Java Lambdas and Streams

Table of contents

Lambdas (anonymous functions) 1
Streams 2

Introduced in 2026-05-14-lambdas-streams-and-events[1] (Guest lecture, Week 11). Practised further in week12-tutorial-lambdas-and-streams[2].

Lambdas (anonymous functions)

A lambda is a **function as a value** — it isn't declared with a name, and can be passed around like any other reference value, such as an object. JavaScript and Python have the same idea.

Scope: like in most languages, a Java lambda **can hold references to things accessible in scope at the time of creation**.

Functional interfaces

Because Java is typed, you cannot have an anonymous function without clearly defined type information — the JVM would not know what to do. So Java supplies a set of interfaces that are **combined with generics** (see java-generics[3]) to give easily-adjustable type descriptions for lambdas. They live in `java.util.function`. Four of them:

[1] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[2] courses/csse2002/week12-tutorial-lambdas-and-streams.pdf
[3] courses/csse2002/java-generics.pdf

Interface	Signature	Purpose
`Consumer<T>`	takes a `T`, returns nothing (`void`)	by convention indicates the function has some kind of side effect — if it returned nothing *and* had no side effect it would do nothing at all
`Predicate<T>`	takes a `T`, returns `boolean`	useful in filtering especially
`Function<T, R>`	takes a `T`, returns an `R`	turning a `T` into an `R`
`BiFunction<T, U, R>`	takes a `T` and a `U`, returns an `R`	deriving one value from two, e.g. the distance between two positions, or custom sorting

A minimal lambda of each shape:

```
Consumer<String> print = s -> System.out.println(s);
Predicate<Integer> isEven = n -> n % 2 == 0;
Function<String, Integer> length = s -> s.length();
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
```

`Consumer<T>` is the one that matters most for events: the toy event system in java-events[4] types every listener as a `Consumer<SimpleEvent>`.

Streams

Added in **Java 8** to enable a different, more **declarative** style of programming. Brian Goetz's IBM article on Java streams is a very good resource on their benefits.

This style is particularly effective when you have data you need to conduct multiple operations on sequentially, or retrieve a single resulting value from a sequence of operations, which is often the case in a real job. At a simple level they are an alternative to writing lots and lots of very samey `for` loops over a sequential set of data (most often a `List` or similar, see java-collections-framework[5]); the JavaScript equivalents are `.filter()`, `.map()` and `.reduce()`.

The worked example

One pipeline, taken a line at a time:

[4] courses/csse2002/java-events.pdf
[5] courses/csse2002/java-collections-framework.pdf

```
int totalHpFromFireEnemies =
    enemies.stream()
        .filter(enemy -> enemy.faction.equals("Fire"))
        .mapToInt(enemy -> enemy.hp)
        .sum();
```

Stage	Line	What it does
`.stream()`	`enemies.stream()`	opens a stream over the collection
`.filter()`	`.filter(enemy -> enemy.faction.equals("Fire"))`	drops every enemy whose faction isn't `Fire`
`.mapToInt()`	`.mapToInt(enemy -> enemy.hp)`	replaces each surviving `Enemy` with its `int hp`
`.sum()`	`.sum()`	reduces what is left to a single total

Anatomy of a stream pipeline

A pipeline has three steps:

1. **Start** a stream with `.stream()`.
2. Chain **any number of intermediate operations**: `.filter()` | `.map()` | `.distinct()` | `.sorted()` | `.mapToInt()` etc.
3. **Resolve** it into a final value with one of the **terminal operations**: `.sum()` | `.count()` | `.toList()` | `.collect()` | `.reduce()`.

You *chain* the first kind and *resolve* with the second. The rule underneath: an intermediate operation hands on another stream, a terminal one resolves to a value. That is what puts `.mapToInt()` in the first list and `.sum()` in the second, and why exactly one terminal operation ends a pipeline.

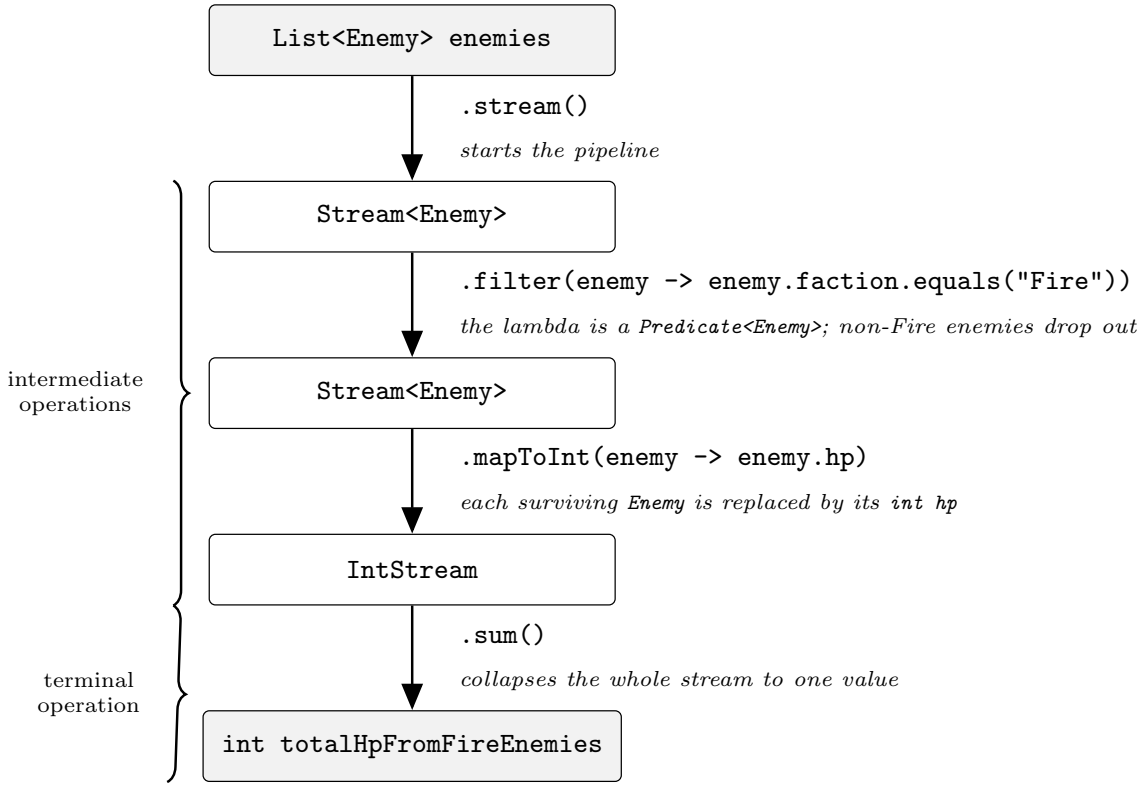

Figure 1: The stream example as a pipeline, one stage per line of the chain. Each box is the type between stages: `.stream()` turns the list into a `Stream<Enemy>`, `.filter()` takes a `Predicate` and hands on a `Stream<Enemy>`, `.mapToInt()` hands on an `IntStream`, and `.sum()` collapses it to one `int`. The braces mark the intermediate operations and the terminal operation.