CSSE2002 — Week 10 Notes

Table of contents

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

```java
@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

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

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

Note `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 exploit the fact that most of a maze's tiles are either walls or empty (i.e. an invariant/regularity in the data), letting the format avoid storing every single tile individually.

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

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(); }
}
```

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

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

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

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

These classes are (unrealistically) tightly coupled. 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, just where it occurs.

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`

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).

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.

Reference material

Java Cohesion and Coupling

Introduced in 2026-04-02-refactoring[4] (Lecture, Week 6, Part 2).

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 a question: what if the resulting modules are highly dependent on each other?

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).

[4] courses/csse2002/2026-04-02-refactoring.pdf

Worked classification examples

Classifying real code against the levels above (my own reasoned classification, following the definitions above — the lecture posed these as open discussion questions without a printed answer key):

```
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
}
```

`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[5]) worth considering bundling into an object.

[5] 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.

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.

Java I/O

Introduced in 2026-04-23-java-io[6] (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.

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. We often don't want to rewrite our program just because the source or destination changed.

- An **output stream** is an abstract destination to be written to.

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

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

Java has two families of streams:

- **Byte streams** (`InputStream`/`OutputStream`) — raw binary data (files, images, network data).
- **Character streams** (`Reader`/`Writer`, see below) — characters/text, handling Unicode.

  A byte stream reads data as bytes, whereas a character stream reads data as characters. (Schildt, *Java: The Complete Reference*, 11th ed.)

`InputStream` / `OutputStream`

`InputStream` and its subclasses represent a stream of bytes, drawn from different sources (`FileInputStream` from a file, `ByteArrayInputStream` from an array, …). Methods that use streams should accept the **superclass** type as a parameter, so any concrete stream can be substituted (see java-solid-principles[7] — Liskov Substitution):

```
private static void sendToStream(OutputStream stream) throws IOException {
    String output = "foo bar baz";
    for (char letter : output.toCharArray()) {
        stream.write(letter);
    }
    stream.flush();
}
// sendToStream(System.out);
// sendToStream(new FileOutputStream("output.txt"));
// sendToStream(new ByteArrayOutputStream());
```

End of file: all input streams need to consider "end of file" — `read()` returns `-1` at EOF, so a loop reading one byte at a time typically continues `while (in != -1)`.

Buffering

Reading a file a byte at a time is slow. `BufferedInputStream` wraps another input stream and buffers reads (the buffer is an area of main memory used to temporarily hold data) — on one test VM, reading a 4MB file took 82ms buffered vs. 18,193ms unbuffered:

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

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

Closing streams

Streams (and Readers) need closure — systems may limit how many files can be open at once, so always `close()` a stream when finished with it. Wrapping cleanup in `try/catch/finally` 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 automatically closed (Effective Java, 3rd ed., Item 9: "Prefer try-with-resources to try-finally"):

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

Readers & Writers

`Reader/Writer` are the character-based counterparts of `InputStream/OutputStream`. `InputStreamReader` bridges the two: it wraps an `InputStream` (like `System.in`) and decodes its bytes into characters.

```
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);
}
```

```
// readAndPrint(new InputStreamReader(System.in));
// readAndPrint(new FileReader("myfile.txt"));
```

`BufferedReader`

Wraps another `Reader`; as well as buffering, it adds `String readLine()`:

```
BufferedReader reader = new BufferedReader(new FileReader("readwithbuffer.txt"));
for (int i = 0; i < 5; i++) {
    System.out.println(reader.readLine());
}
```

`PrintWriter`

`System.out` is actually a `PrintStream`; `PrintWriter` is a better option for character output — it can write to many destinations and supports formatted text (`printf`):

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

`flush()`

If an `OutputStream`/`Writer` is buffered, output might not be sent immediately — `flush()` forces any pending output out. This matters for interactive situations (the other end won't respond if nothing's actually been sent yet) and for debugging/logging (an up-to-date view of what's happening). `close()`-ing a stream flushes 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/strings (a friendlier alternative to `BufferedReader`). It also supports regex-based scanning for advanced use cases.

```java
Scanner scanner = new Scanner(System.in); // internally wraps its own
 ↪  InputStreamReader
int total = 0;
while (scanner.hasNextInt()) {
    total += scanner.nextInt();
}
System.out.println(total);
```

Reading strings: `scanner.next()` reads the next word (skipping leading whitespace, stopping at whitespace); `scanner.nextLine()` reads the entire line up to Enter.

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

`java.io` is mainly stream-oriented — its goal was reading/writing data, not managing files or directories, so it's awkward for file attributes, directory traversal, and path manipulation. `java.nio.file` (since Java 1.7) adds two key abstractions:

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

Creating a `Path`

```java
Path p1 = Path.of("docs/output.txt");        // whole path as one string
Path p2 = Path.of("docs", "output.txt");     // separate name elements, joined by
 ↪  Java
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)`
Manipulate	`Files.copy(Path, Path)`, `Files.move(Path, Path)`

Category	Methods
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.

```
Path myPath = Path.of("src", "Week08", "NIO", "myfile.txt");
List<String> lines = Files.readAllLines(myPath);
for (String line : lines) {
    System.out.println(line);
}
```

Summary: which to use?

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

Java Specification

Introduced in 2026-03-12-object-oriented-programming-ii[8] (Lecture, Week 3, Part 2).

Javadoc

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

[8] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf

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

Common tags:

Tag	Meaning
`@param varname ...`	Describe a parameter
`@return ...`	Describe the return value
`@throws ExceptionType ...`	Describe when a particular exception is thrown
`@requires Precondition`	Assumptions for the method to execute properly
`@ensures Postcondition`	Effects of executing the method
`@author authorname`	Author of the class

What makes a good specification

Ideally, a specification should:

- Allow a method to be *used* by only reading its specification, not its implementation.
- Allow a method to be *re-implemented* without requiring changes to its callers.
- Be **restrictive** enough to rule out unacceptable implementations.
- Be **general** enough to not preclude acceptable (alternative) implementations.
- Be **clear** enough for programmers to understand.

Restrictiveness — keep out incorrect implementations

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

What happens if `x` isn't in `ar`? The spec doesn't say — by its silence it allows *any* return value, so a caller can't distinguish "found at index 0" from "not found". Adding `else, return -1` fixes that, but if `x` appears multiple times, nothing requires the lowest index or a consistent answer across calls — the spec is still non-deterministic unless it says `smallest index`.

Generality — allow acceptable alternative versions

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

This is a **bad** spec even though it's restrictive: it describes *how* (forward iteration order), not *what*. A backward-iterating implementation that returns the same first-match index would violate this over-specific wording despite being equally acceptable — specs should describe outcomes, not implementation strategy. Prefer wording like "return the **smallest** index such that…", which both rules out bad implementations and permits any implementation strategy that achieves the same outcome. Both a backward-iterating and an early-`return`-on-first-match forward implementation satisfy this better wording equally well:

```
public int search(int[] ar, int x) {
    for (int i = 0; i < ar.length; i++) {
        if (ar[i] == x) {
            return i; // early return, still finds the smallest index
        }
    }
    return -1;
}
```

Clarity

A specification should facilitate communication — it can fail either because the reader doesn't understand, or because the reader only *thinks* they understand. Clarity improves by being concise (long specs are more likely to contain contradictions, be skipped, or be misread) and by marking any deliberate redundancy explicitly (e.g. with "e.g." or "i.e.").

Formality

Specifications range from informal to formal:

- **Informal** — plain comments, e.g. `// Withdraws an amount and returns how much is left.` Leaves open questions (what if `amount` is negative? bigger than the balance? is the balance changed on failure?).
- **Semi-formal** — structured English via Javadoc tags (`@param`, `@return`, `@throws`).
- **Formal** — mathematical/boolean constraints, e.g. `@require`/`@ensure` using Java boolean-expression syntax:

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

Contracts

A specification can be written as a **contract**:

- If the caller satisfies the precondition, the method guarantees to satisfy the postcondition.
- If the caller does *not* satisfy the precondition, the method guarantees nothing — any behaviour is allowed, and the method body doesn't need to check for it.

This contrasts with a defensive, "no contract" style that instead documents and handles every failure case explicitly (e.g. returning a sentinel value like `-1` on invalid input) — contracts push that responsibility onto the caller instead.

Defensive programming

> Explicitly checking for invalid inputs and bad situations, ensuring the software does not behave dangerously regardless of input.

Even with a documented precondition, some caller eventually won't check it. When dealing with external input or guarding critical resources, it's often safer to validate defensively at the system boundary (throwing on bad input, e.g. `IllegalArgumentException` for a null/empty array) while relying on well-defined contracts *between* internal methods:

```
/**
 * Finds the maximum value in the given array of integers.
 *
 * @throws IllegalArgumentException if the array is null or empty.
 * @requires numbers != null && numbers.length > 0
```

```
 * @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;
}
```

Further reading: Preconditions, Postconditions, and Class Invariants, Assertions, and Effective Java (3rd ed.), Item 56: Write doc comments for all exposed API elements.