

# 01\_Event-Driven Programming

**Event-driven programming** is a programming paradigm where the program reacts to **events**, such as user actions, sensor inputs, or system-generated signals. Instead of following a strict sequence of commands, the program listens for events and responds when they occur.

---

## Core Concepts

### 1. Events

- Actions or occurrences that the program can respond to.
- Examples: Button clicks, keyboard input, mouse movement.

### 2. Event Handlers

- Functions or methods that execute in response to events.

### 3. Event Loop

- A continuous loop that listens for events and triggers their respective handlers when events occur.
- 

## Key Components

### 1. Basic Event-Driven Program

This example demonstrates a simple event-driven system with custom events and handlers.

```
# Simple Event-Driven Example
class Event:
    def __init__(self, name):
        self.name = name

# Event handlers
def handle_event_1(event):
    print(f"Handling event: {event.name}")

def handle_event_2(event):
    print(f"Handling another event: {event.name}")

# Main event loop
def event_loop(events):
    for event in events:
        if event.name == "event_1":
            handle_event_1(event)
        elif event.name == "event_2":
            handle_event_2(event)

# Example usage
events = [Event("event_1"), Event("event_2"), Event("event_1")]
event_loop(events)
```

## Output:

```
Handling event: event_1
Handling another event: event_2
Handling event: event_1
```

## 2. Keyboard Input Event Handling

Using Python's built-in `keyboard` library to respond to keypresses.

```
import keyboard # Install with: pip install keyboard

def on_key_event(event):
```

```
print(f"Key {event.name} pressed!")

# Attach the event handler
keyboard.on_press(on_key_event)

# Keep the program running to listen for events
print("Press any key (Ctrl+C to exit)")
keyboard.wait() # Blocks and listens for events
```

## What Happens:

- Whenever a key is pressed, the `on_key_event` function is executed.

## 3. Timer-Based Event Handling

This example uses the `threading` library to trigger events based on a timer.

```
import threading

# Event handler
def on_timer_event():
    print("Timer event triggered!")

# Set up a repeating timer
def start_timer():
    threading.Timer(2.0, start_timer).start() # Triggers every 2 seconds
    on_timer_event()

start_timer()
```

## What Happens:

- The program triggers `on_timer_event` every 2 seconds.

## 4. Event Handling with PyQt

PyQt is another popular library for GUI development. It relies on **signals** and **slots** for event handling.

```
from PyQt5.QtWidgets import QApplication, QPushButton, QLabel, QVBoxLayout, QWidget

def on_button_click():
    label.setText("Button clicked!")

app = QApplication([])
window = QWidget()
layout = QVBoxLayout()

label = QLabel("Click the button!")
button = QPushButton("Click Me")
button.clicked.connect(on_button_click) # Connect signal to handler

layout.addWidget(label)
layout.addWidget(button)
window.setLayout(layout)
window.show()
app.exec_()
```

## What Happens:

- Clicking the button emits a signal, triggering the `on_button_click` handler to update the label.

## 5. AR Foundation Plane Detection Event

For AR Foundation, use `ARPlaneManager` to detect planes in an AR session.

```
using UnityEngine;
using UnityEngine.XR.ARFoundation;

public class ARPlaneDetection : MonoBehaviour
{
```

```
[SerializeField] private ARPlaneManager planeManager;

void OnEnable()
{
    planeManager.planesChanged += OnPlanesChanged;
}

void OnDisable()
{
    planeManager.planesChanged -= OnPlanesChanged;
}

private void OnPlanesChanged(ARPlanesChangedEventArgs args)
{
    foreach (var plane in args.added)
    {
        Debug.Log($"Plane added: {plane.trackableId}");
    }
}
}
```

## What Happens:

- **Event:** Plane detection event in AR Foundation.
- **Handler:** `OnPlanesChanged` is executed whenever a plane is added, updated, or removed.

# Best Practices for Event-Driven Programming in Unity

1. **Use Built-in Events Where Possible:**
  - Leverage Unity's `UnityEvent`, UI events, and physics events instead of reinventing the wheel.
2. **Avoid Overusing Global Events:**

- Delegate-based or static events are powerful but can lead to tight coupling and difficulty debugging.

### 3. **Unsubscribe When Done:**

- Always unsubscribe from events to avoid memory leaks or unintended behavior.

```
void OnDisable()
{
    myButton.onClick.RemoveListener(OnButtonClick);
}
```



### 4. **Debugging:**

- Use logs or breakpoints to verify that your events are being triggered and handled correctly.

### 5. **Combine with Coroutines:**

- For delayed or time-based responses to events, pair event handlers with Unity's coroutines.

**Unsubscribing from events** in Unity (or in C# in general) applies only to the **specific listeners** (event handlers) you explicitly unsubscribe. It does not globally remove all listeners from the event.

Here's a breakdown:

---

# How Event Unsubscription Works

## 1. Only Affects Subscribed Handlers

When you unsubscribe from an event, you only remove the **specific handler** (method or delegate) you subscribed to it. Other handlers subscribed to the same event remain unaffected.

### **Example:**

```
using System;
using UnityEngine;

public class EventUnsubscribeExample : MonoBehaviour
{
```

```
public static Action OnCustomEvent;

void Start()
{
    // Subscribe two different handlers to the same event
    OnCustomEvent += HandlerOne;
    OnCustomEvent += HandlerTwo;

    // Invoke the event
    OnCustomEvent?.Invoke();

    // Unsubscribe only HandlerOne
    OnCustomEvent -= HandlerOne;

    // Invoke the event again
    OnCustomEvent?.Invoke();
}

void HandlerOne()
{
    Debug.Log("Handler One called.");
}

void HandlerTwo()
{
    Debug.Log("Handler Two called.");
}
}
```

## Output:

```
Handler One called.
Handler Two called.
Handler Two called.
```

- The first invocation calls both `HandlerOne` and `HandlerTwo`.
- After unsubscribing `HandlerOne`, only `HandlerTwo` is called in the second invocation.

## 2. Why Unsubscription Is Important

### Memory Leaks

If an object subscribes to an event but is not unsubscribed before the object is destroyed, it may cause memory leaks because the event keeps a reference to the object, preventing garbage collection.

**Example:**

```
void OnEnable()
{
    SomeEventManager.OnGameEvent += HandleGameEvent;
}

void OnDisable()
{
    SomeEventManager.OnGameEvent -= HandleGameEvent; // Unsubscribe to prevent memory leaks
}
```

### Avoiding Unexpected Behavior

If you don't unsubscribe properly, the event may trigger a handler for an object that is no longer relevant or expected to respond.

## 3. Applying to Unity Events

For Unity's **UnityEvent** system, you must unsubscribe the same way to remove a specific listener.

**Example:**

```
using UnityEngine;
using UnityEngine.Events;
```



```
public class UnityEventExample : MonoBehaviour
{
    public UnityEvent myUnityEvent;

    void Start()
    {
        myUnityEvent.AddListener(EventHandlerOne);
        myUnityEvent.AddListener(EventHandlerTwo);

        myUnityEvent.Invoke(); // Calls both handlers

        myUnityEvent.RemoveListener(EventHandlerOne); // Unsubscribe EventHandlerOne

        myUnityEvent.Invoke(); // Calls only EventHandlerTwo
    }

    void EventHandlerOne()
    {
        Debug.Log("EventHandlerOne triggered.");
    }

    void EventHandlerTwo()
    {
        Debug.Log("EventHandlerTwo triggered.");
    }
}
```

## 4. Common Mistakes

### A. Unsubscribing from Non-Subscribed Handlers

If you try to unsubscribe a handler that isn't subscribed, nothing happens. C# handles this gracefully without throwing an error.

### Example:

```
OnCustomEvent -= NonExistentHandler; // No error, but nothing is removed
```

## 5. Does This Apply to All Events?

Yes, the principle of unsubscribing applies to all types of events:

- **Unity Built-In Events:** e.g., `Button.onClick`, `Input`, collision events.
- **Custom Events:** Events you create using `Action`, `UnityEvent`, or `delegate`.

You must explicitly unsubscribe only the handlers you no longer need.

## Best Practices for Unsubscribing

1. **Always Unsubscribe When Appropriate:**
  - Use `OnDisable` or `OnDestroy` to clean up event subscriptions.
2. **Track Subscriptions:**
  - Keep a clear record of what you've subscribed to and ensure they're unsubscribed when no longer needed.
3. **Use Weak References if Necessary:**
  - For advanced use cases, consider weak references to avoid holding strong references to objects, preventing memory leaks.
4. **Avoid Excessive Global Subscriptions:**
  - Too many global events can make debugging complex. Use scoped or localized events where possible.

Would you like further guidance on managing event subscriptions or specific examples in Unity? ☐

---

# Building a Simple Event-Driven Framework

Here's how you can build a small event-driven framework in Python:

```
class EventManager:
    def __init__(self):
        self.listeners = {}

    def subscribe(self, event_name, handler):
        if event_name not in self.listeners:
            self.listeners[event_name] = []
        self.listeners[event_name].append(handler)

    def emit(self, event_name, *args, **kwargs):
        if event_name in self.listeners:
            for handler in self.listeners[event_name]:
                handler(*args, **kwargs)

# Example usage
def on_custom_event(data):
    print(f"Custom event received with data: {data}")

event_manager = EventManager()
event_manager.subscribe("custom_event", on_custom_event)
event_manager.emit("custom_event", data="Hello, World!")
```

## Output:

# Advantages of Event-Driven Programming

1. **Modularity:**
    - Event handlers can be written as independent, reusable functions or modules.
  2. **Responsiveness:**
    - Ideal for interactive applications where user input or external events dictate program behavior.
  3. **Scalability:**
    - Can easily add more event types or handlers without major changes to the main program.
- 

# Challenges in Event-Driven Programming

1. **Debugging:**
    - The flow of execution is non-linear, making it harder to trace bugs.
  2. **Performance:**
    - Poorly designed event handlers or excessive events can degrade performance.
  3. **State Management:**
    - Ensuring consistency across multiple event handlers requires careful planning.
- 

Would you like more advanced examples or help applying event-driven programming in a specific context, like AR, gaming, or data pipelines? ☐

---

Revision #6

Created 29 December 2024 22:10:21 by victor

Updated 29 December 2024 23:07:11 by victor