CSSE2002 — Week 8 Notes

Table of contents

Java I/O **2**
- Topics . . . 2
- The thread running through it . . . 2
- Set as homework . . . 3
- Applied class . . . 3
- Practical . . . 3

JUnit Testing **3**
- Test Driven Development (TDD) . . . 3
- `DistinctCounter` . . . 4
- `PalindromeCounter` . . . 12

Class Invariants **16**
- `XFiles` . . . 16
- `Cinema` and `Screening` . . . 18

Reference material **21**
- Java Encapsulation . . . 21
- Java Exceptions . . . 31
- Java I/O . . . 38
- Java JUnit . . . 51
- Java Loop Invariants . . . 58
- Java SOLID Principles . . . 66
- Java Specification . . . 77
- Java Testing . . . 89

Java I/O

See csse2002[1] for course logistics.

Topics

1. Streams (`java.io`, since Java 1.0) — byte-oriented
2. Readers & Writers (`java.io`, since Java 1.1) — character-oriented
3. Scanner (`java.util`, since Java 1.5) — a convenience wrapper, neither stream nor reader
4. New I/O (`java.nio.file`, since Java 1.7) — the file system itself

Instance Saving & Loading — saving and loading an object model to a file — is taken up in the Week 10 practical, week10-lab-java-io[2].

All content is in java-io[3].

The thread running through it

Each generation of the API is a response to what the previous one was awkward at:

- Byte streams give one uniform abstraction over any source or destination, so a method can take an `OutputStream` and not care whether it is a console, a file, or an array (see java-solid-principles[4] — Liskov Substitution).
- They were never ideal for text, so **Readers & Writers** were added to work in `char`s and Unicode. `InputStreamReader` is the bridge back to the byte world.
- Everything in `java.io` is assembled by **wrapping** one object around another — buffering, line-reading and formatting are separate classes you stack, not options on one class.
- None of it was designed for managing files and directories, so **New I/O** adds `Path` and `Files`.
- Every stream and reader must be **closed**; `try`-with-resources is the way to do it (Effective Java Item 9). This is where the topic leans hardest on java-exceptions[5].

Summary — which to use? Streams for binary data, Readers & Writers for text, New I/O when manipulating a file system or file contents.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/week10-lab-java-io.pdf
[3] courses/csse2002/java-io.pdf
[4] courses/csse2002/java-solid-principles.pdf
[5] courses/csse2002/java-exceptions.pdf

Set as homework

- Read through `flush()`.
- Scanner vs. BufferedReader on Stack Overflow.
- Rewrite the stream and reader exercises using New I/O.

Applied class

See week8-tutorial-class-invariants[6] — protecting class invariants (a continuation of java-specification[7] and java-encapsulation[8] from earlier weeks).

Practical

See week8-lab-junit-testing[9] — Test Driven Development with JUnit, extending java-junit[10] from Week 5.

JUnit Testing

Practical for 2026-04-23-java-io[11] (Week 8). Extends java-junit[12] (Week 5) with a full Test Driven Development (TDD) worked example.

Test Driven Development (TDD)

Cycle (see java-testing[13]):

1. Write a test for some piece of functionality.
2. Write **just enough** functionality for the test to pass.
3. Refactor implementation while preserving the passing test.
4. Repeat.

[6] courses/csse2002/week8-tutorial-class-invariants.pdf
[7] courses/csse2002/java-specification.pdf
[8] courses/csse2002/java-encapsulation.pdf
[9] courses/csse2002/week8-lab-junit-testing.pdf
[10] courses/csse2002/java-junit.pdf
[11] courses/csse2002/2026-04-23-java-io.pdf
[12] courses/csse2002/java-junit.pdf
[13] courses/csse2002/java-testing.pdf

Step 3's refactoring can be of the implementation under test **or of the test code itself**.

Stubs: a stub class has all public members, but methods return dummy values based on their return type — `void` methods are empty, primitives return a 'default' value such as `0` for integers, reference types normally return `null`. This lets you write an outline of a class that compiles but may not work.

> In pure test driven development, not compiling is considered a test failure, so a test is written for a class **before it exists**, and creating the class is considered a part of making a test pass.

`DistinctCounter`

Tracks a collection of distinct strings, which may be added to the collection and retrieved in lexicographical order:

- `DistinctCounter()`
- `void add(String element)`
- `int getDistinctCount()`
- `String[] getStrings()` — in lexicographical order

```
DistinctCounter distinct = new DistinctCounter();
distinct.add("Z");
distinct.add("Hello");
distinct.add("Z");
distinct.add("Hello ");
distinct.getDistinctCount(); // 3
distinct.getStrings();       // {"Hello", "Hello ", "Z"}
```

`"Hello"` and `"Hello "` are distinct — the trailing space matters — and `"Hello "` sorts before `"Z"`.

Task 0 — the stub

```
class DistinctCounter {
    public DistinctCounter() {}
    void add(String element) {}
    int getDistinctCount() {
        return 0;
    }
    String[] getStrings() {
        return null;
```

```
    }
}
```

Task 1 — the JUnit 4 test class

```
import org.junit.Test;

import static org.junit.Assert.*;

class DistinctCounterTest {
    @Test
    public void testEmpty() {}
}
```

Task 2 — the test cases

Write each of the following, and after each one write just enough code to make it pass, then refactor. The count cases:

1. An empty counter has a distinct count of zero.
2. A counter with one element has a count of one.
3. A counter with two distinct elements has only a count of two.
4. A counter with two identical elements has a count of one.
5. A counter with two identical elements and one other unique element has a count of two.
6. A counter with N distinct elements has a counter of N for $N > 4$.

The array cases mirror those six, plus three extra cases that ask specifically for a **sorted** array:

7. An empty counter has an empty array of strings.
8. A counter with one element has an array with that element.
9. A counter with two distinct elements has an array of those two elements.
10. A counter with two distinct elements has a **sorted** array of those two elements.
11. A counter with two identical elements has an array of one element.
12. A counter with two identical elements and one other unique element has an array of two elements.
13. A counter with two identical elements and one other unique element has a **sorted** array of two elements.
14. A counter with N distinct elements has an array of N elements for $N > 4$.
15. A counter with N distinct elements has a **sorted** array of N elements for $N > 4$.

Working

There are many different ways to implement `DistinctCounter`, so it becomes an **interface** with four different implementations behind it — which is exactly the point of writing the tests first: all four pass the same fifteen cases of Task 2.

```
public interface DistinctCounter {
    void add(String value);
    int getDistinctCount();
    String[] getStrings();
}
```

1 — a hash set. A set won't add duplicates, so `add` is trivial and the sorting happens on the way out:

```
public class DistinctCounterHashSet implements DistinctCounter {
    private final Set<String> distinct;

    public DistinctCounterHashSet() {
        distinct = new HashSet<>();
    }

    public void add(String word) {
        // set won't add duplicates
        distinct.add(word);
    }

    public int getDistinctCount() {
        return distinct.size();
    }

    public String[] getStrings() {
        String[] elements = new String[distinct.size()];
        int index = 0;
        for (String element : distinct) {
            elements[index] = element;
            index++;
        }
        Arrays.sort(elements);
        return elements;
    }
}
```

2 — a tree set, which automatically keeps elements in sorted order, so `getStrings()` needs no `Arrays.sort` at all:

```java
public class DistinctCounterTreeSet implements DistinctCounter {
    /** TreeSet maintains Strings in lexicographical order **/
    private final Set<String> distinct;

    public DistinctCounterTreeSet() {
        distinct = new TreeSet<>();
    }

    public void add(String word) {
        // set won't add duplicates
        distinct.add(word);
    }

    public int getDistinctCount() {
        return distinct.size();
    }

    public String[] getStrings() {
        return distinct.toArray(new String[]{});
    }
}
```

3 — an array list, sorted on get. Without a set, `add` must check `contains` itself:

```java
public class DistinctCounterArrayListOnGet implements DistinctCounter {
    private final List<String> distinct;

    public DistinctCounterArrayListOnGet() {
        distinct = new ArrayList<>();
    }

    public void add(String word) {
        if (distinct.contains(word)) {
            return;
        }
        distinct.add(word);
    }

    public int getDistinctCount() {
        return distinct.size();
    }

    public String[] getStrings() {
        String[] elements = distinct.toArray(new String[]{});
        Arrays.sort(elements);
        return elements;
    }
}
```

4 — `DistinctCounterArrayListOnAdd`, an array list sorted on add — an insertion sort, walking to the first element that sorts after `word` and inserting there; if no such element exists, `word` sorts after everything already in the list and is appended. Because the list is already ordered, this class's `getStrings()` is the bare `return distinct.toArray(new String[]{});` — no `Arrays.sort`, unlike implementation 3. Only `add` is given here:

```
public void add(String word) {
    if (distinct.contains(word)) {
        return;
    }
    boolean added = false;
    for (int i = 0; i < distinct.size(); i++) {
        if (distinct.get(i).compareTo(word) > 0) {
            distinct.add(i, word);
            added = true;
            break;
        }
    }
    if (!added) {
        distinct.add(word);
    }
}
```

The tests. `@Before` constructs an instance of `DistinctCounter` rather than duplicating the creation in every test method. The `new DistinctCounter()` in `setup()` below no longer compiles once `DistinctCounter` is the interface; in practice you construct whichever of the four implementations you are testing.

```
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class DistinctCounterTest {
    private DistinctCounter counter;

    @Before
    public void setup() {
        counter = new DistinctCounter();
    }
    ...
}
```

Some insist that each test method should have **only one assertion**. The tests here do that because it corresponds to the test cases listed for the exercise, but it is better to refactor tests to have two assertions than to duplicate the initialisation code.
That gives twelve test methods, not fifteen. The six count cases map one-to-one, but the nine array cases collapse to six methods, because each "…has a **sorted** array of…" case is already pinned down by the expected array in the case above it.

```java
@Test
public void testEmptyCounterCount() {
    assertEquals("Empty counter does not have a count of zero",
            0, counter.getDistinctCount());
}

@Test
public void testTwoIdenticalCount() {
    counter.add("A");
    counter.add("A");
    assertEquals("Counter with two identical elements does not have count of
     ↪  one",
            1, counter.getDistinctCount());
}

@Test
public void testEmptyArray() {
    assertArrayEquals("Empty counter does not have an empty array",
            new String[]{}, counter.getStrings());
}

@Test
public void testTwoDistinctArray() {
    counter.add("A");
    counter.add("B");
    assertArrayEquals("Counter with two distinct elements does not have an array
     ↪  of two",
            new String[]{"A", "B"}, counter.getStrings());
}

@Test
public void testNDinstinctElementsArray() {
    counter.add("E"); counter.add("A"); counter.add("B"); counter.add("A");
    counter.add("A"); counter.add("B"); counter.add("D"); counter.add("A");
    counter.add("A"); counter.add("C"); counter.add("B");
    assertArrayEquals("Counter with 5 distinct elements does not have an array
     ↪  of five",
            new String[]{"A", "B", "C", "D", "E"}, counter.getStrings());
}
```

In the last one the elements are added deliberately out of order and with repeats, so the expected array simultaneously pins down *distinctness*, *count*, and *sorted order*.

Four implementations, one interface, one shared test suite — and, looking ahead, one of those four is the class that `PalindromeCounter` actually extends:

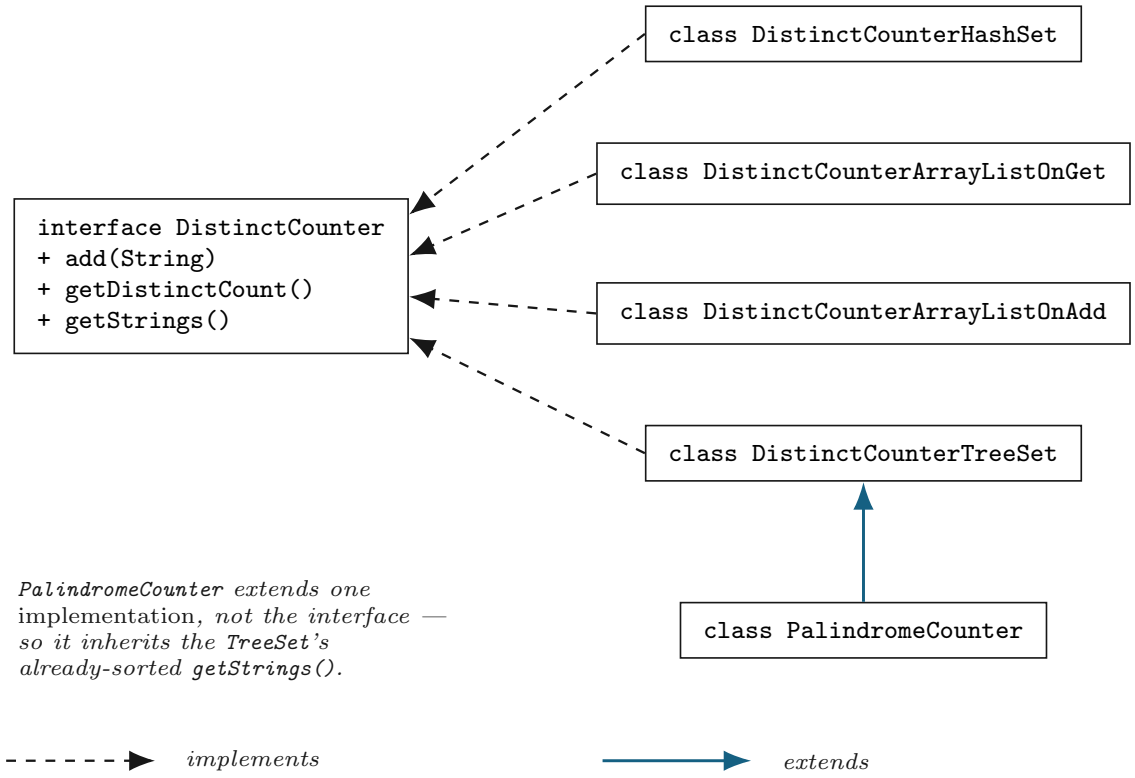


Figure 1: The four interchangeable implementations of one `DistinctCounter` interface, and — below them — the one of the four that `PalindromeCounter` actually extends. The dashed line running from implementer to interface is the *implements* arrow. In the course's dependency diagrams `Uses` is a solid line and `Implements` a dashed line, **both** with the same small **filled** arrowhead; this course does *not* use UML's hollow realisation triangle. The solid teal arrow into the superclass is the generalisation (*extends*) connector.

Task 3 — a boundary case

Test driven development will result in a highly comprehensive test suite, with every component rigorously tested. One possible downside is that it **tends to target "happy path" execution** — so it is worth adding boundary cases (see java-testing[14]) deliberately.

> **i** Working
>
> ```
> @Test
> public void testNullStringCount() {
>     counter.add(null);
>     assertEquals(0, counter.getDistinctCount());
> }
>
> @Test
> public void testNullStringArray() {
>     counter.add(null);
>     assertArrayEquals(new String[]{}, counter.getStrings());
> }
> ```
>
> The exercise asks *how does your implementation handle this case?* Only `DistinctCounterTreeSet` **throws**, and it throws on `add` itself, because a `TreeSet` has to compare the incoming element against what it holds. The other three accept `null`, so the count becomes 1 and both assertions simply **fail**. In particular `DistinctCounterArrayListOnAdd` never reaches `compareTo` here — on a fresh counter the list is empty, so its insertion loop has nothing to walk and the `null` is appended. (It, and the two implementations that call `Arrays.sort` on the way out, would hit a `NullPointerException` on a `null` only once the counter already holds another element to compare against.) Deciding which behaviour is correct is a specification question, not an implementation one — see java-specification[15].

`PalindromeCounter`

`PalindromeCounter` inherits from `DistinctCounter` and extends it with:

1. `int getPalindromeCount()` — number of distinct palindromes
2. `String[] getPalindromes()` — distinct palindromes
3. `String[] getNonPalindromes()` — distinct non-palindromes

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

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

Tasks 4 and 5 repeat Tasks 0 and 1 for this class: write the stub, then create the JUnit 4 test class, following the `DistinctCounter` pattern. **Task 6** then builds up test cases and functionality together, TDD-style.

i Working

It extends `DistinctCounterTreeSet` — one concrete implementation, not the interface — which is why `getStrings()` is already in lexicographical order and `getPalindromes()` can simply filter it in place:

```
public class PalindromeCounter extends DistinctCounterTreeSet {

    private static boolean isPalindrome(String word) {
        if (word.length() < 2) {
            return true;
        }
        if (word.charAt(0) != word.charAt(word.length() - 1)) {
            return false;
        }
        return isPalindrome(word.substring(1, word.length() - 1));
    }

    public int getPalindromeCount() {
        return getPalindromes().length;
    }

    public String[] getPalindromes() {
        List<String> palindromes = new ArrayList<>();
        for (String word : getStrings()) {
            if (isPalindrome(word)) {
                palindromes.add(word);
            }
        }
        return palindromes.toArray(new String[]{});
    }

    public String[] getNonPalindromes() {
        // wrapping in new array list required to allow removeAll
        List<String> distincts = new ArrayList<>(Arrays.asList(getStrings()));
        List<String> palindromes = Arrays.asList(getPalindromes());
        distincts.removeAll(palindromes); // removes the palindromes
        return distincts.toArray(new String[]{});
    }
}
```

Two details in `isPalindrome` worth pinning down: a string shorter than two characters is a palindrome **by definition** (the recursive base case), and the comparison is **case-**

sensitive on the characters as written — `"AbbA"` is a palindrome because `'A' == 'A'` and `'b' == 'b'`, no case-folding involved.
`getNonPalindromes` has to wrap `Arrays.asList(...)` in a `new ArrayList<>(...)`: `Arrays.asList` returns a fixed-size view, which does not support `removeAll`.
The tests, built up one case at a time — empty, a single palindrome, a single non-palindrome, one of each, then several of each. These are only a subset of the tests you could implement using test driven development; three of the five methods are given here:

```java
public class PalindromeCounterTest {
    private PalindromeCounter counter;

    @Before
    public void setup() {
        counter = new PalindromeCounter();
    }

    @Test
    public void testEmptyCounter() {
        assertEquals("Empty counter does not have a count of zero",
                0, counter.getPalindromeCount());
        assertArrayEquals("Empty counter does not have no palindromes",
                new String[]{}, counter.getPalindromes());
        assertArrayEquals("Empty counter does not have no non palindromes",
                new String[]{}, counter.getNonPalindromes());
    }

    @Test
    public void testSingleNonPalindromeCounter() {
        counter.add("car");
        assertEquals("Counter with one non-palindrome does not have count of
        ↪  zero",
                0, counter.getPalindromeCount());
        assertArrayEquals("Counter with one non-palindrome does not have empty
        ↪  palindromes",
                new String[]{}, counter.getPalindromes());
        assertArrayEquals("Counter with one non-palindrome does not have a
        ↪  non-palindromes array of that element",
                new String[]{"car"}, counter.getNonPalindromes());
    }

    @Test
    public void testManyOfEachCounter() {
        counter.add("car");
        counter.add("racecar");
        counter.add("mamma mia");
        counter.add("AbbA");
        counter.add("palindrome");
        counter.add("rufus");
        assertEquals("Counter with two palindromes has incorrect count",
                2, counter.getPalindromeCount());
        assertArrayEquals("Counter with multiple palindromes does not have all
        ↪  in array",
                new String[]{"AbbA", "racecar"}, counter.getPalindromes());
        assertArrayEquals("Counter with multiple non-palindromes does not have
        ↪  all in array",
                new String[]{"car", "mamma mia", "palindrome", "rufus"},
                ↪  counter.getNonPalindromes());
    }
}
```

> These test methods carry **three** assertions each — the refactoring recommended for the `DistinctCounter` tests but not done there, now that the three getters are all views of the same added words.

Class Invariants

Applied class for 2026-04-23-java-io[16] (Week 8). Applies java-encapsulation[17]'s class invariants (and the preconditions/postconditions from java-specification[18]) to worked examples.

Two failure modes run through the sheet, and it is worth naming them up front, because every *fix* it asks for is a response to one or the other. (The grouping is mine, not the sheet's; Tasks 0 and 3, which only ask you to state an invariant, sit outside it.)

1. **The specification lets the invariant break.** No precondition rules out the argument that would violate it.
2. **The implementation lets the invariant break**, even though the specification is fine — because it hands out a reference to its own internal state (see Figure 2).

`XFiles`

`XFiles` maintains the invariant that every stored string must be prefixed with `'X'`:

$$\forall\, 0 \le i < \texttt{getFiles().size()} \implies \texttt{getFiles().get(i).startsWith("X")}$$

```
/**
 * @invariant
 *     \forall i; 0 <= i < getFiles().size(); getFiles().get(i).startsWith("X")
 */
public class XFiles {
    /**
     * @ensures getFiles().contains(newFile)
     * @ensures getFiles().size() == \old(getFiles()).size() + 1
     */
    public void add(String newFile) {...}

    /**
     * @ensures !getFiles().contains(file)
     */
    public void remove(String file) {...}
```

[16] courses/csse2002/2026-04-23-java-io.pdf
[17] courses/csse2002/java-encapsulation.pdf
[18] courses/csse2002/java-specification.pdf

```
    public List<String> getFiles() {...}
}
```

As specified, **`add` allows the invariant to be broken**: given a string that does *not* start with `'X'`, the specification still states that the string will be included in `getFiles()`. Fix — failure mode (1) — add a precondition:

```
/**
 * @requires newFile.startsWith("X")
 */
```

Even with that precondition documented, a naive implementation still has two further leaks (see java-encapsulation[19] — Protecting invariants):

```
public class XFiles {
    public List<String> files = new ArrayList<>(); // (1) public field

    public void add(String newFile) {
        if (newFile.startsWith("X")) {
            files.add(newFile);
        }
    }
    public void remove(String file) { files.remove(file); }

    public List<String> getFiles() {
        return files; // (2) leaks the internal reference
    }
}
```

1. **`files` is public** — callers can grab the list directly and mutate it, bypassing `add`'s check entirely:

   ```
   List<String> secret = xFiles.files;
   secret.add("Doesn't start with X!"); // breaks the invariant
   ```

   Fix: `private List<String> files;`.

2. **`getFiles()` returns the internal reference** — even with `files` made private, the *returned* list is still the real one:

   ```
   List<String> secret = xFiles.getFiles();
   secret.add("Doesn't start with X!"); // still breaks the invariant
   ```

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

Fix: return a defensive copy, `return new ArrayList<>(files);`.

Leak (2) is failure mode (2), and it is the one worth having a picture of — the same leak, in the same shape, is the "real bug" of the `Screening` task below (`return tickets;` on a `Set<Ticket>` in place of `return files;` on a `List<String>`), and is drawn there in Figure 2.

`Cinema` and `Screening`

```
public class Cinema {
    /**
     * @ensures getCapacity() == capacity
     */
    public Cinema(int capacity) {...}
    public int getCapacity() {...}
}
```

> **i** Working
>
> **A useful invariant**: `getCapacity() >= 0`.
> **Precondition to preserve it**: the constructor needs `capacity >= 0`.

```
/**
 * @invariant getEndTime() > getStartTime()
 */
public class Screening {
    /** @ensures \result > 946648800 */
    public int getStartTime() {...}
    /** @ensures \result > 946648800 */
    public int getEndTime() {...}
    /** @ensures getEndTime() == \old(getEndTime()) + amount */
    public int extendRuntime(int amount) {...}
}
```

(All times are unix timestamps — seconds since 1 January 1970. `946648800` is 1 January 2000, when the cinema opened.)

> **i** Working
>
> **Do the methods preserve the invariant?** No — the invariant may be violated if a **negative** `amount` is given to `extendRuntime(int)`, pushing `getEndTime()` below (or equal to) `getStartTime()`. Two sensible preconditions on `extendRuntime`:
>
> a. `amount >= 0`

b. `getEndTime() + amount > getStartTime()`

The name of the method (**extend**Runtime) implies the first is more likely.

Now extend `Screening` with ticket sales:

```
public class Screening {
    // getStartTime(), getEndTime(), extendRuntime(int) as before.
    public Cinema getCinema() {...}

    /**
     * @ensures getSoldTickets().contains(\result)
     * @ensures getSoldTickets().size() == \old(getSoldTickets()).size() + 1
     */
    public Ticket sellTicket() {...}

    public Set<Ticket> getSoldTickets() {...}
}
```

Working

Invariant to stop the screening overselling:

```
/**
 * @invariant getSoldTickets().size() <= getCinema().getCapacity()
 */
```

Precondition to preserve it — added to `sellTicket()`, so we don't sell into a full cinema:

```
getSoldTickets().size() < getCinema().getCapacity()
```

Technically, since the invariant may itself be *assumed* as another precondition, specifying `getSoldTickets().size() != getCinema().getCapacity()` is enough — that inequality and the invariant together imply the strict `<`.

Now consider a concrete (simplified) implementation:

```
public class Screening {
    private Set<Ticket> tickets = new HashSet<>();

    public Ticket sellTicket() {
        Ticket ticket = new Ticket(...);
        tickets.add(ticket);
```

```
        return ticket;
    }

    public Set<Ticket> getSoldTickets() {
        return tickets;
    }
}
```

Working

Does this protect the invariant? Not really, for two reasons.
1. `sellTicket()` never checks its precondition. In true programming by contract that is defensible — we make no guarantees about what happens when a precondition is violated. But if `sellTicket()` is likely to be exposed to others, we should aim to program **defensively** as well as by contract (see java-specification[20]), which improves robustness:

```
public Ticket sellTicket() {
    if (tickets.size() >= getCinema().getCapacity()) {
        return null;
    }
    ...
}
```

That protects the invariant but potentially introduces a difficult-to-track-down `null` bug in the calling code. A better solution:

```
public Ticket sellTicket() {
    if (tickets.size() >= getCinema().getCapacity()) {
        throw new IllegalArgumentException("Screening full!");
    }
    ...
}
```

2. The real bug is that `getSoldTickets()` exposes the reference to the internal `tickets` set, allowing the screening to become oversold without `sellTicket()` ever being called:

```
Screening screening = // screening with cinema of capacity = 100
for (int i = 0; i < 101; i++) {
    screening.getSoldTickets().add(new Ticket());
}
```

Fix by returning a copy:

```java
public Set<Ticket> getSoldTickets() {
    return new HashSet<>(tickets);
}
```

(The solutions sheet writes this fix as `public Ticket getSoldTickets() { return new ArrayList<>(tickets); }` — a typo on two counts: the field is a `Set<Ticket>`, and the return type is `Set<Ticket>`, not `Ticket`. The intent is a defensive copy of the same collection type.)

The pattern to take away: **an invariant is only as strong as the widest way in to the state it constrains.** A precondition closes the front door; `private` closes the side door; a defensive copy closes the one left open by every getter that returns a mutable object.

Reference material

Java Encapsulation

Introduced in 2026-03-05-object-oriented-programming-i[21] (Lecture, Week 2).

Classes and objects

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

Encapsulation

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

The Java that box stands for:

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

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

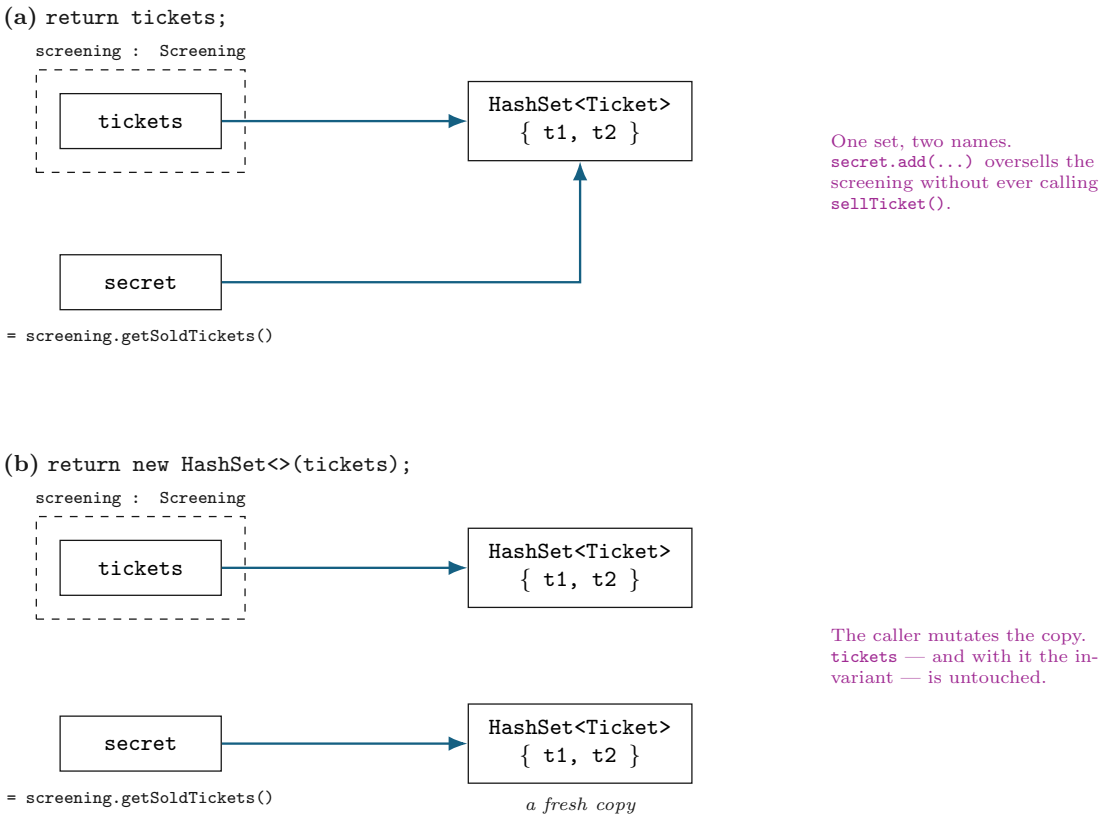


Figure 2: Why returning the internal collection breaks an invariant that the class otherwise enforces. (a) `return tickets;` — the caller's `secret` and the object's own field name **one** set, so `secret.add(...)` writes straight into `Screening`'s state, past every check `sellTicket()` makes. (b) `return new HashSet<>(tickets);` — the caller gets its own set to mutate. Dashed box = the object's encapsulation boundary; teal arrow = a reference. **CSSE2002 establishes no notation for object/reference (boxes-and-pointers) diagrams** — object diagrams, stack-and-heap diagrams and has-a relationships are all left unsettled by the course's own materials — so this rendering is mine. Only the teal reference colour and the magenta verdict text follow the course's own conventions.

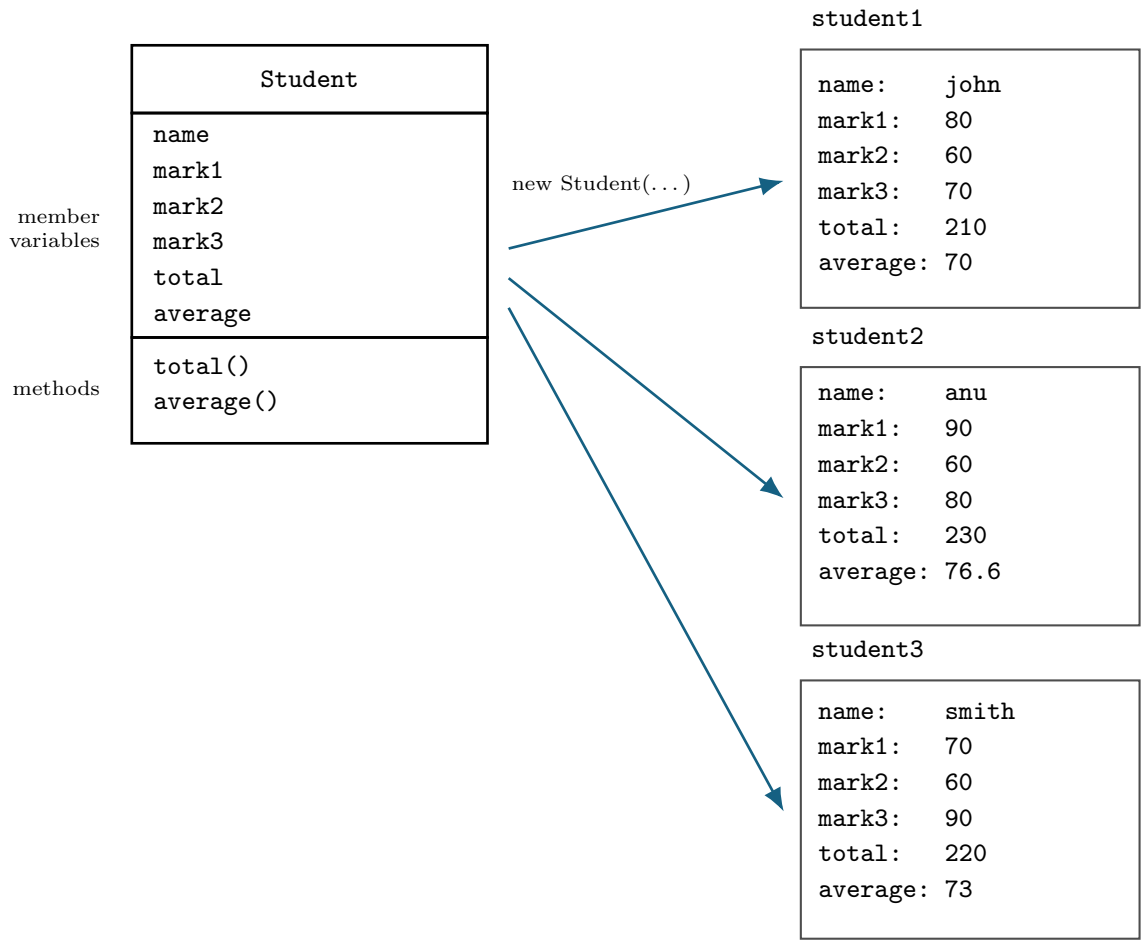


Figure 3: One Student class, three Student objects: the class fixes the member variables and methods, each object carries its own values for them. Each arrow is one new Student(…) call creating an object. anu's average is rounded to 76.6.

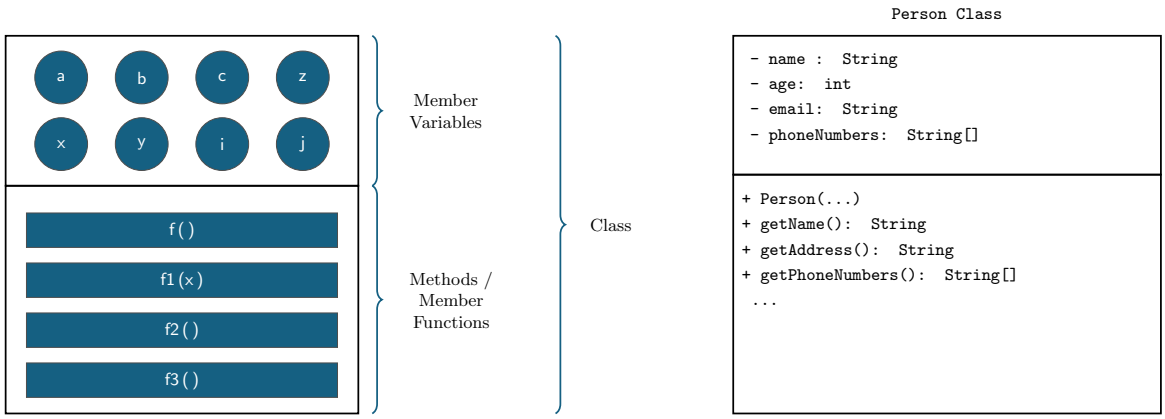


Figure 4: What a class is made of, and how it is written as a diagram. Left: a member-variable compartment above a methods compartment, the two braced together as the class. Right: the same idea as a two-compartment box, attributes above operations, with the class name above the box. The attribute list and the operation list in the box do not quite match (email against getAddress).

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

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

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

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

Constructors

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

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

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

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

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

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

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

Notes on constructors:

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

Access modifiers

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

Modifier	Accessible from
`private`	Only within the declaring class
`public`	Anywhere
`protected`	Same package, plus subclasses (see java-inheritance[22])
(default, no keyword)	Only within the same package

The same `BankAccount` under three of the four:

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

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

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

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

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

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

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

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

Getters and setters

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

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

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

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

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

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

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

The `static` keyword

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

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

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

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

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

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

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

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

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

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

class BankAccount {
    private String accountHolder;
    private double balance;

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

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

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

Class invariants

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

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

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

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

Invariants complement **preconditions** (what must be true before a method executes) and **postconditions** (what must be true after) — together these define a contract for how a method or function should behave. Pre/postconditions themselves are in java-specification[23] and java-hoare-logic[24]; see also week8-tutorial-class-invariants[25].

Protecting invariants

The `Counter` example does not show how an invariant can be broken from outside. Guarding against that is what makes the definition enforceable rather than merely stated; see also week8-tutorial-class-invariants[26] and the defensive-programming material in java-specification[27].

[23] courses/csse2002/java-specification.pdf
[24] courses/csse2002/java-hoare-logic.pdf
[25] courses/csse2002/week8-tutorial-class-invariants.pdf
[26] courses/csse2002/week8-tutorial-class-invariants.pdf
[27] courses/csse2002/java-specification.pdf

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

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

```
private List<String> files;

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

Preserving the invariant may also require adding **preconditions** to methods that could otherwise violate it (e.g. rejecting an addition that wouldn't satisfy the invariant), or, when a method is likely to be called by code outside your control, defensively throwing an exception (e.g. `IllegalArgumentException`/`IllegalStateException`) rather than relying purely on the precondition contract (see java-specification[28] — Defensive programming).

Method signatures

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

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

Worked question — is this allowed?

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

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

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

work in java-inheritance[29]: an override must repeat the superclass method's signature exactly, or it is a new overload rather than an override.

Java Exceptions

Introduced in 2026-03-19-exceptions[30] (Lecture, Week 4, Part 1). Traced in depth in week5-tutorial-exceptions[31].

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:

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

[29] courses/csse2002/java-inheritance.pdf
[30] courses/csse2002/2026-03-19-exceptions.pdf
[31] courses/csse2002/week5-tutorial-exceptions.pdf

Exception hierarchy

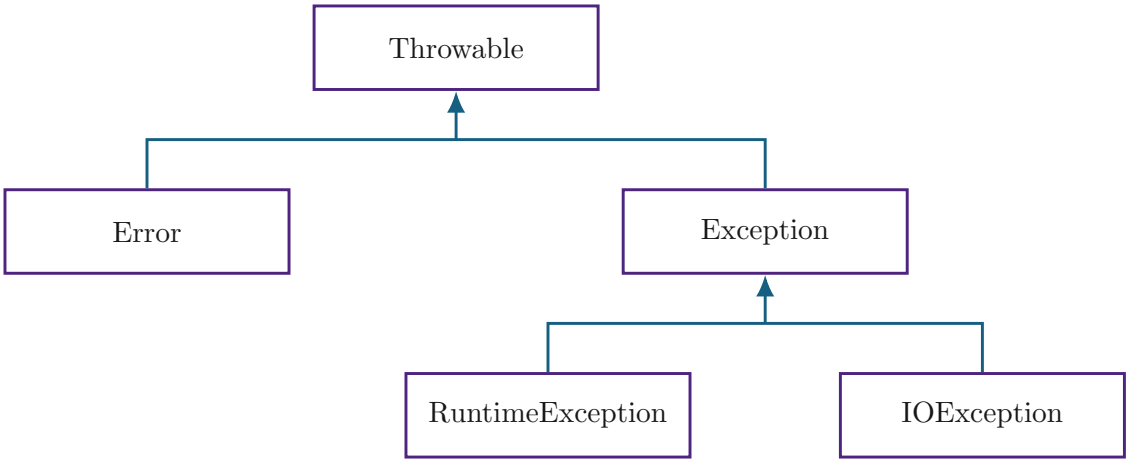


Figure 5: The exception hierarchy. Throwable is the root; Error and Exception sit beneath it; RuntimeException (unchecked) and IOException (checked) sit beneath Exception. Each connector joins the subclasses and points up to their parent. Which branch a class is on is what decides whether the compiler 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:

```
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");
    }
}
```

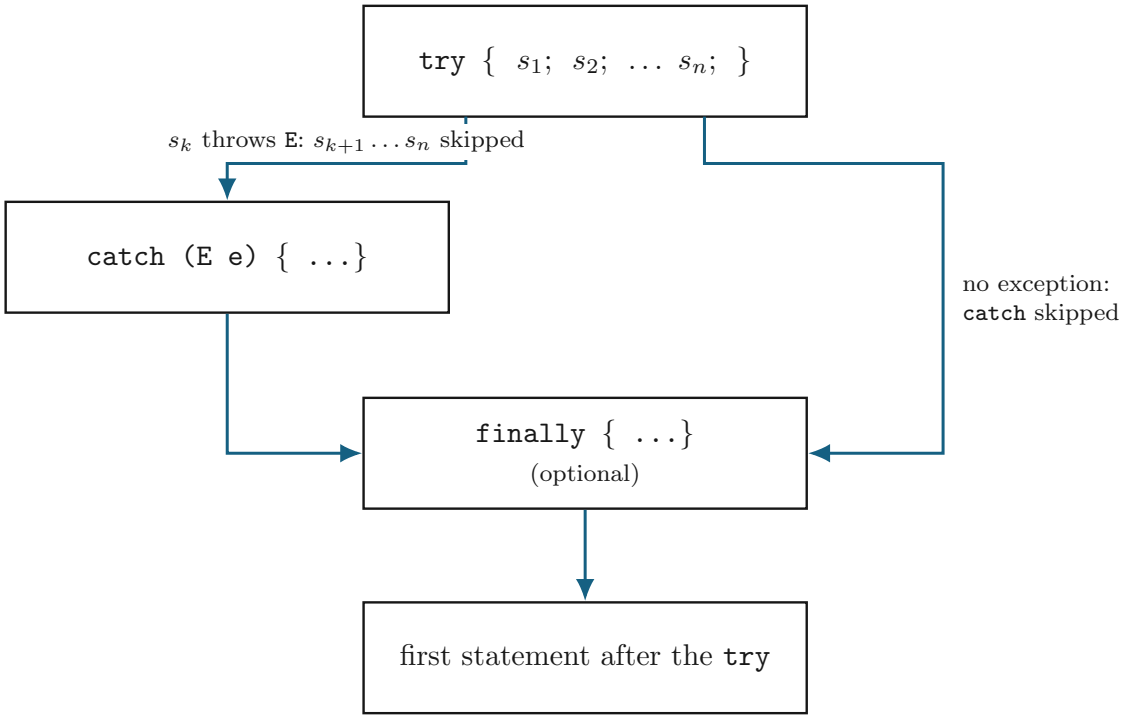


Figure 6: The two paths through a try/catch/finally, and the fact that both of them end up in finally. This is the three numbered rules together with the finally rule, drawn as one picture: the exceptional path abandons the remaining statements of the try and enters catch, the normal path skips catch entirely, and neither can skip finally.

```java
public class Main {
    public static void main(String[] args) {
        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[32]: 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);
        }
    }
    public static void findFile() throws FileNotFoundException {
        // code that may generate FileNotFoundException
        File newFile = new File("file.txt");
        FileInputStream stream = new FileInputStream(newFile);
    }
}
```

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

`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 I/O

Introduced in 2026-04-23-java-io[33] (Lecture, Week 8).

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

1. **Streams** (`java.io`, since 1.0) — byte-oriented.
2. **Readers & Writers** (`java.io`, since 1.1) — character-oriented, added because byte streams weren't ideal for text/Unicode.
3. **New I/O** (`java.nio.file`, since 1.7) — file-system operations (paths, directories, attributes), which `java.io` was never designed for.

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

[33] courses/csse2002/2026-04-23-java-io.pdf

Streams

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

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

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

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

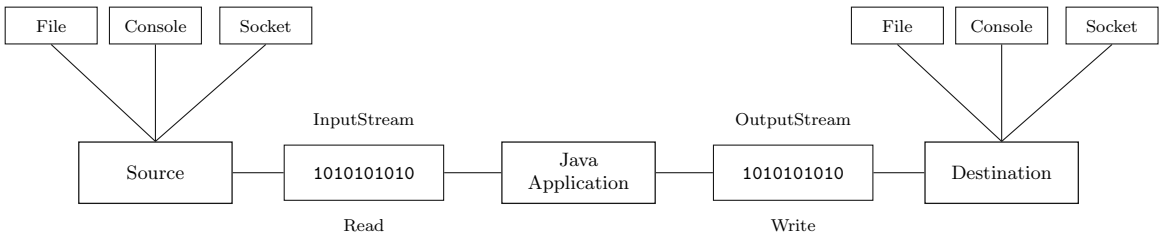


Figure 7: The stream abstraction: whatever the real source (file, console, socket), the application reads bytes through one `InputStream`, and writes them back out through one `OutputStream`, whatever the real destination.

Java has two families of streams:

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

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

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

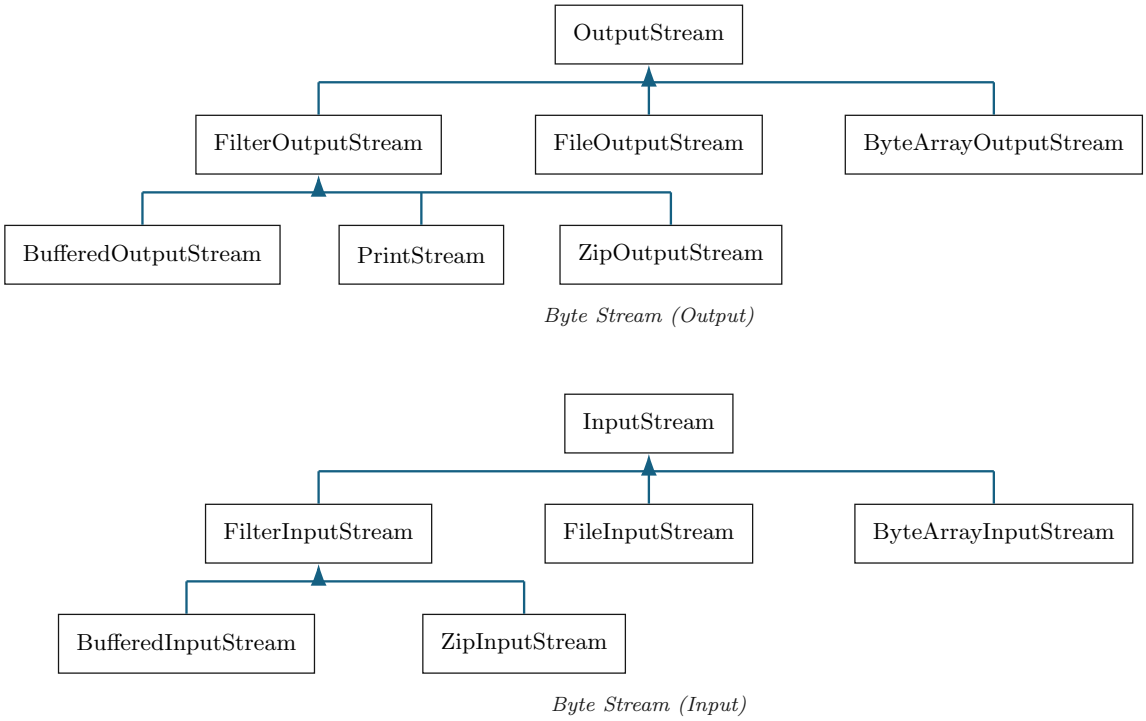


Figure 8: The byte-stream class hierarchies: `OutputStream` and `InputStream`. Each superclass sits above its subclasses, and the arrowhead points into the parent. The wrappers `BufferedInputStream` and `BufferedOutputStream` are `Filter*` streams — streams that wrap another stream.

The byte-stream hierarchy

`InputStream` and its subclasses represent a stream of bytes, drawn from different sources — `FileInputStream` gets bytes from a file, `ByteArrayInputStream` from an array, and so on. Methods that use streams should accept the **superclass** type as a parameter, so any concrete stream can be substituted (see java-solid-principles[34] — Liskov Substitution).

Worked example — writing to an abstract `OutputStream`

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

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

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

        System.out.println(fos);
    }

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

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

End of file

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

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

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

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

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

    private static void readAll(InputStream input) throws IOException {
        int in = 0;
        while (in != -1) {
            in = input.read();
            System.out.println(in);
        }
    }
}
```

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

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

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

Buffering

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

Only the construction changes; `readAll` is untouched:

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

(What each layer actually contributes is drawn in Figure 9.)

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

Closing streams

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

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

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

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

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

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

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

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

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

Wrapping streams together

Nothing in `java.io` is one class doing everything: you assemble the behaviour you want by nesting one stream inside another. That is what the hierarchy's whole `Filter*` branch is for, and why the constructor calls look the way they do. The second chain below uses `InputStreamReader` and `BufferedReader`, which the next section introduces — it is drawn here so both chains can be compared side by side.

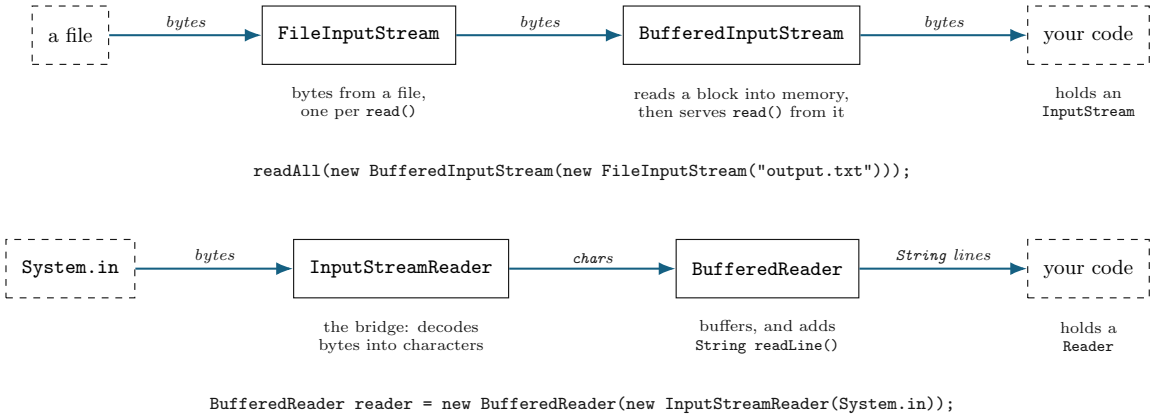


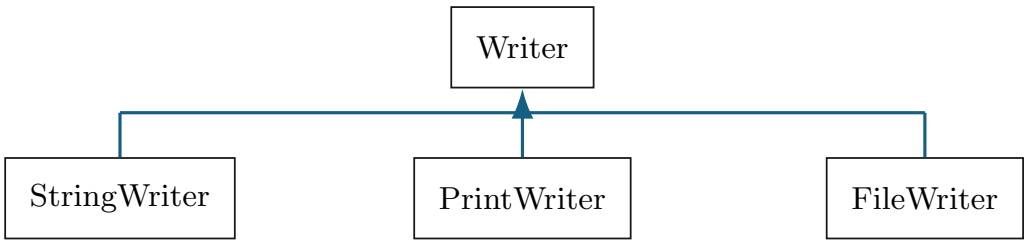
Figure 9: How the nesting in `new BufferedInputStream(new FileInputStream(...))` and `new BufferedReader(new InputStreamReader(System.in))` actually behaves — each wrapper adds one capability and passes the rest through, and your code only ever holds the outermost object. `BufferedInputStream` wraps another input stream; `InputStreamReader` acts as a bridge decoding bytes into characters; `BufferedReader` wraps another `Reader` and adds `readLine()`. The gloss beneath each box says what that stage contributes.

Readers & Writers

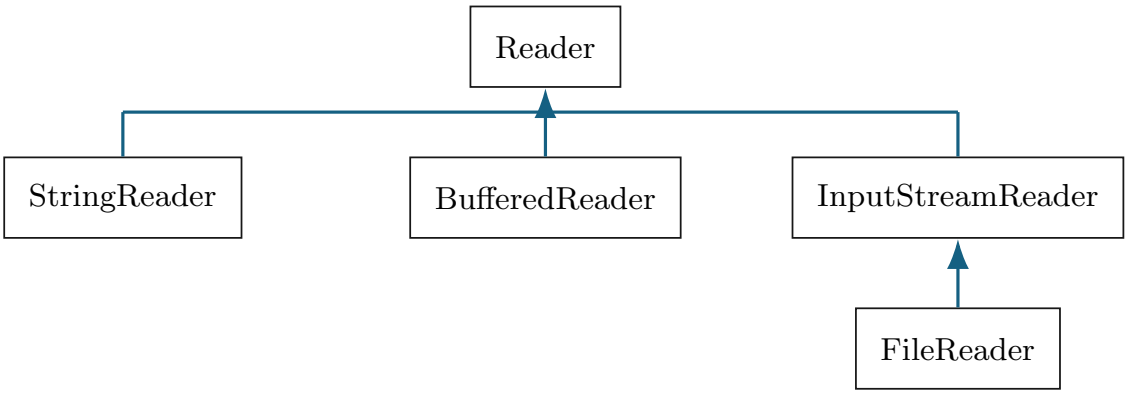
Originally Java I/O was byte-oriented. Byte streams were fine for raw binary data but not ideal for text, so character streams were added in 1.1 to handle text and Unicode characters more conveniently and efficiently.

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

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



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



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

Figure 10: The character-stream hierarchies: `Writer` and `Reader`, drawn the same way as Figure 8. The one two-level branch is the important one — `FileReader` is a kind of `InputStreamReader`, i.e. reading a text file is itself a case of decoding bytes into characters.

Worked example — read the first 10 characters of input

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

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

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

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

`BufferedReader`

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

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

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

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

The exception handling has to be made explicit: `FileReader`'s constructor throws `FileNotFoundException`, and `readLine()` throws `IOException`, so each needs handling.

Below, the second is wrapped in an unchecked `RuntimeException` rather than being declared. See java-exceptions[35].

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

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

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

`PrintWriter`

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

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

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

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

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

`writeroutput.txt` then holds:

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

`flush()`

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

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

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

`Scanner`

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

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

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

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

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

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

Reading strings:

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

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

Further reading: Scanner vs. BufferedReader.

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

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

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

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

Creating a `Path`

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

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

`Files` operations

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

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

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

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

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

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

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

Summary: which to use?

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

Saving and loading a whole object model to a file is picked up later, in week10-lab-java-io[36].

Java JUnit

Introduced in 2026-03-26-testing[37] (Lecture, Week 5). See java-testing[38] for the broader testing concepts JUnit is used to implement.

What is JUnit?

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

Writing tests

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

The full class and its test class:

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

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

    public String getName() {
```

[36] courses/csse2002/week10-lab-java-io.pdf
[37] courses/csse2002/2026-03-26-testing.pdf
[38] courses/csse2002/java-testing.pdf

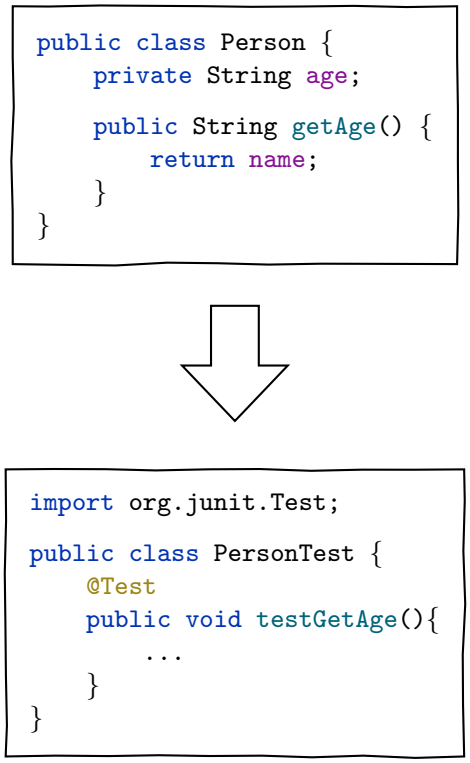


Figure 11: One test class per real class: `Person` produces `PersonTest`, which imports `org.junit.Test` and carries one `@Test`-annotated method per thing being checked. The hollow block arrow pointing down means produces/translates-to. The `Person` snippet does not compile as drawn: `getAge()` returns `name`, which the snippet never declares, and the field is typed `String age`. The full `Person` below types it `int` and is the version that compiles.

```
        return name;
    }

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

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

public class PersonTest {

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

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

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

`@Before` and `@After`

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

```
public class PersonTest {

    private Person person;

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

```
    }

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

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

`Assert`

Some static methods in `Assert`:

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

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

Checking for exceptions

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

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

`.class` is required after the name of the class. The test **passes** if that exception is thrown and **fails** if it is not, or if a different one is. See java-exceptions[39].

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

Black box tests in JUnit

Write the tests purely from the specification/Javadoc, without looking at the implementation, then use smoke tests, boundary tests and equivalence classes (see java-testing[40]) to decide what to check. The hard constraint is that **tests must not violate preconditions**. A test that breaks `@requires` tells you nothing, because the method is entitled to do anything at all in that case.

White box tests in JUnit

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

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

```
public class DiscountCalculator {

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

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

The five tests, one per leaf:

```
public class DiscountCalculatorTest {

    DiscountCalculator calc = new DiscountCalculator();
```

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

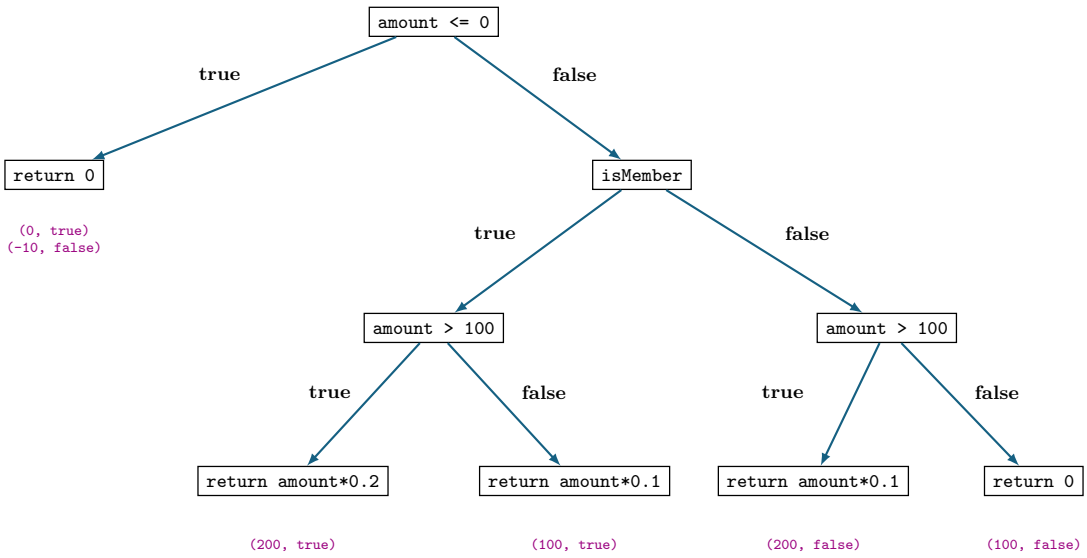


Figure 12: Why `DiscountCalculator` needs exactly five tests: the method as a decision tree, one leaf per test. Each box is a condition from the method or the `return` its branch reaches, so the expected values in the tests can be read off the leaves. The magenta `(amount, isMember)` pair under each leaf is the argument list of the test that reaches that leaf. The early return on `amount <= 0` is the one leaf covered by two assertions in a single test, which is why it carries two pairs.

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

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

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

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

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

Two things worth reading off this example:

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

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

Java Loop Invariants

Introduced in 2026-05-21-correct-programming[41] (Lecture, Week 12). Extends java-hoare-logic[42]'s backward-reasoning technique to loops, where the number of iterations isn't known up front.

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

What is an invariant?

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

Motivating puzzle

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

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

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

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

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

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

Loop invariants

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

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

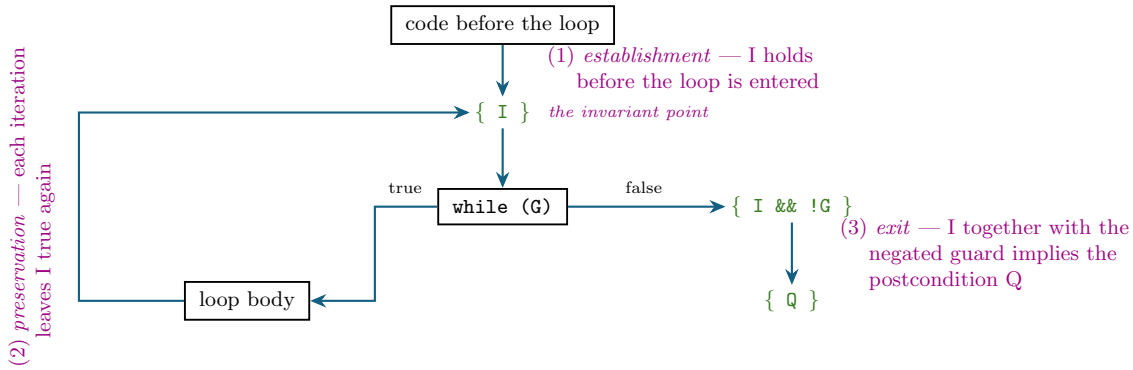


Figure 13: The three obligations in the definition above, each pinned to the point in the loop where it has to be discharged. The invariant point is the top of the loop, before the guard is tested: establishment is the arrow into it, preservation is the arrow back to it, and exit is what the guard's failure adds to it. Connectors are teal and assertions are green.

None of the three clauses in Figure 13 says anything about the loop *finishing*. An invariant that survives all three checks establishes only "*if* the loop ends, the postcondition holds"; termination is a separate argument — in `power` below, that `exp - i` strictly decreases and is bounded below.

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

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

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

Deriving an invariant for `power`

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

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

Step 2 — verify. Three checks:

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

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

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

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

The precondition comes from obligation (1) of Figure 13, failing, and being read backwards. `exp >= 0` was never designed in — it is whatever the establishment check needed in order to succeed.

Designing an algorithm *from* an invariant: fast exponentiation

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

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

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

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

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

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

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

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

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

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

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

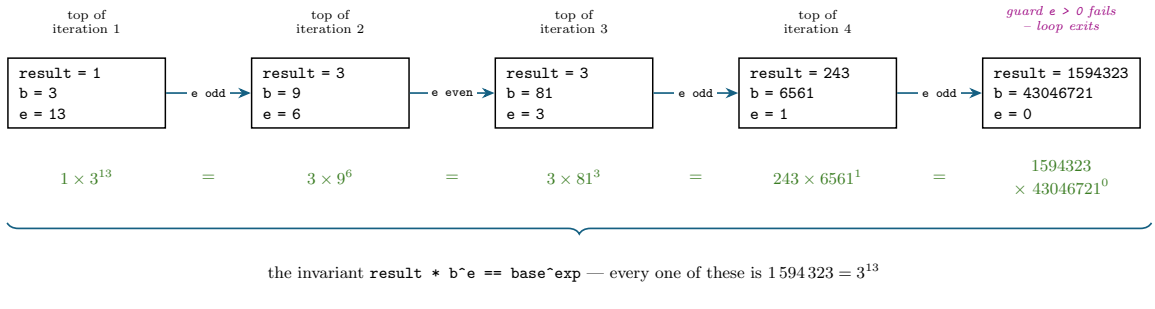


Figure 14: The fast-`power` loop traced on `power(3, 13)`, one box per arrival at the top of the loop. The three variables change on every step; the product `result * b^e` underneath does not, and the five products it takes are exactly the chain the section above works by hand (the section's `3 x 81 x 6561` and this trace's `243 x 6561` are the same number). Connectors are teal and the invariant is green.

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

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

Invariants in the Week 12 exercises

Neither Week 12 exercise set is *about* correctness — the applied class is lambdas and streams (week12-tutorial-lambdas-and-streams[43]) and the practical is generics (week12-lab-generics[44]) — but both contain loops whose behaviour is only explicable in these terms, and the practical writes contracts for two of them.

The loop bound that must not move

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

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

The contract this loop has to establish is its `@ensure` (given in full in java-hoare-logic[45]), whose first clause is `students.getCurrentCount() == 0`. Taking the invariant at the top of the iteration with counter `i` to be

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

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

the three obligations of Figure 13 go through: at `i == 0` nothing has moved and `getCurrentCount()` is still `numStudents`; each iteration takes exactly one passenger off and puts exactly one on before `i++`, restoring it; and exiting with `i == numStudents` gives `students.getCurrentCount() == 0` together with all `numStudents` students on `bus`.

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

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

[43] courses/csse2002/week12-tutorial-lambdas-and-streams.pdf
[44] courses/csse2002/week12-lab-generics.pdf
[45] courses/csse2002/java-hoare-logic.pdf

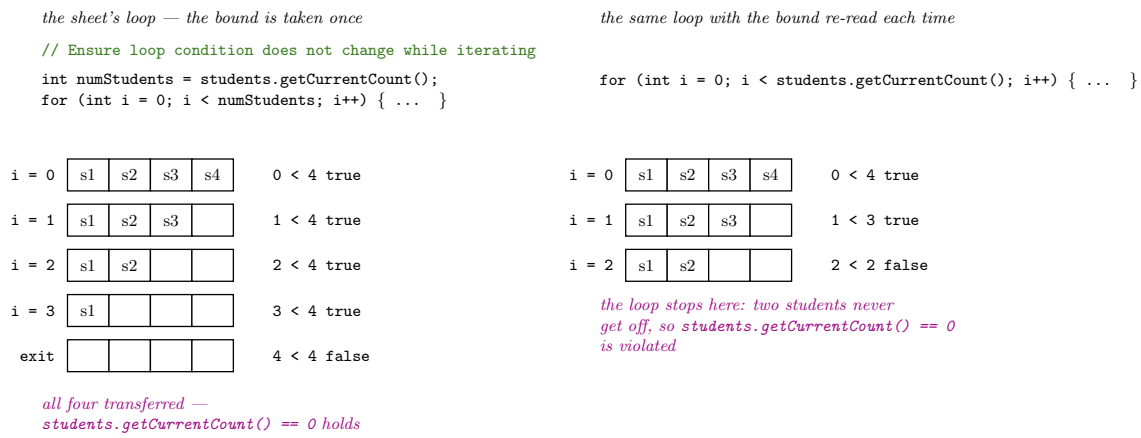


Figure 15: Why the practical snapshots its loop bound. Both columns run the same body on a source bus of four; the left column is the loop with the bound read once into `numStudents`, the right is the same loop with `students.getCurrentCount()` in the guard. Reading the count live makes the guard chase a quantity the body is decreasing, and the two meet after two iterations. Bus contents are drawn as array rows and the verdict on each column is in magenta.

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

An accumulator loop inside a lambda

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

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

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

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

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

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

The same idea, written into a method signature

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

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

`reduce`'s first argument is the identity and its second is the step. Read as a loop, those are precisely obligations (1) and (2): the identity `0` establishes the invariant "the accumulator holds the sum of the elements consumed so far", and `(a, b) -> a + b` is the step that preserves it. When the stream runs out — the negated guard — the accumulator holds the sum of all five elements, $1 + 2 + 4 + 8 + 16 = 31$. Nothing about the reasoning has changed; the fold's signature just makes you write the invariant's two halves down where a `for` loop would have let you leave them implicit. See java-lambdas-and-streams[46] for `reduce` itself.

Pragmatic considerations

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

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

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

A loop invariant is not the only kind of invariant in this course: a **class invariant** is the same idea applied to an object's fields across every public method rather than to a loop's variables across every iteration — established by the constructor, preserved by each method, relied on by all of them. See week8-tutorial-class-invariants[47].

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

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

Java SOLID Principles

Introduced in 2026-04-02-refactoring[48] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[49] (Lecture, Week 7 — Liskov Substitution, Interface Segregation, Dependency Inversion).

Why SOLID?

Large software tends to become:

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

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

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

Single Responsibility Principle (SRP)

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

Closely related to java-cohesion-and-coupling[50]'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.

[48] courses/csse2002/2026-04-02-refactoring.pdf
[49] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[50] courses/csse2002/java-cohesion-and-coupling.pdf

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

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

This class has three reasons to change:

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

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

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

Two details of the split are worth noticing. The `gpa` field is gone: it was a *cached result* of one of the three responsibilities, so it leaves with the responsibility, and `GPACalculator.calculateGPA` returns a value instead of assigning one. And `Student` gains `getCourses()`, because the two 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[51]) and garbage collection.

Open-Closed Principle (OCP)

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

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

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

A class is kept closed for modification by information hiding (see java-encapsulation[52]).

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:

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

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

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

Three things carry the problem: the added `String gradeType` parameter, and the two string literals `"Letter"` and `"Numeric"`. They are every place that has to be *modified* the next time a grading system is added, in a class that was supposed to be finished. The class is open for extension only by editing it, which is exactly what "closed for modification" forbids. (A `gradeType` string also reintroduces the `Vehicle`-style type switch from java-refactoring[53].)

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;
```

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

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

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

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

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

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

What the refactoring bought:

- `GPACalculator` depends on the abstraction `GradingSystem` instead of calculating grade points itself — which makes it open for extension (new grading systems) but closed for modification.
- More grading systems can be added without changing the core logic of `GPACalculator`.
- 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.

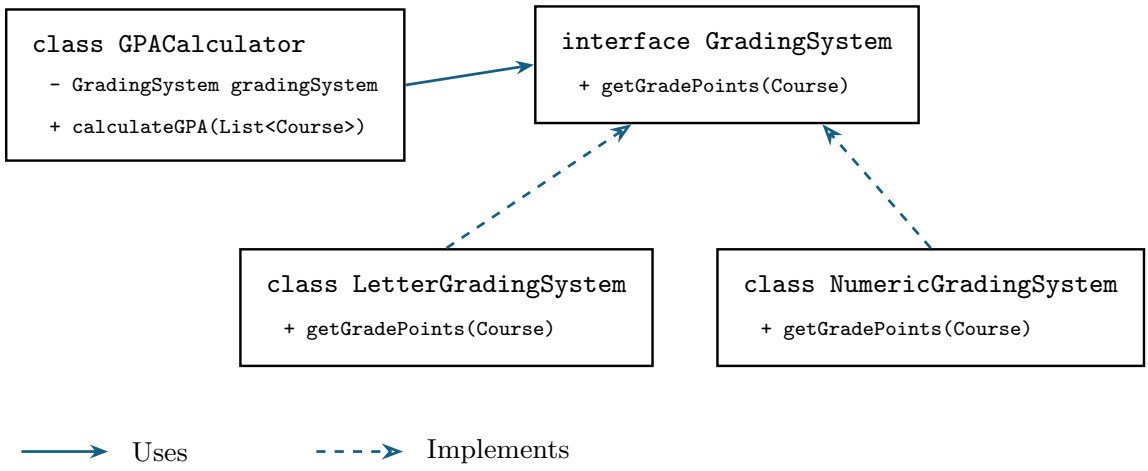


Figure 16: The open-closed refactoring of GPACalculator. GPACalculator uses only the GradingSystem interface; each concrete grading system implements it. A new scheme — pass/fail, say — is a fourth box on the bottom row with a fourth dashed arrow, and no existing box changes: that is 'open for extension, 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.

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[54]):

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

```
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
     */
    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[55]'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.
```

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

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).
- **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[56]):

```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); }
```

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

```
}
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[57] (Lecture, Week 3). Extended in 2026-03-19-exceptions[58] and formalised later in java-hoare-logic[59]; pre/postconditions come

[57] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[58] courses/csse2002/2026-03-19-exceptions.pdf
[59] courses/csse2002/java-hoare-logic.pdf

back as class invariants in week8-tutorial-class-invariants[60] and as the substitution principle in week9-tutorial-substitution-principle-and-pre-post-conditions[61].

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
 * @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:

[60] courses/csse2002/week8-tutorial-class-invariants.pdf
[61] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf

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**.

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) {…}
```

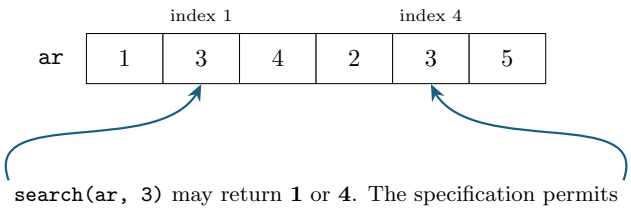


Figure 17: With the spec worded as 'returns an index i such that ar[i] == x', both cells holding 3 are legal answers, and nothing forbids the method returning a different one on each call. That is what makes the specification non-deterministic.

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() {...}
```

```
/**
 * 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[62]. 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`?

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

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:

```
/**
 * 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:

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

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

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

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

Contracts

Specifications can be written using contracts:

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

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

Programming *with* contracts

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

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

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

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

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

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

Programming *without* contracts

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

```java
/**
 * 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;
    }
}
```

```java
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.

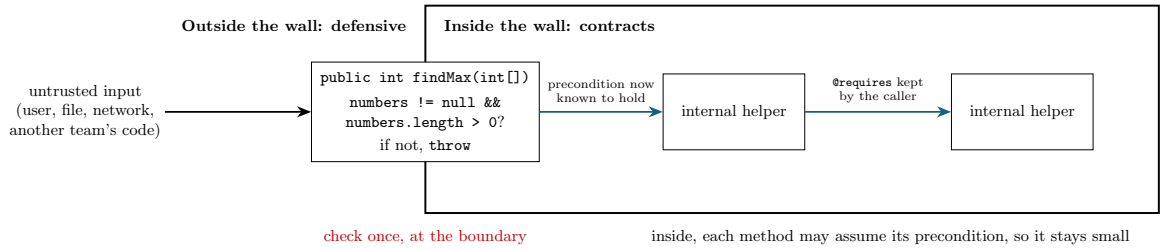


Figure 18: Defensive outside the wall, contracts inside it: validate defensively where untrusted input crosses into the system, and rely on contracts for the internal calls behind that boundary, so the check happens once rather than at every hop. The gate box is the findMax method worked through below.

Worked example: `findMax`

The starting point, with no specification and no checks:

```
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:

```
/**
 * Finds the maximum value in the given array of integers.
 *
 * @param numbers An array of integers from which the maximum value is to be found.
 * @return The maximum value in the array.
 * @throws IllegalArgumentException if the array is null or empty.
 *
 * @requires numbers != null && numbers.length > 0
 *    - The array must not be null.
 *    - The array must contain at least one element.
 *
 * @ensures
 *    - The method returns the maximum value found in the array.
 *
 */
public int findMax(int[] numbers) {
```

```
    if (numbers == null || numbers.length == 0) {
        throw new IllegalArgumentException("Array must not be null or empty.");
    }

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

This version writes the precondition down as `@requires`, promises an exception in `@throws` for callers who break it anyway, and then *checks it in the body* — belt and braces, which is the whole point of being defensive at a boundary. Without the check, a `null` array throws `NullPointerException` and an empty array silently returns `Integer.MIN_VALUE` as if that were a real maximum.

> The loop bound differs between the two versions
>
> The first version loops `i < numbers.length-1` and the documented version loops `i < numbers.length`. The first version therefore never examines the last element, so `findMax(new int[]{1, 2, 9})` returns 2. No specification was attached to the first version, so nothing about it was wrong *according to its spec*, because it did not have one.

Further reading

- Preconditions, Postconditions, and Class Invariants
- Assertions

Key takeaways

- A specification describes the **contract between a method and its client**.
- It explains **what** the method does, not **how** it is implemented.
- A good specification clearly states:
  - **Preconditions** — conditions that must be true before the method is called;
  - **Postconditions** — conditions that will be true after the method finishes successfully;
  - **Side effects** — any additional changes the method makes to the system.
- Clear specifications make code easier to understand, use, and maintain.

Java Testing

Introduced in 2026-03-26-testing[63] (Lecture, Week 5).

Levels of testing

1. **Unit testing** — check that each "unit" in the project behaves correctly.
2. **Integration testing** — check that components work together and that the interfaces between components work as expected.
3. **System testing** — does the system as a whole work correctly?
4. **(User) acceptance testing** — do users of the system agree that the system does what it's supposed to do?

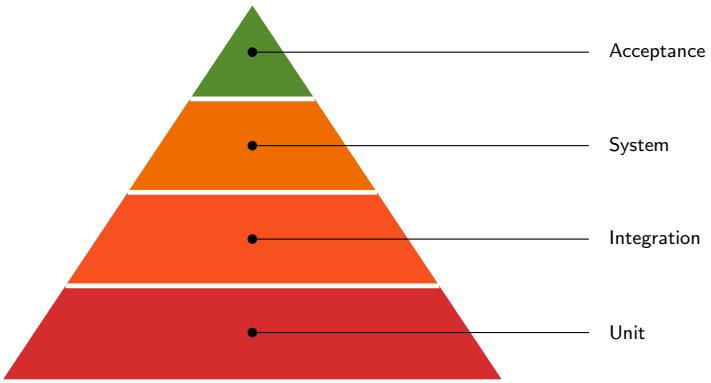


Figure 19: The four levels of testing as a pyramid. Unit tests form the wide base, acceptance tests the narrow apex. In the usual test-pyramid reading, the width of a band stands for how many tests of that kind a project has.

Test frameworks

Libraries that make writing and running tests easier. **xUnit** is the prevalent family — JUnit, NUnit, CPPUnit, PyUnit, … — and this course uses **JUnit 4** (http://junit.org/junit4); see java-junit[64].

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

[63] courses/csse2002/2026-03-26-testing.pdf
[64] courses/csse2002/java-junit.pdf

Regression testing

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

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

Regression testing ensures that old features are still working as they should when new features are introduced.

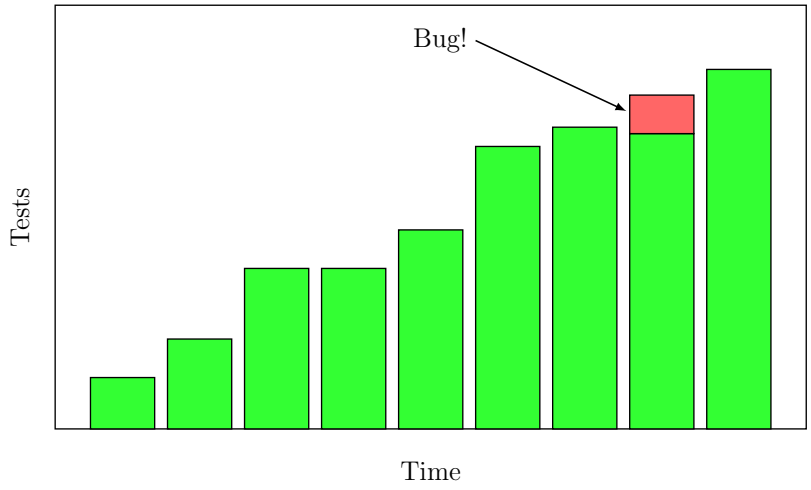


Figure 20: A regression (chart after browserstack.com/guide/regression-testing). Nine bars on *Tests*/*Time* axes: read each bar as a build, and its height as the size of the test suite growing as features are added. The red top slice on the second-to-last bar is the regression, and the arrow labelled Bug! points at it.

Black box vs white/glass box testing

- **Black box** — the software has inputs and outputs to test, but the implementation is unknown to you; you test according to the *specification* (see java-specification[65]) — what it's supposed to do.
- **White/glass box** — testing designed *with* knowledge of the internal implementation, so tests can pay special attention to cases where the implementation is complicated.

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

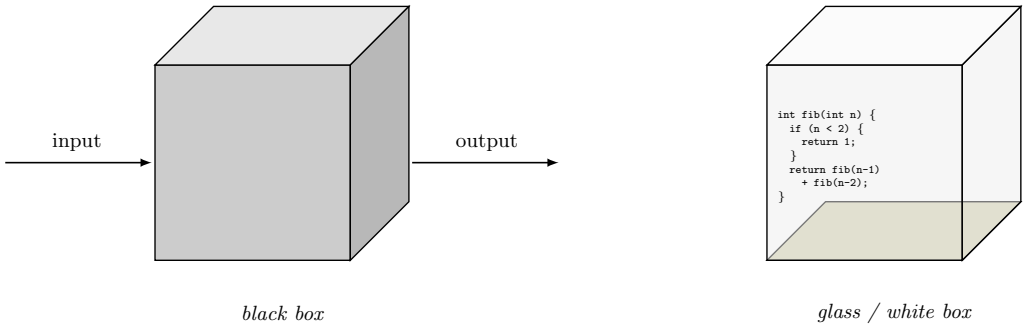


Figure 21: The two testing stances. Both are the same grey cube. The black box is opaque and shows only its input and output arrows. The glass box is translucent, so the implementation inside, a recursive `fib`, can be seen through its walls.

Black box techniques

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

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

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

> **Note**
>
> The tag is written `@returns` here and `@return` in the specification tag table from the exceptions lecture. The course uses both spellings — see csse2002-diagram-notation[66], "Specification tag spelling".

Smoke tests — test the common-case functionality. Usually the *first* test to be performed ("can it run?"), used to determine whether further, more rigorous testing is required. The

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

name comes from electronic hardware testing: plug in a new board, turn on the power, and if you see smoke coming from the board, turn the power off — you don't have to do any more testing. (From *Lessons Learned in Software Testing.*)

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

Input type	Try
Whole number	0, -1, minimum value, maximum value
Floating point	0, -1, `NaN`, infinity
Collection (array, list, set, etc.)	empty, one element, large
Reference type	`null`
Resource (file, link)	non-existent resource

Equivalence classes — groups of inputs that the system treats the same way. For `getCurrentPassengers()` on a bus whose maximum capacity is 80, build the table in two passes: first the classes, then the boundary tests for each class.

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

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

The black-box method

The method in six points:

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

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

White box technique: code coverage

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

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

A worked example:

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

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

Written out, the four paths and their inputs are:

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

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

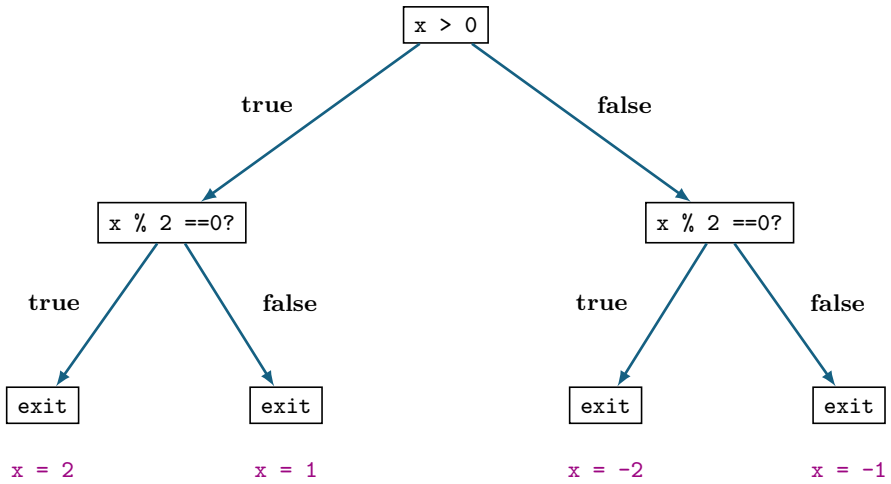


Figure 22: Path coverage for `register`. Each internal box is a condition, each teal arrow an outcome labelled true/false, each leaf one complete path to `exit` — four leaves, so four tests. The magenta input under each leaf is a value of `x` that drives that path.

Loops make exhaustive path coverage infeasible

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

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

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

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

Using JUnit for white-box testing

Design tests based on internal logic. You try to cover **branches**, **loops** and **edge cases**, and you may aim for a code coverage target (e.g. 100%). A worked white-box example — `DiscountCalculator`, the five tests that cover it, and its branch tree — is in java-junit[67].

Test Driven Development (TDD)

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

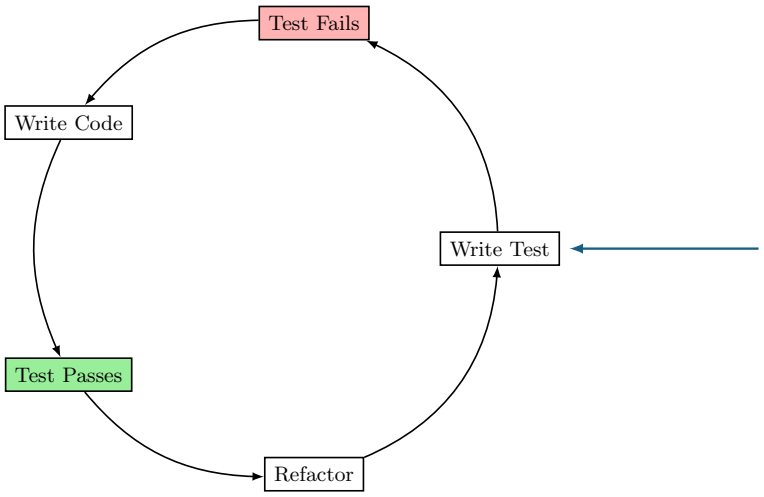


Figure 23: The TDD cycle. Read it counter-clockwise: write a test, watch it fail (pink), write code, watch it pass (green), refactor, and go round again. The teal arrow on the right points at *Write Test* and marks the entry point of the loop: the test comes before the code.

The *Refactor* step connects to java-refactoring[68]: the green test suite is what makes it safe to restructure code without changing its behaviour.

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