CS202

Assignment-4

Kovid Parmar(23110172)
Aniruddh Reddy(23110195)

Contents

Code- https://github.com/kovidparmar/STT_Assignment_4

LAB 11

Introduction

This lab assignment focuses on Events and Delegates within C# Windows Forms Applications. The primary objective is to deepen the understanding of event handling by creating, subscribing to, and invoking custom events using the publisher-subscriber model.

We developed a Windows Forms application named EventPlayground to demonstrate modular and interactive GUI design. The application utilizes two custom events, ColorChangedEvent and TextChangedEvent, to manage functionality across multiple controls. Crucially, we extended the core functionality to implement multicast event handling using a custom ColorEventArgs class, allowing a single event to trigger multiple subscribers simultaneously.

Setup

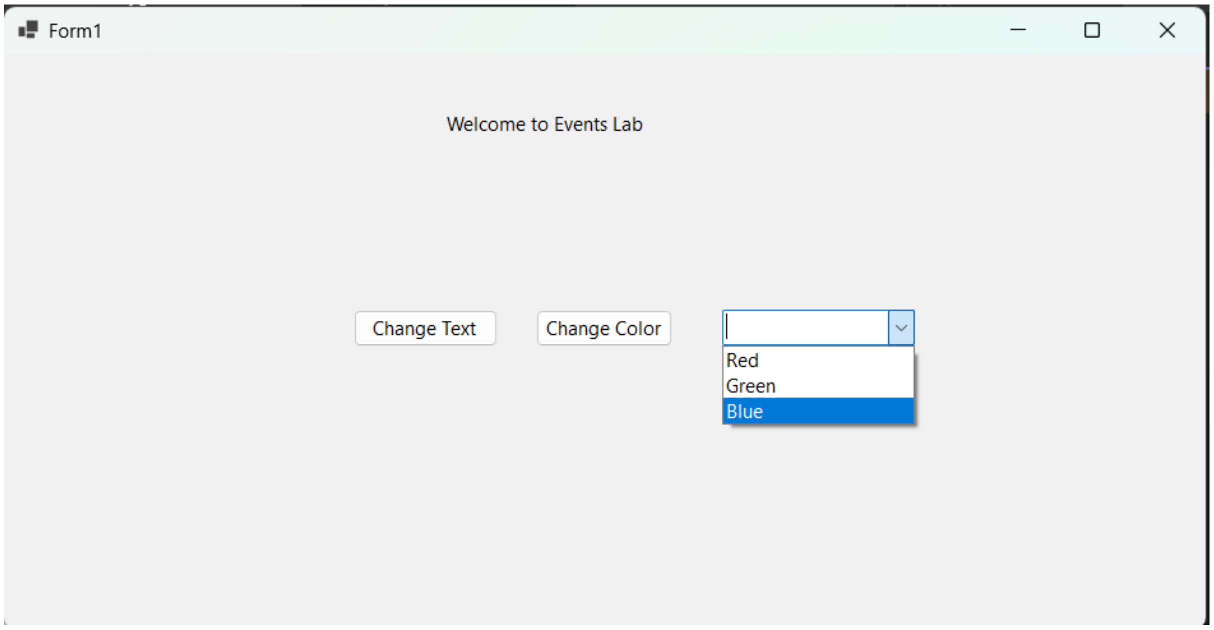
To begin Lab 11, the setup starts with creating the fundamental application structure in Visual Studio. Open **Visual Studio 2022** and initiate a new project by selecting the **Windows Forms App (.NET)** template using C#. Finally, name the application **EventPlayground** and confirm its creation to generate the main Form1.cs file, which serves as the foundation for the entire graphical user interface and subsequent event handling logic.

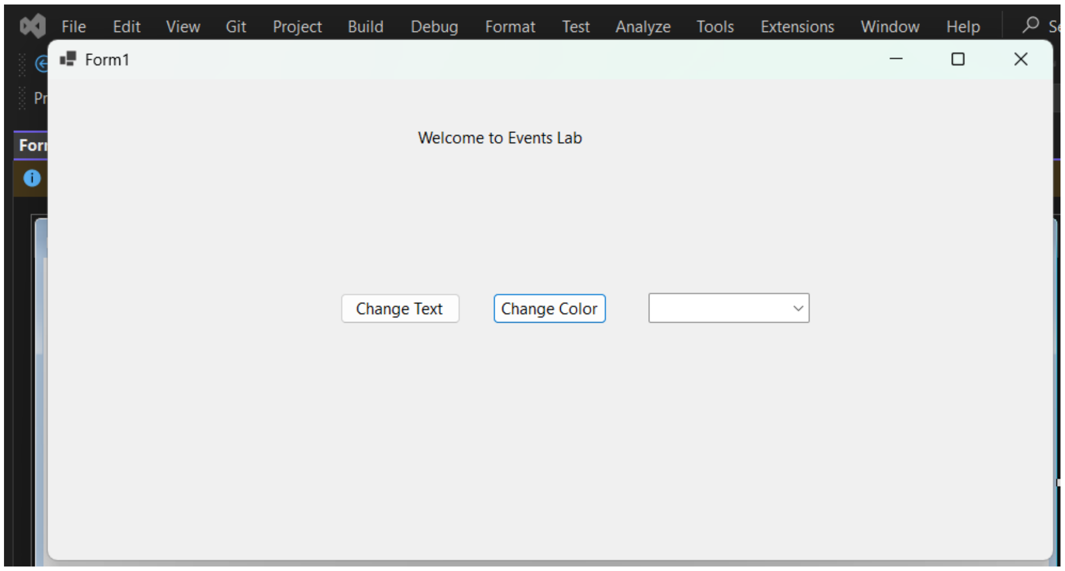
Tools

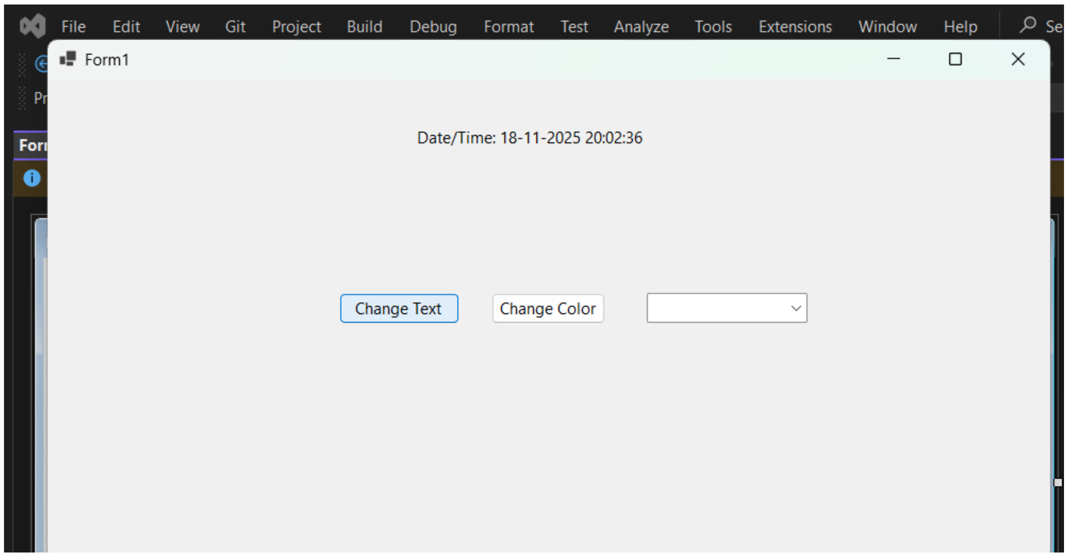
The required integrated development environment (IDE) for this lab is **Visual Studio 2022 (Community Edition)**. This tool will be used for setting up the new C# Windows Form app, writing the code, and executing the programs to test the implemented logic.

Methodology and Execution

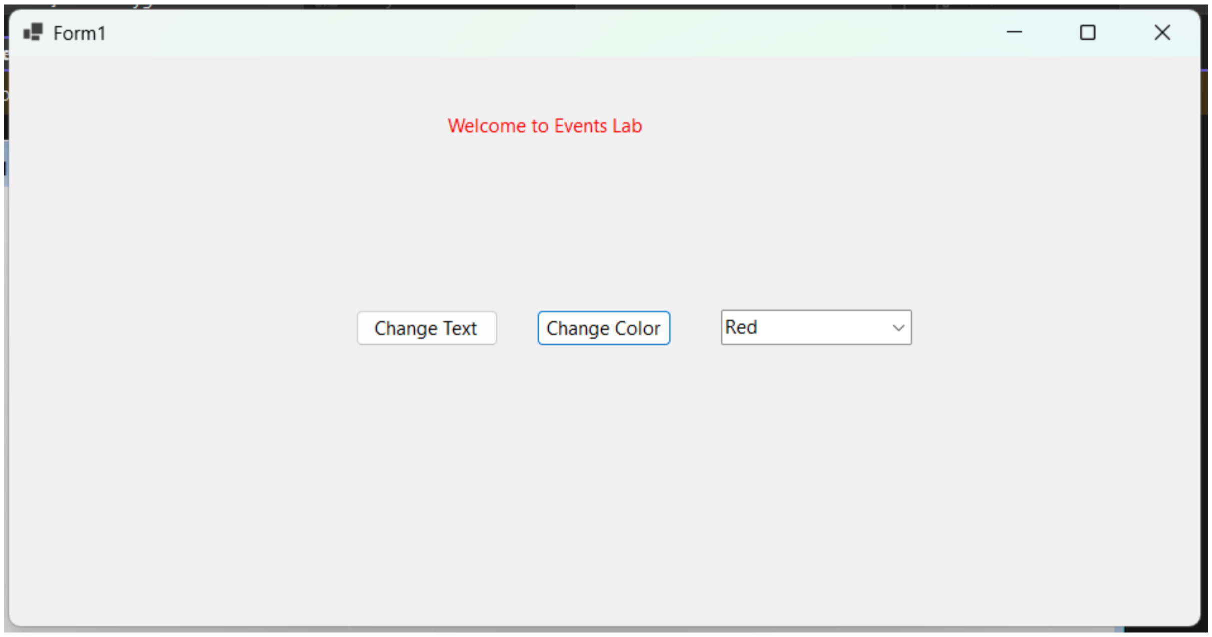
Windows Forms App – Multi-Control Event Interaction:

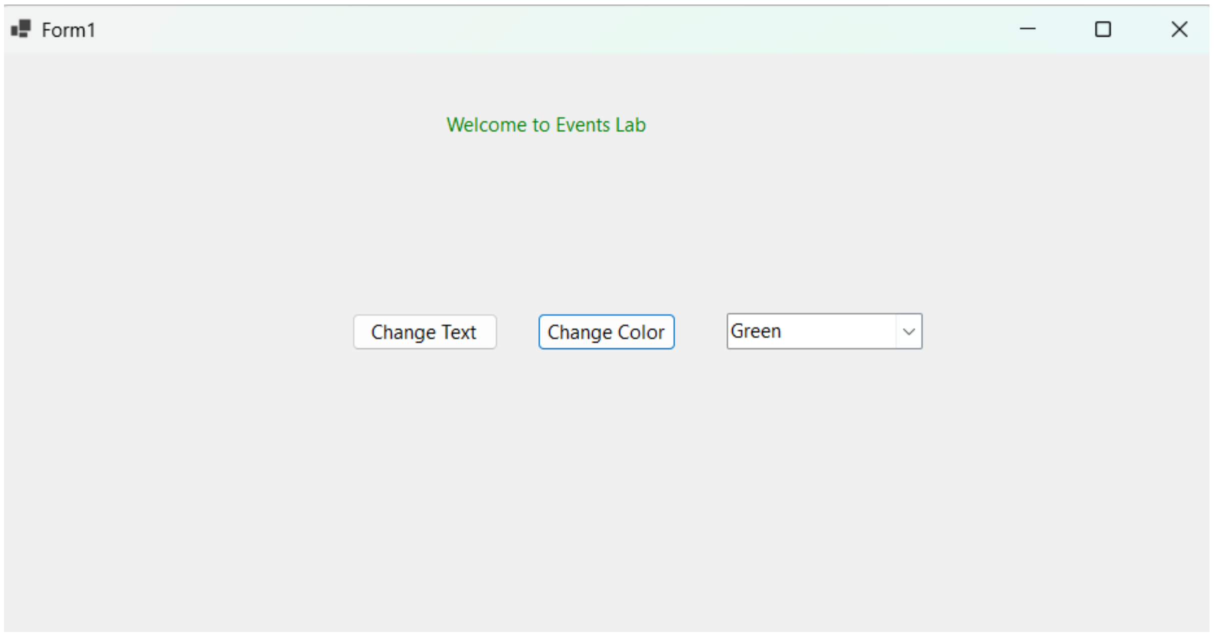

Form1
Welcome to Events Lab
Change Text
Change Color
Red
Green
Blue

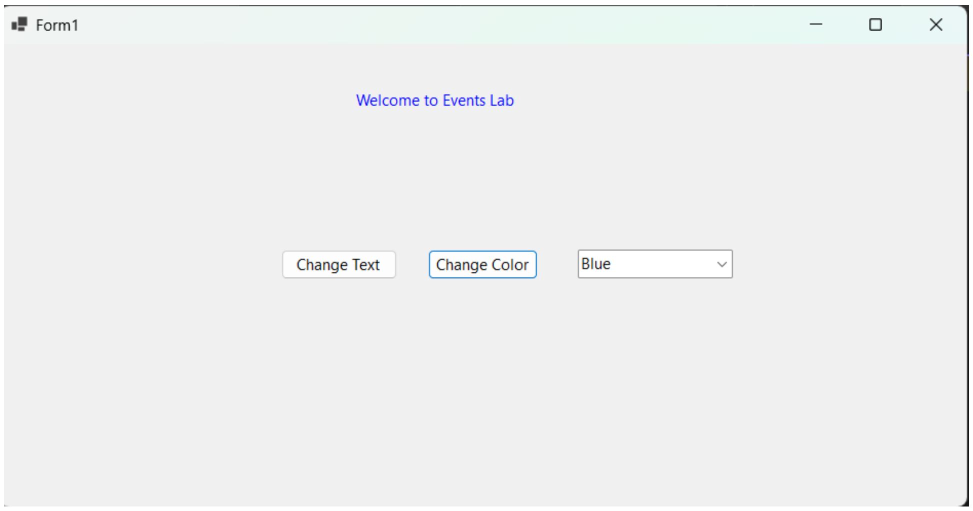

File
Edit
View
Git
Project
Build
Debug
Format
Test
Analyze
Tools
Extensions
Window
Help
Form1
Welcome to Events Lab
Change Text
Change Color

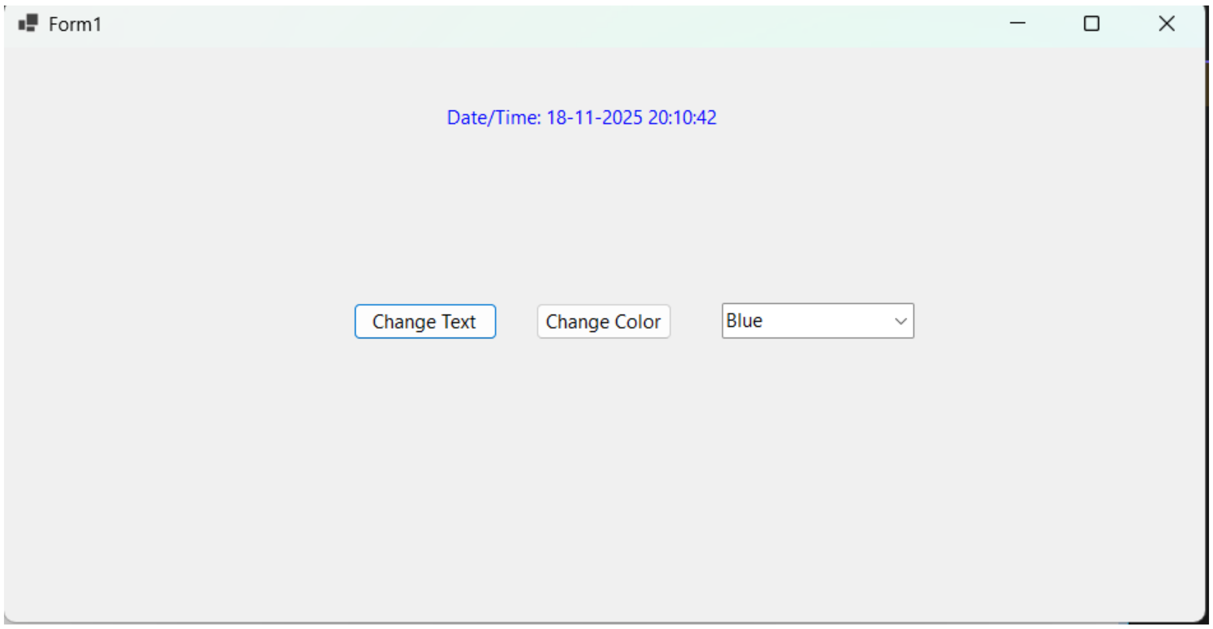


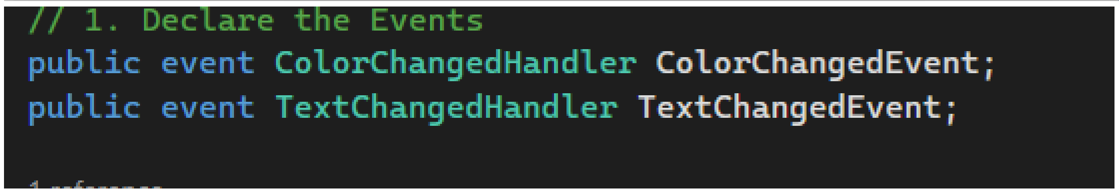
Date/Time: 18-11-2025 20:02:36


Form1
Welcome to Events Lab
Change Text
Change Color
Red

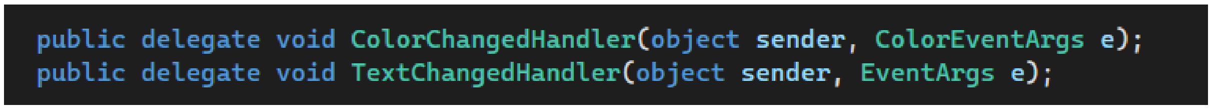

Form1
Welcome to Events Lab
Change Text
Change Color
Green





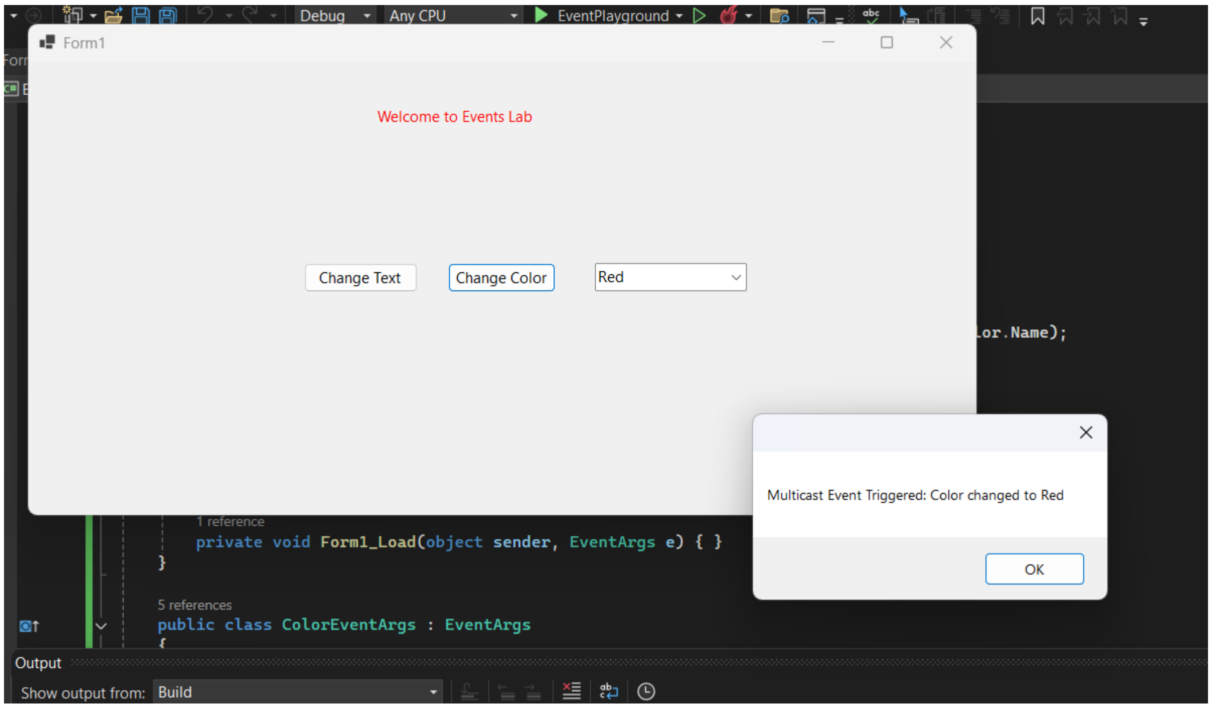


```
// 1. Declare the Events
public event ColorChangedHandler ColorChangedEvent;
public event TextChangedHandler TextChangedEvent;
```



```
public delegate void ColorChangedHandler(object sender, ColorEventArgs e);
public delegate void TextChangedHandler(object sender, EventArgs e);
```

Using EventArgs and Multiple Subscribers:

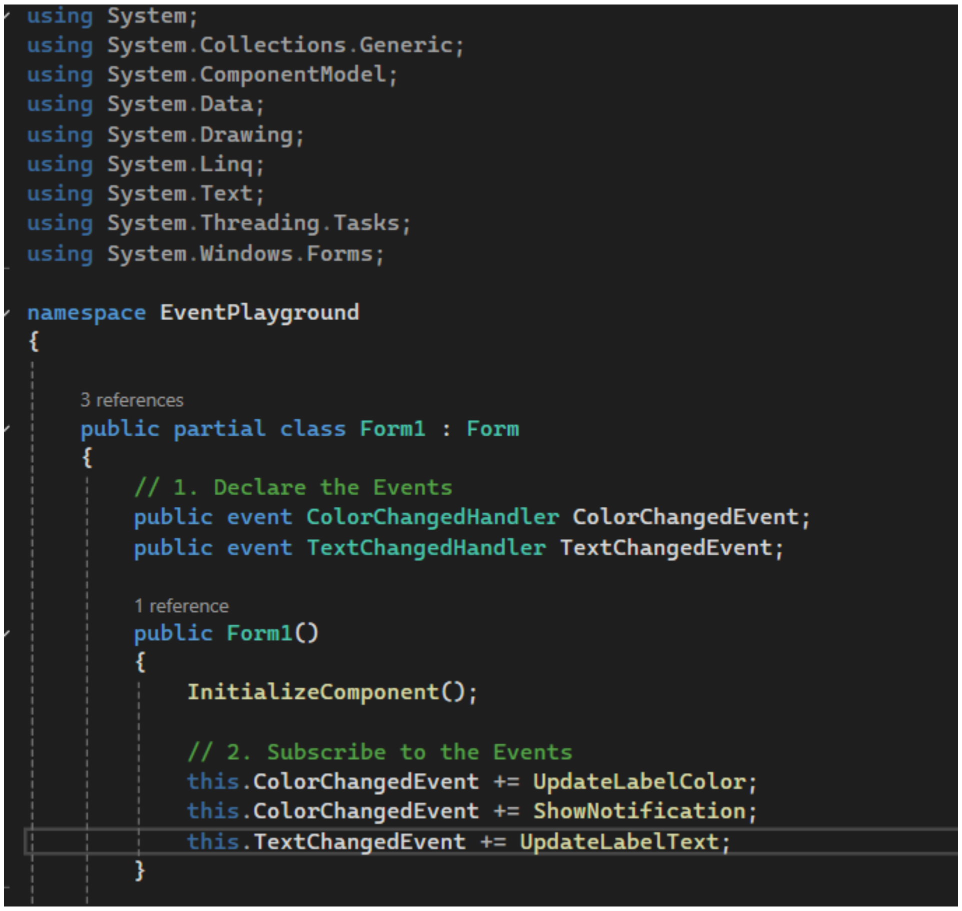


Welcome to Events Lab

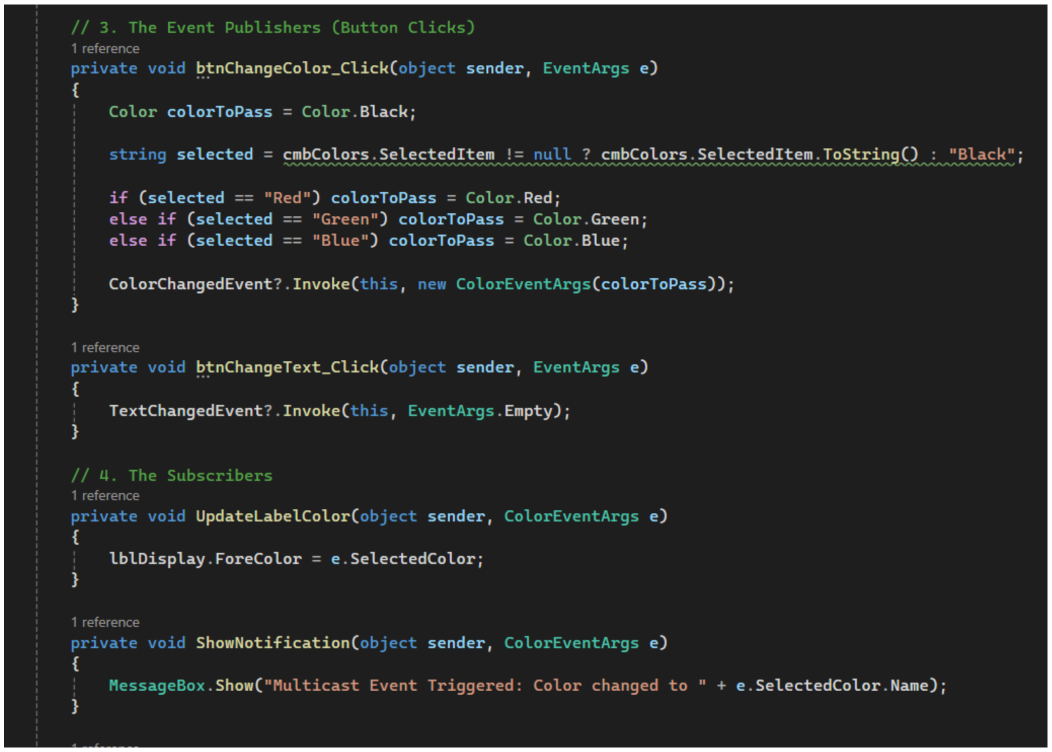
Multicast Event Triggered: Color changed to Red

```
or.Name);
        1 reference
        private void Form1_Load(object sender, EventArgs e) { }
    }

    5 references
    public class ColorEventArgs : EventArgs
    {
```

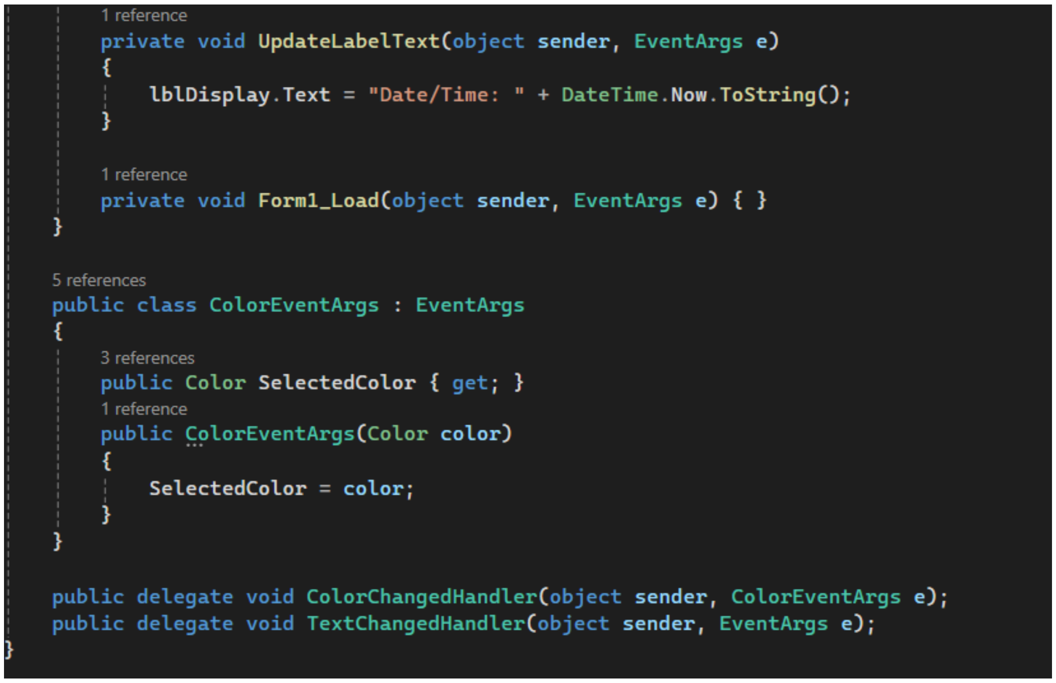
code :



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace EventPlayground
{

    3 references
    public partial class Form1 : Form
    {
        // 1. Declare the Events
        public event ColorChangedHandler ColorChangedEvent;
        public event TextChangedHandler TextChangedEvent;

        1 reference
        public Form1()
        {
            InitializeComponent();

            // 2. Subscribe to the Events
            this.ColorChangedEvent += UpdateLabelColor;
            this.ColorChangedEvent += ShowNotification;
            this.TextChangedEvent += UpdateLabelText;
        }
```



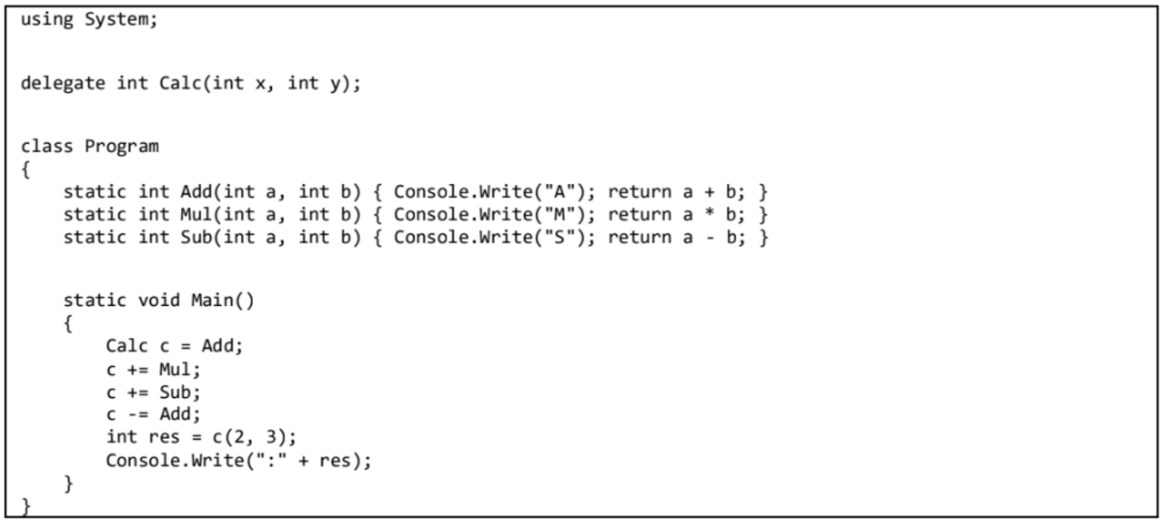
```
        // 3. The Event Publishers (Button Clicks)
        1 reference
        private void btnChangeColor_Click(object sender, EventArgs e)
        {
            Color colorToPass = Color.Black;

            string selected = cmbColors.SelectedItem != null ? cmbColors.SelectedItem.ToString() : "Black";

            if (selected == "Red") colorToPass = Color.Red;
            else if (selected == "Green") colorToPass = Color.Green;
            else if (selected == "Blue") colorToPass = Color.Blue;

            ColorChangedEvent?.Invoke(this, new ColorEventArgs(colorToPass));
        }

        1 reference
        private void btnChangeText_Click(object sender, EventArgs e)
        {
            TextChangedEvent?.Invoke(this, EventArgs.Empty);
        }

        // 4. The Subscribers
        1 reference
        private void UpdateLabelColor(object sender, ColorEventArgs e)
        {
            lblDisplay.ForeColor = e.SelectedColor;
        }

        1 reference
        private void ShowNotification(object sender, ColorEventArgs e)
        {
            MessageBox.Show("Multicast Event Triggered: Color changed to " + e.SelectedColor.Name);
        }
```



```
        1 reference
        private void UpdateLabelText(object sender, EventArgs e)
        {
            lblDisplay.Text = "Date/Time: " + DateTime.Now.ToString();
        }

        1 reference
        private void Form1_Load(object sender, EventArgs e) { }
    }

    5 references
    public class ColorEventArgs : EventArgs
    {
        3 references
        public Color SelectedColor { get; }
        1 reference
        public ColorEventArgs(Color color)
        {
            SelectedColor = color;
        }
    }

    public delegate void ColorChangedHandler(object sender, ColorEventArgs e);
    public delegate void TextChangedHandler(object sender, EventArgs e);
}
```

Output Reasoning (Level 0)

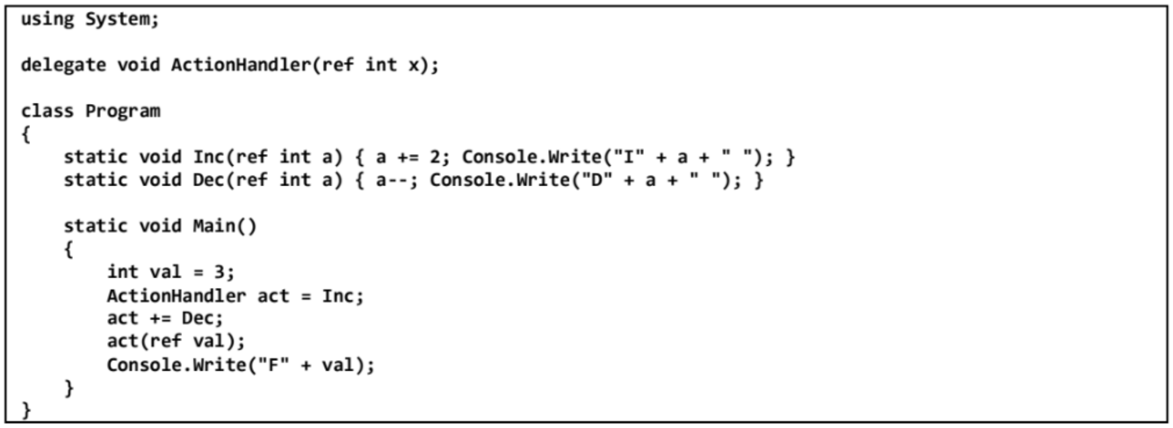
What will be the output of the following C# code? Why?



```
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);
    }
}
```

This code demonstrates how **multicast delegates** work in C# by combining and subtracting methods before calling them. When the delegate c is created, the methods Add, Mul, and Sub are added to its list in that order. The code then specifically removes Add from the list , leaving only **Mul** followed by **Sub** to be executed. When c(2, 3) is called, the console first prints **"M"** (from Mul) and then **"S"** (from Sub). The crucial rule for multicast delegates is that the final return value (res) is always the result from the **last method** executed, which is Sub(2, 3), returning **-1**. Therefore, the total output is the combined console writes followed by the final result: **MS:-1**

What will be the output of the following C# code? Why?



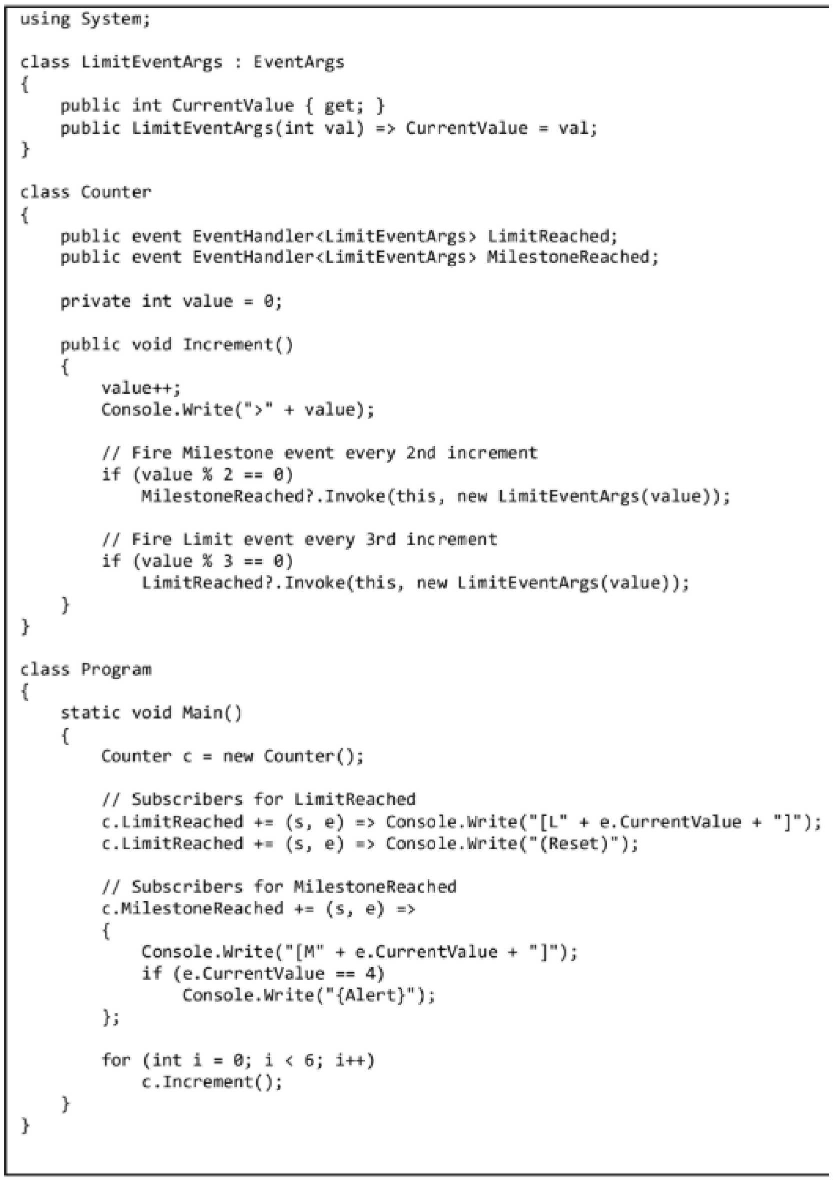
```
using System;

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);
    }
}
```

This code demonstrates the sequence of execution in a multicast delegate, showing how a single call can change a variable passed by reference (ref) multiple times.

- The variable val starts at 3.
- The delegate act contains two methods: **Inc** followed by **Dec**.
- When act(ref val) is called, **Inc** executes first, incrementing val by 2 (to 5) and printing **I5** .
- Next, **Dec** executes using the new value (5), decrementing val by 1 (to 4) and printing **D4** .
- The final line prints **F** followed by the variable's final value, which is 4.

The combined output is **I5 D4 F4**.

Output Reasoning (Level 1)

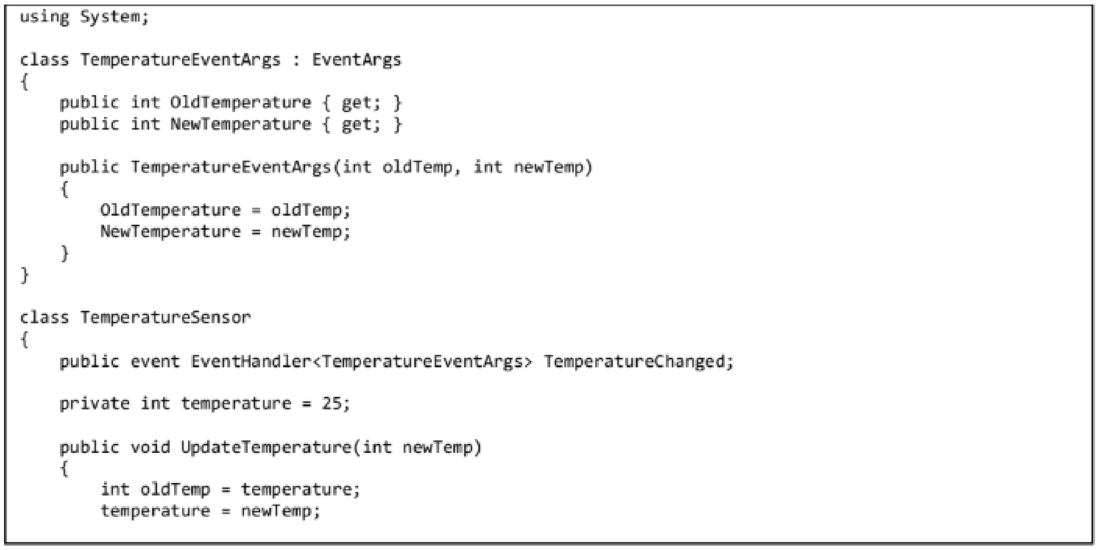


```csharp
using System;

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();
    }
}
```

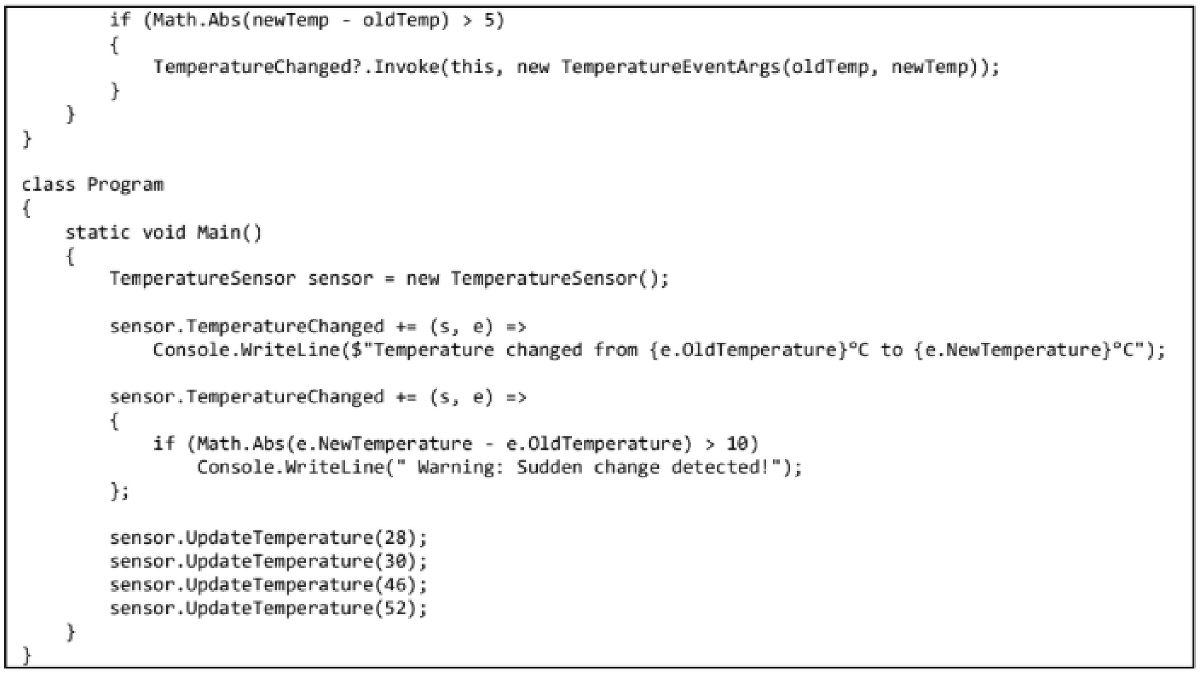
Output: >1>2[M2]>3[L3](Reset)>4[M4]{Alert}>5>6[M6][L6](Reset)

The output is generated by the for loop, which calls the Increment() method 6 times (i = 0 to 5), incrementing the value from 1 to 6. The sequence is governed by the two conditional events: MilestoneReached (fires every 2nd increment) and LimitReached (fires every 3rd increment).

Value (i in loop)	Action & Output	Reason
1 (i=0)	Prints >1	No events fire (not divisible by 2 or 3).
2 (i=1)	Prints >2, then [M2]	value % 2 == 0. MilestoneReached fires, running its subscriber.
3 (i=2)	Prints >3, then [L3](Reset)	value % 3 == 0. LimitReached fires, running its two multicast subscribers ([L3] and (Reset)).
4 (i=3)	Prints >4, then [M4]{Alert}	value % 2 == 0. MilestoneReached fires. Its subscriber checks the condition e.CurrentValue == 4 and prints {Alert} after [M4].
5 (i=4)	Prints >5	No events fire.
6 (i=5)	Prints >6, then [M6][L6](Reset)	Both events fire. MilestoneReached runs ([M6]), then LimitReached runs its two subscribers ([L6](Reset)).



```csharp
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;
```



```csharp
        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
```

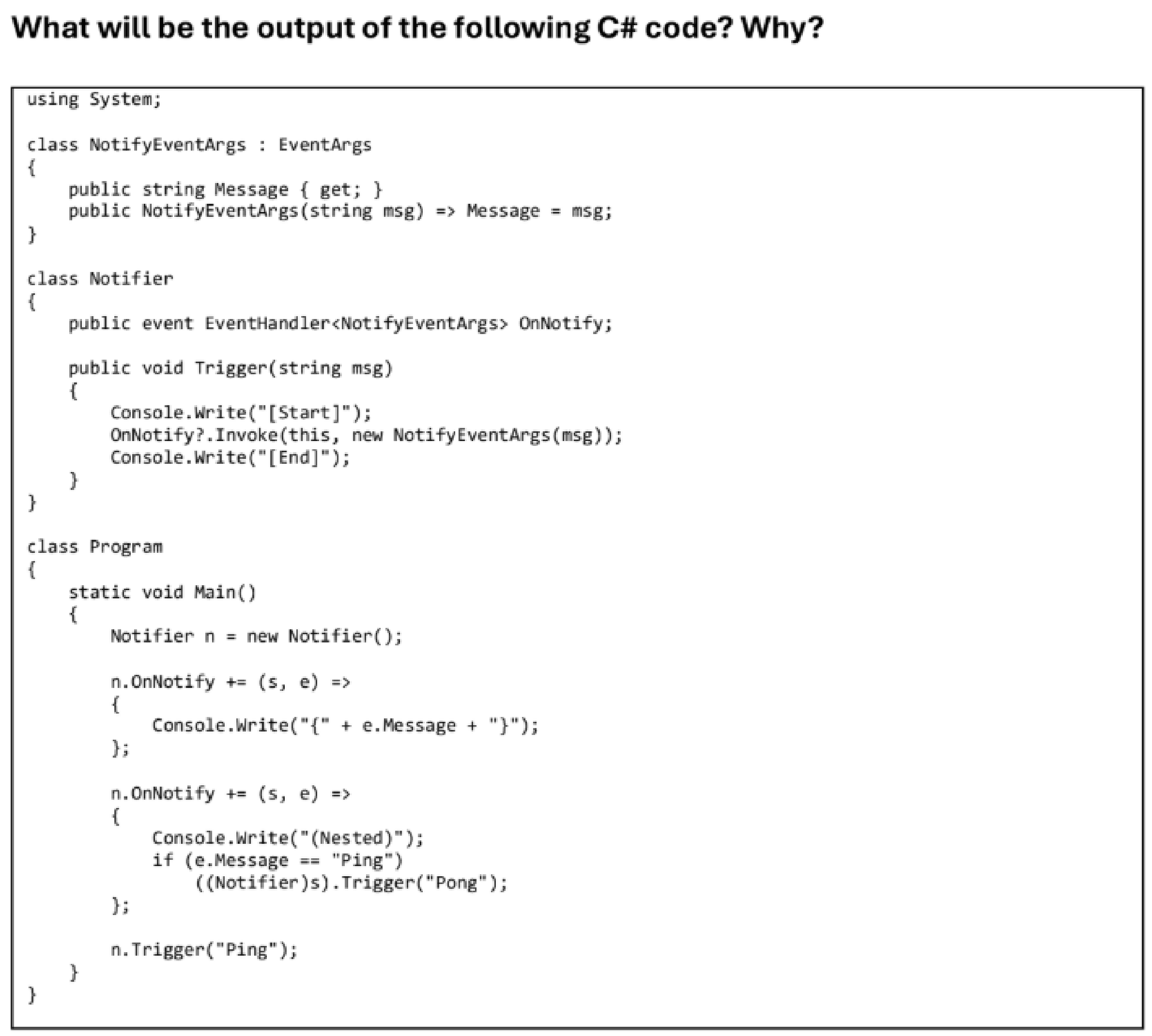
reasoning:

The code will demonstrate event execution based on a **conditional trigger** and **multicast subscription**. The event is only triggered when the temperature change is greater than 5 °C.

1. The first two updates (to 28 and 30) fail the condition.

2. The third update (30 °C to 46 °C) triggers the event (difference of 16), causing two subscribers to run: the first prints the change, and the second prints the **"Warning"** because the change is greater than10 °C
3. The final update(46 °C to 56 °C) also triggers the event (difference of 6), and only the first subscriber prints the change; the warning is skipped.

Output Reasoning (Level 2)



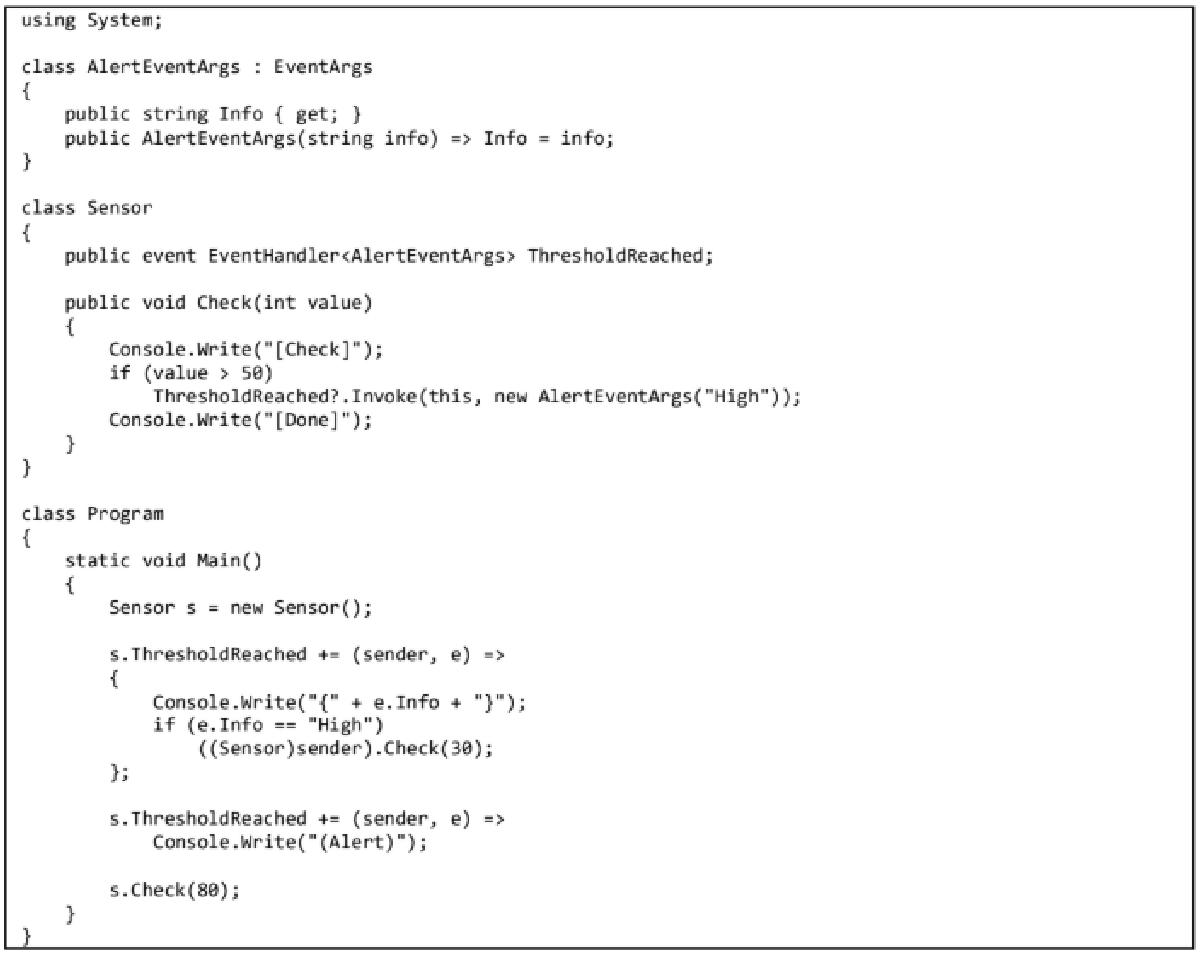
What will be the output of the following C# code? Why?

```
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})[Start]({Pong})(Nested)[End][End]
Reasoning:
This code demonstrates a scenario of **nested event execution** where an event subscriber

recursively triggers the same event, but the order of execution is crucial. The process starts when n.Trigger("Ping") is called.

1. **Outer Trigger ("Ping"):** The Trigger method runs, printing [Start]. It then invokes the OnNotify event.
2. **Subscriber 1 (S1) runs:** S1 prints ({Ping}). Inside S1, the message is checked: since e.Message is "Ping," it calls ((Notifier)s).Trigger("Pong") recursively .
    - **Inner Trigger ("Pong"):** The recursive call starts, printing [Start] again. It invokes OnNotify.
    - **Inner Subscribers run:** The event list for this inner trigger contains **S1** and the **S2 (Nested)** subscriber (since S2 was added just before the main event invocation).
        - S1 runs again, printing ({Pong}). The message is "Pong," so the recursion condition fails.
        - S2 runs, printing (Nested).
    - The inner call finishes, printing [End].
3. **Outer End:** The program returns to the original "Ping" call, which finishes and prints the final [End]



```
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:

This code demonstrates a recursive event execution scenario where one subscriber's action directly triggers the event source again. The process starts when s.Check(80) is called.

1. **First Check (**value = 80**):** The Check method runs, printing [Check]. Since , the ThresholdReached event is invoked, passing **"High"** as information.
2. **Multicast Invocation:** The two subscribers are invoked sequentially:
    - **Subscriber 1 (S1):** Runs, prints {High}. It checks if (e.Info == "High") and, since it's true, it recursively calls ((Sensor)sender).Check(30) .
        - **Nested Check (**value = 30**):** This recursive call runs, printing [Check] again. Since , the event is **not** invoked. It finishes and prints [Done].

- **Subscriber 2 (S2):** Runs after S1 completes its nested call, printing (Alert) .

3. **Outer Check Finishes:** The initial Check(80) call continues and prints [Done].

Results and Analysis

The implementation of the **EventPlayground** application successfully validated all core lab concepts by demonstrating a highly modular **publisher-subscriber model** using custom delegates and a custom EventArgs class.

The key finding confirmed the **multicast behavior** required by Task 2: the single event invocation from clicking **btnChangeColor** simultaneously triggered two distinct subscribers, changing the label's foreground color and displaying a confirmation message box. This verified the effective separation of concerns and data passing via ColorEventArgs.

The analysis of the theoretical snippets further reinforced these principles: **Delegate Arithmetic** showed the return value is always from the last method executed in a multicast chain , while the **Ref** snippet confirmed cumulative modification across methods when passing by reference. The complex **Counter** and **Recursive** problems illustrated conditional event firing and nested execution flow within event subscribers .

Discussion and conclusion

The lab successfully achieved its main goals by implementing a fully functional, event-driven GUI using **custom delegates** and the **publisher-subscriber model**. This approach effectively decoupled the control logic from the visual elements, resulting in a modular design. The crucial **multicast demonstration** was successful, proving that a single click on the "Change Color" button could simultaneously execute two separate actions: updating the label color and displaying a confirmation message, with data passed reliably via the ColorEventArgs class. In summary, the exercise verified that C# events are a flexible and essential tool for building interactive software, a principle further reinforced by the theoretical analyses of delegate execution flow and state modification across multicast chains.

LAB 12

Introduction, Setup, and Tools

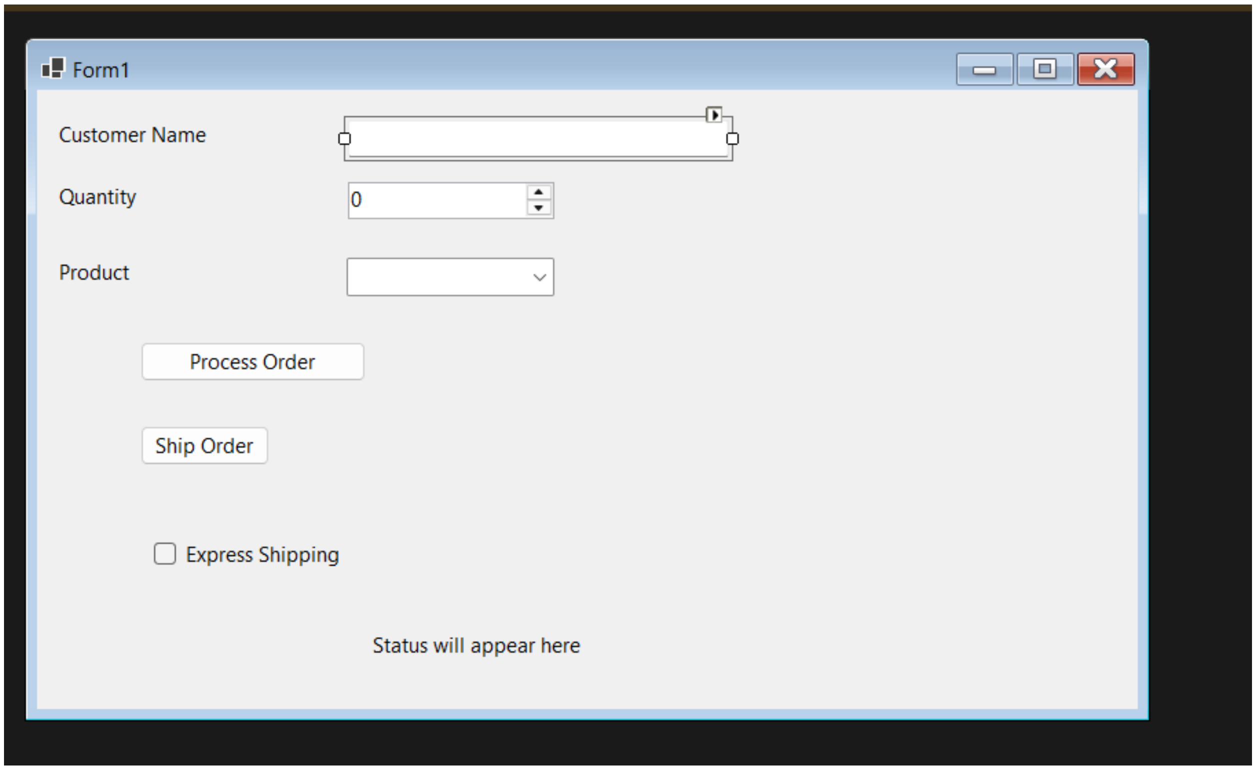
The objective of this lab is to explore advanced event-driven programming concepts in C# Windows Forms applications. It focuses on understanding how events, delegates, and custom EventArgs can be used to build modular and interactive GUI components. The tasks emphasize key ideas such as event chaining, filtered event invocation, contextual data sharing, and dynamic subscriber management. Through a multi-stage order-processing workflow, the lab demonstrates how different parts of an application can communicate efficiently while keeping the user interface decoupled from the underlying logic.

This lab was completed on a Windows 10 environment using Visual Studio 2022 (Community Edition) along with the latest .NET SDK. The project was developed in C#, using the Windows Forms App template. A new project named *OrderPipeline* was created, where the main form (Form1.cs) was designed using the Visual Studio Form Designer. Additional class files such as OrderEventArgs.cs and ShipEventArgs.cs were added to implement custom event data structures. The setup involved configuring GUI components from the Toolbox, arranging them on the form, and preparing the development environment to write, compile, and test the event-driven functionalities required for the lab.

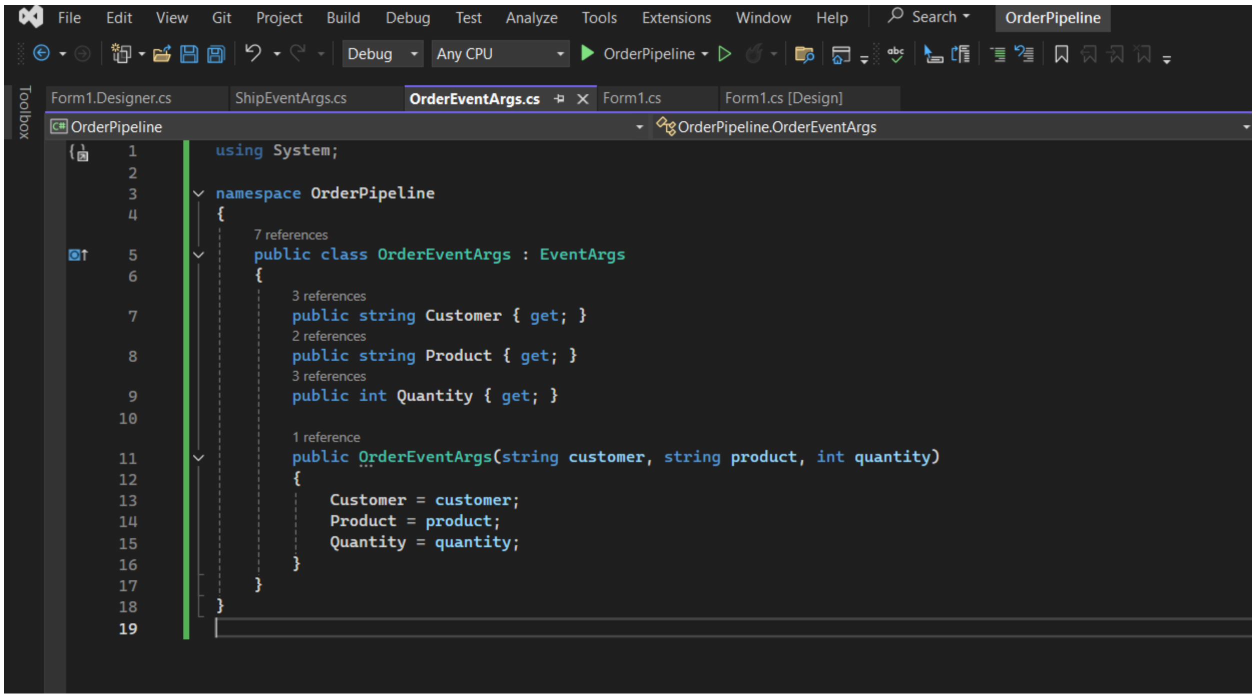
Methodology and Execution

Windows Forms App

To implement the multi-stage event-driven workflow, I began by designing the graphical user interface in Visual Studio using the Windows Forms Designer. The form contained all required components: a TextBox (txtCustomer) for entering the customer name, a ComboBox (cmbProduct) listing “Laptop”, “Mouse”, and “Keyboard”, a NumericUpDown (nudQuantity) to select quantity, a Button (btnProcessOrder) to initiate the workflow, and a Label (lblStatus) to display status updates. These controls were arranged in a clean layout to allow users to input order information and observe how the system responds through event chaining.

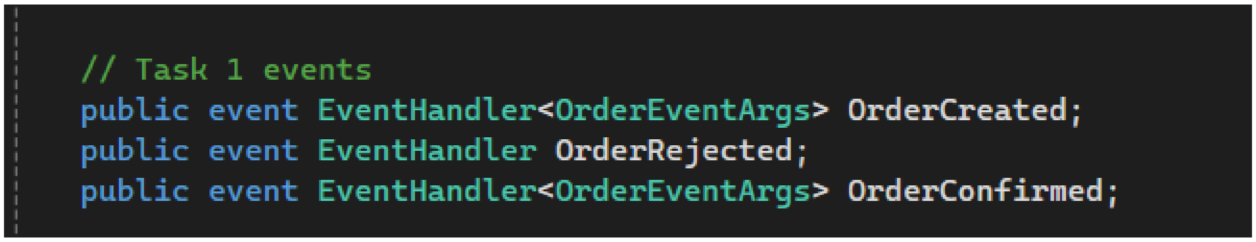


After setting up the UI, I created a custom EventArgs class to encapsulate order details. This was done by adding a new file named **OrderEventArgs.cs**, which defines properties for customer name, product, and quantity. This class allowed meaningful contextual data to be passed from the event publisher to all subscribers.

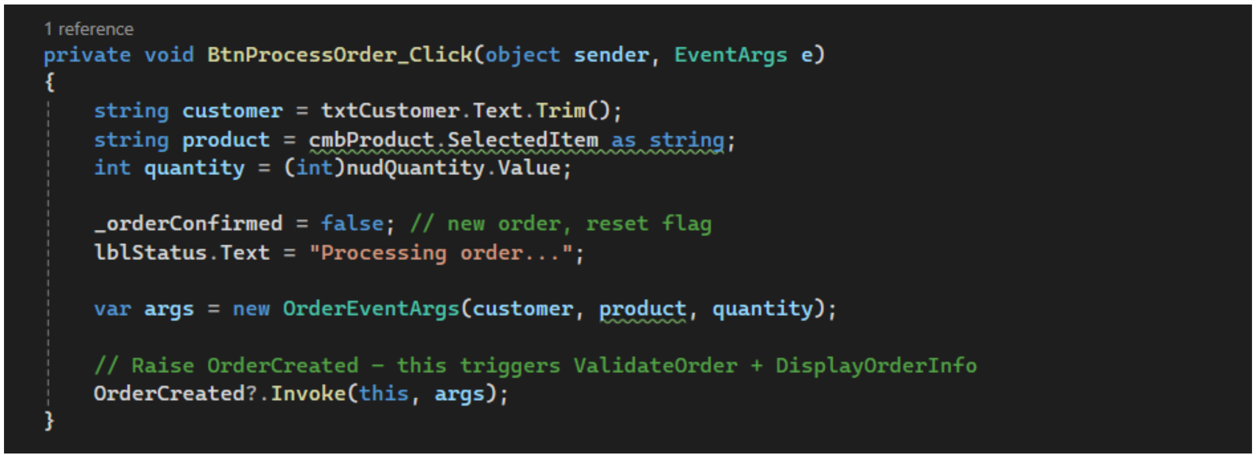

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search OrderPipeline
Debug Any CPU OrderPipeline
Form1.Designer.cs ShipEventArgs.cs OrderEventArgs.cs Form1.cs Form1.cs [Design]
OrderPipeline OrderPipeline.OrderEventArgs
Toolbox

```
using System;

namespace OrderPipeline
{
    7 references
    public class OrderEventArgs : EventArgs
    {
        3 references
        public string Customer { get; }
        2 references
        public string Product { get; }
        3 references
        public int Quantity { get; }

        1 reference
        public OrderEventArgs(string customer, string product, int quantity)
        {
            Customer = customer;
            Product = product;
            Quantity = quantity;
        }
    }
}
```

Next, I implemented the core event-driven workflow inside **Form1.cs**. Three events were created: OrderCreated, OrderRejected, and OrderConfirmed. In the form constructor, each event was linked to one or more subscriber methods to demonstrate multicast delegates. The Process Order button was programmed to collect form input, create an OrderEventArgs instance, and raise the OrderCreated event.

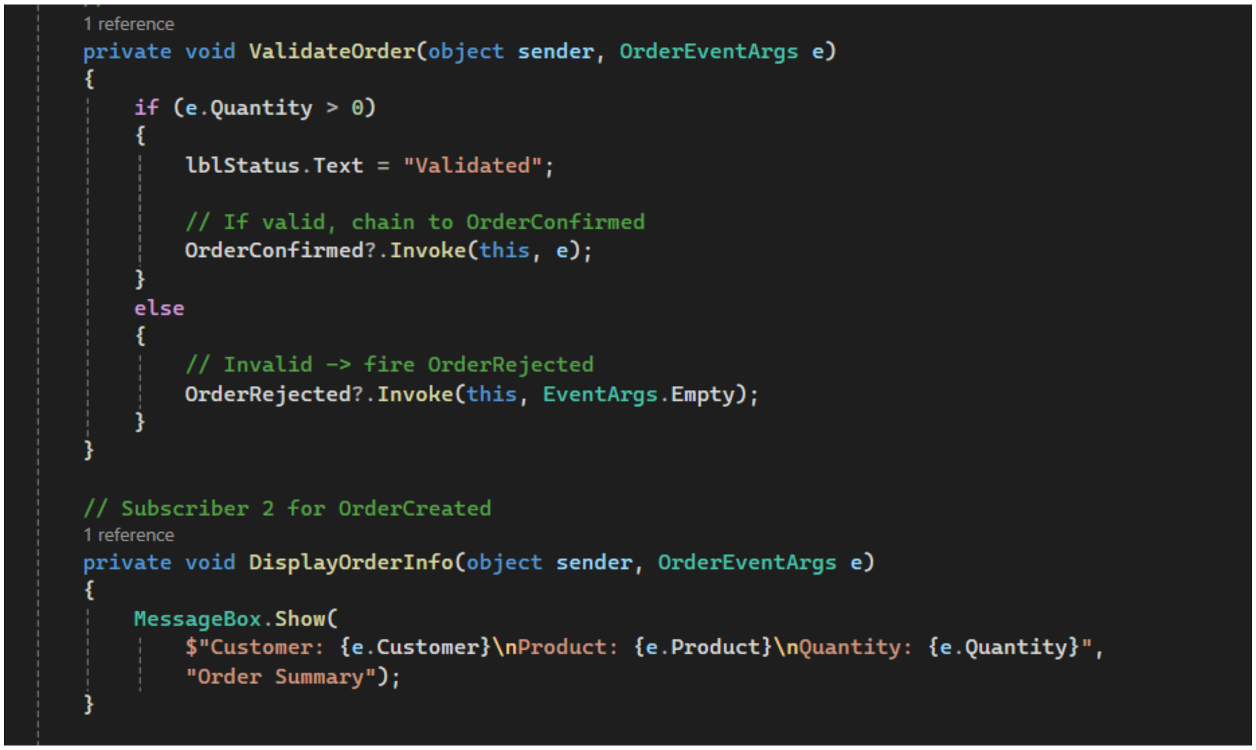


```
// Task 1 events
public event EventHandler<OrderEventArgs> OrderCreated;
public event EventHandler OrderRejected;
public event EventHandler<OrderEventArgs> OrderConfirmed;
```



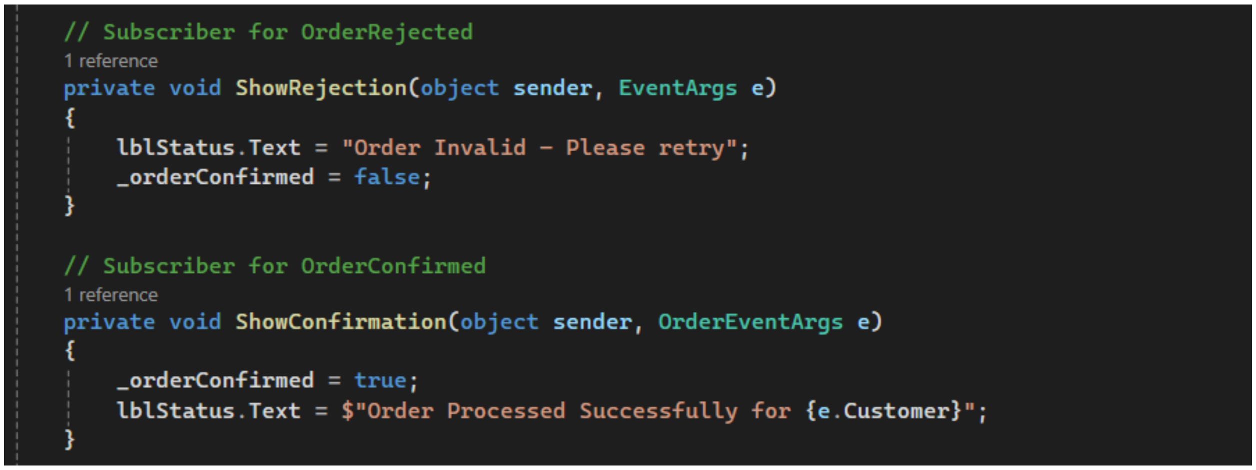
```
1 reference
private void BtnProcessOrder_Click(object sender, EventArgs e)
{
    string customer = txtCustomer.Text.Trim();
    string product = cmbProduct.SelectedItem as string;
    int quantity = (int)nudQuantity.Value;

    _orderConfirmed = false; // new order, reset flag
    lblStatus.Text = "Processing order...";

    var args = new OrderEventArgs(customer, product, quantity);

    // Raise OrderCreated - this triggers ValidateOrder + DisplayOrderInfo
    OrderCreated?.Invoke(this, args);
}
```

The OrderCreated event had two subscribers. The first, ValidateOrder(), verified whether the entered quantity was greater than zero. If valid, it updated the status label to "Validated" and raised another event, OrderConfirmed, demonstrating event chaining. If invalid, it raised the OrderRejected event. The second subscriber, DisplayOrderInfo(), displayed a MessageBox summarizing the order details, illustrating parallel handling of the same event by multiple subscribers.



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

        // If valid, chain to OrderConfirmed
        OrderConfirmed?.Invoke(this, e);
    }
    else
    {
        // Invalid -> fire OrderRejected
        OrderRejected?.Invoke(this, EventArgs.Empty);
    }
}

// Subscriber 2 for OrderCreated
1 reference
private void DisplayOrderInfo(object sender, OrderEventArgs e)
{
    MessageBox.Show(
        $"Customer: {e.Customer}\nProduct: {e.Product}\nQuantity: {e.Quantity}",
        "Order Summary");
}
```

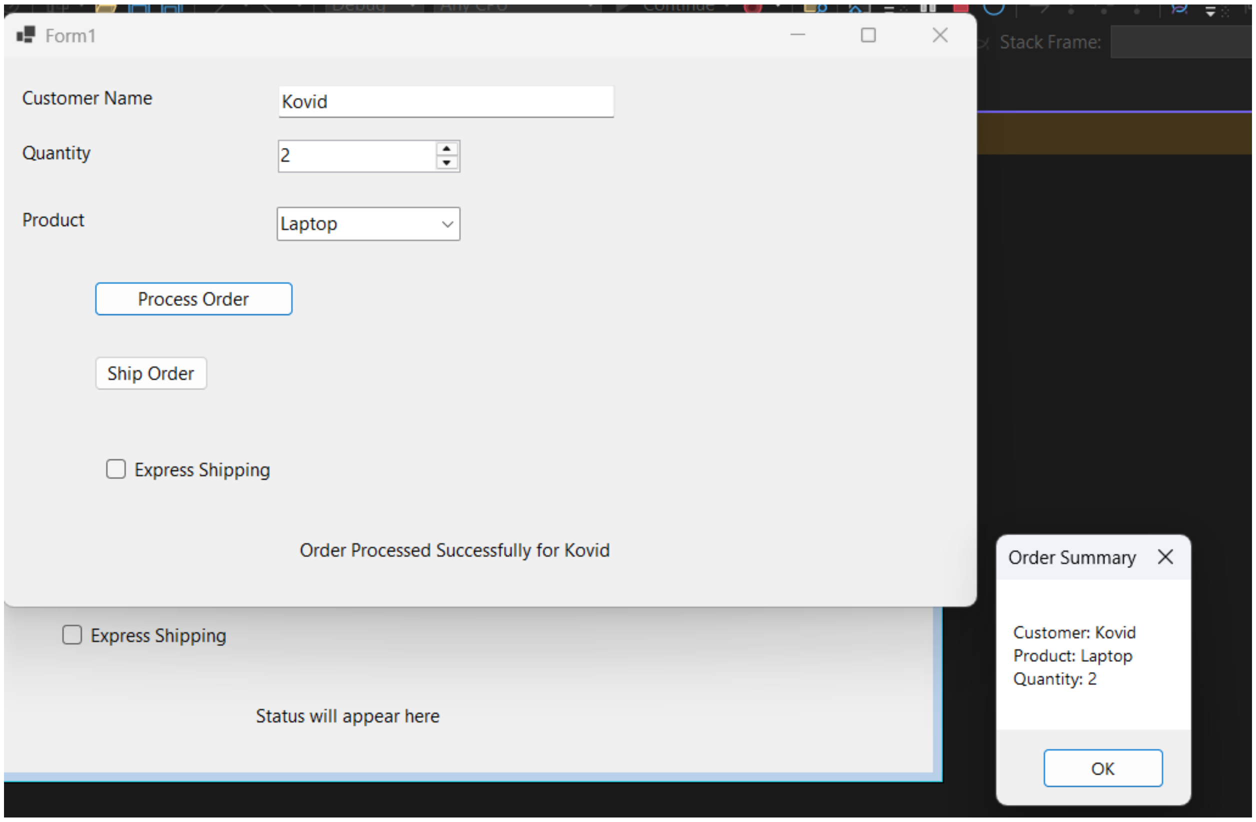
The remaining two event handlers completed the workflow. ShowRejection() responded to invalid orders raised via OrderRejected, updating the label to display “Order Invalid – Please retry”. On the other hand, ShowConfirmation() responded to the OrderConfirmed event, updating the label with “Order Processed Successfully for <Customer>”, confirming to the user that the multi-stage workflow had executed successfully.



```
// Subscriber for OrderRejected
1 reference
private void ShowRejection(object sender, EventArgs e)
{
    lblStatus.Text = "Order Invalid – Please retry";
    _orderConfirmed = false;
}

// Subscriber for OrderConfirmed
1 reference
private void ShowConfirmation(object sender, OrderEventArgs e)
{
    _orderConfirmed = true;
    lblStatus.Text = $"Order Processed Successfully for {e.Customer}";
}
```

Throughout implementation, several errors had to be debugged including designer event handler mismatches (such as accidental *_Click handlers that no longer existed) and warnings regarding nullable event parameters. These were corrected by removing leftover designer event wiring and updating handler signatures. After resolving all issues, the application successfully demonstrated multi-stage event chaining, custom event argument propagation, and clear GUI updates based on user actions.

For testing the order-processing workflow, I used two simple input cases. In the **valid test case**, I entered the customer name **“Kovid”**, selected the product **“Laptop”**, and set the quantity to **2**. When I clicked *Process Order*, the application displayed the order summary in a MessageBox and updated the status label to **“Order Processed Successfully for Kovid”**, showing that both OrderCreated and OrderConfirmed executed correctly. In the **invalid test case**, I used the same name and product but set the quantity to **0**. After clicking *Process Order*, the application immediately triggered the OrderRejected event, and the status label changed to **“Order Invalid – Please retry”**, confirming that the validation logic and rejection workflow were functioning as expected.


Form1

Customer Name: Kovid

Quantity: 2

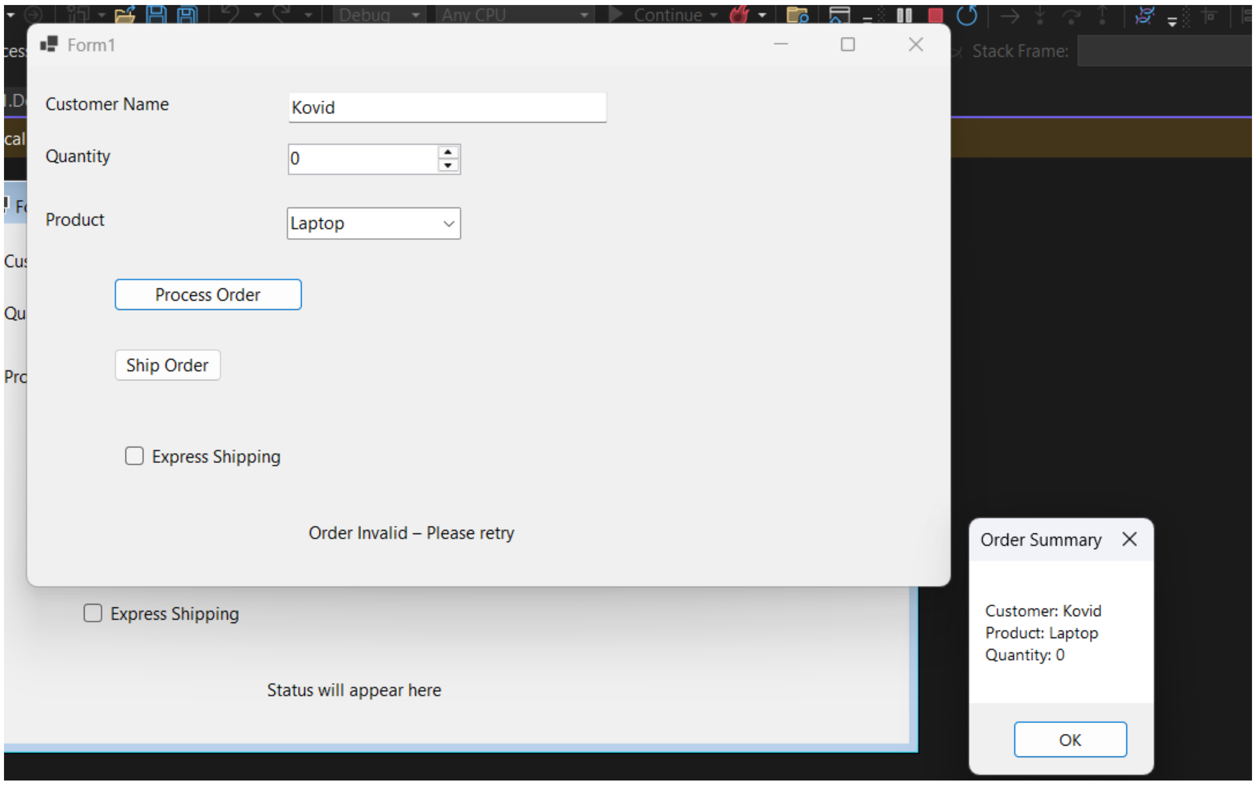
Product: Laptop

Order Processed Successfully for Kovid

Status will appear here

Order Summary

Customer: Kovid
Product: Laptop
Quantity: 2


Form1

Customer Name: Kovid

Quantity: 0

Product: Laptop

Order Invalid – Please retry

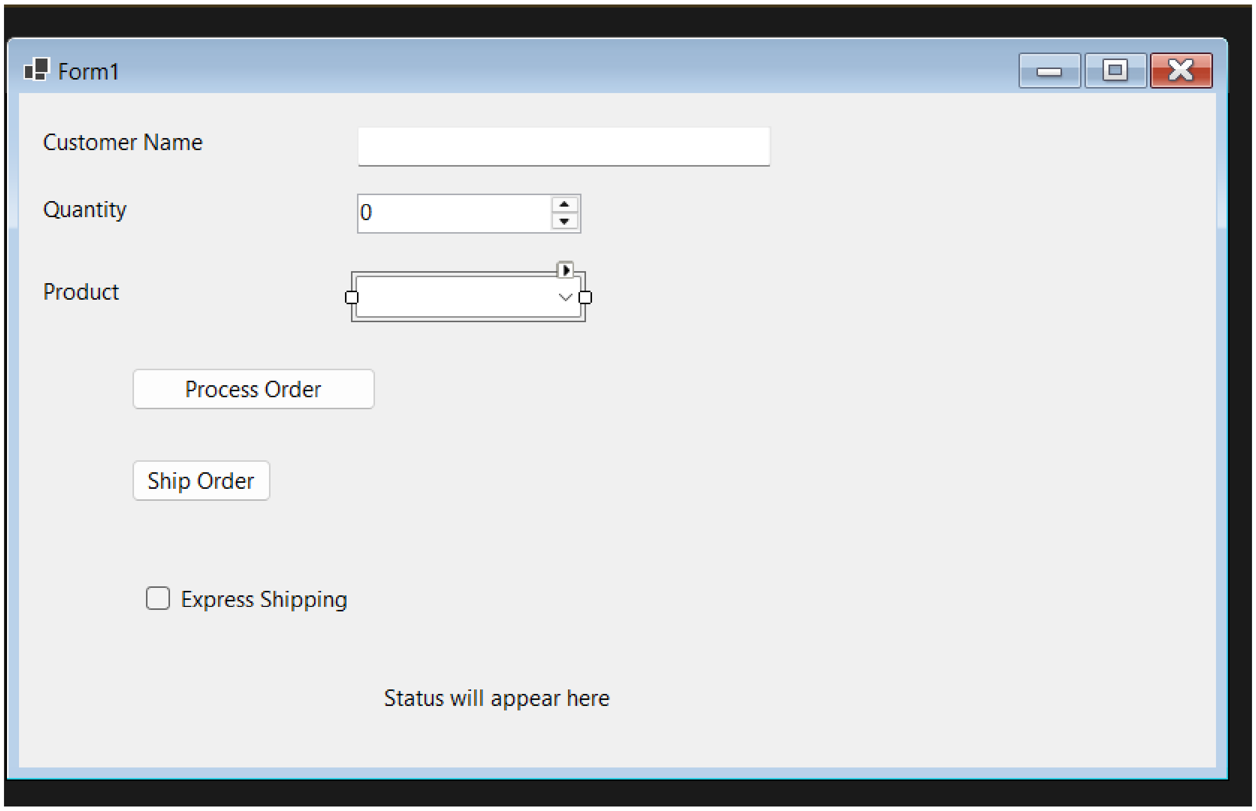
Order Summary

Customer: Kovid
Product: Laptop
Quantity: 0

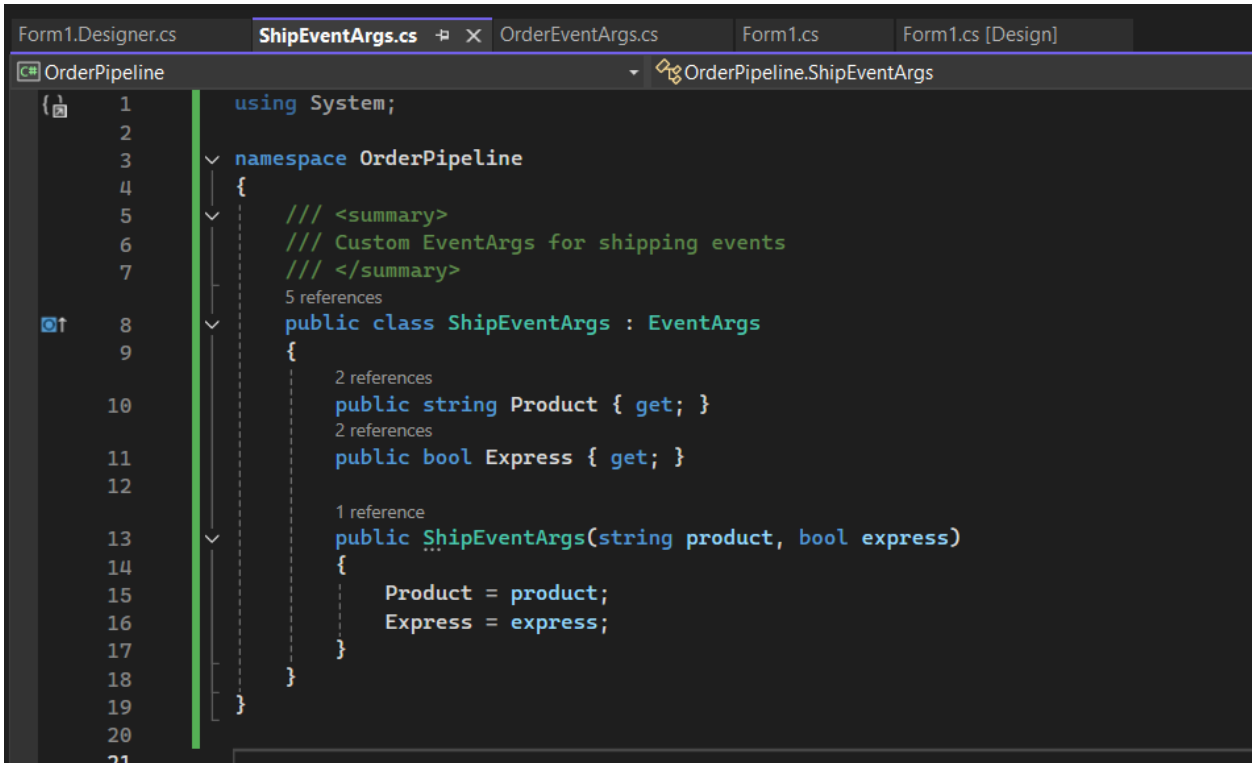
Status will appear here

Event Filtering and Dynamic Subscriber Management

o extend the previous task and demonstrate event filtering and dynamic subscription, the existing *OrderPipeline* application was modified by adding a new shipping stage. First, two additional GUI components were introduced using the Windows Forms Designer: a CheckBox named chkExpress to indicate whether the user wants express delivery, and a Button named btnShipOrder to trigger the shipping process. These controls were placed below the existing order-processing components, so the user could first process an order and then optionally choose express shipping before dispatching it.



To support the shipping operation, a new custom EventArgs class was implemented in a separate file named ShipEventArgs.cs. This class encapsulates the product being shipped and a Boolean flag indicating whether express delivery is selected. Defining this class allowed the shipping event to carry both the product information and the express option to all its subscribers.

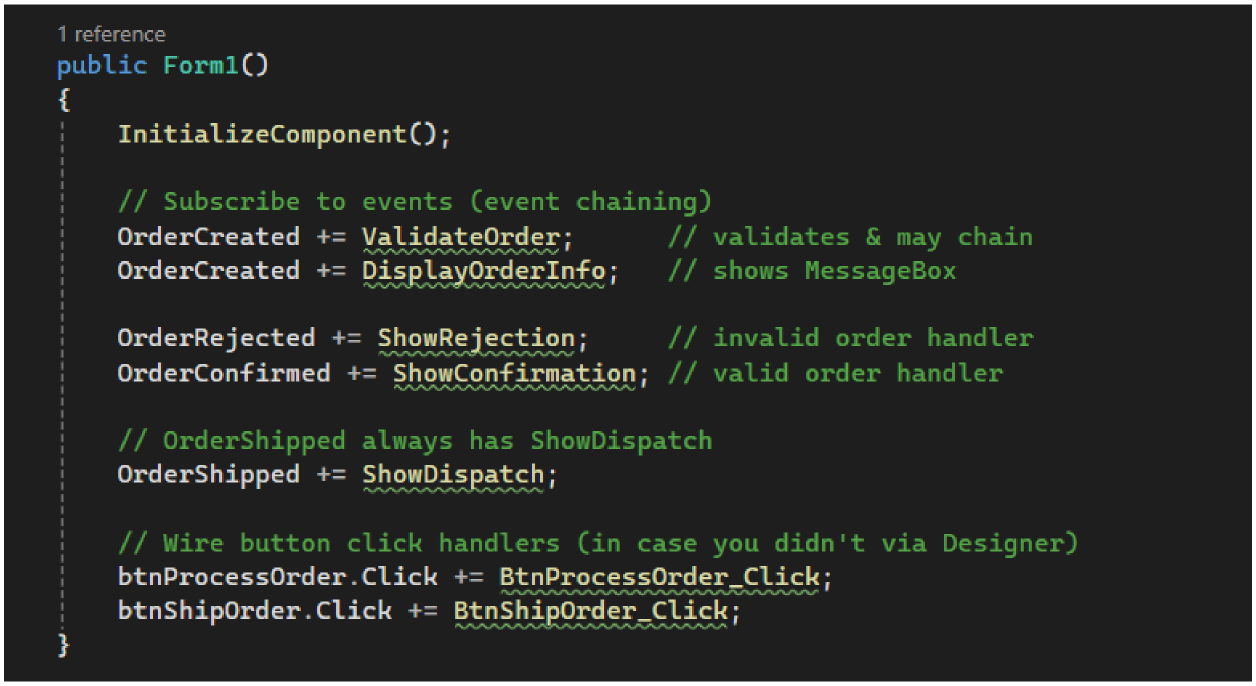


Form1.Designer.cs | ShipEventArgs.cs | OrderEventArgs.cs | Form1.cs | Form1.cs [Design]

OrderPipeline | OrderPipeline.ShipEventArgs

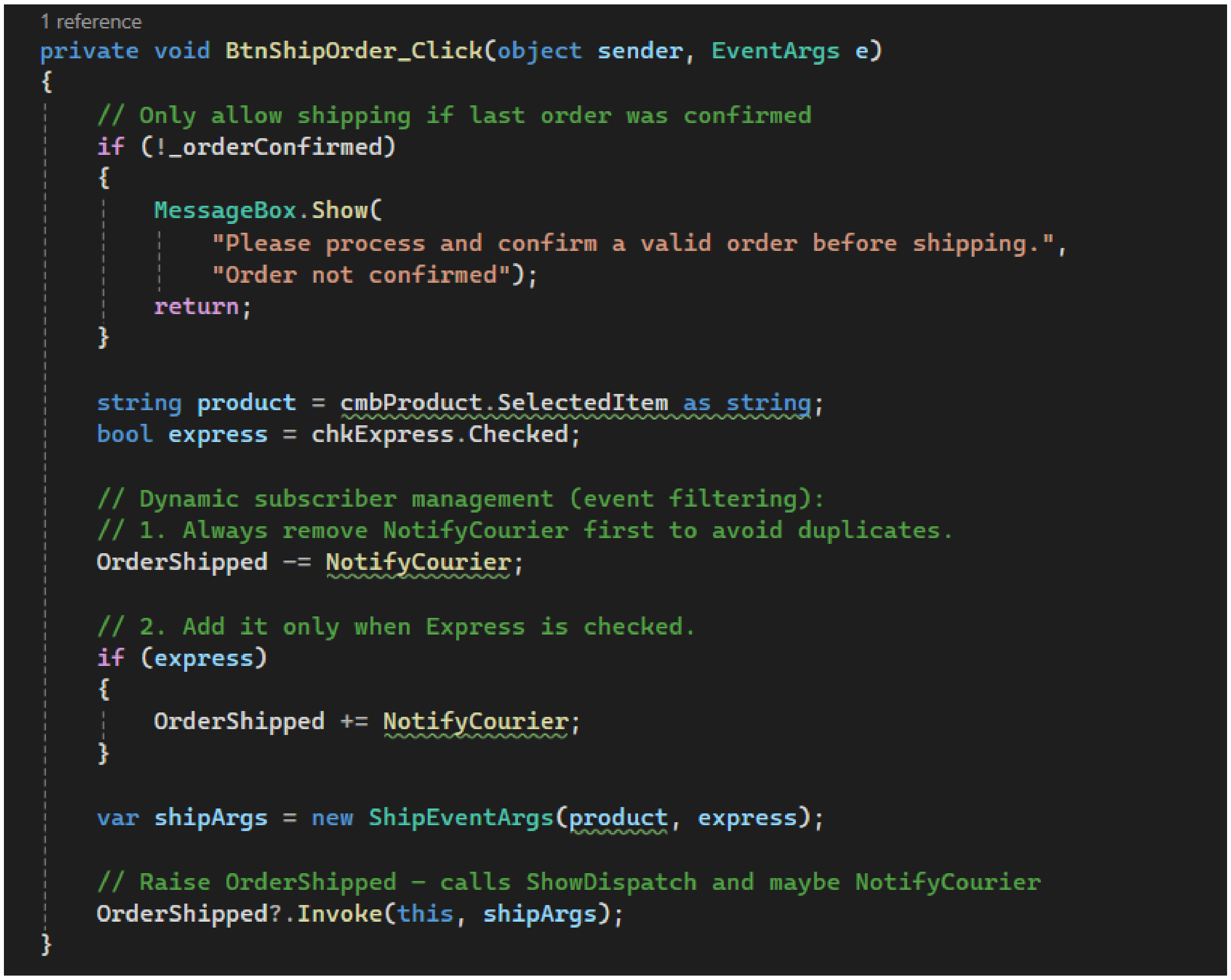
```
using System;

namespace OrderPipeline
{
    /// <summary>
    /// Custom EventArgs for shipping events
    /// </summary>
    5 references
    public class ShipEventArgs : EventArgs
    {
        2 references
        public string Product { get; }
        2 references
        public bool Express { get; }

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

Inside Form1.cs, a new event named OrderShipped was declared, similar to OrderCreated from Task 1, but using ShipEventArgs as the type parameter. The event was then wired to at least one subscriber, ShowDispatch(), which is always attached and responsible for updating the status label when a product is dispatched. A second subscriber, NotifyCourier(), was used to demonstrate dynamic subscription and event filtering; it is conditionally attached based on the state of the express CheckBox. The existing _orderConfirmed Boolean flag from Task 1 was reused to ensure that shipping can only occur if the previous order has been successfully confirmed.



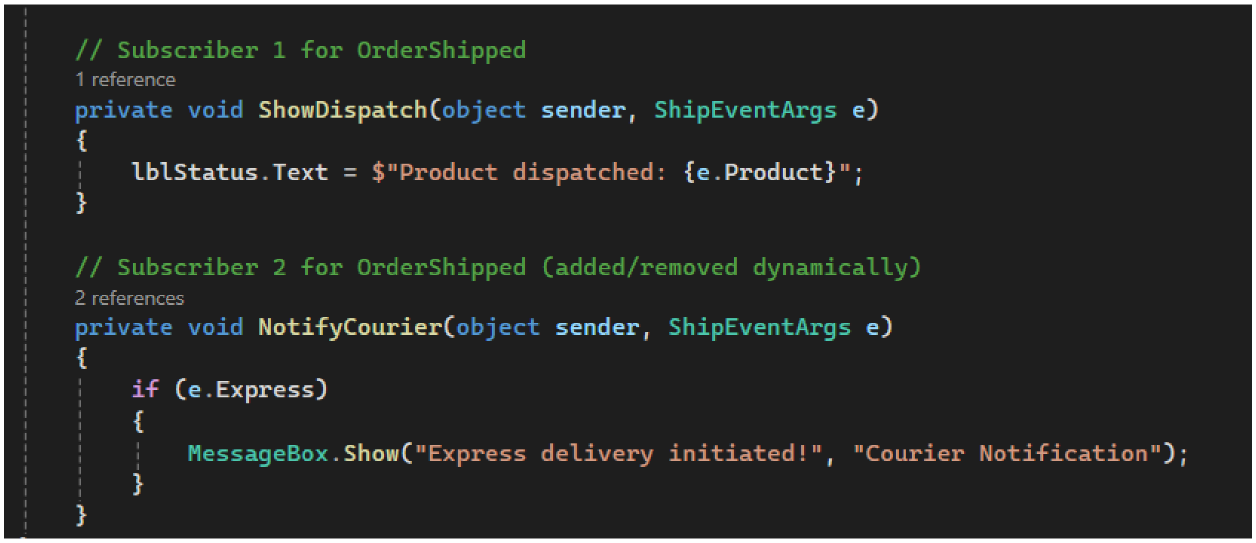
```
1 reference
public Form1()
{
    InitializeComponent();

    // Subscribe to events (event chaining)
    OrderCreated += ValidateOrder;      // validates & may chain
    OrderCreated += DisplayOrderInfo;   // shows MessageBox

    OrderRejected += ShowRejection;     // invalid order handler
    OrderConfirmed += ShowConfirmation; // valid order handler

    // OrderShipped always has ShowDispatch
    OrderShipped += ShowDispatch;

    // Wire button click handlers (in case you didn't via Designer)
    btnProcessOrder.Click += BtnProcessOrder_Click;
    btnShipOrder.Click += BtnShipOrder_Click;
}
```

The core logic for event filtering and dynamic subscriber management was implemented in the BtnShipOrder_Click handler. When the user presses the *Ship Order* button, the method first checks the _orderConfirmed flag. If the order has not been confirmed, a MessageBox is shown to the user and the method returns without firing the event, demonstrating conditional event triggering based on application state. If the order is confirmed, the method reads the selected product and the chkExpress flag, constructs a ShipEventArgs instance, and then dynamically manages the NotifyCourier subscription: it explicitly removes NotifyCourier from the OrderShipped event, and re-adds it only if chkExpress is checked. Finally, the OrderShipped event is raised.



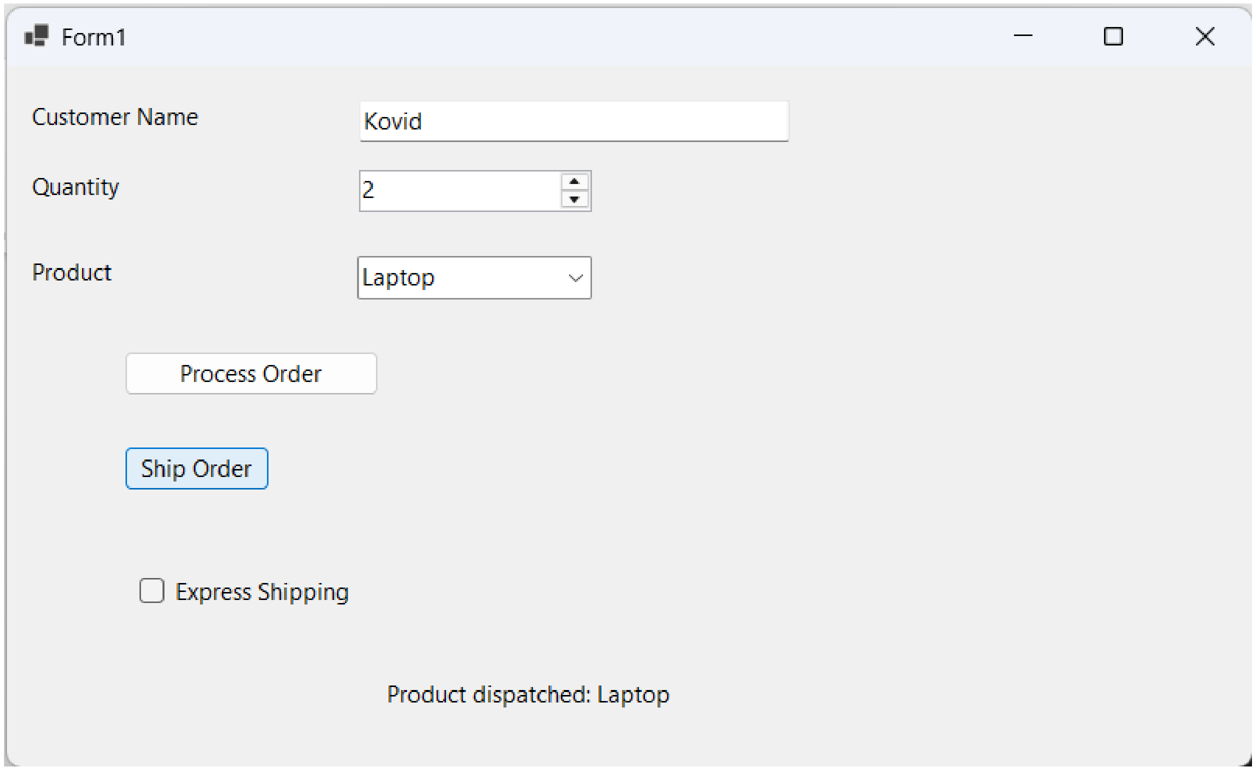
```
1 reference
private void BtnShipOrder_Click(object sender, EventArgs e)
{
    // Only allow shipping if last order was confirmed
    if (!_orderConfirmed)
    {
        MessageBox.Show(
            "Please process and confirm a valid order before shipping.",
            "Order not confirmed");
        return;
    }

    string product = cmbProduct.SelectedItem as string;
    bool express = chkExpress.Checked;

    // Dynamic subscriber management (event filtering):
    // 1. Always remove NotifyCourier first to avoid duplicates.
    OrderShipped -= NotifyCourier;

    // 2. Add it only when Express is checked.
    if (express)
    {
        OrderShipped += NotifyCourier;
    }

    var shipArgs = new ShipEventArgs(product, express);

    // Raise OrderShipped – calls ShowDispatch and maybe NotifyCourier
    OrderShipped?.Invoke(this, shipArgs);
}
```

Two subscribers handle the OrderShipped event. The ShowDispatch() method always executes whenever the shipping event is raised and updates the status label with a message indicating that the product has been dispatched. The NotifyCourier() method is only invoked when it is dynamically attached (i.e., when express shipping is enabled). In that case, it displays a MessageBox confirming that express delivery has been initiated.



```
// Subscriber 1 for OrderShipped
1 reference
private void ShowDispatch(object sender, ShipEventArgs e)
{
    lblStatus.Text = $"Product dispatched: {e.Product}";
}

// Subscriber 2 for OrderShipped (added/removed dynamically)
2 references
private void NotifyCourier(object sender, ShipEventArgs e)
{
    if (e.Express)
    {
        MessageBox.Show("Express delivery initiated!", "Courier Notification");
    }
}
```

To verify the correctness of this dynamic subscription mechanism, I tested three main scenarios. In the **first scenario (regular shipping)**, I processed a valid order using the name **"Kovid"**, selected the product **"Laptop"**, set the quantity to **2**, and left the **Express** CheckBox unchecked. After clicking *Ship Order*, the status label updated to **"Product dispatched: Laptop"** and no extra MessageBox was shown, indicating that OrderShipped invoked only ShowDispatch() and NotifyCourier() was not subscribed.

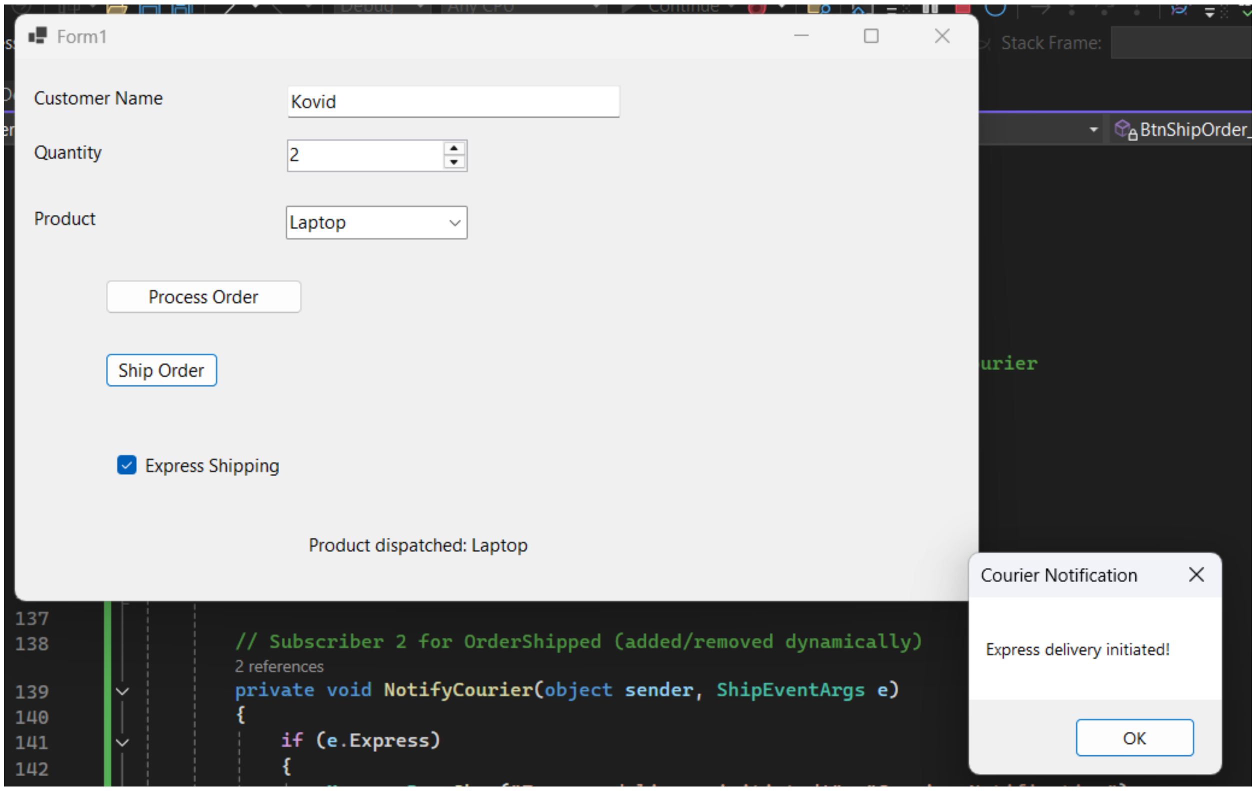


Customer Name	Kovid
Quantity	2
Product	Laptop

☐ Express Shipping

Product dispatched: Laptop

In the **second scenario (express shipping)**, I used the same order details but checked the **Express** option before clicking *Ship Order*. This time, the status label again showed **“Product dispatched: Laptop”**, and an additional MessageBox appeared with the text **“Express delivery initiated!”**, confirming that NotifyCourier() had been dynamically added as a subscriber.



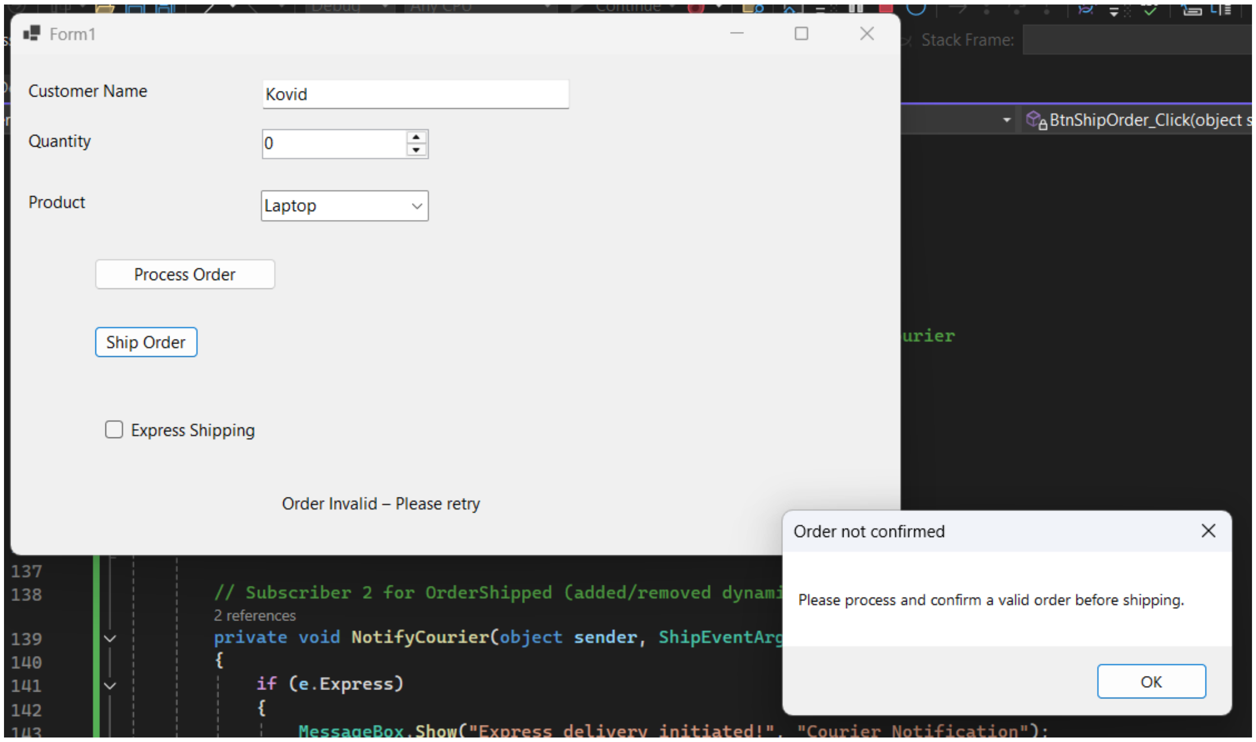
Customer Name: Kovid
Quantity: 2
Product: Laptop
Express Shipping ✓

Product dispatched: Laptop

Express delivery initiated!

```
// Subscriber 2 for OrderShipped (added/removed dynamically)
2 references
private void NotifyCourier(object sender, ShipEventArgs e)
{
    if (e.Express)
    {
```

In the **third scenario (shipping without confirmation)**, I attempted to click *Ship Order* before processing any valid order. In this case, the _orderConfirmed flag was false, and the application showed a MessageBox stating **“Please process and confirm a valid order before shipping.”** No shipping event was raised, and the status label remained unchanged. This confirmed that the system correctly enforces workflow ordering and uses event triggering conditionally based on application state. Throughout this task, any issues such as missing event handlers or incorrect subscriptions were resolved by checking the designer-generated code and ensuring that method signatures matched the expected EventHandler<T> pattern. After debugging, the application successfully demonstrated event filtering, dynamic subscriber management, and controlled shipping behavior based on express selection and order confirmation status.



Form1

Customer Name Kovid

Quantity 0

Product Laptop

Express Shipping

Order Invalid – Please retry

Order not confirmed

Please process and confirm a valid order before shipping.

```
        // Subscriber 2 for OrderShipped (added/removed dynami
        2 references
        private void NotifyCourier(object sender, ShipEventArg
        {
            if (e.Express)
            {
                MessageBox.Show("Express delivery initiated!", "Courier Notification");
```

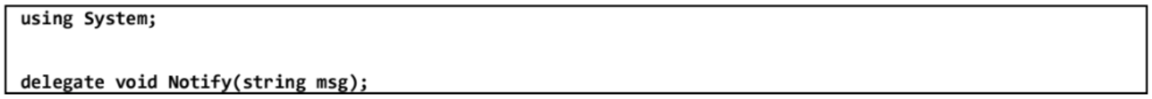
Output Reasoning (Level 0)

- **Does this code compile? If not, identify the error and explain why. If it does, what would print?**

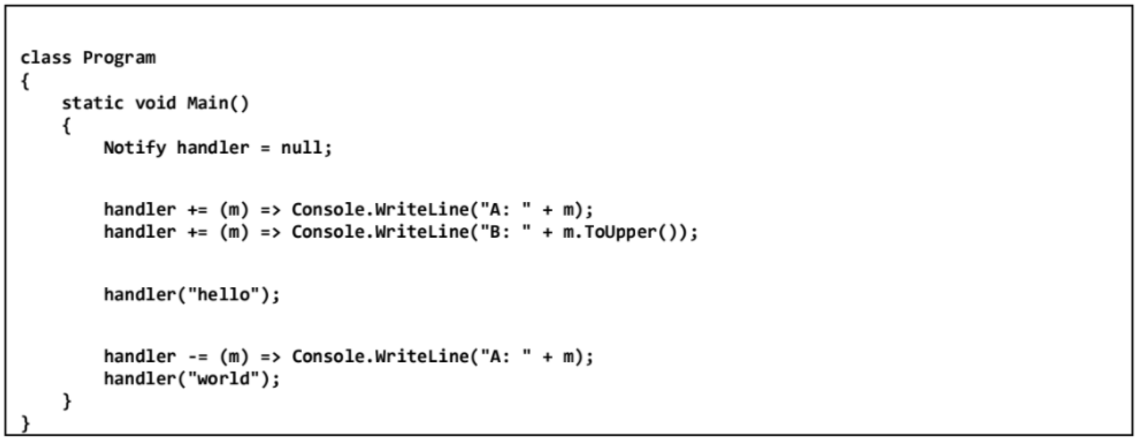
```
public delegate void AuthCallback(bool validUser);
public static AuthCallback loginCallback = Login;
public static void Login()
{
    Console.WriteLine("Valid user!");
}

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

No, the code does not compile because the delegate AuthCallback expects a method that takes one boolean parameter (bool validUser), but the method assigned to it, Login, does not take any parameters. Since the method signature does not match the delegate signature, the compiler throws an error stating that the method cannot be used with the delegate. If the method Login is corrected to accept a boolean parameter, the code will compile and calling loginCallback(true) would print “Valid user!”.

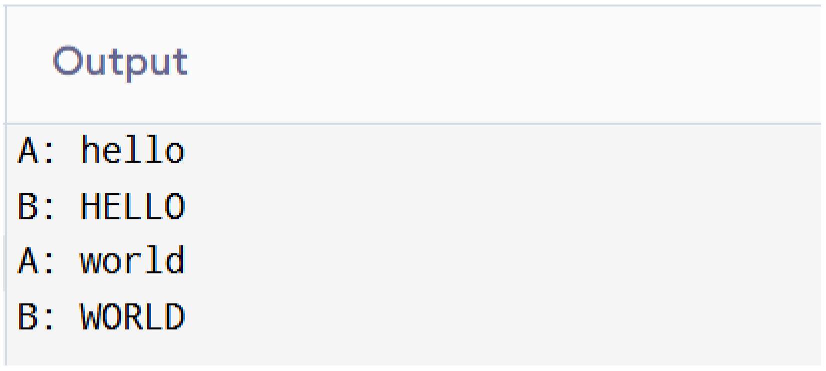
➢ **What will be the output of the following C# code? Why?**



```
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");
    }
}
```



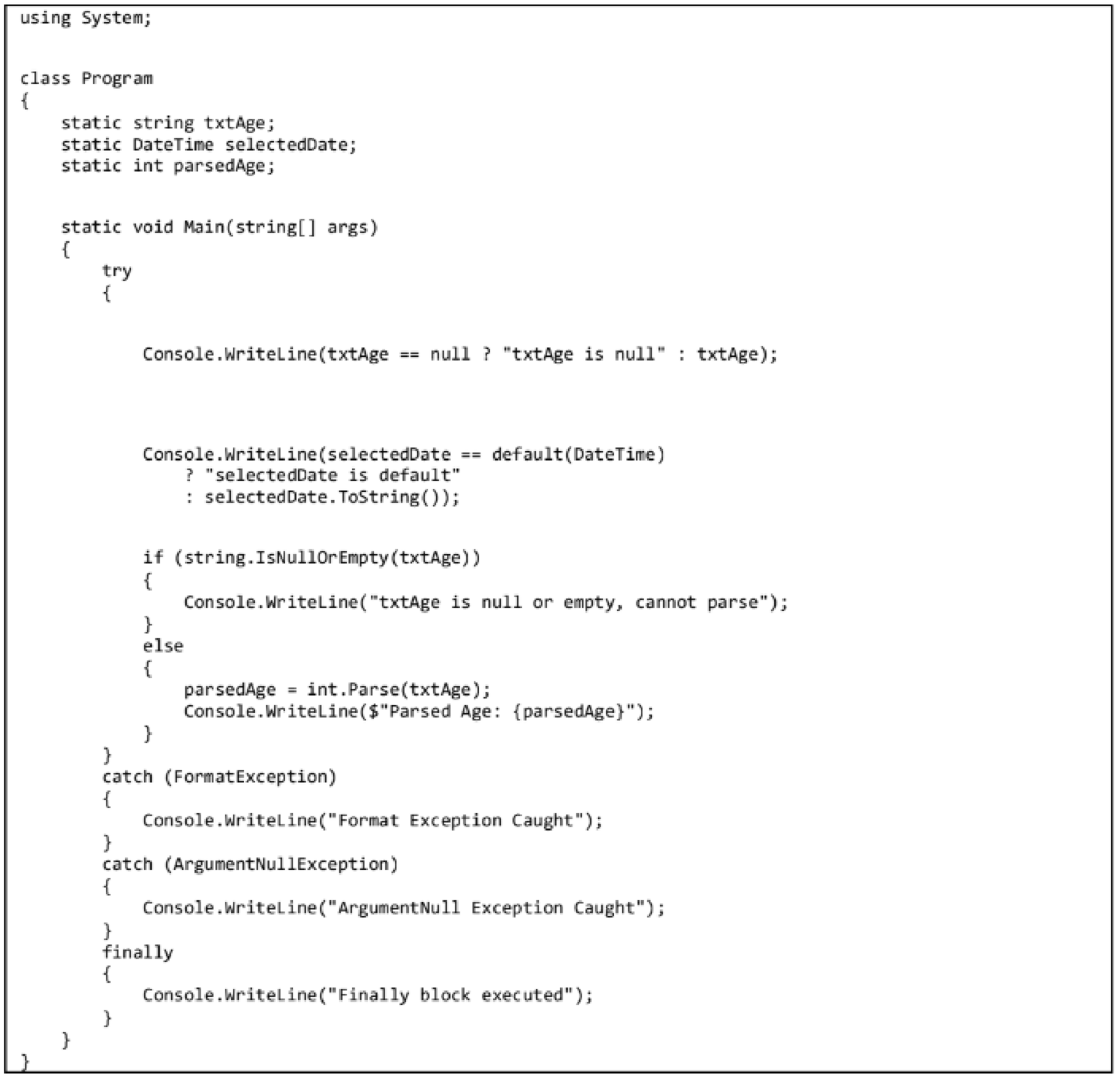
Output

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

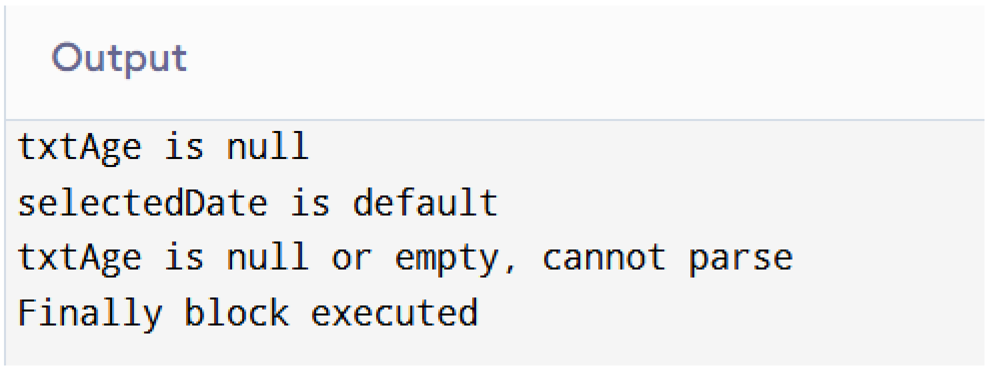
Because both lambda handlers are added to the multicast delegate and the first handler("hello") calls them in order (printing "A: hello" then "B: HELLO"). The subsequent handler -= (m) => Console.WriteLine("A: " + m); does **not** remove the original first handler it creates a new, distinct delegate instance (even though the source looks identical), so no matching delegate is found to remove. Therefore the second handler("world") still invokes both handlers, producing "A: world" and "B: WORLD".

Output Reasoning (Level 1)

➢ **What will be the output of the following C# code? Why?**



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

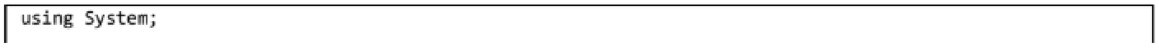


Output

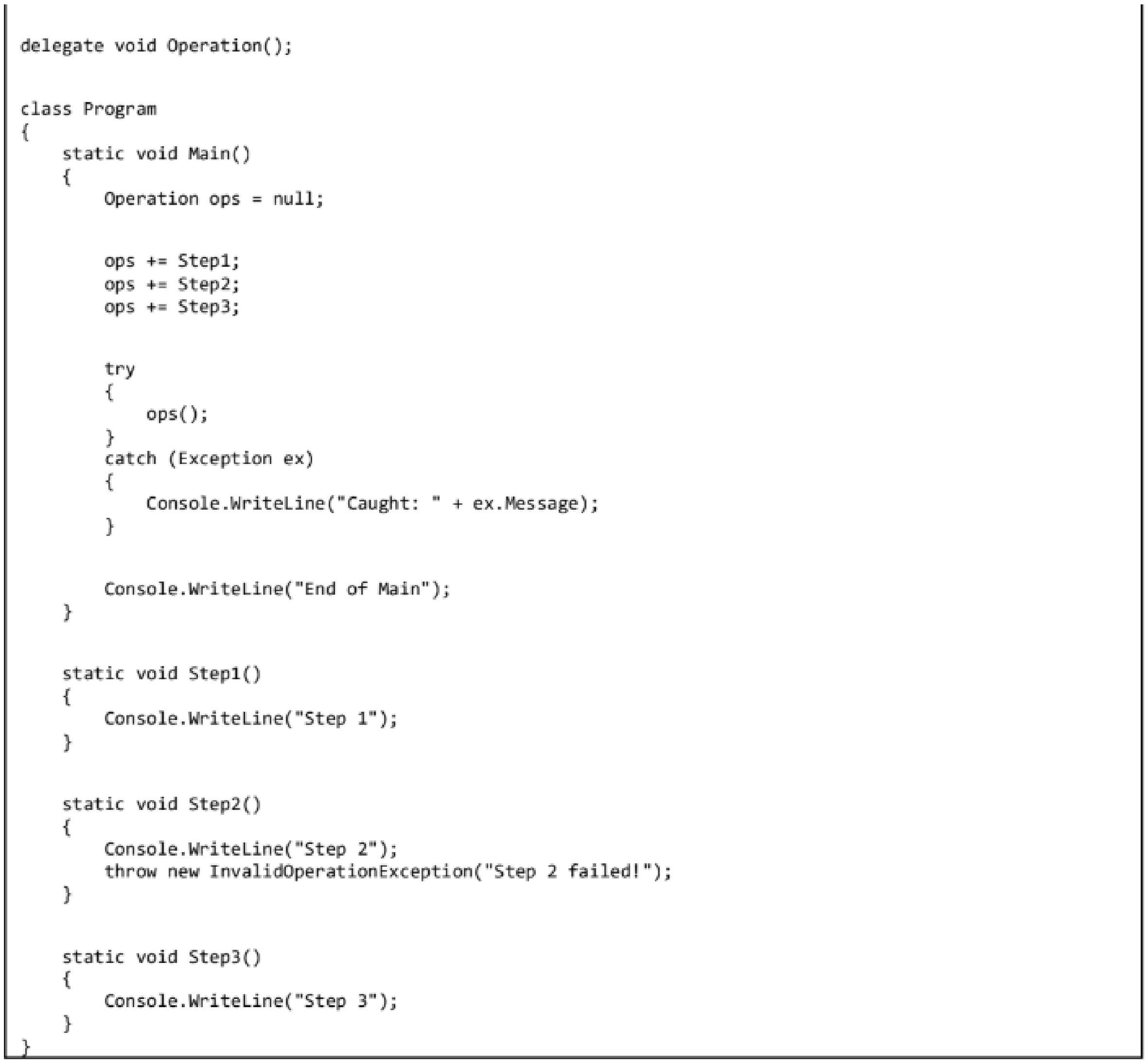
```
txtAge is null
selectedDate is default
txtAge is null or empty, cannot parse
Finally block executed
```

The output of the program is: **"txtAge is null", "selectedDate is default", "txtAge is null or empty, cannot parse", and finally "Finally block executed."** This happens because txtAge is a static string that is never assigned, so its value is null, causing the first line to print that it is null. The selectedDate variable is an uninitialized DateTime, and all uninitialized DateTime variables default to DateTime.MinValue, so the program prints that it is default. Since txtAge is null, string.IsNullOrEmpty(txtAge) returns true, leading the program to print that it cannot parse the age, and no parsing is attempted therefore no exception is thrown. The finally block always runs regardless of errors, so the last line printed is "Finally block executed."

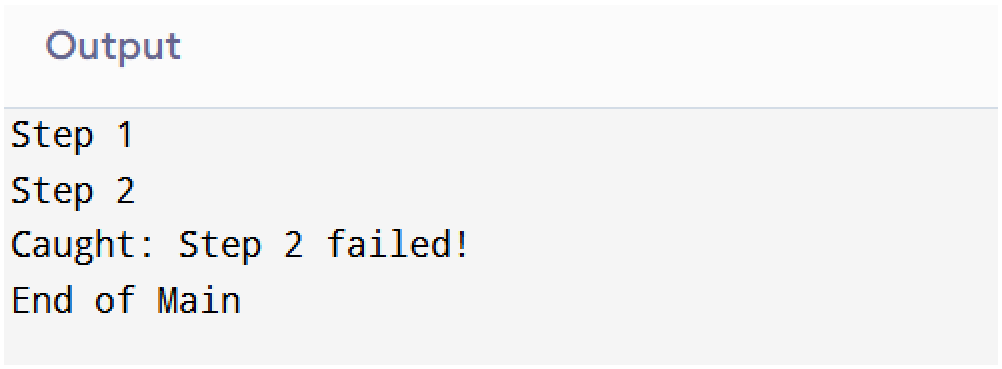
- **What will be the output of the following C# code? Why?**



```
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");
    }
}
```



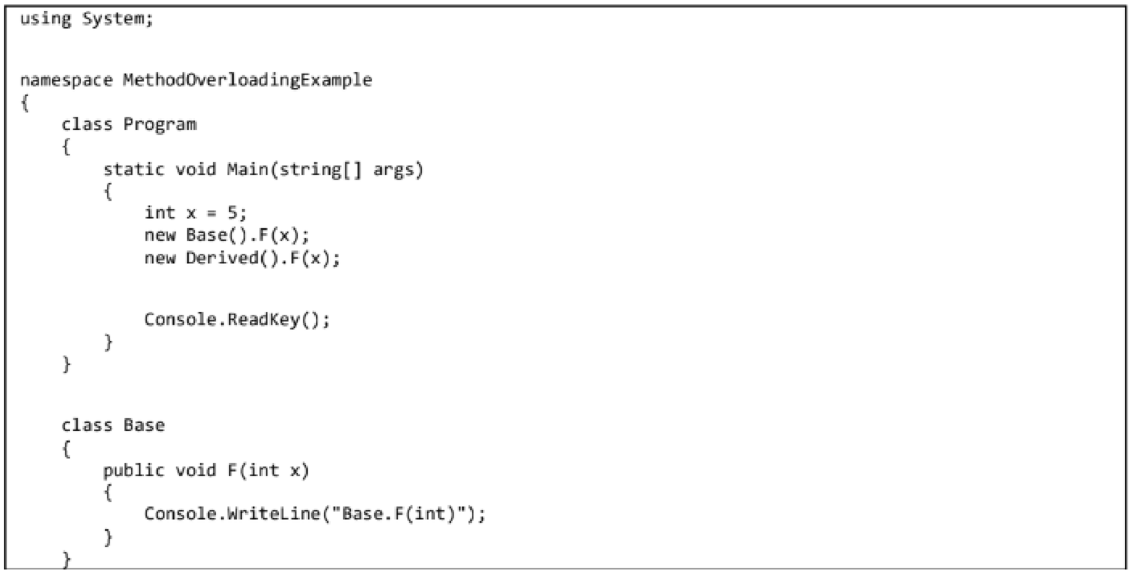
Output

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

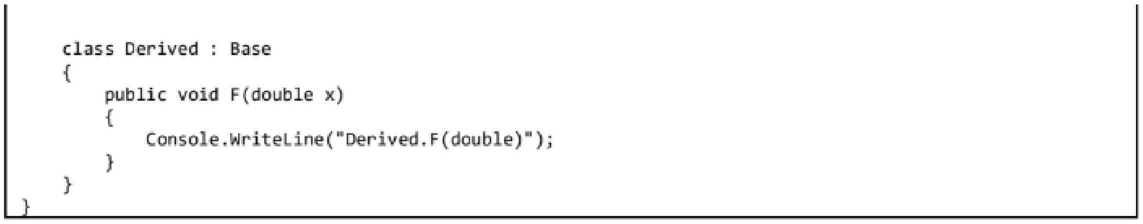
The program prints **Step 1** and then **Step 2**; when Step2 throws an InvalidOperationException the multicast delegate invocation stops immediately and the exception propagates back to the try in Main, so **Step 3 is never executed**. The catch block catches the exception and prints its message ("Caught: Step 2 failed!"), and finally the program continues and prints **End of Main**.

Output Reasoning (Level 2)

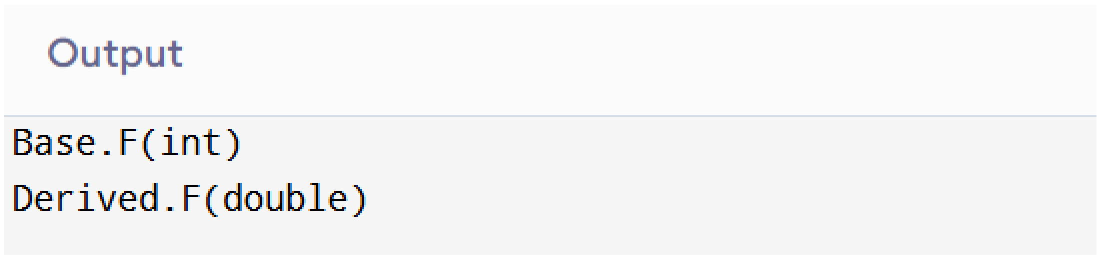
➢ **What will be the output of the following C# code? Why?**



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



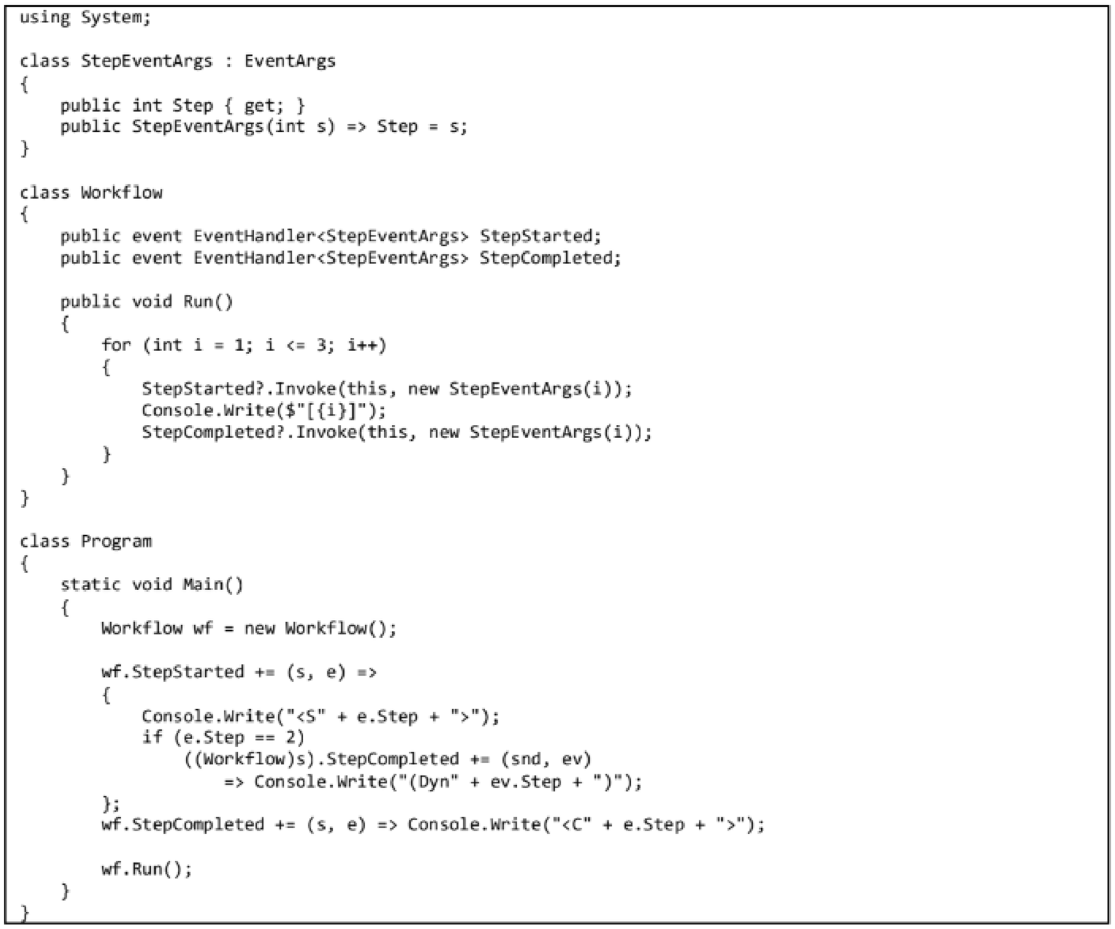
```

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


Output

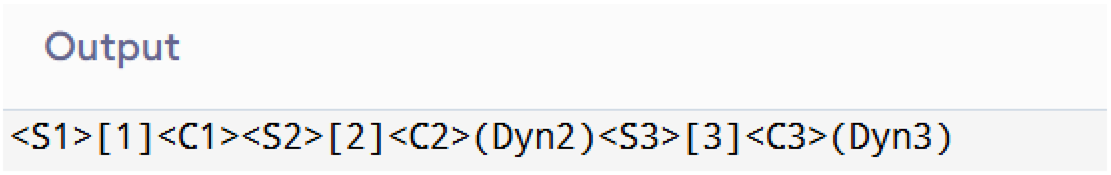
```
Base.F(int)
Derived.F(double)
```

The first call new Base().F(x) prints **Base.F(int)** because the argument x is an integer and matches the F(int) method in the Base class exactly. The second call new Derived().F(x) prints **Derived.F(double)** because the Derived class introduces a new overload F(double) which hides the inherited F(int) method from Base. When a method in the derived class has the same name but a different parameter list, it does not override the base version; instead, it *shadows* it. For this reason, when the call is made through a Derived object, only the Derived class's own overloads are considered first, and the compiler selects F(double) as the best available match, converting the integer argument to a double.

➢ **What will be the output of the following C# code? Why?**



```
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)
```

The program raises StepStarted then prints [{i}] then raises StepCompleted for i = 1..3. Initially StepCompleted has one subscriber that writes <C#>. The StepStarted handler always writes <S#> and, when e.Step == 2, **adds** a new handler to StepCompleted that writes (Dyn#). Because the dynamic subscription happens during the StepStarted invocation for i = 2 (and before StepCompleted is invoked for that same i), the newly added handler is present when StepCompleted runs for i = 2 and it remains for i = 3 as well. Handlers are invoked in subscription order, so for i = 2 and 3 the sequence for completion is the original <C#> then (Dyn#). Concatenating all writes yields the shown single-line output.

Results and Analysis

The implementation of the *OrderPipeline* application produced outputs that clearly demonstrated the functioning of advanced event-handling mechanisms in C#. In **Task 1**, the results confirmed correct behavior for both valid and invalid orders. When a valid input was provided for example, Customer **"Kovid"**, Product **"Laptop"**, and Quantity **2** the system raised the OrderCreated event, invoked both of its subscribers, and displayed an order summary through a MessageBox. The status label ultimately showed **"Order Processed Successfully for Kovid"**, confirming that the event chain (OrderCreated → OrderConfirmed) executed as expected. In contrast, when an invalid input such as quantity **0** was tested, the ValidateOrder() method correctly triggered the OrderRejected event, and the system displayed **"Order Invalid – Please retry"**. These contrasting outputs confirmed that the event validation and branching logic were implemented accurately.

In **Task 2**, the results highlighted the effectiveness of event filtering and dynamic subscriber management. When the order was confirmed and the user left **Express Shipping unchecked**, clicking *Ship Order* resulted in a single output on the status label—**"Product dispatched: Laptop"**—indicating that only the ShowDispatch() handler executed. However, when the **Express** checkbox was selected, the system dynamically added NotifyCourier() as a subscriber to OrderShipped, leading to two outputs: the usual dispatch message and an additional MessageBox stating **"Express delivery initiated!"**. This difference clearly demonstrated the successful runtime addition and removal of subscribers. Additionally, attempting to ship an order without prior confirmation produced an error message prompting users to process the order first, validating that the Boolean control flag prevented incorrect workflow execution.

Overall, the results confirm that the application behaves as intended: events are raised with proper contextual data, chained events activate in the correct sequence, conditional event subscribers behave dynamically, and GUI outputs reflect accurate system state. The contrast between valid vs. invalid orders and express vs. regular shipping illustrates strong separation of logic, flexible event handling, and robust GUI interaction. These observations collectively demonstrate a solid understanding of advanced event-driven programming in C#.

Discussion and Conclusion

During the development of the *OrderPipeline* application, several challenges were encountered, particularly related to Windows Forms Designer behavior and event wiring. One common issue involved automatically generated event handlers such as label3_Click and txtCustomer_TextChanged, which remained in the Designer file even after the corresponding methods were removed. These caused compilation errors until the invalid references were manually deleted. Another challenge was understanding how multicast delegates behave, especially when dynamically adding or removing subscribers. Addressing these issues required careful inspection of designer-generated code and a clear understanding of C# event patterns.

Completing this lab provided valuable insights into how events operate behind the scenes in C#. I gained a deeper appreciation for **event chaining**, where one event can trigger another to create multi-stage workflows. Implementing **custom EventArgs** helped reinforce how contextual data can be passed across components cleanly and safely. The exercise also emphasized the importance of **dynamic subscriber management**, particularly in GUI applications where user choices should change event-handling behavior at runtime. The difference between express and regular shipping illustrated how flexible and powerful event-driven designs can be.

Overall, this lab strengthened my understanding of advanced event-handling mechanisms and their practical applications in user interface programming. It also improved my debugging skills, especially when dealing with designer-related issues and event invocation errors. In conclusion, the lab successfully demonstrated how to design modular, interactive, and maintainable event-driven applications in C#, and provided hands-on experience with concepts that are essential for building responsive software systems.