Course Summary and Exam Preparation

Table of contents

What is not examinable 1
Exam structure 2
Practice questions by week 3
What a real paper looked like 22
Closing 24

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

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

What is not examinable

Four topics are **not examinable**:

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

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-version-control.pdf
[3] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[4] courses/csse2002/2026-05-21-correct-programming.pdf

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

Exam structure

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

Question	Marks	Type
Q1–Q8	16 total	MCQ
Q9	12	short/long answer
Q10	10	short/long answer
Q11	8	short/long answer
Q12	14	short/long answer
Q13	15	short/long answer
Q14	10	short/long answer
Q15	15	short/long answer

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

Conditions, from the cover sheet

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

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

The rest of the cover sheet:

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

[5] courses/csse2002/java-hoare-logic.pdf
[6] courses/csse2002/java-loop-invariants.pdf
[7] courses/csse2002/java-events.pdf

- Two blank pages are added at the end. If you use them, clearly mark which question you are answering, **and** indicate in the designated answer space that your response continues on the blank pages.

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

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

Practice questions by week

Week 1 — Java basics

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

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

Week 1 — Collections and strings

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

Helpful `String` methods (not exhaustive):

Signature	Meaning
`int String.length()`	Return the length of the string.
`char String.charAt(int)`	Return the char at the given index.
`String String.substring(int, int)`	Return the substring between two indexes.

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

Signature	Meaning
`String String.replace(char, char)`	Replace occurrences of char with char.
`int String.indexOf(char)`	Return the index of a given char.

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

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

5. What is the output? — `String name = "CSSE2002"; int times = name.length() / 4; String result = ""; for (int i = 0; i <= times; i++) { result += name.charAt(i * 2); } result += name.substring(2, 5);`
6. Write `contains(int[], int)` returning `true` if the array contains the value.
7. What is the main difference between an array and a list?
8. What are two differences between a list and a set?
9. When would you use a map instead of a set?

i Q6 — answer

```
boolean contains(int[] array, int value) {
    for (int num : array) {
        if (num == value) {
            return true;
        }
    }
    return false;
}
```

The enhanced-for body is the answer. The `public static` modifiers are not part of what was asked.

Weeks 2–3 — OOP

Classes, objects, and the four principles: **inheritance, polymorphism, encapsulation, abstraction**. The internal state of a class must be protected to avoid leakage. In Java the access

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

modifiers that help are `private`, `public`, `protected`, and **default — no keyword required**. See java-encapsulation[10], java-inheritance[11], java-polymorphism[12], java-abstraction[13].

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

    Class invariants must be true after instance construction and after execution of each method invocation. For Q12 and Q13: an invariant must be established after the class constructor; it may be assumed as a precondition of each method (**excluding the constructor**); and it must be established after each method call. See week8-tutorial-class-invariants[15].

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

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

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

[10] courses/csse2002/java-encapsulation.pdf
[11] courses/csse2002/java-inheritance.pdf
[12] courses/csse2002/java-polymorphism.pdf
[13] courses/csse2002/java-abstraction.pdf
[14] courses/csse2002/java-mutability.pdf
[15] courses/csse2002/week8-tutorial-class-invariants.pdf

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

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

Q19–Q21 — the Batman overload/override problems

This is the same two-phase rule as the week-4 applied class: **the overload is chosen at compile time from the static types, and only then is the override resolved at run time.** See java-casting[16] and week4-tutorial-method-dispatch-and-casting[17].

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

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

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

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

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

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

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

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

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

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

`kapow(Enemy)` is locked in.

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

CodeCamp — `BookShelf` / `BargainBin`

Same idea, different names:

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

```
class TextBook    extends Book {}

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

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

i Working

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

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

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

Other question types

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

Week 4 — Exceptions and specification

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

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

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

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

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

23. Same question, different method:

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

24. Rewrite `concat` following defensive programming techniques:

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

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

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

Week 5 — Testing and JUnit

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

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

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

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

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

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

28. Explain the process of test-driven development.

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

Q31–Q33 — coverage of `register`

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

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

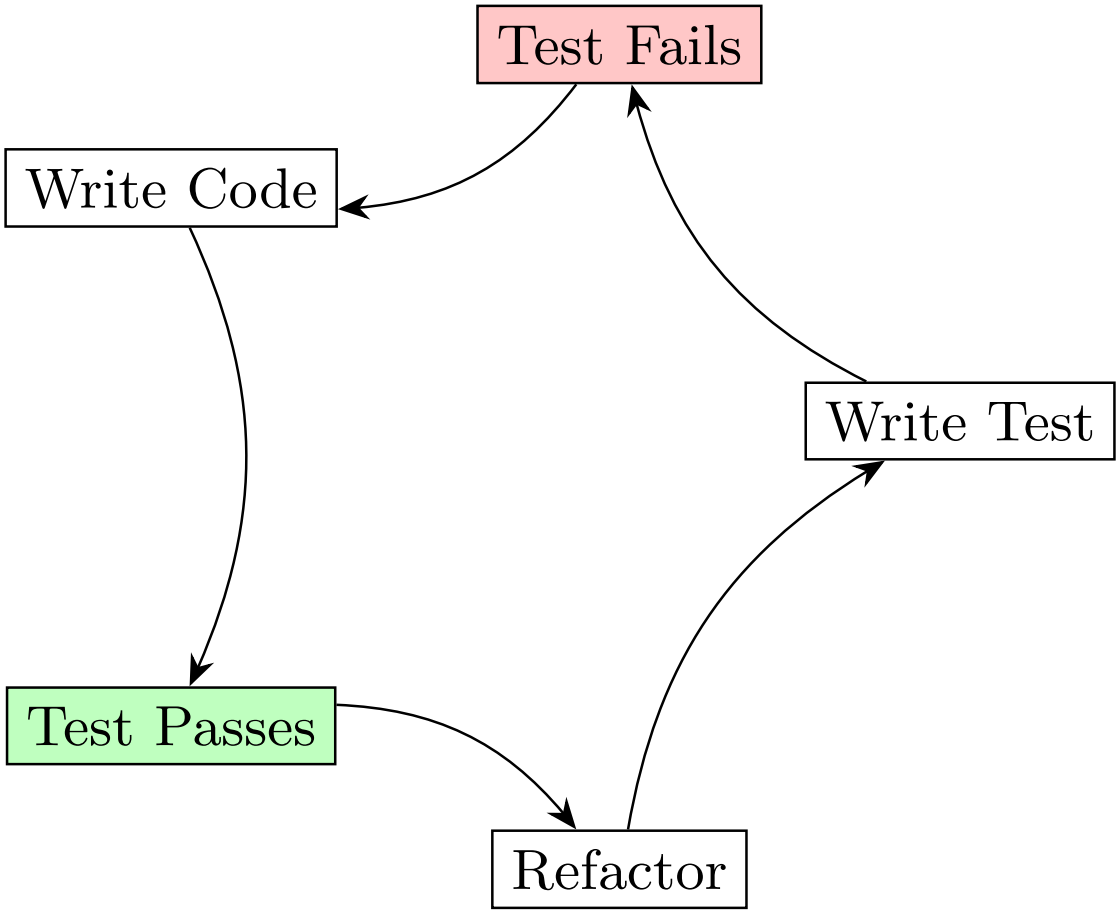


Figure 1: The TDD cycle: write the test first, watch it fail, write just enough code to pass, then refactor and go round again. The failing state is pink and the passing state is green.

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

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

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

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

> **Working**
>
> **Q31/Q32.** `x = 2` makes both conditions true, so every statement in the method runs — **statement coverage**, and no further. Neither `if`'s false branch is ever taken, so branch coverage (and therefore path coverage) is not reached.
> **Q33.** The two `if`s are independent, so there are four paths. One input per path:
>
> | Input | `x > 0` | `x % 2 == 0` | Path |
> |---|---|---|---|
> | `2` | T | T | both counters incremented |
> | `1` | T | F | positives only |
> | `-2` | F | T | evens only |
> | `-1` | F | F | neither |
>
> Four inputs, and four is minimal — one per path.

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

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

```
/**
 * @param array An array of numbers to find the range of.
 * @requires array != null
 * @ensures if array.length > 0 then \result == max(array) - min(array)
 *          otherwise \result == 1
 */
public static int range(int[] array) {
```

```
    int min = Integer.MAX_VALUE;
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < array.length; i++) {
        if (array[i] < min) {
            min = array[i];
        }
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max - min;
}
```

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

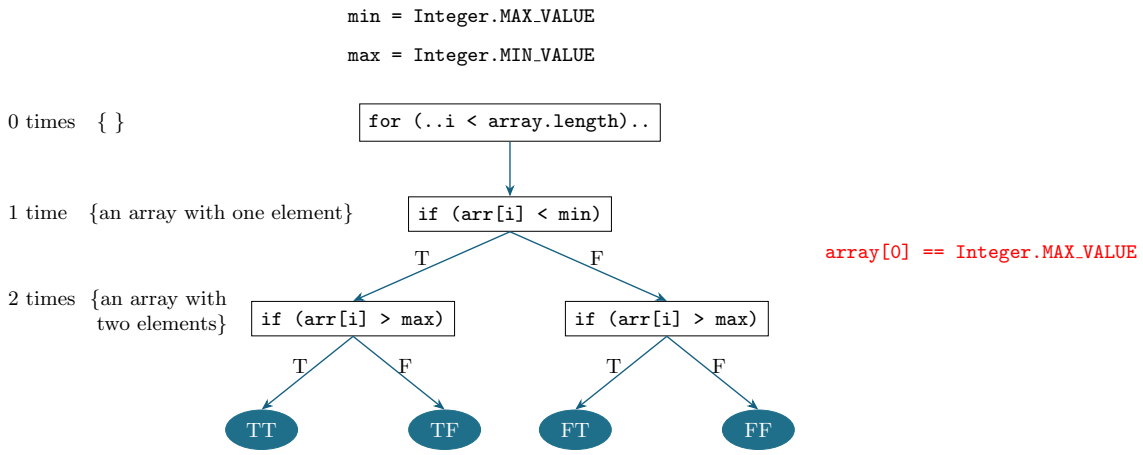


Figure 2: The decision tree for one iteration of the `range` loop, together with the initial values of `min` and `max` that decide which leaves are reachable. Each leaf is one path through the pair of `if`s. The red condition `array[0] == Integer.MAX_VALUE` names the only value that can take the first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

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

One iteration — four leaves, one impossible:

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

Path	Input
`T F`	`[Integer.MIN_VALUE]`
`T T`	`[1]`

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

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

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

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

Week 6 — Cohesion and coupling

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

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

Weeks 6–7 — SOLID principles

Applying SOLID principles helps to prevent rigid and fragile software.

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

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

Q40–Q41 — the substitution principle, drawn

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

40. Do the classes below satisfy the substitution principle?

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

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

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

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

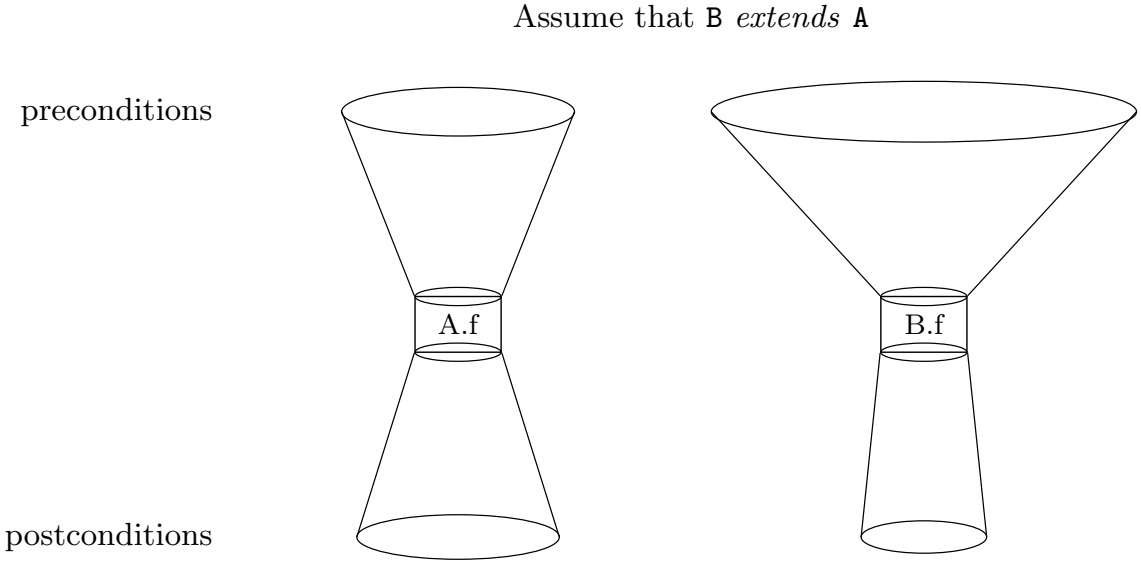


Figure 3: Pre/postcondition funnels. A method is the small box; the cone above it is the set of inputs it accepts (its precondition) and the cone below is the set of outcomes it may produce (its postcondition) — so a **wider** mouth means a **weaker** condition. `B.f` here has a wider top and a narrower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

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

Working

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

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

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

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

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

Q43–Q45 — dependency inversion

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

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

```
    private int height;

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

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

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

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

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

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

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

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

    public Square(int width, int height, Printer printer) {
```

```
        this.width = width;
        this.height = height;
        this.printer = printer;
    }

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

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

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

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

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

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

The substitution rule to remember: **a subclass may weaken preconditions and strengthen postconditions, never the reverse.** See java-solid-principles[25], week9-tutorial-substitution-principle-and-pre-post-conditions[26], week11-tutorial-dependency-inversion[27].

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

Week 8 — Java I/O

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

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

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

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

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

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

Week 9 — Generics

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

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

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

[28] courses/csse2002/java-io.pdf
[29] courses/csse2002/java-generics.pdf

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

    Before, the non-generic version:

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

    After, the generic version:

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

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

Week 11 — Lambdas

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

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

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

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

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

Week 12 — Correct programming *(not examinable)*

Correctness · Deriving Pre-conditions · Loop Invariants.

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

What a real paper looked like

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

Q	Marks	Shape	Answers given
1	10	Five 2-mark traces	`A.f(4)` → `3.46666667`, rounded `3.47`; `d(1234, 10)` → `{4, 3, 2, 1}` and `d(10, 2)` → `{0, 1, 0, 1}`; then the two `BookShelf`/`BargainBin` dispatch parts, `20` and `2`
2	4	Fill three holes in a merge-of-sorted-arrays method	= `array.length`, = `arrays[i].length`, = `i`
3	6	Recursion from a piecewise mathematical definition (Eulerian numbers)	precondition `n > 0 && k >= 0`, then a direct three-case recursive implementation

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

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

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

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

- Given `a.f(5)` **satisfies** `A.f`'s precondition, what do we know about `b.f(5)`? — *"We know that the call to `B.f` will satisfy the precondition … the precondition of a subclass must be weaker than or equal to the precondition of the superclass."*
- Given `a.f(5) == 10` **satisfies** `A.f`'s postcondition, what about `b.f(5) == 10`? — *"May or may not. It is possible for the postcondition of `B.f` to be stronger than the postcondition of `A.f`, therefore `10` may not be a valid return value for `B.f`."*

- Given `a.f(5) == -10` **violates** `A.f`'s postcondition, what about `b.f(5) == -10`? — *"We know that the postcondition of `B.f` will be violated … the postcondition cannot be weakened."*

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

Closing

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