Java Basics Part 02 - Collections and Strings

Table of contents

Outline 1
Java's memory model 1
Arrays 4
Strings 5
Equality and immutability 7
The Collections framework 8

See csse2002[1] 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[2]. 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[3].

Outline

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

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).

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/2026-02-26-course-overview-and-java-basics.pdf
[3] courses/csse2002/week3-tutorial-stack-heap-and-class-design.pdf

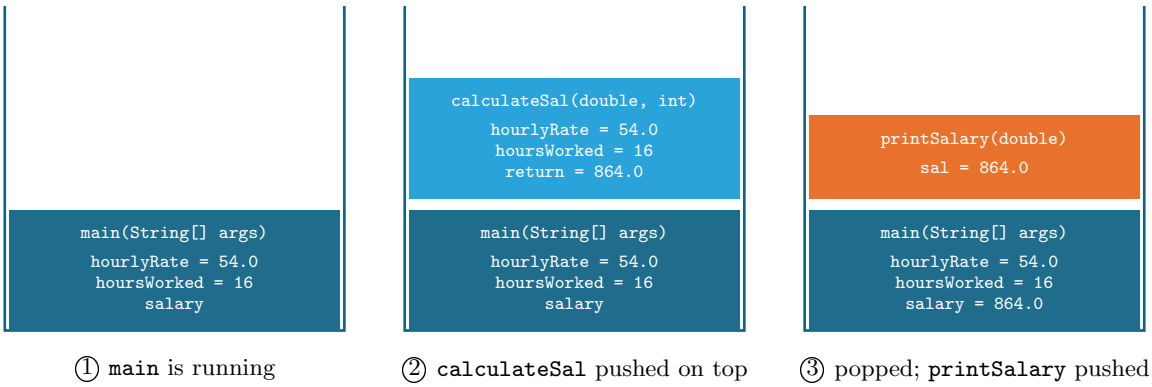
Worked example: stack frames

```java
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:



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

Figure 1: 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.

`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:

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

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

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

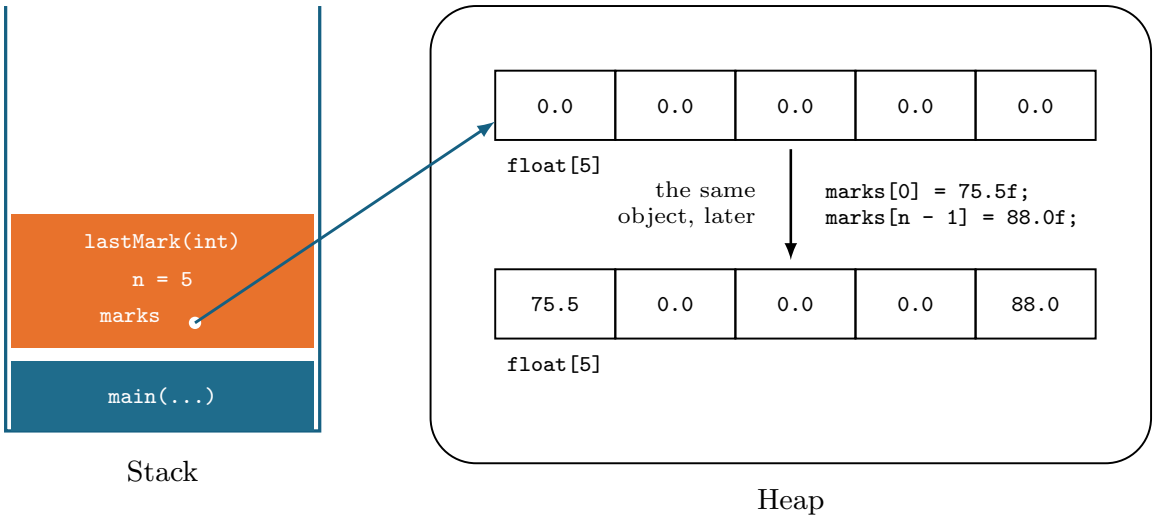


Figure 2: Calling `lastMark(5)`. `n` is a primitive, so its value sits in the frame; `marks` is a reference, so the frame holds only a pointer at a `float[5]` object on the heap. `new float[n]` zero-fills the array, and the two assignments mutate that heap object *through* the reference, leaving `[75.5, 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.

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
```

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:

Strings

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

```
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
```

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

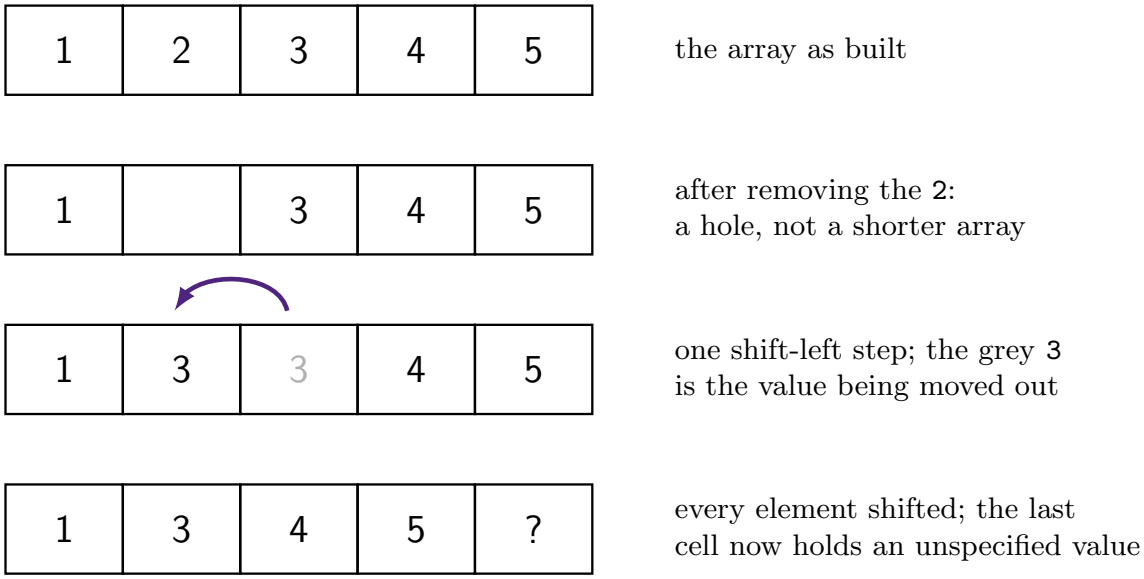


Figure 3: Why arrays motivate collections: deleting an element leaves a hole, and closing it means shifting every later element left by hand — and the array is still the same length, so the trailing cell has to be given some value. The notation is the array convention of csse2002-diagram-notation[4] §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`.

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[5]:

	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

```
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.

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

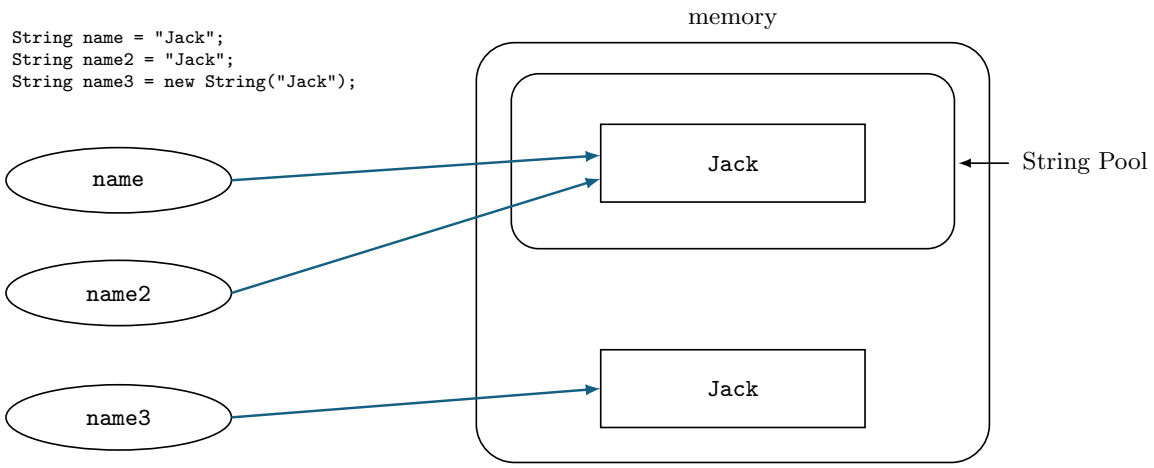


Figure 4: Why `name == name2` is `true` but `name == name3` is `false`. Identical string *literals* are interned once in the string pool and shared; a `new String(...)` call allocates its own object outside the pool. `.equals()` compares contents and so is `true` for all three. Ellipses are variables, rounded regions are memory, and teal arrows are references.

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

```
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[6].

The Collections framework

See java-collections-framework[7] 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):

[6] courses/csse2002/java-mutability.pdf
[7] courses/csse2002/java-collections-framework.pdf

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):

```
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[8]).

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:

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

- 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]
```

> **`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:

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:

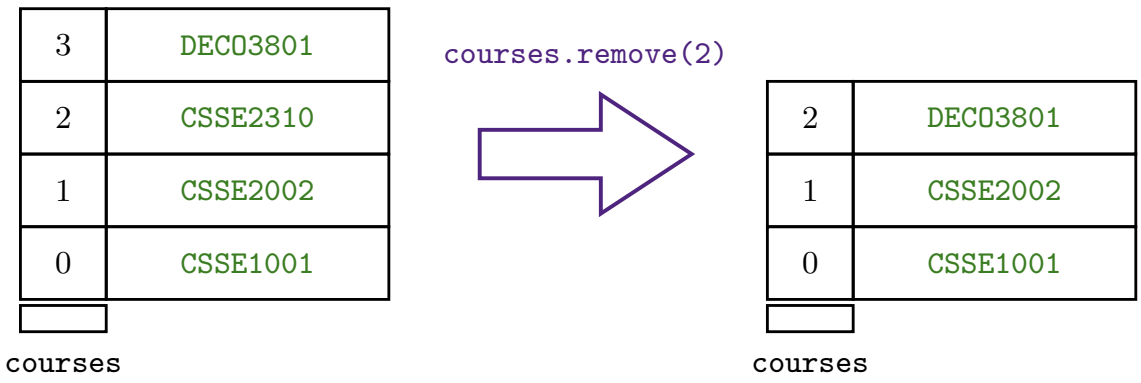


Figure 5: `courses.remove(2)` on the `courses` `ArrayList`, before and after. Everything above the removed index slides down one and the list shrinks — which is exactly the gap-closing the raw array could not do. The list is drawn with **index 0 at the bottom**, growing upward: indices on the left, values in green on the right, and the variable name beneath.

```
Set<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
```