Java Events

Table of contents

Event-driven programming 1
Building a toy event system 2

Introduced in 2026-05-14-lambdas-streams-and-events[1] (Guest lecture, Week 11). Not examinable — see the note at the top of that lecture.

Event-driven programming

A common design pattern for **how to control the flow of your program**: basically a different way to glue together your various chunks of code.

You **subscribe** chunks of your code (often lambdas, see java-lambdas-and-streams[2]) to a *listener* for specific events **emanating from specific sources**; when the event is triggered, the subscribed code is activated. For example, clicking a button spawns a click event, which is then passed to anything subscribed to / listening for the click of *that* button.

The payoff is that the system becomes more discrete: each chunk does not need to know about other irrelevant details. That is low coupling, see java-cohesion-and-coupling[3].

It is incredibly common in two places:

- **GUI programming** — if you have done any GUI work on the web or with a desktop toolkit you have already used events.
- **Network programming** — Node.js, the JavaScript server runtime, is event-driven under the hood.

[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

The event loop

The event system itself is generally run on a separate thread, or via a task-prioritisation (macro vs micro) architecture, and what it runs is an **event loop**:

> An event loop is a program that takes an event from a queue of events, and alerts all subscribed listeners that this event has occurred, dispatching the event information to them.

Mostly you hook into one rather than writing it: many applications ship an event loop for you to use as a developer, and some languages build one into the language itself — **JavaScript is the iconic example**. In Java, **Swing and JavaFX both come with their own event systems** you can hook into.

Building a toy event system

Events are easy to pick up and use. They are often the first design pattern people meet, at high school, without realising it, when they wire up buttons on a basic website. Using them does not show the *internals*. The two things you would not naturally pick up:

1. an implementation of a **basic, naive event system**, so you can vaguely understand how they work under the hood;
2. a variation of the **CSSE2002 game engine** with a basic event system integrated, to show how that simplifies existing code on the developer's end.

The API

A basic event system needs only lambdas and 2 classes:

```
SimpleEvent (String type, String data):  getType():String, getData():String

EventSystem
    addListener(String type, Consumer<SimpleEvent> action): action
    removeListener(Consumer<SimpleEvent> action)
    addEvent(SimpleEvent event): void
    tick(): void
```

`SimpleEvent` is just a **named bundle of data** — a `type` string that says what happened and a `data` string carrying the payload, with a getter for each. The signatures give `EventSystem` no fields. Reading the four method names against the definition of an event loop above, it needs two pieces of state: a **queue** of events not yet dispatched, and a **map** from event type to the listeners subscribed to it.

Two details worth reading off the signatures:

- **`addListener` returns the `action` back to you.** The natural reading is that it hands you a handle to unsubscribe with later: if you passed the listener in as an inline lambda you have no other reference to it, and `removeListener` needs exactly that reference.
- **`removeListener` is the one method with no return type given.** Every other method has one (`:String`, `: action`, `: void`), so treat its return as unspecified rather than assuming `void`.

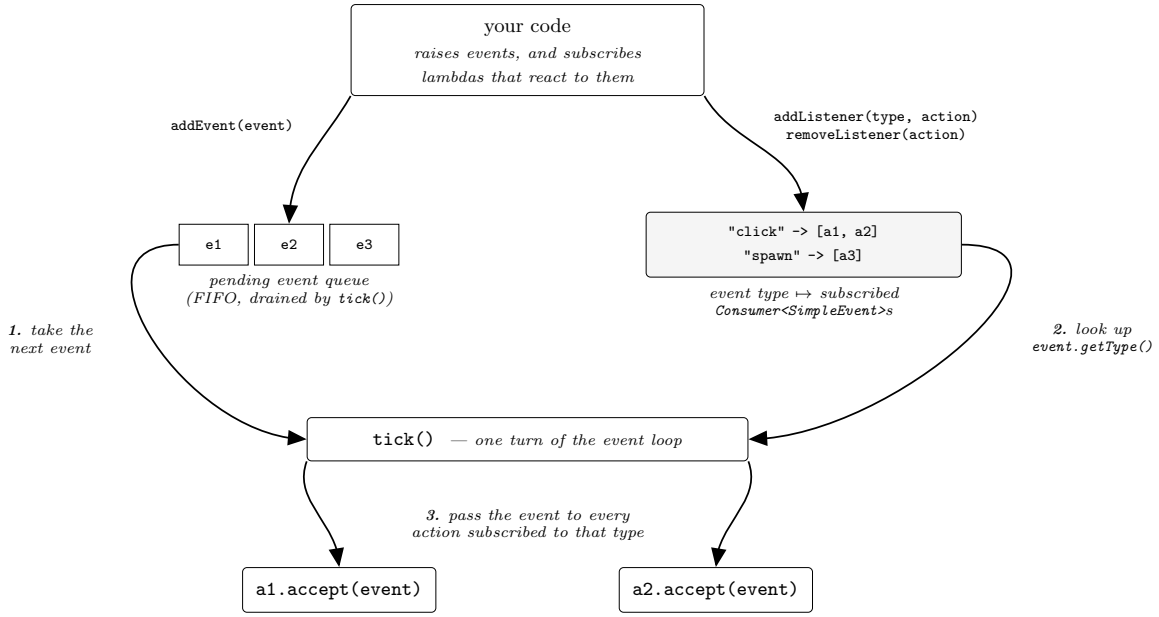

Figure 1: How the four EventSystem methods fit together: a queue of pending events, a map from event type to the listeners subscribed to it, and the numbered steps `tick()` takes to dispatch one event.

On the same reading, `tick()` is what actually drives the loop: it takes an event off the queue, looks up `event.getType()` in the listener map, and calls each subscribed `Consumer<SimpleEvent>` with the event itself as the argument. So nothing happens between ticks, and queuing an event only schedules it.

> **SimpleEvent written out**
>
> Written out as a class, the `SimpleEvent` signature is:

```java
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; }
}
```

Motivating example: the CSSE2002 game engine

The engine you used in the assignments is **very linear**: you called `.tick()`, passed in the various states, and then the parent tiles and managers would tick *their* children, passing the state down accordingly.

That works, but it **tightly links everything around that `tick()` method** as the mechanism for state being passed between the various parts of the program. An event system drastically cuts down on this connective tissue code: the code that exists only to shuttle state through layers that do not otherwise care about it. The saving comes from letting far-apart parts talk through the event system rather than through their common ancestor.

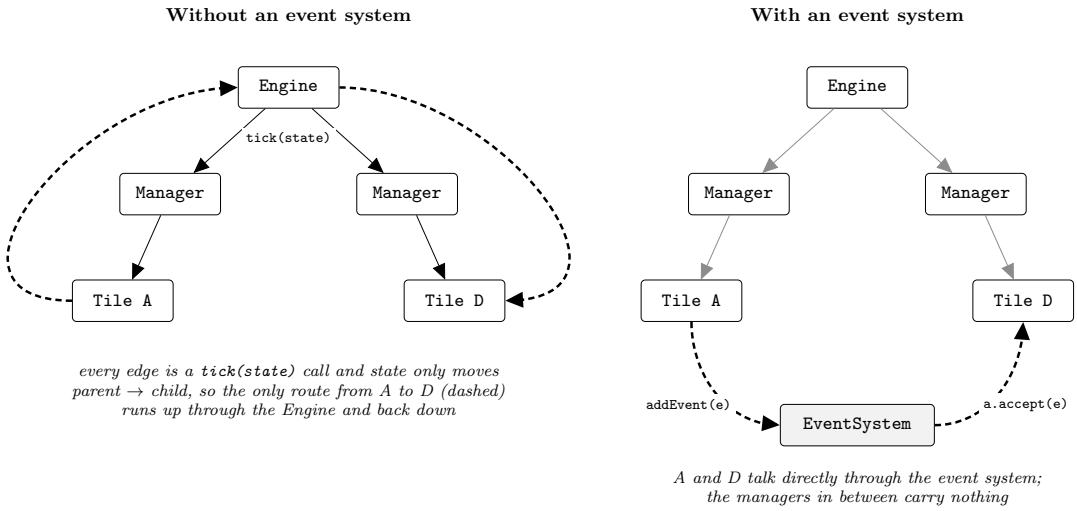

Figure 2: Why the game engine is the motivating example. Without an event system, parent tiles and managers tick their children and pass state down, so the only route from Tile A to Tile D runs up through the Engine and back down. With one, A and D talk through the EventSystem and the managers in between carry nothing.