Java Lambdas and Streams

Table of contents

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

Lambdas (anonymous functions)

A lambda is a function treated as a value — it isn't declared with a name, and can be passed around like any other reference value (similar to anonymous functions in Python or JavaScript).

Scope: like in most languages, a Java lambda can hold references to things accessible in scope at the point it's created.

Functional interfaces

Because Java is statically typed, a lambda can't exist without a clearly-defined type — the JVM needs to know its shape. Java provides a family of generic functional interfaces (in `java.util.function`) to describe common lambda shapes; four commonly used ones:

Interface	Signature	Purpose
`Consumer<T>`	takes a `T`, returns nothing	has some side effect (returning nothing and having no side effect would mean doing nothing at all)
`Predicate<T>`	takes a `T`, returns `boolean`	useful for filtering
`Function<T, R>`	takes a `T`, returns an `R`	general transform

[1] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf

Interface	Signature	Purpose
`BiFunction<T, U, R>`	takes a `T` and a `U`, returns an `R`	deriving one value from two, e.g. distance between two positions, or a custom comparator/sort key

```
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;
```

Streams

Added in Java 8 to enable a more **declarative** style of programming — particularly effective when you need to run multiple operations over data sequentially, or reduce a sequence of operations down to a single resulting value (a very common real-world task). Streams are Java's answer to the same idea as JavaScript's `.filter()`/`.map()`/`.reduce()`: an alternative to writing many near-identical `for` loops.

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

Anatomy of a stream pipeline

1. **Start** a stream from a collection: `.stream()`.
2. Chain any number of **intermediate operations**, each producing a new stream: `.filter()`, `.map()`, `.distinct()`, `.sorted()`, `.mapToInt()`, …
3. **Resolve** the stream into a final value with exactly one **terminal operation**: `.sum()`, `.count()`, `.toList()`, `.collect()`, `.reduce()`, …

In the example above: `enemies.stream()` starts the pipeline; `.filter(...)` keeps only `Fire`-faction enemies; `.mapToInt(...)` transforms each remaining `Enemy` into its `int hp`; `.sum()` (terminal) reduces the stream of `hp` values down to a single total.