CSSE2002 — Week 13 Notes

Table of contents

Course Summary and Exam Preparation **1**
- What is not examinable . . . 1
- Exam structure . . . 2
- Practice questions by week . . . 3
- What a real paper looked like . . . 22
- Closing . . . 24

Reference material **24**
- Java Abstraction . . . 24
- Java Casting . . . 28
- Java Cohesion and Coupling . . . 31
- Java Collections Framework . . . 37
- Java Encapsulation . . . 40
- Java Events . . . 49
- Java Exceptions . . . 52
- Java Generics . . . 59
- Java Hoare Logic . . . 71
- Java Inheritance . . . 79
- Java I/O . . . 83
- Java JUnit . . . 96
- Java Lambdas and Streams . . . 103
- Java Loop Invariants . . . 106
- Java Mutability . . . 114
- Java Polymorphism . . . 118
- Java Primitive and Reference Types . . . 121
- Java SOLID Principles . . . 124
- Java Specification . . . 136
- Java Testing . . . 147
- Version Control . . . 154

Course Summary and Exam Preparation

See csse2002[1] for course logistics. A walk back through the whole course with 52 practice questions attached, plus the shape of the exam.

The practice questions are designed to get you thinking about the content. The actual exam questions will mostly be more challenging.

What is not examinable

Four topics are **not examinable**:

Topic	Where it was taught
Week 03 — Version Control	supplementary lecture — java-version-control[2]
Week 10 — *No lecture*	—
Week 11 — Event Driven Programming & GUIs	the events half of 2026-05-14-lambdas-streams-and-events[3]
Week 12 — Correct Programming	2026-05-21-correct-programming[4]

So java-hoare-logic[5], java-loop-invariants[6] and java-events[7] are course content but not exam content. Lambdas *are* examinable; the events half of the same week is not. Week 12 still gets a practice question, Q52.

Exam structure

15 questions, 100 marks total, **120 minutes plus 10 minutes planning time**.

Question	Marks	Type
Q1–Q8	16 total	MCQ
Q9	12	short/long answer
Q10	10	short/long answer
Q11	8	short/long answer
Q12	14	short/long answer

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-version-control.pdf
[3] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[4] courses/csse2002/2026-05-21-correct-programming.pdf
[5] courses/csse2002/java-hoare-logic.pdf
[6] courses/csse2002/java-loop-invariants.pdf
[7] courses/csse2002/java-events.pdf

Question	Marks	Type
Q13	15	short/long answer
Q14	10	short/long answer
Q15	15	short/long answer

Marks for each question are printed on the paper. Use the planning time to decide the order to attempt questions in, and **note any assumptions** you make.

Conditions, from the cover sheet

The cover sheet of the paper sets the conditions. The main one:

One A4 sheet of handwritten or typed notes double sided is permitted.

The rest of the cover sheet:

- Specified written or printed material permitted.
- **Casio FX82 series or UQ approved and labelled calculator only.** No other electronic aids — explicitly no laptops, no phones.
- During planning time, students are encouraged to review and plan responses to the exam questions.
- Supplied: additional exam materials (answer booklets, rough paper) **on request**, and **1 × Gradescope bubble sheet** (for the MCQ block).
- *"If you believe there is missing or incorrect information impacting your ability to answer any question, please state this when writing your answer."* Marks are awarded out of 100.
- Two blank pages are added at the end. If you use them, clearly mark which question you are answering, **and** indicate in the designated answer space that your response continues on the blank pages.

What to expect: what does the code do · what code completes this method · explain concepts · apply concepts · evaluate code · write code.

Preparation advice: review course material — lectures *and especially applied classes*; practise writing programs **by hand** (try the Ed Lessons problem-set questions); sample exams are roughly similar in structure and style, with **practice exam 3 the most similar**; past exams vary in format but have similar question *types*; and check out the **CodeCamps** for supplementary material.

Practice questions by week

Week 1 — Java basics

Java is a general purpose programming language: **compiled, statically typed, object-oriented, memory safe**. Week 1 topics: arithmetic, logical operations, control flow. See java-primitive-and-reference-types[8].

1. What is the difference between a statically typed and a dynamically typed language?
2. What is the output? — `int x = 0; while (x < 10) { int x = x + 1; } System.out.println(x);`
3. What is the output? — `int x = 6 * 2 - 4; x++; int y = (x % 2) + 3; x *= 3; y += Math.pow(y, 2); System.out.println(x - y);`
4. What is the output? — `int x = 3 * 2; while (x < 24) { x *= 2; } int k = switch (x) { case 24 -> x * 3; case 48 -> x * 2; default -> x * 1; }; System.out.println(k);`

Week 1 — Collections and strings

A `String` is a **reference** datatype constructed from a string literal, e.g. `"hello, world"`. **Double quotes create strings; single quotes create chars.**

Helpful `String` methods (not exhaustive):

Signature	Meaning
`int String.length()`	Return the length of the string.
`char String.charAt(int)`	Return the char at the given index.
`String String.substring(int, int)`	Return the substring between two indexes.
`String String.replace(char, char)`	Replace occurrences of char with char.
`int String.indexOf(char)`	Return the index of a given char.

For `substring`, **`startIndex` is inclusive, `endIndex` is exclusive.** Q5 needs this.

Arrays are ordered, fixed-length, mutable sequences of homogeneous items, accessed with `[]`, sized with `.length`. Collections covered: **Stack, List, Set, Map** — see java-collections-framework[9].

5. What is the output? — `String name = "CSSE2002"; int times = name.length() / 4; String result = ""; for (int i = 0; i <= times; i++) { result += name.charAt(i * 2); } result += name.substring(2, 5);`

[8] courses/csse2002/java-primitive-and-reference-types.pdf
[9] courses/csse2002/java-collections-framework.pdf

6. Write `contains(int[], int)` returning `true` if the array contains the value.
7. What is the main difference between an array and a list?
8. What are two differences between a list and a set?
9. When would you use a map instead of a set?

> **Q6 — answer**
>
> ```
> boolean contains(int[] array, int value) {
>     for (int num : array) {
>         if (num == value) {
>             return true;
>         }
>     }
>     return false;
> }
> ```
>
> The enhanced-for body is the answer. The `public static` modifiers are not part of what was asked.

Weeks 2–3 — OOP

Classes, objects, and the four principles: **inheritance, polymorphism, encapsulation, abstraction**. The internal state of a class must be protected to avoid leakage. In Java the access modifiers that help are `private`, `public`, `protected`, and **default — no keyword required**. See java-encapsulation[10], java-inheritance[11], java-polymorphism[12], java-abstraction[13].

10. Explain the relationship between classes and objects.
11. What are the *two* ways the `BankAccount` class fails to encapsulate its internal state? *(a `public int balance` field, and `getTransactions()` returning the mutable internal `List<Transaction>` — see java-mutability[14])*
12. When can you assume a class invariant is true?
13. When is a class invariant allowed to be false?
14. What differences are there between superclasses and interfaces?
15. Why would you define a superclass and not an interface?
16. What is the purpose of overriding a method?

    Class invariants must be true after instance construction and after execution of each method invocation. For Q12 and Q13: an invariant must be established

[10] courses/csse2002/java-encapsulation.pdf
[11] courses/csse2002/java-inheritance.pdf
[12] courses/csse2002/java-polymorphism.pdf
[13] courses/csse2002/java-abstraction.pdf
[14] courses/csse2002/java-mutability.pdf

after the class constructor; it may be assumed as a precondition of each method (**excluding the constructor**); and it must be established after each method call. See week8-tutorial-class-invariants[15].

Q17–Q18 — `super` and which field gets written

```
class A {
    private int x = 0;
    public int m() { return  x = 1; }
}
class B extends A {
    private int y = 0;
    public int m() { return  y = 2; }
}
class C extends A {
    private int z = 0;
    public int m() { return super.m(); }
}

A a1 = new B();
A a2 = new C();
a1.m();
System.out.println(a1.m());
System.out.println(a2.m());
```

17. What is the value of `x` after calling `a1.m()`?
18. What is the value of `x` after calling `a2.m()`?

> **Working**
>
> - **Q17: `x` is still `0`.** `a1`'s actual type is `B`, so `a1.m()` runs `B.m()`, which assigns to `B`'s own `y`. `A`'s `x` is untouched. `B` cannot see it anyway, because it is `private`.
> - **Q18: `x` is `1`.** `a2`'s actual type is `C`, whose `m()` is `return super.m()`, i.e. `A.m()`, which performs `x = 1`.
>
> The two `println` lines therefore print `2` then `1`.

Q19–Q21 — the Batman overload/override problems

This is the same two-phase rule as the week-4 applied class: **the overload is chosen at compile time from the static types, and only then is the override resolved at run**

[15] courses/csse2002/week8-tutorial-class-invariants.pdf

time. See java-casting[16] and week4-tutorial-method-dispatch-and-casting[17].

```
class Enemy {}
class SuperVillain extends Enemy {}

class BruceWayne {
    public void kapow(Object enemy)       { System.out.println("Smash");  }
    public void bam(SuperVillain enemy)   { System.out.println("Whoosh"); }
}
class Batman extends BruceWayne {
    public void kapow(SuperVillain enemy) { System.out.println("Crunch"); }
    public void bam(Enemy enemy)          { System.out.println("Blam");   }
}
class GeorgeClooney extends Batman {
    public void kapow(Enemy enemy)        { System.out.println("Whack");  }
    public void bam(Enemy enemy)          { System.out.println("Oof");    }
}
```

The call under test is the same in all three questions:

```
BruceWayne manBat  = new Batman();
Enemy      poisonIvy = new SuperVillain();
manBat.kapow(poisonIvy);
```

Q19 — as written above. **`kapow(Object)` is locked in.** The static type of `manBat` is `BruceWayne`, which declares exactly one `kapow`, so that signature is fixed at compile time. `Batman.kapow(SuperVillain)` is an *overload*, not an override, so nothing displaces `BruceWayne.kapow(Object)` at run time. The output is `Smash`.

Q20 (homework) — the parameter type in `Batman.kapow(SuperVillain enemy)` changes from `SuperVillain` to **`Object`**. So `Batman` now declares `kapow(Object)`.

> **i** Working
>
> Compile time is unchanged: `kapow(Object)` is still the only signature visible through a `BruceWayne` variable, so it is still locked in. But `Batman.kapow(Object)` is now a genuine **override** of `BruceWayne.kapow(Object)`, so run-time dispatch finds it. Output: **`Crunch`**.

Q21 (homework) — the class header changes to `class BruceWayne extends Hero {`. The new superclass is:

[16] courses/csse2002/java-casting.pdf
[17] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf

```
class Hero {
    public void kapow(Enemy enemy) { System.out.println("kapow"); }
    public void bam(Enemy enemy)   { System.out.println("bam");   }
}
```

`kapow(Enemy)` is locked in.

> **Working**
>
> Through a `BruceWayne` variable there are now *two* applicable `kapow` overloads — `kapow(Object)` declared on `BruceWayne`, and `kapow(Enemy)` inherited from `Hero`. The argument's **static** type is `Enemy`, so the more specific `kapow(Enemy)` wins the compile-time choice. At run time the actual type is `Batman`, which declares `kapow(SuperVillain)` — not an override of `kapow(Enemy)` — so nothing displaces `Hero`'s version. The output is `kapow`.
> The moral of the three together: **adding a superclass changed the answer without touching a single line of the calling code.**

CodeCamp — `BookShelf` / `BargainBin`

Same idea, different names:

```
class Book {}
class FictionBook extends Book {}
class CrimeBook   extends FictionBook {}
class RomanceBook extends FictionBook {}
class TextBook    extends Book {}

class BookShelf {
    int cost(Book book)           { return 20;  }
    int cost(TextBook textBook)   { return 200; }
    int cost(CrimeBook crimeBook) { return 30;  }
}
class BargainBin extends BookShelf {
    int cost(RomanceBook romanceBook) { return 2; }
}

public static void main(String[] args) {
    BookShelf shelf = new BargainBin();
    Book theNotebook = new RomanceBook();
    System.out.println(shelf.cost(theNotebook));
}
```

Working

20. The static type of `theNotebook` is `Book`, so the only applicable overload in `BookShelf` is `cost(Book)`. `BargainBin.cost(RomanceBook)` is an overload the compiler never even considers, because `RomanceBook` is not the argument's *apparent* type.
The **2025 Sem 2 sample exam asks exactly this as Q1(d) and answers 20** — and then, as Q1(e), changes both declarations to their concrete types:

```
BargainBin shelf = new BargainBin();
RomanceBook theNotebook = new RomanceBook();
System.out.println(shelf.cost(theNotebook));   // 2
```

The same kinds of object, declared with different apparent types — a different answer.

Other question types

Given a base class and an interface, you may be asked to create subclasses that implement the interface.

Week 4 — Exceptions and specification

Specification covers **Javadoc, restrictiveness, generality, formality, contracts, defensive programming**. See java-specification[18] and java-exceptions[19].

22. Suggest ways to improve the specification of `indexOf` in terms of clarity, restrictiveness and generality:

```
/**
 * For arrays less than length 5,
 * iterates through the array and if a number matching the given number is
  ↪  found, returns that index.
 */
public static int indexOf(int[] array, int number) {}
```

(The obvious targets: the arbitrary "less than length 5" restriction, and the silence about what happens when nothing matches.)

23. Same question, different method:

[18] courses/csse2002/java-specification.pdf
[19] courses/csse2002/java-exceptions.pdf

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

24. Rewrite `concat` following defensive programming techniques:

```java
/**
 * Returns the concatenation of two non-null strings.
 *
 * @param first  the left-hand string (must not be null)
 * @param second the right-hand string (must not be null)
 * @return {first} followed by {second}
 * @requiers first != null && second != null
 * @ensures result.equals(first + second)
 */
public static String concat(String first, String second) {
    return first + second;
}
```

`@requiers` is a misspelling of `@requires`. The `{first}` / `{second}` brace notation for "the value of this parameter" appears here and nowhere else in the course.

25. What is printed? — `try { print("A"); throw new RuntimeException(); } catch (Exception e) { print("B"); } finally { print("C"); } print("D");` Answer: `ABCD`. The `catch` swallows the exception, so `finally` runs and control then reaches the statement after the `try`.

Week 5 — Testing and JUnit

Black box · White box · Test Driven Development. See java-testing[20], java-junit[21], week7-tutorial-black-box-and-glass-box-testing[22].

- **Blackbox testing.** *Smoke tests* — does it perform its functionality in the normal use case? *Boundary tests* — boundary inputs or edge cases which often are poorly handled.
- **Glass/White box testing** is used when the software under test is known, and lets you measure **coverage: statement, branch, path**. **Path coverage for loops uses engineers' induction — 0, 1 and 2 times through the loop.**

26. Provide a set of black box test cases for `even`, justifying each.

[20] courses/csse2002/java-testing.pdf
[21] courses/csse2002/java-junit.pdf
[22] courses/csse2002/week7-tutorial-black-box-and-glass-box-testing.pdf

```java
/**
 * Determines whether a given number is even
 * @param number the integer to be evaluated.
 * @return true if the number is an even number,
 *         false otherwise.
 * @throws NegativeNumberException
 *         if number is negative.
 */
public static boolean even (int number) {}
```

27. Write a JUnit test asserting `factorial(5) == 120`. Answer:

```java
public void testFactorialOfFive() {
    int result =  factorial(5);
    assertEquals("Factorial of 5 should be 120", 120, result);
}
```

28. Explain the process of test-driven development.

29. What is the difference between black and glass box testing?
30. Why is code coverage an important concept?

Q31–Q33 — coverage of `register`

Q31 and Q32 are the same question on the same code.

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

The line `evens += 1` is missing its `;`.

Q31 / Q32. Out of statement, branch and path coverage, what is the maximum level of coverage achieved by `x = 2`?

Q33. Provide a minimal set of inputs that achieve **path** coverage for `register`. Justify.

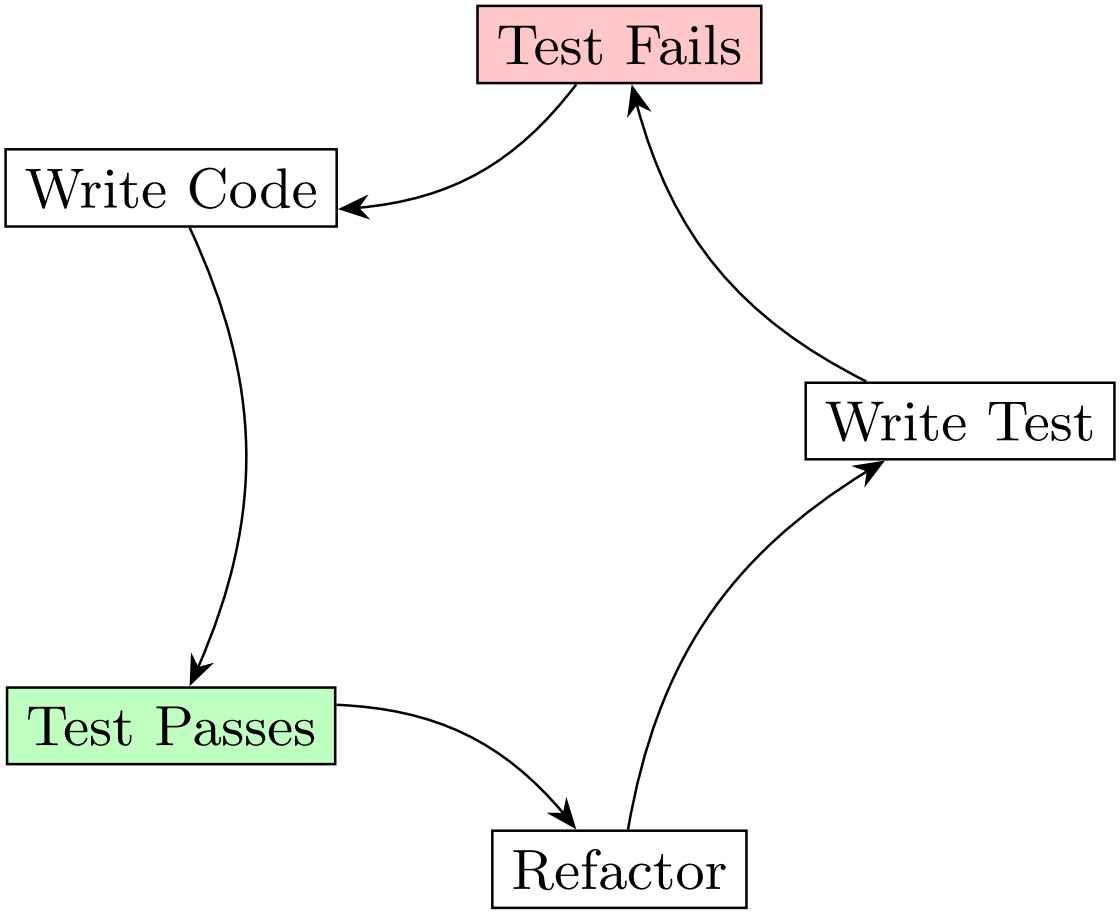


Figure 1: The TDD cycle: write the test first, watch it fail, write just enough code to pass, then refactor and go round again. The failing state is pink and the passing state is green.

Working

Q31/Q32. `x = 2` makes both conditions true, so every statement in the method runs — **statement coverage**, and no further. Neither `if`'s false branch is ever taken, so branch coverage (and therefore path coverage) is not reached.
Q33. The two `if`s are independent, so there are four paths. One input per path:

Input	`x > 0`	`x % 2 == 0`	Path
`2`	T	T	both counters incremented
`1`	T	F	positives only
`-2`	F	T	evens only
`-1`	F	F	neither

Four inputs, and four is minimal — one per path.

Q34 — path coverage of `range`, worked in full

This question reappears verbatim as **Question 6 of the 2025 Sem 2 sample exam**, so it is worth knowing cold.

```
/**
 * @param array An array of numbers to find the range of.
 * @requires array != null
 * @ensures if array.length > 0 then \result == max(array) - min(array)
 *          otherwise \result == 1
 */
public static int range(int[] array) {
    int min = Integer.MAX_VALUE;
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < array.length; i++) {
        if (array[i] < min) {
            min = array[i];
        }
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max - min;
}
```

> Provide a minimal set of inputs to the `range` method that achieves path coverage. Recall that **path coverage for loops only requires executing the loop 0, 1, and 2 times.**

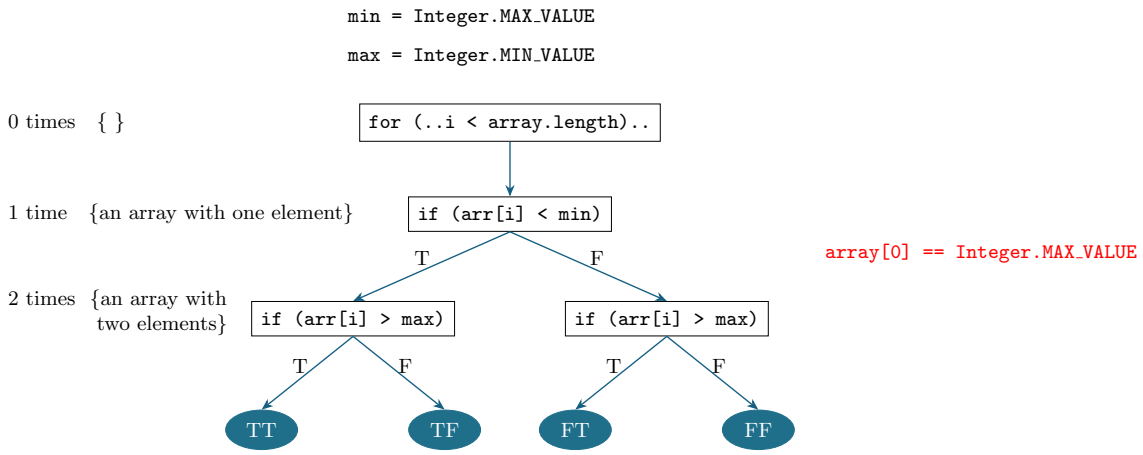


Figure 2: The decision tree for one iteration of the `range` loop, together with the initial values of `min` and `max` that decide which leaves are reachable. Each leaf is one path through the pair of `if`s. The red condition `array[0] == Integer.MAX_VALUE` names the only value that can take the first `if`'s **F** branch, because `min` starts at `Integer.MAX_VALUE`. The labels on the left give the array needed to go 0, 1 or 2 times through the loop.

Zero iterations. The loop is not entered: `{ }`.

One iteration — four leaves, one impossible:

Path	Input
`F F`	*Impossible*
`F T`	`[Integer.MAX_VALUE]`
`T F`	`[Integer.MIN_VALUE]`
`T T`	`[1]`

`FF` needs `array[0] == Integer.MAX_VALUE` (to fail `< min`) **and** `array[0] == Integer.MIN_VALUE` (to fail `> max`) at once.

Two iterations — sixteen combinations, of which nine are impossible. Entering the second iteration along the `F` branch, `array[0]` was `Integer.MAX_VALUE`, so `max` is no longer `Integer.MIN_VALUE`: it has become **`Integer.MAX_VALUE`**, and `min` is still `Integer.MAX_VALUE`. That is what makes the `FT**` rows below come out as they do.

Full path	Input
`F F F F`	*Impossible*
`F F F T`	*Impossible*

Full path	Input
`F F T F`	*Impossible*
`F F T T`	*Impossible*
`F T F F`	`[Integer.MAX_VALUE, Integer.MAX_VALUE]`
`F T F T`	*Impossible*
`F T T F`	`[Integer.MAX_VALUE, 1]`
`F T T T`	*Impossible*
`T F F F`	`[Integer.MIN_VALUE, Integer.MIN_VALUE]`
`T F F T`	`[Integer.MIN_VALUE, Integer.MAX_VALUE]`
`T F T F`	*Impossible*
`T F T T`	*Impossible*
`T T F F`	`[1,1]`
`T T F T`	`[1,2]` or `[1, Integer.MAX_VALUE]`
`T T T F`	`[2,1]`
`T T T T`	*Impossible*

The whole `FF**` block dies because `FF` was already impossible on one element. `TTTT` dies because a second element cannot be simultaneously below the new `min` and above the new `max` once both have been set to the first element. **Eleven inputs in total** — the empty array, three one-element arrays, and seven two-element arrays.

Week 6 — Cohesion and coupling

35. What is meant by a class having high cohesion? Why might it be desirable?
36. Explain the meaning of coupling and give an example of the benefits of low coupling.

The contrast is low cohesion with high coupling against high cohesion with low coupling. See java-cohesion-and-coupling[23] and week10-tutorial-coupling-and-cohesion[24].

Weeks 6–7 — SOLID principles

Applying SOLID principles helps to prevent rigid and fragile software.

- **S**ingle Responsibility — there should never be more than one reason for a class to change
- **O**pen-closed — components should be open for extension but closed for modification
- **L**iskov Substitution — subclasses should be substitutable for their parent classes
- **I**nterface Segregation — shrink interfaces to minimise incidental dependencies
- **D**ependency Inversion — depend on abstractions, not implementations

[23] courses/csse2002/java-cohesion-and-coupling.pdf
[24] courses/csse2002/week10-tutorial-coupling-and-cohesion.pdf

37. What is a "responsibility" in the context of the single responsibility principle?
38. What is prevented by applying the interface segregation principle?
39. Which SOLID principle is violated by `public int sum(ArrayList<Integer> numbers)`? Dependency inversion: it names the concrete `ArrayList` where the `List`/`Collection` abstraction would do.

Q40–Q41 — the substitution principle, drawn

Both questions go with the same figure, a way of reasoning about substitutability.

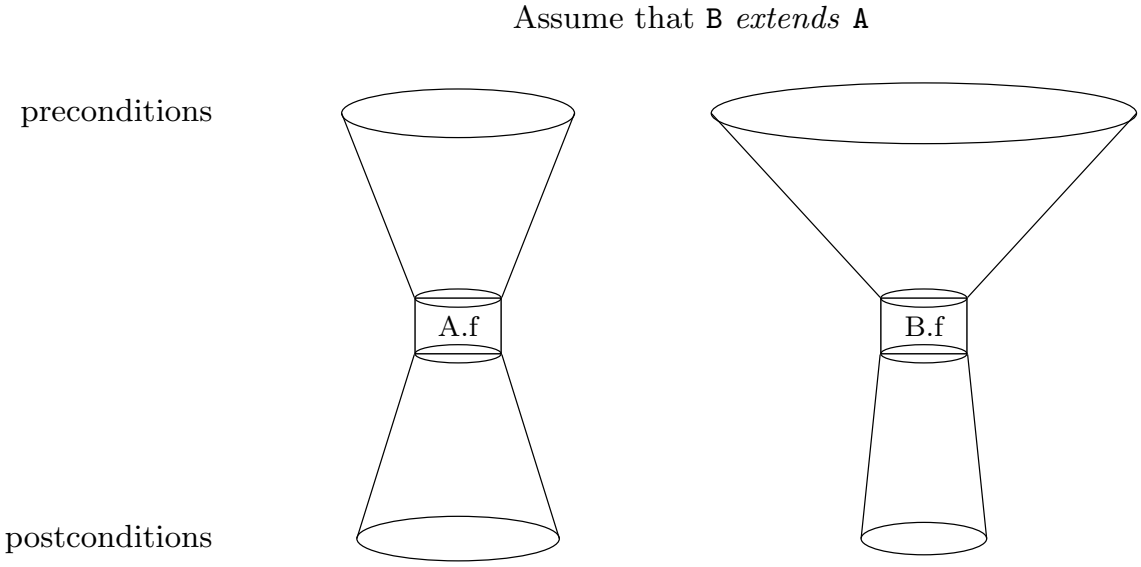


Figure 3: Pre/postcondition funnels. A method is the small box; the cone above it is the set of inputs it accepts (its precondition) and the cone below is the set of outcomes it may produce (its postcondition) — so a **wider** mouth means a **weaker** condition. `B.f` here has a wider top and a narrower bottom than `A.f`: a legal override.

40. Do the classes below satisfy the substitution principle?

```
class Document {
    /** Returns the next 128 chars of a document. */
    char[] nextChunk() {...}
}
class EfficientDocument extend Document {          // should be "extends"
    /** Returns the next 256 chars of a document. */
    @Override
    char[] nextChunk() {...}
}
```

41. Does class `B` satisfy the substitution principle with respect to class `A`?

```
class A {
    /**
     * @requires x > 0
     * @ensures \result > 'A' && \result < 'Z'
     */
    char process(int x);
}
class B extends A {
    /**
     * @requires x != 0
     * @ensures \result > 'J' && \result < 'N'
     */
    char process(int x);
}
```

Working

Q41 is a yes. `x > 0` implies `x != 0`, so `B` accepts everything `A` did and more — a **weakened** precondition, which is allowed. And `'J' < r < 'N'` implies `'A' < r < 'Z'`, so every outcome `B` may produce was already promised by `A` — a **strengthened** postcondition, also allowed. That is exactly the shape drawn: wider top mouth, narrower bottom mouth.

Q40 is the harder one. `EfficientDocument` promises 256 characters where `Document` promises 128. That is not a strengthening of "the next 128 chars" — it is a *different* promise, and a caller written against `Document` that assumes it receives at most 128 will break.

42. *(homework)* For each pair, is the second condition stronger, weaker, or neither?

First	Second	Verdict
`x > 0`	`x > -8`	**weaker**
`x != 0`	`x != 5`	**neither**
`(x != 0) && (x + 7) < 5`	`(x + 7) < 4`	**weaker**
`(x + y) > 5`	`(x > 2) && (y > 3)`	**stronger**

The rule: if everything in the first set is also in the second set, the second precondition is **weaker**, or less strict, than the first.

Q43–Q45 — dependency inversion

43. Explain how these classes violate the Dependency Inversion Principle:

```
class TerminalPrinter {
    /** Draws one black dot at row, column on the terminal. */
    public void typeDot(int row, int column) { /* ... */ }
}
class Square {
    private int width;
    private int height;

    public Square(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public void draw() {
        TerminalPrinter printer = new TerminalPrinter();
        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                printer.typeDot(x, y);
            }
        }
    }
}
```

The offending line is `TerminalPrinter printer = new TerminalPrinter();` inside `draw()`. `Square` both *names* and *constructs* the concrete `TerminalPrinter`, so it depends on an implementation rather than an abstraction, and there is no seam at which another printer could be substituted.

44. Refactor according to the Dependency Inversion Principle. Answer:

```
public interface Printer {
    void typeDot(int row, int column);
}

public class TerminalPrinter implements Printer {
    @Override
    public void typeDot(int row, int column) {
        System.out.println("Dot: " + row + " " + column);
    }
}

public class ScreenPrinter implements Printer {
    @Override
    public void typeDot(int row, int column) {
        System.out.println("…");
    }
}
```

```
public class Square {
    private int width;
    private int height;
    private Printer printer;

    public Square(int width, int height, Printer printer) {
        this.width = width;
        this.height = height;
        this.printer = printer;
    }

    public void draw() {
        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                printer.typeDot(x, y);
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Printer printer = new TerminalPrinter();
        Square square = new Square(5, 5, printer);
        square.draw();
    }
}
```

The dependency is *inverted*: both `Square` and `TerminalPrinter` now depend on `Printer`, and the choice of concrete printer moves out to `Main`.

45. *(homework)* Given that `PNGWriter` now exists —

```
class PNGWriter {
    /** Construct a new PNGWriter */
    public PNGWriter(String path) {...}
    /** Set the x, y pixel within an image to black. */
    public setPixel(int x, int y) {...}
}
```

— use your refactored classes to draw a square of width 8, height 10 to `"mysquare.png"`. You should not need to further modify your refactored classes but may make changes to `PNGWriter`. That constraint is the point of the exercise: if the Q44 refactor was right, the only change needed is to make `PNGWriter implement Printer`.

The substitution rule to remember: **a subclass may weaken preconditions and strengthen**

postconditions, never the reverse. See java-solid-principles[25], week9-tutorial-substitution-principle-and-pre-post-conditions[26], week11-tutorial-dependency-inversion[27].

Week 8 — Java I/O

Streams · Readers & Writers · Instance Saving & Loading · New I/O (`java.nio.file.*`). See java-io[28].

46. Write a method `public int count(char lookFor, Path file)` that counts the number of times a character appears in a file. An answer, taking a `File` in place of the `Path`:

```java
public int count(char lookFor, File file) {
    int count = 0;
    try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
        int currentChar;
        while ((currentChar = reader.read()) != -1) {
            if (currentChar == lookFor) {
                count++;
            }
        }
    } catch (IOException e) {
        System.err.println("Error reading the file: " + e.getMessage());
    }
    return count;
}
```

    Two things to notice: `read()` returns an **`int`**, not a `char`, precisely so that `-1` can signal end-of-stream; and the reader is opened in a **try-with-resources** header so it is closed on every path out.

47. What is the difference between an `InputStream` and a `BufferedInputStream`?

48. What is the purpose of `flush()`?

Week 9 — Generics

Why? · Generic Types · Bounded Generics and wildcards · Type Erasure. See java-generics[29].

49. What is the difference between `List<? extends T>` and `List<? super T>`? Answer:

[25] courses/csse2002/java-solid-principles.pdf
[26] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf
[27] courses/csse2002/week11-tutorial-dependency-inversion.pdf
[28] courses/csse2002/java-io.pdf
[29] courses/csse2002/java-generics.pdf

- **`List<? extends T>`** (upper bounded wildcard) — the list can hold elements of type `T` **or any subtype of** `T`; used when you **read** from the list.
- **`List<? super T>`** (lower bounded wildcard) — the list can hold elements of type `T` **or any supertype of** `T`.

50. Write a generic, type-safe `count` method over an array that returns how many times the given value occurs. The method must be type-safe and ensure that the given value is the same type as the array contents.

    Before, the non-generic version:

```java
public int count(int[] array, int value) {
    int count = 0;
    for (int element : array) {
        if (element == value) {
            count++;
        }
    }
    return count;
}
```

    After, the generic version:

```java
public <T> int count(T[] array, T value) {
    int count = 0;
    for (T element : array) {
        if (element.equals(value)) {
            count++;
        }
    }
    return count;
}
```

    The single substantive change is `==` becoming `.equals()`: once the element type is a reference type `T`, `==` compares identities, not values. Declaring both the array and the value as the *same* `T` is what makes it type-safe.

Week 11 — Lambdas

51. What is a functional interface? Contrast a plain single-method interface —

```java
public interface TestInterface {
    double getValue();
}
```

    — with an `@FunctionalInterface` that declares **two** abstract methods, which does not compile:

```java
@FunctionalInterface
public interface TestInterface {
    double getValue();
    double getAge();
}
```

See java-lambdas-and-streams[30] and week12-tutorial-lambdas-and-streams[31].

Week 12 — Correct programming *(not examinable)*

Correctness · Deriving Pre-conditions · Loop Invariants.

52. What does it mean for a program to be correct?

What a real paper looked like

The 2025 Semester 2 **sample exam** is worth reading alongside these questions: it is eight questions rather than fifteen, with no MCQ block, but the *question shapes* are the ones to expect. Its answer sheet opens with a warning — the model answers are "the most straight-forward solution but … far from the best solution. **They should not be used as examples of good programming practice.**"

Q	Marks	Shape	Answers given
1	10	Five 2-mark traces	`A.f(4)` → `3.46666667`, rounded `3.47`; `d(1234, 10)` → `{4, 3, 2, 1}` and `d(10, 2)` → `{0, 1, 0, 1}`; then the two `BookShelf`/`BargainBin` dispatch parts, `20` and `2`
2	4	Fill three holes in a merge-of-sorted-arrays method	= `array.length`, = `arrays[i].length`, = `i`

[30] courses/csse2002/java-lambdas-and-streams.pdf
[31] courses/csse2002/week12-tutorial-lambdas-and-streams.pdf

Q	Marks	Shape	Answers given
3	6	Recursion from a piecewise mathematical definition (Eulerian numbers)	precondition `n > 0 && k >= 0`, then a direct three-case recursive implementation
4	10	Code style — justify meaningful names, constants over literals, vertical whitespace; then find four violations and cite line numbers	missing Javadoc (line 1); `ArrayList` used instead of the `List` interface (1); magic number `256` (3); missing horizontal whitespace around operators (5, 7)
5	12	Liskov substitution reasoning, first abstractly on `A.f`/`B.f`, then on a `FilingSystem` hierarchy	see below
6	10	Black-box test table, then **minimal path coverage for `range`** — the same `range` method as Q34 above	not answered on the paper
7	14	Implement a specified interface (`JollyJumperSpec`) from its Javadoc	full class given
8	14	Implement a `Library` interface over a Dewey-Decimal call-number scheme	full class given

Two of those are worth naming: `A.f` computes a partial sum of the Leibniz series for , and `d(n, b)` returns the digits of `n` in base `b`, least significant first. The answer sheet gives only the numeric answers.

Question 5 is the clearest statement of the substitution rule anywhere in the course materials. Its three answers:

- Given `a.f(5)` **satisfies** `A.f`'s precondition, what do we know about `b.f(5)`? — *"We*

know that the call to `B.f` will satisfy the precondition … the precondition of a subclass must be weaker than or equal to the precondition of the superclass."

- Given `a.f(5) == 10` **satisfies** `A.f`'s postcondition, what about `b.f(5) == 10`? — *"May or may not. It is possible for the postcondition of `B.f` to be stronger than the postcondition of `A.f`, therefore `10` may not be a valid return value for `B.f`."*
- Given `a.f(5) == -10` **violates** `A.f`'s postcondition, what about `b.f(5) == -10`? — *"We know that the postcondition of `B.f` will be violated … the postcondition cannot be weakened."*

And parts (d)–(f), on a worked hierarchy: weakening `@requires name != null` to `true` **satisfies** LSP; modifying a postcondition "in an incompatible way" **violates** it; strengthening a precondition "to include an additional requirement" **violates** it.

Closing

> *"I hear and I forget. I read and I remember. I do and I understand."* — Confucius (according to popular belief)

Reference material

Java Abstraction

Introduced in 2026-03-05-object-oriented-programming-i[32] (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.

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

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

```
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:

```
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:

```
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.

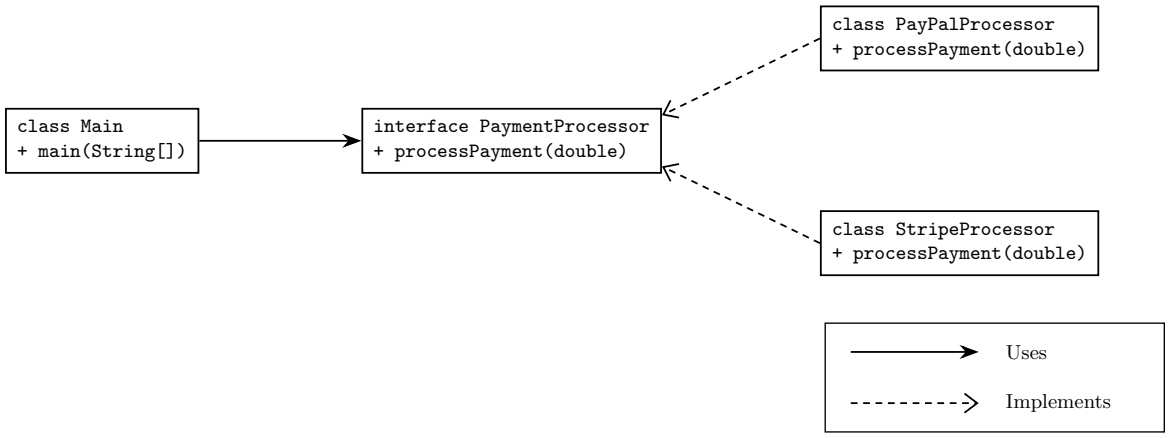


Figure 4: Main depends on the PaymentProcessor interface, not on either implementation - which is what makes swapping PayPal for Stripe a one-line change. Each rectangle is labelled with its Java keyword and lists its members as + lines. A solid arrow with a solid head means uses. A dashed arrow with 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.

Abstract classes

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

```java
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[33] (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.

```java
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");
```

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

```
    }
}

class Cat extends Animal {
    @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[34] — the `@Override` annotation and identical method signatures — and both pay off in Week 3 as java-polymorphism[35].

Java Casting

Introduced in 2026-03-12-object-oriented-programming-ii[36] (Lecture, Week 3). See also java-primitive-and-reference-types[37] and week4-tutorial-method-dispatch-and-casting[38].

Casting is the process of converting a variable from one data type to another. There are two kinds: **primitive casting**, between the built-in numeric types, and **reference casting**, up and down an inheritance hierarchy.

Primitive casting

Set as self-study (homework). The reading is w3schools' type casting page.

- **Widening** — converting a smaller type to a larger type size: `byte -> short -> char -> int -> long -> float -> double`.
- **Narrowing** — converting a larger type to a smaller size type: `double -> float -> long -> int -> char -> short -> byte`.

An example of a narrowing cast: `long floatToLong = (long) inputFloat;`.

[34] courses/csse2002/java-inheritance.pdf
[35] courses/csse2002/java-polymorphism.pdf
[36] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[37] courses/csse2002/java-primitive-and-reference-types.pdf
[38] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf

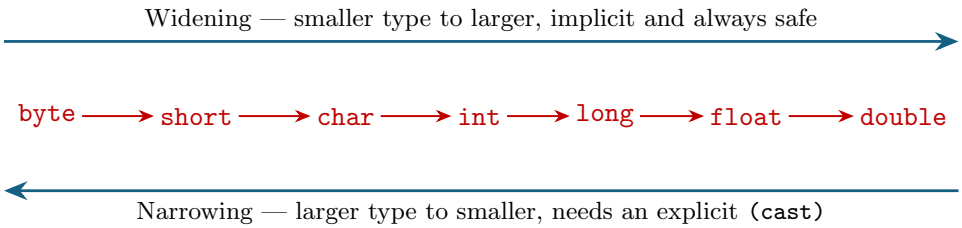


Figure 5: The two primitive conversion chains on a single axis: the narrowing chain is exactly the widening chain reversed. Widening is implicit; narrowing needs an explicit cast.

Reference casting

- **Upcasting** (implicit) — casting a subclass reference to a superclass reference.
- **Downcasting** (explicit) — casting a superclass reference back to a subclass reference.

The two directions are defined against a class hierarchy: up the tree is upcasting, down the tree is downcasting.

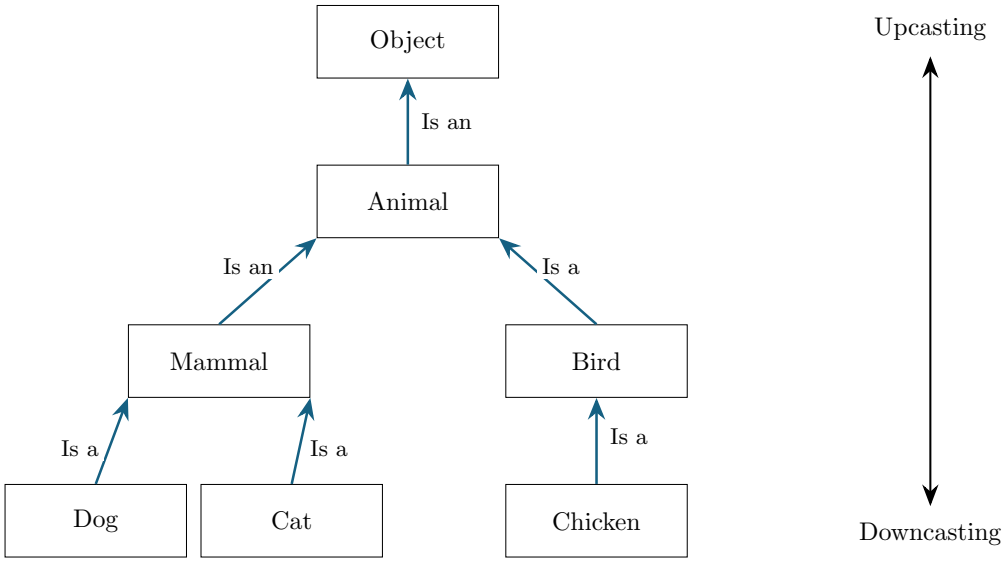


Figure 6: The class hierarchy that upcasting and downcasting move along, with 'Is a' / 'Is an' edge labels and the up/down casting axis at the right. Object sits at the root because every class implicitly extends it.

Upcasting

Upcasting is done implicitly and is always safe: every `Cat` really is an `Animal`, so no check and no cast syntax is needed. In the assignment below the `Cat` is treated as an `Animal`:

```
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"); }
}

Animal myAnimal = new Cat(); // upcasting: Cat is treated as an Animal
myAnimal.makeSound();        // Outputs: Cat meows
```

After the upcast, the *declared* type narrows what you may call, but the *object* still decides which body runs. That is java-polymorphism[39]'s runtime dispatch, and it is why the line above prints `Cat meows` rather than `Animal makes a sound`.

Downcasting

Before downcasting, you must ensure the object is an instance of the subclass using the `instanceof` operator. The cast is written explicitly, and it is the only way to reach members that exist on the subclass but not the superclass:

```
public class Cat extends Animal {
    @Override public void makeSound() { System.out.println("Cat meows"); }
    public void climb() { System.out.println("I can climb trees"); }
}

public class DownCasting {
    public static void main(String[] args) {
        Animal myAnimal = new Cat();     // upcasting

        if (myAnimal instanceof Cat) {
            Cat myCat = (Cat) myAnimal;  // Downcasting
            myCat.climb();               // access methods specific to cat
        }
    }
}
```

The `instanceof` guard is mandatory because an unguarded bad downcast is not a compile error: it compiles and throws `ClassCastException` at run time (see java-exceptions[40]).

Additional reading: upcasting and downcasting in Java.

[39] courses/csse2002/java-polymorphism.pdf
[40] courses/csse2002/java-exceptions.pdf

Every class implicitly extends `Object`

Worked question: what is the parent of class `Student`, a class with no `extends` clause at all?

```
public class Student {
    public void study() {
        System.out.println("CSSE2002");
    }
}
```

The answer is `Object`, the root of the Java class hierarchy and the top box of Figure 6. So every reference type can be upcast to `Object`, and a container declared over `Object` will hold anything. Getting a specific type back out of one means downcasting, which is the problem java-generics[41] exists to remove.

Java Cohesion and Coupling

Introduced in 2026-04-02-refactoring[42] (Lecture, Week 6, Part 2). Practised in week10-tutorial-coupling-and-cohesion[43].

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).

[41] courses/csse2002/java-generics.pdf
[42] courses/csse2002/2026-04-02-refactoring.pdf
[43] 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.

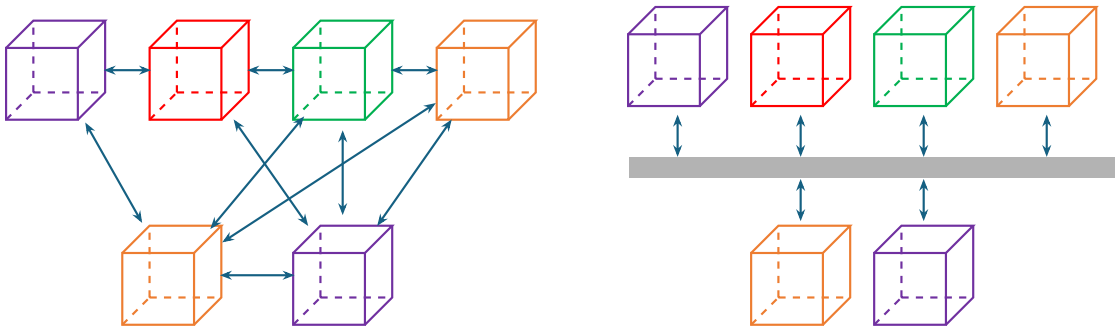


Figure 7: High versus low coupling. Left: every module talks directly to several others, so a change in any one can reach any other. Right: the same six modules each talk only to a shared interface — the grey bar — so nothing propagates further than that bar. A module is a wireframe cube, and its colour 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.

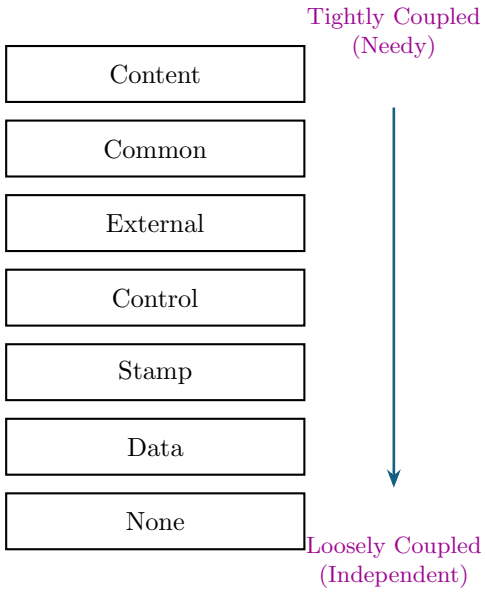


Figure 8: The coupling-levels ladder. Most tightly coupled at the top, least at the bottom, with the arrow running down from tightly coupled (needy) to loosely coupled (independent).

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[44]) 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).

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

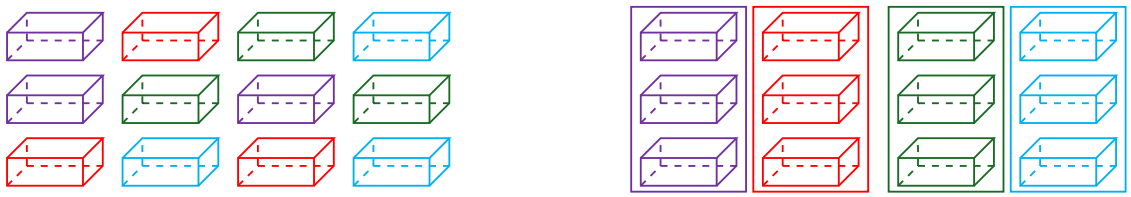


Figure 9: Low versus high cohesion. The same twelve elements in both panels: on the left the four kinds are scattered across the grid, on the right each kind is gathered into one module. Colour means kind only, and each high-cohesion group is enclosed by a rectangle in that group's own colour. This figure 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?

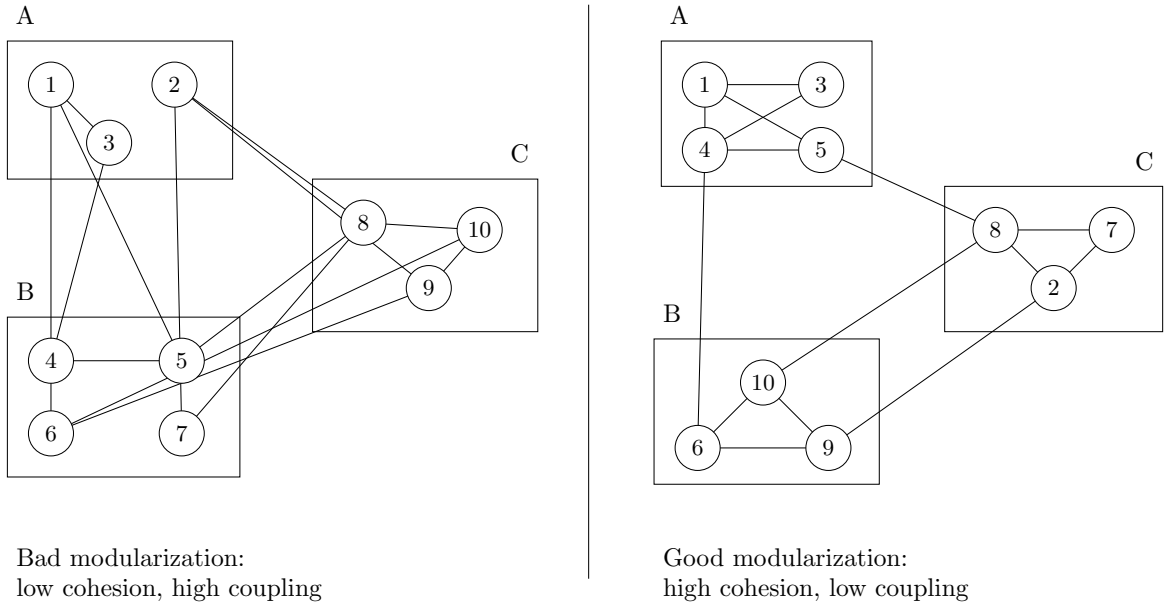
A `Car` class contains: Fuel, Steering, Speed, Route planner, Public holiday calculator.

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 10: 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[45] (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[46] 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`.

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

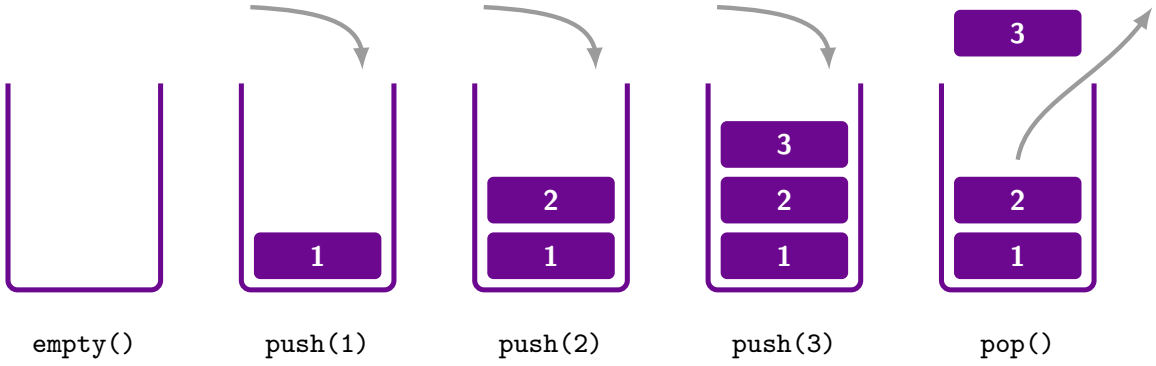


Figure 11: Last In, First Out: `push` always adds at the top and `pop` always removes from the top, so the last value in is the first value out. The stack is an open-topped container whose entries grow upward. An arrow *into* the container is a push and an arrow *out and away* is a pop. Each frame is 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

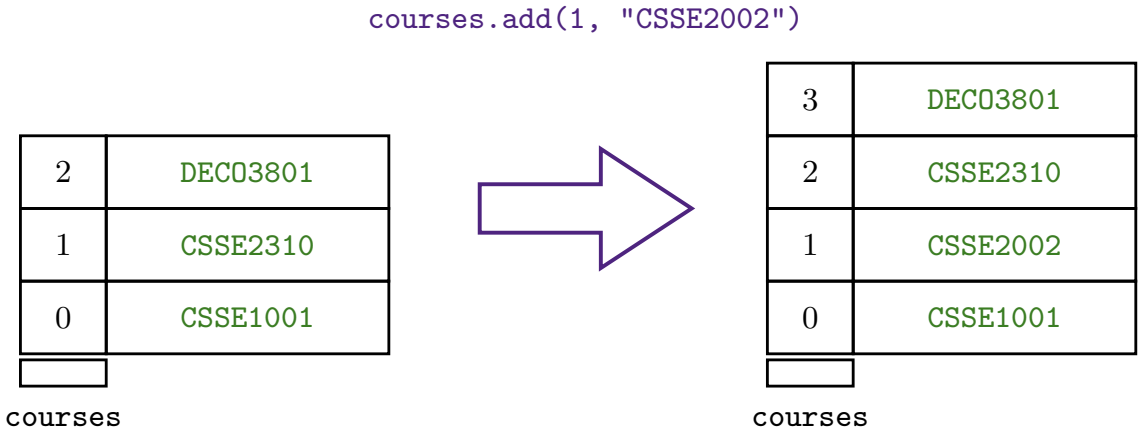


Figure 12: `add(index, e)` on a `List`: `e` lands *at* that index and every entry from there up renumbers, so the list gets longer without you writing a single shift. (`remove(index)` is the mirror image — see the removal step traced in 2026-02-26-java-basics-part-02-collections-and-strings[47].) The 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 Encapsulation

Introduced in 2026-03-05-object-oriented-programming-i[48] (Lecture, Week 2).

Classes and objects

- A **class** describes the contents of the objects that belong to it: an aggregate of data fields (properties) plus the operations (methods) defined on them.
- An **object** is an element (instance) of a class; objects have the behaviours of their class. The object is the actual component of a running program, while the class specifies how instances are created and how they behave.

Encapsulation

Encapsulation is the process of bundling code (methods/member functions) and data (member variables) together into a single unit — a class. It restricts direct access to some of an object's components, preventing the accidental modification of data.

The Java that box stands for:

```java
public class Person {
    private String name;
    private int age;
    private String email;
    private String[] phoneNumber;

    public Person(String name, int age, String email, String[] phoneNumber) {
        this.name = name;
        this.age = age;
        this.email = email;
```

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

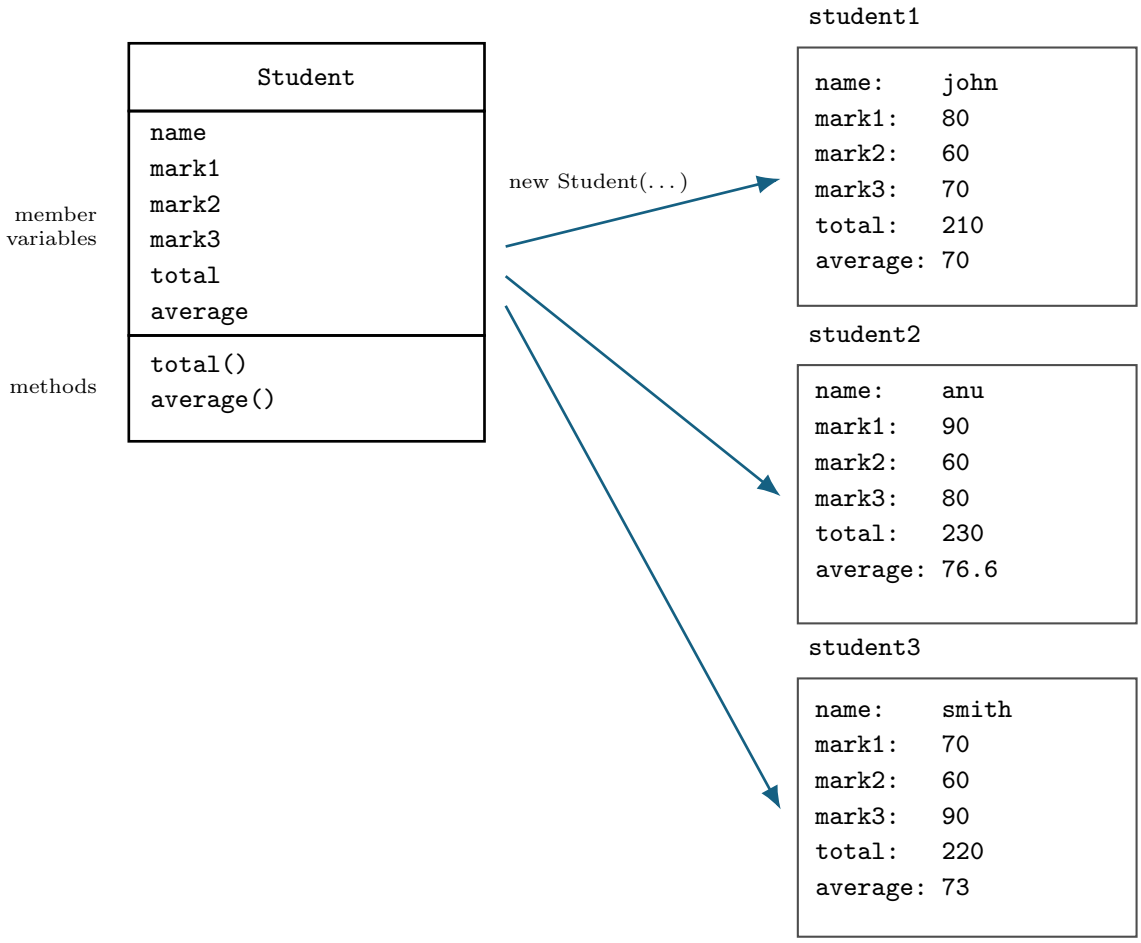


Figure 13: One Student class, three Student objects: the class fixes the member variables and methods, each object carries its own values for them. Each arrow is one new Student(…) call creating an object. anu's average is rounded to 76.6.

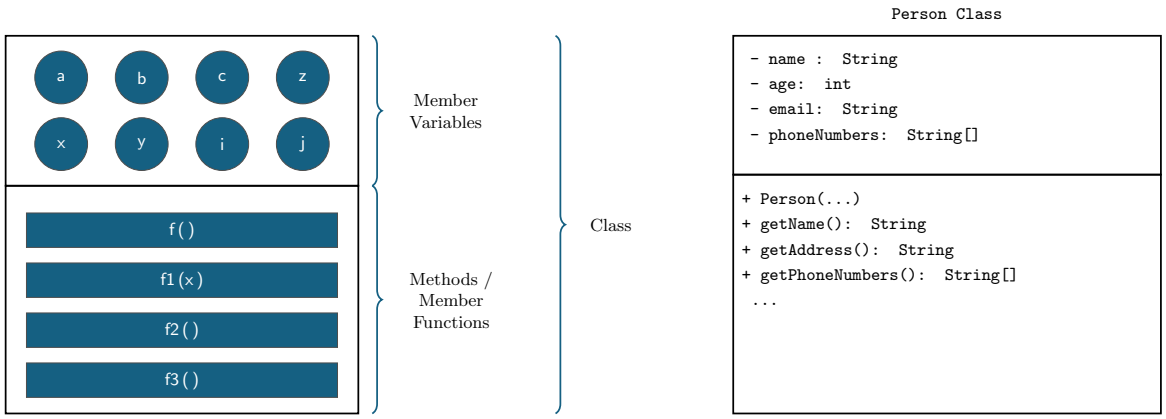


Figure 14: What a class is made of, and how it is written as a diagram. Left: a member-variable compartment above a methods compartment, the two braced together as the class. Right: the same idea as a two-compartment box, attributes above operations, with the class name above the box. The attribute list and the operation list in the box do not quite match (email against getAddress).

```
        this.phoneNumber = phoneNumber;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public String getEmail() { return email; }
    public String[] getPhoneNumber() { return phoneNumber; }
}
```

The box and the code do not match exactly: the box lists an `email` attribute but a `getAddress()` operation (the box has no `address` attribute and no `getEmail()` operation, while the code has `email` and `getEmail()` and no `address` at all), and the box spells the array field `phoneNumbers` where the code spells it `phoneNumber`. The intended mapping is the obvious one — one private attribute per `-` line, one public method per `+` line, the constructor in the operations compartment with no return type.

Constructors

A constructor is a special method invoked when an object of the class is created (via `new`):

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

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

    public static void main(String[] args) {
        Person person = new Person("John", 20);
        System.out.println("The name of the person is " + person.name);
    }
}
```

If a class has no constructor, the Java compiler automatically creates a **default constructor**. Take the same class with the constructor deleted and `new Person()` called instead — it still compiles, because the compiler supplies the no-argument constructor. `name` is never assigned, so the `println` would print `null`.

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

    public static void main(String[] args) {
        Person person = new Person();
        System.out.println("The name of the person is " + person.name);
    }
}
```

Notes on constructors:

- Constructors are invoked implicitly when objects are instantiated.
- The constructor's name MUST be the same as the class.
- A Java constructor must not have a return type.
- If a class does not have a constructor, the compiler creates a **default constructor**. This happens at compile time, not at run time: the compiler emits it into the class file.
- A constructor can be **overloaded** but can **not** be **overridden**.

Access modifiers

Access modifiers set the accessibility (visibility) of classes, interfaces, properties, methods, constructors, and data members:

Modifier	Accessible from
`private`	Only within the declaring class
`public`	Anywhere
`protected`	Same package, plus subclasses (see java-inheritance[49])
(default, no keyword)	Only within the same package

The same `BankAccount` under three of the four:

`private` — the field cannot be touched from outside the class, so this does not compile:

```
public class BankAccount {
    private double balance;
    private String accountNumber;
    private String accountHolder;
    private double interestRate;
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.balance = 1000.0;   // compile error
    }
}
```

```
java: balance has private access in Week02.OOP.AccessModifers.BankAccount
```

`public` — change `balance` to `public double balance;` and the assignment that would not compile above is now legal. Add `System.out.println("Balance: " + account.balance);` to `Main` and it prints `Balance: 1000.0`.

default (no keyword) — drop the modifier entirely (`double balance;`) and, because `Main` is in the *same package*, it also compiles and prints `Balance: 1000.0`. The difference from `public` only shows up across a package boundary.

`protected` is **homework**: read DigitalOcean's Java access modifiers tutorial. It is accessible within the same package *as well as* from subclasses. The subclass clause is what makes `protected` wider than default access, because it reaches subclasses in *other* packages too.

[49] courses/csse2002/java-inheritance.pdf

Getters and setters

Getter and setter methods provide controlled access to an object's private fields, keeping its internal representation hidden from the outside world:

- **Getters** (accessors) retrieve the value of a private field from outside the class.
- **Setters** (mutators) set/update the value of a private field from outside the class.

```
public class BankAccount {
    private double balance;
    private String accountNumber;

    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
    }

    public double getBalance() { return balance; }
    public void setBalance(double balance) { this.balance = balance; }
    public String getAccountNumber() { return accountNumber; }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("111000");
        account.setBalance(10000);
        System.out.println("Balance: " + account.getBalance());
        System.out.println("Account number: " + account.getAccountNumber());
    }
}
```

The example is asymmetric: `balance` gets both a getter and a setter, `accountNumber` gets only a getter. Whether a field is writable from outside is a per-field design decision, not something you owe every field.

The `static` keyword

`static` is a **non-access** modifier for methods and attributes:

- Static methods/attributes can be accessed without creating an object of the class.
- Useful in memory management; can be applied to variables, methods, blocks, and nested classes.
- In *procedural* programming the same keyword means something different: it marks a variable or function whose scope is limited to the file it is declared in.

Static variables (class variables) are shared among all instances of a class — useful for constants and shared properties:

```
class BankAccount {
    public static int totalAccounts = 0;

    private double balance;
    private String accountNumber;
    private String accountHolder;

    public BankAccount(double balance, String accountNumber, String accountHolder) {
        this.balance = balance;
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
    }

    public String getAccountNumber() { return accountNumber; }
    public double getBalance() { return balance; }
    public String getAccountHolder() { return accountHolder; }
}
```

Static methods can be called without creating an instance, but **cannot access non-static (instance) variables or methods directly**:

```
class Bank {
    static double interestRate = 5.0;

    static double calculateInterest(double balance, int years) {
        return (balance * interestRate * years) / 100;
    }
}

class BankAccount {
    private String accountHolder;
    private double balance;

    BankAccount(String name, double balance) {
        this.accountHolder = name;
        this.balance = balance;
    }

    void showInterest(int years) {
        double interest = Bank.calculateInterest(balance, years);
        System.out.println(accountHolder + " earns $" + interest + " in " + years + "
        ↪  years.");
    }
}
```

The call site is the point of the example: it reads `Bank.calculateInterest(...)` — on the *class*, not on an instance — and because `calculateInterest` is static it cannot reach any account's `balance` itself, so the balance has to be handed to it as an argument. The static field `interestRate`, by contrast, it can read directly, because that field belongs to the class too.

Class invariants

A class invariant specifies a condition (or set of conditions) that should always be true throughout the life of an object — used to ensure a system remains in a valid state. Put more precisely, it is a constraint that must hold **at every stable point in time** during the life of an object. A class invariant:

1. Must be established after the class constructor.
2. May be assumed as a precondition of each method (excluding the constructor).
3. Must be established after each method call.

```
class Counter {
    private int count;
    public Counter() { count = 0; }
    public void increment() { count = count + 1; }
}
// Invariant: count >= 0
```

Trace the three rules against `Counter`: the constructor sets `count = 0`, which establishes `count >= 0`; `increment` may *assume* `count >= 0` on entry; and since it only ever adds 1, `count >= 0` still holds on exit. Nothing else can reach `count`, because it is `private` — which is what makes the invariant enforceable at all rather than merely hoped for.

Invariants complement **preconditions** (what must be true before a method executes) and **postconditions** (what must be true after) — together these define a contract for how a method or function should behave. Pre/postconditions themselves are in java-specification[50] and java-hoare-logic[51]; see also week8-tutorial-class-invariants[52].

Protecting invariants

The `Counter` example does not show how an invariant can be broken from outside. Guarding against that is what makes the definition enforceable rather than merely stated; see also week8-tutorial-class-invariants[53] and the defensive-programming material in java-specification[54].

[50] courses/csse2002/java-specification.pdf
[51] courses/csse2002/java-hoare-logic.pdf
[52] courses/csse2002/week8-tutorial-class-invariants.pdf
[53] courses/csse2002/week8-tutorial-class-invariants.pdf
[54] courses/csse2002/java-specification.pdf

A specification can *claim* an invariant while the implementation still allows it to be broken. Two common leaks:

1. **Public fields** — a directly-mutable field lets any caller bypass the class entirely and violate the invariant. Fix: make the field `private`.
2. **Returning an internal reference** — a getter that `return`s its internal mutable collection/object directly hands the caller a way to mutate it from outside, bypassing any checks the class's own methods perform. Fix: return a **defensive copy**:

```
private List<String> files;

public List<String> getFiles() {
    return new ArrayList<>(files); // copy, not the internal reference
}
```

Preserving the invariant may also require adding **preconditions** to methods that could otherwise violate it (e.g. rejecting an addition that wouldn't satisfy the invariant), or, when a method is likely to be called by code outside your control, defensively throwing an exception (e.g. `IllegalArgumentException`/`IllegalStateException`) rather than relying purely on the precondition contract (see java-specification[55] — Defensive programming).

Method signatures

A method signature is a method's unique identifier: its **name** plus its **parameter types**. The return type and the parameter *names* do **not** count towards the signature, and a signature MUST be unique within a class:

```
public void print() { ... }              // signature: print()
public void print(int parameter) { ... } // signature: print(int)
```

Worked question — is this allowed?

```
public class Action {
    public Boot fish(int strength, int rodLength) { ... }
    public Fish fish(int strength, int polLength) { ... }
}
```

No. Both methods have the signature `fish(int, int)`. The differing return types (`Boot` vs `Fish`) and the differing parameter names (`rodLength` vs `polLength`) are exactly the two things a signature ignores, so the class will not compile. This same rule is what makes overriding

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

work in java-inheritance[56]: an override must repeat the superclass method's signature exactly, or it is a new overload rather than an override.

Java Events

Introduced in 2026-05-14-lambdas-streams-and-events[57] (Guest lecture, Week 11). Not examinable — see the note at the top of that lecture.

Event-driven programming

A common design pattern for **how to control the flow of your program**: basically a different way to glue together your various chunks of code.

You **subscribe** chunks of your code (often lambdas, see java-lambdas-and-streams[58]) to a *listener* for specific events **emanating from specific sources**; when the event is triggered, the subscribed code is activated. For example, clicking a button spawns a click event, which is then passed to anything subscribed to / listening for the click of *that* button.

The payoff is that the system becomes more discrete: each chunk does not need to know about other irrelevant details. That is low coupling, see java-cohesion-and-coupling[59].

It is incredibly common in two places:

- **GUI programming** — if you have done any GUI work on the web or with a desktop toolkit you have already used events.
- **Network programming** — Node.js, the JavaScript server runtime, is event-driven under the hood.

The event loop

The event system itself is generally run on a separate thread, or via a task-prioritisation (macro vs micro) architecture, and what it runs is an **event loop**:

> An event loop is a program that takes an event from a queue of events, and alerts all subscribed listeners that this event has occurred, dispatching the event information to them.

[56] courses/csse2002/java-inheritance.pdf
[57] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[58] courses/csse2002/java-lambdas-and-streams.pdf
[59] courses/csse2002/java-cohesion-and-coupling.pdf

Mostly you hook into one rather than writing it: many applications ship an event loop for you to use as a developer, and some languages build one into the language itself — **JavaScript is the iconic example**. In Java, **Swing and JavaFX both come with their own event systems** you can hook into.

Building a toy event system

Events are easy to pick up and use. They are often the first design pattern people meet, at high school, without realising it, when they wire up buttons on a basic website. Using them does not show the *internals*. The two things you would not naturally pick up:

1. an implementation of a **basic, naive event system**, so you can vaguely understand how they work under the hood;
2. a variation of the **CSSE2002 game engine** with a basic event system integrated, to show how that simplifies existing code on the developer's end.

The API

A basic event system needs only lambdas and 2 classes:

```
SimpleEvent (String type, String data):  getType():String, getData():String

EventSystem
    addListener(String type, Consumer<SimpleEvent> action): action
    removeListener(Consumer<SimpleEvent> action)
    addEvent(SimpleEvent event): void
    tick(): void
```

`SimpleEvent` is just a **named bundle of data** — a `type` string that says what happened and a `data` string carrying the payload, with a getter for each. The signatures give `EventSystem` no fields. Reading the four method names against the definition of an event loop above, it needs two pieces of state: a **queue** of events not yet dispatched, and a **map** from event type to the listeners subscribed to it.

Two details worth reading off the signatures:

- **`addListener` returns the `action` back to you.** The natural reading is that it hands you a handle to unsubscribe with later: if you passed the listener in as an inline lambda you have no other reference to it, and `removeListener` needs exactly that reference.
- **`removeListener` is the one method with no return type given.** Every other method has one (`:String`, `: action`, `: void`), so treat its return as unspecified rather than assuming `void`.

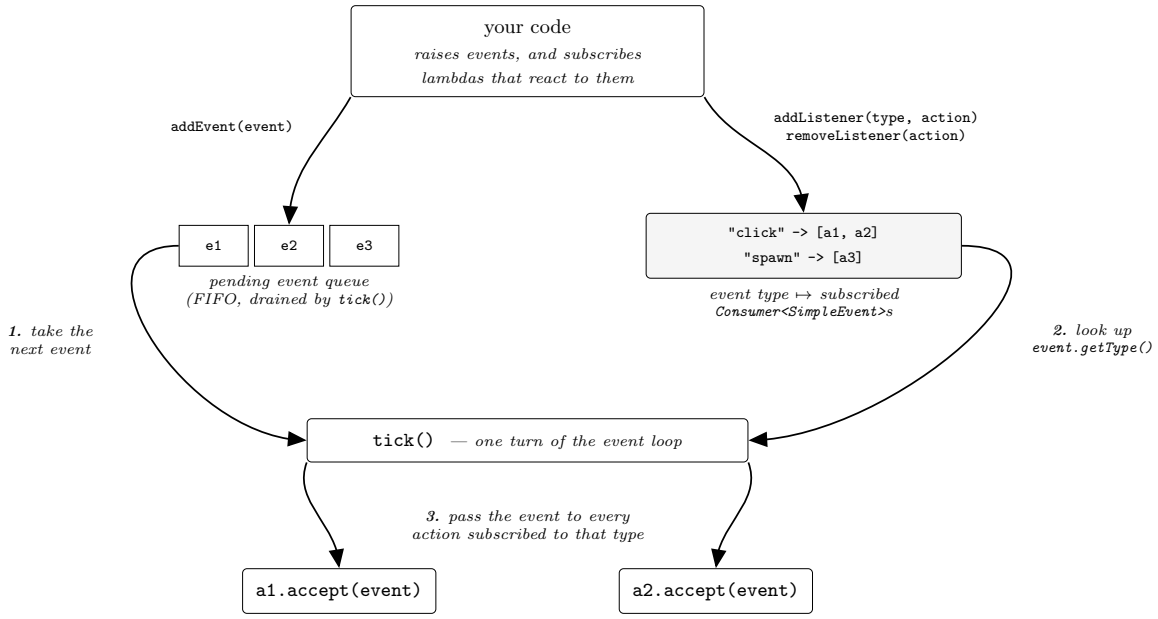


Figure 15: How the four EventSystem methods fit together: a queue of pending events, a map from event type to the listeners subscribed to it, and the numbered steps `tick()` takes to dispatch one event.

On the same reading, `tick()` is what actually drives the loop: it takes an event off the queue, looks up `event.getType()` in the listener map, and calls each subscribed `Consumer<SimpleEvent>` with the event itself as the argument. So nothing happens between ticks, and queuing an event only schedules it.

SimpleEvent written out

Written out as a class, the `SimpleEvent` signature is:

```
class SimpleEvent {
    private String type;
    private String data;

    public SimpleEvent(String type, String data) {
        this.type = type;
        this.data = data;
    }

    public String getType() { return type; }
    public String getData() { return data; }
}
```

Motivating example: the CSSE2002 game engine

The engine you used in the assignments is **very linear**: you called `.tick()`, passed in the various states, and then the parent tiles and managers would tick *their* children, passing the state down accordingly.

That works, but it **tightly links everything around that `tick()` method** as the mechanism for state being passed between the various parts of the program. An event system drastically cuts down on this connective tissue code: the code that exists only to shuttle state through layers that do not otherwise care about it. The saving comes from letting far-apart parts talk through the event system rather than through their common ancestor.

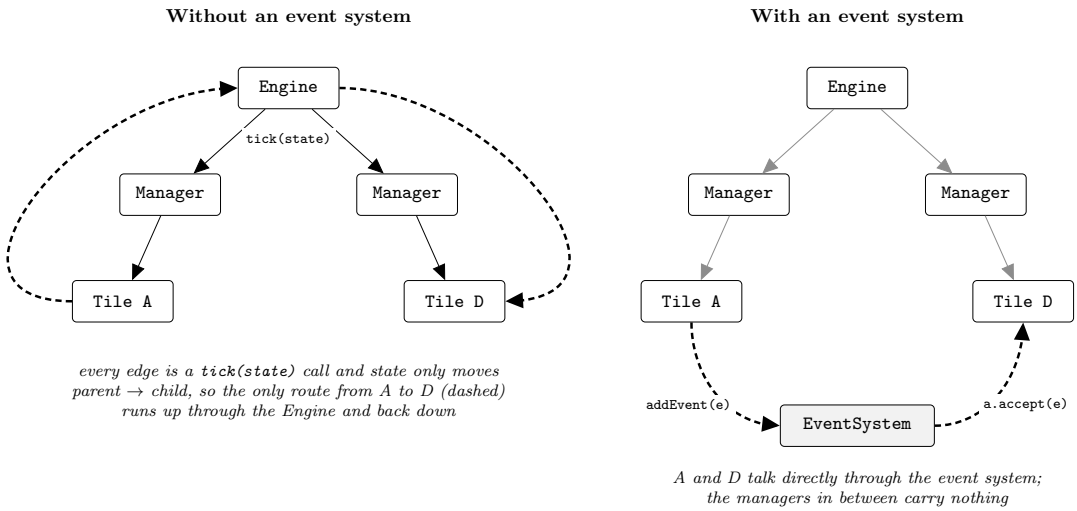


Figure 16: Why the game engine is the motivating example. Without an event system, parent tiles and managers tick their children and pass state down, so the only route from Tile A to Tile D runs up through the Engine and back down. With one, A and D talk through the EventSystem and the managers in between carry nothing.

Java Exceptions

Introduced in 2026-03-19-exceptions[60] (Lecture, Week 4, Part 1). Traced in depth in week5-tutorial-exceptions[61].

What is an exception?

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

[60] courses/csse2002/2026-03-19-exceptions.pdf
[61] courses/csse2002/week5-tutorial-exceptions.pdf

```java
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:

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

Exception hierarchy

`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`

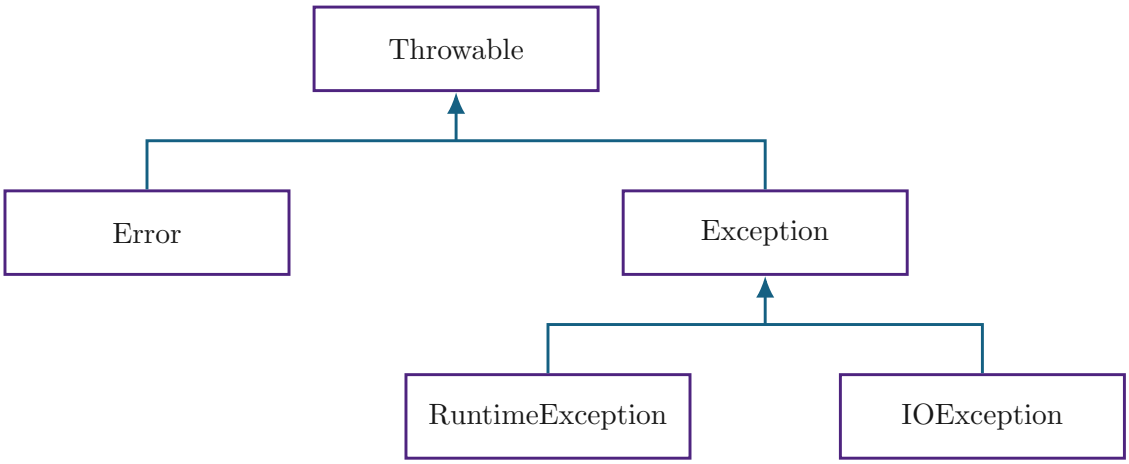


Figure 17: The exception hierarchy. Throwable is the root; Error and Exception sit beneath it; RuntimeException (unchecked) and IOException (checked) sit beneath Exception. Each connector joins the subclasses and points up to their parent. Which branch a class is on is what decides whether the compiler forces you to handle it, so this one picture settles the checked/unchecked question.

`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.**

```java
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**.

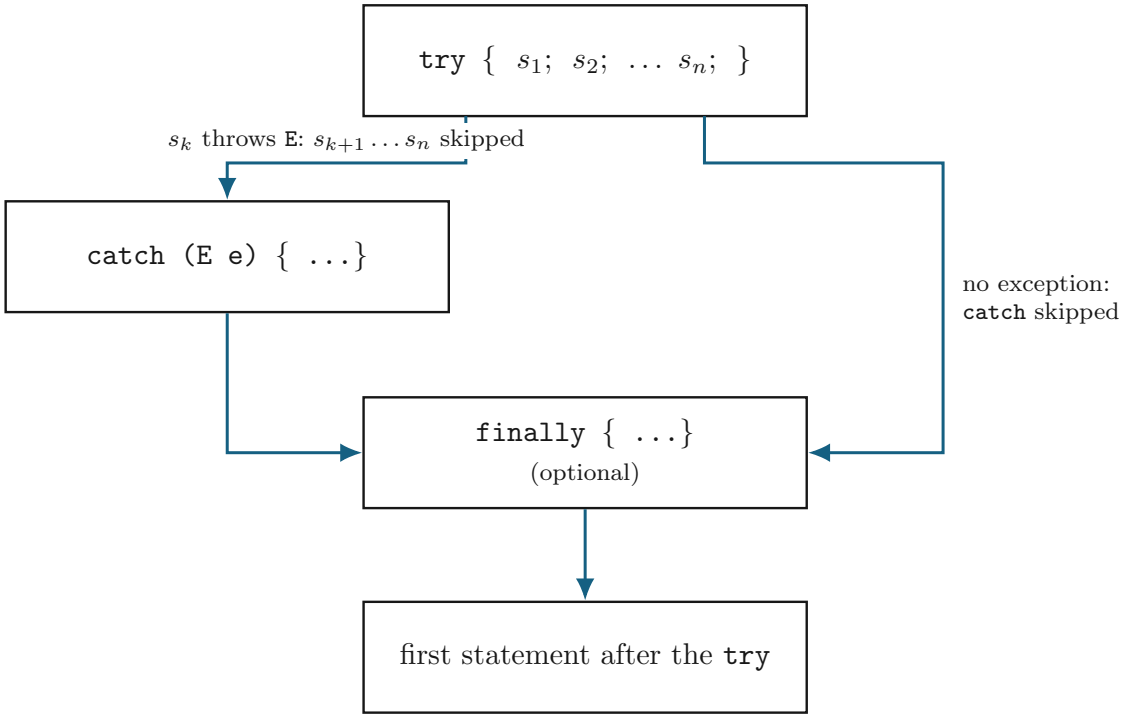


Figure 18: The two paths through a try/catch/finally, and the fact that both of them end up in finally. This is the three numbered rules together with the finally rule, drawn as one picture: the exceptional path abandons the remaining statements of the try and enters catch, the normal path skips catch entirely, and neither can skip finally.

`throw`

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

```
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");
    }
}
```

```
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[62]: 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:

```
public class ThrowsEx {
    public static void main(String[] args) {
        try {
```

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

```
            findFile();
        } catch (FileNotFoundException e) {
            System.out.println(e);
        }
    }
    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 Generics

Introduced in 2026-04-30-generics-in-java[63] (Lecture, Week 9). Practised again in week12-lab-generics[64].

Generics let **reference types** (`Integer`, `String`, … and user-defined types) be *parameters* when defining classes, interfaces and methods. There are two separate motivations for that, and they are worth keeping apart.

[63] courses/csse2002/2026-04-30-generics-in-java.pdf
[64] courses/csse2002/week12-lab-generics.pdf

Why generics? (1) Type safety

Prior to Java 1.5, collection classes could hold *any* type of object — `ArrayList.add` was declared

```
public boolean add(Object o) {
    // code to implement add method.
}
```

so anything could go in, and an explicit cast was needed on the way out (see java-casting[65]):

```
List list = new ArrayList();
list.add("Hello");
list.add(123);          // allowed
list.add(new Date());   // allowed

String str = (String) list.get(0);    // OK
Integer num = (Integer) list.get(1);  // OK
Date date = (Date) list.get(2);       // OK
```

The trap: `Integer num = (Integer) list.get(2);` compiles, because the compiler only knows `get` returns `Object`. It fails at *run time* with a `ClassCastException`, because slot 2 holds a `Date`. Nothing about the list stops a `String` being mixed into a collection that was only ever meant to hold `Cars`, because `add` accepts any `Object`.

Declaring the element type fixes both problems at once:

```
List<String> list = new ArrayList<>();  // <> is the diamond operator
list.add("Hello");
list.add(123);                // compile-time error
String str = list.get(0);     // no cast needed
```

```
List<Car> carList = new ArrayList<Car>();
Car c = carList.get(0);       // the compiler already knows it is a Car
```

This is **type safety**: type-related errors are caught and prevented *early*, at compile time, instead of surfacing as unintended bugs at run time.

[65] courses/csse2002/java-casting.pdf

What about `List<Object>`?

```
List<Object> list = new ArrayList<>();
list.add("Hello");
list.add(123);

Object object1 = list.get(0);
Object object2 = list.get(1);
```

It compiles, and it accepts both. But parameterising by `Object` buys nothing, because everything is an `Object`. Elements come back typed `Object`, so you are casting again, and `add` accepts anything again. Generics only pay off when the parameter is a *specific* type.

Why generics? (2) One class instead of one class per type

The second motivation is duplication, not safety. A class that prints an `Integer`:

```
public class PrintInt {
    Integer i;
    PrintInt(Integer i) { this.i = i; }
    public void print() { System.out.println(i); }
}
```

Wanting the same thing for `Double`, `Float` and `String` means writing `PrintDouble`, `PrintFloat` and `PrintString` — four classes with identical bodies and one word changed. Generics collapse them into one class that still preserves type safety.

Generic classes

A generic class parameterises a type across its whole body:

```
public class Print<T> {
    T i;
    Print(T i) { this.i = i; }
    public void print() { System.out.println(i); }
}

Print<Integer> print  = new Print<>(1);
Print<Double>  print2 = new Print<>(1.1);
Print<String>  print3 = new Print<>("1.1");
```

By convention, type parameters are single letters: `T` (Type), `E` (Element — used extensively by the Java Collections Framework, see java-collections-framework[66]), `K` (Key), `V` (Value), `N` (Number), and `S`/`U`/`V`… for a 2nd/3rd/4th type.

Generic methods

A single *method* can be made generic, independently of whether its enclosing class is. The type parameter is declared between the modifiers and the return type:

```
public <T> void genericsMethod(T data) {
    // do something with data
}

public <T> T genericsMethod(T data) {
    // do something with data here
    return data;                // returning any T-type value
}
```

Multiple type parameters can be declared together, and the return type can be a third one:

```
public <T, S, R> R genericMethod(T value1, S value2) {
    return null;
}

String result = genericMethod(42, 3.14);
```

Type inference

A `Printer` whose `print` method is generic, in place of a fixed-type one:

```
public class Printer {
    public <T> void print(T data) {
        System.out.println("Lets shout " + data + "!!!");
    }
}

Printer printer = new Printer();
printer.print(124);                 // T inferred as Integer
printer.print("I love CSSE2002");    // T inferred as String
```

> The Java compiler can automatically determine the type of `T` based on the type of the argument you pass to the method. This is known as **type inference**.

[66] courses/csse2002/java-collections-framework.pdf

What if `Printer` were a generic *class* instead?

```
public class Printer<T> {
    public void print(T data) {
        System.out.println("Lets shout " + data + "!!!");
    }

    public static void main(String[] args) {
        Printer<String> printer = new Printer<>();
        printer.print("data");
        printer.print(123);
    }
}
```

A *class* type parameter is fixed once, when the object is created. `new Printer<>()` assigned to `Printer<String>` pins `T = String` for the life of that object, so `printer.print(123)` is a compile-time error. A *method* type parameter is inferred afresh at every call, which is why the generic-method version above accepted an `Integer` and a `String` through the same object. Choose the class parameter when one object should work with one type throughout; choose the method parameter when each call is independent.

Bounded generics

A type parameter can normally accept *any* reference type. **Bounded generics** restrict it to a specific type or its subtypes, using `extends`. The form is `<T extends X>`, and it means `T` can only accept data that are subtypes of `X`. Restricting the `Printer` to numbers:

```
public class Printer {
    public <T extends Number> void print(T data) {
        System.out.println("Lets shout " + data + "!!!");
    }
}

printer.print(123);      // OK, Integer is a Number
printer.print(123.2);    // OK, Double is a Number
printer.print("123");    // compile-time error - String is not a Number
```

Despite the keyword, `extends` here also covers implementing an interface.

Bounds through wildcards

Generic-type bounding restricts what may be substituted for the type parameter. Wildcards (`?`) do this *at the use site* — in a method's parameter list — rather than where a generic class or method is declared. Three forms:

Form	Name
`List<?>`	unbounded wildcard
`? extends T`	upper bound wildcard
`? super T`	lower bound wildcard

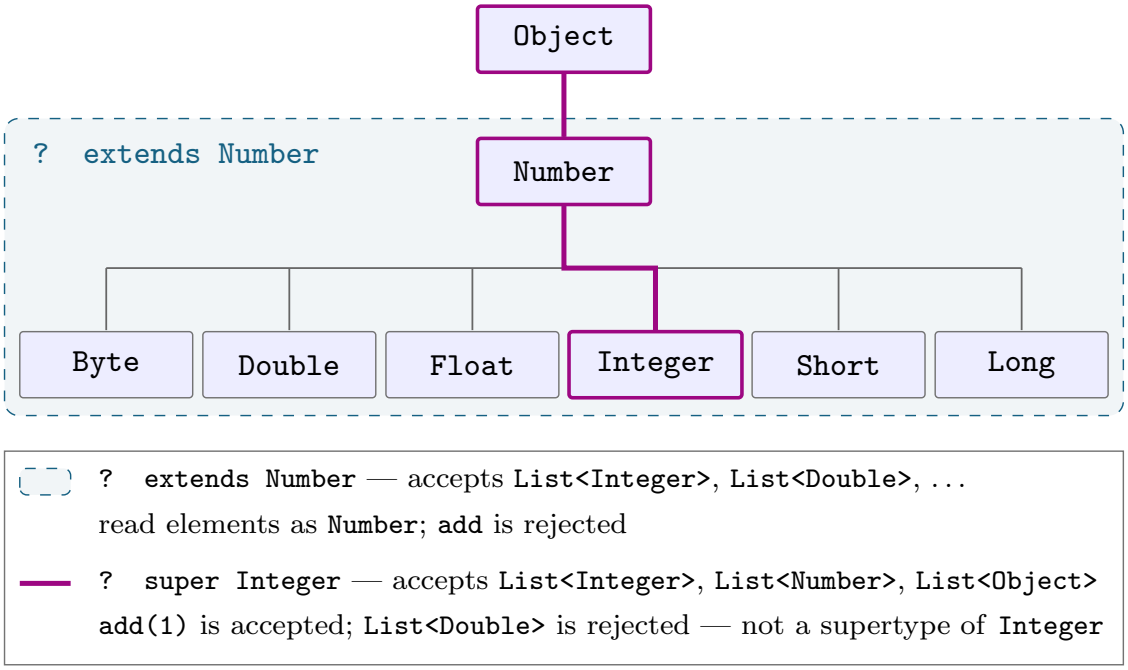


Figure 19: The two bounded wildcards drawn on the `Number` hierarchy. `? extends Number` admits `Number` and everything below it — you may read elements as `Number`, but `add` is refused; `? super Integer` admits `Integer` and everything above it — you may `add` an `Integer`, and `List<Double>` is refused. `Object` is on top because the `addNumbers` example passes a `List<Object>`.

Unbounded wildcard: `List<?>`

`List<?>` means "a list of some unknown type". Java does not know whether it is actually a `List<String>`, a `List<Integer>` or a `List<Product>` — **therefore it prevents the caller**

from writing to the list:

```
public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        addSomething(numbers);
    }

    public static void addSomething(List<?> list) {
        list.add("Hello");   // error
    }
}
```

The compiler has to refuse: if Java allowed `"Hello"` to be added, the `List<Integer>` would now contain a `String`, breaking type safety.

Choosing between `<T>` and `<?>` for a read-only method:

```
public <T> void printList(List<T> toPrint) {
    for (int i = 0; i < toPrint.size(); i++) {
        System.out.println(toPrint.get(i));
    }
}
```

```
public void printList(List<?> toPrint) {
    for (int i = 0; i < toPrint.size(); i++) {
        System.out.println(toPrint.get(i));
    }
}
```

> Use `<T>` when you need to **work with the type**. Use `<?>` when you only need to **read values**.
>
> Wildcards are usually bounded (`<? extends T>`, `<? super T>`) to reduce flexibility and improve safety.

Upper bounded wildcard: `? extends T`

Represents `T` or any **subclass of** `T`. Used mainly to **read data** from a structure — you cannot safely *add* to it, because the exact subtype is unknown:

```
List<? extends Number> nums = new ArrayList<Integer>();

Number n = nums.get(0);  // OK: read as Number
nums.add(10);            // compile-time error
```

The reason this form is needed at all is **invariance**. Write a sum over a `List<Number>`:

```
public static double sum(List<Number> list) {
    double sum = 0;
    for (Number n : list) {
        sum += n.doubleValue();
    }
    return sum;
}

List<Double> numbers = new ArrayList<>();
numbers.add(1.1);
sum(numbers);   // does not compile
```

In standard inheritance `Number n = 5.0;` is fine, because `Double` is a subclass of `Number`. But **generics in Java are invariant**: even though `Double` is a subclass of `Number`, `List<Double>` is *not* a subclass of `List<Number>`. (*Effective Java*, Item 31: parameterized types are invariant; `List<Type1>` is neither a subtype nor a supertype of `List<Type2>`.) Widening the parameter with an upper bound is what makes the method accept both:

```
public static double sum(List<? extends Number> list) {
    double sum = 0;
    for (Number n : list) {
        sum += n.doubleValue();
    }
    return sum;
}
```

…and the price is that the method still *cannot safely write*, because it does not know whether the actual list is a `List<Double>`, a `List<Integer>`, or something else.

Lower bounded wildcard: `? super T`

Represents `T` or any of its **superclasses**. Used when you want to **perform write operations** on a generic collection but keep flexibility in which collections are accepted:

```
public class Main {
    public static void addNumbers(List<? super Integer> list) {
        list.add(1);   // Allowed: Integer is within bounds
        list.add(2);   // Allowed: Integer is within bounds
    }

    public static void main(String[] args) {
        List<Number> numbers = new ArrayList<>();
```

```
        addNumbers(numbers);   // fine: Number is a superclass of Integer

        List<Object> objects = new ArrayList<>();
        addNumbers(objects);   // fine: Object is also a superclass of Integer

        List<Double> doubles = new ArrayList<>();
        addNumbers(doubles);   // compilation error
    }
}
```

Type erasure

Generics provide type checking at **compile time** only. Java implements them by **type erasure**, which removes all type parameters during compilation, replacing each with

- its upper bound (e.g. `Number`, for `<T extends Number>`), or
- `Object`, if unbounded.

As a result, generic type information is **not available at run time**.

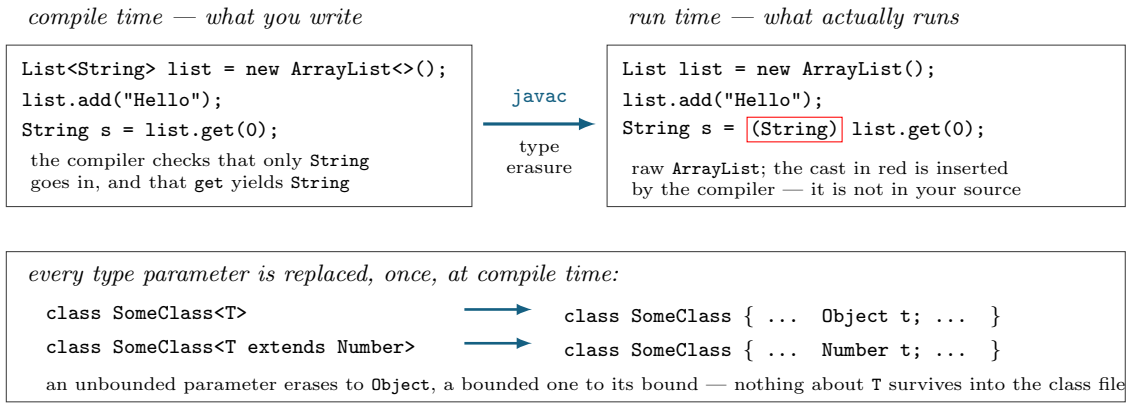


Figure 20: Type erasure as a pipeline: the compiler checks the source against the type parameters, erases each parameter to its bound (`Object` if unbounded), and inserts the casts the erased code needs — so the cast you were told you did not have to write still happens, just not by you. The top row is the compile-time source and what runs after `javac`, with the inserted cast in the red rectangle. The bottom row is what an unbounded and a bounded parameter each erase to.

Spelled out for a bounded parameter — the compiler replaces `T` with the extended class:

```
// Source
class SomeClass<T extends Number> {
    private T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }
}

// After type erasure, the compiler generates
class SomeClass {
    private Number t;
    public void add(Number t) { this.t = t; }
    public Number get() { return t; }
}
```

and for an unbounded one, `T` becomes `Object` (`private Object t;`, `public void add(Object t)`, `public Object get()`).

Practical example: a bookshop

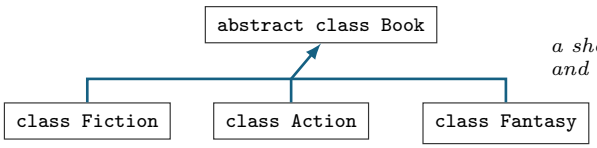
> Imagine you are developing a system for a bookshop that sells books across various genres: `Fiction`, `Action`, and `Fantasy`. Books are in bookshelves. Your job is to design the system in a way that allows the shop to store books of these different genres and retrieve them when needed.

The genres are subclasses of an abstract `Book`:

```
public abstract class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
    public String getTitle()  { return title; }
    public String getAuthor() { return author; }
    public abstract String getGenre();
}

public class Fiction extends Book {
    public Fiction(String title, String author) { super(title, author); }
    @Override
    public String getGenre() { return "Fiction"; }
}
// Action is identical, returning "Action"
```



a shelf has to hold `Book`*s of one genre and give them back as that genre*

```
class BookShelf {
  List<Book> inventory;
  void addItem(Book item)
  Book getItem(int index)
}
```

One shelf, element type `Book`. Reusable, but `getItem` only ever promises a `Book`, so the caller has to cast: `Fiction f = (Fiction) shelf.getItem(0);`

reusable: yes type safe: no
a wrong cast is a `ClassCastException` at run time

```
class FictionShelf {
  List<Fiction> inventory;
  void addItem(Fiction b)
  Fiction getItem(int i)
}
```

A separate shelf class per genre. No cast needed — but `ActionShelf` and `FantasyShelf` are the same class written out again, once per genre.

reusable: no type safe: yes

```
class BookShelf<T extends Book> {
  List<T> inventory;
  void addItem(T item)
  T getItem(int index)
}
```

One generic shelf. `T` is fixed when the shelf is created, so `getItem` gives that genre back directly; the bound `extends Book` is what keeps `getGenre()` legal inside the class:

```
BookShelf<Fiction> shelf = new BookShelf<>();
```

reusable: yes type safe: yes

Figure 21: The three shelf designs, and why only the third is both reusable and type safe. One shelf of `Book` is reusable but needs a cast; a separate shelf class per genre gives better type safety but creates duplicated code; `class BookShelf<T extends Book>` gets both. The stacked outlines behind `FictionShelf` stand for the same class written out again for the other genres.

Without generics

One `BookShelf` storing the abstract `Book` type loses the genre on retrieval:

```
public class BookShelf {
    private List<Book> inventory = new ArrayList<>();

    public void addItem(Book item) { inventory.add(item); }
    public Book getItem(int index) { return inventory.get(index); }

    public void displayBooks() {
        for (Book book : inventory) {
            System.out.println(book.getTitle() + " (" + book.getGenre() + ")");
        }
    }
}
```

The failure, worked through. The commented-out line is the one that breaks:

```
BookShelf shelf = new BookShelf();
shelf.addItem(new Fiction("The Hobbit", "J.R.R. Tolkien"));
shelf.addItem(new Action("Die Hard", "Roderick Thorp"));
shelf.addItem(new Fiction("Harry Potter", "J.K. Rowling"));

Book b = shelf.getItem(0);                      // returns Book

Fiction f  = (Fiction) shelf.getItem(0);        // fine
// Fiction f2 = (Fiction) shelf.getItem(1);     // breaks: index 1 is an
//                                              // Action -> ClassCastException
Action f2  = (Action) shelf.getItem(1);         // the caller has to know
```

`getItem()` returns only `Book`, so retrieving a genre needs a cast, and a wrong cast is a `ClassCastException` — **not type safe**.

Writing separate `FictionShelf`/`ActionShelf` classes instead does give better type safety, but every shelf is the same class copied out again:

```
public class FictionShelf {
    private List<Fiction> inventory = new ArrayList<>();
    public void addItem(Fiction book) { inventory.add(book); }
    public Fiction getItem(int index) { return inventory.get(index); }
    public void displayBooks() { for (Book book : inventory) { /* ... */ } }
}
// class ActionShelf { ...the same again with Action... }
```

With generics

A single generic `BookShelf<T extends Book>` gives type safety *and* reuse from one class. The bound matters twice over: it is what lets `displayBooks` call `book.getGenre()` (the compiler knows every `T` is at least a `Book`), and what stops a `BookShelf<String>` ever being created.

```
public class BookShelf<T extends Book> {
    private List<T> inventory = new ArrayList<>();

    public void addItem(T item) { inventory.add(item); }
    public T getItem(int index) { return inventory.get(index); }

    public void displayBooks() {
        for (T book : inventory) {
            System.out.println(book + "(" + book.getGenre() + ")");
        }
    }
}

BookShelf<Fiction> fictionShelf = new BookShelf<>();
BookShelf<Action>  actionShelf  = new BookShelf<>();

fictionShelf.addItem(new Fiction("The Hobbit", "J.R.R. Tolkien"));
fictionShelf.addItem(new Fiction("Harry Potter", "J.K. Rowling"));
actionShelf.addItem(new Action("Die Hard", "Roderick Thorp"));

fictionShelf.displayBooks();   // getItem() here returns Fiction, no cast
actionShelf.displayBooks();
```

Java Hoare Logic

Introduced in 2026-05-21-correct-programming[67] (Lecture, Week 12). Formalises reasoning about java-specification[68]'s preconditions/postconditions.

What does it mean for a program to be correct?

> A correct program **satisfies its specification**.

Correctness is only meaningful *relative to a specification* — asking "is this program correct?" without one is unanswerable:

[67] courses/csse2002/2026-05-21-correct-programming.pdf
[68] courses/csse2002/java-specification.pdf

```java
public int indexOf(int[] numbers, int number) {
    for (int i = 0; i < numbers.length; i++) {
        if (numbers[i] == number) {
            return i;
        }
    }
    return -1;
}
```

- With **no spec at all**: maybe correct, maybe not — we can't say without knowing what it's *meant* to do.
- Spec'd as `/** Returns how many times number occurs in numbers. */`: **incorrect** — counter-example `indexOf({1, 1, 1}, 1) == 0`, but a correct implementation of *that* spec should return `4`.
- Spec'd as `/** Returns the index of number within numbers, if number is in numbers. */`: **correct**, but the spec is *underspecified* — it says nothing about what happens when `number` isn't in `numbers`.

A specification restricts the space of acceptable implementations: think of all possible input/output pairs as arrows crossing an Implementation/Specification boundary — an implementation is correct exactly when every arrow it draws stays within the region the specification permits. A `@requires`/`@ensures` Javadoc comment (see java-specification[69]) states the precondition and postcondition that bound this region.

Hoare triples

Preconditions and postconditions aren't just for whole methods — **every block of code has them**. We write this as a **Hoare triple**:

$$\{P\}\ S\ \{Q\}$$

meaning: if precondition P holds before statement(s) S run, postcondition Q holds afterwards. E.g.:

```java
// {true}
if (x > y) {
    max = x;
} else {
    max = y;
}
// {max >= x && max >= y}
```

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

Deriving preconditions by propagating backward

Given a method's postcondition, we can work out its precondition (or verify a block of code) by propagating the postcondition **backward** through each statement, substituting as we go.

Straight-line code

```
/**
 * @requires numbers != null && 2 < numbers.length && numbers[2] == number
 * @ensures numbers[\result] == number
 */
public int indexOf(int[] numbers, int number) {
    // {numbers[0 - 10 + 12] == number}
    int result = 0;
    // {numbers[result - 10 + 12] == number}
    result = result + 12;
    // {numbers[result - 10] == number}
    result = result - 10;
    // {numbers[result] == number}
    return result;
    // {numbers[\result] == number}
}
```

Each line's precondition is obtained by substituting the assigned expression into the *next* line's already-derived precondition, working from the `@ensures` postcondition backward to the top of the method — the resulting `@requires` is whatever's left once you reach the very first line.

The direction is the part that trips people up: the listing reads top-to-bottom but is *derived* bottom-to-top, and each rung of the derivation is a single textual substitution.

Through `if/else`

Each branch is handled separately, and the two derived preconditions are combined into a single condition depending on which branch executes:

```
/**
 * @requires numbers != null && 12 < numbers.length
 *               && numbers[12] == number && number % 2 == 0
 * @ensures numbers[\result] == number
 */
public int indexOf(int[] numbers, int number) {
    // {number % 2 == 0 ==> numbers[0 + 12] == number
    //  && number % 2 != 0 ==> numbers[0 - 10] == number}
    int result = 0;
    if (number % 2 == 0) {
```

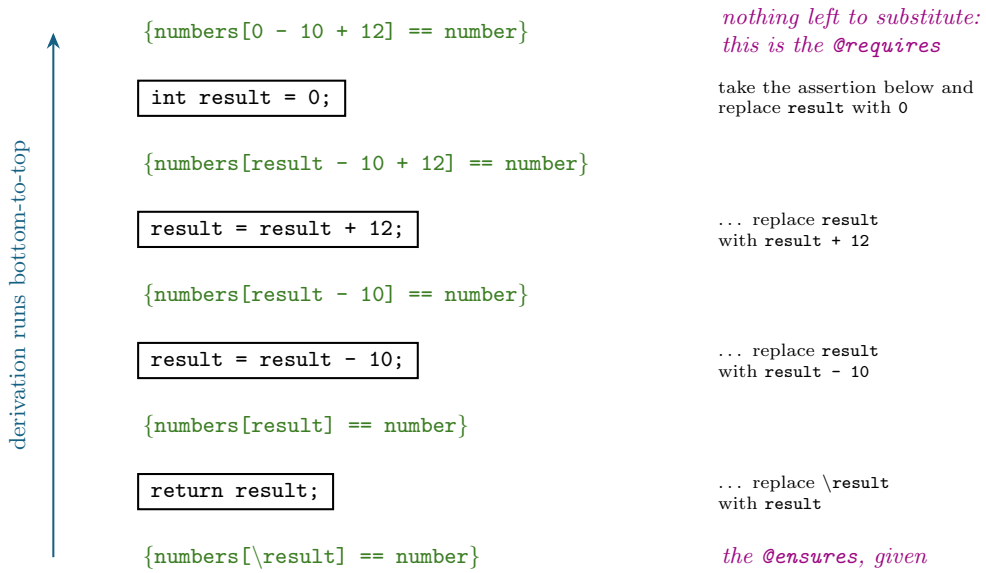


Figure 22: The same derivation as the listing above, laid out so the direction of the reasoning is visible: the `@ensures` at the bottom is given, each statement rewrites the assertion *below* it by replacing the assigned variable with the expression assigned to it, and whatever is left at the top is the `@requires`. The assertions are in green; they are `//` comments in the listing.

```
        // {numbers[result + 12] == number}
        result = result + 12;
    } else {
        // {numbers[result - 10] == number}
        result = result - 10;
    }
    // {numbers[result] == number}
    return result;
}
```

Both branches must reach the *same* postcondition `{numbers[result] == number}`, so the derivation forks there, runs up each branch independently, and only rejoins once each branch's precondition has been guarded by the condition that selects it:

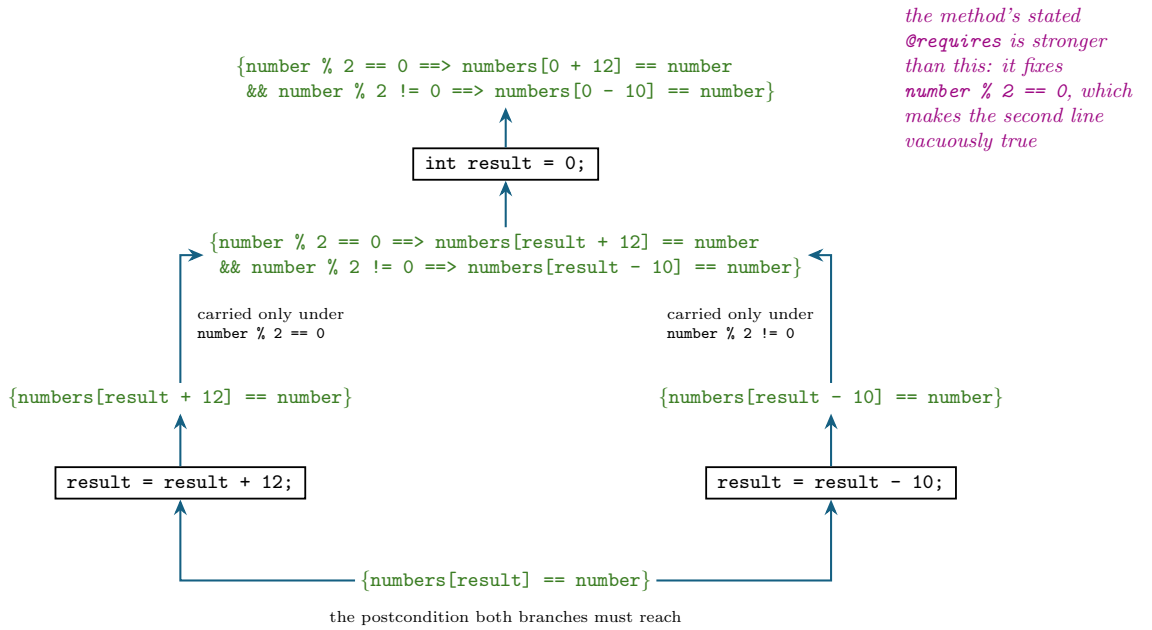


Figure 23: The `if/else` derivation above drawn as a fork and join. Reading upward: one shared postcondition, two independent branch derivations, then a single combined condition in which each branch's requirement is guarded by the test that reaches it, and finally one more substitution through `int result = 0;`. The combined assertion in the middle is the top condition one substitution earlier, a step the listing skips over.

Compare the top of Figure 23 with the `@requires` the method actually declares. The derived condition covers *both* branches; the declared one, `numbers != null && 12 < numbers.length && numbers[12] == number && number % 2 == 0`, pins `number` to the even case. That is

legal — a **stronger** precondition than the derived one is always safe, because it still implies it (with `number % 2 == 0` fixed, the second disjunct is vacuously true) — it just means the method promises less than the code could deliver. Deriving gives you the *weakest* precondition; what you write in the Javadoc may be anything that implies it.

Proving a block correct

The same backward-propagation technique proves an *arbitrary* block of code satisfies a given Hoare triple — substitute back through each branch/statement and confirm the starting precondition (`true`, in these examples) is actually implied:

```
// {true}
if (x > y) {
    // {x > y} ==> {x >= x && x >= y}
    max = x;
    // {max >= x && max >= y}
} else {
    // {y >= x} ==> {y >= x && y >= y}
    max = y;
    // {max >= x && max >= y}
}
// {max >= x && max >= y}
```

A three-statement swap works the same way, substituting each assignment's right-hand side backward through `\old(...)` references:

```
// {true}
// {y == \old(y) && x == \old(x)}
int tmp = x;
// {y == \old(y) && tmp == \old(x)}
x = y;
// {x == \old(y) && tmp == \old(x)}
y = tmp;
// {x == \old(y) && y == \old(x)}
```

A trickier variant swaps `x`/`y` using only arithmetic (no temporary variable), which still propagates the same way:

```
// {true}
// {y == \old(y) && x == \old(x)}
// {y - (y - x) + (y - x) == \old(y) && y - (y - x) == \old(x)}
x = y - x;
// {y - x + x == \old(y) && y - x == \old(x)}
y = y - x;
```

```
// {y + x == \old(y) && y == \old(x)}
x = y + x;
// {x == \old(y) && y == \old(x)}
```

(A similar swap is possible using bitwise XOR instead of subtraction, exploiting $a \oplus b = b \oplus a$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, $a \oplus 0 = a$, $a \oplus a = 0$.)

Writing the triple down: `\result` and `\old(...)`

A Hoare triple written in Java lives in a Javadoc comment (see java-specification[70]): P goes in the precondition tag, Q in the postcondition tag, and S is the method body. Two pieces of vocabulary let a postcondition say things a plain Java boolean expression cannot.

- **`\result`** names the value the method returns. It is the established way to refer to a return value inside a postcondition.
- **`\old(e)`** names the value the expression `e` had *before* the call. It is defined in java-specification[71] and is used again in the Week 12 practical. Without it the swap derivations above could not be stated at all: `x == y && y == x` says nothing, whereas `x == \old(y) && y == \old(x)` is exactly the swap.

> **The contract tags are spelled two ways in this course**
>
> The tags are `@requires` / `@ensures` here and in the tag table in java-specification[72]. The **Week 12 practical sheet** (week12-lab-generics[73]) writes `@require` / `@ensure`, with no trailing `s`, and so do the SOLID contracts. Neither form is canonical. Use whichever form the sheet in front of you uses.

A worked contract from the Week 12 practical

The practical's `transferStudents` (Task 4) is the fullest contract in the Week 12 material, and it uses both pieces of vocabulary. Its `@ensure` writes `bus.getCurrentCount` without the parentheses:

[70] courses/csse2002/java-specification.pdf
[71] courses/csse2002/java-specification.pdf
[73] courses/csse2002/java-specification.pdf
[73] courses/csse2002/week12-lab-generics.pdf

```java
/**
 * Transfer students to the new bus.
 *
 * @param students Bus only containing Concession fare riders.
 * @param bus      Bus to which students will transfer.
 * @require students != null
 *          && bus.getCapacity() >= students.getCurrentCount()
 * @ensure  students.getCurrentCount() == 0
 *          && bus.getCurrentCount >= \old(students).getCurrentCount()
 *          && bus contains all passengers from \old(students)
 */
public static void transferStudents(Bus<Concession> students,
        Bus<? super Concession> bus) { ... }
```

The method returns `void`, so there is no `\result` to talk about; everything the postcondition has to say is about how two objects' states *changed*, which is what makes `\old(...)` indispensable here. `students.getCurrentCount() == 0` is a fact about the state after; `\old(students).getCurrentCount()` reaches back to the state before, so the middle clause can hold the target bus's final count up against the number of passengers the source bus had before the call.

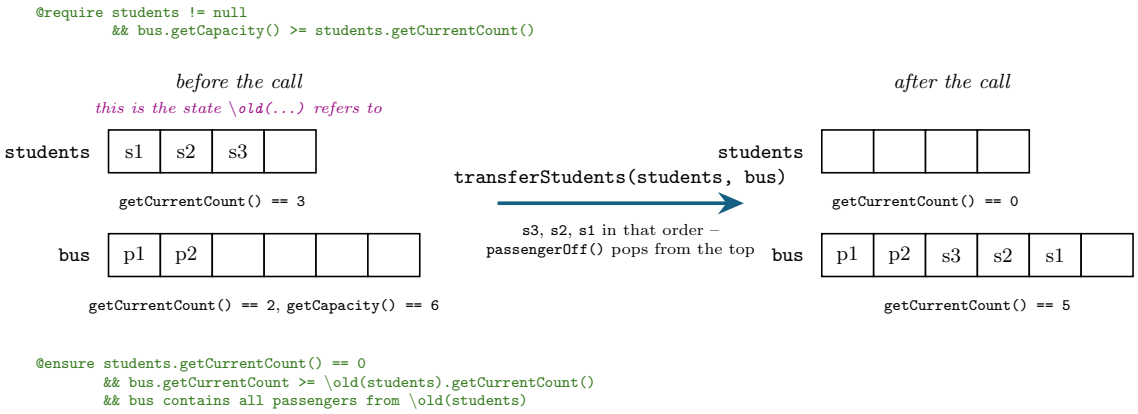


Figure 24: The Week 12 practical's `transferStudents` contract as a before/after picture, on a source bus of capacity 4 carrying three passengers and a target bus of capacity 6 already carrying two. Each bus is an array with its name to the left. The contract text is in green and the annotation in magenta.

Two things about this contract as written.

The postcondition mixes Java and English in the same tag. `students.getCurrentCount() == 0` is a boolean expression; `bus contains all passengers from \old(students)` is a sentence. The course convention is that conditions are written in Java boolean syntax,

not mathematical notation, but the only queries `Bus` exposes are `getCurrentCount()` and `getCapacity()` (its other two methods, `passengerOn` and `passengerOff`, both mutate), so there is no expression in the API that *can* say "contains". The contract falls back to prose rather than inventing one, and the same happens in Task 3, whose whole postcondition is `@ensure \result contains trainee and all trainers`.

The precondition is weaker than the postcondition needs. `bus.getCapacity() >= students.getCurrentCount()` compares the target's *total* capacity against the number of students, not its *free* space. `Bus.passengerOn` (Task 0) silently does nothing when the bus is full:

```
public void passengerOn(T passenger) {
    if (currentCount < capacity) {
        passengers[currentCount++] = passenger;
    }
}
```

so a `bus` of capacity 6 already carrying 4, given a `students` bus of 3, satisfies the precondition (`6 >= 3`) yet drops the third student on the floor — `students.getCurrentCount() == 0` still holds, because `passengerOff` removes them regardless, but `bus contains all passengers from \old(students)` does not. The precondition that actually discharges the postcondition is `bus.getCapacity() - bus.getCurrentCount() >= students.getCurrentCount()`. The marked solution is the contract above, with the weaker precondition.

The loop inside that method — and why its bound is copied into a local variable before the loop starts — is taken up in java-loop-invariants[74].

Java Inheritance

Introduced in 2026-03-05-object-oriented-programming-i[75] (Lecture, Week 2) — "things you have because your parents have them".

Subclasses and superclasses

Inheritance allows creating a new class from an existing class:

- 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:

[74] courses/csse2002/java-loop-invariants.pdf
[75] courses/csse2002/2026-03-05-object-oriented-programming-i.pdf

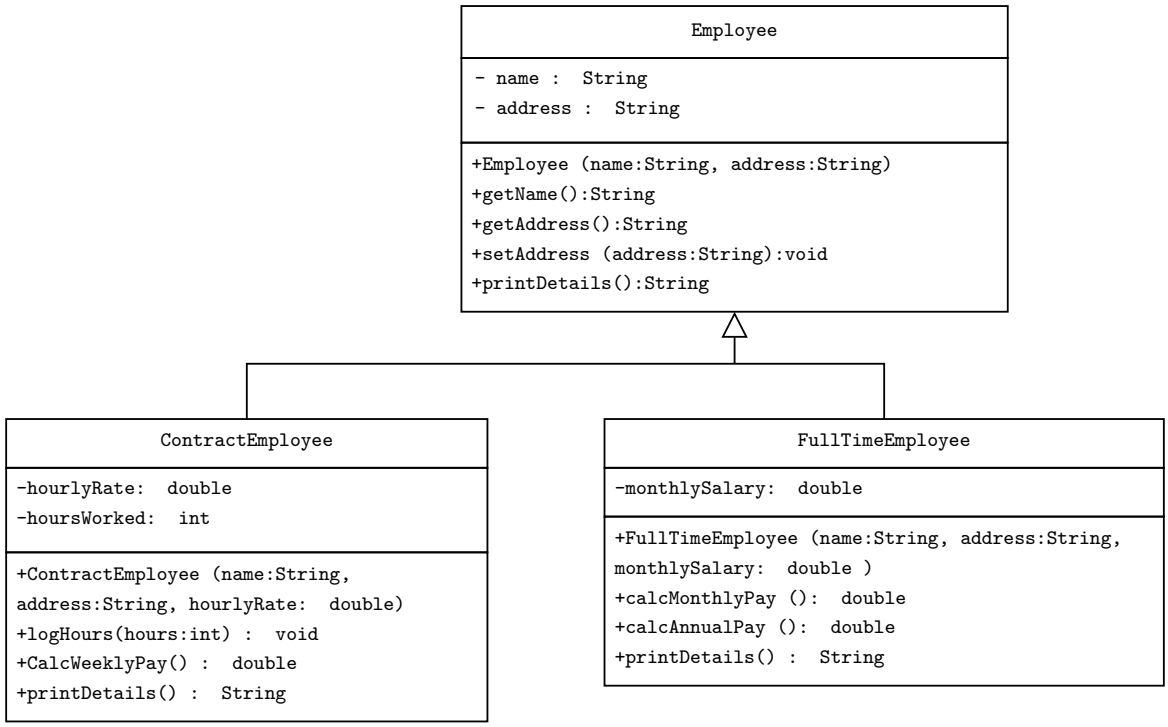


Figure 25: The `Employee` inheritance example as a UML class diagram. Each box has three compartments (name / attributes / operations), with - for private and + for public, and constructors listed in the operations compartment with no return type. Generalisation is a solid line with a single unfilled 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[76] 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 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;
```

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

```
    }
    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[77]; 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;
    }
}
```

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

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

`super` in an override

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[78] and week4-tutorial-method-dispatch-and-casting[79].

Java I/O

Introduced in 2026-04-23-java-io[80] (Lecture, Week 8).

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

1. **Streams** (`java.io`, since 1.0) — byte-oriented.

[78] courses/csse2002/java-polymorphism.pdf
[79] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf
[80] courses/csse2002/2026-04-23-java-io.pdf

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

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

Streams

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

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

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

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

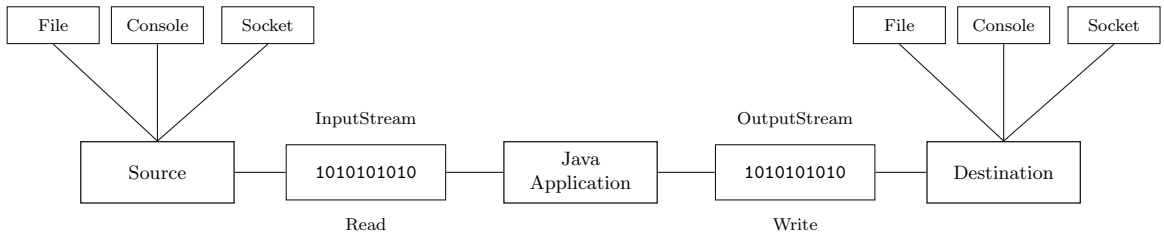


Figure 26: The stream abstraction: whatever the real source (file, console, socket), the application reads bytes through one `InputStream`, and writes them back out through one `OutputStream`, whatever the real destination.

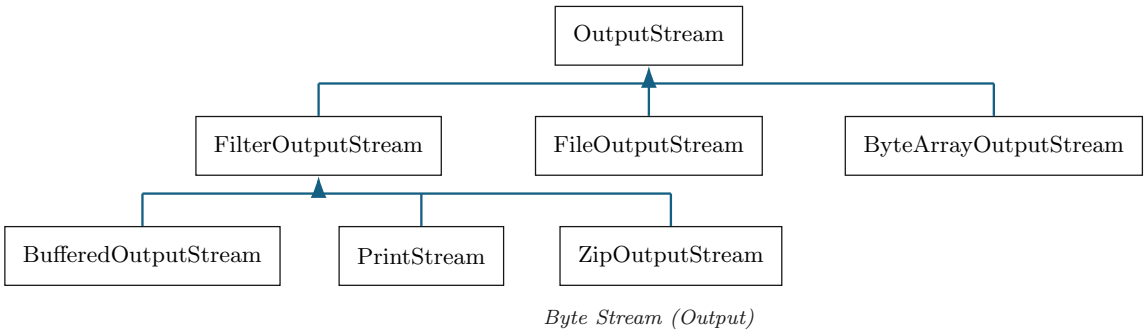
Java has two families of streams:

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

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

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

The byte-stream hierarchy



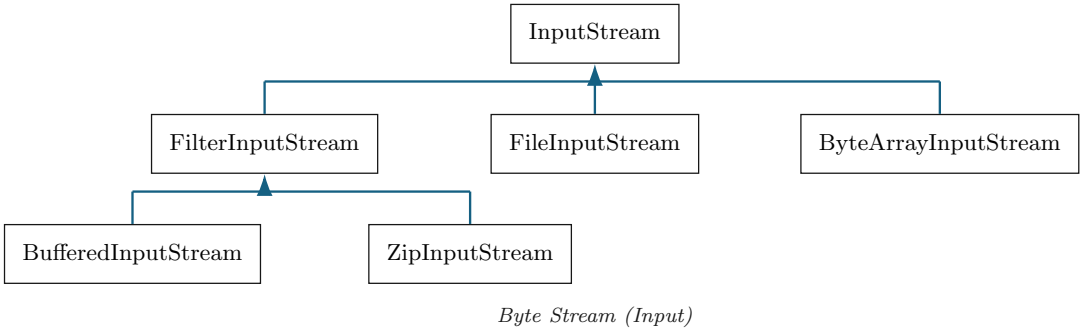


Figure 27: The byte-stream class hierarchies: `OutputStream` and `InputStream`. Each superclass sits above its subclasses, and the arrowhead points into the parent. The wrappers `BufferedInputStream` and `BufferedOutputStream` are `Filter*` streams — streams that wrap another stream.

`InputStream` and its subclasses represent a stream of bytes, drawn from different sources — `FileInputStream` gets bytes from a file, `ByteArrayInputStream` from an array, and so on. Methods that use streams should accept the **superclass** type as a parameter, so any concrete stream can be substituted (see java-solid-principles[81] — Liskov Substitution).

Worked example — writing to an abstract `OutputStream`

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

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

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

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

        System.out.println(fos);
    }

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

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

End of file

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

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

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

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

    private static void readAll(InputStream input) throws IOException {
```

```
        int in = 0;
        while (in != -1) {
            in = input.read();
            System.out.println(in);
        }
    }
}
```

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

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

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

Buffering

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

Only the construction changes; `readAll` is untouched:

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

(What each layer actually contributes is drawn in Figure 28.)

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

Closing streams

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

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

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

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

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

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

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

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

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

Wrapping streams together

Nothing in `java.io` is one class doing everything: you assemble the behaviour you want by nesting one stream inside another. That is what the hierarchy's whole `Filter*` branch

is for, and why the constructor calls look the way they do. The second chain below uses `InputStreamReader` and `BufferedReader`, which the next section introduces — it is drawn here so both chains can be compared side by side.

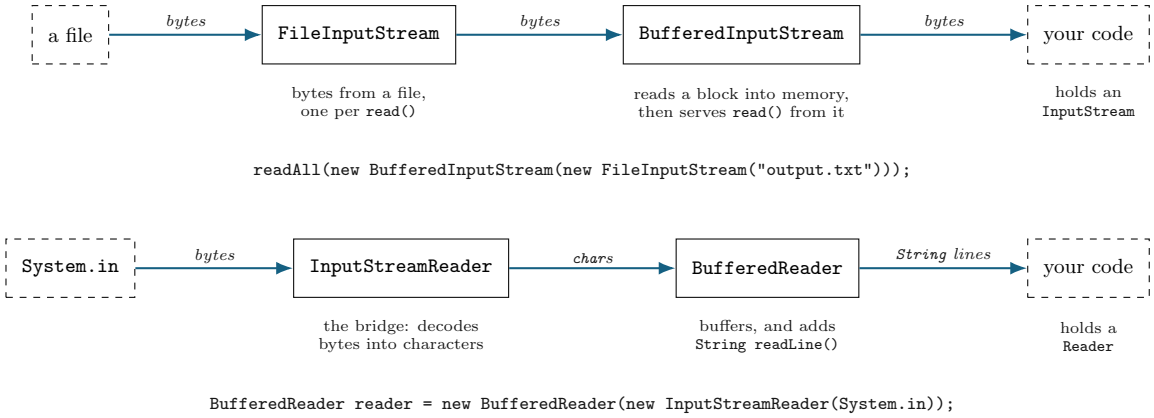


Figure 28: How the nesting in `new BufferedInputStream(new FileInputStream(...))` and `new BufferedReader(new InputStreamReader(System.in))` actually behaves — each wrapper adds one capability and passes the rest through, and your code only ever holds the outermost object. `BufferedInputStream` wraps another input stream; `InputStreamReader` acts as a bridge decoding bytes into characters; `BufferedReader` wraps another `Reader` and adds `readLine()`. The gloss beneath each box says what that stage contributes.

Readers & Writers

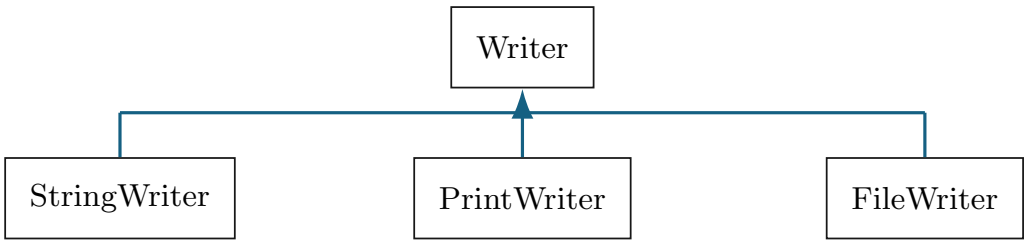
Originally Java I/O was byte-oriented. Byte streams were fine for raw binary data but not ideal for text, so character streams were added in 1.1 to handle text and Unicode characters more conveniently and efficiently.

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

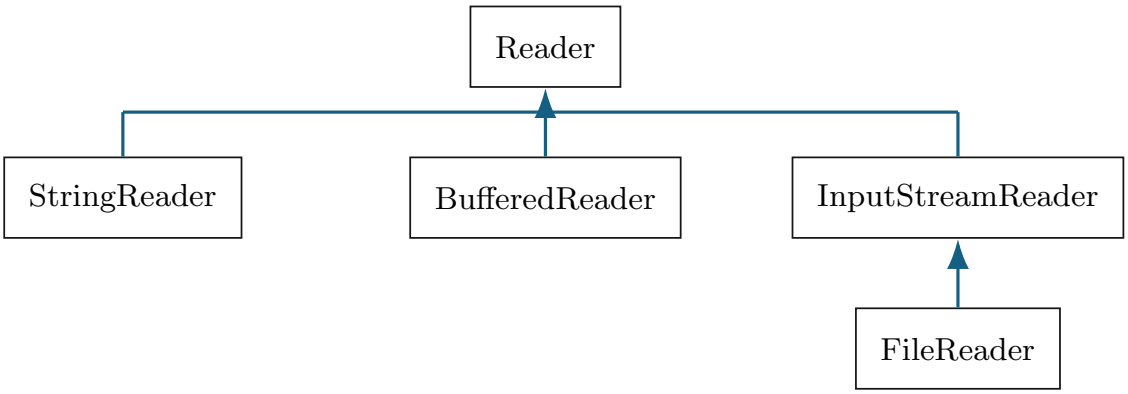
`InputStreamReader` is the **bridge** between the two systems: it wraps an `InputStream` (like `System.in`) and decodes its bytes into characters.

Worked example — read the first 10 characters of input

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



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



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

Figure 29: The character-stream hierarchies: `Writer` and `Reader`, drawn the same way as Figure 27. The one two-level branch is the important one — `FileReader` is a kind of `InputStreamReader`, i.e. reading a text file is itself a case of decoding bytes into characters.

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

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

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

`BufferedReader`

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

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

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

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

The exception handling has to be made explicit: `FileReader`'s constructor throws `FileNotFoundException`, and `readLine()` throws `IOException`, so each needs handling. Below, the second is wrapped in an unchecked `RuntimeException` rather than being declared. See java-exceptions[82].

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

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

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

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

`PrintWriter`

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

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

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

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

`writeroutput.txt` then holds:

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

`flush()`

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

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

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

`Scanner`

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

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

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

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

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

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

Reading strings:

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

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

Further reading: Scanner vs. BufferedReader.

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

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

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

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

Creating a `Path`

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

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

`Files` operations

Category	Methods
Create/delete	`Files.createFile(Path)`, `Files.createDirectory(Path)`, `Files.delete(Path)`, `Files.deleteIfExists(Path)`
Query	`Files.exists(Path)`, `Files.isDirectory(Path)`, `Files.isRegularFile(Path)`
Manipulate	`Files.copy(Path, Path)`, `Files.move(Path, Path)`
Read/write	`Files.readAllLines(Path)`, `Files.readString(Path)`, `Files.write(Path, Iterable<? extends CharSequence>)`, `Files.writeString(Path, CharSequence)`

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

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

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

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

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

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

Summary: which to use?

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

Saving and loading a whole object model to a file is picked up later, in week10-lab-java-io[83].

Java JUnit

Introduced in 2026-03-26-testing[84] (Lecture, Week 5). See java-testing[85] 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.

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.

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() {
```

[83] courses/csse2002/week10-lab-java-io.pdf
[84] courses/csse2002/2026-03-26-testing.pdf
[85] courses/csse2002/java-testing.pdf

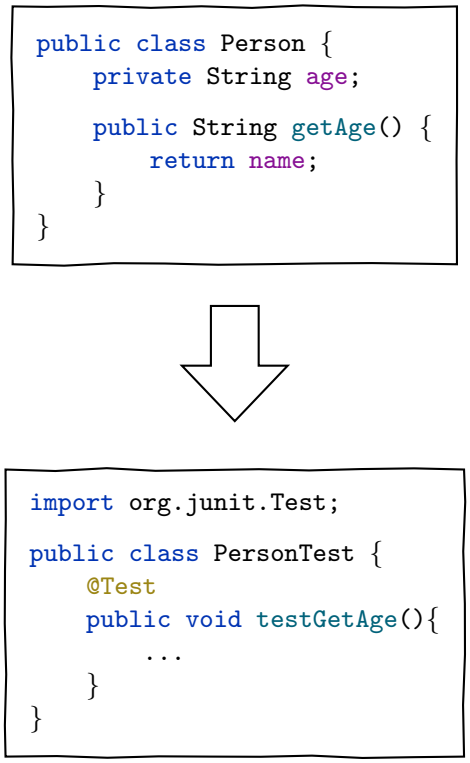


Figure 30: One test class per real class: `Person` produces `PersonTest`, which imports `org.junit.Test` and carries one `@Test`-annotated method per thing being checked. The hollow block arrow pointing down means produces/translates-to. The `Person` snippet does not compile as drawn: `getAge()` returns `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.

```
        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[86].

[86] 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[87]) 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();
```

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

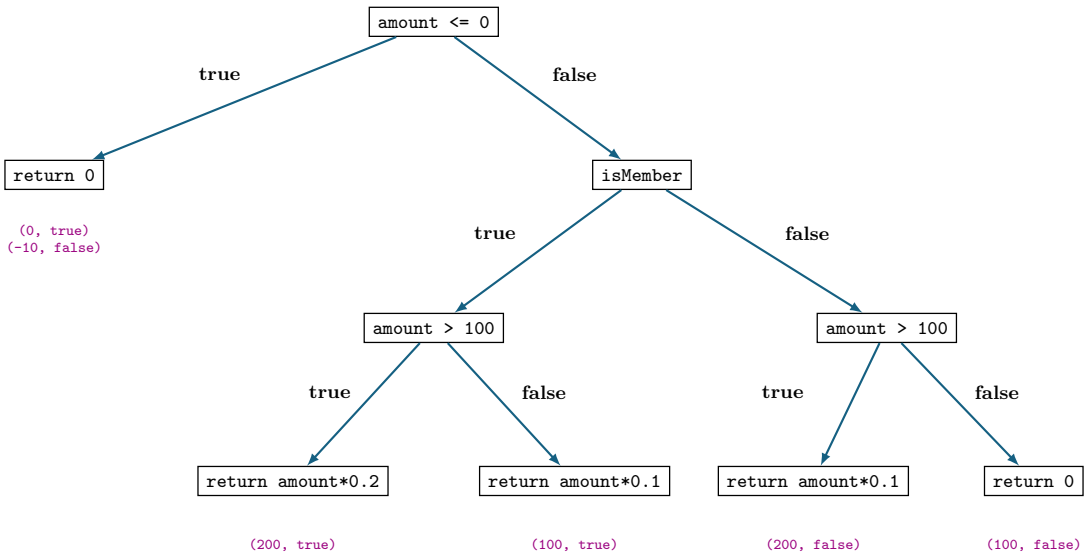


Figure 31: Why `DiscountCalculator` needs exactly five tests: the method as a decision tree, one leaf per test. Each box is a condition from the method or the `return` its branch reaches, so the expected values in the tests can be read off the leaves. The magenta `(amount, isMember)` pair under each 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 Lambdas and Streams

Introduced in 2026-05-14-lambdas-streams-and-events[88] (Guest lecture, Week 11). Practised further in week12-tutorial-lambdas-and-streams[89].

Lambdas (anonymous functions)

A lambda is a **function as a value** — it isn't declared with a name, and can be passed around like any other reference value, such as an object. JavaScript and Python have the same idea.

Scope: like in most languages, a Java lambda **can hold references to things accessible in scope at the time of creation**.

Functional interfaces

Because Java is typed, you cannot have an anonymous function without clearly defined type information — the JVM would not know what to do. So Java supplies a set of interfaces that are **combined with generics** (see java-generics[90]) to give easily-adjustable type descriptions for lambdas. They live in `java.util.function`. Four of them:

Interface	Signature	Purpose
`Consumer<T>`	takes a `T`, returns nothing (`void`)	by convention indicates the function has some kind of side effect — if it returned nothing *and* had no side effect it would do nothing at all
`Predicate<T>`	takes a `T`, returns `boolean`	useful in filtering especially
`Function<T, R>`	takes a `T`, returns an `R`	turning a `T` into an `R`
`BiFunction<T, U, R>`	takes a `T` and a `U`, returns an `R`	deriving one value from two, e.g. the distance between two positions, or custom sorting

A minimal lambda of each shape:

```
Consumer<String> print = s -> System.out.println(s);
Predicate<Integer> isEven = n -> n % 2 == 0;
Function<String, Integer> length = s -> s.length();
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
```

[88] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[89] courses/csse2002/week12-tutorial-lambdas-and-streams.pdf
[90] courses/csse2002/java-generics.pdf

`Consumer<T>` is the one that matters most for events: the toy event system in java-events[91] types every listener as a `Consumer<SimpleEvent>`.

Streams

Added in **Java 8** to enable a different, more **declarative** style of programming. Brian Goetz's IBM article on Java streams is a very good resource on their benefits.

This style is particularly effective when you have data you need to conduct multiple operations on sequentially, or retrieve a single resulting value from a sequence of operations, which is often the case in a real job. At a simple level they are an alternative to writing lots and lots of very samey `for` loops over a sequential set of data (most often a `List` or similar, see java-collections-framework[92]); the JavaScript equivalents are `.filter()`, `.map()` and `.reduce()`.

The worked example

One pipeline, taken a line at a time:

```
int totalHpFromFireEnemies =
    enemies.stream()
        .filter(enemy -> enemy.faction.equals("Fire"))
        .mapToInt(enemy -> enemy.hp)
        .sum();
```

Stage	Line	What it does
`.stream()`	`enemies.stream()`	opens a stream over the collection
`.filter()`	`.filter(enemy -> enemy.faction.equals("Fire"))`	drops every enemy whose faction isn't `Fire`
`.mapToInt()`	`.mapToInt(enemy -> enemy.hp)`	replaces each surviving `Enemy` with its `int hp`
`.sum()`	`.sum()`	reduces what is left to a single total

[91] courses/csse2002/java-events.pdf
[92] courses/csse2002/java-collections-framework.pdf

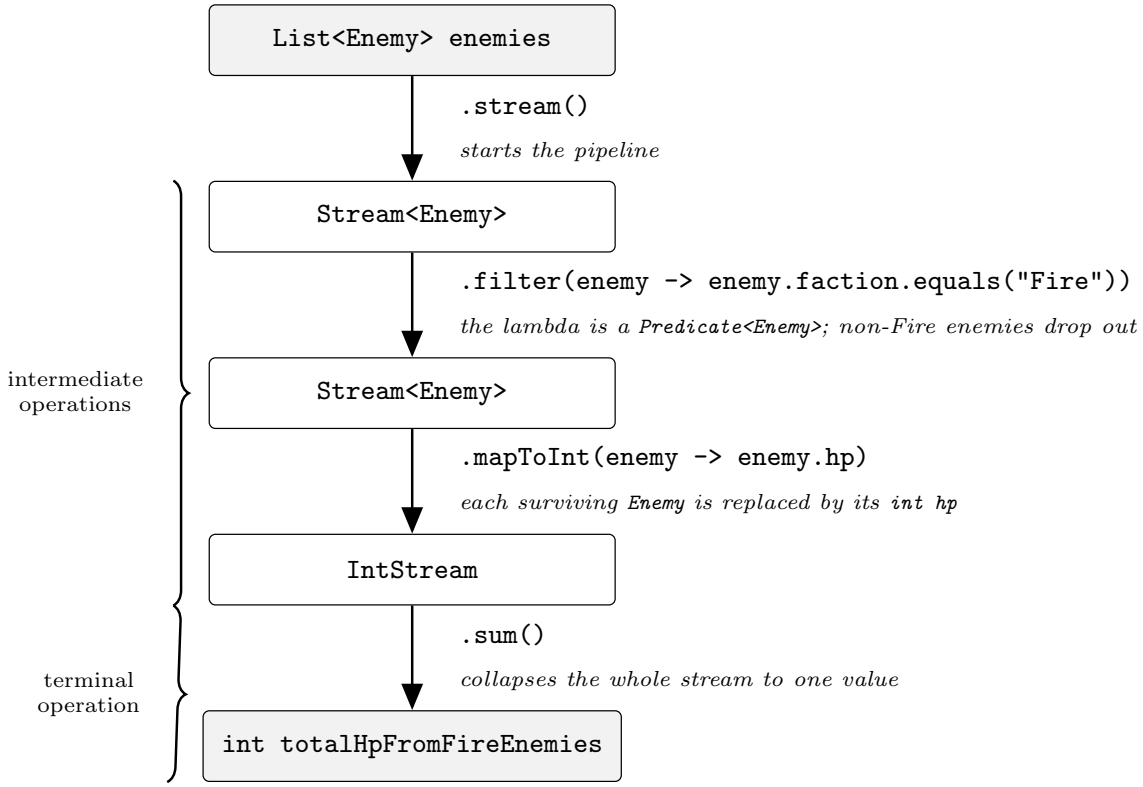


Figure 32: The stream example as a pipeline, one stage per line of the chain. Each box is the type between stages: `.stream()` turns the list into a `Stream<Enemy>`, `.filter()` takes a `Predicate` and hands on a `Stream<Enemy>`, `.mapToInt()` hands on an `IntStream`, and `.sum()` collapses it to one `int`. The braces mark the intermediate operations and the terminal operation.

Anatomy of a stream pipeline

A pipeline has three steps:

1. **Start** a stream with `.stream()`.
2. Chain **any number of intermediate operations**: `.filter()` | `.map()` | `.distinct()` | `.sorted()` | `.mapToInt()` etc.
3. **Resolve** it into a final value with one of the **terminal operations**: `.sum()` | `.count()` | `.toList()` | `.collect()` | `.reduce()`.

You *chain* the first kind and *resolve* with the second. The rule underneath: an intermediate operation hands on another stream, a terminal one resolves to a value. That is what puts `.mapToInt()` in the first list and `.sum()` in the second, and why exactly one terminal operation ends a pipeline.

Java Loop Invariants

Introduced in 2026-05-21-correct-programming[93] (Lecture, Week 12). Extends java-hoare-logic[94]'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	$r \mapsto r,\ b \mapsto b-2$

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

Pick	Action	Effect
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**.

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:

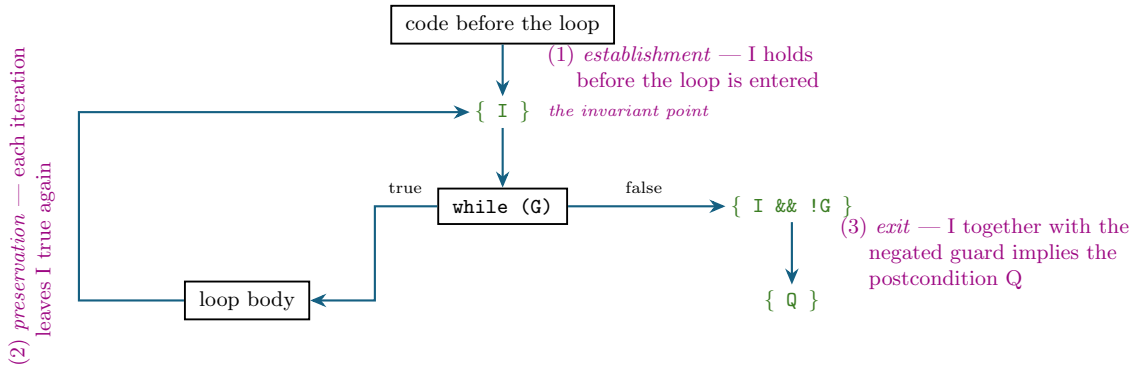


Figure 33: The three obligations in the definition above, each pinned to the point in the loop where it has to be discharged. The invariant point is the top of the loop, before the guard is tested: establishment is the arrow into it, preservation is the arrow back to it, and exit is what the guard's failure adds to it. Connectors are teal and assertions are green.

None of the three clauses in Figure 33 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 ($i \geq \text{exp}$) imply the postcondition? `result = base^i` $\wedge\ i \geq \text{exp} \Rightarrow \text{result} = \text{base}^{\text{exp}}$? **Not quite** — we can only conclude $i \geq \text{exp}$, but we need $i = \text{exp}$.

Step 3 — strengthen. Add the bound $i \leq \text{exp}$ to the invariant: $I : \text{result} = \text{base}^i \wedge i \leq \text{exp}$. Re-checking: I now holds initially iff $\text{exp} \geq 0$ (giving us our precondition), is still preserved by the loop body, and combined with $i \geq \text{exp}$ now forces $i = \text{exp}$ 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 33, 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:

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 $O(\text{exp})$.

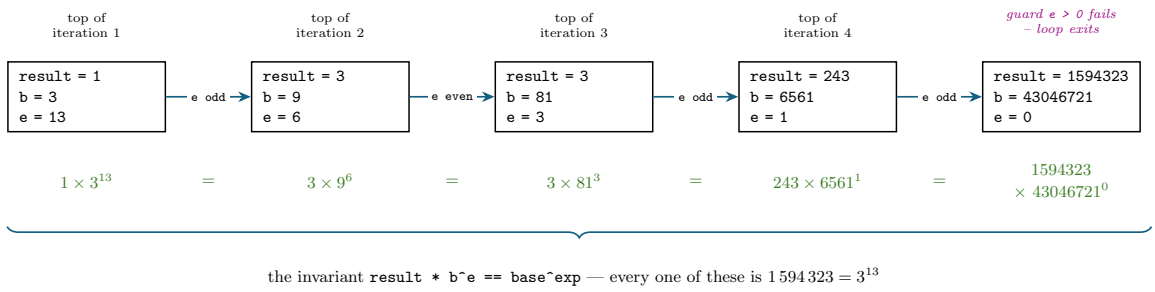


Figure 34: The fast-`power` loop traced on `power(3, 13)`, one box per arrival at the top of the loop. The three variables change on every step; the product `result * b^e` underneath does not, and the five products it takes are exactly the chain the section above works by hand (the section's `3 x 81 x 6561` and this trace's `243 x 6561` are the same number). Connectors are teal and the invariant is green.

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[95]) and the practical is generics (week12-lab-generics[96]) — 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[97]), whose first clause is `students.getCurrentCount() == 0`. Taking the invariant at the top of the iteration with counter `i` to be

[95] courses/csse2002/week12-tutorial-lambdas-and-streams.pdf
[96] courses/csse2002/week12-lab-generics.pdf
[97] courses/csse2002/java-hoare-logic.pdf

`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 33 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.

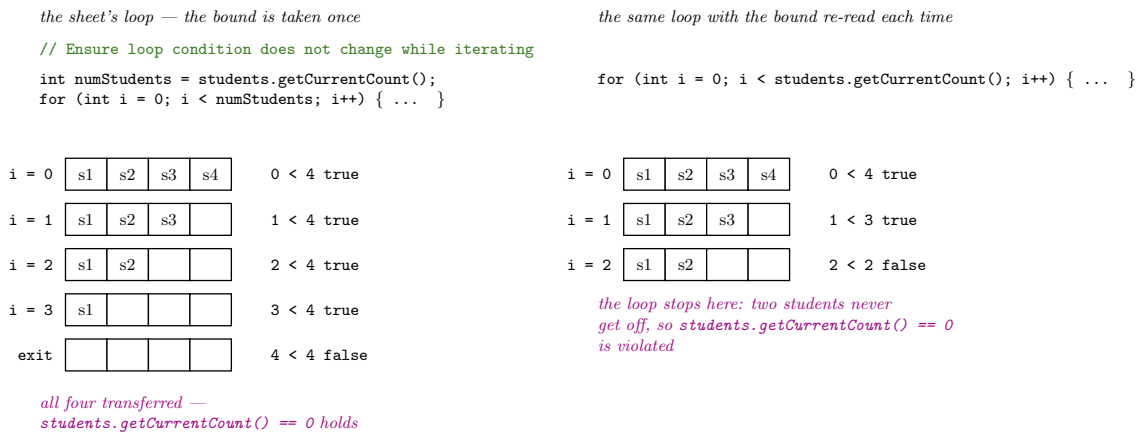


Figure 35: Why the practical snapshots its loop bound. Both columns run the same body on a source bus of four; the left column is the loop with the bound read once into `numStudents`, the right is the same loop with `students.getCurrentCount()` in the guard. Reading the count live makes the guard chase 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.

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 `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[98] for `reduce` itself.

[98] courses/csse2002/java-lambdas-and-streams.pdf

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[99].

> 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 Mutability

Introduced in 2026-03-12-object-oriented-programming-ii[100] (Lecture, Week 3). Builds on java-encapsulation[101].

Mutable vs immutable classes

Objects in Java can either change their state after creation or remain fixed forever.

Two types of class design:

- **Mutable class** — object state *can* change after creation.
- **Immutable class** — object state *cannot* change after creation.

[99] courses/csse2002/week8-tutorial-class-invariants.pdf
[100] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[101] courses/csse2002/java-encapsulation.pdf

Mutable

State is reachable through setters, so it can change at any point in the object's life:

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

    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
}
```

```
class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Alice");
        p.setAge(20);

        p.setAge(30);   // state changed
    }
}
```

This object can change at any time.

Immutable

The immutable version changes four things at once: the class is `final`, both fields are `final`, the values arrive through the constructor, and there are no setters — only getters.

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

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

    public String getName() { return name; }
    public int getAge() { return age; }
}
```

```
class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Alice", 20);
```

```
        Person p2 = new Person("Bob", 30);
        System.out.println(p1.getName() + " " + p1.getAge());
    }
}
```

Changing a value now means constructing a new object, not mutating the old one.

Why `final` on the class

A subclass can introduce mutability (**Effective Java (2nd ed.), Item 16, page 78**). If the class is subclassable, a subclass can add mutable state or override methods to return values that vary, so callers can no longer rely on "an instance of this type never changes". Marking the class `final` is what forecloses that.

Arrays and lists break naive immutability

Even if a class looks immutable, it may not be.

`final` on an array field only prevents **reassigning the reference**. It does nothing to protect the array's *contents*, and a getter that returns the field hands the caller a live handle on the object's own state:

```
public final class Person {
    private final String name;
    private final String[] phoneNumbers;

    public Person(String name) {
        this.name = name;
        this.phoneNumbers = new String[]{"111", "222"};
    }

    public String getName() { return name; }

    public String[] getPhoneNumbers() { return phoneNumbers; }
}
```

```
class Main {
    public static void main(String[] args) {
        Person person = new Person("John");
        person.getPhoneNumbers()[0] = "999999";

        System.out.println(person.getPhoneNumbers()[0]);   // 999999
    }
}
```

Every field is `private final` and there is not a setter in sight, yet the object's state changed. The rule:

- `final` for arrays means the **reference** cannot change, not the contents.
- `final` prevents *reassigning* the array:

```
this.phoneNumbers = new String[]{"123"}; // NOT allowed if already assigned
```

Defensive copying

To protect mutable objects, create copies — on the way in *and* on the way out, so neither the constructor's caller nor the getter's caller keeps a handle on the field (**Effective Java, Item 50: "Make defensive copies when needed"**):

```
public final class Person {
    private final String name;
    private final String[] phoneNumbers;

    public Person(String Name, String[] phoneNumbers) {
        this.name = Name;
        this.phoneNumbers = phoneNumbers.clone();   // defensive copy
    }

    public String[] getPhoneNumbers() {
        return phoneNumbers.clone();                // defensive copy
    }
}
```

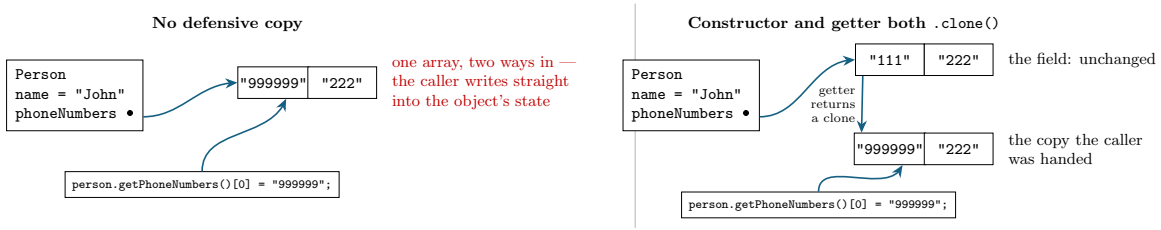


Figure 36: Why a class whose getter returns its array field is not immutable, and what `.clone()` changes. Left: the field and the getter's return value are the same array, so the caller's write lands in the object's state. Right: the getter hands out a copy, so the write lands somewhere the object does not care about.

`clone()` on an array is a shallow copy — it duplicates the row of references, not the objects they point at. That is enough here because `String` is itself immutable, but a `Person[]` of mutable `Person` objects would still leak.

Java Polymorphism

Introduced in 2026-03-12-object-oriented-programming-ii[102] (Lecture, Week 3). Practised in week4-tutorial-method-dispatch-and-casting[103].

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**.

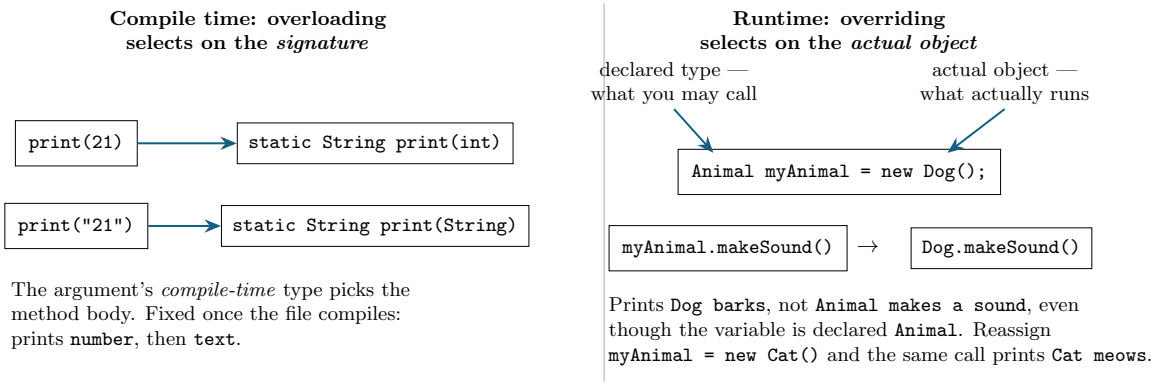


Figure 37: The two selection mechanisms. Left: overloading is resolved by the compiler from the argument's compile-time type, so the choice is fixed once the file compiles. Right: overriding is resolved by the JVM from the object actually on the other end of the reference; the arrow from `myAnimal.makeSound()` 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.

Which one runs is decided from the signature (see java-encapsulation[104]) before the program ever runs:

[102] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[103] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf
[104] courses/csse2002/java-encapsulation.pdf

```java
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[105]).

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.

```java
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"); }
}
```

[105] 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[106].

How it connects to casting

Assigning `new Dog()` to an `Animal` variable is an implicit **upcast** (java-casting[107]). 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 37 are the same reference seen from the compiler's side and the JVM's side.

Java Primitive and Reference Types

Introduced in 2026-02-26-course-overview-and-java-basics[108] (Lecture 1, Week 1).

[106] courses/csse2002/java-abstraction.pdf
[107] courses/csse2002/java-casting.pdf
[108] courses/csse2002/2026-02-26-course-overview-and-java-basics.pdf

Two categories of types

Java distinguishes two sorts of types:

1. **Primitive types** — built-in, fixed representations.
2. **Reference types** — everything else (i.e. classes), including `String`, arrays, and any user-defined class.

Primitive types

Type	Size	Stores
`boolean`	not specified	`true` or `false`
`byte`	1 byte / 8 bits	whole numbers from -128 to 127
`short`	2 bytes / 16 bits	whole numbers from -32,768 to 32,767
`int`	4 bytes / 32 bits	whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes / 64 bits	whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes / 32 bits	fractional numbers, accurate to 6-7 decimal places
`double`	8 bytes / 64 bits	fractional numbers, accurate to 15 decimal places
`char`	2 bytes / 16 bits	a single character/letter

The JVM does not specify a width for `boolean`.

Reference types

Reference types are everything that isn't primitive — classes, including `String` and arrays. **A variable of a reference type stores a reference to an object, not the object's data directly.** That difference is visible in the runtime memory model: a primitive's value lives in the method's stack frame, while a reference type's frame slot holds only an arrow at a heap object.

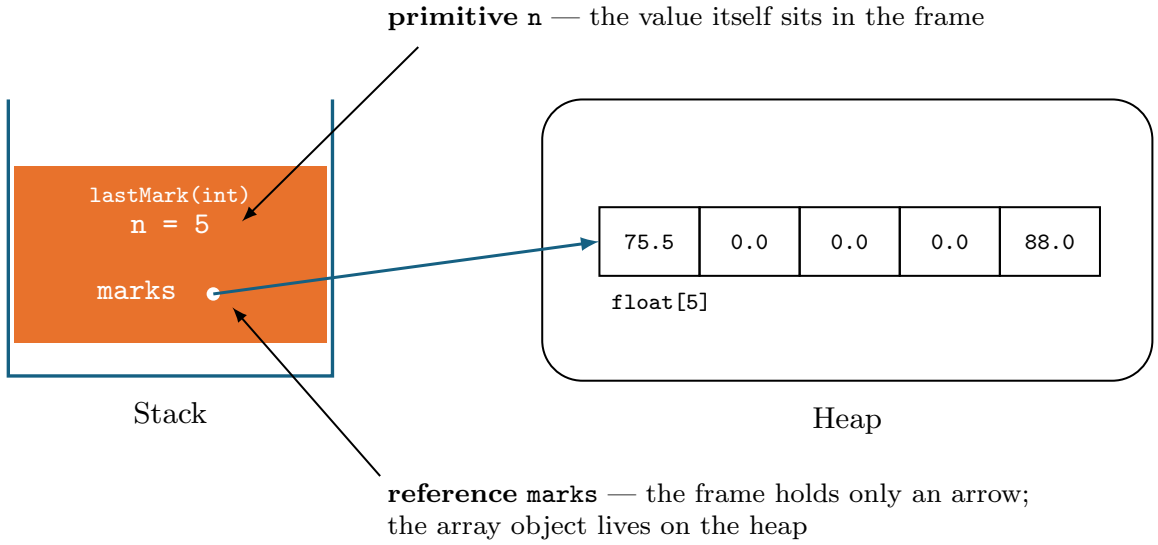


Figure 38: The difference that everything else follows from, in a trace of `lastMark(5)`: the primitive `int n` is stored *as a value* in the stack frame, while the reference `float[] marks` stores only an arrow at an object on the heap. The stack is the open-topped container, the heap is the rounded region, and the reference is the teal arrow.

Why the distinction matters

Assignment and equality mean different things on each side. `x == y` compares stored values only for primitive types:

	Primitive types	Reference types
`x = y`	make `x` store a copy of `y`'s value	make `x` refer to the same object `y` refers to
`x == y`	check if `x` stores the same value as `y`	check if `x` refers to the **same object** as `y`
`x != y`	check if the value stored in `x` differs from `y`'s	check if `x` and `y` reference different objects

So on reference types you almost always want `.equals()`, not `==` — see 2026-02-26-java-basics-part-02-collections-and-strings[109] for the `String` pooling example that makes this concrete, and for how the stack/heap split follows from it.

[109] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf

Wrapper classes

Collections (`Stack`, `List`, `Set`, `Map`) can only store objects/reference types, so Java provides a class for each primitive type:

Primitive	Wrapper class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

`Stack<int>` is illegal; `Stack<Integer>` is what you write. See java-collections-framework[110].

Java SOLID Principles

Introduced in 2026-04-02-refactoring[111] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[112] (Lecture, Week 7 — Liskov Substitution, Interface Segregation, Dependency Inversion).

Why SOLID?

Large software tends to become:

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

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

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

[110] courses/csse2002/java-collections-framework.pdf
[111] courses/csse2002/2026-04-02-refactoring.pdf
[112] courses/csse2002/2026-04-16-solid-principles-ii.pdf

Single Responsibility Principle (SRP)

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

Closely related to java-cohesion-and-coupling[113]'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.

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

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

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[114]) and garbage collection.

Open-Closed Principle (OCP)

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

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

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

A class is kept closed for modification by information hiding (see java-encapsulation[115]).

Example: pluggable grading systems

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

[114] courses/csse2002/java-mutability.pdf
[115] courses/csse2002/java-encapsulation.pdf

```
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();
            }
        }
        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[116].)

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

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

```
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();
    }
}
```

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));
}
```

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`.

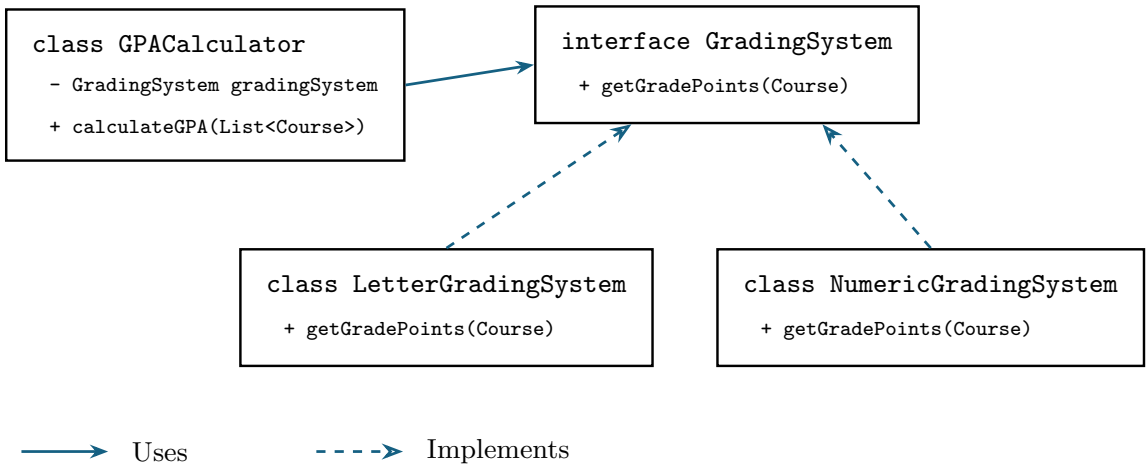


Figure 39: The open-closed refactoring of GPACalculator. GPACalculator uses only the GradingSystem interface; each concrete grading system implements it. A new scheme — pass/fail, say — is a fourth box on the bottom row with a fourth dashed arrow, and no existing box changes: that is 'open for extension, 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.

- 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[117]):

```
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

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

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

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

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

Interface Segregation Principle (ISP)

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

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

```
class Z {
    public void a() {...}
    public void b() {...}
    public void c() {...}
    public void d() {...}
```

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

This mirrors java-cohesion-and-coupling[118]'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:

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

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

Two forms of DIP

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

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

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

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

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

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

Dependency Injection

Dependency Injection (DI) is a design pattern that implements DIP by injecting a class's dependencies into it (e.g. via the constructor) instead of having the class create them internally, promoting loose coupling (see java-cohesion-and-coupling[119]):

[119] 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[120] (Lecture, Week 3). Extended in 2026-03-19-exceptions[121] and formalised later in java-hoare-logic[122]; pre/postconditions come back as class invariants in week8-tutorial-class-invariants[123] and as the substitution principle in week9-tutorial-substitution-principle-and-pre-post-conditions[124].

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
```

[120] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[121] courses/csse2002/2026-03-19-exceptions.pdf
[122] courses/csse2002/java-hoare-logic.pdf
[123] courses/csse2002/week8-tutorial-class-invariants.pdf
[124] 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**.

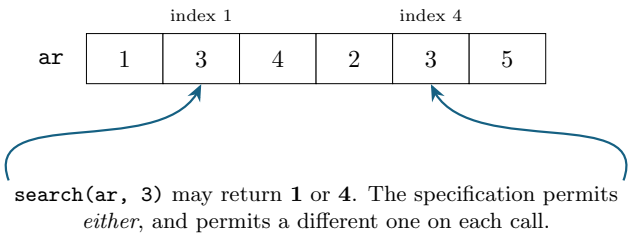


Figure 40: With the spec worded as 'returns an index i such that ar[i] == x', both cells holding 3 are legal answers, and nothing forbids the method returning a different one on each call. That is what makes the specification non-deterministic.

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[125]. 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:

[125] 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.

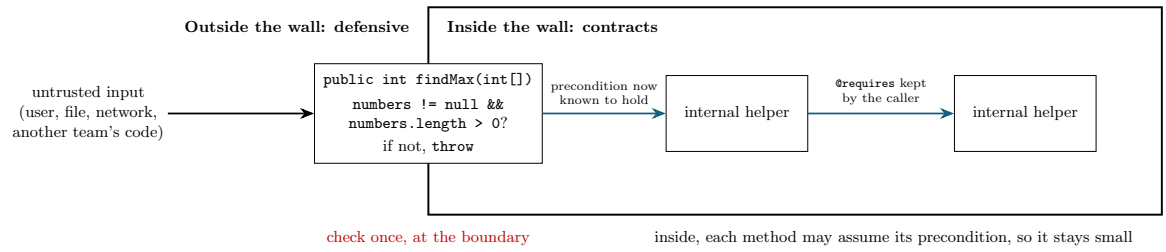


Figure 41: Defensive outside the wall, contracts inside it: validate defensively where untrusted input crosses into the system, and rely on contracts for the internal calls behind that boundary, so the check happens once rather than at every hop. The gate box is the findMax method worked through below.

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[126] (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?

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

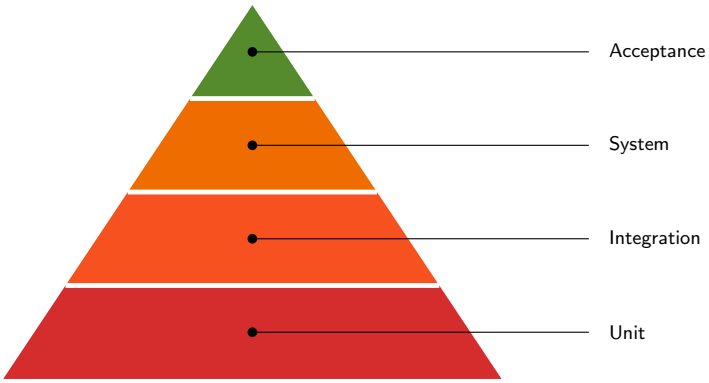


Figure 42: The four levels of testing as a pyramid. Unit tests form the wide base, acceptance tests the narrow apex. In the usual test-pyramid reading, the width of a band stands for how many tests of that kind a project has.

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[127].

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[128]) — what

[127] courses/csse2002/java-junit.pdf
[128] courses/csse2002/java-specification.pdf

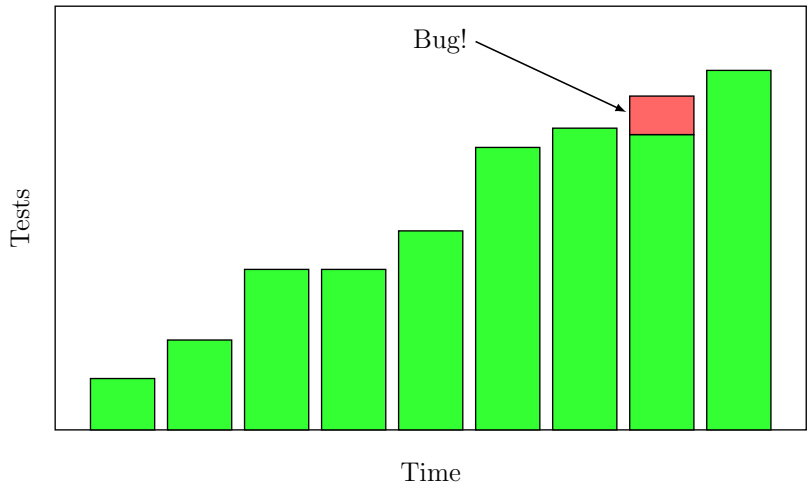


Figure 43: A regression (chart after browserstack.com/guide/regression-testing). Nine bars on *Tests*/*Time* axes: read each bar as a build, and its height as the size of the test suite growing as features are added. The red top slice on the second-to-last bar is the regression, and the arrow labelled 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() {...}
```

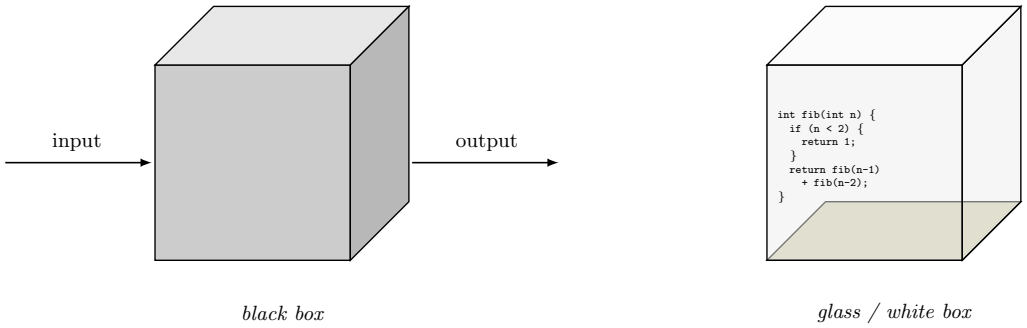


Figure 44: The two testing stances. Both are the same grey cube. The black box is opaque and shows only its input and output arrows. The glass box is translucent, so the implementation inside, a recursive `fib`, can be seen through its walls.

> **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[129], "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.

129 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.

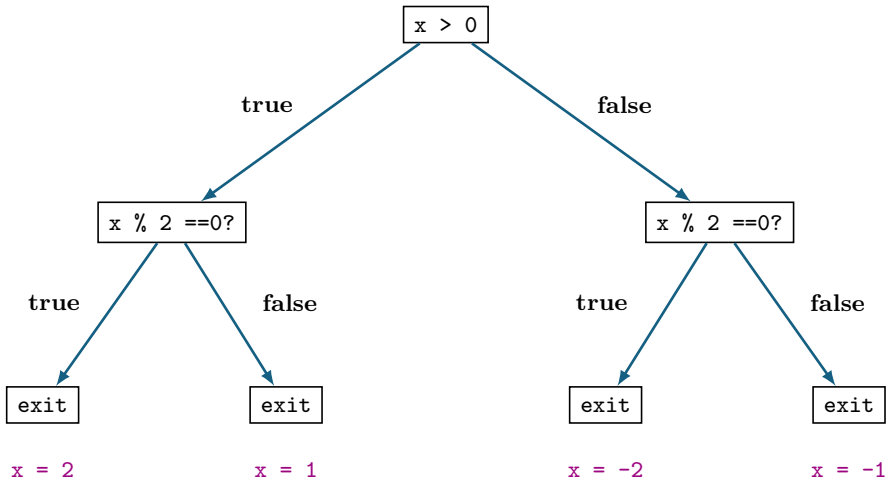


Figure 45: Path coverage for `register`. Each internal box is a condition, each teal arrow an outcome labelled true/false, each leaf one complete path to `exit` — four leaves, so four tests. The magenta input under each leaf is a value of `x` that drives that path.

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[130].

[130] 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".

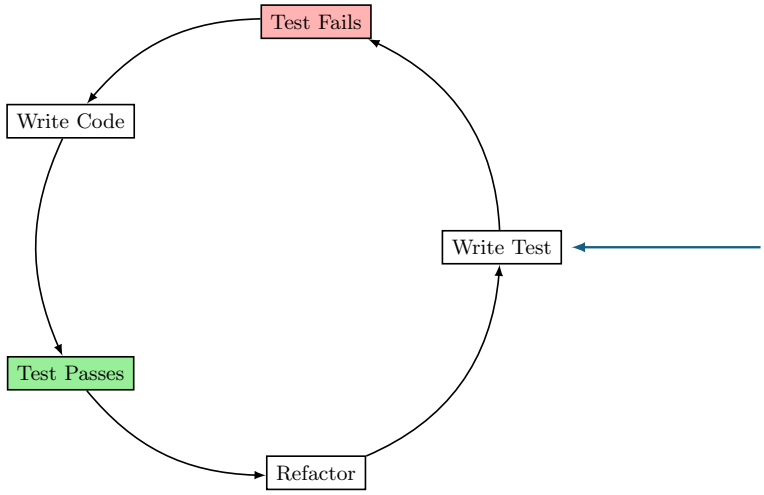


Figure 46: The TDD cycle. Read it counter-clockwise: write a test, watch it fail (pink), write code, watch it pass (green), refactor, and go round again. The teal arrow on the right points at *Write Test* and marks the entry point of the loop: the test comes before the code.

The *Refactor* step connects to java-refactoring[131]: the green test suite is what makes it safe to restructure code without changing its behaviour.

Version Control

A Week 4 supplementary topic, and the only one in the course that isn't about Java. **Not examinable** (confirmed in 2026-05-28-course-summary-and-exam-preparation[132]) — but it's the piece of the course you'll use every day afterwards.

Why

Software changes frequently, and version control preserves the history of those changes. The everyday shape of the thing is a word processor's version-history sidebar: a list of edits, each

[131] courses/csse2002/java-refactoring.pdf
[132] courses/csse2002/2026-05-28-course-summary-and-exam-preparation.pdf

entry timestamped and attributed to a named collaborator, with a **Restore this version** button.

The alternative most students reach for — copying files into another directory — is *incredibly error prone*. It ends up as a folder of `Super Cool Report v1.xlsx … v3.1 … v4a … v4b … vFinal … vFinal_1 … vFinal_Final … vFinal_Final-UPDATED_NEW.xlsx`, or a set of directories running `project-v0.1` through `project-v2.3` with a `project-v2.1-backup` wedged in the middle.

In real development we want three things that copying can't give us:

1. **To work on our own version and send changes back (commit) when they work.** We don't want everyone editing the same code in real time; the software would rarely compile while others were working on it.
2. **Labelled changes** — a record of what changed and why, so bad changes are easy to find and undo. Changes frequently introduce new bugs.
3. **To work with any text files**, not just source.

What a VCS does

A **version control system** tracks a folder and its contents, the history of changes within it, and metadata about that history (when a change was made, and by whom). History is recorded as **snapshots** — the entire contents of the folder at one point in time.

Benefits in this course:

- Inspect old versions of your code — *"this was working before!"*
- Identify what you changed recently — and remember, be suspicious of recently changed code when debugging (see week5-lab-debugging[133]).
- Practice for your future career.

Terminology

Term	Meaning
Version control	Tracking the history of changes such that old versions can be restored
Repository	A folder placed under version control
Commit	A snapshot of a repository at a point in time
History	A list of commits in a repository

By market share: Git 93.87%, Subversion 5.17%, none 4.31%, Mercurial 1.13%.

[133] courses/csse2002/week5-lab-debugging.pdf

When to commit

Commit when you **create something that works**, or **fix something that was broken**.

Commit	Do not commit
Finished the dependencies method of `addCustomer`	It's 5 o'clock
Fixed the code style of `AddressBook`	Changing to a new computer
Fixed a bug in `Farm.core`	Completed the assignment
Fixed a typo in the description of the `update` method	

The pattern: the left column describes *a change to the code*; the right column describes *something that happened to you*. A commit message should be about the former.

The staging area

To prepare a commit you **add** your changes to the **staging area** — the middle ground between what you have done to your files (the **working directory**) and what you last committed.

What should be committed?

Not everything in a folder belongs in version control. A repository should contain **everything required to build or run the project, and not more**.

- Beware **generated files** — if you don't edit the file, don't commit it.
- Beware **proprietary formats** like PowerPoint or Word documents — if your file is not plain text, consider leaving it out.
- Tell Git what to skip with a `.gitignore`.

Common commands

Command	Does
`git init`	Create a new Git repository
`git add`	Add working changes to the staging area
`git commit`	Commit the changes in the staging area
`git log`	View the repository history
`git diff`	View working changes

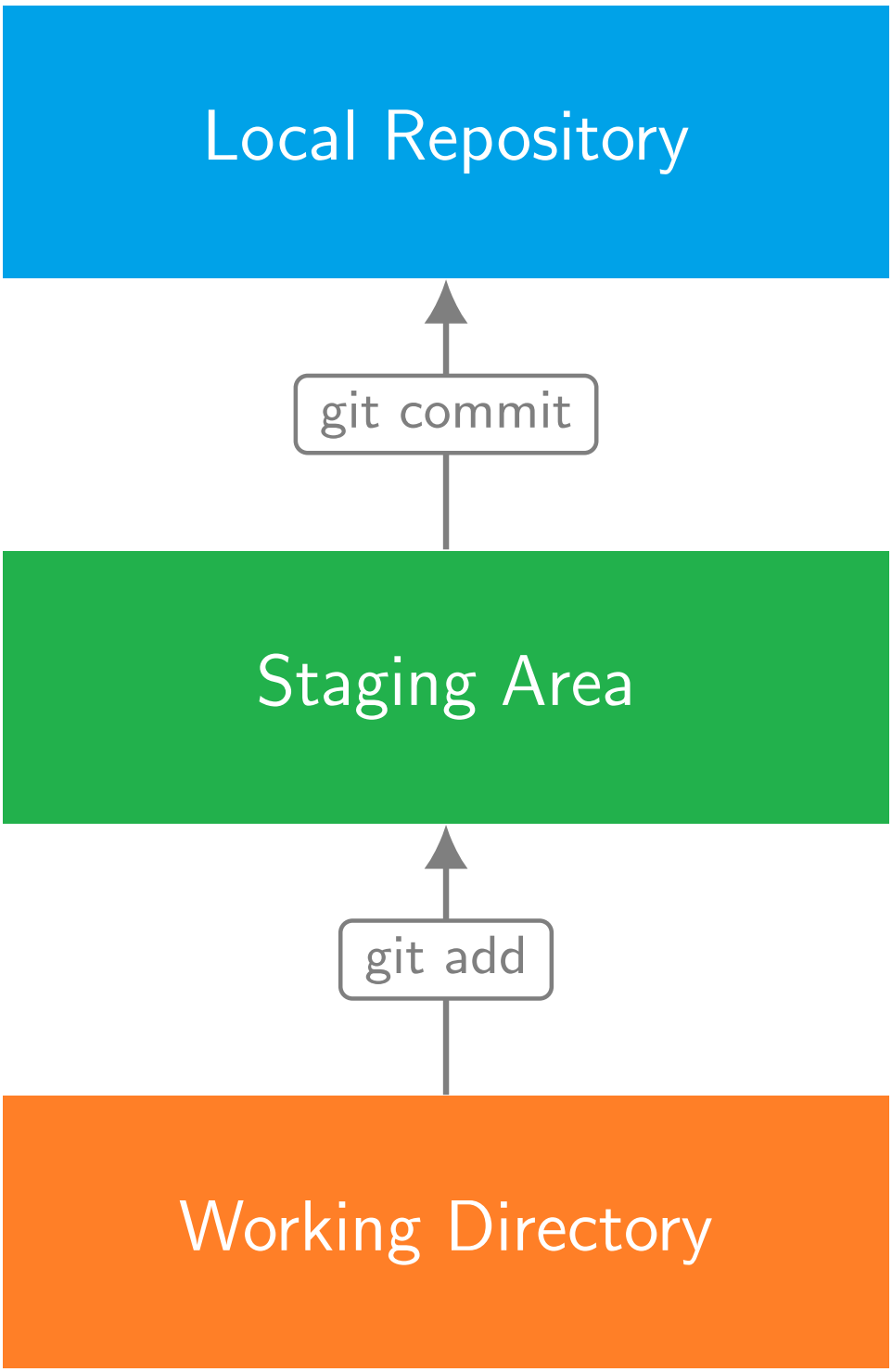


Figure 47: The three places a change can be, and the one command that moves it between each pair. This is the whole reason the staging area is a separate idea: `git add` does not save anything permanently, and `git commit` does not look at your files, it looks at what you staged.

Remotes — and a warning

A repository can be stored on a remote site; popular ones are GitHub, GitLab and BitBucket. An account with a lot of contributions can look good on a resume, but the rule that matters here is:

> **Do not make assessment public on these websites.**

Publishing coursework counts as showing your code, which csse2002[134] lists explicitly under misconduct.

Further learning

Git is a powerful tool you'll use extensively, and there is much more to it than the above. To go further, use the Git edX courses (accessible with a UQ student account) and the Atlassian Git tutorials.

[134] courses/csse2002/csse2002.pdf