100 Java Collections Interview Questions with Answers and Code Snippets

1. What is the Java Collections Framework?

Answer: The Java Collections Framework is a set of classes and interfaces in the `java.util` package that provides a standardized architecture for representing and manipulating collections of objects, such as lists, sets, and maps. It offers reusable data structures and algorithms for common operations like sorting, searching, and iteration.

---

2. What are the main interfaces in the Java Collections Framework?

Answer: The main interfaces are:

- `Collection` : Root interface for all collections.
- `List` : Ordered collection allowing duplicates.
- `Set` : Collection with no duplicates.
- `Map` : Key-value pair mapping.
- `Queue` : Collection for holding elements prior to processing.
- `Deque` : Double-ended queue.

---

3. What is the difference between `Collection` and `Collections`?

Answer: `Collection` is an interface that defines the basic methods for collections (e.g., `add` , `remove` , `size`). `Collections` is a utility class providing static methods for operations like sorting, searching, and synchronizing collections.

Code Snippet:

```
import java.util.*;

List<Integer> list = new ArrayList<>(Arrays.asList(3, 1, 2));
Collections.sort(list); // Using Collections utility class
System.out.println(list); // Output: [1, 2, 3]
```

---

4. What is the difference between `ArrayList` and `LinkedList`?

Answer:

- `ArrayList` uses a dynamic array, offering fast random access (`O(1)`) but slow insertions/deletions (`O(n)`).
- `LinkedList` uses a doubly-linked list, offering fast insertions/deletions (`O(1)`) but slow random access (`O(n)`).

Code Snippet:

```
List<String> arrayList = new ArrayList<>();
arrayList.add("A"); // Fast append
List<String> linkedList = new LinkedList<>();
linkedList.addFirst("B"); // Fast insertion at head
```

---

5. When would you use `ArrayList` over `LinkedList`?

Answer: Use `ArrayList` for frequent random access or iteration, as it has better performance for `get` operations. Use `LinkedList` when frequent insertions or deletions at the beginning or middle are required.

---

6. What is the difference between `HashSet` and `TreeSet`?

Answer:

- `HashSet` uses a hash table, offering `O(1)` average time for add/remove/contains but no order.
- `TreeSet` uses a red-black tree, offering `O(log n)` time for operations and maintains elements in sorted order.

Code Snippet:

```
Set<String> hashSet = new HashSet<>();
hashSet.add("Banana"); // Unordered
Set<String> treeSet = new TreeSet<>();
treeSet.add("Banana"); // Sorted
```

---

7. What is the difference between `HashMap` and `TreeMap`?

Answer:

- `HashMap` uses a hash table, providing `O(1)` average time for operations but no order.
- `TreeMap` uses a red-black tree, providing `O(log n)` time and maintains keys in sorted order.

Code Snippet:

```
Map<Integer, String> hashMap = new HashMap<>();
hashMap.put(1, "One"); // Unordered
Map<Integer, String> treeMap = new TreeMap<>();
treeMap.put(1, "One"); // Sorted by keys
```

---

8. What is `ConcurrentHashMap` and how does it differ from `HashMap`?

Answer: `ConcurrentHashMap` is a thread-safe version of `HashMap` that allows concurrent read and write operations with better performance than synchronized `HashMap`. It uses segment locking instead of locking the entire map.

Code Snippet:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
map.put(1, "One"); // Thread-safe
```

---

9. What is the difference between `Iterator` and `ListIterator`?

Answer:

- `Iterator` allows traversal of any `Collection` in one direction (forward) and supports `remove`.
- `ListIterator` is specific to `List` implementations, allows bidirectional traversal, and supports `add`, `set`, and `remove`.

Code Snippet:

```
List<String> list = new ArrayList<>(Arrays.asList("A", "B"));
ListIterator<String> iterator = list.listIterator();
```

```
iterator.next(); // Move to "B"
iterator.set("C"); // Replace "B" with "C"
System.out.println(list); // Output: [A, C]
```

---

10. What is the `fail-fast` behavior of iterators?

Answer: Iterators in Java Collections (e.g., `ArrayList`, `HashMap`) are fail-fast, meaning they throw `ConcurrentModificationException` if the collection is modified structurally (e.g., elements added/removed) during iteration, except through the iterator's own methods.

Code Snippet:

```
List<String> list = new ArrayList<>(Arrays.asList("A", "B"));
Iterator<String> iterator = list.iterator();
list.add("C"); // Causes ConcurrentModificationException
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

---

11. What is the difference between `HashMap` and `Hashtable`?

Answer:

- `HashMap` is not synchronized, allows null keys/values, and is faster.
- `Hashtable` is synchronized, does not allow nulls, and is thread-safe but slower.

Code Snippet:

```
HashMap<Integer, String> hashMap = new HashMap<>();
hashMap.put(null, "Value"); // Allowed
Hashtable<Integer, String> hashtable = new Hashtable<>();
// hashtable.put(null, "Value"); // Throws NullPointerException
```

---

12. How does `HashMap` handle collisions?

Answer: `HashMap` handles collisions using separate chaining. If multiple keys hash to the same index, they are stored in a linked list (or a balanced tree in Java 8+ for large buckets) at that index.

---

13. What is the load factor in `HashMap`?

Answer: The load factor is the ratio of the number of entries to the number of buckets. When this ratio exceeds the load factor (default 0.75), the `HashMap` resizes by doubling the bucket array size.

Code Snippet:

```
HashMap<String, Integer> map = new HashMap<>(16, 0.75f); // Initial capacity 16, load factor 0.75
```

---

14. What is the time complexity of operations in `HashMap`?

Answer:

- `get`, `put`, `remove`: `O(1)` average case, `O(n)` worst case (due to collisions).