CSSE2002 — Week 9 Notes

Table of contents

Generics in Java

See csse2002[1] for course logistics.

[1] courses/csse2002/csse2002.pdf

Today's outline

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

All content is in java-generics[2].

Applied class

See week9-tutorial-substitution-principle-and-pre-post-conditions[3] — more practice with the Liskov Substitution Principle (from java-solid-principles[4], Week 7) and pre/postconditions (from java-specification[5], Week 3).

Practical

See week9-lab-refactoring[6] — refactoring in the small and in the large, extending java-refactoring[7] (Week 6).

Refactoring

Practical for 2026-04-30-generics-in-java[8] (Week 9). Extends java-refactoring[9] (Week 6) with a full worked refactoring of a Connect 4 implementation, plus a "refactoring in the large" exercise using java-cohesion-and-coupling[10] and java-solid-principles[11] (SRP, DIP).

[2] courses/csse2002/java-generics.pdf
[3] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf
[4] courses/csse2002/java-solid-principles.pdf
[5] courses/csse2002/java-specification.pdf
[6] courses/csse2002/week9-lab-refactoring.pdf
[7] courses/csse2002/java-refactoring.pdf
[8] courses/csse2002/2026-04-30-generics-in-java.pdf
[9] courses/csse2002/java-refactoring.pdf
[10] courses/csse2002/java-cohesion-and-coupling.pdf
[11] courses/csse2002/java-solid-principles.pdf

Setup

Starting point: a `Connect4` class (one `play` method implementing most of the game), a small `Library` helper, and a test suite. Set up version control *before* refactoring, so each refactoring step can be committed separately and reverted if it breaks something.

Refactoring in the small

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

> **i** Working

Style guide fixes (whitespace) — trivial with a linter like 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 ones:

```
// Before
int[] n = new int[7 * 6];
int[] m = new int[7 * 6];
boolean t = false;

// After
int[] boardX = new int[7 * 6];
int[] boardO = new int[7 * 6];
boolean isTurnX = false;
```

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

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

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

Magic numbers — replace with named constants:

```java
private static final int ROW_SIZE = 7;
private static final int COL_SIZE = 6;
// for (int i = 0; i < ROW_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):

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

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

```java
// 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 — the original win-checking logic repeats a near-identical horizontal/vertical check for *both* players. This is refactored in two stages:
Stage 1 — extract a `checkWin(int[] playerBoard)` helper parameterised on which player's board to check, removing the player-specific duplication:

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

    win = true;
    for (int q = 0; q < 4; q++) {                   // vertical
        if (s + (q * 7) >= 42) { return false; }
        if (playerBoard[s + (q * 7)] == 0) win = false;
    }
    return win;
}
// if (checkWin(boardX)) { out.println("Player X wins"); exit = true; }
// if (checkWin(boardO)) { out.println("Player O wins"); exit = true; }
```

Stage 2 — further split `checkWin` into `checkHorizontalWin`/`checkVerticalWin`, each returning as soon as a check fails (removing the need for the `win` bookkeeping variable entirely):

```
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 options (a matter of preference):

1. A nested array of 6 rows × 7 columns instead of a flat 42-slot array (`int[][] boardX = new int[7][6]`) — but this breaks all existing indexing code, so needs a gradual rewrite.

2. Combine `boardX`/`boardO` into a single board (`1` = X, `2` = O), avoiding the risk of the two arrays getting out of sync.
3. Since `boardX`/`boardO` only ever store `1` or `0`, make them `boolean[]` instead of `int[]`.

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 monolithic `Library` class handling everything.

Considerations when doing this:

- Each class should serve one single-minded purpose (high cohesion).
- Each class should have only one reasonable reason to change (SRP, see java-solid-principles[12]) — e.g. changing the classification system shouldn't require touching bookshelf-rendering code.
- Each class's interface should be designed for reasonable programmatic interaction (not too large — see Interface Segregation in java-solid-principles[13]).
- Components should depend on abstractions so sub-components can be substituted/extended later (DIP, see java-solid-principles[14]).

i Working

A possible split of `Library` (not exhaustive — a real system could expand on this considerably):

- **`Book`** — data for a single book (title, author, id).
- **`BookShelf`** — tracks all `Book`s in the library and their availability status.
- **`Borrower`** — data for a borrower (name, id) and the books they're currently borrowing.
- **`BorrowingSystem`** — the "bridge" that handles borrowing/returning, passing data between `BookShelf` and `Borrower` rather than either of those two classes depending directly on each other.

i Working

[12] courses/csse2002/java-solid-principles.pdf
[13] courses/csse2002/java-solid-principles.pdf
[14] courses/csse2002/java-solid-principles.pdf

Substitution Principle and Pre/Post Conditions

Applied class for 2026-04-30-generics-in-java[15] (Week 9). More practice applying the Liskov Substitution Principle (see java-solid-principles[16]) and pre/postconditions (see java-specification[17]).

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`
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`,

[15] courses/csse2002/2026-04-30-generics-in-java.pdf
[16] courses/csse2002/java-solid-principles.pdf
[17] courses/csse2002/java-specification.pdf

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?

i 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.

Substitution principle

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

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

class Z extends X {
    /**
     * @require fontSize > 5
     * @ensure \result > 0
     */
    int detexify(Object symbol, float fontSize) { ... }
}
```

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 substitution principle.

Does `Z` violate the substitution principle?

Working

Yes. `Z`'s postcondition is correctly stronger (`\result > 0` vs. `\result >= 0`), but its precondition (`fontSize > 5`) is *also* stronger than `X`'s (`fontSize >= 5`) — it forbids `fontSize == 5`, 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
 */
```

Add `@requires`/`@ensures` tags.

Working

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

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]
 */
```

Reference material

Java Cohesion and Coupling

Introduced in 2026-04-02-refactoring[18] (Lecture, Week 6, Part 2).

Modularization

The process of dividing a large system (a monolith) into smaller, more manageable pieces that can be worked on and tested independently before being reassembled into the final product. This raises a question: what if the resulting modules are highly dependent on each other?

Coupling

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

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

Levels of coupling (tightly → loosely coupled)

1. **Content** — one class directly manipulates another's data (e.g. public fields instead of private).
2. **Common** — sharing access to the same global data/variables.
3. **External** — sharing something imposed by an external source, e.g. a data format.
4. **Control** — one class controls what happens in another by passing it information/instructions.
5. **Stamp** — sharing a data structure, but each class only needs access to a select part of it.
6. **Data** — sharing data through means such as parameter passing in methods.

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

7. **None** — no dependency at all (most loosely coupled).

Worked classification examples

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

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

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

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

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

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

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

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

Five separate simple parameters — still **data coupling**, though a long parameter list like this is itself often a separate code smell (see java-refactoring[19]) worth considering bundling into an object.

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

Cohesion

The degree of interrelatedness and focus among the elements *within* a module, class, or component.

- **High cohesion** — elements are closely related and contribute collectively to one specific functionality.
- **Low cohesion** — elements are less focused, serving multiple unrelated purposes.

Levels of cohesion (low → high)

1. **Coincidental** — elements grouped arbitrarily, with no clear relationship.
2. **Temporal** — elements grouped because they execute at the same time/phase.
3. **Procedural** — elements grouped by their involvement in a specific sequence of steps.
4. **Communicational** — elements work together to manipulate a shared data structure.
5. **Sequential** — elements organised in a linear sequence, where one's output is the next's input.
6. **Functional** — elements grouped around a single, specific functionality or task (most cohesive).

Worked example

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

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

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

Is this class cohesive?

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

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

Considerations

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

Good modularization (high cohesion, low coupling) groups tightly-interconnected elements together into the same module and minimises connections *between* modules, unlike bad modularization, where connections are scattered across module boundaries with no clear grouping.

Java Generics

Introduced in 2026-04-30-generics-in-java[20] (Lecture, Week 9).

Why generics?

Prior to Java 1.5, collection classes could hold *any* type of object — `ArrayList.add(Object o)` accepted anything, allowing types to be mixed, but required an explicit cast when retrieving elements:

```
List list = new ArrayList();
list.add("Hello");
list.add(123);           // allowed
String str = (String) list.get(0);   // OK, but no compile-time guarantee
Integer num = (Integer) list.get(1);
```

There was no guarantee only one type would end up in the collection, since `add` accepted any `Object` — a `String` could silently get mixed into a list that was only ever meant to hold `Car`s. **Generics** let a class/interface/method declare *which* type(s) it works with, giving:

[20] courses/csse2002/2026-04-30-generics-in-java.pdf

```
List<Car> carList = new ArrayList<Car>(); // or new ArrayList<>() - the diamond
 ↪  operator
carList.add("Not a product"); // compile-time error, not a runtime surprise
Car c = carList.get(0);       // no cast needed - the compiler already knows it's a
 ↪  Car
```

This is called **type safety** — catching and preventing type-related errors *early* (at compile time), rather than allowing unintended bugs (a stray `ClassCastException`) to surface at runtime.

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> p1 = new Print<>(1);
Print<Double> p2 = new Print<>(1.1);
Print<String> p3 = new Print<>("1.1");
```

By convention, type parameters are single letters: `T` (Type), `E` (Element — used throughout the Collections Framework), `K` (Key), `V` (Value), `N` (Number), and `S`/`U`/`V`… for a 2nd/3rd/4th type parameter.

Generic methods

A single method (rather than a whole class) can be made generic, independently of whether its enclosing class is generic:

```
public <T> void print(T data) {
    System.out.println("Lets shout " + data + "!!!");
}
// print(124);              // T inferred as Integer
// print("I love CSSE2002"); // T inferred as String
```

The compiler determines `T` from the argument's type automatically — this is **type inference**. Multiple type parameters can be declared together: `public <T, S, R> R genericMethod(T value1, S value2) { ... }`.

Bounded generics

A type parameter can normally accept *any* reference type. **Bounded generics** restrict it to a specific type (or one of its subtypes) using `extends`:

```
public <T extends Number> void print(T data) {
    System.out.println("Lets shout " + data + "!!!");
}
// print(123);    // OK, Integer is a Number
// print(123.2);  // OK, Double is a Number
// print("123");  // compile-time error, String is not a Number
```

`<T extends X>` means `T` can only accept data that are subtypes of `X` (despite the keyword, this works the same way whether `X` is a class or an interface).

Bounds through wildcards

Wildcards (`?`) bound what types can be substituted for a generic type parameter *at the use site* (e.g. in a method parameter), rather than when a generic class/method is declared.

Unbounded wildcard: `List<?>`

`List<?>` means "a list of some unknown type". Since Java doesn't know whether it's actually `List<String>`, `List<Integer>`, or something else, it **prevents writes**:

```
public static void addSomething(List<?> list) {
    list.add("Hello"); // compile-time error - could break whatever the real element
 ↪  type is
}
```

Use `<T>` when the method needs to *work with* the type; use `<?>` when the method only needs to *read* values (e.g. a `printList` that only calls `.get()`/iterates never needs to know the concrete type).

Upper bounded wildcard: `? extends T`

Represents `T` or any subclass of `T`. Used mainly to **read** data — you can't safely *add* to it, because the method doesn't know the exact subtype:

```
List<? extends Number> nums = new ArrayList<Integer>();
Number n = nums.get(0); // OK: read as Number
nums.add(10);           // compile-time error - could be a List<Double>, etc.
```

This matters because **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>`). So a method that should accept a `List<Integer>` *or* a `List<Double>` and only needs to read `Number`s from it must be written as:

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

Lower bounded wildcard: `? super T`

Represents `T` or any of its superclasses. Used when you want to **write** to a generic collection while keeping some flexibility in what type of collection is accepted:

```
public static void addNumbers(List<? super Integer> list) {
    list.add(1);
    list.add(2);
}
// addNumbers(new ArrayList<Number>());  // OK, Number is a superclass of Integer
// addNumbers(new ArrayList<Object>());  // OK, Object is a superclass of Integer
// addNumbers(new ArrayList<Double>());  // compile-time error, Double isn't a
 ↪  superclass of Integer
```

Wildcards are usually bounded (`? extends T`, `? super T`) rather than left fully unbounded, to reduce flexibility just enough to improve type safety.

Type erasure

Generics only provide type checking at **compile time** — Java implements them via **type erasure**, which removes all type parameter information during compilation. Each type parameter is replaced with:

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

As a result, generic type information is not available at runtime — a compiled `List<String>` and a compiled `List<Integer>` are the same erased `List` class. This is why the compiler needs to insert a hidden cast for you:

```java
// Source (compile time)
List<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0);

// After erasure (what actually runs)
List list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0); // compiler inserts this cast
```

For a bounded type parameter, the compiler substitutes the bound itself:

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

Practical example: a bookshop

A bookshop sells books across genres (`Fiction`, `Action`, `Fantasy`, …), stored on shelves; the shop needs to store and retrieve books of these different genres.

Without generics

A single `BookShelf` storing the abstract `Book` type loses genre information on retrieval — every `getItem()` call needs an explicit (unsafe) cast back to the specific genre:

```java
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); }
}
// Fiction f = (Fiction) shelf.getItem(0); // risk of ClassCastException, not type
 ↪  safe
```

Writing separate `FictionShelf`/`ActionShelf` classes fixes the type safety but duplicates the whole class for every genre.

With generics

A single generic `BookShelf<T extends Book>` gives type safety *and* reuse in one class:

```
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"));
actionShelf.addItem(new Action("Die Hard", "Roderick Thorp"));
fictionShelf.displayBooks(); // getItem() on fictionShelf now returns Fiction
 ↪  directly, no cast
```

Java Refactoring

Introduced in 2026-04-02-refactoring[21] (Lecture, Week 6).

What is refactoring?

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

Why refactor?

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

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

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[22]).
- 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.

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"` case means editing `travelTime` *and* remembering every other place that branches on `type`:

```
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[23]), 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; }
}
```

[22] courses/csse2002/java-testing.pdf
[23] courses/csse2002/java-polymorphism.pdf

```java
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; }
}
```

Example: making it easier to understand

Extracting named boolean-returning methods (`isHotAndSunny()`, `isPleasant()`) out of inline compound conditions makes the *intent* of a branch obvious at a glance, instead of making the reader re-derive it from raw comparisons every time:

```java
// Before
if (temperature > 25 && !isRaining) { ... }
else if (temperature <= 25 && !isRaining) { ... }
else if (isRaining) { ... }

// After
if (isHotAndSunny()) { ... }
else if (isPleasant()) { ... }
else if (isRaining) { ... }
```

Code smells

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

Bad naming — single-letter/cryptic names (`r`, `x`, `y`, `g(x)`) force the reader to trace logic to understand intent; descriptive names (`evenCount`, `number`, `isEven(number)`) make code self-documenting.

Duplication — near-identical blocks repeated for different data (e.g. computing an average for two separate arrays with two copy-pasted loops) should be extracted into a single shared method (`average(int[] numbers)`), so a bug fix or improvement only needs to happen once.

Feature envy — a method that spends most of its time reaching into *another* class's data (`cart.getItems()`, `cart.getVoucher()`) rather than its own is a sign the method's logic actually belongs on the class it's "envious" of. Moving `checkout()` onto `ShoppingCart` itself

(next to `getItems`/`getVoucher`) removes the envy and reduces coupling between `User` and `ShoppingCart` (see java-cohesion-and-coupling[24]).

Many other code smells exist — see Martin Fowler's *Refactoring* (2nd ed., 2019), Robert Martin's *Clean Code* (2008), John Ousterhout's *A Philosophy of Software Design* (2018), and the refactoring.com catalog.

Java SOLID Principles

Introduced in 2026-04-02-refactoring[25] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[26] (Lecture, Week 7 — Liskov Substitution, Interface Segregation, Dependency Inversion).

Why SOLID?

Large software tends to become:

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

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

Single Responsibility Principle (SRP)

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

Closely related to java-cohesion-and-coupling[27]'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.

[24] courses/csse2002/java-cohesion-and-coupling.pdf
[25] courses/csse2002/2026-04-02-refactoring.pdf
[26] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[27] courses/csse2002/java-cohesion-and-coupling.pdf

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

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

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

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

Open-Closed Principle (OCP)

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

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

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

How do we keep a class closed for modification? Information hiding (see java-encapsulation[28]).

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

Example: pluggable grading systems

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

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

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

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

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

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

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

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

Liskov Substitution Principle (LSP)

> Subclasses should be substitutable for their parent classes.

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

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

Example violation

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

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

Substitution with contracts

Even when a subclass overrides a method with a different implementation, it must still honour the parent's contract (see java-specification[29]):

```
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
     */
```

[29] 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[30]'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).

[30] 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[31]):

[31] 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[32] (Lecture, Week 3, Part 2).

Javadoc

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

```
/**
 * Calculates a sum by combining the hash code of the provided string
 * and the long value of the provided float.
 *
 * @param inputString the input string whose hash code will be used
 * @param inputFloat the float value to be converted to long and combined with the
  ↪  string hash code
 * @return a long value representing the sum of the string's hash code and the long
  ↪  value of the float
 */
public long doCalculation(String inputString, float inputFloat) { ... }
```

Common tags:

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

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

Tag	Meaning
`@author authorname`	Author of the class

What makes a good specification

Ideally, a specification should:

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

Restrictiveness — keep out incorrect implementations

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

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

Generality — allow acceptable alternative versions

```
/**
 * Examine ar[0], ar[1], ... in turn and return the index of the
 * first one that is equal to x, if any, else return -1.
 */
public int search(int[] ar, int x) { ... }
```

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

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

Clarity

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

Formality

Specifications range from informal to formal:

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

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

Contracts

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

- If the caller satisfies the precondition, the method guarantees to satisfy the postcondition.

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

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

Defensive programming

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

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

```java
/**
 * Finds the maximum value in the given array of integers.
 *
 * @throws IllegalArgumentException if the array is null or empty.
 * @requires numbers != null && numbers.length > 0
 * @ensures the method returns the maximum value found in the array.
 */
public int findMax(int[] numbers) {
    if (numbers == null || numbers.length == 0) {
        throw new IllegalArgumentException("Array must not be null or empty.");
    }
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < numbers.length; i++) {
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max;
}
```

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