CSSE2002 — Week 1 Notes

Table of contents

Course Overview and Java Basics **2**
- Outline . . . 2
- Programming in the large . . . 2
- Java overview . . . 3
- Data types in Java . . . 5
- Scope . . . 5
- Control flow: sequence, selection, and iteration . . . 6
- Java arithmetic . . . 10
- Summary . . . 11
- Reminders . . . 12

Java Basics Part 02 - Collections and Strings **12**
- Outline . . . 12
- Java's memory model . . . 13
- Arrays . . . 14
- Strings . . . 17
- Equality and immutability . . . 18
- The Collections framework . . . 20

Reference material **24**
- CSSE2002 Diagram & Notation Conventions . . . 24
- Java Collections Framework . . . 36
- Java Mutability . . . 40
- Java Primitive and Reference Types . . . 43

Course Overview and Java Basics

See csse2002[1] for staff, assessment, and course logistics. This is the technical content of Lecture 1, the live lecture. There's also a **recorded** lecture this week, see 2026-02-26-java-basics-part-02-collections-and-strings[2].

Outline

- Programming in the large vs. programming in the small — why this course isn't just "a Java course"
- Java vs Python: compiled vs interpreted, statically vs dynamically typed
- Java's primitive and reference types
- Scope
- Sequence, selection, and iteration in Java
- Java arithmetic

Programming in the large

Code from an intro course (CSSE1001/ENGG1001) has likely been:

1. Small.
2. Written solely by you.
3. For a limited purpose.
4. To exist for a limited time.

…but large software projects are often **none of these**.

CSSE2002 teaches *individual discipline and programming practices such that graduates can design and write software suitable for integration with large software systems.* Professionally you will need to:

- **Document** your code — because someone who didn't write it (or doesn't remember it) may need to work on it.
- **Debug** your code — the more code you write, the more opportunity to make mistakes.
- **Test** your code — it is far better for you to find bugs than for someone else to. You want a reason for confidence in your work beyond "I hope it is okay".

  CSSE2002 teaches practices that support documenting, debugging, and testing of code to allow programming to scale to large software projects.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf

This is not just a Java course. Knowing everything (or even most things) about Java is *not the only goal*. Java is the vehicle. As for why Java: in the Stack Overflow 2025 developer survey, Java is used by 29.6% of professional developers.

Where the course sits in the software lifecycle

In a seven-phase software development lifecycle, CSSE2002 lives in two of the phases, Phase 4 (Development) and Phase 5 (Testing):

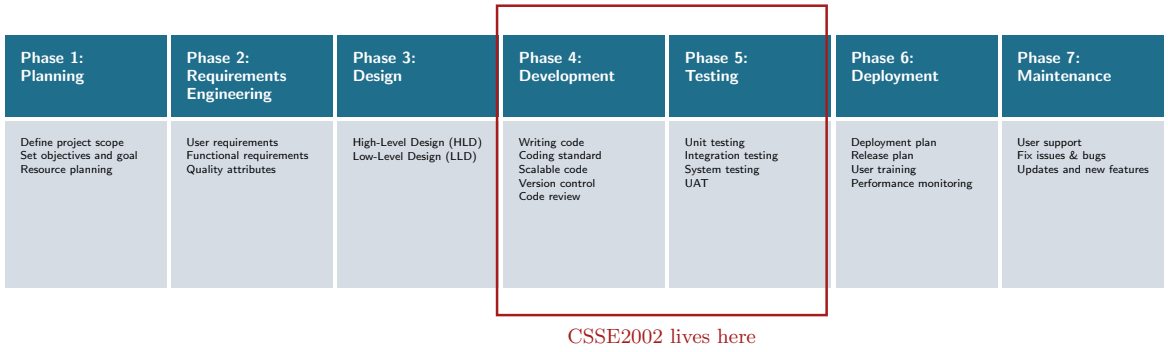


Figure 1: A seven-phase software development lifecycle (SDLC), with a red rectangle around Phases 4–5: CSSE2002 is a *development and testing* course, not a requirements or deployment course.

Java overview

Java is a general-purpose programming language that is:

- **Compiled**
- **Statically typed**
- **Object-oriented**
- **Memory safe**

Java vs Python: compiled vs interpreted

Python is *interpreted*: you run the Python interpreter directly on the source.

```
$ python hello_world.py
```

Java is *compiled*: the source is first compiled to **bytecode**, then run.

```
$ javac HelloWorld.java
$ java HelloWorld
```

Hello World, with the canonical entry point:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Running `javac` on it produces a `.class` file of bytecode. As raw hex, the file begins with the JVM's `cafebabe` magic number:

```
cafebabe00000041001d0a000200030700040c000500060100106a617661
2f6c616e672f4f626a6563740100063c696e69743e010003282956090008
...
```

Homework. What is the benefit of having bytecode? What are the differences between bytecode and machine code?

Java vs Python: static vs dynamic typing

Python is *dynamically* typed — whether an appropriate type is used is determined when the program runs. Java is *statically* typed — types are checked for every scenario at compile time.

> Static typed languages require explicit definition of a data type when they create a piece of data (e.g. variable, parameter, return value). Dynamic languages are the opposite and can infer, or at least try to guess, the type that we're using.

```
x = 1
print(x, type(x))    # 1 <class 'int'>
x = 1.7
print(x, type(x))    # 1.7 <class 'float'>
x = "Hello"
print(x, type(x))    # Hello <class 'str'>
```

Python knows that `1` and `1.7` are different types of value (different "dynamic"/"runtime" types) but doesn't restrict variables on that basis. Java is the opposite — the *variable* carries the type:

```java
int x;
x = 1;      // OK
x = -5000;  // OK
x = 1.0;    // Illegal: not a whole number
x = "15";   // Illegal: cannot convert from a string to a number
```

Python also uses indentation to delimit code blocks, while the Java compiler uses `{}` to define code blocks and `;` to terminate statements.

Data types in Java

See java-primitive-and-reference-types[3] for the full primitive-vs-reference-types table. In short: Java distinguishes **primitive types** (built-in, fixed representations like `int` and `boolean`) from **reference types** — everything else, i.e. classes (including `String` and arrays).

Scope

A variable's scope is where it can be used. In Java, a variable's scope ends at the end of the block in which it's declared. Python has no such rule, so the same code behaves differently in the two languages:

```python
# z does not exist
if 5 > 4:
    z = 2
else:
    z = 3
print(z)          # This works!
```

```java
if (5 > 4) {
    int z = 2;
} else {
    int z = 3;
}
System.out.println(z);   // This does not!
```

Question. What happens when this executes?

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

```java
public static void main(String[] args) {
    if (5 > 4) {
        int z = 2;
    } else {
        int z = 3;
    }
    System.out.println(z);
}
```

It never executes — it's a *compile* error: `z` is declared inside the `if`/`else` blocks, so it's out of scope by the time `System.out.println(z)` is reached.

Answer. Declare the variable inside the same (or a higher) block as the print — hoist the declaration out and leave only the assignments in the branches:

```java
public static void main(String[] args) {
    int z;
    if (5 > 4) {
        z = 2;
    } else {
        z = 3;
    }
    System.out.println(z);
}
```

Control flow: sequence, selection, and iteration

Sequence

1. Each line must end with a semicolon (`;`).
2. Declarations must come before other assignments.
3. Instructions are executed sequentially.

```java
public static void main(String[] args) {
    int x = 1;
    int y = 2;
    int z = x + y;
    System.out.println(z);
}
```

Methods are Java's equivalent of Python functions — but all code must live inside a class:

```python
def add_two_numbers(num1, num2):
    result = num1 + num2
    return result
```

```java
class Arithmetic {           // All code must be inside a class.
    int add_two_numbers(int num1, int num2) {
        int result = num1 + num2;
        return result;
    }
}
```

Calling a method looks the same as calling a Python function: `add_two_numbers(10, 20)`.

The Java method here keeps the Python name unchanged. Java's own convention, used everywhere else in the course (see csse2002-diagram-notation[4] §16), would be `addTwoNumbers`.

Selection: `if` and `switch`

`if` and `switch` are the same shape of control flow, so one diagram covers both:

The `if` chain:

```java
String month;
if (x == 1) {
    month = "Jan";
} else if (x == 2) {
    month = "Feb";
} ... {
    month = "Dec";
} else {
    month = "Unknown";
}
```

The same thing as a `switch`. This is the **arrow form** (`case … ->`), not the old colon-and-`break` form. An arrow case takes a block and does *not* fall through, so no `break` is needed:

```java
String month;
switch (x) {
    case 1 -> {
        month = "Jan";
    }
    case 2 -> {
```

[4] courses/csse2002/csse2002-diagram-notation.pdf

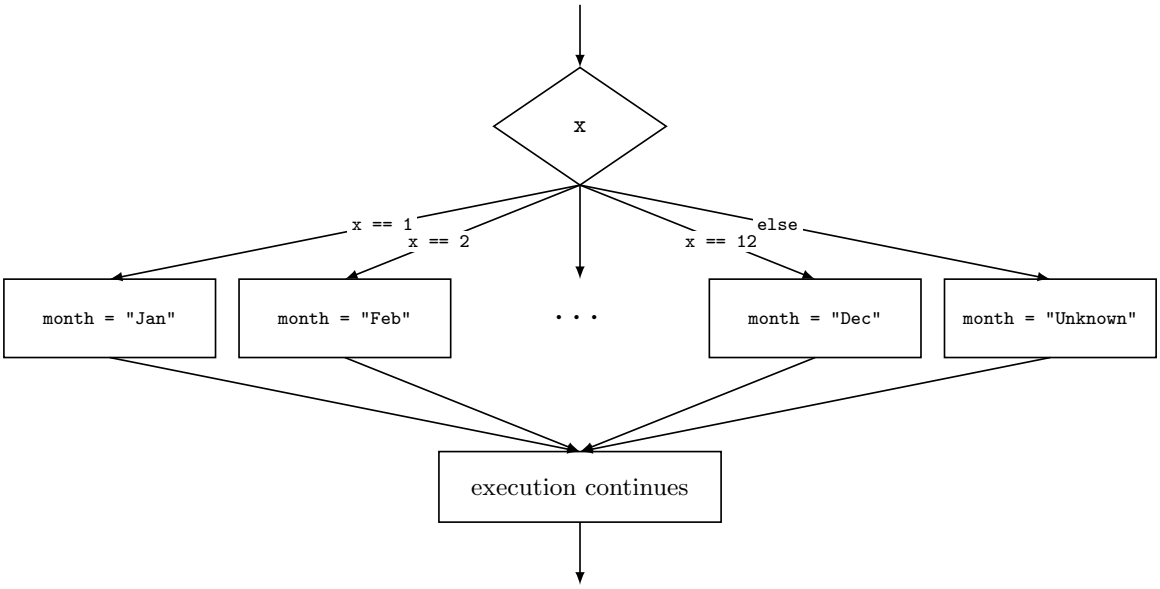


Figure 2: Selection as a flowchart: one test fans out into mutually exclusive branches that all rejoin at a single point. The same diagram describes the `if`-chain *and* the `switch`. The selector `x`, the guards and the assignments are those of the month code below, with one branch box per named outcome.

```
        month = "Feb";
    }
    ...
    default -> {
        month = "Unknown";
    }
}
```

Exercise. Write a method `printGrade` that takes the `mark` of a student and determines the grade:

Mark	Grade Description
83.00 – 100.00	High Distinction
73.00 – 82.99	Distinction
63.00 – 72.99	Credit
50.00 – 62.99	Pass
0.00 – 49.99	Fail

Iteration: `while` and `for` loops

Every counting loop has three moving parts:

- **Initialiser** — initialise a counter variable.
- **Test/condition** — check whether the counter has reached a condition.
- **Update** — increment or decrement the counter.

In a `while` loop the three parts are scattered across three lines:

```
int i = 0;              // initialiser
while (i < 10) {        // test
    System.out.println(i);
    i++;                // update
}
```

```
i = 0
while i < 10:
    print(i)
    i += 1
```

A `for` loop bundles all three into the loop header — the same three fragments, in the same order, separated by semicolons:

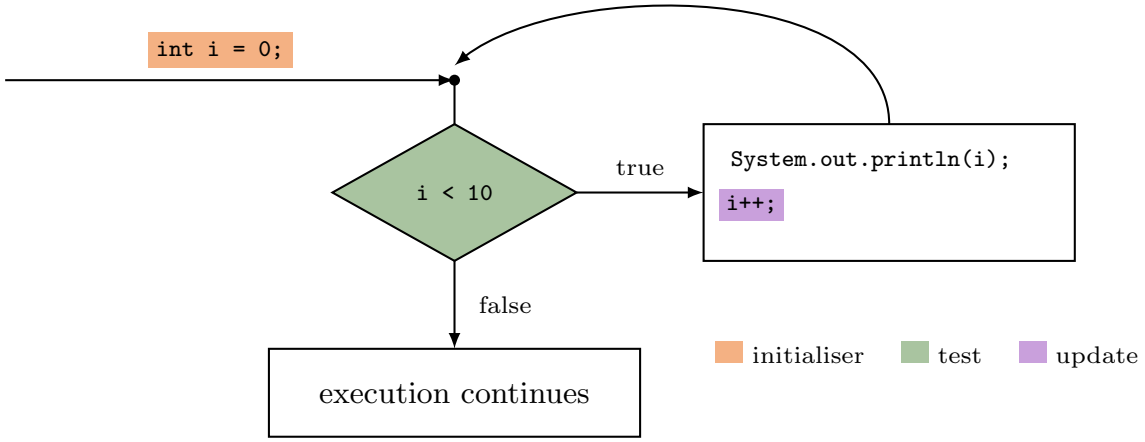


Figure 3: The `while` loop's control flow: the test is evaluated *before* each pass, and the update at the end of the body feeds the back-edge. The initialiser is orange, the test green and the update purple.

```java
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

```python
for i in range(10):
    print(i)
```

Exercise. A factorial of a number n, written $n!$, is the product of all numbers less than or equal to n:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = \prod_{k=1}^{n} k$$

with $0! = 1$, $1! = 1$, $2! = 2 \times 1 = 2$, $3! = 3 \times 2 \times 1 = 6$. Write a method `factorial` that calculates the factorial of its input.

Exercise. For all positive integers it holds that $n! = n \times (n-1)!$. Write an equivalent `factorialRec` method that uses recursion to calculate the result.

Java arithmetic

`+`, `-` and `*` work the same way as Python, except in a very special case.

Division is where the two languages diverge. Java has no `//` operator; instead, `/` *is* integer division whenever **both** operands are integers, and floating-point division as soon as either operand is a floating-point value:

Python	Result	Java	Result
`3 / 2`	`1.5`	`3 / 2`	`1`
`3 // 2`	`1`	*(no // operator)*	—
`3 / 2.0`	`1.5`	`3 / 2.0`	`1.5`
`3.0 / 2`	`1.5`	`3.0 / 2`	`1.5`
`3.0 / 2.0`	`1.5`	`3.0 / 2.0`	`1.5`

Comparison operators are identical in spelling to Python's — `x < y`, `x <= y`, `x > y`, `x >= y`, `x == y`, `x != y`. One caveat applies to `==` alone: this holds **for primitive types**. On reference types, `==` compares object identity, not contents. The equality section of 2026-02-26-java-basics-part-02-collections-and-strings[5] covers this.

Logical operators have the same semantics but different syntax:

Python	Java
`x and y`	`x && y`
`x or y`	x || y
`not x`	`!x`

Augmented assignment and increment/decrement:

Python	Java
`x += y`	`x += y`
`x -= y`	`x -= y`
`x += 1`	`x++`
`x -= 1`	`x--`

Summary

- The entry point for all Java programs is the `main` method: `public static void main(String[] args)`.
- Java is a compiled and statically typed language, compared to Python.
- **Most** arithmetic operations in Java function identically to Python.

[5] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf

- The same core principles of sequence, selection, and iteration apply to Java, but the syntax is different.

Reminders

- Your first Ed Lessons exercise is due by 1pm next Wednesday.
- Applied and practical classes start next week (Week 2).

Tasks this week:

- ☒ Attend the lecture.
- ☐ Week 1 Ed Lessons exercises.
- ☐ Ensure you can access Ed discussion.
- ☐ Enrol in your contact and practical classes.
- ☐ Watch the recorded lecture on Collections — see 2026-02-26-java-basics-part-02-collections-and-strings[6].
- ☐ Install IntelliJ IDEA & Java Development Kit (JDK) 21 — in IntelliJ's *Download JDK* dialog, choose **Version 21**, vendor **Eclipse Temurin (AdoptOpenJDK HotSpot)**.

Java Basics Part 02 - Collections and Strings

See csse2002[7] for staff, assessment, and course logistics. This is the week's **recorded** lecture, a "Tasks this week" item in 2026-02-26-course-overview-and-java-basics[8]. It goes alongside the live Lecture 1. The stack/heap material here is picked up again in the Week 3 applied class, week3-tutorial-stack-heap-and-class-design[9].

Outline

- Java's memory model: stack vs heap
- Arrays and their limitations
- Strings: indexing, substrings, equality, immutability
- The `Stack`, `List`, `Set`, and `Map` collections

[6] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf
[7] courses/csse2002/csse2002.pdf
[8] courses/csse2002/2026-02-26-course-overview-and-java-basics.pdf
[9] courses/csse2002/week3-tutorial-stack-heap-and-class-design.pdf

Java's memory model

Runtime memory (the memory of the program while it's running) splits into:

- **The stack** — local variables and method parameters.
- **The heap** — values with a **dynamic size** (objects and shared data).

Worked example: stack frames

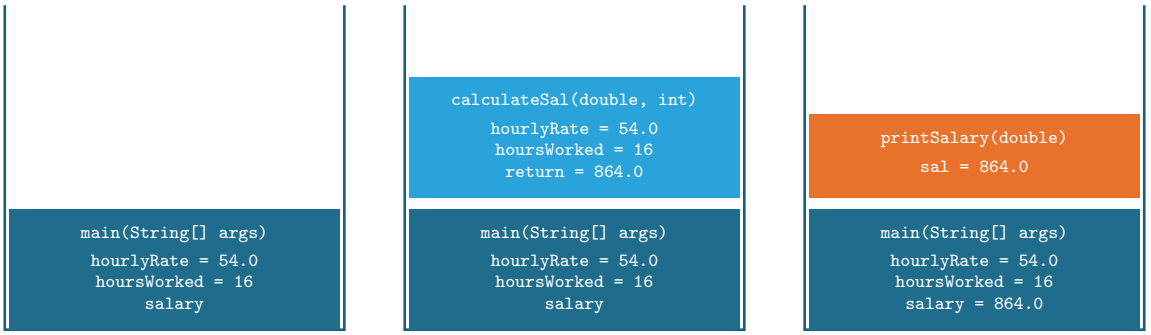
```
public class StackHeap {
    public static void main(String[] args) {
        double hourlyRate = 54.0;
        int hoursWorked = 16;

        double salary = calculateSal(hourlyRate, hoursWorked);

        printSalary(salary);
    }
    public static double calculateSal(double hourlyRate, int hoursWorked) {
        return hourlyRate * hoursWorked;
    }
    public static void printSalary(double sal) {
        System.out.println("Salary: " + sal);
    }
}
```

The call stack, one frame at a time:

`main` pushes a frame holding `hourlyRate = 54.0`, `hoursWorked = 16` and a not-yet-assigned `salary`. Calling `calculateSal` pushes a new frame on top with its *own* copies of `hourlyRate` and `hoursWorked`; it computes `54.0 * 16 = 864.0` and returns, popping its frame and storing the result in `main`'s `salary`. `main` then calls `printSalary`, which pushes a frame holding `sal = 864.0`, prints `Salary: 864.0`, and pops.

Worked example: heap allocation

An array's size isn't known until runtime, so the array *object* can't live in the frame — it goes on the heap, and the frame holds only a reference to it:



① `main` is running ② `calculateSal` pushed on top ③ popped; `printSalary` pushed

Figure 4: The call stack for the salary example at three moments. Each method call gets its own frame holding its parameters and locals; frames are pushed on top and popped in the reverse order (last in, first out), so `calculateSal` is gone by the time `printSalary` is pushed. The `main` frame is dark teal, `calculateSal` blue and `printSalary` orange; the open-topped container is the stack.

```
static float lastMark(int n) {
    float[] marks = new float[n];

    marks[0] = 75.5f;
    marks[n - 1] = 88.0f;

    return marks[n - 1];
}
```

Java never clears heap memory when a method ends. It clears heap memory only when no references exist **and** the Garbage Collection decides to run.

Arrays

An array is an ordered, **fixed-length**, mutable sequence of **homogeneous** items — the closest Java equivalent to a Python list, except that a Python list (`list1 = ["abc", 34, True, 40, "UQ"]`) is neither fixed-length nor homogeneous.

```
int[] numbers = {1, 2, 3, 4, 5};
numbers.length;         // 5   -- a field, not a method call
numbers[0] = 0;         // OK  -- {0, 2, 3, 4, 5}
numbers[0] = "zero";    // illegal -- all elements must be the same type, int
```

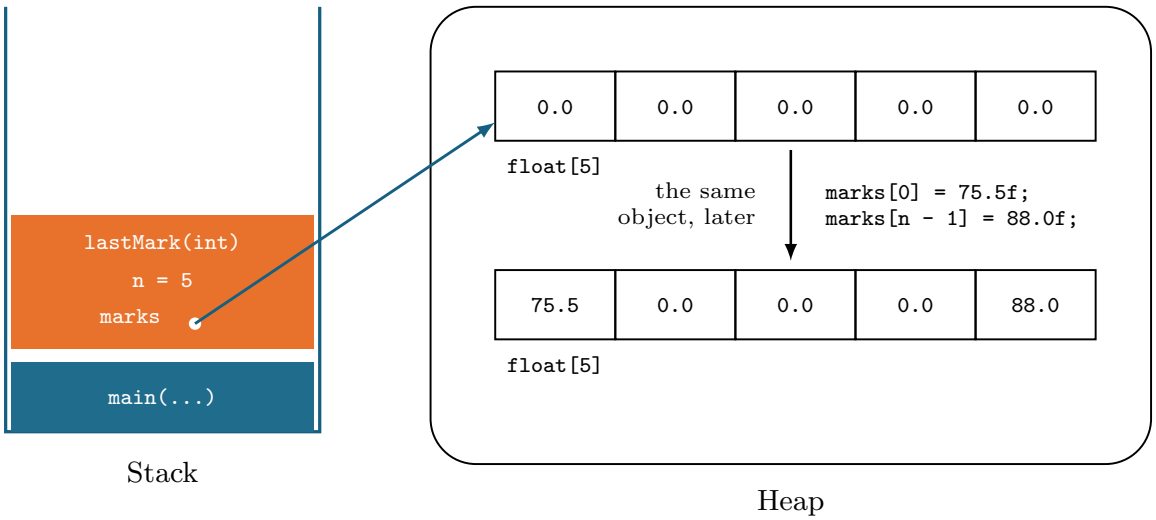


Figure 5: Calling `lastMark(5)`. `n` is a primitive, so its value sits in the frame; `marks` is a reference, so the frame holds only a pointer at a `float[5]` object on the heap. `new float[n]` zero-fills the array, and the two assignments mutate that heap object *through* the reference, leaving `[75.5, 0.0, 0.0, 0.0, 88.0]`. The teal arrow is the reference, the open-topped container is the stack and the rounded region is the heap; the orange callee frame sits over the teal `main` frame.

Two ways to create one:

```
// Approach 01: array literal
float[] marks = {60.3f, 62f, 70.1f, 65.8f, 80.3f};

// Approach 02: allocate, then assign each index
float[] marks = new float[5];
marks[0] = 60.3f;
marks[1] = 62f;
marks[2] = 70.1f;
marks[3] = 65.8f;
marks[4] = 80.3f;
```

The `f` suffixes are needed. Without them, `float[] marks = {60.3, 62, 70.1, 65.8, 80.3}` does not compile: an unsuffixed `60.3` is a `double`, and Java will not narrow a `double` to a `float` implicitly. The heap example writes `75.5f` / `88.0f` the same way.

Querying. Valid indices run from `0` to `length - 1`:

```
float[] marks = {60.3f, 62f, 70.1f, 65.8f, 80.3f};
System.out.println(marks[3]);      // 65.8
System.out.println(marks[0]);      // 60.3
int len = marks.length;            // 5
System.out.println(marks[len]);    // ArrayIndexOutOfBoundsException -- index 5
↪  doesn't exist
```

Iterating. An indexed `for` loop, or the enhanced `for`-each loop:

```
for (int i = 0; i < len; i++) {
    System.out.println(marks[i]);
}

for (float mark : marks) {
    System.out.println(mark);
}
```

The general shape of a for-each is `for (ElementType name : collection) { … }` — e.g. `for (String name : people) { … }`.

Try at home.

- `max(int[])` — takes an array of integers and returns the maximum value.
- `contains(int[], int)` — returns `true` if the input array contains the input integer, `false` otherwise.
- `reverse(int[])` — returns a *new* array with the elements in reverse order.

Limitations of arrays

Arrays are not great for everything:

- **Fixed size at creation.** You do not always know how much space you are going to need.
- **They do not automatically close gaps.**

On the second point: removing an element leaves a *hole*, and closing it is work you have to do yourself:

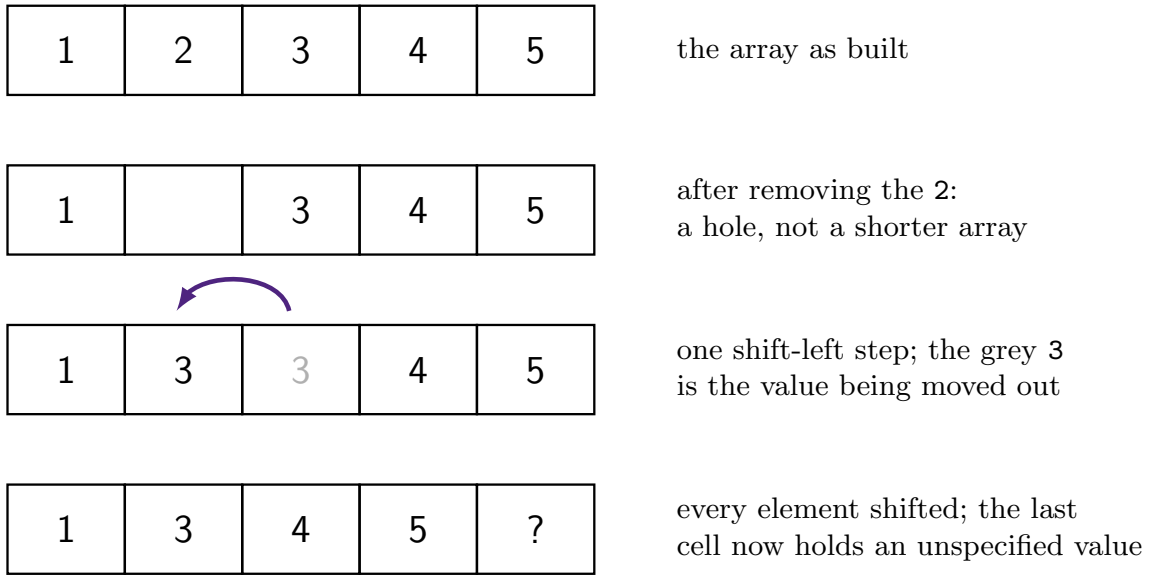


Figure 6: Why arrays motivate collections: deleting an element leaves a hole, and closing it means shifting every later element left by hand — and the array is still the same length, so the trailing cell has to be given some value. The notation is the array convention of csse2002-diagram-notation[10] §13: equal-width cells with no index row, a grey ghosted digit for a value being shifted out, `?` for an unspecified value, and a purple curved arrow above the row for one shift-left step. After the last row, the trailing cell is filled with a sentinel `-1`.

Strings

A `String` is a sequence of characters. The running example:

[10] courses/csse2002/csse2002-diagram-notation.pdf

```
String course = "Programming in the Large";
```

Laid out with indices, this is 24 characters long — indices 0–23:

	P	r	o	g	r	a	m	m	i	n	g	␣	i	n	␣	t	h	e	␣	L	a	r	g	e
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

```
System.out.println(course.length());     // 24  -- length() is a method on String
System.out.println(course.charAt(0));     // P
System.out.println(course.charAt(11));    // (the space between "Programming" and
 ↪  "in")
System.out.println(course.charAt(23));    // e
```

The three `charAt` results are read off the index row above.

One asymmetry trips people up: an array uses the **field** `marks.length`, a `String` uses the **method** `course.length()`.

`substring(start, end)` returns a substring with an **inclusive** start index and an **exclusive** end index. If the end argument is not specified, the substring runs to the end of the string.

```
System.out.println(course.substring(0, 11));  // Programming  -- index 11 (the space)
 ↪  is excluded
System.out.println(course.substring(19));     // Large
```

Equality and immutability

`=`, `==` and `!=` mean different things depending on which kind of type they are applied to. This is the "for primitive types" caveat on `==` in 2026-02-26-course-overview-and-java-basics[11]:

	Primitive types	Reference types
`x = y`	make `x` store a copy of `y`'s value	make `x` refer to the same object `y` refers to
`x == y`	check if `x` stores the same value as `y`	check if `x` refers to the same object as `y`
`x != y`	check if the value stored in `x` is different to `y`	check if `x` and `y` are referencing different objects

[11] courses/csse2002/2026-02-26-course-overview-and-java-basics.pdf

```java
String name  = "Jack";
String name2 = "Jack";
String name3 = new String("Jack");

System.out.println(name == name2);        // true
System.out.println(name == name3);        // false

System.out.println(name.equals(name2));   // true
System.out.println(name.equals(name3));   // true

Object obj1 = new Object();
Object obj2 = new Object();
System.out.println(obj1.equals(obj2));    // false
```

Two literals with the same contents are `==` because of the **string pool**; a `new String(...)` opts out of the pool and allocates a distinct object, so it is `!=` but still `.equals()`. `Object`'s own `equals()` is identity, so two fresh `Objects` are never equal.

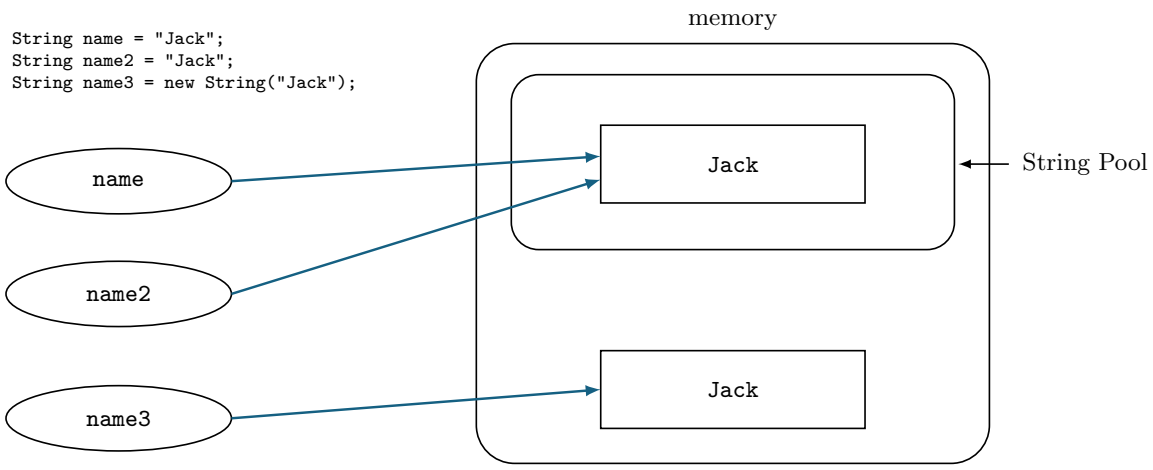


Figure 7: Why `name == name2` is `true` but `name == name3` is `false`. Identical string *literals* are interned once in the string pool and shared; a `new String(...)` call allocates its own object outside the pool. `.equals()` compares contents and so is `true` for all three. Ellipses are variables, rounded regions are memory, and teal arrows are references.

String objects are immutable. A `String` cannot be changed. Reassigning

```java
String name = "Jack";
name = "Jill";
```

does not edit the `"Jack"` object. It leaves `"Jack"` untouched in memory and repoints the `name` reference at a different object, `"Jill"`. Immutability comes back as a design property in java-mutability[12].

The Collections framework

See java-collections-framework[13] for the full `Stack`/`List`/`Set`/`Map` interface reference. The rest of this section works through a trace of each. All of these live in `java.util.*`.

Stack

Last In, First Out (LIFO):

Method	Meaning
`empty()`	Is this stack empty?
`peek()`	Return the object at the top of the stack
`pop()`	Remove the object at the top of the stack
`push(obj)`	Put `obj` on the top of the stack

Assume `letters` is an empty stack:

```
letters.empty();       // true
letters.push("A");
letters.empty();       // false
letters.push("B");
letters.push("C");
letters.peek();        // "C"
letters.push("D");

letters.pop();         // "D"
letters.pop();         // "C"
letters.pop();         // "B"
letters.pop();         // "A"
letters.pop();         // EmptyStackException
```

Creating a stack (must import the `java.util.Stack` package):

[12] courses/csse2002/java-mutability.pdf
[13] courses/csse2002/java-collections-framework.pdf

```
Stack<Type> stacks = new Stack<>();

Stack<Integer> stacks = new Stack<>();   // Integer type stack
Stack<String>  stacks = new Stack<>();   // String type stack
Stack<Cat>     stacks = new Stack<>();   // Cat type stack
```

Collections only store objects, so `Stack<int>` and `Stack<double>` are illegal. Java has a class for each primitive type: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short` (see java-primitive-and-reference-types[14]).

Exercise. Implement a method `int sum(Stack)` that returns the sum of all integers in the given stack.

List

Lists hold items in sequential order, much like an array, but:

- Grow and shrink automatically.
- No fixed size limit.
- Insert an item anywhere in the list.
- Remove an item anywhere in the list.
- Check if an item is in the list.
- Index from zero, start of list.

`List` is an **interface**, not a particular implementation — you can declare a type as a list (`List<String> xs`), but you cannot create a list instance (`new List<>()` is illegal). Popular implementations:

- **`LinkedList<String>`** — better for operations that modify the middle of the list.
- **`ArrayList<String>`** — better for random access.

A trace, with the state after each line:

```
List<String> courses = new ArrayList<>();
courses.add("CSSE1001");     // [CSSE1001]
courses.add("DECO3801");     // [CSSE1001, DECO3801]
courses.get(0);              // "CSSE1001"
courses.add(1, "CSSE2310");  // [CSSE1001, CSSE2310, DECO3801]
courses.add(1, "CSSE2002");  // [CSSE1001, CSSE2002, CSSE2310, DECO3801]
courses.remove(2);           // removes and returns "CSSE2310"
                             // [CSSE1001, CSSE2002, DECO3801]
```

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

`remove(int)` removes by index

`remove(int)` removes by *index*, and index 2 at that moment holds `"CSSE2310"`. So `courses.remove(2)` returns `"CSSE2310"`, not `"CSSE2002"`: `CSSE2002` **survives** at index 1 and `CSSE2310` is gone.

The reason `add(1, …)` twice does not put both new entries at index 1 is that every insertion renumbers everything above it — the same shift work the array section did by hand, done for you:

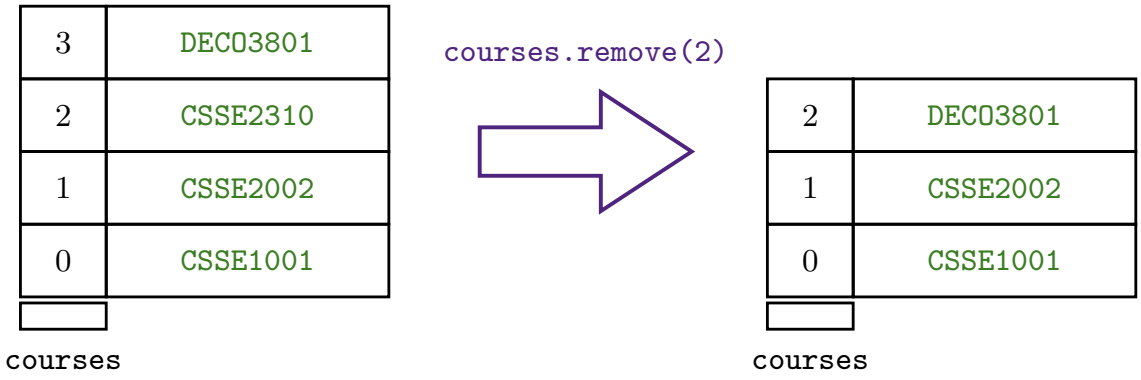


Figure 8: `courses.remove(2)` on the `courses` `ArrayList`, before and after. Everything above the removed index slides down one and the list shrinks — which is exactly the gap-closing the raw array could not do. The list is drawn with **index 0 at the bottom**, growing upward: indices on the left, values in green on the right, and the variable name beneath.

Set

Sets store unique items (no duplicates). You can iterate over the set (**don't assume ordering**), add an item, remove an item, and check if an item is in the set. `add` returns a `boolean` saying whether the set actually changed:

```
Set<String> farm = new HashSet<>();
farm.add("Fox");                    // true
farm.add("Farmer");                 // true
farm.add("Chicken");                // true
farm.add("Fox");                    // false -- already present, set unchanged
farm.add("Grain");                  // true
System.out.println(farm.size());    // 4
```

Implementations: `TreeSet<E>` (`E` must implement `Comparable`, e.g. `String`) and `HashSet<E>` (`E` must have sensible `hashCode()`/`equals()`).

Map

Maps store key → value pairs (similar to a Python dictionary). You need to specify a type for each: `Map<Integer, String>` is `Integer` keys mapping to `String` values, and the value type can itself be a collection, e.g. `Map<String, ArrayList<String>>`.

```
Map<String, Integer> farm = new HashMap<>();
farm.put("Fox", 5);
farm.put("Farmer", 10);
farm.put("Chicken", 0);
farm.get("Farmer");                          // 10
farm.put("Fox", 18);                         // key already present -> value
 ↪  overwritten
farm.put("Grain", 15);
farm.put("Grain", farm.get("Grain") + 5);    // read-modify-write: Grain -> 20
```

Final contents: `Fox → 18`, `Farmer → 10`, `Chicken → 0`, `Grain → 20`. A `HashMap` has **no** meaningful iteration order, so don't read anything into the order the entries are listed in.

Implementations: `TreeMap<K,V>` (`K` must implement `Comparable`) and `HashMap<K,V>` (`K` must have sensible `hashCode()`/`equals()`).

The hashCode/equals contract

`HashSet<E>` needs `E`, and `HashMap<K,V>` needs `K`, to have sensible `hashCode()` and `equals()` methods:

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

A `hashCode()` returns an integer value generated by a hashing algorithm for an object:

```
String name = "Thilina";
System.out.println(name.hashCode());   // 318621125
```

Reference material

CSSE2002 Diagram & Notation Conventions

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

How this note was built

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

Pages actually read

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

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

I did **not** read: L01a 1–17 & 28–62, L01b 1–30 & 43–77, L02 21–30 & 46–54, L03 1–13 & 32–62, L04 1–5 & 37–49, L05 15–42, L06 1–14 & 33–52, L07 1–13 & 31–39, L08 1–2 & 17–50, L09 1–18 & 34–46, L11 1–19 & 33–34, L13 1–25 & 41–89. A few citations below (textbook mentions, "Further Reading" slides) come from **extracted text on pages I did not view as images**; those are marked *(text only)*. Font names come from `pdffonts`; hex colours were sampled from rendered pixels, not from any stated palette.

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

1. The page frame every figure sits in

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

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

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

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

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

2. Typeface roles

Role	Face	Evidence
Slide titles, section headings	Tw Cen MT, purple	everywhere in Template A
Prose, captions, labels on the lecturer's own diagrams	Bell MT (serif)	L03 p27; L06 p21; L07 p16
Java source; Java identifiers inline in prose; text inside class/dependency boxes	**Inconsolata** (hand-set) or **Courier New** (pasted)	L02 p32, p36; L07 p27; L08 p6
Borrowed / stock figures	Arial or similar sans	L02 p15; L06 p17; L08 p3; L09 p27

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

3. What colour means

Colour	Sampled hex	Meaning	Evidence	Consistency
UQ purple	`#4E247F`	chrome, slide titles; nothing semantic inside a figure	everywhere	throughout
Teal	`#156082`	**every relationship line, connector and pointer the lecturer draws**	L03 p27; L04 p7; L06 p18, p20, p21; L08 p7, p11, p14; L09 p25, p29; L13 p27	strongest colour convention in the course — weeks 3–13
Magenta	`#9D0782` – `#A02B93`	subtitle line; a term being defined; a verdict on an example	L02 p35; L04 p27, p30; L05 p5; L08 p5, p8; L09 p21, p27, p30	very consistent
Pure red	`#FF0000`	a hard rule to memorise; the outline of a code-highlight box; deadlines	L01a p24; L02 p38, p40; L03 p28; L04 p14; L07 p15, p20; L08 p16	consistent for "stop and look"
Green	`#3B7D23` (Javadoc), bright green (chart)	Javadoc/comment text in code; passing tests in the one chart	L04 p27; L05 p5	chart usage is a single instance
Near-black	`#171717`	prose, box outlines, arrows in *borrowed* figures	L05 p6, p8; L07 p26	throughout

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

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

4. Class-hierarchy diagrams — the dominant style

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

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

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

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

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

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

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

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

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

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

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

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

7. The pre/postcondition "funnel" figure

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

- A method is a small **square box, thin black outline, monospace name** (`A.f`, `B.f`).
- **Above** it, a cone that narrows downward into the box. Its wide upper ellipse is the set of inputs the method accepts — the **precondition**.
- **Below** it, a cone that widens downward out of the box. Its wide lower ellipse is the set of outcomes it may produce — the **postcondition**.
- **Width = weakness.** A *wider* top mouth = a *weaker* (more permissive) precondition. A *wider* bottom mouth = a *weaker* (less committal) postcondition. So a legal subclass override has a top mouth **at least as wide** and a bottom mouth **no wider** than its parent's.

- Labels `preconditions` and `postconditions` in **lowercase serif** at the far left of the slide, vertically aligned with the two mouths.
- Entirely **black line art, unfilled, no colour**.
- To compare parent and child, the child's cones are **overlaid on the parent's** in the same frame in thinner/greyer stroke (L07 p17).

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

8. Coupling and cohesion figures

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

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

9. The coupling-levels ladder

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

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

10. Annotating code

- **Highlight a token:** a thin **red `#FF0000` rectangle** drawn around it. *Cited:* L02 p38, p40; L03 p28; L04 p14; L08 p16. Dominant across four decks.
- **Point at a token:** a thin **teal straight or diagonal line with a small solid arrowhead**, running between the token and a **serif caption placed outside the code block, unboxed**. *Cited:* L02 p16; L03 p24, p28; L08 p14; L09 p25, p29; L13 p27. Weeks 2–13 — very consistent.

- **Show a replacement: strike through** the token being removed and put the replacement beside it in red. *Cited:* L02 p4; L07 p20; L13 p27.
- **Mark an error inline:** a red squiggly/dashed underline, IntelliJ-style. *Cited:* L09 p22.
- **Aside / rule-to-remember:** a short sentence centred beneath the content in serif — sometimes inside a thin rectangle, sometimes bare. *Cited (boxed black):* L02 p31; L07 p24; L08 p5; L09 p26. *Cited (boxed teal):* L03 p21. *Cited (unboxed):* L03 p23; L06 p31; L08 p12, p15; L07 p14, p25. Inconsistent — see below.
- **Yellow highlighter** behind a heading inside a worked-example box. *Cited:* L07 p19 (single instance).
- **Verdict on an example**, placed in the right margin in magenta serif: "Bad Example", "Better / Still Bad ", "Does not satisfy specification!", "Satisfies specification!". *Cited:* L04 p26, p27, p28, p29, p30, p31. Consistent within L04.

11. Code blocks

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

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

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

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

12. Specification / Javadoc notation

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

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

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

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

13. Arrays and stacks

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

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

14. Black-box / glass-box testing icon

A **grey-filled isometric cube, unlabelled**, with a **black horizontal arrow entering on the left** labelled `input` in serif above the line, and a matching arrow **leaving on the right** labelled `output`. Flow is strictly left→right. The glass-box variant is the same cube rendered

translucent with the source code visible inside it. *Cited:* **L05 p6** (both cubes together), L05 p8 (black-box cube reused as a corner marker).

15. Attribution and textbook lineage

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

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

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

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

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

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

16. Naming and capitalisation

- Java identifiers follow standard Java: `PascalCase` types, `camelCase` methods and fields, `SCREAMING_SNAKE` constants, always monospace. **Consistent everywhere.**
- Running example names: `Person`, `Employee`/`ContractEmployee`/ `FullTimeEmployee`, `Animal`/`Dog`/`Cat`, `BankAccount`/`AccountHolder`, `Foo`, `Printer`, and for abstract structural examples `class A/B/C`, `class Z`, `interface AI/BI/CI`, `interface BSpec`.
- **Interfaces in structural diagrams are named either `<Letter>I`** (`AI`, `BI`, `CI` — L07 p27) **or `<Class>Spec`** (`BSpec` — L07 p29, p30). Both in the same deck.
- Method-signature shorthand in prose uses UML order — `getType():String`, `addListener(String type, Consumer<SimpleEvent> action): action` (L11 p28) — while diagram boxes use the same order (L02 p36). Consistent.

- A right arrow → in serif means "resolves to" / "dispatches to": `kapow(Object) → locked in`. *Cited:* L13 p26, p28.
- ASCII `->` between monospace type names means "widens to" in the casting chains: `byte -> short -> char -> int -> long -> float -> double`. *Cited:* L03 p26.

Inconsistencies

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

1. **Three different ways to draw inheritance.**
   - *H1* — black-outlined box, **serif** class name, teal orthogonal connector, plain solid teal arrowhead into the parent, no edge label: **L08 p7, L08 p11**.
   - *H2* — UML: black-outlined **three-compartment** box, monospace name, black orthogonal connector, **unfilled triangle** arrowhead: **L02 p36, L02 p40**.
   - *H3* — black-outlined box, **serif** name, **teal diagonal** (not orthogonal) lines, plain arrowhead, and every edge **labelled "Is a" / "Is an"** in serif: **L03 p27**. All three put the superclass above. They disagree on arrowhead shape, on routing, and on whether edges are labelled. **Decision needed.**
2. **Hierarchy box border colour.** Purple `#4E247F` at **L04 p7** versus black `#171717` at **L08 p7 and L08 p11** — same figure type, four weeks apart.
3. **Class name typeface inside a box.** Serif at **L04 p7, L08 p7, L08 p11**; monospace at **L02 p36, L07 p27, L07 p29**.
4. **Where the class name goes.** Inside a dedicated name compartment at **L02 p36**; floating **above** an unlabelled two-compartment box as "Person Class" at **L02 p12 and L02 p13** — within one deck.
5. **Specification tag spelling — three variants.**
   - `@requires` / `@ensures` — **L04 p23** (the tag table itself), **L13 p36** (which additionally misspells it `@requiers`).
   - `@require` / `@ensure` (no `s`) — **L07 p18, p19, p20**, used consistently across that whole deck's worked example.
   - `@returns` instead of `@return` — **L05 p8**. **Decision needed.** The tag table (L04 p23) and the exam-prep deck (L13 p36) agree on the `-s` forms; the SOLID deck (L07) does not.

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

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

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

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

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

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

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

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

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

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

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

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

Not established by the source

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

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

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

Java Collections Framework

Introduced in 2026-02-26-java-basics-part-02-collections-and-strings[15] (Lecture 1, Week 1) — see that lecture for the worked push/pop/add/remove traces. All of these live in `java.util.*`, and (unlike arrays) grow/shrink automatically.

Why collections instead of arrays?

An array is an ordered **fixed-length** mutable sequence of **homogeneous** items, and that costs you two things:

1. **Fixed size at creation.** You have to know up front how much space you need.
2. **They do not automatically close gaps.** Removing an element from the middle leaves a hole; closing it means shifting every later element down by hand, and the array is still the same length afterwards, so the trailing cell has to be given some filler value.

Collections solve both, at the cost of only being able to store **objects** (reference types) — see java-primitive-and-reference-types[16] for the primitive wrapper classes (`Integer`, `Double`, etc.) used to store primitives inside them.

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

Stack — LIFO

Method	Description
`empty()`	Is this stack empty?
`peek()`	Return the object at the top of the stack
`pop()`	Remove (and return) the object at the top of the stack
`push(obj)`	Put `obj` on the top of the stack

```
import java.util.Stack;

Stack<Type> stacks = new Stack<>();
```

`pop()` on an empty stack throws `EmptyStackException`.

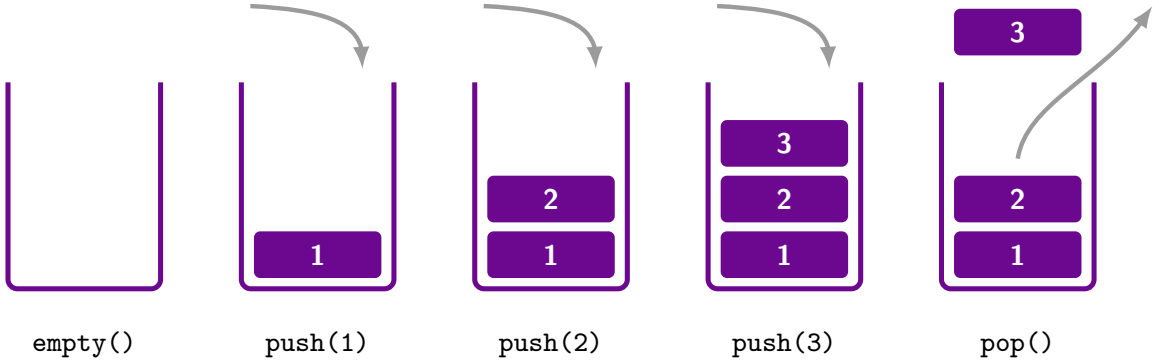


Figure 9: Last In, First Out: `push` always adds at the top and `pop` always removes from the top, so the last value in is the first value out. The stack is an open-topped container whose entries grow upward. An arrow *into* the container is a push and an arrow *out and away* is a pop. Each frame is labelled with its operation, and the frames read left to right as time. In the `pop()` frame the `3` is lifted clear of the container to show it gone.

List

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

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

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

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

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

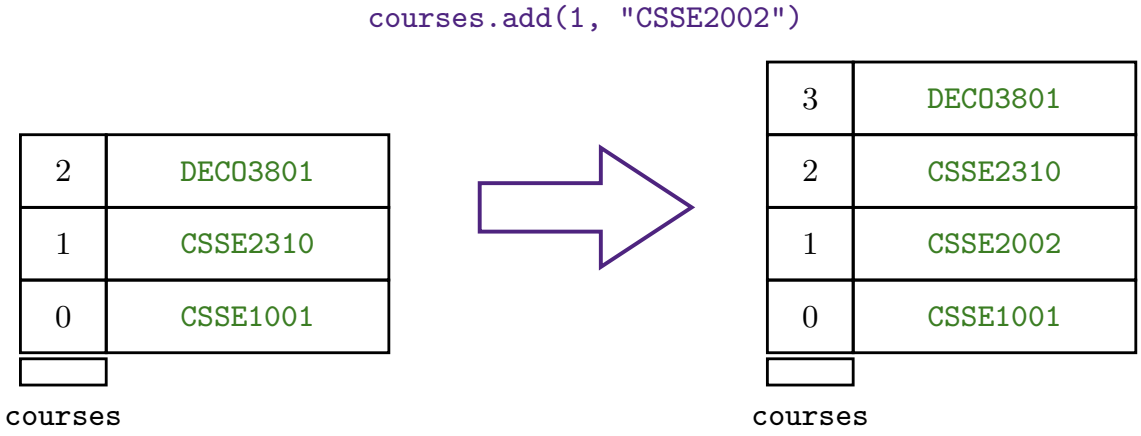


Figure 10: `add(index, e)` on a `List`: `e` lands *at* that index and every entry from there up renumbers, so the list gets longer without you writing a single shift. (`remove(index)` is the mirror image — see the removal step traced in 2026-02-26-java-basics-part-02-collections-and-strings[17].) The list has **index 0 at the bottom** and grows upward, with index cells on the left, values in green on the right, and the variable name in a small box beneath the stack of cells.

Set

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

[17] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf

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

`add` returns `true` if the set changed and `false` if the item was already present — so `farm.add("Fox")` a second time returns `false` and leaves `size()` alone.

- **`TreeSet<E>`** — `E` must implement `Comparable` (e.g. `String`).
- **`HashSet<E>`** — `E` must have sensible `hashCode()`/`equals()`.

Map

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

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

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

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

The hashCode/equals contract

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

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

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

Java Mutability

Introduced in 2026-03-12-object-oriented-programming-ii[18] (Lecture, Week 3). Builds on java-encapsulation[19].

Mutable vs immutable classes

Objects in Java can either change their state after creation or remain fixed forever.

Two types of class design:

- **Mutable class** — object state *can* change after creation.
- **Immutable class** — object state *cannot* change after creation.

Mutable

State is reachable through setters, so it can change at any point in the object's life:

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

    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
}
```

```
class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Alice");
        p.setAge(20);

        p.setAge(30);   // state changed
    }
}
```

This object can change at any time.

[18] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[19] courses/csse2002/java-encapsulation.pdf

Immutable

The immutable version changes four things at once: the class is `final`, both fields are `final`, the values arrive through the constructor, and there are no setters — only getters.

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

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

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

```
class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Alice", 20);
        Person p2 = new Person("Bob", 30);
        System.out.println(p1.getName() + " " + p1.getAge());
    }
}
```

Changing a value now means constructing a new object, not mutating the old one.

Why `final` on the class

A subclass can introduce mutability (**Effective Java (2nd ed.), Item 16, page 78**). If the class is subclassable, a subclass can add mutable state or override methods to return values that vary, so callers can no longer rely on "an instance of this type never changes". Marking the class `final` is what forecloses that.

Arrays and lists break naive immutability

Even if a class looks immutable, it may not be.

`final` on an array field only prevents **reassigning the reference**. It does nothing to protect the array's *contents*, and a getter that returns the field hands the caller a live handle on the object's own state:

```java
public final class Person {
    private final String name;
    private final String[] phoneNumbers;

    public Person(String name) {
        this.name = name;
        this.phoneNumbers = new String[]{"111", "222"};
    }

    public String getName() { return name; }

    public String[] getPhoneNumbers() { return phoneNumbers; }
}
```

```java
class Main {
    public static void main(String[] args) {
        Person person = new Person("John");
        person.getPhoneNumbers()[0] = "999999";

        System.out.println(person.getPhoneNumbers()[0]);   // 999999
    }
}
```

Every field is `private final` and there is not a setter in sight, yet the object's state changed. The rule:

- `final` for arrays means the **reference** cannot change, not the contents.
- `final` prevents *reassigning* the array:

```java
this.phoneNumbers = new String[]{"123"}; // NOT allowed if already assigned
```

Defensive copying

To protect mutable objects, create copies — on the way in *and* on the way out, so neither the constructor's caller nor the getter's caller keeps a handle on the field (**Effective Java, Item 50: "Make defensive copies when needed"**):

```java
public final class Person {
    private final String name;
    private final String[] phoneNumbers;

    public Person(String Name, String[] phoneNumbers) {
        this.name = Name;
```

```
        this.phoneNumbers = phoneNumbers.clone();   // defensive copy
    }

    public String[] getPhoneNumbers() {
        return phoneNumbers.clone();                // defensive copy
    }
}
```

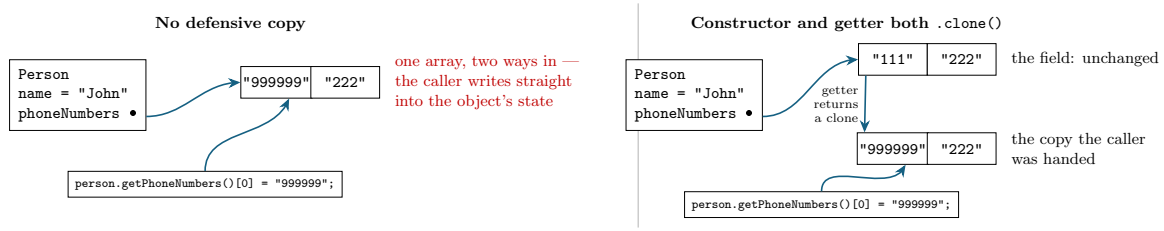


Figure 11: Why a class whose getter returns its array field is not immutable, and what `.clone()` changes. Left: the field and the getter's return value are the same array, so the caller's write lands in the object's state. Right: the getter hands out a copy, so the write lands somewhere the object does not care about.

`clone()` on an array is a shallow copy — it duplicates the row of references, not the objects they point at. That is enough here because `String` is itself immutable, but a `Person[]` of mutable `Person` objects would still leak.

Java Primitive and Reference Types

Introduced in 2026-02-26-course-overview-and-java-basics[20] (Lecture 1, Week 1).

Two categories of types

Java distinguishes two sorts of types:

1. **Primitive types** — built-in, fixed representations.
2. **Reference types** — everything else (i.e. classes), including `String`, arrays, and any user-defined class.

Primitive types

[20] courses/csse2002/2026-02-26-course-overview-and-java-basics.pdf

Type	Size	Stores
`boolean`	not specified	`true` or `false`
`byte`	1 byte / 8 bits	whole numbers from -128 to 127
`short`	2 bytes / 16 bits	whole numbers from -32,768 to 32,767
`int`	4 bytes / 32 bits	whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes / 64 bits	whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes / 32 bits	fractional numbers, accurate to 6-7 decimal places
`double`	8 bytes / 64 bits	fractional numbers, accurate to 15 decimal places
`char`	2 bytes / 16 bits	a single character/letter

The JVM does not specify a width for `boolean`.

Reference types

Reference types are everything that isn't primitive — classes, including `String` and arrays. **A variable of a reference type stores a reference to an object, not the object's data directly.** That difference is visible in the runtime memory model: a primitive's value lives in the method's stack frame, while a reference type's frame slot holds only an arrow at a heap object.

Why the distinction matters

Assignment and equality mean different things on each side. `x == y` compares stored values only for primitive types:

	Primitive types	Reference types
`x = y`	make `x` store a copy of `y`'s value	make `x` refer to the same object `y` refers to
`x == y`	check if `x` stores the same value as `y`	check if `x` refers to the **same object** as `y`

	Primitive types	Reference types
`x != y`	check if the value stored in `x` differs from `y`'s	check if `x` and `y` reference different objects

So on reference types you almost always want `.equals()`, not `==` — see 2026-02-26-java-basics-part-02-collections-and-strings[21] for the `String` pooling example that makes this concrete, and for how the stack/heap split follows from it.

Wrapper classes

Collections (`Stack`, `List`, `Set`, `Map`) can only store objects/reference types, so Java provides a class for each primitive type:

Primitive	Wrapper class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

`Stack<int>` is illegal; `Stack<Integer>` is what you write. See java-collections-framework[22].

[21] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf
[22] courses/csse2002/java-collections-framework.pdf

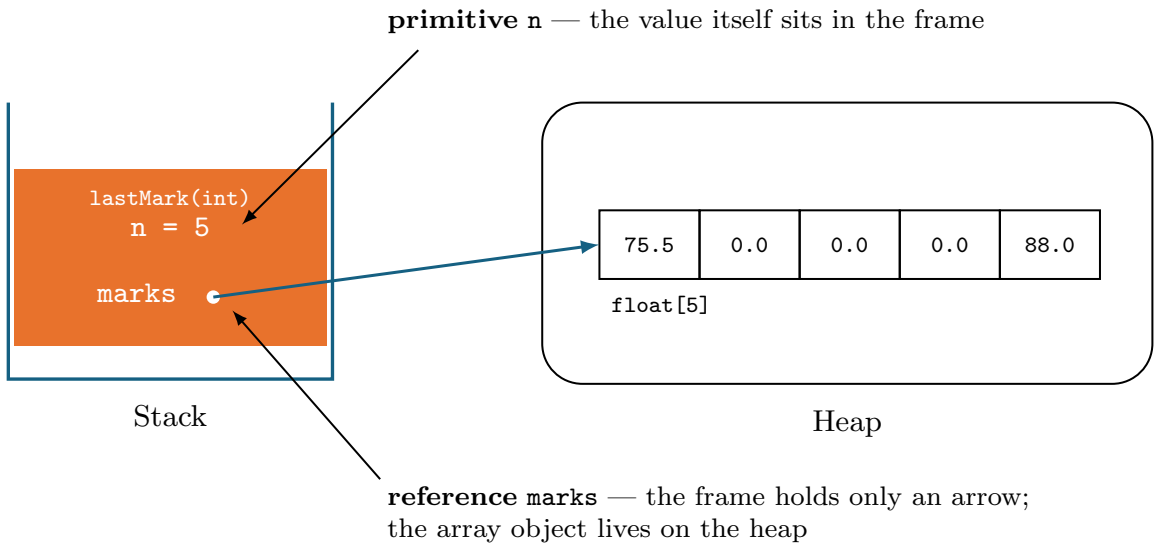


Figure 12: The difference that everything else follows from, in a trace of `lastMark(5)`: the primitive `int n` is stored *as a value* in the stack frame, while the reference `float[] marks` stores only an arrow at an object on the heap. The stack is the open-topped container, the heap is the rounded region, and the reference is the teal arrow.