12214994_Saurabh Rana

Scenario-Based Case Studies for C# (Calculator Program)

Case Study 1: Banking System – Basic Arithmetic Operations

Scenario

A **banking application** needs a utility class to perform basic arithmetic operations such as addition, subtraction, multiplication, and division for internal calculations like balance updates, interest calculation, and transaction summaries.

Requirement

- Create a `Calculator` class
- Support:
  - Addition of two numbers
  - Subtraction without user input
  - Multiplication with input parameters
  - Division using predefined values
- Display results where required

C# Concepts Used

- Class and Object
- Default Constructor
- Parameterized Methods
- Return types and void methods

Mapping to Code

```
Add(int number1, int number2)
Subtract()
Multiply(int number1, int number2)
Divide()
```

Learning Outcome

✔ Understand method overloading
✔ Difference between methods with/without parameters
✔ Difference between methods with/without return type

Answer:

```
class Program
{
    static void Main()
    {
        Calculator calc = new Calculator();


        Console.WriteLine("Addition: " + calc.Add(15, 5));
        Console.WriteLine("Subtraction: " + calc.Subtract());
        calc.Multiply(4, 5);
        calc.Divide();
    }
}


class Calculator
{
    int number1;
    int number2;
    public Calculator()
    {
        number1 = 200;
```

```
    number2 = 20;
}

//addition
public int Add(int n1, int n2)
{
    return n1 + n2;
}
//subtraction
public int Subtract()
{
    return number1 - number2;
}
//multiplication
public void Multiply(int n1, int n2)
{
    Console.WriteLine("Multiplication: " + (n1 * n2));
}
//division
public void Divide()
{
    Console.WriteLine("Division: " + (number1 / number2));
```

}

Case Study 2: Billing Software – Constructor Usage

Scenario

A **retail billing system** initializes calculation objects differently based on whether default values or customer-specific values are available.

Requirement

- Initialize calculator with default values
- Initialize calculator with given numbers for immediate calculation

C# Concepts Used

- Default Constructor
- Parameterized Constructor
- `this` keyword

Mapping to Code

```
public Calculator()
public Calculator(int Number1, int Number2)
```

Learning Outcome

✔ Understand object initialization
✔ Use of constructors in real applications

Ansswer:

```
class Program
{
   public static void Main()
   {
      //default constructor
      Calculator calcDefault = new Calculator();
```

```
        calcDefault.Add();
        //parameterized constructor
        Calculator calcParam = new Calculator(50, 70);
        calcParam.Add();


    }
}

class Calculator
{
    private int number1;
    private int number2;

    public Calculator()
    {
        number1 = 140;
        number2 = 120;
    }

    public Calculator(int number1, int number2)
    {
        this.number1 = number1;
        this.number2 = number2;
    }

    public void Add()
    {
        int result = number1 + number2;
        Console.WriteLine("Addition: " + result);
    }
}
```

Case Study 3: Employee Payroll – Call by Value

Scenario

In a **payroll system**, salary values are passed to a method for calculation, but the original values must **remain unchanged** after method execution.

Requirement

- Pass two salary components to a method
- Swap values internally
- Ensure original values remain unchanged

C# Concepts Used

- Call by Value
- Local scope of variables

Mapping to Code

```
public void Swap(int num1, int num2)
```

Observation

Even after swapping inside the method, original values do not change.

Learning Outcome

✔ Understand call by value
✔ Data safety in applications

Answer:

```
class Program
{
    static void Main()
    {
        PayRoll payroll = new PayRoll();


        int salaryComponent1 = 5000;
        int salaryComponent2 = 7000;


        Console.WriteLine("Before Swap:");
        Console.WriteLine("Salary Component 1: " + salaryComponent1);
        Console.WriteLine("Salary Component 2: " + salaryComponent2);
```

```
        payroll.Swap(salaryComponent1, salaryComponent2);

        Console.WriteLine("\nAfter Swap (Outside Method):");
        Console.WriteLine("Salary Component 1: " + salaryComponent1);
        Console.WriteLine("Salary Component 2: " + salaryComponent2);
    }
}
class Program
{
    static void Main()
    {
        PayRoll payroll = new PayRoll();

        int salaryComponent1 = 5000;
        int salaryComponent2 = 7000;

        Console.WriteLine("Before Swap:");
        Console.WriteLine("Salary Component 1: " + salaryComponent1);
        Console.WriteLine("Salary Component 2: " + salaryComponent2);

        payroll.Swap(salaryComponent1, salaryComponent2);

        Console.WriteLine("\nAfter Swap (Outside Method):");
```

```
        Console.WriteLine("Salary Component 1: " + salaryComponent1);

        Console.WriteLine("Salary Component 2: " + salaryComponent2);

    }

}
```

Case Study 4: HR System – Call by Reference

Scenario

An **HR application** allows administrators to permanently swap employee IDs or role priorities.

Requirement

- Swap two values
- Changes must reflect outside the method

C# Concepts Used

- `ref` keyword
- Call by Reference

Mapping to Code

```
public void Swap1(ref int num3, ref int num4)
```

Learning Outcome

✔ Difference between `ref` and normal parameters
✔ Real-time data modification

Answer:

```
class Program

{

    static void Main()
```

```
    {
        HR hr = new HR();

        int employeeID1 = 101;
        int employeeID2 = 202;

        Console.WriteLine("Before Swap:");
        Console.WriteLine("Employee ID 1: " + employeeID1);
        Console.WriteLine("Employee ID 2: " + employeeID2);

        hr.Swap1(ref employeeID1, ref employeeID2);

        Console.WriteLine("\nAfter Swap (Outside Method):");
        Console.WriteLine("Employee ID 1: " + employeeID1);
        Console.WriteLine("Employee ID 2: " + employeeID2);
    }
}
class HR
{
    public void Swap1(ref int num3, ref int num4)
    {
        int temp = num3;
        num3 = num4;
```

```
        num4 = temp;


        Console.WriteLine("\nAfter Swap (Inside Method):");
        Console.WriteLine("Employee ID 1: " + num3);
        Console.WriteLine("Employee ID 2: " + num4);
    }
}
```

Case Study 5: Examination System – Output Parameters (`out`)

Scenario

An **online examination system** needs to calculate total marks and also return individual subject marks back to the caller.

Requirement

- Accept two marks
- Return:
    - Total marks
    - Individual marks back to calling function

C# Concepts Used

- `out` parameters
- Returning multiple values from a method

Mapping to Code

```
public void Addition(int n1, int n2, out int result, out int n3, out int n4)
```

Learning Outcome

✔ Multiple outputs from a single method
✔ Use of `out` keyword in enterprise applications

```
class Program
{
    static void Main()
    {
        Examination exam = new Examination();

        int subjectMark1 = 85;
        int subjectMark2 = 90;

        exam.Addition(subjectMark1, subjectMark2, out int totalMarks, out int returnedMark1, out int returnedMark2);

        Console.WriteLine("Total Marks: " + totalMarks);
        Console.WriteLine("Returned Subject Mark 1: " + returnedMark1);
        Console.WriteLine("Returned Subject Mark 2: " + returnedMark2);
    }
}
class Examination
{
    public void Addition(int n1, int n2, out int result, out int n3, out int n4)
    {
        result = n1 + n2;
```

```
        n3 = n1;

        n4 = n2;

    }

}
```

Case Study 6: Utility Service – Auto-Implemented Properties

Scenario

A **shared utility service** stores input values and results for reuse across multiple calculations.

Requirement

- Store numbers and results
- Provide controlled access

C# Concepts Used

- Auto-implemented properties
- Encapsulation

Mapping to Code

```
public int Number1 { get; set; }
public int Number2 { get; set; }
public int Result { get; set; }
```

Learning Outcome

✔ Clean property implementation
✔ Encapsulation best practices

Answer:

```
class Program

{
```

```
    static void Main()
    {
        UtilityService utilService = new UtilityService();

        utilService.Number1 = 50;
        utilService.Number2 = 70;

        utilService.Result = utilService.Number1 + utilService.Number2;

        Console.WriteLine("Number 1: " + utilService.Number1);
        Console.WriteLine("Number 2: " + utilService.Number2);
        Console.WriteLine("Result (Sum): " + utilService.Result);
    }
}
class UtilityService
{

    public int Number1 { get; set; }
    public int Number2 { get; set; }
    public int Result { get; set; }
}
```

Case Study 7: Console Application – Main Method Integration

Scenario

A **console-based testing application** is used by trainees to test all calculator features from a single entry point.

Requirement

- Create object
- Call calculation method
- Display final output

Mapping to Code

```
Calculator calculator2 = new Calculator();
calculator2.Addition(num1, num2, out result, out num3, out num4);
```

Learning Outcome

✔ Program flow control
✔ Object interaction in Main method

Answer:

class Program

{

static void Main()

{

Examination exam = new Examination();

int subjectMark1 = 85;

int subjectMark2 = 90;

```
        exam.Addition(subjectMark1, subjectMark2, out int totalMarks, out int returnedMark1, out int returnedMark2);



        Console.WriteLine("Total Marks: " + totalMarks);

        Console.WriteLine("Returned Subject Mark 1: " + returnedMark1);

        Console.WriteLine("Returned Subject Mark 2: " + returnedMark2);

    }

}
```

Summary of Concepts Covered

Concept	Used
Class & Object	✅
Constructors	✅
Methods (all types)	✅
Return vs Void	✅
Call by Value	✅
Call by Reference	✅
`out` Parameters	✅
Encapsulation	✅

Assignment Extension (Optional for Students)

1. Add **exception handling** for division by zero
2. Accept user input instead of hardcoded values
3. Convert calculator to **menu-driven program**
4. Add **method overloading** for `Add()`
5. Store calculation history