ASSESSMENT 4

Burra Saharsh

23110071

—

Software Tools and Techniques For CSE

—

Professor Shouvick Mondal

TABLE OF CONTENTS

Laboratory Session 11 03
- Introduction 03
- Tools 03
- Setup 03
- Methodology and Execution..... 04
- Results and Analysis 15
- Discussion and Conclusion 16
- References 16

Laboratory Session 12 17
- Introduction 17
- Tools 17
- Setup 17
- Methodology and Execution..... 18
- Results and Analysis 30
- Discussion and Conclusion 30
- References 30

LABORATORY SESSION 11

INTRODUCTION

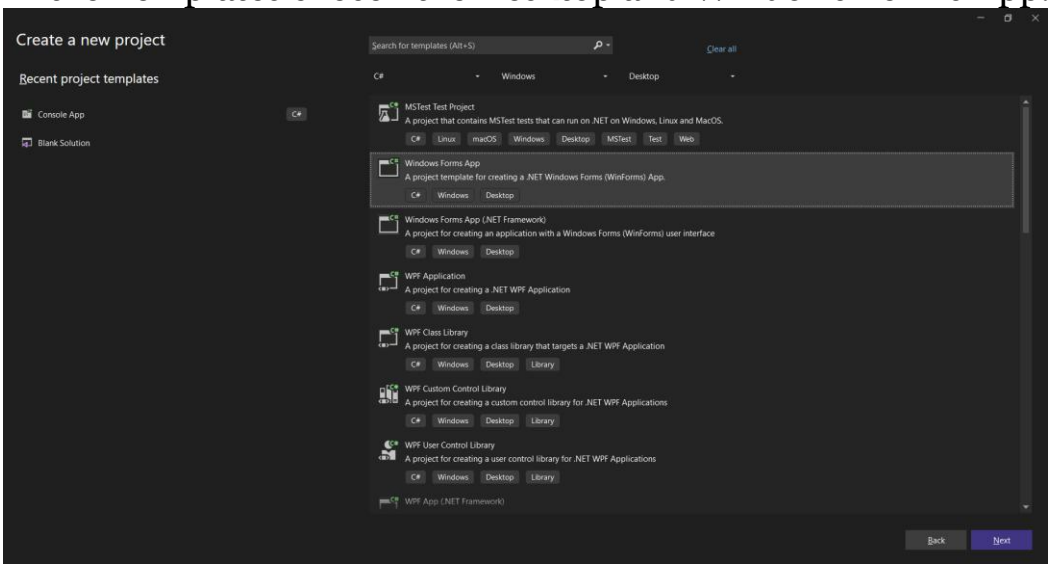
In this lab session we will understand the in-depth concepts of event handling and delegates in C# Windows Forms Applications. We will also learn how to create, subscribe, and invoke custom events to achieve modular, interactive, and reusable GUI designs using Visual Studio. We will learn how to implement and handle custom events in C# Windows Forms applications using the publisher–subscriber model, design and manage interactive forms, apply multicast event handling, use custom EventArgs classes.

TOOLS

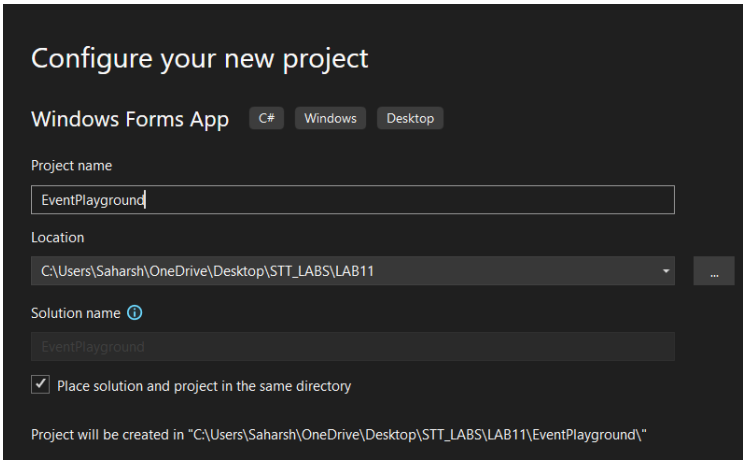
Operating Systems – Windows
Software - Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK
Programming Language: C#
Framework - .NET 8.0

SETUP

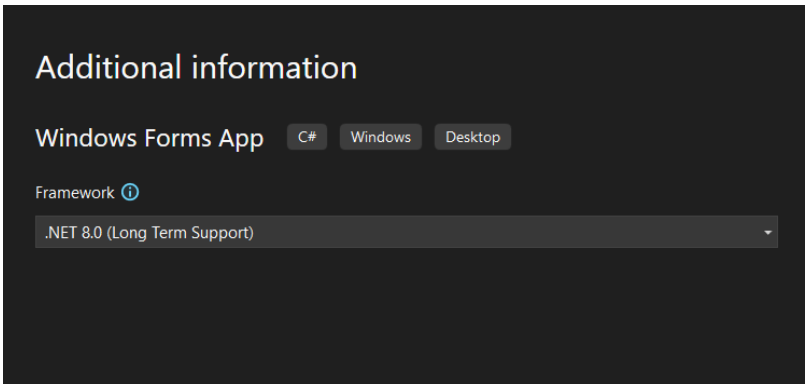
In the Templates chosen the Desktop and Windows Forms App.



Saved this new one into the LAB11 Folder.



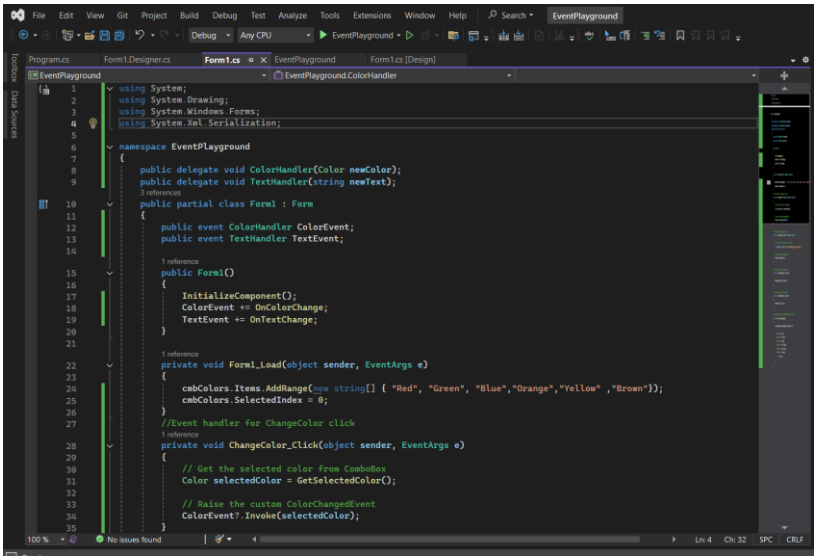
Framework is .NET 8.0



METHODOLOGY AND EXECUTION

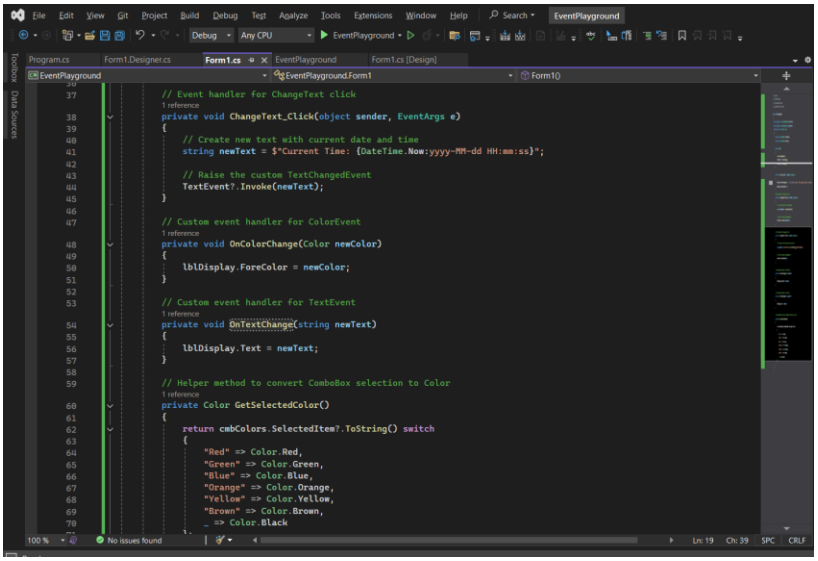
Windows Forms App – Multi-Control Event Interaction:

Form1.cs



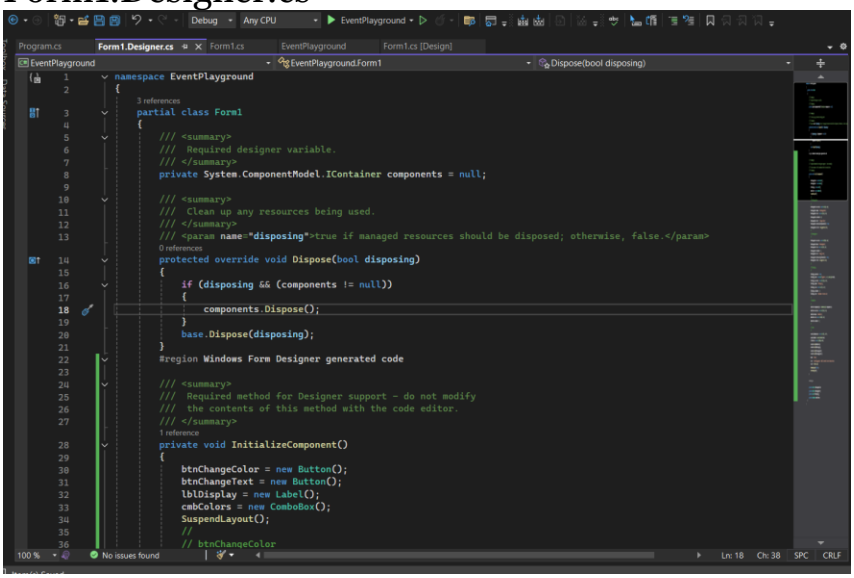
```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Xml.Serialization;

namespace EventPlayground
{
    public delegate void ColorHandler(Color newColor);
    public delegate void TextHandler(string newText);
    3 references
    public partial class Form1 : Form
    {
        public event ColorHandler ColorEvent;
        public event TextHandler TextEvent;

        1 reference
        public Form1()
        {
            InitializeComponent();
            ColorEvent += OnColorChange;
            TextEvent += OnTextChange;
        }

        1 reference
        private void Form1_Load(object sender, EventArgs e)
        {
            cmbColors.Items.AddRange(new string[] { "Red", "Green", "Blue","Orange","Yellow" ,"Brown"});
            cmbColors.SelectedIndex = 0;
        }
        //Event handler for ChangeColor click
        1 reference
        private void ChangeColor_Click(object sender, EventArgs e)
        {
            // Get the selected color from ComboBox
            Color selectedColor = GetSelectedColor();

            // Raise the custom ColorChangedEvent
            ColorEvent?.Invoke(selectedColor);
        }
```

Defined the two delegate declarations of color and text handlers, also initialized the event handlers, implemented code for subscribing to their events on their respective handlers.
Added the code for Changing the text displayed to current time.



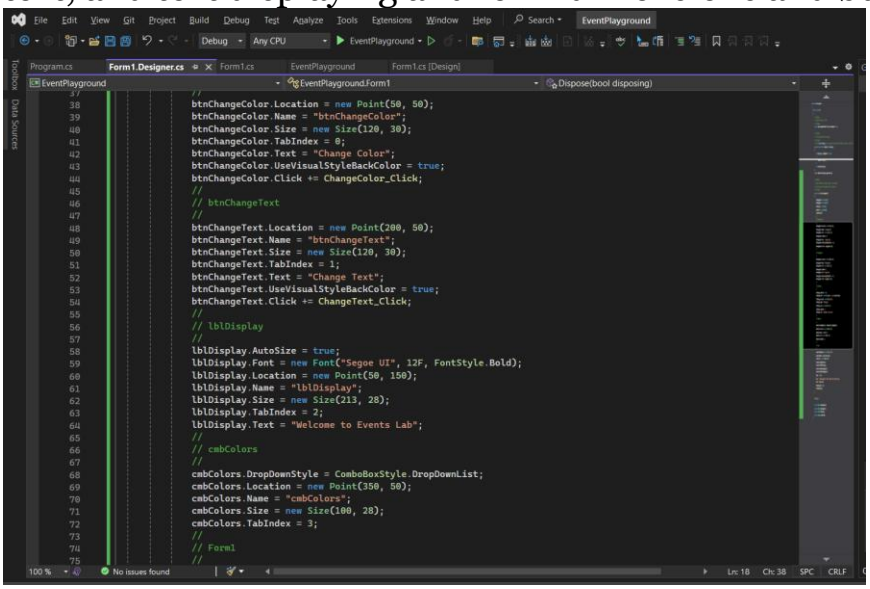
```
// Event handler for ChangeText click
1 reference
private void ChangeText_Click(object sender, EventArgs e)
{
    // Create new text with current date and time
    string newText = $"Current Time: {DateTime.Now:yyyy-MM-dd HH:mm:ss}";

    // Raise the custom TextChangedEvent
    TextEvent?.Invoke(newText);
}

// Custom event handler for ColorEvent
1 reference
private void OnColorChange(Color newColor)
{
    lblDisplay.ForeColor = newColor;
}

// Custom event handler for TextEvent
1 reference
private void OnTextChange(string newText)
{
    lblDisplay.Text = newText;
}

// Helper method to convert ComboBox selection to Color
1 reference
private Color GetSelectedColor()
{
    return cmbColors.SelectedItem?.ToString() switch
    {
        "Red" => Color.Red,
        "Green" => Color.Green,
        "Blue" => Color.Blue,
        "Orange" => Color.Orange,
        "Yellow" => Color.Yellow,
        "Brown" => Color.Brown,
        _ => Color.Black
```

Gave 6 options for the colors and implemented GetSelectedColor function along with event handler for changing text on click and also custom text and color handlers.
Form1.Designer.cs



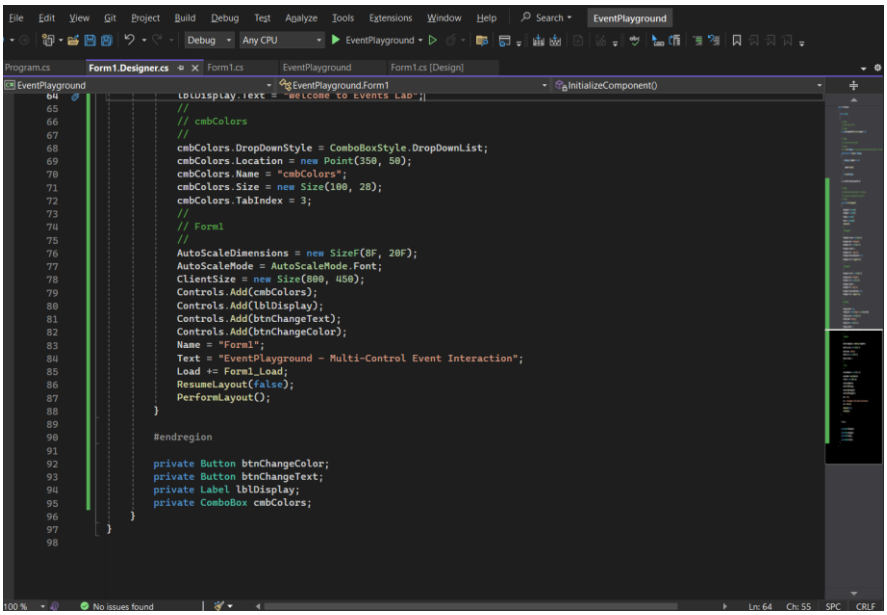
```
namespace EventPlayground
{
    3 references
    partial class Form1
    {
        /// <summary>
        ///  Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        ///  Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        0 references
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
        #region Windows Form Designer generated code

        /// <summary>
        ///  Required method for Designer support - do not modify
        ///  the contents of this method with the code editor.
        /// </summary>
        1 reference
        private void InitializeComponent()
        {
            btnChangeColor = new Button();
            btnChangeText = new Button();
            lblDisplay = new Label();
            cmbColors = new ComboBox();
            SuspendLayout();
            //
            // btnChangeColor
```

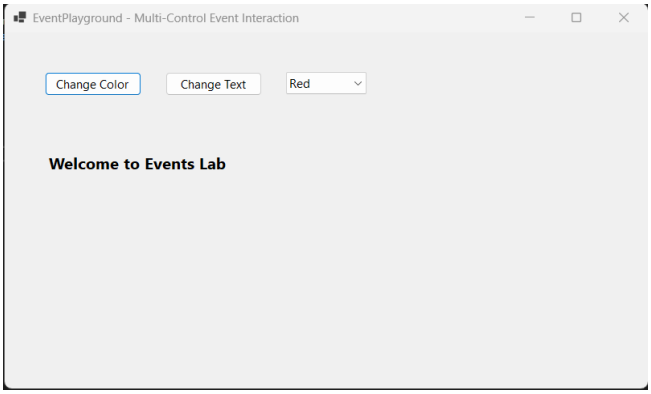
Implemented the Initialize component function, using the button for color, button for text, and text displaying and form dimensions and button dimensions.

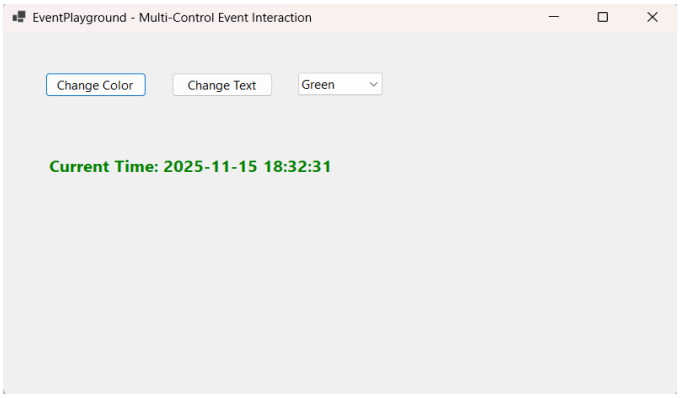


```
//
btnChangeColor.Location = new Point(50, 50);
btnChangeColor.Name = "btnChangeColor";
btnChangeColor.Size = new Size(120, 30);
btnChangeColor.TabIndex = 0;
btnChangeColor.Text = "Change Color";
btnChangeColor.UseVisualStyleBackColor = true;
btnChangeColor.Click += ChangeColor_Click;
//
// btnChangeText
//
btnChangeText.Location = new Point(200, 50);
btnChangeText.Name = "btnChangeText";
btnChangeText.Size = new Size(120, 30);
btnChangeText.TabIndex = 1;
btnChangeText.Text = "Change Text";
btnChangeText.UseVisualStyleBackColor = true;
btnChangeText.Click += ChangeText_Click;
//
// lblDisplay
//
lblDisplay.AutoSize = true;
lblDisplay.Font = new Font("Segoe UI", 12F, FontStyle.Bold);
lblDisplay.Location = new Point(50, 150);
lblDisplay.Name = "lblDisplay";
lblDisplay.Size = new Size(213, 28);
lblDisplay.TabIndex = 2;
lblDisplay.Text = "Welcome to Events Lab";
//
// cmbColors
//
cmbColors.DropDownStyle = ComboBoxStyle.DropDownList;
cmbColors.Location = new Point(350, 50);
cmbColors.Name = "cmbColors";
cmbColors.Size = new Size(100, 28);
cmbColors.TabIndex = 3;
//
// Form1
//
```

Here in the below I have initialized the four variables.



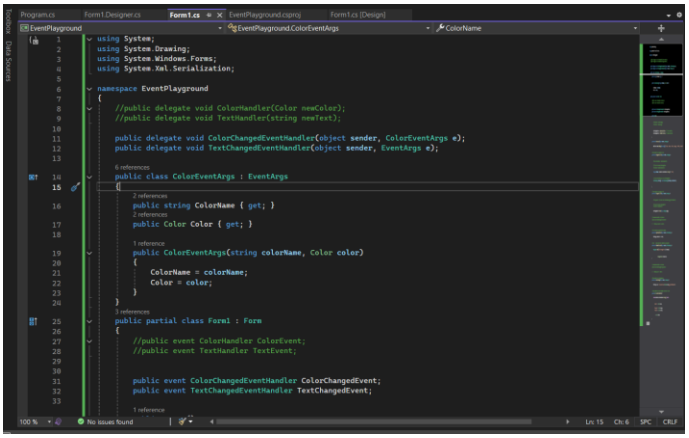




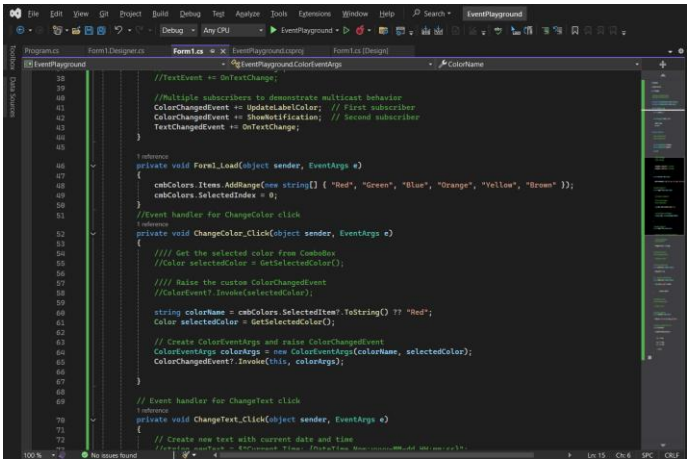
These are the results for the first task.

Using EventArgs and Multiple Subscribers:

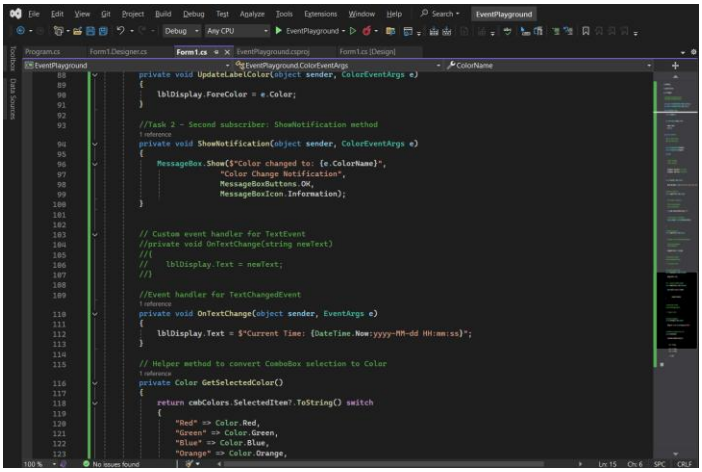
Form1.cs



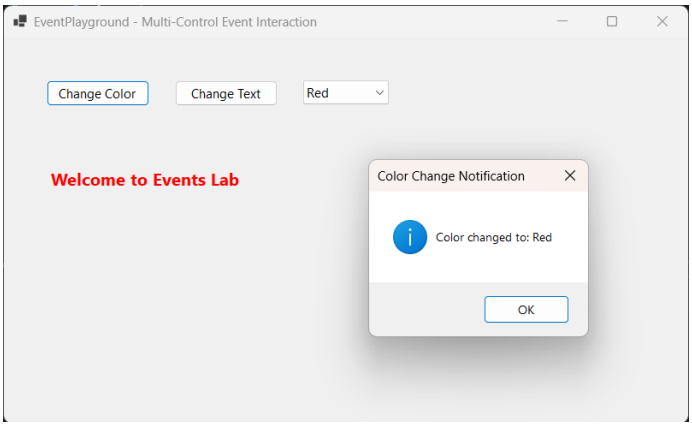
Updated the code for the delegate to handle multiple subscribers.

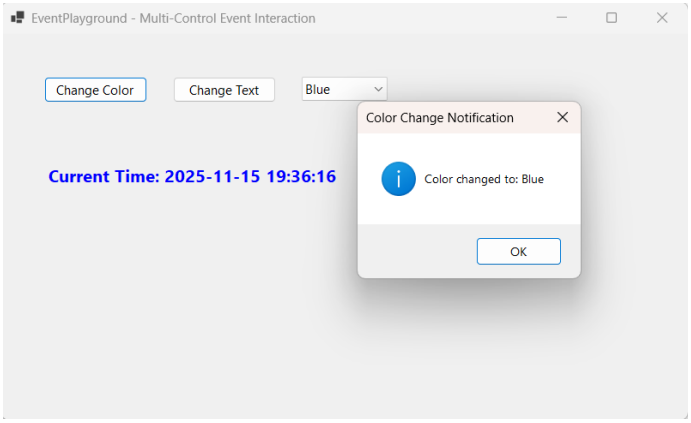


Implemented the code for showing the notification on color change with the help of ColorEventArgs class.



Updated the code to handle multiple methods (multicast behaviour).



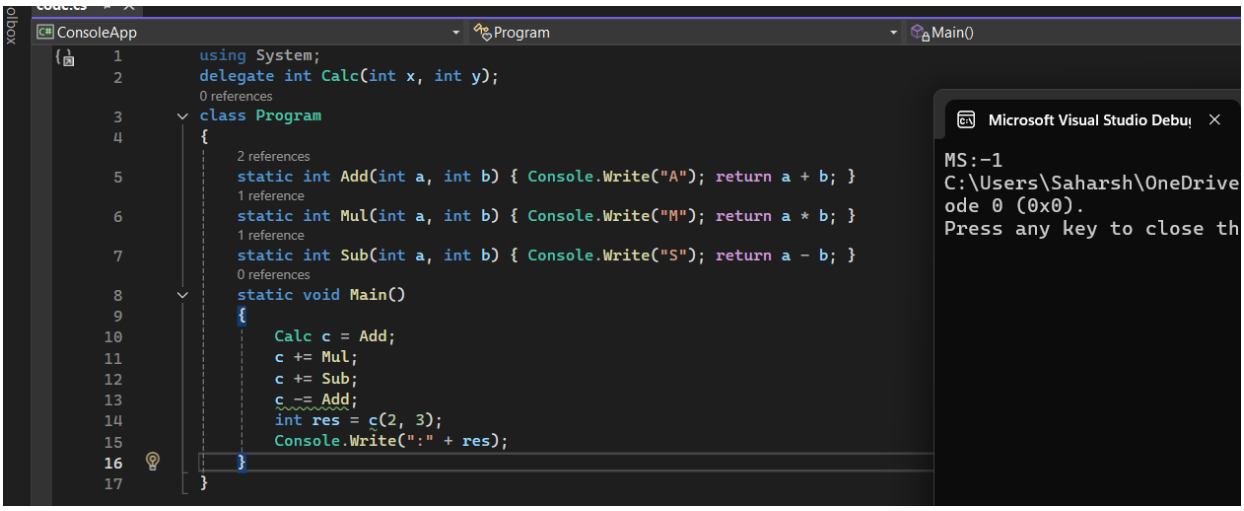


Output Reasoning (Level 0)
Code1:

```
using System;


delegate int Calc(int x, int y);


class Program
{
    static int Add(int a, int b) { Console.Write("A"); return a + b; }
    static int Mul(int a, int b) { Console.Write("M"); return a * b; }
    static int Sub(int a, int b) { Console.Write("S"); return a - b; }


    static void Main()
    {
        Calc c = Add;
        c += Mul;
        c += Sub;
        c -= Add;
        int res = c(2, 3);
        Console.Write(":" + res);
    }
}
```

Output and why?

When the delegate c is called at res = c(2,3), as the handler went through the flow of subscribing to add, then multiply, then subtract and finally unsubscribing the add function, so it only subscribed to multiply and subtract which it executes one by one, so when multiply executes it prints string M then returns 6(2*3), then executes the Subtract and prints S and returns -1(2-3), so the handler will return the last executed functions result to res, hence we get MS:-1 as the output.

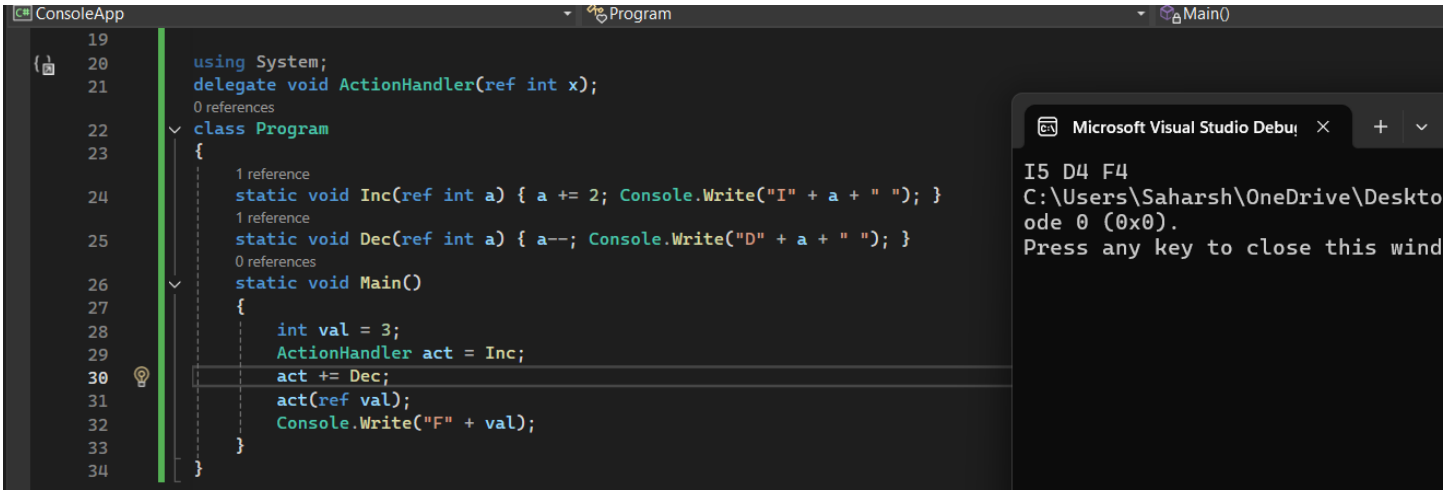


Code 2:

```
using System;

delegate void ActionHandler(ref int x);

class Program
{
    static void Inc(ref int a) { a += 2; Console.Write("I" + a + " "); }
    static void Dec(ref int a) { a--; Console.Write("D" + a + " "); }

    static void Main()
    {
        int val = 3;
        ActionHandler act = Inc;
        act += Dec;
        act(ref val);
        Console.Write("F" + val);
    }
}
```

Output and why?

So here the handler i.e. ActionHandler will first subscribe to Inc then to the Dec function and then for arguments to these functions we use the val variable's reference and passes to the both functions which essentially acts as a pointer by reference(so here the val value also changes due to passing by reference) and not value, so first Inc will execute resulting in value of a to be 5 since(a+=2, and a =3 by reference), then it prints I+a, i.e. I5, then calls Dec where it gets decremented by 1, so prints D+a, D4 and finally checks the val variable value in main function which is 4 as it gets modified.



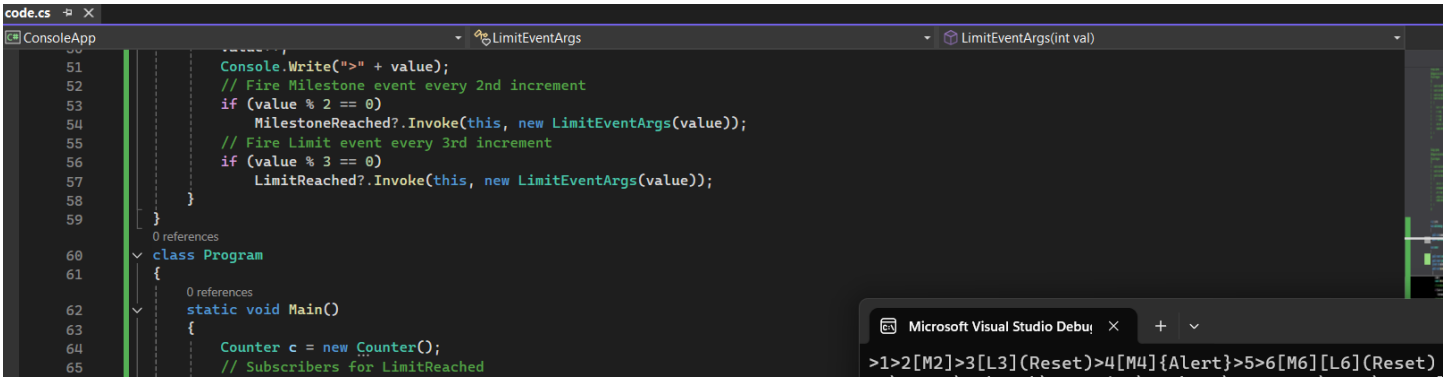
Output Reasoning (Level 1)
Code1:

```
using System;

class LimitEventArgs : EventArgs
{
    public int CurrentValue { get; }
    public LimitEventArgs(int val) => CurrentValue = val;
}

class Counter
{
    public event EventHandler<LimitEventArgs> LimitReached;
    public event EventHandler<LimitEventArgs> MilestoneReached;

    private int value = 0;

    public void Increment()
    {
        value++;
        Console.Write(">" + value);

        // Fire Milestone event every 2nd increment
        if (value % 2 == 0)
            MilestoneReached?.Invoke(this, new LimitEventArgs(value));

        // Fire Limit event every 3rd increment
        if (value % 3 == 0)
            LimitReached?.Invoke(this, new LimitEventArgs(value));
    }
}

class Program
{
    static void Main()
    {
        Counter c = new Counter();

        // Subscribers for LimitReached
        c.LimitReached += (s, e) => Console.Write("[L" + e.CurrentValue + "]");
        c.LimitReached += (s, e) => Console.Write("(Reset)");

        // Subscribers for MilestoneReached
        c.MilestoneReached += (s, e) =>
        {
            Console.Write("[M" + e.CurrentValue + "]");
            if (e.CurrentValue == 4)
                Console.Write("{Alert}");
        };

        for (int i = 0; i < 6; i++)
            c.Increment();
    }
}
```

Output and Why?

When the program starts, it creates a Counter object and subscribes several event handlers to LimitReached and MilestoneReached. The for loop calls Increment() six times. Across the six iterations, the counter increments from 1 to 6, and during each increment the program prints the new value and conditionally triggers events. The event handlers use (s, e) where s is just the sender (the Counter object triggering the event) and e is the LimitEventArgs carrying the current value . At value 1, no event fires. At 2, the value is even, so the MilestoneReached event fires and prints [M2]. At 3,

the value is divisible by 3, so the LimitReached event fires, printing [L3] and (Reset) from its two subscribers. At 4, the value is even again, so MilestoneReached prints [M4] and, because it equals 4, also prints {Alert}. At 5, no conditions are met and only >5 is printed. At 6, both conditions are true: it is even, so MilestoneReached prints [M6], and it is divisible by 3, so LimitReached prints [L6] and (Reset). This sequence completes the control flow for all six iterations.



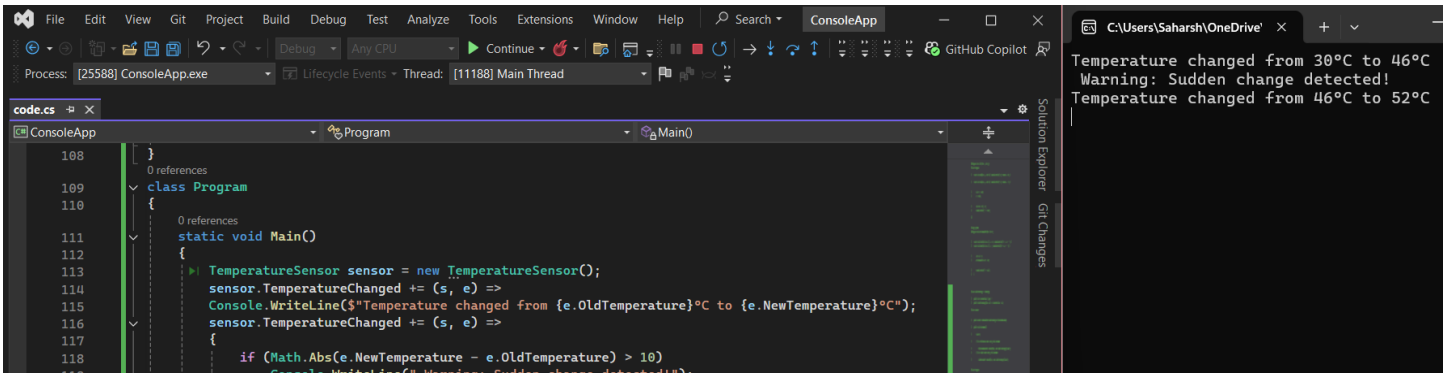
Code 2:

```
using System;

class TemperatureEventArgs : EventArgs
{
    public int OldTemperature { get; }
    public int NewTemperature { get; }

    public TemperatureEventArgs(int oldTemp, int newTemp)
    {
        OldTemperature = oldTemp;
        NewTemperature = newTemp;
    }
}

class TemperatureSensor
{
    public event EventHandler<TemperatureEventArgs> TemperatureChanged;

    private int temperature = 25;

    public void UpdateTemperature(int newTemp)
    {
        int oldTemp = temperature;
        temperature = newTemp;
```

```
        if (Math.Abs(newTemp - oldTemp) > 5)
        {
            TemperatureChanged?.Invoke(this, new TemperatureEventArgs(oldTemp, newTemp));
        }
    }
}

class Program
{
    static void Main()
    {
        TemperatureSensor sensor = new TemperatureSensor();

        sensor.TemperatureChanged += (s, e) =>
            Console.WriteLine($"Temperature changed from {e.OldTemperature}°C to {e.NewTemperature}°C");

        sensor.TemperatureChanged += (s, e) =>
        {
            if (Math.Abs(e.NewTemperature - e.OldTemperature) > 10)
                Console.WriteLine(" Warning: Sudden change detected!");
        };

        sensor.UpdateTemperature(28);
        sensor.UpdateTemperature(30);
        sensor.UpdateTemperature(46);
        sensor.UpdateTemperature(52);
    }
}
```

Output and Why?

When UpdateTemperature() is called each time, the sensor stores the previous temperature and updates to the new one, and only if the absolute difference is more than 5 it fires the TemperatureChanged event. The handlers use (s, e) where s is the sensor object and e carries both the old and new temperatures. On the first call 28, old is 25 so the difference is 3, which is not greater than 5, so no event fires. On the second call 30, old is 28 so the difference is 2, still no event. On the third call 46, old is 30 and the difference becomes 16, so the event triggers and both subscribed handlers run in the order they were added: first handler prints “Temperature changed from 30°C to 46°C”, then the second handler checks if the difference > 10, which is true, so it prints “Warning: Sudden change detected!”. On the final call 52, old is 46 and the difference is 6, which again triggers the event, the first handler prints the temperature change, and the second handler sees difference is 6 (not > 10) so it prints nothing.



Output Reasoning(Level 2)

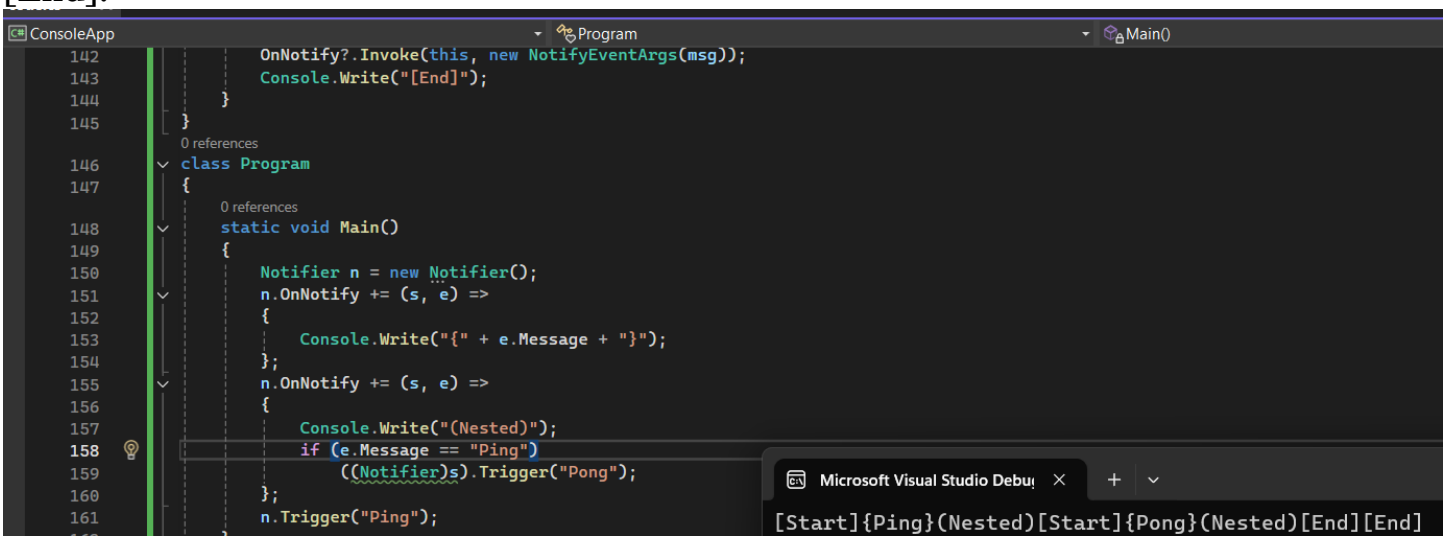
Code1:

```
using System;

class NotifyEventArgs : EventArgs
{
    public string Message { get; }
    public NotifyEventArgs(string msg) => Message = msg;
}

class Notifier
{
    public event EventHandler<NotifyEventArgs> OnNotify;

    public void Trigger(string msg)
    {
        Console.Write("[Start]");
        OnNotify?.Invoke(this, new NotifyEventArgs(msg));
        Console.Write("[End]");
    }
}

class Program
{
    static void Main()
    {
        Notifier n = new Notifier();

        n.OnNotify += (s, e) =>
        {
            Console.Write("{" + e.Message + "}");
        };

        n.OnNotify += (s, e) =>
        {
            Console.Write("(Nested)");
            if (e.Message == "Ping")
                ((Notifier)s).Trigger("Pong");
        };

        n.Trigger("Ping");
    }
}
```

Output and Why?

When Trigger("Ping") is called, it first prints [Start] and then fires the OnNotify event with (s, e) where s is the Notifier object and e.Message holds "Ping". The handlers execute in the order they were subscribed to, so the first handler prints {Ping}. Then the second handler prints (Nested) and since the message is "Ping", it calls Trigger("Pong") on the same object using ((Notifier)s). This starts a completely new Trigger call, again printing [Start], then both handlers run with the new message "Pong": the first prints {Pong}, the second prints (Nested) but this time the condition fails so no further nested trigger happens. That nested trigger then prints its [End],

returns control to the original handler, and finally the very first Trigger call prints its [End].

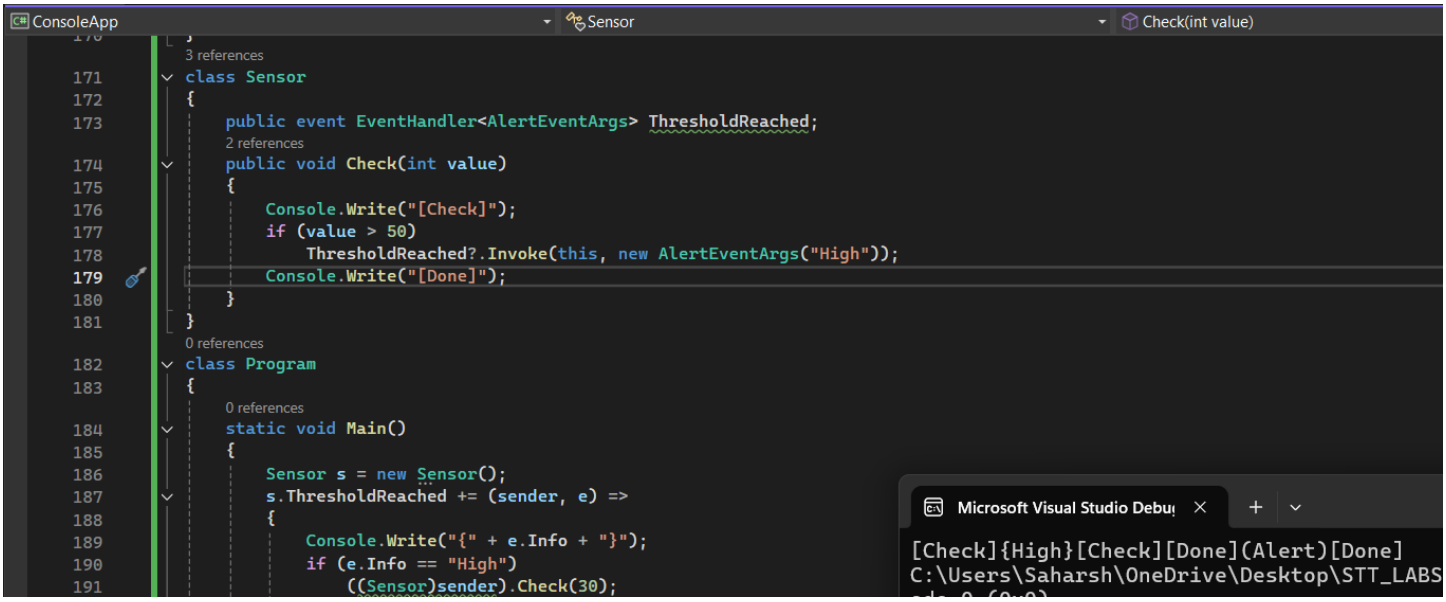


Code2:

```
using System;

class AlertEventArgs : EventArgs
{
    public string Info { get; }
    public AlertEventArgs(string info) => Info = info;
}

class Sensor
{
    public event EventHandler<AlertEventArgs> ThresholdReached;

    public void Check(int value)
    {
        Console.Write("[Check]");
        if (value > 50)
            ThresholdReached?.Invoke(this, new AlertEventArgs("High"));
        Console.Write("[Done]");
    }
}

class Program
{
    static void Main()
    {
        Sensor s = new Sensor();

        s.ThresholdReached += (sender, e) =>
        {
            Console.Write("{" + e.Info + "}");
            if (e.Info == "High")
                ((Sensor)sender).Check(30);
        };

        s.ThresholdReached += (sender, e) =>
            Console.Write("(Alert)");

        s.Check(80);
    }
}
```

Output and Why?

When s.Check(80) runs, it first prints [Check], and since 80 is greater than 50 the ThresholdReached event fires with (sender, e) where sender is the same sensor object and e.Info is "High". The handlers run in the order they were subscribed, so the first handler prints {High} and because the info is "High", it immediately calls a nested Check(30) on the same sensor using ((Sensor)sender). That nested call again prints [Check], but since 30 is not greater than 50 the event does not fire, so it directly prints [Done] and returns. After coming back, the second handler of the original event executes and prints (Alert). Finally, after both handlers finish, the original Check(80) prints its [Done].



RESULTS AND ANALYSIS

A consistent pattern of event behavior can be traced across all the programs - from the console-based examples to the full Windows Forms application. In the Forms app, the button clicks were able to raise our custom events: one changed the label's color based on the "ComboBox" selection, and the other updated the label text with the current date and time. By introducing the custom "ColorEventArgs" class, it became possible to pass the additional information, such as the selected color, to each subscriber, thus making it absolutely clear how the data is flowing through custom events. Besides that, it illustrated multicast behavior, i.e., both "UpdateLabelColor" and "ShowNotification" were executed in the sequence for the same event. The console programs were a perfect support for this statement as they showed events firing only when the conditions were met, handlers executing in the exact order they were subscribed to, and even cases where an event caused another event inside a handler, thus creating a nested and layered flow of execution.

DISCUSSION AND CONCLUSION

These examples made it very clear how event-driven programming lets different parts of an application independently respond to the same user action, thereby maintaining the entire system as clean and modular. Our work on the Windows Forms app with custom delegates of our own—without using the default button click events—helped us get a more vivid, practical and intuitive grasp of the way the code behind the scenes for events is actually happening: how events are declared, subscribed to, and invoked.

The console applications brought the same points home again, in particular, when dealing with the difficult concepts such as events causing other events (re-entrancy), events being fired conditionally and, having multiple subscribers reacting to the same event. There were some issues in particular around making sure that the signatures of the events matched the definitions of our custom delegate and, also, in following the execution paths where an event could trigger itself again.

REFERENCES

[1] LAB11

[2] Lec12

[3] Event-driven_programming

[4] GitHubLink

LABORATORY SESSION 12

INTRODUCTION

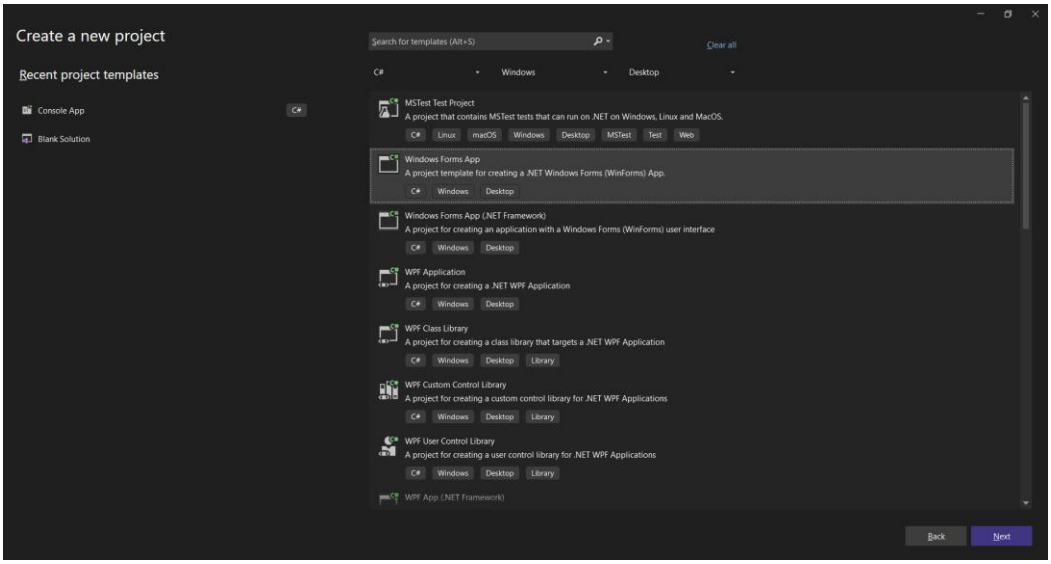
This lab session we will enhance our knowledge of advanced event-driven mechanisms in C# Windows Forms Applications. We will design modular GUIs that demonstrate even chaining, filtered event invocation, and contextual data sharing through custom EventArgs. We will learn how to use the before said classes to exchange contextual data between publishers and subscribers. We also get to understand event chaining where one event dynamically triggers another. We will also learn how to apply conditional event firing and multicast subscription in GUI contexts.

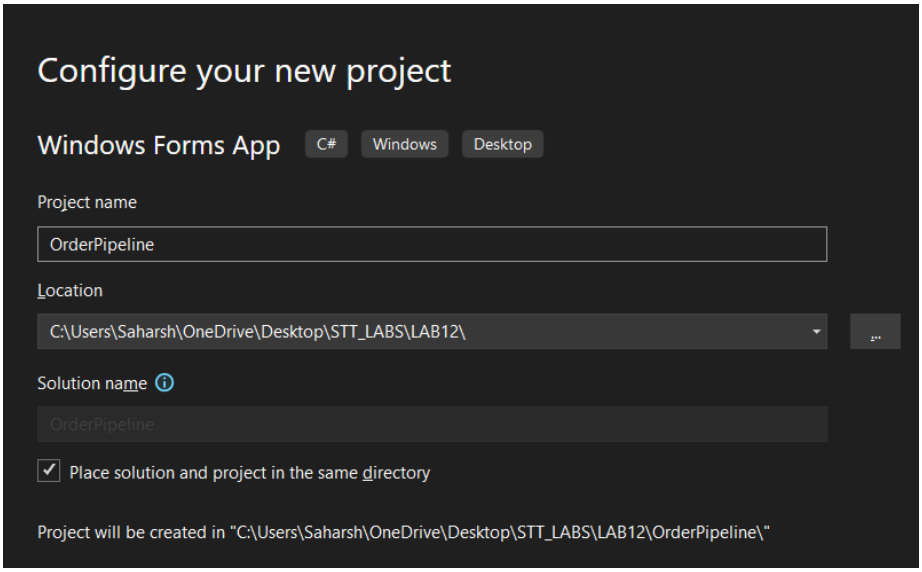
TOOLS

Operating Systems – Windows
Software - Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK
Programming Language: C#
Framework - .NET 8.0

SETUP

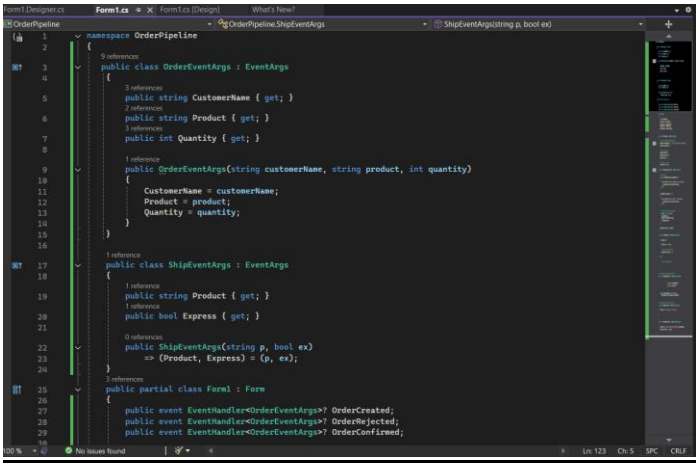
Created a new project in the STT_Lab12 folder.





METHODOLOGY AND EXECUTION

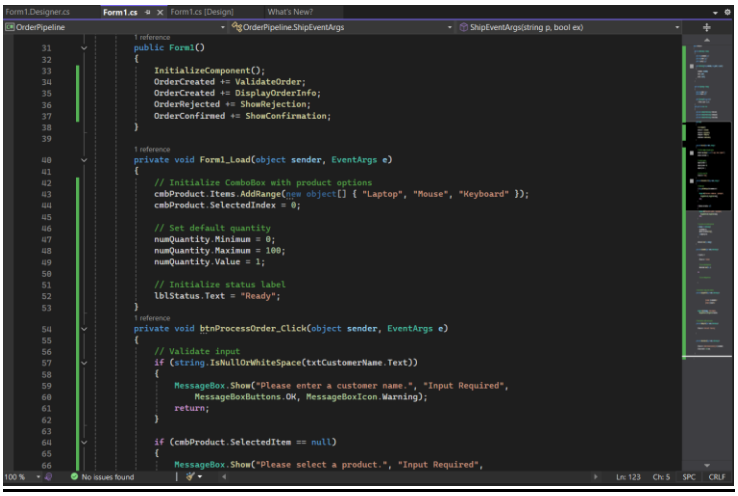
Windows Forms App – Multi-Stage Event Chaining with Custom EventArgs:



```
namespace OrderPipeline
{
    public class OrderEventArgs : EventArgs
    {
        public string CustomerName { get; }
        public string Product { get; }
        public int Quantity { get; }

        public OrderEventArgs(string customerName, string product, int quantity)
        {
            CustomerName = customerName;
            Product = product;
            Quantity = quantity;
        }
    }

    public class ShipEventArgs : EventArgs
    {
        public string Product { get; }
        public bool Express { get; }

        public ShipEventArgs(string p, bool ex)
            => (Product, Express) = (p, ex);
    }
    public partial class Form1 : Form
    {
        public event EventHandler<OrderEventArgs>? OrderCreated;
        public event EventHandler<OrderEventArgs>? OrderRejected;
        public event EventHandler<OrderEventArgs>? OrderConfirmed;
```

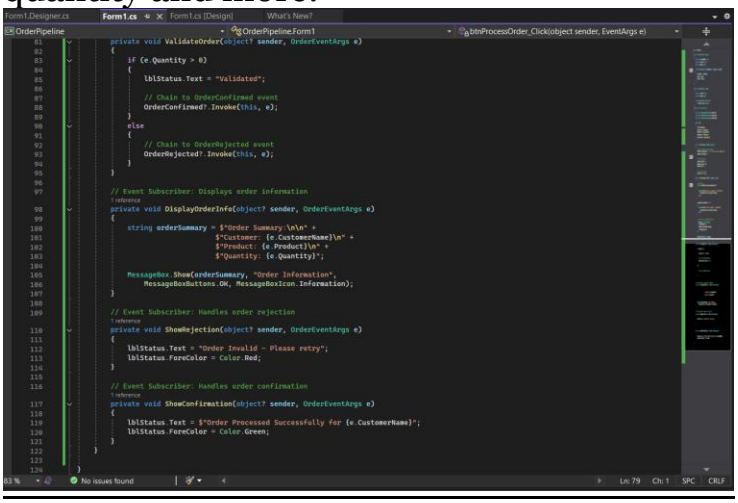
Created a public class which takes care of Ordering Event such as name, product and quantity and initializes these variables. Similarly made the same with Shipping Event.

Defined the event handlers in the Form1 class for order creation, rejection and confirmation. Initialized the Form1 class with subscribing to their respective functions like for object creation, need to check for validation, displaying on success or rejection, and finally confirmation.



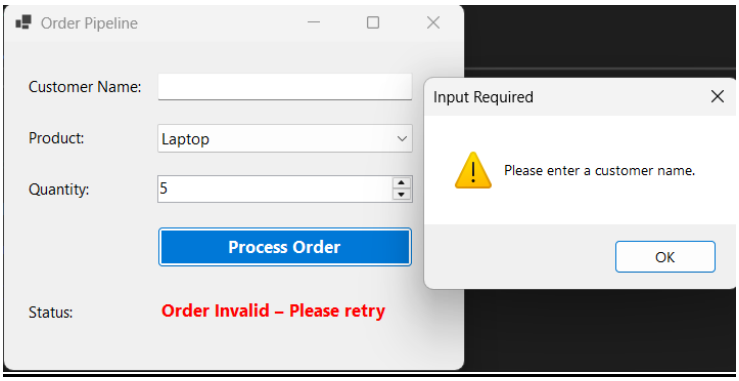
```
public Form1()
{
    InitializeComponent();
    OrderCreated += ValidateOrder;
    OrderCreated += DisplayOrderInfo;
    OrderRejected += ShowRejection;
    OrderConfirmed += ShowConfirmation;
}

private void Form1_Load(object sender, EventArgs e)
{
    // Initialize ComboBox with product options
    cmbProduct.Items.AddRange(new object[] { "Laptop", "Mouse", "Keyboard" });
    cmbProduct.SelectedIndex = 0;

    // Set default quantity
    numQuantity.Minimum = 0;
    numQuantity.Maximum = 100;
    numQuantity.Value = 1;

    // Initialize status label
    lblStatus.Text = "Ready";
}
private void btnProcessOrder_Click(object sender, EventArgs e)
{
    // Validate input
    if (string.IsNullOrWhiteSpace(txtCustomerName.Text))
    {
        MessageBox.Show("Please enter a customer name.", "Input Required",
            MessageBoxButtons.OK, MessageBoxIcon.Warning);
        return;
    }

    if (cmbProduct.SelectedItem == null)
    {
        MessageBox.Show("Please select a product.", "Input Required",
```

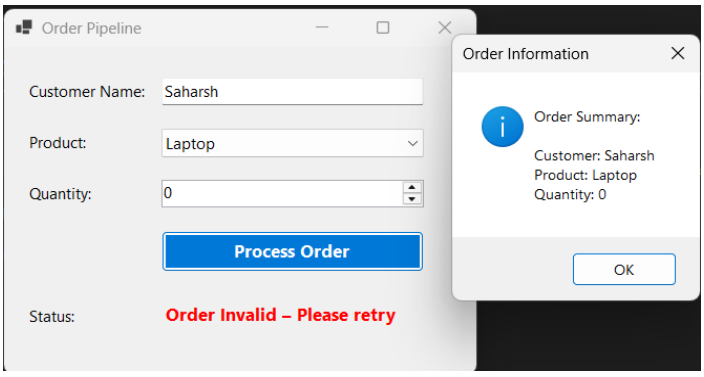
Form1 load contains the information to load like the products Laptop, Mouse and Keyboard and also providing their ranges. Also implemented the above said functions like validation, checking for the correct ranges and not empty name and non zero quantity and more.



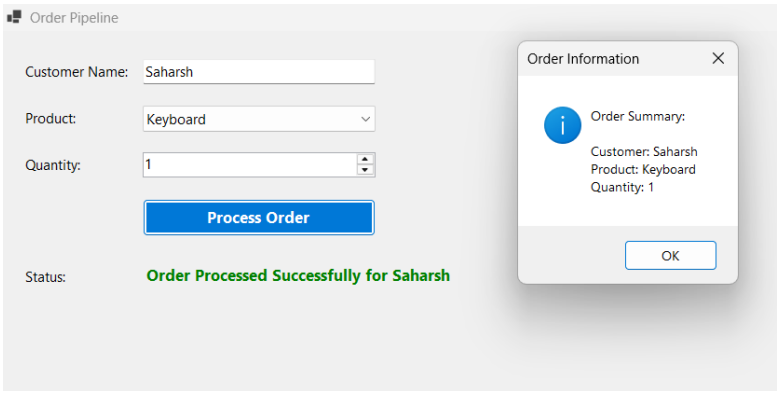
```
private void ValidateOrder(object? sender, OrderEventArgs e)
{
    if (e.Quantity > 0)
    {
        lblStatus.Text = "Validated";

        // Chain to OrderConfirmed event
        OrderConfirmed?.Invoke(this, e);
    }
    else
    {
        // Chain to OrderRejected event
        OrderRejected?.Invoke(this, e);
    }
}

// Event Subscriber: Displays order information
private void DisplayOrderInfo(object? sender, OrderEventArgs e)
{
    string orderSummary = $"Order Summary:\n\n" +
                          $"Customer: {e.CustomerName}\n" +
                          $"Product: {e.Product}\n" +
                          $"Quantity: {e.Quantity}";

    MessageBox.Show(orderSummary, "Order Information",
        MessageBoxButtons.OK, MessageBoxIcon.Information);
}

// Event Subscriber: Handles order rejection
private void ShowRejection(object? sender, OrderEventArgs e)
{
    lblStatus.Text = "Order Invalid – Please retry";
    lblStatus.ForeColor = Color.Red;
}

// Event Subscriber: Handles order confirmation
private void ShowConfirmation(object? sender, OrderEventArgs e)
{
    lblStatus.Text = $"Order Processed Successfully for {e.CustomerName}";
    lblStatus.ForeColor = Color.Green;
}
```

The last functions implemented to show the order made, rejection or acceptance/confirmation.



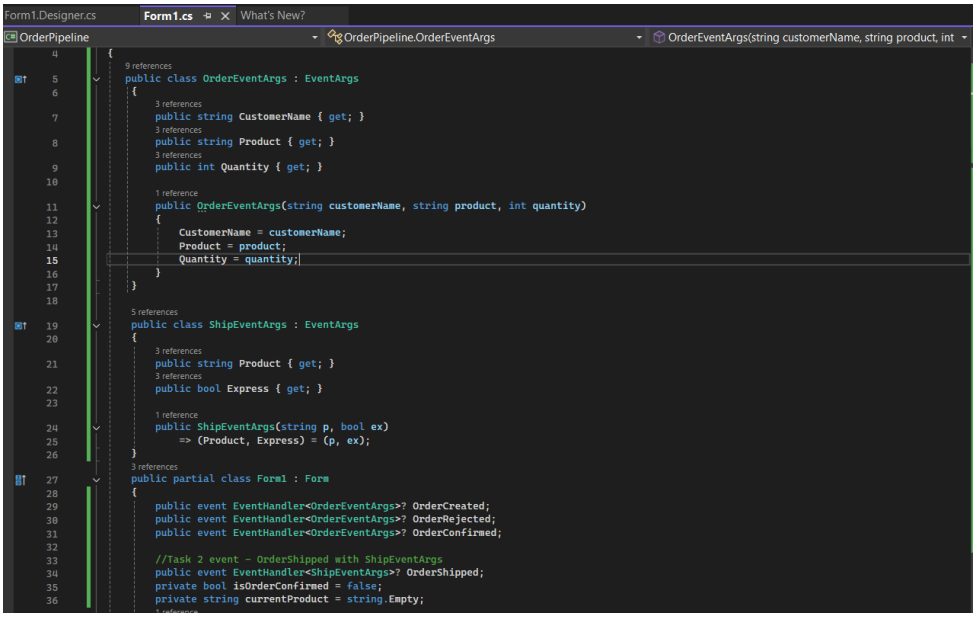


We can see here that when name field is empty it throws an invalid message and also it says order invalid , and also says the order is invalid for zero quantity with the red color “Order Invalid – Please retry”.



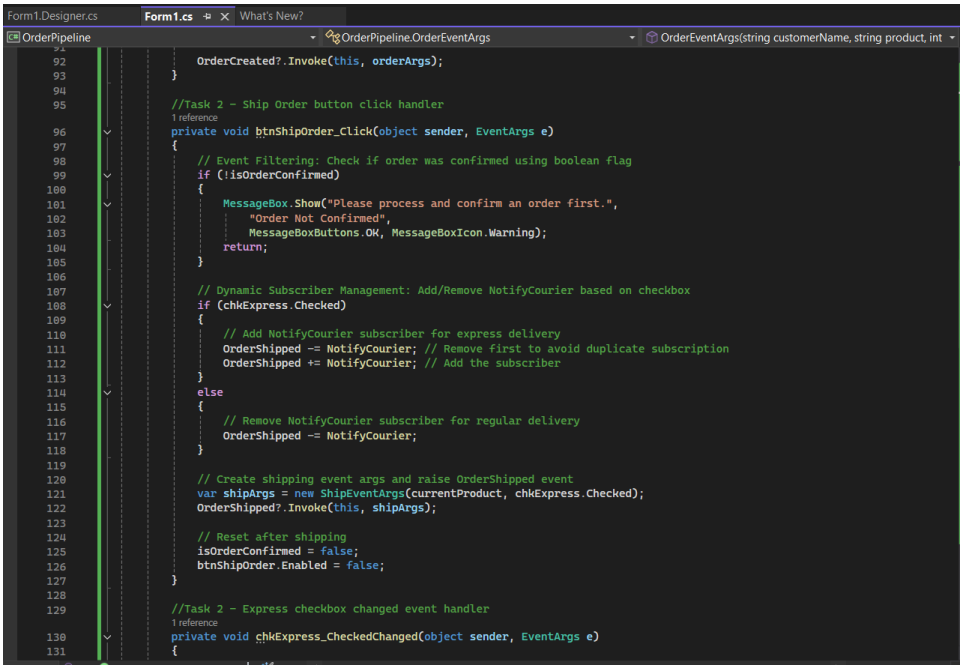
It shows in green color that “Order processed Successfully for User” when an order is valid that order is placed.

2. Event Filtering and Dynamic Subscriber Management:

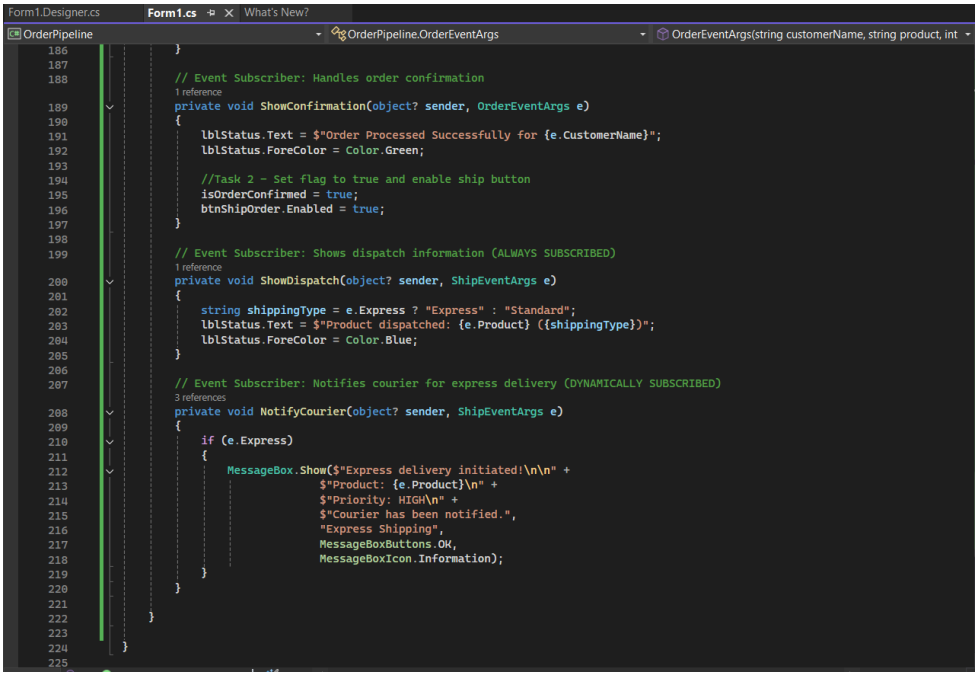


So most of the code still remains the same and few of them get added those are event handler for Order Shipping. And also I have initialized two other variables one is the order confirmation and other is the current products name stored as a string.

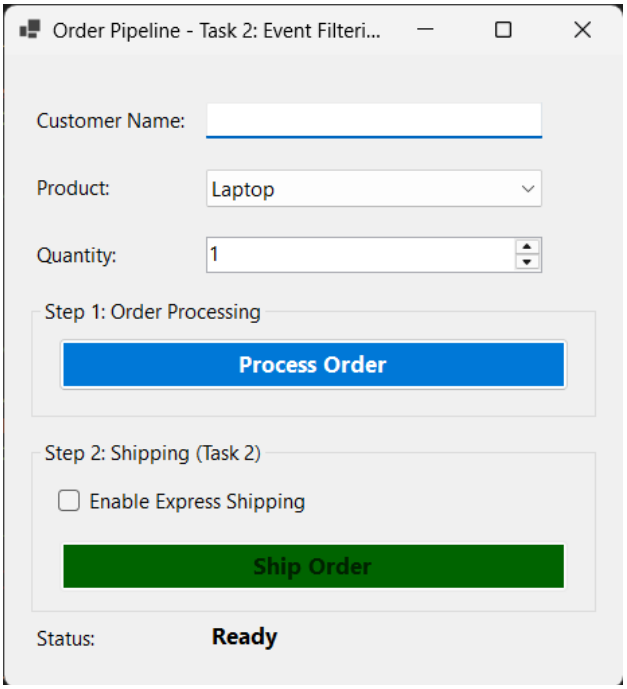
These variables are used to check for the next step of shipping only when the above Boolean is true then only we go to next stage of either express or standard way of shipping.

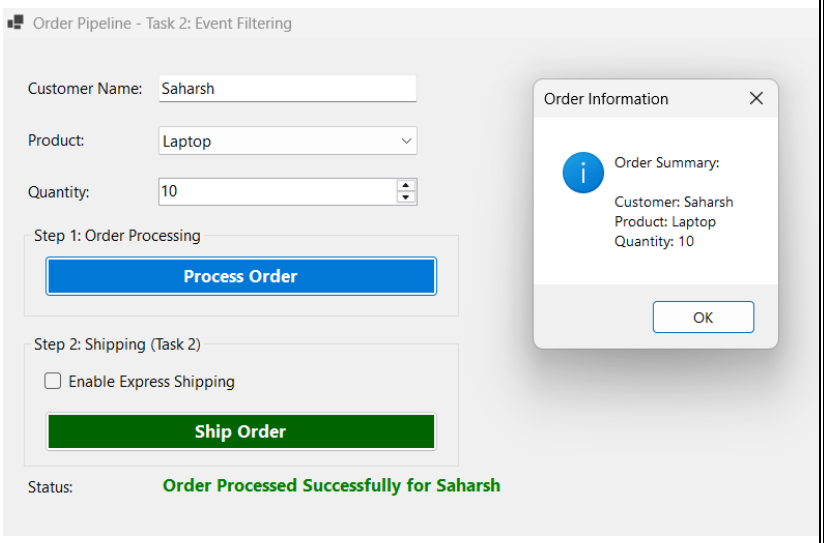


Also in the confirmation function we set this variable to true. And after the shipping this is reset to false.

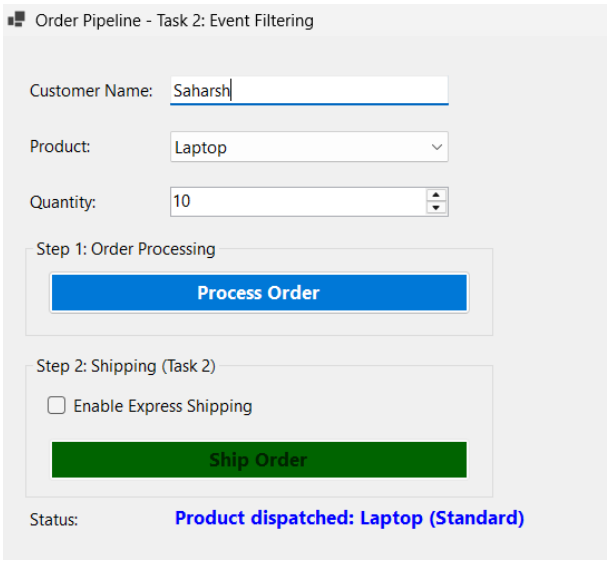


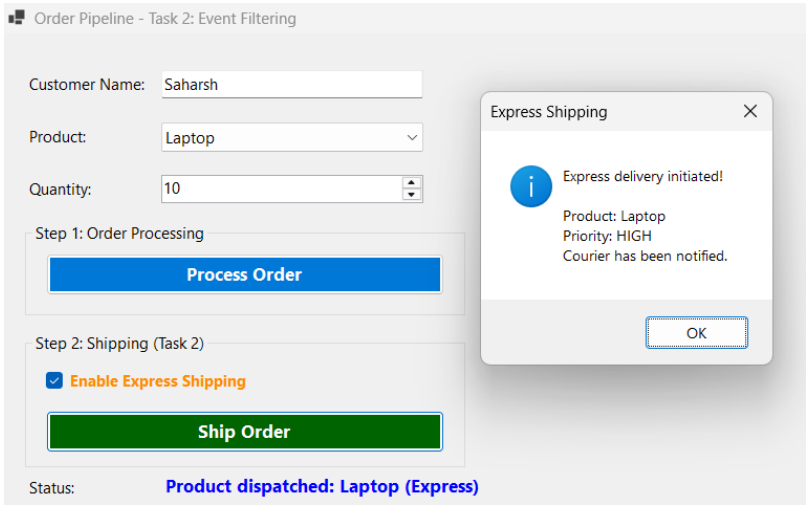
We can see the below images for correct functionality of Order Processing is verified.



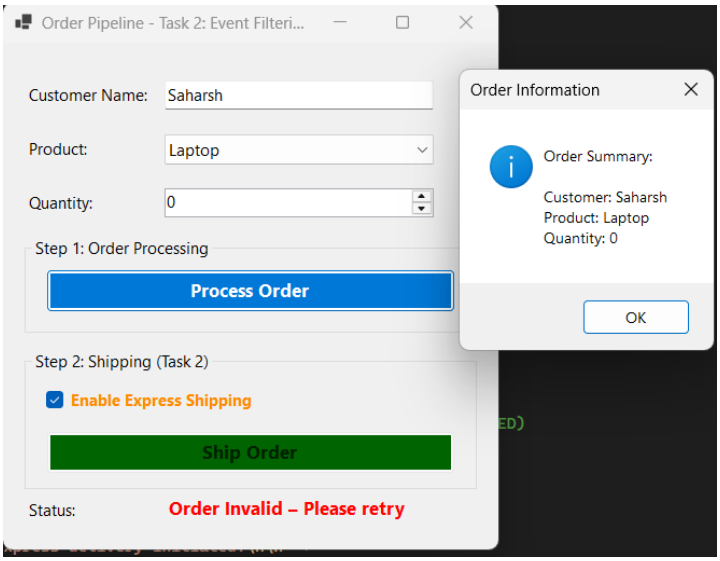


These two images show the correct functionality of the shipping with and without the express shipping.





Finally also checked if zero quantity order is getting the needed order invalid message which is correctly getting executed.



Output Reasoning (Level 0)

Code1:

```
public delegate void AuthCallback(bool validUser);
public static AuthCallback loginCallback = Login;
public static void Login()
{
    Console.WriteLine("Valid user!");
}


public static void Main(string[] args)
{
    loginCallback(true);
}
```

Output and Why?

Upon starting, the program sees Main trying to execute the delegate call loginCallback(true), as if to make a call to a method passing a bool parameter. The delegate type AuthCallback is the definition for wrapping any method with the signature void MethodName(bool), so the flow of control should be Main → loginCallback → Login(bool). Nevertheless, the trouble is happening even before the shot can be fired: loginCallback is equated to Login, but Login doesn't take any parameters, thus it's incompatible with the delegate's signature. Due to this discrepancy, the compiler is unable to link the delegate with the method and thus halts the entire flow; the program does not get to execution stage. The exception is caused by C# demanding the exact same signature for a delegate and the method assigned to it.

Code2:

```
using System;


delegate void Notify(string msg);
class Program
{
    static void Main()
    {
        Notify handler = null;


        handler += (m) => Console.WriteLine("A: " + m);
        handler += (m) => Console.WriteLine("B: " + m.ToUpper());


        handler("hello");


        handler -= (m) => Console.WriteLine("A: " + m);
        handler("world");
    }
}
```

Output and Why?

At the start of the program, a multicast delegate handler is created by adding two anonymous functions: the first prints "A: " + the string and the second prints "B: <string in uppercase >". So when handler("hello") is called, both functions are executed in order, printing A: hello and then B: HELLO. The code then tries to remove the first handler by using handler -= (m) => Console.WriteLine("A: " + m);, but this operation generates a new anonymous lambda rather than the one that was previously added (Each time we write a lambda expression like above C# creates a new delegate instance in memory. Even if two lambda expressions look identical in code, they are still different objects. Delegates support removal (-=) only when the instance we try to remove is exactly the same object reference as the one originally added). Delegates can only remove an invocation if the reference matches the one that was originally added. Since anonymous lambdas are different objects, the removal goes silently without any effect. Hence, when handler("world") is called, both original functions are still executed, printing A: world and B: WORLD.

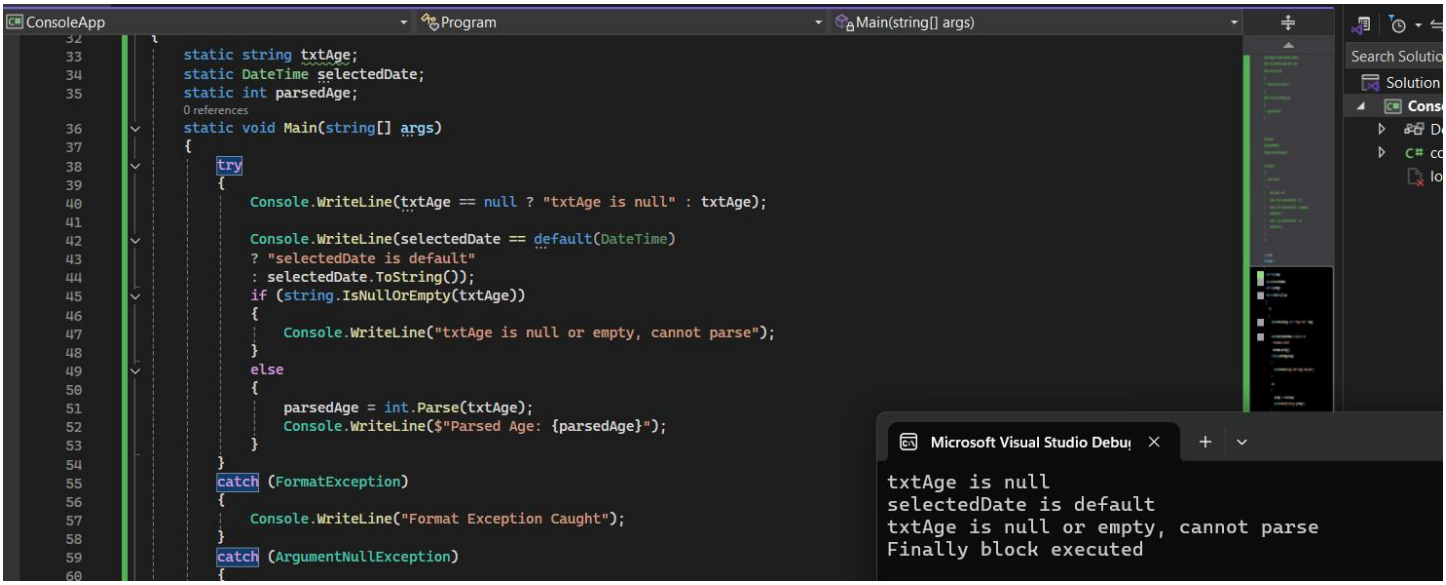
Output Reasoning (Level 1)

Code1:

```
using System;


class Program
{
    static string txtAge;
    static DateTime selectedDate;
    static int parsedAge;


    static void Main(string[] args)
    {
        try
        {


            Console.WriteLine(txtAge == null ? "txtAge is null" : txtAge);



            Console.WriteLine(selectedDate == default(DateTime)
                ? "selectedDate is default"
                : selectedDate.ToString());


            if (string.IsNullOrEmpty(txtAge))
            {
                Console.WriteLine("txtAge is null or empty, cannot parse");
            }
            else
            {
                parsedAge = int.Parse(txtAge);
                Console.WriteLine($"Parsed Age: {parsedAge}");
            }
        }
        catch (FormatException)
        {
            Console.WriteLine("Format Exception Caught");
        }
        catch (ArgumentNullException)
        {
            Console.WriteLine("ArgumentNull Exception Caught");
        }
        finally
        {
            Console.WriteLine("Finally block executed");
        }
    }
}
```

Output and Why?

When the program starts, all static fields (txtAge, selectedDate, and parsedAge) have their default values: txtAge is null, and selectedDate is DateTime.MinValue (the default). So the first Console.WriteLine prints "txtAge is null" because the ternary operator checks txtAge == null. Next, the second check prints "selectedDate is default" because selectedDate still has its default value. Then the if (string.IsNullOrEmpty(txtAge)) condition evaluates to true since txtAge is null, so the program prints "txtAge is null or empty, cannot parse" and never enters the else block where parsing occurs. No exceptions are thrown, so the catch blocks are skipped entirely. Finally, execution always reaches the finally block, printing "Finally block executed" before the program ends.

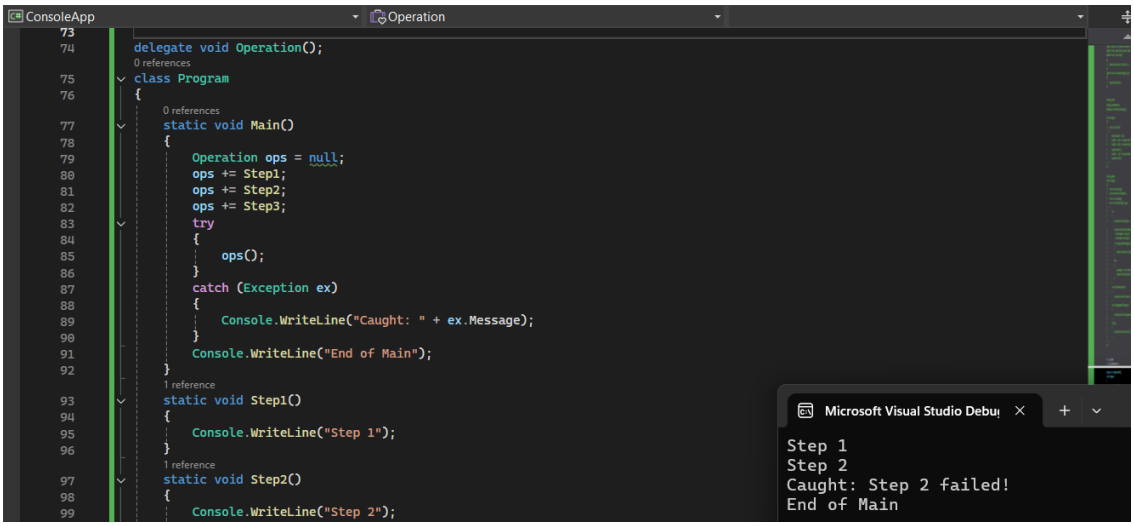


Code2:

```
using System;

delegate void Operation();

class Program
{
    static void Main()
    {
        Operation ops = null;

        ops += Step1;
        ops += Step2;
        ops += Step3;

        try
        {
            ops();
        }
        catch (Exception ex)
        {
            Console.WriteLine("Caught: " + ex.Message);
        }

        Console.WriteLine("End of Main");
    }

    static void Step1()
    {
        Console.WriteLine("Step 1");
    }

    static void Step2()
    {
        Console.WriteLine("Step 2");
        throw new InvalidOperationException("Step 2 failed!");
    }

    static void Step3()
    {
        Console.WriteLine("Step 3");
    }
}
```

Output and Why?

When Main starts, the multicast delegate ops is built by adding Step1, Step2, and Step3 in that order, meaning a single call to ops() will invoke each method sequentially. Inside the try block, the invocation begins: Step1 runs first and prints "Step 1" with no issues; then Step2 runs, prints "Step 2", and immediately throws an InvalidOperationException. As soon as this exception is thrown, the remaining methods in the invocation list—including Step3—are skipped completely, and control jumps to the catch block. The catch prints "Caught: Step 2 failed!", showing the exception message. After handling the exception, execution continues normally, and the final line "End of Main" is printed.



Output Reasoning (Level 2)

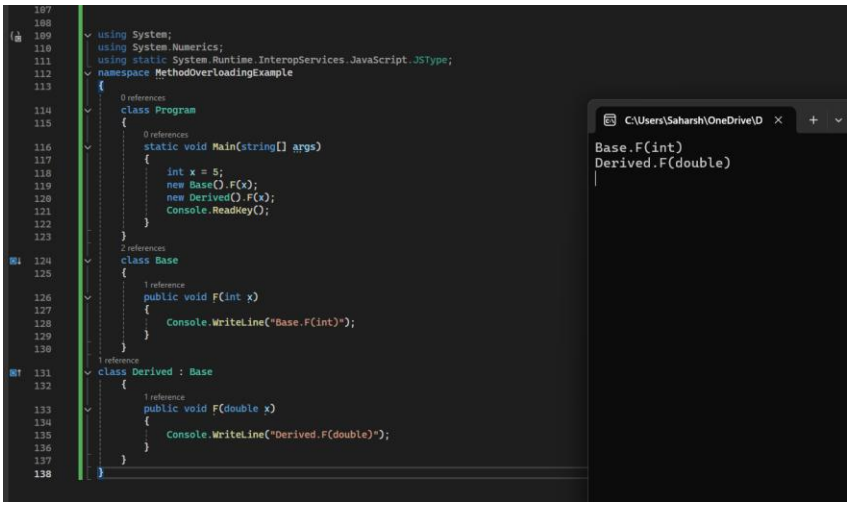
Code1:

```
using System;

namespace MethodOverloadingExample
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            new Base().F(x);
            new Derived().F(x);

            Console.ReadKey();
        }
    }

    class Base
    {
        public void F(int x)
        {
            Console.WriteLine("Base.F(int)");
        }
    }
```

```
    class Derived : Base
    {
        public void F(double x)
        {
            Console.WriteLine("Derived.F(double)");
        }
    }
}
```

Output and Why?

The execution begins in Main, where x is declared as an int with value 5. The first call, new Base().F(x), is resolved by the compiler by searching the Base class, where it finds an exact match F(int), so it prints “Base.F(int)”. Next, the call new Derived().F(x) is evaluated. The compiler first checks the Derived class for methods named F and finds F(double). Since an int can be implicitly converted to double, this method is considered a valid match, and therefore the compiler chooses Derived.F(double) without checking the base class for a more exact match. As a result, the program prints “Derived.F(double)”, demonstrating method hiding, where the presence of a new overload in the derived class prevents the base-class method from being selected.

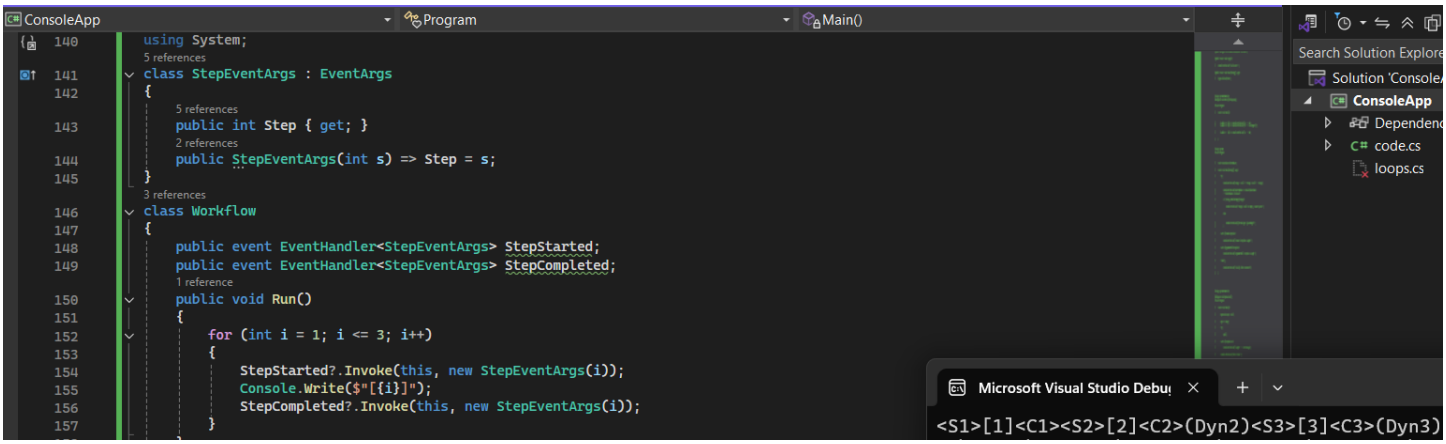


Code2:

```
using System;

class StepEventArgs : EventArgs
{
    public int Step { get; }
    public StepEventArgs(int s) => Step = s;
}

class Workflow
{
    public event EventHandler<StepEventArgs> StepStarted;
    public event EventHandler<StepEventArgs> StepCompleted;

    public void Run()
    {
        for (int i = 1; i <= 3; i++)
        {
            StepStarted?.Invoke(this, new StepEventArgs(i));
            Console.Write($"[{i}]");
            StepCompleted?.Invoke(this, new StepEventArgs(i));
        }
    }
}

class Program
{
    static void Main()
    {
        Workflow wf = new Workflow();

        wf.StepStarted += (s, e) =>
        {
            Console.Write("<S" + e.Step + ">");
            if (e.Step == 2)
                ((Workflow)s).StepCompleted += (snd, ev)
                    => Console.Write("(Dyn" + ev.Step + ")");
        };
        wf.StepCompleted += (s, e) => Console.Write("<C" + e.Step + ">");

        wf.Run();
    }
}
```

Output and Why?

The workflow runs steps 1 through 3, and before printing each step number, it fires StepStarted, then after printing it, it fires StepCompleted. For Step 1, the StepStarted handler prints <S1>, then the body prints [1], then the fixed StepCompleted handler prints <C1>. For Step 2, StepStarted prints <S2>, and because this is step 2, it dynamically adds a second StepCompleted handler that prints (DynN). After printing [2], Step 2's completion phase now has two handlers: the original <C2> and the newly added dynamic handler (Dyn2), so it prints both in that order. For Step 3, StepStarted prints <S3>, body prints [3], and StepCompleted now permanently has two handlers (because the second was attached during step 2), so it prints <C3> and (Dyn3) again.



RESULTS AND ANALYSIS

The OrderPipeline program that was put into action is a perfect example of multi-stage event chaining, using custom EventArgs, and the dynamic subscriber management. The firing of the OrderCreated event, hence the reaction of the two subscribers, is what logically starts the chain of events when the user processes an order. This results both in the validation and in the message-box summary.
The flow of the work system properly distinguishes between valid and invalid orders: in the case of a valid input, the call to ValidateOrder() will lead to the raising of the OrderConfirmed event and the status label will be updated to “Order Processed Successfully,” whereas if the quantities are invalid, OrderRejected will be immediately raised showing “Order Invalid – Please retry.”

DISCUSSION AND CONCLUSION

This project unveiled event-driven programming's massive power and its complex nature. The design of sequential events required a deep consideration of the program flow, especially the verification of the correct subscribers' triggering order and also ensuring that errors of one stage do not crash the subsequent stages. One major learning point was to understand the behaviour of delegates and events in C# which also entails issues like handling dynamic subscriptions, avoiding stale handlers and being certain that event filtering is done before event invocation.

REFERENCES

[1] LAB12

[2] Lec13

[3]Multicast_Delegates

[4] GitHub_Repo