01_Programming Paradigms

- **Event-Driven Programming**:
  - Essential for tool and UI development (e.g., PyQt, AR Foundation).
- **Object-Oriented Programming (OOP)**:
  - Helps with modularity and reusability in tools, game objects, and pipelines.
- **Declarative Programming**:
  - Useful for shaders, procedural workflows, and material systems.
- **Functional Programming**:
  - Great for mathematical operations in shaders and procedural generation.
- **Data-Driven Programming**:
  - Ideal for tool and pipeline flexibility

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:

```
Custom event received with data: Hello, World!
```

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? □□

02_Object Oriented Programming

03_Declarative Programming

04_Functional Programming

Functional programming (FP) is a programming paradigm focused on writing software by composing and applying **pure functions**, avoiding shared state, and minimizing side effects. It's particularly well-suited for **mathematical computations**, **data transformations**, and scenarios requiring **parallel processing**.

Key Principles of Functional Programming

1. Pure Functions

- A **pure function** is a function where:
  - The output depends only on its inputs.
  - It has no side effects (doesn’t modify external state).

Example:

```
# Pure function
def add(a, b):
    return a + b
```

- **Not Pure** (has side effects):

```
result = 0


def add(a, b):
    global result
    result = a + b
    return result
```

Why It Matters:

- Easier to debug: The function's behavior is predictable and testable.
- Parallelization: Pure functions can be executed independently.

---

2. Immutability

- **Data is not modified** after it is created.
- Instead of changing data, new data structures are created.

Example:

```
# Immutable transformation
numbers = [1, 2, 3]
new_numbers = [x * 2 for x in numbers]
```

Why It Matters:

- Reduces bugs caused by unexpected state changes.
- Makes reasoning about program behavior easier.

---

3. Higher-Order Functions

- Functions that take other functions as arguments or return functions.

Example:

```
# Map applies a function to each element
numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

Why It Matters:

- Encourages reusability and modularity by composing small, reusable functions.

---

4. First-Class Functions

- Functions are treated like data: They can be passed as arguments, returned from other functions, and assigned to variables.

Example:

```
def greet(name):
    return f"Hello, {name}!"


def execute(func, arg):
    return func(arg)


print(execute(greet, "Alice"))  # Output: "Hello, Alice!"
```

Why It Matters:

- Enables concise and expressive code.

---

5. Recursion

- Instead of loops, functional programming often uses **recursion** to repeat operations.

Example (factorial with recursion):

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n - 1)


print(factorial(5))  # Output: 120
```

Why It Matters:

- Recursion avoids mutable state and aligns with the FP principle of immutability.

---

6. Lazy Evaluation

- Computation is deferred until the result is actually needed.
- Common in FP languages like **Haskell**, but supported in Python via generators.

Example:

```
# Lazy evaluation with a generator
def generate_numbers():
    for i in range(10):
        yield i

numbers = generate_numbers()
for num in numbers:
    print(num)  # Generates each number one at a time
```

Why It Matters:

- Optimizes performance by avoiding unnecessary computations.
- Handles large or infinite data structures efficiently.

---

7. Function Composition

- Combine smaller functions to build more complex functions.

Example:

```
def double(x):
    return x * 2

def square(x):
    return x ** 2

def compose(f, g):
    return lambda x: f(g(x))

double_then_square = compose(square, double)
print(double_then_square(3))  # Output: 36
```

Why It Matters:

- Encourages modular and reusable code.

Advantages of Functional Programming

1. **Predictable Code**:
    - Pure functions ensure the output is consistent, making debugging easier.
2. **Concurrency and Parallelism**:
    - No shared state or side effects mean functions can run independently.
3. **Modularity**:
    - Encourages writing small, reusable functions that can be combined in powerful ways.
4. **Testability**:
    - Pure functions are easy to test as they don't depend on external states.
5. **Immutable Data**:
    - Reduces bugs caused by unexpected changes to shared data.

Disadvantages of Functional Programming

1. **Learning Curve**:
    - The paradigm requires a shift in thinking for those used to procedural or object-oriented programming.
2. **Performance**:
    - Immutable data structures can sometimes lead to higher memory usage and slower performance compared to mutable ones.
3. **Debugging Recursion**:
    - Heavy reliance on recursion can lead to stack overflow errors if not optimized (e.g., via tail recursion).
4. **Limited Libraries**:
    - Some libraries or APIs are built with OOP in mind and may not work well with FP.

Functional Programming in Popular Languages

Functional-First Languages:

- **Haskell**: Purely functional, lazy evaluation.
- **Erlang**: High concurrency and reliability.

Functional Features in Multi-Paradigm Languages:

1. **Python**:
    - Supports functional constructs like `map`, `filter`, `lambda`, and comprehensions.
    - Example:

    ```
    nums = [1, 2, 3, 4]
    squares = list(map(lambda x: x ** 2, nums))
    print(squares)  # Output: [1, 4, 9, 16]
    ```

2. **JavaScript**:
    - Functional tools like `reduce`, `map`, and `filter`.
    - Example:

    ```
    const nums = [1, 2, 3, 4];
    const squares = nums.map(x => x ** 2);
    console.log(squares);  // Output: [1, 4, 9, 16]
    ```

3. **C++**:
    - Lambdas and standard functional algorithms in the STL (`std::transform`, `std::accumulate`).
    - Example:

    ```
    #include <vector>
    #include <algorithm>
    #include <iostream>

    int main() {
    ```

```
    std::vector<int> nums = {1, 2, 3, 4};
    std::transform(nums.begin(), nums.end(), nums.begin(), [](int x) { return x * x; });
    for (int n : nums) std::cout << n << " ";  // Output: 1 4 9 16
}
```

Applications of Functional Programming

1. **Graphics Programming**:
   - Procedural texture generation and transformations (e.g., GLSL shaders).
   - Functional paradigms simplify operations on immutable data like pixels or vertex buffers.
2. **Data Processing**:
   - Big data frameworks like Apache Spark rely on FP for parallelism and immutability.
3. **Game Development**:
   - Functional constructs help build procedural systems like terrain generation or AI logic.
4. **Concurrency**:
   - Functional programming is ideal for writing highly concurrent and parallel systems due to immutability.

Would you like more hands-on examples in Python or another language, or a deeper dive into functional constructs? □□