CSSE2002 — Week 10 Notes

Table of contents

Java I/O **1**
- Setup . . . 1
- Airline schedule . . . 2
- Maze — designing a file format . . . 7

Coupling and Cohesion **8**
- Coupling . . . 8
- Cohesion — `Customer` . . . 11
- Cohesion — `Employee` . . . 12

Reference material **15**
- Java Cohesion and Coupling . . . 15
- Java I/O . . . 21
- Java Specification . . . 34

Java I/O

Practical for Week 10 (no lecture this week). Extends java-io[1] (Week 8) with saving/loading a data model to/from files, plus a file-format design exercise.

Setup

Provided classes: `Schedule`, `Airport`, `Flight`, `DayOfWeek`, `BadScheduleException` (an airline schedule of flights between airports), plus an unrelated `Maze` class.

[1] courses/csse2002/java-io.pdf

Airline schedule

Goal: make a `Schedule`'s data persistent by saving/loading it to/from a file.

Task 0 — implement `Schedule.toString()` per its Javadoc spec.

> **i** Working

```
@Override
public String toString() {
    StringJoiner joiner = new StringJoiner(System.lineSeparator());
    joiner.add(String.valueOf(getConnectedAirports().size()));
    joiner.add(String.valueOf(this.flights.size()));
    for (Airport airport : getConnectedAirports()) {
        joiner.add(airport.toString());
    }
    for (Flight flight : this.flights) {
        joiner.add(flight.toString());
    }
    return joiner.toString();
}
```

Task 1 — implement `void save(String filename)`, propagating any exceptions rather than handling them:

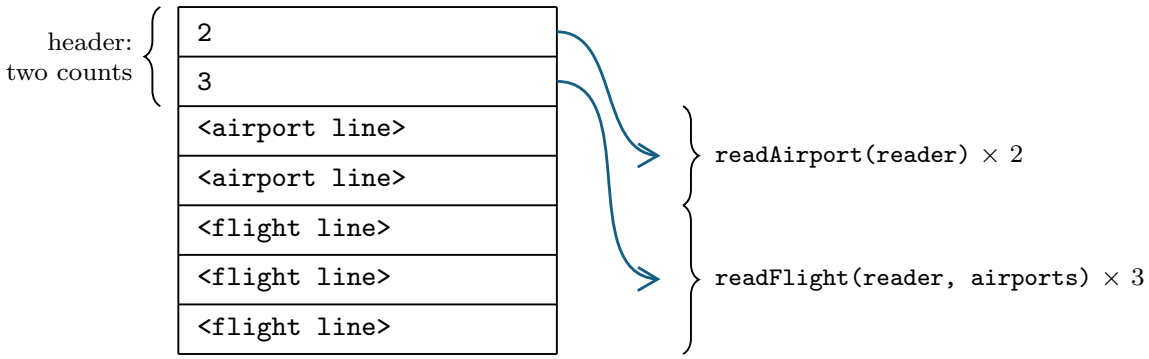
> **i** Working

```
public void save(String filename) throws IOException {
    BufferedWriter writer = new BufferedWriter(new FileWriter(filename));
    writer.write(this.toString());
    writer.close();
}
```

Between them, `toString()` and `load()` fix a file format. `toString()` emits two counts and then the airports and flights in that order; `load()` has to consume exactly the same thing, which is what makes the two counts load-bearing — they are what tells the reader how many times to go round each loop:

Every `BadScheduleException` thrown by `load()` below is one of the ways the file on disk can fail to match Figure 1: a header line missing, a count that isn't a number, a data line missing, a line with the wrong number of `|`-separated fields, or an airport code a flight refers to that never appeared in the airport block.

Task 2 — implement `static Schedule load(String filename)`, using two private helpers (`readFlight`, `readAirport`) and throwing `BadScheduleException` on a failed numeric parse



Within a line, fields are separated by `|`:

`<airport line>`

lineParts[0]	lineParts[1]
new Airport(lineParts[0], lineParts[1])	

`<flight line>`

flight number	origin	destination	day of week
Integer.parseInt	match getCode()	match getCode()	DayOfWeek.valueOf

Figure 1: The Schedule file format fixed by toString() and load(). The two counts shown (2 airports, 3 flights) are an illustrative instance, not a fixture of the format. The teal arrows record the dependency that makes the format parseable at all: each header count governs the trip count of one read loop. The four flight fields are pinned exactly by load() - lineParts[0] goes to Integer.parseInt, [1] and [2] are matched against Airport.getCode(), [3] to DayOfWeek.valueOf - whereas the two airport fields are only ever passed straight to new Airport(lineParts[0], lineParts[1]), so which of them is the code is not settled by load() and they are left unnamed.

or an unknown airport code:

Working

```
public static Schedule load(String filename) throws IOException,
 ↪  BadScheduleException {
    BufferedReader reader = new BufferedReader(new FileReader(filename));

    String airportsLine = reader.readLine();
    String flightsLine = reader.readLine();
    if (airportsLine == null || flightsLine == null) {
        throw new BadScheduleException();
    }

    int numAirports, numFlights;
    try {
        numAirports = Integer.parseInt(airportsLine);
        numFlights = Integer.parseInt(flightsLine);
    } catch (NumberFormatException e) {
        throw new BadScheduleException();
    }

    List<Airport> airports = new ArrayList<>();
    for (int i = 0; i < numAirports; ++i) {
        airports.add(readAirport(reader));
    }
    List<Flight> flights = new ArrayList<>();
    for (int i = 0; i < numFlights; ++i) {
        flights.add(readFlight(reader, airports));
    }

    return new Schedule(flights);
}

private static Flight readFlight(BufferedReader reader, List<Airport> airports)
        throws IOException, BadScheduleException {
    String line = reader.readLine();
    if (line == null) {
        throw new BadScheduleException();
    }

    String[] lineParts = line.split("\\|");
    if (lineParts.length != 4) {
        throw new BadScheduleException();
    }

    int flightNumber;
    try {
        flightNumber = Integer.parseInt(lineParts[0]);
    } catch (NumberFormatException e) {
        throw new BadScheduleException();
    }

    Airport origin = null, destination = null;
    for (Airport airport : airports) {
        if (airport.getCode().equals(lineParts[1])) {
            origin = airport;
        }
        if (airport.getCode().equals(lineParts[2])) {
            destination = airport;
        }
    }
    if (origin == null || destination == null) {
        throw new BadScheduleException();
    }
```

`readAirport` has to declare `BadScheduleException` as well as `IOException`, because its body throws it; with `throws IOException` alone the listing does not compile. `readFlight`/`readAirport` both throw `BadScheduleException` for a malformed line — this is a form of defensive programming (see java-specification[2]) protecting the `Schedule` created by `load()` from corrupt input, rather than letting a `NullPointerException`/`ArrayIndexOutOfBoundsException` leak out from deeper in the parsing logic.

Maze — designing a file format

`Maze` maps `(row, column)` positions to tile types (start, end, wall, empty). Unlike `Schedule`, there's no existing format to follow — the task is to *design* one that's sufficient to save and reload a `Maze`'s full state.

Task 3 — design a file format for `Maze`. (Hint: the class's invariants may let you store less than a full grid dump.)

i Working

Several reasonable designs, each with different tradeoffs:

1. **Visual dump** — store the same characters `render()` would print (e.g. `#` for wall, `S`/`E` for start/end). Human-readable, but more complex to parse back than necessary.
2. **Linear encoding of every tile** — e.g. `#S#E` for a 2×2 grid means wall at (0,0), start at (0,1), wall at (1,0), end at (1,1) (with or without an outer border wall included).
3. **One line per map entry** — e.g. `1,2,#` means "wall at (1, 2)". Simple, but verbose for large mazes with many walls.
4. **Only relevant tiles** — the first two stored positions are always start/end, and every position after that is implicitly a wall; positions can be `(row, column)` pairs or a single linear index (`rows * row + column`).
5. **Start/end positions + a bitmap** — store start and end explicitly, then a bit per remaining tile (`1` = wall, `0` = empty).
6. **Start/end positions + wall runs** — store start and end, then walls as (start, end) position pairs, so a run of adjacent wall tiles can be stored as a single entry instead of one entry per tile.

Options 4-6 are the ones the hint is pointing at: each leans on a regularity in the data rather than dumping the grid verbatim — 4 and 6 store only the tiles that are not empty

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

(6 collapsing adjacent walls into runs), and 5 still visits every remaining tile but spends one bit on it instead of a whole character.

Coupling and Cohesion

Applied class for Week 10 (no lecture this week). More practice applying java-cohesion-and-coupling[3] (Week 6).

Coupling

> **Class coupling**: the strength of the connection or dependence between classes — to what extent does this class depend on other classes? How many methods are called on how many other classes? Can another object influence the flow of control in this object?

Assume the following classes are all in separate files in the same package:

```java
public class X {
    public int num = 5;
    protected Z z;

    public X(Z z) { this.z = z; }

    public void doThis() { sayHello(); }

    public void sayHello() { System.out.println("Hello"); }
}
```

```java
public class Y extends X {
    public Y(Z z) { super(z); }

    public void doThat() { this.z.sayHello(); }

    @Override
    public void sayHello() { super.sayHello(); }
}
```

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

```java
public class Z {
    private X x = new X();

    public void setNum(int num) { x.num = num; }

    public void sayHello() { System.out.println("Hello"); }
}
```

These classes are tightly coupled, to a degree that is possibly unrealistic — every pair of them is connected. Identify the points of coupling between: (0) `X` and `Y`; (1) `X` and `Z`; (2) `Y` and `Z` — you don't need to name the type/level of coupling (see java-cohesion-and-coupling[4] for the ladder from content coupling down to none), just where it occurs.

As written, `Z`'s `private X x = new X()` would not compile, because `X` declares only `X(Z z)` and so has no no-argument constructor. The exercise is about locating coupling, not about compiling the listing.

Working

Class	Coupling	Implication
`X`	stores a `Z` object	`X` is coupled to `Z`
`X`	constructor takes a `Z`	`X` is coupled to `Z`
`X`	`doThis()` just calls `sayHello()`, which is overridden in `Y`	forwarding behaviour
`Y`	constructor takes a `Z`	`Y` is coupled to `Z`
`Y`	constructor calls `super(z)`	`Y` is coupled to `X`
`Y`	`doThat()` accesses the protected `this.z`	`Y` is coupled to `Z` and `X`
`Y`	`sayHello()` calls `X.sayHello()` via `super`	`Y` is coupled to `X`
`Z`	stores an `X` object	`Z` is coupled to `X`
`Z`	`setNum()` accesses the public `X.num`	`Z` is coupled to `X`

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

Drawn as a dependency graph, the table says that **every** pair of these three classes is joined, and that `X` and `Z` are joined in *both* directions — a dependency cycle, so neither can be understood, tested or reused without the other:

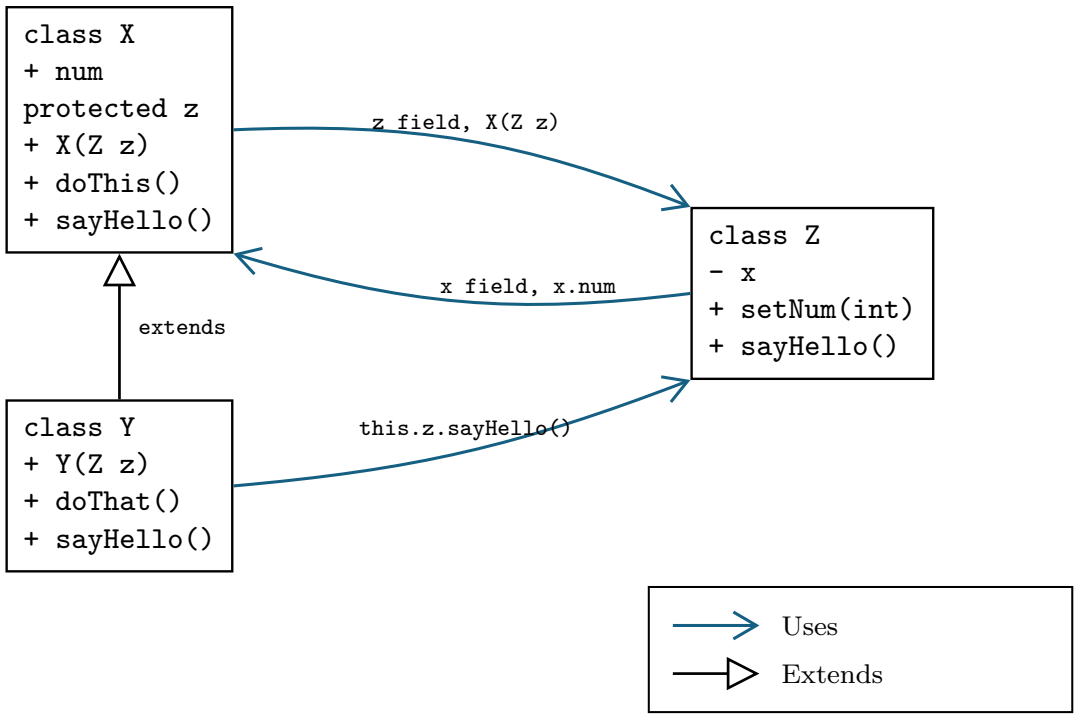


Figure 2: The coupling points of Task 0-2 collected into a dependency graph. The table's nine rows become four edges: X and Z point at each other, Y points at Z, and one extends edge carries every X-Y row at once (super(z), super.sayHello(), and doThis() forwarding to a sayHello() that Y overrides). A solid teal arrow means 'uses' and points from the depender to the thing it depends on. The black line with an unfilled triangle head is the generalisation arrow, with the superclass above. Members carry the visibility prefixes `-` (private) and `+` (public), and X's protected field is spelled out in words.

Two points the table makes that Figure 2 makes visible at once:

- The `X`–`Z` pair is a **cycle**. `X` holds a `Z` and takes one in its constructor; `Z` holds an `X` and writes straight into its public `num` field.
- `Y`'s single line `this.z.sayHello()` reaches a `Z` through a field it did not declare, so it couples `Y` to `Z` *and* to `X` at the same time — which is why the table gives that one row two implications.

Cohesion — `Customer`

> **Class cohesion**: how well components support a central purpose — how focused the components of a unit are. How well do the parts of the class (state and methods) fit together? Do they all contribute to a single, clear purpose?

```java
public class Customer {
    private String name;
    private String streetAddress;
    private String suburb;
    private String postCode;
    private List<Item> orders; // the products that have been ordered

    public Customer(String name, String streetAddress, String suburb, String
     ↪  postCode) {
        this.name = name;
        this.streetAddress = streetAddress;
        this.suburb = suburb;
        this.postCode = postCode;
        this.orders = new ArrayList<>();
    }

    public String getName() { return name; }
    public String getMailingAddress() { return streetAddress + suburb + postCode; }
    public void newOrder(List<Item> order) { orders.addAll(order); }
    public List<Item> getAllOrder() { return orders; }
}
```

Does `Customer` exhibit high or low cohesion? Justify your answer.

> **Working**
>
> Low cohesion:
>
> - `name` is only used once, in a single getter.
> - `streetAddress`/`suburb`/`postCode` are only used once, in a single getter — and these aren't unique to a `Customer`, so belong in a separate class.
> - `orders` (with its getter and add method) isn't cohesive with the rest of `Customer`'s representation, and could live in a dedicated `Order`-related class.
>
> The grouping of member variables looks coincidental, and the methods have no clear relationship to each other's functionality — the state and methods don't contribute to a single, clear purpose.

If `Customer` doesn't have high cohesion, design replacement classes with higher cohesion.

> **Working**
>
> `Customer` is really trying to be at least three concepts at once: a customer, a mailing address, and a customer's order history. Extract a postal-address abstraction (that `Customer` stores an instance of), and an order-history abstraction (stored by `Customer`, or managed separately).

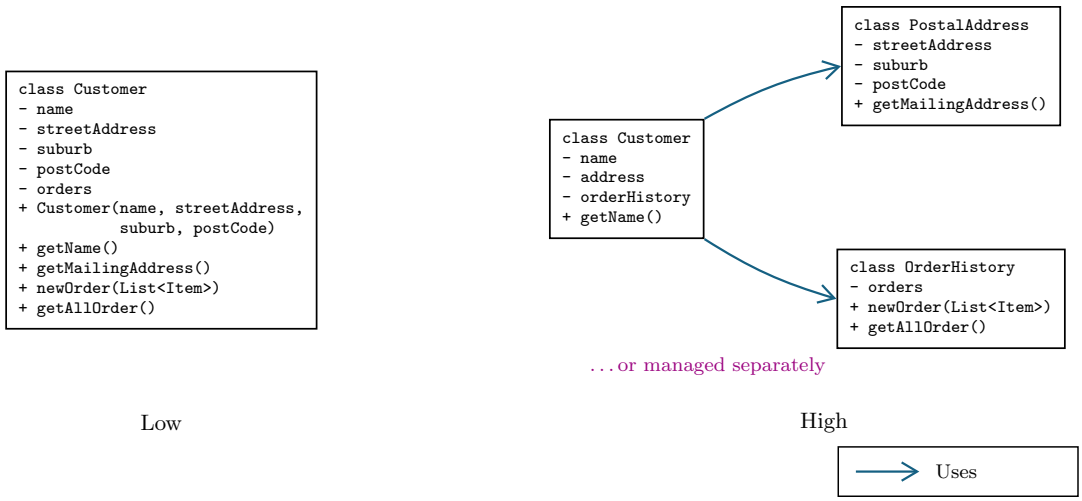


Figure 3: Customer as written, and split into its three concepts: a customer, a mailing address (PostalAddress) and a customer's order history (OrderHistory). The solid teal arrow means 'uses' and should not be read as UML aggregation or composition. The magenta note says that the order history may equally be managed outside Customer. The weaker design is on the left and the stronger on the right, each labelled with one word beneath.

The point of Figure 3 is that nothing is deleted and no behaviour changes — the same five fields and four methods are still there. All that moves is *which class owns them*, and with it the number of reasons any one class has to change.

Cohesion — `Employee`

```
public class Employee {
    private String firstName;
    private String surname;
```

```
    private String homeAddress;
    private String suburb;
    private String postCode;
    private String currentRole;
    private int currentRoleSecurityLevel;
    private int hourlyWage;

    public Employee(String fName, String lName, String homeAddress,
                    String suburb, String postCode, int hourlyWage) {
        this.firstName = fName;
        this.surname = lName;
        this.homeAddress = homeAddress;
        this.suburb = suburb;
        this.postCode = postCode;
        this.hourlyWage = hourlyWage;
    }

    public String getName() { return surname + ", " + firstName; }
    public String getMailingAddress() {
        return String.format("%s%n%s%n%s", homeAddress, suburb, postCode);
    }
    public void setRole(String newRole, int securityLevel) {
        currentRole = newRole;
        currentRoleSecurityLevel = securityLevel;
    }
    public String getCurrentRole() { return currentRole; }
    public boolean accessAllowed(int requiredSecurityLevel) {
        return currentRoleSecurityLevel >= requiredSecurityLevel;
    }
    public int getPay(int hoursWorked) { return hourlyWage * hoursWorked; }
    public void setHourlyWage(int newWage) { hourlyWage = newWage; }
}
```

Does `Employee` exhibit high or low cohesion? Justify your answer.

> **Working**
>
> Low cohesion:
>
> - `firstName`/`surname` are only set in the constructor and returned by a single getter — better as a dedicated personal-details class, or simply a single `name` string.
> - The address fields are assigned once and used in a single getter — they appear arbitrarily grouped in this class and could live in a separate object.
> - The role fields (`currentRole`, `currentRoleSecurityLevel`) are used across several methods, but never interact with the name or address fields — this functionality could move to a `Role` class, shrinking `Employee`'s constructor and letting other

objects reuse `Role`.

`Employee` is really trying to wrap the functionality of at least two other objects (address, role) inside itself; since the grouping of state appears coincidental rather than purposeful, `Employee` has low cohesion.

"The grouping appears coincidental" is a claim you can check mechanically: tabulate which method reads or writes which field, and see whether the marks form one connected mass or several disjoint blocks.

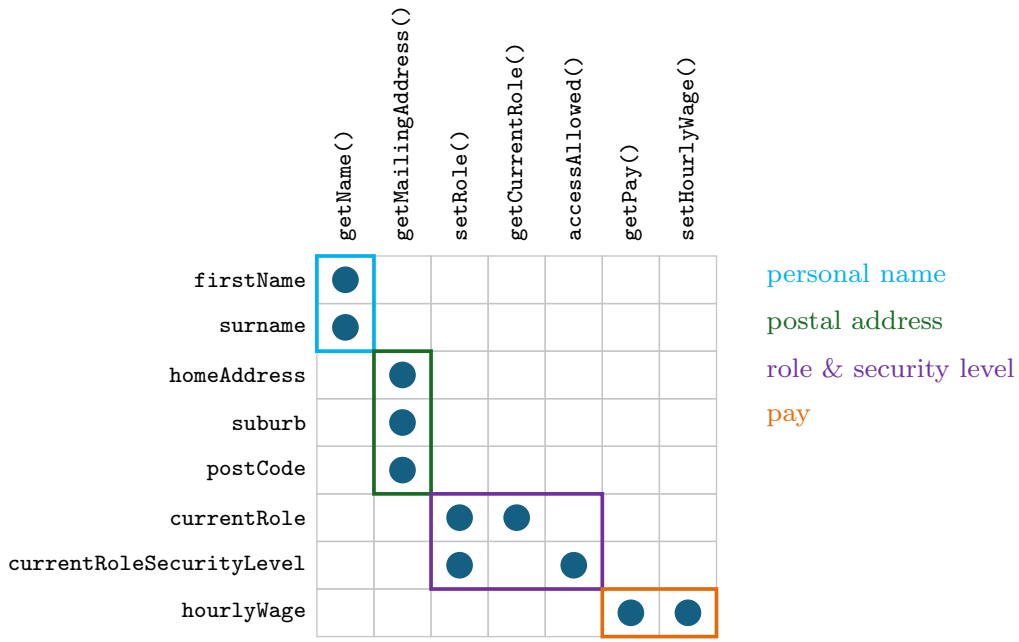


Figure 4: Which Employee method touches which Employee field. The constructor is deliberately left out: it assigns every field by definition, so it says nothing about cohesion. The result is four disjoint blocks with no mark outside them: no method in this class touches fields from two different blocks, which is exactly what 'the grouping of state appears coincidental' means, and exactly why each block can be lifted out into a class of its own. Block outline colours carry group identity only.

Compare the same test applied to a cohesive class: there the marks would spread across the whole grid, because the methods would share the state they operate on. Here they do not overlap at all — Figure 4 is four small classes drawn inside one large one.

Reference material

Java Cohesion and Coupling

Introduced in 2026-04-02-refactoring[5] (Lecture, Week 6, Part 2). Practised in week10-tutorial-coupling-and-cohesion[6].

Modularization

The process of dividing a large system (a monolith) into smaller, more manageable pieces that can be worked on and tested independently before being reassembled into the final product. This raises the question everything below answers: **what if the resulting modules are highly dependent on each other?** Cutting a monolith into pieces buys nothing if every piece still has to know about every other one. The left panel of the coupling figure below shows exactly that failure: six modules tangled together with arrows in all directions.

Two measures separate a real decomposition from a nominal one: **coupling** (how much the modules depend on *each other*) and **cohesion** (how well the contents of *one* module belong together).

Coupling

The degree of interdependence between different modules, classes, or components of a system.

- **High coupling** — strong interconnections, where a change in one module can cascade through others.
- **Low coupling** — greater independence and isolation between modules.

Levels of coupling (tightly → loosely coupled)

1. **Content** — one class directly manipulates another's data (e.g. public fields instead of private).
2. **Common** — sharing access to the same global data/variables.
3. **External** — sharing something imposed by an external source, e.g. a data format.
4. **Control** — one class controls what happens in another by passing it information/instructions.
5. **Stamp** — sharing a data structure, but each class only needs access to a select part of it.
6. **Data** — sharing data through means such as parameter passing in methods.
7. **None** — no dependency at all (most loosely coupled).

[5] courses/csse2002/2026-04-02-refactoring.pdf
[6] courses/csse2002/week10-tutorial-coupling-and-cohesion.pdf

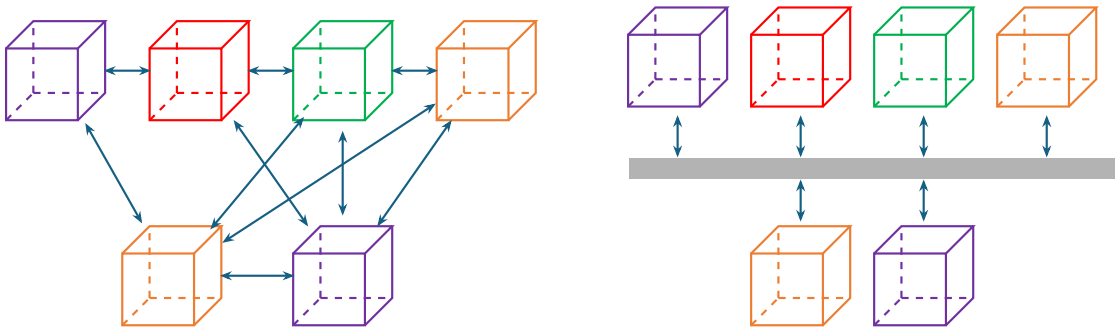


Figure 5: High versus low coupling. Left: every module talks directly to several others, so a change in any one can reach any other. Right: the same six modules each talk only to a shared interface — the grey bar — so nothing propagates further than that bar. A module is a wireframe cube, and its colour only tells the modules apart; a dependency is a teal double-headed arrow. The tangle of arrows on the left is schematic — the count is the point, not which cube joins which.

The list is ordered, and the order is the assessable part: the same two classes can be joined in any of these ways, and the further down the list you can get, the less a change in one can force a change in the other.

Worked classification examples

Four snippets, each with the same question: what type of coupling do these methods have? Each answer is reasoned from the definitions above.

```
public void sinh(int x) {...}
public void cosh(int x) {...}
public void tanh(int x) {
    return sinh(x) / cosh(x);
}
```

`tanh` only interacts with `sinh`/`cosh` by passing/receiving simple parameters — **data coupling**.

```
public class R { int x, y, z; char a, b, c; }
public static void compute(R r) {
    return r.x * r.y; // only touches x and y
}
```

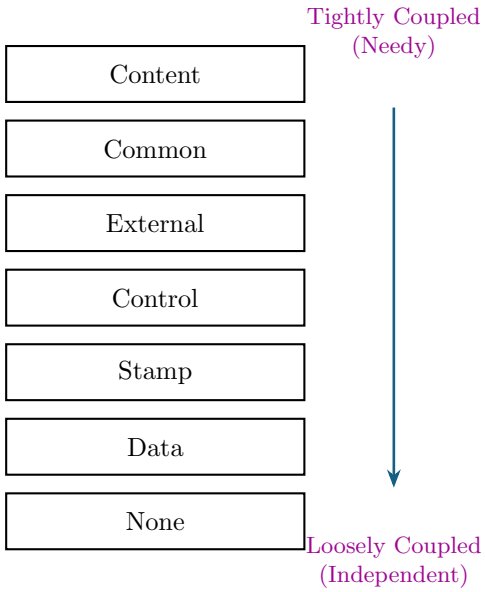


Figure 6: The coupling-levels ladder. Most tightly coupled at the top, least at the bottom, with the arrow running down from tightly coupled (needy) to loosely coupled (independent).

`compute` is handed the whole `R` object but only needs two of its six fields — **stamp coupling**.

```
public static void compute(R r) {
    return r.x * r.y * r.z / r.a + r.b + r.c; // uses every field
}
```

Here every field of `R` is actually used, so passing the whole object is fully justified — this is just **data coupling** via a composite value, not stamp coupling (contrast with the previous example, which only used part of the structure).

```
public static runReport(String name, int age, String phone, Date birth, String
 ↪  address) {...}
```

Five separate simple parameters — still **data coupling**, though a long parameter list like this is itself often a separate code smell (see java-refactoring[7]) worth considering bundling into an object.

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

Cohesion

The degree of interrelatedness and focus among the elements *within* a module, class, or component.

- **High cohesion** — elements are closely related and contribute collectively to one specific functionality.
- **Low cohesion** — elements are less focused, serving multiple unrelated purposes.

Asked of a class, cohesion comes down to two questions: **how well do the parts of the class (state and methods) fit together?** and **do they all contribute to a single, clear purpose?**

> High cohesion is preferable. (Low cohesion usually indicates that a class should be split.)

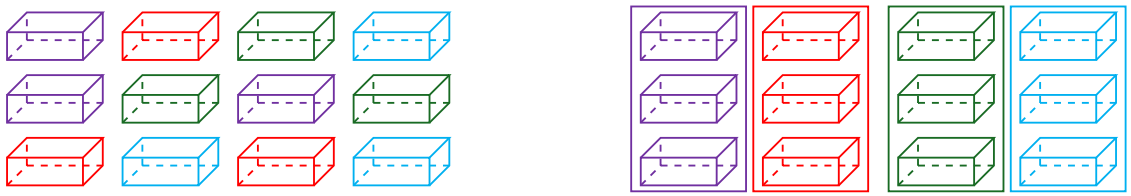


Figure 7: Low versus high cohesion. The same twelve elements in both panels: on the left the four kinds are scattered across the grid, on the right each kind is gathered into one module. Colour means kind only, and each high-cohesion group is enclosed by a rectangle in that group's own colour. This figure says nothing about coupling — it is only about what sits inside one module. The good case is again on the right.

Levels of cohesion (low → high)

1. **Coincidental** — elements grouped arbitrarily, with no clear relationship.
2. **Temporal** — elements grouped because they execute at the same time/phase.
3. **Procedural** — elements grouped by their involvement in a specific sequence of steps.
4. **Communicational** — elements work together to manipulate a shared data structure.
5. **Sequential** — elements organised in a linear sequence, where one's output is the next's input.
6. **Functional** — elements grouped around a single, specific functionality or task (most cohesive).

Worked example

```java
public void performTasks(int[] numbers, String text) {
    // Task 1: sum the numbers
    int sum = 0;
    for (int num : numbers) { sum += num; }
    System.out.println("Sum of numbers: " + sum);

    // Task 2: reverse the text
    StringBuilder reversedText = new StringBuilder();
    for (int i = text.length() - 1; i >= 0; i--) {
     ↪  reversedText.append(text.charAt(i)); }
    System.out.println("Reversed text: " + reversedText);
}
```

Summing numbers and reversing text share no real purpose — they're only in the same method because they happen to run one after another. This is low (coincidental, or at best temporal) cohesion, and a sign `performTasks` should be split into two focused methods.

Is this class cohesive?

A `Car` class contains: Fuel, Steering, Speed, Route planner, Public holiday calculator.

Fuel/Steering/Speed are all core to a car's own physical behaviour, but a route planner and (especially) a public-holiday calculator are unrelated concerns bolted onto the class — this is **low cohesion**, and a sign the class is trying to do too much (a "God class"); the unrelated pieces should be split into their own classes.

Considerations

- Classes should be **highly cohesive** — a single, easily understood concept.
- Classes should have **loose coupling** to each other — this reduces overall system complexity.

Good modularization (high cohesion, low coupling) groups tightly-interconnected elements together into the same module and minimises connections *between* modules, unlike bad modularization, where connections are scattered across module boundaries with no clear grouping. The figure below makes this precise, because both panels are the *same system* — the same ten elements and the same fifteen dependencies — grouped into modules two different ways:

Read off the figure, the two measures are the same count seen from opposite sides: a dependency that stays inside a module is cohesion, and a dependency that leaves one is coupling. Maximising the first and minimising the second is one decision — *where to draw the boundaries* — which is why the goal is always stated as a pair.

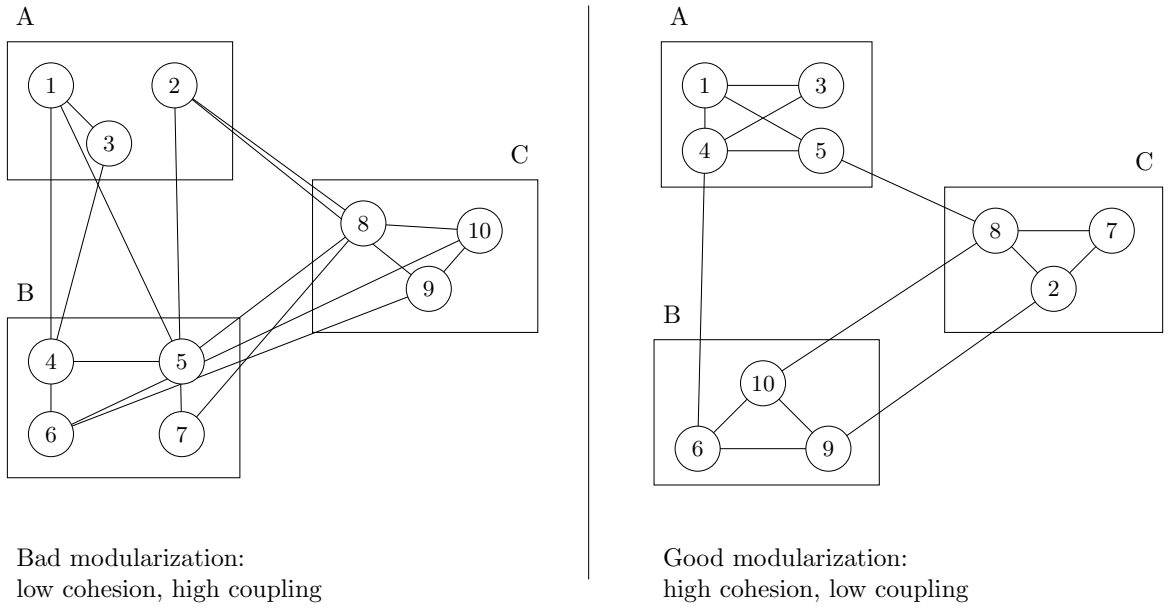


Figure 8: Bad versus good modularization. Both panels draw the same ten elements and the same fifteen dependencies; only the assignment of elements to modules A, B and C differs. Left: 5 dependencies fall inside a module and 10 cross a boundary. Right: 11 fall inside and only 4 cross. Nothing was rewritten to achieve that — the code is identical, the grouping is not.

Java I/O

Introduced in 2026-04-23-java-io[8] (Lecture, Week 8).

Java spreads its I/O functionality across many more classes than most other languages, split roughly into three generations of API:

1. **Streams** (`java.io`, since 1.0) — byte-oriented.
2. **Readers & Writers** (`java.io`, since 1.1) — character-oriented, added because byte streams weren't ideal for text/Unicode.
3. **New I/O** (`java.nio.file`, since 1.7) — file-system operations (paths, directories, attributes), which `java.io` was never designed for.

`Scanner` (`java.util`, since 1.5) sits alongside these as a convenience utility — neither a stream nor a reader.

Streams

A **stream** is an abstraction that either produces or consumes information, letting Java perform I/O uniformly regardless of whether data comes from a file, keyboard, console, network socket, or elsewhere.

- An **output stream** is an abstract destination to be written to.
- An **input stream** is an abstract source of input that can be read without concern for how it's supplied.

What the abstraction hides is threefold: the **source** (keyboard, disk file, network, …), the **destination** (terminal, disk file, network, …), and the **chunking** — whether data arrives one byte at a time (keyboard) or many at once (files).

How a program is written should not depend on whether its input is from a file or a keyboard, or whether its destination is a file, the console or the network. If the source or destination changes, the program should not need rewriting.

Java has two families of streams:

- **Byte streams** (`java.io.InputStream` / `java.io.OutputStream`) — raw binary data (files, images, network data).
- **Character streams** (`Reader`/`Writer`, see below) — characters/text, handling Unicode; better suited to strings, text files and console text.

  A byte stream reads data as **bytes**, whereas a character stream reads data as **characters**.

[8] courses/csse2002/2026-04-23-java-io.pdf

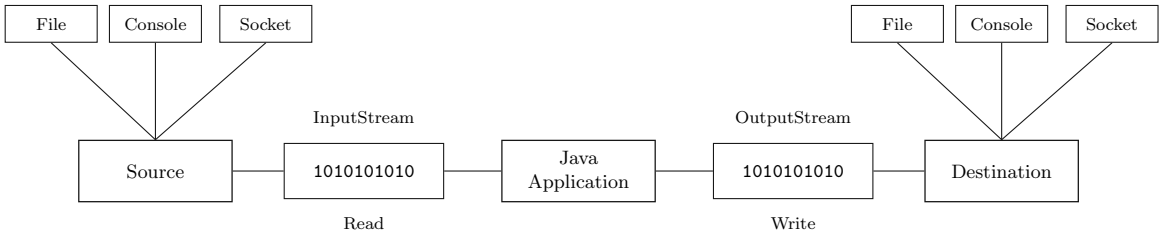


Figure 9: The stream abstraction: whatever the real source (file, console, socket), the application reads bytes through one `InputStream`, and writes them back out through one `OutputStream`, whatever the real destination.

Reference: *Java: The Complete Reference*, Eleventh Edition — Herbert Schildt.

The byte-stream hierarchy

`InputStream` and its subclasses represent a stream of bytes, drawn from different sources — `FileInputStream` gets bytes from a file, `ByteArrayInputStream` from an array, and so on. Methods that use streams should accept the **superclass** type as a parameter, so any concrete stream can be substituted (see java-solid-principles[9] — Liskov Substitution).

Worked example — writing to an abstract `OutputStream`

`System.out` is an output stream already in everyday use. That makes it the first thing to hand a method that takes an abstract `OutputStream`: `sendToStream(System.out)`. The version below keeps `sendToStream` untouched and swaps the argument twice:

```
public class Foo {
    public static void main(String[] args) throws IOException {
        OutputStream fos = new FileOutputStream("output.txt");
        sendToStream(fos);

        fos = new ByteArrayOutputStream();
        sendToStream(fos);

        System.out.println(fos);
    }

    private static void sendToStream(OutputStream stream) throws IOException {
        String output = "foo bar baz";
        for (char letter : output.toCharArray()) {
            stream.write(letter);
```

[9] courses/csse2002/java-solid-principles.pdf

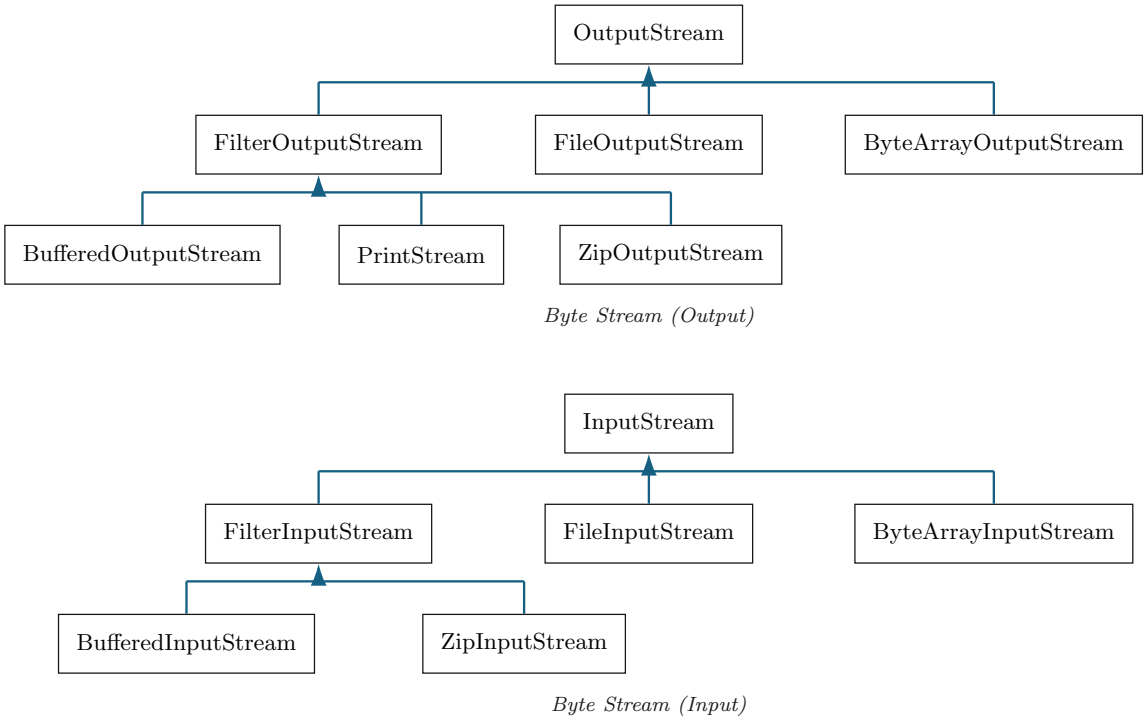


Figure 10: The byte-stream class hierarchies: `OutputStream` and `InputStream`. Each superclass sits above its subclasses, and the arrowhead points into the parent. The wrappers `BufferedInputStream` and `BufferedOutputStream` are `Filter*` streams — streams that wrap another stream.

```
        }
        stream.flush();
    }
}
```

`sendToStream` never changes: first `System.out` (straight to the console), then a `FileOutputStream`, which leaves `output.txt` holding `foo bar baz`, then a `ByteArrayOutputStream`. `main` has to declare `throws IOException`; without the declaration it does not compile. The final `System.out.println(fos)` prints the byte-array stream's accumulated contents, because `ByteArrayOutputStream.toString()` decodes the buffer it has collected.

End of file

All input streams (and everything built on them) need to consider "end of file":

```
public int read() // returns -1 on end of file
```

Reading a whole file one byte at a time therefore loops until `read()` hands back `-1`:

```
public class Foo {
    public static void main(String[] args) throws IOException {
        readAll(new FileInputStream("output.txt"));
    }

    private static void readAll(InputStream input) throws IOException {
        int in = 0;
        while (in != -1) {
            in = input.read();
            System.out.println(in);
        }
    }
}
```

Run against an `output.txt` containing `foo bar baz`, that prints the *byte values* — `102`, `111`, `111`, `32`, … The fix is to cast in the `println` inside the loop, so you see the characters instead:

```
System.out.println((char) in);
```

The loop as written tests `in` before refreshing it, so the final iteration still prints the sentinel — `-1`, or, once cast, the char with all sixteen bits set, , which is not a printable character. Reading into the loop condition, `while ((in = input.read()) != -1)`, avoids that.

Buffering

Getting information from files can be slow a `byte` or `char` at a time. `BufferedInputStream` is an input stream which **wraps around another input stream**. The buffer is an area in the main memory used to store or hold the data temporarily. This is faster because one large read from the device then serves many small `read()` calls.

Only the construction changes; `readAll` is untouched:

```
readAll(new BufferedInputStream(new FileInputStream("output.txt")));
```

(What each layer actually contributes is drawn in Figure 11.)

On a test VM, reading a 4MB file takes **82 milliseconds with buffering** and **18,193 milliseconds without**.

Closing streams

Streams need closure. Streams and Readers have a `close()` method; systems may limit how many files you can have open at a time; when you have finished with a stream, close it.

Getting this right takes three steps. First, just catching the failure to open:

```
public static void main(String[] args) {
    try {
        readAll(new BufferedInputStream(new FileInputStream("output.txt")));
    } catch (IOException e) {
        System.out.println("Error: File Not Found " + e);
    }
}
```

That still never closes anything. Adding a `finally` block does — but the close can itself throw, so it needs its own `try`, and the whole thing gets verbose fast:

```
BufferedInputStream input = null;
try {
    input = new BufferedInputStream(new FileInputStream("output.txt"));
    readAll(input);
} catch (IOException e) {
    System.out.println("Error: File Not Found " + e);
} finally {
    try {
        input.close();
    } catch (IOException e) {
        System.out.println("Error closing the stream: " + e);
```

```
    }
}
```

try-with-resources is much cleaner — any resource declared in the `try(...)` parentheses is closed automatically:

```
try (BufferedInputStream input = new BufferedInputStream(new
 ↪  FileInputStream("output.txt"))) {
    readAll(input);
} catch (IOException e) {
    System.out.println("Error: File Not Found " + e);
}
```

> *Effective Java* (2017, Addison-Wesley) — Item 9: "Prefer try-with-resources to try-finally".

A cautionary example of what production I/O failures cost: a national passport office suspended issuance after a computer-system breakdown (reported November 2022).

Wrapping streams together

Nothing in `java.io` is one class doing everything: you assemble the behaviour you want by nesting one stream inside another. That is what the hierarchy's whole `Filter*` branch is for, and why the constructor calls look the way they do. The second chain below uses `InputStreamReader` and `BufferedReader`, which the next section introduces — it is drawn here so both chains can be compared side by side.

Readers & Writers

Originally Java I/O was byte-oriented. Byte streams were fine for raw binary data but not ideal for text, so character streams were added in 1.1 to handle text and Unicode characters more conveniently and efficiently.

- `java.io.Writer` — an abstraction over writing `char`s to an output.
- `java.io.Reader` — an abstraction over reading `char`s from an input source.

`InputStreamReader` is the **bridge** between the two systems: it wraps an `InputStream` (like `System.in`) and decodes its bytes into characters.

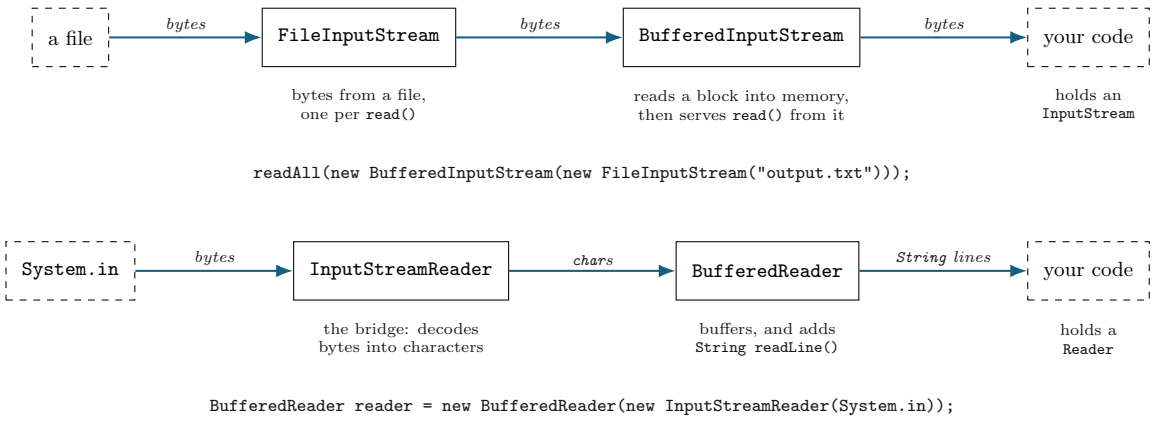


Figure 11: How the nesting in `new BufferedInputStream(new FileInputStream(...))` and `new BufferedReader(new InputStreamReader(System.in))` actually behaves — each wrapper adds one capability and passes the rest through, and your code only ever holds the outermost object. `BufferedInputStream` wraps another input stream; `InputStreamReader` acts as a bridge decoding bytes into characters; `BufferedReader` wraps another `Reader` and adds `readLine()`. The gloss beneath each box says what that stage contributes.

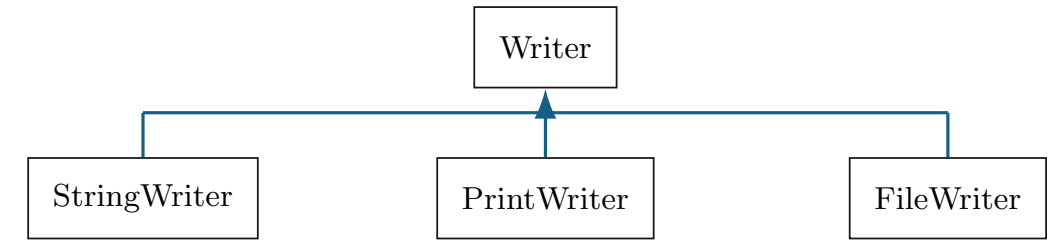
Worked example — read the first 10 characters of input

```
> Words, words, and more words
Words, wor
```

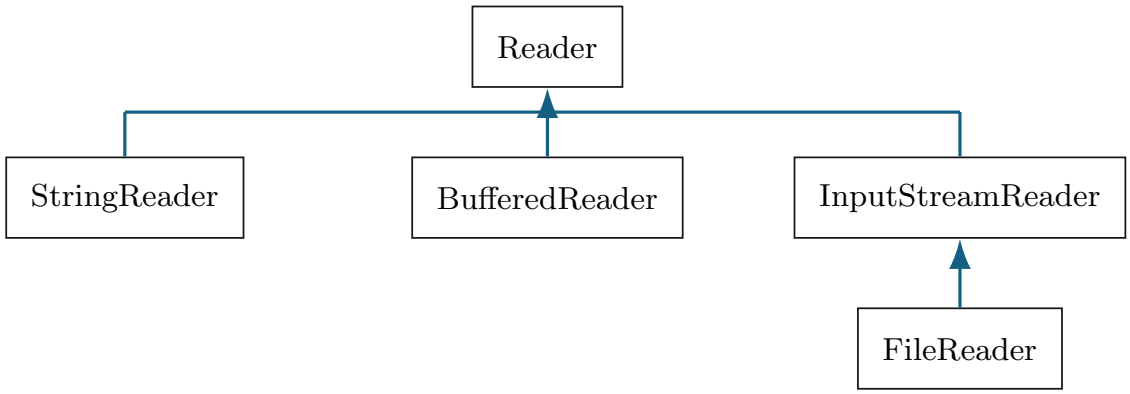
```java
public class ReaderEx {
    public static void main(String[] args) throws IOException {
        readAndPrint(new InputStreamReader(System.in));
    }

    private static void readAndPrint(Reader reader) throws IOException {
        char[] letters = new char[10];
        for (int i = 0; i < 10; i++) {
            letters[i] = (char) reader.read();
        }
        System.out.println(letters);
    }
}
```

Exercise: rewrite the program to use `FileReader`. Because `readAndPrint` takes the abstract `Reader`, only the argument in `main` has to change.



java.io.Writer — writing `chars` to an output



java.io.Reader — reading `chars` from an input source

Figure 12: The character-stream hierarchies: `Writer` and `Reader`, drawn the same way as Figure 10. The one two-level branch is the important one — `FileReader` is a kind of `InputStreamReader`, i.e. reading a text file is itself a case of decoding bytes into characters.

BufferedReader

`BufferedReader` wraps another `Reader`. As well as buffering it adds `String readLine()`:

```
public class BufferedReaderEx {
    public static void main(String[] args) {
        BufferedReader reader = new BufferedReader(new
         ↪  FileReader("readwithbuffer.txt"));
        for (int i = 0; i < 5; i++) {
            System.out.println(reader.readLine());
        }
    }
}
```

Wrapping `System.in` instead of a file is the standard console-input idiom:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
```

The exception handling has to be made explicit: `FileReader`'s constructor throws `FileNotFoundException`, and `readLine()` throws `IOException`, so each needs handling. Below, the second is wrapped in an unchecked `RuntimeException` rather than being declared. See java-exceptions[10].

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("readwithbuffer.txt"));
} catch (FileNotFoundException e) {
    System.err.println("File not found: " + e.getMessage());
}

for (int i = 0; i < 10; i++) {
    try {
        System.out.println(reader.readLine());
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Catching and continuing leaves `reader` `null` when the file is missing, so the loop then throws a `NullPointerException`. `try`-with-resources — as used for streams above — would both scope the reader correctly and close it.

[10] courses/csse2002/java-exceptions.pdf

`PrintWriter`

`System.out` is a `PrintStream`; `PrintWriter` is another (better) option for character output — it can write to many destinations, and it can write **formatted** text. You can construct one straight from `System.out`:

```
PrintWriter printWriter = new PrintWriter(System.out);
printWriter.println("I love CSSE2002");
printWriter.printf("Formatted number: %.2f%n", 123.45336);
printWriter.flush();
```

Or stack it on a `FileWriter` inside a `try`-with-resources, closing both:

```
String fileName = "writeroutput.txt";
try (FileWriter fileWriter = new FileWriter(fileName);
     PrintWriter printWriter = new PrintWriter(fileWriter)) {
    printWriter.println("I love CSSE2002");
    printWriter.printf("Formatted number: %.2f%n", 123.45336);
    System.out.println("Successfully Written.");
} catch (IOException e) {
    e.printStackTrace();
}
```

`writeroutput.txt` then holds:

```
I love CSSE2002
Formatted number: 123.45
```

`flush()`

If an `OutputStream`/`Writer` is buffered, output might not be sent immediately — `flush()` will send any pending output. This matters for:

- **"Interactive" or other situations where you expect a response** — the other end won't respond if you haven't actually sent anything yet.
- **Debugging or logging**, where you need an up-to-date view of what is happening.

`close()`-ing a stream will flush it as well.

`Scanner`

`Scanner` (`java.util`, since 1.5) is **neither a stream nor a reader** — it's a **utility class** that wraps an existing `InputStream` or `Reader`, added to simplify reading user input and parsing primitive types and strings, as a more user-friendly alternative to `BufferedReader`. It allows reading of primitive Java types from the stream, and `regex` scanning is possible for advanced use cases.

Because it accepts either, the same logic can be handed a `Reader`:

```
public class ScannerEx {
    public static void main(String[] args) {
        readInts(new InputStreamReader(System.in));
    }

    private static void readInts(Reader reader) {
        Scanner scanner = new Scanner(reader);
        int total = 0;
        while (scanner.hasNextInt()) {
            int newInt = scanner.nextInt();
            total += newInt;
        }
        System.out.println(total);
    }
}
```

or given `System.in` directly — in which case the `Scanner` internally creates its own `InputStreamReader`:

```
Scanner scanner = new Scanner(System.in);
int total = 0;
while (scanner.hasNextInt()) {
    total += scanner.nextInt();
}
System.out.println(total);
```

Reading strings:

- `scanner.next()` — reads the next **word**, up to whitespace, skipping any leading whitespace.
- `scanner.nextLine()` — reads the **entire line**, until Enter is pressed.

```java
Scanner scanner = new Scanner(System.in);
System.out.println("Enter three words:");
String word1 = scanner.next();   // reads up to first space
String word2 = scanner.next();
String word3 = scanner.next();
System.out.println("You entered: " + word1 + ", " + word2 + ", " + word3);
```

Further reading: Scanner vs. BufferedReader.

New I/O (`java.nio.file`)

The problem with the older `java.io`: it focused mainly on streams (byte and char streams), because the main goal was **reading and writing data, not managing files or directories**. That leaves it with limited support for file-system operations, and makes it inconvenient for file attributes, directory traversal, and path manipulation.

`java.nio.file` (since Java 1.7) introduces two new abstractions:

- `java.nio.file.Path` — represents a file location.
- `java.nio.file.Files` — utility methods for file operations.

Creating a `Path`

`Path` is the key abstraction for locating files, and can be created from path strings or from a URI:

```java
Path p1 = Path.of("docs/output.txt");        // the whole path as one string
Path p2 = Path.of("docs", "output.txt");     // separate name elements, which Java
 ↪  joins into a path
Path p3 = Path.of(new URI("file:///docs/output.txt"));
```

`Files` operations

Category	Methods
Create/delete	`Files.createFile(Path)`, `Files.createDirectory(Path)`, `Files.delete(Path)`, `Files.deleteIfExists(Path)`
Query	`Files.exists(Path)`, `Files.isDirectory(Path)`, `Files.isRegularFile(Path)`

Category	Methods
Manipulate	`Files.copy(Path, Path)`, `Files.move(Path, Path)`
Read/write	`Files.readAllLines(Path)`, `Files.readString(Path)`, `Files.write(Path, Iterable<? extends CharSequence>)`, `Files.writeString(Path, CharSequence)`

`CharSequence` is an interface representing a read-only sequence of characters; `String`, `StringBuilder`, `StringBuffer` and `CharBuffer` all implement it.

```
public static void main(String[] args) throws IOException {
    Path myPath = Path.of("src", "Week08", "NIO", "myfile.txt");
    Files.createFile(myPath);
}
```

Reading a whole file is then a one-liner — no stream, no wrapper, no `close()`:

```
public static void main(String[] args) throws IOException {
    Path myPath = Path.of("src", "Week08", "NIO", "myfile.txt");

    List<String> lines = Files.readAllLines(myPath);
    for (String line : lines) {
        System.out.println(line);
    }
}
```

Exercise: rewrite the earlier stream and reader exercises using New I/O.

Summary: which to use?

- **Streams** — good for binary data (byte at a time).
- **Readers & Writers** — best for text data.
- **New I/O** — when manipulating a file system or file contents.

Saving and loading a whole object model to a file is picked up later, in week10-lab-java-io[11].

[11] courses/csse2002/week10-lab-java-io.pdf

Java Specification

Introduced in 2026-03-12-object-oriented-programming-ii[12] (Lecture, Week 3). Extended in 2026-03-19-exceptions[13] and formalised later in java-hoare-logic[14]; pre/postconditions come back as class invariants in week8-tutorial-class-invariants[15] and as the substitution principle in week9-tutorial-substitution-principle-and-pre-post-conditions[16].

The order below: Javadoc → restrictiveness → generality → formality → contracts → defensive programming.

Javadoc

- Ordinary comments: `//` and `/* ... */`.
- **Javadoc comments**:
  - begin with `/**` — *note the second* `*`; the Javadoc generator ignores all other comments;
  - end with `*/`;
  - must sit **immediately above** the thing being documented;
  - use tags beginning with `@`;
  - some tags take parameters, some just text.

A worked example:

```
/**
 * Calculates a sum by combining the hash code of the provided string
 * and the long value of the provided float.
 *
 * @param inputString the input string whose hash code will be used
 * @param inputFloat  the float value to be converted to long and combined with the
  ↪  string hash code
 * @return a long value representing the sum of the string's hash code and the long
  ↪  value of the float
 */
public long doCalculation(String inputString, float inputFloat){
    long floatToLong = (long) inputFloat;

    // Combine the hash code of the string with the float-to-long value
    long result = inputString.hashCode() + floatToLong;
```

[12] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[13] courses/csse2002/2026-03-19-exceptions.pdf
[14] courses/csse2002/java-hoare-logic.pdf
[15] courses/csse2002/week8-tutorial-class-invariants.pdf
[16] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf

```
    return result;
}
```

Tags

Methods:

Tag	Argument	Meaning
`@param`	*varname …*	Describe a parameter for a method
`@return`	…	Describe the return value
`@throws`	*ExceptionType …*	Describe when a particular type of exception is thrown by the method
`@requires`	*Precondition*	Specify the assumptions for a method to execute properly
`@ensures`	*Postcondition*	Specify the effects of executing the method

Classes:

Tag	Argument	Meaning
`@author`	*authorname*	Specify the author of the class
`@copyright`	…	Provide details on who owns copyright

> **i** The contract tags are spelled two ways
>
> The tag table above and the `findMax` example use `@requires` / `@ensures`. The formal `withdraw` specification and the contract examples use `@require` / `@ensure`, with no `s`. The same split runs across the course as a whole. Neither form is declared canonical, and the examples below use both. `@requires` and `@ensures` are **not** standard Javadoc tags. They are the course's own convention layered on top of it, which is why the generator will not do anything special with them.

What makes a good specification

Ideally a specification will:

- Allow the use of a method **by only reading the specification**, not by reading and understanding the implementation.
- Allow a method to be **re-implemented** without requiring changes to any methods that use it.

And it should:

- Rule out all implementations that are unacceptable — i.e. be sufficiently **restrictive**.
- Not preclude acceptable implementations — i.e. be sufficiently **general**.
- Be easy for users (programmers) to understand — i.e. be **clear**.

The next three sections take one method, `search`, and fail each of those three tests in turn.

Restrictiveness — keep out incorrect implementations

```
/**
 * Returns an index (i) of ar such that
 * ar[i] == x, if any.
 */
public int search(int[] ar, int x) {…}
```

This does not say what happens if `x` is not in `ar`. It is tempting to assume the obvious answer, but the specification doesn't say, so don't assume. By its silence, the specification allows any value to be returned. So a return of 1 or 0 could mean "It's at index 1/0" or "I didn't find it".

Adding the missing case makes it **better, still bad**:

```
/**
 * Returns an index (i) of ar such that
 * ar[i] == x, if any
 * else, return -1
 */
public int search(int[] ar, int x) {…}
```

Suppose `x` appears in `ar` multiple times. There is nothing in the spec that requires:

- that you will get the lowest index;
- that you will get the same answer running multiple times.

The specification is **non-deterministic**.

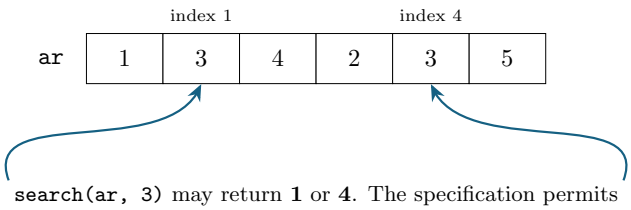


Figure 13: With the spec worded as 'returns an index i such that ar[i] == x', both cells holding 3 are legal answers, and nothing forbids the method returning a different one on each call. That is what makes the specification non-deterministic.

Generality — allow acceptable alternative versions

The obvious fix for the non-determinism is to say which index you get. Saying it like this, however, is a **bad example**:

```
/**
 *  Examine ar[0], ar[1], ... in turn and
 *  return the index of the
 *  first one that is equal to x, if any,
 *  else return -1
 */
public int search(int[] ar, int x) {…}
```

It is restrictive enough, but it over-commits: it describes *how* the method walks the array. A forward-scanning implementation goes with it (see the warning below before trusting this listing):

```
public int search(int[] ar, int x) {
    int found = -1;
    for (int i = 0; i < ar.length; i++) {
        if (ar[i] == x) {
            found = i;
        }
    }
    return found;
}
```

The consequence of the over-committed wording shows when the loop is run the other way:

```
public int search(int[] ar, int x) {
    int found = -1;
```

```
    for (int i = ar.length-1; i >= 0; i--) {
        if (ar[i] == x) {
            found = i;
        }
    }
    return found;
}
```

Scanning backwards and keeping the last match found *is* the smallest matching index — the answer the spec asks for — but the specification said **`ar[0], ar[1], ...` in turn**, so this version **does not satisfy the specification**. A perfectly good implementation has been outlawed by wording that was never about the outcome.

Even if you think that is the most natural way to do something, that doesn't mean it should be part of the specification.

> Specs should describe what happens, not how.

Rewritten to talk about the result only, the specification now accepts the backward implementation:

```
/**
 * Return smallest index (i) of ar such that
 * ar[i] == x, if any,
 * else return -1
 */
public int search(int[] ar, int x) {…}
```

> ⚠ The forward listing does not return the smallest index
>
> The backward scan above returns the smallest index, so it satisfies the rewritten spec. The forward scan from earlier does not:
>
> ```
> for (int i = 0; i < ar.length; i++) {
>     if (ar[i] == x) { found = i; }
> }
> return found;
> ```
>
> That loop overwrites `found` on every match, so it returns the **largest** matching index, not the smallest — and not the *first* one either, which is what the earlier spec asked for. On `ar = [1,3,4,2,3,5]` with `x = 3` it returns 4 where both specs want 1. The forward and backward listings only agree when `x` occurs at most once. A forward scan that satisfies the spec has to stop at the first match:

```
for (int i = 0; i < ar.length; i++) {
    if (ar[i] == x) { return i; }
}
return -1;
```

Clarity

- A specification should facilitate communication **between people**.
- Two ways in which communication could fail:
  - the reader doesn't understand;
  - the reader only *thinks* they understand.
- Clarity can be improved by being as concise as possible. Long specs are:
  - more likely to contain errors/contradictions,
  - less likely to be read properly,
  - more likely to be misunderstood.
- **Redundancy may help** — but be clear that it *is* redundant, by using e.g. "e.g." or "i.e.".

Four specs to judge

Four one-line specifications. The question for each is whether it is a good specification.

```
/**
 * Sorts the given array.
 */
public void sort(int[] ar) {...}
```

```
/**
 * Calculate x to the power of two with Math.pow.
 */
public int square(int x) {...}
```

```
/**
 * Returns the list of ResourceContainers stored by this instance.
 */
public List<ResourceContainer> getResources() {...}
```

```java
/**
 * Calculate the nth value of the fibonacci sequence.
 */
public long fib(int n) {...}
```

Applying the three criteria (restrictive, general, clear):

- **`sort`** — under-restrictive. Ascending or descending? In place, or is the return of `void` supposed to imply in place? What about equal elements?
- **`square`** — over-general in one direction and over-specific in another: it names `Math.pow`, which is a *how*, exactly the generality failure above; and it never says what happens on overflow.
- **`getResources`** — the mutability trap from java-mutability[17]. Is the returned `List` the instance's own list, so that the caller can add to it, or a copy? Restrictiveness: the spec's silence permits either, so no caller can safely do anything with it.
- **`fib`** — under-restrictive on the domain and on the indexing convention. Is `fib(0)` 0 or 1? What happens for negative `n`, or for an `n` whose answer overflows `long`?

Formality

Specifications of software range in formality:

- **informal** — e.g. normal comments;
- **semi-formal** — structured English documentation, e.g. using Javadoc tags;
- **formal** — mathematical constraints, e.g. using Java syntax for boolean expressions.

Here is the *same* method specified three times, once at each level.

Informal:

```java
/**
 * Withdraws an amount from this account and
 * returns how much is left.
 */
public int withdraw(int amount) {...}
```

This leaves three things open: what happens when `amount` is negative, what happens when `amount` is bigger than the balance, and whether the balance is changed when there is a failure.

Semi-formal — structured English, one tag per obligation:

[17] courses/csse2002/java-mutability.pdf

```java
/**
 * Withdraw an amount from this account.
 *
 * @param amount the amount to withdraw, must be a positive integer.
 * @return the new balance after the withdrawal.
 * @throws IllegalArgumentException if the amount is negative.
 * @throws InsufficientFundsException if the amount is greater than the current
  ↪  balance.
 */
public int withdraw(int amount) {...}
```

Formal — boolean expressions in Java syntax, with `\old(...)` naming the value an expression had *before* the call:

```java
/**
 * Withdraws an amount from this account.
 *
 * @require amount >= 0
 * @require amount <= getBalance()
 * @ensure getBalance() == \old(getBalance()) - amount
 */
public int withdraw(int amount) {...}
```

There is no `@throws` in the formal version. The failure cases have not been forgotten: they have been moved into the caller's obligations. The next section covers this.

Contracts

Specifications can be written using contracts:

```java
/**
 * [Description of method's behaviour]
 *
 * @require [Precondition]
 * @ensure [Postcondition]
 */
```

- If the code calling the method **satisfies the precondition**, then the method guarantees to satisfy the postcondition.
- If the code calling the method **does not satisfy the precondition**, then the method **guarantees nothing** — any behaviour is allowed.

Programming *with* contracts

`\result` names the returned value inside a postcondition:

```
/**
 * Withdraws an amount from this account.
 *
 * @require amount >= 0
 * @require amount <= getBalance()
 * @ensure getBalance == \old(getBalance()) - amount
 * @ensure \result == getBalance()
 */
public int withdraw(int amount) {
    // no need to check the conditions
}
```

```
int balance = myAccount.withdraw(amt);
```

In the first `@ensure`, `getBalance` is written without the parentheses it has in the formal specification above.

It is up to the calling code to make sure it fulfils its side of the contract, i.e. `0 <= amt <= myAccount.getBalance()`.

The method body carries no validation at all, and the call site carries no error handling — the postcondition is unconditional, so `balance` is always meaningful.

Programming *without* contracts

The same method written defensively instead, with the failure folded into the return value:

```
/**
 * Withdraw an amount from this account.
 *
 * @param amount The amount to withdraw.
 * @return The balance of this account after
 *         the withdrawal successfully occurs,
 *         or -1 if it fails.
 */
public int withdraw(int amount) {
    if (amount < 0) {
        return -1;
    }
    if (balance < amount) {
        return -1;
    }
}
```

```
int balance = myAccount.withdraw(amt);
If(balance == -1) {
    ...
}
```

The method body shows only the two checks; the success path is left out. The call site writes `If` with a capital where Java needs `if`.

The cost is visible on both sides: the method now checks twice on every call, and every call site has to branch on a sentinel. It also re-introduces exactly the restrictiveness problem from `search` — `-1` has to mean both "a balance" and "it failed".

Defensive programming

> **Definition (Defensive Programming)** Explicitly checking for invalid inputs and bad situations, ensuring the software does not behave dangerously regardless of input.

Sooner or later someone will call the method without checking the precondition. Particularly when dealing with external input sources or when guarding critical resources, it may be better to be defensive **outside the wall** and use contracts **inside** it.

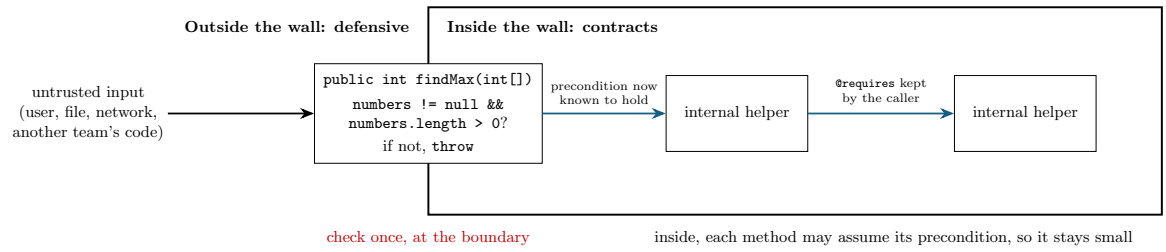


Figure 14: Defensive outside the wall, contracts inside it: validate defensively where untrusted input crosses into the system, and rely on contracts for the internal calls behind that boundary, so the check happens once rather than at every hop. The gate box is the findMax method worked through below.

Worked example: `findMax`

The starting point, with no specification and no checks:

```java
public int findMax(int[] numbers) {
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < numbers.length-1; i++) {
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max;
}
```

Documented, and with the defensive check added:

```java
/**
 * Finds the maximum value in the given array of integers.
 *
 * @param numbers An array of integers from which the maximum value is to be found.
 * @return The maximum value in the array.
 * @throws IllegalArgumentException if the array is null or empty.
 *
 * @requires numbers != null && numbers.length > 0
 *    - The array must not be null.
 *    - The array must contain at least one element.
 *
 * @ensures
 *    - The method returns the maximum value found in the array.
 *
 */
public int findMax(int[] numbers) {
    if (numbers == null || numbers.length == 0) {
        throw new IllegalArgumentException("Array must not be null or empty.");
    }

    int max = Integer.MIN_VALUE;
    for (int i = 0; i < numbers.length; i++) {
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max;
}
```

This version writes the precondition down as `@requires`, promises an exception in `@throws` for callers who break it anyway, and then *checks it in the body* — belt and braces, which is the whole point of being defensive at a boundary. Without the check, a `null` array throws `NullPointerException` and an empty array silently returns `Integer.MIN_VALUE` as if that were a real maximum.

The loop bound differs between the two versions

The first version loops `i < numbers.length-1` and the documented version loops `i < numbers.length`. The first version therefore never examines the last element, so `findMax(new int[]{1, 2, 9})` returns 2. No specification was attached to the first version, so nothing about it was wrong *according to its spec*, because it did not have one.

Further reading

- Preconditions, Postconditions, and Class Invariants
- Assertions

Key takeaways

- A specification describes the **contract between a method and its client**.
- It explains **what** the method does, not **how** it is implemented.
- A good specification clearly states:
  - **Preconditions** — conditions that must be true before the method is called;
  - **Postconditions** — conditions that will be true after the method finishes successfully;
  - **Side effects** — any additional changes the method makes to the system.
- Clear specifications make code easier to understand, use, and maintain.