

TEMA 2. LENGUAJE DE PROGRAMACIÓN EN DISEÑO DE INTERFACES

Gestión de los errores

Hemos mencionado muchas veces en el libro los errores que podrían suceder cuando se ejecuta el código. Es el momento de ponerles nombre: se trata de excepciones. Una excepción es un error que se encuentra en la ejecución. Puede ser imprevisto (debido a un bug) o planificado de manera voluntaria. Primero vamos a ver qué es una excepción.

1. Concepto de una excepción

Como se indica arriba, una excepción es una instancia de una clase especial que representa un error de ejecución. Pero, para poder usar este error, tiene que llevar dos datos:

- El tipo de error.
- Información sobre el error.

Como el lenguaje C# está fuertemente tipado, usamos el tipo de la clase de la excepción para transmitir la información del tipo de error. Por ejemplo, una de las excepciones más habituales es `NullReferenceException`, procedente de la clase del mismo nombre. Esta excepción lleva el siguiente tipo de error: ha habido un intento de acceso a un dato en un objeto que no ha sido asignado (que es igual a null).

Por otro lado, como una excepción es una instancia de una clase, se le puede añadir información mediante propiedades. Sin embargo, una clase de excepción debe heredar forzosamente de la clase del framework `.NET Exception` para que sea posible usar todo el mecanismo de gestión de errores del framework.

Por ejemplo, para crear una excepción personalizada, se escribe el siguiente código:

```
public class MiExcepcion : Exception { }
```

Por defecto, la clase de base `Exception` proporciona varios datos que se pueden usar. Se usan con más frecuencia los siguientes:

- `Message` contiene el mensaje de texto generado automáticamente por el framework o añadido por el desarrollador (a menudo es el primer parámetro del constructor de una excepción).
- `InnerException` contiene una posible excepción interna, que también transmite información. Este dato se usa con frecuencia en el marco de la gestión de excepciones que traspasa varias capas de código.

2. Devolver una excepción

Cuando el desarrollador quiere avisar de que hay que desencadenar un error, hay que devolver una excepción. Para que sea posible, hay que efectuar estas acciones:

- Crear una instancia nueva de una clase que representa una excepción.
- Usar la palabra clave `throw` con la instancia creada de esta manera para decirle a la runtime que devuelva la excepción.

La creación de la instancia y la devolución con la palabra clave `throw` se pueden hacer en una sola instrucción, algo que suele suceder porque no es habitual declarar una variable de tipo excepción para usarla más tarde.

Por ejemplo, si se quiere devolver una excepción de tipo `InvalidOperationException` (indica que la operación solicitada no es válida), se puede proceder de la siguiente manera:

```
public decimal Division(decimal a, decimal b)
{
    if(b == 0)
    {
        throw new InvalidOperationException("No se puede
dividir entre 0");
    }
    return a / b;
}
```

Conceptualmente, hay que considerar que una excepción es como una burbuja, es decir, que subirá por todas las capas de código desde el momento en el que se emita hasta que se gestione (veremos cómo hacerlo en la siguiente subsección) o hasta que el comportamiento predeterminado se ocupe de ella porque el programador no la ha gestionado. Si consideramos el siguiente encadenamiento de métodos:

```
public static void Main(string[] args)
{
    Calcular();
}
public static void Calcular()
{
    Console.WriteLine("Ha empezado la división");
    RecuperarEntrada();
}
public static void RecuperarEntrada()
{
    Console.WriteLine("Introducir el primer número");
    decimal one = decimal.Parse(Console.ReadLine());
    Console.WriteLine("Introducir el segundo número");
}
```

```

        decimal two = decimal.Parse(Console.ReadLine());
        Console.WriteLine("Resultado = " + Dividir(one, two));
    }
    public static decimal Dividir (decimal a, decimal b)
    {
        if(b == 0)
        {
            throw new InvalidOperationException("No es posible
dividir entre 0");
        }
        return a / b;
    }
}

```

Si el usuario introduce 0 como segundo número, la excepción nace dentro del método Dividir, pero, como no se ha gestionado allí, sube al método que hace la llamada RecuperarEntrada. Este método tampoco gestiona la excepción, lo que hace que vuelva a subir a Calcular. Por último, como Calcular tampoco gestiona la excepción, la gestiona el método Main y, dado que no hay ninguna gestión manual, la runtime .NET se encarga de la gestión.

Sin embargo, esto provoca el paro de la aplicación, con el guardado de la información del accidente (especialmente en el gestor de eventos de Windows si el programa se ejecuta bajo Windows, o directamente en la consola):

```

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
PS D:\ENI\CSharp\cap4\Dividir> dotnet run
Ha empezado la división
Introducir el primer número
25
Introducir el segundo número
0
Unhandled exception. System.InvalidOperationException: No es posible dividir entre 0
   at Program.<<Main>>$>g_Dividir|0_2(Decimal a, Decimal b) in D:\ENI\CSharp\cap4\Dividir\Program.cs:line 22
   at Program.<<Main>>$>g_RecuperarEntrada|0_1() in D:\ENI\CSharp\cap4\Dividir\Program.cs:line 16
   at Program.<<Main>>$>g_Calcular|0_0() in D:\ENI\CSharp\cap4\Dividir\Program.cs:line 8
   at Program.<<Main>>$(String[] args) in D:\ENI\CSharp\cap4\Dividir\Program.cs:line 3
PS D:\ENI\CSharp\cap4\Dividir>

```

Resultado de la ejecución

Como se puede comprobar en la captura de la consola que aparece antes, si se intenta realizar una división entre 0, se devuelve y se muestra una excepción del tipo `InvalidOperationException`. Sin embargo, también constatamos que el mensaje empieza por `Unhandled exception`, lo que significa que la excepción ha sido capturada en el nivel más alto del programa, por la runtime .NET, y no ha sido gestionada por el programador en el nivel de su código.

Por fortuna, el lenguaje C# pone a nuestra disposición mecanismos que permiten capturar y gestionar las excepciones.

3. Gestionar una excepción

Cuando se ejecuta un fragmento de código, es posible que se encuentre una excepción. Esta excepción puede llegar de forma voluntaria (como en el ejemplo de arriba, donde validaba un trabajo de limitación) o de manera involuntaria (en el caso de un bug).

a. Bloques try, catch y finally

Se puede crear un contexto de ejecución protegido donde, si encontramos una excepción, podemos pedir que sea capturada y definir cómo queremos gestionarla.

Para hacerlo, el lenguaje C# pone a nuestra disposición tres palabras clave:

- **try**, que permite definir un bloque de ejecución con gestión de las excepciones.
- **catch**, que permite definir un bloque de gestión de una excepción capturada.
- **finally**, que permite definir un bloque para ejecutar en todos los casos (tanto si se encuentra con una excepción como si no).

Un bloque try no puede existir solo; necesariamente va seguido de un bloque catch o de un bloque finally e incluso de los dos.

Así, para gestionar el error en el ejemplo anterior, hay que escribir el siguiente código:

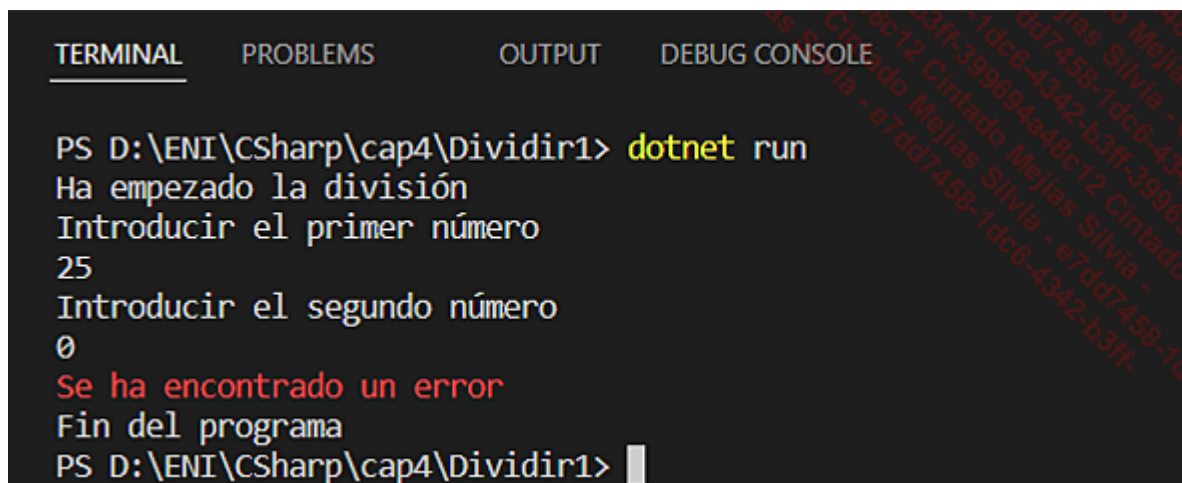
```
public static void Main(string[] args)
{
    try
    {
        Calcular();
    }
    catch
    {
        var color = Console.ForegroundColor;
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Se ha encontrado un error");
        Console.ForegroundColor = color;
    }
    finally
    {
        Console.WriteLine("Fin del programa");
    }
}
```

```

public static void Calcular()
{
    Console.WriteLine("Ha empezado la división");
    RecuperarEntrada();
}
public static void RecuperarEntrada()
{
    Console.WriteLine("Introducir el primer número");
    decimal one = decimal.Parse(Console.ReadLine());
    Console.WriteLine("Introducir el segundo número");
    decimal two = decimal.Parse(Console.ReadLine());
    Console.WriteLine("Resultado = " + Dividir(one, two));
}
public static decimal Dividir(decimal a, decimal b)
{
    if (b == 0)
    {
        throw new InvalidOperationException("No es posible
dividir entre 0");
    }
    return a / b;
}

```

Después de modificarlo de esta manera, al ejecutar este código obtenemos el siguiente resultado:



```

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE

PS D:\ENI\CSharp\cap4\Dividir1> dotnet run
Ha empezado la división
Introducir el primer número
25
Introducir el segundo número
0
Se ha encontrado un error
Fin del programa
PS D:\ENI\CSharp\cap4\Dividir1>

```

Resultado de la ejecución con gestión de los errores

Sin embargo, hay una limitación porque, aquí, nuestro bloque catch es genérico y no permite distinguir el tipo de error que ha sucedido. Por supuesto, lo que se muestra al usuario debe estar lo más separado posible de artefactos técnicos (por ejemplo, es inútil mencionarle que se ha devuelto una excepción de tipo `InvalidOperationException`, esto no le ayudará). Sin embargo, no es posible distinguir las diferentes excepciones con las que podría encontrarse. Por suerte, se pueden añadir limitaciones en un bloque catch en C#.

b. Filtro en bloque catch

En el ejemplo anterior, hemos hecho un bloque catch sencillo, que capta todas las excepciones que se pueden devolver. Con ese fin, no se pueden hacer distinciones entre excepciones de un tipo u otro. Sin lugar a duda, este bloque es más genérico, y de hecho más seguro (no deja pasar ninguna excepción), pero no permite gestionar casos muy precisos.

Con la palabra catch entre paréntesis, se puede especificar el tipo de excepción que se desea gestionar dentro del bloque catch en particular. También se puede añadir un nombre de variable después del tipo para obtener, en el marco del bloque catch, una variable que representa la excepción capturada de esta manera. Usando esta sintaxis se pueden encadenar distintos bloques catch, donde cada uno captura un tipo de excepción diferente.

Esto permite una gestión más detallada y adaptada a cada caso:

```
try
{
    Calcular();
}
catch (InvalidOperationException i) // con variable
{
    var color = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Ha efectuado una operación
prohibida (división entre cero)");
    Console.ForegroundColor = color;
}
catch (NullReferenceException) // sin variable {
    var color = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Se ha encontrado un bug en el código");
    Console.ForegroundColor = color;
}
catch // sin tipo
{
    var color = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Se ha encontrado un error
desconocido");
    Console.ForegroundColor = color;
}
finally
{
    Console.WriteLine("Fin del programa");
}
```

```

static void Calcular()
{
    Console.WriteLine("Ha empezado la división");
    RecuperarEntrada();
}

static void RecuperarEntrada()
{
    Console.WriteLine("Introducir el primer número");
    decimal one = decimal.Parse(Console.ReadLine());
    Console.WriteLine("Introducir el segundo número");
    decimal two = decimal.Parse(Console.ReadLine());
    Console.WriteLine("Resultado = " + Dividir(one, two));
}

static decimal Dividir(decimal a, decimal b)
{
    if (b == 0)
    {
        throw new InvalidOperationException("No es posible
dividir entre 0");
    }
    return a / b;
}

```

Como podemos observar en este ejemplo, se pueden tener distintos bloques catch. Así, la runtime C# entra en el primer bloque para el que se puede cumplir la condición. Por eso es importante el orden; hay que prestar atención y evitar gestionar los casos más genéricos primero porque si no, nunca se llega a los casos específicos.

4. Excepciones y rendimientos

La gestión de las excepciones es un mecanismo muy interesante en cuestión de seguridad del código. Sin embargo, es recomendable prestar atención y no usarlo en exceso para gestionar limitaciones que se podrían tratar de otra manera. Por ejemplo, en el ejercicio de la sección Bases de algoritmia, la versión final usa el método TryParse para definir si se puede convertir el valor. Una alternativa habría podido ser usar el sistema de excepción para obtener un resultado similar, como este:

```

using System;
using System.Collections.Generic;

int limiteMaximo = 0;
string entrada = string.Empty;
bool estInt = false;

```

```

do
{
    Console.WriteLine("Introducir el límite máximo del número
para adivinar");
    entrada = Console.ReadLine();
    if (entrada == "q")
    {
        Environment.Exit(0);
    }
    else
    {
        try
        {
            limiteMaximo = int.Parse(entrada);
            estInt = true;
        }
        catch
        {
            estInt = false;
        }
    }
} while (!estInt && limiteMaximo <= 0);

```

Este planteamiento es funcional, pero no usa el sistema de gestión de las excepciones con moderación. En efecto, el tratamiento de una excepción debe ser excepcional y no utilizarse para gestionar reglas lógicas o trabajo. Así es: la gestión de las excepciones tiene un coste nada despreciable en lo que a rendimiento se refiere. Hay que usar el bloque try/cath solo si se quiere gestionar errores no anticipados.