EXCEPCIONES

PARADIGMAS DE PROGRAMACIÓN II

Carlos Rojas Sánchez

Licenciatura en Informática

Universidad del Mar

Contenido

Conceptos generales

¿Qué es una Excepción?

- Una excepción es un error en tiempo de ejecución.
- Permite manejar fallos como divisiones por cero, índices fuera de rango, etc.
- Todas las excepciones derivan de `System.Exception`.

Sintaxis Básica

```
try
{
    // Código que puede fallar
}
catch (ExceptionTipo ex)
{
    // Manejo del error
}
finally
{
    // Siempre se ejecuta
}
```

Ejemplo Básico

```
try
{
    int[] numeros = { 1, 2, 3 };
    Console.WriteLine(numeros[5]);
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
finally
{
    Console.WriteLine("Finalizando...");
}
```

Tipos Comunes de Excepciones

- `DivideByZeroException`
- `NullReferenceException`
- `IndexOutOfRangeException`
- `FormatException`
- `FileNotFoundException`
- `InvalidOperationException`

Múltiples `catch`

```
try
{
    // ...
}
catch (FormatException)
{
    Console.WriteLine("Error de formato.");
}
catch (DivideByZeroException)
{
    Console.WriteLine("No se puede dividir por cero.");
}
catch (Exception ex)
{
    Console.WriteLine("Otro error: " + ex.Message);
}
```

Lanzar Excepciones Manualmente

```
void VerificarEdad(int edad)
{
    if (edad < 18)
        throw new ArgumentException("Debe ser mayor de edad.");
}
```

Excepciones Personalizadas

```
public class MiExcepcion : Exception
{
    public MiExcepcion(string mensaje) : base(mensaje) { }
}
```

Buenas Prácticas

- Captura solo excepciones que puedas manejar.
- Usa `finally` para cerrar archivos o conexiones.
- No atrapes `Exception` de forma genérica sin necesidad.
- Lanza excepciones con mensajes claros.

Estructura de un Proyecto Windows Forms en Visual Studio 2022

¿Qué es un Proyecto Windows Forms?

- Es una aplicación con interfaz gráfica para Windows.
- Usa formularios (‘Form‘) para mostrar ventanas, botones, cajas de texto, etc.
- En Visual Studio, el proyecto se divide en varios archivos para mantener una buena organización.

Archivos principales en el proyecto

- `Program.cs`
- `Form1.cs`
- `Form1.Designer.cs`

`Program.cs` – Punto de entrada

- Contiene el método `Main()`.
- Es donde comienza la ejecución de la aplicación.
- Inicia el formulario principal con `Application.Run(new Form1());`

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
}
```

`Form1.cs` – Lógica del formulario

- Contiene el código para responder a eventos.
- Aquí se escriben los métodos como `btnCalcular_Click()`.
- Es la parte donde el programador define el comportamiento.

`Form1.Designer.cs` – Diseño de la interfaz

- Es generado automáticamente por Visual Studio.
- Define la posición, tamaño y propiedades de los controles.
- No se recomienda editarlo manualmente.

```
private void InitializeComponent()
{
    this.txtNumero1 = new System.Windows.Forms.TextBox();
    this.btnCalcular = new System.Windows.Forms.Button();
    // ...
}
```

Relación entre los archivos

- `Form1.cs` y `Form1.Designer.cs` forman una **clase parcial** (‘partial class‘).
- Esto permite dividir la lógica del diseño y la programación sin conflictos.
- Visual Studio actualiza automáticamente el diseñador sin borrar tu código.

Resumen

Archivo	Función
`Program.cs`	Inicia la aplicación con el método `Main()`
`Form1.cs`	Contiene la lógica del formulario y eventos
`Form1.Designer.cs`	Define el diseño visual del formulario (auto-generado)

Ventajas de esta estructura

- Mantiene el código organizado y limpio.
- Facilita el trabajo con formularios complejos.
- Permite a Visual Studio separar responsabilidades:
  - Diseño (visual)
  - Lógica (programación)

Aserciones en C#

¿Qué es una aserción?

Una aserción es una expresión que se espera que sea verdadera en tiempo de ejecución.

Si no se cumple, indica un error lógico en el programa.

Se utiliza principalmente durante el desarrollo para detectar fallos internos.

Espacio de nombres necesario

Para utilizar aserciones en C# se necesita importar el siguiente espacio de nombres:

```
using System.Diagnostics;
```

Sintaxis básica

La sintaxis más común es:

```
Debug.Assert(condición, "Mensaje si falla la condición");
```

Ejemplo simple

```
using System;
using System.Diagnostics;

class Program
{
    static void Main()
    {
        int edad = 20;
        Debug.Assert(edad >= 0, "La edad no puede ser negativa"
            );
        Console.WriteLine("Edad válida: " + edad);
    }
}
```

Assert vs Excepciones

Assert (Debug)	Excepciones (try-catch)
Solo en desarrollo	En producción y desarrollo
No se ejecuta en modo Release	Siempre se ejecuta
Detección temprana de bugs	Manejo de errores en tiempo de ejecución
No reemplaza validaciones	Requiere manejo explícito

Ejemplo avanzado

```
int CalcularPromedio(int suma, int cantidad)
{
    Debug.Assert(cantidad > 0, "La cantidad debe ser mayor a 0"
        );
    return suma / cantidad;
}
```

Conclusión

- Las aserciones son útiles para detectar errores lógicos.
- Son parte del proceso de depuración, no de la gestión de errores.
- Se recomienda usarlas en pruebas y desarrollo, pero no en producción.

Estructuras de Datos en C#

¿Qué son las estructuras de datos?

Las estructuras de datos son formas de organizar y almacenar datos para facilitar su acceso y modificación.

En C#, se dividen principalmente en:

- Estructuras lineales
- Estructuras no lineales
- Estructuras genéricas

Listas (List<T>)

- Permiten almacenar elementos de forma ordenada.
- Tamaño dinámico.
- Acceso por índice.

```
using System.Collections.Generic;

List<string> nombres = new List<string>();
nombres.Add("Ana");
nombres.Add("Luis");
Console.WriteLine(nombres[1]); // Luis
```

Pilas (Stack<T>)

- Tipo LIFO (último en entrar, primero en salir).
- Métodos: Push, Pop, Peek.

```
Stack<int> pila = new Stack<int>();
pila.Push(10);
pila.Push(20);
int cima = pila.Pop(); // 20
```

Colas (Queue<T>)

- Tipo FIFO (primero en entrar, primero en salir).
- Métodos: Enqueue, Dequeue, Peek.

```
Queue<string> cola = new Queue<string>();
cola.Enqueue("Cliente1");
cola.Enqueue("Cliente2");
string siguiente = cola.Dequeue(); // Cliente1
```

Diccionarios (Dictionary<TKey, TValue>)

- Almacenan pares clave-valor.
- Acceso rápido a datos mediante claves únicas.

```
Dictionary<string, int> edades = new Dictionary<string, int>();
edades["Ana"] = 30;
edades["Luis"] = 25;
Console.WriteLine(edades["Luis"]); // 25
```

Conjuntos (HashSet<T>)

- Colección de elementos únicos.
- Útil para evitar duplicados.

```
HashSet<string> frutas = new HashSet<string>();
frutas.Add("Manzana");
frutas.Add("Manzana"); // No se agrega de nuevo
```

Árbol Binario (implementación personalizada)

- Estructura jerárquica no lineal.
- Cada nodo tiene como máximo dos hijos.

```
class Nodo {
    public int Valor;
    public Nodo Izq, Der;

    public Nodo(int val) {
        Valor = val;
        Izq = Der = null;
    }
}
```

Uso de Árbol Binario

```
Nodo raiz = new Nodo(10);
raiz.Izq = new Nodo(5);
raiz.Der = new Nodo(15);
Console.WriteLine(raiz.Izq.Valor); // 5
```

- Se puede extender a árboles de búsqueda binaria (BST).

Grafos (básico con listas de adyacencia)

- Representación de relaciones entre nodos.

```
class Grafo {
    Dictionary<int, List<int>> adyacencia = new();

    public void AgregarArista(int origen, int destino) {
        if (!adyacencia.ContainsKey(origen))
        adyacencia[origen] = new List<int>();
        adyacencia[origen].Add(destino);
    }
}
```

Uso del Grafo

```
Grafo g = new Grafo();
g.AgregarArista(1, 2);
g.AgregarArista(1, 3);
```

- Se pueden implementar búsquedas como BFS y DFS.

Estructuras personalizadas

Puedes crear tus propias estructuras con clases o structs:

```
class Contacto {
    public string Nombre;
    public string Telefono;

    public Contacto(string n, string t) {
        Nombre = n;
        Telefono = t;
    }
}
```

Operaciones básicas con archivos de texto en C#

1. Escribir texto en un archivo

```
File.WriteAllText("archivo.txt", "Hola mundo");
```

2. Leer texto completo

```
string contenido = File.ReadAllText("archivo.txt");
Console.WriteLine(contenido);
```

3. Escribir varias líneas

```
string[] lineas = { "Línea 1", "Línea 2" };
File.WriteAllLines("archivo.txt", lineas);
```

4. Leer línea por línea

```
string[] lineas = File.ReadAllLines("archivo.txt");
foreach (string linea in lineas)
Console.WriteLine(linea);
```

5. Agregar contenido

```
File.AppendAllText("archivo.txt", "\nOtra línea");
```

6. Verificar si existe el archivo

```
if (File.Exists("archivo.txt")) {
    Console.WriteLine("El archivo existe");
}
```