Java Collections

Excellent👍 — let's go through **Collections in Java from Scratch — with explanations and examples.**

🔹 What is a Collection in Java?

- **Collection** refers to a group of **objects**.
- It's a framework that **provides a set of classes and interfaces** to efficiently manage groups of related elements.
- All classes related to collections are in **java.util** package.

🔹 Why do we need Collections?

✅ Store multiple elements in a single object
✅ Easily add, delete, search, or modify elements
✅ Provide convenient methods for iteration, sorting, and other operations
✅ Flexible and dynamic in size (unlike arrays)

🔹 Java Collections Framework (Hierarchy)

```
typescript

java.util.Collection (interface)
 └ List
 └ Set
 └ Queue
 └ Deque
 └ Map (not a Collection but related)
```

🔹 List — Stores elements in a sequence (with duplicates)

✅ Implementations:

- **ArrayList** — backed by a dynamic array
- **LinkedList** — doubly-linked list
- **Vector** (Thread-safe, older)

```
java

// ArrayList example
import java.util.*;
class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
```

```
        names.add("Bob");

        names.add("Charlie");

        System.out.println(names);
        // Prints [Alice, Bob, Charlie]
    }
}
```

🔹 Set — Stores unique elements, unordered or ordered:

✅ Implementations:

- **HashSet** — no ordering, faster
- **LinkedHashSet** — maintains insertion order
- **TreeSet** — sorts elements (ascending)

```
java

// HashSet example
Set<String> set = new HashSet<>();
set.add("apple");

set.add("apple"); // duplicates are not allowed
set.add("banana");

System.out.println(set);
// Prints [apple, banana] in unspecified order
```

🔹 Queue — Stores elements in FIFO (First In, First Out):

✅ Implementations:

- **LinkedList** — a doubly-linked list used as a queue
- **PriorityQueue** — a priority-based queue

```
java

// Queue example
Queue<String> queue = new LinkedList<>();
queue.add("apple");

queue.add("banana");

String first = queue.poll();

System.out.println(first); // Prints "apple"

queue.remove("banana");

System.out.println(queue.isEmpty()); // Prints true
```

🔹 Deque — Stores elements at both ends:

✅ Implementations:

- **ArrayDeque** — resizable array
- **LinkedList** — doubly-linked list

```
java

// Deque example
Deque<String> deque = new ArrayDeque<>();
deque.addFirst("apple");

deque.addLast("banana");

String first = deque.pollFirst();

String last = deque.pollLast();

System.out.println(first);
System.out.println(last);
```

🔹 Map — Stores key-value pairs:

✅ Implementations:

- **HashMap** — unordered
- **LinkedHashMap** — maintains insertion order
- **TreeMap** — sorts by keys
- **Hashtable** — thread-safe (but legacy)

```
java

// HashMap example
Map<String, Integer> map = new HashMap<>();
map.put("apple", 1);
map.put("banana", 2);

for (String key : map.keySet()) {
    System.out.println(key + " -> " + map.get(key));
}
```

🔹 Common Operations (examples)

```
java

// adding elements
list.add("item");

// removing elements
list.remove("item");

set.remove("item");

queue.poll();

// accessing elements
list.get(0);
```

```
set.contains("item");

queue.peek();

// iteration
for (String item : list) {
    System.out.println(item);
}

iterator, forEach, streams...
```

🔹 Summary:

✅ **List** — duplicates allowed, maintains order.
✅ **Set** — duplicates prohibited, unordered or ordered.
✅ **Queue** — FIFO structure.
✅ **Deque** — double-ended.
✅ **Map** — key-value pairs.

✨ If you'd like, I can go into **more depth**, with:

✅ Performance differences
✅ Concurrent collections
✅ How to choose which to use
✅ Implementing custom collections
✅ Stream API with collections
✅ And much more...

✨ Just let me know by sending **"MORE"**! 🌟

Excellent — let's go **deeper with Java Collections!** 🌟

🔹 List — Detailed View

✅ **ArrayList**:

- **Internally** a resizable array.
- **Best for** accessing by index and adding at the end.
- **Slower** for adding or removing from the middle.

```
java

List<String> arrayList = new ArrayList<>();
arrayList.add("apple"); // O(1) amortized
arrayList.add("banana");

String item = arrayList.get(0);
arrayList.remove(0);
```

✅ **LinkedList**:

- **Internally** a doubly-linked list.
- **Best for** adding or removing from the start or middle.
- **Slower** for accessing by index.

```
java

List<String> linkedList = new LinkedList<>();
linkedList.addFirst("apple");

linkedList.addLast("banana");

String first = linkedList.removeFirst();
String last = linkedList.removeLast();
```

🔹 Set — Detailed View

✅ **HashSet**:

- Stores elements in a hash table.
- **Offers O(1)** for add, remove, contains.
- **Not ordered**.

```
java

Set<String> hashSet = new HashSet<>();
hashSet.add("apple");

hashSet.add("banana");

if (hashSet.contains("apple")) {
```

```
    System.out.println("apple exists.");
}
```

✅ **LinkedHashSet**:

- Stores elements in a **HashTable + Linked List**.
- **Maintains insertion order**.

```
java

Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("apple");

linkedHashSet.add("banana");

for (String item : linkedHashSet) {
    System.out.println(item);
}
```

✅ **TreeSet**:

- Stores elements in a **Red-Black Tree**.
- **Maintains a naturally ascending order or a custom Comparator**.

```
java

Set<String> treeSet = new TreeSet<>();
treeSet.add("apple");

treeSet.add("banana");

for (String item : treeSet) {
    System.out.println(item);
}
```

🔹 Queue — Detailed View

✅ **PriorityQueue**:

- Stores elements in **min-heap or max-heap**.
- The **poll** always removes the **lowest or highest priority** first.

```
java

Queue<Integer> priorityQueue = new PriorityQueue<>();
priorityQueue.add(5);
priorityQueue.add(1);
priorityQueue.add(10);

System.out.println(priorityQueue.poll()); // Prints 1
```

🔹 Deque — Detailed View (Double-Ended)

✅ **ArrayDeque**:

- Implemented with **resizable array**.
- Allows adding or removing from both ends efficiently.

```
java

Deque<String> arrayDeque = new ArrayDeque<>();
arrayDeque.addFirst("apple");

arrayDeque.addLast("banana");

String first = arrayDeque.pollFirst();
String last = arrayDeque.pollLast();
```

🔹 Concurrent Collections (Thread-safe)

✅ **ConcurrentHashMap** — thread-safe map with high concurrency
✅ **CopyOnWriteArrayList** — thread-safe List with snapshot semantics
✅ **CopyOnWriteArraySet** — thread-safe Set with snapshot semantics
✅ **ConcurrentLinkedQueue** — thread-safe non-blocking FIFO
✅ **ConcurrentSkipListSet/Map** — thread-safe, sorted set or map

---

🔹 How to Choose?

✅ **ArrayList** — if you need index-based, fast, or frequently accessing by index.
✅ **LinkedList** — if you need frequently adding or removing at both ends or in the middle.
✅ **HashSet** — if you want fast set operations with no duplicates.
✅ **LinkedHashSet** — if you want to maintain insertion order while avoiding duplicates.
✅ **TreeSet** — if you want a set that's kept in a sorted order.
✅ **HashMap** — if you need fast key-value lookup with no ordering.
✅ **LinkedHashMap** — if you need fast lookup with insertion order preserved.
✅ **TreeMap** — if you need a map that's kept in a sorted order by keys.
✅ **ArrayDeque** — if you want a stack or a queue with fast operations at both ends.
✅ **PriorityQueue** — if you want to process elements by their priority.
✅ **ConcurrentHashMap**, **ConcurrentSkipListSet**, etc. — for multithreaded applications with high concurrency.

🔹 Stream API with Collections (Bonus)

```
java

List<String> names = List.of("apple", "banana", "cherry");

List<String> bNames = names.stream()
    .filter(s -> s.startsWith("b"))
    .toList();
```

```
bNames.forEach(System.out::println);
```

✨ If you'd like, I can go into **even more depth**:

✅ How the **`Iterator`** works
✅ How **`Comparator`** and **`Comparable`** enable custom ordering
✅ How to perform **thread-safe operations with collections in multithreaded applications**
✅ How **`fail-fast`** mechanisms work
✅ Best practices and performance comparisons

✨ Just let me know with a simple **"MORE"**! 🌟