CSSE2002 — Week 7 Notes

Table of contents

SOLID Principles II

See csse2002[1] for course logistics. Continues 2026-04-02-refactoring[2] (Week 6) — Single Responsibility and Open-Closed were covered there; this lecture covers the remaining three SOLID principles.

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

Today's outline

1. Liskov Substitution Principle (and substitution with contracts)
2. Interface Segregation Principle
3. Dependency Inversion Principle

All content, including the worked examples, is in java-solid-principles[3] (updated this week with the L, I, D sections).

Applied class

No separate applied-class topic on SOLID this week — see week7-tutorial-black-box-and-glass-box-testing[4] for this week's applied class (black-box/glass-box testing, a continuation of java-testing[5] from Week 5).

Black-box and Glass-box Testing

Applied class for 2026-04-16-solid-principles-ii[6] (Week 7). Applies the black-box/white-box testing concepts from java-testing[7] (Week 5) to a worked example. All of these tasks ask for *a* set of inputs achieving some coverage goal — many different answers are equally correct, so answers here are reveal-only worked examples rather than checkable exercises.

The `Bus` class

```
public class Bus {
    public Bus(int capacity) {...}
    public int getCurrent() {...}
    public int getAverageCount() {...}
    public void stop(int on, int off) {...}
}
```

[3] courses/csse2002/java-solid-principles.pdf
[4] courses/csse2002/week7-tutorial-black-box-and-glass-box-testing.pdf
[5] courses/csse2002/java-testing.pdf
[6] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[7] courses/csse2002/java-testing.pdf

Task 0 — write a smoke test

Working

A smoke test just exercises normal/expected operation:

```
Bus bus = new Bus(10);
bus.stop(5, 0);
bus.stop(3, 1);
assertEquals(7, bus.getCurrent());
assertEquals(6, bus.getAverageCount());
```

Task 1 — develop boundary scenarios (with justification)

Working

Scenario	Justification
Construct a bus with capacity 0	A bus that can have no passengers is an uncommon scenario we may expect bugs from.
Construct a bus with a negative capacity	Unclear how this should behave — worth clarifying and testing.
Ask for the average after no stops	Averages usually involve division by count; want to confirm no exception when count is 0.
Negative values for `on`/`off`	Unclear behaviour — does a negative `on` remove passengers?
More passengers get off than are currently on the bus	Results in undefined behaviour.
More passengers on the bus than capacity	Behaviour isn't specified, so worth observing.

Task 2 — how can we make `Bus`'s behaviour fully specified?

> **Working**
>
> Most of the boundary scenarios above come from unspecified/unclear behaviour. If we declare `getCurrent() <= capacity` an **invariant** of `Bus` (see java-specification[8]), we must protect it everywhere:
>
> 1. Constructor throws `IllegalArgumentException` if `capacity < 0`.
> 2. `stop` gets a precondition that both `on` and `off` are `>= 0`.
> 3. `stop` throws `IllegalStateException` if `getCurrent() + on - off` would be negative or exceed `capacity`.

Glass-box testing: code coverage levels

See java-testing[9] for the three levels (statement, branch, path). Consider:

```java
public static int countOccurrences(int[] array, int target) {
    int count = 0;
    for (int num : array) {
        if (num == target) {
            count++;
        }
    }
    return count;
}
```

A single array containing the target at least once (e.g. `[2, 3, 2]`, target `2`) gives statement coverage (the `if` body runs); adding an array where the target never appears (e.g. `[3]`, target `2`) gives branch coverage too.

Worked example: `ascending`

```java
/**
 * @require numbers != null
 * @ensure \result is true iff for all indices j such that
 *         0 <= j < numbers.length - 1, numbers[j] <= numbers[j + 1]
```

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

[9] courses/csse2002/java-testing.pdf

```
 */
public boolean ascending(int[] numbers) {
    boolean result = true;
    for (int i = 0; i < numbers.length - 1; i++) {
        if (numbers[i] > numbers[i + 1]) {
            result = false;
        }
    }
    return result;
}
```

i Working

Statement coverage: `[2, 1]` (loop runs once, `if` body executes).
Branch coverage: `[2, 1, 2]`, or the pair `[2, 1]` and `[1, 2]` (need the `if` to be both true and false).
Path coverage (0, 1, and 2-iteration cases, crossed with the `if` outcome each time):

Input	Justification
`[]` or `[3]`	0 times through loop
`[3, 4]`	1 time through loop, `if` false
`[4, 3]`	1 time through loop, `if` true
`[3, 4, 5]`	2 times through loop, false, false
`[3, 4, 2]`	2 times through loop, false, true
`[4, 3, 5]`	2 times through loop, true, false
`[5, 4, 3]`	2 times through loop, true, true

Worked example: `maximum` (bonus)

```
/**
 * @require numbers != null
 * @ensure numbers.length > 0 ==> (
 *     (\forall int i; 0 <= i < numbers.length ==> \result >= numbers[i]) &&
 *     (\exists int i; 0 <= i < numbers.length && \result == numbers[i]))
 * @ensure numbers.length == 0 ==> \result = Integer.MIN_VALUE
 */
public int maximum(int[] numbers) {
    int currentMaximum = Integer.MIN_VALUE;
    for (int number : numbers) {
        if (number > currentMaximum) {
            currentMaximum = number;
```

```
        }
    }
    return currentMaximum;
}
```

> **Working**
>
> **Statement coverage**: `[1]`.
> **Branch coverage**: `[2, 1]` (first element sets a new max, second doesn't).
> **Path coverage**:

Input	Justification
`[]`	0 times through loop
`[Integer.MIN_VALUE]`	1 time through loop, `if` false
`[1]`	1 time through loop, `if` true
`[Integer.MIN_VALUE, Integer.MIN_VALUE]`	2 times through loop, false, false
`[Integer.MIN_VALUE, 1]`	2 times through loop, false, true
`[2, 1]`	2 times through loop, true, false
`[1, 2]`	2 times through loop, true, true

Worked example: `indexOf` (bonus)

```
/**
 * @require numbers != null
 * @ensure numbers[\result] = number ||
 *     (\result == -1 &&
 *      \forall int j; 0 <= j < numbers.length ==> numbers[j] != number)
 */
public int indexOf(int[] numbers, int number) {
    for (int i = 0; i < numbers.length; i++) {
        if (numbers[i] == number) {
            return i;
        }
    }
    return -1;
}
```

Working

Statement coverage: `indexOf([], 1)` and `indexOf([1], 1)`.
Branch coverage: `indexOf([], 1)` and `indexOf([2, 1], 1)`.
Path coverage — note that with an early `return` inside the loop, "2 iterations, `if` true on the first" is unreachable (the method returns immediately), so two of the theoretically-possible paths for a 2-element input can't actually be tested:

Input	Justification
`indexOf([], 42)`	0 times through loop
`indexOf([1], 2)`	1 time through loop, `if` false
`indexOf([1], 1)`	1 time through loop, `if` true
`indexOf([1, 2], 3)`	2 times through loop, false, false
`indexOf([1, 2], 2)`	2 times through loop, false, true
Cannot be tested	2 times through loop, true, false
Cannot be tested	2 times through loop, true, true

Reference material

Java SOLID Principles

Introduced in 2026-04-02-refactoring[10] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[11] (Lecture, Week 7 — Liskov Substitution, Interface Segregation, Dependency Inversion).

Why SOLID?

Large software tends to become:

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

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

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

Single Responsibility Principle (SRP)

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

Closely related to java-cohesion-and-coupling[12]'s cohesion: cohesion is about how *similar* a class's functionality is, while SRP is about the *reasons for change*. A "responsibility" is a reason for change — if you can think of more than one motive to change a class, it has more than one responsibility.

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

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

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

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

Open-Closed Principle (OCP)

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

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

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

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

How do we keep a class closed for modification? Information hiding (see java-encapsulation[13]).

Example: pluggable grading systems

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

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

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

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

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

```
}

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

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

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

Liskov Substitution Principle (LSP)

> Subclasses should be substitutable for their parent classes.

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

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

Example violation

```
class Student {
    protected List<Course> courses;
    public double calculateGPA() {
        double totalPoints = 0;
        for (Course course : courses) {
            totalPoints += course.getGradePoints();
```

```
        }
        return totalPoints / courses.size();
    }
}
class PostgraduateStudent extends Student {
    @Override
    public double calculateGPA() {
        throw new UnsupportedOperationException("Postgraduates have a different GPA
        ↪ system.");
    }
}
```

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

Substitution with contracts

Even when a subclass overrides a method with a different implementation, it must still honour the parent's contract (see java-specification[14]):

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

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

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

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

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

Worked example

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

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

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

Interface Segregation Principle (ISP)

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

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

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

This mirrors java-cohesion-and-coupling[15]'s stamp coupling at the interface level: despite each client using only a small subset of `Z`, every client depends on *all* of `Z` — so a change to any part of `Z` risks affecting (and forcing a recompile/redeploy of) every client, even ones that never used the changed part. Splitting `Z`'s surface into focused interfaces removes those incidental dependencies:

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

Dependency Inversion Principle (DIP)

> Depend upon abstractions. Do not depend upon concretions.

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

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

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

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

What makes a good dependency? (stability and exposure)

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

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

Two forms of DIP

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

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

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

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

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

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

Dependency Injection

Dependency Injection (DI) is a design pattern that implements DIP by injecting a class's dependencies into it (e.g. via the constructor) instead of having the class create them internally, promoting loose coupling (see java-cohesion-and-coupling[16]):

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

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

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

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

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

Summary

- **S**ingle Responsibility — one reason to change per class.
- **O**pen-Closed — open for extension, closed for modification.
- **L**iskov Substitution — subclasses substitutable for their parents; no stronger preconditions, no weaker postconditions.
- **I**nterface Segregation — many small, client-specific interfaces beat one large general-purpose interface.
- **D**ependency Inversion — depend on abstractions, not concretions; inject dependencies rather than constructing them internally.

Java Specification

Introduced in 2026-03-12-object-oriented-programming-ii[17] (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).

```
/**
 * 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

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

Tag	Meaning
`@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

```
/**
 * 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:

```java
/**
 * 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.

Java Testing

Introduced in 2026-03-26-testing[18] (Lecture, Week 5).

Levels of testing

1. **Unit testing** — check that each "unit" in the project behaves correctly.
2. **Integration testing** — check that components work together and that the interfaces between components work as expected.
3. **System testing** — does the system as a whole work correctly?
4. **(User) acceptance testing** — do users of the system agree that the system does what it's supposed to do?

These form a pyramid, from many low-level unit tests at the base up to a few acceptance tests at the top.

Regression testing

During development, testing helps answer two questions: does the new stuff work, and have we broken things that used to work (**regressed**)? Regression testing ensures old features keep working as new features are introduced.

Black box vs white/glass box testing

- **Black box** — the software has inputs and outputs to test, but the implementation is unknown to you; you test according to the *specification* (see java-specification[19]) — what it's supposed to do.
- **White/glass box** — testing designed *with* knowledge of the internal implementation, so tests can pay special attention to cases where the implementation is complicated.

Black box techniques

Smoke tests — test the common-case functionality first ("can it run?"), to decide whether more rigorous testing is worthwhile. The name comes from electronics testing: plug in a board, turn on the power — if you see smoke, stop.

Boundary tests — investigate extremes and corner cases, since that's where bugs often occur:

[18] courses/csse2002/2026-03-26-testing.pdf
[19] courses/csse2002/java-specification.pdf

Input type	Try
Whole number	0, -1, minimum value, maximum value
Floating point	0, -1, `NaN`, infinity
Collection (array, list, set…)	empty, one element, large
Reference type	`null`
Resource (file, link)	non-existent resource

Equivalence classes — groups of inputs that the system treats the same way. E.g. for `getCurrentPassengers()` on a bus with capacity 80: `Valid-Empty` (0), `Valid-Normal` (1-79), `Valid-Full` (80), `Invalid` (anything else). The corresponding *boundary* tests probe each class's edges: `-1, 0, 1` (around empty), `79, 80, 81` (around full capacity) — giving a minimal boundary set of `-1, 0, 1, 79, 80, 81`.

White box technique: code coverage

Of all the ways a program could run, how many are covered by the tests? Three levels, from weakest to strongest:

1. **Statement coverage** — every statement is executed at least once.
2. **Branch coverage** — every branch is tested for both the `true` and `false` case.
3. **Path coverage** — every distinct path through the code is traversed.

Example:

```
public void register(int x) {
    if (x > 0) {
        positives += 1;
    }
    if (x % 2 == 0) {
        evens += 1;
    }
}
```

- Statement coverage: `x = 2` alone covers every line (both `if` bodies run).
- Branch coverage: `x = 2` and `x = -1` together cover both branches of each `if` (true/false).
- Path coverage: needs all 4 combinations of the two conditions: `x=2` (true, true), `x=1` (true, false), `x=-2` (false, true), `x=-1` (false, false).

Loops make exhaustive path coverage infeasible — a loop like `for (int i = 0; i < 100; i++) { if (f(i, k)) { j++; } }` has 2^{100} paths (over a billion tests/second would still take ~40.2 trillion years, about 2900 times the age of the universe). Instead, path coverage for loops is **approximated**: treat 0, 1, and 2-or-more iterations (of a loop or recursive call) as equivalent ("engineers' induction: one, two, three — that's good enough for me").

Test Driven Development (TDD)

Originates from the Agile manifesto and Extreme Programming. Cycle: **write a (failing) test** → **write code** until the **test passes** → **refactor** → write the next test. Developers write small test cases for every feature based on their initial understanding, and only modify/write new code when a test fails (avoiding duplicated test scripts) — the tests determine when code is "ready".