# Unidad 17

Descargar estos apunte en pdf o html

# Índice

# Gestión de errores en POO

## Introducción

Podemos decir que son la **forma en que los lenguajes orientados a objetos realizan el control de errores**.

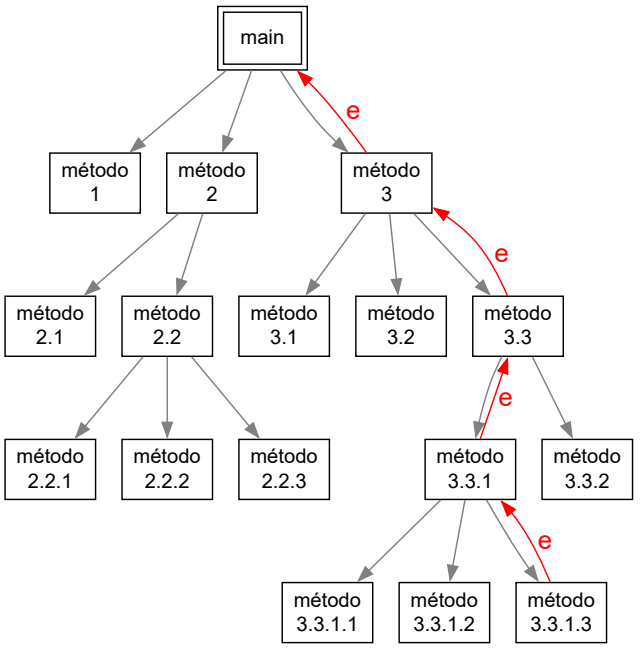
Frente a la programación estructurada tradicional, nos ofrece:

1. **Tratamiento asegurado** de errores.
2. Posibilidad de recuperarnos de un error de forma **centralizada**.
3. **Claridad y simplicidad**, ya que **evitamos lógica adicional** del caso de error.

En la programación estructurada, el control de errores se hacía de forma **dispersa** y **ad-hoc** en cada módulo, lo que hacía que el código fuese más complejo y difícil de mantener.

**Con el control que hemos hecho hasta ahora con aserciones**, no tenemos la posibilidad de recuperarnos de un error, ya que el programa finalizaba. Además, estas solo eran útiles en la fase de desarrollo y pruebas pues solo se producían con la versión de depuración.

Si por el contrario, intentábamos controlar los errores con **códigos de error** devueltos por los métodos, teníamos que añadir lógica adicional en cada módulo para comprobar si se había producido un error y actuar en consecuencia. Esto hacía que el código fuese más complejo y difícil de mantener como se puede ver en el diagrama de ejemplo donde se muestra una situación típica de la programación estructurada, el la cual si se producía un error en un módulo atómico como el `Metodo3313()` del ejemplo. Si queríamos recuperarnos del mismo en algún bucle de opciones por ejemplo en el `Main()`. El error debía propagarse hacia arriba a través de todos los módulos devolviéndose en todas las llamadas.




Esto hacía, que muchos interfaces tuviesen que devolver información adicional con información sobre el error.

# Excepciones en CSharp

Vale, ya podemos sustituir las aserciones por excepciones, pero... **¿Cómo funcionan las excepciones en C#?**

Podemos decir que las excepciones en C# son **objetos** que **contienen un estado con información sobre un error** que se ha producido en tiempo de ejecución. Podemos destacar las siguientes características comunes a muchos lenguajes orientados a objetos:

1. Todas derivan de la clase `System.Exception`
2. Existen ya muchas **predefinidas**.
3. 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 **ArgumentOutOfRangeException** que indicará como su nombre sugiere que un argumento pasado a un método no está dentro del rango permitido.

En el tema anterior creamos una clase `Empleado` que tenía una propiedad `Sueldo`. Donde controlábamos, a través de una aserción en el `set` de la propiedad, que sueldo de un empleado estuviese entre 1200 y 3000 €.

```
public class Empleado : IComparable, ICloneable
{
    private double _sueldo;
    public double Sueldo
    {
        get => _sueldo;
        set
        {
            Debug.Assert(
                condition: value >= 1200D && value <= 3000D,
                message: "El sueldo debe estar entre 1200 y 3000 euros");
            _sueldo = value;
        }
    }
    ...
}
```

Ahora vamos a sustituir la aserción por una excepción, de forma que si se intenta asignar un sueldo fuera de rango, se lance una excepción si lo deseamos podríamos recuperarnos del error, pero

evitando que se asigne el valor erróneo al sueldo.

```
public class Empleado : IComparable, ICloneable
{
    private double _sueldo;
    public double Sueldo
    {
        get => _sueldo;
        set
        {
            if (value < 1200D || value > 3000D)
            {
                throw new ArgumentOutOfRangeException(
                    paramName: nameof(value),
                    message: "El sueldo debe estar entre 1200 y 3000 euros");
            }
            _sueldo = value;
        }
    }}
```

Hemos creado un objeto en memoria del tipo `ArgumentOutOfRangeException` y lo hemos lanzado con la instrucción `throw` hacia algún punto de control.

> **Nota**
>
> A partir de este momento, en todos los casos donde usábamos aserciones para controlar errores, **deberemos sustituirlas por excepciones**.

Otro ejemplo de uso de excepciones podría ser el **control de casting** donde hemos usado el operador `o!` para indicar **que un objeto `o` anulable no es `null`** en ese punto del código.

Por ejemplo, cuando intentábamos hacer un downcast de una abstracción a una clase concreta con el operador `as`.

```
Articulo a = new ArticuloReacondicionado("A005-R", "iPhone 16 Pro", 950,
                                         new(2025, 8, 17), "Foxconn", "Cambio de batería");
// Al usar ()! se generará una excepción de nulo si a no es del tipo ArticuloRebajado
// pero no tenemos ningún control real de que estaba pasando.
ArticuloRebajado ar = (a as ArticuloRebajado)!;

// Ahora generamos una excepción más específica y además con un mensaje
// que nos aporta más información.
ArticuloRebajado ar = a as ArticuloRebajado
                    ?? throw new InvalidCastException(message: $"El artículo {a.Id} no es un artículo rebaja
```

## 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** `catch` o en una condición `when` (la condición `when` será **opcional** y específica del lenguaje C#).

Es importante tener en cuenta que no podremos, poner ningún bloque `catch` que no esté asociado a uno `try`. Además, en C# `try` y `catch` son **instrucciones (statements)** y no expresiones, por lo que **no pueden** formar parte de una expresión más grande o retornarse como valor de una función.

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

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

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

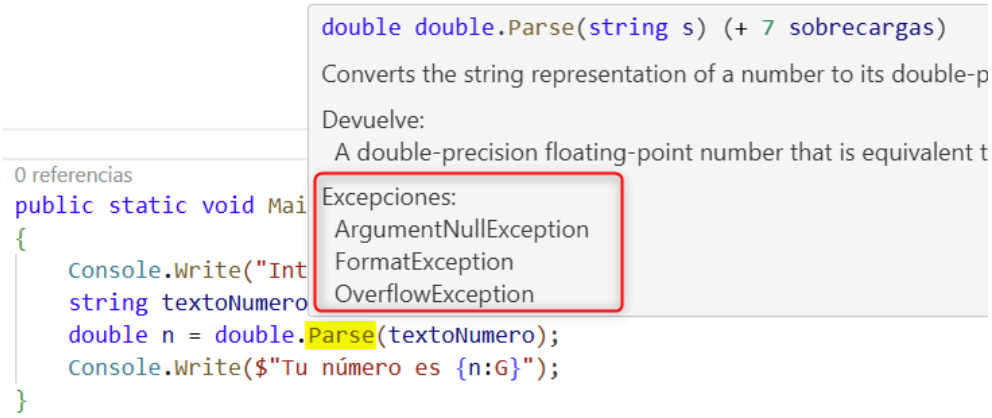
```
Introduce un número real: 25
Tu número es 25
```

Pero si introducimos el texto `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`.

> **Tip**
>
> Para saber que errores/excepciones puede generar una llamada a un método, colocaremos el ratón sobre el método y mostrará una ventana emergente con la documentación del mismo donde se indicarán las excepciones que puede generar.




También podemos hacer `Ctrl + <click izquierdo del ratón>` sobre el método y esto nos abrirá su definición y el la documentación del método podremos verlo.

En el ejemplo, la excepción ha sido capturada por el Runtime de C# (CLR), 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
    // e inicializarla si queremos que sea accesible desde el bloque catch.
    string textoNumero = "sin valor";
    try
    {
        Console.Write("Introduce un número real: ");
        textoNumero = Console.ReadLine()!;
        double n = double.Parse(textoNumero);
        Console.Write($"Tu número es {n}");
    }
    catch (FormatException)
    {
        // Como no usamos e no lo declaramos.
        Console.WriteLine($"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`.

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

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

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

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 CLR.

## 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.

```
public static double Divide(double numerador, double divisor)
{
    if (divisor < 1e-5)
        throw new DivideByZeroException(); // Lanzamos la excepción
    return numerador / divisor;
}

public 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:\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 CLR.

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, **tendremos controlados los dos errores**...

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

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

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

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
{
    //...
}
catch (Exception) // Está cláusula catch nos producirá un error
{
    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`. 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**
>
> La **documentación** del lenguaje, no nos recomienda hacerlo fuera del `Main` por no ser una práctica que puede producir problemas.

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

```
public 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()!);
        double resultado = Divide(numerador, divisor);
        Console.WriteLine($"La división es {resultado:F2}");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

**También tendremos como resultado:**

```
Ejecución 1:
    Introduce el numerador: 4
    Introduce el divisor: 0
    Attempted to divide by zero.

Ejecución 2:
    Introduce el numerador: 4
    Introduce el divisor: cero
    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 cual es **opcional** y debe estar **asociado a un bloque** `try`. Esto es, no necesita de un bloque `catch` para existir. Por tanto, 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` asociados al mismo o **en un ámbito superior**. Además, se **ejecutará en último lugar** respecto a sus bloques try-catch asociados en su ámbito.

En otras palabras, 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 CLR. En este caso, el `finally` se ejecutará siempre y antes que el `catch` del ámbito superior.

Veamos en un simples ejemplos de código esquematizado.

Caso más *'normal'* con **try - catch - finally**

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

**finally sin catch...**

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

## Ejemplo práctico de gestión de errores en POO

En el siguiente ejemplo vamos a ver cómo plantear la gestión de errores/excepciones en una aplicación sencilla aplicando los conceptos vistos. Para ello, vamos a simular una aplicación de gestión de sesiones de usuario en un sistema remoto, donde se instancia algún tipo de recurso asociado a la sesión en el servidor que debe ser liberado al cerrar la sesión. Por ejemplo una máquina virtual, un contenedor, disco virtual, base de datos, etc.

Puedes descargar el código completo de este ejemplo desde el siguiente fichero: **gestion_errores_con_recursos.cs**.

Vamos a definir en primer lugar la clase `Sesion` que tendrá los métodos `Login` y `Logout` para iniciar y cerrar sesión respectivamente. Además, **implementará la interfaz** `IDisposable` para liberar los recursos asociados a la sesión en el servidor. En esta clase se van a ir mostrando una serie de mensajes por consola a modo de log para ver lo que va ocurriendo.

Empezamos definiendo las propiedades `Usuario`, `Clave` e `Iniciada` en el constructor por defecto. Fíjate que no hace falta definir un constructor explícito, basta con inicializar las propiedades en su definición para ello.

```
public class Sesion : IDisposable
{
    public string Usuario { get; private set; } = string.Empty;
    private string Clave { get; set; } = string.Empty;
    public bool Iniciada { get; private set; } = false;
}
```

En el método `Login` hay dos posibles errores que podemos controlar:

1. **Que el usuario o la clave estén vacíos**: En este caso usamos **programación por contrato** y tendremos una precondición donde lanzamos una excepción del tipo `ArgumentException` si alguno de los dos parámetros está vacío o es nulo.
2. **Que ya haya una sesión iniciada**: En este caso, hemos usado **programación defensiva** y si ya hay una sesión iniciada, mostramos un mensaje por consola indicando que se cerrará la sesión actual liberando los recursos del servidor con `Dispose()` y se iniciará la nueva.
   **En el caso de haber optado por programación por contrato**, podríamos haber lanzado una excepción del tipo `InvalidOperationException` y sería **quien llamase al método el que debería controlar que la sesión no estuviese iniciada**.

```
public class Sesion : IDisposable
{
    // ... código anterior
    public void Login(string usuario, string clave)
    {
        if (string.IsNullOrWhiteSpace(usuario) || string.IsNullOrWhiteSpace(clave))
            throw new ArgumentException("Usuario y clave no pueden estar vacíos.");

        if (Iniciada)
        {
            Console.WriteLine($"Ya hay una sesión iniciada. Cerrando la sesión actual de {Usuario} en servidor
            Dispose();
        }

        Usuario = usuario;
        Clave = clave;
        Iniciada = true;

        Console.WriteLine($"Usuario {Usuario} autenticado exitosamente. Registrando sesión en servidor...");
    }
}
```

En el método `Logout` si la sesión aplicamos la precondición de que **debe haber una sesión iniciada** para cerrarla. En caso contrario lanzamos una excepción del tipo `InvalidOperationException` el llamador debe controlar esta condición. Si la sesión está iniciada, mostramos un mensaje por consola y liberamos los recursos del servidor llamando a `Dispose()`.

```
public class Sesion : IDisposable
{
    // ... código anterior
    public void Logout()
    {
        if (!Iniciada)
            throw new InvalidOperationException("No hay una sesión iniciada para cerrar.");
        Console.WriteLine($"Usuario {Usuario} ha cerrado sesión.");
        Dispose();
    }
}
```

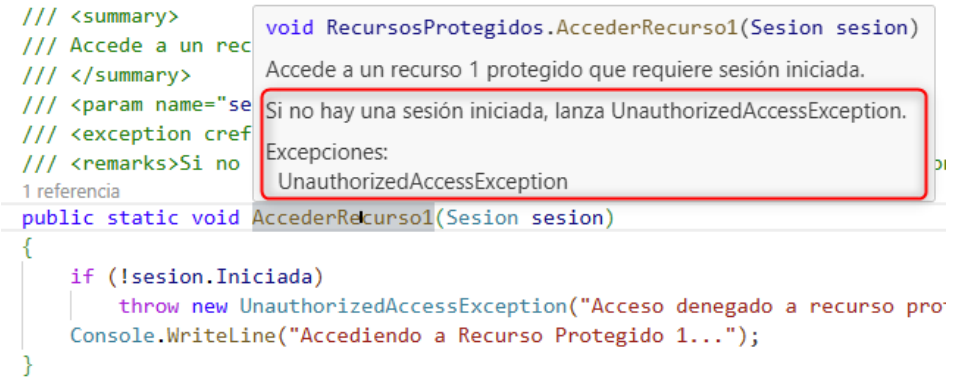
Implementamos el método `Dispose()` de la interfaz `IDisposable` para liberar los recursos asociados a la sesión en el servidor y **vaciamos el estado del objeto sesión si se ha quedado iniciada para evitar su uso**. En este caso, simplemente mostramos un mensaje por consola y reiniciamos las propiedades de la clase. Más adelante veremos el ***'¿Por qué?'*** de implementar esta interfaz.

```
public class Sesion : IDisposable
{
    // ... código anterior
    public void Dispose()
    {
        if (Iniciada)
        {
            Console.WriteLine("Liberando recursos de la sesión en el servidor...");
            Usuario = string.Empty;
            Clave = string.Empty;
            Iniciada = false;
        }
    }
}
```

Definimos ahora una clase de utilidad `RecursosProtegidos`, que puedes encontrar más abajo, con dos métodos estáticos que simulan el acceso a recursos protegidos por sesión. Ambos métodos requieren como precondición de ejecución que la sesión esté iniciada para poder acceder así a los recursos. Si no es así, lanzan una excepción del tipo `InvalidOperationException`.

Fíjate que, a través de la documentación del método, es como se indica que el método debe cumplir la precondición de que la sesión esté iniciada. **De esta forma, aunque no tengamos acceso porque está en una librería, el usuario llamador sabe que debe cumplir esta precondición para usarse sin errores**.

Como hemos comentado antes, al colocar el ratón sobre el método, se muestra la documentación del mismo y las precondiciones que debe cumplir el llamador.




Fíjate además, que en `AccederRecurso2` **simulamos que se produce un error inesperado** al acceder al recurso protegido lanzando una excepción del tipo ** `InvalidOperationException` **aunque la sesión esté iniciada**.

```
public static class RecursosProtegidos
{
    /// <summary>
    /// Accede a un recurso 1 protegido que requiere sesión iniciada.
    /// </summary>
    /// <param name="sesion"></param>
    /// <exception cref="UnauthorizedAccessException"></exception>
    /// <remarks>Si no hay una sesión iniciada, lanza UnauthorizedAccessException.</remarks>
    public static void AccederRecurso1(Sesion sesion)
    {
        if (!sesion.Iniciada)
            throw new UnauthorizedAccessException("Acceso denegado a recurso protegido 1. No hay una sesión in
        Console.WriteLine("Acceso correcto a Recurso Protegido 1...");
    }
    public static void AccederRecurso2(Sesion sesion)
    {
        if (!sesion.Iniciada)
            throw new UnauthorizedAccessException("Acceso denegado a recurso protegido 2. No hay una sesión in

        Console.WriteLine("Acceso correcto a Recurso Protegido 2 se producirá un error inesperado ...");
        throw new InvalidOperationException("Se produjo un error inesperado al acceder al Recurso Protegido 2.
    }
}
```

Por último, definimos el programa principal donde tendremos que tener en cuenta la gestión de errores/excepciones que se puedan producir en los diferentes métodos si se nos ha olvidado cumplir alguna precondición en la llamada.

Definimos en primer lugar un método estático `Menu()` que nos devolverá el menú de opciones a mostrar por consola.

```
public class Program
{
    public static string Menu()
    {
        return """
        1. Iniciar sesión
        2. Acceder a Recurso Protegido 1
        3. Acceder a Recurso Protegido 2
        4. Cerrar sesión
        5. Salir
        """;
    }
}
```

Veamos un poco la gestión del `Main()`, para ello, **lee atentamente los comentarios del código**.

```
public class Program
{
    // ... código anterior

    public static void Main()
    {
        // Iniciamos la sesión fuera del try para que esté accesible
        Sesion sesion = new();
        try
        {
            Console.Clear();
            bool salir = false;
            do
            {
                Console.WriteLine(Menu());
                Console.Write("Seleccione una opción: ");
                string? opcion = Console.ReadLine()!;

                switch (opcion)
                {
                    case "1":
                        Console.Write("Usuario: ");
                        string usuario = Console.ReadLine()!;
                        Console.Write("Clave: ");
                        string clave = Console.ReadLine()!;
                        sesion.Login(usuario, clave);
                        break;
                    case "2":
                        // Debemos ser nosotros quienes verifiquemos si la sesión está iniciada antes de llama
                        // en optro caso, el método lanzaría una excepción por no cumplir la precondición de u
                        // Podemos verlo en la documentación del método AccederRecurso1.
                        if (sesion.Iniciada)
                            RecursosProtegidos.AccederRecurso1(sesion);
                        else
                            Console.WriteLine("Debe iniciar sesión antes de acceder al recurso protegido 1.");
                        break;
                    case "3":
                        // Aquí no verificamos si la sesión está iniciada, para demostrar que el método lanza
                        // si no se cumple la precondición de uso.
                        RecursosProtegidos.AccederRecurso2(sesion);
                        break;
                    case "4":
                        // Cerrar sesión si está iniciada. Como antes, si queremos recuperanos del error,
                        // este es el punto donde debemos hacerlo y no en el método Logout.
                        if (sesion.Iniciada)
                            sesion.Logout();
                        else
                            Console.WriteLine("No hay una sesión iniciada para cerrar.");
                        break;
                    case "5":
```

```
                        Console.WriteLine("Saliendo de la aplicación...");
                        salir = true;
                        break;
                    default:
                        Console.WriteLine("Opción no válida. Intente nuevamente.");
                        break;
                }
            }
            while (!salir);
        }
        catch (Exception e)
        {
            // Gestion de errores centralizada.
            // no se debe usar como control de flujo, sino para capturar errores inesperados.
            Console.WriteLine($"Error inesperado: {e.Message}");
        }
        finally
        {
            // Siempre se ejecuta, haya o no error.
            // Nos aseguramos de liberar los recursos de la sesión.
            sesion.Dispose();
        }
    }
}
```

Ejemplo de ejecución **produciendo un error inesperado una ves iniciada la sesión**...

```
1. Iniciar sesión
2. Acceder a Recurso Protegido 1
3. Acceder a Recurso Protegido 2
4. Cerrar sesión
5. Salir
Seleccione una opción: 1
Usuario: Juan
Clave: 1234
Usuario Juan autenticado exitosamente. Registrando sesión en servidor...
1. Iniciar sesión
2. Acceder a Recurso Protegido 1
3. Acceder a Recurso Protegido 2
4. Cerrar sesión
5. Salir
Seleccione una opción: 3
Acceso correcto a Recurso Protegido 2 se producirá un error inesperado ...
Error inesperado: Se produjo un error inesperado al acceder al Recurso Protegido 2.
Liberando recursos de la sesión en el servidor...
```

Fíjate como se ejecuta el bloque `finally` liberando los recursos de la sesión en el servidor, aunque se haya producido un error inesperado al acceder al recurso protegido 2.

Ejemplo de ejecución **intentando acceder a un recurso protegido sin tener iniciar sesión**...

```
1. Iniciar sesión
2. Acceder a Recurso Protegido 1
3. Acceder a Recurso Protegido 2
4. Cerrar sesión
5. Salir
Seleccione una opción: 1
Usuario: Maria
Clave: 1234
Usuario Maria autenticado exitosamente. Registrando sesión en servidor...
1. Iniciar sesión
2. Acceder a Recurso Protegido 1
3. Acceder a Recurso Protegido 2
4. Cerrar sesión
5. Salir
Seleccione una opción: 2
Acceso correcto a Recurso Protegido 1...
1. Iniciar sesión
2. Acceder a Recurso Protegido 1
3. Acceder a Recurso Protegido 2
4. Cerrar sesión
5. Salir
Seleccione una opción: 4
Usuario Maria ha cerrado sesión.
Liberando recursos de la sesión en el servidor...
1. Iniciar sesión
2. Acceder a Recurso Protegido 1
3. Acceder a Recurso Protegido 2
4. Cerrar sesión
5. Salir
Seleccione una opción: 2
Debe iniciar sesión antes de acceder al recurso protegido 1.
1. Iniciar sesión
2. Acceder a Recurso Protegido 1
3. Acceder a Recurso Protegido 2
4. Cerrar sesión
5. Salir
Seleccione una opción: 4
No hay una sesión iniciada para cerrar.
1. Iniciar sesión
2. Acceder a Recurso Protegido 1
3. Acceder a Recurso Protegido 2
4. Cerrar sesión
5. Salir
Seleccione una opción: 3
Error inesperado: Acceso denegado a recurso protegido 2. No hay una sesión iniciada.
```

Fíjate como hemos liberado bien al hacer logout y controlado por lógica (if-else) todos los posibles errores menos el caso de acceso sin iniciar sesión al recurso protegido 2.

## Instrucción using

Se utiliza para instanciar objetos que contienen recursos no gestionados, esto es, que no son liberados por el recolector de basura. Para poder usarla, las clases que definen los objetos deben implementar el interfaz `IDisposable` y por tanto el método de liberación `Dispose()`.

`using` garantiza que se llama a `Dispose()` **aunque se produzca una excepción**.

> **Aviso**
>
> Dentro del bloque using, el objeto es de solo lectura y no se puede modificar ni reasignar puesto que dejaría de tener una referencia y no se liberaría.

### Sintaxis clásica:

```
// Podemos usar varios recursos liberables en el mismo ámbito así ...
using (TipoIDisposable r1 = new ())
using (TipoIDisposable r2 = new ())
...
using (TipoIDisposable rN = new ())
{
    // Ámbito de uso de solo lectura de r1, r2, … , rN
}

// También podremos anidarlos.
```

### Sintaxis moderna:

En la **documentación oficial** sugiere que usemos un bloque ya definido como ámbito para el recurso liberable. Si esta dentro de un método, se liberará al salir del método y por ejemplo en un bloque `if` se liberará al salir del mismo.

```
if (…)
{
    using TipoIDisposable r = new ();
    // Bloque...
}
```

### Interpretación real de la instrucción `using`:

```
// Cuando instanciemos un objeto disposable de la siguiente manera en un método...
void Metodo()
{
    using TipoIDisposable r = new (); // Cuerpo del método ...
}

// Realmente será un 'syntactic sugar' del siguiente código.
void Metodo()
{
    TipoIDisposable r;
    try
    {
        r = new (); // Cuerpo del método ...
    }
    finally
    {
        if (r != null)
            ((IDisposable)r).Dispose();
    }
}
```

Ejemplo de uso de **`using`** con la clase **`Sesion`** del ejemplo anterior.

**Podemos cambiar este código...**

```
    public static void Main()
    {
        Sesion sesion = new();
        try
        {
            // ... código de la aplicación
        }
        catch (Exception e)
        {
            // ... manejo de excepciones
        }
        finally
        {
            sesion.Dispose();
        }
    }
```

**Por este otro.** Donde la sesion se liberará automáticamente al salir del ámbito del **`try`**.

```
public static void Main()
{
    try
    {
        using Sesion sesion = new();
        // ... código de la aplicación
    }
    catch (Exception e)
    {
        // ... manejo de excepciones
    }
}
```

## 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 EmpresaException : Exception
{
    public EmpresaException(string message) : base (message) {}
}
```

En el código de arriba hemos creado una excepción `EmpresaException` que usaré para saber cuando se ha producido un error dentro de una clase `Empresa`.

Ahora supongamos un método para imprimir nóminas de un departamento dentro de `Empresa`, donde no hemos contemplado un departamento de reciente creación como `Marketing` en el siguiente código.

```
public enum Departamento { Contable, Desarrollo, Marketing };

public class Empresa
{
    public static void ImprimeNominas(Departamento departamento)
    {
        string datosNominas = departamento switch
        {
            Departamento.Contable => "Datos nóminas contabilidad.",
            Departamento.Desarrollo => "Datos nóminas desarrollo.",
            _ => throw new EmpresaException(
                        $"No se pueden imprimir nóminas de este departamento de {departamento}.")
        };
        Console.WriteLine(datosNominas);
    }
}
```

Otra posibilidad es crear excepciones personalizadas más concretas para una clase. Una forma de hacerlo sería **definir la excepción más concreta de forma anidada dentro de la clase**.

```
public class Empresa
{
    // Definición de un tipo anidado
    public class GestionNominasException : Exception
    {
        public GestionNominasException(string message) : base(message) { }
    }
    public static void ImprimeNominas(Departamentos departamento)
    {
        string datosNominas = departamento switch
        {
            Departamentos.Contable => "Imprimiendo nóminas contabilidad.",
            Departamentos.Desarrollo => "Imprimiendo nóminas Desarrollo.",
            // El tipo de la excepción es más especifico
            // y su definición está dentro de empresa.
            _ => throw new Empresa.GestionNominasException(
                        $"No se pueden imprimir nóminas de este departamento de {departamento}.")
        };
        Console.WriteLine(datosNominas);
    }
}
```

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

**Ejemplo de uso de excepción general de la clase**:

```
foreach (var d in Enum.GetValues<Departamento>())
{
    try
    {
        Empresa.ImprimeNominas(d);
    }
    catch (EmpresaException ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }
}
```

**Ejemplo de uso de excepción específica anidada**:

```
foreach (var d in Enum.GetValues<Departamento>())
{
    try
    {
        Empresa.ImprimeNominas(d);
    }
    catch (Empresa.GestionNominasException ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }
}
```

Pudes descargar el código completo de este ejemplo desde el siguiente fichero: **excepcion_personalidada.cs**.

## Uso inadecuado de las excepciones

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 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 oficial**. Que además de ser una lectura complementaria interesante, puede ser **extrapolable a otros lenguajes**.

De entre los consejos del enlace anterior, destacaremos un uso incorrecto de las excepciones que se suele dar con frecuencia en programadores noveles y que debemos evitar.

> ⚠ **Cuidado**
>
> **No utilice excepciones para el flujo de control normal, si es posible.**

Veamos **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:

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

### ❌ Código a evitar

Si ahora nos piden filtrar la entrada de datos, para que no se generase error al producirse una entrada correcta. Una tentación sería implementar el método `Lee` de la siguiente forma...

```
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. Las excepciones deben reservarse para situaciones excepcionales y si está en nuestra mano generarlas en un determinado contexto, debemos hacerlo.

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

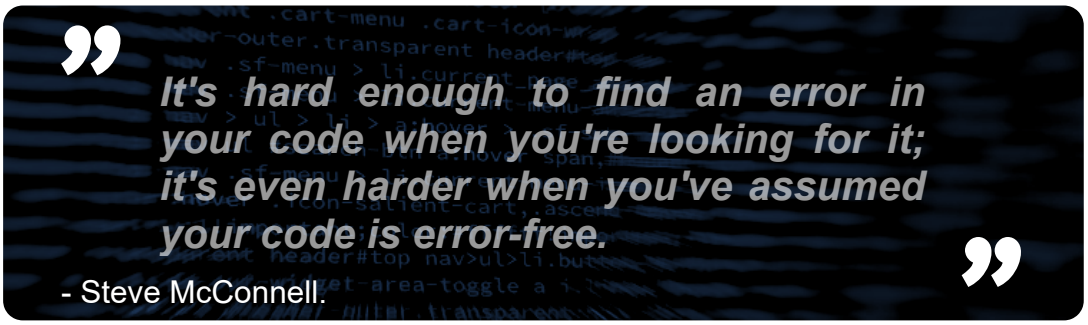
### ✅ Código recomendado

La documentación oficial 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;
}
```



En el caso del programa principal, si nos piden controlar la división por cero, una mala práctica sería hacerlo a través de excepciones.

### ❌ Código a evitar

```
public 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.\n" +
                              "Introduzca de nuevo los valores.");
        }
    }
    while (division == null);
}
```

> **Idea**
>
> En este caso **es fácil de ver**, porque en el ámbito del bucle do-while, estamos lanzando una excepción `throw new DivideByZeroException();` y capturándose en el mismo ámbito con `catch(DivideByZeroException)`. Cunando suceda esto, debe saltarnos una alarma ⚠️.

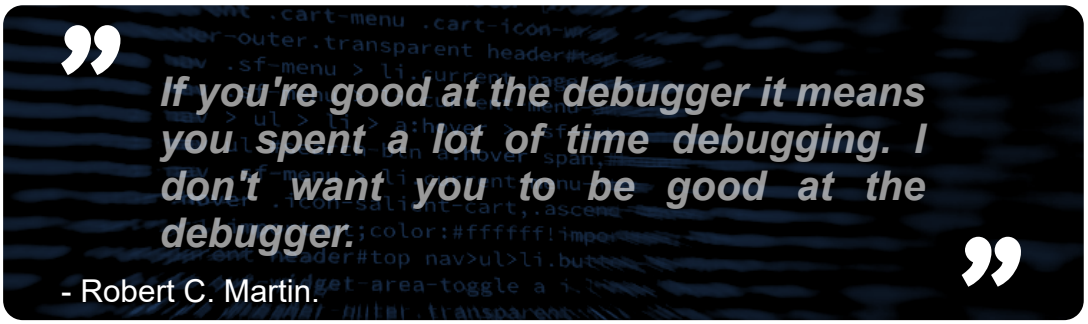
**¿Cual sería la implementación correcta del código anterior para no hacerlo?**

### ✅ Código recomendado

Siempre es más correcto usar lógica de control (**if-else**) para estos casos. Por ejemplo, podríamos hacerlo de la siguiente forma...

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

Vale, pero... **¿Y si la división se hace en una función aparte que no es nuestra y ya no se ve tan claramente que el `throw` y `catch` de la misma excepción están en el mismo ámbito?**



### ✅ Código recomendado

Es código sería exactamente igual, pero ahora es la función `Divide` quien tiene una precondición de que el divisor no sea cero y en caso contrario lanza una excepción. Pero nosotros, como llamadores del método, debemos cumplir esa precondición y no llamar al método si el divisor es cero. **La función debería estar documentada indicando esa precondición**.

```
/// <summary>
/// Divide dos números, lanzando una excepción si el divisor es cero.
/// </summary>
/// <returns>El cociente resultado de la división.</returns>
/// <exception cref="DivideByZeroException"></exception>
/// <remarks>El divisor se considera cero si es menor que 1e-5.</remarks>
public static double Divide(double numerador, double divisor)
{
    if (divisor < 1e-5)
        throw new DivideByZeroException();
    return numerador / divisor;
}
public static void Main()
{
    bool errorDivisionPorCero;
    do
    {
        double numerador = Consola.Lee("Introduce el numerador");
        double divisor = Consola.Lee("Introduce el divisor");
        errorDivisionPorCero = divisor < 1e-5;
        string textoError = errorDivisionPorCero
                            ? $"No se puede dividir por cero.\nIntroduzca de nuevo los valores."
                            : $"La división es {Divide(numerador, divisor)}";
        Console.WriteLine(textoError);
    }
    while (errorDivisionPorCero);
}
```

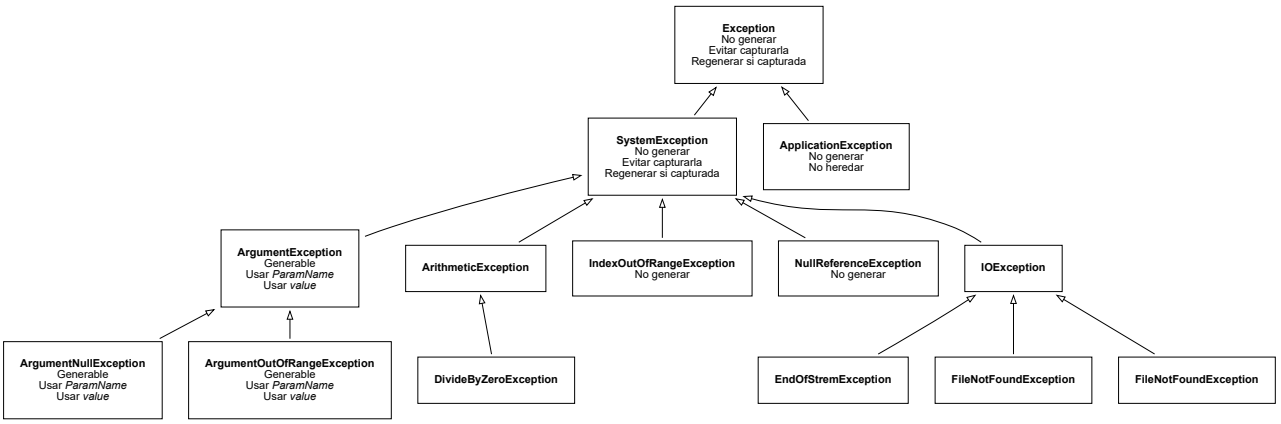
Fíjate que es similar al esquema que hemos seguido cuando hemos gestionado las excepciones en el ejemplo de las sesiones de usuario. Donde hemos cumplido las precondiciones de los métodos que lanzaban excepciones y **no hemos usado excepciones para el control del flujo del programa**.

Puedes descargar el código completo de este ejemplo desde el siguiente fichero: **gestion_errores_correcta.cs**.

## Excepciones estándar en .NET

La **documentación oficial**, 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.

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.



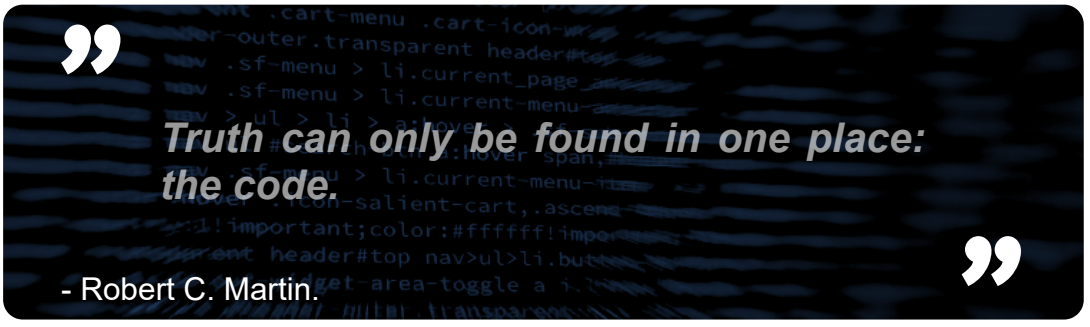
# Anexo I - Ampliación usando encadenamiento de excepciones

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).

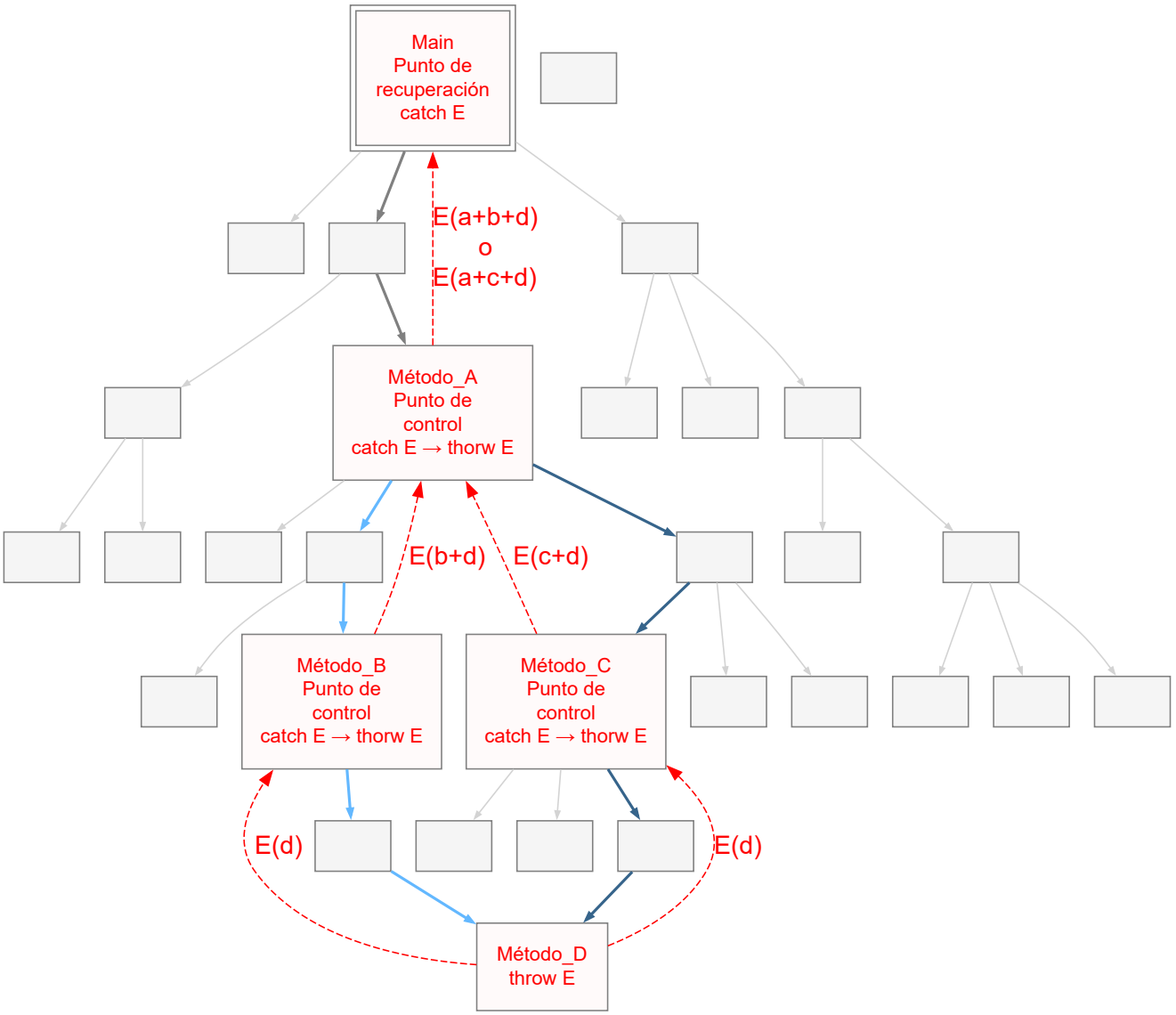
```
try
{
    // Ámbito del try
}
catch (ExcepcionTipoA e)
{
    // En este punto de control de excepciones capturamos la excepción
    // del tipo ExcepcionTipoA y añadimos un mensaje específico de lo
    // que estamos haciendo en el ámbito del try y la relanzamos a un
    // catch en un ámbito superior 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 contol'* en el `Método_A`, si nos fijamos, puede relanzar la excepción añadiendo información de cada uno de los '***`catch`*** *de contol'* 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 sería:

```
public class E : Exception
{
    public E(string message) : base(message) { }
    public E(string message, Exception innerException) : base(message, innerException) { }
}
public class JerarquiaDeLlamadas
{
    static void Metodo_D()
    {
        // En algún punto de pude lanzar la excepción
        throw new E("D");
    }
    public 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);
        }
    }
    public 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);
        }
    }
    public static void Metodo_A()
    {
        try
        {
            // En algún punto de este código en su jerarquía de llamadas,
            // o bien se llemará a Metodo_B() o a Metodo_C()
        }
        catch (E e) // Catch de control en A
        {
            throw new E("A", e);
        }
    }
```

```
    public 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);
        }
    }
}
```