CS202 Software Tools and Techniques for CSE
Lab Report 11: Events and Delegates in C# Windows Forms Applications

Md Sibtain Raza (22110148) | Github Link

November 23, 2025

1 Objective

The objective of this lab is to understand and implement custom events and delegates in C# Windows Forms applications using the publisher–subscriber model. Students design a small GUI named `EventPlayground` that demonstrates multi–control event interaction and multicast event handling with custom `EventArgs`.

2 Lab Requirements

- Operating System: Windows.
- IDE: Visual Studio 2022 (Community Edition) with .NET Desktop Development workload.
- Language and platform: C# on .NET (Windows Forms App).

3 Theory

C# events provide a standard mechanism for implementing the observer (publisher–subscriber) pattern. A *delegate* is a type–safe function pointer that specifies the signature of methods that may handle a given event, and an *event* is a field of delegate type that clients can subscribe to or unsubscribe from.

In Windows Forms, controls expose built–in events such as `Click`, but this lab focuses on defining custom delegates and events on the form itself and explicitly invoking them from within control event handlers. Multicast delegates allow a single event invocation to call multiple subscribers, which is used here to update a label and simultaneously show a notification message box.

4 Experiment 1: Multi–Control Event Interaction

4.1 Problem Statement

Design a Windows Forms application named `EventPlayground` with the following components:

- Two buttons: `btnChangeColor` and `btnChangeText`.
- One label: initially displaying "Welcome to Events Lab".
- One combo box with colour options: `Red`, `Green`, `Blue`.

The application must use custom delegates and events as follows:

- Clicking `btnChangeColor` raises a user–defined event `ColorChangedEvent` that changes the label's foreground colour based on the combo box selection.
- Clicking `btnChangeText` raises another custom event `TextChangedEvent` that updates the label text to the current date and time.

4.2 Design

The form defines two custom delegate types and two events:

- `ColorChangedEventHandler` for colour changes.
- `TextChangedEventHandler` for text changes.

Built–in `Click` events of the buttons are used only as publishers that invoke the custom events; all application logic resides in the subscribed handlers.

4.3 Implementation: Form Code

Form1.cs (core logic)

Listing 1: Form1.cs – custom events and handlers

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

namespace EventPlayground
{
    public partial class Form1 : Form
    {
        // Custom delegate types
        public delegate void ColorChangedEventHandler(object sender,
            ColorEventArgs e);
        public delegate void TextChangedEventHandler(object sender,
            EventArgs e);

        // Custom events
        public event ColorChangedEventHandler ColorChangedEvent;
        public event TextChangedEventHandler TextChangedEvent;

        public Form1()
        {
            InitializeComponent();

            // Initial label text
            lblDisplay.Text = "Welcome to Events Lab";

            // Populate combo box if needed
            if (cmbColors.Items.Count == 0)
            {
                cmbColors.Items.AddRange(new object[] { "Red", "Green", "
                    Blue" });
                cmbColors.SelectedIndex = 0;
            }

            // Subscribe handlers (multicast for ColorChangedEvent)
            ColorChangedEvent += UpdateLabelColor;
            ColorChangedEvent += ShowNotification;
            TextChangedEvent  += UpdateLabelText;

            // Publishers: just fire the custom events
            btnChangeColor.Click += BtnChangeColor_Click;
            btnChangeText.Click  += BtnChangeText_Click;
        }
```

```
        // Publisher for color change
        private void BtnChangeColor_Click(object sender, EventArgs e)
        {
            string selectedName = cmbColors.SelectedItem?.ToString();
            if (string.IsNullOrWhiteSpace(selectedName))
                return;

            Color clr = Color.Black;

            switch (selectedName)
            {
                case "Red":
                    clr = Color.Red;
                    break;
                case "Green":
                    clr = Color.Green;
                    break;
                case "Blue":
                    clr = Color.Blue;
                    break;
            }

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

        // Publisher for text change
        private void BtnChangeText_Click(object sender, EventArgs e)
        {
            TextChangedEvent?.Invoke(this, EventArgs.Empty);
        }

        // Subscriber 1: update label foreground colour
        private void UpdateLabelColor(object sender, ColorEventArgs e)
        {
            lblDisplay.ForeColor = e.SelectedColor;
        }

        // Subscriber 2: show message box notification
        private void ShowNotification(object sender, ColorEventArgs e)
        {
            MessageBox.Show(
                $"Color changed to {e.ColorName}",
                "Color Notification",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information);
        }

        // Subscriber: update label to current date and time
        private void UpdateLabelText(object sender, EventArgs e)
        {
            lblDisplay.Text = DateTime.Now.ToString("F");
        }
    }
}
```

5 Experiment 2: Using EventArgs and Multiple Subscribers

The second activity extends the previous design with a custom `ColorEventArgs` class and multiple subscribers to demonstrate multicast behaviour. In the implementation above, `ColorChangedEvent` passes an instance of `ColorEventArgs` to both `UpdateLabelColor` and `ShowNotification`, thus using one event invocation to trigger two logically distinct reactions.

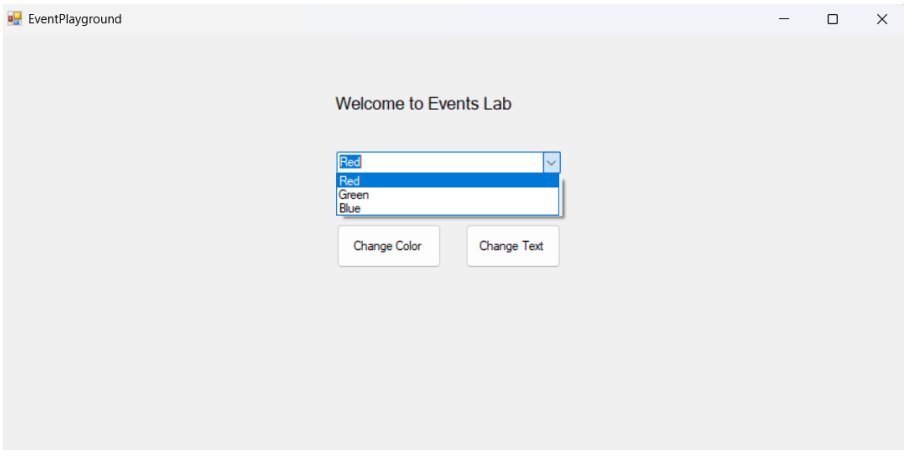
5.1 Implementation: Custom `EventArgs`

Listing 2: ColorEventArgs.cs – custom EventArgs

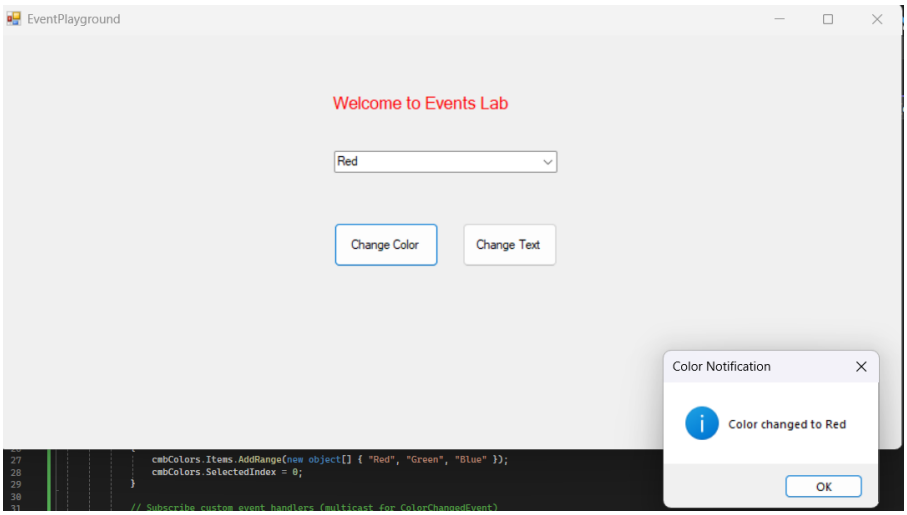
```
using System;
using System.Drawing;

namespace EventPlayground
{
    public class ColorEventArgs : EventArgs
    {
        public string ColorName { get; }
        public Color SelectedColor { get; }

        public ColorEventArgs(string colorName, Color selectedColor)
        {
            ColorName = colorName;
            SelectedColor = selectedColor;
        }
    }
}
```

6 Execution and Observations

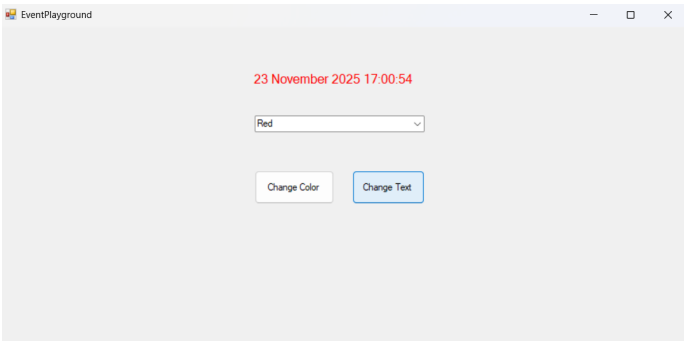
1. On startup, the label displays "Welcome to Events Lab" in the default colour, and the combo box shows the list of colours.

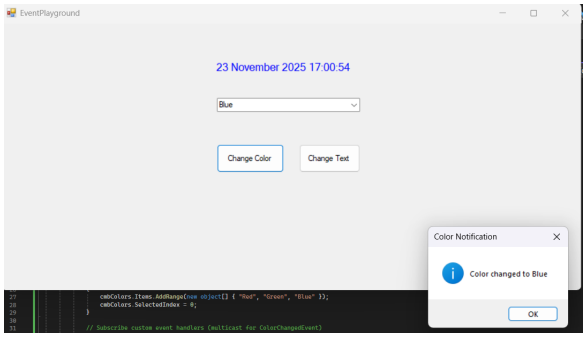


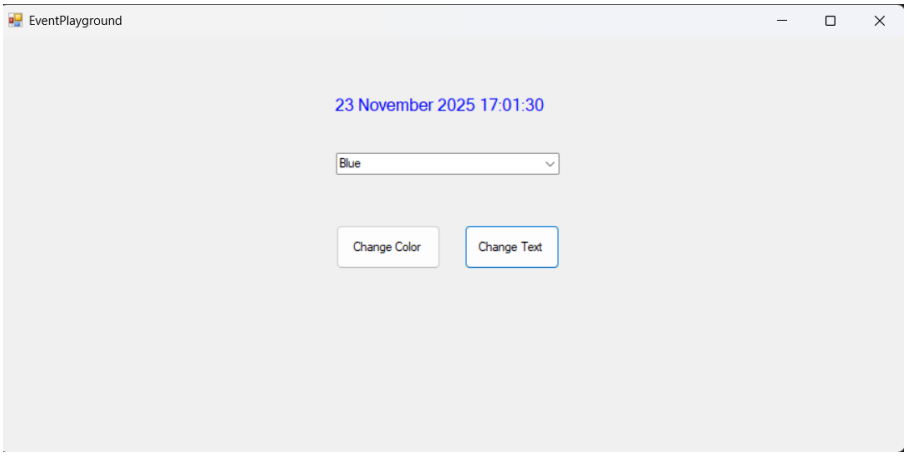
2. When a colour (e.g., `Red`) is chosen and the `Change Color` button is clicked, the form raises `ColorChangedEvent`, which simultaneously executes both subscribers:
   - `UpdateLabelColor` changes the label's `ForeColor`.
   - `ShowNotification` displays a message box indicating the selected colour.



3. When the `Change Text` button is clicked, the form raises `TextChangedEvent`, and `UpdateLabelText` sets the label text to the current date and time obtained from `DateTime.Now`.







7 Output Reasoning Tasks

This section records answers and reasoning for the delegate and event tracing problems provided in the lab sheet.

7.1 Level 0

Problem 1

For the delegate `Calc` with methods `Add`, `Mul`, and `Sub`, the final combined delegate after subscription and unsubscription is `[Mul, Sub]`. Invoking `c(2,3)` therefore prints "M" from `Mul` and "S" from `Sub`, and returns the last method's result -1, so the overall **console output** is:

```
MS:-1
```

Problem 2

For the multicast delegate `ActionHandler` with methods `Inc` and `Dec`, the integer is passed by reference and updated sequentially. Starting from 3, `Inc` makes it 5 and prints "I5 ", then `Dec` makes it 4 and prints "D4 ", and finally the main method prints "F4", yielding:

```
I5 D4 F4
```

7.2 Level 1

Problem 1: Counter with Milestones and Limits

The `Counter` class fires `MilestoneReached` on every second increment and `LimitReached` on every third increment, with multiple subscribers to `LimitReached`. Tracing values from 1 to 6 gives the single-line output:

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

Problem 2: Temperature Sensor

The sensor raises `TemperatureChanged` only when the absolute temperature difference exceeds 5°C, and the second subscriber prints a warning when the change exceeds 10°C. With calls `UpdateTemperature(28)`, `30`, `46`, `52`, only the last two increments trigger events, producing:

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

7.3 Level 2

Problem 1: Notifier with Nested Trigger

The `Notifier` object has two subscribers; the second subscriber recursively calls `Trigger("Pong")` when it receives the message "Ping". Following the nested call sequence yields the output:

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

Problem 2: Sensor with Recursive Check

The `Sensor` raises `ThresholdReached` when `value > 50`, and the first subscriber calls `Check(30)` inside the handler when the info is "High". Executing `s.Check(80)` results in a nested check and the complete output:

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

8 Conclusion

This lab reinforced the concepts of delegates and events in C# by requiring the design of a GUI that explicitly uses custom events for communication between controls, rather than relying solely on default control events. Through the practical implementation of `EventPlayground` and detailed reasoning about delegate invocation order and nested event triggering, the experiment clarified how multicast delegates, custom `EventArgs` classes, and event chaining behave in real applications.