GLA University Mathura

Academic Year 2023-2024

Department: MCA

C#

Lab Practical File

Submitted To:-
Mr. Sachindra Singh Chauhan
Assistant Professor
(CEA Dept.)

Submitted By:-
Naresh Varshney
MCA - A
Roll No:52
Batch :- A2

2284200130

Ques 1: Write the installation steps of Visual Studio?

Ans :

1. **Get Visual Studio:**
    - Head over to the Microsoft Visual Studio website at https://visualstudio.microsoft.com/.
    - Pick your preferred edition (such as Community).
    - Download the provided installer.
2. **Run the Installer:**
    - Run the installer you've downloaded**.**
3. **Customize Your Installation:**
    - Tailor your installation by choosing development workloads (like ".NET desktop," "ASP.NET," etc.).
    - Optionally, select specific components according to your requirements**.**
4. **Initiate the Installation:**
    - Click on "Install" to kick off the installation process.
    - Wait patiently until the installation wraps up**.**
5. **Launch and Setup:**
    - Open Visual Studio once the installation is complete.
    - Sign in or create a Microsoft account if desired (this step is optional).
    - Opt for default development settings.
    - Start utilizing Visual Studio for your project needs.

These steps encompass the key actions involved in downloading, installing, customizing, and configuring Visual Studio to cater to your development endeavors.

Ques 2: Develop a .net platform program with classes library and functions.

Ans :

Step 1: Class Library Creation

1. **Open Visual Studio:**
    - Launch your Visual Studio application.
2. **Create a Class Library Project:**
    - Start by selecting "Create a new project."
    - Choose "Class Library" within the relevant language category (e.g., C#) under "Class Library."
    - Name your project (e.g., NareshLibrary) and hit "Create."
3. **Define a Class and Function:**
    - Access the default class file (e.g., Class1.cs).
    - Replace its content with a simple mathematical function.
4. **Build the Class Library**:
    - Execute the build process for the solution to identify and resolve any potential errors.

Step 2: Console Application Setup

1. **Add a Console Application to the Solution**:
    - Right-click on the solution in the Solution Explorer.
    - Navigate to "Add" -> "New Project."
    - Choose "Console App" in the language of choice (e.g., C#).
    - Name your project (e.g., ConsoleApp) and create it.
2. **Reference the Class Library**:
    - Right-click on the console application project.

- Select "Add" -> "Reference."
- Pick the class library project (MathLibrary) and confirm with "OK."

3. **Utilize the Class Library in the Console Application**:

    - Open the Program.cs file within the console application.

Step 3 : Build and Execute:

- Build the solution to verify and compile the combined projects.
- Run the console application to see the functionality with the incorporated class library.

Ques 3. Write in detail about Conditional statement in c#?

Ans :

Conditional statements in C# enable you to control the flow of your code based on certain conditions. Here are the primary conditional statements:

1. if Statement:

The **if** statement executes a block of code only if a specified condition is true.

```
int num = 10;

if (num > 0)

{        // Code executes if 'num' is greater than 0

         Console.WriteLine("Number is positive");

}
```

2. else Statement:

The **else** statement is paired with an **if** statement and executes a block of code if the **if** condition is false.

```
int num = -5; if (num > 0)

{

         Console.WriteLine("Number is positive");

}

else

{

         // Code executes if 'num' is not greater than 0

         Console.WriteLine("Number is non-positive");

}
```

3. else if Statement:

The **else if** statement allows checking multiple conditions after the initial **if** statement.

```
int num = 0;

if (num > 0)

{

        Console.WriteLine("Number is positive");

}

else if (num < 0)

{

        Console.WriteLine("Number is negative");

}

else

{

         // Code executes if 'num' is neither greater nor less than 0

        Console.WriteLine("Number is zero");

}
```

4. Nested if Statements:

You can place an **if** statement inside another **if** statement, allowing for more complex conditional checks.

```
int num1 = 10, num2 = 20;

if (num1 > 0)

{

        if (num2 > 0)

        {

                Console.WriteLine("Both numbers are positive");

        }
        else
        {
                Console.WriteLine("Number 1 is positive but Number 2 is non-positive");
        }
}
```

5. switch Statement:

The **switch** statement allows selecting one of many code blocks to be executed based on different cases.

```
int day = 3;
switch (day)
{
        case 1: Console.WriteLine("Monday");
        break;
        case 2: Console.WriteLine("Tuesday");
        break;
        // More cases can follow...
        default: Console.WriteLine("Invalid day");
        break;
}
```

These conditional statements provide ways to control the program's flow based on conditions, enabling you to create dynamic and responsive code.

Ques 4: Write about Loops statements in C#?

Ans :

Loops in C# are structures that allow you to execute a block of code repeatedly based on a condition. There are several types of loops available:

1. for Loop:

The for loop repeats a block of code a specified number of times.

```
for (int i = 0; i < 5; i++)

{

    Console.WriteLine($"Number: {i}");

}
```

- The loop starts with an initialization (int i = 0), followed by a condition (i < 5), and an increment/decrement (i++) statement.

2. while Loop:

The while loop executes a block of code as long as a specified condition is true.

```
int counter = 0;

while (counter < 5)

{

    Console.WriteLine($"Counter: {counter}");

    counter++;

}
```

- The loop checks the condition (counter < 5) before executing the block of code.

3. do-while Loop:

The do-while loop executes a block of code at least once and then repeatedly executes it as long as a specified condition is true.

```
int number = 5;

do
```

```
{
    Console.WriteLine($"Number: {number}");
    number--;
} while (number > 0);
```

- The loop checks the condition (number > 0) after executing the block of code.

4. foreach Loop:

The foreach loop iterates over a collection of items (arrays, lists, etc.).

```
int[] numbers = { 1, 2, 3, 4, 5 };
foreach (int n in numbers)
{
    Console.WriteLine($"Number: {n}");
}
```

- It iterates through each element in the collection (numbers) without needing an explicit index.

These loops offer flexibility in controlling how many times a block of code runs or how it interacts with collections. They provide efficient ways to perform repetitive tasks and iterate through data structures.

Ques 5: Write about Classes in C# in detail?

Ans :
In C#, classes serve as the blueprint for creating objects—a fundamental aspect of object-oriented programming (OOP). They encapsulate data and behavior into a single unit. Here's an overview of classes in C#:

1. Class Declaration:

```
// Class declaration

public class MyClass

{

        // Fields (attributes/variables)

        public int myField;

        private string myPrivateField;

        // Constructor

        public MyClass()

        {

                // Constructor initializes the object when created

                myField = 10;

                myPrivateField = "Hello";

        }

        // Methods (functions)

        public void MyMethod()

        {

        // Method logic

        Console.WriteLine("MyMethod called");

        }

}
```

2. Fields:

- Fields are variables that hold data within a class.
- They can be accessed and modified by the class methods and other classes depending on their access level (public, private, protected, etc.).

3. Constructor:

- Constructors initialize objects of the class when they are created.
- They have the same name as the class and may have parameters.
- They set initial values to class fields or perform other necessary initialization tasks.

4. Methods:

- Methods are functions defined inside a class.
- They contain the behavior or actions that objects of the class can perform.
- Methods can take parameters and return values.

5. Access Modifiers:

- C# uses access modifiers (public, private, protected, etc.) to control the accessibility of class members (fields, methods, constructors) from other parts of the program.
- public: Accessible from anywhere.
- private: Accessible only within the same class.
- protected: Accessible within the same class or derived classes.

6. Creating Objects (Instantiating a Class):

```
// Creating an instance/object of MyClass

MyClass obj = new MyClass();

// Accessing fields and methods

obj.myField = 20;

obj.MyMethod();
```

7. Inheritance:

- Classes can inherit properties and behavior from other classes through inheritance.

- The : (colon) syntax is used to derive a class from another class.

```
public class MyDerivedClass : MyClass

{

        // Additional fields and methods specific to the derived class

}
```

Classes form the building blocks of C# programs, allowing for encapsulation, abstraction, inheritance, and polymorphism—essential principles of object-oriented programming. They facilitate the creation of structured and reusable code by organizing data and functionality into manageable units.

Ques 6 : Write about delegates in C#?

Ans:

Delegate Declaration:

To declare a delegate, define its method signature using the delegate keyword.

```
public delegate void MyDelegate(string message);
```

Here, MyDelegate is a delegate capable of referencing methods that take a string parameter and return void.

Using Delegates:

After declaring a delegate, it can be instantiated and linked with methods that match its signature.

```
public class MyClass

{

   public void PerformAction1(string message)

   {

      Console.WriteLine($"PerformAction1: {message}");

   }


   public void PerformAction2(string message)

   {

      Console.WriteLine($"PerformAction2: {message}");

   }

}


class Program
```

```
{
   static void Main()
   {
      MyClass myActions = new MyClass();
      // Creating delegate instances and associating them with methods
      MyDelegate actionDelegate1 = myActions.PerformAction1;
      MyDelegate actionDelegate2 = myActions.PerformAction2;

      // Invoking delegates
      actionDelegate1("Hello");
      actionDelegate2("World");
   }
}
```

Multicast Delegates:

Combining delegates creates a multicast delegate, enabling the invocation of multiple methods.

```
MyDelegate combinedDelegate = actionDelegate1 + actionDelegate2;
combinedDelegate("Combined invocation");
```

Built-in Delegates:

C# offers pre-defined generic delegate types like Action and Func in the System namespace, reducing the need for custom delegates.

```
Action<string> predefinedActionDelegate = myActions.PerformAction1;
predefinedActionDelegate("Using Action");
Func<int, int, int> addDelegate = (a, b) => a + b;
int result = addDelegate(3, 5);
Console.WriteLine($"Result: {result}");
```

Events:

Delegates are commonly used to implement events, allowing one object to notify others when specific events occur.

```
public class EventPublisher

{

   public event MyDelegate OnEventTriggered;

   public void TriggerEvent(string message)

   {

      OnEventTriggered?.Invoke(message);

   }

}

class Program

{

   static void Main()

   {

      EventPublisher eventSource = new EventPublisher();

      MyClass eventSubscriber = new MyClass();

      eventSource.OnEventTriggered += eventSubscriber.PerformAction1;

      eventSource.TriggerEvent("Event triggered");

   }

}
```

Ques 7: Write about File Handling in C#?

Ans:

1.Reading from a File:

```
string filePath = "example.txt";

if (File.Exists(filePath))

{

    string[] lines = File.ReadAllLines(filePath);

    // Process lines

}
```

2.Writing to a File:

```
string filePath = "example.txt";

string[] lines = { "Line 1", "Line 2", "Line 3" };

File.WriteAllLines(filePath, lines);
```

3.Appending to a File:

```
string filePath = "example.txt";

string[] newLines = { "New Line 1", "New Line 2" };

File.AppendAllLines(filePath, newLines);
```

4.Reading and Writing Binary Files:

```
string filePath = "binaryfile.bin";

byte[] data = { 0x48, 0x65, 0x6C, 0x6C, 0x6F };

File.WriteAllBytes(filePath, data);

byte[] buffer = File.ReadAllBytes(filePath);
```

Ques 8 :Create a windows form app individual studio with C#

Ans :

Steps to Create a Windows Forms Application:

1.Open Visual Studio:

- Launch Visual Studio.

2.Create a New Project:

- Go to "File" -> "New" -> "Project..."
- In the "Create a new project" dialog, select "Windows Forms App (.NET Core)" or "Windows Forms App (.NET Framework)" based on your preference and system setup.
- Click "Next."

3.Configure Project:

- Enter a name for your project.
- Choose the location where you want to save the project.
- Set the solution name (optional).
- Choose the framework version (e.g., .NET Core 3.1 or .NET Framework 4.8).
- Click "Create."

4.Design the Form:

- Once the project is created, you'll see the default form (Form1.cs) in the designer.
- You can design your form by dragging and dropping controls from the Toolbox (View -> Toolbox) onto the form.
- Customize the properties of the controls using the Properties window.

5.Add Code to the Form:

- Double-click on a control to create an event handler.
- Add your C# code to handle events and perform actions.
- For example, you can add code to the button click event:

```
private void button1_Click(object sender, EventArgs e)

{
    MessageBox.Show("Hello, Windows Forms!");

}
```

6.Build and Run:

- Build your project by clicking on "Build" -> "Build Solution."
- Run your application by pressing F5 or clicking on the "Start Debugging" button.

That's it! You've created a simple Windows Forms application. You can further enhance your application by adding more controls, implementing additional functionality, and exploring features provided by the Windows Forms framework.

Ques 9: Write about Assemblies in .net?

Ans :

Assemblies in .NET serve as vital units for deployment, version control, reuse, and security permissions within applications. They encapsulate types and resources that collaborate to form a logical unit of functionality. Assemblies typically manifest as executable (.exe) or dynamic link library (.dll) files, acting as the foundational elements of .NET applications. They equip the Common Language Runtime (CLR) with necessary information about type implementations.

There exist two primary types of assemblies in .NET:

Private Assemblies: These are exclusively accessible to the application that generates them. They remain isolated, unable to be utilized by other applications. This isolation prevents conflicts and side effects, aiding in a more controlled environment.

Advantages:

Enhanced isolation and reduced potential for conflicts

Simplified deployment, bundled alongside the application

Independent versioning control

Reduced security risks due to limited accessibility

Drawbacks:

Complex deployment for applications relying on multiple private assemblies

Additional testing efforts to ensure compatibility and avoid conflicts

Limited reusability across different applications

Shared Assemblies: Intended for use by multiple applications, these assemblies are typically stored in the Global Assembly Cache (GAC). This shared location allows various applications to reference and utilize them without needing individual copies. Shared assemblies facilitate code reuse, simplifying deployment across multiple applications.

Advantages:

Encourages code reuse, reducing development time

Simplifies deployment by referencing the assembly from the GAC

Centralized management and versioning in the GAC

Additional Information:

Promotes improved performance due to pre-loading in memory

Both private and shared assemblies offer distinct advantages and drawbacks. While private assemblies ensure isolation and simplified deployment for their respective applications, shared assemblies enable code reuse and centralized management but may require extra care to ensure compatibility and versioning across multiple applications.

Ques 10: Design a Window based calculator in C# ?

Ans :

1.Create a new Windows Forms Application:

- Open Visual Studio.
- Create a new project: "File" -> "New" -> "Project..."
- Choose "Windows Forms App (.NET Core)" or "Windows Forms App (.NET Framework)" based on your preference.
- Name your project and click "Create."

2.Design the Calculator Form:

- In the Solution Explorer, double-click on "Form1.cs" to open the designer.
- Drag and drop buttons for digits (0-9), operators (+, -, *, /), equals (=), clear (C), and a TextBox for display.

3.Add Code to Handle Button Clicks:

Double-click on each button to create event handlers. Add the following code to handle button clicks and perform calculations:

```
using System;

using System.Windows.Forms;

namespace RenamedCalculator

{

    public partial class CalculationForm : Form

    {

        private string currentEntry = "";

        private double currentResult = 0;

        private char currentOperation;

        public CalculationForm()

        {

            InitializeComponent();
        }
        private void NumberButtonClick(object sender, EventArgs e)
        {
            Button button = (Button)sender;
            currentEntry += button.Text;
            DisplayEntry(currentEntry);
        }
        private void OperationButtonClick(object sender, EventArgs e)
        {
            Button button = (Button)sender;
            if (!string.IsNullOrEmpty(currentEntry))
            {
                currentResult = double.Parse(currentEntry);
                currentEntry = "";
                currentOperation = button.Text[0];
            }
        }
        private void EqualsButtonClick(object sender, EventArgs e)
        {
            if (!string.IsNullOrEmpty(currentEntry))
            {
                double secondResult = double.Parse(currentEntry);
                double result = PerformComputation(currentResult, secondResult,
currentOperation);
                DisplayEntry(result.ToString());
                currentEntry = result.ToString();
            }
        }
        private void ClearButtonClick(object sender, EventArgs e)
        {
            currentEntry = "";
            currentResult = 0;
            currentOperation = '\0';
            DisplayEntry("");
        }
        private double PerformComputation(double firstResult, double secondResult, char
operation)
        {
            switch (operation)
            {
                case '+':
                    return firstResult + secondResult;
                case '-':
                    return firstResult - secondResult;
                case '*':
                    return firstResult * secondResult;
                case '/':
                    return secondResult != 0 ? firstResult / secondResult : double.NaN;
                default:
                    return double.NaN;
            }

        }

        private void DisplayEntry(string text)

        {

            textBox1.Text = text;

        }

    }

}
```

4.Build and Run:

- Build your project by clicking on "Build" -> "Build Solution."
- Run your application by pressing F5 or clicking on the "Start Debugging" button.