ChatGPT

Mastering Java Collections: SCJP-Level Mastery

---

1. List Implementations

1.1 ArrayList

Description: Resizable array implementation of the List interface.

Key Features:

- Allows duplicate elements.
- Maintains insertion order.
- Fast random access (`O(1)`).
- Slower for insertions/deletions (`O(n)`).

Example:

```
import java.util.ArrayList;
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("SCJP");
System.out.println(list);
```

1.2 LinkedList

Description: Doubly-linked list implementation of the List interface.

Key Features:

- Allows duplicate elements.
- Maintains insertion order.
- Better for insertions/deletions (`O(1)` at both ends).
- Slower random access (`O(n)`).

Example:

```
import java.util.LinkedList;
LinkedList<String> list = new LinkedList<>();
list.add("One");
list.add("Two");
System.out.println(list);
```

1.3 Vector

Description: Synchronized resizable array implementation of the List interface.

Key Features:

- Allows duplicate elements.
- Maintains insertion order.
- Thread-safe (synchronized).
- Slower than ArrayList due to synchronization.

Example:

```
import java.util.Vector;
Vector<String> vec = new Vector<>();
vec.add("Thread");
vec.add("Safe");
System.out.println(vec);
```

---

2. Set Implementations

2.1 HashSet

Description: Hash table implementation of the Set interface.

Key Features:

- No duplicate elements.
- No guaranteed order.
- Fast operations (`O(1)` for add, remove, contains).

Example:

```
import java.util.HashSet;
HashSet<String> set = new HashSet<>();
set.add("A");
set.add("B");
set.add("A");
System.out.println(set);
```

2.2 LinkedHashSet

Description: HashSet with predictable iteration order (insertion order).

Key Features:

- No duplicate elements.

- Maintains insertion order.
- Slightly slower than HashSet due to linked list overhead.

Example:

```
import java.util.LinkedHashSet;
LinkedHashSet<String> set = new LinkedHashSet<>();
set.add("X");
set.add("Y");
System.out.println(set);
```

2.3 TreeSet

Description: NavigableSet implementation based on a red-black tree.

Key Features:

- No duplicate elements.
- Maintains sorted order (natural/custom comparator).
- Slower than HashSet (`O(log n)` for operations).

Example:

```
import java.util.TreeSet;
TreeSet<Integer> set = new TreeSet<>();
set.add(30);
set.add(10);
System.out.println(set);
```

---

3. Map Implementations

3.1 HashMap

Description: Hash table implementation of the Map interface.

Key Features:

- Allows null keys and values.
- No guaranteed order.
- Fast operations (`O(1)` for put, get).

Example:

```
import java.util.HashMap;
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "One");
```

```
map.put(null, "Null");
System.out.println(map);
```

3.2 LinkedHashMap

Description: HashMap with predictable iteration order (insertion order).

Key Features:

- Allows null keys and values.
- Maintains insertion order.
- Slightly slower than HashMap.

Example:

```
import java.util.LinkedHashMap;
LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
map.put(2, "Two");
map.put(1, "One");
System.out.println(map);
```

3.3 TreeMap

Description: NavigableMap based on red-black tree.

Key Features:

- No null keys (allows null values).
- Maintains sorted order.
- Slower than HashMap (`O(log n)`).

Example:

```
import java.util.TreeMap;
TreeMap<Integer, String> map = new TreeMap<>();
map.put(5, "Five");
map.put(3, "Three");
System.out.println(map);
```

3.4 Hashtable

Description: Synchronized implementation of Map.

Key Features:

- No null keys/values.
- Thread-safe.
- Slower due to synchronization.

Example:

```
import java.util.Hashtable;
Hashtable<Integer, String> table = new Hashtable<>();
table.put(1, "First");
System.out.println(table);
```

---

4. Queue Implementations

4.1 PriorityQueue

Description: Queue with natural/custom ordering.

Key Features:

- Allows duplicates.
- No guaranteed iteration order.
- `O(log n)` for insertions.

Example:

```
import java.util.PriorityQueue;
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(20);
pq.add(10);
System.out.println(pq);
```

4.2 ArrayDeque

Description: Resizable array implementation of Deque.

Key Features:

- No capacity restriction.
- Allows null elements.
- Faster than LinkedList for stack/queue operations.

Example:

```
import java.util.ArrayDeque;
ArrayDeque<String> deque = new ArrayDeque<>();
deque.push("A");
deque.push("B");
System.out.println(deque);
```

---

5. Fail-Fast vs Fail-Safe

Fail-Fast

- Throws `ConcurrentModificationException` if modified during iteration.
- Examples: `ArrayList`, `HashMap`, `HashSet`.

Fail-Safe

- Uses snapshot of collection; safe from `ConcurrentModificationException`.
- Examples: `CopyOnWriteArrayList`, `ConcurrentHashMap`.

Example:

```
List<String> list = new ArrayList<>();
list.add("One");
Iterator<String> it = list.iterator();
list.add("Two");
while(it.hasNext()) {
    System.out.println(it.next()); // Exception
}
```

---

6. equals() and hashCode()

Purpose

- Ensures objects behave correctly in collections like `HashMap`, `HashSet`.

equals()

- Defines equality logic.
- Must be consistent and transitive.

hashCode()

- Returns an integer used in hashing.
- Equal objects **must** return equal hash codes.

Contract:

- If `a.equals(b)` is true, then `a.hashCode() == b.hashCode()`.
- But `a.hashCode() == b.hashCode()` does **not** mean `a.equals(b)`.

Example:

```
class Employee {
    int id;
    String name;
```

```
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Employee)) return false;
        Employee e = (Employee) o;
        return id == e.id && name.equals(e.name);
    }

    public int hashCode() {
        return Objects.hash(id, name);
    }
}
```

---

7. Comparable vs Comparator

Comparable

- Natural ordering.
- Method: `int compareTo(T o)`
- Used in TreeSet/TreeMap.

Example:

```
class Fruit implements Comparable<Fruit> {
  String name;
  public int compareTo(Fruit other) {
    return this.name.compareTo(other.name);
  }
}
```

Comparator

- Custom ordering.
- Method: `int compare(T o1, T o2)`
- Useful when different order is needed without modifying class.

Example:

```
Comparator<Fruit> byName = (f1, f2) -> f1.name.compareTo(f2.name);
Collections.sort(list, byName);
```

---

8. Generics with Hidden Traps

Benefits

- Type safety.
- Eliminates casting.

- Readability.

Hidden Traps

- **Raw Types**: `List list = new ArrayList();` – not type safe.
- **Type Erasure**: Generic type info is erased at runtime.
- **No new T()**: Cannot create instance of generic type.

Example:

```
List<String> list = new ArrayList<>();
list.add("Apple");

class Box<T> {
  T item;
  void set(T item) { this.item = item; }
  T get() { return item; }
}
```

---

9. Mistakes & Learning Points

- Misunderstood fail-fast behavior → runtime crashes.
- Raw types used → `ClassCastException`.
- Confused Comparable vs Comparator → sorting errors.
- Skipping equals/hashCode in HashMap/HashSet → bugs.
- Used generics with primitives → compilation error.

---

10. Hidden Traps and SCJP-Level Tricks

- Modifying collections while iterating = danger zone.
- Not all collections support null (e.g., TreeMap).
- Choose collection based on performance needs.
- Casting with generics needs caution.
- Using wrong `equals()` or `hashCode()` ruins data integrity.

Conclusion

This guide gives you a strategic edge for cracking SCJP-style questions on Java Collections. Deep dive into the nuances, code each type out, and get your hands dirty testing fail-fast, sorting, and generics. Let the concepts stick by writing, breaking, and debugging them in real code.

Let me know if you'd like this in styled PDF or export-ready markdown.