CSSE2002 — Week 11 Notes

Table of contents

Lambdas, Streams and Events **1**
- Topics . . . 2
- Events . . . 2
- Applied class . . . 2
- Aside: anecdotal local industry survey . . . 2

Dependency Inversion **3**
- Dependencies . . . 3
- Dependency Inversion . . . 4
- Dependency Injection . . . 9

Reference material **10**
- Java Events . . . 10
- Java Lambdas and Streams . . . 13
- Java SOLID Principles . . . 17

Lambdas, Streams and Events

See csse2002[1] for course logistics.

> **Lambdas are examinable; event-driven programming and GUIs are not** — confirmed in 2026-05-28-course-summary-and-exam-preparation[2]. The two halves carry very different revision weight.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/2026-05-28-course-summary-and-exam-preparation.pdf

Topics

1. **Lambdas in Java** — anonymous functions, scope, and the functional interfaces that give them a type. Examples: an anonymous function in JS, in Python and in Java; what a lambda captures from its enclosing scope; filtering with a `Predicate`.
2. **Streams in Java** — a basic `.stream()` example, then one worked pipeline, built up one line at a time.
3. **Events** — including how to make your own toy event system.
4. An anecdotal survey of QLD dev teams (see below).

All the technical content lives in java-lambdas-and-streams[3] and java-events[4].

Events

Using an event API is easy to pick up. It is often the first design pattern people use at high school, wiring up buttons on a website, without realising it. Two things are not picked up naturally:

1. a basic, naive **event system implementation**, which shows what is going on under the hood;
2. a variant of the **CSSE2002 game engine** (the engine used in the assignments) with an event system integrated, which shows how much "connective tissue" code it removes.

Both are written up in java-events[5].

Applied class

See week11-tutorial-dependency-inversion[6] — more practice with Dependency Inversion, extending java-solid-principles[7] (Week 7).

Aside: anecdotal local industry survey

An informal, anonymised survey of one engineer's professional network: **14 QLD-based dev teams** (13 excluding the surveyor's own) across **12 distinct organisations**. It is not representative data. It reflects only one person's connections in the dev scene, with *no* QLD teams in big tech, consultancies or banks. The findings (on AI tool usage and graduate hiring) are not course content.

[3] courses/csse2002/java-lambdas-and-streams.pdf
[4] courses/csse2002/java-events.pdf
[5] courses/csse2002/java-events.pdf
[6] courses/csse2002/week11-tutorial-dependency-inversion.pdf
[7] courses/csse2002/java-solid-principles.pdf

Dependency Inversion

Applied class for 2026-05-14-lambdas-streams-and-events[8] (Week 11). More practice applying Dependency Inversion and Dependency Injection (see java-solid-principles[9], where the principle itself is stated).

(The handout is headed *Week 10*. It is the week 11 applied class.)

Dependencies

> A class `A` has a **dependency** on class `B` if `A` refers to `B` in code — an `import B;`, or (if in the same package) simply referring to `B` anywhere in the code.

```
import furniture.Lamp;

class Desk {
    HardwoodFloor placedOn;
    List<Junk> items = new ArrayList<>();
    LogitechKeyboard keyboard = new LogitechKeyboard();

    public Desk() {
        placedOn = new HardwoodFloor();
        items.add(new Book("Pragmatic Programmer"));
        items.add(new Monitor());
    }

    public void rotate(Angle angle) {
        if (angle == Angle.LEFT) {
            placedOn.scratch("clockwise");
        }
        // implementation
    }

    public boolean clean() {
        // implementation
    }
}
```

List the classes `Desk` depends upon.

[8] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[9] courses/csse2002/java-solid-principles.pdf

> **Working**
>
> `Lamp`, `HardwoodFloor`, `Junk`, `List`, `ArrayList`, `Book`, `Monitor`, `Angle`, `LogitechKeyboard`.

Not all dependencies are equally bad. Two properties determine how good/bad a dependency is:

- **Stability** — how likely the dependency is to change. `ArrayList` is a good dependency because it's very unlikely to change; in general, treat classes *you* write as unstable until proven otherwise. Interfaces are usually assumed more stable than concrete classes (unless they're poorly designed and change often).
- **Exposure** — how much of the class uses the dependency. `Book` (used only in the constructor) is a better dependency than `HardwoodFloor` (used in any method) simply because less of the class is entangled with it.

Roughly order `Desk`'s dependencies from best to worst.

> **Working**
>
> 1. `List`
> 2. `ArrayList`
> 3. `Lamp` (assuming that it isn't used anywhere)
> 4. `Book`
> 5. `Monitor`
> 6. `Angle`
> 7. `LogitechKeyboard`
> 8. `HardwoodFloor`
> 9. `Lamp` (assuming that it can be used anywhere)
>
> One caveat: since `Angle` appears to be an `enum` type, we may reasonably assume it is stable and may consider it a better dependency.

`Lamp` appears twice because it is ranked under both assumptions: the `import` is there but no use of `Lamp` is visible in the listing, so its exposure is either nil (best) or unbounded (worst). The caveat makes `Angle` a better dependency, which would move it up from sixth.

Dependency Inversion

Dependency inversion is a tool for building software that is more flexible and amenable to future changes.

Dependency Inversion Principle (DIP): depend on abstractions, not concretions (implementations).

To apply it to `Desk` you first need to know which of its dependencies already have an abstraction behind them. In the diagram, a dashed box indicates an interface.

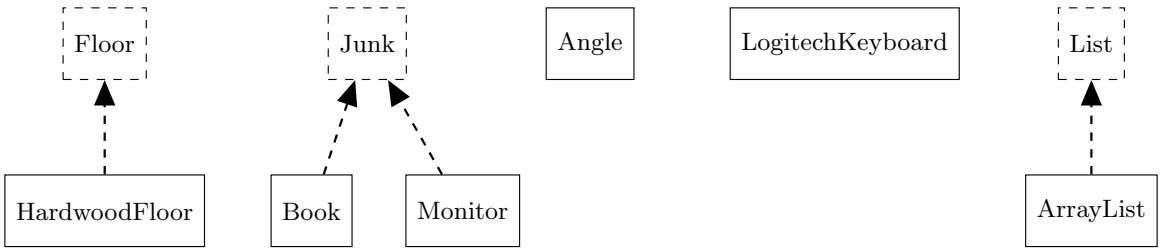


Figure 1: The structure behind Desk's dependencies. It carries eight of the nine dependencies listed above - Lamp is absent - plus the new Floor interface. A dashed box indicates an interface. A dashed line with a solid filled arrowhead runs from a concrete class up to the interface it implements.

Read off the diagram: `HardwoodFloor`, `Book`, `Monitor` and `ArrayList` each already sit under an interface, so the simple form of the DIP applies to them immediately. `Angle` and `LogitechKeyboard` sit alone, with no abstraction to switch to. `LogitechKeyboard` is the one fixed below, with the proper form of the DIP.

Minimising concrete dependencies (the simple form)

The simplest step towards DIP: change a field's compile-time type from a concrete class to whatever interface it implements (assuming the interface exposes everything the field is used for):

```
- HardwoodFloor placedOn;
+ Floor placedOn;
```

```
public class Warrior {
    private BronzeSword weapon = new BronzeSword();
    private BronzeShield shield = new BronzeShield();

    public void attack(Opponent opponent) { weapon.use(opponent); }
    public void defend(Attack attackType) { shield.block(attackType); }
}
```

Minimise concrete dependencies for `Warrior`, based on the class diagram below.

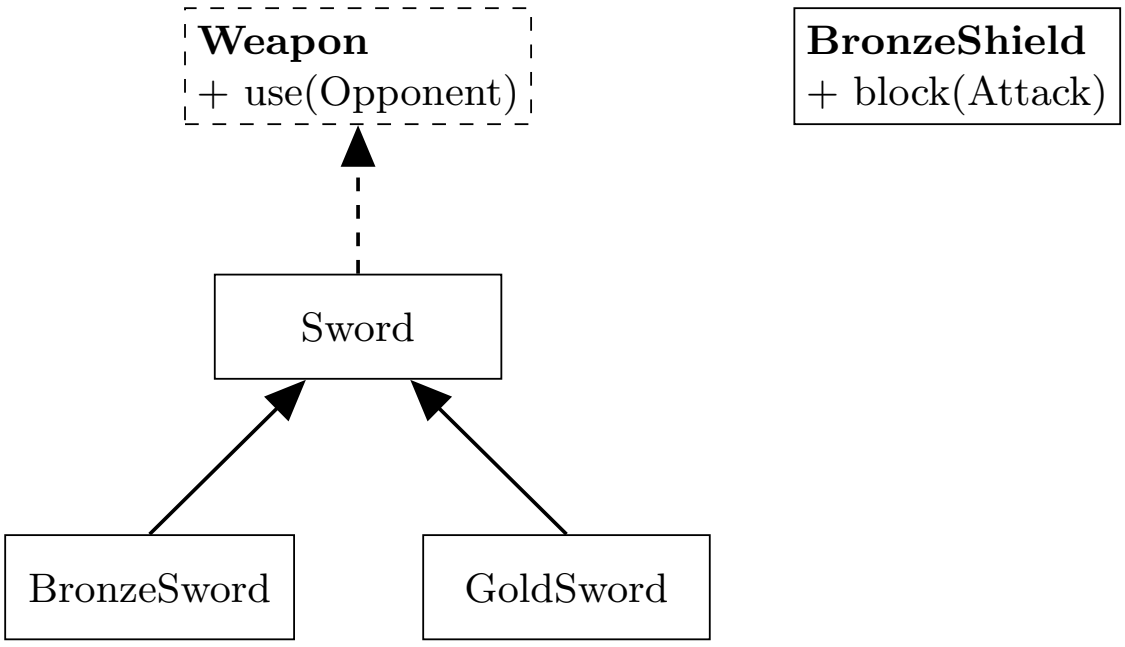


Figure 2: The class diagram for Task 2: dashed box = interface (Weapon), solid box = concrete class, dashed arrow = implements, solid arrow = extends, arrowheads solid-filled and the supertype above. Weapon and BronzeShield are the only two boxes that list a member, on the line below the name. Those two members are the only two methods Warrior actually calls.

`BronzeSword` has `Sword` above it and `Weapon` above that; `BronzeShield` stands alone, with no interface to retreat to.

> **i** Working

```
- private BronzeSword weapon = new BronzeSword();
+ private Weapon weapon = new BronzeSword();
```

The solution touches only `weapon`; `shield` has to wait for the proper form below.

The proper form of DIP

Minimising concrete dependencies only helps where an abstraction already exists. The *proper* form of DIP is:

> Remove dependencies on low-level components from high-level components — make **both** depend on an abstraction.

High-level vs low-level isn't a strict distinction: high-level components implement application logic (e.g. a browser's back/forward navigation history); low-level components do the grunt work the high-level logic depends on (e.g. actually requesting a webpage's data). The goal is for high-level components to talk to low-level components *exclusively* through an abstraction, so a low-level component changing doesn't force a change to the high-level component depending on it.

Applying this to `Desk`'s `LogitechKeyboard` dependency (`LogitechKeyboard` has no existing interface to fall back on):

1. **Create an abstraction** of the low-level component:

```
interface Keyboard {
    void type(char key);
    // include other ways to interface with the low-level component
    // multiple interfaces can be used if the low-level component
    // has multiple responsibilities
}
```

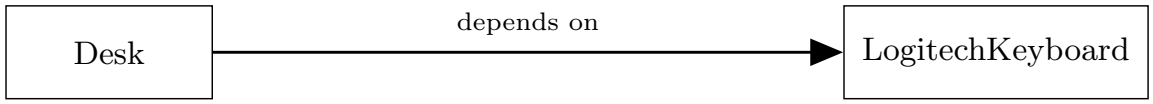
2. **Modify the low-level component** to depend on (implement) the abstraction:

```
- class LogitechKeyboard {
+ class LogitechKeyboard implements Keyboard {
```

3. **Minimise concrete dependencies** in the high-level component:

```
- LogitechKeyboard keyboard = new LogitechKeyboard();
+ Keyboard keyboard = new LogitechKeyboard();
```

The name of the principle is in what those three steps do to the arrows:



Before: *the high-level class names the low-level one*



After: *both ends point at the abstraction — the right-hand edge has reversed*

Figure 3: What is inverted in dependency inversion, for the Desk-to-LogitechKeyboard example. A dashed box is an interface and a dashed arrow with a filled head means implements. A depends-on edge is a solid line with a solid head, pointing from the depender to what it depends upon.

`Desk` still depends on something, and `LogitechKeyboard` is still the thing that runs, yet the edge that used to run *from* the high-level component *to* the low-level one now runs the other way — from the low-level component up to the shared abstraction. That reversal is the "inversion".

Apply the proper form of DIP to `Warrior`'s `BronzeShield` dependency.

Working

```
interface Shield {
    void block(Attack attack);
}
```

```
- class BronzeShield {
+ class BronzeShield implements Shield {
```

```
- private BronzeShield shield = new BronzeShield();
+ private Shield shield = new BronzeShield();
```

Dependency Injection

Even after minimising concrete dependencies, `Desk` still depends on the concrete types when *constructing* them. **Dependency Injection (DI)** completes DIP: provide concrete instances to a class as parameters, rather than having the class instantiate them itself:

```
- public Desk() {
-     placedOn = new HardwoodFloor();
+ public Desk(Floor floor) {
+     placedOn = floor;
```

`Desk` no longer needs to change to support a different floor type — the caller just passes a different `Floor` implementation in. A constructor isn't the only injection point — a setter is appropriate when the dependency is likely to change over the object's lifetime:

```
public changeKeyboard(Keyboard keyboard) {
    this.keyboard = keyboard;
}
```

Modify `Warrior` so its concrete types are dependency injected.

> **i** Working
>
> ```
> public class Warrior {
>     private Weapon weapon;
>     private Shield shield;
>
>     public Warrior(Weapon weapon, Shield shield) {
>         this.weapon = weapon;
>         this.shield = shield;
>     }
>
>     // it's quite likely that a warrior will change their weapon
>     public void equip(Weapon weapon) { this.weapon = weapon; }
>
>     // overloading gives us a nice consistent interface for this action
>     public void equip(Shield shield) { this.shield = shield; }
>
>     public void attack(Opponent opponent) { weapon.use(opponent); }
>     public void defend(Attack attackType) { shield.block(attackType); }
> }
> ```

`weapon` and `shield` are injected via the constructor *and* can be swapped later through the overloaded `equip(...)` setters — setter injection layered on top of constructor injection.

Dependency injection frameworks

Something (typically an entry-point method, e.g. `main`) still has to construct the concrete dependencies before injecting them. DI frameworks let you defer that construction to runtime, specifying which concrete classes to build via the framework's own configuration rather than in code. Unless a project needs multiple *levels* of injected dependencies, good design alone can avoid the extra conceptual overhead of adopting a DI framework.

Reference material

Java Events

Introduced in 2026-05-14-lambdas-streams-and-events[10] (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[11]) 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[12].

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.

[10] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[11] courses/csse2002/java-lambdas-and-streams.pdf
[12] 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`.

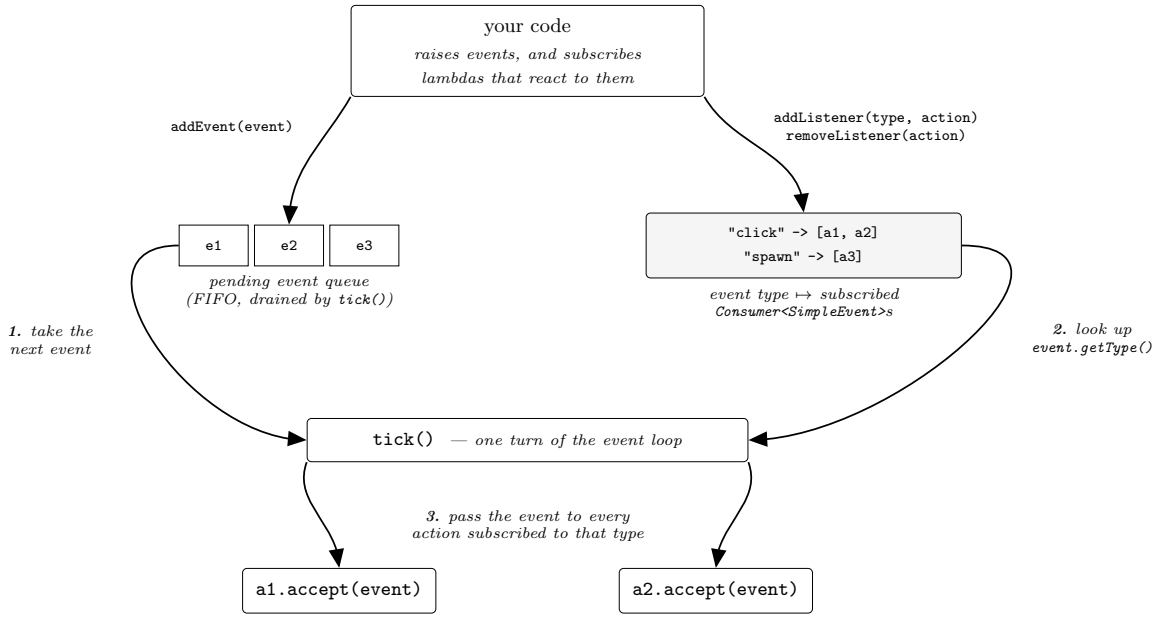


Figure 4: How the four EventSystem methods fit together: a queue of pending events, a map from event type to the listeners subscribed to it, and the numbered steps `tick()` takes to dispatch one event.

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.

Java Lambdas and Streams

Introduced in 2026-05-14-lambdas-streams-and-events[13] (Guest lecture, Week 11). Practised further in week12-tutorial-lambdas-and-streams[14].

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**.

[13] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[14] courses/csse2002/week12-tutorial-lambdas-and-streams.pdf

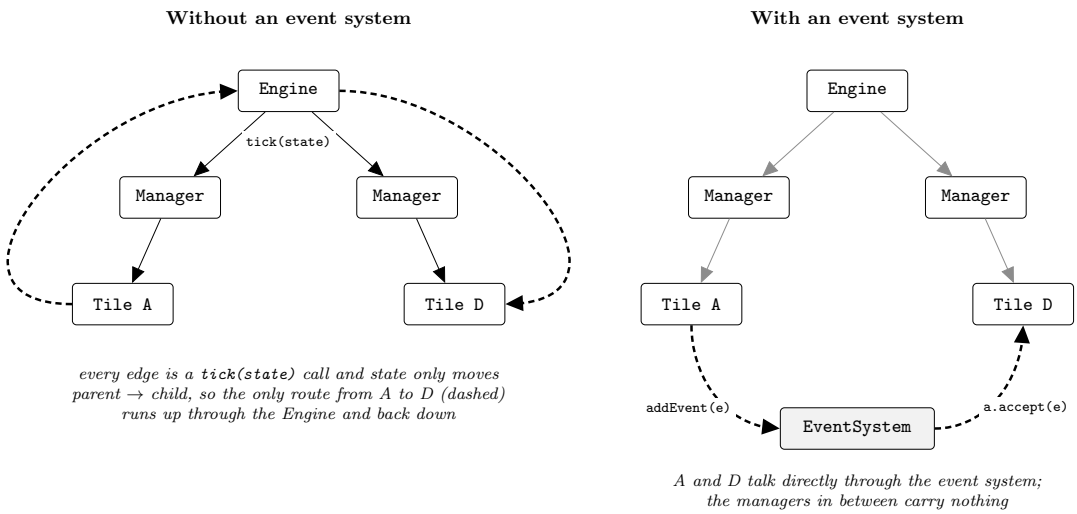


Figure 5: Why the game engine is the motivating example. Without an event system, parent tiles and managers tick their children and pass state down, so the only route from Tile A to Tile D runs up through the Engine and back down. With one, A and D talk through the EventSystem and the managers in between carry nothing.

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[15]) to give easily-adjustable type descriptions for lambdas. They live in `java.util.function`. Four of them:

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:

[15] courses/csse2002/java-generics.pdf

```
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[16] 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[17]); the JavaScript equivalents are `.filter()`, `.map()` and `.reduce()`.

The worked example

One pipeline, taken a line at a time:

```
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

[16] courses/csse2002/java-events.pdf
[17] courses/csse2002/java-collections-framework.pdf

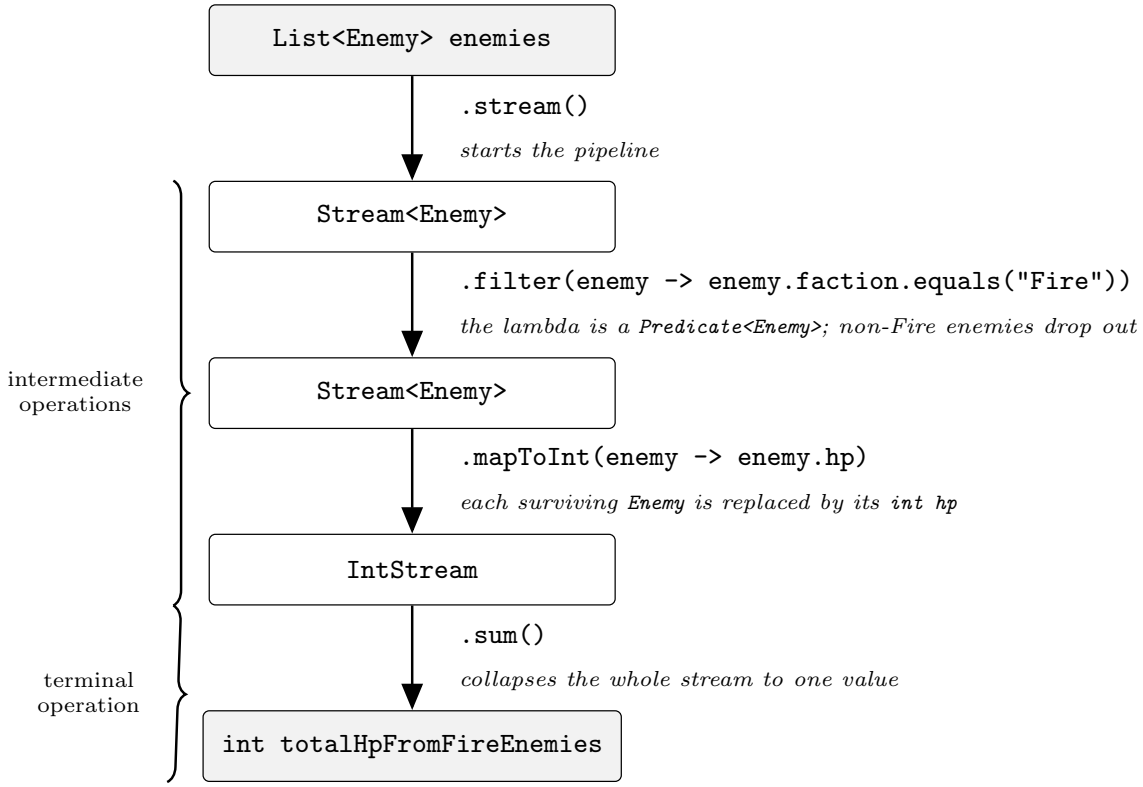


Figure 6: The stream example as a pipeline, one stage per line of the chain. Each box is the type between stages: `.stream()` turns the list into a `Stream<Enemy>`, `.filter()` takes a `Predicate` and hands on a `Stream<Enemy>`, `.mapToInt()` hands on an `IntStream`, and `.sum()` collapses it to one `int`. The braces mark the intermediate operations and the terminal operation.

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.

Java SOLID Principles

Introduced in 2026-04-02-refactoring[18] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[19] (Lecture, Week 7 — Liskov Substitution, Interface Segregation, Dependency Inversion).

Why SOLID?

Large software tends to become:

- **Rigid** — difficult to change; even small changes require modifications across many parts of the codebase.
- **Fragile** — likely to break when changed, sometimes in parts of the codebase that appear unrelated to the change.

The SOLID principles (S-ingle responsibility, O-pen-closed, L-iskov substitution, I-nterface segregation, D-ependency inversion) are a set of guidelines — not rules — to help prevent rigidity and fragility. Many of the individual ideas predate the acronym; Robert Martin combined them in *Design Principles and Design Patterns* (2000).

Week 6 covers the first two: **Single Responsibility** and **Open-closed**.

[18] courses/csse2002/2026-04-02-refactoring.pdf
[19] courses/csse2002/2026-04-16-solid-principles-ii.pdf

Single Responsibility Principle (SRP)

> There should never be more than one reason for a class to change.

Closely related to java-cohesion-and-coupling[20]'s cohesion: cohesion is about how *similar* a class's functionality is, while SRP is about the *reasons for change*. A "responsibility" is a reason for change — if you can think of more than one motive to change a class, it has more than one responsibility.

```
class Student {
    private String name;
    private List<Course> courses;
    private double gpa;

    public void enrollInCourse(Course course) { courses.add(course); }
    public void calculateGPA() { /* ... */ }
    public void printTranscript() { /* ... */ }
}
```

This class has three reasons to change:

1. Changing how to enrol to courses.
2. Changing how overall GPAs are calculated.
3. Changing how transcripts are generated.

Splitting each concern into its own class gives each one a single responsibility:

```
class Student {
    private String name;
    private List<Course> courses;
    public void enrollInCourse(Course course) { courses.add(course); }
    public List<Course> getCourses() { return courses; }
}
class GPACalculator {
    public double calculateGPA(List<Course> courses) { /* ... */ }
}
class TranscriptPrinter {
    public void printTranscript(Student student) { /* ... */ }
}
```

Two details of the split are worth noticing. The `gpa` field is gone: it was a *cached result* of one of the three responsibilities, so it leaves with the responsibility, and `GPACalculator.calculateGPA` returns a value instead of assigning one. And `Student` gains `getCourses()`, because the two

[20] courses/csse2002/java-cohesion-and-coupling.pdf

extracted classes now need the data from outside — the responsibilities separate cleanly, but they do not become independent.

The same shape, stated more briefly:

```
// Before                          // After
class Report {                     class ReportCalculator { }
    void calculateData() {}        class ReportSaver { }
    void saveToFile() {}           class ReportPrinter { }
    void printReport() {}
}
```

Calculating, saving and printing are three different reasons to change one `Report` class — a new output format, a new storage backend and a new formula are unrelated events — so they become three classes.

Homework. Check whether SRP improves memory management *indirectly*, by improving overall design structure — think about object creation, immutability (see java-mutability[21]) and garbage collection.

Open-Closed Principle (OCP)

> Components should be **open for extension** but **closed for modification**.

Introduced by Bertrand Meyer in *Object Oriented Software Construction* (1988): "if the open-closed principle is applied well, then further changes are achieved by adding new code, not by changing old code that already works".

- **Open for extension** — a class's behaviour can be extended to support changing requirements.
- **Closed for modification** — no one may change the behaviour of an existing class; it's closed once it's available for other modules to depend on.

A class is kept closed for modification by information hiding (see java-encapsulation[22]).

Example: pluggable grading systems

The example picks up the `GPACalculator` extracted under SRP above, which assumes a single grading scheme:

[21] courses/csse2002/java-mutability.pdf
[22] courses/csse2002/java-encapsulation.pdf

```
class GPACalculator {
    public double calculateGPA(List<Course> courses) {
        double totalPoints = 0;
        for (Course course : courses) {
            totalPoints += course.getGradePoints();
        }
        return totalPoints / courses.size();
    }
}
```

Now a requirement arrives: the university needs to handle different grading systems, such as letter grades (A, B, C) and numerical grades (90, 80, 70). A first attempt branches on a `gradeType` string:

```
class GPACalculator {
    public double calculateGPA(List<Course> courses, String gradeType) {
        double totalPoints = 0;
        if (gradeType.equals("Letter")) {
            for (Course course : courses) {
                totalPoints += convertLetterGradeToPoints(course.getLetterGrade());
            }
        } else if (gradeType.equals("Numeric")) {
            for (Course course : courses) {
                totalPoints += course.getNumericGrade();
            }
        }
        return totalPoints / courses.size();
    }

    private double convertLetterGradeToPoints(String letterGrade) {
        // Convert letter grade to GPA points logic
        return 0;
    }
}
```

Three things carry the problem: the added `String gradeType` parameter, and the two string literals `"Letter"` and `"Numeric"`. They are every place that has to be *modified* the next time a grading system is added, in a class that was supposed to be finished. The class is open for extension only by editing it, which is exactly what "closed for modification" forbids. (A `gradeType` string also reintroduces the `Vehicle`-style type switch from java-refactoring[23].)

Extracting the grading logic behind an interface makes `GPACalculator` open for extension (new `GradingSystem` implementations) without ever touching its own code again:

[23] courses/csse2002/java-refactoring.pdf

```java
interface GradingSystem {
    double getGradePoints(Course course);
}
class LetterGradingSystem implements GradingSystem {
    @Override
    public double getGradePoints(Course course) { /* convert letter grade to points
     ↪  */ return 0; }
}
class NumericGradingSystem implements GradingSystem {
    @Override
    public double getGradePoints(Course course) { return course.getNumericGrade(); }
}

class GPACalculator {
    private GradingSystem gradingSystem;
    public GPACalculator(GradingSystem gradingSystem) { this.gradingSystem =
     ↪  gradingSystem; }

    public double calculateGPA(List<Course> courses) {
        double totalPoints = 0;
        for (Course course : courses) {
            totalPoints += gradingSystem.getGradePoints(course);
        }
        return totalPoints / courses.size();
    }
}
```

The caller picks the grading system and hands it to the calculator:

```java
public static void main(String[] args) {
    List<Course> letterGradeCourses = new ArrayList<>();
    letterGradeCourses.add(new Course("CS", "A"));

    GradingSystem letterGradingSystem = new LetterGradingSystem();
    GPACalculator gpaCalculator = new GPACalculator(letterGradingSystem);
    System.out.println("GPA (Letter Grades): " +
     ↪  gpaCalculator.calculateGPA(letterGradeCourses));
}
```

What the refactoring bought:

- `GPACalculator` depends on the abstraction `GradingSystem` instead of calculating grade points itself — which makes it open for extension (new grading systems) but closed for modification.
- More grading systems can be added without changing the core logic of `GPACalculator`.

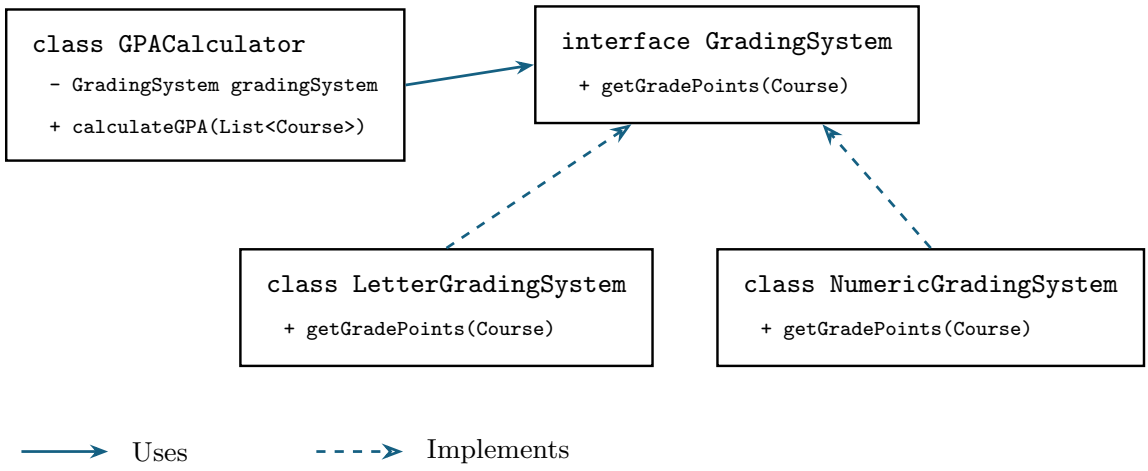


Figure 7: The open-closed refactoring of GPACalculator. GPACalculator uses only the GradingSystem interface; each concrete grading system implements it. A new scheme — pass/fail, say — is a fourth box on the bottom row with a fourth dashed arrow, and no existing box changes: that is 'open for extension, closed for modification' as a picture. Each label carries the Java keyword and members are listed as '+ m()'. A solid line with a solid head means 'uses', a dashed line with an open head means 'implements', and every arrow points from the depender to the thing depended upon. This is not UML, which would draw realisation as a hollow triangle.

- The interface guarantees every grading system implements the same method (`getGradePoints()`), making it easy to switch between them.
- `main` demonstrates using either `LetterGradingSystem` or `NumericGradingSystem` by passing a different grading system to the `GPACalculator`.

Two arrows into an abstraction that nobody edits is also how the class stays closed: the calculator is closed for modification because the *only* thing it can see of a grading system is the interface. That is information hiding used structurally, and it is the same move as the Dependency Inversion Principle below; the constructor parameter here is dependency injection.

Liskov Substitution Principle (LSP)

> Subclasses should be substitutable for their parent classes.

Defined by Barbara Liskov in a 1987 keynote: an instance of a subclass type can be used wherever an instance of a superclass type is expected, *without breaking the correctness of the program*:

```
List<Animal> animals = new ArrayList<>();
animals.add(new Cat());
animals.add(new Mouse());
for (Animal animal : animals) {
    animal.eat(new Food());
}
```

Example violation

```
class Student {
    protected List<Course> courses;
    public double calculateGPA() {
        double totalPoints = 0;
        for (Course course : courses) {
            totalPoints += course.getGradePoints();
        }
        return totalPoints / courses.size();
    }
}
class PostgraduateStudent extends Student {
    @Override
    public double calculateGPA() {
        throw new UnsupportedOperationException("Postgraduates have a different GPA
        ↪  system.");
    }
```

```
}
```

Code written against `Student` (e.g. calling `calculateGPA()` on every student in a list) will blow up the moment a `PostgraduateStudent` is substituted in — `PostgraduateStudent` is not actually usable wherever a `Student` is expected, so it violates LSP.

Substitution with contracts

Even when a subclass overrides a method with a different implementation, it must still honour the parent's contract (see java-specification[24]):

```
class Parent {
    /**
     * @requires x > 0
     * @ensures \result > 'A' && \result < 'Z'
     */
    char f(int x);
}
```

Preconditions — suppose an overriding `Child.f()` can also handle negative numbers. Changing its precondition from `x > 0` to `x != 0` is fine: every input `Parent.f()` accepted (`x > 0`) is still accepted by `Child.f()` (`x != 0`), since $x > 0 \Rightarrow x \neq 0$ — so the precondition became *weaker* (less strict), not stronger. **Preconditions in a subclass must be no stronger than the superclass's — they may be weaker or stay the same.**

Postconditions — suppose `Child.f()` only ever returns `'K'`, `'L'`, or `'M'`. A postcondition of `\result > 'J' && \result < 'N'` is fine: every result still satisfies the parent's promise (`\result > 'J' && \result < 'N' \Rightarrow \result > 'A' && \result < 'Z'`), so the postcondition became *stronger* (more restrictive), which is allowed. **Postconditions in a subclass must be no weaker than the superclass's — they may be stronger or stay the same.**

> A subclass must not strengthen preconditions or weaken postconditions — it should accept everything the parent accepts, and still guarantee everything the parent guarantees.

Worked example

[24] courses/csse2002/java-specification.pdf

```java
class One {
    /**
     * @require row != null && col != null && 0 < val && val < 100
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in either row or col
     */
    public List<Integer> calc(int[] row, int[] col, int val) { ... }
}
class Two extends One {
    /**
     * @require row != null && col != null && 0 < val &&
     *     val < 100 && row only contains positive integers
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in either row or col
     */
    @Override
    public List<Integer> calc(int[] row, int[] col, int val) { ... }
}
```

`Two`'s precondition adds an extra requirement (`row only contains positive integers`) that `One` never required — so `One`'s precondition does *not* imply `Two`'s: there are inputs `One` would accept (a `row` containing a negative number) that `Two` rejects. This **strengthens** the precondition and violates LSP: code written against `One` that happens to pass a negative-containing `row` would break if a `Two` were substituted in.

Homework variant: if `Two`'s postcondition is instead changed to return integers appearing in **both** `row` and `col` (rather than *either*), does this satisfy LSP? Reasoning it through: the guarantee changes from a union to an intersection, which for most inputs returns strictly *fewer* elements than `One` promised — that's a **weaker** postcondition (some outputs a caller was guaranteed under `One` are no longer guaranteed under `Two`), which also violates LSP.

Interface Segregation Principle (ISP)

> Many client-specific interfaces are better than one general-purpose interface.

Shrink interfaces to minimize *incidental dependencies* — large interfaces should be split into smaller ones, so implementing/using classes only need to be concerned about the methods that actually interest them.

```java
class Z {
    public void a() {...}
    public void b() {...}
    public void c() {...}
    public void d() {...}
```

```
    public void e() {...}
}
// A only uses a() and b(), B only uses c(), C only uses d() and e() ...
// ...but all three classes still depend on the whole of Z.
```

This mirrors java-cohesion-and-coupling[25]'s stamp coupling at the interface level: despite each client using only a small subset of `Z`, every client depends on *all* of `Z` — so a change to any part of `Z` risks affecting (and forcing a recompile/redeploy of) every client, even ones that never used the changed part. Splitting `Z`'s surface into focused interfaces removes those incidental dependencies:

```
interface AI { void a(); void b(); }
interface BI { void c(); }
interface CI { void d(); void e(); }
// Z implements AI, BI, CI; A depends only on AI, B only on BI, C only on CI.
```

Dependency Inversion Principle (DIP)

> Depend upon abstractions. Do not depend upon concretions.

Entities must depend on abstractions, not concretions: high-level modules must not depend on low-level modules — both should depend on abstractions. If `A` depends directly on the concrete class `B`, changes to `B`'s implementation can propagate to `A`; if `A` instead depends only on an abstraction (`BSpec`) that `B` implements, changes inside `B` are far less likely to affect `A`, as long as `BSpec` itself stays the same.

Even a small step towards this helps — programming against the `List` interface instead of the concrete `ArrayList` type means `Student` no longer depends on which list implementation the caller chose:

```
class Student {
    private List<Course> courses; // not ArrayList<Course>
    public Student(List<Course> courses) { this.courses = courses; }
    public double calculateTotalGradePoints() { ... }
}
```

What makes a good dependency? (stability and exposure)

A class `A` has a **dependency** on class `B` if `A` refers to `B` in code (an `import B;`, or, in the same package, simply referring to `B` anywhere). Not all dependencies are equally risky — two properties determine how good/bad one is:

[25] courses/csse2002/java-cohesion-and-coupling.pdf

- **Stability** — how likely the dependency is to change. `ArrayList` is a good dependency because it's very unlikely to change; treat classes *you* write as unstable until proven otherwise, and generally assume interfaces are more stable than concrete classes (unless they're poorly designed and change often).
- **Exposure** — how much of the depending class is entangled with it. A dependency only used in a constructor is better than one used across every method, since less of the class would need to change if that dependency changed.

Two forms of DIP

Minimising concrete dependencies (the simple form) — change a field's compile-time type from a concrete class to whatever interface it already implements, wherever one already exists:

```
- HardwoodFloor placedOn;
+ Floor placedOn;
```

This only helps where an abstraction already exists, though. The **proper form of DIP** goes further:

> Remove dependencies on low-level components from high-level components — make *both* depend on an abstraction.

"High-level" vs "low-level" isn't a strict distinction: high-level components implement application logic (e.g. a browser's back/forward navigation history); low-level components do the grunt work that logic depends on (e.g. actually requesting a webpage's data). The goal is for high-level components to talk to low-level ones *exclusively* through an abstraction, so a low-level component changing doesn't force the high-level component depending on it to change too. Applied to a `Desk` class depending on a concrete `LogitechKeyboard` with no existing interface:

1. **Create an abstraction** of the low-level component: `interface Keyboard { void type(char key); ... }`.
2. **Modify the low-level component** to implement it: `class LogitechKeyboard implements Keyboard`.
3. **Minimise concrete dependencies** in the high-level component: `Keyboard keyboard = new LogitechKeyboard();`.

Dependency Injection

Dependency Injection (DI) is a design pattern that implements DIP by injecting a class's dependencies into it (e.g. via the constructor) instead of having the class create them internally, promoting loose coupling (see java-cohesion-and-coupling[26]):

[26] courses/csse2002/java-cohesion-and-coupling.pdf

```java
// Before: AlertService is tightly coupled to EmailSender, and creates its own
↪ dependency.
class AlertService {
    public void sendAlert(String msg) {
        EmailSender sender = new EmailSender();
        sender.send(msg);
    }
}
```

```java
// After: AlertService depends on the MessageSender abstraction, injected via the
↪ constructor.
interface MessageSender {
    void send(String msg);
}
class EmailSender implements MessageSender {
    @Override
    public void send(String msg) { System.out.println("Sending EMAIL: " + msg); }
}
class SmsSender implements MessageSender {
    @Override
    public void send(String msg) { System.out.println("Sending SMS: " + msg); }
}
class AlertService {
    private MessageSender sender;
    public AlertService(MessageSender sender) { this.sender = sender; }
    public void sendAlert(String msg) { sender.send(msg); }
}
```

`AlertService` can now send alerts via email, SMS, or any future `MessageSender` implementation, without ever changing its own code.

A constructor isn't the only injection point — a setter is appropriate when the dependency is likely to change over the object's lifetime, e.g. `public changeSender(MessageSender sender) { this.sender = sender; }`. Eventually *something* (typically an entry-point method like `main`) has to construct the concrete dependencies before injecting them; DI frameworks let you defer that construction to runtime via the framework's own configuration instead of in code, but unless a project needs multiple *levels* of injected dependencies, good design alone can avoid the extra conceptual overhead of adopting one.

Summary

- **S**ingle Responsibility — one reason to change per class.
- **O**pen-Closed — open for extension, closed for modification.

- **L**iskov Substitution — subclasses substitutable for their parents; no stronger preconditions, no weaker postconditions.
- **I**nterface Segregation — many small, client-specific interfaces beat one large general-purpose interface.
- **D**ependency Inversion — depend on abstractions, not concretions; inject dependencies rather than constructing them internally.