Java Collections Framework

Table of contents

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?

Arrays have a fixed size at creation and don't automatically close gaps when an element is removed from the middle. Collections solve both problems, 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.

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

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

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

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

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.

Set

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

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

- **`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>`.

```
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();
}
```

- **`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.