SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 2nd semester
PROGRAMMING USING C# (PCA20D05J)

Lab Manual

Lab 1: Initialization and Declaration, Data types

Aim: To understand and implement variable initialization, declaration, and various data types in C#.

Procedure:

1. Open Visual Studio and create a new C# Console Application project.
2. In the `Program.cs` file, write code to declare variables of different data types (e.g., `int`, `double`, `char`, `string`, `bool`).
3. Initialize these variables with appropriate values.
4. Use `Console.WriteLine()` to display the values and their types.
5. Compile and run the program to observe the output.

Source Code:

```
using System;

public class Lab1
{
    public static void Main(string[] args)
    {
        // Variable Declaration and Initialization
        int integerValue = 10;
        double doubleValue = 20.5;
        char charValue = 'A';
        string stringValue = "Hello C#";
        bool boolValue = true;

        Console.WriteLine("Integer Value: " + integerValue);
        Console.WriteLine("Double Value: " + doubleValue);
        Console.WriteLine("Character Value: " + charValue);
        Console.WriteLine("String Value: " + stringValue);
        Console.WriteLine("Boolean Value: " + boolValue);

        // Demonstrating type inference (var keyword)
        var inferredInt = 100;
        var inferredString = "Inferred Type";
        Console.WriteLine("Inferred Integer: " + inferredInt);
        Console.WriteLine("Inferred String: " + inferredString);
    }
}
```

Input: No specific input required for this program.

Expected Output:

```
Integer Value: 10
Double Value: 20.5
Character Value: A
String Value: Hello C#
Boolean Value: True
Inferred Integer: 100
Inferred String: Inferred Type
```

Lab 2: Control Statements

Aim: To implement and understand conditional (`if-else`, `switch`) and looping (`for`, `while`, `do-while`, `foreach`) control statements in C#.

Procedure:

1. Create a new C# Console Application.
2. Write code to demonstrate an `if-else` statement based on a user-provided number.
3. Implement a `switch` statement to handle different cases for a character input.
4. Use a `for` loop to print numbers from 1 to 5.
5. Use a `while` loop to sum numbers until a certain condition is met.
6. Compile and run the program.

Source Code:

```
using System;

public class Lab2
{
    public static void Main(string[] args)
    {
        // If-Else Statement
        Console.Write("Enter a number: ");
        int num = Convert.ToInt32(Console.ReadLine());

        if (num > 0)
        {
            Console.WriteLine("The number is positive.");
        }
        else if (num < 0)
        {
            Console.WriteLine("The number is negative.");
        }
        else
        {
            Console.WriteLine("The number is zero.");
        }

        // Switch Statement
        Console.Write("Enter a grade (A, B, C, D, F): ");
        char grade = Convert.ToChar(Console.ReadLine().ToUpper());

        switch (grade)
        {
            case 'A':
                Console.WriteLine("Excellent!");
                break;
            case 'B':
                Console.WriteLine("Good!");
                break;
            case 'C':
                Console.WriteLine("Fair.");
                break;
            case 'D':
                Console.WriteLine("Pass.");
                break;
            case 'F':
                Console.WriteLine("Fail.");
```

```
                break;
            default:
                Console.WriteLine("Invalid grade.");
                break;
        }

        // For Loop
        Console.WriteLine("\nNumbers from 1 to 5 (using for loop):");
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine(i);
        }

        // While Loop
        int count = 0;
        Console.WriteLine("\nCounting to 3 (using while loop):");
        while (count < 3)
        {
            Console.WriteLine("Count: " + count);
            count++;
        }
    }
}
```

Input:

```
Enter a number: 5
Enter a grade (A, B, C, D, F): A
```

Expected Output:

```
The number is positive.
Excellent!

Numbers from 1 to 5 (using for loop):
1
2
3
4
5

Counting to 3 (using while loop):
Count: 0
Count: 1
Count: 2
```

Lab 3: Arrays

Aim: To learn how to declare, initialize, and manipulate single-dimensional and multi-dimensional arrays in C#.

Procedure:

1. Create a new C# Console Application.
2. Declare and initialize a single-dimensional integer array.
3. Iterate through the array using a `for` loop and print its elements.
4. Declare and initialize a two-dimensional array (matrix).
5. Use nested `for` loops to print the elements of the two-dimensional array.
6. Compile and run the program.

Source Code:

```
using System;

public class Lab3
{
    public static void Main(string[] args)
    {
        // Single-dimensional array
        int[] numbers = { 10, 20, 30, 40, 50 };
        Console.WriteLine("Elements of the single-dimensional array:");
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine("Element at index " + i + ": " + numbers[i]);
        }

        // Multi-dimensional array (2x3 matrix)
        int[,] matrix = { { 1, 2, 3 }, { 4, 5, 6 } };
        Console.WriteLine("\nElements of the two-dimensional array (matrix):");
        for (int i = 0; i < matrix.GetLength(0); i++) // Rows
        {
            for (int j = 0; j < matrix.GetLength(1); j++) // Columns
            {
                Console.Write(matrix[i, j] + "\t");
            }
            Console.WriteLine(); // New line after each row
        }
    }
}
```

Input: No specific input required for this program.

Expected Output:

```
Elements of the single-dimensional array:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50

Elements of the two-dimensional array (matrix):
1       2       3
```

4 5 6

Lab 4: Classes, Constructors

Aim: To understand object-oriented programming concepts by creating classes and using constructors in C#.

Procedure:

1. Create a new C# Console Application.
2. Define a class named `Person` with properties like `Name` and `Age`.
3. Implement a default constructor and a parameterized constructor for the `Person` class.
4. Create objects of the `Person` class using both constructors.
5. Call a method (e.g., `DisplayInfo()`) on the created objects to show their details.
6. Compile and run the program.

Source Code:

```
using System;

// Define a class
public class Person
{
    // Properties
    public string Name { get; set; }
    public int Age { get; set; }

    // Default Constructor
    public Person()
    {
        Name = "Unknown";
        Age = 0;
        Console.WriteLine("Default constructor called.");
    }

    // Parameterized Constructor
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
        Console.WriteLine("Parameterized constructor called for " + name);
    }

    // Method to display information
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}

public class Lab4
{
    public static void Main(string[] args)
    {
        // Create an object using the default constructor
        Person person1 = new Person();
        person1.DisplayInfo();

        Console.WriteLine();

        // Create an object using the parameterized constructor
        Person person2 = new Person("Alice", 30);
```

```
        person2.DisplayInfo();

        Console.WriteLine();

        Person person3 = new Person("Bob", 25);
        person3.DisplayInfo();
    }
}
```

Input: No specific input required for this program.

Expected Output:

```
Default constructor called.
Name: Unknown, Age: 0

Parameterized constructor called for Alice
Name: Alice, Age: 30

Parameterized constructor called for Bob
Name: Bob, Age: 25
```

Lab 5: Inheritance

Aim: To implement inheritance in C# to demonstrate code reusability and hierarchical relationships between classes.

Procedure:

1. Create a new C# Console Application.
2. Define a base class named `Animal` with a method like `Eat()`.
3. Define a derived class named `Dog` that inherits from `Animal`.
4. Add a specific method to the `Dog` class, e.g., `Bark()`.
5. Create an object of the `Dog` class and call both inherited and specific methods.
6. Compile and run the program.

Source Code:

```
using System;

// Base class
public class Animal
{
    public string Name { get; set; }

    public Animal(string name)
    {
        Name = name;
    }

    public void Eat()
    {
        Console.WriteLine($"{Name} is eating.");
    }
}

// Derived class inheriting from Animal
public class Dog : Animal
{
    public string Breed { get; set; }

    public Dog(string name, string breed) : base(name) // Call base class
constructor
    {
        Breed = breed;
        Console.WriteLine($"{Name} the {Breed} is a dog.");
    }

    public void Bark()
    {
        Console.WriteLine($"{Name} is barking: Woof! Woof!");
    }
}

public class Lab5
{
    public static void Main(string[] args)
    {
        // Create an object of the derived class
        Dog myDog = new Dog("Buddy", "Golden Retriever");

        // Call inherited method
```

```
        myDog.Eat();

        // Call specific method of the derived class
        myDog.Bark();
    }
}
```

Input: No specific input required for this program.

Expected Output:

```
Buddy the Golden Retriever is a dog.
Buddy is eating.
Buddy is barking: Woof! Woof!
```

Lab 6: Interface, Operator Overloading

Aim: To understand and implement interfaces and operator overloading in C#.

Procedure:

1. Create a new C# Console Application.
2. Define an interface, e.g., `IShape` with a method `CalculateArea()`.
3. Implement the `IShape` interface in a class, e.g., `Rectangle`.
4. Demonstrate operator overloading by defining an overloaded `+` operator for a custom class, e.g., `Vector2D`.
5. Create objects and use the overloaded operator.
6. Compile and run the program.

Source Code:

```
using System;

// Define an interface
public interface IShape
{
    double CalculateArea();
}

// Implement the interface in a class
public class Rectangle : IShape
{
    public double Length { get; set; }
    public double Width { get; set; }

    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double CalculateArea()
    {
        return Length * Width;
    }
}

// Class for operator overloading
public class Vector2D
{
    public int X { get; set; }
    public int Y { get; set; }

    public Vector2D(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Overload the '+' operator
    public static Vector2D operator +(Vector2D v1, Vector2D v2)
    {
        return new Vector2D(v1.X + v2.X, v1.Y + v2.Y);
    }
```

```
    public override string ToString()
    {
        return $"({X}, {Y})";
    }
}

public class Lab6
{
    public static void Main(string[] args)
    {
        // Interface demonstration
        Rectangle rect = new Rectangle(5, 4);
        Console.WriteLine($"Area of Rectangle: {rect.CalculateArea()}");

        // Operator Overloading demonstration
        Vector2D vec1 = new Vector2D(1, 2);
        Vector2D vec2 = new Vector2D(3, 4);
        Vector2D sumVec = vec1 + vec2; // Using overloaded '+' operator

        Console.WriteLine($"\nVector 1: {vec1}");
        Console.WriteLine($"Vector 2: {vec2}");
        Console.WriteLine($"Sum of Vectors: {sumVec}");
    }
}
```

Input: No specific input required for this program.

Expected Output:

```
Area of Rectangle: 20

Vector 1: (1, 2)
Vector 2: (3, 4)
Sum of Vectors: (4, 6)
```

Lab 7: Delegates

Aim: To understand and implement delegates for type-safe function pointers in C#.

Procedure:

1. Create a new C# Console Application.
2. Define a delegate type that matches the signature of a method.
3. Create a few methods that match the delegate's signature.
4. Create instances of the delegate and assign methods to them.
5. Invoke the methods using the delegate instances.
6. Demonstrate multicast delegates by adding multiple methods to a single delegate instance.
7. Compile and run the program.

Source Code:

```
using System;

public class Lab7
{
    // 1. Define a delegate type
    public delegate void MyDelegate(string message);

    // Methods that match the delegate's signature
    public static void Method1(string msg)
    {
        Console.WriteLine("Method1 called: " + msg);
    }

    public static void Method2(string msg)
    {
        Console.WriteLine("Method2 called: " + msg.ToUpper());
    }

    public static void Main(string[] args)
    {
        // 4. Create instances of the delegate and assign methods
        MyDelegate del1 = new MyDelegate(Method1);
        MyDelegate del2 = Method2; // Shorthand syntax

        // 5. Invoke methods using delegate instances
        Console.WriteLine("--- Single Delegate Invocation ---");
        del1("Hello from delegate 1!");
        del2("hello from delegate 2!");

        // 6. Multicast Delegate
        Console.WriteLine("\n--- Multicast Delegate Invocation ---");
        MyDelegate multiDel = del1 + del2; // Combine delegates
        multiDel("This message goes to both methods.");

        // Remove a method from multicast delegate
        multiDel -= del1;
        Console.WriteLine("\n--- Multicast Delegate after removing Method1 ---
");
        multiDel("Only Method2 should be called now.");
    }
}
```

Input: No specific input required for this program.

Expected Output:

```
--- Single Delegate Invocation ---
Method1 called: Hello from delegate 1!
Method2 called: HELLO FROM DELEGATE 2!

--- Multicast Delegate Invocation ---
Method1 called: This message goes to both methods.
Method2 called: THIS MESSAGE GOES TO BOTH METHODS.

--- Multicast Delegate after removing Method1 ---
Method2 called: ONLY METHOD2 SHOULD BE CALLED NOW.
```

Lab 8: Exception Handling

Aim: To implement `try-catch-finally` blocks for robust error handling in C# applications.

Procedure:

1. Create a new C# Console Application.
2. Write a program that attempts a division by zero, which will cause a `DivideByZeroException`.
3. Enclose the potentially problematic code within a `try` block.
4. Add a `catch` block to specifically handle `DivideByZeroException` and print an informative error message.
5. Add a generic `catch` block to handle any other unexpected exceptions.
6. Include a `finally` block to demonstrate code that always executes, regardless of whether an exception occurred.
7. Test with valid input and input that causes an exception.
8. Compile and run the program.

Source Code:

```
using System;

public class Lab8
{
    public static void Main(string[] args)
    {
        int numerator = 10;
        int denominator;

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

        try
        {
            denominator = Convert.ToInt32(Console.ReadLine());

            int result = numerator / denominator;
            Console.WriteLine($"Result of division: {result}");
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine($"Error: Cannot divide by zero. {ex.Message}");
        }
        catch (FormatException ex)
        {
            Console.WriteLine($"Error: Invalid input. Please enter a valid
number. {ex.Message}");
        }
        catch (Exception ex) // Generic catch for any other exceptions
        {
            Console.WriteLine($"An unexpected error occurred: {ex.Message}");
        }
        finally
        {
            Console.WriteLine("This block always executes (finally block).");
        }

        Console.WriteLine("Program continues after exception handling.");
    }
}
```

Input:

Case 1 (Valid Input):

```
Enter a denominator: 2
```

Case 2 (Divide by Zero):

```
Enter a denominator: 0
```

Case 3 (Invalid Format):

```
Enter a denominator: abc
```

Expected Output:

Case 1 (Valid Input):

```
Enter a denominator: 2
Result of division: 5
This block always executes (finally block).
Program continues after exception handling.
```

Case 2 (Divide by Zero):

```
Enter a denominator: 0
Error: Cannot divide by zero. Attempted to divide by zero.
This block always executes (finally block).
Program continues after exception handling.
```

Case 3 (Invalid Format):

```
Enter a denominator: abc
Error: Invalid input. Please enter a valid number. Input string was not in a
correct format.
This block always executes (finally block).
Program continues after exception handling.
```

Lab 9: Custom Exception, Thread

Aim: To create and use custom exceptions and implement multithreading in C#.

Procedure:

1. Create a new C# Console Application.
2. Define a custom exception class that inherits from `Exception`.
3. Write a method that throws this custom exception under a specific condition.
4. Implement a `try-catch` block to handle the custom exception.
5. Create a method that simulates a long-running task.
6. Create a new `Thread` and execute the long-running task in it.
7. Observe the concurrent execution.
8. Compile and run the program.

Source Code:

```
using System;
using System.Threading;

// 2. Define a custom exception class
public class InvalidValueException : Exception
{
    public InvalidValueException() : base("Value is invalid.") { }
    public InvalidValueException(string message) : base(message) { }
    public InvalidValueException(string message, Exception innerException) :
base(message, innerException) { }
}

public class Lab9
{
    // Method that throws a custom exception
    public static void CheckValue(int value)
    {
        if (value < 0)
        {
            throw new InvalidValueException("Value cannot be negative.");
        }
        Console.WriteLine($"Value is valid: {value}");
    }

    // Method for thread execution
    public static void DoWork()
    {
        Console.WriteLine("Worker thread started.");
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine($"Worker thread: {i}");
            Thread.Sleep(500); // Simulate work
        }
        Console.WriteLine("Worker thread finished.");
    }

    public static void Main(string[] args)
    {
        // Custom Exception Handling
        Console.WriteLine("--- Custom Exception Demo ---");
        try
        {
```

```
            CheckValue(10);
            CheckValue(-5); // This will throw the custom exception
        }
        catch (InvalidValueException ex)
        {
            Console.WriteLine($"Caught Custom Exception: {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Caught Generic Exception: {ex.Message}");
        }

        Console.WriteLine("\n--- Threading Demo ---");
        // Create and start a new thread
        Thread workerThread = new Thread(DoWork);
        workerThread.Start();

        // Main thread continues execution
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine($"Main thread: {i}");
            Thread.Sleep(700); // Simulate work
        }

        Console.WriteLine("Main thread finished.");
        workerThread.Join(); // Wait for the worker thread to complete
        Console.WriteLine("Both threads have completed.");
    }
}
```

Input: No specific input required for this program.

Expected Output:

```
--- Custom Exception Demo ---
Value is valid: 10
Caught Custom Exception: Value cannot be negative.

--- Threading Demo ---
Worker thread started.
Main thread: 0
Worker thread: 0
Worker thread: 1
Main thread: 1
Worker thread: 2
Main thread: 2
Worker thread: 3
Worker thread: 4
Worker thread finished.
Main thread finished.
Both threads have completed.
```

(Note: The exact interleaving of "Worker thread" and "Main thread" output may vary slightly due to thread scheduling.)

Lab 10: Create Windows Applications

Aim: To develop basic desktop applications using Windows Forms or WPF in C#.

Procedure:

1. Open Visual Studio and create a new C# "Windows Forms App" or "WPF App" project.
2. Design a simple UI with a `Button` and a `Label` control.
3. Double-click the button to generate its `Click` event handler.
4. In the event handler, write code to change the text of the `Label` when the button is clicked.
5. Compile and run the application. Interact with the UI elements.

Source Code (Windows Forms Example):

```
// This code typically resides in Form1.cs [Design] and Form1.cs
// Form1.cs (Designer-generated code for controls - simplified)
/*
namespace WindowsApp
{
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.Label label1;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.label1 = new System.Windows.Forms.Label();
            this.SuspendLayout();
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(100, 50);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(75, 23);
            this.button1.Text = "Click Me!";
            this.button1.UseVisualStyleBackColor = true;
            this.button1.Click += new System.EventHandler(this.button1_Click);
            //
            // label1
            //
            this.label1.AutoSize = true;
            this.label1.Location = new System.Drawing.Point(100, 100);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(70, 13);
            this.label1.Text = "Hello World!";
            //
            // Form1
            //
            this.ClientSize = new System.Drawing.Size(284, 261);
```

```
            this.Controls.Add(this.label1);
            this.Controls.Add(this.button1);
            this.Name = "Form1";
            this.Text = "My First Windows App";
            this.ResumeLayout(false);
            this.PerformLayout();
        }
    }
}
*/

// Form1.cs (Your custom code)
using System;
using System.Windows.Forms;

namespace WindowsApp
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent(); // This method is generated by the designer
        }

        private void button1_Click(object sender, EventArgs e)
        {
            label1.Text = "Button Clicked!";
        }
    }
}

// Program.cs (Standard entry point)
/*
using System;
using System.Windows.Forms;

namespace WindowsApp
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
*/
```

Input: User clicks the "Click Me!" button.

Expected Output: Initially, the label displays "Hello World!". After clicking the button, the label text changes to "Button Clicked!".

Lab 11: Develop Web Applications using Validation and Navigation Controls

Aim: To develop web applications using ASP.NET Web Forms with validation and navigation controls.

Procedure:

1. Open Visual Studio and create a new C# "ASP.NET Web Application (.NET Framework)" project, choosing the "Empty" or "Web Forms" template.
2. Add a new Web Form (`.aspx` file) to your project.
3. Drag and drop a `TextBox` control, a `RequiredFieldValidator`, and a `Button` onto the form.
4. Configure the `RequiredFieldValidator` to validate the `TextBox`.
5. Add a `HyperLink` control for navigation to another page (create a second `.aspx` page if needed).
6. Compile and run the web application. Test the validation by submitting an empty text box and then with valid input. Test the navigation.

Source Code (Example for Default.aspx):

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebAppControls.Default" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Validation & Navigation</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <h2>User Registration</h2>
            <p>
                Name:
                <asp:TextBox ID="txtName" runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="rfvName" runat="server"
ControlToValidate="txtName"
                    ErrorMessage="Name is required!"
ForeColor="Red"></asp:RequiredValidator>
            </p>
            <p>
                <asp:Button ID="btnSubmit" runat="server" Text="Submit"
OnClick="btnSubmit_Click" />
            </p>
            <p>
                <asp:Label ID="lblMessage" runat="server" Text=""
ForeColor="Green"></asp:Label>
            </p>
            <hr />
            <h3>Navigation</h3>
            <p>
                Go to <asp:HyperLink ID="HyperLink1" runat="server"
NavigateUrl="~/About.aspx">About Us</asp:HyperLink>
            </p>
        </div>
    </form>
```

```
</body>
</html>
```

Source Code (Example for Default.aspx.cs):

```
using System;
using System.Web.UI;

namespace WebAppControls
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            // Optional: Any page load logic
        }

        protected void btnSubmit_Click(object sender, EventArgs e)
        {
            if (Page.IsValid) // Checks all validators on the page
            {
                lblMessage.Text = $"Hello, {txtName.Text}! Form submitted
successfully.";
            }
            else
            {
                lblMessage.Text = "Please correct the errors.";
            }
        }
    }
}
```

Input:

1. Leave the Name textbox empty and click "Submit".
2. Enter a name (e.g., "John Doe") in the textbox and click "Submit".
3. Click the "About Us" hyperlink.

Expected Output:

1. "Name is required!" message appears next to the textbox.
2. "Hello, John Doe! Form submitted successfully." message appears.
3. The browser navigates to `About.aspx` (assuming you created this page).

Lab 12: Develop Web Applications using Data Controls

Aim: To develop web applications using ASP.NET data controls (e.g., GridView, DetailsView) to display and manipulate data.

Procedure:

1. Create a new ASP.NET Web Forms project.
2. Add a new Web Form.
3. Add a simple data source (e.g., an `ArrayList` or `List<T>` in the code-behind for simplicity, or connect to a database if available).
4. Drag and drop a `GridView` control onto the form.
5. Bind the `GridView` to your data source.
6. Configure the `GridView` to display the data.
7. Compile and run the application.

Source Code (Example for Default.aspx):

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebAppDataControls.Default" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Data Controls</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <h2>Product List</h2>
            <asp:GridView ID="gvProducts" runat="server"
AutoGenerateColumns="true"
                EmptyDataText="No products to display.">
            </asp:GridView>
        </div>
    </form>
</body>
</html>
```

Source Code (Example for Default.aspx.cs):

```
using System;
using System.Collections.Generic;
using System.Web.UI;

namespace WebAppDataControls
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                // Create a simple list of objects to act as data source
                List<Product> products = new List<Product>
                {
```

```
                    new Product { Id = 1, Name = "Laptop", Price = 1200.00 },
                    new Product { Id = 2, Name = "Mouse", Price = 25.50 },
                    new Product { Id = 3, Name = "Keyboard", Price = 75.00 }
                };

                // Bind the GridView to the data source
                gvProducts.DataSource = products;
                gvProducts.DataBind();
            }
        }
    }

    // Simple Product class for demonstration
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public double Price { get; set; }
    }
}
```

Input: No direct user input.

Expected Output: A table (rendered by `GridView`) displaying the product data:

Id	Name	Price
1	Laptop	1200
2	Mouse	25.5
3	Keyboard	75

Lab 13: Develop Web Applications Using Object Model

Aim: To develop web applications by interacting with the ASP.NET object model (e.g., `Request`, `Response`, `Session`, `Application` objects).

Procedure:

1. Create a new ASP.NET Web Forms project.
2. Add a new Web Form.
3. In the `Page_Load` event, use `Request.QueryString` to read values from the URL.
4. Use `Session` state to store and retrieve user-specific data across multiple page requests.
5. Use `Response.Redirect` to navigate to another page.
6. Display information from these objects on the page.
7. Compile and run the application. Test by passing query string parameters and observing session data.

Source Code (Example for Default.aspx):

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebAppObjectModel.Default" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Object Model Demo</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <h2>ASP.NET Object Model Demo</h2>
            <p>
                Query String Value (Name): <asp:Label ID="lblQueryString"
runat="server"></asp:Label>
            </p>
            <p>
                Session Value (User ID): <asp:Label ID="lblSession"
runat="server"></asp:Label>
            </p>
            <p>
                <asp:Button ID="btnSetSession" runat="server" Text="Set Session
& Redirect" OnClick="btnSetSession_Click" />
            </p>
        </div>
    </form>
</body>
</html>
```

Source Code (Example for Default.aspx.cs):

```
using System;
using System.Web.UI;

namespace WebAppObjectModel
{
    public partial class Default : System.Web.UI.Page
    {
```

```
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                // Reading from Request.QueryString
                string name = Request.QueryString["name"];
                if (!string.IsNullOrEmpty(name))
                {
                    lblQueryString.Text = name;
                }
                else
                {
                    lblQueryString.Text = "No name in query string.";
                }

                // Reading from Session
                if (Session["UserID"] != null)
                {
                    lblSession.Text = Session["UserID"].ToString();
                }
                else
                {
                    lblSession.Text = "Session not set.";
                }
            }
        }

        protected void btnSetSession_Click(object sender, EventArgs e)
        {
            // Setting Session value
            Session["UserID"] = Guid.NewGuid().ToString(); // Example: set a
unique ID

            // Redirecting to the same page with a query string for
demonstration
            Response.Redirect($"Default.aspx?name=DemoUser");
        }
    }
}
```

Input:

1. Access the page: `Default.aspx`
2. Access the page with a query string: `Default.aspx?name=Alice`
3. Click the "Set Session & Redirect" button.

Expected Output:

1. `Query String Value (Name): No name in query string. Session Value (User ID): Session not set.`
2. `Query String Value (Name): Alice Session Value (User ID): Session not set.`
3. After clicking the button, the page reloads, and you'll see: `Query String Value (Name): DemoUser Session Value (User ID): [a new GUID]`

Lab 14: Develop Web Application Using Data Source Control

Aim: To develop web applications using ASP.NET data source controls (e.g., `SqlDataSource`, `ObjectDataSource`) for simplified data access.

Procedure:

1. Create a new ASP.NET Web Forms project.
2. Add a new Web Form.
3. Add a database (e.g., a simple SQL Server Express `.mdf` file or use a connection string to an existing database). Create a sample table (e.g., `Products`).
4. Drag and drop a `SqlDataSource` control onto the form.
5. Configure the `SqlDataSource` to connect to your database and select data from your table.
6. Drag and drop a `GridView` control and set its `DataSourceID` property to the ID of your `SqlDataSource`.
7. Compile and run the application.

Source Code (Example for Default.aspx):

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebAppDataSrc.Default" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Data Source Control</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <h2>Products from Database</h2>
            <asp:SqlDataSource ID="SqlDataSource1" runat="server"
                ConnectionString="<%$
ConnectionStrings:MyDatabaseConnectionString %>"
                SelectCommand="SELECT [ProductId], [ProductName], [Price] FROM
[Products]">
            </asp:SqlDataSource>

            <asp:GridView ID="GridView1" runat="server"
AutoGenerateColumns="true"
                DataSourceID="SqlDataSource1" AllowPaging="true" PageSize="5">
            </asp:GridView>
        </div>
    </form>
</body>
</html>
```

Source Code (Example for Web.config - Connection String):

```
<?xml version="1.0"?>
<configuration>
  <connectionStrings>
    <add name="MyDatabaseConnectionString" connectionString="Data
Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\MyDatabase.mdf;In
tegrated Security=True"
      providerName="System.Data.SqlClient" />
```

```
  </connectionStrings>
  <system.web>
    <compilation debug="true" targetFramework="4.7.2" />
    <httpRuntime targetFramework="4.7.2" />
  </system.web>
</configuration>
```

(Note: You would need to create a `MyDatabase.mdf` file in your `App_Data` folder with a `Products` table containing `ProductId`, `ProductName`, `Price` columns.)

Input: No direct user input.

Expected Output: A `GridView` displaying data retrieved from the `Products` table in your database.

Lab 15: Develop Web Application Using Form View and Repeater Control

Aim: To develop web applications using ASP.NET `FormView` and `Repeater` controls for flexible data presentation.

Procedure:

1. Create a new ASP.NET Web Forms project.
2. Add a new Web Form.
3. Create a simple data source (e.g., a `List<T>` in the code-behind).
4. Drag and drop a `Repeater` control onto the form. Define its `ItemTemplate` to customize how each data item is displayed.
5. Drag and drop a `FormView` control. Configure its templates (`ItemTemplate`, `EditItemTemplate`, `InsertItemTemplate`) to display, edit, and insert single records.
6. Bind both controls to your data source.
7. Compile and run the application. Observe the flexible rendering of data.

Source Code (Example for Default.aspx):

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebAppFormRepeater.Default" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>FormView & Repeater</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <h2>Products (using Repeater)</h2>
            <asp:Repeater ID="rptProducts" runat="server">
                <HeaderTemplate>
                    <table border="1" cellpadding="5" cellspacing="0">
                        <tr>
                            <th>ID</th>
                            <th>Name</th>
                            <th>Price</th>
                        </tr>
                </HeaderTemplate>
                <ItemTemplate>
                    <tr>
                        <td><%# Eval("Id") %></td>
                        <td><%# Eval("Name") %></td>
                        <td><%# Eval("Price", "{0:C}") %></td> <%-- Format as
currency --%>
                    </tr>
                </ItemTemplate>
                <FooterTemplate>
                    </table>
                </FooterTemplate>
                <SeparatorTemplate>
                    </SeparatorTemplate>
            </asp:Repeater>

            <hr />
```

```
            <h2>Selected Product Details (using FormView)</h2>
            <asp:FormView ID="fvProduct" runat="server" DataKeyNames="Id"
                EmptyDataText="No product selected.">
                <ItemTemplate>
                    <div>
                        <b>Product ID:</b> <%# Eval("Id") %><br />
                        <b>Product Name:</b> <%# Eval("Name") %><br />
                        <b>Price:</b> <%# Eval("Price", "{0:C}") %><br />
                        <asp:Button ID="EditButton" runat="server"
CommandName="Edit" Text="Edit" />
                    </div>
                </ItemTemplate>
                <EditItemTemplate>
                    <div>
                        <b>Product ID:</b> <%# Eval("Id") %><br />
                        <b>Product Name:</b>
                        <asp:TextBox ID="txtNameEdit" runat="server" Text='<%#
Bind("Name") %>'></asp:TextBox><br />
                        <b>Price:</b>
                        <asp:TextBox ID="txtPriceEdit" runat="server" Text='<%#
Bind("Price") %>'></asp:TextBox><br />
                        <asp:Button ID="UpdateButton" runat="server"
CommandName="Update" Text="Update" />
                        <asp:Button ID="CancelButton" runat="server"
CommandName="Cancel" Text="Cancel" />
                    </div>
                </EditItemTemplate>
                <InsertItemTemplate>
                    </InsertItemTemplate>
            </asp:FormView>

            <p>
                <asp:Button ID="btnSelectFirst" runat="server" Text="Select
First Product" OnClick="btnSelectFirst_Click" />
            </p>
        </div>
    </form>
</body>
</html>
```

Source Code (Example for Default.aspx.cs):

```
using System;
using System.Collections.Generic;
using System.Linq; // For .FirstOrDefault()
using System.Web.UI;
using System.Web.UI.WebControls; // For FormViewMode

namespace WebAppFormRepeater
{
    public partial class Default : System.Web.UI.Page
    {
        // Simple Product class (reused from Lab 12)
        public class Product
        {
            public int Id { get; set; }
            public string Name { get; set; }
            public double Price { get; set; }
        }

        // Sample data source
        private List<Product> products = new List<Product>
        {
            new Product { Id = 101, Name = "Monitor", Price = 300.00 },
```

```
            new Product { Id = 102, Name = "Webcam", Price = 50.00 },
            new Product { Id = 103, Name = "Headphones", Price = 150.00 }
        };

        protected void Page_Load(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                BindData();
            }
        }

        private void BindData()
        {
            rptProducts.DataSource = products;
            rptProducts.DataBind();

            // Initially, bind FormView to the first product (or leave empty)
            fvProduct.DataSource = products.Take(1); // Select first item
            fvProduct.DataBind();
        }

        protected void btnSelectFirst_Click(object sender, EventArgs e)
        {
            // Select the first product in the FormView
            fvProduct.DataSource = products.Take(1);
            fvProduct.DataBind();
            fvProduct.ChangeMode(FormViewMode.ReadOnly); // Ensure it's in read-
only mode
        }

        protected void fvProduct_ModeChanging(object sender,
FormViewModeEventArgs e)
        {
            fvProduct.ChangeMode(e.NewMode);
            BindData(); // Rebind to reflect mode change
        }

        protected void fvProduct_ItemUpdating(object sender,
FormViewUpdateEventArgs e)
        {
            // Get the ID of the product being updated
            int productId = (int)fvProduct.DataKey.Value;

            // Find the product in the list
            Product productToUpdate = products.FirstOrDefault(p => p.Id ==
productId);

            if (productToUpdate != null)
            {
                // Get updated values from the textboxes in the EditItemTemplate
                productToUpdate.Name =
((TextBox)fvProduct.FindControl("txtNameEdit")).Text;
                productToUpdate.Price =
Convert.ToDouble(((TextBox)fvProduct.FindControl("txtPriceEdit")).Text);
            }

            fvProduct.ChangeMode(FormViewMode.ReadOnly); // Switch back to read-
only mode
            BindData(); // Rebind to show updated data
        }
    }
}
```

Input:

1. Load the page.
2. Click "Select First Product" button.
3. Click "Edit" button in FormView.
4. Change values in textboxes and click "Update".

Expected Output:

1. A table displaying products from the `Repeater`.
2. The `FormView` displays details of the first product.
3. After clicking "Edit", the `FormView` switches to edit mode with textboxes.
4. After clicking "Update", the `FormView` returns to read-only mode, and the updated values are reflected in both the `FormView` and `Repeater` (if the data source is updated in code).