12214994_SaurabhRana

Technical Specification: Banking Transaction Module

1. Project Overview

Objective: Develop a robust C# backend logic for an automated teller system that handles deposits and withdrawals while ensuring data integrity through structured Exception Handling.

---

2. Functional Requirements

A. Data Model (`Account` Class)

The application must maintain the state of a bank account using the following structure:

Member Type	Name	Data Type	Description
Property	`AccountNumber`	`string`	The unique identifier for the account.
Property	`Balance`	`decimal`	The current monetary standing of the account.

B. Business Logic Methods

`Deposit(decimal amount)`

- **Success Criteria:** If `amount > 0`, add value to `Balance`.
- **Error Handling:** Implement a single `try-catch` block.
- **Exception:** Throw `ArgumentException` if the amount is non-positive.
- **Message:** "Deposit amount must be positive."
- **Return:** The updated or current `Balance`.

`Withdraw(decimal amount)`

- **Success Criteria:** If `amount > 0` AND `amount <= Balance`, deduct value from `Balance`.

- **Error Handling:** Implement one `try` block with **two** specific `catch` blocks.
- **Exception 1:** `ArgumentException` if `amount <= 0`.
    - *Message:* "Withdrawal amount must be positive."
- **Exception 2:** `InvalidOperationException` if `amount > Balance`.
    - *Message:* "Insufficient funds."
- **Return:** The updated or current `Balance`.

---

3. Interface Requirements (`Program` Class)

The `Main` method must drive the application following this execution flow:

1. **Menu Display:** Present options for Deposit and Withdraw.
2. **Input Collection:** Capture `Choice`, `AccountNumber`, and `Initial Balance`.
3. **Operation Execution:** Prompt for the transaction `amount` and invoke the corresponding method.
4. **Reporting:** Display the final balance to the user.

---

4. Constraint Checklist

- [ ] **Access Modifiers:** All classes and methods must be `public`.
- [ ] **Control Flow:** Do not use `Environment.Exit()`.
- [ ] **Architecture:** Logic must reside in the `Account` class; UI must reside in `Program`.
- [ ] **Validation:** Exceptions must be caught and their messages displayed to the console.

---

5. Expected Test Scenarios

Scenario	Input Amount	Expected Exception/Message	Final Balance Output
Valid Deposit	500	None	Original + 500
Negative Deposit	-100	"Deposit amount must be positive."	Original

Scenario	Input Amount	Expected Exception/Message	Final Balance Output
Valid Withdrawal	200	None	Original - 200
Zero Withdrawal	0	"Withdrawal amount must be positive."	Original
Overdraft	1000 (Bal: 500)	"Insufficient funds."	Original

```
using System;

namespace BankingApplication
{
   // --- Domain Logic Class ---
   public class Account
   {
      public string AccountNumber { get; set; }
      public decimal Balance { get; set; }

      public decimal Deposit(decimal amount)
      {
         try
         {
            if (amount > 0)
            {
```

```
            Balance += amount;
        }
        else
        {
            // Manually triggering the exception for validation
            throw new ArgumentException("Deposit amount must be positive.");
        }
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
    }
    return Balance;
}

public decimal Withdraw(decimal amount)
{
    try
    {
        if (amount <= 0)
        {
            throw new ArgumentException("Withdrawal amount must be positive.");
        }
        else if (amount > Balance)
        {
            throw new InvalidOperationException("Insufficient funds.");
        }
        else
        {
```

```
            Balance -= amount;
        }
    }
    // Handling specific exception types separately
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine(ex.Message);
    }
    return Balance;
  }
}

// --- User Interface / Entry Point ---
public class Program
{
  public static void Main(string[] args)
  {
    Account userAccount = new Account();

    Console.WriteLine("1. Deposit");
    Console.WriteLine("2. Withdraw");
    Console.WriteLine("Enter the choice");

    if (!int.TryParse(Console.ReadLine(), out int choice)) return;
```

```
        Console.WriteLine("Enter the account number");
        userAccount.AccountNumber = Console.ReadLine();

        Console.WriteLine("Enter the balance");
        userAccount.Balance = decimal.Parse(Console.ReadLine());

        if (choice == 1)
        {
           Console.WriteLine("Enter the amount to be deposit");
           decimal amount = decimal.Parse(Console.ReadLine());
           decimal currentBalance = userAccount.Deposit(amount);
           Console.WriteLine("Balance amount " + currentBalance);
        }
        else if (choice == 2)
        {
           Console.WriteLine("Enter the amount to be withdraw");
           decimal amount = decimal.Parse(Console.ReadLine());
           decimal currentBalance = userAccount.Withdraw(amount);
           Console.WriteLine("Balance amount " + currentBalance);
        }
     }
  }
}
```

---

mplementing `TryParse` is a great move for John. It prevents the application from crashing ("blowing up") if a user accidentally types something like "abc" instead of "500". This makes the application **production-ready**.

```
using System;

namespace BankingApplication
{
    public class Account
    {
        public string AccountNumber { get; set; }
        public decimal Balance { get; set; }

        public decimal Deposit(decimal amount)
        {
            try
            {
                if (amount > 0)
                {
                    Balance += amount;
                }
                else
                {
                    throw new ArgumentException("Deposit amount must be positive.");
                }
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine(ex.Message);
            }
            return Balance;
        }
```

```
    public decimal Withdraw(decimal amount)
    {
        try
        {
            if (amount <= 0)
            {
                throw new ArgumentException("Withdrawal amount must be positive.");
            }
            else if (amount > Balance)
            {
                throw new InvalidOperationException("Insufficient funds.");
            }
            else
            {
                Balance -= amount;
            }
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine(ex.Message);
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine(ex.Message);
        }
        return Balance;
    }
}
```

```
public class Program
{
   public static void Main(string[] args)
   {
      Account userAccount = new Account();

      Console.WriteLine("1. Deposit");
      Console.WriteLine("2. Withdraw");
      Console.WriteLine("Enter the choice");

      if (!int.TryParse(Console.ReadLine(), out int choice))
      {
         Console.WriteLine("Invalid choice. Please enter 1 or 2.");
         return;
      }

      Console.WriteLine("Enter the account number");
      userAccount.AccountNumber = Console.ReadLine();

      Console.WriteLine("Enter the balance");
      userAccount.Balance = ReadDecimalInput();

      if (choice == 1)
      {
         Console.WriteLine("Enter the amount to be deposit");
         decimal amount = ReadDecimalInput();
         Console.WriteLine("Balance amount " + userAccount.Deposit(amount));
      }
      else if (choice == 2)
```

```
        {
            Console.WriteLine("Enter the amount to be withdraw");
            decimal amount = ReadDecimalInput();
            Console.WriteLine("Balance amount " + userAccount.Withdraw(amount));
        }
    }

    // Helper method to ensure we get a valid number from the user
    private static decimal ReadDecimalInput()
    {
        decimal result;
        while (!decimal.TryParse(Console.ReadLine(), out result))
        {
            Console.WriteLine("Invalid input. Please enter a valid numeric amount:");
        }
        return result;
    }
  }
}
```