Java Events

Table of contents

Introduced in 2026-05-14-lambdas-streams-and-events[1] (Guest lecture, Week 11).

Event-driven programming

A common design pattern for controlling program flow — a different way to glue chunks of code together than direct method calls. Code (often a lambda, see java-lambdas-and-streams[2]) **subscribes** as a listener for specific events from specific sources; when triggered, the event is dispatched to every subscribed listener. E.g. clicking a button spawns a click event, which is passed to anything listening for clicks on that button. Extremely common in GUI programming (Java's Swing and JavaFX both have built-in event systems) and network programming (Node.js is event-driven under the hood).

A well-designed event system lets each part of a system stay **decoupled** (see java-cohesion-and-coupling[3]) — a chunk of code doesn't need to know about the irrelevant details of whatever eventually reacts to the event it raises, only about the event itself.

The event loop

The event system itself is usually run on a separate thread, or via a task-prioritisation architecture, executing an **event loop**: a program that repeatedly takes an event off a queue and notifies every subscribed listener, dispatching the event's information to each of them. Many environments provide an event loop for you to hook into (JavaScript has one built into the language itself).

[1] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[2] courses/csse2002/java-lambdas-and-streams.pdf
[3] courses/csse2002/java-cohesion-and-coupling.pdf

Building a toy event system

A minimal event system needs only two classes plus lambdas:

```
class SimpleEvent {
    private String type;
    private String data;

    public SimpleEvent(String type, String data) {
        this.type = type;
        this.data = data;
    }

    public String getType() { return type; }
    public String getData() { return data; }
}
```

```
class EventSystem {
    // addListener(type, action): subscribe a Consumer<SimpleEvent> to a named event
     ↪  type.
    public Consumer<SimpleEvent> addListener(String type, Consumer<SimpleEvent>
     ↪  action) { ... }

    // removeListener(action): unsubscribe a previously-added listener.
    public void removeListener(Consumer<SimpleEvent> action) { ... }

    // addEvent(event): queue an event to be dispatched on the next tick().
    public void addEvent(SimpleEvent event) { ... }

    // tick(): the event loop step - dispatch queued events to their listeners.
    public void tick() { ... }
}
```

`SimpleEvent` is just a named bundle of data (`type` + `data`); `EventSystem` maintains the queue of pending events and the map of listeners per event type. Calling `tick()` is what actually drives the event loop — it drains the queue, and for each event, calls every `Consumer<SimpleEvent>` subscribed to that event's `type`, passing the event itself as the argument.

Motivating example: the CSSE2002 game engine

Without events, the game engine's `tick()` method is the *only* mechanism for passing state between parts of the program — every parent tile/manager ticks its children, explicitly passing state down through the whole hierarchy. This tightly links everything around that single `tick()` call. Introducing an event system lets far-apart parts of the program communicate

directly through events instead, cutting down substantially on this “connective tissue” code that exists purely to shuttle state through layers that don’t otherwise care about it.