

ASSESSMENT 4

Burra Saharsh

23110071

—

**Software Tools and Techniques
For CSE**

—

Professor Shouvik 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

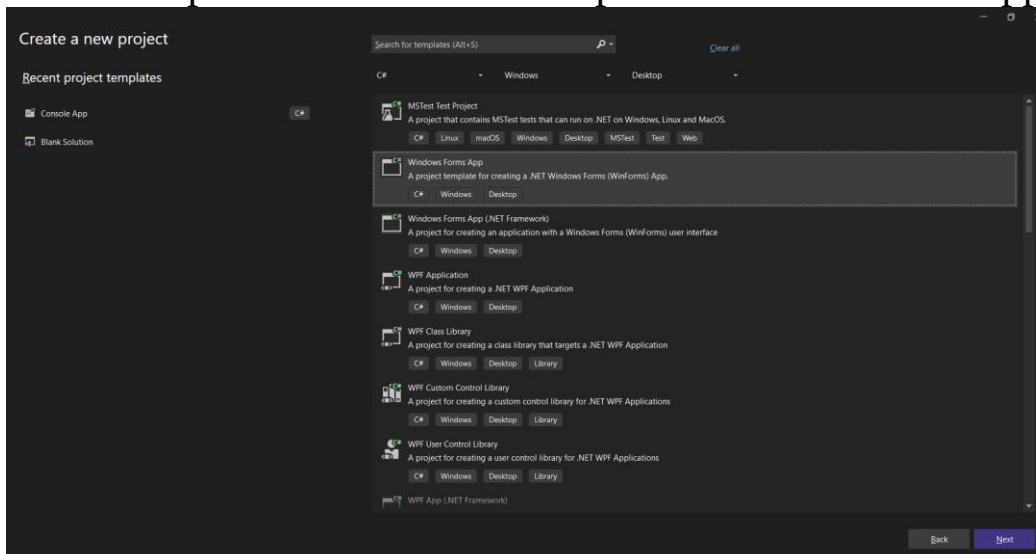
Software - Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK

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.

Configure your new project

Windows Forms App C# Windows Desktop

Project name
EventPlayground

Location
C:\Users\Saharsh\OneDrive\Desktop\STT_LABS\LAB11

Solution name ?
EventPlayground

☒ Place solution and project in the same directory

Project will be created in "C:\Users\Saharsh\OneDrive\Desktop\STT_LABS\LAB11\EventPlayground\"

Framework is .NET 8.0

Additional information

Windows Forms App C# Windows Desktop

Framework ?
.NET 8.0 (Long Term Support)

METHODOLOGY AND EXECUTION

Windows Forms App – Multi-Control Event Interaction: Form1.cs

```

1  using System;
2  using System.Drawing;
3  using System.Windows.Forms;
4  using System.Xml.Serialization;
5
6  namespace EventPlayground
7  {
8      public delegate void ColorHandler(Color newColor);
9      public delegate void TextHandler(string newText);
10
11     public partial class Form1 : Form
12     {
13         public event ColorHandler ColorEvent;
14         public event TextHandler TextEvent;
15
16         1 reference
17         public Form1()
18         {
19             InitializeComponent();
20             ColorEvent += OnColorChange;
21             TextEvent += OnTextChange;
22         }
23
24         1 reference
25         private void Form1_Load(object sender, EventArgs e)
26         {
27             cmbColors.Items.AddRange(new string[] { "Red", "Green", "Blue", "Orange", "Yellow", "Brown" });
28             cmbColors.SelectedIndex = 0;
29
30             //Event handler for ChangeColor click
31             1 reference
32             private void ChangeColor_Click(object sender, EventArgs e)
33             {
34                 // Get the selected color from ComboBox
35                 Color selectedColor = GetSelectedColor();
36
37                 // Raise the custom ColorChangedEvent
38                 ColorEvent?.Invoke(selectedColor);
39             }
40         }
41     }

```

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.

```

37 // Event handler for ChangeText click
38 1 reference
39 private void ChangeText_Click(object sender, EventArgs e)
40 {
41     // Create new text with current date and time
42     string newText = $"Current Time: {DateTime.Now:yyyy-MM-dd HH:mm:ss}";
43     // Raise the custom TextChangeEvent
44     TextEvent?.Invoke(newText);
45 }
46
47 // Custom event handler for ColorEvent
48 1 reference
49 private void OnColorChange(Color newColor)
50 {
51     lblDisplay.ForeColor = newColor;
52 }
53
54 // Custom event handler for TextEvent
55 1 reference
56 private void OnTextChange(string newText)
57 {
58     lblDisplay.Text = newText;
59 }
60
61 // Helper method to convert ComboBox selection to Color
62 1 reference
63 private Color GetSelectedColor()
64 {
65     return cmbColors.SelectedItem?.ToString() switch
66     {
67         "Red" => Color.Red,
68         "Green" => Color.Green,
69         "Blue" => Color.Blue,
70         "Orange" => Color.Orange,
71         "Yellow" => Color.Yellow,
72         "Brown" => Color.Brown,
73         _ => Color.Black
74     };
75 }

```

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

```

1 namespace EventPlayground
2 {
3     3 references
4     partial class Form1
5     {
6         /// <summary>
7         /// Required designer variable.
8         /// </summary>
9         private System.ComponentModel.IContainer components = null;
10
11         /// <summary>
12         /// Clean up any resources being used.
13         /// </summary>
14         /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
15         0 references
16         protected override void Dispose(bool disposing)
17         {
18             if (disposing && (components != null))
19             {
20                 components.Dispose();
21             }
22             base.Dispose(disposing);
23         }
24
25         #region Windows Form Designer generated code
26
27         /// <summary>
28         /// Required method for Designer support - do not modify
29         /// the contents of this method with the code editor.
30         /// </summary>
31         1 reference
32         private void InitializeComponent()
33         {
34             btnChangeColor = new Button();
35             btnChangeText = new Button();
36             lblDisplay = new Label();
37             cmbColors = new ComboBox();
38             SuspendLayout();
39             // btnChangeColor
40             //
41             // btnChangeText
42             //
43             // lblDisplay
44             //
45             // cmbColors
46             //
47             // Form1
48             //
49         }
50     }
51 }

```

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

```

37 // btnChangeColor
38 btnChangeColor.Location = new Point(50, 50);
39 btnChangeColor.Name = "btnChangeColor";
40 btnChangeColor.Size = new Size(120, 30);
41 btnChangeColor.TabIndex = 0;
42 btnChangeColor.Text = "Change Color";
43 btnChangeColor.UseVisualStyleBackColor = true;
44 btnChangeColor.Click += ChangeColor_Click;
45
46 // btnChangeText
47 //
48 btnChangeText.Location = new Point(200, 50);
49 btnChangeText.Name = "btnChangeText";
50 btnChangeText.Size = new Size(120, 30);
51 btnChangeText.TabIndex = 1;
52 btnChangeText.Text = "Change Text";
53 btnChangeText.UseVisualStyleBackColor = true;
54 btnChangeText.Click += ChangeText_Click;
55
56 // lblDisplay
57 //
58 lblDisplay.AutoSize = true;
59 lblDisplay.Font = new Font("Segoe UI", 12F, FontStyle.Bold);
60 lblDisplay.Location = new Point(50, 150);
61 lblDisplay.Name = "lblDisplay";
62 lblDisplay.Size = new Size(213, 28);
63 lblDisplay.TabIndex = 2;
64 lblDisplay.Text = "Welcome to Events Lab";
65
66 // cmbColors
67 //
68 cmbColors.DropDownStyle = ComboBoxStyle.DropDownList;
69 cmbColors.Location = new Point(50, 50);
70 cmbColors.Name = "cmbColors";
71 cmbColors.Size = new Size(180, 28);
72 cmbColors.TabIndex = 3;
73
74 // Form1
75 //

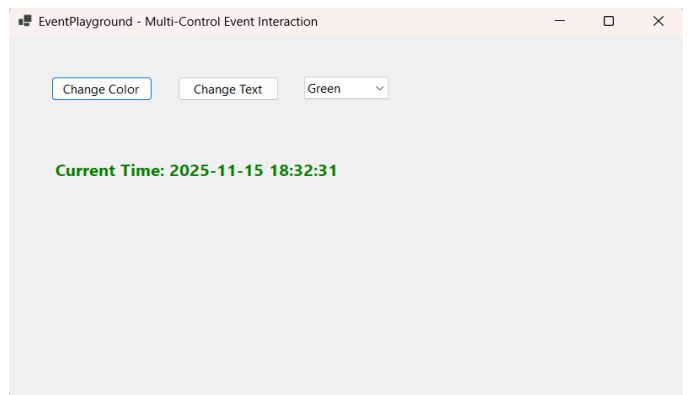
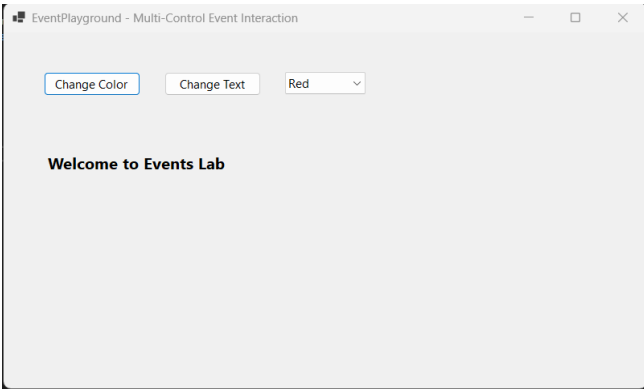
```

Here in the below I have initialized the four variables.

```

84 // LblDisplay.Text = "Welcome To Events Lab.";
85 //
86 // cmbColors
87 //
88 cmbColors.DropDownStyle = ComboBoxStyle.DropDownList;
89 cmbColors.Location = new Point(350, 50);
90 cmbColors.Name = "cmbColors";
91 cmbColors.Size = new Size(100, 20);
92 cmbColors.TabIndex = 3;
93 //
94 // Font1
95 //
96 AutoScaleDimensions = new.SizeF(8F, 20F);
97 AutoScaleMode = AutoScaleMode.Font;
98 ClientSize = new.SizeF(800, 450);
99 Controls.Add(cmbColors);
100 Controls.Add(lblDisplay);
101 Controls.Add(btnChangeText);
102 Controls.Add(btnChangeColor);
103 Name = "Form1";
104 Text = "EventPlayground - Multi-Control Event Interaction";
105 Load += Form1_Load;
106 ResumeLayout(false);
107 PerformLayout();
108 }
109 #endregion
110 private Button btnChangeColor;
111 private Button btnChangeText;
112 private Label lblDisplay;
113 private ComboBox cmbColors;

```



These are the results for the first task.

Using EventArgs and Multiple Subscribers: Form1.cs

```

1  using System;
2  using System.Drawing;
3  using System.Windows.Forms;
4  using System.Xml.Serialization;
5
6  namespace EventPlayground
7  {
8      //public delegate void ColorHandler(Color newColor);
9      //public delegate void TextHandler(string newText);
10
11     public delegate void ColorChangedEventHandler(object sender, ColorEventArgs e);
12     public delegate void TextChangedEventHandler(object sender, EventArgs e);
13
14     [Serializable]
15     public class ColorEventArgs : EventArgs
16     {
17         2 references
18         public string ColorName { get; }
19         2 references
20         public Color Color { get; }
21
22         1 reference
23         public ColorEventArgs(string colorName, Color color)
24         {
25             ColorName = colorName;
26             Color = color;
27         }
28     }
29
30     1 reference
31     public partial class Form1 : Form
32     {
33         //public event ColorHandler ColorEvent;
34         //public event TextHandler TextEvent;
35
36         public event ColorChangedEventHandler ColorChangedEvent;
37         public event TextChangedEventHandler TextChangedEvent;
38     }
39
40     1 reference

```

Updated the code for the delegate to handle multiple subscribers.


```

38 //TestEvent += OnTextChanged;
39
40 //Multiple subscribers to demonstrate multicast behavior
41 ColorChangeEvent += UpdateLabelColor; // First subscriber
42 ColorChangeEvent += ShowNotification; // Second subscriber
43 TestChangeEvent += OnTextChanged;
44
45
46
47 //reference
48 private void Form1_Load(object sender, EventArgs e)
49 {
50     cmColors.Items.AddRange(new string[] { "Red", "Green", "Blue", "Orange", "Yellow", "Brown" });
51     cmColors.SelectedIndex = 0;
52 }
53
54 //Event handler for ChangeColor click
55 //reference
56 private void ChangeColor_Click(object sender, EventArgs e)
57 {
58     //Get the selected color from ComboBox
59     Color selectedColor = GetSelectedColor();
60
61     //Raise the custom ColorChangeEvent
62     ColorEvent?.Invoke(selectedColor);
63
64     string colorName = cmColors.SelectedItem?.ToString() ?? "Red";
65     Color selectedColor = GetSelectedColor();
66
67     // Create ColorEventArgs and raise ColorChangeEvent
68     ColorEventArgs colorArgs = new ColorEventArgs(colorName, selectedColor);
69     ColorChangeEvent?.Invoke(this, colorArgs);
70 }
71
72 // Event handler for ChangeText click
73 //reference
74 private void ChangeText_Click(object sender, EventArgs e)
75 {
76     // Create new text with current date and time
77     string newText = $"Current Time: {DateTime.Now:yyyy-MM-dd HH:mm:ss}";
78     lblDisplay.Text = newText;
79 }

```

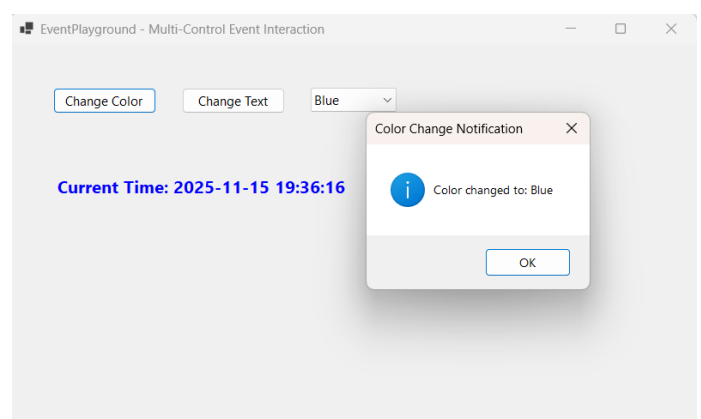
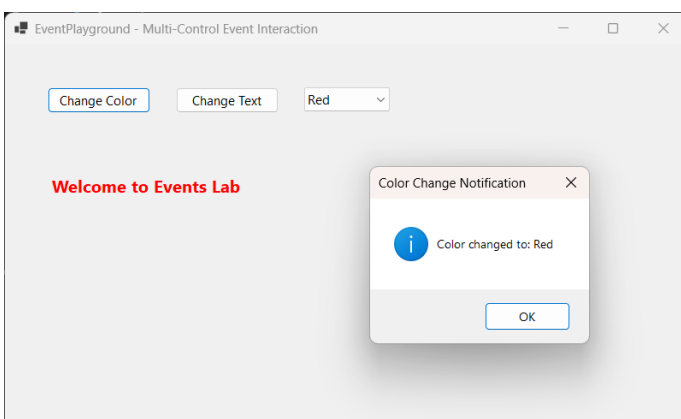
Implemented the code for showing the notification on color change with the help of ColorEventArgs class.

```

88 private void UpdateLabelColor(object sender, ColorEventArgs e)
89 {
90     lblDisplay.ForeColor = e.Color;
91 }
92
93 //Task 2 - Second subscriber: ShowNotification method
94 //reference
95 private void ShowNotification(object sender, ColorEventArgs e)
96 {
97     MessageBox.Show($"Color changed to: {e.ColorName}",
98         "Color Change Notification",
99         MessageBoxButtons.OK,
100         MessageBoxIcon.Information);
101 }
102
103 // Custom event handler for TestEvent
104 //private void OnTextChanged(string newText)
105 //{
106 //    lblDisplay.Text = newText;
107 //}
108
109 //Event handler for TestChangeEvent
110 //reference
111 private void OnTextChanged(object sender, EventArgs e)
112 {
113     lblDisplay.Text = $"Current Time: {DateTime.Now:yyyy-MM-dd HH:mm:ss}";
114 }
115
116 // Helper method to convert ComboBox selection to Color
117 //reference
118 private Color GetSelectedColor()
119 {
120     return cmColors.SelectedItem?.ToString() switch
121     {
122         "Red" => Color.Red,
123         "Green" => Color.Green,
124         "Blue" => Color.Blue,
125         "Orange" => Color.Orange,
126     };
127 }

```

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



Output Reasoning (Level o)

Code1:

```
using System;

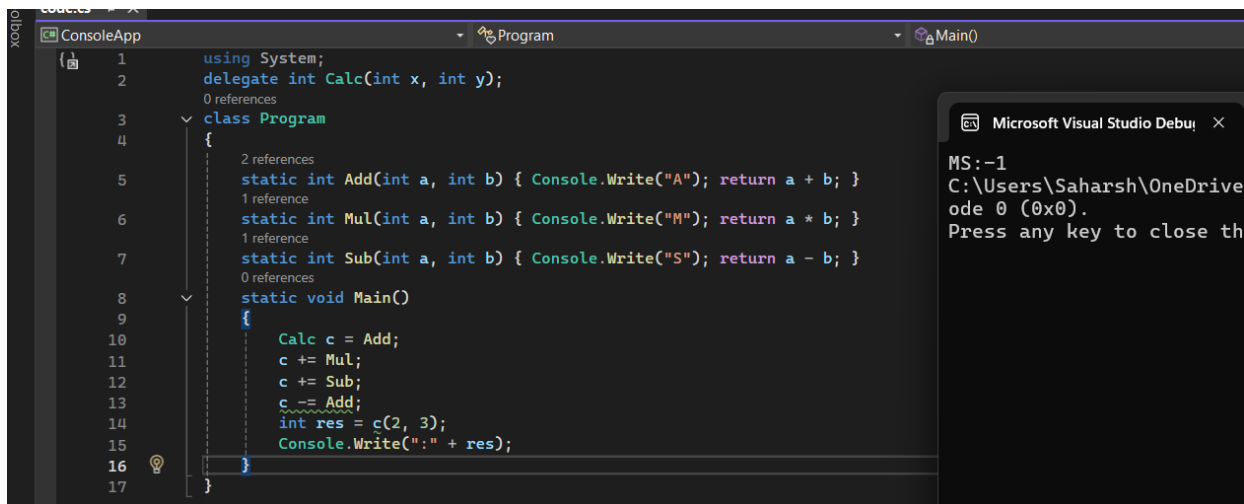
delegate int Calc(int x, int y);

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.WriteLine(":" + 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.

The screenshot shows the Visual Studio IDE with the same C# code as above. The code is highlighted with line numbers from 19 to 34. The output window on the right shows the execution results: "I5 D4 F4", the file path "C:\Users\Saharsh\OneDrive\Desktop\ode 0 (0x0)", and the prompt "Press any key to close this window".

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.WriteLine("[L] + e.CurrentValue + "]");
        c.LimitReached += (s, e) => Console.WriteLine("(Reset)");

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

The screenshot shows a Visual Studio code editor with a C# file named `code.cs`. The code defines a `LimitEventArgs` class and a `Program` class with a `Main` method. The `Main` method creates a `Counter` object and subscribes to its `LimitReached` event. The output window at the bottom shows the sequence of events: `>1>2[M2]>3[L3](Reset)>4[M4]{Alert}>5>6[M6][L6](Reset)`.

```
code.cs
51 Console.WriteLine(">" + value);
52 // Fire Milestone event every 2nd increment
53 if (value % 2 == 0)
54     MilestoneReached?.Invoke(this, new LimitEventArgs(value));
55 // Fire Limit event every 3rd increment
56 if (value % 3 == 0)
57     LimitReached?.Invoke(this, new LimitEventArgs(value));
58 }
59 }
60
61 class Program
62 {
63     static void Main()
64     {
65         Counter c = new Counter();
66         // Subscribers for LimitReached
67     }
68 }
69
70 Microsoft Visual Studio Debug Console
>1>2[M2]>3[L3](Reset)>4[M4]{Alert}>5>6[M6][L6](Reset)
```

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.

The screenshot shows the Visual Studio IDE with the following components:

- Code Editor:** Displays the `Program.cs` file. The `Main` method is visible, showing the sequence of `UpdateTemperature` calls (28, 30, 46, 52) and the two event handlers.
- ConsoleApp Window:** Shows the output of the program:


```

Temperature changed from 30°C to 46°C
Warning: Sudden change detected!
Temperature changed from 46°C to 52°C
      
```
- Solution Explorer:** Shows the project structure with `Program` and `Main()` visible.

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.WriteLine("[Start]");
        OnNotify?.Invoke(this, new NotifyEventArgs(msg));
        Console.WriteLine("[End]");
    }
}

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

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

        n.OnNotify += (s, e) =>
        {
            Console.WriteLine("(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].

```
142 OnNotify?.Invoke(this, new NotifyEventArgs(msg));
143 Console.WriteLine("End");
144 }
145 }
146 class Program
147 {
148     static void Main()
149     {
150         Notifier n = new Notifier();
151         n.OnNotify += (s, e) =>
152         {
153             Console.WriteLine("{0} {1}", e.Message, "Nested");
154         };
155         n.OnNotify += (s, e) =>
156         {
157             Console.WriteLine("Nested");
158             if (e.Message == "Ping")
159                 ((Notifier)s).Trigger("Pong");
160         };
161         n.Trigger("Ping");
162     }
163 }
```

Microsoft Visual Studio Debug Console: [Start]{Ping}(Nested)[Start]{Pong}(Nested)[End][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.WriteLine("[Check]");
        if (value > 50)
            ThresholdReached?.Invoke(this, new AlertEventArgs("High"));
        Console.WriteLine("[Done]");
    }
}

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

        s.ThresholdReached += (sender, e) =>
        {
            Console.WriteLine("{0} {1}", e.Info, "Nested");
            if (e.Info == "High")
                ((Sensor)sender).Check(30);
        };

        s.ThresholdReached += (sender, e) =>
        {
            Console.WriteLine("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]`.

```
ConsoleApp
170
171 3 references
172 class Sensor
173 {
174     public event EventHandler<AlertEventArgs> ThresholdReached;
175     2 references
176     public void Check(int value)
177     {
178         Console.WriteLine("[Check]");
179         if (value > 50)
180             ThresholdReached?.Invoke(this, new AlertEventArgs("High"));
181         Console.WriteLine("[Done]");
182     }
183 }
184 0 references
185 class Program
186 {
187     0 references
188     static void Main()
189     {
190         Sensor s = new Sensor();
191         s.ThresholdReached += (sender, e) =>
192         {
193             Console.WriteLine("{ " + e.Info + " }");
194             if (e.Info == "High")
195                 ((Sensor)sender).Check(30);
196         }
197     }
198 }
```

Microsoft Visual Studio Debug Console

[Check]{High}[Check][Done](Alert)[Done]
C:\Users\Saharsh\OneDrive\Desktop\STT_LABS

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

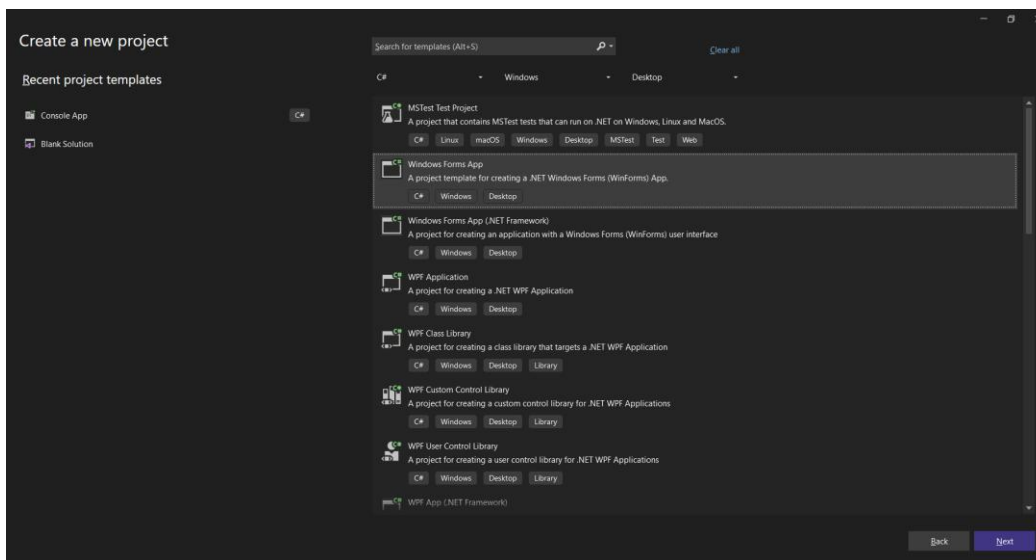
Software - Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK

Programming Language: C#

Framework - .NET 8.0

SETUP

Created a new project in the STT_Lab12 folder.



Configure your new project

Windows Forms App C# Windows Desktop

Project name

OrderPipeline

Location

C:\Users\Saharsh\OneDrive\Desktop\STT_LABS\LAB12\

Solution name ⓘ

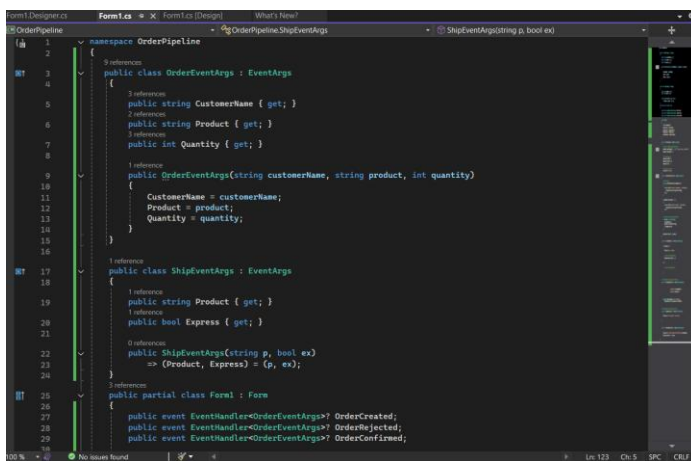
OrderPipeline

☒ Place solution and project in the same directory

Project will be created in "C:\Users\Saharsh\OneDrive\Desktop\STT_LABS\LAB12\OrderPipeline\"

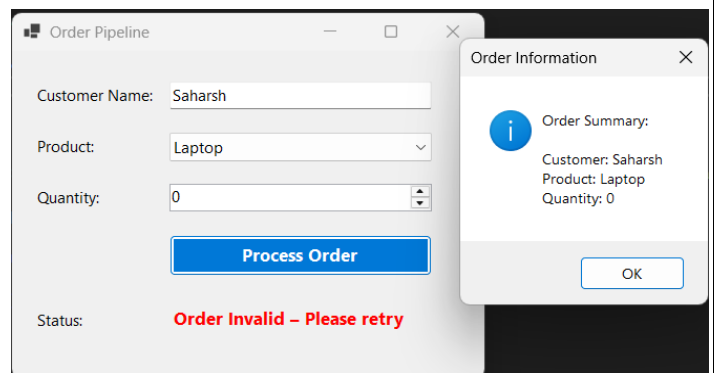
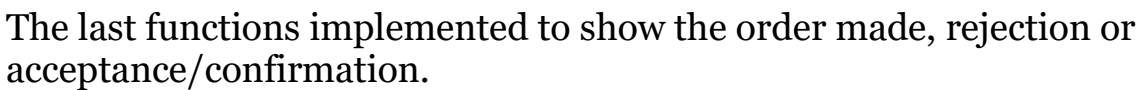
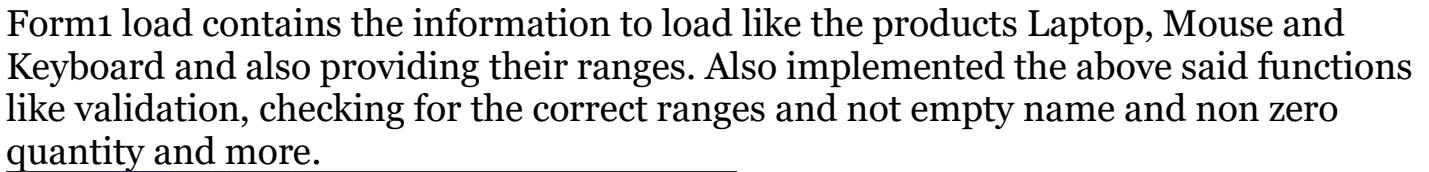
METHODOLOGY AND EXECUTION

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



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.



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”.

The screenshot shows a web application titled "Order Pipeline". It has three input fields: "Customer Name" with the value "Saharsh", "Product" with a dropdown menu showing "Keyboard", and "Quantity" with a value of "1". Below these fields is a blue button labeled "Process Order". At the bottom, the status is displayed in green text: "Status: Order Processed Successfully for Saharsh". An "Order Information" dialog box is open, showing an "Order Summary" with the details: "Customer: Saharsh", "Product: Keyboard", and "Quantity: 1". The dialog has an "OK" button.

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:

```

Form1.Designer.cs  Form1.cs  What's New?
OrderPipeline
4
5 9 references
6 public class OrderEventArgs : EventArgs
7 {
8     3 references
9     public string CustomerName { get; }
10    3 references
11    public string Product { get; }
12    3 references
13    public int Quantity { get; }
14
15    1 reference
16    public OrderEventArgs(string customerName, string product, int quantity)
17    {
18        CustomerName = customerName;
19        Product = product;
20        Quantity = quantity;
21    }
22
23    5 references
24    public class ShipEventArgs : EventArgs
25    {
26        3 references
27        public string Product { get; }
28        3 references
29        public bool Express { get; }
30
31        1 reference
32        public ShipEventArgs(string p, bool ex)
33        => (Product, Express) = (p, ex);
34    }
35
36    3 references
37    public partial class Form1 : Form
38    {
39        public event EventHandler<OrderEventArgs>? OrderCreated;
40        public event EventHandler<OrderEventArgs>? OrderRejected;
41        public event EventHandler<OrderEventArgs>? OrderConfirmed;
42
43        //Task 2 event - OrderShipped with ShipEventArgs
44        public event EventHandler<ShipEventArgs>? OrderShipped;
45        private bool isOrderConfirmed = false;
46        private string currentProduct = string.Empty;
47    }

```

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.


```

Form1.Designer.cs  Form1.cs  What's New?
OrderPipeline  OrderPipeline.OrderEventArgs  OrderEventArgs(string customerName, string product, int
92      OrderCreated?.Invoke(this, orderArgs);
93  }
94
95  //Task 2 - Ship Order button click handler
96  1 reference
97  private void btnShipOrder_Click(object sender, EventArgs e)
98  {
99      // Event Filtering: Check if order was confirmed using boolean flag
100      if (!isOrderConfirmed)
101      {
102          MessageBox.Show("Please process and confirm an order first.",
103              "Order Not Confirmed",
104              MessageBoxButtons.OK, MessageBoxIcon.Warning);
105          return;
106      }
107
108      // Dynamic Subscriber Management: Add/Remove NotifyCourier based on checkbox
109      if (chkExpress.Checked)
110      {
111          // Add NotifyCourier subscriber for express delivery
112          OrderShipped -= NotifyCourier; // Remove first to avoid duplicate subscription
113          OrderShipped += NotifyCourier; // Add the subscriber
114      }
115      else
116      {
117          // Remove NotifyCourier subscriber for regular delivery
118          OrderShipped -= NotifyCourier;
119      }
120
121      // Create shipping event args and raise OrderShipped event
122      var shipArgs = new ShipEventArgs(currentProduct, chkExpress.Checked);
123      OrderShipped?.Invoke(this, shipArgs);
124
125      // Reset after shipping
126      isOrderConfirmed = false;
127      btnShipOrder.Enabled = false;
128  }
129
130  //Task 2 - Express checkbox changed event handler
131  1 reference
132  private void chkExpress_CheckedChanged(object sender, EventArgs e)
133  {

```

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

```

Form1.Designer.cs  Form1.cs  What's New?
OrderPipeline  OrderPipeline.OrderEventArgs  OrderEventArgs(string customerName, string product, int
186  }
187
188  // Event Subscriber: Handles order confirmation
189  1 reference
190  private void ShowConfirmation(object? sender, OrderEventArgs e)
191  {
192      lblStatus.Text = $"Order Processed Successfully for {e.CustomerName}";
193      lblStatus.ForeColor = Color.Green;
194
195      //Task 2 - Set flag to true and enable ship button
196      isOrderConfirmed = true;
197      btnShipOrder.Enabled = true;
198  }
199
200  // Event Subscriber: Shows dispatch information (ALWAYS SUBSCRIBED)
201  1 reference
202  private void ShowDispatch(object? sender, ShipEventArgs e)
203  {
204      string shippingType = e.Express ? "Express" : "Standard";
205      lblStatus.Text = $"Product dispatched: {e.Product} ({shippingType})";
206      lblStatus.ForeColor = Color.Blue;
207  }
208
209  // Event Subscriber: Notifies courier for express delivery (DYNAMICALLY SUBSCRIBED)
210  3 references
211  private void NotifyCourier(object? sender, ShipEventArgs e)
212  {
213      if (e.Express)
214      {
215          MessageBox.Show($"Express delivery initiated!\n\n" +
216              $"Product: {e.Product}\n" +
217              $"Priority: HIGH\n" +
218              $"Courier has been notified.",
219              "Express Shipping",
220              MessageBoxButtons.OK,
221              MessageBoxIcon.Information);
222      }
223  }
224
225  }

```

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

Order Pipeline - Task 2: Event Filteri...

Customer Name:

Product:

Quantity:

Step 1: Order Processing

Step 2: Shipping (Task 2)

☐ Enable Express Shipping

Status: **Ready**

Order Pipeline - Task 2: Event Filtering

Customer Name:

Product:

Quantity:

Step 1: Order Processing

Step 2: Shipping (Task 2)

☐ Enable Express Shipping

Status: **Order Processed Successfully for Saharsh**

Order Information

Order Summary:

Customer: Saharsh

Product: Laptop

Quantity: 10

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

Order Pipeline - Task 2: Event Filtering

Customer Name:

Product:

Quantity:

Step 1: Order Processing

Step 2: Shipping (Task 2)

☐ Enable Express Shipping

Status: **Product dispatched: Laptop (Standard)**

Order Pipeline - Task 2: Event Filtering

Customer Name:

Product:

Quantity:

Step 1: Order Processing

Step 2: Shipping (Task 2)

☒ Enable Express Shipping

Status: **Product dispatched: Laptop (Express)**

Express Shipping

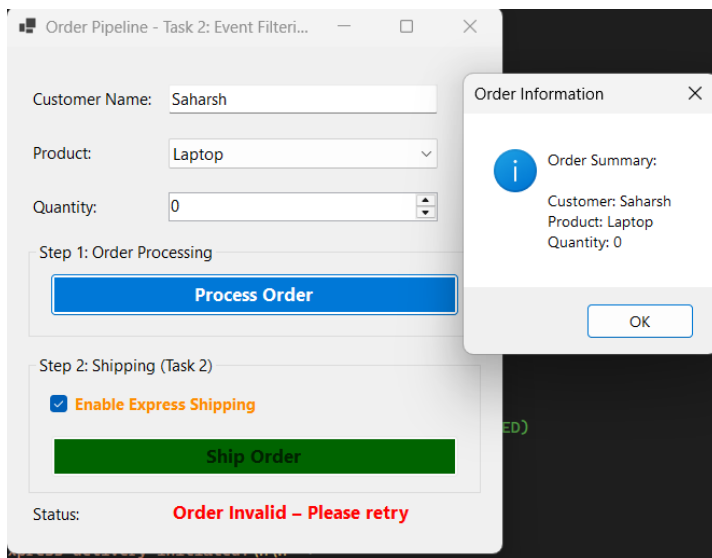
Express delivery initiated!

Product: Laptop

Priority: HIGH

Courier has been notified.

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.



The screenshot shows the Visual Studio Code editor with the C# code from the previous block. The code is highlighted in blue and green. To the right, the 'Microsoft Visual Studio Debug Console' window is open, displaying the output of the program: A: hello, B: HELLO, A: world, 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);

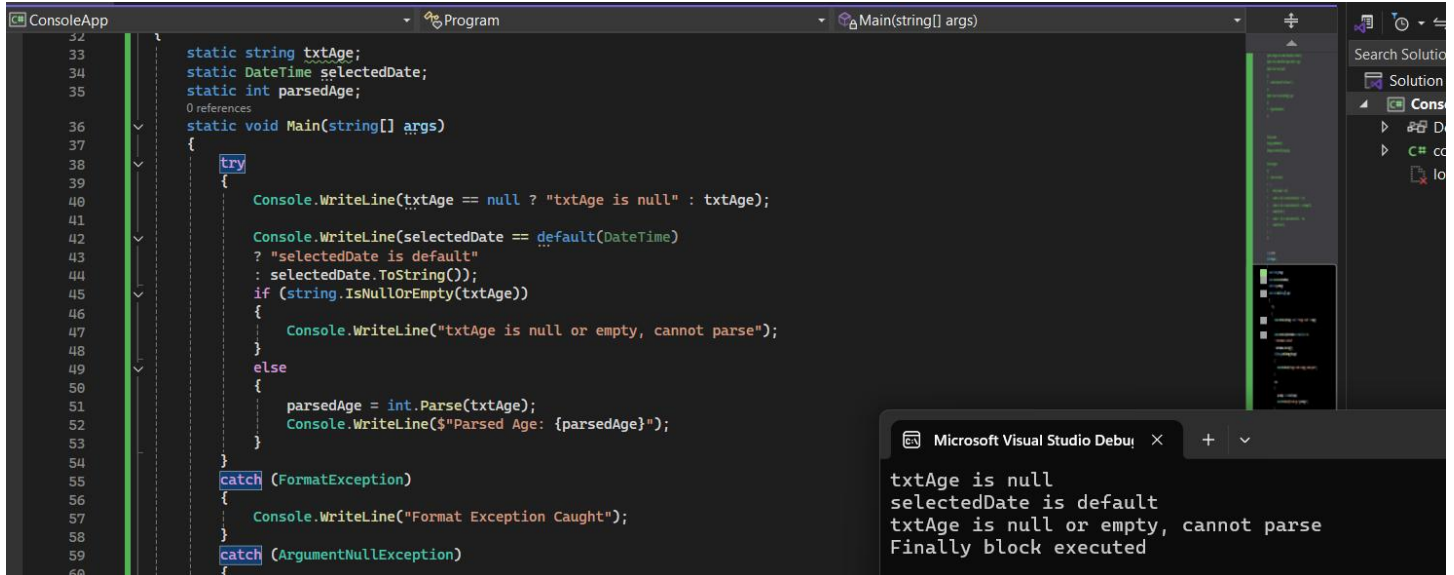
            Console.WriteLine(selectedDate == default(DateTime)
                ? "selectedDate is default"
                : selectedDate.ToString());

            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");
    }

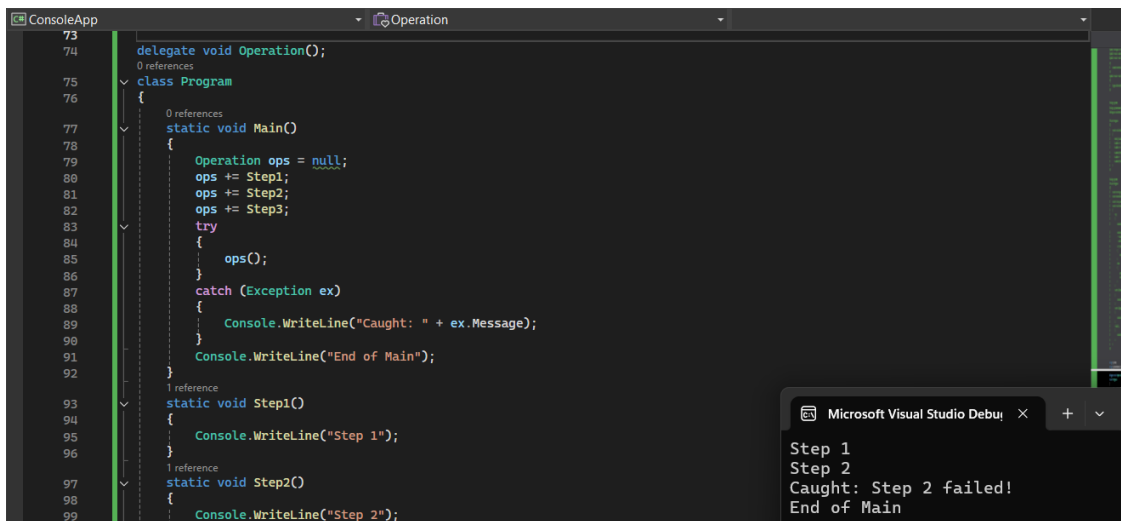
    static void Step2()
    {
        Console.WriteLine("Step 2");
        throw new InvalidOperationException("Step 2 failed!");
    }

    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.



The screenshot shows a Visual Studio IDE with a C# file named `ConsoleApp`. The code defines a delegate `Operation` and a class `Program` with a `Main` method. The `Main` method builds a multicast delegate `ops` by adding `Step1`, `Step2`, and `Step3`. It then calls `ops()` inside a `try` block. `Step1` prints "Step 1", `Step2` prints "Step 2" and throws an `InvalidOperationException`. The `catch` block prints "Caught: Step 2 failed!". Finally, it prints "End of Main". The output window shows the execution results: "Step 1", "Step 2", "Caught: Step 2 failed!", and "End of Main".

```
73 delegate void Operation();
74
75 class Program
76 {
77     static void Main()
78     {
79         Operation ops = null;
80         ops += Step1;
81         ops += Step2;
82         ops += Step3;
83         try
84         {
85             ops();
86         }
87         catch (Exception ex)
88         {
89             Console.WriteLine("Caught: " + ex.Message);
90         }
91         Console.WriteLine("End of Main");
92     }
93     static void Step1()
94     {
95         Console.WriteLine("Step 1");
96     }
97     static void Step2()
98     {
99         Console.WriteLine("Step 2");
100     }
101     static void Step3()
102     {
103         Console.WriteLine("Step 3");
104     }
105 }
```

Microsoft Visual Studio Debug Console Output:

```
Step 1
Step 2
Caught: Step 2 failed!
End of Main
```

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)");
        }
    }
}
```

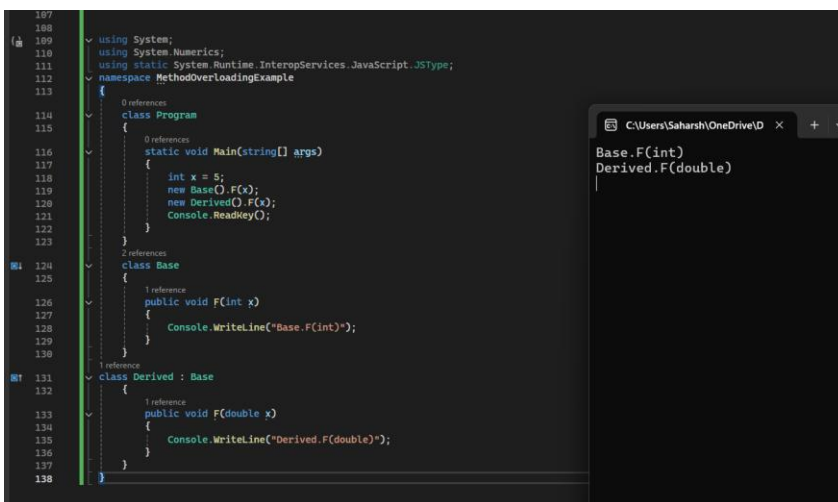
```

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.WriteLine($"[{i}]");
            StepCompleted?.Invoke(this, new StepEventArgs(i));
        }
    }
}

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

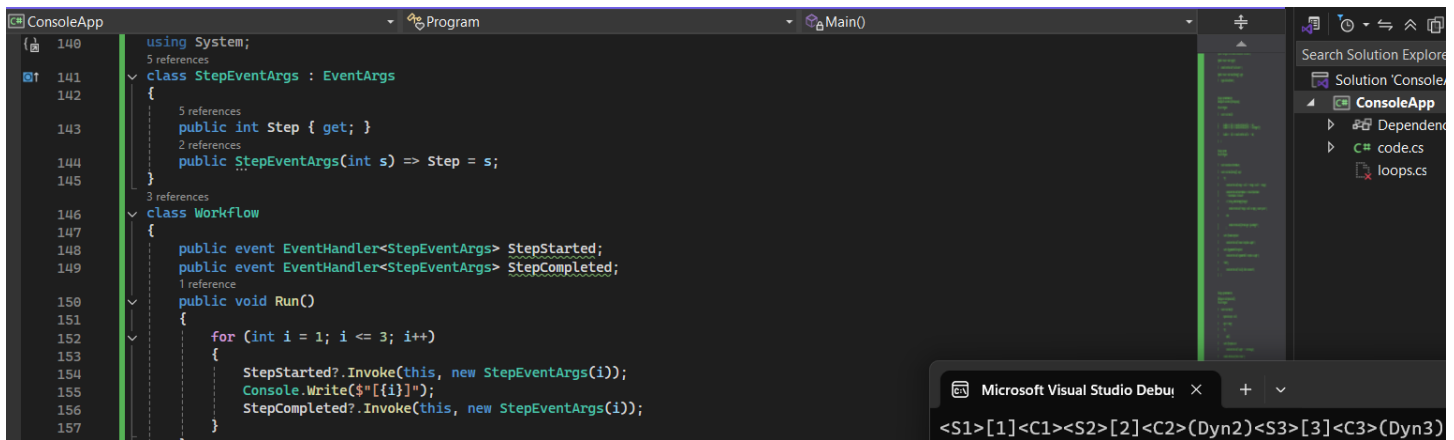
        wf.StepStarted += (s, e) =>
        {
            Console.WriteLine("<S" + e.Step + ">");
            if (e.Step == 2)
                ((Workflow)s).StepCompleted += (snd, ev)
                    => Console.WriteLine("(Dyn" + ev.Step + ")");
        };
        wf.StepCompleted += (s, e) => Console.WriteLine("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

A screenshot of the Visual Studio IDE. The main window shows a C# code file with two classes: `StepEventArgs` and `Workflow`. `StepEventArgs` inherits from `EventArgs` and has a `Step` property and a constructor. `Workflow` has two events, `StepStarted` and `StepCompleted`, and a `Run()` method. The `Run()` method contains a `for` loop that iterates from 1 to 3, invoking `StepStarted` and `StepCompleted` events. The right sidebar shows the Solution Explorer with a project named `ConsoleApp` containing `code.cs` and `loops.cs`. The bottom status bar shows the Microsoft Visual Studio Debugger with a list of variables: `<S1>[1]<C1><S2>[2]<C2>(Dyn2)<S3>[3]<C3>(Dyn3)`.

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](#)