CSSE2002 — Week 9 Notes

Table of contents

Generics in Java **2**
- Outline . . . 2
- Why generics . . . 2
- Generic methods, and the two exercises . . . 4
- Bounded generics . . . 7
- Bounds through wildcards . . . 7
- Type erasure . . . 8
- Practical example: the bookshop . . . 8
- Homework . . . 8
- Further reading . . . 8
- Applied class . . . 9
- Practical . . . 9

Refactoring **9**
- Setup . . . 10
- Refactoring in the small . . . 10
- Refactoring in the large . . . 16

Substitution Principle and Pre/Post Conditions **17**
- Predicate strength . . . 17
- Substitution principle . . . 19
- Writing pre/postconditions . . . 21

Reference material **23**
- CSSE2002 Diagram & Notation Conventions . . . 23
- Java Casting . . . 35
- Java Cohesion and Coupling . . . 38
- Java Collections Framework . . . 44
- Java Exceptions . . . 47
- Java Generics . . . 54

Java Inheritance 66
Java Refactoring 70
Java SOLID Principles 77
Java Specification 89
Version Control 100

Generics in Java

See csse2002[1] for course logistics.

Outline

1. Generics
2. Bounded Generics
3. Bounds through Wildcards
4. Type Erasure
5. Practical Example

Definitions, the full listings and the bookshop worked example are in java-generics[2]. What follows is the order of the argument and the exercises.

Why generics

There are two reasons for generics, and they have nothing to do with each other. Keep them apart:

1. **Type safety** — without a declared element type, a collection accepts anything and hands everything back as `Object`, so the caller casts, and a wrong cast is a run-time crash.
2. **Reuse** — without a type parameter, a class that works on one type has to be copied once per type.

Everything after that is the same parameter `T` being progressively *constrained*: unbounded (`<T>`) → bounded at the declaration site (`<T extends Number>`) → bounded at the use site (`? extends T`, `? super T`) → and finally erased entirely, so none of it survives compilation.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-generics.pdf

Why 01 — the cast you should not have to write

Prior to Java 1.5 collection classes could hold any type of object: `ArrayList` declared `public boolean add(Object o)`. Every element comes back as `Object`, so each `get` needs a cast, and the compiler cannot check that the cast is right.

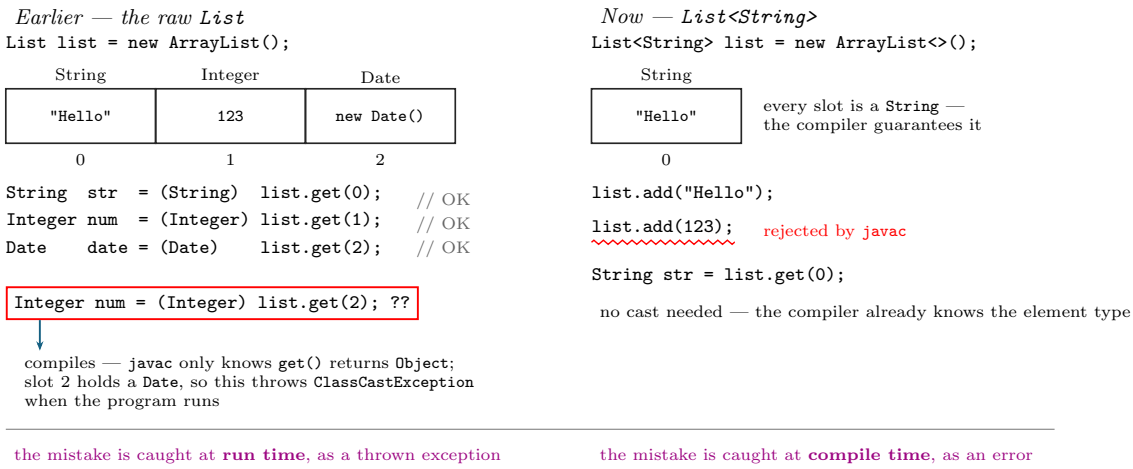


Figure 1: Type safety with and without generics. Left: a raw `List` holding a String, an Integer and a Date, with each cell's type above it and its index below. The three casts that match the element types all succeed. The boxed line, `Integer num = (Integer) list.get(2);`, compiles, but slot 2 holds a `Date`, so it throws `ClassCastException` when the program runs. Right: the same list parameterised by `String`, where the bad line `list.add(123)` is refused before the program ever runs and `get` needs no cast. The red squiggle marks a compile-time error. The magenta lines give the verdict: the raw list's mistake is caught at run time, the generic list's at compile time.

Generics **provide type safety**. Type safety allows us to catch and prevent type-related errors early and thus prevent unintended bugs. A shopping cart is a second example: a raw `List cart` accepts `new Product("Laptop")` and then, by accident, `"Not a product"`. The generic version turns that second `add` into a compile-time error and drops the `(Product)` cast from the loop.

See java-casting[3] for what the cast itself is doing, and java-exceptions[4] for `ClassCastException`.

[3] courses/csse2002/java-casting.pdf
[4] courses/csse2002/java-exceptions.pdf

> **Fun fact: a list of `Object`**
>
> `List<Object> list = new ArrayList<>();` accepts both `"Hello"` and `123`, and both come back as `Object`. What that shows is written up in java-generics[5].

Why 02 — the class you should not have to copy

The second motivation is duplication. A class that prints an `Integer` has to be written again for `Double`, `Float` and `String`. The four classes differ in one word, and one generic class replaces them.

Generics enable **reference types** — `Integer`, `String`, … and user-defined types — to be *parameters* when defining classes, interfaces and methods. The word *reference* matters: `Print<int>` is not legal, only `Print<Integer>`.

Naming the type parameter

The convention is a single letter. These show up often in the Java libraries:

Letter	Stands for
`T`	Type
`E`	Element — used extensively by the Java Collections Framework (java-collections-framework[6])
`K`	Key
`V`	Value
`N`	Number
`S`, `U`, `V`…	2nd, 3rd, 4th types

Generic methods, and the two exercises

A *method* can carry its own type parameter, declared between the modifiers and the return type, whether or not its class is generic:

[5] courses/csse2002/java-generics.pdf

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

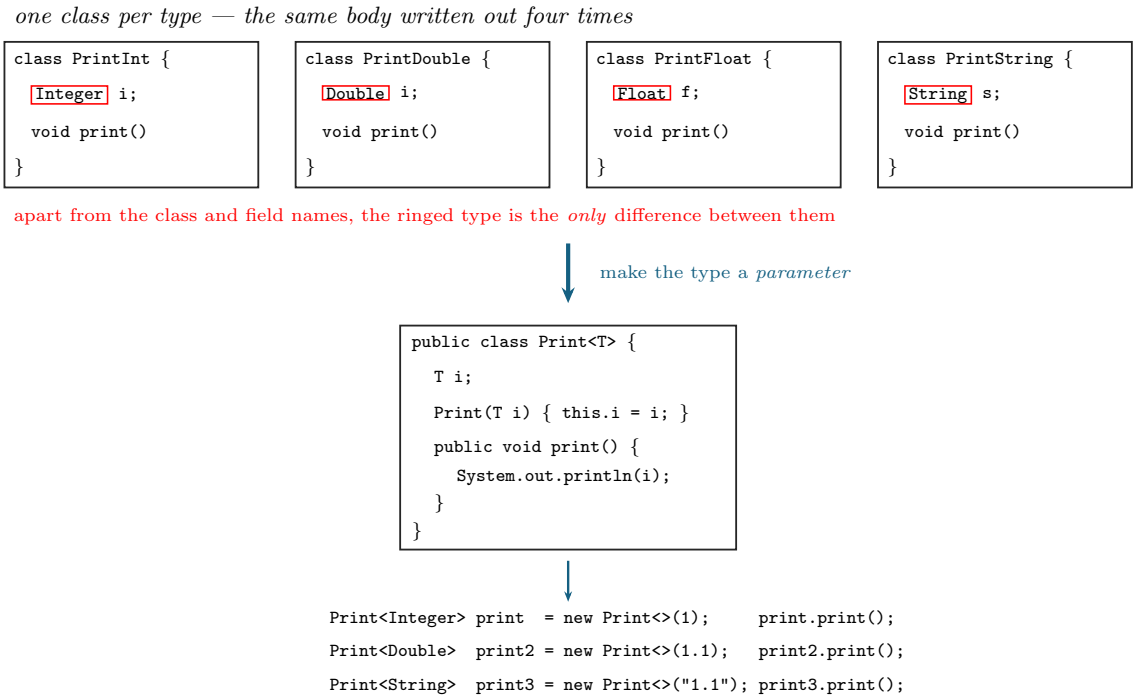


Figure 2: Reuse: four classes whose bodies are identical apart from one word, and the single generic class `Print<T>` that replaces them. Apart from the class and field names, the type outlined in red (`Integer`/`Double`/`Float`/`String`) is the only difference between the four. Each of the four is a `public class` and also carries a constructor and a one-line `print()` body (`System.out.println(...)`); all of that is elided here so the four fit in one row. At the bottom, three instantiations of the generic class with three different element types.

```java
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
}
```

Several at once are allowed, and the return type may be a third one:

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

String result = genericMethod(42, 3.14);   // T = Integer, S = Double, R = String
```

The two exercises take **one** method — `Printer.print` — through three signatures. Exercise 02 is an edit to Exercise 01's answer, so revise them as one sequence.

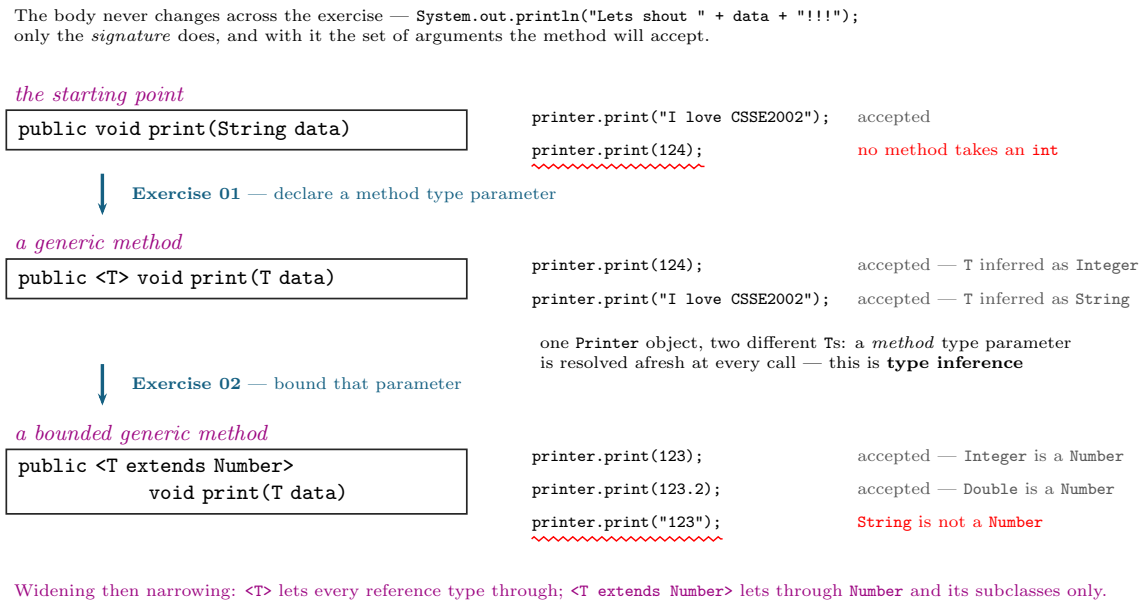


Figure 3: Exercises 01 and 02 as one progression. The body is the same at every stage; only the signature changes, and with it the set of arguments the method accepts. A red squiggle marks a call the compiler rejects. At the starting point `printer.print(124)` is rejected, and removing that restriction is what Exercise 01 exists to do. Exercise 02 then bounds the parameter, so the String call on the bottom row is rejected. The teal arrows are the two exercises, each a change of signature.

At the middle stage the Java compiler automatically determines the type of `T` from the type of the argument passed to the method. This is known as **type inference**.

Bounded generics

In general, the type parameter can accept any reference data type. Bounded generics are used if we want to use generics for some specific types.

The form is `<T extends X>`: `T` can only accept data that are subtypes of `X`. It is the java-inheritance[7] relation being used as a constraint. The example classes are `Person`, `Student` and `Professor`.

Despite the keyword, `extends` in a bound also covers *implementing an interface*. There is no `implements` form of a bound. This is needed to read library signatures.

Bounds through wildcards

The bound above sits at the *declaration* site. Wildcards move the same idea to the *use* site — a parameter list. There are three forms:

Form	Name	Use
`List<?>`	unbounded wildcard	read only; writing is refused outright
`? extends T`	upper bound wildcard	mainly to read data from a structure
`? super T`	lower bound wildcard	to perform write operations

Choosing between `<T>` and `<?>`:

- 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.

The reason upper bounds are needed at all is **invariance** (*Effective Java*, Item 31): parameterized types are invariant, so `List<Type1>` is neither a subtype nor a supertype of `List<Type2>`. So `sum(List<Number>)` rejects a `List<Double>` even though `Number n = 5.0;` is fine. Widening the parameter to `sum(List<? extends Number>)` fixes it.

java-generics[8] carries the code for all three forms, the `addSomething` / `sum` / `addNumbers` examples, and a figure of the `Number` tree showing which lists each wildcard admits.

[7] courses/csse2002/java-inheritance.pdf
[8] courses/csse2002/java-generics.pdf

Type erasure

All of the above is **compile time only**. Java implements generics by erasing every type parameter during compilation — replacing it with its upper bound, or with `Object` if unbounded — so generic type information is not available at run time. The compiler then inserts the casts the erased code needs. The `(String)` cast that generics took out of the source is still there in the compiled code; you just did not type it.

The two erasure examples and the compile-time/run-time pair are drawn out in java-generics[9].

Practical example: the bookshop

A worked design problem: a bookshop selling `Fiction`, `Action` and `Fantasy` books held in bookshelves. There are three designs: one `BookShelf` of `Book` (reusable, needs casts, and one line throws), a separate shelf class per genre (type safe, duplicated), and finally `BookShelf<T extends Book>` (both). The listings and a figure comparing the three designs are in java-generics[10].

The bound is needed in that last version: `displayBooks()` calls `book.getGenre()`, which is only legal because the compiler knows every `T` is at least a `Book`. An unbounded `BookShelf<T>` would not compile.

Homework

- **What if we use a generic class instead?** Rewrite the `Printer` exercise with the type parameter on the *class* rather than the method, then work out why `printer.print(123)` is now an error on a `Printer<String>`. The reasoning is written up in java-generics[11].

Further reading

- https://docs.oracle.com/javase/tutorial/java/generics/index.html
- https://www.baeldung.com/java-generics
- https://youtu.be/K1iu1kXkVoA

The course's CodeCamp playlist has supplementary material, and the course Git repo has the code samples.

Generics come back in the Week 12 practical, week12-lab-generics[12].

[9] courses/csse2002/java-generics.pdf
[10] courses/csse2002/java-generics.pdf
[11] courses/csse2002/java-generics.pdf
[12] courses/csse2002/week12-lab-generics.pdf

Applied class

See week9-tutorial-substitution-principle-and-pre-post-conditions[13] — more practice with the Liskov Substitution Principle (from java-solid-principles[14], Week 7) and pre/postconditions (from java-specification[15], Week 3). It runs in three parts: ranking six pairs of predicates by **strength**; deciding, for each of those pairs, whether putting them in as *preconditions* or as *postconditions* keeps substitutability; and then writing Javadoc for a small `q2` method — `@param`/`@return` first, then `@requires`/`@ensures`, then a frame condition saying the method changes nothing in the array.

The worked `X`/`Y`/`Z` example is the one to remember: `Y` widens the precondition and narrows the postcondition, so it is legal; `Z` narrows the postcondition correctly but *also* narrows the precondition (`> 5` instead of `>= 5`, forbidding 5), and that is the violation.

Practical

See week9-lab-refactoring[16] — refactoring in the small and in the large, extending java-refactoring[17] (Week 6). The "in the small" half is a Connect 4 class taken through style fixes, better names, `while`→`for`, magic numbers to constants, decomposition into helper methods, comments, and finally removing the duplicated win-checking code into `checkWin` / `checkHorizontalWin` / `checkVerticalWin`. The "in the large" half breaks a monolithic `Library` class into `Book`, `BookShelf`, `Borrower` and `BorrowingSystem` — see java-cohesion-and-coupling[18].

That practical solution lands on `Book` and `BookShelf` — the same two classes the bookshop generics example is built from, arrived at from cohesion rather than from type safety.

Refactoring

Practical for 2026-04-30-generics-in-java[19] (Week 9). Extends java-refactoring[20] (Week 6) with a full worked refactoring of a Connect 4 implementation, plus a "refactoring in the large" exercise using java-cohesion-and-coupling[21] and java-solid-principles[22] (SRP, DIP).

[13] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf
[14] courses/csse2002/java-solid-principles.pdf
[15] courses/csse2002/java-specification.pdf
[16] courses/csse2002/week9-lab-refactoring.pdf
[17] courses/csse2002/java-refactoring.pdf
[18] courses/csse2002/java-cohesion-and-coupling.pdf
[19] courses/csse2002/2026-04-30-generics-in-java.pdf
[20] courses/csse2002/java-refactoring.pdf
[21] courses/csse2002/java-cohesion-and-coupling.pdf
[22] courses/csse2002/java-solid-principles.pdf

Setup

The provided `.zip` contains:

```
resources/oTowerWin.txt
resources/simpleWin.txt
resources/towerWin.txt
src/library/Library.java
src/Connect4.java
test/Connect4Test.java
```

`Connect4` has one method, `play`, implementing (most of) the gameplay; the three `resources/*.txt` files are recorded games the tests replay. `Library` is a separate library-management system, used for the second half. Set up version control *before* refactoring (see java-version-control[23]), so each refactoring step is a separate commit that can be reverted if it breaks something.

The running program prints the seven column headings `0123456`, then the board, then prompts the current player (`Player O: 6`, `Player X: 0`, …) for the column to drop into, and finally prints `Player X wins`. That console transcript is the *external behaviour* the refactoring must not change.

Refactoring in the small

Goal: make an existing, working class more **readable and understandable** without changing its behaviour — the unit tests must keep passing after every change. "In the small" refactorings touch one method/class's internals, not the overall class structure.

Checklist:

1. Code adheres to the course style guide.
2. Complex operations are explained.
3. Logical blocks of code are grouped with whitespace.
4. Methods have been decomposed such that each method has one clear purpose.
5. Appropriate data structures are used.

> **Working**
>
> Each item below demonstrates *one* type of refactoring; none is fully refactored or perfectly styled, and the list is a sample rather than an exhaustive treatment.
> **Style guide fixes** (vertical and horizontal whitespace) — trivial with a linter like

[23] courses/csse2002/java-version-control.pdf

Checkstyle:

```
// Before
int x=0;
while (x < 7){out.print(x); x++;}

// After
int x = 0;
while (x < 7) {
    out.print(x);
    x++;
}
```

Bad naming — replace cryptic names with meaningful descriptions:

```
// Before
public static void play(Scanner i, PrintWriter out) {
    int[] n = new int[7 * 6];
    int[] m = new int[7 * 6];
    boolean t = false;
}

// After
public static void play(Scanner in, PrintWriter out) {
    int[] boardX = new int[7 * 6];
    int[] boardO = new int[7 * 6];
    boolean isTurnX = false;
}
```

`while` loops that execute a fixed number of times — replace with `for`:

```
// Before
int x = 0;
while (x < 7) { out.print(x); x++; }
x = 0;
while (x < 42) { ... x++; }

// After
for (int x = 0; x < 7; x++) { out.print(x); }
for (int x = 0; x < 42; x++) { ... }
```

Magic numbers — replace with named constants:

```
private static final int ROW_SIZE = 7;
private static final int COL_SIZE = 6;

for (int i = 0; i < ROW_SIZE; i++) { out.print(i); }
for (int i = 0; i < ROW_SIZE * COL_SIZE; i++) { ... }
```

Decomposition — even code that isn't duplicated can be pulled into a helper method if it has a single, nameable purpose (here, printing the board):

```
private static void printBoard(int[] boardX, int[] boardO,
                               PrintWriter out) {
    for (int i = 0; i < ROW_SIZE; i++) { out.print(i); }
    for (int i = 0; i < ROW_SIZE * COL_SIZE; i++) { ... }
}

public static void play(Scanner in, PrintWriter out) {
    ...
    printBoard(boardX, boardO, out);
    ...
}
```

Comments — explain complex lines or the purpose of a block, not what is already obvious from the code:

```
for (int s = 0; s < 42; s++) {
    boolean nf = true;
    boolean mf = true;
    // Check horizontal win
    for (int q = 0; q < 4; q++) {
        if ((s + q) / 7 != s / 7) {
            // Found row out of bounds, not a win
            nf = false;
            break;
        }
        if (boardX[s + q] == 0) {
            // Found position that isn't an X, not a win
            nf = false;
        }
    }
    ...
}
```

Duplication — repeated blocks that do the same thing become helper methods whose differences are the parameters. The original win check repeats a near-identical horizontal/vertical scan for *both* players, and is refactored in two stages.

Stage 1 — extract a `checkWin(int[] playerBoard)` helper parameterised on which player's board to check, removing the per-player duplication:

```
private static boolean checkWin(int[] playerBoard) {
    boolean win = true;
    // horizontal win
    for (int q = 0; q < 4; q++) {
        if ((s + q) / 7 != s / 7) { return false; }     // out of bounds
        if (playerBoard[s + q] == 0) { win = false; }   // not the player's
         ↪  piece
    }
    if (win) { return true; }

    win = true;
    // vertical win
    for (int q = 0; q < 4; q++) {
        if (s + (q * 7) >= 42) { return false; }
        if (playerBoard[s + (q * 7)] == 0) { win = false; }
    }
    return win;
}

// in play():
for (int s = 0; s < 42; s++) {          // check win
    if (checkWin(boardX)) { out.println("Player X wins"); exit = true; }
    if (checkWin(boardO)) { out.println("Player O wins"); exit = true; }
}
```

Stage 2 — split `checkWin` again into `checkHorizontalWin`/`checkVerticalWin`, each returning as soon as a check fails. This is mostly about comprehension, but it also removes the need to store a `win` variable at all:

```java
private static boolean checkHorizontalWin(int[] playerBoard) {
    for (int q = 0; q < 4; q++) {
        if ((s + q) / 7 != s / 7) { return false; }
        if (playerBoard[s + q] == 0) { return false; }
    }
    return true;
}

private static boolean checkVerticalWin(int[] playerBoard) {
    for (int q = 0; q < 4; q++) {
        if (s + (q * 7) >= 42) { return false; }
        if (playerBoard[s + (q * 7)] == 0) { return false; }
    }
    return true;
}

private static boolean checkWin(int[] playerBoard) {
    return checkHorizontalWin(playerBoard) || checkVerticalWin(playerBoard);
}
```

Data structures — the flat 42-slot array representing the board is clumsy. Three things to try (partly a matter of personal preference):

1. Change the flat array of 42 slots to a nested array of 6 rows with 7 slots (`int[] boardX = new int[7 * 6];` becomes `int[][] boardX = new int[7][6];`). This breaks every existing use of the board, so a lot of the class has to be (gradually) rewritten.
2. Combine the two per-player arrays into one board — `1` at a position means X, `2` means O. This avoids the two data structures accidentally getting out of sync.
3. Turn the array into a `boolean[]`. Since `boardX` and `board0` only ever store `1` or `0`, they are effectively boolean arrays already.

Why the win check indexes the way it does

The two guards in `checkWin` are the least obvious lines in the class, and they are exactly what the "comments" refactoring above was for: both exist because the board is *flat*, so a run of four can silently walk off the edge of one row and onto the next, or off the end of the array.

This is also the clearest argument for the "data structures" refactoring: with `int[][] boardX = new int[7][6]`, the horizontal guard becomes a plain column-index bound and the `/7` arithmetic disappears — which is why it is worth the rewrite it forces.

`int[] boardX = new int[7 * 6];` — one flat array, 7 slots per row, 6 rows

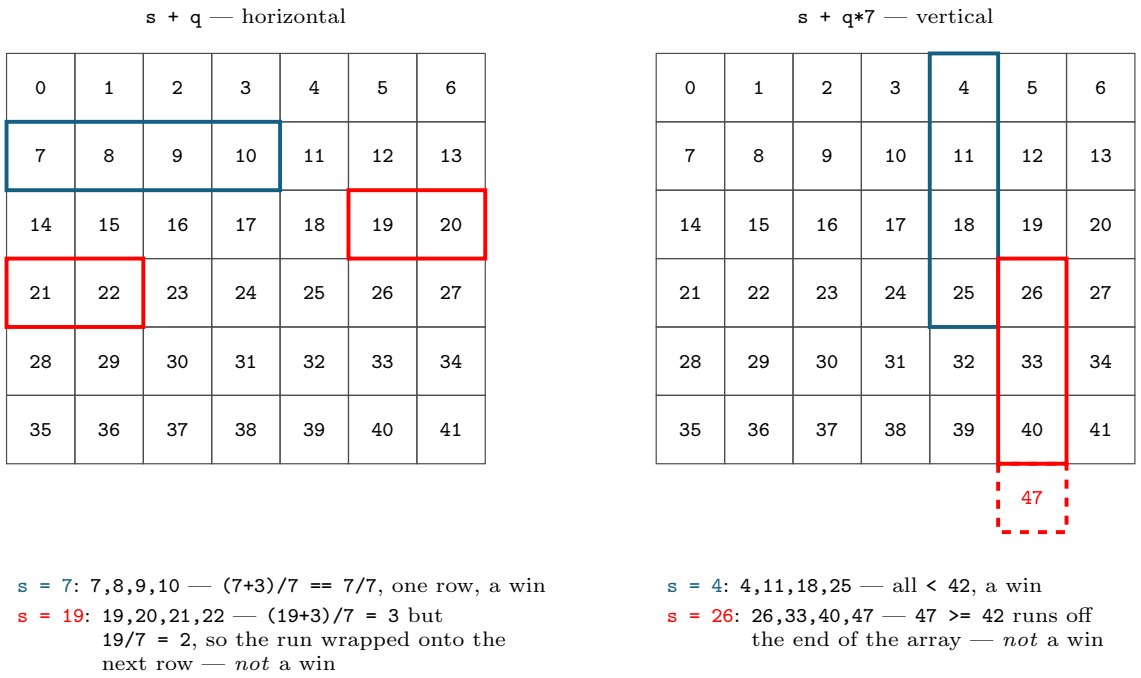


Figure 4: The two bounds checks in the win scan, drawn on the flat board. With 7 slots per row, a horizontal run is `s, s+1, s+2, s+3` and stays on one row only while `(s+q)/7 == s/7` — the left grid shows a run that satisfies it and one that wraps onto the next row. A vertical run is `s, s+7, s+14, s+21`, which needs no row test but must stay inside the array, hence `s + q*7 >= 42` — the right grid shows the run that falls off the end. The four example values of `s` exercise each branch.

Refactoring in the large

Goal: given a reasonably well-*styled* class that has **low cohesion** and too many responsibilities, split it into smaller, more cohesive classes with clear responsibilities — to make the code reusable and extensible, not just readable. Starting point: a library management system implemented entirely in the `Library` class.

Considerations when doing this:

- Each class serves one single-minded purpose (high cohesion).
- Each class has only one *reasonable* reason to change (SRP, see java-solid-principles[24]) — e.g. a change of classification system, a change of bookshelf rendering, or a change of customer management process should each land on one class.
- The interface of each class is designed to have reasonable programmatic interaction.
- Each component depends on abstractions such that each sub-component can be substituted or extended later (DIP, see java-solid-principles[25]).

For this exercise, don't worry about "over-engineering" — make the refactored system as flexible as possible.

> **Working**
>
> A possible split of `Library` (these are the basic classes — a real system could expand on this considerably):
>
> - **`Book`** — keeps track of data related to a single book (eg. title, author, id).
> - **`BookShelf`** — keeps track of all the `Book`s in the library and their availability status.
> - **`Borrower`** — keeps track of data related to a borrower (eg. name, id) and the books they're currently borrowing.
> - **`BorrowingSystem`** — handles borrowing and returning; the "bridge" that passes data between `BookShelf` and `Borrower`.

[24] courses/csse2002/java-solid-principles.pdf
[25] courses/csse2002/java-solid-principles.pdf

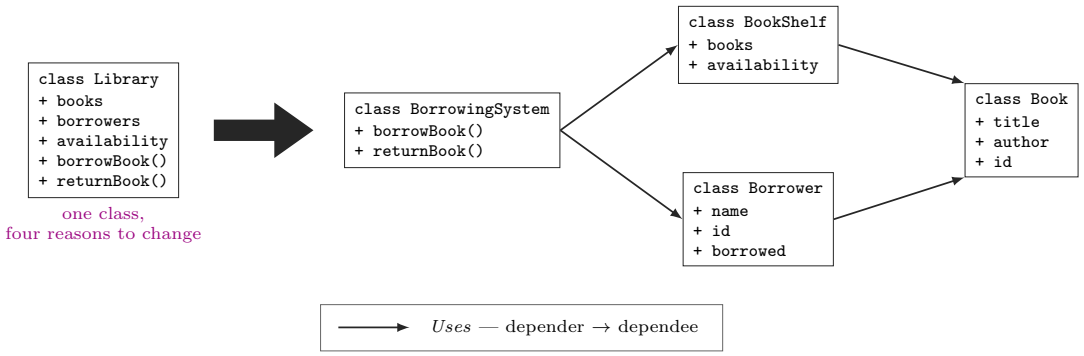


Figure 5: The `Library` monolith and the four-class split, with the dependencies that make `BorrowingSystem` a bridge: it is the only class that knows about both `BookShelf` and `Borrower`, so neither of those has to know about the other. A solid line with a solid arrowhead means *Uses* and points from depender to dependee; the block arrow means 'becomes'. All four classes are concrete and no interface is drawn, although the DIP consideration asks for dependencies on *abstractions*.

Substitution Principle and Pre/Post Conditions

Applied class for 2026-04-30-generics-in-java[26] (Week 9). More practice applying the Liskov Substitution Principle (see java-solid-principles[27]) and pre/postconditions (see java-specification[28]).

Everything here turns on one idea: **a condition is *stronger* when it is more restrictive** — when satisfying it implies satisfying the other, but not the reverse. Every task below is an application of that, first to bare predicates, then to a subclass's contract.

Predicate strength

	A	B
1	`a < 0 && b < 0`	`a != 0 && b < 0`
2	`a instanceof Animal`	`a instanceof String`
3	`a instanceof Animal || b instanceof Zebra`	`a instanceof Animal && b instanceof Zebra`

[26] courses/csse2002/2026-04-30-generics-in-java.pdf
[27] courses/csse2002/java-solid-principles.pdf
[28] courses/csse2002/java-specification.pdf

	A	B
4	`a instanceof Animal && b instanceof Zebra`	`a instanceof Animal`
5	`a instanceof Animal && b instanceof Object`	`a instanceof Animal`
6	`a instanceof Zebra`	`a instanceof Tiger`

For each row, identify which column is a **stronger** (more restrictive) condition, or state that there's no relation.

Working

1. **A is stronger than B** — `a < 0 && b < 0` implies `a != 0 && b < 0` (every `a < 0` is also `a != 0`), but not vice versa.
2. **No relation** — `Animal` and `String` are unrelated types; neither instance check implies the other.
3. **B is stronger than A** — A only needs *either* condition to hold (a looser OR), while B requires *both* (a stricter AND) — satisfying B always satisfies A, not the reverse.
4. **A is stronger than B** — A requires everything B requires, plus more (`b instanceof Zebra` on top of `a instanceof Animal`).
5. **A looks stronger than B, but they're equivalent** — since *everything* is an instance of `Object`, `b instanceof Object` is always true, so it adds no actual restriction.
6. **No relation** — `Zebra` and `Tiger` are siblings (both presumably subtypes of `Animal`, say), so neither instance check implies the other.

Now consider two possible class structures:

```
// Option 1: Preconditions
class ClassA {
    /** @requires A */
    void f(Object a, Object b) {}
}
class ClassB extends ClassA {
    /** @requires B */
    void f(Object a, Object b) {}
}
```

```
// Option 2: Postconditions
class ClassA {
    /** @ensures A */
```

```
    void f(Object a, Object b) {}
}
class ClassB extends ClassA {
    /** @ensures B */
    void f(Object a, Object b) {}
}
```

For each row above, if column A and column B are inserted as the specification text, which option (if any) satisfies the substitution principle?

> **Working**
>
> Recall: a subclass must not *strengthen* preconditions (they may only stay the same or weaken) and must not *weaken* postconditions (they may only stay the same or strengthen).
>
> 1. **Option 1 (precondition)** — `ClassB`'s precondition (B) is weaker than `ClassA`'s (A), which is allowed for preconditions.
> 2. **Neither** — unrelated conditions satisfy neither the "no stronger precondition" nor "no weaker postcondition" rule.
> 3. **Option 2 (postcondition)** — `ClassB`'s postcondition (B) is stronger than `ClassA`'s (A), which is allowed for postconditions.
> 4. **Option 1 (precondition)** — same reasoning as row 1: B is weaker than A.
> 5. **Both** — since A and B are equivalent here, substituting either as precondition or postcondition preserves the (unchanged) strength relationship.
> 6. **Neither** — unrelated conditions again satisfy neither rule.

Rows 1, 3 and 4 each land in exactly *one* column: a strength relation that is legal for a precondition is illegal for a postcondition and vice versa, which is why "B is weaker" is never on its own an answer — you have to know which end of the contract it sits at.

Substitution principle

```
class X {
    /**
     * @require fontSize >= 5
     * @ensure \result >= 0
     */
    int detexify(Object symbol, float fontSize) {
        // Things happen here
    }
}
```

```
class Y extends X {
    /**
     * @require fontSize > 0
     * @ensure \result > 0
     */
    int detexify(Object symbol, float fontSize) {
        // Other things happen here
    }
}

class Z extends X {
    /**
     * @require fontSize > 5
     * @ensure \result > 0
     */
    int detexify(Object symbol, float fontSize) {
        // Do other things happen here?
    }
}
```

The tags are spelled `@require`/`@ensure` here but `@requires`/`@ensures` in the sections either side. Both spellings appear across the course — see csse2002-diagram-notation[29], Inconsistencies 5 — so neither is safe to treat as *the* house form.

Why would a programmer choose to use a precondition at all?

> **i** Working
>
> As a restriction on the allowable range of inputs, to avoid having to handle huge positive/negative numbers (or other awkward edge cases) inside the method body.

Does `Y` violate the substitution principle?

> **i** Working
>
> No. `Y`'s precondition (`fontSize > 0`) is *weaker* than `X`'s (`fontSize >= 5` implies `fontSize > 0`, but not vice versa — the range of acceptable inputs expanded), and `Y`'s postcondition (`\result > 0`) is *stronger* than `X`'s (`\result >= 0` — the range of possible outputs contracted). Both changes are allowed by the principle.

Does `Z` violate the substitution principle?

[29] courses/csse2002/csse2002-diagram-notation.pdf
[30] courses/csse2002/2026-04-16-solid-principles-ii.pdf

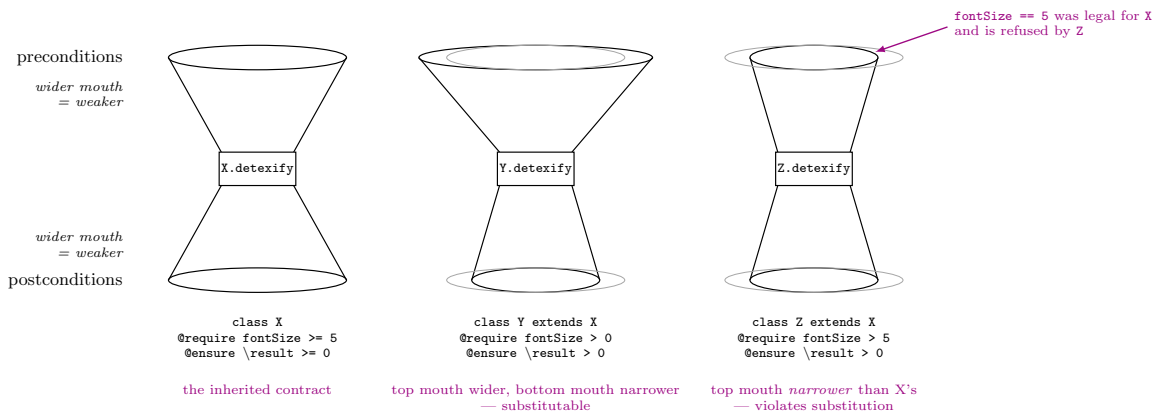


Figure 6: `X`, `Y` and `Z` drawn as the pre/postcondition funnels from 2026-04-16-solid-principles-ii[30]. The mouth above each box is the set of inputs the method accepts, the mouth below is the set of outcomes it may produce, and a *wider* mouth is a *weaker* condition — so a legal override needs a top mouth at least as wide as its parent's and a bottom mouth no wider. `X`'s two mouths are ghosted over `Y` and `Z` for comparison. Read off: `Y` widens the top and narrows the bottom, both legal; `Z` narrows the top, which is what breaks it. The verdicts are in magenta, with a pointer to the input `Z` refuses.

> **Working**
>
> Yes. `Z`'s postcondition is correctly *more restrictive* of possible outputs (`\result > 0` vs. `\result >= 0`). However, the precondition is *also* more restrictive (`fontSize > 5` vs. `fontSize >= 5`) — it forbids `5` as a possible input, which `X` explicitly allowed. Strengthening a precondition violates the principle, regardless of what happens to the postcondition.

Writing pre/postconditions

```
public boolean q2(String[] strArray, int firstIndex, int secondIndex) {
    return strArray[firstIndex] == strArray[secondIndex];
}
```

Write a Javadoc comment for `q2`.

Working

```
/**
 * Determines if two indicated elements of an array of Strings
 * refer to the same String object.
 *
 * @param strArray an array of Strings
 * @param firstIndex index of the first element to compare
 * @param secondIndex index of the second element to compare
 * @return true if the indicated elements in the array refer to
 *         the same string, false otherwise
 */
public boolean q2(String[] strArray, int firstIndex, int secondIndex) {
    return strArray[firstIndex] == strArray[secondIndex];
}
```

The description says *refer to the same* ***`String`*** *object* rather than "are equal": the method uses `==` on references, not `.equals`.

Add `@requires`/`@ensures` tags for all appropriate pre- and postconditions.

Working

```
/**
 * ...
 * @requires strArray != null && 0 <= firstIndex < strArray.length
 *               && 0 <= secondIndex < strArray.length
 * @ensures \result == true
 *              <==> strArray[firstIndex] == strArray[secondIndex]
 */
```

The precondition is everything the body needs in order not to throw: a non-null array and two in-range indices. The postcondition is written as an `<==>` (if and only if), not just "returns true when …", so it also pins down when the result is `false`.

Specify that `q2` does not change any elements of `strArray`.

Working

```
/**
 * ...
 * @ensures \forall int i; 0 <= i && i < strArray.length
 *              ==> \old(strArray)[i] == strArray[i]
 */
```

> `\old(...)` is the value on entry to the method, so this says every slot holds the same reference afterwards as it did before.

Reference material

CSSE2002 Diagram & Notation Conventions

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

How this note was built

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

Pages actually read

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

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

I did **not** read: L01a 1–17 & 28–62, L01b 1–30 & 43–77, L02 21–30 & 46–54, L03 1–13 & 32–62, L04 1–5 & 37–49, L05 15–42, L06 1–14 & 33–52, L07 1–13 & 31–39, L08 1–2 & 17–50,

L09 1–18 & 34–46, L11 1–19 & 33–34, L13 1–25 & 41–89. A few citations below (textbook mentions, "Further Reading" slides) come from **extracted text on pages I did not view as images**; those are marked *(text only)*. Font names come from `pdffonts`; hex colours were sampled from rendered pixels, not from any stated palette.

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

---

1. The page frame every figure sits in

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

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

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

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

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

2. Typeface roles

Role	Face	Evidence
Slide titles, section headings	Tw Cen MT, purple	everywhere in Template A
Prose, captions, labels on the lecturer's own diagrams	Bell MT (serif)	L03 p27; L06 p21; L07 p16

Role	Face	Evidence
Java source; Java identifiers inline in prose; text inside class/dependency boxes	**Inconsolata** (hand-set) or **Courier New** (pasted)	L02 p32, p36; L07 p27; L08 p6
Borrowed / stock figures	Arial or similar sans	L02 p15; L06 p17; L08 p3; L09 p27

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

3. What colour means

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

Colour	Sampled hex	Meaning	Evidence	Consistency
Near-black	`#171717`	prose, box outlines, arrows in *borrowed* figures	L05 p6, p8; L07 p26	throughout

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

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

4. Class-hierarchy diagrams — the dominant style

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

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

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

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

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

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

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

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

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

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

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

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

7. The pre/postcondition "funnel" figure

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

- A method is a small **square box, thin black outline, monospace name** (`A.f`, `B.f`).

- **Above** it, a cone that narrows downward into the box. Its wide upper ellipse is the set of inputs the method accepts — the **precondition**.
- **Below** it, a cone that widens downward out of the box. Its wide lower ellipse is the set of outcomes it may produce — the **postcondition**.
- **Width = weakness.** A *wider* top mouth = a *weaker* (more permissive) precondition. A *wider* bottom mouth = a *weaker* (less committal) postcondition. So a legal subclass override has a top mouth **at least as wide** and a bottom mouth **no wider** than its parent's.
- Labels `preconditions` and `postconditions` in **lowercase serif** at the far left of the slide, vertically aligned with the two mouths.
- Entirely **black line art, unfilled, no colour**.
- To compare parent and child, the child's cones are **overlaid on the parent's** in the same frame in thinner/greyer stroke (L07 p17).

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

8. Coupling and cohesion figures

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

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

9. The coupling-levels ladder

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

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

10. Annotating code

- **Highlight a token:** a thin **red `#FF0000` rectangle** drawn around it. *Cited:* L02 p38, p40; L03 p28; L04 p14; L08 p16. Dominant across four decks.
- **Point at a token:** a thin **teal straight or diagonal line with a small solid arrowhead**, running between the token and a **serif caption placed outside the code block, unboxed**. *Cited:* L02 p16; L03 p24, p28; L08 p14; L09 p25, p29; L13 p27. Weeks 2–13 — very consistent.
- **Show a replacement: strike through** the token being removed and put the replacement beside it in red. *Cited:* L02 p4; L07 p20; L13 p27.
- **Mark an error inline:** a red squiggly/dashed underline, IntelliJ-style. *Cited:* L09 p22.
- **Aside / rule-to-remember:** a short sentence centred beneath the content in serif — sometimes inside a thin rectangle, sometimes bare. *Cited (boxed black):* L02 p31; L07 p24; L08 p5; L09 p26. *Cited (boxed teal):* L03 p21. *Cited (unboxed):* L03 p23; L06 p31; L08 p12, p15; L07 p14, p25. Inconsistent — see below.
- **Yellow highlighter** behind a heading inside a worked-example box. *Cited:* L07 p19 (single instance).
- **Verdict on an example**, placed in the right margin in magenta serif: "Bad Example", "Better / Still Bad ", "Does not satisfy specification!", "Satisfies specification!". *Cited:* L04 p26, p27, p28, p29, p30, p31. Consistent within L04.

11. Code blocks

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

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

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

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

12. Specification / Javadoc notation

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

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

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

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

13. Arrays and stacks

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

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

14. Black-box / glass-box testing icon

A **grey-filled isometric cube, unlabelled**, with a **black horizontal arrow entering on the left** labelled `input` in serif above the line, and a matching arrow **leaving on the right** labelled `output`. Flow is strictly left→right. The glass-box variant is the same cube rendered translucent with the source code visible inside it. *Cited:* **L05 p6** (both cubes together), L05 p8 (black-box cube reused as a corner marker).

15. Attribution and textbook lineage

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

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

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

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

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

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

16. Naming and capitalisation

- Java identifiers follow standard Java: `PascalCase` types, `camelCase` methods and fields, `SCREAMING_SNAKE` constants, always monospace. **Consistent everywhere.**
- Running example names: `Person`, `Employee`/`ContractEmployee`/ `FullTimeEmployee`, `Animal`/`Dog`/`Cat`, `BankAccount`/`AccountHolder`, `Foo`, `Printer`, and for abstract structural examples `class A`/`B`/`C`, `class Z`, `interface AI`/`BI`/`CI`, `interface BSpec`.

- **Interfaces in structural diagrams are named either `<Letter>I`** (`AI`, `BI`, `CI` — L07 p27) **or `<Class>Spec`** (`BSpec` — L07 p29, p30). Both in the same deck.
- Method-signature shorthand in prose uses UML order — `getType():String`, `addListener(String type, Consumer<SimpleEvent> action): action` (L11 p28) — while diagram boxes use the same order (L02 p36). Consistent.
- A right arrow `→` in serif means "resolves to" / "dispatches to": `kapow(Object) → locked in`. *Cited:* L13 p26, p28.
- ASCII `->` between monospace type names means "widens to" in the casting chains: `byte -> short -> char -> int -> long -> float -> double`. *Cited:* L03 p26.

Inconsistencies

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

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

- `@returns` instead of `@return` — **L05 p8**. **Decision needed.** The tag table (L04 p23) and the exam-prep deck (L13 p36) agree on the `-s` forms; the SOLID deck (L07) does not.

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

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

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

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

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

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

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

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

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

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

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

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

Not established by the source

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

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

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

Java Casting

Introduced in 2026-03-12-object-oriented-programming-ii[31] (Lecture, Week 3). See also java-primitive-and-reference-types[32] and week4-tutorial-method-dispatch-and-casting[33].

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;`.

[31] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[32] courses/csse2002/java-primitive-and-reference-types.pdf
[33] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf

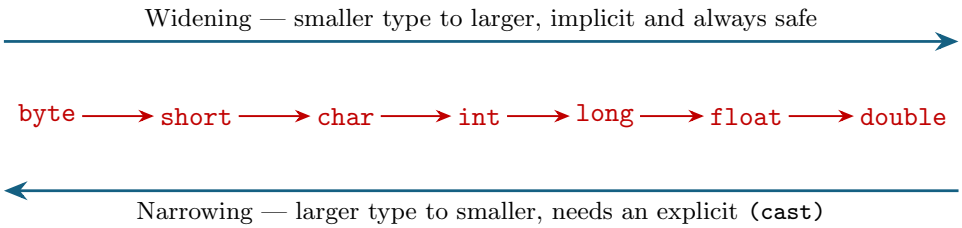


Figure 7: The two primitive conversion chains on a single axis: the narrowing chain is exactly the widening chain reversed. Widening is implicit; narrowing needs an explicit cast.

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.

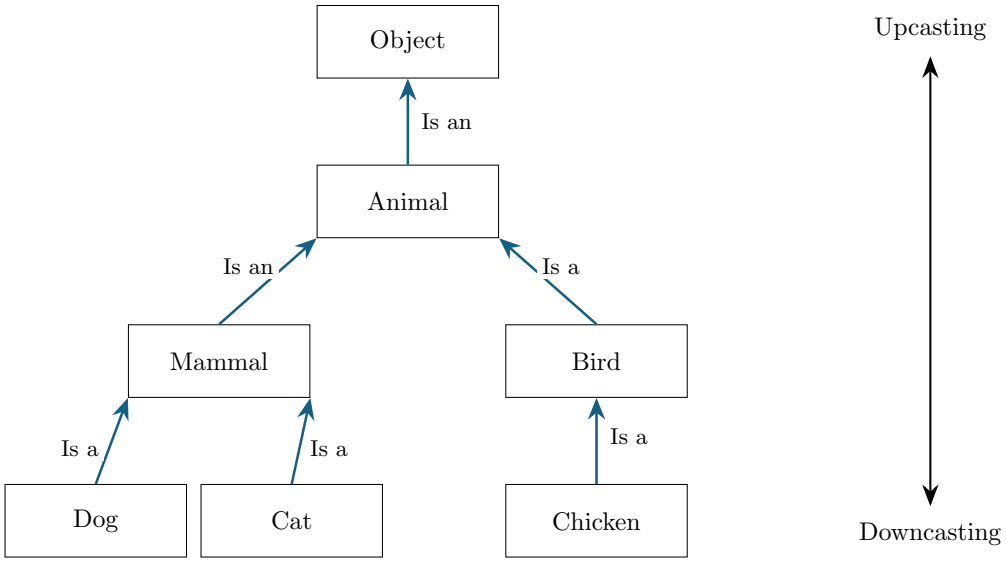


Figure 8: The class hierarchy that upcasting and downcasting move along, with 'Is a' / 'Is an' edge labels and the up/down casting axis at the right. Object sits at the root because every class implicitly extends it.

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[34]'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[35]).

Additional reading: upcasting and downcasting in Java.

[34] courses/csse2002/java-polymorphism.pdf
[35] 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 8. 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[36] exists to remove.

Java Cohesion and Coupling

Introduced in 2026-04-02-refactoring[37] (Lecture, Week 6, Part 2). Practised in week10-tutorial-coupling-and-cohesion[38].

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).

[36] courses/csse2002/java-generics.pdf
[37] courses/csse2002/2026-04-02-refactoring.pdf
[38] courses/csse2002/week10-tutorial-coupling-and-cohesion.pdf

Coupling

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

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

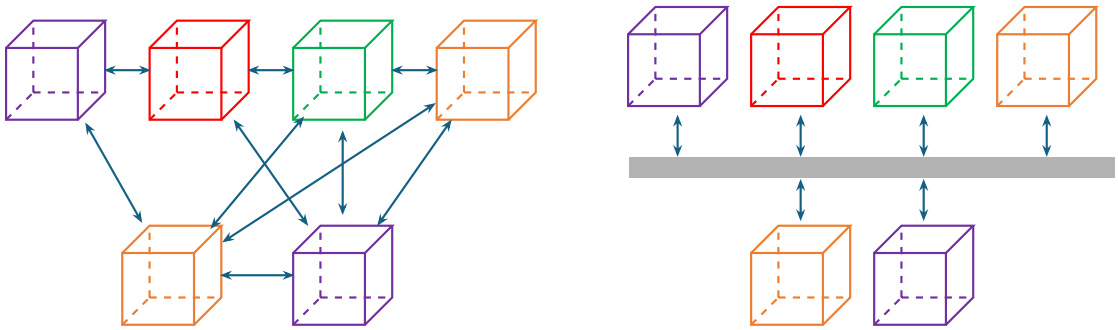


Figure 9: High versus low coupling. Left: every module talks directly to several others, so a change in any one can reach any other. Right: the same six modules each talk only to a shared interface — the grey bar — so nothing propagates further than that bar. A module is a wireframe cube, and its colour only tells the modules apart; a dependency is a teal double-headed arrow. The tangle of arrows on the left is schematic — the count is the point, not which cube joins which.

Levels of coupling (tightly → loosely coupled)

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

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

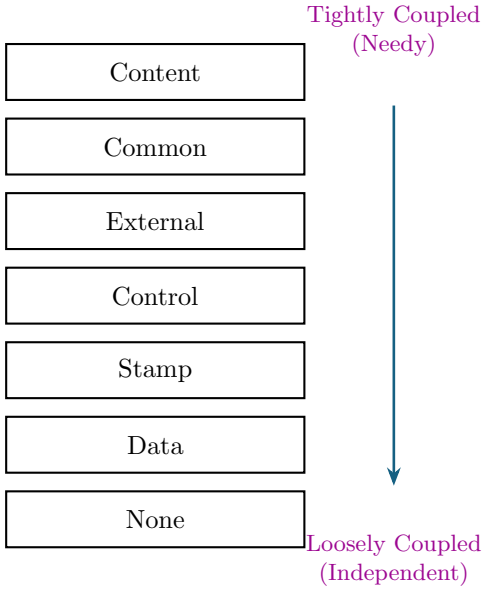


Figure 10: The coupling-levels ladder. Most tightly coupled at the top, least at the bottom, with the arrow running down from tightly coupled (needy) to loosely coupled (independent).

Worked classification examples

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

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

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

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

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

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

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

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

Five separate simple parameters — still **data coupling**, though a long parameter list like this is itself often a separate code smell (see java-refactoring[39]) 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).

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

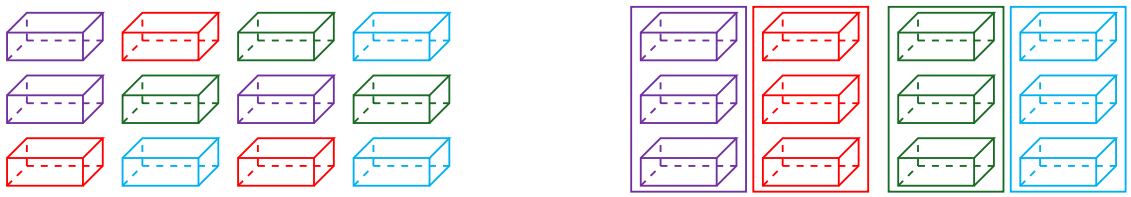


Figure 11: Low versus high cohesion. The same twelve elements in both panels: on the left the four kinds are scattered across the grid, on the right each kind is gathered into one module. Colour means kind only, and each high-cohesion group is enclosed by a rectangle in that group's own colour. This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

Worked example

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

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

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

Is this class cohesive?

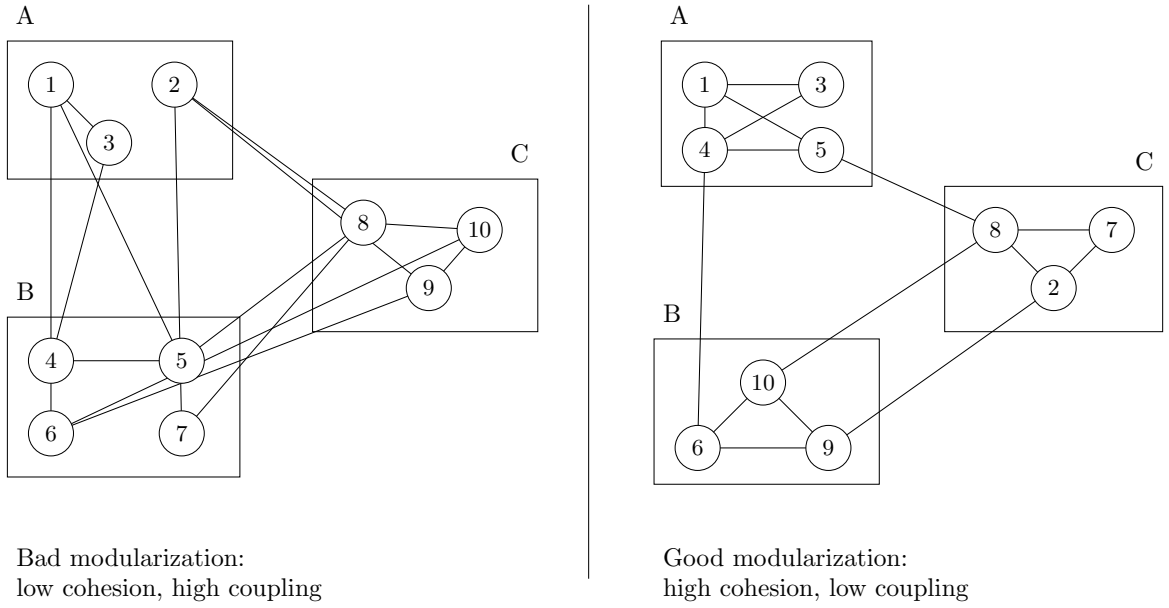
A `Car` class contains: Fuel, Steering, Speed, Route planner, Public holiday calculator.

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

Considerations

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

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



Bad modularization:
low cohesion, high coupling

Good modularization:
high cohesion, low coupling

Figure 12: Bad versus good modularization. Both panels draw the same ten elements and the same fifteen dependencies; only the assignment of elements to modules A, B and C differs. Left: 5 dependencies fall inside a module and 10 cross a boundary. Right: 11 fall inside and only 4 cross. Nothing was rewritten to achieve that — the code is identical, the grouping is not.

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

Java Collections Framework

Introduced in 2026-02-26-java-basics-part-02-collections-and-strings[40] (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[41] 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`.

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

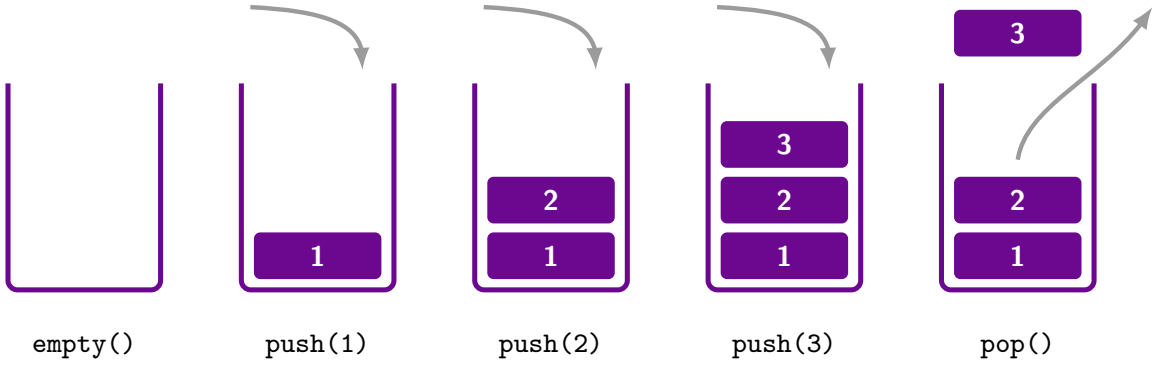


Figure 13: Last In, First Out: `push` always adds at the top and `pop` always removes from the top, so the last value in is the first value out. The stack is an open-topped container whose entries grow upward. An arrow *into* the container is a push and an arrow *out and away* is a pop. Each frame is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

List

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

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

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

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

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

Set

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

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


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

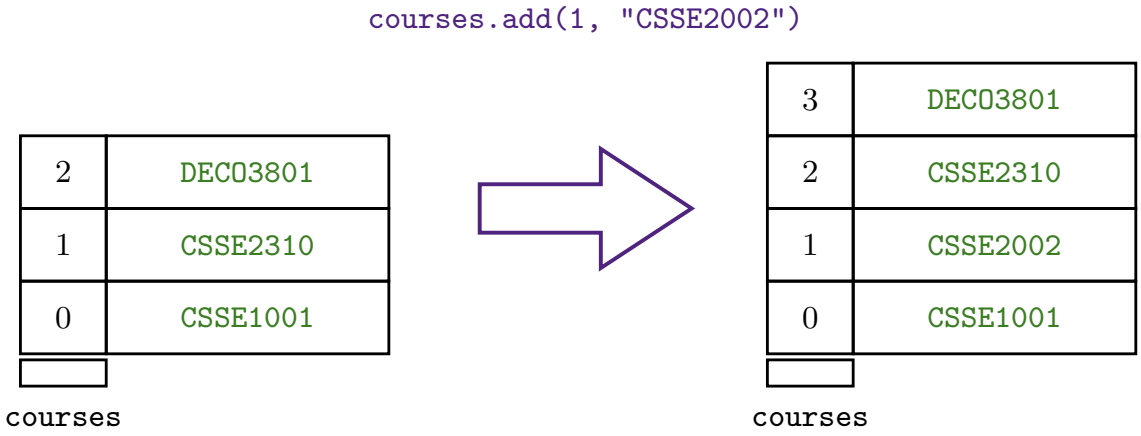


Figure 14: `add(index, e)` on a `List`: `e` lands *at* that index and every entry from there up renumbers, so the list gets longer without you writing a single shift. (`remove(index)` is the mirror image — see the removal step traced in 2026-02-26-java-basics-part-02-collections-and-strings[42].) The list has **index 0 at the bottom** and grows upward, with index cells on the left, values in green on the right, and the variable name in a small box beneath the stack of cells.

Map

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

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

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

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

The hashCode/equals contract

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

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

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

Java Exceptions

Introduced in 2026-03-19-exceptions[43] (Lecture, Week 4, Part 1). Traced in depth in week5-tutorial-exceptions[44].

What is an exception?

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

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

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

Common causes:

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

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

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

[43] courses/csse2002/2026-03-19-exceptions.pdf
[44] courses/csse2002/week5-tutorial-exceptions.pdf

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

Exception hierarchy

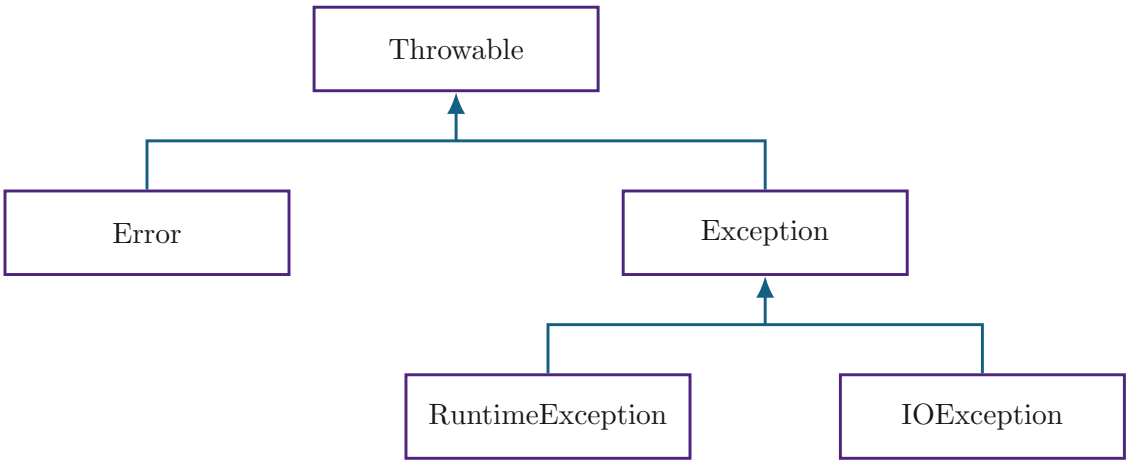


Figure 15: The exception hierarchy. Throwable is the root; Error and Exception sit beneath it; RuntimeException (unchecked) and IOException (checked) sit beneath Exception. Each connector joins the subclasses and points up to their parent. Which branch a class is on is what decides whether the compiler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

`RuntimeException` — unchecked

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

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

`IOException` — checked

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

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

Java exception handling

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

`try … catch`

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

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

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

`throw`

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

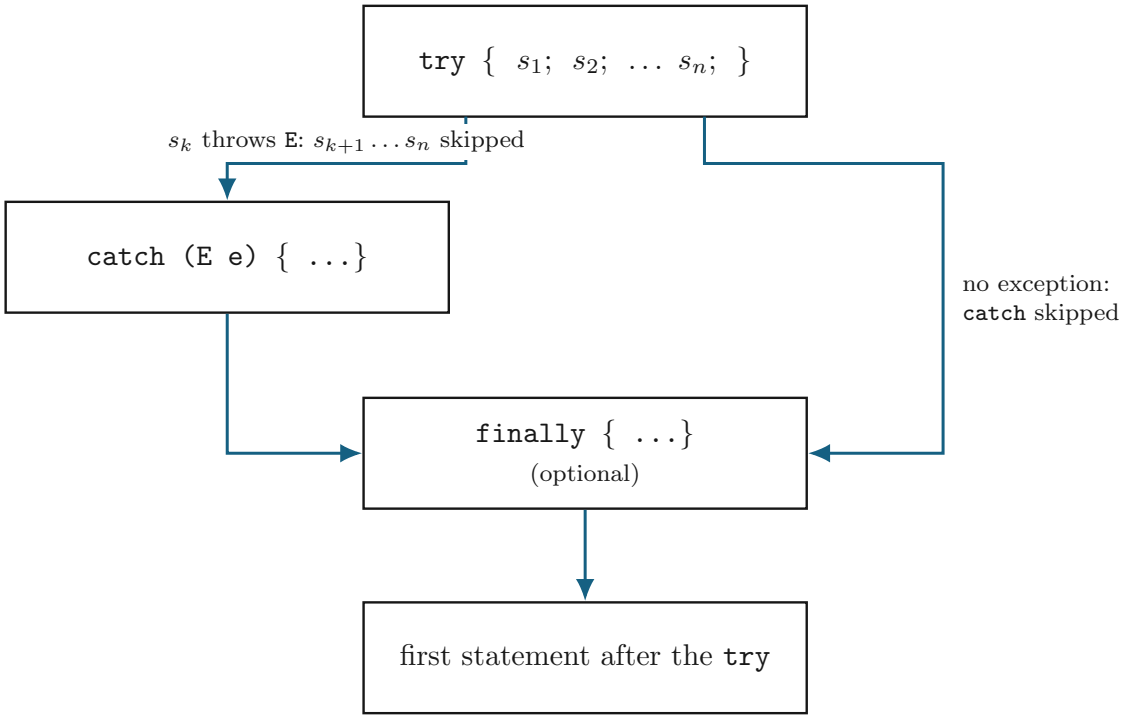


Figure 16: The two paths through a try/catch/finally, and the fact that both of them end up in finally. This is the three numbered rules together with the finally rule, drawn as one picture: the exceptional path abandons the remaining statements of the try and enters catch, the normal path skips catch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

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

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

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

`throw` is used to manually trigger an error when your program detects something invalid, so it doesn't continue in a bad state. That is the point of the whole keyword, and it is also the bridge to java-specification[45]: throwing is how a method refuses input that breaks its precondition instead of limping on.

`throws`

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

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

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

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

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

`finally`

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

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

Exercise — what will the following print?

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

Answer:

```
Good Morning
hello
world
```

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

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

Custom exceptions

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

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

}

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

Worked example — `ItemNotFoundException`

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

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

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

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

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

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

Java Generics

Introduced in 2026-04-30-generics-in-java[46] (Lecture, Week 9). Practised again in week12-lab-generics[47].

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.

[46] courses/csse2002/2026-04-30-generics-in-java.pdf
[47] 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[48]):

```
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.

[48] 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[49]), `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**.

[49] 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

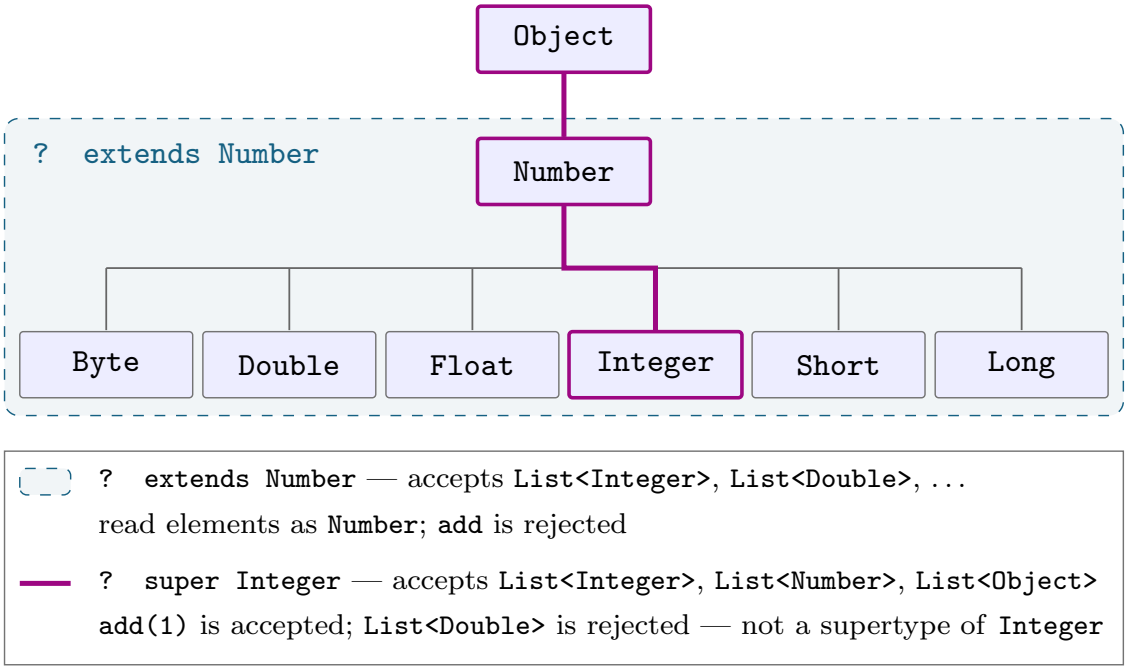


Figure 17: The two bounded wildcards drawn on the `Number` hierarchy. `? extends Number` admits `Number` and everything below it — you may read elements as `Number`, but `add` is refused; `? super Integer` admits `Integer` and everything above it — you may `add` an `Integer`, and `List<Double>` is refused. `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**.

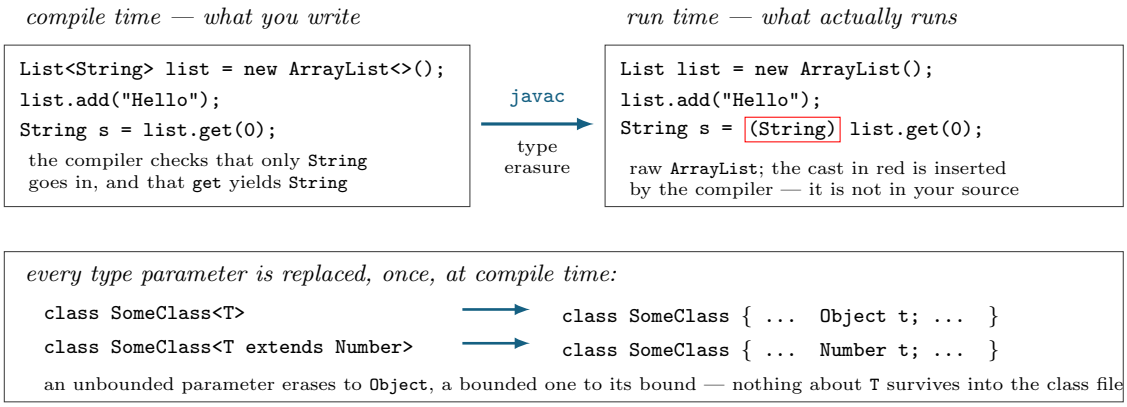


Figure 18: Type erasure as a pipeline: the compiler checks the source against the type parameters, erases each parameter to its bound (`Object` if unbounded), and inserts the casts the erased code needs — so the cast you were told you did not have to write still happens, just not by you. The top row 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

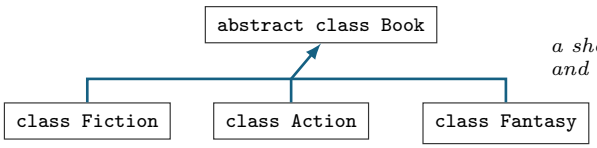
> Imagine you are developing a system for a bookshop that sells books across various genres: `Fiction`, `Action`, and `Fantasy`. Books are in bookshelves. Your job is to design the system in a way that allows the shop to store books of these different genres and retrieve them when needed.

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 19: 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 Inheritance

Introduced in 2026-03-05-object-oriented-programming-i[50] (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:

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

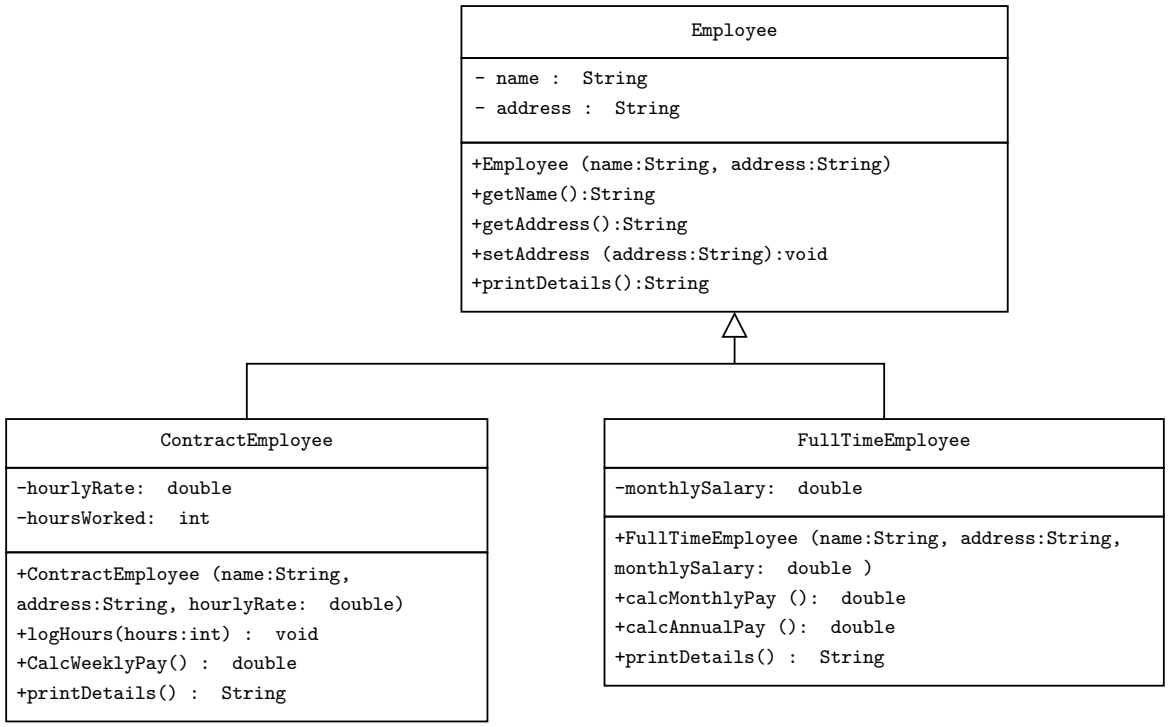


Figure 20: The `Employee` inheritance example as a UML class diagram. Each box has three compartments (name / attributes / operations), with - for private and + for public, and constructors listed in the operations compartment with no return type. Generalisation is a solid line with a single unfilled 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[51] 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;
```

[51] 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[52]; 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;
    }
}
```

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

```java
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:

```java
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[53] and week4-tutorial-method-dispatch-and-casting[54].

Java Refactoring

Introduced in 2026-04-02-refactoring[55] (Lecture, Week 6). Practised in week9-lab-refactoring[56].

[53] courses/csse2002/java-polymorphism.pdf
[54] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf
[55] courses/csse2002/2026-04-02-refactoring.pdf
[56] courses/csse2002/week9-lab-refactoring.pdf

What is refactoring?

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

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

Why refactor?

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

Good practice

- **Don't** refactor and add functionality at the same time — keep the two activities separate.
- Have good tests *before* refactoring, so you'll know immediately if you've broken something (see java-testing[57], java-junit[58]). This is what puts `Refactor` where it is in the development cycle, drawn in 2026-04-02-refactoring[59]: you only ever enter it from a passing test.
- Take short, deliberate steps: move a member variable, split a method, rename a variable. Refactoring is usually many small, localised changes that add up to a larger-scale change — small steps + testing after each one avoids prolonged debugging.
- Refactor early, refactor often.

When to refactor

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

[57] courses/csse2002/java-testing.pdf
[58] courses/csse2002/java-junit.pdf
[59] courses/csse2002/2026-04-02-refactoring.pdf

Example: making it easier to add a feature

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

```
public class Vehicle {
    private String type;

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

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

Refactored to use inheritance/polymorphism (see java-polymorphism[60]), adding a `Scooter` is just one new class — no existing code needs editing:

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

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

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

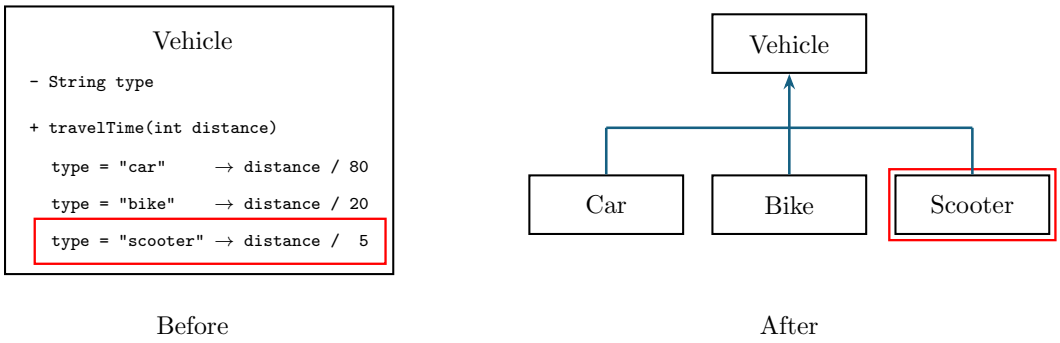


Figure 21: Adding a scooter, before and after. Left: one class, and the new type is another branch spliced into an existing method — the red outline marks the edit. Right: an abstract Vehicle with one subclass per type, joined by a connector with a single arrowhead into the parent. The new type is a whole new box that no existing box points at, again outlined in red.

What has *not* changed: `travelTime(int distance)` still exists and still returns the same answers for cars and bikes. That is what makes this a refactoring rather than a rewrite. The `Vehicle` abstraction being open to new subclasses without being edited is the Open-Closed Principle arriving early — see java-solid-principles[61].

Example: making it easier to understand

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

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

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

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

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

Code smells

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

Bad naming

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

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

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

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

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

Duplication

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

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

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

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

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

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

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

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

Feature envy

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

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

class User {
    private ShoppingCart cart;

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

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

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

class User {
    private ShoppingCart cart;
```

```
}
```

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

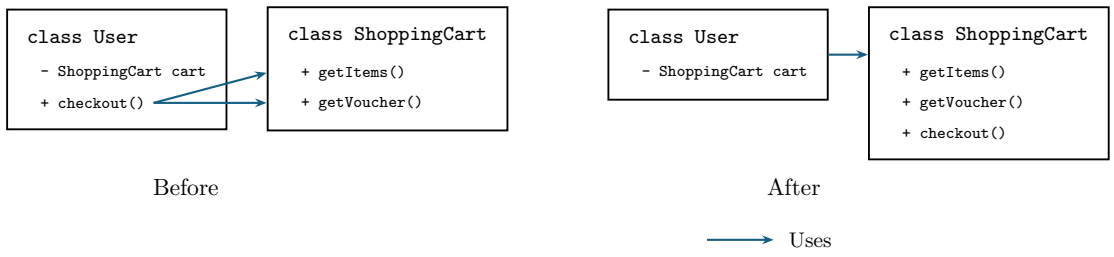


Figure 22: Feature envy, before and after. Before: User.checkout() reaches across the class boundary into two of ShoppingCart's members. After: checkout() sits with the data it uses, and User's only link to ShoppingCart is the field it holds. Each box is a class with its members listed as '+ m()'. A solid arrow means 'uses' and points from the depender to the thing depended upon; it may land on an individual member row.

Two things improve at once: `ShoppingCart` gets more cohesive (the checkout logic now sits with the data it operates on) and `User` gets less coupled to it (one link instead of three). Both are the subject of java-cohesion-and-coupling[62].

Others

Many other code smells exist. Further reading:

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

Java SOLID Principles

Introduced in 2026-04-02-refactoring[63] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[64] (Lecture, Week 7 — Liskov Substitution, Interface Segregation, Dependency Inversion).

[62] courses/csse2002/java-cohesion-and-coupling.pdf
[63] courses/csse2002/2026-04-02-refactoring.pdf
[64] courses/csse2002/2026-04-16-solid-principles-ii.pdf

Why SOLID?

Large software tends to become:

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

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

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

Single Responsibility Principle (SRP)

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

Closely related to java-cohesion-and-coupling[65]'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.

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

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

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

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

The same shape, stated more briefly:

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

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

Homework. Check whether SRP improves memory management *indirectly*, by improving overall design structure — think about object creation, immutability (see java-mutability[66]) and garbage collection.

Open-Closed Principle (OCP)

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

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

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

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

A class is kept closed for modification by information hiding (see java-encapsulation[67]).

Example: pluggable grading systems

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

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

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

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

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

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

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

Three things carry the problem: the added `String gradeType` parameter, and the two string literals `"Letter"` and `"Numeric"`. They are every place that has to be *modified* the next time a grading system is added, in a class that was supposed to be finished. The class is open for extension only by editing it, which is exactly what "closed for modification" forbids. (A `gradeType` string also reintroduces the `Vehicle`-style type switch from java-refactoring[68].)

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

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

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

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

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

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

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

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

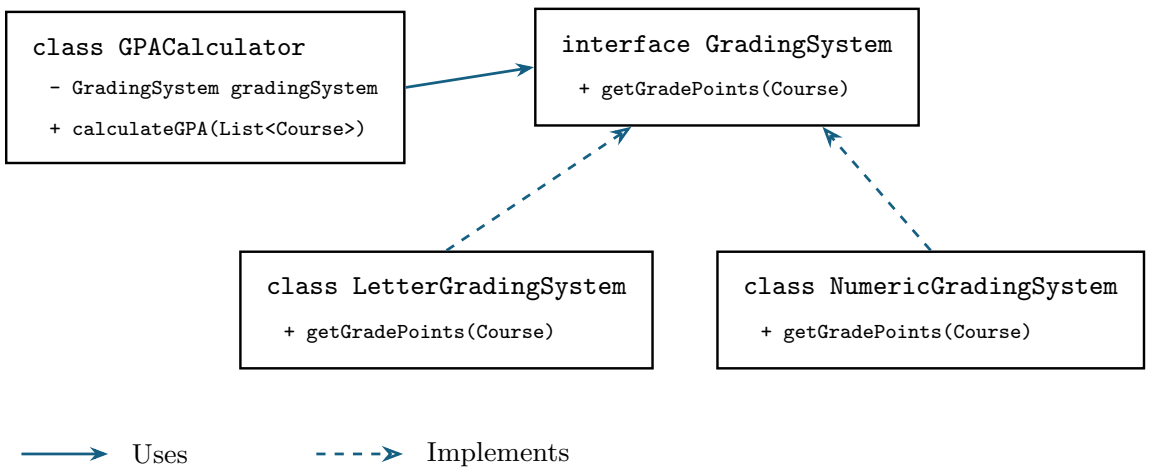


Figure 23: The open-closed refactoring of GPACalculator. GPACalculator uses only the GradingSystem interface; each concrete grading system implements it. A new scheme — pass/fail, say — is a fourth box on the bottom row with a fourth dashed arrow, and no existing box changes: that is 'open for extension, closed for modification' as a picture. Each label carries the Java keyword and members are listed as '+ m()'. A solid line with a solid head means 'uses', a dashed line with an open head means 'implements', and every arrow points from the depender to the thing depended upon. This is not UML, which would draw realisation as a hollow triangle.

What the refactoring bought:

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

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

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

Liskov Substitution Principle (LSP)

> Subclasses should be substitutable for their parent classes.

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

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

Example violation

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

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

Substitution with contracts

Even when a subclass overrides a method with a different implementation, it must still honour the parent's contract (see java-specification[69]):

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

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

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

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

Worked example

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

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

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

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

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

Interface Segregation Principle (ISP)

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

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

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

This mirrors java-cohesion-and-coupling[70]'s stamp coupling at the interface level: despite each client using only a small subset of `Z`, every client depends on *all* of `Z` — so a change to any part of `Z` risks affecting (and forcing a recompile/redeploy of) every client, even ones that never used the changed part. Splitting `Z`'s surface into focused interfaces removes those incidental dependencies:

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

Dependency Inversion Principle (DIP)

> Depend upon abstractions. Do not depend upon concretions.

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

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

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

What makes a good dependency? (stability and exposure)

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

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

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

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

Two forms of DIP

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

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

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

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

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

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

Dependency Injection

Dependency Injection (DI) is a design pattern that implements DIP by injecting a class's dependencies into it (e.g. via the constructor) instead of having the class create them internally, promoting loose coupling (see java-cohesion-and-coupling[71]):

[71] 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[72] (Lecture, Week 3). Extended in 2026-03-19-exceptions[73] and formalised later in java-hoare-logic[74]; pre/postconditions come back as class invariants in week8-tutorial-class-invariants[75] and as the substitution principle in week9-tutorial-substitution-principle-and-pre-post-conditions[76].

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
```

[72] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[73] courses/csse2002/2026-03-19-exceptions.pdf
[74] courses/csse2002/java-hoare-logic.pdf
[75] courses/csse2002/week8-tutorial-class-invariants.pdf
[76] 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**.

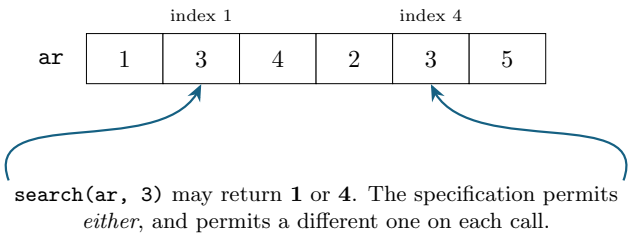


Figure 24: With the spec worded as 'returns an index i such that ar[i] == x', both cells holding 3 are legal answers, and nothing forbids the method returning a different one on each call. That is what makes the specification non-deterministic.

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[77]. 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:

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

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

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

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

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

Contracts

Specifications can be written using contracts:

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

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

Programming *with* contracts

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

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

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

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

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

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

Programming *without* contracts

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

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

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

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

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

Defensive programming

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

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

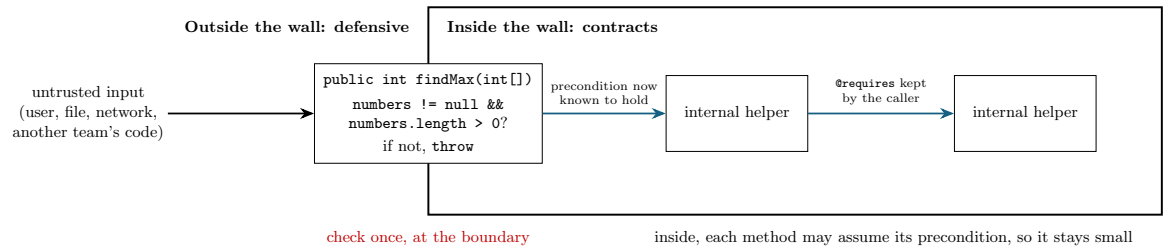


Figure 25: Defensive outside the wall, contracts inside it: validate defensively where untrusted input crosses into the system, and rely on contracts for the internal calls behind that boundary, so the check happens once rather than at every hop. The gate box is the findMax method worked through below.

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.

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[78]) — 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 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`

[78] courses/csse2002/2026-05-28-course-summary-and-exam-preparation.pdf

`… 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[79]).
- 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%.

[79] 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

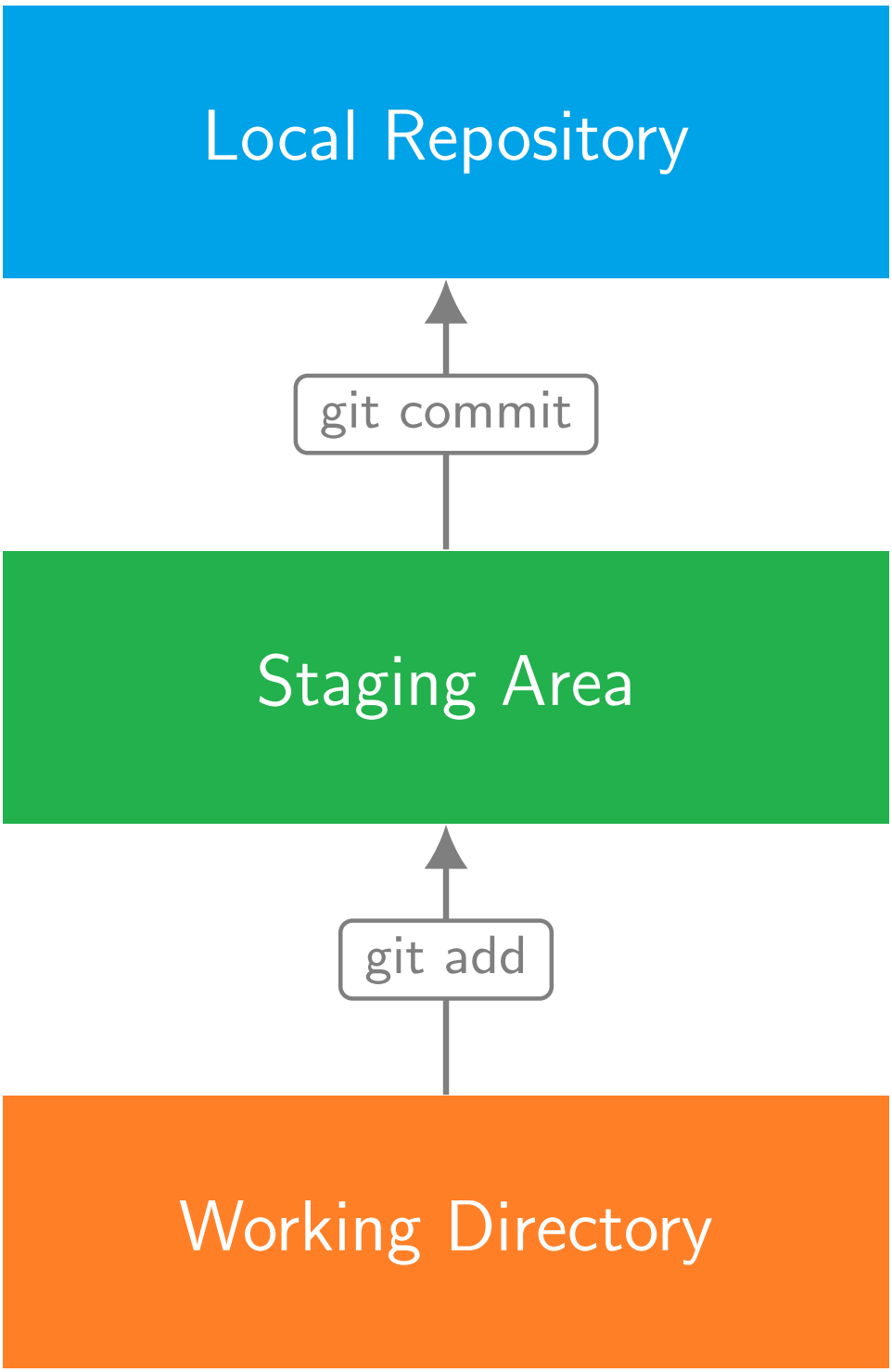


Figure 26: The three places a change can be, and the one command that moves it between each pair. This is the whole reason the staging area is a separate idea: `git add` does not save anything permanently, and `git commit` does not look at your files, it looks at what you staged.

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[80] 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.

[80] courses/csse2002/csse2002.pdf