



International University of Business Agriculture and Technology

Department: Computer Science and Engineering

Semester: Spring 2025

Course Name: Visual Programming

Course Code: CSC 440

Section: A

Lab Report topic: Lab task 06

Submitted To:

Suhala Lamia

Assistant Professor

Department of Computer Science and Engineering

Submitted By:

Samiul Karim Mazumder

22303308

Date of Submission: 17/04/25

Experiment No. 06: Learning about Try-Catch-Exception and abstract classes using C#.

6.1

Objective: To develop a basic calculator program in C# that performs division and handles exceptions such as:

- Division by zero (DivideByZeroException)
- Invalid input (FormatException)
- Any other unexpected error (Exception)

Algorithm:

1. Start the program.
2. Use a loop to ask the user to input two numbers repeatedly.
3. Convert the inputs to double.
4. Call a method to perform division.
5. If division is successful, display the result and exit the loop.
6. Catch and handle:
 - DivideByZeroException: show a message for zero division.
 - FormatException: show a message for non-numeric input.
 - General Exception: show a generic error message.
7. Repeat the process until valid input is provided and division succeeds.
8. End the program.

Code:

```
using System;

namespace lab_6
{
    class SimpleCalculator
    {
        static void Main()
```

```
{
    Console.WriteLine("Simple Division Calculator");
    while (true)
    {
        try
        {
            Console.Write("Enter the first number: ");
            double num1 = Convert.ToDouble(Console.ReadLine());
            Console.Write("Enter the second number: ");
            double num2 = Convert.ToDouble(Console.ReadLine());
            if (num2 == 0)
            {
                throw new DivideByZeroException();
            }
            double result = DivideNumbers(num1, num2);
            Console.WriteLine("Result: " + result);
            break;
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("You cannot divide by zero.\n");
        }
        catch (FormatException)
        {

```

```
        Console.WriteLine("Invalid input.\n");
    }
    catch (Exception)
    {
        Console.WriteLine("Something went wrong.\n");
    }
}

static double DivideNumbers(double a, double b)
{
    return a / b;
}
}
```

Output:

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard window controls (minimize, maximize, close). The console output is as follows:
Simple Division Calculator
Enter the first number: 10
Enter the second number: 0
You cannot divide by zero.

Enter the first number: 5a
Invalid input.

Enter the first number: 4
Enter the second number: 10
Result: 0.4

6.2

Objective:

To create a simple banking application in C# that handles:

- **Custom exceptions** like `InsufficientFundsException` when the user tries to withdraw more than their account balance.
- **Illegal input** such as negative withdrawal amounts using `ArgumentException`.
- **General exceptions** for unexpected errors using the base `Exception` class.

Algorithm:

1. Define a **custom exception class** named `InsufficientFundsException`.
2. Create a `BankAccount` class with:
 - A balance field.
 - A method `Withdraw(double amount)` that:
 - Throws `InsufficientFundsException` if `amount > balance`.
 - Throws `ArgumentException` if `amount < 0`.
 - Subtracts amount from balance otherwise.
3. In the `Main` method:
 - Ask the user for an amount to withdraw.
 - Try to perform the withdrawal.
 - Catch and handle:
 - `InsufficientFundsException`

- ArgumentException

- General Exception

4. Display proper error messages for each case.

Code:

```
using System;
```

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

```
class BankAccount
{
    public double Balance { get; private set; }

    public BankAccount(double initialBalance)
    {
        Balance = initialBalance;
    }
}
```

```
public void Withdraw(double amount)
{
}
```

```
        if (amount < 0)
        {
            throw new ArgumentException("Withdrawal amount cannot be negative.");
        }
        else if (amount > Balance)
        {
            throw new InsufficientFundsException("Insufficient funds.");
        }
        else
        {
            Balance -= amount;

            Console.WriteLine($"Withdrawal successful. Remaining balance: {Balance}");
        }
    }
}

class SimpleBankingApp
{
    static void Main()
    {
        BankAccount myAccount = new BankAccount(1000);

        Console.WriteLine("Welcome to Simple Bank!");

        Console.WriteLine($"Your current balance is: {myAccount.Balance}");
    }
}
```

```
try
{
    Console.Write("Enter amount to withdraw: ");
    double amount = Convert.ToDouble(Console.ReadLine());

    myAccount.Withdraw(amount);
}
catch (InsufficientFundsException ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
catch (ArgumentException ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("An unexpected error occurred: " + ex.Message);
}
}
```


Output:

```
Microsoft Visual Studio Debug Console
Welcome to Simple Bank!
Your current balance is: 1000
Enter amount to withdraw: 500
Withdrawal successful. Remaining balance: 500
```

```
Microsoft Visual Studio Debug Console
Welcome to Simple Bank!
Your current balance is: 1000
Enter amount to withdraw: 1300
Error: Insufficient funds.
```

```
Select Microsoft Visual Studio Debug Console
Welcome to Simple Bank!
Your current balance is: 1000
Enter amount to withdraw: gdh
An unexpected error occurred: The input string 'gdh' was not in a correct format.
```

```
Microsoft Visual Studio Debug Console
Welcome to Simple Bank!
Your current balance is: 1000
Enter amount to withdraw: -100
Error: Withdrawal amount cannot be negative.
```

6.3

Objective: To design a Transportation Management System in C# that Represents different types of vehicles (Car, Truck), uses interfaces and abstract classes for shared and specific behaviors, implements startEngine(), stopEngine(), and calculateFuelEfficiency() methods for all vehicles and adds an extra loadCargo() method for Trucks.

Algorithm:

1. Define an **interface IVehicle** with:
 - void StartEngine()
 - void StopEngine()
 - double CalculateFuelEfficiency()
2. Create an **abstract class Vehicle** that implements **IVehicle** and provides:
 - Common implementation for StartEngine() and StopEngine().
 - An abstract method CalculateFuelEfficiency() to be overridden.

3. Create a **Car** class that inherits from **Vehicle** and provides specific fuel efficiency logic.
4. Create a **Truck** class that:
 - Inherits from **Vehicle**
 - Overrides **CalculateFuelEfficiency()**
 - Adds a new method **LoadCargo()** for loading.
5. In the **Main()** method:
 - Create objects of **Car** and **Truck**.
 - Demonstrate their behaviors using interface references.

Code:

```
using System;
```

```
interface IVehicle
```

```
{
```

```
    void StartEngine();
```

```
    void StopEngine();
```

```
    double CalculateFuelEfficiency();
```

```
}
```

```
abstract class Vehicle : IVehicle
```

```
{
```

```
    public void StartEngine()
```

```
    {
```

```
        Console.WriteLine("Engine started.");
```

```
    }
```

```
    public void StopEngine()
```

```
    {
```

```
        Console.WriteLine("Engine stopped.");
    }

    public abstract double CalculateFuelEfficiency();
}

class Car : Vehicle
{
    public override double CalculateFuelEfficiency()
    {
        Console.WriteLine("Calculating Car fuel efficiency...");
        return 15.5;
    }
}

class Truck : Vehicle
{
    public override double CalculateFuelEfficiency()
    {
        Console.WriteLine("Calculating Truck fuel efficiency...");
        return 8.0;
    }

    public void LoadCargo()
    {
        Console.WriteLine("Cargo loaded into the truck.");
    }
}
```

```
class TransportSystem
{
    static void Main()
    {
        IVehicle myCar = new Car();
        Console.WriteLine("Car:");
        myCar.StartEngine();
        Console.WriteLine($"Fuel Efficiency: {myCar.CalculateFuelEfficiency()} km/l");
        myCar.StopEngine();
        Console.WriteLine();
        Truck myTruck = new Truck();
        Console.WriteLine("Truck:");
        myTruck.StartEngine();
        Console.WriteLine($"Fuel Efficiency: {myTruck.CalculateFuelEfficiency()} km/l");
        myTruck.LoadCargo();
        myTruck.StopEngine();
    }
}
```

Output:

```
Microsoft Visual Studio Debug Console

Car:
Engine started.
Calculating Car fuel efficiency...
Fuel Efficiency: 15.5 km/l
Engine stopped.

Truck:
Engine started.
Calculating Truck fuel efficiency...
Fuel Efficiency: 8 km/l
Cargo loaded into the truck.
Engine stopped.
```

6.4

Objective: To develop a File Management System that supports reading and writing to multiple file formats (e.g., CSV, JSON, XML), Schema validation for XML, and future flexibility using OOP concepts like interfaces and abstract classes.

Algorithm:

1. Create an **interface IFileOperations** with:
 - `void ReadFile();`
 - `void WriteFile();`
2. Create an **abstract class FileHandler** that implements **IFileOperations**.
 - Provides base structure or default behavior (if needed).
3. Create **concrete classes**:
 - **CSVFileHandler** – Implements `ReadFile()` and `WriteFile()`.
 - **XMLFileHandler** – Implements `ReadFile()`, `WriteFile()`, and adds a new method `ValidateSchema()`.
4. Demonstrate the functionality in the `Main()` method by creating instances and calling the methods.

Code:

```
using System;
```

```
interface IFileOperations
{
    void ReadFile();
    void WriteFile();
}

abstract class FileHandler : IFileOperations
{
    public abstract void ReadFile();
    public abstract void WriteFile();
}

class CSVFileHandler : FileHandler
{
    public override void ReadFile()
    {
        Console.WriteLine("Reading data from CSV file...");
    }

    public override void WriteFile()
    {
        Console.WriteLine("Writing data to CSV file...");
    }
}

class XMLFileHandler : FileHandler
{
    public override void ReadFile()
```

```

    {
        Console.WriteLine("Reading data from XML file...");
    }

    public override void WriteFile()
    {
        Console.WriteLine("Writing data to XML file...");
    }

    public void ValidateSchema()
    {
        Console.WriteLine("Validating XML schema...");
    }
}

```

```

class FileManagementSystem
{
    static void Main()
    {
        Console.WriteLine("CSV File Operations:");
        CSVFileHandler csv = new CSVFileHandler();
        csv.ReadFile();
        csv.WriteFile();
        Console.WriteLine();
        Console.WriteLine("XML File Operations:");
        XMLFileHandler xml = new XMLFileHandler();
    }
}

```

```
        xml.ReadFile();  
        xml.WriteFile();  
        xml.ValidateSchema();  
    }  
}
```

Output:

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard window controls (minimize, maximize, close). The console output is as follows:
CSV File Operations:
Reading data from CSV file...
Writing data to CSV file...

XML File Operations:
Reading data from XML file...
Writing data to XML file...
Validating XML schema...
The text is displayed in a monospaced font on a dark background.