Java Collections Framework — Complete Notes (Up to List)

Prepared for: Akhila Vk

This document explains Java Collections up to the List interface in a single, neat PDF. All code examples use Style-B formatting (clear comments inside code) and outputs are shown after each example.

1. What is a Collection?

1. A Collection is a group of objects stored together as a single unit.

2. Collections allow storing, retrieving, and manipulating multiple objects efficiently.

3. Collections were introduced to overcome limitations of arrays in Java.

4. Collections grow and shrink dynamically based on data.

5. Collections are part of the java.util package.

2. Why Collections? (Reasons)

1. Arrays are fixed in size; Collections are dynamic and resize automatically.

2. Arrays lack built-in operations like add(), remove(), sort(); Collections provide these methods.

3. Collections offer ready-made data structures such as List, Set, Queue, and Map.

4. Collections reduce boilerplate code and simplify complex data handling.

5. Collections provide optimized internal algorithms for better performance.

3. Important Characteristics of Collections

1. Collections grow and shrink automatically as elements are added or removed.

2. Collections store objects (boxed types for primitives).

3. Collections provide searching, sorting and manipulation utilities via Collections and Comparator.

4. Collections use Generics for compile-time type safety.

5. Collections provide Iterators for consistent element traversal.

6. Collections form a structured hierarchy of interfaces and implementations.

4. Collection Framework Architecture (Simplified)

1. Interfaces define behavior: Collection, List, Set, Queue, Map (Map is separate).

2. Implementations provide concrete behavior: ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue, HashMap.

3. Algorithms are provided by the Collections utility class: sort(), reverse(), shuffle(), binarySearch().

4. Iterators and ListIterator standardize traversal and modification during traversal.

5. Collection vs Collections

1. Collection is an interface representing a group of objects and core operations like add(), remove().

2. Collections is a utility class that contains static methods to operate on or return collections (sort, synchronized wrappers).

3. Use Collections when you need algorithms or thread-safe wrappers for existing collections.

6. Main Interfaces in the Framework

1. List — ordered collection that allows duplicates and indexed access.

2. Set — collection that prevents duplicates (HashSet, TreeSet).

3. Queue — collection for holding elements prior to processing (FIFO behavior).

4. Map — stores key-value pairs; keys are unique (HashMap, TreeMap).

7. What is List?

1. List is an ordered collection that allows duplicate elements and maintains insertion order.

2. List supports positional (index-based) access and manipulation.

3. List provides ListIterator for bi-directional traversal and modification at iterator position.

4. List implementations include ArrayList, LinkedList, Vector, and CopyOnWriteArrayList.

8. Core List Operations (Short)

1. add(element) — appends element to list.

2. add(index, element) — inserts element at position index.

3. get(index) — returns element at index.

4. set(index, element) — replaces element at index.

5. remove(index) or remove(object) — deletes element.

6. indexOf(object) and lastIndexOf(object) — find positions.

7. size(), isEmpty(), clear(), subList(from, to) — utility operations.

9. ArrayList — Detailed Explanation

1. ArrayList is backed by a dynamic array and provides fast random access (get(index) is O(1)).

2. ArrayList grows automatically (internal array resizing) and shifting occurs on insert/remove at middle positions.

3. ArrayList allows duplicates and null values and is not synchronized by default.

Example — ArrayList (Style-B)

```
import java.util.*;

public class ArrayListDemo {
    public static void main(String[] args) {

        // Creating an ArrayList
        ArrayList list = new ArrayList<>();

        // Adding elements
        list.add("Akhila");
        list.add("Neha");
        list.add("Akhila");   // duplicate allowed

        // Printing the list
        System.out.println("List = " + list);

        // Accessing index element
        System.out.println("Element at index 1 = " + list.get(1));
    }
}
```

Output: List = [Akhila, Neha, Akhila] Element at index 1 = Neha

10. LinkedList — Detailed Explanation

1. LinkedList uses a doubly linked node structure and is efficient for frequent insertions and deletions.

2. LinkedList implements List and Deque, allowing addFirst, addLast, removeFirst, removeLast methods.

3. LinkedList has O(n) access time for get(index) because of traversal.

Example — LinkedList (Style-B)

```
import java.util.*;

public class LinkedListDemo {
    public static void main(String[] args) {

        // Creating LinkedList
        LinkedList list = new LinkedList<>();

        // Adding values
        list.add("Start");
        list.add("Middle");
        list.addFirst("Beginning");
        list.addLast("End");

        // Printing the list
        System.out.println("List = " + list);

        // Removing first and last elements
        list.removeFirst();
        list.removeLast();

        // Printing after removal
        System.out.println("After removal: " + list);
    }
}
```

Output: List = [Beginning, Start, Middle, End] After removal: [Start, Middle]

11. Vector — Detailed Explanation

1. Vector is a legacy, synchronized list implementation suitable for thread-safe scenarios but slower due to locking.

2. Prefer Collections.synchronizedList(new ArrayList<>()) or concurrent collections instead of Vector in new code.

Example — Vector (Style-B)

```
import java.util.*;

public class VectorDemo {
    public static void main(String[] args) {

        // Creating Vector
        Vector v = new Vector<>();

        // Adding elements
        v.add(10);
        v.add(20);
        v.add(30);

        // Printing vector
        System.out.println("Vector = " + v);
    }
}
```

Output: Vector = [10, 20, 30]

12. CopyOnWriteArrayList — Detailed Explanation

1. CopyOnWriteArrayList from java.util.concurrent creates a fresh array copy on each write operation, giving safe iteration without ConcurrentModificationException.

2. Best for scenarios with many readers and few writers (listener lists, event handlers).

Example — CopyOnWriteArrayList (Style-B)

```
import java.util.concurrent.CopyOnWriteArrayList;

public class COWListDemo {
    public static void main(String[] args) {

        // Creating CopyOnWriteArrayList
        CopyOnWriteArrayList list = new CopyOnWriteArrayList<>();

        // Adding initial value
        list.add("Java");

        // Adding while iterating is safe
        for (String s : list) {
            list.add("NewItem");
        }

        // Printing final list
        System.out.println("List = " + list);
    }
}
```

Output: List = [Java, NewItem]

13. Differences Between List Implementations

1. ArrayList is faster for random access; LinkedList is faster for frequent insertion and deletion.

2. ArrayList uses less memory per element; LinkedList uses extra memory for node pointers.

3. Vector is synchronized and slower; use concurrent collections for thread-safe operations.

14. Important Notes and Pitfalls

1. Iterators are fail-fast and will throw ConcurrentModificationException if collection is structurally modified outside iterator methods.

2. Arrays.asList(...) returns a fixed-size list backed by the array; modifications to size will throw UnsupportedOperationException.

3. subList(from, to) returns a view backed by the parent list; concurrent structural changes can cause issues.

4. equals() and hashCode() must be implemented correctly for objects stored in collections when logical equality matters.

15. Interview Questions (Till List)

1. What is the Java Collections Framework?

2. Why use Collections instead of arrays?

3. What is the difference between Collection and Collections?

4. What is List and when to use it?

5. Compare ArrayList and LinkedList and give use-cases.

6. What is fail-fast and how to avoid ConcurrentModificationException?

7. When to use CopyOnWriteArrayList?

End of Document