CSSE2002 — Week 7 Notes

Table of contents

SOLID Principles II **2**
- Outline . . . 2
- Why SOLID (recap) . . . 2
- Liskov Substitution Principle . . . 3
- Interface Segregation Principle . . . 9
- Dependency Inversion Principle . . . 10
- Summary and key takeaways . . . 15
- Further reading . . . 15
- Applied class . . . 16

Black-box and Glass-box Testing **16**
- 1. Black box testing . . . 16
- 2. Glass box testing . . . 19
- 3. Ascending . . . 20
- 4. Maximum — bonus . . . 23
- 5. Index Of — bonus . . . 24

Reference material **27**
- CSSE2002 Diagram & Notation Conventions . . . 27
- Java Abstraction . . . 39
- Java Cohesion and Coupling . . . 43
- Java Collections Framework . . . 49
- Java Exceptions . . . 52
- Java Inheritance . . . 59
- Java JUnit . . . 63
- Java Loop Invariants . . . 70
- Java Polymorphism . . . 78
- Java Refactoring . . . 81
- Java SOLID Principles . . . 88
- Java Specification . . . 100

Java Testing 111

SOLID Principles II

See csse2002[1] for course logistics. Continues 2026-04-02-refactoring[2] (Week 6), which has Single Responsibility and Open-Closed. The remaining three SOLID principles are below. The code-level treatment of all five is in java-solid-principles[3].

Outline

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

Why SOLID (recap)

Large software tends towards becoming **rigid** and **fragile**.

- **Rigid software** — difficult to change. Changes, even small changes, require many modifications in many parts of the codebase.
- **Fragile software** — likely to break when a change is made. Sometimes the breakage occurs in parts of the codebase that *appear to have no relation to the change*.

SOLID principles are a way to help prevent rigidity and fragility. Many of them were floating around before becoming "SOLID"; Robert Martin combined them in *Design Principles and Design Patterns* (2000).

	Principle	Notes
S	Single Responsibility	Week 6 — 2026-04-02-refactoring[4]
O	Open-closed Principle	Week 6 — 2026-04-02-refactoring[5]
L	Liskov Substitution Principle	Week 7 — below
I	Interface Segregation Principle	Week 7 — below
D	Dependency Inversion	Week 7 — below

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/2026-04-02-refactoring.pdf
[3] courses/csse2002/java-solid-principles.pdf
[4] courses/csse2002/2026-04-02-refactoring.pdf
[5] courses/csse2002/2026-04-02-refactoring.pdf

Liskov Substitution Principle

> Subclasses should be substitutable for their parent classes.

Defined by Barbara Liskov in a 1987 keynote. In full: 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**. That last clause is the whole content of the principle.

The motivating example is an `Animal` supertype with `Cat` and `Mouse` subclasses, each declaring `+ eat(f: Food) : void`, and a client that only ever talks to the supertype (see java-polymorphism[6], java-inheritance[7]):

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

A violation: the postgraduate GPA

> Different types of students (undergraduates and postgraduates) should behave consistently when calculating GPA.

```
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.");
    }
}
```

[6] courses/csse2002/java-polymorphism.pdf
[7] courses/csse2002/java-inheritance.pdf

The failure is entirely in the client code:

```
Student undergraduate = new UndergraduateStudent();
double gpa = undergraduate.calculateGPA();

Student postgraduate = new PostgraduateStudent();
// This will throw an exception!
gpa = postgraduate.calculateGPA();
```

Both variables are declared `Student`. The second one is not usable where a `Student` is expected, so `PostgraduateStudent` is not a substitutable subclass. This compiles perfectly and only breaks at run time — see java-exceptions[8].

Substitution with contracts

Substitutability is made precise using the method's contract (see java-specification[9]):

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

A subclass of `Parent` can override `f`. Even when the implementation changes, **the override must still follow the contract commitments of the original version.**

> **Note**
>
> The course spells these tags two ways: `@requires` / `@ensures` as above, and `@require` / `@ensure` (no trailing `s`) as in the examples further down. The Week 4 tag table and the exam preparation both use `@requires` / `@ensures`. Neither spelling is "the" course form — see csse2002-diagram-notation[10].

[8] courses/csse2002/java-exceptions.pdf
[9] courses/csse2002/java-specification.pdf
[10] courses/csse2002/csse2002-diagram-notation.pdf

Preconditions

Suppose `Child extends Parent`, but `Child.f()` can deal with negative numbers as well. What should the precondition of `Child.f()` be? There are two options:

1. Leave it as `x > 0`.
2. Change it to `x != 0`.

A possible problem with option 1 is confusion for readers about why the limitation is there when the code is more capable.

Option 2 does not violate the substitution principle. Compare the **sets of inputs each precondition allows**. Everything in the first set (`x > 0`) is also in the second set (`x != 0`), so the second precondition is **weaker**, that is, less strict. It does not reject any state the first one would allow. Stated as an implication:

`x > 0` $\Rightarrow$ `x != 0`

> Preconditions in a subclass must be **no stronger** than those in the superclass; they may be weaker or remain the same.

Postconditions

What if `Child.f()` only returned `'K'`, `'L'`, or `'M'` — would `\result > 'J' && \result < 'N'` be an acceptable postcondition?

Yes. Every result still fits within the original postcondition, so the new postcondition is **stronger** (more restrictive). Stated as an implication, the `Child` postcondition implies the `Parent` postcondition:

`\result > 'J' && \result < 'N'` $\Rightarrow$ `\result > 'A' && \result < 'Z'`

> Postconditions in a subclass must be **no weaker** than those in the superclass; they may be stronger or remain the same.

The rule

> **A subclass must not strengthen preconditions or weaken postconditions.**

The implications run in opposite directions: for preconditions the *parent's* condition implies the *child's*; for postconditions the *child's* implies the *parent's*.

The funnel picture

The rule can be drawn as well as stated. A method is a box; the cone above it is the set of inputs it will accept, the cone below is the set of outcomes it may produce. Width means permissiveness, so a wide top is a weak precondition and a wide bottom is a weak postcondition.

Assume that B `extends` A

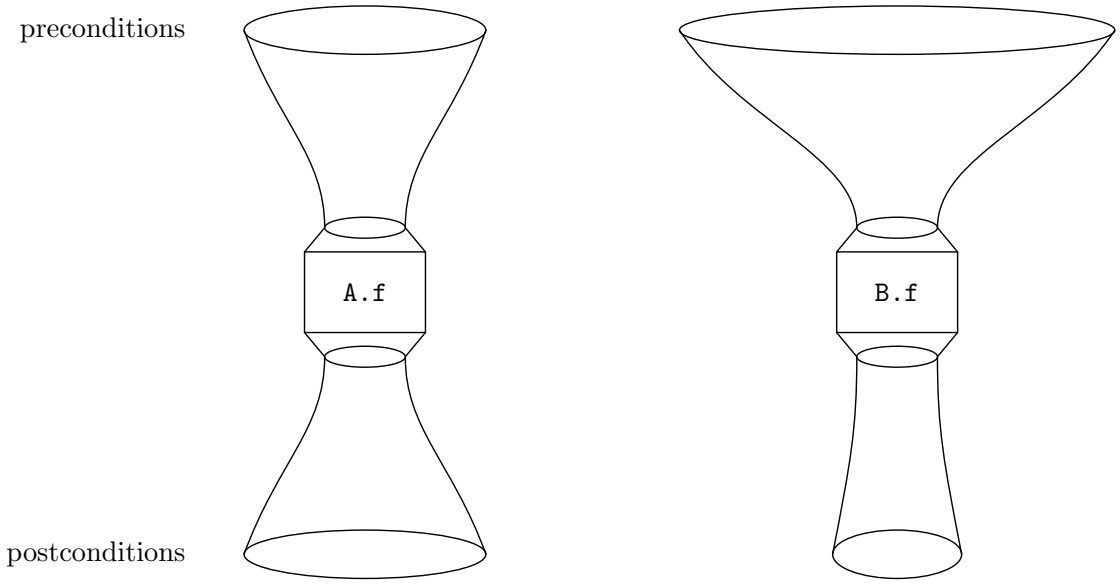


Figure 1: A method's contract drawn as a funnel. The cone above `A.f` is the set of inputs it accepts — its precondition; the cone below is the set of outcomes it may produce — its postcondition. A wider mouth is a weaker condition, so the legal override `B.f` has a wider top (weaker precondition) and a narrower bottom (stronger postcondition).

Superimposing the two funnels turns the rule into a containment statement:

Worked example: `One` and `Two`

The question. Does `Two` satisfy the substitution principle with respect to `One`?

```
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
```

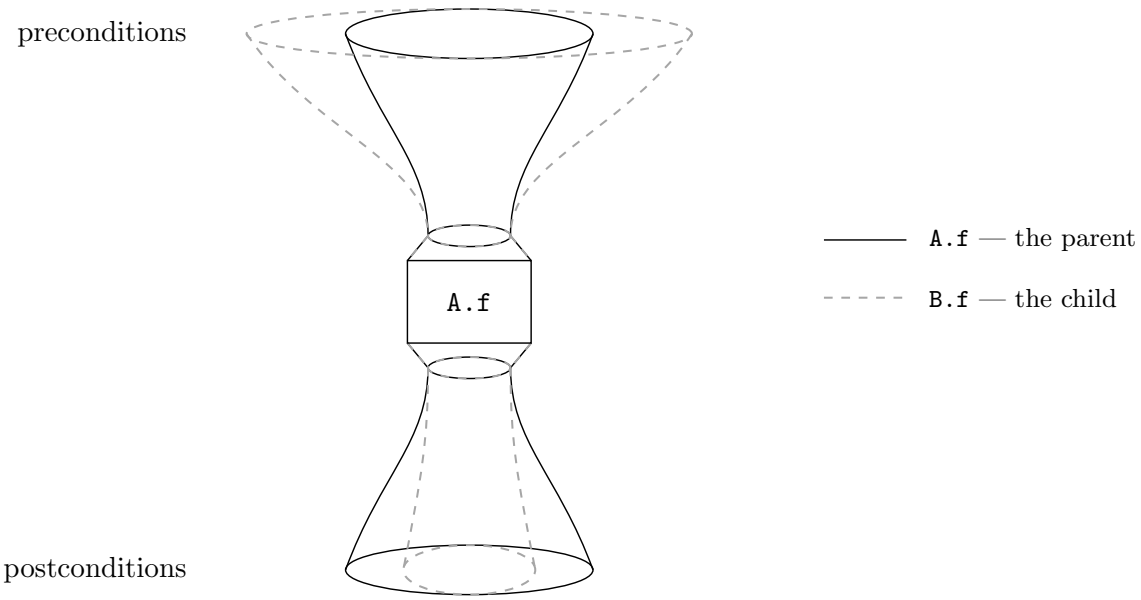


Figure 2: The same two funnels superimposed, the parent `A.f` solid and the child `B.f` dashed: the child's precondition mouth contains the parent's, and the parent's postcondition mouth contains the child's. The child should accept all inputs the parent accepts, and it should still guarantee outputs that satisfy everything the parent guarantees.

```
     */
    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 and col
     */
    @Override
    public List<Integer> calc(int[] row, int[] col, int val) {
    }
}
```

The answer. Read `Two`'s `@ensure` as "either row **or** col" (identical to `One`'s), so that the only thing left differing is the precondition. The input sets each precondition admits are:

	Class One	Class Two
`row`	…, -1, 0, 1, 2, 3 …	1, 2, 3, …
`col`	…, -1, 0, 1, …	…, -1, 0, 1, …
`val`	1, 2, 3, 4, …, 99	1, 2, 3, 4, …, 99

`col` and `val` admit exactly the same values in both. `row` does not: `One` accepts any integers, `Two` accepts only positive ones. So there are inputs `One` would accept that `Two` rejects — `One`'s precondition does **not** imply `Two`'s. That is a **strengthened precondition**, and it breaks substitutability: client code written against `One` that happens to pass a `row` containing a negative number would break the moment a `Two` were substituted in.

Reading it off Figure 1: `Two.calc`'s top mouth is *narrower* than `One.calc`'s, which is exactly the shape the rule forbids.

A variant (homework). Remove `&& val < 100 && row only contains positive integers` from `Two`'s precondition and change its postcondition to the integers appearing in **BOTH** `row` and `col`, leaving:

```
class Two extends One {
    /**
     * @require row != null && col != null && 0 < val
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in BOTH row and col
     */
```

```
    @Override
    public List<Integer> calc(int[] row, int[] col, int val) {
    }
}
```

`val < 100` is gone as well, so `Two`'s precondition is now *weaker* than `One`'s, not equal to it. That is allowed, so the precondition side is settled. The question left is entirely about the postcondition: does *that* satisfy the substitution principle?

> **Working**
>
> `One` promises the *union* — the positive integers below `val` appearing in `row` **or** `col`. `Two` now promises the *intersection* — those appearing in **both**. Take `row = [1, 2]`, `col = [2, 3]`, `val = 10`: `One` must return `[1, 2, 3]`, `Two` returns `[2]`. Both postconditions pin the result down to a single list, and those lists differ, so `Two`'s postcondition does **not** imply `One`'s. Since the rule demands exactly that implication, it fails and LSP is violated. (The two are in fact incomparable — neither implies the other — which is still a failure, because "no weaker" means "implies".)
>
> In funnel terms: `Two.calc`'s bottom mouth is not contained in `One.calc`'s; it juts outside it, and containment is what the rule demands. Client code that relied on `One`'s promise — say, looking up `3` in the returned list — silently gets a wrong answer rather than an exception, which is the fragility[11] SOLID exists to prevent.

This example is drilled again in week9-tutorial-substitution-principle-and-pre-post-conditions[12].

Interface Segregation Principle

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

ISP builds on **coupling** — the degree of interdependence between modules, classes or components — and its seven levels from `Content` (tightly coupled, "needy") down to `None` (loosely coupled, "independent"). The levels are from Week 6: see java-cohesion-and-coupling[13] for the levels and worked classifications, and week10-tutorial-coupling-and-cohesion[14] for the applied class.

The bridge from coupling to ISP is the `class R` example and its **risk of causing incidental dependencies**:

[11] courses/csse2002/java-solid-principles.pdf
[12] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf
[13] courses/csse2002/java-cohesion-and-coupling.pdf
[14] courses/csse2002/week10-tutorial-coupling-and-cohesion.pdf

```
public class R {
    int x, y, z;
    char a, b, c;
}

public static void compute(R r) {
    return r.x * r.y;
}

public static void run() {
    R r = new R(1, 2, 3, 'a', 'b', 'c');
    compute(r);
}
```

`compute` is handed the whole of `R` but only reads two of its six fields (stamp coupling). ISP is that observation applied to a type's public surface:

- **Shrink interfaces to minimize incidental dependencies.**
- **Larger interfaces should be split into smaller ones**, to ensure that implementing classes only need to be concerned about the methods that are of interest to them.

The problem

The cost: when `Z` changes, clients `A`, `B`, and `C` may all be affected, even if each uses only a small part of `Z`, so they may need to be recompiled and redeployed.

The solution

`Z` is unchanged — it still has all five methods. What changed is that no client can *see* more of `Z` than it needs, so a change to `d()` can no longer force `A` or `B` to be recompiled.

Dependency Inversion Principle

> Depend upon Abstractions. Do not depend upon concretions.

Entities must depend on abstractions, not on concretions. The high-level module must not depend on the low-level module; both should depend on abstractions.

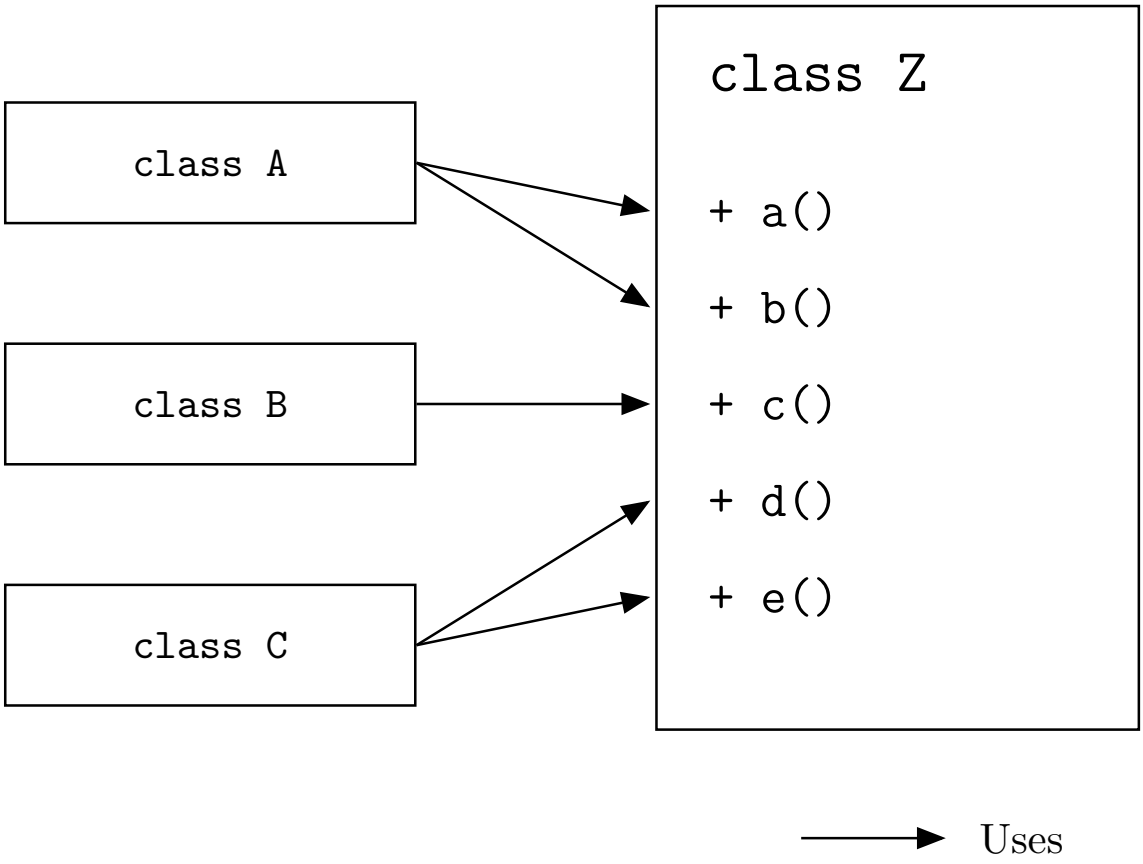


Figure 3: Despite only using a small subset of `Z`, each class inadvertently depends on all of `Z`. A node is a rectangle labelled with its Java keyword and name, with its members listed inside the same box as `+ a()` and no compartment rule. The solid `Uses` arrow points from the depender to the thing depended on, and ends on the member row it uses, not on the box edge.

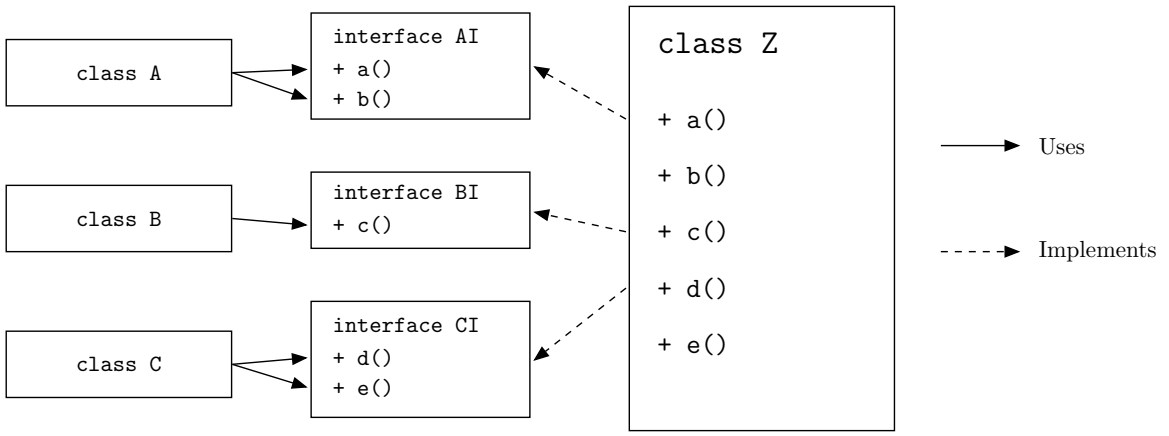
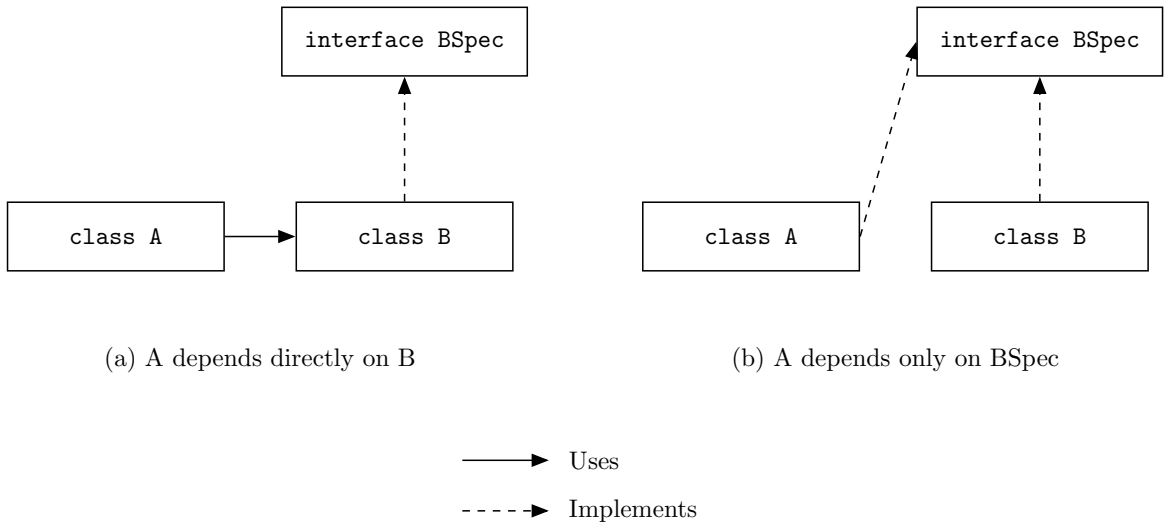


Figure 4: Create interfaces to minimize the incidental dependencies: `A` now depends only on `AI`, `B` only on `BI`, `C` only on `CI`, and `Z` implements all three. A solid line is Uses and a dashed line is Implements, both pointing from the depender to the thing depended on. Both have the same filled arrowhead, so only the line style tells them apart. The interfaces are named `AI`/`BI`/`CI` here and `BSpec` in the dependency inversion figure; the course does not settle on one naming scheme.



(a) A depends directly on B

(b) A depends only on BSpec

Figure 5: Inverting a dependency. In (a) `A` depends directly on the concrete `B`, so changes in the implementation can propagate to `A`. In (b) `A` depends only on `BSpec`, so changes inside `B` are less likely to affect `A` as long as `BSpec` stays the same. The edge from `A` to `BSpec` in (b) is drawn *dashed*, the `Implements` style, even though `A` merely uses `BSpec`.

The cheapest version: program to the interface you already have

The first DIP example changes one type name, in three places. Before:

```
public static void main(String[] args) {
    ArrayList<Course> courses = new ArrayList<>();
    Student student = new Student(courses);
    double totalGradePoints = student.calculateTotalGradePoints();
}

class Student {
    private ArrayList<Course> courses;
    public Student(ArrayList<Course> courses) { this.courses = courses; }

    public double calculateTotalGradePoints() {
        return totalGradePoints;
    }
}
```

After: every `ArrayList` in a *declared type* position becomes `List`, while `new ArrayList<>()` is left alone. The constructor call is the one place a concrete class has to be named.

```
public static void main(String[] args) {
    List<Course> courses = new ArrayList<>();
    Student student = new Student(courses);
    double totalGradePoints = student.calculateTotalGradePoints();
}

class Student {
    private List<Course> courses;
    public Student(List<Course> courses) { this.courses = courses; }

    public double calculateTotalGradePoints() {
        return totalGradePoints;
    }
}
```

See java-collections-framework[15] for the `List`/`ArrayList` split this relies on, and java-abstraction[16].

[15] courses/csse2002/java-collections-framework.pdf
[16] courses/csse2002/java-abstraction.pdf

Dependency Injection

> Dependency Injection (DI) is a design pattern that implements the Dependency Inversion Principle (DIP), by injecting dependencies into a class instead of creating them internally, promoting loose coupling.

In the `Student` code above, the `Student` object receives its dependency through the constructor. The dependency is created in `main` and handed in — `Student` never calls `new` on it.

The full example runs the same move on a service that creates its own collaborator:

```
class AlertService {
    public void sendAlert(String msg) {
        EmailSender sender = new EmailSender();
        sender.send(msg);
    }
}
```

- `AlertService` is tightly coupled to `EmailSender`.
- `AlertService` can only send emails.
- `AlertService` creates its own dependency.

Introducing the abstraction and injecting it fixes all three:

```
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` no longer depends directly on `EmailSender`.
- It depends on the interface `MessageSender`.
- The actual sender is injected from outside.

Stability, exposure, and the "proper" form of DIP (creating an abstraction where none exists yet) are worked through in java-solid-principles[17] and drilled in week11-tutorial-dependency-inversion[18].

Summary and key takeaways

The takeaways for the whole block, not only this lecture:

- Test Driven Development — java-testing[19]
- Refactoring — java-refactoring[20]
- Coupling and Cohesion — java-cohesion-and-coupling[21]
- SOLID Principles — java-solid-principles[22]
  - **S**ingle Responsibility
  - **O**pen-closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion

Further reading

- *Design Principles and Design Patterns* by Robert C Martin
- https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/
- https://www.baeldung.com/solid-principles
- https://youtu.be/kF7rQmSRlq0

[17] courses/csse2002/java-solid-principles.pdf
[18] courses/csse2002/week11-tutorial-dependency-inversion.pdf
[19] courses/csse2002/java-testing.pdf
[20] courses/csse2002/java-refactoring.pdf
[21] courses/csse2002/java-cohesion-and-coupling.pdf
[22] courses/csse2002/java-solid-principles.pdf

Applied class

No separate applied-class topic on SOLID this week — see week7-tutorial-black-box-and-glass-box-testing[23] for this week's applied class (black-box/glass-box testing, a continuation of java-testing[24] from Week 5).

Black-box and Glass-box Testing

Applied class for 2026-04-16-solid-principles-ii[25] (Week 7). Applies the black-box/white-box testing concepts from java-testing[26] (Week 5, 2026-03-26-testing[27]) to a worked example. Most 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.

1. Black box testing

The following class tracks how many passengers are on a bus. Only the specification is given — no bodies — because black-box tests are written against the specification[28], not the implementation.

```
public class Bus {
    /**
     * Construct a new instance of the bus with a specified capacity.
     *
     * @param capacity The maximum passengers at any one time.
     */
    public Bus(int capacity) {...}

    /**
     * Calculate how many passengers are currently on the bus.
     *
     * @returns The number of passengers on the bus.
     */
    public int getCurrent() {...}

    /**
     * Calculate the average number of passengers on the bus between stops.
```

[23] courses/csse2002/week7-tutorial-black-box-and-glass-box-testing.pdf
[24] courses/csse2002/java-testing.pdf
[25] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[26] courses/csse2002/java-testing.pdf
[27] courses/csse2002/2026-03-26-testing.pdf
[28] courses/csse2002/java-specification.pdf

```
     *
     * @returns The average passengers on the bus.
     */
    public int getAverageCount() {...}

    /**
     * Exchange passengers at a bus stop.
     * Passengers get off before new passengers get on.
     *
     * @param on The number of passengers boarding the bus.
     * @param off The number of passengers disembarking the bus.
     */
    public void stop(int on, int off) {...}
}
```

The class above spells the tag `@returns`, not the standard `@return`. The course is inconsistent about specification-tag spellings (see csse2002-diagram-notation[29]).

Task 0 — write a smoke test

> **Working**
>
> There is a huge range of smoke tests you could come up with. Your smoke test should utilize the `Bus` class under normal/expected operation:
>
> ```
> Bus bus = new Bus(10);
> bus.stop(5, 0);
> bus.stop(3, 1);
> assertEquals(7, bus.getCurrent());
> assertEquals(6, bus.getAverageCount());
> ```
>
> The Javadoc says passengers get off **before** new passengers get on, so `stop(3, 1)` on a bus holding 5 goes $5 - 1 = 4$, then $4 + 3 = 7$; and the average between the two stops is $(5 + 7)/2 = 6$.
> See java-junit[30] for `assertEquals` and week8-lab-junit-testing[31] for writing these as real tests.

[29] courses/csse2002/csse2002-diagram-notation.pdf

[31] courses/csse2002/java-junit.pdf

[31] courses/csse2002/week8-lab-junit-testing.pdf

Task 1 — develop boundary scenarios (with justification)

Develop some boundary scenarios/inputs to use for testing. For each scenario you should justify why it is a good boundary case. You are **not** expected to know the expected output for each scenario.

> **i** Working
>
> There are many different situations to try; the following are just some examples.

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	It is not clear how a bus with a negative capacity should behave; we should aim to clarify its functionality and test it.
Ask for the average number of passengers after no stops	An average often involves division by the number of times — we want to ensure that asking for the average after zero stops doesn't create an exception.
Negative values for `on` and `off`	Behaviour for negative values is unclear — will a negative number of `on` passengers result in passengers getting off?
More passengers get off at the stop than are currently on the bus	Results in undefined behaviour.
More passengers on the bus than capacity	Behaviour is not specified, so we wish to see how `Bus` behaves.

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

Many of the boundary scenarios/inputs above are a result of unspecified or unclear behaviour of the bus class. How can we fix `Bus` to make all behaviour specified and clear?

> **i** Working
>
> If we specify that `getCurrent() <= capacity` is an **invariant** of the `Bus` class (see java-specification[32]), then we must protect the invariant throughout by either throwing exceptions or adding preconditions:
>
> 1. Add a `throws IllegalArgumentException` to the constructor of `Bus` when

`capacity < 0`.
2. Add a precondition to `stop` that both `on` and `off` are greater than or equal to zero.
3. Add a `throws IllegalStateException` to the `stop` method if `getCurrent() + on - off` is less than zero or greater than capacity.

The two exception types do different jobs (see java-exceptions[33]): the argument itself is nonsense in case 1, whereas in case 3 the arguments are individually fine but would drive the object into a state the invariant forbids.

2. Glass box testing

Glass-box testing helps us cover edge cases of program control flow. This is possible as we are allowed to use the internal structure (control-flow) of a method to construct our test cases.

There are different levels of *code coverage*; in this course we focus on three:

1. **Statement coverage**
2. **Branch coverage**
3. **Path coverage**

Consider what inputs would achieve each level of code coverage for `countOccurrences`:

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

Working

- **Statement coverage** — every statement must run at least once, and the only one that isn't run unconditionally is `count++`. So the array must contain the target at least once: `[2]` with target `2`.
- **Branch coverage** — the `if` must additionally be false once, and the `for` must additionally terminate (it always does). `[2, 3]` with target `2` is enough on its own;

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

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

so is `[2, 3, 2]`.

- **Path coverage** — approximated as 0, 1 and 2 iterations crossed with the `if` outcome each time, exactly the seven-row shape used for `ascending`. Because there is no early `return`, all seven are reachable.

3. Ascending

```
/**
 * Returns true if and only if numbers is an array of ascending
 * integers.
 *
 * @require numbers != null
 * @ensure \result is true iff for all indices j such that
 * 0 <= j < numbers.length - 1, numbers[j] <= numbers[j + 1]
 */
public boolean ascending(int[] numbers) {
    boolean result = true;
    for (int i = 0; i < numbers.length - 1; i++) {
        // invariant: for all j, 0 <= j < i, numbers[j] <= numbers[j + 1]
        if (numbers[i] > numbers[i + 1]) {
            result = false;
        }
    }
    return result;
}
```

The `// invariant:` line is the loop invariant — see java-loop-invariants[34]. It plays no part in the coverage answers below, which turn on the loop's shape: the body has no early exit, so every combination of iteration count and `if` outcome is reachable.

The three coverage levels are questions about the method's **control-flow graph**, so it helps to have it in front of you:

Task 3 — statement coverage

Provide a minimal set of inputs that achieve *statement* coverage.

[34] courses/csse2002/java-loop-invariants.pdf

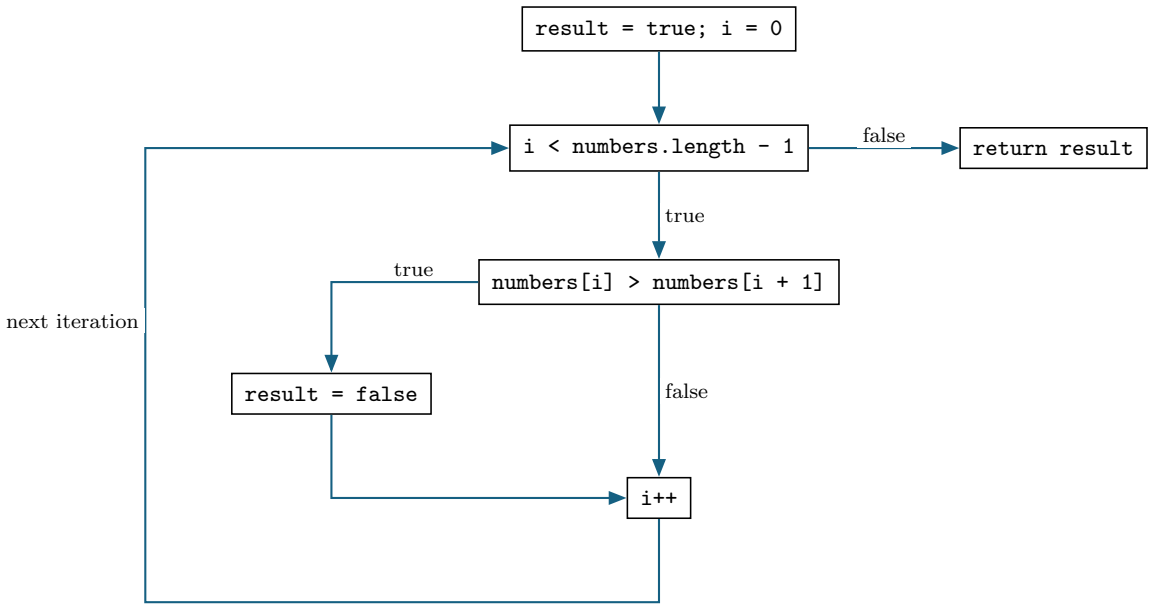


Figure 6: Control-flow graph of `ascending`. Every statement lies on some path; the two labelled edges out of `numbers[i] > numbers[i + 1]` are the only branch in the loop body, and the back edge is what makes exhaustive path coverage infeasible and forces the 0/1/2-iteration approximation. Statements and decisions are both drawn as plain rectangles; the teal arrows are the flow of control.

Working

`[2, 1]`
One iteration, with the `if` true, is enough to touch `result = false` — the only statement not on the unconditional path.

Task 4 — branch coverage

Provide a minimal set of inputs that achieve *branch* coverage.

Working

`[2, 1, 2]` **or** the pair `[2, 1]` `[1, 2]`
Either way the `if` has to come out true once and false once.

Task 5 — path coverage

Provide a minimal set of inputs that achieve *path* coverage.

Working

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, if false, false
`[3, 4, 2]`	2 times through loop, if false, true
`[4, 3, 5]`	2 times through loop, if true, false
`[5, 4, 3]`	2 times through loop, if true, true

Writing the justification for each row helps ensure no cases are missed.
Both `[]` and `[3]` give zero iterations, because the loop bound is `numbers.length - 1`.
The 0/1/2-iteration approximation is the one from java-testing[35] — exhaustive path coverage of a loop is infeasible.

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

4. Maximum — bonus

Returns the maximum number within an array of integers, if one exists, else returns `Integer.MIN_VALUE`.

```
/**
 * Returns the maximum number within an array of integer,
 * if one exists, else returns Integer.MIN_VALUE.
 *
 * @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;
}
```

Task 6 — statement coverage

> **Working**
>
> `[1]`

Task 7 — branch coverage

> **Working**
>
> `[2, 1]`
> The first element beats `Integer.MIN_VALUE` (if true), the second does not beat `2` (if false).

Task 8 — path coverage

i Working

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, if false, false
`[Integer.MIN_VALUE, 1]`	2 times through loop, if false, true
`[2, 1]`	2 times through loop, if true, false
`[1, 2]`	2 times through loop, if true, true

`Integer.MIN_VALUE` is doing real work here: it is the only value for which `number > currentMaximum` can be false on the *first* iteration, because the accumulator starts there and the comparison is strict.

5. Index Of — bonus

```
/**
 * Returns the index of the given number within the array of numbers,
 * or -1 if number is not in numbers.
 *
 * @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++) {
        // invariant: \forall int j; 0 <= j < i ==> numbers[j] != number
        if (numbers[i] == number) {
            return i;
        }
    }
    return -1;
}
```

This one differs from the other two in exactly one structural way: the `if` body **returns**, so the true branch leaves the method instead of continuing round the loop.

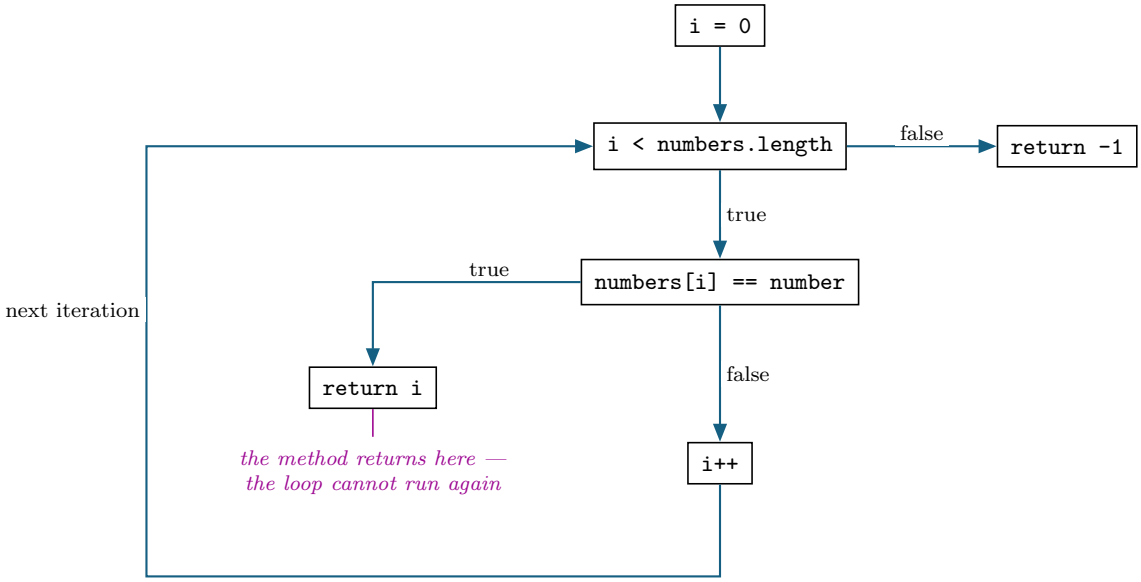


Figure 7: Control-flow graph of `indexOf`. Unlike `ascending`, the true branch of the `if` is an exit, not a step — so no path can take the `if` true and then iterate again, which is why two of the seven path-coverage rows in Task 11 read `Cannot be tested`. The exit is labelled in magenta.

Task 9 — statement coverage

Working

`[] 1` and `[1] 1`
Two calls are needed: `return i` and `return -1` are both statements, and no single input reaches both.

Task 10 — branch coverage

Working

`[] 1` and `[2, 1] 1`
`[2, 1]` with target `1` takes the `if` false then true; `[]` takes the loop condition false immediately.

Task 11 — path coverage

Working

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

The last two rows are unreachable for the reason drawn in Figure 7: taking the `if` true on the first iteration returns immediately, so there is no second iteration to have an outcome at all.

Reference material

CSSE2002 Diagram & Notation Conventions

The house style for every figure drawn in CSSE2002 — inferred from the lecture decks themselves, because **this course ships no written notation guide**.

How this note was built

The anchor manifest for csse2002 lists **12 lecture decks, 656 pages, and an empty `guides` array**. There is no style guide, no drawing standard, no "how to draw a class diagram" handout. Every convention below is *inferred from the drawings*, not quoted from a rule. Where a convention rests on a single figure, the entry says so.

Pages actually read

I read **210 of the 656 pages** as rendered images, sampled to go wide across the semester and to land on pages that carry drawings rather than bullets:

Deck	Pages read	of
L01a Course Overview	18–27	62
L01b Java Basics, Collections & Strings	31–42	77
L02 Object Oriented Programming I	1–20, 31–45	54
L03 OOP II & Specification	14–31	62
L04 Exceptions & Specification	6–36	49
L05 Testing	3–14	42
L06 Refactoring, Cohesion & SOLID I	15–32	52
L07 SOLID Principles II	14–30	39
L08 Java I/O	3–16	50
L09 Generics in Java	19–33	46
L11 Lambdas, Streams & Events	20–32	34
L13 Course Summary & Exam Prep	26–40	89

I did **not** read: L01a 1–17 & 28–62, L01b 1–30 & 43–77, L02 21–30 & 46–54, L03 1–13 & 32–62, L04 1–5 & 37–49, L05 15–42, L06 1–14 & 33–52, L07 1–13 & 31–39, L08 1–2 & 17–50, L09 1–18 & 34–46, L11 1–19 & 33–34, L13 1–25 & 41–89. A few citations below (textbook mentions, "Further Reading" slides) come from **extracted text on pages I did not view as images**; those are marked *(text only)*. Font names come from `pdffonts`; hex colours were sampled from rendered pixels, not from any stated palette.

Citations use the form **L07 p29** = the deck for lecture 07, PDF page 29.

1. The page frame every figure sits in

Template A — the course template. White ground. Slide title in UQ purple **#4E247F**, set in **Tw Cen MT** (a geometric sans), left-aligned at roughly x = 7% and y = 15%. An optional second title line sits directly beneath in smaller magenta **#9D0792 / #A02B93** Tw Cen MT. Body prose is **Bell MT** (a serif) in near-black **#171717**, round bullets, up to four indent levels with decreasing size. UQ crest bottom-left; solid purple **CSSE 2002** badge bottom-right; page number immediately left of the badge in small grey serif.

Cited: L02 p2, p14; L04 p12; L06 p20; L07 p14; L08 p5; L09 p33; L13 p39. **Holds across 11 of the 12 decks, weeks 1–13.**

Template B — the guest deck. L11 alone uses a completely different template: **Raleway Bold** black titles, **Source Sans Pro** black body, hyphen bullets, teal underlined links, occasional full-bleed purple block, and **no crest, no badge, no page number**. *Cited:* L11 p20, p23, p24, p28, p31. Seen only in L11 — see Inconsistencies.

Section dividers. Purple Tw Cen MT heading alone, placed low-left or low-centre, everything else blank. *Cited:* L02 p34, p43, p44; L06 p16, p19, p26; L09 p20, p23, p32; L13 p31, p32, p38. **Throughout.**

Deck title slides. Centred stack: deck title in purple Tw Cen MT → "Programming in the Large (CSSE2002)" in purple Bell MT → "Dr. Thilina Halloluwa" in black Bell MT → "Week NN – Part NN" in black Bell MT. *Cited:* L02 p1; L04 p19; L06 p15. **Consistent wherever a title slide appears.**

2. Typeface roles

Role	Face	Evidence
Slide titles, section headings	Tw Cen MT, purple	everywhere in Template A
Prose, captions, labels on the lecturer's own diagrams	Bell MT (serif)	L03 p27; L06 p21; L07 p16
Java source; Java identifiers inline in prose; text inside class/dependency boxes	**Inconsolata** (hand-set) or **Courier New** (pasted)	L02 p32, p36; L07 p27; L08 p6
Borrowed / stock figures	Arial or similar sans	L02 p15; L06 p17; L08 p3; L09 p27

Inline Java identifiers in serif prose are always switched to monospace: `byte`, `char`, `BufferedInputStream`, `java.io.InputStream`, `instanceof`, `? extends T`. *Cited:* L03 p29; L08 p15; L09 p24, p30. **Very consistent.**

3. What colour means

Colour	Sampled hex	Meaning	Evidence	Consistency
UQ purple	`#4E247F`	chrome, slide titles; nothing semantic inside a figure	everywhere	throughout
Teal	`#156082`	**every relationship line, connector and pointer the lecturer draws**	L03 p27; L04 p7; L06 p18, p20, p21; L08 p7, p11, p14; L09 p25, p29; L13 p27	strongest colour convention in the course — weeks 3–13
Magenta	`#9D0782` – `#A02B93`	subtitle line; a term being defined; a verdict on an example	L02 p35; L04 p27, p30; L05 p5; L08 p5, p8; L09 p21, p27, p30	very consistent
Pure red	`#FF0000`	a hard rule to memorise; the outline of a code-highlight box; deadlines	L01a p24; L02 p38, p40; L03 p28; L04 p14; L07 p15, p20; L08 p16	consistent for "stop and look"
Green	`#3B7D23` (Javadoc), bright green (chart)	Javadoc/comment text in code; passing tests in the one chart	L04 p27; L05 p5	chart usage is a single instance
Near-black	`#171717`	prose, box outlines, arrows in *borrowed* figures	L05 p6, p8; L07 p26	throughout

Nothing anywhere in any deck states what a colour means. Teal-for-relations and magenta-for-defined-terms are inferred from repetition alone.

Cube outline colours in the coupling/cohesion figures (purple `#7030A0`, red `#FF0000`, green `#196B24`, cyan `#00B0F0`, orange) carry **module identity only** — the same colour means "same module", not any property of it. *Cited:* L06 p18, p20, p27; L07 p24.

4. Class-hierarchy diagrams — the dominant style

Style H1 (dominant). Node = rectangle, thin border, white fill, class name centred in **Bell MT serif** (not monospace). Superclass sits **above**. Connector is a **teal orthogonal polyline**: one vertical stub rising into the parent's bottom edge carrying a **single solid teal arrowhead**, a horizontal teal bus, and vertical drop-legs to each child. **One arrowhead per parent, not per child.** No edge labels, no multiplicities, no stereotypes.

Cited: **L04 p7** (`Throwable → Error, Exception → RuntimeException, IOException`), **L08 p7** (`OutputStream` tree), **L08 p11** (`InputStream` tree). Three figures, two decks, weeks 4 and 8. This is the closest thing the course has to a settled hierarchy notation — but see Inconsistencies for two rival styles.

5. UML-style class boxes (lecture 02 only)

Three compartments separated by thin horizontal rules: **name / attributes / operations**. Thin black border, white fill, **monospace** throughout. Name centred and bold-ish; attribute and operation lines left-aligned.

- Visibility prefix: `-` private, `+` public. No `#` or `~` appears anywhere.
- Attribute line: `- name : String`, `-hourlyRate: double`.
- Operation line: `+getName():String`, `+logHours(hours:int) : void`. Parameters written `name:Type`, return type after the closing paren's colon.
- **Constructors live in the operations compartment**, written `+Employee (name:String, address:String)` with no return type.
- Generalisation: **solid black line, unfilled triangle arrowhead at the superclass**, orthogonal routing, superclass above.

Cited: L02 p12, p13 (two-compartment variant), **L02 p36** (the full hierarchy — the canonical figure), L02 p37, p38, p40.

This notation appears in lecture 02 and nowhere else I read. L09 p21 shows a small corner decoration in a similar spirit, but it is a borrowed graphic in a different palette (pale blue-grey fill, sans labels, `+ name : string` with a lowercase type). Do not treat L09 p21 as course style.

6. Dependency / interface diagrams — the only figures with a legend

These are the only diagrams in the course that **draw their own key on the slide**, so this is the most reliable convention in the note.

- **Node** = plain rectangle, thin black border, white fill, monospace label that **includes the Java keyword**: `class A`, `class B`, `class Z`, `interface AI`, `interface BSpec`.
- **Members** are listed inside the same box, one per line, as `+ a()` — with a space after the `+`, and **no compartment rule** separating them from the name.
- **Legend, drawn at the right of the slide:**
  - solid line + solid arrowhead → **"Uses"**
  - dashed line + open arrowhead → **"Implements"**
- **Direction:** an arrow always points **from the depender to the thing depended upon**. `class A` `interface AI` = A uses AI. `class Z` — the dashed implements arrow runs **from the implementing class to the interface** (L07 p27, p29, p30).
- **Layout:** clients left, abstractions centre, concrete classes right (L07 p26, p27); or dependents at the bottom with the abstraction above (L07 p29, p30). Flow is left→right for "uses" and bottom→top for "implements".
- An arrow may terminate on an individual **member row** inside a box, not only on the box edge (L07 p26).

Cited: L07 p26 (legend shows "Uses" only), **L07 p27, p29, p30** (full legend). Four figures, one deck, week 7.

Note the divergence from standard UML: UML draws realisation as a *dashed line with a hollow triangle* and dependency as a *dashed open arrow*. This course draws implements as a dashed **open arrowhead** and uses as a **solid** line with a solid head. Follow the course, not UML — but be aware you are diverging.

7. The pre/postcondition "funnel" figure

A distinctive, course-specific idiom for reasoning about the Liskov Substitution Principle:

- A method is a small **square box, thin black outline, monospace name** (`A.f`, `B.f`).
- **Above** it, a cone that narrows downward into the box. Its wide upper ellipse is the set of inputs the method accepts — the **precondition**.
- **Below** it, a cone that widens downward out of the box. Its wide lower ellipse is the set of outcomes it may produce — the **postcondition**.
- **Width = weakness.** A *wider* top mouth = a *weaker* (more permissive) precondition. A *wider* bottom mouth = a *weaker* (less committal) postcondition. So a legal subclass override has a top mouth **at least as wide** and a bottom mouth **no wider** than its parent's.

- Labels `preconditions` and `postconditions` in **lowercase serif** at the far left of the slide, vertically aligned with the two mouths.
- Entirely **black line art, unfilled, no colour**.
- To compare parent and child, the child's cones are **overlaid on the parent's** in the same frame in thinner/greyer stroke (L07 p17).

Cited: **L07 p16** (A.f vs B.f side by side), **L07 p17** (overlaid), L07 p18 (reused as a thumbnail). Three pages, one deck.

8. Coupling and cohesion figures

- **A module is an isometric wireframe cube** — outline only, no fill, hidden edges dashed. One colour per module, identity only.
- **Coupling is a teal double-headed arrow** between cubes. High coupling = many crossing arrows; low coupling = each cube attached by one short teal double arrow to a **horizontal grey bar** standing for a shared interface.
- **Cohesion** uses the same cubes flattened. Low cohesion = colours scattered; high cohesion = same-coloured cubes grouped inside a rectangle **outlined in that same colour**.
- **Comparison layout: the bad case on the LEFT, the good case on the RIGHT**, each labelled with one serif word beneath (`High/Low`, `Low/High`).

Cited: L06 p18 (tangled), **L06 p20** (High vs Low coupling), **L06 p27** (Low vs High cohesion); reused verbatim in **L07 p24**. Two decks, weeks 6–7.

9. The coupling-levels ladder

Seven identical rectangles — thin black border, white fill, serif label — stacked vertically with even gaps. Ordered **most-coupled at the top** (`Content`) down to **least at the bottom** (`None`). A single long thin **teal arrow points downward** to the right of the stack. Its endpoints are captioned in magenta serif with a parenthetical gloss: "Tightly Coupled (Needy)" at the top, "Loosely Coupled (Independent)" at the bottom.

Cited: **L06 p21**; reused verbatim **L07 p25**. Two decks.

10. Annotating code

- **Highlight a token:** a thin **red `#FF0000` rectangle** drawn around it. *Cited:* L02 p38, p40; L03 p28; L04 p14; L08 p16. Dominant across four decks.
- **Point at a token:** a thin **teal straight or diagonal line with a small solid arrowhead**, running between the token and a **serif caption placed outside the code block, unboxed**. *Cited:* L02 p16; L03 p24, p28; L08 p14; L09 p25, p29; L13 p27. Weeks 2–13 — very consistent.

- **Show a replacement: strike through** the token being removed and put the replacement beside it in red. *Cited:* L02 p4; L07 p20; L13 p27.
- **Mark an error inline:** a red squiggly/dashed underline, IntelliJ-style. *Cited:* L09 p22.
- **Aside / rule-to-remember:** a short sentence centred beneath the content in serif — sometimes inside a thin rectangle, sometimes bare. *Cited (boxed black):* L02 p31; L07 p24; L08 p5; L09 p26. *Cited (boxed teal):* L03 p21. *Cited (unboxed):* L03 p23; L06 p31; L08 p12, p15; L07 p14, p25. Inconsistent — see below.
- **Yellow highlighter** behind a heading inside a worked-example box. *Cited:* L07 p19 (single instance).
- **Verdict on an example**, placed in the right margin in magenta serif: "Bad Example", "Better / Still Bad ", "Does not satisfy specification!", "Satisfies specification!". *Cited:* L04 p26, p27, p28, p29, p30, p31. Consistent within L04.

11. Code blocks

Java is shown as **light-theme IntelliJ IDEA output**, confirmed by the sampled hex values: keywords `#0033B3` blue, fields/constants `#871094` magenta, strings and Javadoc `#3B7D23` green, comments grey italic, annotations (`@Override`, `@Test`, `@Before`) olive/mustard, numeric literals blue. *Cited:* L02 p4, p16; L04 p22; L06 p24; L09 p22.

Some slides show real IntelliJ inlay hints (`3 usages`, `accountNumber:`) and even an editor error popup, i.e. they are screenshots rather than set text. *Cited:* L02 p5, p6, p9, p20.

Container: either bare on white, or inside a **thin black rectangle**. Two files at once are shown as **two overlapping black rectangles**, z-order indicating which is in front. *Cited:* L02 p39, p41, p42; L03 p16; L04 p13, p18; L07 p18.

A **trace** puts line numbers in the left margin in monospace and expected results in a right-aligned column of `//` comments. *Cited:* L01b p40, p41.

12. Specification / Javadoc notation

The course's specification "algebra" lives inside Javadoc comments, not in a separate formalism.

- Javadoc opens `/**` and closes `*/`, sits immediately above the thing documented, and tags begin with `@`. *Cited:* L04 p21.
- **The tag table** (L04 p23) is set as a two-column list with no rules: tag name in monospace, its argument placeholder in *italic serif*, gloss in upright serif.

Tag	Argument	Meaning (as printed)
`@param`	*varname …*	Describe a parameter for a method
`@return`	…	Describe the return value
`@throws`	*ExceptionType …*	Describe when a particular exception is thrown
`@requires`	*Precondition*	Specify the assumptions for a method to execute properly
`@ensures`	*Postcondition*	Specify the effects of executing the method
`@author`	*authorname*	Specify the author of the class
`@copyright`	…	Provide details on who owns copyright

- `\result` denotes the returned value inside a postcondition. *Cited:* L07 p18, p19, p20.
- Conditions are written in **Java boolean syntax**, not mathematical notation: `row != null && col != null && 0 < val && val < 100`. No , , appears anywhere I read. *Cited:* L07 p18; L13 p36.
- `{param}` braces denote a parameter's value inside prose in a tag: `@return {first} followed by {second}`. *Cited:* L13 p36 (single instance).

13. Arrays and stacks

Array. A horizontal row of equal-width square cells sharing thin black borders; values centred; **no index row**. An empty cell is a hole. A **grey ghosted digit** is a value being shifted out. `?` is an unspecified value. A **purple curved arrow** drawn above the row = one shift-left step. *Cited:* L01b p31–p37 (a seven-frame build). A second, smaller variant puts the array's name in monospace immediately to the left of the row and sets the values in serif: L04 p27.

Stack. *Cited:* L01b p39 — an **open-topped U-shaped container** in bright purple `#6B088F` (no top edge), entries as purple-filled rounded rectangles with white bold numerals, **growing upward** so the top of stack is at the top. A grey curved arrow **into** the container = push; a grey curved arrow **out and away** = pop. Each frame is captioned in monospace with the operation (`empty()`, `push(1)`, `pop()`), and frames read **left→right as time**.

14. Black-box / glass-box testing icon

A **grey-filled isometric cube, unlabelled**, with a **black horizontal arrow entering on the left** labelled `input` in serif above the line, and a matching arrow **leaving on the right** labelled `output`. Flow is strictly left→right. The glass-box variant is the same cube rendered

translucent with the source code visible inside it. *Cited:* **L05 p6** (both cubes together), L05 p8 (black-box cube reused as a corner marker).

15. Attribution and textbook lineage

No figure anywhere I read carries a "Figure from ..." caption, and no single textbook governs the drawing style. The course cites books for *content*, never for *notation*.

Placement convention: a source line sits at the **top of the slide, right-aligned, small serif, no box and no rule**.

- "Effective Java (2017) - Item 75: Include failure-capture information in detail messages" — **L04 p8**
- "Java: The Complete Reference, Eleventh Edition- Herbert Schild" — **L08 p4**
- "*Effective Java*, Item 31: parameterized types are invariant; `List<Type1>` is neither a subtype nor supertype of `List<Type2>`" — **L09 p28**
- "Effective Java (2017, Addison-Wesley) - Item 9" — **L08 p20** *(text only)*
- Bare URLs in the same position: L01b p38; L05 p5, p9; L08 p9, p10; L09 p26.

Borrowed figures carry an adjacent link instead: "Source : Javatpoint" in small serif at the figure's lower right (**L08 p3**); a bare hyperlink reading just "source" centred beneath the figure (**L06 p28**).

A book cover can stand in for a citation: L03 p15 shows the *Effective Java* **2nd edition** cover as the whole visual for the slide.

Books named in prose *(text only — pages not viewed as images)*: Martin Fowler, *Refactoring: Improving the Design of Existing Code* (2nd edition, 2019) and Robert Martin, *Clean Code* at **L06 p14**; Robert C. Martin, *Design Principles and Design Patterns* at **L06 p35 & p50, L07 p3, L07 p38**; *Effective Java* items at **L03 p18, p22** and **L04 p47**.

16. Naming and capitalisation

- Java identifiers follow standard Java: `PascalCase` types, `camelCase` methods and fields, `SCREAMING_SNAKE` constants, always monospace. **Consistent everywhere.**
- Running example names: `Person`, `Employee`/`ContractEmployee`/ `FullTimeEmployee`, `Animal`/`Dog`/`Cat`, `BankAccount`/`AccountHolder`, `Foo`, `Printer`, and for abstract structural examples `class A/B/C`, `class Z`, `interface AI/BI/CI`, `interface BSpec`.
- **Interfaces in structural diagrams are named either `<Letter>I`** (`AI`, `BI`, `CI` — L07 p27) **or `<Class>Spec`** (`BSpec` — L07 p29, p30). Both in the same deck.
- Method-signature shorthand in prose uses UML order — `getType():String`, `addListener(String type, Consumer<SimpleEvent> action): action` (L11 p28) — while diagram boxes use the same order (L02 p36). Consistent.

- A right arrow → in serif means "resolves to" / "dispatches to": `kapow(Object) → locked in`. *Cited:* L13 p26, p28.
- ASCII `->` between monospace type names means "widens to" in the casting chains: `byte -> short -> char -> int -> long -> float -> double`. *Cited:* L03 p26.

Inconsistencies

Where the course does the same thing more than one way. **A decision is needed on each of these before drawing a new figure — I am recording both, not choosing.**

1. **Three different ways to draw inheritance.**
   - *H1* — black-outlined box, **serif** class name, teal orthogonal connector, plain solid teal arrowhead into the parent, no edge label: **L08 p7, L08 p11**.
   - *H2* — UML: black-outlined **three-compartment** box, monospace name, black orthogonal connector, **unfilled triangle** arrowhead: **L02 p36, L02 p40**.
   - *H3* — black-outlined box, **serif** name, **teal diagonal** (not orthogonal) lines, plain arrowhead, and every edge **labelled "Is a" / "Is an"** in serif: **L03 p27**. All three put the superclass above. They disagree on arrowhead shape, on routing, and on whether edges are labelled. **Decision needed.**
2. **Hierarchy box border colour.** Purple `#4E247F` at **L04 p7** versus black `#171717` at **L08 p7 and L08 p11** — same figure type, four weeks apart.
3. **Class name typeface inside a box.** Serif at **L04 p7, L08 p7, L08 p11**; monospace at **L02 p36, L07 p27, L07 p29**.
4. **Where the class name goes.** Inside a dedicated name compartment at **L02 p36**; floating **above** an unlabelled two-compartment box as "Person Class" at **L02 p12 and L02 p13** — within one deck.
5. **Specification tag spelling — three variants.**
   - `@requires` / `@ensures` — **L04 p23** (the tag table itself), **L13 p36** (which additionally misspells it `@requiers`).
   - `@require` / `@ensure` (no `s`) — **L07 p18, p19, p20**, used consistently across that whole deck's worked example.
   - `@returns` instead of `@return` — **L05 p8**. **Decision needed.** The tag table (L04 p23) and the exam-prep deck (L13 p36) agree on the `-s` forms; the SOLID deck (L07) does not.

6. **Spacing inside UML member lines**, within one figure (**L02 p36**): `- name : String` (spaces both sides of the colon) in the `Employee` box, but `-hourlyRate: double` (no leading space, none before the colon) in `ContractEmployee`. Likewise `+getName():String` versus `+logHours(hours:int) : void` versus `+printDetails() : String`.

7. **Code-highlight rectangle colour.** Red at **L02 p38, L03 p28, L04 p14, L08 p16**; **purple** at **L02 p16 and L02 p17**; plain black at **L08 p14**.

8. **The "translates to / produces" block arrow.** Solid **teal** block arrow pointing **right** at **L02 p13** (UML box → code); hollow black-outlined block arrow pointing **down** at **L05 p10** (class → test class); solid **black** block arrow pointing **right** at **L06 p17** (monolith → modular). Same semantic role, three renderings.

9. **Aside / rule boxes.** Thin **black** rectangle at L02 p31, L07 p24, L08 p5, L09 p26; thin **teal** rectangle at **L03 p21**; **no box at all** at L03 p23, L06 p31, L07 p14, L08 p12, L08 p15 — all carrying the same kind of one-line rule-to-remember.

10. **Two stack renderings inside one deck. L01b p39**: purple open-topped container, purple filled blocks, grey push/pop arrows, monospace operation captions. **L01b p40 & p41**: white cells with black borders and a heavy bottom rule, quoted string values in bold sans, variable name in monospace *below* the stack, no container and no arrows.

11. **Two array renderings. L01b p31–p37**: large cells, no name label, sans/mono digits. **L04 p27**: small cells, serif digits, array name `ar` in monospace to the left of the row.

12. **Code-block borders.** Crisp thin black rectangle (L02 p39, L04 p13); **hand-drawn wobbly "sketched" rectangle** at **L05 p10 and L05 p11** only; or no border at all (L02 p32, L04 p22).

13. **Two whole slide templates.** Template A (Tw Cen MT / Bell MT / crest / badge) versus Template B in **L11** (Raleway / Source Sans Pro / no chrome). A figure drawn for L11's week will not match a figure drawn for any other week.

14. **Attribution placement.** Top-right of the slide (L04 p8, L08 p4, L09 p28) versus adjacent to the figure itself (L08 p3, L06 p28) versus a cover image standing in for the citation (L03 p15).

15. **Which edition of *Effective Java*.** 2nd edition cover shown at **L03 p15**; "(2017)" i.e. 3rd at **L04 p8** and **L08 p20**; "(3rd Edition)" spelled out at **L04 p47**; unlabelled at **L03 p18, p22** and **L09 p28**.

16. **Syntax-colour theme drifts between pasted screenshots.** `new` renders blue `#0033B3` at **L02 p16** but magenta at **L01b p42**; some blocks carry IntelliJ inlay hints (L02 p5, p6) and others do not.

17. **Sub-title-line capitalisation.** "Levels" at **L06 p21** but "levels" at **L06 p29**; "Byte Stream (Output)" (L08 p7) beside "Abstraction" (L08 p5). Slide titles likewise: "Black Box testing" (L05 p8) versus "Blackbox testing" (L13 p40); "Bounded generics" (L09 p21) versus "Bounded Generics" (L09 p20).
18. **The interface naming scheme.** `AI/BI/CI` at **L07 p27** but `BSpec` at **L07 p29 and p30** — two slides apart.
19. **Table styling.** The only formal data table I found (**L01a p18**, the semester schedule) uses a **navy** header band and pale-blue zebra rows — not the UQ purple used by everything else on the same slide deck.

Not established by the source

Things a new CSSE2002 diagram will plausibly need that the materials simply do not settle. **This is where the risk of inventing notation lives.**

- **Aggregation and composition.** No filled or hollow diamond appears in any page I read. L02 p9 has a code comment reading `// Association with AccountHolder`, but no diagram of that association is ever drawn. How to show "has-a" is unestablished.
- **Multiplicities.** No `1`, `0..1`, `1..*` or `*` ever appears on an edge.
- **Abstract classes and interfaces in a UML box.** No `«interface»` stereotype, no italicised abstract class name, no dashed realisation triangle. L07 writes the keyword into the label text instead (`interface BSpec`), but that deck's boxes have no compartments, so the two idioms cannot simply be merged.
- **Static members.** The usual UML underline is never shown.
- **Package or namespace boxes.** Never drawn, despite packages appearing in code (L02 p20).
- **Sequence diagrams, state machines, activity diagrams, flowcharts, ER diagrams.** None in any of the 210 pages I read.
- **Control-flow graphs.** Glass-box testing is taught (L05 p6) and path/branch coverage is implied, but no CFG, decision tree or coverage diagram is drawn. There is no notation for one.
- **Stack-and-heap memory diagrams.** The schedule (**L01a p18**) lists a Week 3 applied class called "Stack & Heap", but no lecture deck I sampled draws one. Boxes-and-pointers, reference vs value, and how to render `null` are all unestablished from the lecture materials.
- **Object (instance-level) diagrams.** Nothing like `p1:Person`. The nearest thing is a borrowed clip-art figure at **L02 p15** in a completely foreign style (yellow label chips, grey value cards, child cartoons) that should not be copied.

- **Generic types inside a diagram box.** `<T>`, `? extends T` and `? super T` appear only in code and in a borrowed `Number`-tree graphic (L09 p27, p31). How to render a parameterised type in a class box is unestablished.
- **Exact palette values.** No deck states a hex code. Every colour in this note was sampled from rendered pixels. Line weights, arrowhead sizes and box corner radii are likewise nowhere specified.
- **Dark mode / greyscale printing.** Every figure assumes a white ground and colour output. There is no stated fallback, and the coupling/cohesion cube figures (L06 p20, p27) become unreadable in greyscale because colour is the only thing distinguishing modules.
- **Figure numbering and captions.** Not one figure in the decks is numbered or captioned "Figure N." Captions, where they exist, are bare serif phrases beneath the art. There is no established way to cross-reference a figure.
- **Whether new hierarchy figures should be UML-correct.** This is the single biggest open question. The dominant house style (H1) uses a plain arrowhead where UML requires a hollow triangle; lecture 02 uses the triangle. Neither is declared canonical.
- **Font substitutes.** Tw Cen MT, Bell MT and Inconsolata are the embedded faces, but nothing says what to substitute when they are unavailable.

Java Abstraction

Introduced in 2026-03-05-object-oriented-programming-i[36] (Lecture, Week 2).

What is abstraction?

The ability to hide complex implementation details and expose only the essential behaviours. In Java, abstraction is often implemented using **interfaces** and **abstract classes** — both define methods that must be implemented by subclasses, providing a contract that specifies *what* operations can be performed on an object without specifying *how* those operations are implemented.

Interfaces

- An interface is a group of related methods **with empty bodies**.
- An interface defines a **contract** that implementing classes must fulfil.
- To access the interface methods, the interface must be **implemented** — and a class can implement multiple interfaces if required.

  Interfaces provide abstraction in Java by defining behaviour without specifying implementation.

[36] courses/csse2002/2026-03-05-object-oriented-programming-i.pdf

```java
public interface PaymentProcessor {
    void processPayment(double amount);
}
```

Any payment processor must have a ***`processPayment()`*** *method.* Two classes take the interface up, each with its own body for that one method:

```java
public class PayPalProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment: " + amount);
    }
}

public class StripeProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Stripe payment: " + amount);
    }
}
```

and the caller is written against the interface, never against either class:

```java
public class Main {
    public static void main(String[] args) {
        PaymentProcessor processor = new PayPalProcessor();
        processor.processPayment(100);
    }
}
```

The declared type of `processor` is `PaymentProcessor`, so `Main` can only call what the interface promises: the program only interacts with the interface. Swapping `new PayPalProcessor()` for `new StripeProcessor()` is therefore a one-line change: it changes what runs without touching anything else in `Main`.

> An interface in Java represents an abstraction. It defines a set of behaviours that classes must implement, but it does not define how those behaviours are implemented. The implementation details are hidden inside the classes that implement the interface.

Abstract classes

- An abstract class is a class that **cannot be instantiated**.

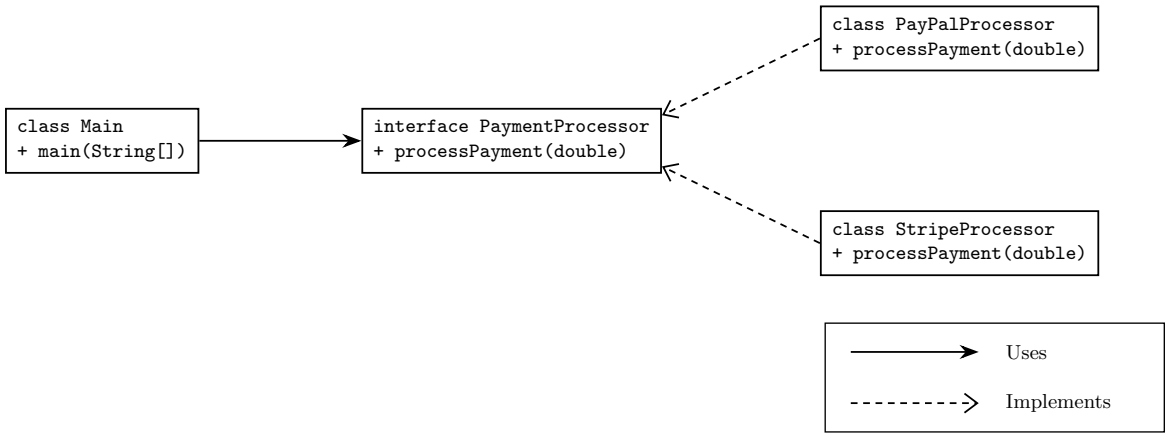


Figure 8: Main depends on the PaymentProcessor interface, not on either implementation - which is what makes swapping PayPal for Stripe a one-line change. Each rectangle is labelled with its Java keyword and lists its members as + lines. A solid arrow with a solid head means uses. A dashed arrow with an open head means implements and runs from the implementer to the interface. The client is on the left and the abstraction in the centre.

- It can contain **both** abstract methods and fully implemented methods.
- The `abstract` keyword is used to declare abstract classes and methods.
- Subclasses must implement all abstract methods.

One qualification to the first bullet: an instance of an abstract class can be created using an anonymous class. The two statements are consistent. The ban is on `new Payment()`, while `new Payment() { ... }` instantiates an anonymous subclass rather than `Payment` itself.

```
public abstract class Payment {
    public void validateTransaction() {
        System.out.println("Validating transaction...");
    }

    abstract void processPayment(double amount);
}
```

That class **defines common behaviour for every payment** (`validateTransaction`), and does **NOT** define how a payment is processed. A subclass supplies the missing half:

```
public class PayPalPayment extends Payment {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment: $" + amount);
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Payment payment = new PayPalPayment();

        payment.validateTransaction();
        payment.processPayment(150.0);
    }
}
```

The abstract class defines the common behaviour of all payments while leaving the specific logic to subclasses. The two calls in `main` do different things structurally: `validateTransaction()` runs the body inherited from `Payment`, while `processPayment()` runs the body `PayPalPayment` had to supply.

Interfaces vs abstract classes

A **homework / self-study** item in Week 2, taken up in 2026-03-12-object-oriented-programming-ii[37] (Lecture, Week 3).

- **Use an interface** to define a capability or contract, e.g. `Flyable`, `Runnable`, `Payable`. A class can implement multiple interfaces.
- **Use an abstract class** to define a shared base implementation (common behaviour), e.g. an `Animal` abstract class shared by `Dog` and `Cat`. A class can only extend one (super)class.

```
abstract class Animal {
    public void eat() {
        System.out.println("Animal is eating");
    }
    public abstract void makeSound();
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
```

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

```
    @Override
    public void makeSound() {
        System.out.println("Cat Meows");
    }
}
```

A class can combine both: extend at most one abstract/concrete class, while implementing as many interfaces as needed:

```
class Dog extends Animal implements Runnable, Swimmable
class Duck extends Animal implements Flyable, Swimmable
```

Both mechanisms lean on the same machinery as java-inheritance[38] — the `@Override` annotation and identical method signatures — and both pay off in Week 3 as java-polymorphism[39].

Java Cohesion and Coupling

Introduced in 2026-04-02-refactoring[40] (Lecture, Week 6, Part 2). Practised in week10-tutorial-coupling-and-cohesion[41].

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).

[38] courses/csse2002/java-inheritance.pdf
[39] courses/csse2002/java-polymorphism.pdf
[40] courses/csse2002/2026-04-02-refactoring.pdf
[41] courses/csse2002/week10-tutorial-coupling-and-cohesion.pdf

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.

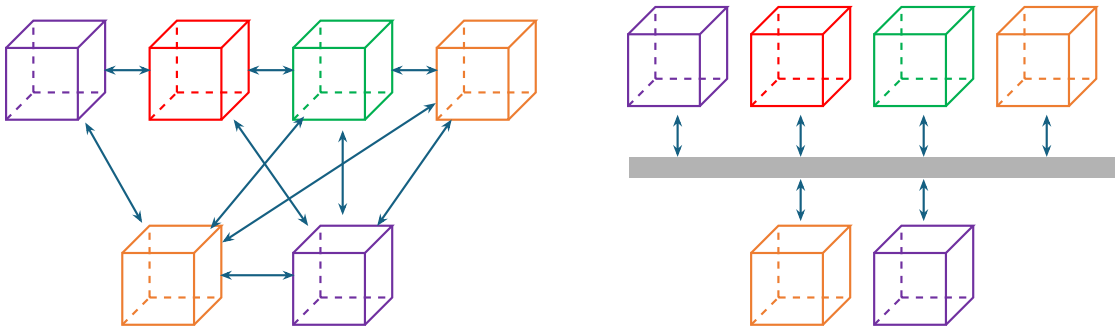


Figure 9: High versus low coupling. Left: every module talks directly to several others, so a change in any one can reach any other. Right: the same six modules each talk only to a shared interface — the grey bar — so nothing propagates further than that bar. A module is a wireframe cube, and its colour 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.

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).

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.

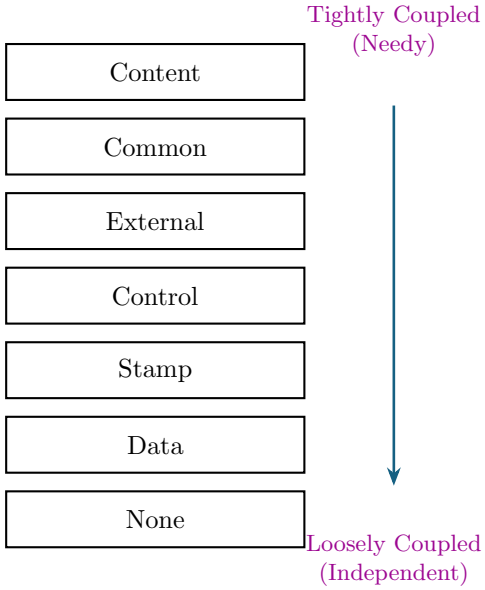


Figure 10: The coupling-levels ladder. Most tightly coupled at the top, least at the bottom, with the arrow running down from tightly coupled (needy) to loosely coupled (independent).

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
}
```

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

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.)

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).

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

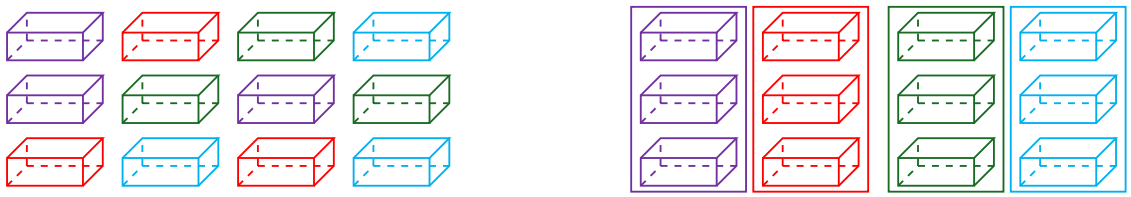


Figure 11: Low versus high cohesion. The same twelve elements in both panels: on the left the four kinds are scattered across the grid, on the right each kind is gathered into one module. Colour means kind only, and each high-cohesion group is enclosed by a rectangle in that group's own colour. This figure says nothing about coupling — it is only about what sits inside one module. The good case is again on the right.

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?

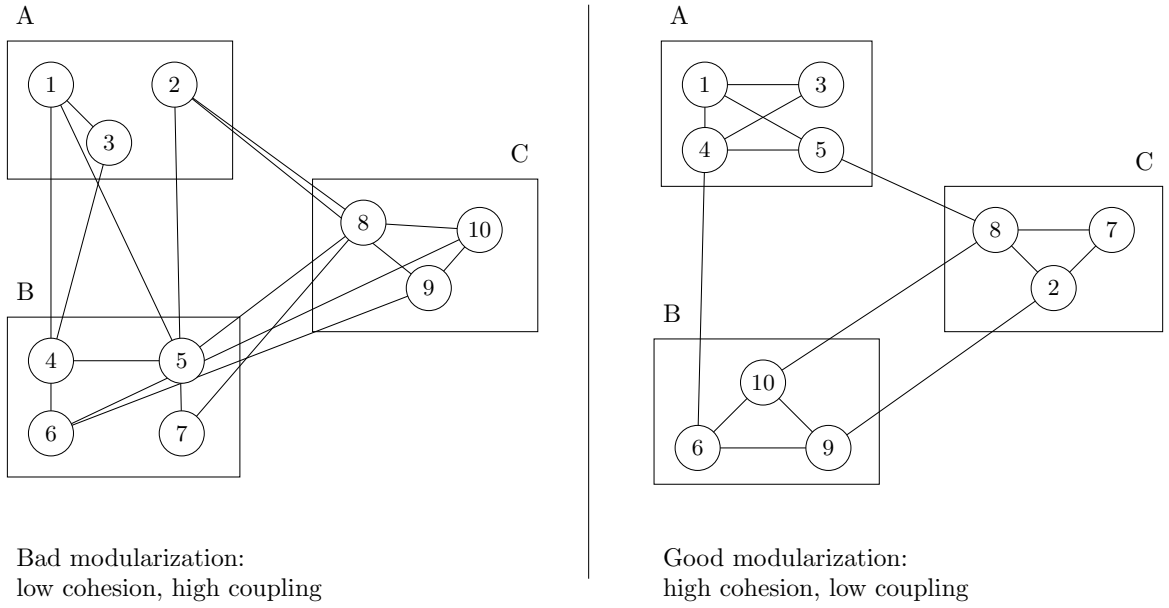
A `Car` class contains: Fuel, Steering, Speed, Route planner, Public holiday calculator.

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:



Bad modularization:
low cohesion, high coupling

Good modularization:
high cohesion, low coupling

Figure 12: 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.

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.

Java Collections Framework

Introduced in 2026-02-26-java-basics-part-02-collections-and-strings[43] (Lecture 1, Week 1) — see that lecture for the worked push/pop/add/remove traces. All of these live in `java.util.*`, and (unlike arrays) grow/shrink automatically.

Why collections instead of arrays?

An array is an ordered **fixed-length** mutable sequence of **homogeneous** items, and that costs you two things:

1. **Fixed size at creation.** You have to know up front how much space you need.
2. **They do not automatically close gaps.** Removing an element from the middle leaves a hole; closing it means shifting every later element down by hand, and the array is still the same length afterwards, so the trailing cell has to be given some filler value.

Collections solve both, at the cost of only being able to store **objects** (reference types) — see java-primitive-and-reference-types[44] for the primitive wrapper classes (`Integer`, `Double`, etc.) used to store primitives inside them.

Stack — LIFO

Method	Description
`empty()`	Is this stack empty?
`peek()`	Return the object at the top of the stack
`pop()`	Remove (and return) the object at the top of the stack
`push(obj)`	Put `obj` on the top of the stack

```
import java.util.Stack;

Stack<Type> stacks = new Stack<>();
```

`pop()` on an empty stack throws `EmptyStackException`.

[43] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf
[44] courses/csse2002/java-primitive-and-reference-types.pdf

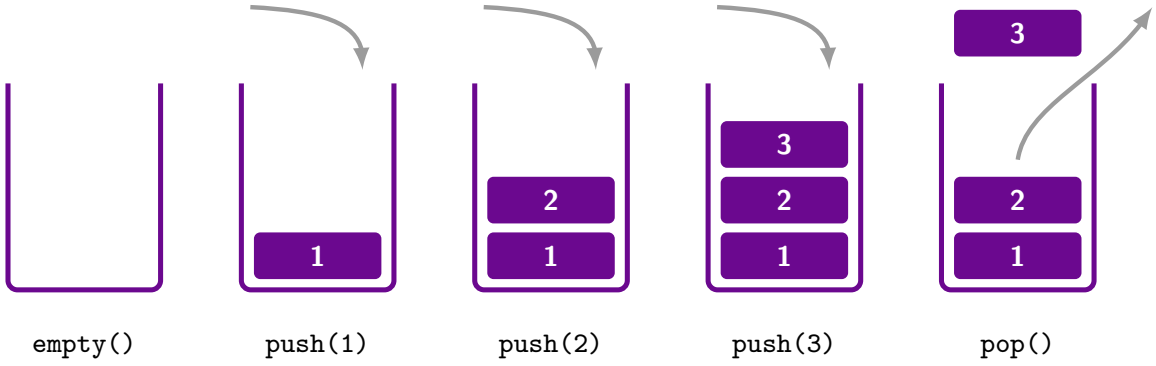


Figure 13: Last In, First Out: `push` always adds at the top and `pop` always removes from the top, so the last value in is the first value out. The stack is an open-topped container whose entries grow upward. An arrow *into* the container is a push and an arrow *out and away* is a pop. Each frame is labelled with its operation, and the frames read left to right as time. In the `pop()` frame the `3` is lifted clear of the container to show it gone.

List

An interface, not a particular implementation — you can declare a variable as `List`, but can't do `new List()`.

- **`ArrayList`** — better for random access (`get(i)`).
- **`LinkedList`** — better for operations that modify the middle of the list.

Holds items in sequential order (like an array), 0-indexed, with no fixed size limit; supports inserting/removing at any position and checking whether an item is present.

Method	Description
`add(e)`	Append `e` to the end
`add(index, e)`	Insert `e` **at** `index`, shifting everything from `index` up by one
`get(index)`	Return the element at `index`
`remove(index)`	Remove the element **at** `index`, shifting everything above it down by one

The shifting is the whole point — it is the gap-opening and gap-closing an array makes you write yourself:

Set

Stores unique items (no duplicates); don't assume any iteration order.

```
interface Set<E> {
    int size();
    boolean contains(E item);
    boolean add(E e);
    boolean remove(E item);
}
```


`add` returns `true` if the set changed and `false` if the item was already present — so

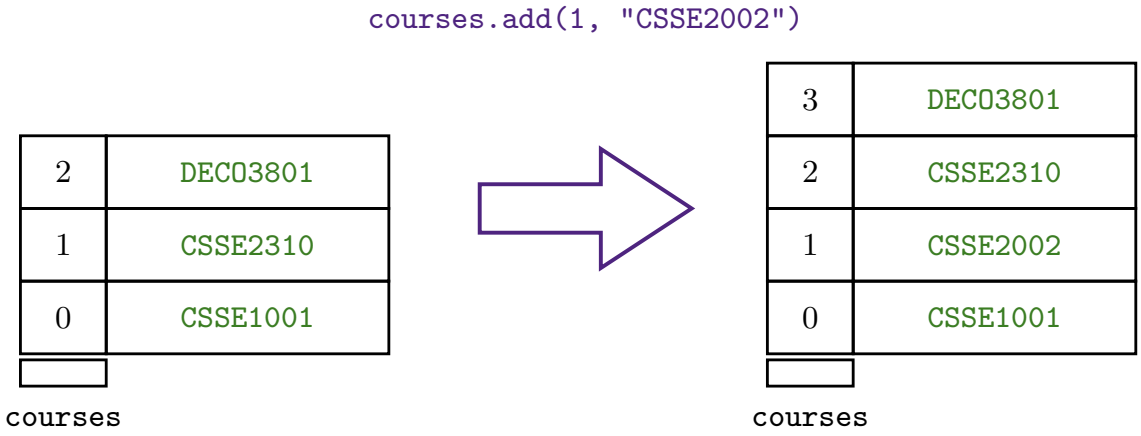


Figure 14: `add(index, e)` on a `List`: `e` lands *at* that index and every entry from there up renumbers, so the list gets longer without you writing a single shift. (`remove(index)` is the mirror image — see the removal step traced in 2026-02-26-java-basics-part-02-collections-and-strings[45].) The list has **index 0 at the bottom** and grows upward, with index cells on the left, values in green on the right, and the variable name in a small box beneath the stack of cells.

Map

Stores key → value pairs (like a Python dict) — specify a type for both the key and the value, e.g. `Map<Integer, String>`. The value type can itself be a collection: `Map<String, ArrayList<String>>`.

```
interface Map<K, V> {
    int size();
    boolean containsKey(K key);
    boolean containsValue(V value);
    V get(K key);
    V put(K key, V value);
    V remove(K key);
    Set<K> keySet();
}
```

`put` on a key that is already present **overwrites** its value, so the read-modify-write idiom is `m.put(k, m.get(k) + 1)`. `put` also *returns* the displaced value. That is standard `Map` behaviour, and it is the `V` return type in `V put(K, V)`.

- **`TreeMap<K,V>`** — `K` must implement `Comparable`.
- **`HashMap<K,V>`** — `K` must have sensible `hashCode()`/`equals()`.

The hashCode/equals contract

`HashSet`/`HashMap` rely on their elements/keys satisfying:

1. `x.equals(y)` $\iff$ `y.equals(x)` (symmetric).
2. `x.equals(y)` $\implies$ `x.hashCode() == y.hashCode()`.

`hashCode()` returns an integer generated by a hashing algorithm for the object — two objects that are `.equals()` must hash the same, or a `HashSet`/`HashMap` won't be able to find them correctly. For example `"Thilina".hashCode()` is `318621125`.

Java Exceptions

Introduced in 2026-03-19-exceptions[46] (Lecture, Week 4, Part 1). Traced in depth in week5-tutorial-exceptions[47].

What is an exception?

An **unexpected event that occurs during program execution**, e.g.:

```
System.out.println(5 / 0); // Infinity??
```

No — integer division by zero causes a `java.lang.ArithmeticException: / by zero`. Infinity is the tempting wrong answer: it is what floating-point division gives, not integer division.

Common causes:

- invalid user input
- loss of network connection
- physical limitations (out of disk memory)
- code errors
- opening an unavailable file

When an exception occurs within a method **it creates an object** — the **exception object** — which contains information about the exception. *Effective Java* (2017), Item 75, says to *include failure-capture information in detail messages*: that object is where the diagnostic information has to live.

We can catch the exception to handle it, and the catch block is where the program is put back into a usable state:

[46] courses/csse2002/2026-03-19-exceptions.pdf
[47] courses/csse2002/week5-tutorial-exceptions.pdf

```java
try {
    System.out.println(5 / 0);
}  catch (ArithmeticException e) {
    System.out.println(e); // print out error message
    //do something to recover
}
```

Exception hierarchy

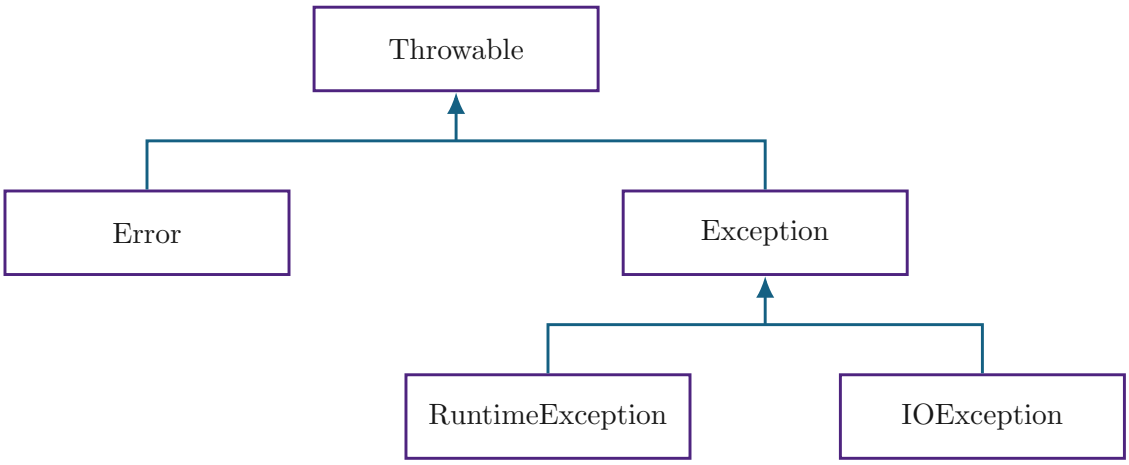


Figure 15: The exception hierarchy. Throwable is the root; Error and Exception sit beneath it; RuntimeException (unchecked) and IOException (checked) sit beneath Exception. Each connector joins the subclasses and points up to their parent. Which branch a class is on is what decides whether the compiler forces you to handle it, so this one picture settles the checked/unchecked question.

`RuntimeException` — unchecked

A runtime exception happens **due to a programming error**. Also known as **unchecked** exceptions: the compiler does not force you to handle them.

Cause	Exception
Improper use of an API	`IllegalArgumentException`
Null pointer access	`NullPointerException`
Out-of-bounds array access	`ArrayIndexOutOfBoundsException`
Dividing a number by 0	`ArithmeticException`

`IOException` — checked

Also known as a **checked** exception: it is checked by the compiler at compile time, and the programmer is prompted to handle it.

Cause	Exception
Trying to open a file that doesn't exist	`FileNotFoundException`
Trying to read past the end of a file	`EOFException`
A connection attempt to a remote host fails	`ConnectException`

Java exception handling

There are three mechanisms: the `try…catch` block, the `throw` and `throws` keywords, and the `finally` block.

`try … catch`

Place the code that might generate an exception inside the `try` block. **Every `try` block is followed by a `catch` block.**

```
int result;
try {
    result = 5 / 0;
    …
} catch (ArithmeticException e) {
    System.out.println(e);      // print out error message
    result = Integer.MAX_VALUE; // Assign a default or fallback value
}
```

1. When an exception occurs, **the rest of the code inside the `try` block is skipped**.
2. The `catch` block catches the exception and the statements inside the `catch` block are executed.
3. If none of the statements in the `try` block generates an exception, **the `catch` block is skipped**.

`throw`

The `throw` keyword is used to **explicitly throw a single exception**. Two examples:

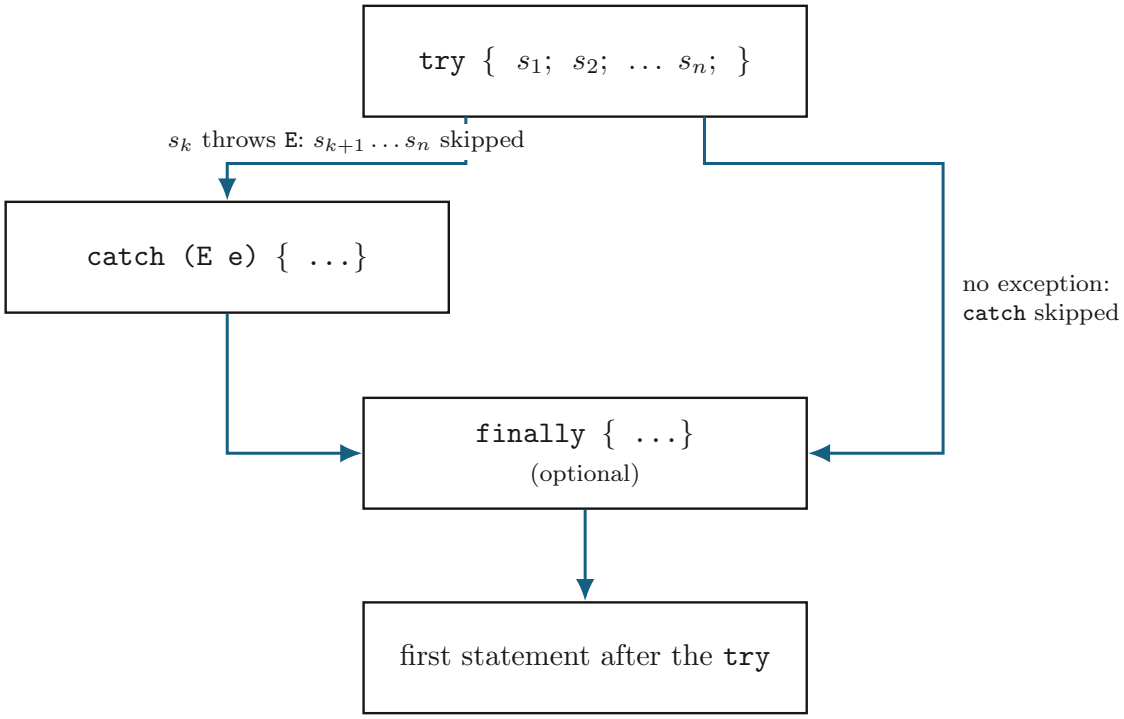


Figure 16: The two paths through a try/catch/finally, and the fact that both of them end up in finally. This is the three numbered rules together with the finally rule, drawn as one picture: the exceptional path abandons the remaining statements of the try and enters catch, the normal path skips catch entirely, and neither can skip finally.

```java
public class Main {
    public static void main(String[] args) {
        divideByZero();
    }
    public static void divideByZero() {
        // throw an exception
        throw new ArithmeticException("Trying to divide by 0");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        checkEntry(17);
    }

    public static void checkEntry(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Access denied ");
        }
        else {
            System.out.println("Access granted ");
        }
    }
}
```

`throw` is used to manually trigger an error when your program detects something invalid, so it doesn't continue in a bad state. That is the point of the whole keyword, and it is also the bridge to java-specification[48]: throwing is how a method refuses input that breaks its precondition instead of limping on.

`throws`

The `throws` keyword **declares the type of exceptions that might occur within the method**, and it is used **in the method declaration**. The caller is then the one prompted to handle it:

```java
public class ThrowsEx {
    public static void main(String[] args) {
        try {
            findFile();
        } catch (FileNotFoundException e) {
            System.out.println(e);
        }
```

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

```
    }
    public static void findFile() throws FileNotFoundException {
        // code that may generate FileNotFoundException
        File newFile = new File("file.txt");
        FileInputStream stream = new FileInputStream(newFile);
    }
}
```

`throw` and `throws` are not two spellings of one thing: `throw` (statement, inside a body) *raises* an exception; `throws` (clause, in a signature) *declares* that one can escape. Only checked exceptions have to be declared — that is exactly what "checked by the compiler" buys you.

`finally`

- The `finally` block is executed **regardless of whether an exception is thrown**.
- The `finally` block is **optional**.

```
try {
    //code
}
catch (ExceptionType1 e1) {
    // catch block
}
finally {
    // finally block always executes
}
```

Exercise — what will the following print?

```
try{
    System.out.println("Good Morning");
    throw  new FileNotFoundException();
    System.out.println("the earth says");
}catch (Exception e){
    System.out.println("hello");
}finally {
    System.out.println("world");
}
```

Answer:

```
Good Morning
hello
world
```

Working through it against the rules above: `"Good Morning"` prints; the `throw` immediately abandons the rest of the `try`, so `"the earth says"` never prints; the `catch` parameter is `Exception`, and `FileNotFoundException` is an `Exception`, so this catch matches and `"hello"` prints; `finally` always runs, so `"world"` prints last.

As written this snippet would not actually compile. `System.out.println("the earth says");` sits directly after an unconditional `throw` in the same block, and Java rejects that at compile time as an unreachable statement. Read it as a tracing exercise about the *order* of the three rules, not as something to paste into a file.

Custom exceptions

Define your own by extending `Exception` (or a subclass of it), then declare it with `throws` and raise it with `throw`:

```
public class MyCustomException extends Exception {
...

}

int myFunction() throws MyCustomException {
throw new MyCustomException();
}
```

Worked example — `ItemNotFoundException`

The exception class gives a no-argument constructor with a fixed message and a second constructor taking a custom one, both delegating to `super(...)` — the `Throwable` message is the failure-capture information from Item 75:

```
public class ItemNotFoundException extends Exception{
    public ItemNotFoundException(){
        super("Item not found");
    }
    // Constructor that accepts a custom message
    public ItemNotFoundException(String message) {
        super(message);
    }
}
```

The class that uses it declares `throws` on the searching method and throws when the loop falls through without a match:

```
public class Store {
    private List<String> items;
    public Store() {
        items = new ArrayList<>();
        items.add("apple");
        items.add("banana");
        items.add("orange");
    }
    public String findItem(String itemName) throws ItemNotFoundException {
        for (String item : items) {
            if (item.equals(itemName)) {
                return item;
            }
        }
        throw new ItemNotFoundException("Item '" + itemName + "' not found in the
        ↪ list.");
    }

    public static void main(String[] args) {
        Store store = new Store();
        try {
            String item = store.findItem("grape");
            System.out.println("Found item: " + item);
        } catch (ItemNotFoundException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

`findItem("grape")` misses, so the `try` is abandoned at the call — `"Found item: …"` never prints — and the catch reports `Item 'grape' not found in the list.` on `System.err`.

Java Inheritance

Introduced in 2026-03-05-object-oriented-programming-i[49] (Lecture, Week 2) — "things you have because your parents have them".

Subclasses and superclasses

Inheritance allows creating a new class from an existing class:

[49] courses/csse2002/2026-03-05-object-oriented-programming-i.pdf

- The new class is the **subclass** (child/derived class).
- The existing class it's derived from is the **superclass** (parent/base class).

The running example is one `Employee` superclass with two subclasses:

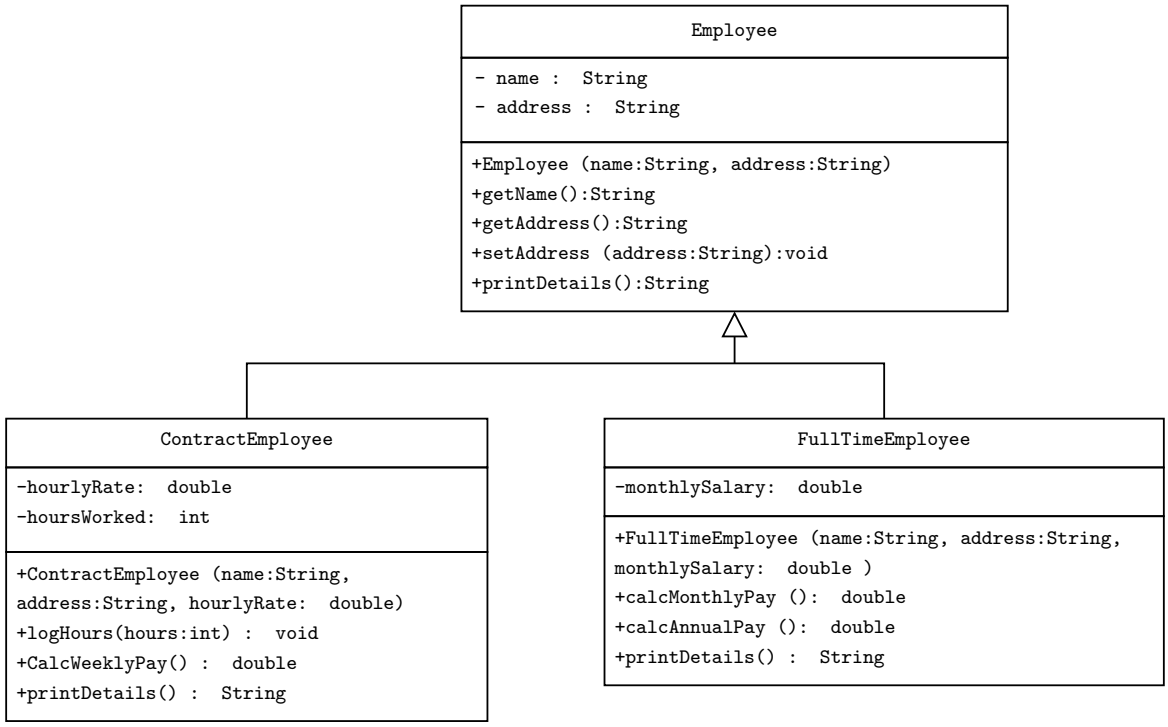


Figure 17: The `Employee` inheritance example as a UML class diagram. Each box has three compartments (name / attributes / operations), with - for private and + for public, and constructors listed in the operations compartment with no return type. Generalisation is a solid line with a single unfilled triangle arrowhead entering the superclass.

This form is not a course-wide standard. The course draws inheritance in **three** ways: this UML three-compartment form with an unfilled triangle, a labelled box with a plain arrowhead, and a diagonal form with every edge labelled "Is a". **None of them is declared canonical.** See csse2002-diagram-notation[50] for the compartment, visibility and arrowhead conventions this figure uses.

Two things to read off the diagram before looking at any code:

- `name` and `address` are declared **`private in Employee`** and are *not* repeated in either subclass. Every `ContractEmployee` carries them, but the subclass's own code cannot

[50] courses/csse2002/csse2002-diagram-notation.pdf

name them — `private` members are not accessible in a subclass — which is what forces the `super(...)` call below.
- `printDetails()` appears in **all three** boxes. That repetition is the setup for method overriding.

(The diagram writes `CalcWeeklyPay()` with a capital C; the code calls the same method `calculateWeeklyPay()`. The code spelling is the one to trust.)

`extends`

`extends` is the keyword used to implement inheritance in Java:

```
public class Employee {
    private String name;
    private String address;

    public Employee(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public String getName() { return name; }
    public String getAddress() { return address; }
    public void setAddress(String address) { this.address = address; }
    public String printDetails() { return name + " " + address; }
}

public class ContractEmployee extends Employee {
    private double hourlyRate;
    private int hoursWorked;

    public ContractEmployee(String name, String address, double hourlyRate) {
        super(name, address);
        this.hourlyRate = hourlyRate;
        this.hoursWorked = 0;
    }
    public void logHours(int hours) { this.hoursWorked += hours; }
    public double calculateWeeklyPay() { return hourlyRate * hoursWorked; }
}
```

From the diagram, the shape of `FullTimeEmployee` is `private double monthlySalary`, a constructor `FullTimeEmployee(String name, String address, double monthlySalary)`, and `calcMonthlyPay()` / `calcAnnualPay()` both returning `double`.

The `super` keyword

`super` is used in a subclass to access superclass **members — attributes, constructors and methods**.

As a constructor call, `super(name, address)` runs `Employee`'s constructor first, so the inherited `name` and `address` are initialised before the subclass's own constructor body runs. This is not merely tidy: `name` and `address` are `private` in `Employee`, so `ContractEmployee` has no other way to set them.

Method overriding

The subclass inherits the attributes and methods of its superclass. If the **same method** is defined in both the superclass and the subclass, the subclass's version **overrides** the superclass's version — and both **MUST** have the same method signature (see java-encapsulation[51]; a different signature gives you an overload, not an override).

```
public class Employee {
    ...
    public String printDetails() {
        return name + " " + address;
    }
}

public class ContractEmployee extends Employee {
    ...
    @Override
    public String printDetails() {
        return "Role: Contract Employee, Hourly Rate: $" + hourlyRate
             + ", Hours Worked: " + hoursWorked;
    }
}

public class FullTimeEmployee extends Employee {
    @Override
    public String printDetails() {
        return "Role: Full-Time Employee, Monthly Salary: $" + monthlySalary;
    }
}
```

`super` in an override

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

The superclass's method is still reachable after overriding it. `super.methodName(...)` calls the superclass's version from inside the override, so the subclass can **extend** the inherited behaviour rather than replace it:

```
public class FullTimeEmployee extends Employee {
    ...
    @Override
    public String printDetails() {
        return super.printDetails()
             + ", Role: Full-Time Employee, Monthly Salary: $" + monthlySalary;
    }
}
```

With `Employee.printDetails()` returning `name + " " + address`, this version prints the name and address *and* the role and salary, where the previous version threw the inherited half away.

Choosing whether `Employee`'s `printDetails()` or a subclass's override actually runs for a given object is **dynamic dispatch**, taken up with polymorphism — see java-polymorphism[52] and week4-tutorial-method-dispatch-and-casting[53].

Java JUnit

Introduced in 2026-03-26-testing[54] (Lecture, Week 5). See java-testing[55] for the broader testing concepts JUnit is used to implement.

What is JUnit?

An open-source test automation framework for Java — one of many *xUnit* frameworks (JUnit, NUnit, CPPUnit, PyUnit…). This course uses **JUnit 4** (http://junit.org/junit4, and the cookbook at https://junit.org/junit4/cookbook.html). Depending on how software is structured, unit-testing frameworks like JUnit can also be used for other kinds of automated testing, but "unit testing" is sometimes *incorrectly* used to mean any automated test.

[52] courses/csse2002/java-polymorphism.pdf
[53] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf
[54] courses/csse2002/2026-03-26-testing.pdf
[55] courses/csse2002/java-testing.pdf

Writing tests

Tests are defined in classes — **one test class per real class**. `@Test` marks a method as a test JUnit should run; test methods **take no parameters and return `void`**, and should be named for what they check.

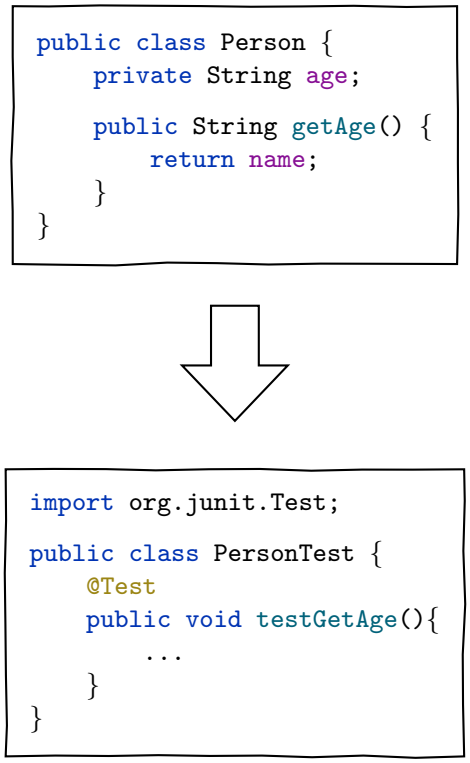


Figure 18: One test class per real class: `Person` produces `PersonTest`, which imports `org.junit.Test` and carries one `@Test`-annotated method per thing being checked. The hollow block arrow pointing down means produces/translates-to. The `Person` snippet does not compile as drawn: `getAge()` returns `name`, which the snippet never declares, and the field is typed `String age`. The full `Person` below types it `int` and is the version that compiles.

The full class and its test class:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
```

```
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class PersonTest {

    @Test
    public void TestGetAge() {
        Person p1 = new Person("Jack", 10);
        int age = p1.getAge();
        assertEquals(10, age);
    }

    @Test
    public void TestGetName() {
        Person p1 = new Person("Jack", 10);
        String name = p1.getName();
        assertEquals("Jack", name);
    }
}
```

The test class uses two imports: `org.junit.Test` for the annotation, and a **static** import of `org.junit.Assert.assertEquals` so the assertion can be called bare rather than as `Assert.assertEquals(...)`.

`@Before` and `@After`

Each test method should be independent, but you may want common things set up before each test (e.g. object creation):

```
public class PersonTest {

    private Person person;
```

```
    @Before
    public void setUp() {
        person = new Person("Jack", 10);
    }

    @After
    public void tearDown() {
        person = null;
    }
    // ...
}
```

`@Before` will be run before **each** test; `@After` will be run after **each** test. Every test method gets a freshly built fixture rather than whatever the previous test left behind, which is how the tests are kept independent.

`Assert`

Some static methods in `Assert`:

- `assertEquals` — asserts that two objects are equal.
- `assertArrayEquals` — asserts that two object arrays are equal.
- `assertFalse` / `assertTrue` — asserts that a condition is false or true.
- `assertSame` / `assertNotSame` — asserts that two objects refer to the same object or not.

The difference between `assertSame` and `assertEquals` is reference identity vs `.equals`: two distinct `Person` objects with the same fields can be `assertEquals`-equal (if `equals` is overridden) but are never `assertSame`.

Checking for exceptions

Don't catch the exception — the test runner does. Tell the test runner which exception it should catch, via `@Test(expected = ...)`:

```
@Test(expected = EOFException.class)
public void testExceptions() throws EOFException {
    callThatShouldThrowEOFException();
}
```

`.class` is required after the name of the class. The test **passes** if that exception is thrown and **fails** if it is not, or if a different one is. See java-exceptions[56].

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

Black box tests in JUnit

Write the tests purely from the specification/Javadoc, without looking at the implementation, then use smoke tests, boundary tests and equivalence classes (see java-testing[57]) to decide what to check. The hard constraint is that **tests must not violate preconditions**. A test that breaks `@requires` tells you nothing, because the method is entitled to do anything at all in that case.

White box tests in JUnit

Design tests based on the **internal logic**. You try to cover branches, loops and edge cases, and you may aim for a code coverage target (e.g. 100%).

The worked example is a discount calculator with two nested levels of branching:

```
public class DiscountCalculator {

    public double calculateDiscount(double amount, boolean isMember) {
        if (amount <= 0) {
            return 0;
        }

        if (isMember) {
            if (amount > 100) {
                return amount * 0.2;
            } else {
                return amount * 0.1;
            }
        } else {
            if (amount > 100) {
                return amount * 0.1;
            } else {
                return 0;
            }
        }
    }
}
```

The five tests, one per leaf:

```
public class DiscountCalculatorTest {

    DiscountCalculator calc = new DiscountCalculator();
```

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

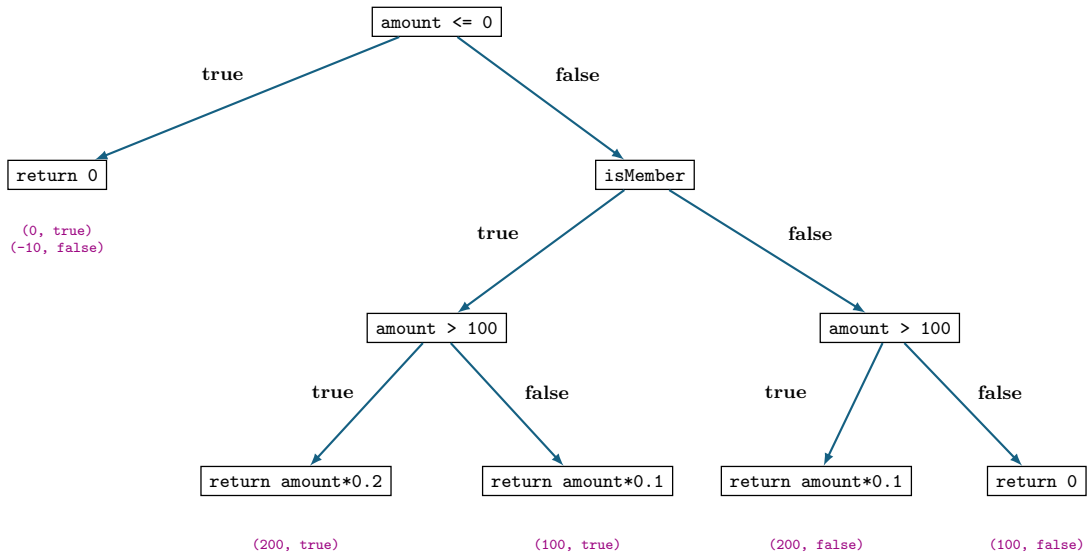


Figure 19: Why `DiscountCalculator` needs exactly five tests: the method as a decision tree, one leaf per test. Each box is a condition from the method or the `return` its branch reaches, so the expected values in the tests can be read off the leaves. The magenta `(amount, isMember)` pair under each leaf is the argument list of the test that reaches that leaf. The early return on `amount <= 0` is the one leaf covered by two assertions in a single test, which is why it carries two pairs.

```
    @Test
    public void testAmountLessOrEqualZero() {
        assertEquals(0.0, calc.calculateDiscount(0, true));
        assertEquals(0.0, calc.calculateDiscount(-10, false));
    }

    @Test
    public void testMemberAbove100() {
        assertEquals(40.0, calc.calculateDiscount(200, true));
    }

    @Test
    public void testMemberBelowOrEqual100() {
        assertEquals(10.0, calc.calculateDiscount(100, true));
    }

    @Test
    public void testNonMemberAbove100() {
        assertEquals(20.0, calc.calculateDiscount(200, false));
    }

    @Test
    public void testNonMemberBelowOrEqual100() {
        assertEquals(0.0, calc.calculateDiscount(100, false));
    }
}
```

Two things worth reading off this example:

- The tests are **named for the branch they cover**, not for the method (`testMemberAbove100`, not `testCalculateDiscount3`), which is what naming a test for what it checks looks like in practice.
- `amount = 100` is the argument in two of the five tests. It is the boundary of `amount > 100`, so it is both a white-box branch probe and a black-box boundary test.

> **i** assertEquals on doubles
>
> The tests above call `assertEquals(0.0, ...)` on `double` values. JUnit 4's two-argument `assertEquals(double, double)` is deprecated precisely because floating-point equality is unreliable; the three-argument `assertEquals(expected, actual, delta)` is the form that compiles without a deprecation warning. The code above will warn if you type it in.

Java Loop Invariants

Introduced in 2026-05-21-correct-programming[58] (Lecture, Week 12). Extends java-hoare-logic[59]'s backward-reasoning technique to loops, where the number of iterations isn't known up front.

> …proving the correctness of algorithms has another aspect that is even more important: it mirrors the way we understand an algorithm.
>
> — Donald Knuth, *The Art of Computer Programming* (Vol. 1), 1997

What is an invariant?

> An **invariant** is a property of a system that does not change.

Motivating puzzle

A jar contains 100 red and 100 blue beans. Repeatedly: pick two random beans; if they're the same colour, discard both; if they're different colours, discard the blue one. **What colour is the last bean?**

Let r, b be the current counts of red/blue beans. Each step does exactly one of:

Pick	Action	Effect
two reds	discard both	$r \mapsto r-2,\ b \mapsto b$
two blues	discard both	$r \mapsto r,\ b \mapsto b-2$
one of each	discard blue	$r \mapsto r,\ b \mapsto b-1$

What never changes? The *parity* of r — every action either leaves r unchanged or decreases it by exactly 2, so r stays even throughout (it starts at 100, itself even). When one bean remains, $r + b = 1$; since r must be even, $r \neq 1$, so $r = 0$ and $b = 1$ — **the last bean is blue**.

The property "r is even" is an invariant of the process: it (1) holds before any step, (2) is preserved by every step, and (3), combined with the termination condition ($r + b = 1$), implies what we wanted to prove. This is exactly the structure of a **loop invariant**.

[58] courses/csse2002/2026-05-21-correct-programming.pdf
[59] courses/csse2002/java-hoare-logic.pdf

Loop invariants

> A **loop invariant** is a condition that: (1) holds before the loop is entered, (2) is preserved by each iteration, and (3) combined with the negation of the guard, implies the postcondition.

Those three clauses are three separate proof obligations, and they attach to three different places around the loop:

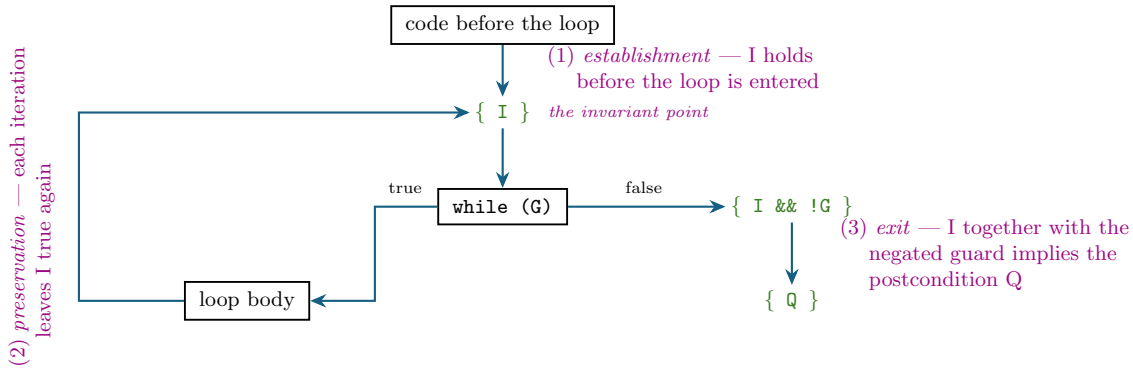


Figure 20: The three obligations in the definition above, each pinned to the point in the loop where it has to be discharged. The invariant point is the top of the loop, before the guard is tested: establishment is the arrow into it, preservation is the arrow back to it, and exit is what the guard's failure adds to it. Connectors are teal and assertions are green.

None of the three clauses in Figure 20 says anything about the loop *finishing*. An invariant that survives all three checks establishes only "*if* the loop ends, the postcondition holds"; termination is a separate argument — in `power` below, that `exp - i` strictly decreases and is bounded below.

```
int count = 0;
while (count < n) {
    // invariant: count >= 0
    count = count + 1;
}
```

```
// {0 >= 0}
int count = 0;
// {count >= 0}                    <-- invariant holds before the loop
while (count < n) {
    // invariant: count >= 0
    // {count + 1 >= 0}
```

```
    count = count + 1;
    // {count >= 0}                <-- preserved by the body
}
// {count >= 0 && count >= n}
```

Deriving an invariant for `power`

```
/**
 * @requires ????
 * @ensures \result == base^exp
 */
int power(int base, int exp) {
    int result = 1;
    int i = 0;
    while (i < exp) {
        result = result * base;
        i = i + 1;
    }
    return result;
}
```

Step 1 — guess an invariant. Replace `exp` in the postcondition with the loop variable `i`: I : result $=$ base^i.

Step 2 — verify. Three checks:

1. Does I hold *before* the loop first executes? `result = base^i` $\equiv 1 =$ base$^0 \equiv$ true.
2. Assuming I holds at the top of the loop body, does it still hold after? `result * base = base^i * base = base^{i+1}`, and after `i = i + 1`, that's exactly `result = base^i` again.
3. Does I combined with the negated guard ($i \geq$ exp) imply the postcondition? `result = base^i` $\wedge\ i \geq$ exp $\Rightarrow$ result $=$ base^{exp}? **Not quite** — we can only conclude $i \geq$ exp, but we need $i =$ exp.

Step 3 — strengthen. Add the bound $i \leq$ exp to the invariant: I : result $=$ base$^i \wedge i \leq$ exp. Re-checking: I now holds initially iff exp ≥ 0 (giving us our precondition), is still preserved by the loop body, and combined with $i \geq$ exp now forces $i =$ exp exactly, giving the postcondition:

```
/**
 * @requires exp >= 0
 * @ensures \result == base^exp
 */
```

```
int power(int base, int exp) {
    // {0 <= exp}
    // {1 == base^0 && 0 <= exp}
    int result = 1;
    int i = 0;
    // {result == base^i && i <= exp}
    while (i < exp) {
        ...
    }
    // {result == base^i && i <= exp && i >= exp}
    // {result == base^exp}
    return result;
}
```

The precondition comes from obligation (1) of Figure 20, failing, and being read backwards. `exp >= 0` was never designed in — it is whatever the establishment check needed in order to succeed.

Designing an algorithm *from* an invariant: fast exponentiation

The `power` above runs in $O(\text{exp})$ multiplications. We can do better using the identity:

$$b^e = \begin{cases} (b^{e/2})^2 & \text{if } e \text{ is even} \\ b \times (b^{(e-1)/2})^2 & \text{if } e \text{ is odd} \end{cases}$$

E.g. computing 3^{13}: $3^{13} = 3 \times 3^{12}$ [13 odd] $= 3 \times (3^2)^6$ [12 even] $= 3 \times 9^6 = 3 \times (9^2)^3$ [6 even] $= 3 \times 81^3 = 3 \times 81 \times 81^2$ [3 odd] $= 3 \times 81 \times 6561 = 1{,}594{,}323$ — only a handful of multiplications instead of 12 sequential ones.

Design from the invariant: rather than writing code and *then* finding its invariant, pick the invariant *first* and let the code follow. We want a loop maintaining:

$$I : \text{result} \times b^e = \text{base}^{\text{exp}}$$

starting from `result = 1`, `b = base`, `e = exp`, terminating when `e == 0`:

```
/**
 * @requires exp >= 0
 * @ensures \result == base^exp
 */
int power(int base, int exp) {
    int result = 1;
```

```
    int b = base;
    int e = exp;
    while (e > 0) {
        // invariant: result * b^e == base^exp
        if (e % 2 == 1) {
            result = result * b;
            e = e - 1;
        }
        b = b * b;
        e = e / 2;
    }
    return result;
}
```

Verifying I (using $b^e = (b^{e/2})^2$ when even, $b \times (b^{(e-1)/2})^2$ when odd, and $(a^b)^c = (a^c)^b$):

1. Before the loop: `result * b^e == base^exp` $\equiv 1 \times \text{base}^{\text{exp}} = \text{base}^{\text{exp}} \equiv$ true.
2. Through the body: splitting on `e % 2`, both branches reduce back to exactly `result * b^e == base^exp` after the update (the `if` branch peels off one factor of `b` into `result` and decrements `e` first; both branches then square `b` and halve `e`).
3. On exit (`e <= 0`): `result * b^e == base^exp` with `e <= 0` only concludes the post-condition when `e >= 0` too (i.e. `e == 0`) — which holds as long as `exp >= 0`, giving the same precondition as before.

The invariant is easiest to believe by watching it *not move*. Running the loop on the same 3^{13} worked above:

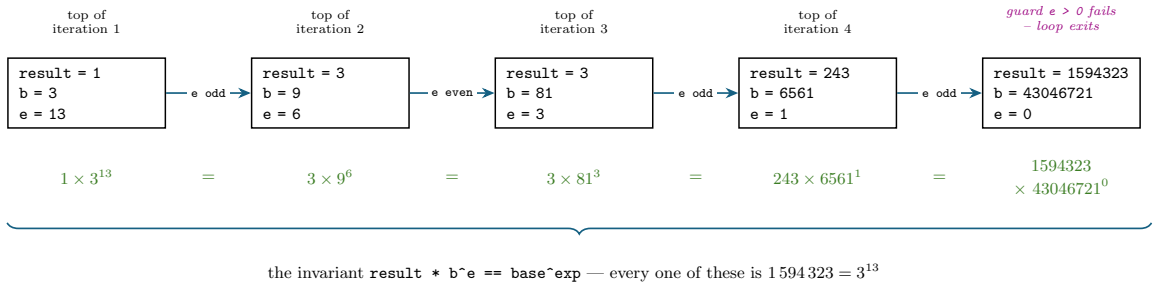


Figure 21: The fast-`power` loop traced on `power(3, 13)`, one box per arrival at the top of the loop. The three variables change on every step; the product `result * b^e` underneath does not, and the five products it takes are exactly the chain the section above works by hand (the section's `3 x 81 x 6561` and this trace's `243 x 6561` are the same number). Connectors are teal and the invariant is green.

This preserves the *exact same specification* as the slow version — both satisfy `@requires exp >= 0` / `@ensures \result == base^exp` — but the fast version reaches it in $O(\log \text{exp})$

multiplications instead of $O(\text{exp})$.

Invariants in the Week 12 exercises

Neither Week 12 exercise set is *about* correctness — the applied class is lambdas and streams (week12-tutorial-lambdas-and-streams[60]) and the practical is generics (week12-lab-generics[61]) — but both contain loops whose behaviour is only explicable in these terms, and the practical writes contracts for two of them.

The loop bound that must not move

The practical's `transferStudents` (Task 4) empties one bus into another. Its loop carries a one-line comment:

```
// Ensure loop condition does not change while iterating
int numStudents = students.getCurrentCount();
for (int i = 0; i < numStudents; i++) {
    Concession student = students.passengerOff();
    bus.passengerOn(student);
}
```

The contract this loop has to establish is its `@ensure` (given in full in java-hoare-logic[62]), whose first clause is `students.getCurrentCount() == 0`. Taking the invariant at the top of the iteration with counter `i` to be

`0 <= i && i <= numStudents && students.getCurrentCount() == numStudents - i`,

and `bus` holds everything it started with plus the `i` students already moved,

the three obligations of Figure 20 go through: at `i == 0` nothing has moved and `getCurrentCount()` is still `numStudents`; each iteration takes exactly one passenger off and puts exactly one on before `i++`, restoring it; and exiting with `i == numStudents` gives `students.getCurrentCount() == 0` together with all `numStudents` students on `bus`.

Now write the guard the obvious way instead, reading the count live, and the invariant has nothing to hold onto: `i` climbs by one per iteration while `students.getCurrentCount()` falls by one, so the two meet halfway and the loop stops with half the bus still loaded.

In general the live-bound version runs $\lceil n/2 \rceil$ iterations and leaves $\lfloor n/2 \rfloor$ students behind, because `i` and the count meet at $i = n - i$. The fix is the standard one: **a loop guard should not read state the body is mutating.** Copying the bound into a local before the loop is what turns `numStudents` into something the invariant can be stated against. Task 5's

[60] courses/csse2002/week12-tutorial-lambdas-and-streams.pdf
[61] courses/csse2002/week12-lab-generics.pdf
[62] courses/csse2002/java-hoare-logic.pdf

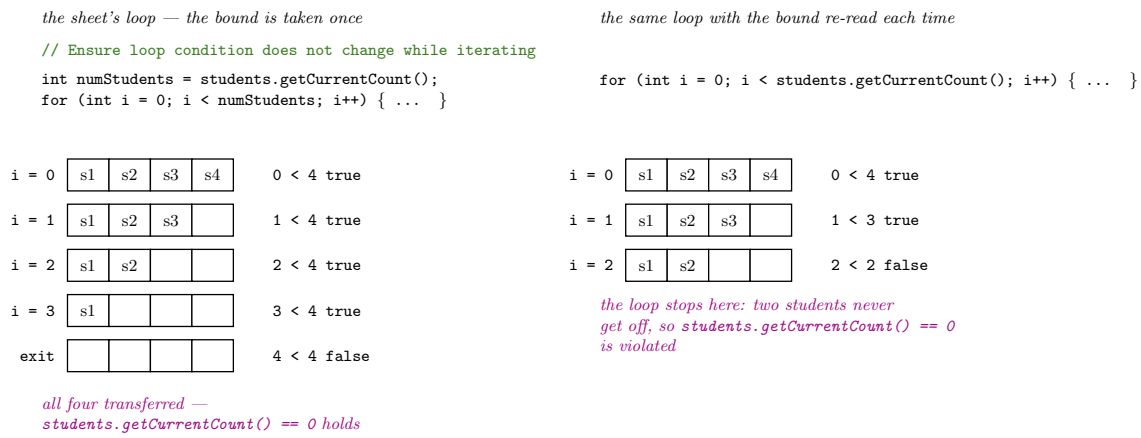


Figure 22: Why the practical snapshots its loop bound. Both columns run the same body on a source bus of four; the left column is the loop with the bound read once into `numStudents`, the right is the same loop with `students.getCurrentCount()` in the guard. Reading the count live makes the guard chase a quantity the body is decreasing, and the two meet after two iterations. Bus contents are drawn as array rows and the verdict on each column is in magenta.

`transferAllConcession` uses the same body and the same snapshot, with the type parameter `T` in place of `Concession`, so the same invariant carries over unchanged.

An accumulator loop inside a lambda

The applied class's Task 9 hides an ordinary counting loop inside a `Function`:

```
Function<Integer, Integer> factorial = n -> {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
};
System.out.println(factorial.apply(4));   // 24
```

The task asks only for the printed value. The invariant, following the `power` recipe of replacing the target quantity's bound with the loop variable, is I : result $= (i-1)!\ \wedge\ 2 \le i \le n+1$:

1. **Establishment.** At the top, `i == 2` and `result == 1 == 1!`. — but the bound clause needs $2 \le n+1$, so this invariant establishes only for `n >= 1`, which is the derived precondition. (`factorial.apply(0)` still returns `1`, correctly; that just isn't something *this* invariant proves.)

2. **Preservation.** `result * i == (i-1)! * i == i!`, and `i++` then makes that `(i-1)!` again.
3. **Exit.** The guard fails at `i > n`, which with `i <= n + 1` forces `i == n + 1`, so `result == n!`.

At `n == 4` that is `4! == 24`, derived rather than simulated.

The same idea, written into a method signature

The applied class's Task 18 computes the same shape of thing with no loop at all:

```
Stream<Integer> stream = Stream.iterate(1, n -> n * 2).limit(5);
Integer result = stream.reduce(0, (a, b) -> a + b);   // 31
```

`reduce`'s first argument is the identity and its second is the step. Read as a loop, those are precisely obligations (1) and (2): the identity `0` establishes the invariant "the accumulator holds the sum of the elements consumed so far", and `(a, b) -> a + b` is the step that preserves it. When the stream runs out — the negated guard — the accumulator holds the sum of all five elements, $1 + 2 + 4 + 8 + 16 = 31$. Nothing about the reasoning has changed; the fold's signature just makes you write the invariant's two halves down where a `for` loop would have let you leave them implicit. See java-lambdas-and-streams[63] for `reduce` itself.

Pragmatic considerations

Proving correctness can be vital, but it's also time-consuming — you want code running on a plane or controlling trains to be correct; you may not care nearly as much whether a weekend web-scraping script is correct. Where formal reasoning tends to pay off:

- **Combinatorial algorithms** — proving a faster/trickier implementation still satisfies the original spec (as with `power` above).
- **Distributed and concurrent algorithms** — notoriously hard to reason about by testing alone (see CSSE3610/7610).

Reasoning about programs this way provides (1) a more rigorous way to think about your software, and (2) a better interface design for others to interact with it — it doesn't remove the need to think about your programs, but gives a framework to structure that thinking.

A loop invariant is not the only kind of invariant in this course: a **class invariant** is the same idea applied to an object's fields across every public method rather than to a loop's variables across every iteration — established by the constructor, preserved by each method, relied on by all of them. See week8-tutorial-class-invariants[64].

[63] courses/csse2002/java-lambdas-and-streams.pdf
[64] courses/csse2002/week8-tutorial-class-invariants.pdf

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> — Donald Knuth (memo to Peter van Emde Boas, 1977)

Java Polymorphism

Introduced in 2026-03-12-object-oriented-programming-ii[65] (Lecture, Week 3). Practised in week4-tutorial-method-dispatch-and-casting[66].

What is polymorphism?

From the Greek *"poly"* (many) and *"morphe"* (form). In programming, it allows the same method call to produce different behaviour depending on the object that executes it.

There are three named kinds. The first two differ in **when** the method body is chosen; the third is about **what types a call site will accept**.

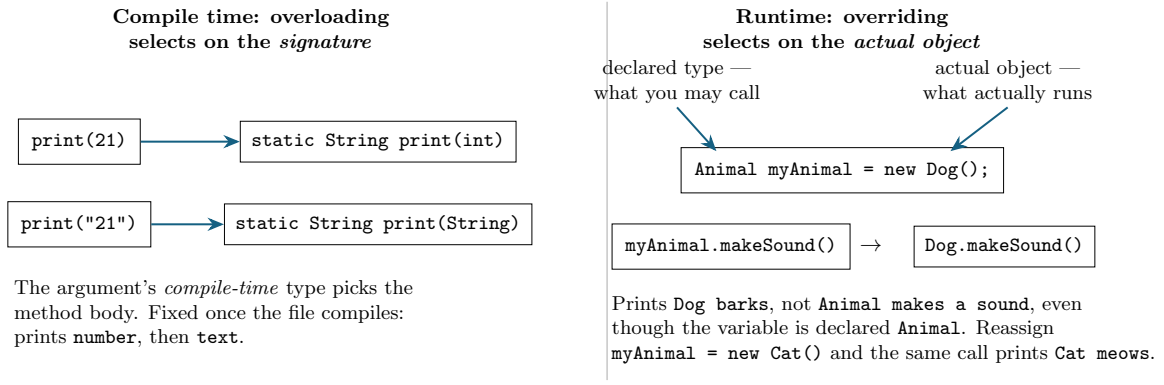


Figure 23: The two selection mechanisms. Left: overloading is resolved by the compiler from the argument's compile-time type, so the choice is fixed once the file compiles. Right: overriding is resolved by the JVM from the object actually on the other end of the reference; the arrow from `myAnimal.makeSound()` to `Dog.makeSound()` means the call dispatches to that method.

Compile-time polymorphism (method overloading)

> **Method overloading** occurs when multiple methods have the same name but different parameter lists (different type, number, or both). The method to be executed is determined at compile time based on the method signature.

[65] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[66] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf

Which one runs is decided from the signature (see java-encapsulation[67]) before the program ever runs:

```
public class CompPolymorphism {
    static String print(int value) {
        return "number";
    }
    static String print(String value) {
        return "text";
    }

    public static void main(String[] args) {
        System.out.println(print(21));    // number
        System.out.println(print("21"));  // text
    }
}
```

Runtime polymorphism (dynamic method dispatch)

- Runtime polymorphism is also known as **dynamic method dispatch**.
- It occurs when the method that gets executed is determined **at runtime based on the actual object type, not the variable type**.
- It is implemented through **method overriding** — when a subclass provides a specific implementation for a method already defined in its superclass, the subclass's method is the *overridden* method (see java-inheritance[68]).

The rule:

> The subclass's overriding method executes, regardless of the compile-time type of the subclass instance. — *Effective Java*

The example below **reassigns the same variable**: nothing about the variable changed, only the object it points at.

```
public class Animal {
    public void makeSound() { System.out.println("Animal makes a sound"); }
}
public class Cat extends Animal {
    @Override public void makeSound() { System.out.println("Cat meows"); }
}
public class Dog extends Animal {
    @Override public void makeSound() { System.out.println("Dog barks"); }
}
```

[67] courses/csse2002/java-encapsulation.pdf
[68] courses/csse2002/java-inheritance.pdf

```java
public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.makeSound();   // Dog barks

        myAnimal = new Cat();
        myAnimal.makeSound();   // Cat meows
    }
}
```

Even though the object type is `Animal`, the `Dog` version of `makeSound()` runs because the actual object is `Dog`.

Subtype polymorphism

- Subtype polymorphism occurs when objects of different subclasses can be treated as objects of a common superclass **or interface**.
- This is possible because subclasses are *subtypes* of their superclass.
- It is achieved through inheritance (`extends`) or interface implementation (`implements`).
- It allows code to work with the abstraction (supertype) instead of specific implementations.

The rule the whole idea rests on:

> A subclass object can be used anywhere a superclass is expected.

Where runtime polymorphism is about *which body runs*, subtype polymorphism is about *what a call site will accept*: `performAnimalSound` is written once against `Animal` and never mentions `Dog` or `Cat`.

```java
public class TestSubtypePolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();

        performAnimalSound(myAnimal);   // a Dog object
    }

    public static void performAnimalSound(Animal animal) {
        animal.makeSound();             // Outputs: Dog barks
    }
}
```

The same method works unchanged when the supertype is an **interface** rather than a superclass. Any class that implements the interface can be passed:

```java
public interface Animal {
    void makeSound();
}

public class Cat implements Animal {
    @Override public void makeSound() { System.out.println("Cat meows"); }
}
public class Dog implements Animal {
    @Override public void makeSound() { System.out.println("Dog barks"); }
}
```

```java
public class TestSubtypePolymorphism {
    public static void main(String[] args) {
        performAnimalSound(new Dog());
        performAnimalSound(new Cat());
    }

    public static void performAnimalSound(Animal animal) {
        animal.makeSound();
    }
}
```

Choosing between the two supertypes is the interface-vs-abstract-class question — see java-abstraction[69].

How it connects to casting

Assigning `new Dog()` to an `Animal` variable is an implicit **upcast** (java-casting[70]). That is why the declared type restricts what you may call (`myAnimal.climb()` will not compile even when the object is a `Cat`) while the actual object still decides what runs — the left and right halves of Figure 23 are the same reference seen from the compiler's side and the JVM's side.

Java Refactoring

Introduced in 2026-04-02-refactoring[71] (Lecture, Week 6). Practised in week9-lab-refactoring[72].

[69] courses/csse2002/java-abstraction.pdf
[70] courses/csse2002/java-casting.pdf
[71] courses/csse2002/2026-04-02-refactoring.pdf
[72] courses/csse2002/week9-lab-refactoring.pdf

What is refactoring?

> A disciplined technique for continuously restructuring an existing body of code, altering its internal structure *without changing its external behaviour*.

Two halves make it a *technique* rather than a mood: **continuous** (not a one-off clean-up) and **behaviour-preserving** (the outside of the system cannot tell you did it). See also refactoring.guru.

Why refactor?

Over time, as features are added: quick fixes ("hacks") accumulate, the design becomes messy, and code becomes harder to change.

Good practice

- **Don't** refactor and add functionality at the same time — keep the two activities separate.
- Have good tests *before* refactoring, so you'll know immediately if you've broken something (see java-testing[73], java-junit[74]). This is what puts `Refactor` where it is in the development cycle, drawn in 2026-04-02-refactoring[75]: you only ever enter it from a passing test.
- Take short, deliberate steps: move a member variable, split a method, rename a variable. Refactoring is usually many small, localised changes that add up to a larger-scale change — small steps + testing after each one avoids prolonged debugging.
- Refactor early, refactor often.

When to refactor

There are two triggers: refactor when it would make the next change **easier to add**, and refactor when it would make the code **easier to understand**. Code smells (below) are a third: symptoms that point at where one of the first two applies.

[73] courses/csse2002/java-testing.pdf
[74] courses/csse2002/java-junit.pdf
[75] courses/csse2002/2026-04-02-refactoring.pdf

Example: making it easier to add a feature

A `Vehicle` class using a `type` string and a chain of `if/else` branches is fragile to extend. Adding a scooter means adding the *same* `else if (type.equals("scooter")) { return ..; }` block **three times over** — one for each place in the codebase that switches on `type`. Adding one vehicle is not one edit, it is an edit everywhere the string is tested, and nothing tells you when you have found them all.

```
public class Vehicle {
    private String type;

    public Vehicle(String type) { this.type = type; }

    public int travelTime(int distance) {
        if (type.equals("car")) {
            return distance / 80;
        } else if (type.equals("bike")) {
            return distance / 20;
        }
        return distance;
    }
}
```

Refactored to use inheritance/polymorphism (see java-polymorphism[76]), adding a `Scooter` is just one new class — no existing code needs editing:

```
abstract class Vehicle {
    public abstract int travelTime(int distance);
}
class Car extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 80; }
}
class Bike extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 20; }
}
class Scooter extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 5; }
}
```

The refactoring has moved *where* a new vehicle type lands. Before, it lands in the middle of an existing method (and in every other method that branches on `type`); after, it lands in a new leaf of the hierarchy that nothing existing has to know about:

[76] courses/csse2002/java-polymorphism.pdf

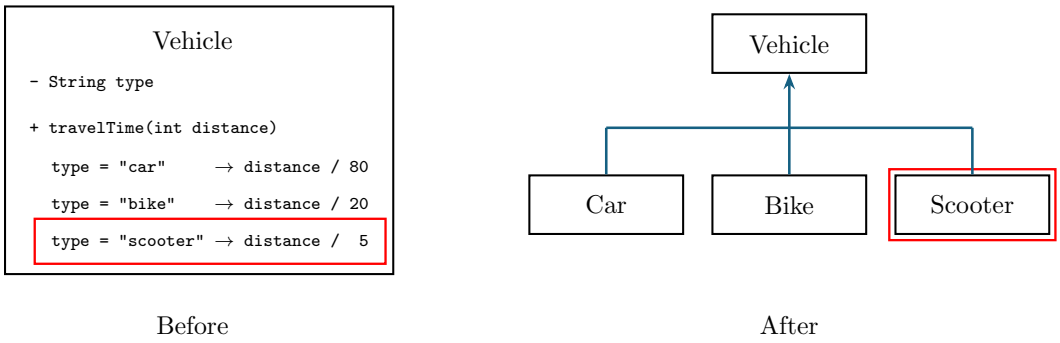


Figure 24: Adding a scooter, before and after. Left: one class, and the new type is another branch spliced into an existing method — the red outline marks the edit. Right: an abstract Vehicle with one subclass per type, joined by a connector with a single arrowhead into the parent. The new type is a whole new box that no existing box points at, again outlined in red.

What has *not* changed: `travelTime(int distance)` still exists and still returns the same answers for cars and bikes. That is what makes this a refactoring rather than a rewrite. The `Vehicle` abstraction being open to new subclasses without being edited is the Open-Closed Principle arriving early — see java-solid-principles[77].

Example: making it easier to understand

The second trigger. Here nothing about the structure changes — the same three branches run in the same order — but the conditions are replaced by named boolean-returning methods, so the reader sees the *intent* of a branch instead of re-deriving it from raw comparisons every time:

```
// Before
private int temperature;
private boolean isRaining;
public void provideWeatherAdvice() {
    if (temperature > 25 && !isRaining) {
        wearSunscreen();
        hydrate();
    } else if (temperature <= 25 && !isRaining) {
        play();
    } else if (isRaining) {
        equipUmbrella();
    }
}
```

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

```
// After
private int temperature;
private boolean isRaining;
public void provideWeatherAdvice() {
    if (isHotAndSunny()) {
        wearSunscreen();
        hydrate();
    } else if (isPleasant()) {
        play();
    } else if (isRaining) {
        equipUmbrella();
    }
}
```

`isRaining` is left as it was, because a field already named for the condition it tests reads fine — only the two *compound* conditions needed a name. The bodies of the extracted methods (`isHotAndSunny()`, `isPleasant()`) are the conditions they replaced.

Code smells

A "code smell" is a surface indication that usually corresponds to a deeper problem in the design. Three are worked here.

Bad naming

Single-letter and cryptic names force the reader to trace the logic before they can know what it is *for*. The two snippets compile to the same thing; only the second one tells you it counts even numbers:

```
// Before
int r = 0;
for (int x : y) {
    if (g(x)) {
        r++;
    }
}
return r;
```

```
// After
int evenCount = 0;
for (int number : numbers) {
```

```
    if (isEven(number)) {
        evenCount++;
    }
}
return evenCount;
```

Every name is renamed, including the loop variable and the collection — `r → evenCount`, `x → number`, `y → numbers`, `g → isEven`.

Duplication

Near-identical blocks repeated for different data. Two arrays, two copy-pasted loops; a fix to one has to be remembered for the other:

```
// Before
int[] first = new int[]{1, 2, 3};
int[] second = new int[]{4, 5, 6};

int firstSum = 0;
for (int i = 0; i < 3; i++) {
    firstSum += first[i];
}
int firstAverage = firstSum / 3;

int secondSum = 0;
for (int i = 0; i < 3; i++) {
    secondSum += second[i];
}
int secondAverage = secondSum / 3;
```

```
// After
int[] first = new int[]{1, 2, 3};
int[] second = new int[]{4, 5, 6};

int firstAverage = average(first);
int secondAverage = average(second);

int average(int[] numbers) {
    int sum = 0;
    for (int i = 0; i < 3; i++) {
        sum += numbers[i];
    }
    return sum / 3;
}
```

The duplicated loop now exists once, and the two call sites read as what they mean. The extracted `average` still hardcodes the length `3` in both the loop bound and the divisor, so it silently only works for three-element arrays — `numbers.length` in both places is the obvious next step. Extracting the duplication and *then* generalising the extracted method is exactly the "short, deliberate steps" style above.

Feature envy

A method that spends most of its time reaching into *another* class's data rather than its own. `User.checkout()` touches `cart` twice and `User`'s own state never — the logic is envious of `ShoppingCart`:

```
// Before
class ShoppingCart {
    public List<Item> getItems() {...}
    public int getVoucher() {...}
}

class User {
    private ShoppingCart cart;

    public double checkout() {
        double totalPrice = 0.0;
        for (Item item : cart.getItems()) {
            totalPrice += item.getPrice();
        }
        return totalPrice - cart.getVoucher();
    }
}
```

```
// After
class ShoppingCart {
    public List<Item> getItems() {...}
    public int getVoucher() {...}

    public double checkout() {
        double totalPrice = 0.0;
        for (Item item : getItems()) {
            totalPrice += item.getPrice();
        }
        return totalPrice - getVoucher();
    }
}

class User {
    private ShoppingCart cart;
```

```
}
```

Moving `checkout()` onto `ShoppingCart` turns two cross-class calls into two calls on `this` — the giveaway is that the `cart.` qualifier simply disappears:

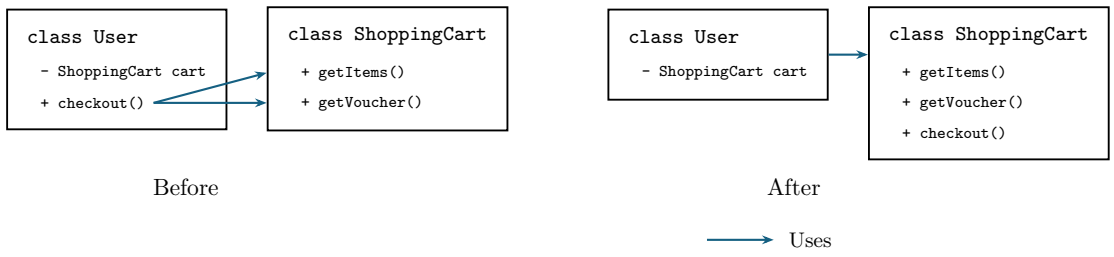


Figure 25: Feature envy, before and after. Before: User.checkout() reaches across the class boundary into two of ShoppingCart's members. After: checkout() sits with the data it uses, and User's only link to ShoppingCart is the field it holds. Each box is a class with its members listed as '+ m()'. A solid arrow means 'uses' and points from the depender to the thing depended upon; it may land on an individual member row.

Two things improve at once: `ShoppingCart` gets more cohesive (the checkout logic now sits with the data it operates on) and `User` gets less coupled to it (one link instead of three). Both are the subject of java-cohesion-and-coupling[78].

Others

Many other code smells exist. Further reading:

- Martin Fowler, *Refactoring: Improving the Design of Existing Code* (2nd edition, 2019)
- Robert Martin, *Clean Code: A Handbook of Agile Software Craftsmanship* (2008)
- John Ousterhout, *A Philosophy of Software Design* (2018)
- and the catalog of refactorings at refactoring.com.

Java SOLID Principles

Introduced in 2026-04-02-refactoring[79] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[80] (Lecture, Week 7 — Liskov Substitution, Interface Segregation, Dependency Inversion).

[78] courses/csse2002/java-cohesion-and-coupling.pdf
[79] courses/csse2002/2026-04-02-refactoring.pdf
[80] courses/csse2002/2026-04-16-solid-principles-ii.pdf

Why SOLID?

Large software tends to become:

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

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

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

Single Responsibility Principle (SRP)

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

Closely related to java-cohesion-and-coupling[81]'s cohesion: cohesion is about how *similar* a class's functionality is, while SRP is about the *reasons for change*. A "responsibility" is a reason for change — if you can think of more than one motive to change a class, it has more than one responsibility.

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

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

This class has three reasons to change:

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

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

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

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

Two details of the split are worth noticing. The `gpa` field is gone: it was a *cached result* of one of the three responsibilities, so it leaves with the responsibility, and `GPACalculator.calculateGPA` returns a value instead of assigning one. And `Student` gains `getCourses()`, because the two extracted classes now need the data from outside — the responsibilities separate cleanly, but they do not become independent.

The same shape, stated more briefly:

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

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

Homework. Check whether SRP improves memory management *indirectly*, by improving overall design structure — think about object creation, immutability (see java-mutability[82]) and garbage collection.

Open-Closed Principle (OCP)

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

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

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

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

A class is kept closed for modification by information hiding (see java-encapsulation[83]).

Example: pluggable grading systems

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

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

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

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

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

```
        return totalPoints / courses.size();
    }

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

Three things carry the problem: the added `String gradeType` parameter, and the two string literals `"Letter"` and `"Numeric"`. They are every place that has to be *modified* the next time a grading system is added, in a class that was supposed to be finished. The class is open for extension only by editing it, which is exactly what "closed for modification" forbids. (A `gradeType` string also reintroduces the `Vehicle`-style type switch from java-refactoring[84].)

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

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

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

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

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

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

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

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

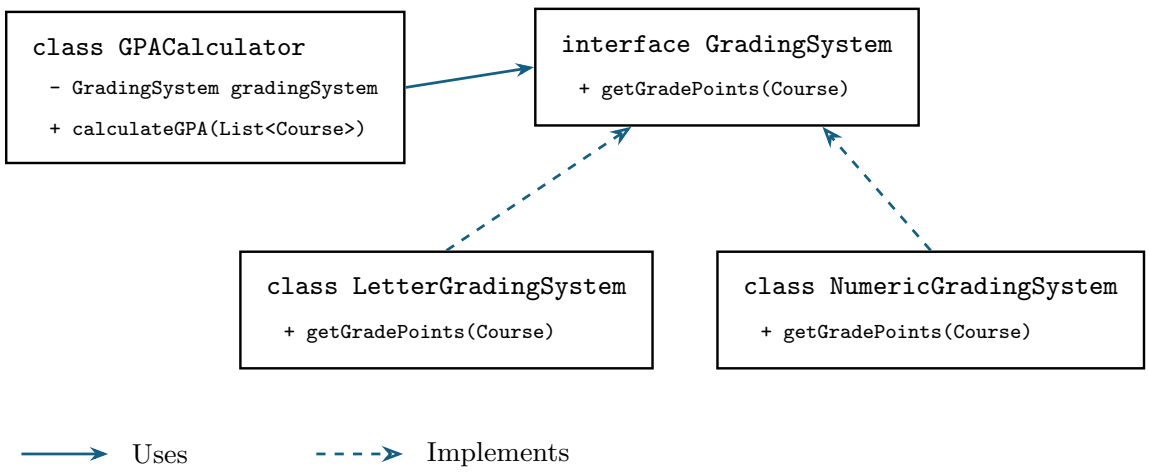


Figure 26: The open-closed refactoring of GPACalculator. GPACalculator uses only the GradingSystem interface; each concrete grading system implements it. A new scheme — pass/fail, say — is a fourth box on the bottom row with a fourth dashed arrow, and no existing box changes: that is 'open for extension, closed for modification' as a picture. Each label carries the Java keyword and members are listed as '+ m()'. A solid line with a solid head means 'uses', a dashed line with an open head means 'implements', and every arrow points from the depender to the thing depended upon. This is not UML, which would draw realisation as a hollow triangle.

What the refactoring bought:

- `GPACalculator` depends on the abstraction `GradingSystem` instead of calculating grade points itself — which makes it open for extension (new grading systems) but closed for modification.
- More grading systems can be added without changing the core logic of `GPACalculator`.
- The interface guarantees every grading system implements the same method (`getGradePoints()`), making it easy to switch between them.

- `main` demonstrates using either `LetterGradingSystem` or `NumericGradingSystem` by passing a different grading system to the `GPACalculator`.

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

Liskov Substitution Principle (LSP)

> Subclasses should be substitutable for their parent classes.

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

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

Example violation

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

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

Substitution with contracts

Even when a subclass overrides a method with a different implementation, it must still honour the parent's contract (see java-specification[85]):

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

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

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

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

Worked example

```
class One {
    /**
     * @require row != null && col != null && 0 < val && val < 100
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in either row or col
     */
```

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

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

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

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

Interface Segregation Principle (ISP)

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

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

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

This mirrors java-cohesion-and-coupling[86]'s stamp coupling at the interface level: despite each client using only a small subset of `Z`, every client depends on *all* of `Z` — so a change to any part of `Z` risks affecting (and forcing a recompile/redeploy of) every client, even ones that never used the changed part. Splitting `Z`'s surface into focused interfaces removes those incidental dependencies:

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

Dependency Inversion Principle (DIP)

> Depend upon abstractions. Do not depend upon concretions.

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

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

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

What makes a good dependency? (stability and exposure)

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

- **Stability** — how likely the dependency is to change. `ArrayList` is a good dependency because it's very unlikely to change; treat classes *you* write as unstable until proven otherwise, and generally assume interfaces are more stable than concrete classes (unless they're poorly designed and change often).

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

- **Exposure** — how much of the depending class is entangled with it. A dependency only used in a constructor is better than one used across every method, since less of the class would need to change if that dependency changed.

Two forms of DIP

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

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

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

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

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

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

Dependency Injection

Dependency Injection (DI) is a design pattern that implements DIP by injecting a class's dependencies into it (e.g. via the constructor) instead of having the class create them internally, promoting loose coupling (see java-cohesion-and-coupling[87]):

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

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

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

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

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

Summary

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

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

Java Specification

Introduced in 2026-03-12-object-oriented-programming-ii[88] (Lecture, Week 3). Extended in 2026-03-19-exceptions[89] and formalised later in java-hoare-logic[90]; pre/postconditions come back as class invariants in week8-tutorial-class-invariants[91] and as the substitution principle in week9-tutorial-substitution-principle-and-pre-post-conditions[92].

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
```

[88] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[89] courses/csse2002/2026-03-19-exceptions.pdf
[90] courses/csse2002/java-hoare-logic.pdf
[91] courses/csse2002/week8-tutorial-class-invariants.pdf
[92] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf

```
 * @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;

    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**.

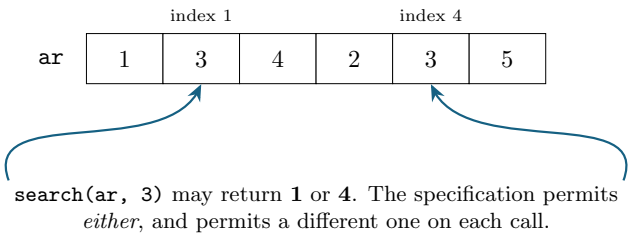


Figure 27: With the spec worded as 'returns an index i such that ar[i] == x', both cells holding 3 are legal answers, and nothing forbids the method returning a different one on each call. That is what makes the specification non-deterministic.

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[93]. 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:

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

```
/**
 * 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:

```
/**
 * 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:

```
/**
 * [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.

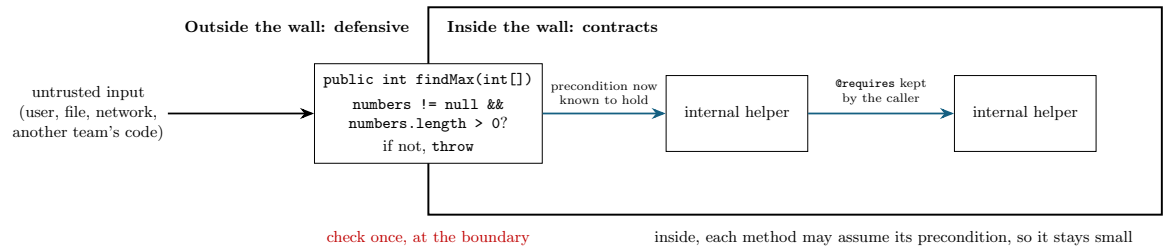


Figure 28: Defensive outside the wall, contracts inside it: validate defensively where untrusted input crosses into the system, and rely on contracts for the internal calls behind that boundary, so the check happens once rather than at every hop. The gate box is the findMax method worked through below.

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.

Java Testing

Introduced in 2026-03-26-testing[94] (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?

[94] courses/csse2002/2026-03-26-testing.pdf

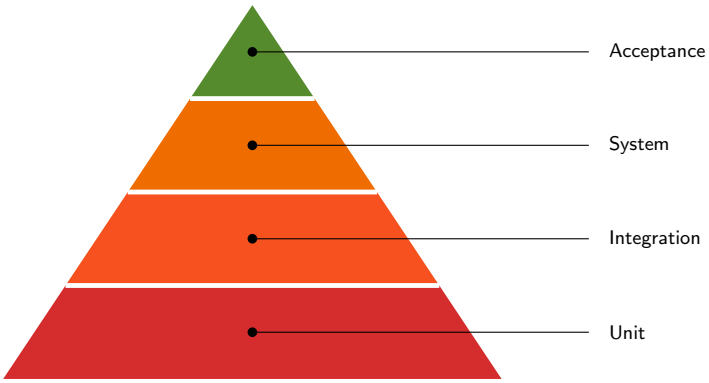


Figure 29: The four levels of testing as a pyramid. Unit tests form the wide base, acceptance tests the narrow apex. In the usual test-pyramid reading, the width of a band stands for how many tests of that kind a project has.

Test frameworks

Libraries that make writing and running tests easier. **xUnit** is the prevalent family — JUnit, NUnit, CPPUnit, PyUnit, … — and this course uses **JUnit 4** (http://junit.org/junit4); see java-junit[95].

Depending on how the software is structured, a unit-testing framework can also be driven for other kinds of automated testing. A caveat: *"unit testing" is sometimes incorrectly used to mean any automated test.*

Regression testing

During development, testing helps answer (at least) two questions:

- Does the new stuff work?
- Have we broken things which used to work — have we **regressed**?

Regression testing ensures that old features are still working as they should when 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[96]) — what

[95] courses/csse2002/java-junit.pdf
[96] courses/csse2002/java-specification.pdf

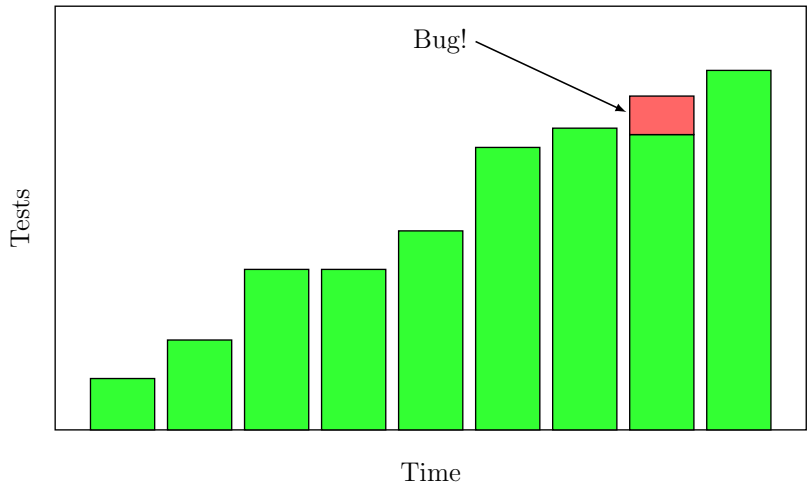


Figure 30: A regression (chart after browserstack.com/guide/regression-testing). Nine bars on *Tests*/*Time* axes: read each bar as a build, and its height as the size of the test suite growing as features are added. The red top slice on the second-to-last bar is the regression, and the arrow labelled Bug! points at it.

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

Black box testing works from the Javadoc alone. The running example is a bus, and the *specification is all you get*:

```
/**
 * Calculate how many passengers are currently on the bus.
 *
 * @returns The number of passengers on the bus.
 */
public int getCurrentPassengers() {...}

/**
 * Calculate the average number of passengers on the bus between stops.
 *
 * @returns The average passengers on the bus.
 */
public int getAverageCount() {...}
```

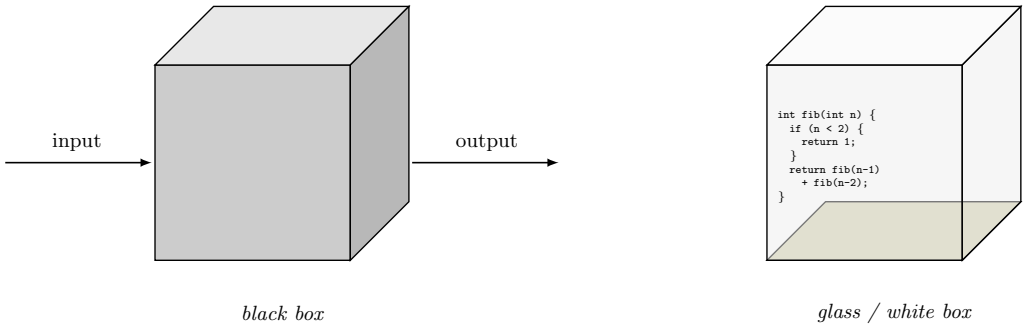


Figure 31: The two testing stances. Both are the same grey cube. The black box is opaque and shows only its input and output arrows. The glass box is translucent, so the implementation inside, a recursive `fib`, can be seen through its walls.

> **Note**
>
> The tag is written `@returns` here and `@return` in the specification tag table from the exceptions lecture. The course uses both spellings — see csse2002-diagram-notation[97], "Specification tag spelling".

Smoke tests — test the common-case functionality. Usually the *first* test to be performed ("can it run?"), used to determine whether further, more rigorous testing is required. The name comes from electronic hardware testing: plug in a new board, turn on the power, and if you see smoke coming from the board, turn the power off — you don't have to do any more testing. (From *Lessons Learned in Software Testing.*)

Boundary tests — investigate extremes and corner cases, i.e. where bugs often occur:

Input type	Try
Whole number	0, -1, minimum value, maximum value
Floating point	0, -1, `NaN`, infinity
Collection (array, list, set, etc.)	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. For `getCurrentPassengers()` on a bus whose maximum capacity is 80, build the table in two passes: first the classes, then the boundary tests for each class.

97 courses/csse2002/csse2002-diagram-notation.pdf

Class	Values	Boundary tests
Valid – Empty	0	-1, 0, 1
Valid – Normal	1 – 79 (any number within the range)	Covered by above and below
Valid – Full capacity	80	79, 80, 81
Invalid	–	-1, 81

Minimal boundary set: `-1, 0, 1, 79, 80, 81`. The "Normal" class contributes no probes of its own because its edges are already the outer probes of the classes on either side.

The black-box method

The method in six points:

1. Black box testing: no knowledge of implementation, only specification.
2. JUnit is a test framework.
3. Tests must not violate preconditions.
4. Start with smoke tests.
5. Expand test suite with boundary tests.
6. Stop when confidence is high.

Point 3: a test that breaks `@requires` proves nothing, because the method is free to do anything at all once its precondition is violated. Point 6: testing is never exhaustive, so you stop on judgement rather than on a completeness proof. The loop argument below makes the same admission concrete.

White box technique: code coverage

With tests *and* an implementation, code coverage measures how much of the code is actually tested: of all the different ways your program could run, how many are covered by your tests. There are three levels, in this order:

1. **Statement coverage** — each statement is executed.
2. **Branch coverage** — each branch is tested for the `true` case and the `false` case.
3. **Path coverage** — every 'path' through the code is traversed.

A worked example:

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

- **Statement coverage:** any odd positive value, e.g. `x = 1`, gives a partial run: `if (x > 0)`, `positives += 1` and the `if (x % 2 == 0)` test are executed while `evens += 1` is missed. `x = 2` alone executes every statement, since it is positive *and* even so both `if` bodies run.
- **Branch coverage:** `x = 2` makes both conditions `true` and `x = -1` makes both `false`, so `x = 2` and `x = -1` together give each condition both outcomes, `true, false`. Two tests.
- **Path coverage:** all four combinations of the two conditions, so four tests. The four inputs are in the table below.

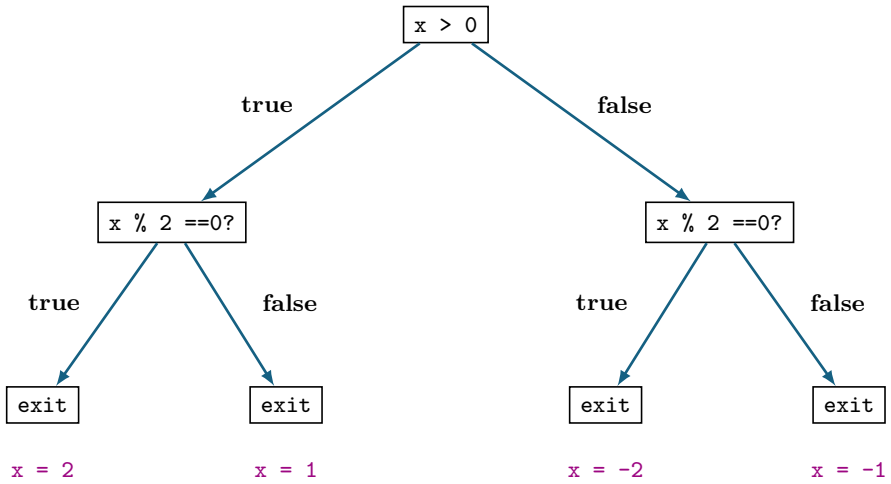


Figure 32: Path coverage for `register`. Each internal box is a condition, each teal arrow an outcome labelled true/false, each leaf one complete path to `exit` — four leaves, so four tests. The magenta input under each leaf is a value of `x` that drives that path.

Written out, the four paths and their inputs are:

Path	x
`x > 0 && x % 2 == 0`	2
`x > 0 && !(x % 2 == 0)`	1

Path	x
`!(x > 0) && x % 2 == 0`	-2
`!(x > 0) && !(x % 2 == 0)`	-1

The three levels run from weakest to strongest, and this exercise makes that concrete: the same `x = 2` that gives full *statement* coverage misses three of the four paths. That is why "100% statement coverage" is a weak claim.

Loops make exhaustive path coverage infeasible

```
for (int i = 0; i < 100; i++) {
    if (f(i, k)) {
        j++;
    }
}
```

There are 2^{100} paths through this program — 1,267,650,600,228,229,401,496,703,205,376 of them. At a billion tests per second that is 40.2 trillion years, roughly 2900 times the age of the universe.

So path coverage is **approximated**. The common approximation is that **2 or more iterations through a loop, or recursive calls, are equivalent**. This is *engineers' induction*: "One, two, three — that's good enough for me." For `for (init; test; iterate) { body }` the three cases to test are:

- **0 iterations:** `init`; `test` (**fails**); `nextStatement`;
- **1 iteration:** `init`; `test`; `body`; `iterate`; `test` (**fails**); `nextStatement`;
- **2 iterations:** `init`; `test`; `body`; `iterate`; `test`; `body`; `iterate`; `test` (**fails**); `nextStatement`;

Using JUnit for white-box testing

Design tests based on internal logic. You try to cover **branches**, **loops** and **edge cases**, and you may aim for a code coverage target (e.g. 100%). A worked white-box example — `DiscountCalculator`, the five tests that cover it, and its branch tree — is in java-junit[98].

[98] courses/csse2002/java-junit.pdf

Test Driven Development (TDD)

The TDD approach originates from the Agile manifesto principles and Extreme Programming. In TDD, developers create small test cases for every feature based on their initial understanding. The primary intention is to **modify or write new code only if the tests fail** — this prevents duplication of test scripts. Use the tests to determine when code is "ready".

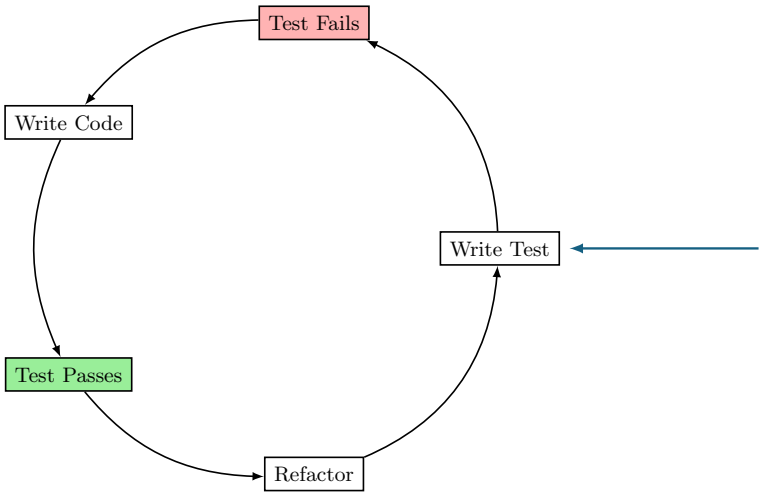


Figure 33: The TDD cycle. Read it counter-clockwise: write a test, watch it fail (pink), write code, watch it pass (green), refactor, and go round again. The teal arrow on the right points at *Write Test* and marks the entry point of the loop: the test comes before the code.

The *Refactor* step connects to java-refactoring[99]: the green test suite is what makes it safe to restructure code without changing its behaviour.

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