CSSE2002 — Week 8 Notes

Table of contents

Java I/O

See csse2002[1] for course logistics.

[1] courses/csse2002/csse2002.pdf

Today's outline

1. Streams (`java.io`, since Java 1.0)
2. Readers & Writers (`java.io`, since Java 1.1)
3. Scanner (`java.util`, since Java 1.5)
4. New I/O (`java.nio.file`, since Java 1.7)

All content is in java-io[2].

Applied class

See week8-tutorial-class-invariants[3] — protecting class invariants (a continuation of java-specification[4] and java-encapsulation[5] from earlier weeks).

Practical

See week8-lab-junit-testing[6] — Test Driven Development with JUnit, extending java-junit[7] from Week 5.

JUnit Testing

Practical for 2026-04-23-java-io[8] (Week 8). Extends java-junit[9] (Week 5) with a full Test Driven Development (TDD) worked example.

Test Driven Development (TDD)

Cycle (see java-testing[10]):

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.

[2] courses/csse2002/java-io.pdf
[3] courses/csse2002/week8-tutorial-class-invariants.pdf
[4] courses/csse2002/java-specification.pdf
[5] courses/csse2002/java-encapsulation.pdf
[6] courses/csse2002/week8-lab-junit-testing.pdf
[7] courses/csse2002/java-junit.pdf
[8] courses/csse2002/2026-04-23-java-io.pdf
[9] courses/csse2002/java-junit.pdf
[10] courses/csse2002/java-testing.pdf

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 (e.g. `0`), reference types normally return `null`. This lets you write an outline of a class that compiles but doesn't yet work — in pure TDD, not compiling counts as a test failure, so a stub is often the very first thing written to make a not-yet-existing class's test compile.

`DistinctCounter`

Tracks a collection of distinct strings, retrievable 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"}
```

> **i** Working
>
> **Stub**:
>
> ```
> class DistinctCounter {
>     public DistinctCounter() {}
>     void add(String element) {}
>     int getDistinctCount() { return 0; }
>     String[] getStrings() { return null; }
> }
> ```
>
> **JUnit 4 test class skeleton**:
>
> ```
> import org.junit.Test;
> import static org.junit.Assert.*;
>
> class DistinctCounterTest {
>     @Test
>     public void testEmpty() {}
> }
> ```

Implementation (one of several equally valid designs — a `HashSet` naturally rejects duplicates, so `getStrings()` just needs to sort on the way out):

```
public class DistinctCounterHashSet implements DistinctCounter {
    private final Set<String> distinct = new HashSet<>();

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

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

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

(Other equally valid implementations: a `TreeSet`, which keeps elements sorted automatically without an explicit `Arrays.sort`; or an `ArrayList`-backed version that checks `contains()` before adding, sorting either on every `getStrings()` call or by inserting in sorted position on every `add()`.)

Representative tests (`@Before` constructs a fresh `counter` before each test, avoiding duplicated setup code in every method):

```java
public class DistinctCounterTest {
    private DistinctCounter counter;

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

    @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 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());
    }
}
```

The full test suite (not reproduced in full here) mirrors this pattern across every case worth naming: empty / one element / two distinct / two identical / two identical + one other / N distinct (for both `getDistinctCount()` and `getStrings()`, plus that the returned array is sorted). TDD like this tends to produce a very thorough test suite, but one that mostly targets "happy path" execution — it's still worth adding boundary cases (see java-testing[11]) on top, e.g.:

```java
@Test
public void testNullStringCount() {
    counter.add(null);
    assertEquals(0, counter.getDistinctCount());
}
```

11 courses/csse2002/java-testing.pdf

`PalindromeCounter`

Extends `DistinctCounter` with:

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

Working

```
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() {
        List<String> distincts = new ArrayList<>(Arrays.asList(getStrings()));
        ↪  // wrap to allow removeAll
        distincts.removeAll(Arrays.asList(getPalindromes()));
        return distincts.toArray(new String[]{});
    }
}
```

Representative test (built up incrementally via TDD, one case at a time — empty, single palindrome, single non-palindrome, one of each, then several of each):

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

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

    @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());
    }
}
```

Note `isPalindrome` treats strings shorter than 2 characters as palindromes by definition (the base case), and is case-sensitive (`"AbbA"` is a palindrome as written — comparing first/last characters directly, not after case-folding).

Class Invariants

Applied class for 2026-04-23-java-io[12] (Week 8). Applies java-encapsulation[13]'s class invariants (and the preconditions/postconditions from java-specification[14]) to worked examples.

`XFiles`

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

[12] courses/csse2002/2026-04-23-java-io.pdf
[13] courses/csse2002/java-encapsulation.pdf
[14] courses/csse2002/java-specification.pdf

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

```
/**
 * @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) {...}

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

As specified, `add` allows the invariant to be broken (nothing stops a non-`'X'` string being added). Fix: add a precondition

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

Even with that precondition documented, a naive implementation still has two further leaks (see java-encapsulation[15] — 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
```

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

```
    }
}
```

1. **`files` is public** — callers can grab the list directly and mutate it, bypassing `add`'s check entirely: `xFiles.files.add("Doesn't start with X!")`. Fix: make it `private`.
2. **`getFiles()` returns the internal reference** — even with `files` made private, the *returned* list is still the real one: `xFiles.getFiles().add("Doesn't start with X!")` still breaks the invariant. Fix: return a defensive copy, `return new ArrayList<>(files);`.

`Cinema` and `Screening`

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

> **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) {...}
}
```

(Timestamps are unix time; `946648800` = 1st January 2000, when the cinema opened.)

Working

Do the methods preserve the invariant? No — a negative `amount` passed to `extendRuntime` could push `getEndTime()` below (or equal to) `getStartTime()`. Two sensible fixes:

a. `amount >= 0`
b. `getEndTime() + amount > getStartTime()`

The method's name (**extend**Runtime) implies (a) is the more likely intended precondition.

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 prevent overselling:

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

Precondition to preserve it — add to `sellTicket()`: `getSoldTickets().size() < getCinema().getCapacity()`. (Technically, since the invariant can be *assumed* as a precondition too, `getSoldTickets().size() != getCinema().getCapacity()` is enough — the invariant plus that inequality 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. In pure programming-by-contract, `sellTicket()` doesn't check the precondition at all — under a pure contract it's *allowed* to do anything if the caller violates the precondition, but that's fragile if `sellTicket()` is exposed to code you don't control. Defensive programming (see java-specification[16]) is safer here:

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

(Returning `null` instead of throwing protects the invariant too, but risks a hard-to-trace `NullPointerException` later in the caller — throwing immediately at the point of misuse is clearer.)

2. The *real* bug is that `getSoldTickets()` leaks the internal `tickets` reference — external code can oversell the screening directly: `screening.getSoldTickets().add(new Ticket())`, bypassing `sellTicket()` entirely. Fix with a defensive copy:

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

16 courses/csse2002/java-specification.pdf

Reference material

Java Encapsulation

Introduced in 2026-03-05-object-oriented-programming-i[17] (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.

```
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; }
}
```

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

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** at runtime.

Notes on constructors:

- Constructors are invoked implicitly when objects are instantiated.
- The constructor's name MUST be the same as the class.
- A constructor must not have a return type.
- A constructor can be overloaded but cannot 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[18])
(default, no keyword)	Only within the same package

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

Attempting to access a `private` field from outside its class is a compile error (`X has private access in Y`).

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; }
}
```

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.

Static variables (class variables) are shared among all instances of a class — useful for constants and shared properties, e.g. `public static int totalAccounts = 0;`.

Static methods can be called without creating an instance of the class, 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 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. 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
```

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 should behave.

Protecting invariants

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[19] — Defensive programming).

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

Method signatures

A method signature is a method's unique identifier: its name plus its parameter types. The return type and 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)
```

Java I/O

Introduced in 2026-04-23-java-io[20] (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.

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. We often don't want to rewrite our program just because the source or destination changed.

- 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.

Java has two families of streams:

- **Byte streams** (`InputStream`/`OutputStream`) — raw binary data (files, images, network data).
- **Character streams** (`Reader`/`Writer`, see below) — characters/text, handling Unicode.

  A byte stream reads data as bytes, whereas a character stream reads data as characters. (Schildt, *Java: The Complete Reference*, 11th ed.)

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

`InputStream` / `OutputStream`

`InputStream` and its subclasses represent a stream of bytes, drawn from different sources (`FileInputStream` from a file, `ByteArrayInputStream` from an array, …). Methods that use streams should accept the **superclass** type as a parameter, so any concrete stream can be substituted (see java-solid-principles[21] — Liskov Substitution):

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

End of file: all input streams need to consider "end of file" — `read()` returns `-1` at EOF, so a loop reading one byte at a time typically continues `while (in != -1)`.

Buffering

Reading a file a byte at a time is slow. `BufferedInputStream` wraps another input stream and buffers reads (the buffer is an area of main memory used to temporarily hold data) — on one test VM, reading a 4MB file took 82ms buffered vs. 18,193ms unbuffered:

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

Closing streams

Streams (and Readers) need closure — systems may limit how many files can be open at once, so always `close()` a stream when finished with it. Wrapping cleanup in `try/catch/finally` 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 {
```

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

```
    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 automatically closed (Effective Java, 3rd ed., Item 9: "Prefer try-with-resources to try-finally"):

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

Readers & Writers

`Reader`/`Writer` are the character-based counterparts of `InputStream`/`OutputStream`. `InputStreamReader` bridges the two: it wraps an `InputStream` (like `System.in`) and decodes its bytes into characters.

```
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);
}
// readAndPrint(new InputStreamReader(System.in));
// readAndPrint(new FileReader("myfile.txt"));
```

`BufferedReader`

Wraps another `Reader`; as well as buffering, it adds `String readLine()`:

```
BufferedReader reader = new BufferedReader(new FileReader("readwithbuffer.txt"));
for (int i = 0; i < 5; i++) {
    System.out.println(reader.readLine());
}
```

`PrintWriter`

`System.out` is actually a `PrintStream`; `PrintWriter` is a better option for character output — it can write to many destinations and supports formatted text (`printf`):

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

`flush()`

If an `OutputStream`/`Writer` is buffered, output might not be sent immediately — `flush()` forces any pending output out. This matters for interactive situations (the other end won't respond if nothing's actually been sent yet) and for debugging/logging (an up-to-date view of what's happening). `close()`-ing a stream flushes 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/strings (a friendlier alternative to `BufferedReader`). It also supports regex-based scanning for advanced use cases.

```
Scanner scanner = new Scanner(System.in); // internally wraps its own
↪  InputStreamReader
int total = 0;
while (scanner.hasNextInt()) {
    total += scanner.nextInt();
}
System.out.println(total);
```

Reading strings: `scanner.next()` reads the next word (skipping leading whitespace, stopping at whitespace); `scanner.nextLine()` reads the entire line up to Enter.

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

`java.io` is mainly stream-oriented — its goal was reading/writing data, not managing files or directories, so it's awkward for file attributes, directory traversal, and path manipulation. `java.nio.file` (since Java 1.7) adds two key abstractions:

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

Creating a `Path`

```
Path p1 = Path.of("docs/output.txt");         // whole path as one string
Path p2 = Path.of("docs", "output.txt");      // separate name elements, joined by
 ↪  Java
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.

```
Path myPath = Path.of("src", "Week08", "NIO", "myfile.txt");
List<String> lines = Files.readAllLines(myPath);
for (String line : lines) {
    System.out.println(line);
}
```

Summary: which to use?

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

Java JUnit

Introduced in 2026-03-26-testing[22] (Lecture, Week 5). See java-testing[23] 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. Depending on how software is structured, unit-testing frameworks like JUnit can also be used for other kinds of automated testing (the term "unit testing" is sometimes loosely used to mean *any* automated test).

Writing tests

Tests are defined in classes — conventionally 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.

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

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

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

[22] courses/csse2002/2026-03-26-testing.pdf
[23] courses/csse2002/java-testing.pdf

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

public class PersonTest {
    @Test
    public void testGetAge() {
        Person p1 = new Person("Jack", 10);
        assertEquals(10, p1.getAge());
    }

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

`@Before` and `@After`

Each test method should be independent, but common setup (e.g. object creation) can be factored out:

```java
public class PersonTest {
    private Person person;

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

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

`@Before` runs before **each** test; `@After` runs after **each** test.

`Assert`

Some static methods on `Assert`:

- `assertEquals` — asserts two objects are equal.

- `assertArrayEquals` — asserts two object arrays are equal.
- `assertFalse` / `assertTrue` — asserts a condition is false/true.
- `assertSame` / `assertNotSame` — asserts two objects do/don't refer to the same object.

Checking for exceptions

Don't catch the exception yourself — tell the test runner which exception type to expect via `@Test(expected = ...)`, and let it catch it:

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

Note the `.class` after the exception type is required.

Black box vs white box JUnit tests

- **Black box**: write tests purely from the specification/Javadoc, without looking at the implementation, then apply smoke tests / boundary tests / equivalence classes (see java-testing[24]) to decide what to check — tests must not violate the method's preconditions.
- **White box**: design tests *using* knowledge of the internal logic, aiming to cover branches, loops, and edge cases (e.g. targeting 100% branch coverage) — e.g. for a `calculateDiscount(amount, isMember)` method with nested `if`/`else` branches on both parameters, white-box tests are written to exercise every branch combination (amount 0; member above/at-or-below 100; non-member above/at-or-below 100).

Java Specification

Introduced in 2026-03-12-object-oriented-programming-ii[25] (Lecture, Week 3, Part 2).

Javadoc

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

[24] courses/csse2002/java-testing.pdf
[25] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf

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

Common tags:

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

What makes a good specification

Ideally, a specification should:

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

Restrictiveness — keep out incorrect implementations

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

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

Generality — allow acceptable alternative versions

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

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

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

Clarity

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

Formality

Specifications range from informal to formal:

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

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

Contracts

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

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

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

Defensive programming

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

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

```
/**
 * Finds the maximum value in the given array of integers.
 *
 * @throws IllegalArgumentException if the array is null or empty.
 * @requires numbers != null && numbers.length > 0
```

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

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

Java Testing

Introduced in 2026-03-26-testing[26] (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?

These form a pyramid, from many low-level unit tests at the base up to a few acceptance tests at the top.

Regression testing

During development, testing helps answer two questions: does the new stuff work, and have we broken things that used to work (**regressed**)? Regression testing ensures old features keep working as new features are introduced.

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

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[27]) — 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.

Black box techniques

Smoke tests — test the common-case functionality first ("can it run?"), to decide whether more rigorous testing is worthwhile. The name comes from electronics testing: plug in a board, turn on the power — if you see smoke, stop.

Boundary tests — investigate extremes and corner cases, since that's 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…)	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. E.g. for `getCurrentPassengers()` on a bus with capacity 80: `Valid-Empty` (0), `Valid-Normal` (1-79), `Valid-Full` (80), `Invalid` (anything else). The corresponding *boundary* tests probe each class's edges: `-1, 0, 1` (around empty), `79, 80, 81` (around full capacity) — giving a minimal boundary set of `-1, 0, 1, 79, 80, 81`.

White box technique: code coverage

Of all the ways a program could run, how many are covered by the tests? Three levels, from weakest to strongest:

1. **Statement coverage** — every statement is executed at least once.
2. **Branch coverage** — every branch is tested for both the `true` and `false` case.
3. **Path coverage** — every distinct path through the code is traversed.

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

Example:

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

- Statement coverage: `x = 2` alone covers every line (both `if` bodies run).
- Branch coverage: `x = 2` and `x = -1` together cover both branches of each `if` (true/false).
- Path coverage: needs all 4 combinations of the two conditions: `x=2` (true, true), `x=1` (true, false), `x=-2` (false, true), `x=-1` (false, false).

Loops make exhaustive path coverage infeasible — a loop like `for (int i = 0; i < 100; i++) { if (f(i, k)) { j++; } }` has 2^{100} paths (over a billion tests/second would still take ~40.2 trillion years, about 2900 times the age of the universe). Instead, path coverage for loops is **approximated**: treat 0, 1, and 2-or-more iterations (of a loop or recursive call) as equivalent ("engineers' induction: one, two, three — that's good enough for me").

Test Driven Development (TDD)

Originates from the Agile manifesto and Extreme Programming. Cycle: **write a (failing) test** → **write code** until the **test passes** → **refactor** → write the next test. Developers write small test cases for every feature based on their initial understanding, and only modify/write new code when a test fails (avoiding duplicated test scripts) — the tests determine when code is "ready".