Event Handling Mechanisms

ITCC1123- Event Driven Programming

Event arguments and delegate-based event handling.

Regular Delegates

A delegate in C# is a type-safe function pointer that holds references to methods with a specific signature and return type.

It allows you to pass methods as arguments to other methods, store methods in variables, and define callback methods

Event arguments and delegate-based event handling.

Sample

```
// Define a delegate
public delegate void PlayerActions();
```

```
// Create a function
1 reference
public void PlayerInitialize()
{
    Debug.WriteLine("Player is initialized")
}
```

Event arguments and delegate-based event handling.

Sample

```
// Point delegate into the function
PlayerActions initialization = PlayerInitialize;
```

```
// use the delegate
initialization();
```

Relation of Events and delegates in C#

In C#, an event is like a messenger that uses delegates to let one part of your program (the sender) tell other parts (the receivers) when something happens.

Workflow Behind Delegates and Events

Define the Delegate: This is like setting up the type of methods that can be referenced. The delegate specifies the return type and parameters of the methods it can point to.

Create Methods: These are the actions you want to perform, like creating buttons or setting up an fire alarm system.

Assign Methods to Delegates: You can attach the methods to the delegate, allowing the delegate to call those methods.

Workflow Behind Delegates and Events

Define an Event: Events are based on delegates and act as a broadcaster. It’s like setting up a system that notifies subscribers when something happens.

Subscribe to the Event: Other parts of the program subscribe to the event, just like how different responders (fire department, management, occupants) are prepared to react to a fire alarm.

Trigger the Event: When the event occurs (like a fire is detected), all the subscribed methods are called automatically.

Example of Events in Delegate

```
1 reference
public class Notifications
{
    public delegate void ExternalActions();
    public event ExternalActions OnExternalActionTriggered;

    1 reference
    public void NotificationTriggered()
    {
        OnExternalActionTriggered?.Invoke();
    }
}
```

Example of Events in Delegate

```
Notifications Notifications;
public delegate void PlayerActions();
1 reference
public Form1()
{
    InitializeComponent();

    PlayerActions initialize = PlayerInitialize;

    initialize();

    Notifications = new Notifications();

    Notifications.OnExternalActionTriggered += showAlert;

    Notifications.NotificationTriggered();
}
```

Custom event handlers

Custom event handlers allow you to create events that can pass specific information when they are triggered.

This involves:

1. Defining a custom EventArgs class to hold extra data.
2. Declaring an event using a delegate.
3. Subscribing to the event with methods that match the delegate's signature. Raising the event to notify subscribers

Custom event handlers

Sample

Defining a custom EventArgs class to hold extra data.

```
public class TaskEventDetails : EventArgs
{
    3 references
    public string TaskName { get; }
    2 references
    public int Status { get; }

    2 references
    public TaskEventDetails(string taskName, int isSuccessful)
    {
        TaskName = taskName;
        Status = isSuccessful;
    }
}
```

Custom event handlers

Declaring an event using a delegate.

```
public class notifications{


    public event EventHandler<TaskEventDetails> Alerted;

    1 reference
    public void showAlert(String AlertMessage)
    {
        Alerted?.Invoke(this, new TaskEventDetails(AlertMessage, 1));
    }
```

Custom event handlers

Subscribing to the event with methods that match the delegate's signature. Raising the event to notify subscribers

```
notifications notification = new notifications();
private Label lblNotification;
```

```
1 reference
public void notification_Alerted(object sender, TaskEventDetails e)
{
    MessageBox.Show(e.TaskName, "Alert", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
}
```

```
notification.Alerted += notification_Alerted;
```

Custom event handlers

```
lblNotification = new Label
{
    Text = "Show Alerts",
    Location = new System.Drawing.Point(200, 50),
    AutoSize = false
};

lblNotification.Click += (sender, e) => notification.showAlert("This is a UI alert!");
Controls.Add(lblNotification);
```

Multi-control event handling.

In C#, you are able to bind the same event handler to multiple controls

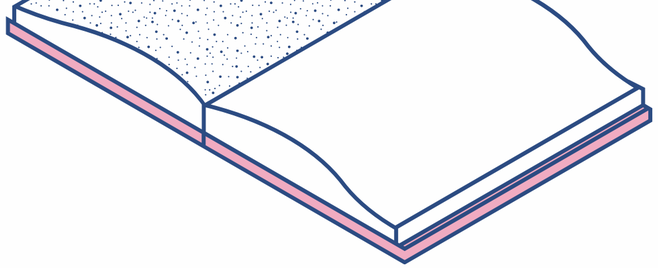
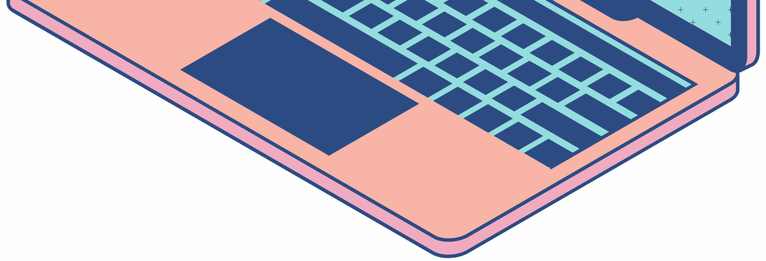
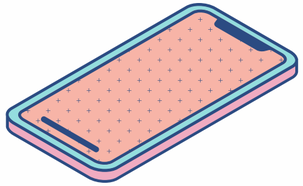
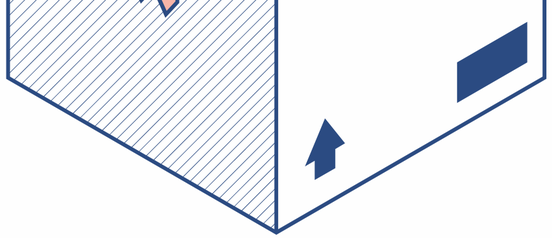
```
private Label lblNotification;
private Button ProgButton;
```

```
lblNotification = new Label
{
    Text = "Show Alerts",
    Location = new System.Drawing.Point(200, 50),
    AutoSize = false
};

ProgButton = new Button
{
    Text = "Show Alert",
    Location = new System.Drawing.Point(500, 50),
    AutoSize = false
};

lblNotification.Click += (sender, e) => notification.showAlert("This is an alert from label!");
ProgButton.Click += (sender, e) => notification.showAlert("This is an alert from Button");

Controls.Add(lblNotification);
Controls.Add(ProgButton);
```

The End