pasar al siguiente punto, es importante remarcar que la interfaz `IEnumerable` es "Lazy" (ya veremos un post) lo que significa que no va a ser ejecutada hasta que necesitamos su ejecución u obtener un elemento fuera del `IEnumerable`, por ejemplo, en un `foreach`, o si convertimos todo el `IEnumerable` en una `List<T>`.

4- Funcionalidades para construir queries en LINQ

Por defecto LINQ nos trae una gran variedad de métodos para construir nuestras consultas, y en el 99.9% de los casos, serán más que suficientes, más adelante en este post veremos cómo crear un filtro personalizado.

Las funcionalidades de las que disponemos para construir queries en LINQ son muy similares al SQL normal y así son sus nombres, en este post no voy a entrar en detalle en todas ellas pero sí en las principales.

La gran mayoría de extension methods en LINQ utilizan un delegado como parámetro de entrada que es `Func<T, bool>`

4.1 - Where en LINQ

El principal y más importante es el método `.Where` el cual nos permite filtrar utilizando los parámetros de nuestra consulta. por ejemplo como el caso que hemos visto, Comparando el nombre.

```
var libros = arrayLibros.Where(a => a.Autor == autor);
```

Cuando utilizamos where nos devuelve una lista, esto quiere decir que tendrá 0 o más elementos.

4.2 - Obtención de un único elemento con LINQ

Para obtener un único elemento tenemos varias opciones, TODAS ellas, reciben como parámetro el delegado `Func<T, bool>` (igual que el `where`) y devuelven un elemento del tipo `T`.

- `.First()` -> devuelve el primer elemento
- `.FirstOrDefault()` -> devuelve el primer elemento o uno por defecto.
- `.Single()` -> si hay más de un elemento o no hay ninguno, devuelve una excepción, si hay un elemento lo devuelve.
- `.SingleOrDefault()` -> en caso de haber más de un elemento salta una excepción y si hay solo uno o ningún devuelve el elemento o uno por defecto.

Podría hacer un post entero, quizá lo haga, sobre que utilizar si `.Single`, o `.First`, la respuesta es que ninguno, ya que `.First` puede devolver falsos positivos si hay más de un elemento y `.Single` necesita leer toda la enumeración para comprobar que no hay ninguno más.

Depende un poco del escenario, pero para obtener elementos únicos que no sabemos a ciencia cierta que son un ID único en la base de datos recomiendo una combinación entre `.Count == 1` y `.First`. haciendo que si count no es 1 salte una excepción.

4.3 - Último elemento con LINQ

Similar al anterior.

- `.Last()` -> devuelve el último elemento.
- `.LastOrDefault()` -> devuelve el último elemento o uno por defecto (vacío).

4.4 - Ordenar elementos en LINQ

Para ordenar elementos debemos llamar al método `.OrderBy` u `.OrderByDescending` el cual ordenará de forma descendente.

Ambos métodos contienen un método adicional en el que puedes indicar un objeto del tipo `IComparer` para poder ordenar a tu gusto.

El resultado que obtendremos es la propia lista ordenada como hemos indicado:

```
var librosOrdenados = arrayLibros.Where(a => a.Autor ==  
"Santiago Posteguillo").OrderBy(a=>a.Titulo);
```


4.5 - Agrupar elementos en LINQ

Para agrupar elementos debemos utilizar el método `.GroupBy()` y el resultado será una lista `IEnumerable<Grouping<Key, T>>` en nuestro caso, agrupamos por autor, y podemos iterar sobre la lista, accediendo al grupo, key contiene el nombre del autor del libro.

```
var agrupacion = arrayLibros.GroupBy(a => a.Autor);

foreach(var autorLibro in agrupacion)
{
    Console.WriteLine(autorLibro.Key);

    foreach(var libro in autorLibro)
    {
        Console.WriteLine(libro.Titulo);
    }
}
```

5 - Creación de un filtro para LINQ

bueno ahora hemos visto lo básico de LINQ, consultas, filtros etc. pero el cómo funcionan por detrás es muy importante. para ello vamos a crear un filtro nosotros.

Lo primero que tenemos que hacer es una clase y la llamaremos `ExLinq` y dentro de esta clase un *extension method* que utiliza `IEnumerable<T>`, junto con un parámetro de entrada el cual será un delegado `Func` el cual contendrá `T` y devolverá un `bool`.

Si hemos consultado la interfaz `IEnumerable`, hemos podido observar que cuando el resultado devuelve varios elementos los métodos devuelven `IEnumerable<T>` con lo que aquí realizaremos lo mismo.

Únicamente vamos a hacer un bucle sobre la lista inicial y compararla con la segunda.

```
public static class ExLinq
{
    public static IEnumerable<T> Filter<T> (this
IEnumerable<T> source, Func<T, bool> predicado)
    {
        var result = new List<T>();
    }
}
```

```

        foreach (var item in source)
        {
            if (predicado(item))
            {
                result.Add(item);
            }
        }
        return result;
    }
}

```

Pero esta no es la forma más correcta de realizar filtros en `IEnumerable` como he comentado antes, el código es “*Lazy*” y no se va a ejecutar hasta que lo necesitemos de verdad.

Para ello C# nos provee de la palabra clave `yield` la cual para todos aquellos que no la hayan visto nunca, es algo similar a echar el ojo en el método, cuando accedemos a la lista ya sea con un `foreach` o un `ToList()`, el código dirá, espera, que hay que comprobar este filtro.

Cambiamos el código a la siguiente manera:

```

public static class ExLinq
{
    public static IEnumerable<T> Filtro<T> (this
IEnumerable<T> source, Func<T, bool> predicado)
    {
        foreach(var item in source)
        {
            if (predicado(item))
            {
                yield return item;
            }
        }
    }
}

public bool FuncionEvaluadora() {

    return Autor == "Santiago Posteguillo";

}

var librosOrdenados =
arrayLibros.Filtro(FuncionEvaluadora).OrderBy(a=>a.Titulo);

```

De esta forma tenemos creado nuestro filtro personalizado y podemos llamarlo desde el código:

```

IEnumerable <Libro> librosExtension = arrayLibros.Filtro(a =>
a.Autor == "Santiago Posteguillo");

```

Delegados en C#

Índice

- 1 - Qué es un delegado
 - 1.1 - Ejemplo delegado
- 2 - Trabajando con delegados y generics
- 3 - Delegado Action
 - 3.1 - Métodos delegados anónimos
- 4 - Delegado Func
- 5 - Delegado Predicate
- Conclusión

1 - Qué es un delegado

Podemos indicar que un delegado es una referencia a un método.

Cuando definimos un delegado lo que estamos haciendo es declarar una variable que apunta a otro método.

Podemos definir un delegado utilizando la palabra clave `delegate` seguido de la declaración de la función como vemos a continuación:

```
<modificador de acceso> delegate <tipo de retorno>
<nombre>(<parametros>[]);
```

Lo cual lo podemos traducir al siguiente código:

```
public delegate void ImprimirDelegado(string value);
```

1.1 - Ejemplo delegado

Ahora llega la hora del pequeño ejemplo;

Tenemos el siguiente código:

```
namespace PA.Delegates
{
    public delegate void ImprimirDelegado(string value);
    public class EjemploDelegado
    {
        private void Imprimir(string valor)
        {
            Console.WriteLine(valor);
        }

        public void EjemploDelegate()
        {
            //Declaración
            ImprimirDelegado imprimirDelegado = new
            ImprimirDelegado(Imprimir);

            //invocación
            imprimirDelegado("texto de ejemplo");
        }
    }
}
```

```
}  
}
```

Debemos fijarnos en varios puntos :

- Primero, disponemos fuera de la clase, aunque puede estar dentro el delegado.
 - Posteriormente una clase, que contiene:
 - En método llamado "imprimir" que tiene como tipo de entrada y como tipo de retorno los mismos que el delegado.
 - Método principal el cual declara el delegado, pasando por parámetro el método "imprimir" que acabamos de crear.
- La invocación al delegado, pasando texto por parámetro.

Es importante tener en cuenta que el tipo de retorno y los parámetros de entrada al método deben coincidir entre método y delegado, de lo contrario nos dará error de compilación.

2 - Trabajando con delegados y generics

El ejemplo que acabamos de ver es muy sencillo y nos puede servir de introducción, y antes de pasar al grueso, hay que indicar que los delegados también pueden funcionar con generics.

Si no se tienen claros los conceptos de generics, es conviene revisar [este link](#) sobre generics en C#.

Primero modificamos el valor para que acepte generics

```
public delegate void ImprimirDelegado<T>(T value);
```

Y posteriormente hacemos lo propio con el método, en este caso crearemos dos métodos.

El que ya tenemos, y uno que acepte int como parámetro.

Y cuando lo declaramos únicamente debemos indicar el tipo que va a contener:

```
public class EjemploDelegado  
{  
    private void Imprimir(string valor)  
    {
```

```

        Console.WriteLine(valor);
    }

    private void Imprimir(int valor)
    {
        Console.WriteLine($"el valor es {valor}");
    }

    public void EjemploDelegate()
    {
        //Declaración
        ImprimirDelegado<string> imprimirDelegado = new
        ImprimirDelegado<string>(Imprimir);
        ImprimirDelegado<int> imprimirDelegadoEntero = new
        ImprimirDelegado<int>(Imprimir);

        //invocación
        imprimirDelegado("texto de ejemplo");
        imprimirDelegadoEntero(25);
    }
}

```

Dentro de .NET hay varios tipos de delegados ya escritos que se utilizan prácticamente cada día, estos son `Func`, `Action` y `Predicate`

3 - Delegado Action<T>

Un delegado `Action` nos permite que será un delegado que apunta a un método, el cual devuelve `void`, ya que los `Action<T>` siempre devuelven `void`, pero puede contener de 0 a 16 parámetros. Y los tipos de estos parámetros están determinados por los tipos genericos.

Por ejemplo: un `Action<string>` es un método que devuelve `void` pero que necesita recibir un `string`.

Lo cual es exactamente el método `Imprimir` que ya tenemos implementado con el método `Imprimir`.

Por lo que podemos asignarle el método al `Action<string>` y posteriormente invocarlo.

```
private void Imprimir(string valor)
{
    Console.WriteLine(valor);
}

public void EjemploAction()
{
    Action<string> imprimirAction = Imprimir;
    imprimirAction("ejemplo");
}
```

3.1 - Métodos delegados anónimos

Muy similar a lo que entendemos como tipo anonimo disponemos de los métodos anónimos.

Cuando definimos un `Action`, podemos asignar directamente el valor del delegado al que hace referencia:

```
public void EjemploActionAnonimo()
{
    Action<string> imprimirAction = delegate (string valor)
    {
        Console.WriteLine(valor);
    };

    imprimirAction("ejemplo");
}
```

Este código se puede mejorar con una `lambda expression` las cuales veremos lo que son más adelante:

```
Action<string> imprimirAction = v => Console.WriteLine(v);
```

Lo dicho, lo veremos en detalle más adelante.

4 - Delegado Func<in T, out TResult>

Similar al caso anterior, acepta de 1 a 16 parámetros y los tipos son genéricos. Con la diferencia de que en este caso, `Func<in T, out TResult>` debe devolver un valor, y el último tipo que se le asigna a `Func` es el tipo de retorno.

En otras palabras un `Func<int, string>` es una función que recibe un `int` y devuelve un `string`:

```
Func<int, string> resultado = v => $"el resultado es{v}";  
Console.WriteLine(resultado(5));
```

Como hemos indicado acepta múltiples parámetros por ejemplo si realizamos una multiplicación.

```
Func<int, int, int> multiplicacion = (v1, v2) => v1 * v2;  
int valor = multiplicacion(3, 2);  
Console.WriteLine($"El resultado es {valor}");
```

5 - Delegado Predicate<T>

Similar a los dos anteriores, en este caso, un predicate SIEMPRE devuelve un `boolean`. Por ejemplo `Predicate<int>` recibe un `int` y devuelve un `booleano`.

```
Predicate<int> mayorDeEdad = e => e >= 18;  
bool esMayorDeEdad = mayorDeEdad(10);
```

Conclusión

En este post hemos visto cómo instanciar un delegado y cómo utilizar estos delegados `Func`, `Action` y `Predicate`. Tener claros los conceptos de delegados es extremadamente útil.

Utilizaremos delegados en nuestro día a día continuamente, lo cual nos ayudará a tener un código limpio y de mayor calidad.

Pero no solo eso, sino que cuando utilizamos `LINQ` estamos continuamente utilizando `Func<T>`.

Principios SOLID

S Single Responsibility Principle

O Open/Closed Principle

L Liskov Substitution Principle

I Interface Segregation Principle

D Dependency Inversion Principle

Responsabilidad única

Índice

- 1 - Qué es el principio de responsabilidad única?
- 2 - Ejemplo de principio de responsabilidad única
 - 2.1 - Arreglar código para cumplir el principio de responsabilidad única.
 - 2.2 - En qué mejora nuestro código
- Conclusión

Veremos que es el principio de responsabilidad única o *Single Responsibility Principle* (SRP) de sus siglas en inglés.

1 - Qué es el principio de responsabilidad única?

La descripción más famosa de este principio es la del propio autor del mismo Robert C. Martin, que dijo “Una clase debe tener solo una razón para cambiar”.

Bien pero esto, qué quiere decir.

Una parte crucial cuando escribimos código es que nuestro código sea fácil de mantener y de leer. La forma de cumplir con esta premisa es que cada clase debe hacer una cosa y hacerla bien.

Cuando tenemos clases que realizan más de una tarea acaban acoplándose unas con otras cuando no deberían estar juntos haciendo esa clase mucho más difícil de usar y entender, comprender y por supuesto mantener.

Una ventaja de mantener código con las clases muy diferenciadas es que son más fáciles de testear, lo que implica que es mucho más difícil tener un bug.

Un ejemplo rápido de que el principio de responsabilidad única funciona es unix ya que unix originalmente era un sistema de línea de comandos donde cada comando es un pequeño script y este hace lo que se le pide correctamente.

2 - Ejemplo de principio de responsabilidad única

Para entender el concepto de principio de responsabilidad única lo más fácil es que veamos un ejemplo.

Este ejemplo está ultra reducido para entenderlo correctamente.

Disponemos de una clase en nuestro código en nuestro caso que tiene un método el cual nos permite insertar un artículo y otro método que permite leer ese artículo.

```
public void GuardarArticulo(string contenido, string titulo)
{
    Log.Information($"vamos a insertar el articulo {titulo}");
    File.WriteAllText($"{path}/{titulo}.txt", contenido);
    Log.Information($"articulo {titulo} insertado");
    this.Cache.Add(titulo, contenido);
}

public string ConsultarArticulo(string titulo)
{
    Log.Information($"Consultar artículo {titulo}");

    string contenido = this.Cache.Get(titulo);
    if (!string.IsNullOrEmpty(contenido))
    {
        Log.Information($"Artículo encontrado en la cache {titulo}");
        return contenido;
    }

    Log.Information($"buscar articulo en el sistema de archivos {titulo}");
    contenido = File.ReadAllText($"{path}/{titulo}.txt");

    return contenido;
}
```

En estos dos métodos debemos identificar qué factores o que código nos llevarían a tener que modificar la clase y podemos encontrar los siguientes:

- Realizamos múltiples logs.
- Almacenamos el artículo, en este caso en el sistema de archivos.
- El sistema de cache.

Como vemos disponemos de 3 puntos.

- Por ejemplo podríamos cambiar los logs, de serilog a log4net o a otro personalizado por nosotros.
- Podríamos querer almacenar los artículos en una base de datos en vez de en el sistema de ficheros.
- Si queremos cambiar el sistema de cache.

Por estos motivos el principio de responsabilidad única no se estaría cumpliendo, ya que si cambiamos el sistema de log, deberemos cambiar una clase que lee artículos, lo cual no es correcto.

Pero no solo eso, en este ejemplo disponemos de un cuarto motivo, que es la lógica de los propios métodos. osea, como las tres funcionalidades actúan entre sí para devolvernos la información que estamos buscando.

2.1 - Arreglar código para cumplir el principio de responsabilidad única.

Para cumplir con el principio de responsabilidad única lo que debemos hacer es coger cada uno de esos motivos que nos pueden hacer cambiar la clase y extraerlos en una clase nueva.

Por ejemplo el sistema de log. Debemos separarlo en una clase nueva.

Y esta clase es la que va a encargarse de que nuestros logs se inserten correctamente ya sea utilizando serilog o log4net

```
Using Serilog;
public class Logging
{
    public void Info(string message)
    {
        Log.Information(message);
    }
    public void Error(string message, Exception e)
    {
        Log.Error(e, message);
    }
    public void Fatal(string message, Exception e)
    {
        Log.Fatal(e, message);
    }
}
```

Posteriormente deberemos indicar en la clase que estabamos que utilize esa clase para hacer logs y cambiar las variables.

```
public class ArticulosServicio
{
    private readonly Logging _logging;

    public ArticulosServicio()
    {
        _logging = new Logging();
    }

    public void GuardarArticulo(string contenido, string
titulo)
    {
        _logging.Info($"vamos a insertar el articulo
{titulo}");
        File.WriteAllText($"{path}/{titulo}.txt", contenido);
        _logging.Info($"articulo {titulo} insertado");
        this.Cache.Add(titulo, contenido);
    }
    //Resto del código
}
```

2.2 - En qué mejora nuestro código

Ahora la cuestión es entender en qué mejora realizar esta acción nuestro código.

En el ejemplo estoy utilizando *serilog* el cual cuando queremos logear una excepción lo haríamos de la manera

```
Log.Error(exception, message);
```

Mientras que si cambiamos a *log4net* (otra librería para logs) se realiza con un pequeño cambio.

```
Log.Error(message, exception);
```

```

using Serilog;
public class Logging
{
    public void Info(string message)
    {
        Log.Information(message);
    }
    public void Error(string message, Exception e)
    {
        Log.Error(e, message);
    }
    public void Fatal(string message, Exception e)
    {
        Log.Fatal(e, message);
    }
}

```

```

using Log4Net;
public class Logging
{
    public void Info(string message)
    {
        Log.Info(message);
    }
    public void Error(string message, Exception e)
    {
        Log.Error(message, e);
    }
    public void Fatal(string message, Exception e)
    {
        Log.Fatal(message, e);
    }
}

```

Este cambio implica que si queremos cambiar la forma en la que la aplicación registra los logs deberemos hacerlo una única vez en la clase Logging y no en cada una donde estamos llamando a los logs.

Debemos hacer exactamente lo mismo para cada uno de los motivos o factores que teníamos para cambiar nuestro código. con lo que tendremos que crear una clase para hacer logs, una clase para hacer controlar la caché y una clase para acceder/crear a los ficheros.

```

public class Cache
{
    private Dictionary<string, string> CacheDicctionary;
    public Cache()
    {
        CacheDicctionary = new Dictionary<string, string>();
    }

    public void Add(string titulo, string contenido)
    {
        if(!CacheDicctionary.TryAdd(titulo, contenido))
        {
            CacheDicctionary[titulo] = contenido;
        }
    }

    public string Get(string titulo)
    {
        CacheDicctionary.TryGetValue(titulo, out string
contenido);
        return contenido;
    }
}

```

```

}

public class Almacenamiento
{
    string path="C:/temp";
    public void EscribirFichero(string titulo, string
contenido)
    {
        File.WriteAllText($"{path}/{titulo}.txt", contenido);
    }

    public string LeerFichero(string titulo)
    {
        return File.ReadAllText($"{path}/{titulo}.txt");
    }
}

```

Como puedes imaginar, si una clase es muy grande puede contener muchas clases adicionales a las que hace referencia, para ello SOLID incluye *dependency injection* que veremos más adelante en esta serie de posts, este es el resultado final:

```

public class ArticulosServicio
{
    private readonly Cache _cache;
    private readonly Logging _logging;
    private readonly Almacenamiento _almacenamiento;

    public ArticulosServicio()
    {
        _logging = new Logging();
        _almacenamiento = new Almacenamiento();
        _cache = new Cache();
    }

    public void GuardarArticulo(string contenido, string
titulo)
    {
        _logging.Info($"vamos a insertar el articulo
{titulo}");
        _almacenamiento.EscribirFichero(titulo, contenido);
        _logging.Info($"articulo {titulo} insertado");
        _cache.Add(titulo, contenido);
    }
}

```



```

public string ConsultarArticulo(string titulo)
{
    _logging.Info($"Consultar artículo {titulo}");

    string contenido = _cache.Get(titulo);
    if (!string.IsNullOrEmpty(contenido))
    {
        _logging.Info($"Artículo encontrado en la cache {titulo}");
        return contenido;
    }

    _logging.Info($"buscar articulo en el sistema de archivos {titulo}");

    return _almacenamiento.LeerFichero(titulo);
}
}

```

Conclusión

- Cuando aplicamos el principio de responsabilidad única estamos haciendo que nuestro código sea más fácil de entender, ya que solo realiza una única funcionalidad.
- El código será más fácil de mantener, ya que los cambios serán únicamente en las clases que se ven afectadas directamente y reduciremos el riesgo de romper código que no está afectado.
- Nuestro código es más reusable ya que cada clase tiene una responsabilidad, si queremos acceder a cierta funcionalidad deberemos pasar por esa clase.

Abierto Cerrado

Índice

- 1 - Qué es el principio de abierto/cerrado
 - 1.1 - Ejemplo caso práctico
- 2 - Ejemplo principio abierto cerrado
 - 2.1 - Abierto cerrado utilizando herencia
 - 2.2 - Principio abierto cerrado utilizando composition

Este es el segundo post dentro de la serie de SOLID en el que vamos a ver la “O” que es el patrón abierto cerrado u *Open Closed Principle* (OCP) en inglés.

1 - Qué es el principio de abierto/cerrado

Este principio nos indica que una clase debe estar abierta para poder extenderla pero cerrada para su modificación.

Lo que quiere decir que cuando estamos escribiendo una clase y la ponemos en producción no debemos hacer cambios a esa clase. Si queremos cambiar lo que esa clase realiza, debe estar abierta para ser extendida si queremos cambiar su comportamiento.

Sin embargo tenemos una excepción y es que SI podemos cambiar una clase si encontramos un bug. En ese caso está permitido modificar la clase.

1.1 - Ejemplo caso práctico

Suena muy extremo, pero la realidad, y más ahora con los microservicios, es que si tenemos aplicaciones clientes que utilizan esa clase y nos dedicamos a cambiar su comportamiento puede afectar directamente a todos los clientes que la utilizan. Y si nadie respeta este principio podría, y se da, darse el caso que afecte a toda la cadena de clientes, con lo que múltiples proyectos o empresas deban cambiar su código.

Lo que implica que hacer un cambio simple en una clase puede llevar a un cambio mucho más gordo ya no solo en el proyecto que estás utilizando sino en todos los que lo referencian en alguna manera.

nota: para este post voy a utilizar el mismo ejemplo que en el post anterior.

2 - Ejemplo principio abierto cerrado

Este principio se basa básicamente en no permitir que una clase se vea modificada una vez está en producción sino que se extienda su funcionalidad o se modifique en las clases hijo que la implementan si utilizamos herencia o se modifique en otra clase si utilizamos *composition*.

2.1 - Abierto cerrado utilizando herencia

Tenemos dos formas de implementar este principio, la primera es por herencia.

La primera de las formas de conseguir esto es modificar los métodos de la clase padre con la palabra clave `virtual` el cual nos permite, como vimos en el post de las clases abstractas, sobrescribir un método por completo.

Si tu lenguaje es java, los métodos son virtual por defecto, así que no tienes que añadir nada.

```
public class Almacenamiento
{
    string path="C:/temp";
    public virtual void EscribirFichero(string titulo, string
contenido)
    {
        File.WriteAllText($"{path}/{titulo}.txt", contenido);
    }

    public virtual string LeerFichero(string titulo)
    {
        return File.ReadAllText($"{path}/{titulo}.txt");
    }
}
```

Al añadir la palabra clave virtual hemos modificado esta clase para ser abierta para extender, y podemos realizar lo mismo con la clase de los logs y de cache.

En caso de que no quieras extender la clase NO debes añadir la palabra clave `virtual`. Si ha de ser extendida ya se cambiará si hiciera falta en el futuro.

Por ejemplo, queremos cambiar nuestro sistema de logs en cierta parte crítica de nuestra aplicación, para que además de loguear el error en un fichero de log, lo almacene en una base de datos.

Para este escenario NO debemos modificar el sistema actual ya que ya esta en produccion. entonces lo que podemos hacer es basándonos en `Logging` crear una clase nueva :

```
public class DatabaseLogger : Logging
{
    private LogRepository logRepo { get; set; }
    public DatabaseLogger()
    {
        logRepo = new LogRepository();
    }

    public override void Fatal(string message, Exception e)
    {
        logRepo.AlmacenarError(message, e);
        base.Fatal(message, e);
    }
}
```

La clase nueva `DatabaseLogger` está sobrescribiendo la funcionalidad que tenemos sobre el método `Fatal` de la clase padre añadiendo el guardado en la base de datos.

Además vemos que sigue llamando a la clase padre para mantener la funcionalidad antigua además de la nueva, pero si removemos `base.Fatal()` del método dejará de guardar en el fichero de log y solo guardaría en la base de datos.

Podemos cambiar en el método principal para ver cómo todo sigue funcionando.

```
public class ArticulosServicio
{
    private readonly Cache _cache;
    private readonly DatabaseLogger _logging;
    private readonly Almacenamiento _almacenamiento;

    public ArticulosServicio()
    {
        _logging = new DatabaseLogger();
    }
}
```

```

        _almacenamiento = new Almacenamiento();
        _cache = new Cache();
    }

    public void GuardarArticulo(string contenido, string
titulo)
    {
        _logging.Info($"vamos a insertar el articulo
{titulo}");
        _almacenamiento.EscribirFichero(titulo, contenido);
        _logging.Info($"articulo {titulo} insertado");
        _cache.Add(titulo, contenido);
    }

    public string ConsultarArticulo(string titulo)
    {
        _logging.Info($"Consultar artículo {titulo}");

        string contenido = _cache.Get(titulo);
        if (!string.IsNullOrEmpty(contenido))
        {
            _logging.Info($"Artículo encontrado en la cache
{titulo}");
            return contenido;
        }

        _logging.Info($"buscar articulo en el sistema de
archivos {titulo}");

        return _almacenamiento.LeerFichero(titulo);
    }
}

```

2.2 - Principio abierto cerrado utilizando composition

La segunda forma y la más recomendable ya que suele dar lugar a menos errores es utilizar un patrón composition, el cual nos permite cambiar el funcionamiento en tiempo de ejecución. Para entender correctamente el funcionamiento deberemos entender sobre inyección de dependencias, con el que podemos inyectar un tipo en lugar de instanciarlo. Por ahora en este ejemplo, voy a instanciar las clases, pero deberían ser inyectadas.

El ejemplo es el mismo, estamos creando una clase y queremos que ahora además guarde en una base de datos.

Nuestra clase de `Logging` vuelve a su estado original donde no tenía la palabra clave `virtual` ya que ahora no vamos a sobrescribir los métodos.

```
public class Logging
{
    public void Info(string message)
    {
        Log.Information(message);
    }
    public void Error(string message, Exception e)
    {
        Log.Error(e, message);
    }
    public void Fatal(string message, Exception e)
    {
        Log.Fatal(e, message);
    }
}
```

Lo que debemos hacer es crear una nueva clase, la cual contiene una propiedad del tipo de nuestra clase original `Logging` que inicializamos en el constructor (deberíamos inyectarla) y como vemos creamos métodos que se llaman igual que los métodos de la clase original, sustituyendo el funcionamiento del que deseamos cambiar.

```
public class DatabaseLogger
{
    private readonly LogRepository logRepo;
    private readonly Logging loggingOriginal;
    public DatabaseLogger()
    {
        logRepo = new LogRepository();
        loggingOriginal = new Logging();
    }

    public void Info(string message) =>
loggingOriginal.Info(message);

    public void Error(string message, Exception e) =>
loggingOriginal.Error(message, e);
}
```

```

    public void Fatal(string message, Exception e)
    {
        logRepo.AlmacenarError(message, e);
        loggingOriginal.Fatal(message, e);
    }
}

```

La forma óptima de asegurarnos que estas clases van a contener todos los métodos que necesitamos es obligando a ambas clases a implementar una interfaz.

```

public interface ILogger
{
    void Info(string message);
    void Error(string message, Exception e);
    void Fatal(string message, Exception e);
}

public class DatabaseLogger : ILogger
{
    //Resto del código
}

public class Logging : ILogger
{
    //resto del código
}

```

Y ahora en nuestra clase principal podemos cambiar el código para que inyecte la interfaz `ILogger` y utilizando inyección de dependencias decidiremos si esa interfaz será referencia a `Logging` o `DatabaseLogger`.

```

public class ArticulosServicio
{
    private readonly Cache _cache;
    private readonly ILogger _logging;
    private readonly Almacenamiento _almacenamiento;

    public ArticulosServicio(ILogger logger)
    {
        _logging = logger;
        _almacenamiento = new Almacenamiento();
        _cache = new Cache();
    }
}

```

```

    public void GuardarArticulo(string contenido, string
titulo)
    {
        _logging.Info($"vamos a insertar el articulo
{titulo}");
        _almacenamiento.EscribirFichero(titulo, contenido);
        _logging.Info($"articulo {titulo} insertado");
        _cache.Add(titulo, contenido);
    }

    public string ConsultarArticulo(string titulo)
    {
        _logging.Info($"Consultar artículo {titulo}");

        string contenido = _cache.Get(titulo);
        if (!string.IsNullOrEmpty(contenido))
        {
            _logging.Info($"Artículo encontrado en la cache
{titulo}");
            return contenido;
        }

        _logging.Info($"buscar articulo en el sistema de
archivos {titulo}");

        return _almacenamiento.LeerFichero(titulo);
    }
}

```

Como podemos observar si hubiéramos implementado el principio abierto/cerrado desde el principio a la hora de extender la funcionalidad, no hubiéramos modificado la clase `ArticuloServicio` con lo que cumpliríamos con el principio de responsabilidad única visto en el post anterior.

Sustitución de Liskov

Índice

- Qué es el principio de substitución de Liskov
- Violación del principio de substitución de Liskov
- Ejemplo de principio de substitución Liskov
- Conclusión

Por ahora hemos visto los principios de responsabilidad única y de abierto cerrado en este post veremos el tercero de los principios solid el principio de sustitución de Liskov o Liskov substitution principle.

Qué es el principio de substitución de Liskov

El motivo por el que se denomina a este principio “principio de substitución de Liskov” es porque lo describió una señora llamada barbara Liskov en los 70, pero lo hizo de una forma muy matemática y es un paper que está casi todo en álgebra, por ello Robert C. Martin lo redefinió con la siguiente frase *“subtipos deben poder ser sustituidos por sus tipos base”*.

Aún así, sigue siendo una descripción muy abstracta y que no se entiende muy bien. en resumidas cuentas principio de substitución de Liskov nos habla sobre el polimorfismo. En el mundo real el uso de polimorfismo debe utilizarse con cautela ya que es una herramienta poderosa a par de complicada y nos puede hacer acabar en un callejón sin salida.

Una forma muy sencilla de comprender el principio de substitución de Liskov es mostrando un ejemplo de lo opuesto.



Como podemos ver ambos son patos, ambos hacen cuack pero resulta que necesita pilas, por lo que tenemos una abstracción errónea.

Cuando aplicamos el principio de sustitución de Liskov, no hay una forma general de hacerlo y es exclusivo de nuestra aplicación y del funcionamiento de la misma. Con la excepción de la siguiente norma “El sistema no debe romperse”.

En resumen: Si tenemos un cliente/servicio que llama a una interfaz A y cambiamos el cliente para llamar a la implementación B de la misma interfaz, el sistema no debe romperse.

Violación del principio de sustitución de Liskov

Una forma de ver que estamos violando el principio de sustitución de liskov es lanzar una excepción `NotSupportedException`. Obviamente esta situación no debe de estar en producción jamás.

```
public void Metodo1() {  
    throw new NotSupportedException();  
}
```

La forma más común de violar el principio de sustitución de Liskov es si intentamos extraer interfaces. Cuando tenemos una clase, y queremos extraer su interfaz, debemos de tener en cuenta cada uno de los métodos y propiedades que queremos extraer, y no hacerlo con todos ellos. Debemos extraer los únicos que queramos que sean privados.

Ejemplo de principio de substitución Liskov

Para el ejemplo vamos a modificar la clase `Almacenamiento` llamandola `AlmacenamientoFichero` y he añadido un método llamado `InformacionFichero` para poder ver el ejemplo de la violación del principio mencionado anteriormente.

```
public class AlmacenamientoFichero
{
    readonly string path = "C:/temp";

    public void Guardar(string titulo, string contenido)
    {
        File.WriteAllText($"{path}/{titulo}.txt", contenido);
    }

    public string Leer(string titulo)
    {
        return File.ReadAllText($"{path}/{titulo}.txt");
    }

    public FileInfo InformacionFichero(string titulo)
    {
        return new FileInfo($"{path}/{titulo}.txt");
    }
}
```

En nuestro ejemplo, queremos cambiar el sistema para que en vez de almacenar en un fichero guarde en una base de datos, por lo que creamos una clase la cual implementa los mismos métodos que `Almacenamiento fichero`.

```
public class AlmacenamientoSQL
```

```

{

    public void Guardar(string titulo, string contenido)

    {

    }

    public string Leer(string titulo)

    {

    }

    public FileInfo InformacionFichero(string titulo)

    {

    }

}

```

Como vemos estamos implementando los mismos métodos que en nuestra clase `AlmacenamientoFichero` pero ya no trabajamos por ficheros así que lo recomendable sería cambiar los nombres.

```

public class AlmacenamientoFichero

{

    readonly string path = "C:/temp";

    public void Guardar(string titulo, string contenido)

    {

        //código

    }

    public string Leer(string titulo)

    {

```

```
        //código
    }

    public FileInfo InformacionFichero(string titulo)
    {
        //código
    }
}

public class AlmacenamientoSQL
{
    public void Guardar(string titulo, string contenido)
    {
        //código
    }

    public string Leer(string titulo)
    {
        //código
    }

    public FileInfo InformacionFichero(string titulo)
    {
        throw new NotSupportedException();
    }
}
```

```
}
```

Para asegurarnos que estamos indicando todo correctamente lo suyo sería que extraigamos una interfaz de la primera de nuestras clases:

```
public interface IAlmacenamiento  
{  
    void Guardar(string titulo, string contenido);  
    string Leer(string titulo);  
    FileInfo InformacionFichero(string titulo);  
}
```

```
public class AlmacenamientoSQL : IAlmacenamiento{}  
public class AlmacenamientoFichero : IAlmacenamiento{}
```

Como vemos estamos indicando un método que no va a ser utilizado, debido a que cuando almacenamos en SQL no podemos coger la información del fichero. Esta es la violación indicada anteriormente. ese método no debería estar indicado en la interfaz.

Además vamos a lanzar una excepción en ese método pues nunca podremos ejecutarlo si estamos consultando la base de datos.

Por supuesto en nuestra clase principal `ArticulosServicio` debemos cambiar el almacenamiento por la interfaz `IAlmacenamiento` y he cambiado un poco el código para comprobar que el fichero existe antes de leerlo.

```
public class ArticulosServicio  
{  
    private readonly Cache _cache;  
    private readonly ILogger _logging;  
    private readonly IAlmacenamiento _almacenamiento;  
  
    public ArticulosServicio()
```

```
{

    _logging = new DatabaseLog();

    _almacenamiento = new AlmacenamientoFichero();

    _cache = new Cache();

}

public void GuardarArticulo(string contenido, string
titulo)

{

    _logging.Info($"vamos a insertar el articulo
{titulo}");

    _almacenamiento.Guardar(titulo, contenido);

    _logging.Info($"articulo {titulo} insertado");

    _cache.Add(titulo, contenido);

}

public string ConsultarArticulo(string titulo)

{

    _logging.Info($"Consultar artículo {titulo}");

    string contenido = _cache.Get(titulo);

    if (!string.IsNullOrEmpty(contenido))

    {

        _logging.Info($"Artículo encontrado en la cache
{titulo}");

        return contenido;

    }

}
```

```

    }

    _logging.Info($"buscar articulo en el sistema de
archivos {titulo}");

                                                                    if
(!_almacenamiento.InformacionFichero(titulo).Exists)

    return null;

    return _almacenamiento.Leer(titulo);

}

}

```

Como podemos arreglar el problema de nuestro método `LeerInformacionFichero` ya que no tiene sentido ninguno. Pero cómo hemos creado una mala implementación de nuestra interfaz no podemos quitar el método.

Como hemos indicado debido a este método, el código se romperá cuando sea ejecutado. Para arreglarlo, podemos cambiar la implementación. Podemos devolver una simulación de un fichero, o indicar un fichero falso, etc. pero esta solución no es apropiada, debido a que puede llevarnos a falsos positivos. Como vemos esta implementación de la interfaz es muy problemática, porque solo sirve cuando trabajamos con el sistema de ficheros.

Cómo arreglar el escenario?

Está claro que el objetivo final es quitar el método `LeerInformacionFichero` de la interfaz `IAlmacenamiento`.

Para ello, debemos cambiar la lógica que nuestra clase `AlmacenamientoFichero` está ejecutando, y debemos cambiar el método `Leer` de una forma que nos indique si ese fichero existe o no, antes de leerlo.

```

public class AlmacenamientoFichero : IAlmacenamiento
{

```



```

    readonly string path = "C:/temp";

    public void Guardar(string titulo, string contenido)
    {
        File.WriteAllText($"{path}/{titulo}.txt", contenido);
    }

    public string Leer(string titulo)
    {
        if (!InformacionFichero(titulo).Exists)
            return null;

        return File.ReadAllText($"{path}/{titulo}.txt");
    }

    private FileInfo InformacionFichero(string titulo)
    {
        return new FileInfo($"{path}/{titulo}.txt");
    }
}

```

Obviamente este ejemplo es muy sencillo, pero en un ejemplo real puede ser mucho más complejo.

```

public interface IAlmacenamiento
{
    void Guardar(string titulo, string contenido);

    string Leer(string titulo);
}

```

}

Conclusión

Mi consejo para evitar este tipo de situaciones es que cuando estés desarrollando y quieras hacer una clase, pienses en su implementación o en cómo podrías extraerlo en una interfaz en caso de que algún día quieras cambiar.

Los principios SOLID están altamente relacionados unos con otros y es muy difícil explicar uno al 100% sin mencionar o pasar por alto una mención a uno de los otros

Segregación de interfaces

Índice

- Qué es el principio de segregación de interfaces?
- Ejemplo de segregación de interfaces
- CONCLUSIÓN

En este post vamos a ver el principio de segregación de interfaces dentro de los principios SOLID.

Qué es el principio de segregación de interfaces?

Si has ojeado al post anterior donde hablo del principio de sustitución de Liskov, verías que en cierto punto nombramos las interfaces.

Y sobre ellas es donde nos vamos a centrar en este post, ya que las interfaces son las que nos ayudan a arreglar los problemas que nos puede causar una mala implementación del principio de sustitución de Liskov.

Como he indicado anteriormente que los principios solid interactúan entre ellos y por eso es muy difícil verlos exactamente por separado.

El principio de segregación de interfaces nos indica que *“Los clientes no deben ser forzados a depender de métodos que no utilizan”*

Una forma muy fácil de entender el ejemplo es pensar en quién es el dueño de la interfaz, y para mi, la forma más fácil es ver la interfaz como una dependencia.

Por ejemplo nuestra clase cliente utiliza una interfaz, esta interfaz es una dependencia. Lo que quiere decir que si nuestra clase cliente no necesita un método en concreto la interfaz no debe indicarlo.

Una interfaz debe ser definida por el cliente que consume la interfaz, y no por la clase que la implementa. Debido a este motivo, podemos tener interfaces más pequeñas, lo cual se hace mucho más sencillo de manejar.

Es muy común crear interfaces que tienen un único miembro, así que si eres nuevo en un trabajo, y ves muchas interfaces con un único miembro, no te preocupes, es algo común.

Ejemplo de segregación de interfaces

Para este ejemplo volvemos a utilizar el mismo código que hemos estado utilizando durante esta sección.

Como habrás podido comprobar la última parte del post anterior es el objetivo que queremos hacer en este básicamente es eliminar el método `InformacionFichero` y lo que queremos es deshacernos de él porque no lo utilizamos en nuestra clase de `AlmacenamientoSQL`; Aquí disponemos de la interfaz `IAlmacenamiento`

```
public interface IAlmacenamiento
{
    void Guardar(string titulo, string contenido);

    string Leer(string titulo);

    FileInfo InformacionFichero(string titulo);
}
```

Pero en esta ocasión lo haremos de una forma diferente, como he dicho debemos dejar al nuestro cliente en este caso la clase `ArticulosServicio` definir la interfaz.

Lo primero que vamos a hacer es mover la responsabilidad de leer la información del fichero que estamos comprobando a un método dentro de nuestra clase `ArticuloServicio`.

```
public FileInfo GetFicheroInformation(string titulo)
{
    return _almacenamiento.InformacionFichero(titulo);
}
```

Además debemos de cambiar el `if` para que lea de nuestro nuevo método.

```
if (!GetFicheroInformation(titulo).Exists)
{
    return null;
}
```

Esta acción es para ver cómo podemos fraccionar un poco más nuestra interfaz `IAlmacenamiento` ya que ahora tenemos un método que no está directamente relacionado, y así lo podemos utilizar como ejemplo, vamos a extraer esta funcionalidad en una nueva Interfaz, la cual va a tener un único método, el cual va a ser leer la información del fichero.

```
public interface IFicheroInformacion
{
    FileInfo GetInformacion(string titulo);
}
```

De esta forma podemos realizar un cambio el cual nos va a añadir esta nueva interfaz a nuestra clase `ArticuloServicio` y desde el método que acabamos de crear llamamos a nuestra nueva interfaz `IFicheroInformacion` y no a `IAlmacenamiento`

```
public class ArticulosServicio
{
    private readonly Cache _cache;
    private readonly ILogger _logging;
    private readonly IAlmacenamiento _almacenamiento;
    private readonly IFicheroInformacion _ficheroInformacion;

    public ArticulosServicio(IFicheroInformacion ficheroInformacion)
    {
        _logging = new DatabaseLog();
        _almacenamiento = new AlmacenamientoSQL();
        _cache = new Cache();
        _ficheroInformacion = ficheroInformacion;
    }
}
```

```
    public void GuardarArticulo(string contenido, string
titulo)
    {
        _logging.Info($"vamos a insertar el articulo
{titulo}");
        _almacenamiento.Guardar(titulo, contenido);
        _logging.Info($"articulo {titulo} insertado");
        _cache.Add(titulo, contenido);
    }
```

```
    public string ConsultarArticulo(string titulo)
    {
        _logging.Info($"Consultar artículo {titulo}");

        string contenido = _cache.Get(titulo);
        if (!string.IsNullOrEmpty(contenido))
        {
            _logging.Info($"Artículo encontrado en la cache
{titulo}");
            return contenido;
        }
```

```
        _logging.Info($"buscar articulo en el sistema de
archivos {titulo}");
```

```

        if (!GetFicheroInformation(titulo).Exists)

            return null;

        return _almacenamiento.Leer(titulo);

    }

    public FileInfo GetFicheroInformation(string titulo)
    {
        return _ficheroInformacion.GetInformacion(titulo);
    }
}

```

Basándonos en el caso de uso del post anterior, vamos a realizar la misma acción, vamos a utilizar nuestra clase `AlmacenamientoSQL` en vez de `AlmacenamientoFichero` por lo que conseguir la información del fichero carece de sentido.

Para ello lo que vamos a hacer es refactorizar la interfaz `IAlmacenamiento` para quitarle `InformacionFichero` ya que no es necesario para nuestro caso de uso.

```

public interface IAlmacenamiento
{
    void Guardar(string titulo, string contenido);

    string Leer(string titulo);
}

```

Al SQL tampoco necesitamos el método que acabamos de crear. así que debemos eliminarlo y eliminar también la referencia a la interfaz `IFicheroInformacion`.

Pero, ahora nos puede fallar en caso de que utilicemos la clase de escribir y guardar en el sistema de archivos (`AlmacenamientoFichero`). lo que tenemos que hacer

es mover esa interfaz como un parámetro en el constructor de nuestra clase `AlmacenamientoFichero` ya que es la clase cliente de `IFicheroInformacion`.

```
public class AlmacenamientoFichero : IAlmacenamiento
{
    readonly string path = "C:/temp";

    private readonly IFicheroInformacion _ficheroInformacion;

    public AlmacenamientoFichero(IFicheroInformacion
ficheroInformacion)
    {
        _ficheroInformacion = ficheroInformacion;
    }

    public void Guardar(string titulo, string contenido)
    {
        File.WriteAllText($"{path}/{titulo}.txt", contenido);
    }

    public string Leer(string titulo)
    {
        return File.ReadAllText($"{path}/{titulo}.txt");
    }

    public FileInfo InformacionFichero(string titulo)
```



```

    {

        if (!GetFicheroInformation(titulo).Exists)

            return null;

        return new FileInfo($"{path}/{titulo}.txt");

    }

    private FileInfo GetFicheroInformation(string titulo)

    {

        return _ficheroInformacion.GetInformacion(titulo);

    }

}

```

Debemos hacer lo mismo con todos los ejemplos.

CONCLUSIÓN

Una práctica muy común es tener interfaces con un único miembro, bueno a mi esto me parece una locura, porque depende cómo sea el sistema, no es necesario. por ejemplo, en el caso de los log, sería una interfaz para guardar el log como error, otra para guardar el log como fatal, etc.

Esto tiene una ventaja que es muy fácil de extender pero, hay que hacer las cosas con cabeza, si extendemos el caso de uso a un proyecto donde llevamos un par de años trabajando, en caso de hacer una interfaz por método, podemos acabar con millones de interfaces, así que lo mejor, en mi opinión es agruparlas por funcionalidad. siempre y cuando no sean mega-interfaces.

Inversión de dependencias

Índice

- 1 - Qué es el principio de inversión de dependencias.
- 2 - Utilizar inversión de dependencias
 - 3 - Inyección de dependencias en .NET
 - 3.1 - Inyección de dependencias en ASP.NET Core
 - 3.2 - Inyección de dependencias en una aplicación de consola.
- 4 - Beneficios de implementar inversión de dependencias
- Conclusión

Hemos llegado ya al último punto de los principios SOLID el cual es la inversión de dependencias.

En .NET cumplimos este principio utilizando inyección de dependencias. Lo más normal es ver la inversión de dependencias en aplicaciones web, pero también podemos utilizarla en aplicaciones de consola.

La única diferencia, es que desde .NET Core, cuando creamos una aplicación web, la clase que va a contener las dependencias viene creado por defecto y es muy fácil añadir nuevas dependencias en el, mientras que en una aplicación de consola, hay que crearlo manualmente.

Más adelante veremos cómo funcionan ambas.

1 - Qué es el principio de inversión de dependencias.

El principio de inversión de dependencias es utilizado para permitirnos crear clases que están acopladas entre sí de una forma sencilla.

Este principio, nos indica que debemos separar la lógica en módulos o servicios que puedan ser reutilizados y en caso de tener que modificarlos, únicamente esos servicios se verán afectados.

De esta forma quitamos también parte de la responsabilidad sobre nuestra clase.

Los principios SOLID están relacionados todos entre sí, el principio de inversión de dependencias se nutre de la interfaces que hemos visto a lo largo de los módulos previos.

2 - Utilizar inversión de dependencias

Como he indicado antes, en .NET la forma de utilizar inversión de dependencias es a través de la inyección de dependencias.

Y lo que tenemos que hacer, como su nombre indica es Inyectar estas dependencias en nuestros servicios.

Para ello lo realizamos a través del constructor.

```
public class AlmacenamientoFichero : IAlmacenamiento
{
    readonly IFicheroInformacion _ficheroInformacion;

    public AlmacenamientoFichero(IFicheroInformacion
ficheroInformacion)
    {
        _ficheroInformacion = ficheroInformacion;
    }
    //Resto de métodos
}
```

Como vemos estamos inyectando a través del constructor la interfaz `IFicheroInformacion` la cual contiene el método el cual nos permite consultar la información de un fichero. Pero “no sabemos” cómo está implementado, o las dependencias de esa otra clase, ya que está abstraída a través de una interfaz.

Como puedes observar ahora el constructor recibe un parámetro, y cuando instanciamos la clase, deberíamos pasarlo, pero no lo haremos, ya que todo nuestro código va a estar cortado por el principio de inversión de dependencias.

Por lo que debemos inyectar las dependencias en todas nuestras clases.

```
public class ArticulosServicio
{
    private readonly Cache _cache;
    private readonly ILogger _logging;
    private readonly IAlmacenamiento _almacenamiento;
```

```

    public ArticulosServicio(IAlmacenamiento almacenamiento,
        ILogger logging, Cache cache)
    {
        _logging = logging;
        _almacenamiento = almacenamiento;
        _cache = cache;
    }

    //resto de métodos
}

```

3 - Inyección de dependencias en .NET

Debido a que realizaré un post más en detalle sobre inyección de dependencias en el futuro. voy a pasar solo por encima, para que nuestro código entienda a qué clase hace referencia cada interfaz.

3.1 - Inyeccion de dependencias en ASP.NET Core

Desde la salida de Net Core tenemos un contenedor de dependencias para asp.net en el core del sistema, al cual podemos acceder desde el fichero/clase `startup.cs` en el método `ConfigureServices`. simplemente debemos indicar `services.AddScoped<Interface, Class>`.

De esta forma cuando el código llegue a un constructor que contiene dicha interfaz, sabrá a qué clase le hace referencia.

```

services.AddScoped<IAlmacenamiento, AlmacenamientoSQL>()
    .AddScoped<ILogger, Logging>()
    .AddScoped<DBRepository>()
    .AddScoped<Cache>();

```

3.2 - Inyección de dependencias en una aplicación de consola.

El objetivo es el mismo que en el ejemplo de ASP.NET pero en este caso debemos construir el contenedor de las dependencias nosotros mismos.

Para ello debemos utilizar los dos siguientes paquetes:

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
```

De esta forma tendremos acceso a la clase `ServiceCollection` la cual nos permite construir nuestras dependencias con el método `BuildServiceProvider`.

```
//Base
var serviceProvider = new ServiceCollection()
    .AddScoped<Interface, Claseservicio>()
    .BuildServiceProvider();

var servicio = serviceProvider
    .GetService<Interface>();

//Ejemplo en nuestro código
var serviceProvider = new ServiceCollection()
    .AddScoped<IAlmacenamiento, AlmacenamientoFichero>()
    .BuildServiceProvider();

var servicio = serviceProvider
    .GetService<IAlmacenamiento>();
```

nota: veremos la diferencia entre scoped, singleton, transient en otro post.

4 - Beneficios de implementar inversión de dependencias

Gracias a la inversión de dependencias podemos crear clases más pequeñas e independientes, permitiéndonos además, reutilizar lógica de una forma muy sencilla. Lo cual es su mayor ventaja en mi opinión.

Agregar inversión de dependencias nos proporciona flexibilidad y estabilidad a la hora de desarrollar nuestras aplicaciones. Ya que estaremos más seguros a la hora de ampliar funcionalidades, etc.

Como última gran ventaja, utilizar inversión de dependencias nos permite la creación de test unitarios con mucha más facilidad, permitiéndonos crear mock de las mismas.

Conclusión

Finalmente decir que sí, debemos implementar inversión de dependencias en todos nuestros proyectos ya que eso nos dará servicios más reusables y fáciles de implementar en otros servicios.

Con este post hemos visto todos los principios SOLID, los cuales siguiendolos a rajatabla nos permiten crear aplicaciones robustas que van a estar muy bien creadas para adaptarlas a futuros cambios.