

Lab 1

Objective: To develop a console application in C# that takes user input for their name and age, and prints a welcome message.

Theory

C# is a modern, object-oriented programming language developed by Microsoft and widely used for building applications on the .NET platform. A console application is a basic form of .NET application that runs in the command line and interacts with users through text.

This lab focuses on the use of:

- `Console.ReadLine()` to receive input from the user.
- `Convert.ToInt32()` to convert string input to integer.
- `Console.WriteLine()` and string interpolation to display output.

Understanding how to handle input and output in a console environment is essential for building interactive applications. These fundamentals form the basis for advanced topics like file handling, database access, and GUI development in .NET.

Code

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter your name: ");
        string name = Console.ReadLine();

        Console.Write("Enter your age: ");
        string ageInput = Console.ReadLine();
        int age = Convert.ToInt32(ageInput);
        Console.WriteLine($"Welcome, {name}! You are {age} years old.");
    }
}
```

Output

```
Enter your name: deep  
Enter your age: 11  
Welcome, deep! You are 11 years old.  
  
=== Code Execution Successful ===
```

Discussion

The code demonstrates how to create a simple C# console application that interacts with the user. The program takes string input for the name and age, converts the age to an integer, and displays a formatted welcome message using string interpolation.

Conclusion

The console application successfully captures user input and displays a customized welcome message. This lab helped in understanding essential input/output operations in C#, providing a strong foundation for building more complex .NET applications.

Lab 2

Objectives: To create a C# program that demonstrates valid and invalid identifiers and shows the use of various C# keywords

Theory

In C#, identifiers are the names used to identify variables, methods, classes, etc. Valid identifiers must follow specific rules:

- Can contain letters, digits, and underscores.
- Must not begin with a digit.
- Cannot be the same as reserved keywords
- Are case-sensitive.

C# also provides a rich set of keywords that serve specific purposes in the language, such as data types (int, string), control structures (if, else), and modifiers (public, private, static).

Understanding valid naming and how to correctly use keywords is critical in writing readable and syntactically correct C# code.

Code

```
using System;

class IdentifierDemo
{
    static void Main(string[] args)
    {
        // Valid identifiers

        int age = 11;           // Simple variable

        string _name = "deep"; // Starts with an underscore

        string @class = "BCA5"; // Reserved keyword with @ prefix


        // Invalid identifiers

        // int 123number = 10; // Starts with a number

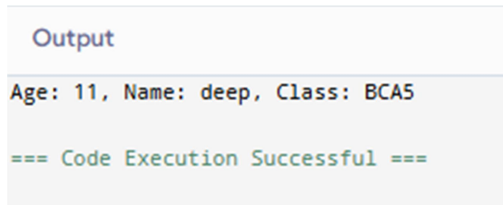
        // string void = "Invalid"; // Reserved keyword without @

        // string user name = "Invalid"; // Contains spaces
    }
}
```

```
// Output valid identifiers

        Console.WriteLine($"Age:    {age},    Name:    {_name},    Class:
{@class}");
    }
}
```

Output



The screenshot shows a code execution environment. At the top, the word "Output" is displayed in a blue font. Below it, the program's output is shown in a monospaced font: "Age: 11, Name: deep, Class: BCAS". At the bottom, a green status message reads "=== Code Execution Successful ===".

Discussion

This program demonstrates:

- The rules for naming identifiers in C#.
- How reserved keywords can be used as identifiers with the @ prefix.
- Examples of common errors when defining identifiers.

Errors from invalid identifiers: If invalid identifier lines are uncommented following compilation errors occurs:

1. Identifiers cannot start with a number.
2. 'void' is a reserved keyword.
3. Invalid syntax due to spaces in identifier names.

Conclusion

The program successfully demonstrates valid and invalid identifiers and shows how reserved keywords can be used as identifiers. By understanding and adhering to syntax rules, developers can avoid common pitfalls and write cleaner, error-free code.

Lab 3

Objective: To modify a C# program by adding appropriate single-line and multi-line comments, making the code more understandable and maintainable.

Theory

Comments in C# are non-executable lines that describe or explain code. They are useful for documentation, debugging, and collaboration.

- **Single-line comments** use `//` and are placed before or next to code statements.
- **Multi-line comments** use `/* */` and are used for larger descriptions or block comments.

Commenting is an essential practice for writing readable and professional code, especially in collaborative or long-term projects.

Code

```
using System;

class CommentedProgram
{
    static void Main()
    {
        /*
            This program collects a user's name and age,
            then prints a personalized welcome message.
        */

        // Ask for the user's name
        Console.Write("Enter your name: ");

        string name = Console.ReadLine(); // Store user input into
'name'

        // Ask for the user's age
        Console.Write("Enter your age: ");
```

```
        string ageInput = Console.ReadLine(); // Read age as a
string

        // Convert age to integer for further processing

        int age = Convert.ToInt32(ageInput);

        // Display a welcome message

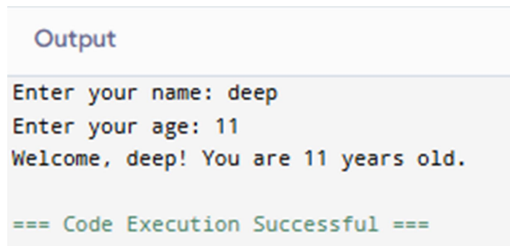
        Console.WriteLine($"Welcome, {name}! You are {age} years
old.");

        // End of the program

    }

}
```

Output



The screenshot shows a terminal window titled "Output". It contains the following text: "Enter your name: deep", "Enter your age: 11", "Welcome, deep! You are 11 years old.", and "=== Code Execution Successful ===".

Discussion

This program demonstrates how to add both single-line and multi-line comments in C#. Each part of the code is explained, making it easier to understand the purpose of every section. Such comments are useful not only for others reading the code but also for the original programmer when revisiting the code after some time.

Conclusion

The modified program now includes explanatory comments that improve code readability and understanding. Adding comments is a best practice in programming, and this lab helps students develop good documentation habits early in their coding journey.

Lab 4

Objective: To develop a basic calculator in C# that demonstrates the use of different data types and various arithmetic and logical operators.

Theory

C# supports several numerical data types, including:

- int – for whole numbers.
- float – for single-precision floating-point numbers.
- double – for double-precision floating-point numbers.

Arithmetic operators include +, -, *, /, % and are used to perform mathematical operations.

Logical operators like &&, ||, and ! are used in boolean expressions for decision making.

This lab demonstrates how to work with different data types and operators in one program to perform basic calculator operations.

Code

```
using System;

class Program
{
    static void ArithmeticCalculator()
    {
        int intNum1 = 5, intNum2 = 3;

        float floatNum1 = 7.5f, floatNum2 = 2.2f;

        double doubleNum1 = 12.345, doubleNum2 = 4.567;

        Console.WriteLine("Arithmetic Operations:");

        // Integer arithmetic

        Console.WriteLine($"Int: {intNum1} + {intNum2} = {intNum1 + intNum2}");

        Console.WriteLine($"Int: {intNum1} - {intNum2} = {intNum1 - intNum2}");

        // Float arithmetic
```

```

        Console.WriteLine($"Float: {floatNum1} * {floatNum2} =
{floatNum1 * floatNum2}");

        Console.WriteLine($"Float: {floatNum1} / {floatNum2} =
{floatNum1 / floatNum2}");

        // Double arithmetic

        Console.WriteLine($"Double: {doubleNum1} + {doubleNum2} =
{doubleNum1 + doubleNum2}");

        Console.WriteLine($"Double: {doubleNum1} - {doubleNum2} =
{doubleNum1 - doubleNum2}");

    }

    static void LogicalCalculator()

    {

        int val1 = 1, val2 = 0;

        Console.WriteLine("\nLogical Operations:");

        Console.WriteLine($"Logical AND: {val1} && {val2} =
{(val1 != 0 && val2 != 0)}");

        Console.WriteLine($"Logical OR: {val1} || {val2} = {(val1 !=
0 || val2 != 0)}");

        Console.WriteLine($"Logical NOT: !{val1} = {!(val1 != 0)}");

    }

    static void Main()

    {

        ArithmeticCalculator();

        LogicalCalculator();

    }

}

```


Output

```
Arithmetic Operations:
Int: 5 + 3 = 8
Int: 5 - 3 = 2
Float: 7.5 * 2.2 = 16.5
Float: 7.5 / 2.2 = 3.409091
Double: 12.345 + 4.567 = 16.912
Double: 12.345 - 4.567 = 7.778

Logical Operations:
Logical AND: 1 && 0 = False
Logical OR: 1 || 0 = True
Logical NOT: !1 = False
```

Discussion

This program performs calculations using three different numeric types (int, float, double) and demonstrates the use of arithmetic operators like +, /, *, %. It also shows how logical operators (&&, !, ==, >) work in C# to evaluate boolean expressions.

By mixing data types and operators, the program reinforces how operations behave differently depending on the context and type used.

Conclusion

The calculator program effectively showcases multiple data types and how they interact with arithmetic and logical operators. This lab builds a practical foundation for writing mathematical and decision-based logic in more complex .NET applications.

Lab 5

Objective: To develop a C# program that performs basic string manipulations

Theory

In C#, the string class provides built-in methods and properties for text manipulation. String operations are essential in many applications like form processing, file reading, and data formatting.

This lab demonstrates three common string operations:

- **Reversing a string:** By converting the string to a character array and reversing it.
- **Counting vowels:** Using a loop and checking each character.
- **Converting to uppercase:** Using the ToUpper() method of the string class.

Mastering these operations helps in handling and processing textual data effectively.

Code

```
using System;

class StringManipulation
{
    static void Main()
    {
        // Input string

        Console.Write("Enter a string: ");

        string input = Console.ReadLine();

        // Reverse the string

        char[] charArray = input.ToCharArray();
        Array.Reverse(charArray);

        string reversed = new string(charArray);

        Console.WriteLine("Reversed String: " + reversed);
    }
}
```

```

        // Count vowels

        int vowelCount = 0;

        foreach (char c in input.ToLower())
        {
            if ("aeiou".Contains(c))
            {
                vowelCount++;
            }
        }

        Console.WriteLine("Number of Vowels: " + vowelCount);

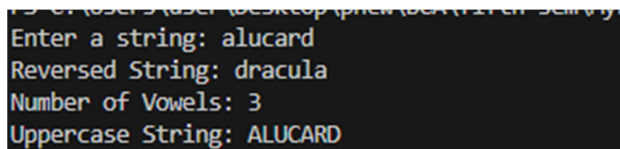
        // Convert to uppercase

        string upperCase = input.ToUpper();

        Console.WriteLine("Uppercase String: " + upperCase);
    }
}

```

Output



```

C:\Users\User\Desktop\New IDE\Programs>
Enter a string: alucard
Reversed String: dracula
Number of Vowels: 3
Uppercase String: ALUCARD

```

Discussion

The program demonstrates key string manipulation tasks:

- Reversing is done by converting the string to a character array and using `Array.Reverse()`.
- Vowel counting uses a loop and `Contains()` with lowercase conversion for case-insensitivity.
- Uppercasing is handled by the `ToUpper()` method.

These are fundamental string operations that form the basis for more advanced tasks like validation, formatting, and parsing.

Conclusion

This lab effectively shows how to manipulate strings using built-in C# methods. It reinforces the importance of string operations in everyday programming tasks and builds confidence in handling textual data.

Lab 6

Objective: To create a C# program that takes 5 integer inputs from the user, stores them in an array, and calculates the total sum and average

Theory

In C#, an array is a fixed-size collection of elements of the same type. Arrays are useful for storing and processing multiple values together.

This lab also demonstrates:

- Accepting input using `Console.ReadLine()` and converting to `int`.
- Calculating **sum** using a loop.
- Computing **average** by dividing the sum by the number of elements.

These operations build essential skills in array handling and numeric processing.

Code

```
using System;

class ArraySumAverage
{
    static void Main()
    {
        int[] numbers = new int[5]; // Array to store 5 integers

        int sum = 0;

        Console.WriteLine("Enter 5 integers:");

        for (int i = 0; i < 5; i++)
        {
            Console.Write($"Number {i + 1}: ");

            numbers[i] = Convert.ToInt32(Console.ReadLine());

            sum += numbers[i];
        }

        double average = (double)sum / numbers.Length;
```

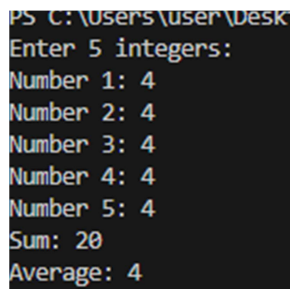
```
        Console.WriteLine("Sum: " + sum);

        Console.WriteLine("Average: " + average);

    }

}
```

Output



```
PS C:\Users\user\Desktop>
Enter 5 integers:
Number 1: 4
Number 2: 4
Number 3: 4
Number 4: 4
Number 5: 4
Sum: 20
Average: 4
```

Discussion

This program collects user input, stores it in an array, and uses a loop to calculate the **sum**. The average is computed using typecasting to ensure decimal division.

This reinforces how arrays and loops can be combined to solve basic data processing problems, such as calculating totals and averages.

Conclusion

The program successfully demonstrates array usage in C#, user input collection, and numeric operations. Understanding how to work with arrays and calculate statistical values is vital in many real-world programming scenarios.

Lab 7

Objective: To write a C# program that demonstrates the difference between passing parameters by value and by reference using methods

Theory

In C#, method parameters can be passed:

- By value (default): A copy of the variable is passed. Changes made inside the method do not affect the original variable.
- By reference (ref keyword): The method works with the original variable. Changes made inside the method affect the original variable.

This concept is important for memory management and understanding how variables behave across method calls.

Code

```
using System;

class ValueVsReference
{
    // Method that takes parameter by value
    static void PassByValue(int number)
    {
        number = number + 10;

        Console.WriteLine("Inside PassByValue: " + number);
    }

    // Method that takes parameter by reference
    static void PassByReference(ref int number)
    {
        number = number + 10;

        Console.WriteLine("Inside PassByReference: " + number);
    }
}
```

```

static void Main()
{
    int a = 5;

    int b = 5;

    Console.WriteLine("Original value of a (by value): " + a);

    PassByValue(a);

    Console.WriteLine("After PassByValue: " + a);

    Console.WriteLine();

    Console.WriteLine("Original value of b (by reference): " +
b);

    PassByReference(ref b);

    Console.WriteLine("After PassByReference: " + b);

}
}

```

Output

```

Original value of a (by value): 5
Inside PassByValue: 15
After PassByValue: 5

Original value of b (by reference): 5
Inside PassByReference: 15
After PassByReference: 15

```

Discussion

- In PassByValue(), the original variable a remains unchanged because the method only worked on a copy.
- In PassByReference(), the original variable b is updated since the method operated on the original memory location.

This difference is essential when deciding how to manipulate data within functions, especially with large objects or when you want a method to modify original data.

Conclusion

The program clearly demonstrates how passing by value and by reference behaves differently in C#. Understanding this concept helps in designing methods that either preserve or modify input data intentionally.

Lab 8

Objective: To create a C# program that determines and prints a student's grade based on their marks using both if-else and switch-case statements

Theory

Control flow statements like if-else and switch-case are used to make decisions based on conditions:

- if-else is best for range-based decisions.
- switch-case is suitable for discrete values.

This program uses if-else to determine the grade range and a switch-case to display a message based on the grade.

Code

```
using System;

class StudentGrade
{
    static void Main()
    {
        Console.Write("Enter marks (0-100): ");

        int marks = Convert.ToInt32(Console.ReadLine());

        string grade;

        if (marks >= 90 && marks <= 100)
            grade = "A";

        else if (marks >= 80)
            grade = "B";

        else if (marks >= 70)
            grade = "C";

        else if (marks >= 60)
            grade = "D";

        else if (marks >= 0)
```

```
        grade = "F";

else

    grade = "Invalid";

Console.WriteLine("Grade: " + grade);

switch (grade)
{
    case "A":

        Console.WriteLine("Excellent!");

        break;

    case "B":

        Console.WriteLine("Very Good!");

        break;

    case "C":

        Console.WriteLine("Good.");

        break;

    case "D":

        Console.WriteLine("Needs Improvement.");

        break;

    case "F":

        Console.WriteLine("Fail. Try harder next time.");

        break;

    default:

        Console.WriteLine("Invalid input.");

        break;

}

}

}
```

Output

```
Enter marks (0-100): 86
Grade: B
Very Good!
PS C:\Users\user\Desktop> gcc.c
```

Discussion

The program uses if-else to categorize marks into grade letters (A–F). Then, a switch-case statement is used to print a corresponding message based on the grade. This approach shows how to combine both decision-making constructs effectively in a real-world scenario.

Conclusion

This lab demonstrates how to use if-else for complex conditions and switch-case for specific outcomes. These control structures are fundamental in making programs behave dynamically based on input.

Lab 9

Objective: To write a C# program that generates the Fibonacci series up to N terms using all three types of loops: for, while, and do-while

Theory

The Fibonacci series is a sequence where each term is the sum of the two preceding ones, starting from 0 and 1.

Example: 0, 1, 1, 2, 3, 5, 8...

This lab shows how different loops (for, while, do-while) can be used to perform the same logic with slight variations in structure and flow control.

Code

```
using System;

class FibonacciDemo
{
    static void Main()
    {
        Console.WriteLine("Enter the number of terms (N): ");
        int n = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("\nUsing for loop:");
        int a = 0, b = 1, c;
        for (int i = 1; i <= n; i++)
        {
            Console.Write(a + " ");

            c = a + b;
            a = b;
            b = c;
        }
    }
}
```

```
Console.WriteLine("\n\nUsing while loop:");

a = 0; b = 1;

int count = 1;

while (count <= n)
{
    Console.Write(a + " ");

    c = a + b;

    a = b;

    b = c;

    count++;
}

Console.WriteLine("\n\nUsing do-while loop:");

a = 0; b = 1;

count = 1;

if (n > 0)
{
    do
    {
        Console.Write(a + " ");

        c = a + b;

        a = b;

        b = c;

        count++;

    } while (count <= n);
}
```

```
}  
  
}
```

Output

```
Enter the number of terms (N): 3
```

```
Using for loop:
```

```
0 1 1
```

```
Using while loop:
```

```
0 1 1
```

```
Using do-while loop:
```

```
0 1 1
```

Discussion

The program calculates the Fibonacci series using:

- A for loop for fixed iterations.
- A while loop with manual incrementing.
- A do-while loop which guarantees at least one iteration.

Each loop correctly produces the same output, showing how different control structures can be used interchangeably depending on the situation.

Conclusion

This lab successfully demonstrates generating the Fibonacci sequence using all three fundamental loop types. Understanding these loop structures is essential for iterative tasks in programming.

Lab 10

Objective: To create a user-defined namespace `PNC` and define a `Student` class inside it with fields for `Name`, `Gender`, and `RollNo`, including a constructor to initialize these fields.

Theory

Code

Output

Discussion

Conclusion