

# Índice

## ▼ Índice

- [Ejercicio 1. Calculadora avanzada y excepciones predefinidas](#)
- [Ejercicio 2. Sistema de biblioteca con excepciones personalizadas](#)
- [Ejercicio 3. Encadenamiento de Excepciones en Gestión de Archivos](#)
- [Ejercicio 4. Gestión de Cuentas Bancarias y Validación de Números de Cuenta](#)
- [Ejercicio 5. Procesamiento de Archivos y Conteo de Palabras](#)

## Ejercicios Unidad 17

[Descargar estos ejercicios](#)



### Antes de empezar

Estos ejercicios están pensados para trabajar de forma avanzada el manejo de excepciones en C#. Se recomienda revisar los apuntes de la unidad antes de resolverlos.

### Ejercicio 1. Calculadora avanzada y excepciones predefinidas

Crea una clase `CalculadoraAvanzada` con los siguientes métodos:

- `Dividir(int a, int b)` : Devuelve la división de `a` entre `b`.
- `RaizCuadrada(double x)` : Devuelve la raíz cuadrada de `x`.
- `ObtenerElemento(int[] array, int indice)` : Devuelve el elemento del array en la posición indicada.
- `ParsearEntero(string texto)` : Convierte el texto a entero.

En el método `Main`, pide al usuario que introduzca los datos necesarios para probar cada método y gestiona las siguientes excepciones predefinidas:

- `DivideByZeroException` (al dividir por cero)
- `ArgumentOutOfRangeException` (al pedir una raíz cuadrada de un número negativo)
- `IndexOutOfRangeException` (al acceder a una posición no válida del array)

- `FormatException` (al intentar convertir un texto no numérico a entero)

```
Introduce el dividendo: 10
Introduce el divisor: 0
Error: No se puede dividir por cero.

Introduce un número para la raíz cuadrada: -9
Error: No se puede calcular la raíz cuadrada de un número negativo.

Introduce el índice del array (array de 3 elementos): 5
Error: Índice fuera de rango.

Introduce un texto para convertir a entero: hola
Error: Formato no válido para convertir a entero.
```

### Requisitos:

- Utiliza bloques try-catch para capturar cada excepción.
- Muestra un mensaje específico para cada tipo de error.
- En caso de que no haya error, simplemente mostrar el resultado sin ningún tipo de cadena añadida.
- No permitas que el programa termine abruptamente ante un error.

## Ejercicio 2. Sistema de biblioteca con excepciones personalizadas

Crea un sistema de gestión de biblioteca con las siguientes clases:

- **Libro:**
  - Propiedades: `Titulo` (string), `Autor` (string), `Disponible` (bool).
  - Método: `Prestar()` . Si el libro no está disponible, lanza una excepción personalizada `LibroNoDisponibleException` .
- **Usuario:**
  - Propiedades: `Nombre` (string), `LibrosPrestados` (array de `Libro` , máximo 3).
  - Método: `TomarPrestado(Libro libro)` . Si el usuario ya tiene 3 libros prestados, lanza una excepción personalizada `LimitePrestamosException` .
- **Biblioteca:**
  - Propiedad: `Libros` (array de `Libro` ).

- Método: `BuscarLibro(string titulo)` . Si no existe el libro, lanza una excepción predefinida `InvalidOperationException` .

En el método `Main` :

- Crea una biblioteca con varios libros y un usuario.
- Permite al usuario buscar y tomar prestados libros, gestionando todas las excepciones lanzadas y mostrando mensajes adecuados.

```
Introduce el título del libro a buscar: El Quijote
Libro encontrado. ¿Desea tomarlo prestado? (s/n): s
Libro prestado correctamente.
```

```
Introduce el título del libro a buscar: El Quijote
Error: El libro no está disponible.
```

```
Introduce el título del libro a buscar: Libro Inexistente
Error: El libro no existe en la biblioteca.
```

```
Intentando tomar un cuarto libro prestado...
Error: Has alcanzado el límite de préstamos.
```

### Requisitos:

- Implementa y lanza excepciones personalizadas.
- Utiliza arrays para almacenar libros y préstamos.
- Gestiona todas las excepciones en el programa principal.

## Ejercicio 3. Encadenamiento de Excepciones en Gestión de Archivos

Crea un sistema con las siguientes clases:

### 1. Excepción personalizada

- `ArchivoLecturaException` (hereda de `Exception`). Su constructor debe aceptar un mensaje y una excepción interna.

### 2. Clase `GestorArchivos`

- Método `LeerArchivo(string ruta)` :
  - Si la ruta es nula o vacía, lanza un `ArgumentException` .

- Si la extensión del archivo no es `.txt` , lanza un `NotSupportedException` .
- Intenta leer el archivo con `File.ReadAllText` .
  - Si el archivo no existe, captura el `FileNotFoundException` y lanza un `ArchivoLecturaException` con la excepción original como inner exception.
  - Si ocurre cualquier otro error de E/S ( `IOException` ), lanza también un `ArchivoLecturaException` encadenando la excepción original.

### 3. Clase `ProcesadorTexto`

- Método `ContarPalabras(string texto)` :
  - Si el texto es nulo o vacío, lanza un `ArgumentException` .
  - Devuelve el número de palabras del texto.

### 4. En el método `Main`:

- Pide al usuario la ruta de un archivo.
- Usa `GestorArchivos.LeerArchivo` para leer el contenido.
- Usa `ProcesadorTexto.ContarPalabras` para contar las palabras.
- Gestiona todas las excepciones posibles:
  - Si ocurre un `ArchivoLecturaException` , muestra el mensaje y el mensaje de la excepción interna.
  - Si ocurre cualquier otra excepción, muestra un mensaje genérico.

```
Introduce la ruta del archivo: datos.txt
El archivo tiene 120 palabras.
```

```
Introduce la ruta del archivo: archivo_inexistente.txt
Error al leer el archivo: No se pudo leer el archivo.
Causa original: El archivo especificado no se encontró.
```

```
Introduce la ruta del archivo: datos.pdf
Error: Solo se permiten archivos de texto (.txt).
```

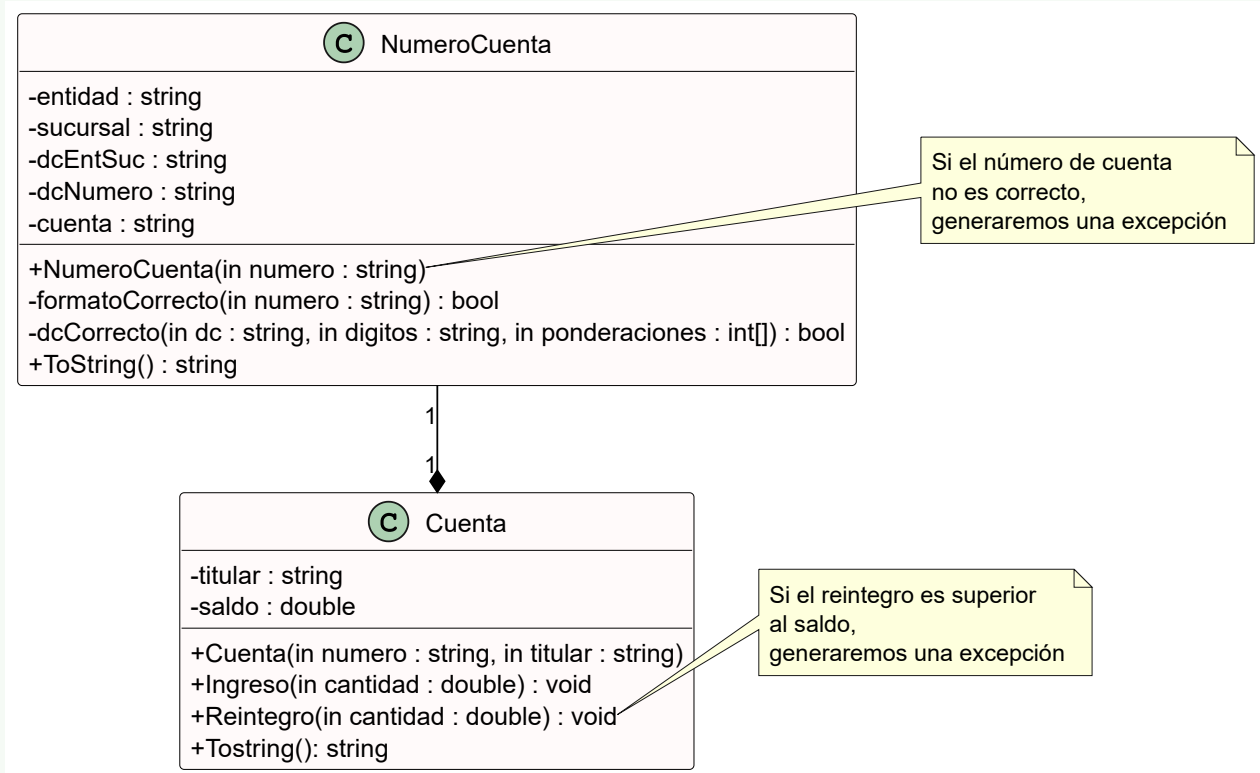
```
Introduce la ruta del archivo:
Error: La ruta no puede estar vacía.
```

### Requisitos:

- Implementa el encadenamiento de excepciones usando inner exception.
- Gestiona correctamente las excepciones predefinidas y personalizadas.
- Usa expresiones regulares para validar la extensión del archivo.

## Ejercicio 4. Gestión de Cuentas Bancarias y Validación de Números de Cuenta

Crea un programa a partir del siguiente diagrama de clases en UML:



La aplicación permitirá recoger datos de una cuenta bancaria. Deberás crear los siguientes métodos y comprobar que funcionan correctamente. Prueba la aplicación con más de un número de cuenta y captura las posibles excepciones en el programa principal.

- El constructor **NumeroCuenta** recibirá un string con el número de cuenta completo y si este no se corresponde con un número de cuenta correcto se lanzara una excepción **NumeroCuentaIncorrectoException**. Esta excepción se puede lanzar por dos motivos:
  - i. Porque no cumple el patrón de un número de cuenta (para ello llamará al método `formatoCorrecto`)
  - ii. Porque la cuenta no se corresponde con los dígitos de control (en este caso usara `dcCorrecto`).
- El método privado de **NumeroDeCuenta**, **formatoCorrecto**. Comprobará el formato del número de cuenta con expresiones regulares y si **el formato** es correcto rellenará los campos del objeto. En caso contrario lanzará una excepción **NumeroCuentaIncorrectoException**, indicando que el formato de la cuenta no cumple las condiciones necesarias.

- El método privado `dcCorrecto` , comprobará si el dígito de control corresponde a la cuenta, devolviendo false en caso contrario.
- El método `Reintegro` , lanzará una excepción `SaldoInsuficienteException` , cuando se intente retirar una cantidad y no haya suficiente saldo.

```
Introduce el número de cuenta: 2085 0103 92 0300731702
Introduce el titular: Ana
Cuenta creada correctamente.
```

```
Introduce cantidad a retirar: 1000
Error: Saldo insuficiente.
```

```
Introduce el número de cuenta: 1234 5678 90 1234567890
Error: El formato del número de cuenta no es válido.
```

```
Introduce el número de cuenta: 2100 0721 09 0200601249
Introduce el titular: Luis
Cuenta creada correctamente.
```

### Requisitos:

- Valida el formato del número de cuenta con expresiones regulares.
- Calcula los dígitos de control según las ponderaciones indicadas.
- Lanza y captura excepciones personalizadas para errores de formato y saldo insuficiente.
- Prueba la aplicación con varios números de cuenta.

## Ejercicio 5. Procesamiento de Archivos y Conteo de Palabras

Crea un sistema con las siguientes clases:

### 1. Excepción personalizada

- `ArchivoLecturaException` (hereda de `Exception`). Su constructor debe aceptar un mensaje y una excepción interna.

### 2. Clase `GestorArchivos`

- Método `LeerArchivo(string ruta)` :
  - Si la ruta es nula o vacía, lanza un `ArgumentException` .
  - Si la extensión del archivo no es `.txt` , lanza un `NotSupportedException` .

- Intenta leer el archivo con `File.ReadAllText` .
  - Si el archivo no existe, captura el `FileNotFoundException` y lanza un `ArchivoLecturaException` con la excepción original como inner exception.
  - Si ocurre cualquier otro error de E/S ( `IOException` ), lanza también un `ArchivoLecturaException` encadenando la excepción original.

### 3. Clase `ProcesadorTexto`

- Método `ContarPalabras(string texto)` :
  - Si el texto es nulo o vacío, lanza un `ArgumentException` .
  - Devuelve el número de palabras del texto.

### 4. En el método `Main`:

- Pide al usuario la ruta de un archivo.
- Usa `GestorArchivos.LeerArchivo` para leer el contenido.
- Usa `ProcesadorTexto.ContarPalabras` para contar las palabras.
- Gestiona todas las excepciones posibles:
  - Si ocurre un `ArchivoLecturaException` , muestra el mensaje y el mensaje de la excepción interna.
  - Si ocurre cualquier otra excepción, muestra un mensaje genérico.

```
Introduce la ruta del archivo: datos.txt
El archivo tiene 120 palabras.
```

```
Introduce la ruta del archivo: archivo_inexistente.txt
Error al leer el archivo: No se pudo leer el archivo.
Causa original: El archivo especificado no se encontró.
```

```
Introduce la ruta del archivo: datos.pdf
Error: Solo se permiten archivos de texto (.txt).
```

```
Introduce la ruta del archivo:
Error: La ruta no puede estar vacía.
```

### Requisitos:

- Implementa el encadenamiento de excepciones usando inner exception.
- Gestiona correctamente las excepciones predefinidas y personalizadas.
- Usa expresiones regulares para validar la extensión