ASSIGNMENT 4

SOFTWARE TOOLS AND TECHNIQUES FOR CSE

Revathi Katta, Sai Keerthana Pappala

Roll No: 23110159 , 23110229

November 19, 2025

Table of Contents

1 LABORATORY SESSION 11

INTRODUCTION, SETUP AND TOOLS

This report documents the development and analysis conducted on Events and Delegates in C# Windows Forms Applications. The core objective was to deepen understanding and practical application of the publisher–subscriber model using custom events and delegates in a Graphical User Interface (GUI) environment. The primary learning outcomes targeted were implementing and handling custom events in C# using the publisher–subscriber model, designing interactive controls that respond dynamically to user actions, applying multicast event handling to connect multiple subscribers to a single event, and utilizing custom EventArgs classes to pass contextual data between publishers and subscribers.

Environment and Tools:

- **Operating System:** Windows 10 (or later)
- **Development Environment:** Visual Studio 2022 Community Edition
- **.NET SDK:** Latest stable version (e.g., .NET 8.0)
- **Programming Language:** C#
- **Project Type:** Windows Forms App (.NET)

METHODOLOGY AND EXECUTION

This section details the step-by-step procedure for developing the EventPlayground application (Tasks 1 and 2) and provides the in-depth reasoning for the C# code analysis questions (Tasks 3, 4, and 5).

1.1 Windows Forms App – Multi-Control Event Interaction (Task 1 & 2 Implementation)

Form Layout and Component Setup

A new Windows Forms application named EventPlayground was created. The following controls were added and configured in the Designer: Buttons: `btnChangeColor` ("Change Color"), `btnChangeText` ("Change Text"); Label: `lblInfo` (initial text: "Welcome to Events Lab"); ComboBox: `cmbColors` (Items: Red, Green, Blue).

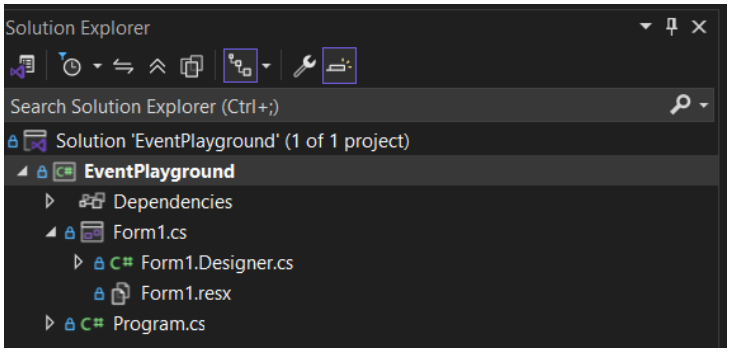


Figure 1: Solution Explorer - Default Project Structure

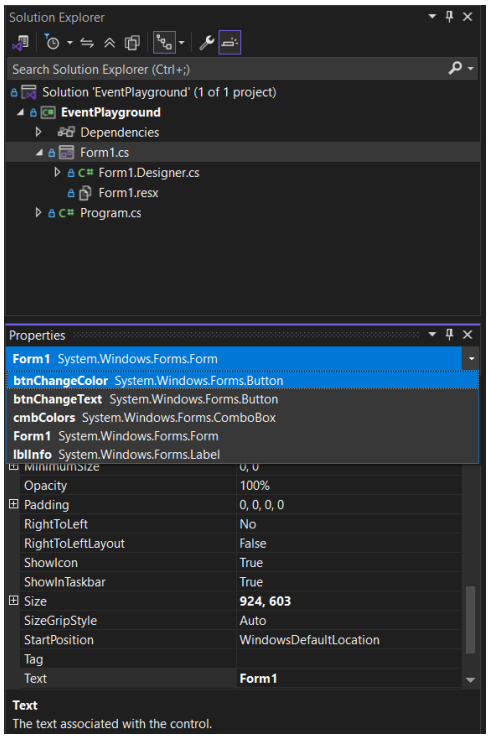


Figure 2: Form Designer showing controls with their Name

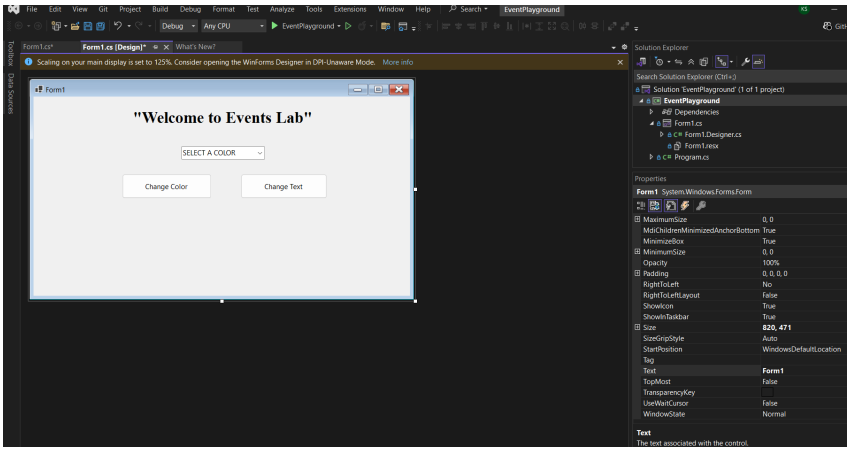


Figure 3: Designer View - Placing Controls

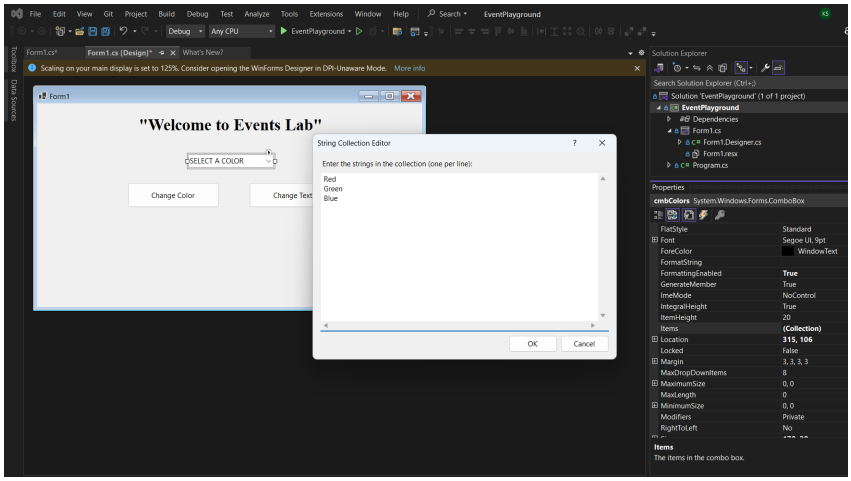


Figure 4: ComboBox Items Setup

Custom Delegate and EventArgs Definitions (Task 2 Extension)

Here I defined two custom delegates: one for color changes and one for text changes and a custom `EventArgs` class to pass selected color name with the event.

Listing 1: Custom Delegate and EventArgs Definitions

```
// Custom Delegate for Color Change Event (passing ColorEventArgs)
public delegate void ColorChangedEventHandler(object sender, ColorEventArgs
    e);

// Custom Delegate for Text Change Event (using standard EventArgs)
public delegate void TextChangedEventHandler(object sender, EventArgs e);

// Custom EventArgs class to carry the selected color data (Task 2)
public class ColorEventArgs : EventArgs
{
    public string SelectedColorName { get; set; }
    public ColorEventArgs(string colorName)
    {
        SelectedColorName = colorName;
    }
}
```

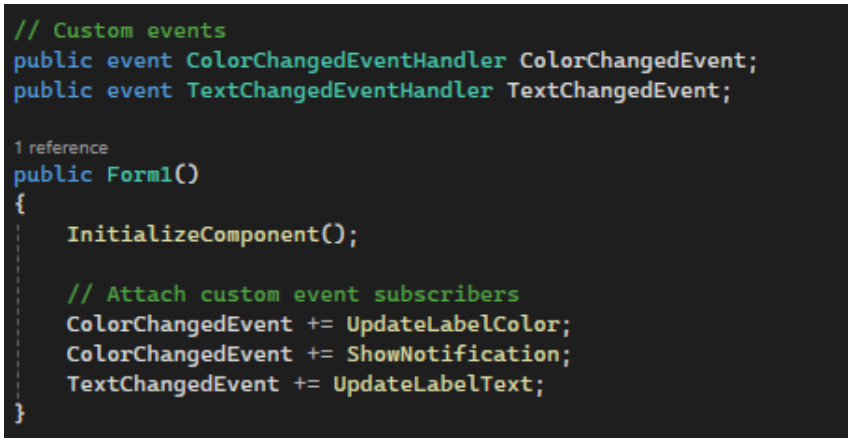
```
// Custom delegate for color change event
public delegate void ColorChangedEventHandler(object sender, ColorEventArgs e);
// Custom delegate for text change event
public delegate void TextChangedEventHandler(object sender, EventArgs e);

// Custom EventArgs for Color
5 references
public class ColorEventArgs : EventArgs
{
    3 references
    public string SelectedColorName { get; set; }
    1 reference
    public ColorEventArgs(string colorName)
    {
        SelectedColorName = colorName;
    }
}
```

Declaring and Connecting Events

Declared `ColorChangedEvent` and `TextChangedEvent` using the custom delegates and Subscribed two methods to `ColorChangedEvent` for multicast event handling.

Listing 2: Declaring and Connecting Events

```
// Event Declarations
public event ColorChangedEventHandler ColorChangedEvent;
public event TextChangedEventHandler TextChangedEvent;

// Inside the Form1() Constructor:
public Form1()
{
    InitializeComponent();

    // Task 2: Multicast for ColorChangedEvent
    ColorChangedEvent += UpdateLabelColor;
    ColorChangedEvent += ShowNotification;

    // Subscription for TextChangedEvent
    TextChangedEvent += UpdateLabelText;
}
```



Defining Subscriber Methods

The subscriber methods implement the specific logic that runs when the events are raised. UpdateLabelColor uses `ColorEventArgs` to retrieve the selected color based on the combusted ComboBox value and update `lblInfo.ForeColor`. ShowNotification is a multicast subscriber that Pops up a message box with the selected color name. UpdateLabelText updates `lblInfo.Text` to the current system date and time.

Listing 3: Subscriber Methods

```
private void UpdateLabelColor(object sender, ColorEventArgs e)
{
    switch (e.SelectedColorName)
    {
        case "Red": lblInfo.ForeColor = Color.Red; break;
        case "Green": lblInfo.ForeColor = Color.Green; break;
        case "Blue": lblInfo.ForeColor = Color.Blue; break;
        default: lblInfo.ForeColor = Color.Black; break;
```

```
    }
}

private void ShowNotification(object sender, ColorEventArgs e)
{
    // Task 2: Notification subscriber for multicast
    MessageBox.Show($"Color changed to {e.SelectedColorName}", "
       Notification");
}

private void UpdateLabelText(object sender, EventArgs e)
{
    lblInfo.Text = DateTime.Now.ToString("F");
}
```

```
// Event subscriber methods:

// 1. Change label foreground color
1 reference
private void UpdateLabelColor(object sender, ColorEventArgs e)
{
    switch (e.SelectedColorName)
    {
        case "Red":
            lblInfo.ForeColor = Color.Red;
            break;
        case "Green":
            lblInfo.ForeColor = Color.Green;
            break;
        case "Blue":
            lblInfo.ForeColor = Color.Blue;
            break;
        default:
            lblInfo.ForeColor = Color.Black;
            break;
    }
}

// 2. Show notification popup with selected color
1 reference
private void ShowNotification(object sender, ColorEventArgs e)
{
    MessageBox.Show($"Color changed to {e.SelectedColorName}", "Notification");
}

// 3. Change label text to current date/time
1 reference
private void UpdateLabelText(object sender, EventArgs e)
{
    lblInfo.Text = DateTime.Now.ToString("F"); // Full date and time
}
```

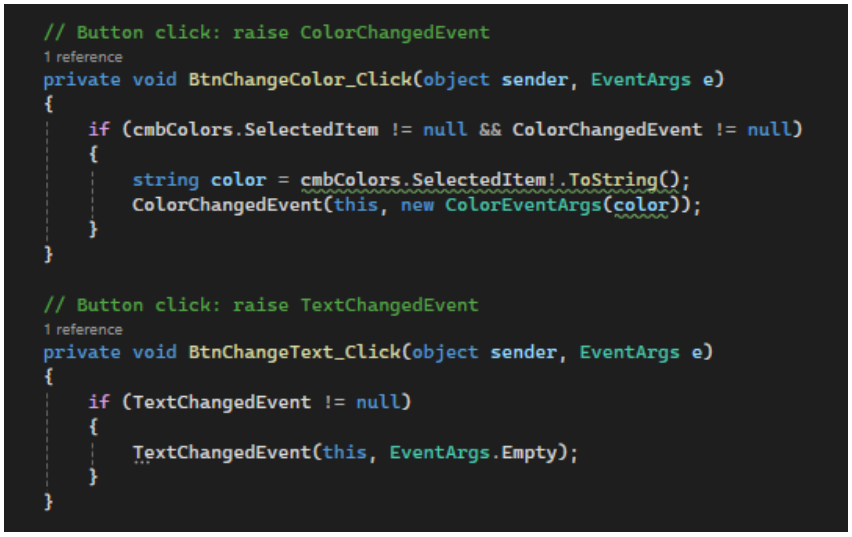
Raising Custom Events in Button Handlers

The standard Click events of the buttons act as the initial trigger, which then invoke (raise) the custom delegates/events, passing the necessary EventArgs objects. The `?.Invoke` syntax or an explicit null check is used for safety.

Listing 4: Raising Custom Events

```
private void BtnChangeColor_Click(object sender, EventArgs e)
{
    // Check for selected item and subscribers before invoking
    if (cmbColors.SelectedItem != null && ColorChangedEvent != null)
    {
        string color = cmbColors.SelectedItem!.ToString();

        // Invoke the custom event, passing the custom EventArgs
```

```
        ColorChangedEvent(this, new ColorEventArgs(color));
    }
}

private void BtnChangeText_Click(object sender, EventArgs e)
{
    if (TextChangedEvent != null)
    {
        TextChangedEvent(this, EventArgs.Empty);
    }
}
```



```
// Button click: raise ColorChangedEvent
1 reference
private void BtnChangeColor_Click(object sender, EventArgs e)
{
    if (cmbColors.SelectedItem != null && ColorChangedEvent != null)
    {
        string color = cmbColors.SelectedItem!.ToString();
        ColorChangedEvent(this, new ColorEventArgs(color));
    }
}

// Button click: raise TextChangedEvent
1 reference
private void BtnChangeText_Click(object sender, EventArgs e)
{
    if (TextChangedEvent != null)
    {
        TextChangedEvent(this, EventArgs.Empty);
    }
}
```

1.2 Output Reasoning Level 0

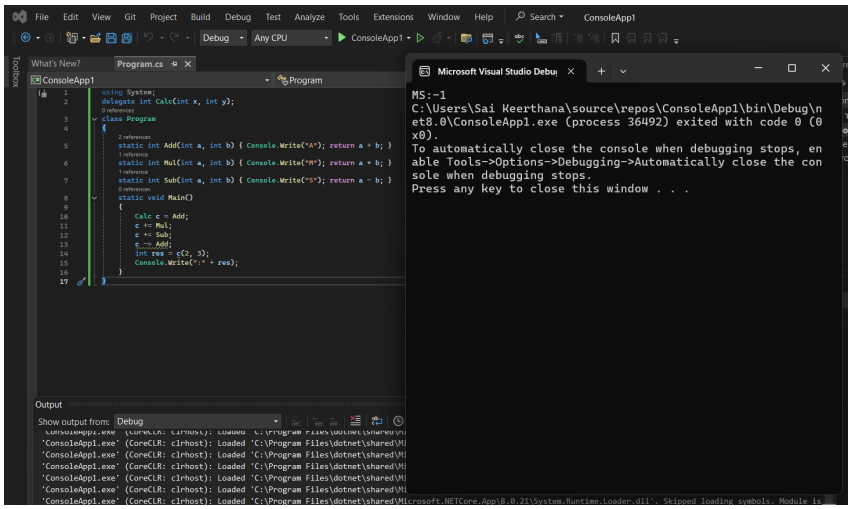
1. **Code 1: Multicast Delegate Return Value**:

```
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:

```
MS:-1
```

Reasoning: The Calc delegate becomes a multicast delegate with the methods Mul and Sub remaining in the invocation list after Add is added and then removed. When c(2, 3) is called, all methods in the invocation list execute in order: Mul(2, 3) executes, prints M, and returns 6. Sub(2, 3) executes, prints S, and returns $2 - 3 = -1$. In C, for a non-void multicast delegate, the final return value (res) is the result of the last method invoked, which is Sub(2, 3), resulting in -1. The final output concatenates the method prints and the result.



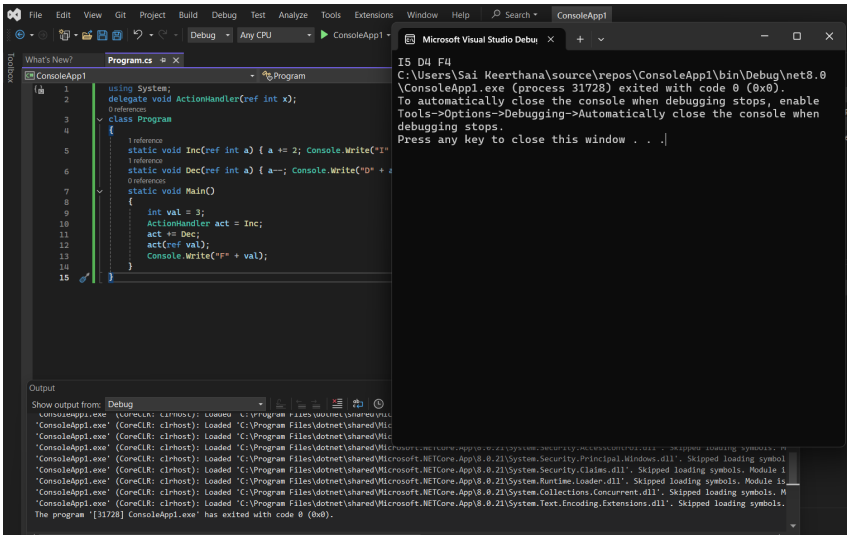
2. **Code 2:Multicast Delegate with `ref` Parameter**:

```
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:

```
I5 D4 F4
```

Reasoning: The ActionHandler delegate uses a ref parameter, meaning all methods in the multicast invocation list modify the same variable in memory. The initial value of val is 3. Inc(ref val) executes: val becomes $3 + 2 = 5$. It prints I5 . Dec(ref val) executes: val becomes $5 - 1 = 4$. It prints D4 . After the delegate invocation, val holds the final value of 4. The main method then prints F4.



1.3 Output Reasoning Level 1

1. **Code 1: Custom Events and Multicast Firing**:

```
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:

```
>1>2[M2]>3[L3](Reset)>4[M4]{Alert}>5>6[M6][L6](Reset)
```

Reasoning: The loop runs 6 times, incrementing `value` from 1 to 6.

`MilestoneReached` fires every 2nd increment (`value % 2 == 0`). `LimitReached` fires every 3rd increment (`value % 3 == 0`). The `value` is printed with > before any events fire.

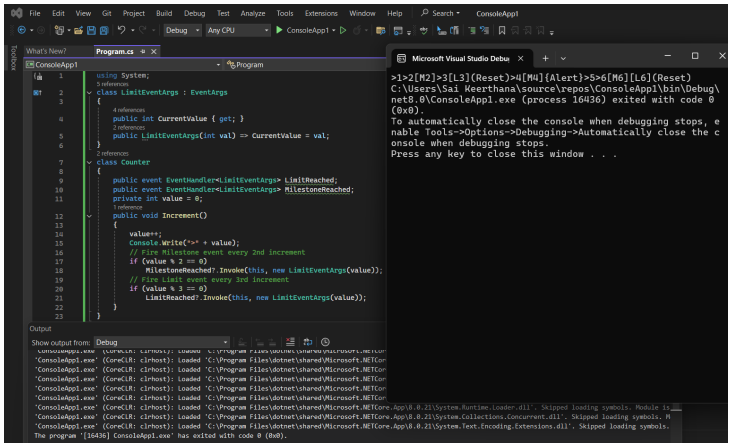
`value = 1, 5:` Prints >1, >5. No events fire.

`value = 2:` Prints >2. Milestone fires → [M2].

`value = 3:` Prints >3. Limit fires → [L3], (Reset).

`value = 4:` Prints >4. Milestone fires → [M4]. The second subscriber checks for $e.CurrentValue == 4$ and prints {Alert}.

`value = 6:` Prints >6. Both events fire. Milestone → [M6]. Limit → [L6], (Reset).



2. **Code 2: Conditional Event Firing**:

```
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:

```
Temperature changed from 30°C to 46°C
 Warning: Sudden change detected!
Temperature changed from 46°C to 52°C
```

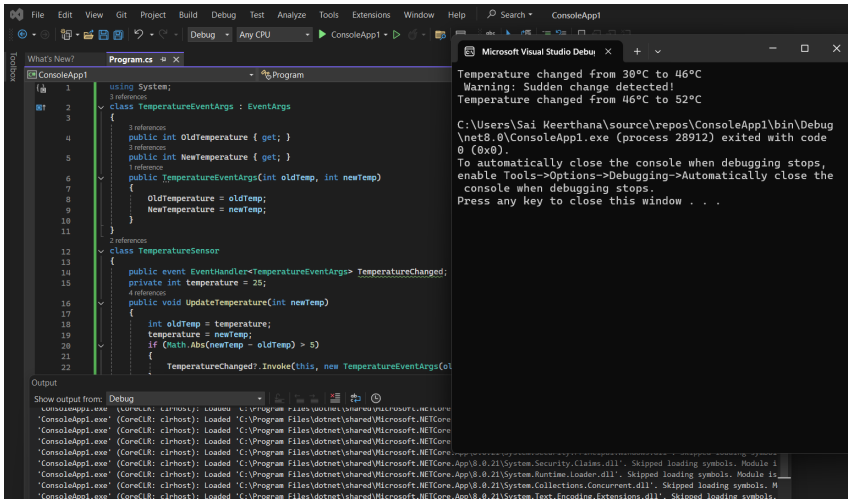
Reasoning: The `TemperatureChanged` event only fires if the absolute change in temperature is greater than 5 degrees (`Math.Abs(newTemp - oldTemp) > 5`). The initial temperature is 25°C.

`UpdateTemperature(28)`: Change is $|28 - 25| = 3$. No event. Current Temp: 28°C.

`UpdateTemperature(30)`: Change is $|30 - 28| = 2$. No event. Current Temp: 30°C.

`UpdateTemperature(46)`: Change is $|46 - 30| = 16$. Event fires. Subscriber 1 prints: Temperature changed from 30°C to 46°C. Subscriber 2 checks: $|46 - 30| = 16$. Since $16 > 10$, it prints: `Warning: Sudden change detected!`.

`UpdateTemperature(52)`: Change is $|52 - 46| = 6$. Event fires. Subscriber 1 prints: Temperature changed from 46°C to 52°C. Subscriber 2 checks: $|52-46| = 6$. Since 6 is not > 10, no warning is printed.



1.4 Output Reasoning Level 2

1. **Code 1: Nested Event Invocation**:

```
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:

```
[Start]{Ping}(Nested)[Start]{Pong}(Nested)[End][End]
```

Reasoning: This demonstrates a recursive, nested event invocation.

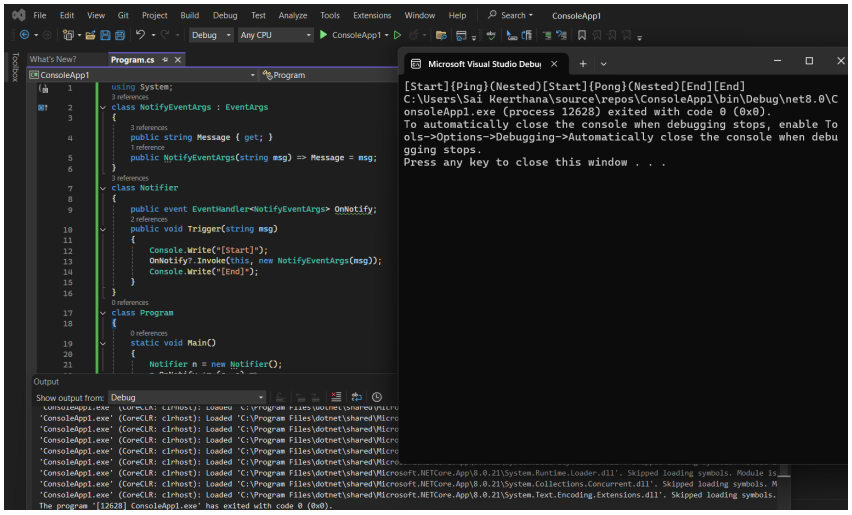
Initial Call: `n.Trigger("Ping")` starts → Prints [Start].

`OnNotify` (with message "Ping") fires: Subscriber 1: Prints {Ping}. Subscriber 2: Prints (Nested). Condition `e.Message == "Ping"` is true, triggering a nested call (`((Notifier)s).Trigger("Pong")`).

Nested Call: `n.Trigger("Pong")` starts → Prints [Start].

`OnNotify` (with message "Pong") fires: Subscriber 1: Prints {Pong}. Subscriber 2: Prints (Nested). Condition `e.Message == "Pong"` is false.

Nested Call completes → Prints [End]. Initial Call completes → Prints [End].



2. **Code 2: Nested Event Invocation with Different Input**:

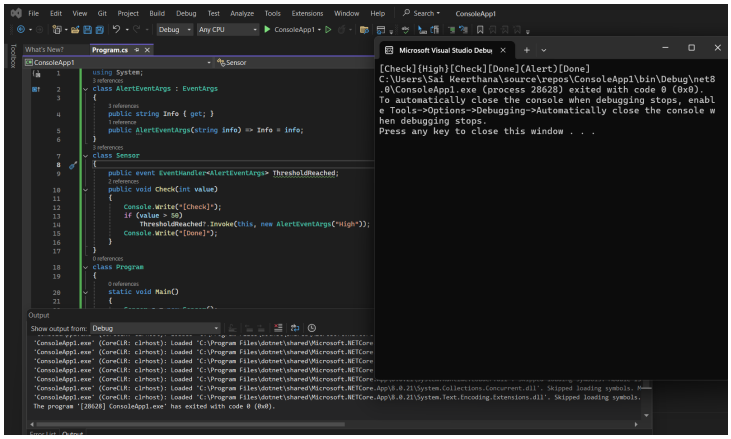
```
using System;
class AlertEventArgs : EventArgs
{
    public string Info { get; }
    public AlertEventArgs(string info) => Info = info;
}
class Sensor
{
    public event EventHandler<AlertEventArgs> ThresholdReached;
    public void Check(int value)
    {
        Console.Write("[Check]");
        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:

```
[Check]{High}[Check][Done](Alert)[Done]
```

Reasoning: The initial call is s.Check(80), which prints [Check] and raises ThresholdReached (Info="High"). Subscriber 1 executes: Prints High. The if (e.Info == "High") condition triggers the nested call s.Check(30). The nested s.Check(30) executes completely: It prints [Check], the condition 30 ¿ 50 is false (no event fire), and it prints [Done]. Execution returns to the main event. Subscriber 2 executes: Prints (Alert). The event invocation completes, and the original s.Check(80) call finishes by printing [Done].



RESULTS AND ANALYSIS

Outputs and Observations: The developed EventPlayground application successfully implemented the custom event and delegate requirements.

Color Change Event: When a color is selected and Change Color is clicked, the ColorChangedEvent is raised. This event successfully triggered two separate subscribers (UpdateLabelColor and ShowNotification), demonstrating the multicast behavior. The label color changed, and a notification dialog appeared.

Text Change Event: Clicking Change Text correctly raised the TextChangedEvent, which updated the lblInfo.Text dynamically to the current date and time.

Theoretical Analysis: The reasoning for the C# code snippets in Tasks 3, 4, and 5 successfully demonstrated understanding of core delegate concepts: the final return value of non-void multicast delegates, the sequential modification of ref parameters in multicast delegates, conditional event firing, and the execution flow of nested event invocations.

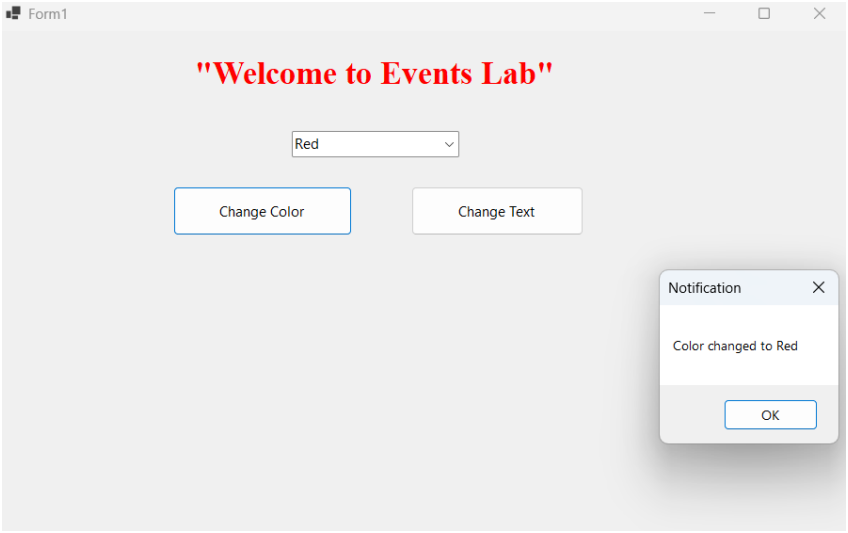


Figure 5: Running application showing label color change for Red selection

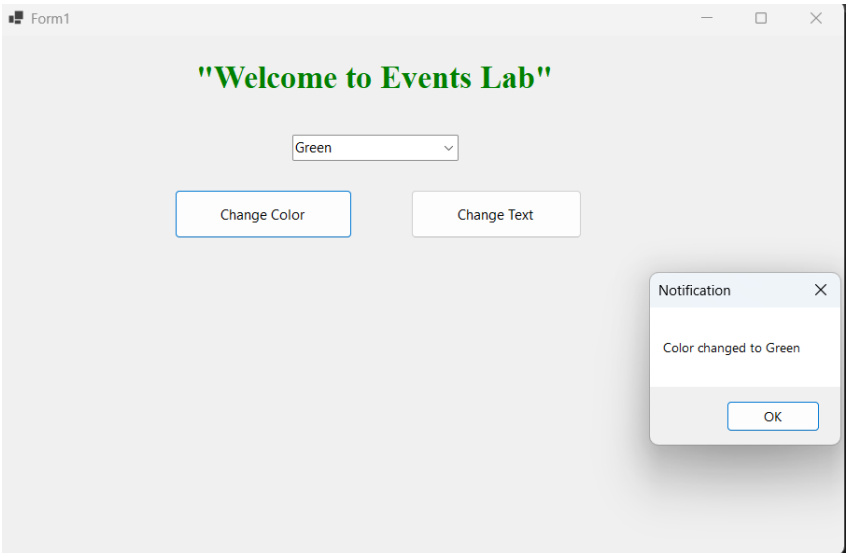


Figure 6: Running application showing label color change for Green selection

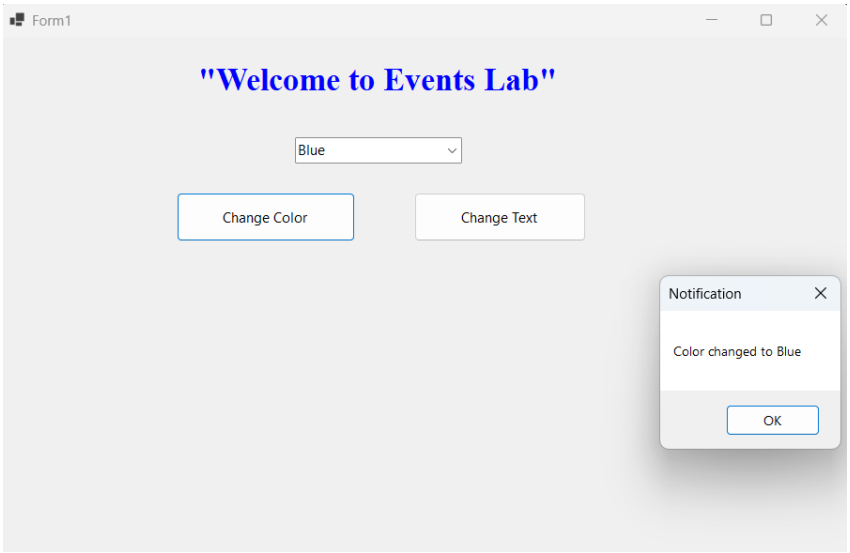


Figure 7: Running application showing label color change forBlue selection

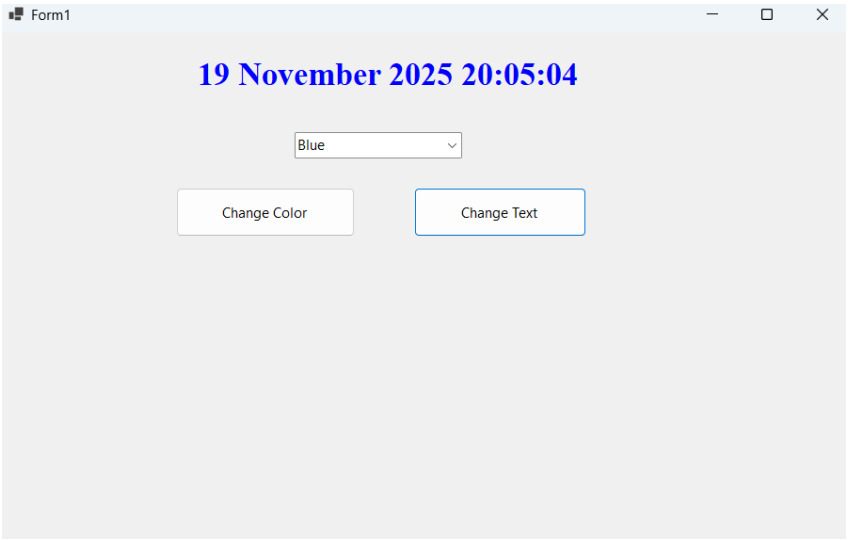


Figure 8: Running application with the label text updated to show current date/time.

Key Insights: Decoupling with Events: Using custom delegates and events separates the publisher (the button click handler logic) from the subscribers (the methods that change the label color or show the notification). This creates modular, reusable code where the publisher doesn't need to know what subscribers do, only that they exist.

Data Transfer: The custom ColorEventArgs class proved crucial for transferring contextual information (the selected color name) from the publisher to the subscribers, which is a fundamental requirement for interactive GUIs.

Multicast Execution: The multicast setup confirmed that C# executes all subscribed methods sequentially and in the order they were added, which is essential for coordination (e.g., updating the label and showing a notification).

DISCUSSION AND CONCLUSION SUMMARY

Challenges and Solutions:

This lab had several practical challenges which we successfully addressed:

- **Multicast Implementation:** Ensured the correct custom delegate (`ColorChanged EventHandler`) was declared and that two distinct methods were explicitly subscribed to the event in the constructor.
- **Data Flow:** Designed the `ColorEventArgs` class to encapsulate the selected color and modified the event raising call to instantiate and pass this object.
- **C# Nullability:** Resolved compiler warnings (CS8600/CS8604) using explicit null-checks (`if (ColorChangedEvent != null)`) or the null-conditional operator (`TextChangedEvent?.Invoke(...)`).
- **Nested Invocation Flow:** The Level 2 problems required precise tracing of execution, recognizing that a nested call fully executes before the original event's remaining subscribers complete.

Lessons Learned and Summary:

This lab reinforced the practical necessity of **event-driven programming** in GUI development. Transitioning from default control events to implementing custom events and delegates provides a highly structured and scalable pattern, especially for complex forms where components need to communicate without tight coupling.

The successful implementation of custom delegates, the publisher-subscriber pattern, custom `EventArgs`, and multicast events confirms that all stipulated learning objectives were achieved. The theoretical analysis further validated a deep understanding of delegate invocation rules, return value handling, and flow control in advanced C# event scenarios.

REFERENCES

1. CS202 Lecture 12 Slides
2. Microsoft Learn – Delegates and Events in C#
3. C# Delegates Overview
4. Windows Forms Events Guide
5. Event-driven Programming Concepts

2 LABORATORY SESSION 12

INTRODUCTION

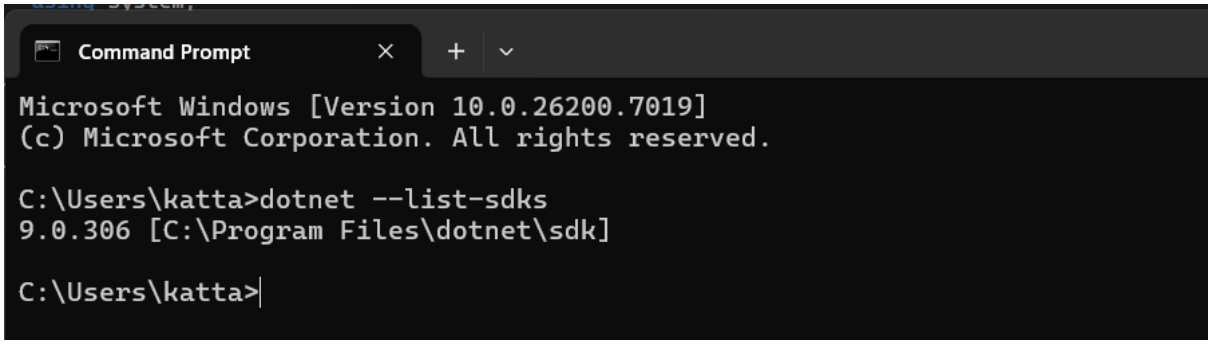
This laboratory session focused on understanding and analyzing C programming concepts related to delegates, callbacks, event handling, exception management, and method overloading through various output reasoning exercises. The primary objective was to evaluate program snippets to determine their compilation validity, the correctness of delegate usage, event-driven method calls, and exception handling outcomes. Additionally, exploration of method overloading behavior and event subscription dynamics was undertaken to deepen proficiency with C language features important for robust .NET application development. This hands-on experience aimed to consolidate theoretical knowledge by engaging with practical C code examples illustrating delegate invocation, event chaining, and control flow using try-catch-finally blocks.

SETUP AND TOOLS

Before starting the lab activities, I ensured that my system was properly configured with all necessary components for effective .NET development. System and Tools Used:

Component	Description / Version
Operating System	Windows 11
Development IDE	Visual Studio 2022 (Community Edition)
.NET SDK	.NET 9.0.306 (compatible with .NET 6 and later)
Programming Language	C# (latest stable version)
Version Control	Git (for project version management)

Visual Studio 2022 was installed with the "`.NET desktop development`" workload, enabling creation and management of C# console applications targeting frameworks from .NET 6 onwards. The .NET SDK was verified via command line using `dotnet --list-sdks` to confirm the installed versions. This configuration provided a robust and modern environment to develop, debug, and run the console projects as required by the lab.



METHODOLOGY AND EXECUTION

Leveraging the Visual Studio IDE, the development workflow involved designing, coding, building, and testing two key Windows Forms applications to implement event-driven order processing workflows using custom events and dynamic event subscription.

- **Code Development:** The initial step involved creating a Windows Forms application named OrderPipeline. Source code was written in Form and supporting class files. Object-oriented programming principles were applied to define custom EventArgs classes encapsulating event data and to implement modular event handlers that model the order workflow with chained events.
- **Build Process:** T The project was built using Visual Studio's integrated build tools. The compiler verified syntax and semantics, highlighting errors and warnings. Successful builds produced executable binaries targeting the appropriate .NET runtime for Windows Forms applications.
- **Execution and Testing:** The application was run within Visual Studio's debugging environment. User inputs were provided via the graphical interface components: TextBox for customer name, ComboBox for product selection, NumericUpDown for quantity, and Buttons for processing and shipping orders. Outputs like label text updates and message boxes were observed to validate the correct event firing and subscriber chaining.
- **Iteration and Refinement:** Based on test results and runtime behavior, the code was iteratively improved to add event filtering (dynamic subscription management) and extend functionalities, such as conditional subscriber addition/removal depending on user input (e.g., express shipping selection).

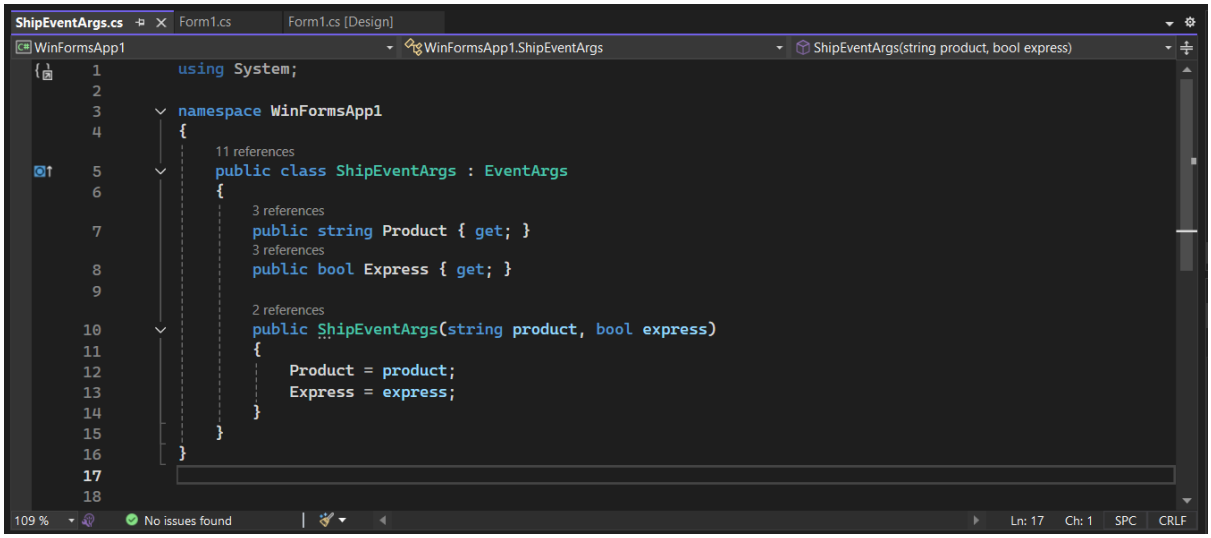
2.1 Windows Forms App – Multi-Stage Event Chaining with Custom EventArgs:

1. Add the Required Controls in Designer

- Toolbox Access: If the Toolbox is not visible, open it using View > Toolbox from the menu bar or press Ctrl+Alt+X.
- Adding Controls:
  - TextBox: Drag from "Common Controls" and drop onto the form. Name: txtCustomerName.
  - ComboBox: Drag and drop; set its Name to cmbProduct. In Properties, set Items to include "Laptop", "Mouse", and "Keyboard".
  - NumericUpDown: Drag and drop; set Name to numQuantity.
  - Button: Drag and drop; set Name to btnProcessOrder, change Text to "Process Order".
  - Label: Drag and drop; set Name to lblStatus, set Text to "Status will be shown here"

2. Create the Custom EventArgs Class Add a new class to the project:

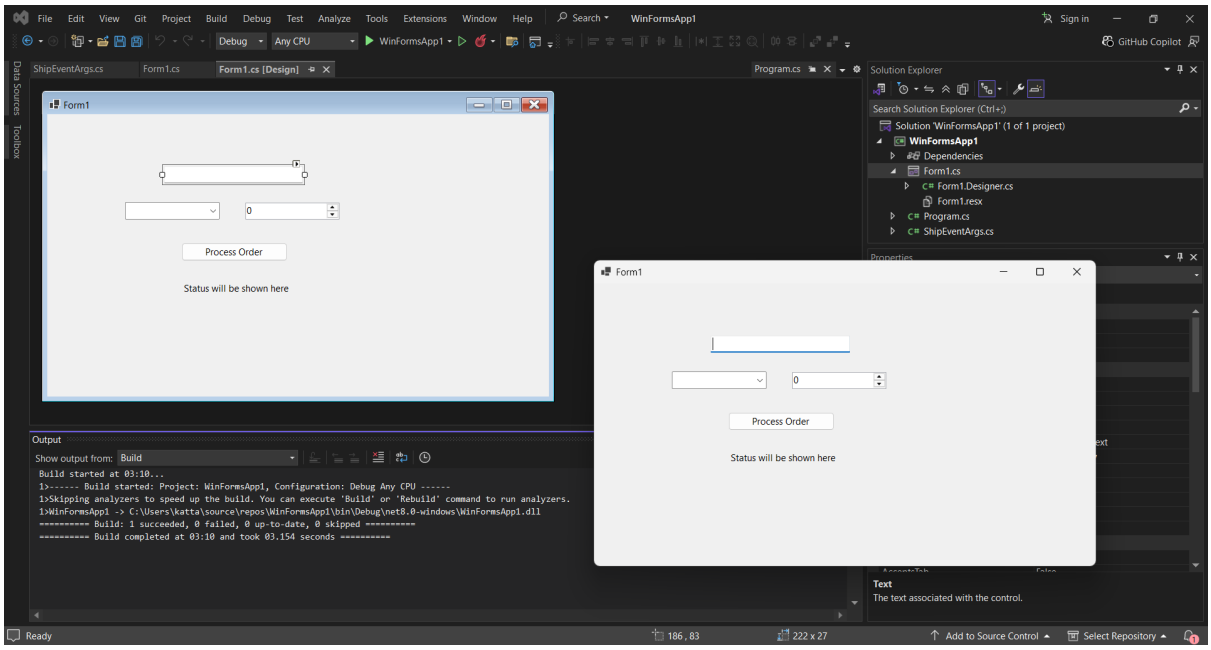
- Right-click on your project in Solution Explorer > *Add* > *Class* > Name it `ShipEventArgs.cs`.
- This defines your custom data for events, as required.



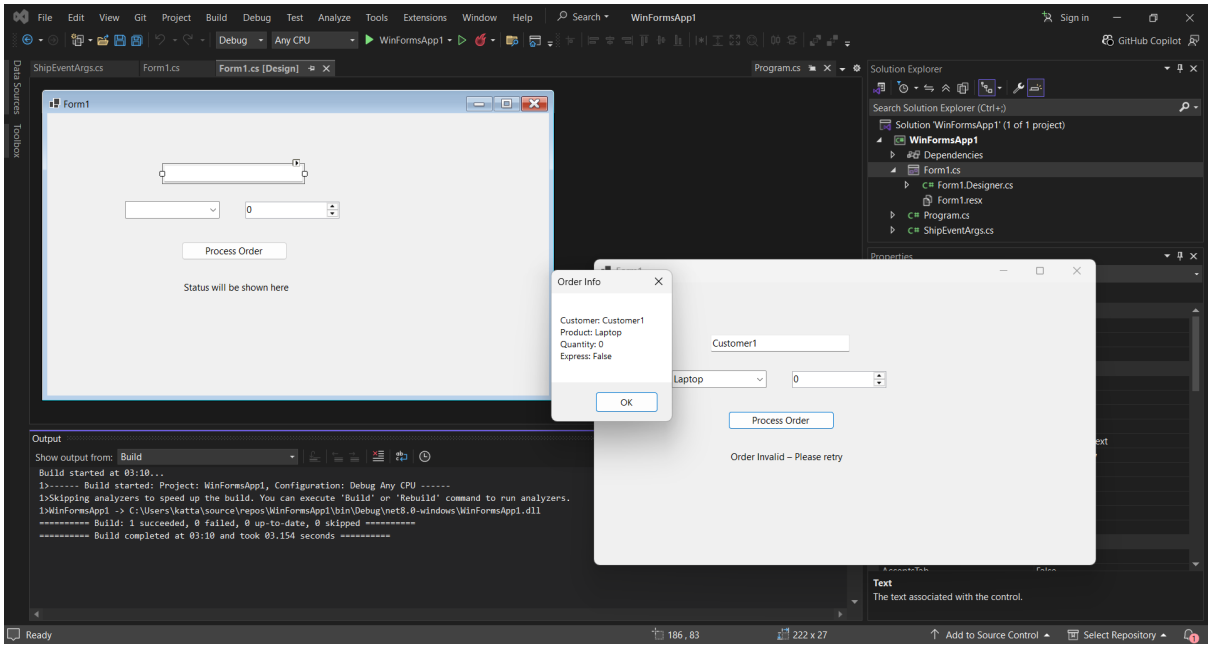
3. Define Events in Form

- This wires up events and links them to handlers.
- Implementing `ValidateOrder()`, `DisplayOrderInfo()`, `ShowRejection()`, `ShowConfirmation()`, the code can be seen in activity 2 where I have added `form.cs` code.

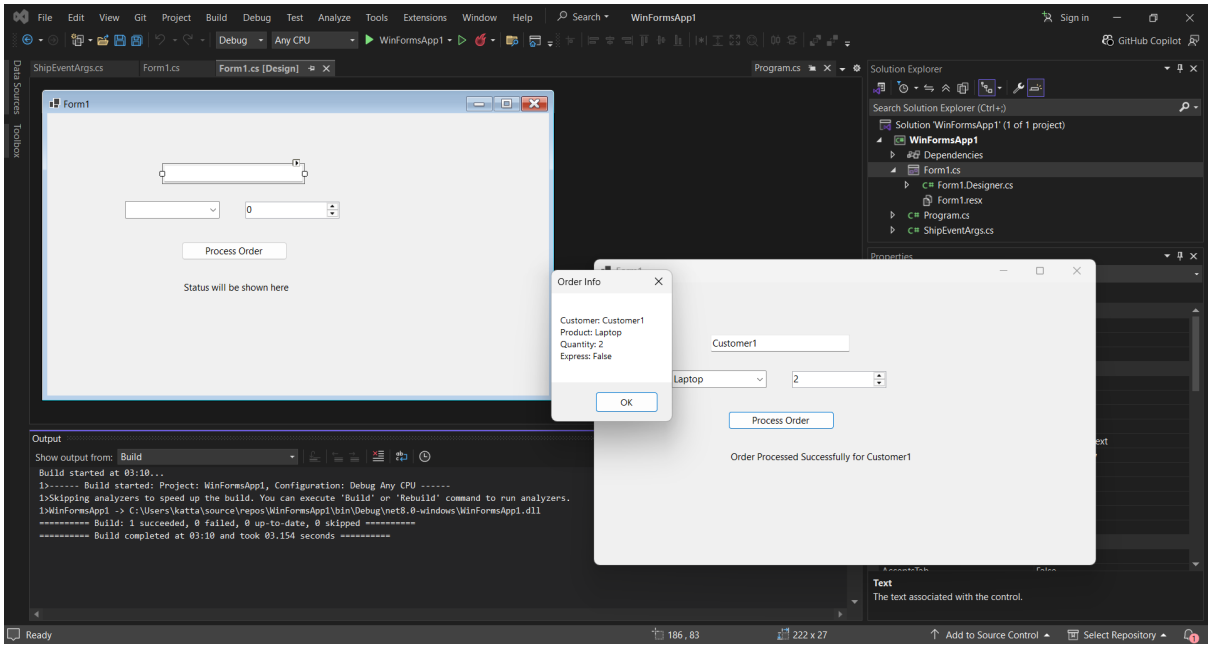
Interface after build:



Placing invalid order with quantity 0:



Placing valid order with quantity $>$ 0:



2.2 Event Filtering and Dynamic Subscriber Management:

To extend our current order processing application with the new features: Add New Controls in Designer:

- Add a CheckBox named `chkExpress` for the Express delivery option.
- Add a new Button named `btnShipOrder` labeled "Ship Order".

Define New Event and EventArgs:

- Define a new event `OrderShipped` using a modified `ShipEventArgs` class that includes Product and Express data.

Implement Dynamic Subscribers for `OrderShipped`:

- Subscriber `ShowDispatch` updates label with dispatch info.
- Subscriber `NotifyCourier` shows a MessageBox only when the `chkExpress` checkbox is checked.
- Dynamically add or remove `NotifyCourier` subscriber based on the checkbox before raising `OrderShipped`.

Track Confirmation Status:

- Maintain a boolean flag `orderConfirmed` to verify that shipping only proceeds if the order was previously confirmed.

Listing 5: Final Form1.cs code:

```
using System;
using System.Windows.Forms;

namespace WinFormsApp1
{
    public partial class Form1 : Form
    {
        // Define custom events
        public event EventHandler<ShipEventArgs> OrderCreated;
        public event EventHandler OrderRejected;
        public event EventHandler OrderConfirmed;
        public event EventHandler<ShipEventArgs> OrderShipped; // New event

        // Track order confirmation status
        private bool orderConfirmed = false;

        public Form1()
        {
            InitializeComponent();

            // Subscribe event handlers
            OrderCreated += ValidateOrder;
            OrderCreated += DisplayOrderInfo;
            OrderRejected += ShowRejection;
            OrderConfirmed += ShowConfirmation;
            btnProcessOrder.Click += BtnProcessOrder_Click;

            // New Ship Order button event
            btnShipOrder.Click += BtnShipOrder_Click;
        }

        private void BtnProcessOrder_Click(object sender, EventArgs e)
        {
            string product = cmbProduct.SelectedItem?.ToString() ?? "";
            bool express = chkExpress.Checked; // Read from checkbox

            ShipEventArgs args = new ShipEventArgs(product, express);
```

```
        // Reset confirmation flag before new processing
        orderConfirmed = false;

        // Raise order created event
        OrderCreated?.Invoke(this, args);
    }

    private void ValidateOrder(object sender, ShipEventArgs e)
    {
        if (numQuantity.Value > 0)
        {
            lblStatus.Text = "Validated";
            orderConfirmed = true;
            OrderConfirmed?.Invoke(this, EventArgs.Empty);
        }
        else
        {
            orderConfirmed = false;
            OrderRejected?.Invoke(this, EventArgs.Empty);
        }
    }

    private void DisplayOrderInfo(object sender, ShipEventArgs e)
    {
        string customer = txtCustomerName.Text;
        int quantity = (int)numQuantity.Value;
        string summary = $"Customer: {customer}\nProduct: {e.Product}\
           nQuantity: {quantity}\nExpress: {e.Express}";
        MessageBox.Show(summary, "Order Info");
    }

    private void ShowRejection(object sender, EventArgs e)
    {
        lblStatus.Text = "Order Invalid - Please retry";
    }

    private void ShowConfirmation(object sender, EventArgs e)
    {
        lblStatus.Text = $"Order Processed Successfully for {
           txtCustomerName.Text}";
    }

    // --- New handlers for OrderShipped event

    private void BtnShipOrder_Click(object sender, EventArgs e)
    {
        if (!orderConfirmed)
        {
            MessageBox.Show("Cannot ship: order not confirmed.", "
               Shipping Error", MessageBoxButtons.OK, MessageBoxIcon.
               Warning);
            return;
        }

        string product = cmbProduct.SelectedItem?.ToString() ?? "";
        bool express = chkExpress.Checked;
        ShipEventArgs args = new ShipEventArgs(product, express);
```

```
            // Dynamic subscription management for NotifyCourier
            if (express)
            {
                // Subscribe NotifyCourier only if checked
                OrderShipped += NotifyCourier;
            }
            else
            {
                // Unsubscribe NotifyCourier if unchecked
                OrderShipped -= NotifyCourier;
            }

            // Always subscribe ShowDispatch
            // To avoid duplicate subscriptions, unsubscribe first then
               subscribe
            OrderShipped -= ShowDispatch;
            OrderShipped += ShowDispatch;

            // Raise OrderShipped event
            OrderShipped?.Invoke(this, args);
        }

        private void ShowDispatch(object sender, ShipEventArgs e)
        {
            lblStatus.Text = $"Product dispatched: {e.Product}";
        }

        private void NotifyCourier(object sender, ShipEventArgs e)
        {
            if (e.Express)
            {
                MessageBox.Show("Express delivery initiated!", "Courier
                   Notification", MessageBoxButtons.OK, MessageBoxIcon.
                   Information);
            }
        }

        // Auto-generated or placeholder methods below
        private void Form1_Load(object sender, EventArgs e) { }
        private void textBox1_TextChanged(object sender, EventArgs e) { }
        private void cmbProduct_SelectedIndexChanged(object sender,
           EventArgs e) { }
        private void lblStatus_Click(object sender, EventArgs e) { }
    }
}
```

Listing 6: Final ShipEventArgs.cs code:

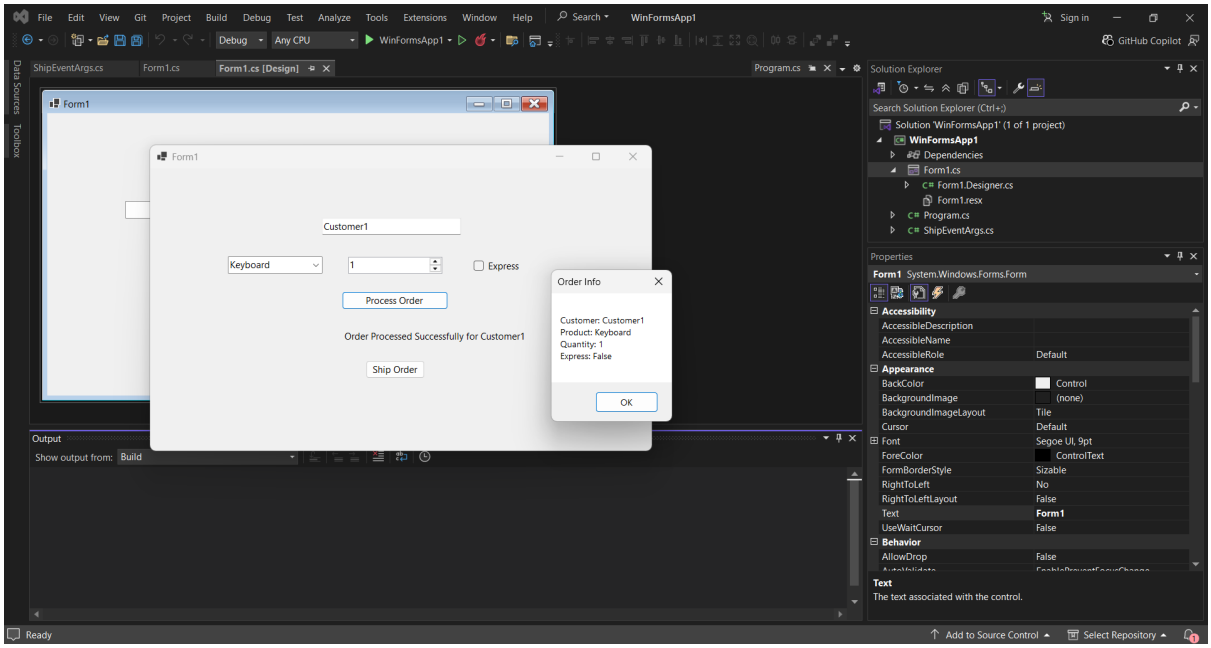
```
using System;

namespace WinFormsApp1
{
    public class ShipEventArgs : EventArgs
    {
        public string Product { get; }
        public bool Express { get; }
```

```
        public ShipEventArgs(string product, bool express)
        {
            Product = product;
            Express = express;
        }
    }
}
```

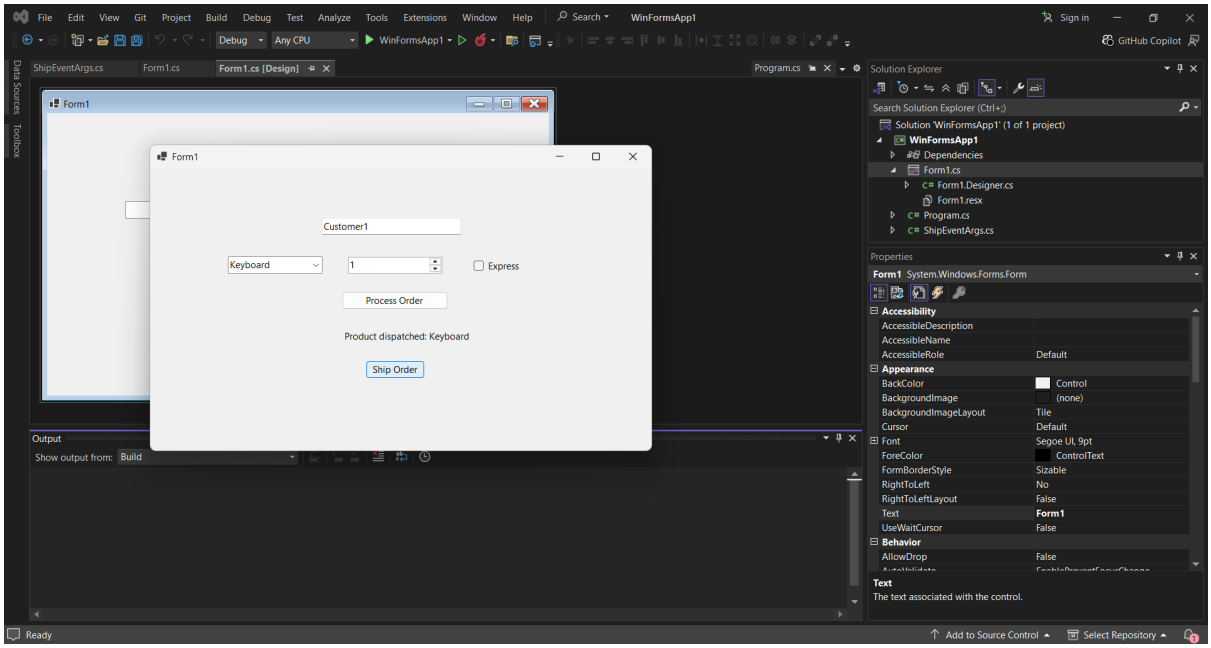
OUTPUTS:

Regular (Non-Express) Shipping Path

- Placing order

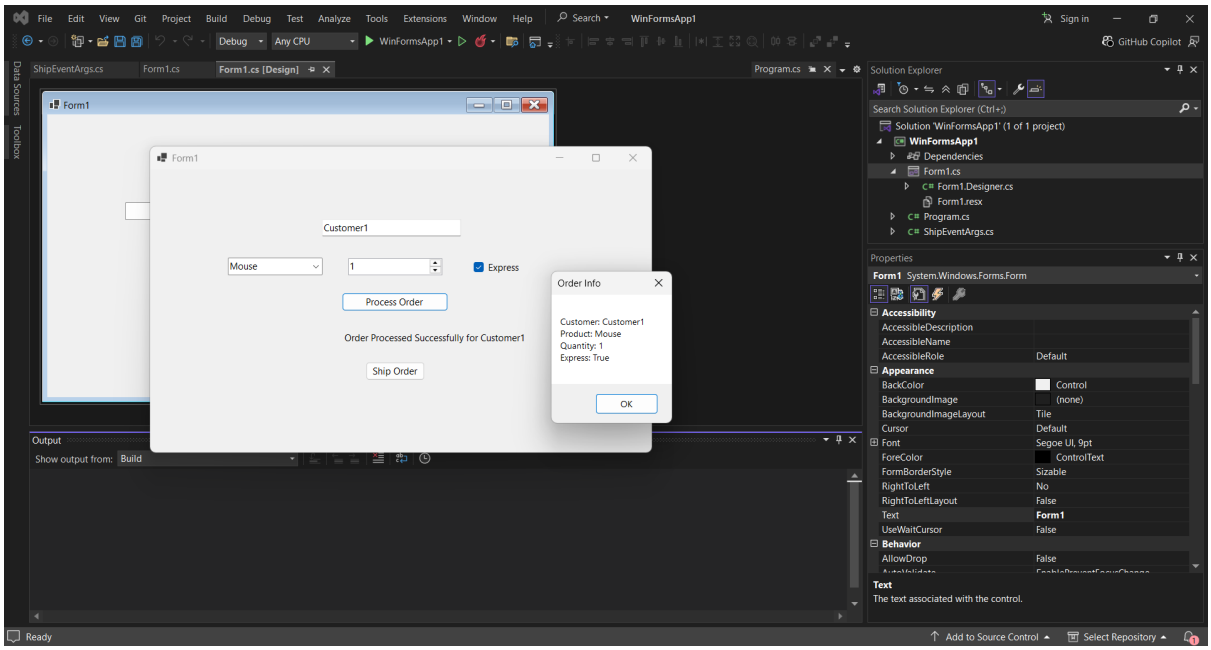


- Shipping order

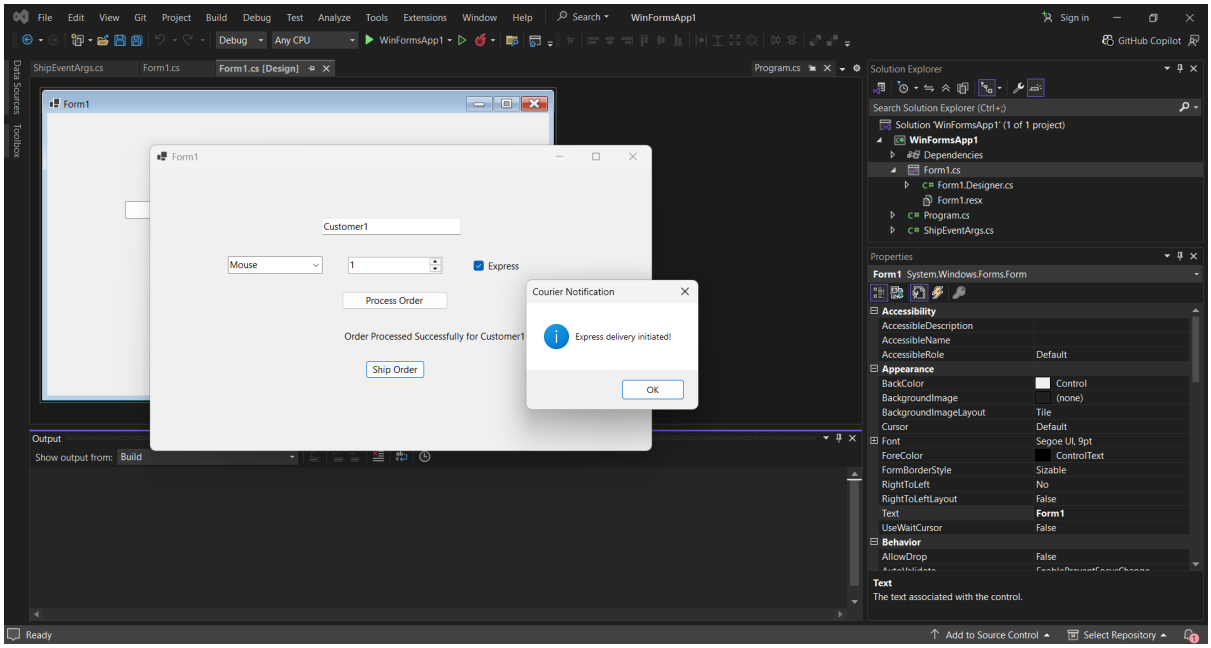


Express Shipping Path

- Placing order



- Shipping order

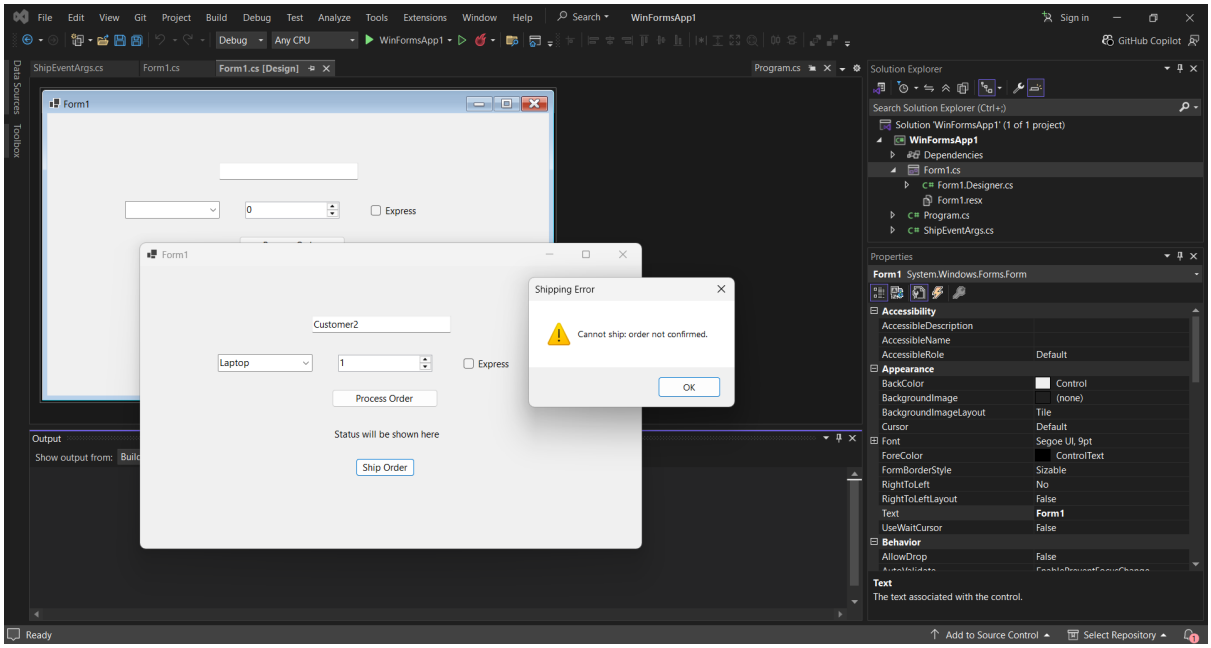


Dynamic Subscriber Addition/Removal

- When chkExpress is checked, the NotifyCourier subscriber is added at ship time (via OrderShipped += NotifyCourier;).
- When unchecked, it is removed (via OrderShipped -= NotifyCourier;) before the event.
- The code dynamically manages the subscription each time Ship Order is pressed.
- There are no duplicate notifications if you toggle the checkbox between shipments

Prevent Unconfirmed Orders from Shipping

- Trying to ship without placeorder



2.3 Output Reasoning (Level 0)

1. **Code:**

```
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:

```
N/A
```

Explanation: This code does not compile. Due to below Compilation Errors:
1. Delegate Signature Mismatch:

   - AuthCallback expects a method with a bool parameter: void Method(bool validUser).
   - The method Login() is declared as public static void Login() (no parameters), which does not match the delegate's required signature.

2. Incorrect Use of Top-Level Statements:

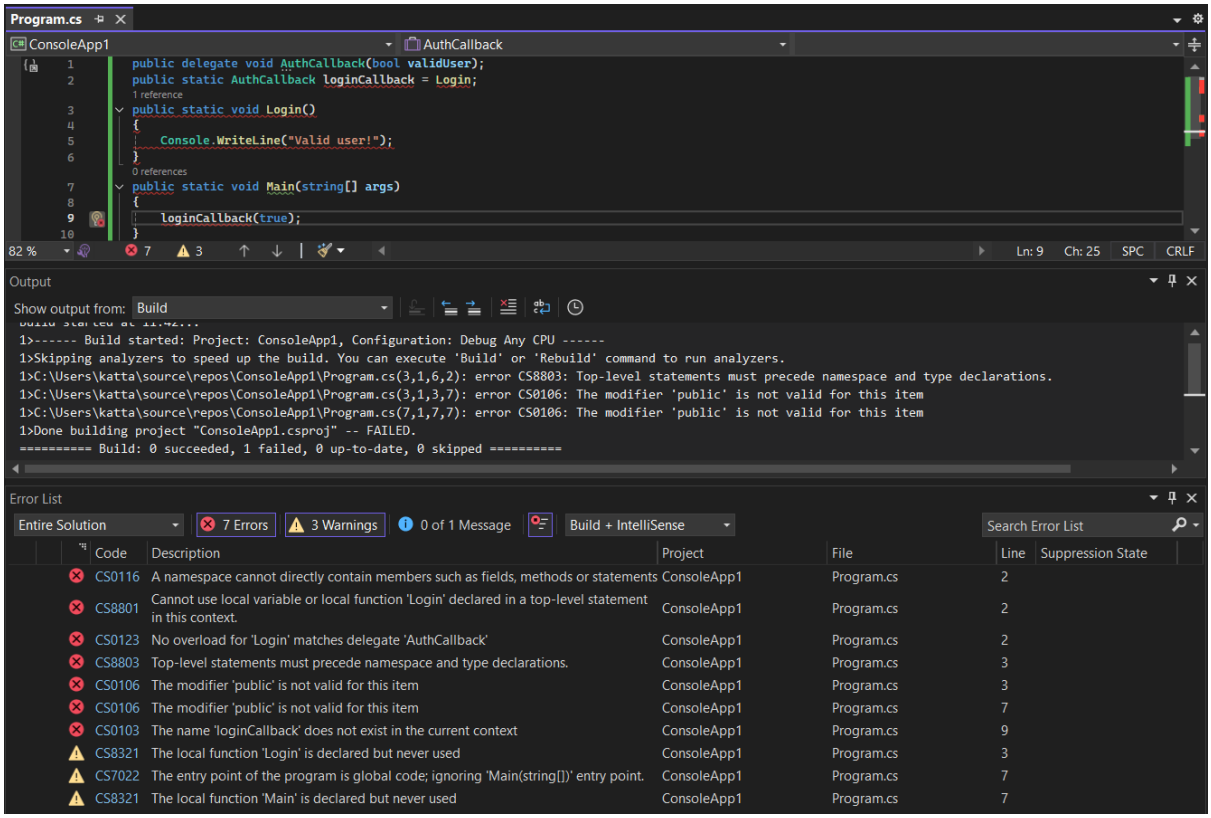
   - C# expects namespaces and class declarations; methods cannot be declared at the top level in traditional C syntax.

3. Public Modifier Not Valid for This Context:

   - The code uses public static modifiers directly in top-level, which isn't valid outside of a class declaration.

4. Other Context Errors:

   - These stem from the lack of a proper class or namespace block.



2. **Code:**

```
using System;
using System.Numerics;
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:

```
A: hello
B: HELLO
A: world
B: WORLD
```

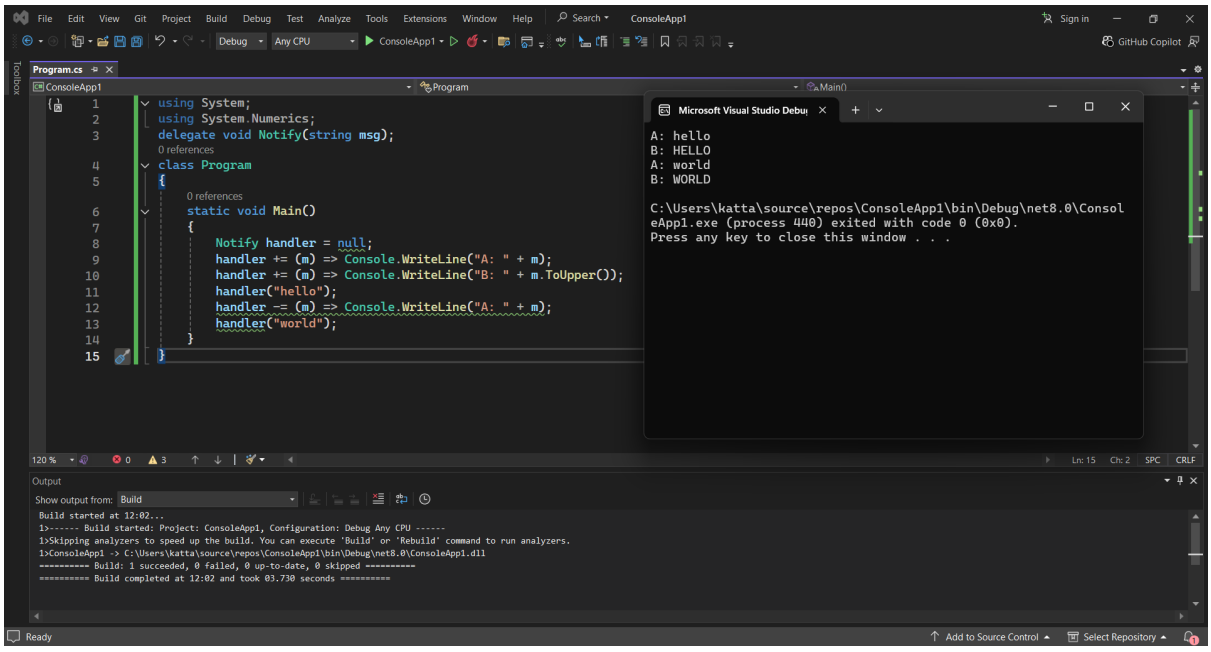
Explanation: 1. First handler invocation:

- handler(”hello”);
- Calls both attached methods:

  - Prints "A: hello"
  - Prints "B: HELLO"

2. Attempted removal of handler:
   - handler -= (m) =¿ Console.WriteLine("A: " + m);
   - Lambda expressions without explicit variable references create new delegates each time, so this statement does NOT remove the earlier lambda. Thus, both handlers are still present.
3. Second handler invocation:
   - handler("world");
   - Again calls both attached methods:
     - Prints "A: world"
     - Prints "B: WORLD"

The code prints both lines for each call because removing the handler via a lambda does not actually remove the previously-added delegate, so both remain in the invocation list for both calls.



2.4 Output Reasoning (Level 1)

1. **Code:**

```
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:

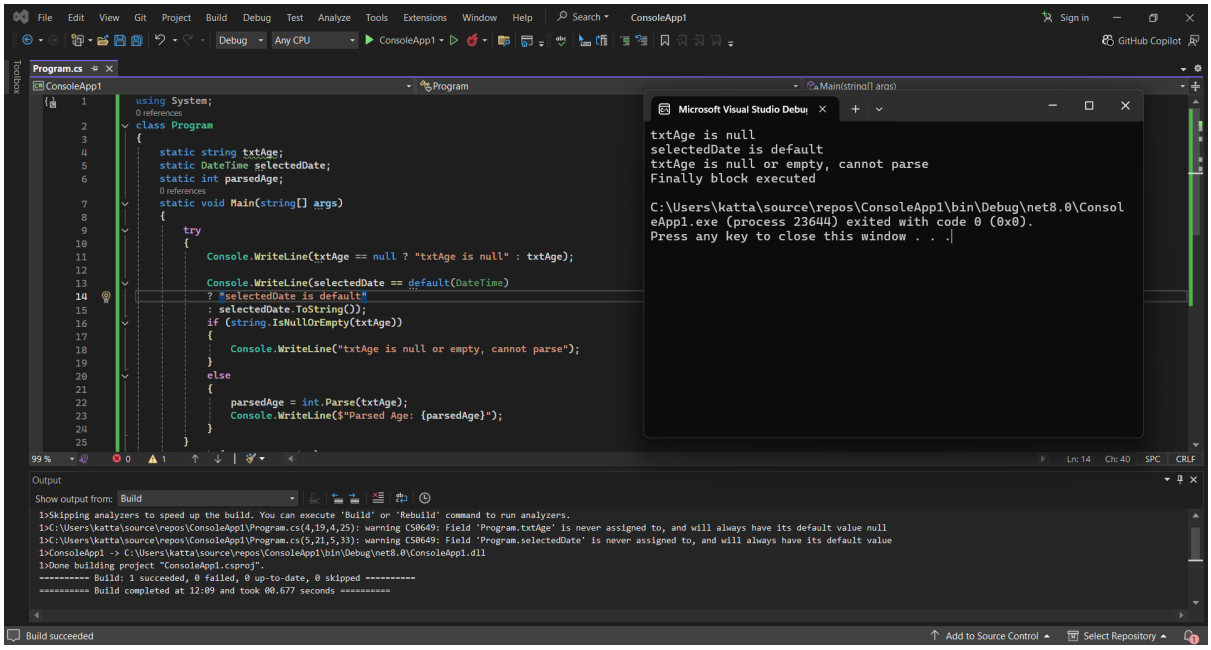
```
txtAge is null
selectedDate is default
txtAge is null or empty, cannot parse
Finally block executed
```

Explanation:

- txtAge is a static string field that is never assigned a value, so its value is null by default.
- selectedDate is a static DateTime field that is never assigned, so its value is default(DateTime) (i.e., DateTime.MinValue).
- In Main:
  - The first line checks if txtAge == null. This prints "txtAge is null".
  - The next line checks if selectedDate == default(DateTime). This prints "selectedDate is default".
  - The code then checks if (string.IsNullOrEmpty(txtAge)). Since txtAge is null, this is true, so it prints "txtAge is null or empty, cannot parse".

- The else block is not executed, so parsedAge = int.Parse(txtAge) is not attempted.
- No exceptions are thrown, so no catch blocks run.
- Finally, the finally block prints "Finally block executed".

No parsing is attempted, so no exceptions are caught. The output comes from simple value checks and the finally block.



2. **Code:**

```
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()
```

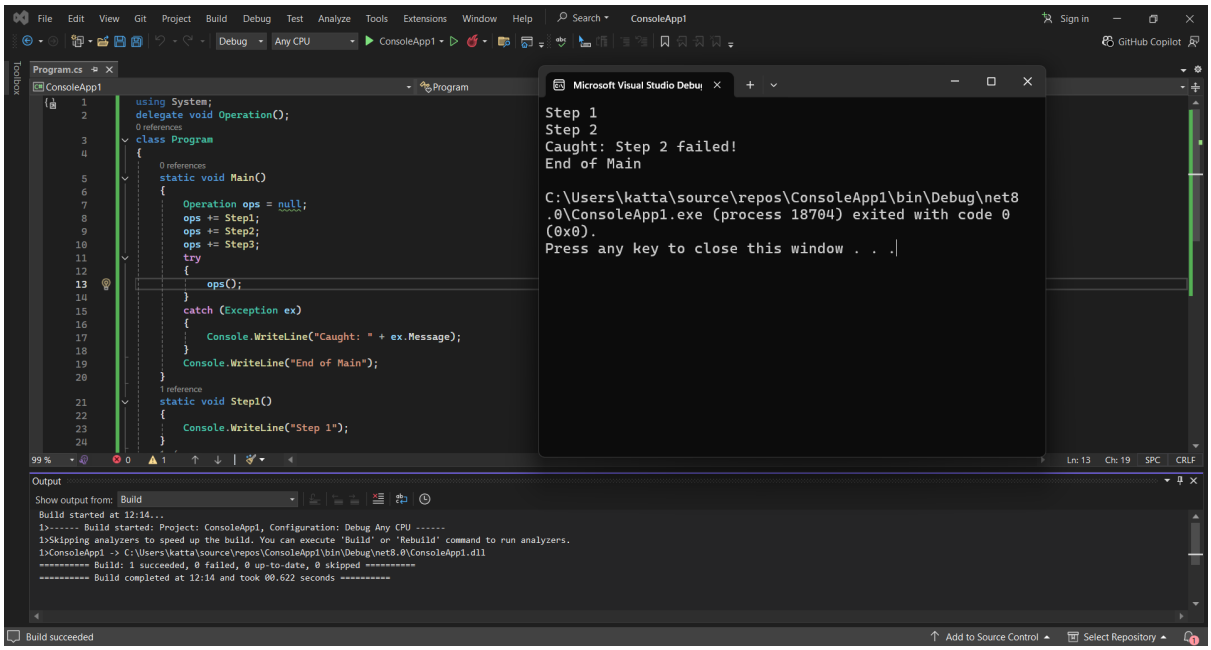
```
    {
        Console.WriteLine("Step 2");
        throw new InvalidOperationException("Step 2 failed!");
    }
    static void Step3()
    {
        Console.WriteLine("Step 3");
    }
}
```

Output:

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

Explanation:

- The delegate ops has three methods attached: Step1, Step2, and Step3.
- Invoking ops() will run them in order:
  - Step1 prints "Step 1".
  - Step2 prints "Step 2" and then throws an exception.
  - The exception interrupts further invocation, so Step3 is never called.
- The try-catch in Main catches the exception from Step2 and prints: Caught: Step 2 failed!
- After catching, the program prints: End of Main



2.5 Output Reasoning (Level 2)

1. **Code:**

```
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)");
        }
    }
}
```

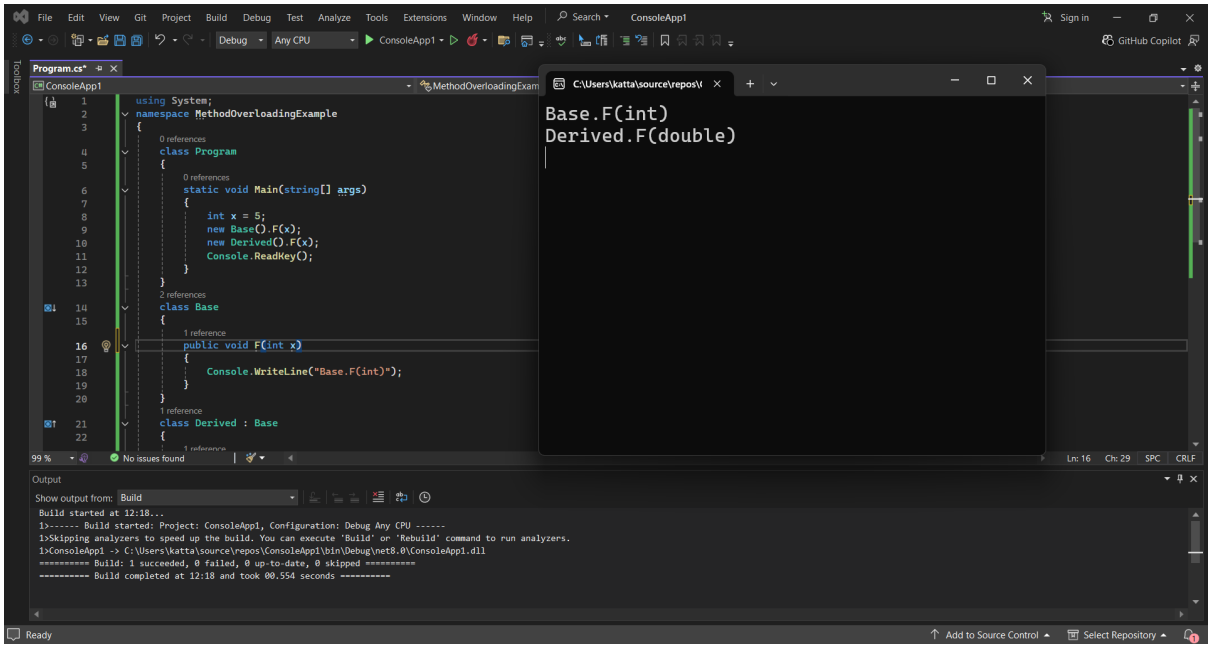
Output:

```
Base.F(int)
Derived.F(double)
```

Explanation:

- In Main, x is an integer 5.
- new Base().F(x); calls the F(int x) method of Base, so prints: Base.F(int)
- new Derived().F(x); is more interesting:
  - Although Derived inherits from Base, it does not override F(int) but instead overloads F by providing F(double).
  - When calling F(x) on a Derived instance with x as an integer, the compiler looks for the best method match in Derived first.
  - Derived has a method F(double x). The integer argument is implicitly converted to double, so this overload is chosen.
  - This means prints: Derived.F(double)

This demonstrates method overloading (not overriding) and type promotion in method resolution.



2. **Code:**

```
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:

```
<S1>[1]<C1><S2>[2]<C2>(Dyn2)<S3>[3]<C3>(Dyn3)
```

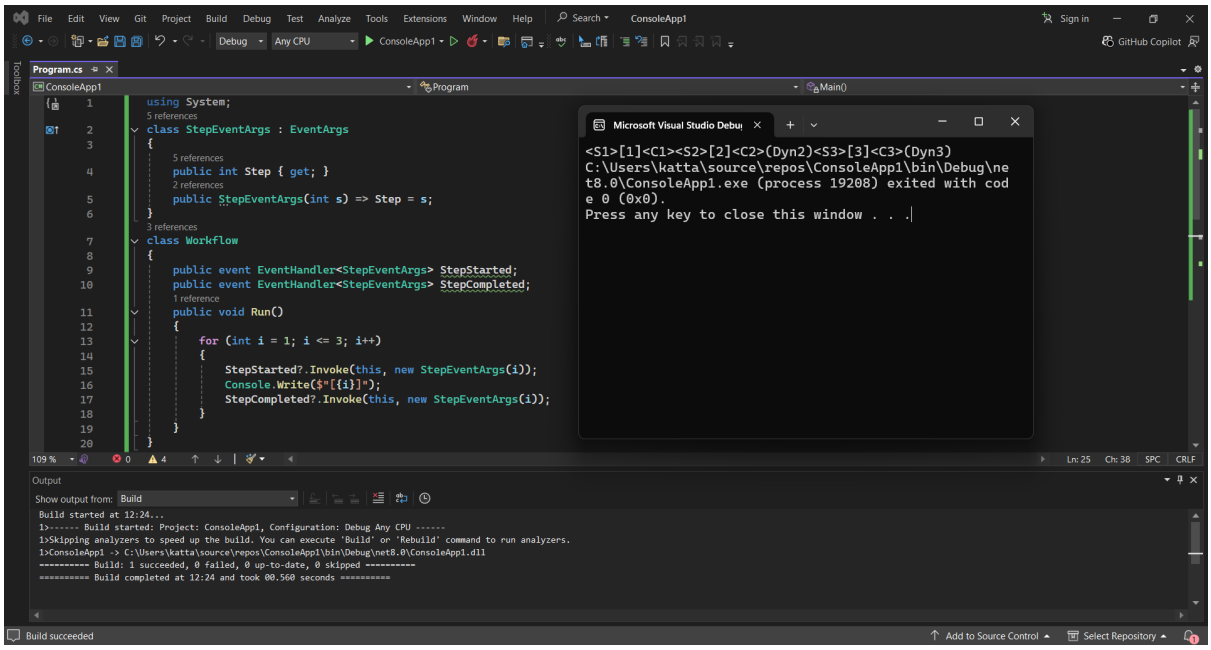
Explanation: Let's step through each iteration of the loop in Run() and see what happens, observing how event subscriptions change dynamically: Before Loop Starts

- StepStarted:
  - Writes ¡S¿ for current step.
  - At step 2, adds another handler to StepCompleted that writes (Dyn) for current step.
- Initial StepCompleted:
  - Always writes ¡C¿ for current step.

Loop Unrolling

- For i = 1
  - StepStarted fires: ¡S1¿
  - Print [1]
  - StepCompleted fires: ¡C1¿
- For i = 2
  - StepStarted fires: ¡S2¿
    Inside handler, since e.Step == 2, dynamically adds another handler to StepCompleted to print (Dyn).
  - Print [2]
  - StepCompleted fires:
    - Old handler: ¡C2¿
    - Newly added handler: (Dyn2)
- For i = 3
  - StepStarted fires: ¡S3¿
  - Print [3]
  - StepCompleted fires:
    - Old handler: ¡C3¿
    - Newly added handler: (Dyn3) (added in the previous iteration, still attached!)

Dynamic event subscription on step 2 means any further invocations of StepCompleted (steps 2, 3) will fire both handlers and print tags accordingly.



DISCUSSION

The set of lab activities provided a comprehensive, hands-on exploration of C programming concepts in both Windows Forms and console applications. Through these exercises, several advanced topics were tackled, including event-driven programming, custom delegate and event management, exception handling, and method overloading.

The Windows Forms tasks focused on modeling real-world workflows, such as order processing, by leveraging custom events and EventArgs subclasses. Implementing multi-stage event chaining and subscriber methods fostered a deeper understanding of how event-driven architectures enable modularity and maintainable code. The extension task on event filtering and dynamic subscriber management demonstrated the flexibility of .NET's event system—allowing runtime attachment and detachment of event handlers to selectively respond to user input (for example, enabling express-courier logic only when required).

In the console-based reasoning questions, concepts were reinforced through direct code execution and output analysis. These exercises addressed how delegates operate, how method resolution works in the context of inheritance and overloading, and the practical impact of exception handling when multiple methods are invoked via a multicast delegate. With dynamic event subscription, students saw firsthand how handlers could be attached in response to program state and events, and how these changes affect output.

Additionally, the debugging and output reasoning tasks emphasized careful attention to compilation errors, type signatures, and runtime logic. The importance of understanding C's rules on method matching, delegate invocation lists, and exception propagation became clear, as did the subtleties of the language's handling of null/default variable initialization and branching.

CONCLUSION

By methodically implementing and testing these lab tasks, the objectives of strengthening C fundamentals and deepening understanding of event-driven and object-oriented programming principles were achieved. Students gained practical knowledge of:

- Designing modular programs using events, delegates, and custom EventArgs.
- Dynamically managing event subscribers to adapt program behavior at runtime.
- Debugging and analyzing program output to ensure correctness and refine control flow.
- Using inheritance and method overloading to create flexible program designs.

These exercises laid the groundwork for further exploration in application development, building confidence in using Visual Studio's tooling, structuring code for maintainability, and leveraging C# and .NET language features. The skills acquired in dynamic logic, error diagnosis, and UI-event-model thinking are fundamental for tackling more complex scenarios in both desktop and enterprise programming contexts.

REFERENCES

1. CS202 Lecture 12, 13 Slides
2. Object.Equals Method
   `https://learn.microsoft.com/en-us/dotnet/api/system.object.equals?view=net-9.0`
3. Introduction to delegates and events in C#
   `https://learn.microsoft.com/en-us/dotnet/csharp/delegates-overview`
4. How to combine delegates (Multicast Delegates)]
   `https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/how-to-combine-delegates-multicast-delegates`