

# Tema 7 parte 3

[Descargar estos apuntes](#)

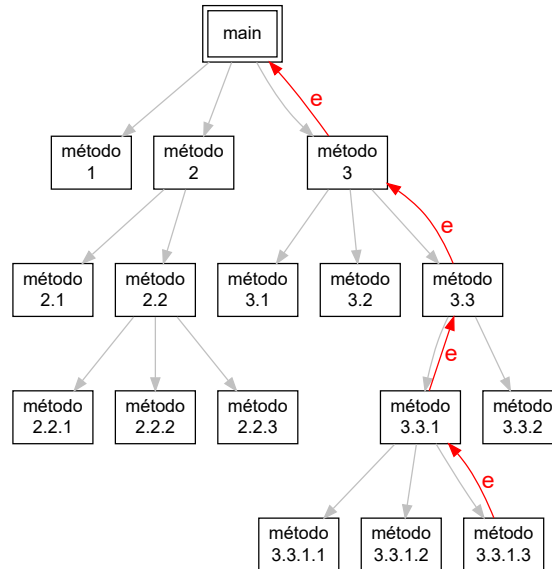
## Índice

1. [Gestión de errores en POO](#)
  1. [Excepciones Conceptos Previos](#)
  2. [Excepciones en C#](#)
    1. [Generación de excepciones](#)
      1. [Captura y control de excepciones](#)
      2. [Capturando excepciones diferentes](#)
    2. [Liberando recursos con finally](#)
      1. [Ejemplo práctico de uso de finally](#)
    3. [Creando nuestras propias excepciones](#)
      1. [Ejemplo de creación de una excepción propia](#)
    4. [Encadenamiento de excepciones \(Avanzado\)](#)
      1. [Caso de uso de encadenamiento de excepciones](#)
    5. [🔗 Tips de uso de excepciones en .NET](#)
      1. [Excepciones estándar en .NET](#)

# Gestión de errores en POO

## Excepciones Conceptos Previos

- **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 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 necesita más terapia.",
            PacienteTriste _ => $"{p.GetNombre()} tome fluoxetina 20mg y vuelva en un mes.",
            _ => $"{p.GetNombre()} déjeme que estudie un poco más su caso y vuelva la semana que viene."
        };
    }
    ...
}
```

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**.

 **Importante:** 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, además de ver el error que produce, como en el caso anterior, lo más normal es consultar la [documentación del método](#). En el 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)
    {
11  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 principal nos pedirá 2 números e intentará dividirlos y si no puede finaliza.

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

        static void Main()
        {
            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:\ejemplo\Program.cs:line 11
   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.

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...

```
Ejecución 1:
  Introduce el numerador: 4
  Introduce el divisor: 0
4  No se puede dividir por cero.

Ejecución 2:
  Introduce el numerador: 4
  Introduce el divisor: cero
9  Has introducido un valor que no es un número real.

Ejecución 3:
  Introduce el numerador: 4
  Introduce el divisor: 2
  La división es 2
```

Fíjate que ahora capturamos 'nosotros' los 2 errores.

👉 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()
{
    try
    {
5      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);
    }
}
```

Ejecución 1:

Introduce el numerador: 4

Introduce el divisor: 0

4 Attempted to divide by zero.

Ejecución 2:

Introduce el numerador: 4

Introduce el divisor: cero

9 Input string was not in a correct format.

Ejecución 3:

Introduce el numerador: 4

Introduce el divisor: 2

La división es 2



## 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();
        }
    }
}
```

## Creando nuestras propias excepciones

En ocasiones nos interesará crear nuestras propias excepciones para **capturar errores de tipos de excepciones específicos** en nuestras clases, a la hora de pasar test por ejemplo.

En este caso las definiremos el tipo de la excepción nosotros y heredando de una excepción ya creada si queremos concertarla más o de la clase base para excepciones `System.Exception`.

### Ejemplo de creación de una excepción propia

El convenio en C# es acabar el nombre del tipo de nuestra excepción con el sufijo `Exception` y en este ejemplo lo hemos hecho.

```
class DepartamentoException : Exception
{
    public DepartamentoException(string message) : base (message) {}
}
```

- En el código de arriba hemos creado una excepción `ExcepcionDepartamento` que usaré para saber cuando no he controlado algo en los departamentos de mi compañía.
- Ahora supongamos un método para imprimir nóminas de un departamento, donde no hemos contemplado uno de reciente creación como en el siguiente código.

```
public class Departamento
{
    enum Departamentos { Contable, Desarrollo, Marketing };
    static void ImprimeNominas(Departamentos departamento)
    {
        switch (departamento)
        {
            case Departamentos.Contable:
                Console.WriteLine("Imprimiendo nóminas contabilidad.");
                break;
            case Departamentos.Desarrollo:
                Console.WriteLine("Imprimiendo nóminas Desarrollo.");
                break;
            default:
                throw new DepartamentoException(
                    $"No se pueden imprimir nóminas de este departamento de {departamento}.");
        }
    }
}
```

Otra posibilidad es crear excepciones personalizadas para una clase, podría ser definir la excepción de forma anidada dentro de la clase.

```
public class Departamento
{
    public class NoGestionadoException : Exception
    {
        public NoGestionadoException(string message) : base(message) { }
    }

    enum Departamentos { Contable, Desarrollo, Marketing };
    static void ImprimeNominas(Departamentos departamento)
    {
        switch (departamento)
        {
            case Departamentos.Contable:
                Console.WriteLine("Imprimiendo nóminas contabilidad."); break;
            case Departamentos.Desarrollo:
                Console.WriteLine("Imprimiendo nóminas contabilidad."); break;
            default:
                throw new Departamento.NoGetionadoException(
                    $"No se pueden imprimir nóminas de este departamento de {departamento}.");
        }
    }
}
```

Fíjate que al estar anidada en el tipo `Departamento.NoGestionadoException` queda claro que es una excepción relacionada con la clase `Departamento`.

## Encadenamiento de excepciones (Avanzado)

En ocasiones, puedo tener la necesidad de crear un bloque `catch` para capturar una excepción en un ámbito, añadir un **mensaje específico para ese ámbito** y posteriormente **relanzarla** para ser capturada en otro ámbito superior. Además, este proceso se puede repetir de forma sucesiva.

La mayoría de constructores de excepciones de las BCL, admiten una sobrecarga con el parámetro **Exception innerException**. (a NULL por defecto).

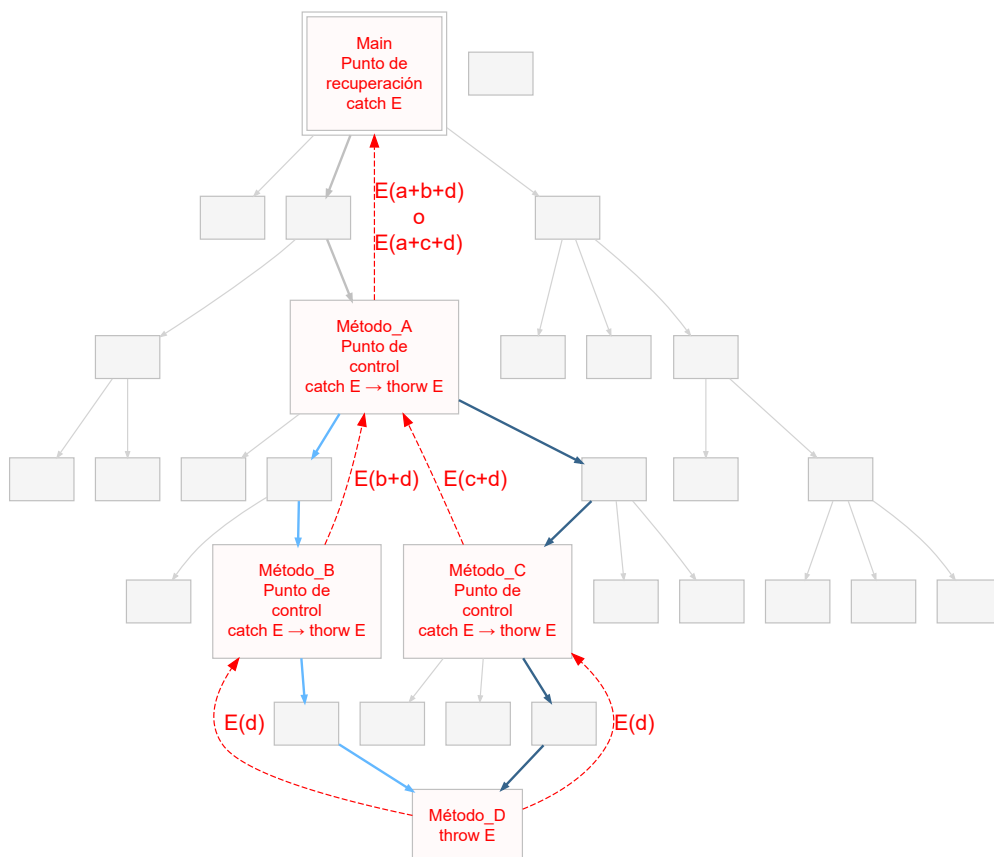
```
try
{
    // Ámbito del try
}
catch (ExcepcionTipoA e)
{
    // En este punto de control de excepciones capturamos la excepción
    // del tipo ExcepcionTipoA y añadimos un mensaje específico de lo
    // que estamos haciendo en el ámbito del try y la relanzamos a un
    // catch en un ámbito superior donde puede haber otro punto
    // de control o de recuperación de excepciones.
    throw new ExcepcionTipoA("mensaje específico en ámbito del try", e);
}
```

Fíjate que en al hacer `throw new ExcepcionTipoA("mensaje específico en ámbito del try", e);` estamos creando otro objeto `ExcepcionTipoA` al que le pasamos una referencia al objeto `ExcepcionTipoA e`, donde nos llegaba la información del error.

Esto me permitirá recorrer todos los objetos excepción encadenados en el orden que se han ido relanzando y así acceder a mensajes específicos en cada ámbito.

### Caso de uso de encadenamiento de excepciones

Veamos un caso de uso de encadenamiento de excepciones a través del siguiente esquema, donde desde el Main se iría llamando a métodos de diferentes objetos para realizar una tarea. Tenemos un `Método_D` que es usado en **dos ramas diferentes de la pila de llamadas** y que puede generar la excepción/error `E`



Si sólo tuviera un punto de recuperación de errores (catch) en el **Main** al producirse el la excepción **E(d)** en **Método\_D**, no sabría por donde se ha producido.

Sin embargo, si vamos añadiendo '**catch de control**' donde relanzamos la misma excepción, añadiendo una capa de información de que proceso estamos haciendo en ese momento. El '**catch de control**' en el **Método\_A**, si nos fijamos, puede relanzar la excepción añadiendo información de cada uno de los '**catch de control**' por donde ha pasado **E(a+b+d)** o **E(a+c+d)** y saber de forma más específica en un '**catch de mensaje o recuperación**' del **Main**, por donde se ha producido el error.

Una posible implementación esquemática del la situación descrita en el diagrama, podría ser la siguiente:

```
namespace EsquemaExcepcionesEncadenadas {
    class E : Exception {
        public E(string message) : base(message) {}
        public E(string message, Exception innerException) : base(message, innerException){ }
    }
    class JerarquiaDeLlamadas {
        static void Metodo_D()
        {
            // En algún punto de pude lanzar la excepción
            throw new E("D");
        }
        static void Metodo_C() {
            try {
                // En algún punto de este código en su jerarquía de llamadas,
                // se llamará a Metodo_D()
            }
            catch(E e) // Catch de control en C
            {
                throw new E("C", e);
            }
        }
        static void Metodo_B() {
            try {
                // En algún punto de este código en su jerarquía de llamadas,
                // se llamará a Metodo_D()
            }
            catch(E e) // Catch de control en B
            {
                throw new E("B", e);
            }
        }
        static void Metodo_A() {
            try {
                // En algún punto de este código en su jerarquía de llamadas,
                // o bien se llamará a Metodo_B() o a Metodo_C()
            }
            catch(E e) // Catch de control en A
            {
                throw new E("A", e);
            }
        }
        static void Main() {
            try {
                // En algún punto de este código en su jerarquía de llamadas,
                // se llamará a Metodo_A()
            }
            catch (E e) // Catch de mensaje o recuperación en Main
            {
                StringBuilder mensaje = new StringBuilder();
                while (e != null)
                {
                    mensaje.Append(e.Message);
                    e = e.InnerException as E;
                    if (e != null)
                        mensaje.Append("+");
                }
                Console.WriteLine(mensaje);
            }
        }
    }
}
```

## 💡 Tips de uso de excepciones en .NET

En ocasiones usar adecuadamente las excepciones es complicado incluso para programadores **experimentados**, y se han de establecer convenios y patrones en los equipos de desarrollo.

Aunque en este curso no vamos a tratar más que los conceptos básicos. Si deseas profundizar, en el siguiente enlace puedes encontrar una serie de instrucciones para la correcta generación de excepciones descritas en la documentación de Microsoft. Que además de ser una lectura complementaria interesante, puede ser extrapolable a otros lenguajes.

- [Instrucciones para la correcta generación de excepciones.](#)

De entre los tips del enlace anterior, destacaremos un uso incorrecto y muy común en programadores noveles.

### ✗ No utilice excepciones para el flujo de control normal, si es posible.

Veámos **un ejemplo similar al del inicio** del tema, donde queríamos pedir dos números y mostrar su división.

Un código algo más modularizado, pero sin gestión de excepciones sería el siguiente:

```
static class Consola
{
    public static double Lee(string etiqueta)
    {
        Console.Write($"{etiqueta}: ");
        return double.Parse(Console.ReadLine());
    }
}
class Principal
{
    static void Main()
    {
        double numerador = Consola.Lee("Introduce el numerador");
        double divisor = Consola.Lee("Introduce el divisor");
        Console.WriteLine($"La división es {numerador / divisor}");
    }
}
```

Si ahora nos piden filtrar la entrada de datos, para que no se generase error al producirse una entrada correcta.

✗ 😬 Una tentación sería hacer la siguiente implementación...

```
public static double Lee(string etiqueta)
{
    string textoEntrada = default;
    double? valor = null;

    do
    {
        try
        {
            Console.Write($"{etiqueta}: ");
            textoEntrada = Console.ReadLine();
            valor = double.Parse(textoEntrada);
        }
        catch (FormatException)
        {
            Console.WriteLine($"El valor introducido {textoEntrada} no es un valor real válido.");
        }
    }
    while(valor == null);

    return (double)valor;
}
```

En esta implementación, utilizamos excepciones para el **control del flujo de código** y no para una situación de error.

## ¿Cual sería la implementación correcta del código anterior para no hacerlo?

✓ 👍 Microsoft nos propone usar el **patrón try-parse** en su lugar, de la siguiente forma...

```
public static double Lee(string etiqueta)
{
    double valor;
    bool error;

    do
    {
        Console.Write($"{etiqueta}: ");
        string textoEntrada = Console.ReadLine();
        error = double.TryParse(textoEntrada, out valor);
        if (error)
            Console.WriteLine($"El valor introducido {textoEntrada} no es un valor real válido.");
    }
    while(error);

    return valor;
}
```

✗ 😬 De forma similar sucedería si intentásemos controlar la división por cero en el programa principal de la siguiente forma ....

```
static void Main()
{
    double? division = default;
    do
    {
        try
        {
            double numerador = Consola.Lee("Introduce el numerador");
            double divisor = Consola.Lee("Introduce el divisor");
            if (divisor == 0)
                throw new DivideByZeroException();
            division = numerador / divisor;
            Console.WriteLine($"La división es {division}");
        }
        catch(DivideByZeroException)
        {
            Console.WriteLine($"No se puede dividir por cero.\nIntroduzca de nuevo los valores.");
        }
    }
    while (division == null);
}
```

En esta implementación, estamos cometiendo un doble mal uso

- ✗ Utilizamos excepciones para el **control del flujo de código**.
- ✗ En **el mismo ámbito** estamos lanzando una excepción ( Línea 11 ) y capturándola ( Línea 15 ) y esto, **nunca se debe producir**.

## ¿Cual sería la implementación correcta del código anterior para no hacerlo?

✓ 👍 Usaríamos estructuras condicionales tradicionales.

```
static void Main()
{
    bool errorDivisionPorCero;
    do
    {
        double numerador = Consola.Lee("Introduce el numerador");
        double divisor = Consola.Lee("Introduce el divisor");
        errorDivisionPorCero = divisor == 0;
        if (errorDivisionPorCero)
            Console.WriteLine($"No se puede dividir por cero.\nIntroduzca de nuevo los valores.");
        else
            Console.WriteLine($"La división es {numerador / divisor}");
    }
    while (errorDivisionPorCero);
}
```

## Excepciones estándar en .NET

Además, Microsoft nos proporciona unas recomendaciones de uso de excepciones estándar ya definidas por las propias BCL. Fíjate que algunas **no** son recomendables capturarlas, lanzarlas o derivarlas.

- [Recomendaciones de uso de excepciones estándar](#)

Aunque cuando necesitemos lanzar algún tipo de excepción estándar se te indicará en el enunciado. En el siguiente diagrama tienes un resumen **simplificado** de las principales excepciones en .NET y las recomendaciones de uso del cuadro anterior.

