Java Collections Framework

Table of contents

Why collections instead of arrays? 1
Stack — LIFO 2
List 2
Set 3
Map 4
The hashCode/equals contract 4

Introduced in 2026-02-26-java-basics-part-02-collections-and-strings[1] (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[2] for the primitive wrapper classes (`Integer`, `Double`, etc.) used to store primitives inside them.

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

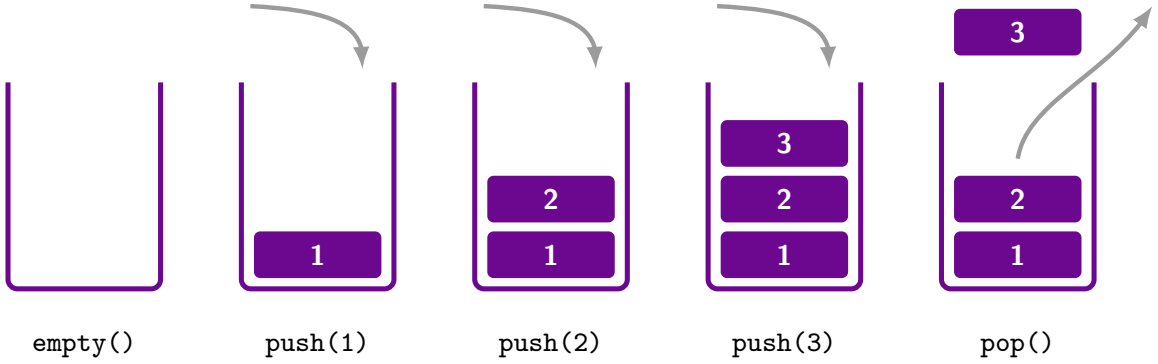

Figure 1: Last In, First Out: `push` always adds at the top and `pop` always removes from the top, so the last value in is the first value out. The stack is an open-topped container whose entries grow upward. An arrow *into* the container is a push and an arrow *out and away* is a pop. Each frame is 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:

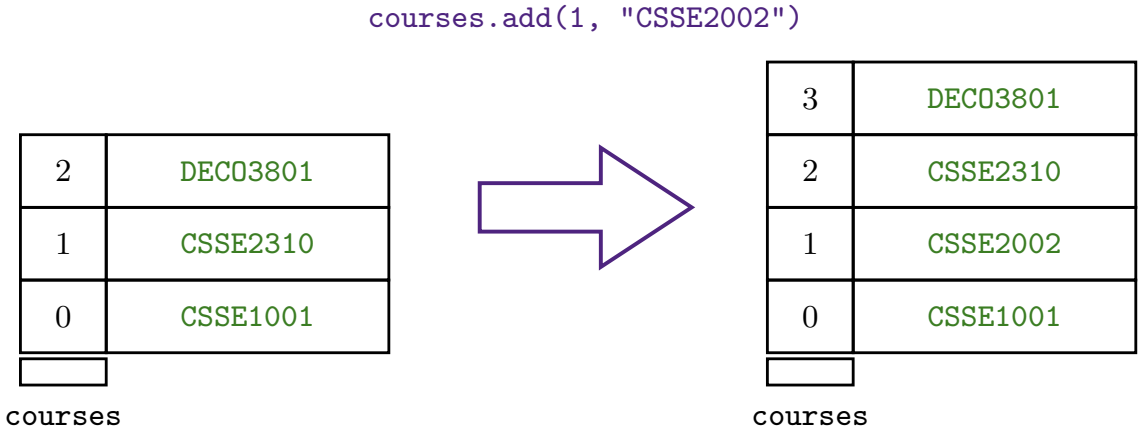

Figure 2: `add(index, e)` on a `List`: `e` lands *at* that index and every entry from there up renumbers, so the list gets longer without you writing a single shift. (`remove(index)` is the mirror image — see the removal step traced in 2026-02-26-java-basics-part-02-collections-and-strings[3].) The list 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.

[3] 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`.