Java Collections Framework

Advanced Java Programming

Masoud Hamad

State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

What is the Collections Framework?

Definition

A unified architecture for representing and manipulating groups of objects.

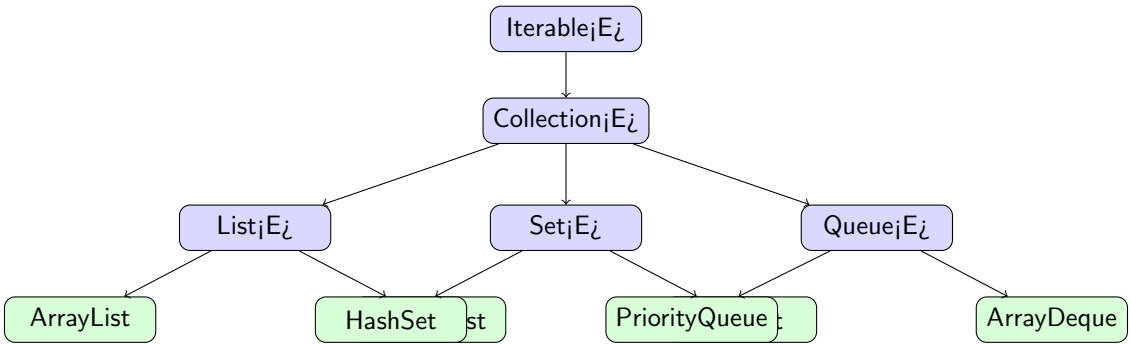
Why not just use arrays?

- Fixed size — cannot grow/shrink
- No built-in search, sort, insert
- No type safety (before generics)
- Manual memory management

Collections provide:

- Dynamic sizing
- Built-in algorithms
- Type safety with generics
- Rich API (add, remove, search, sort)
- Thread-safe variants

Collections Hierarchy



Blue = Interface, Green = Implementation class

ArrayList — Dynamic Array

```
import java.util.ArrayList;
import java.util.Collections;

ArrayList<String> names = new ArrayList<>();
names.add("Ali");                       // Add to end
names.add(0, "Fatma");                  // Add at index 0
names.set(1, "Hassan");                 // Replace at index 1
String first = names.get(0);            // Access by index
names.remove("Hassan");                 // Remove by value
names.remove(0);                        // Remove by index
int size = names.size();                // Get size
boolean has = names.contains("Ali");// Check existence

// Sort and shuffle
Collections.sort(names);
Collections.shuffle(names);

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

ArrayList vs LinkedList

Operation	ArrayList	LinkedList
Access by index	$O(1)$	$O(n)$
Add at end	$O(1)$ amortized	$O(1)$
Add at beginning	$O(n)$	$O(1)$
Remove from middle	$O(n)$	$O(1)$ if at node
Memory	Compact	Extra pointers
Cache performance	Good	Poor

When to use what?

- **ArrayList**: Most cases — random access, iteration
- **LinkedList**: Frequent insertions/deletions at both ends (use as Deque)

HashSet and TreeSet

```java
// HashSet: unordered, no duplicates
HashSet<String> set = new HashSet<>();
set.add("Java");
set.add("Python");
set.add("Java");  // ignored!
// Size = 2

// Fast: O(1) add/remove/contains
System.out.println(
    set.contains("Java")); // true
```

```java
// TreeSet: sorted, no duplicates
TreeSet<Integer> sorted =
    new TreeSet<>();
sorted.add(30);
sorted.add(10);
sorted.add(20);
// Prints: [10, 20, 30]

// Extra methods:
sorted.first();   // 10
sorted.last();    // 30
sorted.headSet(20); // [10]
```

Important

Objects in `HashSet` must override `equals()` and `hashCode()`.
Objects in `TreeSet` must implement `Comparable` or provide a `Comparator`.

HashMap — Key-Value Pairs

```
import java.util.HashMap;
import java.util.Map;

HashMap<String, Integer> scores = new HashMap<>();
scores.put("Ali", 85);
scores.put("Fatma", 92);
scores.put("Hassan", 78);

int aliScore = scores.get("Ali");            // 85
scores.getOrDefault("Unknown", 0);           // 0
scores.putIfAbsent("Ali", 100);              // keeps 85
scores.replace("Hassan", 80);                // update to 80
scores.remove("Fatma");                      // delete entry

// Iterate over entries
for (Map.Entry<String, Integer> entry : scores.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}

// Iterate keys only
for (String key : scores.keySet()) { ... }
// Iterate values only
for (int val : scores.values()) { ... }
```

HashMap vs TreeMap vs LinkedHashMap

Property	HashMap	TreeMap	LinkedHashMap
Ordering	None	Sorted by key	Insertion order
get/put	$O(1)$	$O(\log n)$	$O(1)$
Null keys	1 allowed	Not allowed	1 allowed
Implementation	Hash table	Red-Black tree	Hash + linked list

Use Cases

- **HashMap**: General-purpose mapping, frequency counting
- **TreeMap**: When you need sorted keys (e.g., alphabetical listing)
- **LinkedHashMap**: When insertion order matters (e.g., LRU cache)

Queue and PriorityQueue

```
// Queue interface (FIFO)
Queue<String> queue =
    new LinkedList<>();
queue.offer("First");
queue.offer("Second");
queue.offer("Third");

queue.peek();    // "First" (no remove)
queue.poll();    // "First" (remove)
queue.size();    // 2
```

```
// PriorityQueue (min-heap)
PriorityQueue<Integer> pq =
    new PriorityQueue<>();
pq.offer(30);
pq.offer(10);
pq.offer(20);

pq.poll();  // 10 (smallest first)
pq.poll();  // 20
pq.poll();  // 30

// Max-heap:
PriorityQueue<Integer> maxPQ =
  new PriorityQueue<>(
    Collections.reverseOrder());
```

Iterator Pattern and for-each

```
List<String> names = Arrays.asList("Ali", "Fatma", "Hassan", "Amina");

// 1. Enhanced for-each loop (preferred)
for (String name : names) {
    System.out.println(name);
}

// 2. Iterator (needed for safe removal during iteration)
Iterator<String> it = names.iterator();
while (it.hasNext()) {
    String name = it.next();
    if (name.startsWith("A")) {
        it.remove();  // Safe removal! Don't use list.remove() here
    }
}

// 3. ListIterator (bidirectional, add/set support)
ListIterator<String> lit = names.listIterator();
while (lit.hasNext()) {
    String name = lit.next();
    lit.set(name.toUpperCase());  // Modify in place
}
```

Collections Utility Class

```
List<Integer> nums = new ArrayList<>(Arrays.asList(5, 2, 8, 1, 9, 3));

Collections.sort(nums);                          // [1, 2, 3, 5, 8, 9]
Collections.sort(nums, Collections.reverseOrder()); // [9, 8, 5, 3, 2, 1]
Collections.shuffle(nums);                       // Random order
Collections.reverse(nums);                       // Reverse current order
int max = Collections.max(nums);                 // Maximum element
int min = Collections.min(nums);                 // Minimum element
int freq = Collections.frequency(nums, 5);       // Count of 5
Collections.swap(nums, 0, 3);                   // Swap indices 0 and 3

// Binary search (list must be sorted first!)
Collections.sort(nums);
int idx = Collections.binarySearch(nums, 5); // Returns index

// Unmodifiable view
List<Integer> readOnly = Collections.unmodifiableList(nums);
// readOnly.add(10); --> throws UnsupportedOperationException
```

Comparable vs Comparator

```
// Comparable: natural ordering
class Student implements
    Comparable<Student> {
    String name;
    double gpa;

    @Override
    public int compareTo(Student o) {
        return this.name.compareTo(
            o.name); // by name
    }
}

// Usage:
Collections.sort(students);
```

```
// Comparator: custom ordering
Comparator<Student> byGPA =
    new Comparator<Student>() {
        public int compare(
            Student a, Student b) {
            return Double.compare(
                b.gpa, a.gpa); // desc
        }
    };

// Lambda version:
Comparator<Student> byGPA =
    (a, b) -> Double.compare(
        b.gpa, a.gpa);

Collections.sort(students, byGPA);
```

Collections Summary

Need	Interface	Best Choice	Alternative
Ordered list	List	ArrayList	LinkedList
Unique elements	Set	HashSet	TreeSet
Key-value pairs	Map	HashMap	TreeMap
FIFO queue	Queue	LinkedList	ArrayDeque
Priority queue	Queue	PriorityQueue	—
Stack (LIFO)	Deque	ArrayDeque	—
Sorted elements	SortedSet	TreeSet	—
Sorted keys	SortedMap	TreeMap	—

Key Takeaways

- Program to the **interface**, not the implementation
- Override `equals()`/`hashCode()` for objects in Hash-based collections
- Use `Iterator` for safe removal during iteration