

# Tema 7 parte 3

[Descargar estos apuntes](#)

## Índice

1. [Índice](#)
2. [Gestión de errores en POO](#)
  1. [Programación defensiva](#)
  2. [Excepciones](#)
    1. [Excepciones en C#](#)
      1. [Generación de excepciones](#)
      2. [Captura y control de excepciones](#)
      3. [Liberando recursos con finally](#)

# Gestión de errores en POO

## Programación defensiva

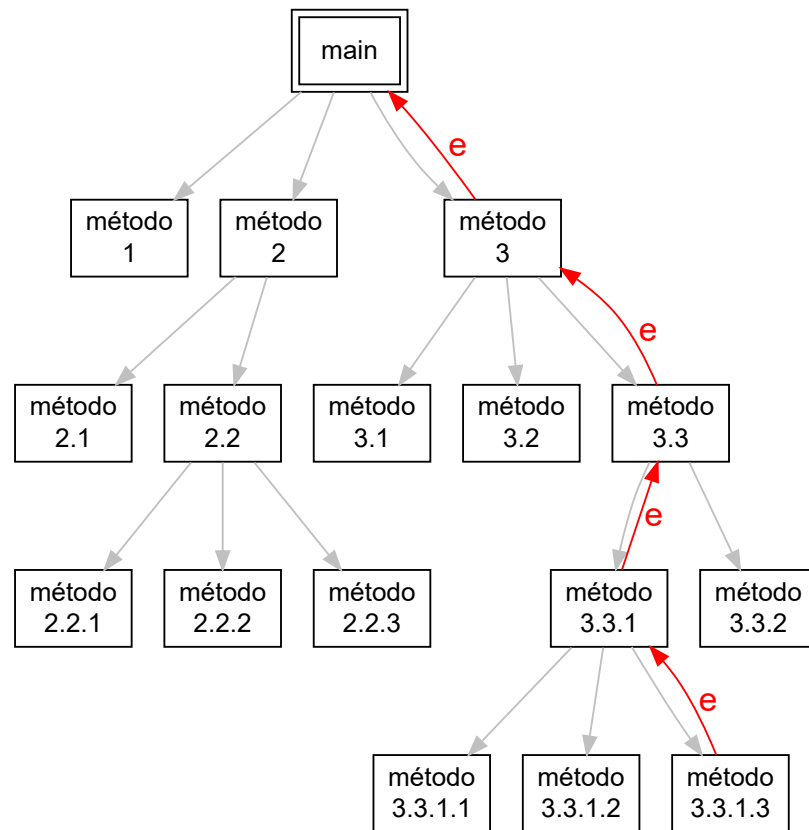
- En este tema vamos a explicar el

[Steve McConnell](#)

[Steve McConnell](#)

# Excepciones

- **Definición:** Es la forma en que los lenguajes orientados a objetos realizan el control de errores.
- **Frente al control de errores en la programación estructurada tradicional nos ofrecen:**
  - **Claridad y simplicidad**, ya que **evitamos lógica adicional** del caso de error.  
Por ejemplo, evitando que un método devuelva un error como parámetro a través de la pila de llamadas como sucedía en la **programación estructurada**.



En el diagrama de arriba podemos ver una situación típica de la programación estructurada donde si se producía un error en un módulo básico como el **módulo 3313()** del ejemplo. Este tenía que propagarse hacia arriba a través de todos los módulos devolviéndose en todas las llamadas. Esto hacía, que muchos interfaces tuviesen que devolver información adicional para el error con un interfaz similar al siguiente.

```
void Metodo3(..., out bool error, out string mensajeError) {...}
void Metodo33(..., out bool error, out string mensajeError) {...}
void Metodo331(..., out bool error, out string mensajeError) {...}
void Metodo3313(..., out bool error, out string mensajeError) {...}
```

- **Tratamiento asegurado** de errores.

## Excepciones en C#

- Todas derivan de la clase `System.Exception`
- Existen algunas ya muchas **predefinidas**.
- Podemos definir **excepciones propias** mediante el mecanismo de herencia.

### Generación de excepciones

Utilizaremos la instrucción **throw** → `throw new <TipoExcepción>(...);`

Veamos un **ejemplo** en el que vamos a usar la excepción ya predefinida **ArgumentNullException** que me servirá para detectar aquellos tipos referencia que sean `null` antes de usarlos.

Por ejemplo, en el método **Diagnostico** del bloque anterior que creamos para la clase **Sicologo** podemos comprobar que el objeto **Paciente** que nos llega como parámetro no sea `null`

```
class Sicologo
{
    ...
    private static string Diagnostico(Paciente p)
    {
        6         if (p is null)
        7             throw new ArgumentNullException("paciente");
        return p switch
        {
            PacienteAlegre _ => $"{p.GetNombre()} le veo estupendamente. Enhorabuena!! no
            PacienteTriste _ => $"{p.GetNombre()} tome fluoxetina 20mg y vuelva en un mes
            _ => $"{p.GetNombre()} déjeme que estudie un poco más su caso y vuelva la ser
        };
    }
    ...
}
```

Nadie nos asegura que cuando se llame al método Diagnóstico no se pase un `null` como parámetro. Si eso sucediese la instrucción `p.GetNombre()` generaría la excepción del sistema **NullReferenceException** finalizando el programa.

Este es uno de los errores más típicos en muchos lenguajes y nosotros lo filtramos generando una excepción de **ArgumentNullException** que podremos gestionar de una manera más específica.

👉 Fíjate, que al lanzar la excepción realmente lo que hacemos es crear una referencia objeto en memoria con `new ArgumentNullException("paciente");`. Este objeto contendrá información sobre el error y una referencia al mismo será pasada al punto donde controlemos el error.

## Captura y control de excepciones

En la gran mayoría de lenguajes orientados a objetos se realiza con las palabras reservadas **try** y **catch**.

Una sintaxis básica de este tipo de estructura podría ser...

```
try
{
    // Código del que quiero controlar errores.
}
catch (TipoDeLaExcepciónACapturar e) when (<expresión con e>)
{
    // Tratamiento del error o excepción del tipo TipoDeLaExcepciónACapturar
}
```

Donde **catch** es un punto de control de errores, en el cual el identificador **e** será opcional y lo definiremos solo si lo vamos a usar dentro del bloque o en una condición **when**.

La cláusula **when** será **opcional**.

**Nota:** No podremos, poner ningún bloque **catch** que no esté asociado a uno **try**

Veamos un ejemplo de sintaxis muy **simple** para ver cómo funcionan. Supongamos el siguiente código...

```
static void Main()
{
    Console.Write("Introduce un número real: ");
    textoNumero = Console.ReadLine();
    double n = double.Parse(textoNumero);
    Console.WriteLine($"Tu número es {n:G}");
}
```


Si lo ejecutamos e introducimos **25** obtendremos...

```
Introduce un número real: 25
Tu número es 25
```

Pero si introducimos **veinticinco** obtendremos...

```
Introduce un número real: veinticinco
Unhandled exception. System.FormatException: Input string was not in a correct format.
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)
   at System.Double.Parse(String s)
   at EjemploExcepciones.EjemploExcepciones.Main() in C:\ejemplo\Program.cs:line 15
```

Si nos fijamos `double.Parse(textoNumero);` ha generado una excepción de tipo `FormatException` porque no ha podido pasar la cadena de entrada `"veinticinco"` a `double`.

 **Importante:** Para saber que errores/excepciones puede generar una llamada a un método de. Además, de ver el error que produce como en el caso anterior. Lo más normal es consultar la [documentación del método](#) y en apartado **Excepciones** nos lo indicará.

La excepción ha sido capturada por el Runtime de C# nos la ha mostrado y ha finalizado la ejecución. A efectos prácticos, es cómo si hubiera un bloque `try - catch` que englobara todo nuestro código y que *'barriera'* o capturase cualquier error/excepción que se pudiera producir aunque nosotros no lo hagamos.

Veamos cómo sería el código siguiendo la sintaxis del `try - catch` para capturar la excepción

`FormatException`

```
static void Main()
{
    // Tendremos que definir la variable textoNumero fuera del bloque try
    // si queremos que sea accesible desde el bloque catch.
5   string textoNumero = default;
6   try
    {
        Console.Write("Introduce un número real: ");
        textoNumero = Console.ReadLine();
        double n = double.Parse(textoNumero);
        Console.WriteLine($"Tu número es {n}");
    }
13  catch (FormatException)
    {
        // Como no usamos e no lo declaramos.
16  Console.WriteLine($"Lo siento, el valor '{textoNumero}' no es Real.");
    }
}
```

Si lo ejecutamos e introducimos **'veinticinco'** ahora obtendremos...


```
Introduce un número real: veinticinco
Lo siento, el valor 'veinticinco' no es Real.
```

También tendremos la opción de mostrar el mensaje de error que devuelven las BCL a través del **objeto** `e` que contiene la información de la excepción y que se creó al generarse la misma en el `throw`.

```
static void Main()
{
    try
    {
        Console.Write("Introduce un número real: ");
        double n = double.Parse(Console.ReadLine());
        Console.WriteLine($"Tu número es {n}");
    }
    catch (FormatException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Si lo ejecutamos e introducimos **'veinticinco'** ahora obtendremos...

```
Introduce un número real: veinticinco
Input string was not in a correct format.
```

 **Importante:** Al especificar que capturamos solo la excepción **FormatException** solo se entrará en este bloque catch si se produce la misma. Por tanto, cualquier otro error/excepción será capturado por el Runtime.

## Capturando excepciones diferentes

Si `catch (FormatException e)` solo captura las excepciones de formato de entrada incorrecto.  
**¿Cómo haremos para controlar diferentes tipos errores/excepciones?**

Supongamos el siguiente **ejemplo** donde hemos creado un método **Divide** que genera una excepción **DivideByZeroException** al intentar dividir por cero. Pero si nos fijamos en el **Main** solo gestionamos **FormatException**.

El programa será un bucle infinito que siempre nos pedirá 2 números e intentará dividirlos.

```
class EjemploExcepciones
{
    static double Divide(double numerador, double divisor)
    {
        if (divisor < 1e-5)
            throw new DivideByZeroException();
        return numerador / divisor;
    }

    static void Main()
    {
        while(true)
        {
            try
            {
                Console.Write("Introduce el numerador: ");
                double numerador = double.Parse(Console.ReadLine());
                Console.Write("Introduce el divisor: ");
                double divisor = double.Parse(Console.ReadLine());
                Console.WriteLine($"La división es {Divide(numerador, divisor)}");
            }
            catch (FormatException)
            {
                Console.WriteLine($"Has introducido un valor que no es un número real.");
            }
        }
    }
}
```

Al ejecutar el código e intentar dividir por cero obtendremos...

```
Introduce el numerador: 4
Introduce el divisor: 0
Unhandled exception. System.DivideByZeroException: Attempted to divide by zero.
   at EjemploExcepciones.EjemploExcepciones.Divide(Double numerador, Double divisor) in C:\ej
   at EjemploExcepciones.EjemploExcepciones.Main() in C:\ejemplo\Program.cs:line 23
```



Si nos fijamos el programa finaliza porque la excepción es capturada por el catch del Runtime que está fuera del bucle infinito.


Para capturar también este error, lo que haremos es añadir **dos bloques catch consecutivos** para el mismo bloque **try**.

```
try
{
    //...
}
catch (FormatException)
{
    Console.WriteLine("Has introducido un valor que no es un número real.");
}
catch (DivideByZeroException)
{
    Console.WriteLine("No se puede dividir por cero.");
}
```

Al ejecutar ahora con la misma entrada obtendremos...

```
Introduce el numerador: 4
Introduce el divisor: 0
3 No se puede dividir por cero.
Introduce el numerador: 4
Introduce el divisor: cero
6 Has introducido un valor que no es un número real.
Introduce el numerador: 4
Introduce el divisor: 2
La división es 2
Introduce el numerador: ...
```

Fíjate que ahora capturamos '*nosotros*' los 2 errores y además nos '*recuperamos*' porque estamos en un bucle infinito.

 Importante: Si el **tipo** del primer bloque **catch** es una **superclase** del tipo del segundo. **El segundo bloque catch nunca se ejecutará** y además nos avisará con un **error de compilación**.

Por ejemplo el siguiente código ...

```
try
{
    //...
}
5 catch (Exception)
{
    Console.WriteLine("Hay un error.");
}
catch (FormatException)
{
    Console.WriteLine("Has introducido un valor que no es un número real.");
}
catch (DivideByZeroException)
{
    Console.WriteLine("No se puede dividir por cero.");
}
```

Generará el siguiente error ....

✗ **"Una cláusula catch previa ya detecta todas las excepciones de este tipo o de tipo superior ('Exception')"**

ya que todas las excepciones heredan de `Exception` y por tanto `FormatException` y `DivideByZeroException` lo hacen. Lo cual implicaría que es un código inalcanzable porque cualquier error entraría primero por el primer `catch`.

👉 **Importante:** De lo anterior se deduce que siempre podremos los `catch` excepciones más concretas primero y a continuación las excepciones más generales.

Si añadimos un único bloque `catch (Exception e)` en el `Main` controlaríamos cualquier error/excepción que se produjese y seguiríamos con nuestro bucle infinito.

⚠ **Aviso:** Microsoft no nos recomienda hacerlo\*\* fuera del `Main`, indicándonos en su [documentación](#) que no es una buena práctica.

Por tanto, si probamos el siguiente código....

```
static void Main()
{
    while (true)
    {
        try
        {
            Console.Write("Introduce el numerador: ");
            double numerador = double.Parse(Console.ReadLine());
            Console.Write("Introduce el divisor: ");
            double divisor = double.Parse(Console.ReadLine());
            Console.WriteLine($"La división es {Divide(numerador, divisor)}");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Introduce el numerador: 4

Introduce el divisor: 0

3 Attempted to divide by zero.

Introduce el numerador: 4

Introduce el divisor: cero

6 Input string was not in a correct format.

Introduce el numerador: 4

Introduce el divisor: 2

La división es 2

Introduce el numerador: ...

## Liberando recursos con finally

En ocasiones se pueden dar casos en los que queramos, **independientemente de si ha producido un error o no**, hacer algo siempre después de un bloque de instrucciones. Por ejemplo:

- Liberar un recurso de memoria asociado a un fichero cómo una imagen o una fuente.
- Cerrar una conexión remota o a una base de datos, un fichero abierto por el sistema, etc.
- Parar algún proceso en paralelo iniciado.

Para eso tendremos un bloque **finally** ...

- El bloque **finally** es **opcional** y debe estar **asociado a un bloque try**.
- Libera o cierra los recursos usados dentro del un bloque **try** al que está asociado.
- El bloque **finally** **se ejecutará siempre**, tanto si ha ido bien el bloque try, como si ha entrado por alguno de los bloques catch.
- Se **ejecutará en último lugar** respecto a sus bloques try-catch asociados en su ámbito, esto es, después del **try** si todo ha ido bien o después del **catch** correspondiente.

👉 Se puede tener un bloque **try** seguido de un **finally** (sin bloques catch), y realizar la captura de la excepción en un catch de ámbito más externo o incluso por el Runtime. En este caso, el finally se ejecutará siempre y antes que el catch del ámbito superior.

Veámoslo en un simples ejemplos de código esquematizado.

```
// Si correcto: BT -> BF
// Si error:    BT -> BC -> BF
try
{
    // BT
}
catch(...)
{
    // BC
}
finally
{
    // BF
}
```

```
// Si correcto: BT1 -> BT2
// Si error:    BT1 -> BT2 -> BF2 -> BC1
try
{
    // BT1
    try
    {
        // BT2
    }
    finally
    {
        // BF2
    }
}
catch(...)
{
    // BC1
}
```

## Ejemplo práctico de uso de finally

Vemos un ejemplo típico de uso de `finally` que se nos va a dar en el futuro.

**Nota:** Cómo aún no hemos visto E/S con ficheros, lee los comentarios para entender que estamos haciendo en cada bloque y el porqué de cada uno. Este ejemplo **solo pretende aplicar un esquema práctico de uso real de try - catch - finally** y no abordar la lectura de ficheros en este punto del curso.

```
public static void LeeFichero(int posicion)
{
    char[] buffer = new char[10];
    StreamReader streamReader = default;
    string ruta = @"c:\test.txt";
    try
    {
        // Intentamos abrir un fichero
        streamReader = new StreamReader(ruta);
        // Si llegamos aquí le hemos dicho al SO que ocupamos un recurso
        // (En nuestro caso un fichero) y por tanto deberemos liberarlo
        // en el bloque finally, tanto si todo va bien como sino.

        // Intentamos leer el fichero (podemos obtener una excepción de lectura)
        streamReader.ReadBlock(buffer, posicion, buffer.Length);
    }
    catch (FileNotFoundException) // Capturamos el error de que el fichero no existe.
    {
        Console.WriteLine($"El fichero {ruta} no existe.");
    }
    catch (IOException e) // Capturamos un error de lectura devuelto por el dispositivo.
    {
        Console.WriteLine($"Error leyendo de {ruta}. {e.Message}");
    }
    finally // ESTE BLOQUE SE EJECUTARÁ SIEMPRE
    {
        // En el caso de que el fichero no exista, llegaremos aquí desde el
        // catch (FileNotFoundException). Por tanto, la referencia streamReader será null
        // y no deberemos liberar el recurso con Close().
        if (streamReader != null)
        {
            // En el caso de que el fichero exista, llegaremos aquí:
            // 1. Después de haber leído correctamente el fichero.
            // 2. Tras una excepción de lectura capturada en el catch (IOException e) y en
            // ambos casos deberemos liberar el recurso (el fichero) indicándoselo al SO.
            streamReader.Close();
        }
    }
}
```