# CSSE2002 — Week 11 Notes

## Table of contents



## Lambdas, Streams and Events

See csse2002[1] for course logistics. Guest lecture by James Baker (Principal Software Engineer, School of EECS).

### Today's outline

1. Lambdas in Java (briefly)
2. Streams in Java (briefly)
3. Events, including a toy event system built from scratch

[1] courses/csse2002/csse2002.pdf

All content is in java-lambdas-and-streams[2] and java-events[3].

### Applied class

See week11-tutorial-dependency-inversion[4] — more practice with Dependency Inversion, extending java-solid-principles[5] (Week 7).

### Aside: anecdotal local industry survey

The lecture closed with an informal, small-sample survey (~14 Queensland-based dev teams from the lecturer's own professional network) on AI tool usage and graduate hiring — explicitly framed as anecdotal rather than representative data, so not reproduced here as course content.

## Dependency Inversion

Applied class for 2026-05-14-lambdas-streams-and-events[6] (Week 11). More practice applying Dependency Inversion and Dependency Injection (see java-solid-principles[7]).

### 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());
    }
```

[2] courses/csse2002/java-lambdas-and-streams.pdf
[3] courses/csse2002/java-events.pdf
[4] courses/csse2002/week11-tutorial-dependency-inversion.pdf
[5] courses/csse2002/java-solid-principles.pdf
[6] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[7] courses/csse2002/java-solid-principles.pdf

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

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

**List the classes `Desk` depends upon.**

> **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` (interface, low exposure)
> 2. `ArrayList`
> 3. `Lamp` (assuming it isn't actually used anywhere — the `import` is unused)
> 4. `Book`
> 5. `Monitor`
> 6. `Angle` — likely an `enum`, so reasonably assumed stable
> 7. `LogitechKeyboard`
> 8. `HardwoodFloor`
> 9. `Lamp` (assuming it actually *is* used somewhere, contradicting the "unused import"

assumption above)

## Dependency Inversion

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

### 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); }
}
```

Given a hierarchy `Weapon` (interface) ← `Sword` ← `BronzeSword`/`GoldSword`, and a standalone concrete `BronzeShield` (no interface):

**Minimise concrete dependencies for `Warrior`.**

> **i** Working
>
> ```
> - private BronzeSword weapon = new BronzeSword();
> + private Weapon weapon = new BronzeSword();
> ```
>
> (`BronzeShield` can't be minimised this way yet — there's no interface for it, see 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 it 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();
```

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

> **i** 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.**

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 a warrior will change their weapon
    public void equip(Weapon weapon) { this.weapon = weapon; }

    // overloading gives 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`/`shield` are injected via the constructor (initial equipment) *and* can be swapped later via the overloaded `equip(...)` setters (since it's plausible a warrior changes weapon or shield mid-game) — this is 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[8] (Guest lecture, Week 11).

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

### 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[9]) **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[10]) — 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).

### 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; }
}
```

[9] courses/csse2002/java-lambdas-and-streams.pdf
[10] courses/csse2002/java-cohesion-and-coupling.pdf

```
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.

### Java Lambdas and Streams

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

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

### 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 |
| `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.

## Java SOLID Principles

Introduced in 2026-04-02-refactoring[12] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[13] (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).

### Single Responsibility Principle (SRP)

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

Closely related to java-cohesion-and-coupling[14]'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.

[12] courses/csse2002/2026-04-02-refactoring.pdf
[13] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[14] courses/csse2002/java-cohesion-and-coupling.pdf

```java
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 `Student` class has (at least) three reasons to change: how enrolment works, how GPA is calculated, and how transcripts are generated. Splitting each concern into its own class gives each one a single responsibility:

```java
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) { /* ... */ }
}
```

### 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.

**How do we keep a class closed for modification?** Information hiding (see java-encapsulation[15]).

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

### Example: pluggable grading systems

A university needs to support multiple grading systems (letter grades, numeric grades). A first attempt branches on a `gradeType` string inside `GPACalculator` — every new grading system means *editing* this method (violates OCP):

```
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();
    }
}
```

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

```
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();
    }
}
```

`GPACalculator` now depends on the `GradingSystem` *abstraction* rather than calculating grade points itself — new grading systems (e.g. pass/fail) can be added just by writing a new class that implements `GradingSystem`, with zero changes to `GPACalculator`'s own code.

### 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[16]):

```
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

```
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
     */
```

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

```
    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.

```
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[17]'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:

- **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).

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

- **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[18]):

[18] 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.