Collections in Java

Java Collections Framework in simple terms

List, Set, Map and Deque: how to store data and choose the right collection

1. What is a collection?

A **collection** allows us to store several objects together and perform common operations on them.
For example, a list of names:

```
List<String> names = new ArrayList<>();

names.add("Anna");
names.add("Bohdan");
names.add("Denys");
```

Now all the names are stored in one object called `names`.
With collections, we can:

- add elements;
- get elements;
- search for elements;
- remove elements;
- go through all elements.

The type `String` inside the angle brackets

```
List<String>
```

means that this list stores strings.

Main idea

A collection is useful when we want to work with a group of objects instead of just one object.

2. Main types of collections

Java has several important collection interfaces.

List
Stores elements in order. Duplicates are allowed.
Example implementation:

```
ArrayList
```

Set
Stores unique elements only.
Example implementation:

```
HashSet
```

Map
Stores pairs:

key ⟶ value.

Example implementation:

```
HashMap
```

Deque

Useful for queues and stacks.
Example implementation:

```
ArrayDeque
```

Important

`Map` is slightly different from the other collection types.
It does not extend `Collection` because it stores pairs of keys and values rather than individual elements.

3. List and ArrayList

A **List** is an ordered list of elements.
It:

- keeps the order of elements;
- allows access by index;
- allows duplicate elements.

Example:

```
List<String> names = new ArrayList<>();

names.add("Anna");
names.add("Bohdan");
names.add("Anna");
```

The list now contains three elements.
Get an element by index:

```
String first = names.get(0);
```

The first element has index `0`.
Change an element:

```
names.set(1, "Denys");
```

Remove an element:

```
names.remove(0);
```

Find the number of elements:

```
int size = names.size();
```

Important

For most ordinary lists, `ArrayList` is a good first choice.

4. Set and HashSet

A **Set** is used when elements must be unique.
Example:

```
Set<String> languages = new HashSet<>();

languages.add("Java");
languages.add("Python");
languages.add("Java");
```

Although `"Java"` was added twice, it will be stored only once.
Check whether an element exists:

```
boolean exists =
    languages.contains("Java");
```

Remove an element:

```
languages.remove("Python");
```

It is important to remember:
`HashSet` does not guarantee the order of its elements.
If the insertion order must be preserved, we can use:

```
LinkedHashSet
```

If the elements should be kept sorted:

```
TreeSet
```

5. Map and HashMap

A **Map** stores pairs:

$$\text{key} \longrightarrow \text{value}.$$

For example, we can connect a student's name with their score:

```
Map<String, Integer> scores =
    new HashMap<>();

scores.put("Anna", 10);
scores.put("Bohdan", 8);
scores.put("Denys", 9);
```

Get a value by its key:

```
int annaScore = scores.get("Anna");
```

Check whether a key exists:

```
boolean exists =
    scores.containsKey("Bohdan");
```

Remove a key-value pair:

```
scores.remove("Denys");
```

Keys in a `Map` are unique.
If we put another value using the same key:

```
scores.put("Anna", 12);
```

the old value `10` will be replaced by `12`.

Important

`HashMap` is useful when we need to quickly find a value by its key.

6. Queue, Deque and ArrayDeque

A **queue** works according to the rule:

first in ⟶ first out.

This is called **FIFO**:

First In, First Out.

In Java, `ArrayDeque` is a convenient choice for an ordinary queue.

```
Deque<String> tasks =
    new ArrayDeque<>();

tasks.offerLast("Task A");
tasks.offerLast("Task B");

String next = tasks.pollFirst();
```

The variable `next` will contain:

```
"Task A"
```

Look at the first element without removing it:

```
String first = tasks.peekFirst();
```

A `Deque` can also be used as a stack.
A stack works according to the rule:

last in ⟶ first out.

This is called **LIFO**:

Last In, First Out.

```
Deque<String> stack =
    new ArrayDeque<>();

stack.push("A");
stack.push("B");

String top = stack.pop();
```

Here, the element `"B"` will be removed.

7. How to choose a collection quickly

For most learning tasks, the following rules are enough.

- Need an ordinary list with indices:

```
ArrayList
```

- Need unique elements only:

```
HashSet
```

- Need to find a value by a key:

```
HashMap
```

- Need unique elements while preserving insertion order:

```
LinkedHashSet
```

- Need key-value pairs while preserving insertion order:

```
LinkedHashMap
```

- Elements should automatically stay sorted:

```
TreeSet
```

- Map keys should stay sorted:

```
TreeMap
```

- Need a queue or a stack:

```
ArrayDeque
```

8. Iterating over elements

One of the simplest ways to go through all elements of a list is a `for-each` loop.

```
for (String name : names) {
    System.out.println(name);
}
```

We can iterate over a `Set` in the same way:

```
for (String language : languages) {
    System.out.println(language);
}
```

For a `Map`, we can go through all key-value pairs:

```
for (Map.Entry<String, Integer> entry
        : scores.entrySet()) {

    System.out.println(
        entry.getKey()
        + ": "
        + entry.getValue()
    );
}
```

Here:

- `entry.getKey()` returns the key;
- `entry.getValue()` returns the value.

9. Sorting

An ordinary list can be sorted.
For example:

```
List<String> names =
    new ArrayList<>();

names.add("Denys");
names.add("Anna");
names.add("Bohdan");

Collections.sort(names);
```

After sorting, the elements will be in alphabetical order:

```
Anna
Bohdan
Denys
```

Numbers are naturally sorted from smaller to larger values.
If we want unique elements to remain automatically sorted, we can use `TreeSet`:

```
Set<Integer> numbers =
    new TreeSet<>();

numbers.add(5);
numbers.add(2);
numbers.add(8);
```

The elements will be ordered as:

```
2, 5, 8
```

10. Speed of common operations

When choosing a collection, it is useful to understand how quickly common operations usually work.
Notation:

- $O(1)$ — the operation usually takes approximately constant time;
- $O(\log n)$ — the time grows slowly as the number of elements increases;

- $O(n)$ — in the worst case, many or all elements may need to be checked.

Collection	Operation	Usually
`ArrayList`	access by index	$O(1)$
`ArrayList`	search for an element	$O(n)$
`HashSet`	search / add	$O(1)^*$
`HashMap`	search / add by key	$O(1)^*$
`TreeSet`	search / add	$O(\log n)$
`TreeMap`	search / add	$O(\log n)$
`ArrayDeque`	add / remove at the ends	$O(1)$

* On average. In some cases, the operation may take longer.

Important

For your first programs, do not choose a collection only because of its time-complexity formulas.
First think about **what you actually need to do with the data**.

11. equals() and hashCode()

For `HashSet` and the keys of `HashMap`, it is important to define correctly when two objects are considered equal.
Java uses these methods:

```
equals()
hashCode()
```

In simple terms:
equals()
answers the question:

"Are these objects considered equal?"

hashCode()
returns a special number that helps hash-based collections find an object more quickly.
The main rule is:

Important

If two objects are equal according to `equals()`, they must have the same `hashCode()`.

For standard Java types such as

```
String
Integer
Long
```

these methods are already implemented correctly.
When you start creating your own classes, this topic can be studied in more detail.

12. Unmodifiable collections

Sometimes we want to create a small collection that cannot be changed after it is created.
For example:

```
List<String> colors =
    List.of("red", "green", "blue");
```

We can read from this list:

```
String first = colors.get(0);
```

But we cannot add a new element:

```
colors.add("yellow"); // Error
```

For a map:

```
Map<String, Integer> scores =
    Map.of(
        "Anna", 10,
        "Bohdan", 8
    );
```

Such collections are useful when the set of data should remain unchanged.

13. Common mistakes

Mistake 1.
Expecting a particular order of elements in a `HashSet` or `HashMap`.
Their order is not guaranteed.

Mistake 2.
Forgetting that list indices start at zero.
For a list containing three elements, the valid indices are:

$$0, \quad 1, \quad 2.$$

Mistake 3.
Expecting a `Set` to keep duplicates.

```
Set<String> set = new HashSet<>();

set.add("Java");
set.add("Java");
```

Only one element will remain in the set.

Mistake 4.
Forgetting that putting a new value into a `Map` using the same key replaces the previous value.

```
scores.put("Anna", 10);
scores.put("Anna", 12);
```

The value associated with `"Anna"` is now `12`.

Mistake 5.
Removing elements directly from a collection inside a `for-each` loop.
For simple removal based on a condition, it is often easier to use:

```
names.removeIf(String::isBlank);
```

14. Quick reference

ArrayList
An ordinary ordered list. It has indices and allows duplicates.
HashSet
A set of unique elements. Order is not guaranteed.
HashMap
Stores key-value pairs. Each key has one value.
LinkedHashSet
Unique elements while preserving insertion order.
LinkedHashMap
Key-value pairs while preserving insertion order.
TreeSet
Unique elements kept in sorted order.
TreeMap
Key-value pairs with sorted keys.
ArrayDeque
A convenient implementation for a queue or a stack.

Main rule for choosing a collection

First ask yourself three questions.

1. Do I need the elements to have a particular order?
2. Can elements be repeated?
3. How do I want to find the data: by index, by value or by key?

After that, the choice is usually simple:

- ordinary list — `ArrayList`;
- unique elements — `HashSet`;
- key and value — `HashMap`;
- queue or stack — `ArrayDeque`.

You do not need to choose the most complicated collection.
Choose the collection that best matches the task.

This reference sheet is intended as a first introduction to Java collections. The examples are intentionally simple to explain the main ideas and the most common operations.