Java Basics Part 02 - Collections and Strings

Table of contents

See csse2002[1] for staff, assessment, and course logistics. This is the **recorded** lecture flagged as a "Tasks this week" item in 2026-02-26-course-overview-and-java-basics[2] — watch/read this alongside the live Lecture 1.

Today's 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 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

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);
    }
}
```

Tracing the stack: `main` pushes a frame holding `hourlyRate = 54.0` and `hoursWorked = 16`. Calling `calculateSal` pushes a new frame on top (with its own `hourlyRate`/`hoursWorked` parameters); it computes `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. Each method call gets its own frame, and frames are popped in the reverse order they were pushed (last in, first out).

Worked example: heap allocation

```
static float lastMark(int n) {
    float[] marks = new float[n];
    marks[0] = 75.5f;
    marks[n - 1] = 88.0f;
    return marks[n - 1];
}
```

Called as `lastMark(5)` from `main`: the array itself (`float[5]`, initially `[0.0, 0.0, 0.0, 0.0, 0.0]`) is allocated **on the heap**. The stack frame for `lastMark` only holds a *reference* (`marks`) pointing at that heap object, plus the parameter `n = 5`. Assigning `marks[0] = 75.5f` and `marks[n-1] = 88.0f` mutates the heap array directly through the reference, giving `[75.5, 0.0, 0.0, 0.0, 88.0]`.

Java never clears heap memory just because a method ends — it's only reclaimed once no references to it exist and the garbage collector decides to run.

Arrays

Recall: an array is an ordered, fixed-length, mutable sequence of homogeneous items.

```
int[] numbers = {1, 2, 3, 4, 5};
numbers.length;         // 5
numbers[0] = 0;         // OK
numbers[0] = "zero";    // illegal -- all elements must be the same type (int)
```

Two ways to create an array:

```
// Approach 1: array literal
float[] marks = {60.3, 62, 70.1, 65.8, 80.3};

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

Querying: `marks[3]` and `marks[0]` are valid, but `marks[marks.length]` throws `ArrayIndexOutOfBoundsException` — valid indices are `0` to `length - 1`.

Iterating: an indexed `for` loop, or an enhanced `for`-each loop:

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

Try at home.

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

Limitations of arrays

Arrays have a **fixed size at creation** (you must know the space you need up-front), and don't automatically close gaps when an element is removed from the middle. This motivates the built-in collections below.

Strings

A `String` is a sequence of characters:

```
String course = "Programming in the Large";
System.out.println(course.length());     // 24
System.out.println(course.charAt(0));     // 'P'
System.out.println(course.charAt(11));
System.out.println(course.charAt(23));    // 'e'
```

`substring(start, end)` returns a substring with an **inclusive** start index and an **exclusive** end index (if `end` is omitted, it defaults to the end of the string):

```
System.out.println(course.substring(0, 11));  // "Programming"
```

Equality and immutability

Equality means something different for primitive types vs reference 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`
`x != y`	check if `x`'s value differs from `y`'s	check if `x` and `y` refer to different objects

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

System.out.println(name == name2);        // true  -- same pooled literal
System.out.println(name == name3);        // false -- different object

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

Object obj1 = new Object();
Object obj2 = new Object();
System.out.println(obj1.equals(obj2));    // false -- Object's default .equals() is
 ↪  identity
```

String objects are immutable. Reassigning `name = "Jill"` doesn't mutate the original `"Jack"` object — it just repoints the `name` reference to a different (or newly pooled) string. String literals with the same value are shared via the **string pool** (`name` and `name2` above both point at the same pooled `"Jack"`); `new String("Jack")` opts out of the pool and allocates a distinct object.

The Collections framework

See java-collections-framework[3] for the full `Stack`/`List`/`Set`/`Map` interface reference — the rest of this section walks through the lecture's worked traces. All of these live in `java.util.*`.

Stack

LIFO (Last In, First Out): `empty()`, `peek()`, `pop()`, `push(obj)`.

```
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 `java.util.Stack`):

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

Collections only store objects, so `Stack<int>` is illegal — Java provides a wrapper class for each primitive type (`Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`; see java-primitive-and-reference-types[4]).

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

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

List

Lists hold items in sequential order like an array, but grow/shrink automatically, have no fixed size limit, support inserting/removing an item anywhere, and are indexed from zero.

`List` is an **interface**, not a particular implementation — you can declare a variable as `List`, but can't do `new List()`. Popular implementations: `ArrayList` (better for random access) and `LinkedList` (better for modifying the middle of the list).

```
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/returns "CSSE2310" -> [CSSE1001,
 ↪  CSSE2002, DECO3801]
```

Set

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

```
Set<String> farm = new HashSet<>();
farm.add("Fox");       // true
farm.add("Farmer");    // true
farm.add("Chicken");   // true
farm.add("Fox");       // false -- already present
farm.add("Grain");     // true
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, like a Python dict.

```
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);                      // overwrites Fox's value
```

```
farm.put("Grain", 15);
farm.put("Grain", farm.get("Grain") + 5);    // Grain -> 20
// final map: {Fox: 18, Farmer: 10, Chicken: 0, Grain: 20}
```

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

The hashCode/equals contract

For `HashSet`/`HashMap` to behave correctly, elements/keys need:

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, e.g. `"Thilina".hashCode()` $\rightarrow$ `318621125`.