Exception Handling in C#.NET

1. Introduction to Exceptions

- An **exception** is an unexpected or erroneous situation that occurs during program execution.
- In C#, exceptions are runtime errors that disrupt the normal flow of a program.
- The .NET Framework provides a structured way to handle exceptions using the `try`, `catch`, `finally`, and `throw` keywords.

2. Key Concepts

- **Exception Class**: All exceptions in C# inherit from the `System.Exception` class, which has properties like `Message`, `StackTrace`, and `InnerException`.
- **Types of Exceptions**:
  - Predefined: `DivideByZeroException`, `NullReferenceException`, `FileNotFoundException`, etc.
  - User-defined: Custom exceptions created by extending `Exception`.

3. Exception Handling Syntax

```
try
{
    // Code that might throw an exception
}
catch (ExceptionType ex)
{
    // Handle specific exception
}
finally
{
    // Code that runs regardless of exception (optional)
}
```

- `try` **Block**: Contains code that might throw an exception.
- `catch` **Block**: Handles the exception. Multiple `catch` blocks can be used for different exception types.
- `finally` **Block**: Executes cleanup code (e.g., closing files or releasing resources) whether an exception occurs or not.
- `throw`: Manually throws an exception.

4. Best Practices

- Catch specific exceptions rather than a generic `Exception` to avoid masking unrelated errors.
- Use `finally` for resource cleanup.
- Log exceptions for debugging (e.g., using `Console.WriteLine` or a logging framework).
- Avoid using exceptions for normal flow control—reserve them for exceptional cases.

5. Creating Custom Exceptions

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

6. Common Exceptions in C#

- `ArgumentNullException`: Parameter is null.
- `IndexOutOfRangeException`: Array index is invalid.
- `InvalidOperationException`: Operation is not valid in the current state.

Case Study 1: Division Calculator

Scenario: A program takes two user inputs (numerator and denominator) and performs division.

```
class Program
{
    static void Main()
    {
        Console.Write("Enter numerator: ");
        int numerator = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter denominator: ");
        int denominator = Convert.ToInt32(Console.ReadLine());

        try
        {
            int result = numerator / denominator;
            Console.WriteLine($"Result: {result}");
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine("Error: Cannot divide by zero.");
        }
        catch (FormatException ex)
        {
            Console.WriteLine("Error: Please enter valid numbers.");
        }
        finally
        {
            Console.WriteLine("Calculation attempt completed.");
        }
    }
}
```

Explanation:

- Handles `DivideByZeroException` for division by zero.
- Handles `FormatException` for invalid input (e.g., letters instead of numbers).
- `finally` confirms the attempt regardless of success.

Case Study 2: File Reader

Scenario: A program reads content from a file specified by the user.

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        Console.Write("Enter file path: ");
        string filePath = Console.ReadLine();

        try
        {
            string content = File.ReadAllText(filePath);
            Console.WriteLine("File content: " + content);
        }
        catch (FileNotFoundException ex)
        {
            Console.WriteLine("Error: File not found.");
        }
        catch (IOException ex)
        {
            Console.WriteLine("Error: An I/O error occurred.");
        }
        finally
        {
            Console.WriteLine("File operation completed.");
        }
    }
}
```

Explanation:

- `FileNotFoundException` catches missing files.
- `IOException` handles general I/O issues (e.g., file locked).
- `finally` ensures cleanup or logging can occur.

Case Study 3: Custom Exception for Age Validation

Scenario: A program validates a person's age for voting eligibility.

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

class Program
{
    static void Main()
    {
        Console.Write("Enter your age: ");
        int age;

        try
        {
            age = Convert.ToInt32(Console.ReadLine());
            if (age < 18)
            {
                throw new InvalidAgeException("You must be 18 or older to vote.");
            }
            Console.WriteLine("You are eligible to vote!");
        }
        catch (InvalidAgeException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        catch (FormatException ex)
        {
            Console.WriteLine("Error: Please enter a valid number.");
        }
    }
}
```

Explanation:

- A custom `InvalidAgeException` is thrown if age < 18.
- `FormatException` catches invalid input.
- Demonstrates user-defined exception handling.
