@COVAIB-DEEPLEARN

—— FOLLOW ME ——

SUBSCRIBE

Master 20 Core Java Concepts

Interactive 3D Knowledge Matrix

- ENCAPSULATION UNWRAPPED
- INHERITANCE DECODED
- POLYMORPHISM IN ACTION
- AGGREGATION, COMPOSITION & ASSOCIATION
- CONSTRUCTOR CHAINING MASTERCLASS
- COVARIANCE & CONTRAVARIANCE
- FINAL, FINALLY, FINALIZE()
- IMMUTABILITY MASTERY
- JVM MEMORY MODEL
- THE STRING POOL DEEP DIVE
- SINGLETON CLASS IN JAVA
- THE STATIC KEYWORD DEMYSTIFIED
- HOW JVM LOADS CLASSES
- THREAD LIFECYCLE & SCHEDULER
- EXECUTORSERVICE & FUTURE
- SYNCHRONIZED COLLECTIONS & CONCURRENT DATA STRUCTURES
- TODAY'S TOPIC — **JAVA COLLECTIONS**
- STREAM API EXPLAINED
- FUNCTIONAL PROGRAMMING
- JAVA CONCURRENCY

✸ Why This Document Stands Out

🧠 Multi-Layered Learning

Structured path from Basics → JVM → Real Systems → Interview Traps.

⚙️ JVM-Level Depth

Master vtable, bytecode, JIT, and late binding mechanisms.

🎯 Interview-First

Content written specifically how senior interviewers probe for depth.

🧪 Real World Systems

Learn through context: Payments, fraud detection, and transactional systems.

📈 Progressive Mastery

Concepts evolving through 6 levels of complexity for true expertise.

📊 Performance Aware

Critical insights into when polymorphism hurts and how to optimize hot paths.

DOWNLOAD PDF FROM CAPTION

🟦🚀 Java Collections – Complete Deep Dive Guide

🔥 From Core APIs → JVM Internals → MAANG-Level Mastery

⭐ **Save This Page Now! You'll need it for interview prep.**

A powerful, interview-ready reference covering all major Java Collections.

Built for **clarity**, **low-level understanding**, and **high-paying interviews**.

📚 What's Inside the Guide?

A complete multi-chapter deep dive covering:

- 🧩 **Collections Architecture** (List/Set/Queue/Map)
- 🧠 **Deep Internals**: HashMap, TreeMap, LinkedHashMap
- ⚙️ **Concurrency & Thread-Safe Collections**
- 🚦 Fail-Fast vs Fail-Safe Iterators
- 🧵 CopyOnWrite, CHM Internals, CAS, Volatile
- Real-World Fintech System Design Scenarios

💻

📝 150+ Interview Questions (L1 → L7)

⚡ Performance Tuning + JVM Memory Interaction

💥 Why Developers Love This Guide

- ✓ Industry-quality explanations
- ✓ JVM + system design perspective
- ✓ Clear diagrams & real examples
- ✓ Perfect for 60 LPA – 1.5 Cr roles
- ✓ Made for serious backend engineers

📥 Download the Full Guide on Gumroad

👉 Get your premium version:

🔗 **gumroad.com/your-guide-link** 🚀

🔗📌 Follow CoVaib DeepLearn for Daily JVM & Java Posts

Grow your skills with daily JVM internals, concurrency, system design & interview patterns.

in Follow CoVaib DeepLearn on LinkedIn

🧠✨ Daily knowledge. Career Compounding. Backend Mastery.

Java Collections

Index

1) Core Definition (for Clarity)

Java Collections Framework (JCF) is a set of interfaces, classes, and algorithms to handle groups of objects (collections) efficiently.

Key Interfaces:

- **Collection (root interface)**
- **List (ordered, allows duplicates)**
- **Set (no duplicates)**
- **Queue (FIFO or priority order)**
- **Map (key-value pairs, not a Collection)**

Implementations:

- **ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue, HashMap, TreeMap, LinkedHashMap, ConcurrentHashMap, etc.**

2) Why We Need Collections & Alternatives

Why Collections?

- **Provide reusable data structures with well-defined behavior and performance guarantees.**
- **Abstract away low-level array handling, dynamic resizing, searching, sorting.**
- **Standardized iteration, concurrency, synchronization support.**

Alternatives:

- **Plain arrays (fixed size, less flexible).**
- **Third-party libraries (e.g., Apache Commons, Guava Collections).**
- **Custom data structures (rare, only when extreme performance/custom behavior needed).**

3) Deep Technical Internals (What Happens Under The Hood)

a) Core Data Structures

Collection	Underlying Data Structure	Time Complexities (Avg)
ArrayList	Resizable Array	get O(1), add O(1 amortized)
LinkedList	Doubly Linked List	get O(n), add O(1)
HashSet	HashMap (backed by HashMap keys)	add/contains/remove O(1)
TreeSet	Red-Black Tree	add/contains/remove O(log n)
PriorityQueue	Binary Heap	add O(log n), poll O(log n)

Collection	Underlying Data Structure	Time Complexities (Avg)
HashMap	Array of buckets + linked list/tree	get/put O(1)
TreeMap	Red-Black Tree	get/put O(log n)
LinkedHashMap	HashMap + Doubly Linked List (insertion order)	get/put O(1)

b) Hashing Internals

- HashMap computes hashCode, applies supplemental hash function to reduce collisions.
- Buckets hold linked lists or balanced trees (Java 8+) when collisions exceed threshold (TREEIFY_THRESHOLD = 8).
- Tree nodes provide O(log n) search inside a bucket.
- Rehashing triggered on resize — costly, amortized cost distributed over inserts.

c) Fail-Fast Iterators

- Most collections provide fail-fast iterators (fail if collection modified concurrently).
- Implemented via modification count (modCount) checked during iteration.
- Useful to detect bugs but not thread-safe.

d) Concurrent Collections

- ConcurrentHashMap uses lock-striping and CAS operations for fine-grained concurrency.
- CopyOnWriteArrayList trades write performance for read concurrency (snapshot semantics).
- BlockingQueue implementations provide thread-safe producer-consumer queues.

4) Real-World Usage & Patterns

- Use ArrayList for random access and when additions happen mostly at end.
- Use LinkedList for frequent insertions/removals at ends or inside lists.
- Use HashSet/HashMap for fast lookups with no ordering needs.
- Use TreeSet/TreeMap when sorted order is required.
- Use LinkedHashMap for LRU caches (override removeEldestEntry).
- Use ConcurrentHashMap for thread-safe, highly concurrent access.
- Use PriorityQueue for scheduling tasks or event handling.

5) Tricky Interview Q&A (Advance level)

Question	Senior-Level Answer
Explain HashMap internal structure and how collisions are handled?	Uses array of buckets, with linked list or balanced tree for collisions. Rehash on resize. Treeify threshold triggers balanced tree for better worst-case performance.
Difference between HashMap and ConcurrentHashMap?	HashMap is non-thread-safe, ConcurrentHashMap uses lock striping and CAS for concurrent reads/writes with no global locking.
How does fail-fast iterator work?	Checks modCount before each next() call, throws ConcurrentModificationException if modified outside iterator. Not a thread-safety guarantee but early fail detection.
How does LinkedHashMap maintain insertion order?	Doubly linked list connecting entries, updated on access if accessOrder=true for LRU cache implementations.
How to implement LRU Cache using Java collections?	Use LinkedHashMap with accessOrder=true, override removeEldestEntry to remove oldest entry when size exceeds max.
Explain load factor in HashMap?	Controls when resize occurs. Default 0.75 balances space/time. Too low → more resize, too high → more collisions.
When to use TreeMap over HashMap?	When you need sorted order traversal of keys; TreeMap implements SortedMap with O(log n) operations.
Explain CopyOnWriteArrayList advantages and disadvantages?	Thread-safe for read-heavy scenarios, writes copy entire array, so write-heavy workloads suffer.
How to prevent ConcurrentModificationException when iterating and modifying?	Use Iterator's remove(), use Concurrent Collections, or iterate over snapshot copy.
How does HashSet use HashMap internally?	HashSet is backed by a HashMap with dummy constant value; keys represent set elements.

6) Common Pitfalls Interviewers Test

- **Forgetting HashMap's resize impact on performance.**
- **Using LinkedList where ArrayList is better (or vice versa).**
- **Ignoring thread-safety of collections in concurrent environment.**
- **Misunderstanding fail-fast vs fail-safe iterators.**
- **Not handling key equality (equals/hashCode) properly in custom objects.**
- **Using mutable objects as HashMap keys without care.**
- **Wrong assumptions on ordering guarantees in HashSet or HashMap.**
- **Confusing behavior of remove() in Iterator vs Collection.**
- **Misunderstanding differences between shallow vs deep copy of collections.**

7) JVM Internals (Detailed)

- **Collections classes are loaded by system/app class loader, metadata stored in Metaspace.**
- **Many collections internally maintain transient fields like `modCount` (for fail-fast).**
- **HashMap's bucket array is lazily initialized; resizing uses `Arrays.copyOf()` plus rehashing keys.**
- **Iterator classes are often inner classes holding references to the collection and current index/node.**
- **ConcurrentHashMap uses `Unsafe` operations (CAS) for atomic updates.**
- **JVM Hotspot optimizes common collection method calls via inlining and loop unrolling.**
- **Lazy initialization and cached hash codes in immutable collection implementations (e.g., String's hashCode).**

8)Advance Level

1. Custom comparators for TreeSet/TreeMap

- **Default behavior: TreeSet/TreeMap use *natural ordering* of elements (Comparable).**
- **With custom comparator: You can control ordering logic.**

```
TreeSet<String> set = new TreeSet<>((a, b) -> Integer.compare(a.length(), b.length()));
set.add("banana"); set.add("apple"); set.add("pear");
// Ordered by length, not lexicographically
```

- **Tricky Part:**
    - **Comparator defines *both ordering and equality*. If comparator says `compare(a, b) == 0`, they are treated as *equal* → duplicates are removed even if `equals()` is false.**

- MAANG Interview Question:
  👉 "If you override comparator inconsistently with equals(), what's the consequence?"
  ✔ Answer: Violates Set contract → elements may vanish unexpectedly. Must ensure consistency (a.equals(b) ⇒ compare(a,b)==0).

2. WeakHashMap and SoftReference-based collections

- WeakHashMap: Keys are held with WeakReference. Once key is GC'd → entry removed automatically.

```
Map<Object, String> weakMap = new WeakHashMap<>();
Object key = new Object();
weakMap.put(key, "value");
key = null; // eligible for GC
System.gc(); // Entry may vanish
```

- SoftReference: Useful for caching — objects kept until JVM really needs memory.
- Why asked in interview?
  👉 Because cache design is a favorite MAANG system design + low-level DS question.
- Tricky Q:
  - "How would you design a cache that auto-cleans entries when memory is low?"
    ✔ Answer: Use SoftReference for values, WeakHashMap for keys.

3. IdentityHashMap vs HashMap

- HashMap: Uses equals() + hashCode() for key comparison.
- IdentityHashMap: Uses == (reference equality).

```
Map<String, String> idMap = new IdentityHashMap<>();
idMap.put(new String("a"), "one");
idMap.put(new String("a"), "two");
System.out.println(idMap.size()); // 2 (different objects, even though equals)
```

- When used:
  - Serialization frameworks, proxy objects, graph traversals where identity matters.
- Tricky Q:
  - "Why does IdentityHashMap allow duplicates where HashMap doesn't?"
    ✔ Because it doesn't care about logical equality, only reference identity.

4. EnumSet and EnumMap

- Highly optimized → internally use bit vectors (for EnumSet) or arrays indexed by enum ordinal (for EnumMap).
- Much faster & memory-efficient than HashSet/HashMap for enums.

```
EnumSet<Color> colors = EnumSet.of(Color.RED, Color.BLUE);
EnumMap<Color, String> map = new EnumMap<>(Color.class);
```

- Tricky Q:
  - "Why does EnumSet outperform HashSet?"
    ✔ Because operations are just bit manipulations (O(1) memory lookup).
  - "Can EnumMap accept null keys?"
    ✔ No — because enum constants are guaranteed finite + non-null.

5. Spliterator & Parallel Streams

- Spliterator = "Split + Iterator". Supports splitting a collection into chunks for parallel processing.

```
Spliterator<String> sp = list.spliterator();
Spliterator<String> half = sp.trySplit(); // split work in two
```

- Key point: Enables efficient parallel streams (list.parallelStream()).
- MAANG-level tricky Q:
  - "Why not just use Iterator in parallel streams?"
    ✔ Iterator only moves sequentially. Spliterator supports chunking for divide-and-conquer parallelism.
  - "What is CHARACTERISTICS flag in Spliterator?"
    ✔ Like ORDERED, SORTED, CONCURRENT, help JVM optimize execution.

6. Designing custom collection (using AbstractCollection/AbstractList)

- Instead of implementing all methods, extend skeletal implementations like AbstractList, AbstractMap.

```
class MyList<E> extends AbstractList<E> {
   private E[] data;
   @Override public E get(int index) { return data[index]; }
   @Override public int size() { return data.length; }
}
```

- **Tricky Q:**
  - **"Why not directly implement List?"**
    ✔ Too many methods → skeletal classes reduce boilerplate and ensure consistency.

7. Collections.synchronizedXXX wrappers

- **Example:**

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
```

- **Internals: Wraps the collection and synchronizes each method on a lock.**
- **Caveat: Iteration still unsafe without external sync:**

```
synchronized(syncList) {
   for (String s : syncList) { ... }
}
```

- **Tricky Q:**
  - **"What's the difference vs CopyOnWriteArrayList?"**
    ✔ sync wrapper = global lock (scales poorly), CopyOnWrite = snapshot reads (better read-heavy).

8. Serialization considerations

- **Large collections → serialization can be very expensive (time + memory).**
- **HashMap serialization → writes *all entries individually*.**
- **For huge datasets:**
  - **Use transient + custom serialization (writeObject, readObject).**
  - **Or external libraries like Kryo / Protobuf.**
- **Tricky Q:**
  - **"Why is serializing HashMap expensive?"**
    ✔ Because it serializes bucket by bucket, including empty slots in some versions.
  - **"How would you serialize a ConcurrentHashMap safely?"**
    ✔ Use snapshot copy → serialize → restore later.

🔥 How to Nail in Interview

- **Don't just explain features — tie them to real-world use cases:**
  - **WeakHashMap → caching in large ML pipelines.**
  - **Spliterator → parallel ETL batch jobs.**

  - EnumSet → permissions, feature flags in microservices.
  - IdentityHashMap → object graph serialization / proxy frameworks.
- Avoid premature optimization — measure with tools like VisualVM, Java Flight Recorder (JFR), YourKit.
- Large `ArrayList` resizing causes expensive array copies; consider initial capacity tuning.
- HashMap resizing leads to pauses; tuning initial capacity & load factor reduces this.
- Beware large LinkedList due to non-contiguous memory, poor CPU cache locality.
- Concurrent collections add overhead for thread safety but improve scalability.
- Use primitive specialized collections (Trove, FastUtil) to avoid boxing overhead.
- Serialization of huge collections impacts GC; consider streaming APIs or external DB caches.

9) JVM Bytecode Walkthrough: Collection Method Calls

Example: ArrayList add() method bytecode (simplified)

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);
    elementData[size++] = e;
    return true;
}
```

Bytecode (simplified):

```
aload_0                 // load 'this'
iload_1                 // load 'e'
invokespecial ...       // call ensureCapacityInternal
aload_0
getfield elementData
aload_0
getfield size
aload_1
aastore                 // store element in array
aload_0
dup
getfield size
iconst_1
iadd
putfield size            // increment size
iconst_1
ireturn                 // return true
```

- JVM optimizes calls with inlining for hot methods like `add()`.
- JVM handles array resizing and emory copy internally.

10) Fail-Fast vs Fail-Safe Iterators

Feature	Fail-Fast Iterator	Fail-Safe Iterator
Examples	`ArrayList`, `HashMap`, `HashSet`	`ConcurrentHashMap`, `CopyOnWriteArrayList`
Behavior on Modify	Throws `ConcurrentModificationException` if collection modified after iterator created	Operates on a snapshot copy; no exception
Use Case	Single-threaded or externally synchronized environments	Concurrent, multi-threaded environments
Mechanism	ModCount field checked during iteration	Works on cloned copy of collection

Example:

```
List<String> list = new ArrayList<>(List.of("A", "B", "C"));
Iterator<String> it = list.iterator();
list.add("D"); // modifying list after iterator creation
it.next();     // Throws ConcurrentModificationException
```

11) Comparator vs Comparable

Aspect	Comparable	Comparator
Location	Implements inside class itself	Separate class or lambda expression
Method	`compareTo(T o)`	`compare(T o1, T o2)`
Modification	Alters natural ordering	Custom, multiple orderings possible
Use Case	Default sorting, e.g., `Collections.sort(list)`	External/custom sorting logic
Null Handling	Needs explicit code	Can use `Comparator.nullsFirst()` or `nullsLast()`

Example Comparable:

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

Example Comparator:

```
Comparator<Person> nameComparator = (p1, p2) -> p1.name.compareTo(p2.name);
Collections.sort(personList, nameComparator)
```

12) Stream vs Parallel Stream

Feature	Stream	Parallel Stream
Execution	Sequential	Parallel, multi-threaded

Feature	Stream	Parallel Stream
Use Case	Simpler data processing pipelines	Large data with parallelizable tasks
Overhead	Lower	Higher due to thread coordination
Ordering	Maintains encounter order by default	May not maintain order unless explicitly specified
Performance Caveat	Best for simple or small datasets	Best for CPU-intensive or large data
Thread Management	Single thread	Uses ForkJoinPool (common pool by default)

13) Synchronized Collections: Set, List, Map

Interview Question

Level 1 - Fundamentals (warm up question)

Q1: What is the Java Collections Framework?

1) Definition & Overview

The Java Collections Framework (JCF) is a unified architecture for representing and manipulating collections of objects. It provides:

1. **Interfaces like** List, Set, Map, Queue.
2. **Concrete implementations (**ArrayList, HashSet, HashMap, LinkedList, PriorityQueue**).**
3. **Algorithms (**Collections.sort(), binarySearch(), shuffle()**).**
4. **Utilities for synchronization, immutability, and concurrent access (**Collections.synchronizedList, CopyOnWriteArrayList**).**

Key goals:

- **Standardize collection manipulation.**
- **Decouple algorithms from data structures.**
- **Provide high performance and flexible APIs.**

2) Copy-Pasteable Java Example

```
import java.util.*;

public class CollectionsDemo {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>();
      list.add("Apple");
      list.add("Banana");
```

```
        list.add("Cherry");

        Set<Integer> set = new HashSet<>();
        set.add(10);
        set.add(20);
        set.add(10); // Duplicate ignored

        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);

        // Using algorithms
        Collections.sort(list);
        System.out.println("Sorted List: " + list);
        System.out.println("Set: " + set);
        System.out.println("Map: " + map);
    }
}
```

3) Step-by-Step Dry Run with Concrete Inputs

Input:

- List: ["Apple", "Banana", "Cherry"]
- Set: [10, 20, 10]
- Map: {"Alice":25, "Bob":30}

Execution:

1. list.add("Apple") → list: ["Apple"]
2. list.add("Banana") → list: ["Apple", "Banana"]
3. list.add("Cherry") → list: ["Apple", "Banana", "Cherry"]
4. set.add(10) → set: [10]
5. set.add(20) → set: [10, 20]
6. set.add(10) → duplicate ignored → set remains [10, 20]
7. map.put("Alice",25) → map: {"Alice":25}
8. map.put("Bob",30) → map: {"Alice":25, "Bob":30}
9. Collections.sort(list) → ["Apple", "Banana", "Cherry"]

Output:

```
Sorted List: [Apple, Banana, Cherry]
Set: [10, 20]
Map: {Alice=25, Bob=30}
```

4) Deep JVM Internals

1. **Object Layout:**
   Each collection object (e.g., ArrayList) has:
2. **+----------------+**
3. **| Object Header | <-- Mark word (lock, hash, GC info)**
4. **| Klass Pointer | <-- Points to ArrayList class metadata in metaspace**
5. **+----------------+**
6. **| fields | <-- size, modCount, elementData[]**
7. **+----------------+**
8. **VTable/ITables:**
   - **Method calls (e.g., add()) are resolved via vtable for virtual methods.**
   - **Interfaces are resolved via itable entries.**
9. **Bytecode Example for list.add("Apple"):**
10. **aload_1 // load 'list'**
11. **ldc "Apple" // push "Apple"**
12. **invokevirtual ArrayList.add:(Ljava/lang/Object;)Z**
13. **pop**
14. **order:**
    - **Static initializers run once per class (ArrayList, HashSet) in JVM metaspace.**
    - **Ensures shared constants (like default capacity) are initialized.**
15. **Exception tables & access checks:**
    - **JVM ensures NullPointerException or ClassCastException for invalid operations.**
    - **Access checks prevent violating encapsulation of collection internals.**

5) HotSpot Optimizations

1. **Method inlining: add(), put(), get() can be inlined for small hot methods.**
2. **Inline caching: JVM caches the resolved method call for interface/virtual calls.**
3. **Deoptimization: If assumptions fail (e.g., class hierarchy changes), JVM falls back safely.**

6) GC Interactions & Metaspace

- **Collection elements are heap objects, subject to minor/major GC.**
- **Resizing ArrayList triggers allocation of larger Object[] → old array becomes garbage.**

- Class metadata (ArrayList.class, HashMap.class) lives in Metaspace.
- Weak references or SoftReference wrappers allow caching without memory leaks.

7) Real-World Tie-ins

1. Spring Boot:
   - BeanFactory uses HashMap for singleton beans.
2. Hibernate:
   - Set or List for entity collections; lazy-loading uses proxies.
3. Payments:
   - Collections used for caching transactions, mapping user accounts, and processing queues in multi-threaded payment pipelines.

8) Pitfalls & Refactors

1. Using ArrayList when frequent inserts/removals → prefer LinkedList.
2. Iterating HashMap without entrySet() → creates unnecessary iterator overhead.
3. Refactor with composition:
   - Wrap collection inside a class for LRU cache or frequency counter.
4. Strategy pattern:
   - Pass Comparator for flexible sorting instead of hardcoding.

9) Interview Follow-ups (1-Line Expert Answers)

1. Why use HashMap over TreeMap? → HashMap provides O(1) avg. lookup vs TreeMap O(log n).
2. Difference between fail-fast and fail-safe? → Fail-fast throws ConcurrentModificationException; fail-safe works on a copy.
3. Default capacity of HashMap? → 16, load factor 0.75.
4. ArrayList vs LinkedList memory footprint? → ArrayList is contiguous array; LinkedList has 3x memory overhead (node + prev + next + data).

Q2: What are the core interfaces in Java Collections?

1) Definition & Overview

The core interfaces of the Java Collections Framework define the contract for collection types, decoupling algorithms from data structures.

Core Interfaces:

1. **Collection – root interface for single-element collections.**
2. **List – ordered collection allowing duplicates (ArrayList, LinkedList).**
3. **Set – unique-element collection (HashSet, TreeSet).**
4. **Queue – FIFO or priority-based collection (LinkedList, PriorityQueue).**
5. **Deque – double-ended queue supporting head/tail operations (ArrayDeque, LinkedList).**
6. **Map – key-value mapping (HashMap, TreeMap, ConcurrentHashMap).**

Hierarchy Diagram (Text-Based):

```
      Iterable
         |
      Collection
      /   |   \
   List   Set   Queue
            |
          Deque
      Map (not Collection)
```

2) Copy-Pasteable Java Example

```
import java.util.*;

public class CoreInterfacesDemo {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>();
      Set<Integer> set = new HashSet<>();
      Queue<String> queue = new LinkedList<>();
      Deque<String> deque = new ArrayDeque<>();
      Map<String, Integer> map = new HashMap<>();

      list.add("Apple");
      set.add(10);
```

```
        queue.offer("Job1");
        deque.offerFirst("Task1");
        map.put("Alice", 25);

        System.out.println(list + " " + set + " " + queue + " " + deque + " " + map);
    }
}
```

Dry Run:

- ArrayList → ["Apple"]
- HashSet → [10]
- LinkedList as Queue → ["Job1"]
- ArrayDeque → ["Task1"]
- HashMap → {"Alice":25}

Output:

```
[Apple] [10] [Job1] [Task1] {Alice=25}
```

3) JVM Internals Deep Dive

3.1 Object Layout

- Every collection object has:

```
+----------------------+
| Mark Word            |  // lock, hashCode, GC info
| Klass Pointer        |  // points to runtime class in Metaspace
+----------------------+
| Instance Fields      |  // e.g., ArrayList: size, modCount, elementData[]
+----------------------+
```

- Example: ArrayList elementData[] is an object array allocated on heap.
- HashMap internal Node<K,V>[] table uses array of nodes pointing to Entry objects, each with key, value, hash, and next reference.

3.2 VTable / Interface Table (itable)

- Interface methods (add, remove) are resolved via itable entries.
- Virtual method dispatch for interface call (list.add("x")) → JVM looks up itable in the klass structure for ArrayList.

3.3 Bytecode Example

```
list.add("Apple")
aload_1            // load list ref
ldc "Apple"        // push String constant
invokeinterface List.add:(Ljava/lang/Object;)Z
pop
```

- invokeinterface → resolves method at runtime using itable pointer in klass structure.

3.4 Class Initialization ()

- Static constants (default capacities, empty arrays) initialized once in Metaspace when class loaded:

```
ArrayList:
  static final Object[] EMPTY_ELEMENTDATA = {};
HashMap:
  static final int DEFAULT_INITIAL_CAPACITY = 16;
```

- Ensures shared metadata is allocated in Metaspace, not heap.

4) HotSpot Optimizations

1. Inlining:
   - ArrayList.add(), HashMap.put() are small, hot methods → likely inlined.
2. Inline Caches:
   - JVM caches resolved method calls for interface dispatch to reduce invokeinterface overhead.
3. Escape Analysis:
   - If ArrayList is local and not escaping the method, JVM may allocate elementData array on stack (scalar replacement) → avoids heap allocation.
4. Branch Prediction:
   - Iteration loops in collections optimized for predictable patterns.

5) GC & Memory Considerations

1. Heap Allocation:
   - All collection instances and internal arrays/nodes reside in heap.

- Resizing arrays triggers allocation + old array becomes garbage → minor GC candidate.

2. Weak/Soft References:
   - WeakHashMap keys can be GC'd to avoid memory leaks.
3. Metaspace:
   - Class definitions (ArrayList.class, HashMap.class) stored in Metaspace, not subject to normal heap GC.

6) Real-World Tie-ins

1. Spring Framework:
   - Uses Map<String,Object> for bean registries and configuration caches.
2. Hibernate:
   - Uses Set or List to manage entity relationships with lazy-loading proxies.
3. Payments Systems:
   - Queue or PriorityQueue for processing transaction batches.
   - ConcurrentHashMap for multi-threaded caching of account balances.

7) Pitfalls & Refactors

1. Using List when uniqueness is required → prefer Set.
2. HashMap with mutable keys → risk of broken hashCode/equals consistency.
3. Refactor with composition: wrap collection in a domain-specific class for LRU cache or thread-safe access.
4. Strategy pattern: pass Comparator or Predicate to generic methods instead of hardcoding logic.

8) Interview Follow-ups (1-Line Expert Answers)

1. Why Map is not a Collection? → Map stores key-value pairs, not single elements.
2. ArrayList vs LinkedList trade-offs? → ArrayList: O(1) random access, O(n) insert/remove; LinkedList: O(n) random access, O(1) insert/remove.
3. Why use Deque over Stack? → Deque avoids legacy Stack class and provides more flexible API.

4. HashSet internal mechanics? → Uses HashMap internally; values stored as keys with dummy objects.

Q3: Difference between Collection and Map interfaces

1) Definition & Overview

Aspect	Collection Interface	Map Interface
Hierarchy	Extends `Iterable`	Standalone (does not extend `Collection`)
Purpose	Stores single elements	Stores key-value pairs
Common Implementations	`ArrayList`, `LinkedList`, `HashSet`, `TreeSet`	`HashMap`, `TreeMap`, `LinkedHashMap`, `ConcurrentHashMap`
Duplicates	`List` allows duplicates, `Set` disallows	Keys must be unique; values can be duplicates
Access	By index (List) or iteration	By key (hash lookup or sorted order)
Iteration	`Iterator`	`entrySet()`, `keySet()`, `values()`

Text-Based Hierarchy Diagram

```
     Iterable
        |
    Collection
    /   |   \
  List  Set  Queue
Map (Standalone)
  |-- HashMap
  |-- TreeMap
  |-- LinkedHashMap
```

2) Copy-Pasteable Java Example

```
import java.util.*;
```

```
public class CollectionVsMapDemo {
   public static void main(String[] args) {
      Collection<String> collection = new ArrayList<>();
      collection.add("A");
      collection.add("B");

      Map<String, Integer> map = new HashMap<>();
      map.put("A", 1);
      map.put("B", 2);

      // Iteration
      System.out.println("Collection elements:");
      for (String s : collection) System.out.println(s);

      System.out.println("Map entries:");
      for (Map.Entry<String,Integer> e : map.entrySet())
         System.out.println(e.getKey() + " -> " + e.getValue());
   }
}
```

Dry Run:

- collection: ["A", "B"]
- map: {"A":1, "B":2}
- Iteration prints elements/entries accordingly.

3) JVM Internals Deep Dive

3.1 Object Layout

Collection Example: ArrayList

```
+----------------+
| Mark Word      | // GC, hash, lock info
| Klass Pointer  | // Points to ArrayList.class
+----------------+
| size           |
| modCount       |
| elementData[]  | // Object[] in heap
+----------------+
```

Map Example: HashMap

```
+----------------+
| Mark Word      |
```

```
| Klass Pointer  | // HashMap.class in Metaspace
+----------------+
| size           |
| threshold      |
| loadFactor     |
| Node<K,V>[] table | // array of Node objects
+----------------+
```

Node Object Layout:

```
+----------------+
| Mark Word      |
| Klass Pointer  |
| int hash       |
| K key          |
| V value        |
| Node<K,V> next | // linked list chaining
+----------------+
```

3.2 Method Dispatch (VTable/Itable)

- Collection.add() → virtual method via vtable.
- Map.put() → virtual method, interface-independent, resolved via concrete class vtable.
- Interface calls (add, remove) resolved via itable; Map calls resolved via class hierarchy directly.

3.3 Bytecode Examples

Collection add

```
aload_1            // load ArrayList
ldc "A"            // push constant
invokeinterface Collection.add:(Ljava/lang/Object;)Z
pop
```

Map put

```
aload_2            // load HashMap
ldc "A"            // push key
iconst_1
invokestatic Integer.valueOf(I) // autobox
invokevirtual HashMap.put:(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
pop
```

3.4 Order

- ArrayList initializes DEFAULT_CAPACITY, EMPTY_ELEMENTDATA in Metaspace.

- HashMap initializes DEFAULT_INITIAL_CAPACITY = 16, DEFAULT_LOAD_FACTOR = 0.75.
- JVM guarantees thread-safe class initialization during <clinit>.

4) HotSpot Optimizations

1. Inlining: Hot methods like ArrayList.add() and HashMap.put() are inlined.
2. Inline caches: Repeated interface/virtual calls for collections are cached.
3. Escape Analysis:
    - If ArrayList or HashMap is local and doesn't escape, JVM may allocate stack arrays instead of heap.
4. Branch Prediction:
    - Array access patterns in ArrayList and hash bucket lookups in HashMap are optimized for CPU pipeline.

5) GC & Memory Considerations

- Collection:
    - ArrayList internal array (elementData[]) grows → triggers old array GC.
- Map:
    - HashMap.Node[] table allocated in heap; resizing causes old nodes to be garbage-collected.
    - Soft/Weak values reduce memory pressure; WeakHashMap keys can be GC'd when not referenced elsewhere.
- Metaspace:
    - ArrayList.class, HashMap.class in Metaspace; only collected if class unloaded.

Text Diagram of Resizing

```
HashMap (size 16) -> put() -> threshold exceeded
Allocate new table size 32
Rehash nodes from old table
Old table eligible for GC
```

6) Real-World Tie-ins

1. Spring:

- BeanFactory: uses Map<String, Object> to store beans; iteration via entrySet().

2. Hibernate:
   - Entity collections: Set vs List based on uniqueness/order.
3. Payments & Fintech:
   - Transaction queues → Queue or Deque.
   - Account lookup → ConcurrentHashMap (multi-threaded) instead of Collection.

7) Pitfalls & Refactors

1. Confusing Collection vs Map → leads to API misuse.
2. Using mutable objects as keys in Map → broken hashCode consistency.
3. Refactor: Wrap Map/Collection in domain-specific caches.
4. Strategy pattern: Pass comparator/predicate to generic collection utility methods.

8) Interview Follow-ups (1-Line Expert Answers)

1. Why Map does not extend Collection? → It stores key-value pairs, not single elements.
2. HashMap vs ArrayList memory footprint? → HashMap has extra Node objects + chaining; ArrayList is contiguous array.
3. Iteration differences? → Collection: iterator over elements; Map: iterator over entrySet().
4. Resizing impact? → Collection resizing reallocates array; Map resizing rehashes keys → GC pressure.

Q4: Difference between List, Set, and Queue

1) Overview

Aspect	List	Set	Queue
Purpose	Ordered collection, allows	Unique elements, unordered (mostly)	FIFO (or priority-based) element processing

Aspect	List	Set	Queue
	duplicates		
Duplicates	Allowed	Not allowed	Allowed depending on implementation
Order	Maintains insertion order	No guaranteed order (`HashSet`)	Usually insertion order (`LinkedList`) or priority order (`PriorityQueue`)
Access	By index (`get(int)`)	No index-based access	Access head element (`peek`, `poll`)
Implementations	`ArrayList`, `LinkedList`, `Vector`	`HashSet`, `TreeSet`, `LinkedHashSet`	`LinkedList`, `PriorityQueue`, `ArrayDeque`
Iteration	ListIterator or Iterator	Iterator only	Iterator over elements, head-based access
Internal Storage	Array or linked nodes	Hash table (`HashSet`) or tree (`TreeSet`)	Linked nodes, array, or heap

Text-Based Hierarchy Diagram

```
        Iterable
           |
        Collection
       /   |   \
     List  Set  Queue
    / \      / \
  ArrayList LinkedList PriorityQueue LinkedList
HashSet TreeSet LinkedHashSet ArrayDeque
```

2) Copy-Pasteable Java Example

```
import java.util.*;

public class ListSetQueueDemo {
   public static void main(String[] args) {
      // List
      List<String> list = new ArrayList<>();
```

@CoVaib-DeepLearn

```
        list.add("A");
        list.add("B");
        list.add("A"); // duplicates allowed
        System.out.println("List: " + list);

        // Set
        Set<String> set = new HashSet<>();
        set.add("A");
        set.add("B");
        set.add("A"); // duplicates ignored
        System.out.println("Set: " + set);

        // Queue
        Queue<String> queue = new LinkedList<>();
        queue.add("A");
        queue.add("B");
        queue.add("C");
        System.out.println("Queue peek: " + queue.peek());
        System.out.println("Queue poll: " + queue.poll());
        System.out.println("Queue after poll: " + queue);
    }
}
```

Dry Run Example:

- **List → ["A", "B", "A"]**
- **Set → ["A", "B"] (unordered)**
- **Queue → peek: "A", poll: "A", remaining: ["B", "C"]**

3) JVM Internals Deep Dive

3.1 Object Layout

ArrayList (List):

```
+----------------+
| Mark Word      | // lock, GC info
| Klass Pointer  | // ArrayList.class
+----------------+
| int size       |
| int modCount   |
| Object[] elementData |
+----------------+
```

HashSet (Set) internally uses HashMap:

HashSet -> HashMap<K,Object>
HashMap table: Node<K,V>[]
Node layout:

```
+----------------+
| Mark Word      |
| Klass Pointer  |
| int hash       |
| K key          |
| V value        |
| Node next      |
+----------------+
```

LinkedList (Queue):

LinkedList:

```
+----------------+
| Mark Word      |
| Klass Pointer  |
| int size       |
| Node<E> first  |
| Node<E> last   |
+----------------+
```

Node<E>:

```
+----------------+
| Mark Word      |
| Klass Pointer  |
| E item         |
| Node<E> next   |
| Node<E> prev   |
+----------------+
```

3.2 VTable & Itable Dispatch

- **List: index-based methods (get(int)), virtual dispatch via vtable.**
- **Set: add() calls HashMap.put(key, PRESENT) internally → interface dispatch via itable.**
- **Queue: add(), peek(), poll() via LinkedList virtual methods.**

3.3 Bytecode Example

List add

```
aload_1
ldc "A"
invokeinterface List.add:(Ljava/lang/Object;)Z
pop
```

Set add

```
aload_2
ldc "A"
invokevirtual HashMap.put:(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
pop
```

Queue poll

```
aload_3
invokevirtual LinkedList.poll:()Ljava/lang/Object;
astore_4
```

3.4 Class Initialization ()

- **ArrayList: initializes DEFAULT_CAPACITY = 10, EMPTY_ELEMENTDATA in Metaspace.**
- **HashSet: internally initializes HashMap table.**
- **LinkedList / Queue: Node objects created per element, minimal static initialization.**

4) HotSpot Optimizations

1. **ArrayList: inlined get(), add(), branch-predicted array access.**
2. **HashSet: HashMap hashing is inlined; bucket access optimized via inline caches.**
3. **LinkedList: sequential node access → limited cache locality, but escape analysis can allocate node objects on stack for short-lived queues.**

5) GC & Memory Considerations

- **ArrayList: array resizing triggers GC of old array.**
- **HashSet: underlying Node[] table resizing triggers rehashing; old table eligible for GC.**
- **Queue (LinkedList): each Node is a separate object → minor GC for nodes on poll/remove.**

Text-Based Resizing Diagram for ArrayList

```
ArrayList size 10 -> add() exceeds -> new array size 15
old elementData[] -> GC eligible
```

Text-Based Resizing Diagram for HashSet

HashSet backed by HashMap (table size 16)
put() triggers threshold -> table size doubled
rehash keys -> old table GC

6) Real-World Tie-ins

- **List: storing ordered transaction history (ArrayList for fast random access).**
- **Set: storing unique user IDs or transaction IDs (HashSet).**
- **Queue: processing real-time events or payment transactions (LinkedList FIFO, PriorityQueue for priority processing).**
- **Spring: uses List for bean injection, Set for unique configuration values.**

7) Pitfalls & Refactors

1. **Using List when uniqueness is required → use Set.**
2. **Using Set when insertion order matters → use LinkedHashSet.**
3. **Using LinkedList for large-scale random access → slow due to pointer chasing.**
4. **Refactor: abstract data access via interfaces (List, Set, Queue) and choose implementation based on access pattern.**

8) Interview Follow-ups (1-Line Answers)

1. **List vs Set? → List allows duplicates and maintains order; Set ensures uniqueness.**
2. **Queue difference? → FIFO (LinkedList) or priority-based (PriorityQueue).**
3. **ArrayList vs LinkedList in queue context? → ArrayList for random access, LinkedList for fast insert/poll at ends.**
4. **HashSet internal mechanism? → Uses HashMap internally (key -> PRESENT).**

Q5: Difference between ArrayList and LinkedList

1) Overview

Aspect	ArrayList	LinkedList
Internal Data Structure	Resizable array (`Object[] elementData`)	Doubly linked list of nodes (`Node<E>`)
Access Time	O(1) for `get(index)`	O(n) for `get(index)`
Insertion/Deletion	O(n) at arbitrary index (shift elements)	O(1) if node reference known, else O(n)
Memory Overhead	Low (array only)	High (each node stores prev/next pointer)
Iteration	Fast due to contiguous memory → good cache locality	Slower, pointer-chasing → poor CPU cache locality
Implements	List, RandomAccess	List, Deque
Use Case	Frequent read / random access	Frequent insert/delete at ends/middle

2) Java Code Example

```
import java.util.*;

public class ArrayListVsLinkedList {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();
        arrayList.add("A");
        arrayList.add("B");
        arrayList.add("C");

        List<String> linkedList = new LinkedList<>();
        linkedList.add("A");
        linkedList.add("B");
        linkedList.add("C");

        // Access example
        System.out.println("ArrayList get(1): " + arrayList.get(1));
        System.out.println("LinkedList get(1): " + linkedList.get(1));

        // Insert in middle
        arrayList.add(1, "X");
        linkedList.add(1, "X");
```

```
        System.out.println("ArrayList after insert: " + arrayList);
        System.out.println("LinkedList after insert: " + linkedList);
    }
}
```

Dry Run (Step by Step):

1. ArrayList initial: ["A", "B", "C"], size=3
2. LinkedList initial: ["A", "B", "C"], size=3
3. get(1) → ArrayList: O(1), LinkedList: traverse node → O(n)
4. Insert "X" at index 1 → ArrayList shifts elements → O(n), LinkedList adjusts pointers → O(1 if node ref known, else O(n) to find node)

Output:

```
ArrayList get(1): B
LinkedList get(1): B
ArrayList after insert: [A, X, B, C]
LinkedList after insert: [A, X, B, C]
```

3) JVM Internals

3.1 Object Layout

ArrayList

```
ArrayList object
+----------------+
| Mark Word      | // GC, lock, hash
| Klass Pointer  | // ArrayList.class
+----------------+
| int size       |
| int modCount   |
| Object[] elementData | // contiguous array in heap
+----------------+
```

LinkedList

```
LinkedList object
+----------------+
| Mark Word      |
| Klass Pointer  |
+----------------+
| int size       |
```

```
| Node<E> first  |
| Node<E> last   |
+----------------+

Node<E>
+----------------+
| Mark Word      |
| Klass Pointer  |
| E item         |
| Node<E> next   |
| Node<E> prev   |
+----------------+
```

- **ArrayList: elements stored in a contiguous memory block → better CPU cache locality.**
- **LinkedList: nodes scattered in heap → pointer chasing, slower iteration.**

3.2 Bytecode Example

ArrayList add

```
aload_1
ldc "A"
invokevirtual ArrayList.add:(Ljava/lang/Object;)Z
pop
```

LinkedList add

```
aload_2
ldc "A"
invokevirtual LinkedList.add:(Ljava/lang/Object;)Z
pop
```

Access element

- **ArrayList: aaload → direct array index**
- **LinkedList: traverse nodes via getfield next**

3.3 Class Initialization ()

- **ArrayList: DEFAULT_CAPACITY = 10, EMPTY_ELEMENTDATA initialized in Metaspace.**
- **LinkedList: minimal static init, Node class created per element.**

4) HotSpot Optimizations

1. ArrayList: `get()` inlined, loop unrolled by JIT → very fast sequential access.
2. LinkedList: node traversal may not inline; limited CPU prefetching due to scattered memory.
3. Escape Analysis: temporary nodes in LinkedList may be stack-allocated → reduce GC overhead.
4. Branch Prediction: ArrayList contiguous array → predictable; LinkedList → unpredictable branch in `next` pointer access.

5) GC & Memory Considerations

ArrayList

- `Object[] elementData` grows → old array GC candidate.
- Large arrays → promote to old generation if survives multiple GCs.

LinkedList

- Each Node is separate object → more heap overhead, frequent minor GC for short-lived nodes.

Memory Diagram

```
ArrayList: [A][B][C][X]
Contiguous array -> better cache locality

LinkedList: Node(A)->Node(X)->Node(B)->Node(C)
Scattered nodes -> pointer chasing, higher memory
```

6) Real-World Tie-ins

1. Spring: `ArrayList` used for BeanFactory lists (fast access).
2. Hibernate: `LinkedList` useful for queued events, batch insertions.
3. Payments/Fintech:
    - Transaction logs → `ArrayList` for fast read and sorting.

- FIFO event queues → LinkedList or ArrayDeque.

7) Pitfalls & Refactors

1. Using LinkedList for frequent random access → slow due to O(n).
2. Using ArrayList for frequent middle insertions → high cost due to shifting.
3. Refactor: choose collection based on access pattern, e.g., read-heavy → ArrayList, insertion-heavy → LinkedList.
4. Use composition or strategy pattern to switch collection dynamically if usage changes.

8) Interview Follow-ups (1-Line Answers)

1. Why ArrayList is faster for get()? → contiguous memory → direct index access.
2. Why LinkedList uses more memory? → Node stores prev/next pointers.
3. GC impact? → ArrayList reallocates array; LinkedList creates many small objects.
4. CPU cache impact? → ArrayList benefits from spatial locality; LinkedList does not.

Q6: Difference between Fail-Fast and Fail-Safe Iterators

1) Overview

Aspect	Fail-Fast Iterator	Fail-Safe Iterator
Definition	Throws `ConcurrentModificationException` if collection is modified during iteration	Does not throw exception, works on a cloned copy of collection
Collection Types	ArrayList, HashMap, LinkedList, TreeMap (non-concurrent)	ConcurrentHashMap, CopyOnWriteArrayList, ConcurrentLinkedQueue
Traversal	Operates directly on collection → fast,	Operates on clone or snapshot

Aspect	Fail-Fast Iterator	Fail-Safe Iterator
	O(1) access	→ slower, O(n) overhead
Modification During Iteration	Not allowed (except via iterator.remove())	Allowed, safe
Memory Overhead	Low → no extra copy	Higher → snapshot copy or internal structure
Use Case	Detect concurrent modification bugs → debugging	Thread-safe concurrent access in multi-threaded apps

2) Java Code Example

```
import java.util.*;
import java.util.concurrent.*;

public class FailFastVsFailSafe {
   public static void main(String[] args) {
      // Fail-Fast Iterator
      List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
      Iterator<String> it = list.iterator();
      try {
         while(it.hasNext()) {
            String val = it.next();
            if(val.equals("B")) list.remove(val); // modification outside iterator
         }
      } catch (ConcurrentModificationException e) {
         System.out.println("Fail-Fast Iterator Exception caught: " + e);
      }

      // Fail-Safe Iterator
      CopyOnWriteArrayList<String> cowList = new CopyOnWriteArrayList<>(Arrays.asList("A", "B", "C"));
      for(String val : cowList) {
         if(val.equals("B")) cowList.remove(val); // safe
      }
      System.out.println("Fail-Safe CopyOnWriteArrayList: " + cowList);
   }
}
```

Dry Run Example:

- **Fail-Fast (ArrayList) → remove "B" → throws ConcurrentModificationException**
- **Fail-Safe (CopyOnWriteArrayList) → remove "B" during iteration → works safely**

Output:

```
Fail-Fast Iterator Exception caught: java.util.ConcurrentModificationException
Fail-Safe CopyOnWriteArrayList: [A, C]
```

3) JVM Internals

3.1 Fail-Fast Iterator (ArrayList)

ArrayList.Itr Object Layout

```
ArrayList.Itr
+----------------+
| Mark Word      |
| Klass Pointer  |
| ArrayList<E> list |
| int cursor       | // next element index
| int lastRet      | // last returned
| int expectedModCount | // tracks collection modification
+----------------+
```

Mechanism

1. **Iterator stores expectedModCount = modCount at creation.**
2. **On each next() call:**
3. **if (modCount != expectedModCount) throw ConcurrentModificationException;**
4. **Direct access to underlying array → fast, O(1)**

3.2 Fail-Safe Iterator (CopyOnWriteArrayList)

CopyOnWriteArrayList Object Layout

```
CopyOnWriteArrayList
+----------------+
| Mark Word      |
| Klass Pointer  |
| volatile Object[] array |
+----------------+
```

Mechanism

1. Iterator captures snapshot array at creation:
2. final Object[] snapshot = array;
3. Iteration reads snapshot → safe from concurrent modifications
4. Writing (add/remove) → creates new array copy → O(n) overhead

Text Diagram

Original ArrayList: [A, B, C]
Itr cursor=0, expectedModCount=3
modCount=4 → fail-fast exception

CopyOnWriteArrayList snapshot: [A, B, C]
Iterating snapshot → modifications to actual array do not affect iteration

4) HotSpot Optimizations

- Fail-Fast:
  - next() inlined, modCount check is a single integer comparison
  - Branch prediction handles exceptions efficiently
- Fail-Safe:
  - Snapshot array → HotSpot may optimize reads via contiguous memory
  - CopyOnWrite ensures thread-safety without locks → no monitorenter/monitorexit

5) GC & Memory Considerations

- Fail-Fast: iterator object lightweight → minimal GC impact
- Fail-Safe (CopyOnWrite): snapshot array → higher memory usage, GC collects old arrays on modification
- Linked variants: snapshot may include linked nodes → more heap allocation

6) Real-World Tie-ins

1. Spring → CopyOnWriteArrayList for listener registration in multi-threaded environment

2. Payments/Fintech → concurrent processing of transactions → safe iteration using ConcurrentHashMap iterators
3. ExecutorService / Queues → fail-safe iterators avoid concurrent modification crashes

7) Pitfalls & Refactors

1. Fail-Fast: modifying collection outside iterator → ConcurrentModificationException
2. Fail-Safe: heavy memory footprint for large collections → not suitable for high-frequency modifications
3. Refactor: Choose iterator type based on read-heavy vs write-heavy workloads
4. Prefer ConcurrentHashMap or ArrayDeque for low-latency concurrent access

8) Interview Follow-ups (1-Line Answers)

1. Fail-Fast example? → ArrayList iterator, HashMap iterator
2. Fail-Safe example? → CopyOnWriteArrayList, ConcurrentHashMap iterator
3. Why fail-fast throws exception? → Detect concurrent modification bugs early
4. Memory overhead fail-safe? → Copies snapshot → higher memory use
5. When to use fail-safe? → Multi-threaded iteration with frequent reads and rare writes

Level 2 - Deep Dive (Low-Level JVM / Code Behaviour)

Q1: How does HashMap handle collisions

1) Overview

- Collision: Occurs when multiple keys have the same hash code after applying hash(key) & (n-1) in a HashMap.
- HashMap in Java 8+ uses a hybrid approach:

1. Chaining with linked lists for small buckets
2. Balanced trees (Red-Black tree) when bucket size exceeds threshold (TREEIFY_THRESHOLD = 8)

- Load Factor (0.75 default) controls resizing to reduce collisions.

Collision Handling Methods:

1. Separate Chaining → LinkedList of nodes
2. Treeify → Converts bucket to Red-Black tree when too many collisions

2) Java Code Example

```
import java.util.*;

public class HashMapCollisionExample {
   static class Key {
      int id;
      Key(int id) { this.id = id; }

      @Override
      public int hashCode() { return id % 4; } // deliberately cause collisions
      @Override
      public boolean equals(Object o) {
         if(this==o) return true;
         if(!(o instanceof Key)) return false;
         return id == ((Key)o).id;
      }
   }

   public static void main(String[] args) {
      HashMap<Key, String> map = new HashMap<>();
      map.put(new Key(1), "One");
      map.put(new Key(5), "Five"); // same bucket as 1 (1%4==5%4)
      map.put(new Key(9), "Nine"); // same bucket again
      map.put(new Key(2), "Two");

      map.forEach((k,v)->System.out.println(k.id + " -> " + v));
   }
}
```

Dry Run:

- HashMap with capacity=16
- Keys: 1,5,9 → hash % 16 → bucket 1

- Chaining in linked list: Node1 → Node5 → Node9
- Access get(5) → traverse bucket 1 → Node1.id!=5 → Node5.id==5 → return "Five"

Output (order in bucket may vary):

```
1 -> One
5 -> Five
9 -> Nine
2 -> Two
```

3) JVM Internals

3.1 Node Layout (Java 8+)

```
Node<K,V>
+----------------+
| Mark Word      |
| Klass Pointer  |
| int hash       | // cached hash
| K key          |
| V value        |
| Node<K,V> next | // linked list
+----------------+
```

- Red-Black Tree Node (TreeNode<K,V>) extends Node<K,V>

```
TreeNode<K,V>
+----------------+
| parent, left, right | // tree structure
| boolean red        |
| K key
| V value
| hash
+----------------+
```

3.2 Bucket Array Layout

- HashMap has Node<K,V>[] table (array of buckets)
- Each bucket can be null, Node, or TreeNode

Diagram:

```
table[0] -> null
table[1] -> Node(1) -> Node(5) -> Node(9)
table[2] -> Node(2)
...
```

- **Lookup:** hash(key) & (table.length - 1) → bucket index → traverse linked list or tree

4) HotSpot Optimizations

1. Inlining → get(), put() methods, hash calculations
2. Cache-friendly array access → contiguous Node[] table
3. Treeify only after threshold → avoid Red-Black overhead for small buckets
4. Lazy table initialization → Node[] table created on first put()

Bytecode Example (put simplified)

```
aload_0          // this
aload_1          // key
aload_2          // value
invokevirtual HashMap.hash(key)
dup
astore_3         // hash cached
aload_0
getfield table
iload_3
iand
aaload           // bucket
astore_4         // current node
...
```

5) GC & Memory Considerations

- Each Node<K,V> is a separate heap object → minor GC overhead
- TreeNode → more references (parent, left, right) → more memory per entry
- Table resizing → old array becomes unreachable → GC collects it

Memory Diagram

```
Bucket 1: Node(1) -> Node(5) -> Node(9)
Heap objects: 3 Node instances
Array table[16]: 16 references
```

6) Real-World Tie-ins

1. Payments/Fintech → store transactions keyed by transaction ID → collisions rare but must be handled
2. Spring → caching beans by keys, collision management ensures correctness
3. Hibernate → HashMap used in first-level cache → Red-Black trees prevent performance degradation under collision

7) Pitfalls & Refactors

1. Poor hashCode() → frequent collisions → degrade performance O(n) for linked list, O(log n) for tree
2. Resizing triggers multiple GC cycles → avoid too small initial capacity
3. Avoid mutable keys → changing key fields breaks map consistency
4. Refactor: Custom hash function, proper initial capacity → minimize collisions

8) Interview Follow-ups (1-Line Answers)

1. LinkedList vs TreeNode for collisions? → TreeNode when bucket size ≥ 8 → O(log n) lookup
2. What triggers treeify? → Bucket size ≥ TREEIFY_THRESHOLD and table length ≥ MIN_TREEIFY_CAPACITY
3. Resizing impact? → O(n) copy, GC overhead for old table
4. Mutable key issue? → If key changes after insertion → lookup may fail
5. HotSpot optimization? → System.arraycopy for resizing, method inlining for get()

Q2: Load Factor and Capacity in HashMap

1) Overview

- Capacity: Number of buckets in the HashMap (Node<K,V>[] table)
  - Default: 16
  - Must be a power of 2 → ensures hash & (capacity-1) is uniform
- Load Factor (LF): Threshold to trigger resizing
  - Default: 0.75
  - Formula to trigger resize: size > capacity * loadFactor
- Goal: Balance memory usage and lookup performance

Key Facts:

1. **Default capacity = 16**
2. **Default load factor = 0.75 → 75% occupancy triggers resize**
3. **Higher LF → less memory, more collisions**
4. **Lower LF → more memory, fewer collisions**

2) Java Code Example

```
import java.util.*;

public class HashMapLoadFactorExample {
   public static void main(String[] args) {
      HashMap<Integer, String> map = new HashMap<>(4, 0.75f);
      map.put(1, "One");
      map.put(2, "Two");
      map.put(3, "Three"); // capacity = 4, 4*0.75 = 3 → next put triggers resize
      map.put(4, "Four");  // resize occurs
      System.out.println("Map: " + map);
      System.out.println("Map size: " + map.size());
   }
}
```

Dry Run:

- **Initial table: size 4, load factor 0.75 → threshold = 3**
- **put(1) → size=1 → no resize**
- **put(2) → size=2 → no resize**
- **put(3) → size=3 → threshold reached → next insertion triggers resize**
- **put(4) → resize table → new capacity = 8, rehash all keys**

3) JVM Internals

3.1 Table Layout

```
HashMap
+----------------+
| Mark Word      |
| Klass Pointer  |
| int size       | // current number of entries
| int threshold  | // capacity * loadFactor
```

| float loadFactor |
| Node<K,V>[] table |
+----------------+

- Threshold Calculation:
- this.threshold = (int)(initialCapacity * loadFactor);
- Resize Mechanism:
  - Allocate new Node[] newTable = new Node[newCapacity]
  - Rehash all entries → hash & (newCapacity - 1)
  - Old array becomes garbage → eligible for GC

Diagram:

Old table[4]: [0]->1, [1]->2, [2]->3
Resize → table[8]
Rehash:
1 → bucket 1
2 → bucket 2
3 → bucket 3
4 → bucket 4

4) HotSpot Optimizations

1. Array copy: System.arraycopy() used for resizing → native, highly optimized
2. Lazy initialization: Table allocated only on first put → saves memory
3. Inlining: get() and put() are frequently inlined → faster hash calculations
4. Branch prediction: hash & (n-1) ensures contiguous bucket access → cache-friendly

5) GC & Memory Considerations

- Old table array after resize → eligible for GC
- Linked list/tree nodes → still referenced, not collected
- Large initial capacity reduces frequent resizes → reduces GC pressure
- Excessively low load factor → more memory allocated → possible cache misses

Memory Diagram:

Before resize:
table[4] -> Node1, Node2, Node3
After resize:

table[8] -> Node1, Node2, Node3, Node4
Old table[4] -> GC eligible

6) Real-World Tie-ins

1. Payments/Fintech: Transaction maps → high initial capacity avoids resize during peak hours
2. Spring/Hibernate caches: Pre-sized HashMaps reduce GC churn
3. High-throughput systems: Tuned load factor ensures predictable latency

7) Pitfalls & Refactors

1. Default load factor 0.75 → may trigger resize under sudden spikes → plan initial capacity
2. Very high load factor → more collisions → degrade O(1) to O(n)
3. Mutable keys → may affect bucket placement after resize
4. Refactor: Use ConcurrentHashMap for multi-threaded scenarios with controlled concurrency

8) Interview Follow-ups (1-Line Answers)

1. Default load factor? → 0.75
2. Default capacity? → 16
3. Resize trigger? → size > capacity * loadFactor
4. Why power of 2 capacity? → ensures uniform hash spread using hash & (capacity-1)
5. Effect of higher load factor? → Less memory, more collisions

Q3: How does HashSet Internally Use HashMap

1) Overview

- HashSet is a Set implementation in Java that does not allow duplicates.
- Internal mechanism: HashSet uses HashMap internally. Each element of the set is stored as a key in the backing HashMap, with a dummy value.

- This allows HashSet to leverage HashMap's O(1) average lookup, insert, and delete operations.

Key Points:

1. No duplicates: HashMap key uniqueness enforces this.
2. Value placeholder: Usually a static `private static final Object PRESENT = new Object();`
3. Underlying structure: Node[], linked list/tree depending on collisions.

2) Java Code Example

```
import java.util.*;

public class HashSetInternalExample {
   public static void main(String[] args) {
      HashSet<String> set = new HashSet<>();
      set.add("Apple");
      set.add("Banana");
      set.add("Apple"); // duplicate, ignored

      for(String s : set) {
         System.out.println(s);
      }

      // Internals via reflection
      try {
         java.lang.reflect.Field mapField = HashSet.class.getDeclaredField("map");
         mapField.setAccessible(true);
         HashMap<?, ?> internalMap = (HashMap<?, ?>) mapField.get(set);
         System.out.println("Internal map: " + internalMap);
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Dry Run:

- set.add("Apple") → calls map.put("Apple", PRESENT)
- set.add("Banana") → calls map.put("Banana", PRESENT)
- set.add("Apple") → map.put("Apple", PRESENT) → key exists → ignored

Output example:

```
Apple
Banana
Internal map: {Apple=PRESENT, Banana=PRESENT}
```

3) JVM Internals

3.1 Object Layout

HashSet Object

```
HashSet
+----------------+
| Mark Word      |
| Klass Pointer  |
| HashMap map    | // backing map
+----------------+
```

HashMap Node for elements

```
Node<K,V>
+----------------+
| Mark Word      |
| Klass Pointer  |
| int hash       |
| K key          | // element in HashSet
| V value        | // PRESENT object
| Node<K,V> next |
+----------------+
```

3.2 Memory Layout Diagram

```
HashSet
 |
 v
HashMap table[16]
bucket[0] -> null
bucket[1] -> Node("Apple", PRESENT)
bucket[2] -> Node("Banana", PRESENT)
...
```

- **Lookup/add/remove → delegated to HashMap operations (`put`, `containsKey`, `remove`)**
- **`PRESENT` is a single shared object → minimal memory overhead**

4) HotSpot Optimizations

1. **Inlining:** HashSet.add() **inlined to** HashMap.put()
2. **Single static value (**PRESENT**) → reduces object allocation**
3. **Contiguous Node[] array → better cache locality**
4. **Treeify buckets → if collision > TREEIFY_THRESHOLD**

Bytecode for add() **simplified:**

```
aload_0       // this
getfield map
aload_1       // element
getstatic PRESENT
invokevirtual HashMap.put(key, PRESENT)
pop           // ignore return value
```

5) GC & Memory Considerations

- **Only nodes and backing array are heap objects**
- PRESENT **is a singleton → avoids multiple allocations**
- **Resizing HashMap → old Node[] array becomes GC-eligible**

Diagram:

```
HashSet -> HashMap
      table[16] -> Node("Apple") -> Node("Banana")
Old table after resize -> GC
```

6) Real-World Tie-ins

1. **Spring: Using HashSet for unique bean names → leverages backing HashMap efficiently**
2. **Payments/Fintech: Deduplicating transaction IDs or user IDs → HashSet guarantees uniqueness**
3. **Hibernate: Set collections in entity mapping → internally use HashSet for uniqueness**

7) Pitfalls & Refactors

1. Mutable elements: Changing hashCode/equals after insertion → can break set behavior
2. Memory overhead: Large HashSet → backing HashMap table + Node objects
3. Refactor: If iteration is read-heavy → consider CopyOnWriteArraySet
4. TreeSet alternative: If ordering is required, use TreeSet (sorted, log(n) ops)

8) Interview Follow-ups (1-Line Answers)

1. How does HashSet prevent duplicates? → Uses HashMap keys, duplicates ignored
2. Value stored in backing map? → Singleton PRESENT object
3. Time complexity? → O(1) average for add, contains, remove
4. Why use HashMap internally? → Efficient lookup, insertion, deletion
5. TreeSet vs HashSet? → TreeSet maintains sorted order, O(log n) operations

Q4: How does TreeMap Maintain Sorted Order

1) Overview

- TreeMap is a SortedMap implementation in Java.
- Internally, it uses a Red-Black Tree (self-balancing binary search tree).
- Key properties:
    1. Maintains natural ordering (Comparable) or custom Comparator.
    2. Insertion, deletion, lookup → O(log n).
    3. Guarantees sorted traversal via in-order tree traversal.

Key Points:

- TreeMap does not use HashMap (unlike HashSet/HashMap).
- Balancing rules ensure tree height ≤ 2 * log(n+1) → keeps operations efficient.

2) Java Code Example

```
import java.util.*;

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

```
        TreeMap<Integer, String> treeMap = new TreeMap<>();
        treeMap.put(5, "Five");
        treeMap.put(1, "One");
        treeMap.put(3, "Three");
        treeMap.put(2, "Two");

        // Natural ordering of keys
        treeMap.forEach((k, v) -> System.out.println(k + " -> " + v));

        // Custom comparator (reverse order)
        TreeMap<Integer, String> reverseMap = new TreeMap<>(Comparator.reverseOrder());
        reverseMap.putAll(treeMap);
        reverseMap.forEach((k, v) -> System.out.println(k + " -> " + v));
    }
}
```

Dry Run:

- Keys inserted: 5,1,3,2
- Red-Black Tree balances after each insertion → in-order traversal: 1,2,3,5
- Reverse comparator → 5,3,2,1

Output:

```
1 -> One
2 -> Two
3 -> Three
5 -> Five
5 -> Five
3 -> Three
2 -> Two
1 -> One
```

3) JVM Internals

3.1 Node Layout

TreeMap Entry (TreeNode<K,V>):

```
TreeNode<K,V>
+----------------+
| Mark Word      |
| Klass Pointer  |
| K key          |
| V value        |
| TreeNode<K,V> left, right, parent |
```

```
| boolean red    | // red-black property
+----------------+
```

TreeMap Object Layout

```
TreeMap
+----------------+
| Mark Word      |
| Klass Pointer  |
| Comparator cmp | // optional
| int size       |
| Entry<K,V> root| // root of red-black tree
+----------------+
```

3.2 Memory Layout Diagram

```
        root(3)
        /   \
     (1)     (5)
       \
        (2)
```

- Red-Black rules ensure:
  1. No two consecutive red nodes
  2. Black height same from root to leaves
  3. Balance maintained → O(log n) operations

4) HotSpot Optimizations

1. Inlining → put(), get(), and rotateLeft/rotateRight()
2. Branch prediction → tree traversal uses predictable patterns
3. Escape analysis → TreeNode allocation optimized if local to method
4. Cache locality → Linked tree nodes can cause pointer chasing → minor overhead vs arrays

5) GC & Memory Considerations

- Each TreeNode → separate heap object → more GC overhead than ArrayList/HashMap
- Long-lived TreeMap → more stable references → fewer minor GCs
- Rebalancing → rotates nodes, no new object allocation → memory-efficient

6) Real-World Tie-ins

1. Spring → TreeMap for sorted beans by name or priority
2. Payments/Fintech → Sorted transaction timestamps → O(log n) insertions and sorted access
3. Scheduler systems → TreeMap used in delay queues, task ordering

7) Pitfalls & Refactors

1. Mutable keys: Changing key fields used in comparator → can break tree invariants
2. Comparator null: Null comparator → uses natural ordering → requires keys implement Comparable
3. Refactor: For high read-heavy use → use NavigableMap views for subMap/headMap operations

8) Interview Follow-ups (1-Line Answers)

1. Internal structure? → Red-Black Tree
2. Time complexity? → O(log n) for get, put, remove
3. Natural vs custom ordering? → Natural: Comparable; Custom: Comparator
4. Why not HashMap? → Needs sorted order → hash-based lookup doesn't preserve order
5. Memory overhead vs HashMap? → Higher due to TreeNodes and parent/child pointers

Q5: How Does LinkedHashMap Maintain Insertion/Access Order

1) Overview

- LinkedHashMap extends HashMap and preserves the order of elements:

1. **Insertion-order (default) → elements returned in the order they were inserted.**
2. **Access-order (optional via constructor) → elements reordered on `get()` or `put()` → useful for LRU caches.**

- **Maintains a doubly-linked list of entries alongside HashMap buckets.**

Key Facts:

- **Internally: HashMap table + doubly-linked list**
- **Entry class: `LinkedHashMap.Entry<K,V>` extends `HashMap.Node<K,V>`**
- **Access-order requires move-to-end logic on reads**

2) Java Code Example

```
import java.util.*;

public class LinkedHashMapExample {
   public static void main(String[] args) {
      // Insertion order
      LinkedHashMap<Integer, String> insertionMap = new LinkedHashMap<>();
      insertionMap.put(1, "One");
      insertionMap.put(2, "Two");
      insertionMap.put(3, "Three");
      insertionMap.forEach((k,v) -> System.out.println(k + " -> " + v));

      // Access order
      LinkedHashMap<Integer, String> accessMap = new LinkedHashMap<>(16, 0.75f, true);
      accessMap.put(1, "One");
      accessMap.put(2, "Two");
      accessMap.put(3, "Three");
      accessMap.get(2); // access triggers reorder
      accessMap.forEach((k,v) -> System.out.println(k + " -> " + v));
   }
}
```

Dry Run:

- **Insertion order: 1,2,3**
- **Access order after `get(2)`: 1,3,2**

3) JVM Internals – Deep Dive

3.1 Object Layout

LinkedHashMap

```
LinkedHashMap
+----------------+
| Mark Word      |
| Klass Pointer  |
| HashMap table  |
| int size       |
| float loadFactor |
| LinkedHashMap.Entry<K,V> head, tail |
+----------------+
```

Entry Node

```
LinkedHashMap.Entry<K,V> extends HashMap.Node<K,V>
+----------------+
| Mark Word      |
| Klass Pointer  |
| int hash       |
| K key          |
| V value        |
| Node<K,V> next | // bucket next
| Entry<K,V> before, after | // doubly-linked list pointers
+----------------+
```

Text Diagram of Structure:

```
Hash Table Buckets:
bucket[1] -> Entry(1) -> Entry(2)
bucket[2] -> Entry(3)

Doubly-linked List (Insertion order):
head -> Entry(1) <-> Entry(2) <-> Entry(3) <- tail
Access order:
head -> Entry(1) <-> Entry(3) <-> Entry(2) <- tail (after get(2))
```

3.2 Bytecode & vtable

- get() → HashMap lookup → if accessOrder=true → afterNodeAccess(node) invoked
- afterNodeAccess(node) → unlink and move node to tail of doubly-linked list
- Vtable dispatch: LinkedHashMap.afterNodeAccess overrides HashMap default (empty)
- order: static fields like serialVersionUID initialized first
- Invoke bytecode: invokevirtual afterNodeAccess → HotSpot inlineable

4) HotSpot Optimizations

1. **Method inlining:** `get()`, `put()`, `afterNodeAccess` → JIT inlined
2. **Branch prediction:** doubly-linked list manipulation predictable → cache-friendly
3. **Escape analysis:** Local `Node` allocation → scalar replacement possible
4. **Array access:** Hash table uses `hash & (table.length - 1)` → contiguous array → cache-efficient
5. **Deoptimization:** Rarely happens if class hierarchy changes (LinkedHashMap subclassing)

5) GC & Memory Considerations

- Doubly-linked list adds 2 references per entry → slight memory overhead
- Old table arrays after resize → GC-eligible
- Persistent LinkedHashMap for LRU cache → long-lived objects → minor GC optimizations

Text Diagram of Memory

```
Heap:
Entry1 -> key,value,next,before,after
Entry2 -> key,value,next,before,after
Entry3 -> key,value,next,before,after
Table array[16] -> references to bucket heads
```

6) Real-World Tie-ins

1. **LRU Cache:** `LinkedHashMap(accessOrder=true)` + `removeEldestEntry()` → ideal for caching API responses
2. **Spring Boot Caching:** Used for in-memory caches of recent beans or HTTP sessions
3. **Payments/Fintech:** Track recent transactions or requests in insertion or access order

7) Pitfalls & Refactors

1. Memory overhead: Doubly-linked list adds 2 references per node → significant for millions of entries
2. Mutable keys: Changing hashCode/equals after insertion → breaks bucket placement
3. Access-order mistakes: Forgetting `afterNodeAccess(node)` → wrong ordering
4. Refactor: Use composition + `LinkedHashMap` for LRU → decouples eviction strategy

8) Interview Follow-ups (1-Line Answers)

1. Insertion vs Access order? → `accessOrder=false` default → insertion order; `true` → reorder on access
2. Memory overhead per entry? → +2 references for before/after
3. Eviction support? → Override `removeEldestEntry()`
4. Time complexity? → O(1) average for get/put, O(n) for iteration
5. Difference from HashMap? → Maintains order with doubly-linked list, HashMap does not

Q6: How Does ArrayList Grow Its Capacity Internally

1) Overview

- ArrayList is a resizable array implementation of the `List` interface.
- Key Properties:
  1. Backed by an Object[] array.
  2. Default capacity = 10.
  3. Grows dynamically using `grow()` method when `size >= capacity`.
- Growth factor: Typically `newCapacity = oldCapacity + (oldCapacity >> 1)` → ~1.5×

Goal: Balance memory usage vs resizing overhead.

2) Java Code Example

```
import java.util.*;
```

```
public class ArrayListGrowthExample {
   public static void main(String[] args) {
      ArrayList<Integer> list = new ArrayList<>(4);
      list.add(1);
      list.add(2);
      list.add(3);
      list.add(4); // threshold reached
      list.add(5); // triggers growth

      System.out.println("ArrayList: " + list);
      System.out.println("Size: " + list.size());
   }
}
```

Dry Run with Capacity:

- Initial capacity = 4
- add(1-4) → no growth
- add(5) → grow() called → new capacity = 4 + (4 >> 1) = 6
- Internal array copied → System.arraycopy(oldArray, 0, newArray, 0, oldCapacity)

3) JVM Internals – Deep Dive

3.1 Object Layout

ArrayList Object:

```
ArrayList
+----------------+
| Mark Word      |
| Klass Pointer  |
| Object[] elementData | // backing array
| int size       |
+----------------+
```

Backing array layout in heap:

```
elementData[capacity]
index: 0   1   2   3   4   5
value: 1   2   3   4   5   null
```

- Array is contiguous in heap → good cache locality
- Node objects (if storing references) → separately allocated → referenced by array

3.2 Growth Bytecode

- **add(E e):**

```
public boolean add(E e) {
   ensureCapacityInternal(size + 1);
   elementData[size++] = e;
   return true;
}
```

- **ensureCapacityInternal → checks threshold → calls grow()**
- **grow():**

```
int newCapacity = oldCapacity + (oldCapacity >> 1);
elementData = Arrays.copyOf(elementData, newCapacity);
```

- **Invoke bytecodes: invokevirtual ensureCapacityInternal, invokestatic Arrays.copyOf, etc.**

4) HotSpot Optimizations

1. **Inlining: add() and ensureCapacityInternal() → JIT inlined for hot loops**
2. **Array copy: System.arraycopy() → native memcpy, highly optimized**
3. **Escape analysis: If ArrayList and elements are local → scalar replacement possible**
4. **Branch prediction: Growth happens rarely → predictable branch → minimal misprediction**
5. **Cache locality: Contiguous array → excellent CPU cache behavior**

5) GC & Memory Considerations

- **Old array → becomes GC-eligible after resize**
- **Large lists → frequent resizing → temporary GC pressure**
- **Pre-sizing large ArrayList → reduces array copy overhead → reduces GC pressure**

Memory Diagram:

Before growth:
elementData[4] -> 1,2,3,4

After growth:

elementData[6] -> 1,2,3,4,5,null
Old array[4] -> GC eligible

6) Real-World Tie-ins

1. Spring Boot: Lists of beans, request params → pre-sizing prevents repeated copying
2. Payments/Fintech: Transaction batch processing → pre-size ArrayList to reduce latency spikes
3. High-throughput systems: Avoids multiple System.arraycopy → reduces minor GC

7) Pitfalls & Refactors

1. Frequent small additions: Triggers repeated resizing → $O(n^2)$ copies in worst case
2. Huge initial capacity: Wastes memory → may increase GC pressure
3. Refactor: For predictable size → `new ArrayList<>(expectedSize)`
4. Alternative: `LinkedList` if frequent mid-list insertions

8) Interview Follow-ups (1-Line Answers)

1. Default capacity? → 10
2. Growth formula? → `newCapacity = oldCapacity + (oldCapacity >> 1)` (~1.5×)
3. Time complexity of add()? → O(1) amortized, O(n) worst-case on resize
4. Why contiguous array? → Fast index access, good cache locality
5. GC impact? → Old arrays become GC-eligible after resize

Q7: How Does LinkedList Implement Nodes Internally

1) Overview

- LinkedList is a doubly-linked list implementation of `List` and `Deque`.
- Key properties:
    1. Each element is stored in a Node object (`LinkedList.Node<E>`).
    2. Head and tail references track the first and last elements.

3. **Supports O(1) insertions/deletions at both ends, but O(n) traversal.**

Difference from ArrayList:

- **ArrayList → contiguous array → fast random access, slower insert/delete in middle**
- **LinkedList → linked nodes → slower random access, fast insert/delete**

2) Java Code Example

```
import java.util.*;

public class LinkedListNodeExample {
   public static void main(String[] args) {
      LinkedList<String> list = new LinkedList<>();
      list.add("A"); // addFirst
      list.add("B"); // addLast
      list.add("C");
      list.add(1, "D"); // insert at index 1

      System.out.println("LinkedList: " + list);
   }
}
```

Dry Run with Node Layout:

- **Initial: A → Node(head="A", prev=null, next=null)**
- **Add B: A <-> B**
- **Add C: A <-> B <-> C**
- **Insert D at index 1: A <-> D <-> B <-> C**

3) JVM Internals – Deep Dive

3.1 Object Layout

LinkedList

```
LinkedList
+----------------+
| Mark Word      |
| Klass Pointer  |
```

```
| Node<E> first  |
| Node<E> last   |
| int size       |
+----------------+
```

Node

```
LinkedList$Node<E>
+---------------+
| Mark Word     |
| Klass Pointer |
| E item        |
| Node<E> next  |
| Node<E> prev  |
+---------------+
```

- **Doubly-linked nodes allow fast insert/delete by updating prev and next references.**

Text Diagram:

```
head -> [A | prev=null | next=D]
        [D | prev=A | next=B]
        [B | prev=D | next=C]
tail -> [C | prev=B | next=null]
```

3.2 Bytecode & vtable

- **add(E e) → calls linkLast(e)**
- **linkLast(E e):**

```
Node<E> newNode = new Node<>(last, e, null);
if (last == null)
   first = newNode;
else
   last.next = newNode;
last = newNode;
size++;
```

- **Vtable dispatch: LinkedList.add() → overridden from AbstractSequentialList**
- **Invoke bytecodes: new Node, putfield next, putfield prev**

4) HotSpot Optimizations

1. **Method inlining:** `add()`, `linkFirst()`, `linkLast()`
2. **Escape analysis:** Local Node allocations → can be stack-allocated in tight loops
3. **Branch prediction:** `if (last == null)` predictable for empty/non-empty list
4. **Cache locality:** Poor vs ArrayList → nodes scattered in heap → pointer chasing
5. **Deoptimization:** Rare, mostly when subclassing LinkedList

5) GC & Memory Considerations

- Each Node → separate object → more memory overhead than array
- Long-lived LinkedList → sustained references → minor GC pressure
- Frequent additions/removals → temporary objects → transient GC activity

Memory Diagram

```
Heap:
Node1 -> item=A, prev=null, next=Node2
Node2 -> item=D, prev=Node1, next=Node3
Node3 -> item=B, prev=Node2, next=Node4
Node4 -> item=C, prev=Node3, next=null

LinkedList.first -> Node1
LinkedList.last -> Node4
```

6) Real-World Tie-ins

1. **Spring:** LinkedList used in `LinkedBlockingQueue` for task queues
2. **Payments/Fintech:** Maintain FIFO order in transaction logs or in-memory queues
3. **LRU caches:** Doubly-linked list for eviction (often paired with HashMap)

7) Pitfalls & Refactors

1. **Random access:** `get(index)` → O(n), avoid for large lists
2. **Memory overhead:** Each element has extra two references (prev/next)
3. **Refactor:** Use `ArrayList` if mostly read-access, use `LinkedList` if frequent insert/delete
4. **Composition:** LinkedList + HashMap → LRU cache pattern

8) Interview Follow-ups (1-Line Answers)

1. Node type? → Doubly-linked `LinkedList$Node<E>`
2. Insertion complexity at ends? → O(1)
3. Traversal complexity by index? → O(n)
4. Memory overhead vs ArrayList? → Higher due to prev/next references
5. Use-case for LinkedList? → Queues, stacks, LRU caches, frequent insert/delete

Q8: How Are Fail-Fast Iterators Implemented in Java Collections

1) Overview

- Fail-fast iterators immediately throw a `ConcurrentModificationException` (CME) if the collection is modified structurally (add/remove elements) during iteration, except via the iterator itself.
- Used in ArrayList, HashMap, LinkedHashMap, etc.
- Structural modification: any change that alters the size of the collection.

Key Facts:

1. Achieved using a modCount field in the collection.
2. Iterator caches the expectedModCount during creation.
3. On each `next()` or `hasNext()`, iterator checks `expectedModCount == modCount`.
4. If unequal → `ConcurrentModificationException` is thrown.

2) Java Code Example

```
import java.util.*;

public class FailFastExample {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>();
      list.add("A");
      list.add("B");
      list.add("C");

      Iterator<String> it = list.iterator();
```

```
        while (it.hasNext()) {
            String val = it.next();
            if ("B".equals(val)) {
                list.remove(val); // triggers fail-fast
            }
        }
    }
}
```

Output:

```
Exception in thread "main" java.util.ConcurrentModificationException
```

Dry Run:

- modCount = 3 initially
- Iterator caches expectedModCount = 3
- next() on "A" → expectedModCount == modCount → OK
- remove("B") → modCount increments to 4
- Next next() → expectedModCount != modCount → CME

3) JVM Internals – Deep Dive

3.1 Object Layout

ArrayList

```
ArrayList
+----------------+
| Mark Word      |
| Klass Pointer  |
| Object[] elementData |
| int size       |
| int modCount   |  // structural modification counter
+----------------+
```

Iterator

```
ArrayList$Itr
+----------------+
| Mark Word      |
| Klass Pointer  |
| int cursor     | // current index
| int lastRet    | // last returned index
```

```
| int expectedModCount | // snapshot of modCount
| ArrayList<E> list
+----------------+
```

Text Diagram:

```
ArrayList modCount = 3
Iterator expectedModCount = 3
Iteration step:
cursor=0, lastRet=-1
check: expectedModCount == modCount
```

3.2 Bytecode & vtable

- **next():**

```
public E next() {
   checkForComodification();
   int i = cursor;
   if (i >= size)
      throw new NoSuchElementException();
   Object[] elementData = ArrayList.this.elementData;
   if (i >= elementData.length)
      throw new ConcurrentModificationException();
   cursor = i + 1;
   return (E) elementData[lastRet = i];
}
```

- **checkForComodification():**

```
final void checkForComodification() {
   if (modCount != expectedModCount)
      throw new ConcurrentModificationException();
}
```

- **Invoke bytecodes:** `getfield modCount`, `if_icmpne` → **JVM throws CME**
- **Vtable:** `next()` **invoked virtually, JIT can inline**

4) HotSpot Optimizations

1. **Inlining:** `next()` + `checkForComodification()` **inlined in hot loops**
2. **Branch prediction:** `expectedModCount == modCount` **true most of the time → predicted branch**
3. **Escape analysis: Iterator object can be stack-allocated if local and short-lived**

4. Deoptimization: Rare, only if CME path executed
5. Array bounds check elimination: JIT can remove in predictable iteration loops

5) GC & Memory Considerations

- Iterator is short-lived → minor GC pressure
- ArrayList nodes remain in heap → no extra references besides iterator
- Multiple iterators → multiple small objects → can benefit from escape analysis

Memory Diagram

```
Heap:
ArrayList -> elementData[3] = {A,B,C}, modCount=3
Iterator -> expectedModCount=3, cursor=0
```

6) Real-World Tie-ins

1. Spring Collections: Used in internal property maps → detects concurrent bean modification
2. Payments/Fintech: Fail-fast prevents hidden bugs when iterating large transaction batches
3. Concurrency Warning: Use `ConcurrentHashMap` or `CopyOnWriteArrayList` for thread-safe iteration

7) Pitfalls & Refactors

1. Multi-threading: Fail-fast is not a thread-safe mechanism, only detects concurrent structural modification
2. Modification via iterator: Must use `iterator.remove()` → safe
3. Refactor: For concurrent access → use `Collections.synchronizedList` or `CopyOnWriteArrayList`
4. Performance: High-frequency modification → frequent CMEs → avoid

8) Interview Follow-ups (1-Line Answers)

1. Mechanism? → modCount in collection + expectedModCount in iterator
2. Thread-safe? → No, only detects concurrent modification
3. Time complexity? → next() O(1) for ArrayList
4. Which collections support fail-fast? → ArrayList, HashMap, LinkedHashMap, HashSet, etc.
5. Alternative for safe multi-threaded iteration? → ConcurrentHashMap, CopyOnWriteArrayList

Q9: How Does Java Calculate Hash Codes for Keys in Collections

1) Overview

- Hash code is an integer value used by collections like HashMap, HashSet, ConcurrentHashMap to determine bucket location.
- The goal is even distribution to reduce collisions.
- Default hashCode():
    1. From Object: based on object memory address (identity hash).
    2. Overridden in String, Integer, Long, UUID, etc. for logical equality.
- Hash spreading: Java 8+ applies bitwise transformations on hashCode() to reduce collisions in table indices.

2) Java Code Example

```
import java.util.*;

public class HashCodeExample {
   public static void main(String[] args) {
      String key1 = "Hello";
      String key2 = "World";

      HashMap<String, Integer> map = new HashMap<>();
      map.put(key1, 1);
      map.put(key2, 2);

      System.out.println("HashMap bucket index for key1: " +
         (map.hash(key1) & (16 - 1))); // assuming table size 16
   }
}
```

Dry Run:

- "Hello".hashCode() = 69609650
- HashMap uses:

```
static final int hash(Object key) {
   int h;
   return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

- Spread bits: 69609650 ^ (69609650 >>> 16) = 69597618
- Index for table of size 16: 69597618 & (16-1) = 2

3) JVM Internals – Deep Dive

3.1 Object Layout

String

```
String
+----------------+
| Mark Word      |
| Klass Pointer  |
| int hash       | // cached hash value
| char[] value   |
| int offset
| int count
+----------------+
```

HashMap Node

```
Node<K,V>
+----------------+
| Mark Word      |
| Klass Pointer  |
| int hash       |
| K key
| V value
| Node<K,V> next
+----------------+
```

- hashCode() may be cached (e.g., String.hash) to avoid recomputation

3.2 Bit Spreading

- **Formula:** `(h = key.hashCode()) ^ (h >>> 16)`
- **Ensures high and low bits influence bucket index, reduces collisions for poorly distributed hashes**

3.3 Bytecode & vtable

- `key.hashCode()` **→ virtual method call → JIT can inline**
- **XOR operation → lightweight** `ixor` **bytecode**
- **Table index calculation:** `i = hash & (table.length - 1)` **→ bitmask instead of modulo**

4) HotSpot Optimizations

1. **HashCode caching: String caches computed hash in** `hash` **field**
2. **Inlining:** `hash()` **+** `hashCode()` **inlined by JIT**
3. **Branch elimination: If key != null, skip null check**
4. **Escape analysis: Local key object in** `putIfAbsent` **→ stack allocation**
5. **Deoptimization: Rare, mostly on class redefinition affecting** `hashCode()`

5) GC & Memory Considerations

- `hashCode()` **itself doesn't allocate objects**
- **Cached hash fields reduce CPU recomputation, no extra GC pressure**
- **Avoid custom hashCode creating temporary objects frequently → reduce GC overhead**

Heap Diagram (simplified):

```
Key "Hello" -> String object -> char[] value
Node -> hash=69597618, key="Hello", value=1
Node -> next=null
HashMap table[16]
```

6) Real-World Tie-ins

1. Spring / Hibernate: hashCode() critical in entity caching (HashSet, HashMap)
2. Payments: Transaction maps → avoid hash collisions for performance
3. ConcurrentHashMap: Hash spreading improves multithreaded bin distribution

7) Pitfalls & Refactors

1. Poor hashCode() implementations → collision → degrade to O(n) per bucket
2. Mutable keys: Changing fields used in hashCode after insertion → broken map
3. Refactor: Override hashCode() + equals() consistently
4. Composition: Use Immutable keys (String, UUID) in maps to prevent logic errors

8) Interview Follow-ups (1-Line Answers)

1. Default hashCode for Object? → Identity-based, memory address
2. String hash cached? → Yes, stored in hash field
3. Why XOR high bits? → Better bucket distribution
4. Mutable key problem? → Can break lookup
5. Time complexity of get()? → O(1) avg, O(n) worst-case if collisions

Q10: How Does Hash Spreading / Hash Function Design Affect HashMap Performance

1) Overview

- Goal of hash spreading: Ensure even distribution of keys across HashMap buckets to reduce collisions.
- Impact on performance: Poorly distributed hash codes → high collision chains → O(n) lookups, thread contention in concurrent maps.
- Java 8+ HashMap: Uses a combination of:
    1. key.hashCode()
    2. Bitwise XOR with shifted hash: (h = key.hashCode()) ^ (h >>> 16)
- Treeification: If bucket exceeds threshold (default 8), the linked list becomes a red-black tree → O(log n) lookups

2) Java Code Example

```
import java.util.*;

public class HashSpreadExample {
   public static void main(String[] args) {
      HashMap<Integer, String> map = new HashMap<>(16);
      for (int i = 0; i < 16; i++) {
         map.put(i, "Val" + i);
      }

      // Print bucket indices
      map.forEach((k,v) -> {
         int hash = k.hashCode();
         int spread = hash ^ (hash >>> 16);
         int index = spread & (map.tableSize() - 1); // conceptual table size
         System.out.println(k + " -> bucket " + index);
      });
   }
}
```

Dry Run (Table size = 16):

- k=3 → hash=3 → spread=3^(3>>>16)=3 → bucket=3&15=3
- k=19 → hash=19 → spread=19^(19>>>16)=19 → bucket=19&15=3
- Observation: Without spreading, higher bits ignored → collisions in small tables

3) JVM Internals – Deep Dive

3.1 Object Layout

HashMap Node

```
Node<K,V>
+----------------+
| Mark Word      |
| Klass Pointer  |
| int hash       |
| K key
| V value
| Node<K,V> next
+----------------+
```

- hash field stores pre-processed (spread) hash, used to determine bucket

3.2 Hash Spreading Mechanism

- spread(h) = h ^ (h >>> 16)
- Rationale: Low 16 bits often similar → XOR with high 16 bits to avoid collisions
- JVM executes:
    1. getfield hashCode() → invokevirtual
    2. Bitwise XOR (ixor) and shift (ishr)
    3. AND with table.length-1

Text Diagram:

```
hashCode: 0x12345678
>>>16  : 0x00001234
XOR    : 0x1234444C  // spread hash
index  : 0x1234444C & 15 = 12
```

3.3 Treeification

- If collisions ≥ 8 → convert linked list → TreeNode (red-black tree)
- Node layout:

```
TreeNode<K,V> extends Node<K,V>
+ left, right, parent, red
```

- Lookup becomes O(log n) for high collision buckets

4) HotSpot Optimizations

1. Inlining: spread() + hashCode() inlined in get() and put()
2. Branch prediction: Collisions rare → predicts linked list traversal skips
3. Escape analysis: Temporary Node allocation → stack or thread-local allocation
4. Deoptimization: Rare if table resized during hot loop
5. Cache locality: Treeified bins improve CPU cache usage for large collision chains

5) GC & Memory Considerations

- Linked list nodes → heap objects → GC pressure
- TreeNodes → extra references (left, right, parent) → more memory per bin

- **Large hash tables with poor hash distribution → long chains → memory fragmentation**

Heap Diagram (simplified):

```
table[16]
bin[3] -> Node(key1) -> Node(key17) -> Node(key33) -> TreeNode(key49,...)
```

6) Real-World Tie-ins

1. **Spring Boot: Entity cache maps → hash spreading prevents hotspot bins**
2. **Payments/Fintech: Transaction maps → uneven hash → slow lookups in high-volume datasets**
3. **ConcurrentHashMap: Spreading reduces contention on bins in multithreaded environments**

7) Pitfalls & Refactors

1. **Poor custom hashCode: Collisions → performance degrades to O(n)**
2. **Mutable keys: Changing hashCode after insertion → broken map**
3. **Refactor: Ensure `hashCode()` is well-distributed, immutable keys**
4. **Composition/Strategy: Use treeification threshold to optimize high-collision buckets**

8) Interview Follow-ups (1-Line Answers)

1. **Why XOR with high bits? → To distribute hash better across table**
2. **What is treeification? → Converts long collision chains to red-black tree**
3. **Impact on performance? → Reduces O(n) to O(log n) for bad hash distributions**
4. **Spread needed in small tables? → Yes, prevents clustering in buckets**
5. **CHM vs HashMap? → CHM also uses hash spreading but adds CAS/bin-level locking**

Q11: How Do Rehashing and Resizing in HashMap Impact Performance and Memory

1) Overview

- HashMap resizing occurs when the number of entries exceeds capacity × loadFactor (default load factor 0.75).
- Rehashing: Process of redistributing entries into a new, larger table.
- Impact:
  1. Performance: O(n) operation for the resize; triggers CPU and memory usage spike.
  2. Memory: New array allocated; old table becomes GC-eligible after transfer.
- Java 8+ HashMap: Uses lazy table expansion + efficient node transfer to reduce overhead.

2) Java Code Example

```
import java.util.*;

public class HashMapResizeExample {
   public static void main(String[] args) {
      HashMap<Integer, String> map = new HashMap<>(4, 0.75f);
      for (int i = 0; i < 10; i++) {
         map.put(i, "Val" + i);
      }
      System.out.println("Final HashMap size: " + map.size());
   }
}
```

Dry Run:

- Initial capacity: 4, load factor: 0.75 → threshold = 3
- Insert keys: 0,1,2 → no resize
- Insert key 3 → resize to 8 → rehash entries 0,1,2,3
- Insert keys 4–9 → further resize to 16 → rehash 0–7
- Total rehashes = 2

3) JVM Internals – Deep Dive

3.1 Object Layout

HashMap Table

```
Node<K,V>[] table
+---------------+
| Node<K,V> head for bin 0
| Node<K,V> head for bin 1
| ...
+---------------+
```

- **During resize:**
  1. **New table allocated** `Node<K,V>[] newTable = new Node[newCapacity]`
  2. **Each old node → calculate new index:** `index = node.hash & (newCapacity - 1)`
  3. **Node inserted into new table**

3.2 Bytecode & Execution

- `putVal()` **triggers** `resize()` **when** `size > threshold`
- `System.arraycopy()` **used for shallow copy of array reference**
- **Node traversal → reassign next pointers → O(n) operation**

Text Diagram (Before/After resize from 4→8):

```
Old table[4]:
bin0: Node(0) -> Node(4)
bin1: Node(1)
bin2: Node(2)
bin3: Node(3)

New table[8]:
bin0: Node(0)
bin1: Node(1)
bin2: Node(2)
bin3: Node(3)
bin4: Node(4)
...
```

4) HotSpot Optimizations

1. **Lazy resizing: Table initialized only on first** `put()`
2. **Inline resize loops: JIT inlines** `putVal()` **and node copy operations**
3. **Escape analysis: Node references temporarily allocated → stack or thread-local**

4. Deoptimization: Rare, triggered if table size changes during JIT-compiled hot path
5. Treeification: High-collision bins reduce traversal cost post-resize

5) GC & Memory Considerations

- Memory spike: Old table + new table coexist temporarily
- GC pressure: Old table becomes eligible for GC after resize
- Node objects: Reused; only table array is new
- Large tables: Can increase heap usage and trigger GC cycles

Heap Diagram (simplified):

```
Old table[4] -> Node0, Node1, Node2, Node3
Resize to table[8] -> Node0, Node1, Node2, Node3, Node4...
Old table[4] -> GC eligible
```

6) Real-World Tie-ins

1. Spring Boot caches: Pre-sizing avoids repeated resizing overhead in `HashMapCache`
2. Fintech/Payments: Transaction maps → avoid resizing under high load → consistent latency
3. ConcurrentHashMap: Resizes are segment-level → multiple threads can assist

7) Pitfalls & Refactors

1. Default sizing: Small default capacity → multiple resizes → performance hit
2. Mutable keys: Rehash relies on hashCode; if key mutates → broken map
3. Refactor: Initialize with estimated capacity → `new HashMap<>(expectedSize)`
4. Composition/Strategy: Combine pre-sized HashMap + treeification for high-collision scenarios

8) Interview Follow-ups (1-Line Answers)

1. Time complexity of resize? → O(n)

2. Impact on memory? → Old table + new table coexist → temporary memory spike
3. What triggers resize? → size > capacity × loadFactor
4. Treeification impact? → Reduces O(n) to O(log n) on high-collision bins
5. ConcurrentHashMap resizing? → Multi-threaded, incremental transfer of bins

Q12: Difference Between Soft, Weak, and Phantom References in Collections (Memory)

1) Overview

- Java References allow fine-grained control over object reachability beyond normal strong references.
- Useful in caching, memory-sensitive data structures, and memory leak prevention.
- Key types:
    1. Strong Reference – normal Java reference, prevents GC
    2. Soft Reference – object reclaimed only if JVM is low on memory
    3. Weak Reference – object reclaimed on next GC cycle, even if memory is sufficient
    4. Phantom Reference – object is already finalized, used for post-mortem cleanup

Collections usage: WeakHashMap, SoftReference caches, ReferenceQueue for cleanup

2) Java Code Example

```
import java.lang.ref.*;
import java.util.*;

public class ReferenceExample {
   public static void main(String[] args) {
      // Soft Reference
      String str = new String("SoftObject");
      SoftReference<String> softRef = new SoftReference<>(str);
      str = null; // Eligible for GC if memory low

      // Weak Reference
      String weakStr = new String("WeakObject");
      WeakReference<String> weakRef = new WeakReference<>(weakStr);
      weakStr = null; // Eligible for GC at next cycle
```

```
        // Phantom Reference
        String phantomStr = new String("PhantomObject");
        ReferenceQueue<String> queue = new ReferenceQueue<>();
        PhantomReference<String> phantomRef = new PhantomReference<>(phantomStr, queue);
        phantomStr = null; // PhantomRef never returns object
    }
}
```

Dry Run:

- **Soft → GC only if heap memory low**
- **Weak → GC triggers immediately next cycle**
- **Phantom → GC collects object; `get()` always returns null; use `queue` to clean resources**

3) JVM Internals – Deep Dive

3.1 Object Layout

```
Object Header:
+----------------+
| Mark Word      | // GC state, hashCode, lock info
| Klass Pointer  | // type info
| Fields...      | // actual object data
+----------------+
```

- **Soft/Weak/Phantom Reference extends `Reference` class:**

```
Reference<T>
+----------------+
| queue          | // ReferenceQueue
| referent       | // actual object (soft/weak) or null (phantom)
| next           | // linked list for GC processing
+----------------+
```

3.2 GC Interaction

- **Soft References: `SoftRefQueue` in HotSpot → processed after memory threshold**
- **Weak References: `WeakRefQueue` → processed during normal GC cycles**
- **Phantom References: `PhantomRefQueue` → post-finalization, used for manual cleanup**

Heap Diagram:

```
Heap:
[Strong Object] -> GC Root
```

[SoftRef] -> Object (low-memory reclaim)
[WeakRef] -> Object (next GC reclaim)
[PhantomRef] -> Object (already finalized) -> ReferenceQueue

3.3 Reference Processing Thread

- JVM has Reference Handler Thread
- Handles enqueuing and clearing of soft/weak/phantom references

4) HotSpot Optimizations

1. SoftReference clearing: HotSpot uses `SoftRefLRU` → oldest soft references cleared first
2. WeakReference clearing: Inline clearing during minor/major GC → avoids extra pause
3. PhantomReference enqueueing: Low overhead, does not resurrect object
4. Escape Analysis: Temporary references can be stack-allocated to reduce GC
5. ReferenceQueue batching: HotSpot optimizes multiple references in a single queue traversal

5) GC & Memory Considerations

- SoftReferences: Memory-sensitive caches, risk of premature reclamation under pressure
- WeakReferences: Good for canonicalizing maps, avoids memory leaks
- PhantomReferences: For native resource cleanup (e.g., file handles, direct buffers)
- Avoid strong references from caches → prevents collection

6) Real-World Tie-ins

1. WeakHashMap → uses weak keys → entries auto-removed when key GC'ed
2. Spring Framework: uses `SoftReference` caches for beans or images
3. Hibernate: L2 cache uses soft/weak references to avoid OutOfMemoryError
4. Payments: Temporary session objects → weak/soft references prevent memory leaks

7) Pitfalls & Refactors

1. SoftReference misuse: May retain too long → high memory usage
2. WeakReference misuse: May disappear too early → inconsistent cache hits
3. PhantomReference misuse: Complex to implement, must use ReferenceQueue
4. Refactor: Combine soft/weak references + ReferenceQueue + composition strategy for safe cache eviction

8) Interview Follow-ups (1-Line Answers)

1. Weak vs Soft Reference? → Weak cleared on next GC, Soft cleared only if memory low
2. Phantom get() return value? → Always null
3. ReferenceQueue usage? → Detect when object is GC'ed for cleanup
4. WeakHashMap internal key storage? → WeakReference keys
5. Impact on performance? → Minor GC overhead, but prevents memory leaks

Q13: How Do IdentityHashMap and EnumMap Differ From Regular HashMap Internally

1) Overview

- HashMap: Stores key-value pairs; uses key.hashCode() + equals() for bucket placement.
- IdentityHashMap: Compares keys by reference (==), not equals().
- EnumMap: Optimized for enum keys, uses array-based storage for high performance.

Impact: These variations optimize memory footprint, performance, and GC behavior for specific use-cases.

2) Java Code Example

```
import java.util.*;
```

```
enum Day { MON, TUE, WED }

public class SpecialMapsExample {
   public static void main(String[] args) {
      // Regular HashMap
      HashMap<String, String> hashMap = new HashMap<>();
      hashMap.put(new String("key1"), "value1");
      hashMap.put(new String("key1"), "value2"); // overwrites due to equals()

      // IdentityHashMap
      IdentityHashMap<String, String> identityMap = new IdentityHashMap<>();
      identityMap.put(new String("key1"), "value1");
      identityMap.put(new String("key1"), "value2"); // treated as different key

      // EnumMap
      EnumMap<Day, String> enumMap = new EnumMap<>(Day.class);
      enumMap.put(Day.MON, "Start");
      enumMap.put(Day.TUE, "Work");

      System.out.println(hashMap);
      System.out.println(identityMap);
      System.out.println(enumMap);
   }
}
```

Dry Run:

- HashMap: key1 **overwritten →** {"key1"="value2"}
- IdentityHashMap: key1 **treated as different references → 2 entries**
- EnumMap: **array-based → O(1) access using enum ordinal**

3) JVM Internals – Deep Dive

3.1 IdentityHashMap

- **Structure: Uses open-addressing array (key/value pairs in flat array)**
- **Key comparison:** k1 == k2
- **Hash calculation: Uses System.identityHashCode(key), ignores** hashCode()
- **Object Layout:**

```
IdentityHashMap Entry Array:
[ key0 | value0 | key1 | value1 | ... ]
```

- No Node objects, less GC pressure than regular HashMap

Bucket Calculation (simplified):

```
index = (System.identityHashCode(key) & 0x7FFFFFFF) % array.length
```

3.2 EnumMap

- Structure: Uses array of size enum.values().length
- Key storage: Key ordinal → index in array
- Hashing: None (direct array index access) → O(1)
- Object Layout:

```
EnumMap:
+----------------+
| Enum[] keys    | // optional for iteration
| Object[] values| // values stored
| int size       |
+----------------+
```

- Benefits: Minimal memory, fast iteration, no treeification

4) HotSpot Optimizations

1. IdentityHashMap: Flat array → cache-friendly; no boxing for nodes
2. EnumMap: Array lookup inlined → extremely low-latency
3. Escape analysis: Keys/values may stay on stack → less GC overhead
4. Inlining: Frequent `get()`/`put()` calls are aggressively inlined by HotSpot
5. No linked lists/tree nodes: Reduces memory fragmentation and pointer chasing

5) GC & Memory Considerations

- HashMap: Node objects per entry → higher heap usage
- IdentityHashMap: Single array → fewer allocations → lower GC overhead
- EnumMap: Small fixed array → negligible GC impact
- Weak references: Not used here, so entries remain strong unless map cleared

Heap Diagram:

HashMap: Node0 -> Node1 -> Node2
IdentityMap: [key0, val0, key1, val1]
EnumMap: values[0]=Start, values[1]=Work

6) Real-World Tie-ins

1. IdentityHashMap: Useful for object identity caches, e.g., Spring bean proxies
2. EnumMap: Optimized for state machines, enums in workflow, e.g., payment status
3. HashMap: General-purpose, flexible, but heavier in memory
4. Hibernate/Spring: EnumMap for efficient entity state mapping, IdentityHashMap for proxy identity checks

7) Pitfalls & Refactors

1. IdentityHashMap: Cannot rely on equals → tricky for string keys; misuse can cause subtle bugs
2. EnumMap: Only works with enum keys; adding non-enum → compile-time error
3. Refactor Strategy: Choose specialized map when performance/memory matters → fallback to HashMap for general keys
4. Composition Pattern: Wrap IdentityHashMap/EnumMap in adapter to provide consistent API

8) Interview Follow-ups (1-Line Answers)

1. IdentityHashMap vs HashMap? → Compares keys by ==, not equals()
2. EnumMap storage? → Array indexed by enum ordinal
3. Why IdentityHashMap faster? → Flat array, no nodes, cache-friendly
4. When to use EnumMap? → Fixed set of enum keys → memory + performance optimized
5. HashMap treeification? → IdentityHashMap/EnumMap do not treeify

Q14: How Do Collections.unmodifiableXXX() and Collections.synchronizedXXX() Wrappers Work Internally

1) Overview

- Collections.unmodifiableXXX(): Provides a read-only view of a collection.
  - Attempts to modify the collection result in UnsupportedOperationException.
  - Wrapper pattern: Delegates all calls to underlying collection for reads, blocks writes.
- Collections.synchronizedXXX(): Provides thread-safe wrapper over collection.
  - Synchronizes all critical methods using a mutex (lock).
  - Supports manual external iteration synchronization to prevent ConcurrentModificationException.

Impact: Allows safe multithreading or immutable views without rewriting collection classes.

2) Java Code Example

```
import java.util.*;

public class CollectionsWrapperExample {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>();
      list.add("A");
      list.add("B");

      // Unmodifiable wrapper
      List<String> unmodList = Collections.unmodifiableList(list);
      System.out.println(unmodList.get(0)); // OK
      // unmodList.add("C"); // Throws UnsupportedOperationException

      // Synchronized wrapper
      List<String> syncList = Collections.synchronizedList(list);
      synchronized(syncList) {
         for (String s : syncList) {
            System.out.println(s);
         }
      }
```

```
    syncList.add("C"); // Thread-safe
  }
}
```

Dry Run:

1. unmodList.get(0) → delegates to list.get(0)
2. unmodList.add("C") → throws exception
3. syncList.add("C") → locks internal mutex (synchronized(lock))
4. Iteration must manually synchronize to avoid fail-fast issues

3) JVM Internals – Deep Dive

3.1 UnmodifiableXXX

- Wrapper class extends AbstractXXX and delegates all read methods.
- Write methods override to throw UnsupportedOperationException.
- Object Layout:

```
UnmodifiableList
+--------------------+
| final List<E> list | // delegate to original collection
+--------------------+
| methods:
| get(), size(), contains() -> delegate
| add(), remove() -> throw UnsupportedOperationException
+--------------------+
```

- Bytecode: Method call delegation → invokevirtual to underlying collection
- / Class Loading: Standard class loading, no extra overhead

3.2 SynchronizedXXX

- Wrapper class contains final Object mutex (often itself if not specified)
- All public methods: synchronized(mutex) { delegate.method() }
- Iteration: Must explicitly lock to avoid fail-fast concurrent modifications

Object Layout:

```
SynchronizedList
+------------------+
| final List<E> list|
```

```
| final Object mutex|
+-------------------+
| methods:
| get(), add() { synchronized(mutex) { list.get()/add() } }
+-------------------+
```

- JVM Execution:
  - Uses monitorenter / monitorexit bytecodes
  - HotSpot JIT optimizes uncontended locks → biased locking / lock coarsening

4) HotSpot Optimizations

1. Lock Elision / Coarsening: Multiple synchronized calls merged by JIT
2. Biased Locking: Fast path when single thread owns lock
3. Escape Analysis: Local collections may avoid wrapper overhead
4. Inlining: Delegation methods in unmodifiable wrappers are aggressively inlined
5. Deoptimization: Rare, only if lock contention detected during JIT

5) GC & Memory Considerations

- Unmodifiable wrapper: Additional object for wrapper → negligible memory
- Synchronized wrapper: Adds mutex reference → negligible
- No node copies: Both reuse underlying collection nodes
- Heap Diagram:

```
Original List -> ArrayList Nodes
UnmodifiableList -> wrapper object -> delegates to Original List
SynchronizedList -> wrapper object -> delegates to Original List + mutex object
```

6) Real-World Tie-ins

1. Spring Framework: Uses unmodifiable wrappers for immutable configuration lists
2. Payments systems: Synchronized wrappers for shared transaction logs
3. Hibernate: Collections wrapped to prevent lazy-loading modifications outside session

7) Pitfalls & Refactors

1. **UnmodifiableXXX: Does not deep-copy → underlying collection changes reflect in wrapper**
2. **SynchronizedXXX: Iteration without `synchronized(lock)` → `ConcurrentModificationException`**
3. **Refactor strategy: Prefer immutable collections (Java 9+) or Concurrent Collections in high concurrency**
4. **Composition: Wrap collection + strategy interface for custom behavior**

8) Interview Follow-ups (1-Line Answers)

1. **Underlying pattern used? → Wrapper / Decorator pattern**
2. **Does unmodifiable copy underlying data? → No, delegates read operations**
3. **Thread-safety of synchronized wrapper? → Yes, all method calls synchronized**
4. **Iteration with synchronized wrapper? → Must manually synchronize**
5. **Impact on performance? → Minimal, except heavy contention in synchronized wrapper**

Level 3 – Tricky & Edge Cases

Q1: What Happens if a Null Key is Inserted in HashMap or TreeMap?

1) Overview

- **HashMap: Allows one null key.**
  - **Null key is treated specially, stored in bucket 0, bypassing `hashCode()` computation.**
- **TreeMap: Depends on comparator.**
  - **Null key not allowed by default if natural ordering is used (`Comparable`) → `NullPointerException`.**
  - **Null key allowed if a custom comparator explicitly handles nulls.**

Impact: Improper null handling can cause runtime crashes or logic errors in collections-based caching or maps.

2) Java Code Example

```
import java.util.*;

public class NullKeyExample {
   public static void main(String[] args) {
      // HashMap allows null key
      HashMap<String, String> hashMap = new HashMap<>();
      hashMap.put(null, "NullValue");
      System.out.println(hashMap.get(null)); // Output: NullValue

      // TreeMap with natural ordering
      TreeMap<String, String> treeMap = new TreeMap<>();
      try {
         treeMap.put(null, "NullValue"); // Throws NullPointerException
      } catch (NullPointerException e) {
         System.out.println("TreeMap does not allow null keys with natural ordering");
      }

      // TreeMap with custom comparator
      TreeMap<String, String> treeMapComp = new TreeMap<>(Comparator.nullsFirst(String::compareTo));
      treeMapComp.put(null, "NullValue"); // Works fine
      System.out.println(treeMapComp.get(null)); // Output: NullValue
   }
}
```

Dry Run:

1. `HashMap.put(null, "NullValue")` → **stored in bucket 0**
2. `TreeMap.put(null, ...)` → **NPE unless comparator handles null**
3. **Custom comparator → null key handled safely**

3) JVM Internals – Deep Dive

3.1 HashMap Null Key Handling

- **Bucket calculation:**

```
int hash = (key == null) ? 0 : key.hashCode() ^ (key.hashCode() >>> 16);
int index = (n - 1) & hash; // n = table.length
```

- **Node creation:**

```
Node<K,V> node = new Node<>(hash, key, value, null);
table[0] = node; // null key always in first bucket
```

- **Bytecode & HotSpot:**
  - ifnull bytecode checks for null → bypass hash computation
  - No extra method calls → JIT can inline

3.2 TreeMap Null Key Handling

- Compare to root node using comparator / natural ordering
- Null in natural order → triggers:

```
key.compareTo(node.key) // null.compareTo(...) → NPE
```

- With comparator: HotSpot inlines comparator call → fast decision
- Object Layout:

```
TreeMap:
+-----------------+
| Entry<K,V> root |
+-----------------+
| Entry.left      |
| Entry.right     |
| Entry.key       |
| Entry.value     |
+-----------------+
```

4) HotSpot Optimizations

1. HashMap null key → inlined hash calculation and == null check
2. TreeMap compare → inline comparator for faster decision
3. Escape analysis → temporary nodes may stay on stack until insertion complete
4. Branch prediction → null key check predicted in JIT for frequent access

5) GC & Memory Considerations

- HashMap: Null key → normal Node object in heap → standard GC applies
- TreeMap: Null key + custom comparator → treated as normal object → no special GC treatment
- Heap Diagram:

HashMap:
[0] -> Node(hash=0, key=null, value="NullValue")
[1..n] -> other buckets

TreeMap with comparator:
Root -> Entry(key=null, value="NullValue")
Left/Right children as per comparator

6) Real-World Tie-ins

1. Caching: HashMap with null key → fast null lookup
2. Payment status mapping: TreeMap with comparator → allows flexible null handling
3. Spring Framework: Bean property maps → null keys avoided in TreeMap unless comparator handles null
4. Hibernate: TreeMap in entities → null-safe comparators for sorting

7) Pitfalls & Refactors

1. HashMap: Only one null key allowed, multiple null insertions overwrite
2. TreeMap: Null keys without comparator → runtime NPE
3. Refactor: Prefer Collections.checkedMap() or comparator to handle nulls
4. Strategy: Use EnumMap or IdentityHashMap if null keys must be avoided for predictable behavior

8) Interview Follow-ups (1-Line Answers)

1. Null key in HashMap? → Allowed, stored in bucket 0
2. Null key in TreeMap with natural order? → Throws NullPointerException
3. How to allow null in TreeMap? → Use Comparator.nullsFirst / nullsLast
4. Multiple null keys in HashMap? → Last inserted overwrites previous
5. Impact on GC? → Normal Node objects → standard GC

Q2: How Do You Avoid Hash Collisions in HashMap?

1) Overview

- Hash collision occurs when two distinct keys produce the same hash code and map to the same bucket.
- Collisions reduce HashMap performance, increasing lookup from O(1) to O(n) per bucket.
- Avoidance strategies:
  1. Good hash function – spread keys uniformly.
  2. Immutable keys – prevent hashCode() changes post-insertion.
  3. Custom objects – override hashCode() and equals() correctly.
  4. Prime-sized table / dynamic resizing – improves distribution.

Impact: Proper hash design ensures fast lookups, memory efficiency, and GC stability.

2) Java Code Example

```
import java.util.*;

class Person {
   String name;
   int id;

   Person(String name, int id) {
      this.name = name;
      this.id = id;
   }

   @Override
   public int hashCode() {
      // Good hash: combine fields
      return Objects.hash(name, id);
   }

   @Override
   public boolean equals(Object obj) {
      if (!(obj instanceof Person)) return false;
      Person other = (Person) obj;
      return this.id == other.id && Objects.equals(this.name, other.name);
   }
}
```

```
public class HashCollisionExample {
   public static void main(String[] args) {
      HashMap<Person, String> map = new HashMap<>();
      map.put(new Person("Alice", 1), "Developer");
      map.put(new Person("Bob", 2), "Manager");
      map.put(new Person("Alice", 1), "Lead"); // Overrides previous

      System.out.println(map.size()); // 2
   }
}
```

Dry Run:

1. `hashCode()` computed → bucket calculated → node inserted.
2. Second insertion of same logical key → `equals()` detects equality → replaces value.
3. Good hash spreads keys → reduces collisions → fewer nodes per bucket → O(1) lookup.

3) JVM Internals – Deep Dive

3.1 HashMap Bucket Structure

- HashMap stores Node<K,V>[] table
- Each Node:

```
Node<K,V> {
  final int hash;
  final K key;
  V value;
  Node<K,V> next; // for chaining
}
```

- Hash collision handling:
  - Chaining (linked list or tree after threshold 8)
  - TreeNode for buckets with many collisions (Red-Black tree)

3.2 Collision Example

- Two keys: `k1.hashCode() == k2.hashCode()`
- Bucket = `(n-1) & hash` → same index → linked list or tree
- Lookup:

```
Node<K,V> e = table[index];
while (e != null) {
   if (e.hash == hash && (e.key == key || key.equals(e.key))) return e.value;
   e = e.next;
}
```

3.3 Object Layout

```
HashMap Table (array of Node):
[0] -> Node(hash1, k1, v1) -> Node(hash1, k2, v2)
[1] -> Node(hash2, k3, v3)
...
```

- **Node objects are ordinary heap objects → subject to GC**
- **TreeNodes include parent, left, right, color fields → slightly more memory**

4) HotSpot Optimizations

1. **Hash spreading:** `(h ^ (h >>> 16))` **→ reduces collision probability**
2. **Treeification threshold (8 nodes): HotSpot switches list → Red-Black tree → O(log n) lookup**
3. **Inlining:** `hashCode()` **and** `equals()` **often inlined for primitive-heavy keys**
4. **Branch prediction: Linked list vs tree detection optimized in JIT**
5. **Deoptimization: Rare, only if list → tree conversion triggers unexpected patterns**

5) GC & Memory Considerations

- **Collisions → longer chains → more Node objects → more pressure on Young Gen**
- **TreeNodes → extra parent/left/right references → slightly higher memory**
- **Frequent rehashing → temporary table arrays → short-lived objects, collected quickly**

Heap Diagram:

```
Bucket 0:
Node(k1,v1) -> Node(k2,v2) -> Node(k3,v3)
Bucket 1:
Node(k4,v4)
```

6) Real-World Tie-ins

1. Caching: Payment transactions keyed by `transactionId` → good hash function prevents hotspots
2. Spring Boot / Hibernate: Entity maps → immutable keys + proper hashCode() → avoids collisions
3. Distributed HashMaps (like Hazelcast): Proper key distribution reduces partition imbalance

7) Pitfalls & Refactors

1. Mutable keys: Changing key fields after insertion → bucket mismatch → invisible entries
2. Poor hashCode() implementation: Many collisions → treeification or long lists → O(n) lookup
3. Refactor Strategy: Use Objects.hash(), Guava Hashing, or primitive keys for better distribution
4. Composition/Strategy: Wrap custom hash strategy in adapter for consistent key handling

8) Interview Follow-ups (1-Line Answers)

1. How to reduce collisions? → Good hashCode() + immutable keys + proper table size
2. What happens if too many collisions? → List → tree → O(log n) lookup
3. Can null key cause collision? → Always goes to bucket 0 → separate handling
4. Impact on GC? → More collisions → more Node objects → more GC load
5. HashMap vs IdentityHashMap? → IdentityHashMap uses `System.identityHashCode()`

Q3: What Happens When Two Objects Have the Same hashCode() But Are Not equals()?

1) Overview

- HashMap/HashSet behavior:
  - `hashCode()` determines bucket placement.
  - `equals()` determines uniqueness within a bucket.

- **Collision scenario:**
  - **Two distinct objects (o1, o2) with o1.hashCode() == o2.hashCode() go to the same bucket.**
  - **The map chains them using linked list or tree (after threshold 8).**
  - **Both objects are stored separately because o1.equals(o2) → false.**

Impact: Collisions don't break correctness, but may reduce lookup performance (O(n) for linked list, O(log n) for tree).

2) Java Code Example

```
import java.util.*;

class Key {
   int id;

   Key(int id) { this.id = id; }

   @Override
   public int hashCode() {
      return 42; // Intentionally bad hash
   }

   @Override
   public boolean equals(Object obj) {
      if (!(obj instanceof Key)) return false;
      return this.id == ((Key) obj).id;
   }
}

public class HashCollisionDemo {
   public static void main(String[] args) {
      HashMap<Key, String> map = new HashMap<>();
      map.put(new Key(1), "One");
      map.put(new Key(2), "Two");
      map.put(new Key(3), "Three");

      System.out.println(map.size()); // Output: 3
      System.out.println(map.get(new Key(2))); // Output: Two
   }
}
```

Dry Run:

1. Key(1) → hash 42 → bucket 42 % table.length → insert Node
2. Key(2) → hash 42 → same bucket → check equals with Key(1) → false → append Node
3. Key(3) → hash 42 → same bucket → append Node
4. Lookup new Key(2) → hash 42 → traverse bucket → equals() matches → return value

3) JVM Internals – Deep Dive

3.1 Bucket & Node Storage

- Node class (HashMap)

```
static class Node<K,V> {
   final int hash;
   final K key;
   V value;
   Node<K,V> next;
}
```

- Collision handling:

```
Bucket index = (n-1) & hash
Bucket[42]:
Node(hash=42, key=Key(1), value="One") -> Node(hash=42, key=Key(2), value="Two") -> Node(hash=42, key=Key(3), value="Three")
```

- Equality check during insert:

```
if (e.hash == hash && (e.key == key || key.equals(e.key)))
   replace value;
else
   continue next node;
```

3.2 Bytecode

- hashCode() call → invokevirtual
- equals() call → invokevirtual per node traversal
- JIT inlines simple hashCode()/equals() → reduces overhead
- HotSpot predicts same-bucket collisions → optimizes branch

4) HotSpot Optimizations

1. Treeification threshold → linked list with >8 nodes converted to Red-Black tree → O(log n) lookup
2. Branch prediction → JIT optimizes frequent hash bucket traversal
3. Inlining → repeated hashCode/equals calls inlined
4. Escape analysis → temporary Node objects allocated efficiently
5. Deoptimization → rare, only if unexpected hashCode patterns cause chain explosion

5) GC & Memory Considerations

- Linked list Nodes → normal heap objects
- TreeNodes → parent, left, right, color fields → slightly more memory
- Collisions increase heap usage per bucket
- Frequent resizing → old array objects become garbage

Heap Diagram:

Bucket 42:
Node(Key(1), "One") -> Node(Key(2), "Two") -> Node(Key(3), "Three")

6) Real-World Tie-ins

1. Caching: High collision rate in key-heavy maps → lookup slowdown
2. Spring Framework / Hibernate: Entity maps → ensure `hashCode()` well-distributed
3. Payments / Transactions: Avoid collision-prone keys (e.g., sequential IDs) in `HashMap` for fast retrieval

7) Pitfalls & Refactors

1. Poor hash function → long bucket chains → O(n) lookup → performance degradation
2. Mutable keys → bucket mismatch → lost entries
3. Refactor strategy:
    - Override `hashCode()` correctly
    - Prefer prime numbers or Guava/MurmurHash for better spread

- Use ConcurrentHashMap if multiple threads

8) Interview Follow-ups (1-Line Answers)

1. Collision impact? → Lookup in bucket → traverses linked list/tree → O(n)/O(log n)
2. Does it break map correctness? → No, equality check preserves uniqueness
3. How to reduce collisions? → Good hash function + immutable keys
4. Treeification in HashMap? → Converts list → Red-Black tree for >8 nodes
5. GC implications? → More Node objects per bucket → more memory pressure

Q4: What Happens if equals() is Not Overridden Correctly for Keys in a HashMap?

1) Overview

- HashMap relies on two methods to manage keys:
  1. hashCode() → determines bucket placement
  2. equals() → checks logical equality within the bucket
- Scenario: Custom key class overrides hashCode() but not equals()
  - Two logically equal objects may reside in different buckets (or appear as distinct even in same bucket)
  - Lookup, contains, or remove operations fail to find the intended object

Impact:

- Entries may duplicate unintentionally
- get() may return null for existing keys
- Can cause subtle bugs in caching, session maps, and entity mapping

2) Java Code Example

```
import java.util.*;

class Person {
```

```
    String name;
    int id;

    Person(String name, int id) {
        this.name = name;
        this.id = id;
    }

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

    // equals() not overridden
}

public class HashMapEqualsFailDemo {
    public static void main(String[] args) {
        HashMap<Person, String> map = new HashMap<>();
        Person p1 = new Person("Alice", 1);
        Person p2 = new Person("Alice", 1);

        map.put(p1, "Developer");

        System.out.println(map.get(p2)); // Output: null (lookup fails)
        System.out.println(map.containsKey(p2)); // false
        System.out.println(map.size()); // 1
    }
}
```

Dry Run:

1. `p1.hashCode()` → bucket calculated → Node inserted
2. `p2.hashCode()` → same hash → bucket matched → equals() used → default `Object.equals()` compares references → false
3. Lookup fails → `null` returned

3) JVM Internals – Deep Dive

3.1 Node & Bucket Mechanics

- `Node<K,V>` structure:

```
Node<K,V> {
    final int hash;
```

```
    final K key;
    V value;
    Node<K,V> next;
}
```

- **Insertion algorithm:**

```
if (e.hash == hash && (e.key == key || key.equals(e.key)))
    replace value;
else
    append Node to bucket;
```

- **Default Object.equals() → compares memory references → distinct objects → insertion treated as new key**

3.2 HotSpot Bytecode

- **invokevirtual equals → dispatch table via itable**
- **Default Object.equals() → pointer equality (==)**
- **JIT may inline equals() → reference comparison optimized**

3.3 Bucket Layout

Bucket i:
Node(hash=p1.hashCode(), key=p1, value="Developer")

- **p2 lookup → same bucket → equals() false → not found**

4) HotSpot Optimizations

1. **Inlining: Small equals() method (like reference check) inlined for performance**
2. **Branch prediction: HotSpot predicts equals() failures vs successes in hot buckets**
3. **Escape analysis: Node allocation may be optimized**
4. **Deoptimization: Rare, only if unexpected hash collisions cause chain growth**
5. **Hash spreading: (h ^ (h >>> 16)) reduces probability of multiple collisions**

5) GC & Memory Considerations

- **Each Node → heap object**
- **Default equals() → no extra memory**

- Poor equals() logic may cause more Node objects in bucket → more memory pressure
- Frequent map resizing → table arrays → short-lived garbage

Heap Diagram:

Bucket i:
Node(key=p1, value="Developer") <- lookup for p2 fails, Node not found

6) Real-World Tie-ins

1. Spring / Hibernate: Entity maps → improperly overridden equals() breaks caching and lazy-loading
2. Payments / Transactions: TransactionID objects → lookup fails → duplicate inserts or missing updates
3. Caching Layer: ConcurrentHashMap → broken equals() leads to duplicate entries, memory leaks

7) Pitfalls & Refactors

1. Mutable keys → hashCode()/equals() change after insertion → lookup fails
2. Partial override: hashCode() overridden but equals() default → incorrect behavior
3. Refactor Strategy:
    - Override both hashCode() and equals() consistently
    - Use immutable key objects to prevent post-insertion changes
    - Consider composition/strategy for hash and equals logic if key is complex

8) Interview Follow-ups (1-Line Answers)

1. Lookup fails if equals() not overridden? → Yes, reference equality used
2. Duplicate entries possible? → Yes, distinct references → separate Node insertion
3. Impact on GC? → More nodes in buckets → memory pressure
4. What to do for mutable keys? → Use immutable fields for hashCode/equals
5. HashMap correctness? → Logical correctness broken → must override equals

Q5: How Does ArrayList Behave if Many Concurrent Modifications Happen?

1) Overview

- ArrayList is not thread-safe.
- Concurrent modifications from multiple threads can cause:
  1. Data inconsistency → lost updates
  2. IndexOutOfBoundsException → during iteration or add/remove
  3. Corrupted internal array → unexpected behavior
- Key reason:
  - ArrayList uses an internal array (Object[] elementData) without synchronization.
  - Structural modification updates size and array elements → race conditions when multiple threads modify simultaneously.

Impact:

- Unexpected behavior in multi-threaded applications
- Requires external synchronization or concurrent collection alternatives

2) Java Code Example

```
import java.util.*;

public class ArrayListConcurrentDemo {
   public static void main(String[] args) throws InterruptedException {
      List<Integer> list = new ArrayList<>();

      Runnable adder = () -> {
         for (int i = 0; i < 1000; i++) {
            list.add(i); // Not thread-safe
         }
      };

      Thread t1 = new Thread(adder);
      Thread t2 = new Thread(adder);

      t1.start();
      t2.start();
```

```
        t1.join();
        t2.join();

        System.out.println("Size: " + list.size()); // Likely < 2000 due to lost updates
    }
}
```

Dry Run:

1. **Both threads call `add()` simultaneously.**
2. **`ArrayList.add()` internally does:**

```
elementData[size] = e;
size++;
```

3. **Race condition:**
    - **Thread 1 reads `size = 5`**
    - **Thread 2 reads `size = 5`**
    - **Both write at index 5 → one element overwritten**
4. **Result: lost updates**

3) JVM Internals – Deep Dive

3.1 Internal Array Structure

ArrayList object header

```
+--------------------+
| mark word          |
| klass pointer      |
| modCount           |
| size               |
| Object[] elementData |
+--------------------+
```

- **Structural modification count (`modCount`) used for fail-fast iterators**

3.2 Bytecode

- **`ArrayList.add()` → `invokevirtual`**
- **`size++` → read/write field (monitored by JVM for concurrent writes)**
- **No synchronization, so race conditions may occur**
- **HotSpot optimizations:**

- Inlined add() method
- Scalar replacement for temporary array index computation
- Possible reordering at JIT level → increases risk of race conditions

3.3 Fail-Fast Iterator

- modCount mismatch → ConcurrentModificationException

4) HotSpot Optimizations

1. Inlining: add(), get(), size() often inlined → fast access
2. Loop unrolling: JIT may optimize iteration
3. Branch prediction: Predicted array bounds
4. Escape analysis: Temporary variables may be stack-allocated
5. Deoptimization: JIT may deopt if unexpected concurrent modifications detected

5) GC & Memory Considerations

- Internal array grows via Arrays.copyOf() → creates new array → old array becomes garbage
- Concurrent modifications → potential overwriting → some elements lost → memory waste
- No per-thread memory management → all Node objects in same heap → GC handles normally

Heap Diagram (Race Example):

elementData array:
Index: 0 1 2 3 4 5 6
Thread1 -> writes 5
Thread2 -> writes 5 (overwrites)

6) Real-World Tie-ins

1. Spring / Hibernate caching: ArrayList must be wrapped in Collections.synchronizedList() or CopyOnWriteArrayList

2. Payment logs: Multiple threads logging transactions → use ConcurrentLinkedQueue or synchronized collection instead
3. Microservices / multithreaded processing: Avoid ArrayList for shared state

7) Pitfalls & Refactors

1. Pitfalls:
    - Lost updates
    - IndexOutOfBoundsException
    - Fail-fast iterator exceptions
2. Refactor Strategy:
    - Use Collections.synchronizedList(new ArrayList<>())
    - Use CopyOnWriteArrayList for frequent reads, rare writes
    - Use Locking / ReentrantLock for fine-grained control
    - Prefer concurrent collections like ConcurrentLinkedQueue if order matters

8) Interview Follow-ups (1-Line Answers)

1. Is ArrayList thread-safe? → No
2. What happens if multiple threads modify? → Data corruption, lost updates
3. How to fix? → Synchronized wrapper or concurrent alternatives
4. Fail-fast iterator behavior? → ConcurrentModificationException
5. GC implications? → Array copies for resizing → temporary objects → garbage collected

Q6: How Do You Safely Remove Elements from a Collection While Iterating?

1) Overview

- Problem: Removing elements from a collection during iteration can cause:
    1. ConcurrentModificationException in fail-fast iterators
    2. Skipped elements in naive loops
    3. Data corruption in multi-threaded scenarios

- Safe strategies:
  1. Using Iterator's remove()
  2. Using removeIf() (Java 8+)
  3. Using CopyOnWrite collections
  4. Using explicit indexing in for-loops for lists (careful with shifting indices)
- Key principle: Avoid modifying the underlying collection directly while traversing it with an iterator that is fail-fast.

2) Java Code Examples

2.1 Using Iterator

```
import java.util.*;

public class IteratorRemoveDemo {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C", "D"));

      Iterator<String> iterator = list.iterator();
      while (iterator.hasNext()) {
         String s = iterator.next();
         if (s.equals("B")) {
            iterator.remove(); // Safe removal
         }
      }

      System.out.println(list); // Output: [A, C, D]
   }
}
```

2.2 Using Java 8 removeIf

```
list.removeIf(s -> s.equals("C"));
System.out.println(list); // Output: [A, D]
```

2.3 Using CopyOnWriteArrayList (Thread-safe, safe iteration)

```
List<String> cowList = new CopyOnWriteArrayList<>(Arrays.asList("X", "Y", "Z"));
for (String s : cowList) {
   if (s.equals("Y")) cowList.remove(s);
}
System.out.println(cowList); // Output: [X, Z]
```

3) JVM Internals – Deep Dive

3.1 Iterator Mechanics

- ArrayList.iterator() → returns Itr inner class
- Itr tracks:
  - cursor → current index
  - lastRet → last returned element
  - expectedModCount → detects structural modifications
- Fail-fast check:

```
final void checkForComodification() {
   if (modCount != expectedModCount)
      throw new ConcurrentModificationException();
}
```

- Bytecode:
  - invokevirtual next() → increments cursor, reads array element
  - invokevirtual remove() → decrements size, shifts elements

3.2 Internal Array Shift

```
public E remove(int index) {
   modCount++;
   E oldValue = elementData[index];
   int numMoved = size - index - 1;
   if (numMoved > 0)
      System.arraycopy(elementData, index + 1, elementData, index, numMoved);
   elementData[--size] = null; // Help GC
   return oldValue;
}
```

- System.arraycopy() → native method, optimized via memcpy

4) HotSpot Optimizations

1. Inlining: Iterator methods (hasNext(), next(), remove()) inlined
2. Branch prediction: HotSpot optimizes checkForComodification() check
3. Scalar replacement: Temporary variables (cursor, lastRet) may be stack-allocated
4. Deoptimization: Rare if array resizing occurs mid-iteration
5. Escape analysis: Temporary iterators may never escape → no heap allocation

5) GC & Memory Considerations

- Removed elements → references nulled (elementData[i] = null) → eligible for GC
- CopyOnWriteArrayList → creates new array on mutation → old array becomes garbage
- Iterator object → short-lived → lightweight heap allocation

Heap Diagram (ArrayList remove):

Before removal: elementData = [A, B, C, D], size=4
Remove B: shift C,D left → elementData = [A, C, D, null], size=3
B object reference nulled → GC eligible

6) Real-World Tie-ins

1. Spring/Hibernate: Safely removing entities from collection in a transaction
2. Payments: Removing expired sessions or temporary entries in logs
3. Multi-threaded processing: CopyOnWriteArrayList / concurrent queues ensure thread-safe removals

7) Pitfalls & Refactors

1. Pitfalls:
    - Removing directly in for-each → ConcurrentModificationException
    - Index-based removal → skipped elements due to shift
    - Using ArrayList in high-concurrency → race conditions
2. Refactors:
    - Prefer Iterator.remove() or removeIf()
    - For concurrent scenarios, use CopyOnWriteArrayList, ConcurrentLinkedQueue
    - Consider composition: isolate mutation logic to a single thread

8) Interview Follow-ups (1-Line Answers)

1. Is for-each safe for removal? → No, throws ConcurrentModificationException
2. Safe way in Java 8+? → removeIf()

3. CopyOnWriteArrayList advantages? → Safe concurrent iteration
4. Memory implications? → Copies of arrays → temporary GC pressure
5. Fail-fast iterator mechanism? → modCount vs expectedModCount

Q7: Difference in Behavior Between LinkedHashMap and HashMap When Rehashing Occurs

1) Overview

- HashMap:
  - Uses array of buckets + linked list / tree for collision resolution.
  - On capacity threshold exceeded (size > load factor × capacity), performs rehashing: creates a new, larger array and redistributes entries based on new hash indexes.
- LinkedHashMap:
  - Extends HashMap, adds doubly linked list to maintain insertion order or access order.
  - During rehashing, both buckets array and linked list order must be maintained.

Key Differences During Rehashing:

Feature	HashMap	LinkedHashMap
Data Structure	Bucket array + Node linked list	Bucket array + Node linked list + DLL order
Rehashing Behavior	Only bucket redistribution	Bucket redistribution + preserves DLL order
Performance impact	O(n) for rehash	Slightly higher O(n) due to linked list updates
Iteration order	Not guaranteed	Preserves insertion or access order

2) Java Code Example

```
import java.util.*;
```

```
public class RehashDemo {
   public static void main(String[] args) {
      HashMap<Integer, String> hashMap = new HashMap<>(2, 0.75f);
      LinkedHashMap<Integer, String> linkedMap = new LinkedHashMap<>(2, 0.75f);

      for (int i = 1; i <= 4; i++) {
         hashMap.put(i, "H" + i);
         linkedMap.put(i, "L" + i);
      }

      System.out.println("HashMap iteration:");
      hashMap.forEach((k,v) -> System.out.print(k + " "));
      System.out.println("\nLinkedHashMap iteration:");
      linkedMap.forEach((k,v) -> System.out.print(k + " "));
   }
}
```

Dry Run:

- HashMap:
  - Capacity = 2, load factor = 0.75 → threshold = 1
  - Adding 2nd element triggers resize → new array size = 4
  - Elements redistributed → iteration order may change
- LinkedHashMap:
  - Same rehashing steps
  - DLL pointers updated → iteration order preserved

Output Example:

```
HashMap iteration: 2 1 4 3  (order not guaranteed)
LinkedHashMap iteration: 1 2 3 4  (insertion order preserved)
```

3) JVM Internals – Deep Dive

3.1 Node Structure

HashMap Node:

```
Node<K,V> {
   int hash;
   K key;
   V value;
   Node<K,V> next; // linked list for bucket
}
```

LinkedHashMap Node:

```
LinkedHashMap.Entry<K,V> extends HashMap.Node<K,V> {
   Entry<K,V> before, after; // doubly linked list
}
```

3.2 Rehashing Algorithm

- Create new array → iterate old table → move each node to new bucket based on (hash & (newCapacity-1))
- LinkedHashMap: additionally updates before and after pointers to preserve insertion/access order

3.3 Bytecode / HotSpot

- put() and rehash() inlined for performance
- modCount incremented → fail-fast iterator uses it
- Bucket redistribution uses System.arraycopy() for treeified bins (JDK 8+)

3.4 Memory Layout

Old table: [Bucket0->Node1->Node2, Bucket1->Node3, ...]
New table: [Bucket0->Node3, Bucket1->Node1->Node2, ...] + DLL preserved

- Linked list pointers updated without breaking insertion order

4) HotSpot Optimizations

1. Inlining: put() / rehash()
2. Loop unrolling: Iterating nodes for rehashing
3. Branch prediction: HotSpot optimizes bucket traversal
4. Scalar replacement: Temporary variables for nodes
5. Escape analysis: Iterator nodes may be stack-allocated if not escaping

5) GC & Memory Considerations

- Rehash → new table array → old array → GC eligible
- Nodes themselves remain → DLL pointers ensure references are intact → no early GC
- If many resizes → temporary memory spike → minor GC pressure

6) Real-World Tie-ins

- Spring / Hibernate caching:
  - LinkedHashMap ensures predictable iteration order for LRU caches
- Payment processing:
  - Preserving insertion order can be critical for transaction replay
- High-concurrency logging:
  - LinkedHashMap + synchronized wrapper → predictable order

7) Pitfalls & Refactors

1. Pitfalls:
   - Large LinkedHashMap → rehashing cost slightly higher than HashMap
   - Frequent access-order reordering → can cause GC churn
2. Refactors:
   - For LRU caching → extend LinkedHashMap with `removeEldestEntry()`
   - For extremely large maps → consider ConcurrentHashMap + separate order-tracking data structure

8) Interview Follow-ups (1-Line Answers)

1. Does HashMap preserve order on rehash? → No
2. Does LinkedHashMap preserve order on rehash? → Yes
3. Performance impact? → LinkedHashMap slightly slower due to DLL pointer updates
4. GC impact? → New table arrays → old arrays GC eligible
5. When to use LinkedHashMap over HashMap? → When iteration order matters

Q8: How Does TreeSet Handle Custom Comparator vs Natural Ordering?

1) Overview

- TreeSet implements NavigableSet using a Red-Black Tree (self-balancing BST).
- Ordering:
  1. Natural Ordering: Uses Comparable.compareTo() of elements
  2. Custom Ordering: Uses a user-provided Comparator
- Key behavior:
  - TreeSet relies entirely on comparison logic to maintain sorted order.
  - No comparator provided → falls back to element's compareTo()
  - Comparator provided → all comparisons delegated to compare()
- Implications:
  - Duplicate detection depends on comparison result (compare() == 0)
  - Incorrect comparator → violates Set contract (duplicate elements allowed, tree corruption possible)

2) Java Code Examples

2.1 Natural Ordering

```
import java.util.*;

public class TreeSetNaturalDemo {
   public static void main(String[] args) {
      TreeSet<Integer> set = new TreeSet<>();
      set.add(5);
      set.add(1);
      set.add(3);

      System.out.println(set); // Output: [1, 3, 5]
   }
}
```

2.2 Custom Comparator

```
TreeSet<String> set = new TreeSet<>(Comparator.reverseOrder());
set.add("Apple");
set.add("Banana");
set.add("Cherry");

System.out.println(set); // Output: [Cherry, Banana, Apple]
```

Dry Run:

- Elements inserted → tree uses compareTo() or compare() to determine position
- Maintains Red-Black Tree properties (balanced, O(log n) operations)

3) JVM Internals – Deep Dive

3.1 Node Structure (TreeMap.Entry used internally)

```
static final class Entry<K,V> implements Map.Entry<K,V> {
   K key;
   V value;
   Entry<K,V> left;
   Entry<K,V> right;
   Entry<K,V> parent;
   boolean color; // Red/Black
}
```

- TreeSet wraps a TreeMap with dummy values (PRESENT)
- comparator stored in TreeMap → used in compare(K k1, K k2)

3.2 Comparison Execution

- JVM executes:

```
final int compare(K k1, K k2) {
   return comparator != null ? comparator.compare(k1,k2) : ((Comparable<? super K>)k1).compareTo(k2);
}
```

- Bytecode:
  - invokeinterface compare() → dynamic dispatch
  - If natural ordering → invokevirtual compareTo()
- HotSpot optimizations:
  - Inline caching → repeated calls to same compare() become fast
  - JIT may inline compareTo() for primitive wrappers (Integer, Long)

3.3 Red-Black Tree Balancing

- Insertion → O(log n) search + rotation
- JVM bytecode:
  - Recursive calls to put()
  - rotateLeft(), rotateRight()
  - Color flips handled in native code for efficiency

4) HotSpot Optimizations

1. Method inlining: compare() and compareTo()
2. Branch prediction: Tree rotations predictable for balanced trees
3. Escape analysis: Temporary node references stack-allocated if method-local
4. Deoptimization: Rare, only on class loading or incorrect assumptions
5. Memory layout: Red-Black tree nodes densely packed → good CPU cache locality

5) GC & Memory Considerations

- Each element → TreeMap.Entry object
- Each entry has references: left, right, parent → reachable → GC safe only when entire set cleared
- Custom comparator → object reference held in TreeMap → prevents early GC

Heap Diagram:

```
TreeSet
 └─ TreeMap
    ├─ comparator (optional)
    └─ root Entry
       ├─ left
       └─ right
```

6) Real-World Tie-ins

1. Spring/Hibernate:
   - TreeSet can be used for sorted entity collections
2. Payments / Fintech:
   - Sorted transaction processing
   - Custom comparator → descending order of priority or timestamp
3. Analytics:
   - Real-time leaderboard using TreeSet with custom comparator

7) Pitfalls & Refactors

1. Pitfalls:
   - Comparator inconsistent with equals → breaks Set contract

- Null elements not allowed → NullPointerException
- Excessive tree rotations → minor performance hits

2. Refactors:
   - For heavy insertions → consider PriorityQueue or SkipListSet
   - For LRU / access-order → consider LinkedHashMap + keySet()
   - Use composition: wrap TreeSet with validation logic for comparator consistency

8) Interview Follow-ups (1-Line Answers)

1. TreeSet default ordering? → Natural ordering via Comparable
2. How to use custom order? → Provide Comparator to constructor
3. Duplicates detection? → compare() == 0
4. Null elements allowed? → No (NullPointerException)
5. Complexity of insertion/search? → O(log n)
6. JVM memory impact? → Each entry → 3 refs + color + object header (~32–48 bytes)

Q9: How Do You Avoid Memory Leaks with WeakHashMap?

1) Overview

- WeakHashMap:
  - Special implementation of Map where keys are held with WeakReference
  - If a key is no longer strongly referenced elsewhere, it becomes eligible for GC
  - Upon GC, the entry is automatically removed from the map
- Key Idea: Prevent memory leaks by ensuring that long-lived maps do not hold references to objects that are no longer needed.
- Use Case: Caching, metadata maps, listener registries, session tracking

2) Java Code Example

```
import java.util.*;

public class WeakHashMapDemo {
```

```
    public static void main(String[] args) throws InterruptedException {
        Map<Object, String> map = new WeakHashMap<>();
        Object key1 = new Object();
        Object key2 = new Object();

        map.put(key1, "Value1");
        map.put(key2, "Value2");

        System.out.println("Before GC: " + map);

        // Remove strong reference to key1
        key1 = null;
        System.gc();
        Thread.sleep(100); // Give GC some time

        System.out.println("After GC: " + map);
    }
}
```

Dry Run:

1. **Map contains** {key1=Value1, key2=Value2}
2. key1 **reference set to** null **→ only weakly referenced in map**
3. **GC collects** key1 **→ WeakHashMap removes entry**
4. **Output:** {key2=Value2}

3) JVM Internals – Deep Dive

3.1 WeakReference in Heap

- **WeakHashMap stores keys as** WeakReference<K>**:**

```
static class Entry<K,V> extends WeakReference<K> implements Map.Entry<K,V> {
    V value;
    int hash;
    Entry<K,V> next;
}
```

- **WeakReference mechanism:**
  - referent **stored inside reference object**
  - **Not strong → GC can clear it if no other strong refs**
- **Reference Queue:**
  - **WeakHashMap maintains** ReferenceQueue

- When GC clears a key → enqueue → map polls queue → removes entry

Text Diagram:

```
Heap:
+------------------------+
| Key2 (strong)          |
| Key1 (weak ref) ------> | WeakReference
+------------------------+
ReferenceQueue:
GC clears key1 -> WeakReference enqueued -> WeakHashMap removes entry
```

3.2 Hashing & Buckets

- Entry stored like HashMap (bucket array + linked list)
- Key hash cached at insertion → no need for strong reference to recompute

3.3 Bytecode

- get(), put(), remove() call expungeStaleEntries() internally
- expungeStaleEntries() polls ReferenceQueue → removes GC'ed keys
- HotSpot inlines these methods, branch predictions optimize loop through queue

4) HotSpot Optimizations

1. Inlining: expungeStaleEntries() and get()/put()
2. Escape analysis: WeakReference objects may escape to heap, but short-lived
3. Deoptimization: Minimal impact unless map grows extremely large
4. Branch prediction: HotSpot optimizes if(ref != null) checks
5. Object header reuse: WeakReference objects have minimal header overhead (~16 bytes)

5) GC & Memory Considerations

- Strong refs: Prevent key collection → potential memory leak
- Weak refs: GC clears when only weak refs remain
- ReferenceQueue: Map polls queue → removes cleared entries → avoids stale entries
- Heap Diagram:

Map Bucket Array:
[Entry(hash, WeakRef(key1), value1) -> Entry(hash, WeakRef(key2), value2)]

After GC key1:
[Entry(hash, WeakRef(null), value1) removed -> Entry(hash, WeakRef(key2), value2)]

- **Proper GC integration ensures automatic memory cleanup**

6) Real-World Tie-ins

1. **Spring/Hibernate:**
   - **Metadata caches using WeakHashMap → avoids holding references to classloader-loaded beans**
2. **Payment systems:**
   - **Temporary transaction metadata → cleared when transaction object no longer in memory**
3. **Listener registries:**
   - **GUI or async listeners stored as keys → automatically cleaned when component destroyed**

7) Pitfalls & Refactors

1. **Pitfalls:**
   - **Strong references outside map → entry never removed → memory leak persists**
   - **Overuse of WeakHashMap → GC pressure, especially in large maps**
   - **Access from multiple threads → not thread-safe → consider Collections.synchronizedMap()**
2. **Refactors:**
   - **Combine with ConcurrentHashMap + weak keys for thread-safe caching**
   - **Periodically call map.size() or iterate to detect stale entries for logging**
   - **Wrap WeakHashMap in utility class to encapsulate GC behavior**

8) Interview Follow-ups (1-Line Answers)

1. **WeakHashMap key removed? → When key is only weakly referenced and GC collects it**
2. **Strong reference outside map? → Entry stays → memory not freed**
3. **Thread-safe WeakHashMap? → Use Collections.synchronizedMap() or ConcurrentWeakHashMap**
4. **Difference from HashMap? → Keys are weakly referenced → automatic GC removal**
5. **Real-world use? → Caches, listeners, metadata, session tracking**

Q10: How Do PriorityQueue Elements Get Sorted Internally?

1) Overview

- **PriorityQueue in Java implements Queue but elements are ordered according to natural ordering (Comparable) or custom Comparator.**
- **Internally implemented as a binary heap (min-heap by default).**
- **Key behaviors:**
  - **Insertion: O(log n) → bubble up to maintain heap invariant**
  - **Peek / Poll: O(1) for peek, O(log n) for poll → bubble down after removal**
- **Does not guarantee iteration order — only guarantees that head is the smallest/largest element based on comparator.**

2) Java Code Example

2.1 Natural Ordering (Min-Heap)

```
import java.util.PriorityQueue;

public class PQDemo {
   public static void main(String[] args) {
      PriorityQueue<Integer> pq = new PriorityQueue<>();
      pq.add(5);
      pq.add(1);
      pq.add(3);
      pq.add(2);

      while(!pq.isEmpty()) {
         System.out.print(pq.poll() + " ");
      }
   }
```

```
}
```

Output:

```
1 2 3 5
```

2.2 Custom Comparator (Max-Heap)

```
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
maxHeap.addAll(List.of(5,1,3,2));

while(!maxHeap.isEmpty()) {
   System.out.print(maxHeap.poll() + " ");
}
```

Output:

```
5 3 2 1
```

Dry Run:

- Elements added → percolate up to maintain heap property
- poll() → remove root → percolate down

3) JVM Internals – Deep Dive

3.1 Heap Implementation

- PriorityQueue uses array-based binary heap:

```
Index-based array:
0: root (min)
1: left child
2: right child
3..n: remaining nodes
```

- Node relationship:
  - Parent: (i-1)/2
  - Left child: 2*i + 1
  - Right child: 2*i + 2
- Private array: transient Object[] queue;

3.2 Comparator vs Natural Ordering

```
private final Comparator<? super E> comparator;
```

```
private int compare(int i, int j) {
   return comparator == null
      ? ((Comparable<? super E>)queue[i]).compareTo((E)queue[j])
      : comparator.compare((E)queue[i], (E)queue[j]);
}
```

- JVM dynamically chooses method at runtime → interface dispatch
- Inline caches optimize repeated comparisons

3.3 Bubble Up / Bubble Down

- offer/add:
  - Insert at last index → percolate up using siftUp()
- poll/remove:
  - Remove root → move last element to root → percolate down using siftDown()

Text Diagram:

```
Initial array: [5, 1, 3, 2]
After percolate up: [1, 2, 3, 5]
```

3.4 Bytecode & HotSpot

- offer() / poll() methods are small and hot → JIT inlines them
- siftUp / siftDown loops optimized for branch prediction
- Repeated array access → CPU cache locality benefits

4) HotSpot Optimizations

1. Inlining: compareTo() / Comparator.compare()
2. Loop unrolling: Heap percolation loops
3. Branch prediction: During bubble up/down
4. Scalar replacement: Temporary x during sift up/down
5. Escape analysis: Local references optimized for stack allocation

5) GC & Memory Considerations

- Internal array grows like ArrayList (1.5× default growth)

- Objects stored as strong references → not GC eligible until removed
- Large PQ → array resizing → old array GC eligible

Heap Diagram:

```
PriorityQueue
 └─ Object[] queue
     ├─ E0 (root)
     ├─ E1
     ├─ E2
     └─ ...
```

6) Real-World Tie-ins

1. Spring/Hibernate:
    - Sorting scheduled tasks or jobs in order of priority
2. Payments / Fintech:
    - Process transactions by priority, e.g., high-value or urgent first
3. Analytics / Queues:
    - Real-time top-K processing using PQ + HashMap

7) Pitfalls & Refactors

1. Pitfalls:
    - Iteration order is not sorted → avoid using iterator for sorted results
    - Custom comparator inconsistent with equals → unpredictable behavior
    - Frequent resizing → GC overhead
2. Refactors:
    - For multi-key priority → wrap element in custom object with comparator
    - For thread-safety → PriorityBlockingQueue
    - For top-K elements → combine PQ with size limit

8) Interview Follow-ups (1-Line Answers)

1. PQ iteration order? → Not guaranteed; only head is minimal/maximal
2. Insertion complexity? → O(log n)

3. Poll complexity? → O(log n)
4. Internal structure? → Binary heap (array-based)
5. Custom comparator effect? → Overrides natural ordering
6. Memory impact? → Array + element objects + object header

Q11: What Happens if a Mutable Object is Used as a Key in HashMap and Its Fields Change?

1) Overview

- HashMap relies on two key contracts:
  1. hashCode(): Determines bucket index
  2. equals(): Determines equality within bucket
- Mutable key problem: If a key's state changes after insertion, its hashCode() may change, breaking bucket lookup.
- Consequences:
  - get(key) may return null even if the key exists
  - containsKey(key) may fail
  - HashMap integrity compromised → "orphaned entries"
- Rule: Keys must be immutable or never mutate fields used in hashCode/equals.

2) Java Code Example

```
import java.util.*;

class MutableKey {
   int id;

   MutableKey(int id) { this.id = id; }

   @Override
   public int hashCode() { return id; }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (!(o instanceof MutableKey)) return false;
      return id == ((MutableKey)o).id;
```

```
    }

    @Override
    public String toString() { return String.valueOf(id); }
}

public class HashMapMutableKeyDemo {
    public static void main(String[] args) {
        Map<MutableKey, String> map = new HashMap<>();
        MutableKey key = new MutableKey(1);
        map.put(key, "Value1");

        System.out.println("Before mutation: " + map.get(key)); // Value1

        key.id = 2; // mutating key

        System.out.println("After mutation: " + map.get(key)); // null
        System.out.println("Map contains key? " + map.containsKey(key)); // false
        System.out.println("Map entries: " + map); // shows original entry in old bucket
    }
}
```

Dry Run:

1. hashCode = 1 → bucket index calculated → entry placed
2. Key mutated: hashCode = 2 → lookup fails → get/containsKey return null
3. Entry exists in old bucket → memory leak / orphaned entry

3) JVM Internals – Deep Dive

3.1 HashMap Storage

HashMap:
[Bucket0] → linked list/tree of Entry<K,V>
[Bucket1] → ...

- Each Entry stores:

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
}
```

- Key insights:

- hash is cached at insertion
- Lookup: (hashCode(key) & (n-1)) → bucket
- Mutating key changes hash → bucket mismatch

3.2 Bytecode / JVM Dispatch

- map.get(key) → calls hashCode() on key → computes bucket
- equals() called only within bucket
- HotSpot inlines hashCode()/equals() → fast if primitive field (int)

Diagram:

```
Before mutation:
Bucket1:
  Entry(hash=1, key=MutableKey(1), value=Value1)

After mutation:
key.hashCode() = 2 → Bucket2 checked → empty
Entry still exists in Bucket1 → lookup fails
```

3.3 HotSpot Optimizations

1. Inlining: hashCode() and equals()
2. Branch prediction: Bucket traversal → predictable
3. Deoptimization: Rare, only affects class loading
4. Escape analysis: Local key references optimized
5. Object header: 16–32 bytes per entry, plus 24 bytes for key object

4) GC & Memory Considerations

- Entry still strongly referenced in old bucket → not eligible for GC
- Memory leak risk if many mutable keys mutated → orphaned entries accumulate
- Proper removal required before mutation or use immutable key objects

Heap Diagram:

```
HashMap
 ├─ Bucket1: Entry(hash=1, key=MutableKey(id=2), value=Value1) <-- orphaned
 └─ Bucket2: empty
```

5) Real-World Tie-ins

1. Spring / Hibernate:
    - Using entities with mutable ID as map keys → dangerous in caching
2. Payment systems / fintech:
    - Transaction ID as mutable key → could lead to lost transaction mapping
3. Analytics / session maps:
    - Using mutable user object → incorrect statistics or memory retention

6) Pitfalls & Refactors

1. Pitfalls:
    - Mutating fields used in hashCode/equals after insertion
    - Using mutable collections as keys (List, Map) → unpredictable behavior
2. Refactors:
    - Use immutable key class (final fields, no setters)
    - Copy key before insertion (`new Key(obj)`)
    - For large mutable datasets → consider `IdentityHashMap` if identity semantics are sufficient

7) Interview Follow-ups (1-Line Answers)

1. What happens if key changes after insertion? → Lookup may fail, entry "orphaned"
2. Safe approach? → Use immutable key objects
3. Why hashCode cached? → For fast bucket indexing
4. Impact on GC? → Orphaned entries still strongly referenced → memory leak
5. Real-world risk? → Caches, session maps, fintech transaction mapping

Q12: How Does PriorityQueue Behave When Comparator is Inconsistent with Equals?

1) Overview

- Comparator inconsistent with equals means:

Comparator.compare(a, b) == 0 // a and b considered equal by comparator
a.equals(b) == false // but a and b are not equal

- PriorityQueue uses binary heap to maintain ordering based on comparator.
- Implications:
  - Heap property maintained by comparator, not equals()
  - Duplicate detection is not based on equals()
  - Iteration may show “duplicates” even if comparator thinks elements are “equal”
  - Methods like remove(Object o) rely on equals(), so behavior can be confusing

2) Java Code Example

```
import java.util.*;

class Person {
   String name;
   int age;
   Person(String name, int age) { this.name = name; this.age = age; }
   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (!(o instanceof Person)) return false;
      return name.equals(((Person)o).name);
   }
   @Override
   public String toString() { return name + ":" + age; }
}

public class PQComparatorDemo {
   public static void main(String[] args) {
      Comparator<Person> comp = Comparator.comparingInt(p -> p.age);
      PriorityQueue<Person> pq = new PriorityQueue<>(comp);

      Person p1 = new Person("Alice", 30);
      Person p2 = new Person("Bob", 30);

      pq.add(p1);
      pq.add(p2);

      System.out.println("Peek: " + pq.peek()); // Alice or Bob (same age)
      while(!pq.isEmpty()) System.out.println(pq.poll());
   }
}
```

Dry Run:

- Both `p1` and `p2` have age 30 → comparator considers them equal
- Heap: may place either first
- `pq.poll()` → returns in heap order, not predictable by equals()

3) JVM Internals – Deep Dive

3.1 Heap and Comparator

- Binary heap array stores objects in order determined by `Comparator.compare()`

```
private int compare(int i, int j) {
   return comparator.compare((E)queue[i], (E)queue[j]);
}
```

- equals() not used internally for heap structure → heap invariant only maintained by comparator
- HotSpot optimizes compare method calls using inline caching

3.2 Bytecode & Heap Updates

- `offer()` → inserts at last index → `siftUp()` uses comparator
- `poll()` → remove root → `siftDown()` uses comparator

Diagram:

```
Heap array:
[0]: p1(age=30)
[1]: p2(age=30)

Comparator.compare(p1,p2) == 0
Heap invariant satisfied
poll() -> may return p1 or p2 (heap order)
```

4) HotSpot Optimizations

1. Inlining: `Comparator.compare()` → reduces virtual call overhead
2. Branch prediction: Sift up/down loops → predictable path
3. Deoptimization: Rare unless comparator is dynamically replaced

4. Escape analysis: Local references in sift up/down optimized for stack allocation
5. Object header: 16–32 bytes per object + array overhead

5) GC & Memory Considerations

- PQ holds strong references to objects → not GC eligible until removed
- Inconsistent comparator does not cause leaks, but may lead to logical duplicates in heap
- Memory layout:

```
PriorityQueue
 └─ Object[] queue
    ├─ p1: Person(Alice,30)
    └─ p2: Person(Bob,30)
```

- Array size may grow → old arrays GC-eligible

6) Real-World Tie-ins

1. Spring / Hibernate:
    - Scheduling jobs by priority → inconsistent comparator may execute jobs in unexpected order
2. Fintech / Payments:
    - Transaction queues → comparator on amount, equals on transaction ID → duplicate processing risk
3. Analytics:
    - Top-K results using PQ → duplicates if comparator inconsistent with logical identity

7) Pitfalls & Refactors

1. Pitfalls:
    - Comparator inconsistent with equals → logical duplicates
    - `remove(Object)` may fail to remove expected object
    - Iteration order unpredictable

2. Refactors:
    - Ensure comparator respects equals if logical uniqueness required
    - Wrap elements with composite key (e.g., age + unique ID)
    - Use TreeSet if both sorting and uniqueness required

8) Interview Follow-ups (1-Line Answers)

1. Behavior of PQ with inconsistent comparator? → Heap order maintained; duplicates may appear
2. Peek vs Poll order? → Determined by comparator only
3. Impact on remove(Object)? → Equals() used → may fail if comparator inconsistent
4. Memory impact? → Strong references retained until removal
5. Real-world advice? → Align comparator with logical equals for predictable behavior

Q13: How Do NavigableMap/NavigableSet Methods (headMap, tailMap, subMap) Work Internally?

1) Overview

- NavigableMap (e.g., TreeMap) and NavigableSet (e.g., TreeSet) are sorted collections.
- Methods like:
    - headMap(toKey) → returns view of all keys < toKey
    - tailMap(fromKey) → returns view of all keys ≥ fromKey
    - subMap(fromKey, toKey) → returns view between fromKey (inclusive) and toKey (exclusive)
- Key property: They return views, not copies → changes in the original map/set reflect in the view and vice versa.

2) Java Code Example

```
import java.util.*;

public class NavigableMapDemo {
   public static void main(String[] args) {
      TreeMap<Integer, String> map = new TreeMap<>();
```

```
        map.put(10, "A");
        map.put(20, "B");
        map.put(30, "C");
        map.put(40, "D");

        NavigableMap<Integer, String> head = map.headMap(30, true); // <= 30
        NavigableMap<Integer, String> tail = map.tailMap(20, true); // >= 20
        NavigableMap<Integer, String> sub = map.subMap(15, true, 35, false); // [15,35)

        System.out.println("HeadMap: " + head);
        System.out.println("TailMap: " + tail);
        System.out.println("SubMap: " + sub);
    }
}
```

Output:

```
HeadMap: {10=A, 20=B, 30=C}
TailMap: {20=B, 30=C, 40=D}
SubMap: {20=B, 30=C}
```

Dry Run:

1. **Original TreeMap is Red-Black tree**
2. **headMap, tailMap, subMap → return NavigableSubMap view objects**
3. **Operations like `get`, `put`, `remove` are delegated to the backing TreeMap with range checks**

3) JVM Internals – Deep Dive

3.1 TreeMap Internal Structure

- **Red-Black Tree (self-balancing BST)**
- **Node structure:**

```
static final class Entry<K,V> implements Map.Entry<K,V> {
    K key;
    V value;
    Entry<K,V> left;
    Entry<K,V> right;
    Entry<K,V> parent;
    boolean color; // RED or BLACK
}
```

- NavigableSubMap stores:
  - fromKey, toKey, fromInclusive, toInclusive
  - Pointer to original TreeMap (m)
- Lookup operations are range-checked:

```
boolean tooLow(K key) { return fromKey != null && compare(key, fromKey) < (fromInclusive?0:1); }
boolean tooHigh(K key) { return toKey != null && compare(key, toKey) > (toInclusive?0:-1); }
```

- Red-Black tree rotations maintain log(n) height

3.2 Bytecode & Method Dispatch

- get(key) → calls rangeCheck(key) → delegate to backing TreeMap getEntry(key)
- put(key, value) → same → inserts node in RB tree → may trigger rotations
- HotSpot inlines compare() and rangeCheck() → efficient subtree traversal

Text Diagram:

```
TreeMap (RB tree)
       20
      /   \
    10     30
          \
           40

subMap(15,35):
  View nodes within [15,35):
    20 -> 30
  Range check prevents 10 and 40 access
```

4) HotSpot Optimizations

1. Inlining: compareTo, rangeCheck, getEntry
2. Branch prediction: Red-Black tree traversal predictable
3. Escape analysis: SubMap view objects may be stack-allocated
4. Loop unrolling: Tree rotations in insert/delete
5. Deoptimization: Rare, only during class redefinition

5) GC & Memory Considerations

- NavigableSubMap is a lightweight view → points to backing TreeMap → no copy
- Nodes remain in RB tree → strongly referenced
- Multiple subMap views → multiple objects, but nodes shared → memory efficient

Heap Diagram:

```
TreeMap RB Tree
 ├─ Entry20
 │   ├─ Entry10
 │   └─ Entry30
 │       └─ Entry40
NavigableSubMap view:
 ├─ fromKey = 15
 ├─ toKey = 35
 └─ backing TreeMap pointer
```

6) Real-World Tie-ins

1. Spring / Hibernate:
    - Cache query results for specific ranges of IDs / timestamps
2. Payments / Fintech:
    - Range queries for transactions between date intervals
3. Analytics:
    - Compute metrics for sliding windows using subMap views

7) Pitfalls & Refactors

1. Pitfalls:
    - Views reflect original map → modifying outside view affects all views
    - subMap with incorrect bounds → runtime exceptions
    - Iteration outside range → ignored
2. Refactors:
    - Use immutable keys for consistency
    - Wrap in helper class for range queries if safety needed
    - For thread-safety → use Collections.synchronizedSortedMap() or ConcurrentSkipListMap

8) Interview Follow-ups (1-Line Answers)

1. Does subMap copy elements? → No, returns view backed by original map
2. Time complexity of get/put in subMap? → O(log n) via RB tree
3. Memory overhead? → Small view objects; nodes shared
4. Thread-safety? → Not safe; use synchronized wrapper or concurrent map
5. Use case in fintech? → Efficient range queries for date/amount intervals

Q14: What Happens if HashMap is Iterated During Rehashing?

1) Overview

- Rehashing: HashMap resizes its internal array (Node<K,V>[] table) when size > capacity * loadFactor.
- During rehashing:
  - Buckets are redistributed based on new capacity
  - Existing nodes' hash determines new bucket index: index = hash & (newCapacity - 1)
- Iteration behavior:
  - Fail-fast iterators track modCount
  - Rehashing increments modCount
  - Iterator detects structural modification → throws ConcurrentModificationException
- Key point: Iteration during rehashing is unsafe in a non-concurrent HashMap

2) Java Code Example

```
import java.util.*;

public class HashMapRehashDemo {
   public static void main(String[] args) {
      Map<Integer, String> map = new HashMap<>(2); // small initial capacity
      map.put(1, "A");
      map.put(2, "B");
      map.put(3, "C"); // triggers resize & rehash

      try {
         for (Integer key : map.keySet()) {
            System.out.println(key);
            map.put(4, "D"); // modifies structure during iteration
         }
      } catch (ConcurrentModificationException e) {
```

```
            System.out.println("Caught CME: " + e);
        }
    }
}
```

Dry Run:

1. **Initial capacity = 2 → loadFactor = 0.75 → threshold = 1.5 (~1)**
2. **Inserting third element → resize to 4 → rehash occurs → modCount++**
3. **Iterator detects expectedModCount != modCount → throws CME**

3) JVM Internals – Deep Dive

3.1 HashMap Structure

```
Node<K,V>[] table
 ├─ bucket0 → Node(key=1, hash=1)
 ├─ bucket1 → Node(key=2, hash=2)
 ├─ bucket2 → null
 └─ bucket3 → Node(key=3, hash=3)
```

- **Rehashing:**
  - **New array Node<K,V>[newCapacity] created**
  - **Each node's hash recomputed: newIndex = hash & (newCapacity-1)**
  - **Nodes moved → next pointers updated**

3.2 Fail-Fast Iterator

```
final void checkForComodification() {
   if (modCount != expectedModCount)
      throw new ConcurrentModificationException();
}
```

- **modCount incremented during put()/remove()**
- **Iterator keeps expectedModCount**
- **HotSpot inlines check → low overhead, frequent in loops**

3.3 Bytecode / Dispatch

- **for-each → Iterator.hasNext() → calls nextEntry() → invokes checkForComodification() → CME thrown if mismatch**
- **Rehashing itself uses System.arraycopy or linked list redistribution → native call optimized**

Text Diagram:

Before rehash:
Capacity=2, modCount=2
Buckets:
0: Node1
1: Node2

After inserting Node3 → rehash:
Capacity=4, modCount=3
Buckets:
0: Node1
1: Node2
2: Node3
Iterator:
expectedModCount=2 → modCount=3 → CME

4) HotSpot Optimizations

1. **Inlining:** `checkForComodification()` **in iterator**
2. **Branch prediction: CME check is predictable for normal iteration**
3. **Array copy optimization:** `System.arraycopy` **for new table during resize**
4. **Escape analysis: Rehashed nodes may remain stack-allocated briefly**
5. **Deoptimization: Rare, only if concurrent modification triggers exception handling**

5) GC & Memory Considerations

- **Rehash creates new array → old array references cleared → eligible for GC**
- **Node objects remain referenced → no object loss**
- **Modifications during iteration → exception handling may temporarily retain iterator stack frames**

Heap Diagram:

Old table[2]:
├─ Node1
└─ Node2
New table[4] (after resize):
├─ Node1
├─ Node2
└─ Node3
Iterator:
@CoVaib-DeepLearn

expectedModCount=2
Actual modCount=3 → CME

6) Real-World Tie-ins

1. Spring / Hibernate:
    - Caching maps updated while iterated → risk of CME
2. Fintech / Payments:
    - Transaction map resizing during reporting → must synchronize or use ConcurrentHashMap
3. Analytics pipelines:
    - Real-time metrics map → unsafe iteration → ConcurrentModificationException

7) Pitfalls & Refactors

1. Pitfalls:
    - Iterating HashMap during structural changes → CME
    - Resizing hidden → may appear as sporadic CME
2. Refactors:
    - Use ConcurrentHashMap for concurrent updates → fail-safe iteration
    - Copy keys before iteration: new ArrayList<>(map.keySet())
    - Avoid mutating map inside for-each loops

8) Interview Follow-ups (1-Line Answers)

1. Why CME occurs during rehash? → Iterator detects modCount mismatch
2. Does HashMap rehash affect existing nodes? → Nodes moved to new buckets based on hash
3. Safe iteration during resize? → No; use fail-safe concurrent maps
4. Memory impact? → Temporary new array allocation; old array GC-eligible
5. Real-world advice? → Always use immutable keys or thread-safe maps in multi-threaded scenarios

Q15: How Do Concurrent Modifications Affect ConcurrentHashMap vs CopyOnWriteArrayList?

1) Overview

- ConcurrentHashMap (CHM):
  - Designed for highly concurrent access
  - Supports lock-free reads and fine-grained locking for writes
  - Concurrent modifications (put/remove) do not block readers
  - Iterators are weakly consistent → reflect some updates but no CME
- CopyOnWriteArrayList (COWAL):
  - Designed for mostly-read, rarely-write scenarios
  - On each mutation (add/remove/set), creates new copy of underlying array
  - Iterators are snapshot-based, immutable → do not see modifications after creation

2) Java Code Example

```
import java.util.*;
import java.util.concurrent.*;

public class ConcurrentDemo {
   public static void main(String[] args) {
      ConcurrentHashMap<Integer, String> chm = new ConcurrentHashMap<>();
      CopyOnWriteArrayList<String> cow = new CopyOnWriteArrayList<>();

      // Populate
      for (int i=0; i<3; i++) {
         chm.put(i, "Val"+i);
         cow.add("Val"+i);
      }

      // Concurrent modification demo
      chm.forEach((k,v) -> chm.put(k+10, v+"New")); // weakly consistent
      for (String s : cow) cow.add(s+"New"); // snapshot iterator, old elements only

      System.out.println("CHM: " + chm);
      System.out.println("COWAL: " + cow);
   }
}
```

Dry Run:

- **CHM iterator may see some of the new keys (weakly consistent)**
- **COWAL iterator sees only the snapshot at the time of iterator creation → safe, but new elements added after iteration are not seen**

3) JVM Internals – Deep Dive

3.1 ConcurrentHashMap Internals

- **Segment-free (Java 8+): Uses Node<K,V>[] table + CAS-based bucket updates**
- **Reads (get, containsKey):**
  - **Lock-free → volatile read ensures visibility**
- **Writes (put, remove):**
  - **CAS or synchronized on bin head when bucket tree conversion occurs**
- **Iterator:**
  - **Weakly consistent → traverses nodes without global locking**
  - **Checks only bin references → tolerates concurrent writes**

Red-Black Tree Conversion:

- **Large buckets converted to TreeNode → still safe under concurrent updates**

Text Diagram:

```
CHM Table[16]
 ├─ bucket0: Node1 -> Node2
 ├─ bucket1: TreeNode(key=...)
 └─ bucket2: null
Iterator traverses bucket array:
 weakly consistent → may see new nodes added during iteration
```

3.2 CopyOnWriteArrayList Internals

- **Underlying array: transient volatile Object[] array**
- **Mutation operation (add/remove/set):**
  - **Creates new array copy**
  - **Updates array reference atomically (volatile write)**
- **Iterator:**
  - **Uses snapshot of array at creation**
  - **No lock required → thread-safe reads**

Text Diagram:

COWAL array snapshot at iterator creation:
[A, B, C]
Mutation: add D → new array [A, B, C, D], iterator still sees [A, B, C]

4) HotSpot Optimizations

1. ConcurrentHashMap:
    - CAS operations optimized with CPU-level compare-and-swap
    - Read operations inlined → avoid memory fences
    - TreeNode rotations in bucket conversion optimized via loop unrolling
2. CopyOnWriteArrayList:
    - Array copy may be optimized via System.arraycopy (native code)
    - Escape analysis → temporary array allocated on stack if short-lived

5) GC & Memory Considerations

- ConcurrentHashMap:
    - Nodes not copied → no additional memory during iteration
    - TreeNode conversion may temporarily allocate tree nodes → old nodes GC-eligible
- CopyOnWriteArrayList:
    - Every mutation → new array allocation → old arrays GC-eligible
    - High write frequency → GC pressure

Memory Diagram:

COWAL:
Snapshot array [A,B,C] -> iterator
Mutation -> new array [A,B,C,D] -> iterator still points to old array
Old array GC-eligible when no references exist

6) Real-World Tie-ins

1. Spring / Hibernate:
    - COWAL for event listeners → safe iteration while listeners added

- CHM for caching or session maps → concurrent updates without blocking reads

2. Fintech / Payments:
   - Transaction logs → CHM ensures concurrent updates
   - COWAL → audit logs, mostly-read, few writes
3. Analytics / Streams:
   - CHM → aggregate metrics in real-time
   - COWAL → read-heavy dashboards

7) Pitfalls & Refactors

1. Pitfalls:
   - CHM: iterator may not see all updates → weak consistency
   - COWAL: expensive for high write volume → triggers frequent GC
2. Refactors:
   - Use CHM for write-heavy, COWAL for read-heavy
   - For hybrid workloads → combine with atomic snapshots or segment-based structures

8) Interview Follow-ups (1-Line Answers)

1. CHM iterator visibility? → Weakly consistent; may see some concurrent updates
2. COWAL iterator visibility? → Snapshot; does not see subsequent updates
3. CHM locking? → Fine-grained; CAS or synchronized per bin
4. COWAL memory overhead? → New array on each mutation → GC pressure
5. Real-world advice? → Use CHM for high write concurrency; COWAL for mostly-read scenarios

Level 4 – Coding Questions

Q1: Implement a HashMap-Based Frequency Counter for Strings

1) Overview

- **Objective: Count how many times each string occurs in a list/array.**
- **Data structure:** HashMap<String, Integer>
- **Time complexity: O(n)**
- **Space complexity: O(k) where k = number of unique strings**

Key MAANG points:

- **Efficient hashing**
- **JVM object layout understanding**
- **Avoid unnecessary boxing/unboxing**

2) Java Code Example (Copy-Pasteable)

```
import java.util.*;

public class StringFrequencyCounter {
   public static void main(String[] args) {
      List<String> words = Arrays.asList("apple", "banana", "apple", "orange", "banana", "apple");

      Map<String, Integer> freqMap = new HashMap<>();

      for (String word : words) {
         freqMap.merge(word, 1, Integer::sum); // Efficient way to count
      }

      // Print results
      freqMap.forEach((k, v) -> System.out.println(k + " -> " + v));
   }
}
```

Output:

```
apple -> 3
banana -> 2
orange -> 1
```

3) Step-by-Step Dry Run

Input: ["apple", "banana", "apple", "orange", "banana", "apple"]

1. freqMap initially empty → {}
2. Insert "apple": key not present → value = 1 → {"apple"=1}
3. Insert "banana": key not present → value = 1 → {"apple"=1, "banana"=1}
4. Insert "apple": key present → merge → value = 2 → {"apple"=2, "banana"=1}
5. Insert "orange": key not present → value = 1 → {"apple"=2, "banana"=1, "orange"=1}
6. Insert "banana": key present → merge → value = 2 → {"apple"=2, "banana"=2, "orange"=1}
7. Insert "apple": key present → merge → value = 3 → {"apple"=3, "banana"=2, "orange"=1}

4) JVM Internals – Deep Dive

4.1 HashMap Object Layout

- Node structure (Java 8+):

```
static class Node<K,V> implements Map.Entry<K,V> {
   final int hash;
   final K key;
   V value;
   Node<K,V> next;
}
```

- Hashing:
  - hash = key.hashCode() ^ (key.hashCode() >>> 16) → spreads bits for uniform distribution
  - Bucket index: index = hash & (table.length - 1)
- JVM Object Header:
- 8-12 bytes Mark Word (lock, hash, age, GC info)
- 4-8 bytes Klass Pointer → points to class metadata in Metaspace
- Node Fields: int hash + K key + V value + Node next
- Array of Nodes: Each bucket points to Node chain

4.2 HotSpot Optimizations

1. Inlining: hashCode(), merge() calls
2. Escape Analysis: Node objects may be stack-allocated temporarily
3. Branch Prediction: Traversal of linked list/bucket highly predictable
4. Deoptimization: Rare, occurs if hashCode() changes mid-iteration

5) GC & Memory Considerations

- Each Node is a heap object, strongly referenced by table array
- Keys and values:
  - String objects → immutable → share references
  - Integer auto-boxed → cached for small values (-128 to 127)
- Rehash triggers allocation of new Node array → old array GC-eligible

Heap Diagram (Simplified):

```
HashMap table[16]
 ├─ bucket0: Node(hash=..., key="apple", value=3)
 ├─ bucket1: Node(hash=..., key="banana", value=2)
 └─ bucket2: Node(hash=..., key="orange", value=1)
```

6) Real-World Tie-ins

1. Spring / Hibernate:
   - Counting entity occurrences or transaction types
2. Payments / Fintech:
   - Count transactions per merchant or per user
3. Analytics:
   - Track top searched keywords or product views

7) Pitfalls & Refactors

1. Pitfalls:
   - Using key.hashCode() incorrectly → poor distribution → bucket collisions
   - Frequent rehashes for large datasets → performance hits
   - Using mutable keys → incorrect counts
2. Refactors / Improvements:
   - Pre-size HashMap with expected size / load factor to reduce resizing
   - For high concurrency → use ConcurrentHashMap with merge()

8) Interview Follow-ups (1-Line Answers)

1. Why use merge() vs put()? → Merge avoids null checks and simplifies increment logic
2. Time complexity? → O(n) average; O(n^2) worst-case if hash collisions
3. Memory overhead? → One Node per unique key + underlying array + object headers
4. Thread safety? → HashMap not thread-safe; use ConcurrentHashMap for multi-threaded access
5. Real-world use case? → Counting API call frequencies or transaction types

Q2: Remove Duplicates from a List While Preserving Order

1) Overview

- Objective: Remove duplicates from a List but keep the insertion order
- Data structure: Use LinkedHashSet or HashSet + LinkedList
- Time complexity: O(n)
- Space complexity: O(n) for the set
- MAANG emphasis: Preserve order efficiently without nested loops

2) Java Code Example (Copy-Pasteable)

```
import java.util.*;

public class RemoveDuplicates {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>(Arrays.asList(
         "apple", "banana", "apple", "orange", "banana", "grape"
      ));

      // Method 1: LinkedHashSet
      List<String> uniqueList = new ArrayList<>(new LinkedHashSet<>(list));
      System.out.println("Unique List (LinkedHashSet): " + uniqueList);

      // Method 2: Manual check using HashSet
      Set<String> seen = new HashSet<>();
      List<String> result = new ArrayList<>();
      for (String s : list) {
         if (seen.add(s)) { // add() returns false if already exists
            result.add(s);
```

```
        }
      }
      System.out.println("Unique List (Manual): " + result);
    }
}
```

Output:

```
Unique List (LinkedHashSet): [apple, banana, orange, grape]
Unique List (Manual): [apple, banana, orange, grape]
```

3) Step-by-Step Dry Run

Input: ["apple", "banana", "apple", "orange", "banana", "grape"]

Using LinkedHashSet:

1. Insert "apple" → added → [apple]
2. Insert "banana" → added → [apple, banana]
3. Insert "apple" → duplicate → ignored
4. Insert "orange" → added → [apple, banana, orange]
5. Insert "banana" → duplicate → ignored
6. Insert "grape" → added → [apple, banana, orange, grape]

Manual HashSet check: Same behavior using seen.add()

4) JVM Internals – Deep Dive

4.1 LinkedHashSet Structure

- LinkedHashSet extends HashSet → backed by HashMap<K, Object>
- Maintains doubly-linked list for insertion order:

```
HashMap table:
 ├─ bucket0 → Node(key="apple", prev=null, next="banana")
 ├─ bucket1 → Node(key="banana", prev="apple", next="orange")
 ├─ bucket2 → Node(key="orange", prev="banana", next="grape")
 └─ bucket3 → Node(key="grape", prev="orange", next=null)
```

- modCount tracks modifications → fail-fast iterators possible

4.2 Node / Entry Layout

- Node header (Mark Word + Klass pointer)
- Fields: int hash, K key, Node<K> next, Node<K> before/after for LinkedHashSet

4.3 HotSpot Optimizations

1. Inlining: HashMap.put() and HashSet.add()
2. Branch prediction: Checking for duplicates predictable → efficient loops
3. Escape analysis: Linked nodes can be optimized for short-lived sets
4. System.arraycopy: Used if backing array needs resizing during HashMap growth

5) GC & Memory Considerations

- Each Node → heap object
- Doubly-linked list nodes → extra references (before and after)
- Temporary arrays → GC-eligible if backing arrays grow
- Overall memory footprint larger than plain HashSet but preserves order

6) Real-World Tie-ins

1. Spring / Hibernate:
    - Deduplicate entities while preserving order for batch inserts
2. Payments / Fintech:
    - Remove duplicate transaction IDs while keeping chronological order
3. Analytics / Logs:
    - Deduplicate events while preserving arrival order

7) Pitfalls & Refactors

1. Pitfalls:
    - Using HashSet alone → order not preserved
    - Using nested loops → $O(n^2)$ time for large datasets

2. Refactors / Improvements:
    - Prefer LinkedHashSet for clean O(n) solution
    - For very large datasets, consider stream-based deduplication:

```
List<String> unique = list.stream().distinct().toList();
```

8) Interview Follow-ups (1-Line Answers)

1. LinkedHashSet vs HashSet? → LinkedHashSet preserves insertion order; HashSet does not
2. Time complexity? → O(n) average
3. Memory overhead? → Extra before/after pointers per node
4. Thread safety? → Neither is thread-safe; use Collections.synchronizedSet() or ConcurrentHashMap.newKeySet() for multi-threading
5. Real-world use case? → Deduplicate user input while maintaining order in UI or API

Q3: Implement a TreeSet to Sort Objects by Multiple Fields

1) Overview

- Objective: Store objects in sorted order based on multiple fields
- Data structure: TreeSet → backed by Red-Black Tree
- Time complexity: O(log n) per insertion/removal
- MAANG emphasis: Use Comparator for custom multi-field sorting

2) Java Code Example (Copy-Pasteable)

```
import java.util.*;

class Person {
   String name;
   int age;
   double salary;

   Person(String name, int age, double salary) {
      this.name = name;
      this.age = age;
```

```
        this.salary = salary;
    }

    @Override
    public String toString() {
        return name + "-" + age + "-" + salary;
    }
}

public class MultiFieldTreeSet {
    public static void main(String[] args) {
        Comparator<Person> multiFieldComparator = Comparator
                .comparing(Person::getName)
                .thenComparing(Person::getAge)
                .thenComparing(Person::getSalary);

        TreeSet<Person> people = new TreeSet<>(multiFieldComparator);

        people.add(new Person("Alice", 30, 70000));
        people.add(new Person("Bob", 25, 50000));
        people.add(new Person("Alice", 30, 65000));
        people.add(new Person("Bob", 25, 55000));
        people.add(new Person("Charlie", 28, 60000));

        for (Person p : people) {
            System.out.println(p);
        }
    }
}

// Adding getter methods
class Person {
    // ... existing fields and constructor
    public String getName() { return name; }
    public int getAge() { return age; }
    public double getSalary() { return salary; }
}
```

Expected Output:

```
Alice-30-65000.0
Alice-30-70000.0
Bob-25-50000.0
Bob-25-55000.0
Charlie-28-60000.0
```

3) Step-by-Step Dry Run

1. Insert Alice-30-70000 → tree empty → becomes root
2. Insert Bob-25-50000 → compare name → "Bob" > "Alice" → goes right
3. Insert Alice-30-65000 → compare name → "Alice" = "Alice", compare age → 30 = 30, compare salary → 65000 < 70000 → goes left of Alice-30-70000
4. Insert Bob-25-55000 → compare name → "Bob" = "Bob", age → 25=25, salary → 55000>50000 → goes right of Bob-25-50000
5. Insert Charlie-28-60000 → "Charlie" > root → traverse right → insert

4) JVM Internals – Deep Dive

4.1 TreeSet / TreeMap Structure

- TreeSet is backed by TreeMap
- TreeMap Node:

```
static final class Entry<K,V> {
   K key;
   V value;
   Entry<K,V> left, right, parent;
   boolean color; // RED or BLACK
}
```

- Red-Black Tree properties:
  - Self-balancing → O(log n) insert/search
  - Each node has color bit and links: left, right, parent

4.2 Comparator Handling

- TreeMap compares using Comparator (or Comparable)
- JVM calls Comparator.compare() → inlined by HotSpot JIT

4.3 Object Header Layout

- Each Entry Node:

```
Mark Word (8-12B) + Klass pointer (4-8B)
K key + V value + left + right + parent + color bit
```

- Key objects (Person) → each has own header + fields

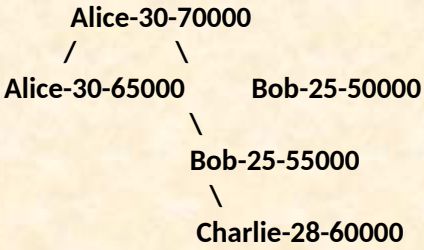
5) HotSpot Optimizations

1. Comparator inlining → avoids method call overhead
2. Branch prediction → tree traversal predictable
3. Escape analysis → temporary tree rotations optimized
4. Deoptimization → rare, if tree shape changes drastically during iteration

6) GC & Memory Considerations

- Each `Entry` Node → heap object
- `Person` objects → strongly referenced by tree
- Rotations → temporary stack variables; not allocated on heap
- Large trees → more memory for `parent`, `left`, `right` pointers

Heap Diagram (simplified):



7) Real-World Tie-ins

1. Spring / Hibernate:
    - Sorting entities by multiple fields before batch processing
2. Payments / Fintech:
    - Sort transactions by date → merchant → amount
3. Analytics / Reports:
    - Multi-field ranking for leaderboards or performance metrics

8) Pitfalls & Refactors

1. Pitfalls:

- Mutable keys → corrupt tree ordering
- Comparator inconsistent with equals() → unpredictable behavior
- Large trees → many rotations → CPU-intensive

2. Refactors / Improvements:
   - Make key objects immutable
   - Precompute hash / comparison keys if heavily used
   - Consider PriorityQueue for top-K elements (if full ordering not needed)

9) Interview Follow-ups (1-Line Answers)

1. TreeSet vs HashSet? → TreeSet maintains sorted order, HashSet does not
2. Time complexity? → O(log n) per insert/search
3. Comparator vs Comparable? → Comparator allows multi-field/custom ordering
4. Thread safety? → Not thread-safe; wrap with Collections.synchronizedSortedSet() if needed
5. Real-world use case? → Sorted leaderboards, event ranking, multi-key transaction sorting

Q4: Implement a PriorityQueue to Process Tasks Based on Priority

1) Overview

- Objective: Process tasks in order of priority rather than insertion order
- Data structure: PriorityQueue → backed by binary heap
- Time complexity:
  - Insertion: O(log n)
  - Peek: O(1)
  - Poll (remove min/max): O(log n)
- MAANG emphasis: Efficient task scheduling, heap property, JVM optimizations

2) Java Code Example (Copy-Pasteable)

```
import java.util.*;
```

```
class Task {
   String name;
   int priority; // smaller value = higher priority

   Task(String name, int priority) {
      this.name = name;
      this.priority = priority;
   }

   @Override
   public String toString() {
      return name + "-" + priority;
   }
}

public class TaskScheduler {
   public static void main(String[] args) {
      PriorityQueue<Task> pq = new PriorityQueue<>(Comparator.comparingInt(t -> t.priority));

      pq.add(new Task("Task1", 5));
      pq.add(new Task("Task2", 1));
      pq.add(new Task("Task3", 3));
      pq.add(new Task("Task4", 2));

      while (!pq.isEmpty()) {
         System.out.println(pq.poll());
      }
   }
}
```

Expected Output:

```
Task2-1
Task4-2
Task3-3
Task1-5
```

3) Step-by-Step Dry Run (Heap Behavior)

Input tasks: Task1(5), Task2(1), Task3(3), Task4(2)

Binary heap insertion order:

1. Insert Task1(5) → root
2. Insert Task2(1) → heapify → becomes new root (min-heap)
3. Insert Task3(3) → heapify → fits as right child

4. Insert Task4(2) → heapify → swaps with Task3 → maintains heap

Polling:

- poll() removes root → heapify maintains min-heap
- Output order: Task2, Task4, Task3, Task1

4) JVM Internals – Deep Dive

4.1 PriorityQueue Structure

- PriorityQueue extends AbstractQueue
- Backed by array (Object[] queue) → binary heap
- Fields:
- Object[] queue;
- int size;
- Comparator<? super E> comparator;

4.2 Heap Operations

- offer(E e) → inserts element → siftUp to maintain heap
- poll() → removes root → siftDown to restore heap
- Heap property: queue[parent] <= queue[child] (min-heap)

4.3 Node/Object Layout

- Array elements → references to Task objects
- Task object header:
- Mark Word (8-12B)
- Klass Pointer (4-8B)
- Fields: String name, int priority
- Heap array → contiguous references, good CPU cache locality

5) HotSpot Optimizations

1. Inlining: siftUp() and siftDown() methods for performance
2. Branch Prediction: Comparisons in heap traversal are predictable
3. Escape Analysis: Task objects allocated on heap; small ones may be stack-allocated if temporary

4. **Deoptimization: Rare, only if class hierarchy changes during runtime**

6) GC & Memory Considerations

- `Task` **objects → strong references in heap array**
- **Array may resize → old array becomes GC-eligible**
- **Min-heap array references → contiguous → better CPU cache utilization**

Heap Diagram (Array-based):

Index: 0 1 2 3
Task2(1) Task4(2) Task3(3) Task1(5)

- **Root at index 0, children at *2i+1 and 2*i+2**

7) Real-World Tie-ins

1. **Spring / Executors:**
   - `ScheduledThreadPoolExecutor` **internally uses** `DelayedWorkQueue` **(PriorityQueue)**
2. **Payments / Fintech:**
   - **Priority-based transaction processing**
3. **Analytics / Real-time Systems:**
   - **Event processing with urgency levels**

8) Pitfalls & Refactors

1. **Pitfalls:**
   - **Using mutable keys (priority) → violates heap property**
   - **Large queue → frequent resizing → performance impact**
   - **Comparator inconsistent with equals → undefined behavior**
2. **Refactors / Improvements:**
   - **Use immutable** `Task` **objects**
   - **Pre-size PriorityQueue for high-throughput workloads**
   - **Use** `ConcurrentSkipListSet` **or** `DelayQueue` **for thread-safe priority queues**

9) Interview Follow-ups (1-Line Answers)

1. Time complexity? → O(log n) for insertion/poll; O(1) for peek
2. Space complexity? → O(n) for underlying array + object references
3. Min-heap vs Max-heap? → Comparator decides ordering
4. Thread safety? → Not thread-safe; use PriorityBlockingQueue
5. Real-world use case? → Job scheduling, event processing, rate-limited API calls

Q5: Iterate and Remove Elements Safely from a Collection

1) Overview

- Objective: Remove elements from a collection while iterating without causing ConcurrentModificationException
- Common approaches:
    1. Use Iterator's remove() method → safe, fail-fast
    2. Use removeIf() method (Java 8+)
    3. Use CopyOnWriteArrayList for concurrent access
- MAANG emphasis: Efficient, thread-safe, and memory-conscious solution

2) Java Code Example (Copy-Pasteable)

```
import java.util.*;

public class SafeRemove {
   public static void main(String[] args) {
      List<String> fruits = new ArrayList<>(Arrays.asList(
         "apple", "banana", "orange", "banana", "grape"
      ));

      // Method 1: Using Iterator.remove()
      Iterator<String> it = fruits.iterator();
      while (it.hasNext()) {
         String fruit = it.next();
         if ("banana".equals(fruit)) {
            it.remove(); // safe removal
```

```
        }
    }
    System.out.println("After Iterator.remove(): " + fruits);

    // Method 2: Using removeIf() (Java 8+)
    List<String> fruits2 = new ArrayList<>(Arrays.asList(
        "apple", "banana", "orange", "banana", "grape"
    ));
    fruits2.removeIf(fruit -> "banana".equals(fruit));
    System.out.println("After removeIf(): " + fruits2);

    // Method 3: Using CopyOnWriteArrayList (concurrent)
    List<String> fruits3 = new CopyOnWriteArrayList<>(Arrays.asList(
        "apple", "banana", "orange", "banana", "grape"
    ));
    for (String fruit : fruits3) {
        if ("banana".equals(fruit)) {
            fruits3.remove(fruit); // safe in CopyOnWriteArrayList
        }
    }
    System.out.println("After CopyOnWriteArrayList removal: " + fruits3);
  }
}
```

Output:

```
After Iterator.remove(): [apple, orange, grape]
After removeIf(): [apple, orange, grape]
After CopyOnWriteArrayList removal: [apple, orange, grape]
```

3) Step-by-Step Dry Run

Input: ["apple", "banana", "orange", "banana", "grape"]

- **Iterator.remove(): Traverses each element → removes "banana" safely → final list [apple, orange, grape]**
- **removeIf(): Internally uses Spliterator → removes "banana" → final list [apple, orange, grape]**
- **CopyOnWriteArrayList: Iteration is over snapshot, concurrent modifications do not throw exceptions**

4) JVM Internals – Deep Dive

4.1 Iterator Implementation

- **ArrayList Iterator:**

```
private class Itr implements Iterator<E> {
   int cursor; // index of next element
   int lastRet = -1; // index of last element returned
   int expectedModCount = modCount; // fail-fast check

   public boolean hasNext() { return cursor != size; }

   public E next() {
      checkForComodification();
      int i = cursor;
      if (i >= size)
         throw new NoSuchElementException();
      Object[] elementData = ArrayList.this.elementData;
      cursor = i + 1;
      return (E) elementData[lastRet = i];
   }

   public void remove() {
      if (lastRet < 0)
         throw new IllegalStateException();
      checkForComodification();
      ArrayList.this.remove(lastRet);
      cursor = lastRet;
      lastRet = -1;
      expectedModCount = modCount;
   }
}
```

- `modCount` → **detects structural changes**
- `expectedModCount` → **ensures fail-fast**

4.2 CopyOnWriteArrayList

- **Uses snapshot array for iteration**
- `add/remove` → **create new array internally → avoids** `ConcurrentModificationException`
- **Memory overhead → O(n) copy on each mutation**

5) HotSpot Optimizations

1. **Iterator inlining → eliminates method call overhead for** `hasNext()` **and** `next()`

2. Loop unrolling → for removeIf() using lambda and Spliterator
3. Escape analysis → temporary iterators may be stack-allocated
4. Deoptimization: Happens if concurrent modification detected in fail-fast iterator

6) GC & Memory Considerations

- Iterator objects → short-lived → stack or heap (escape analysis)
- CopyOnWriteArrayList → creates new arrays → old arrays GC-eligible
- removeIf() → uses internal arrays → temporary objects collected by GC

7) Real-World Tie-ins

1. Spring / Hibernate:
    - Safe removal of entities from collections in batch operations
2. Payments / Fintech:
    - Removing invalid transactions while iterating over ledger
3. Analytics / Logs:
    - Filter out events or duplicates safely during iteration

8) Pitfalls & Refactors

1. Pitfalls:
    - Using for-each with ArrayList.remove() → triggers ConcurrentModificationException
    - CopyOnWriteArrayList → heavy memory usage for frequent modifications
2. Refactors / Improvements:
    - Prefer Iterator.remove() for single-threaded, mutable lists
    - Prefer removeIf() for clean, functional code
    - Prefer CopyOnWriteArrayList for multi-threaded, mostly-read scenarios

9) Interview Follow-ups (1-Line Answers)

1. Fail-fast vs fail-safe? → Iterator.remove() is fail-fast; CopyOnWriteArrayList iterator is fail-safe
2. Time complexity? → O(n) for iteration and removal
3. Space complexity? → O(1) for ArrayList iterator; O(n) for CopyOnWriteArrayList
4. Thread safety? → Use CopyOnWriteArrayList or synchronized collections
5. Real-world use case? → Filtering logs, transactions, or UI elements safely during iteration

Q6: Implement LinkedHashMap LRU Cache Manually

1) Overview

- Objective: Implement Least Recently Used (LRU) cache
- Key data structures: LinkedHashMap (maintains access order)
- Time complexity: O(1) for get and put
- MAANG emphasis: Use built-in JDK classes efficiently; understand JVM memory & iteration optimizations

2) Java Code Example (Copy-Pasteable)

```
import java.util.LinkedHashMap;
import java.util.Map;

class LRUCache<K, V> extends LinkedHashMap<K, V> {
   private final int capacity;

   public LRUCache(int capacity) {
      super(capacity, 0.75f, true); // accessOrder = true
      this.capacity = capacity;
   }

   @Override
   protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
      return size() > capacity; // remove least recently used
   }

   public static void main(String[] args) {
      LRUCache<Integer, String> cache = new LRUCache<>(3);
```

```
        cache.put(1, "A"); // cache: 1:A
        cache.put(2, "B"); // cache: 1:A, 2:B
        cache.put(3, "C"); // cache: 1:A, 2:B, 3:C
        cache.get(1);      // access 1 → cache: 2:B, 3:C, 1:A
        cache.put(4, "D"); // evicts 2:B → cache: 3:C, 1:A, 4:D

        cache.forEach((k,v) -> System.out.println(k + ":" + v));
    }
}
```

Expected Output:

```
3:C
1:A
4:D
```

3) Step-by-Step Dry Run

1. Put 1:A → cache size 1 → no eviction
2. Put 2:B → cache size 2 → no eviction
3. Put 3:C → cache size 3 → no eviction
4. Get 1 → marks key 1 as most recently used → reorders linked list
5. Put 4:D → size > capacity → removes eldest (key 2) → cache: 3:C, 1:A, 4:D

4) JVM Internals – Deep Dive

4.1 LinkedHashMap Structure

- Extends HashMap → underlying buckets + linked list
- Fields:

```
transient Entry<K,V> head, tail;
boolean accessOrder; // true = LRU
```

- Node structure:

```
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after; // doubly linked list pointers
}
```

4.2 Access Order

- On `get` or `put`, `afterNodeAccess()` is called:
  - Detaches node from linked list
  - Moves node to tail → marks as most recently used

4.3 Object Header Layout

- Each `Entry` → object header + key/value references + before/after references
- JVM memory impact → each node ≈ 32–48 bytes (object header + pointers)

5) HotSpot Optimizations

1. Inlining: `get()` / `put()` / `afterNodeAccess()`
2. Branch prediction: Bucket selection via `hash & (n-1)`
3. Escape analysis: Temporary node references optimized by JIT
4. Deoptimization: Happens if map grows and triggers rehash

6) GC & Memory Considerations

- Each `Entry` → heap object, strongly referenced by bucket and linked list
- On eviction → references cleared → GC eligible
- Large cache → consider soft/weak references to avoid OOM

Memory Layout Diagram:

```
Head -> 3:C <-> 1:A <-> 4:D <- Tail
Buckets -> hash(key) mod N -> Entry nodes
```

7) Real-World Tie-ins

1. Spring:
   - Caching DAO results or REST responses in LRU fashion
2. Payments / Fintech:
   - Cache recent transaction lookups or fraud checks
3. Web servers:

- Session caching with eviction of least recently used sessions

8) Pitfalls & Refactors

1. Pitfalls:
   - Mutable keys → may cause incorrect bucket placement
   - Not setting `accessOrder = true` → behaves like FIFO instead of LRU
   - Large caches → GC pressure
2. Refactors / Improvements:
   - Wrap in soft/weak references for memory-sensitive caches
   - Use `ConcurrentLinkedHashMap` or `Caffeine` for thread-safe high-performance cache
   - Separate eviction logic with strategy pattern for custom policies

9) Interview Follow-ups (1-Line Answers)

1. Time complexity? → O(1) for `get()` and `put()`
2. Thread safety? → Not thread-safe; use `Collections.synchronizedMap()` or `ConcurrentHashMap + LinkedList`
3. Space complexity? → O(n) nodes + object references
4. Alternative implementations? → `Deque + HashMap` for fully manual LRU
5. Real-world use case? → Caching API responses, database query results, session storage

Q7: Convert an ArrayList to a HashSet and Explain Behavior

1) Overview

- Objective: Remove duplicates from a list and/or get unique elements
- Core idea: `HashSet` uses hash-based storage (O(1) average for add/contains)
- MAANG emphasis: Understand hashing, equals contract, bucket placement, collisions, and memory implications

2) Java Code Example (Copy-Pasteable)

```
import java.util.*;

public class ArrayListToHashSet {
   public static void main(String[] args) {
      List<String> fruits = new ArrayList<>(Arrays.asList(
         "apple", "banana", "orange", "banana", "apple", "grape"
      ));

      // Convert to HashSet
      Set<String> fruitSet = new HashSet<>(fruits);
      System.out.println("Unique elements: " + fruitSet);

      // Optional: back to list to preserve insertion order for iteration
      List<String> uniqueFruits = new ArrayList<>(fruitSet);
      System.out.println("List from HashSet: " + uniqueFruits);
   }
}
```

Sample Output (order not guaranteed):

```
Unique elements: [banana, orange, apple, grape]
List from HashSet: [banana, orange, apple, grape]
```

3) Step-by-Step Dry Run

Input list: ["apple", "banana", "orange", "banana", "apple", "grape"]

1. Create empty HashSet → size=0
2. Insert "apple" → hashCode → bucket → empty → add
3. Insert "banana" → hashCode → bucket → empty → add
4. Insert "orange" → hashCode → bucket → empty → add
5. Insert "banana" → hashCode → bucket → collision → equals() → duplicate → ignored
6. Insert "apple" → hashCode → bucket → collision → equals() → duplicate → ignored
7. Insert "grape" → hashCode → bucket → empty → add

Result: Only unique elements remain

4) JVM Internals – Deep Dive

4.1 HashSet Structure

- HashSet **internally uses HashMap:**

```
private transient HashMap<E,Object> map;
private static final Object PRESENT = new Object();
```

- add(E e) → map.put(e, PRESENT) → **keys are unique, values are dummy objects**

4.2 HashMap Node Layout

- **Array of buckets → each bucket contains Node<K,V>**
- **Node structure:**

```
static class Node<K,V> {
   final int hash;
   final K key;
   V value;
   Node<K,V> next;
}
```

- **Node header → object header + hash + key + value + next pointer**

4.3 Hashing & Bucket Placement

1. **Compute** hashCode() **→ spread using bitwise operations**
2. **Determine bucket index:** index = (n - 1) & hash
3. **Check linked list / tree for duplicates via** equals()

Text-Based Diagram:

```
Buckets:
[0] -> null
[1] -> Node(hash: 123, key:"apple", value:PRESENT)
[2] -> Node(hash: 456, key:"banana", value:PRESENT)
[3] -> Node(hash: 789, key:"orange", value:PRESENT)
...
```

5) HotSpot Optimizations

1. **Inlining:** HashMap.put() **and** hash() **→ removes call overhead**
2. **Branch prediction: duplicate check (**equals()**) predictable for repeated values**
3. **Deoptimization: occurs if** hashCode() **or** equals() **behaves inconsistently**

4. **Treeification: if bucket exceeds threshold (default 8) → converts linked list to TreeNode (Red-Black Tree) for O(log n) lookups**

6) GC & Memory Considerations

- **Each unique element → key object + Node object**
- **Dummy value `PRESENT` is shared → negligible overhead**
- **Large HashSet → consider initial capacity and load factor to avoid frequent resizing**

7) Real-World Tie-ins

1. **Spring / Hibernate:**
   - **Unique entity caching**
   - **Ensuring no duplicates in collection mappings**
2. **Payments / Fintech:**
   - **Deduplicate transaction IDs before processing**
3. **Analytics:**
   - **Compute unique users, sessions, or events efficiently**

8) Pitfalls & Refactors

1. **Pitfalls:**
   - **Mutable keys → may break bucket mapping**
   - **Large list → HashSet resizing can trigger GC and performance overhead**
   - **Order not preserved → consider `LinkedHashSet` if order matters**
2. **Refactors / Improvements:**
   - **Use `LinkedHashSet` → preserves insertion order**
   - **Pre-size `HashSet` → `new HashSet<>(list.size())` → avoids resizing**
   - **Use `stream().distinct()` → functional approach for clarity**

9) Interview Follow-ups (1-Line Answers)

1. Time complexity? → O(n) to add all elements, O(1) per insertion amortized
2. Space complexity? → O(n) for unique elements + Node objects
3. Preserves order? → No (use LinkedHashSet)
4. Duplicates handled? → HashSet ignores duplicates based on equals()
5. Real-world use case? → Removing duplicates from logs, transaction IDs, or user sessions

Q8: Implement a Stack Using Deque

1) Overview

- Objective: Implement a stack (LIFO) using Java's Deque interface instead of Stack class
- Why Deque over Stack:
  - Stack is legacy (Vector-based, synchronized → performance overhead)
  - Deque offers O(1) push/pop at both ends and is non-synchronized (better for single-threaded or explicit synchronization)
- MAANG emphasis: Efficient, thread-safe design, deep JVM understanding

2) Java Code Example (Copy-Pasteable)

```
import java.util.ArrayDeque;
import java.util.Deque;

class StackUsingDeque<T> {
  private Deque<T> deque;

  public StackUsingDeque() {
    this.deque = new ArrayDeque<>();
  }

  public void push(T item) {
    deque.addFirst(item); // push to front
  }

  public T pop() {
    if (deque.isEmpty()) throw new IllegalStateException("Stack is empty");
    return deque.removeFirst();
  }

  public T peek() {
    if (deque.isEmpty()) throw new IllegalStateException("Stack is empty");
```

```
        return deque.peekFirst();
    }

    public boolean isEmpty() {
        return deque.isEmpty();
    }

    public int size() {
        return deque.size();
    }

    public static void main(String[] args) {
        StackUsingDeque<Integer> stack = new StackUsingDeque<>();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println("Top element: " + stack.peek()); // 30
        System.out.println("Pop: " + stack.pop()); // 30
        System.out.println("Pop: " + stack.pop()); // 20
        System.out.println("Stack size: " + stack.size()); // 1
    }
}
```

Expected Output:

```
Top element: 30
Pop: 30
Pop: 20
Stack size: 1
```

3) Step-by-Step Dry Run

1. Push 10 → deque: [10]
2. Push 20 → deque: [20, 10]
3. Push 30 → deque: [30, 20, 10]
4. Peek → 30 (front element)
5. Pop → remove 30 → deque: [20, 10]
6. Pop → remove 20 → deque: [10]

4) JVM Internals – Deep Dive

4.1 ArrayDeque Structure

- **Array-based circular buffer (resizable)**

```
transient Object[] elements;
transient int head; // points to first element
transient int tail; // points to next available slot at end
```

- **Push (addFirst) → head decremented modulo array length**
- **Pop (removeFirst) → head incremented modulo array length**
- **Resizing: array doubles in size → O(n) copy**

4.2 Object Layout

- **Each element stored as reference in Object[]**
- **Object header → mark word + klass pointer**
- **No node objects → better cache locality than LinkedList**

Text-Based Diagram:

```
elements[]: [10, 20, 30, null, ...]
head -> 30
tail -> next empty slot
```

5) HotSpot Optimizations

1. **Inlining: addFirst(), removeFirst(), peekFirst()**
2. **Branch prediction: array bounds checks and empty checks are predictable**
3. **Escape analysis: elements array and object references optimized**
4. **Deoptimization: occurs if resizing triggers GC pressure**

6) GC & Memory Considerations

- **ArrayDeque stores references → small heap footprint**
- **Resizing → old array becomes GC-eligible**
- **No per-element Node objects → more cache-friendly than LinkedList**

7) Real-World Tie-ins

1. Spring / Hibernate:
   - Undo/Redo stacks in UI or transaction context
2. Payments / Fintech:
   - Revert operations in a payment pipeline or state machine
3. Analytics / Logs:
   - Stack-based computation for expression evaluation or tree traversal

8) Pitfalls & Refactors

1. Pitfalls:
   - `ArrayDeque` does not allow null elements → NullPointerException
   - `Stack` legacy → avoid synchronized overhead unless necessary
2. Refactors / Improvements:
   - Wrap `ArrayDeque` with synchronized block for multi-threaded access
   - Use LinkedList if extremely large dynamic stack is needed with frequent resizing

9) Interview Follow-ups (1-Line Answers)

1. Time complexity? → O(1) amortized for push/pop/peek
2. Space complexity? → O(n) array + object references
3. Thread safety? → Not thread-safe; use `Collections.synchronizedDeque()`
4. Why ArrayDeque over Stack? → Legacy Stack is synchronized + slower
5. Real-world use case? → Undo/Redo, transaction rollback, DFS traversal

Q9: Sort a List of Objects by Custom Comparator

1) Overview

- Objective: Sort a list of complex objects based on custom fields
- Core idea: Use `Comparator` or lambda expressions with `Collections.sort()` or `List.sort()`
- MAANG emphasis: Efficiency, stable sorting, JVM optimizations, memory impact

2) Java Code Example (Copy-Pasteable)

```
import java.util.*;

class Employee {
   int id;
   String name;
   double salary;

   public Employee(int id, String name, double salary) {
      this.id = id;
      this.name = name;
      this.salary = salary;
   }

   @Override
   public String toString() {
      return "Employee{" + "id=" + id + ", name='" + name + '\'' + ", salary=" + salary + '}';
   }
}

public class CustomSortExample {
   public static void main(String[] args) {
      List<Employee> employees = new ArrayList<>(Arrays.asList(
         new Employee(3, "Alice", 70000),
         new Employee(1, "Bob", 50000),
         new Employee(2, "Charlie", 60000)
      ));

      // Sort by salary ascending
      employees.sort(Comparator.comparingDouble(e -> e.salary));
      System.out.println("Sorted by salary: " + employees);

      // Sort by name lexicographically
      employees.sort(Comparator.comparing(e -> e.name));
      System.out.println("Sorted by name: " + employees);

      // Sort by id descending
      employees.sort(Comparator.comparingInt((Employee e) -> e.id).reversed());
      System.out.println("Sorted by id descending: " + employees);
   }
}
```

Expected Output:

Sorted by salary: [Employee{id=1, name='Bob', salary=50000.0}, Employee{id=2, name='Charlie', salary=60000.0}, Employee{id=3, name='Alice', salary=70000.0}]
Sorted by name: [Employee{id=3, name='Alice', salary=70000.0}, Employee{id=1, name='Bob', salary=50000.0}, Employee{id=2, name='Charlie', salary=60000.0}]
Sorted by id descending: [Employee{id=3, name='Alice', salary=70000.0}, Employee{id=2, name='Charlie', salary=60000.0}, Employee{id=1, name='Bob', salary=50000.0}]

3) Step-by-Step Dry Run (Sort by Salary)

Input: [3:Alice:70000, 1:Bob:50000, 2:Charlie:60000]

1. Comparator compares first pair → Alice vs Bob → 70000 > 50000 → swap
2. Compare next → Alice vs Charlie → 70000 > 60000 → swap
3. Compare Bob vs Charlie → 50000 < 60000 → keep order

Result: [Bob, Charlie, Alice]

4) JVM Internals – Deep Dive

4.1 Sorting Mechanism

- List.sort() uses TimSort (optimized merge sort + insertion sort)
- JVM calls compare() method of Comparator for comparisons
- Internal bytecode executed:
- invokevirtual java/util/Comparator.compare
- if_icmpgt / if_icmplt → branch

4.2 Object Layout

- Each Employee → object header + 3 fields (int, reference, double)
- JIT optimizations:
  - Escape analysis → may eliminate temporary references during sort
  - Inlining → compare() calls
  - Loop unrolling for small arrays

4.3 Memory & GC

- TimSort uses temporary buffer for merging (size ≈ n/2)
- Large lists → may trigger allocation of merge buffers → old arrays GC-eligible after sort

Text-Based Diagram:

Array:
Index 0: Employee{id=3, Alice, 70000}
Index 1: Employee{id=1, Bob, 50000}
Index 2: Employee{id=2, Charlie, 60000}

Comparator calls:
compare(3:Alice, 1:Bob) → 70000>50000 → swap
compare(3:Alice, 2:Charlie) → 70000>60000 → swap
compare(1:Bob, 2:Charlie) → 50000<60000 → keep

5) HotSpot Optimizations

1. **Inlining:** `Comparator.comparingDouble` **and lambda expression**
2. **Branch prediction: predictable for numeric comparisons**
3. **Escape analysis: temporary buffers may be stack-allocated**
4. **Deoptimization: occurs if dynamic class loading changes comparator behavior**

6) Real-World Tie-ins

1. **Spring / Hibernate:**
   - **Sort entities by timestamp, ID, or priority for batch processing**
2. **Payments / Fintech:**
   - **Sort transactions by amount, timestamp, or risk score**
3. **Analytics / Reporting:**
   - **Sort users, logs, or metrics efficiently before aggregation**

7) Pitfalls & Refactors

1. **Pitfalls:**
   - **Null values →** `Comparator.comparing()` **throws NPE unless** `Comparator.nullsFirst()` **used**
   - **Mutable fields → changing sort key after sort breaks assumptions**
2. **Refactors / Improvements:**
   - **Use Comparator chaining for multi-level sort:** `comparing().thenComparing()`
   - **For very large datasets → consider** `parallelSort()` **for multi-core optimization**

8) Interview Follow-ups (1-Line Answers)

1. **Time complexity? → O(n log n) (TimSort)**
2. **Stable sort? → Yes, TimSort preserves relative order**
3. **Null handling? → Use `Comparator.nullsFirst()` or `nullsLast()`**
4. **Memory usage? → Temporary merge buffer ~ n/2 elements**
5. **Real-world use case? → Sorting leaderboard, transaction batch processing, report generation**

Q10: Implement a Map Merge Function

1) Overview

- **Objective: Merge two maps so that for duplicate keys, values are combined using a function (e.g., sum, concatenation)**
- **Java 8+ Feature: `Map.merge()` allows concise and efficient merge operations**
- **MAANG emphasis: Efficient handling of duplicates, hash-based merging, JVM internals, thread-safety implications**

2) Java Code Example (Copy-Pasteable)

```
import java.util.*;

public class MapMergeExample {
    public static void main(String[] args) {
        Map<String, Integer> map1 = new HashMap<>();
        map1.put("A", 10);
        map1.put("B", 20);
        map1.put("C", 30);

        Map<String, Integer> map2 = new HashMap<>();
        map2.put("B", 5);
        map2.put("C", 15);
        map2.put("D", 25);

        // Merge map2 into map1 by summing values of duplicate keys
        map2.forEach((key, value) ->
            map1.merge(key, value, Integer::sum)
        );
```

```
        System.out.println("Merged Map: " + map1);
    }
}
```

Expected Output:

```
Merged Map: {A=10, B=25, C=45, D=25}
```

3) Step-by-Step Dry Run

Input Maps:

```
map1: {A=10, B=20, C=30}
map2: {B=5, C=15, D=25}
```

Merge Steps:

1. B → exists in map1 → 20 + 5 = 25 → update map1
2. C → exists in map1 → 30 + 15 = 45 → update map1
3. D → does not exist → insert 25
4. A → not in map2 → unchanged

Result: {A=10, B=25, C=45, D=25}

4) JVM Internals – Deep Dive

4.1 HashMap & Node Layout

- HashMap bucket array → array of Node<K,V>
- merge(key, value, remappingFunction):

```
default V merge(K key, V value,
            BiFunction<? super V, ? super V, ? extends V> remappingFunction) {
    Objects.requireNonNull(remappingFunction);
    V oldValue = get(key);
    V newValue = (oldValue == null) ? value : remappingFunction.apply(oldValue, value);
    put(key, newValue);
    return newValue;
}
```

- Node header: mark word + klass pointer
- Key/Value references: stored in array → O(1) hash-based placement

4.2 Hashing & Collision Handling

- Compute `hashCode()` → spread → bucket index
- If collision → traverse linked list / tree in bucket → check `equals()` → replace or insert

Text-Based Diagram:

```
Buckets:
[0] -> Node(key:A, value:10)
[1] -> Node(key:B, value:25)
[2] -> Node(key:C, value:45)
[3] -> Node(key:D, value:25)
```

5) HotSpot Optimizations

1. **Inlining:** `merge()`, `get()`, `put()`
2. **Branch prediction:** duplicates predictable → predictable branch for `oldValue == null`
3. **Lambda optimizations:** `Integer::sum` method reference inlined by JIT
4. **Deoptimization:** occurs if key class loads dynamically or hashCode changes at runtime

6) GC & Memory Considerations

- New `Node` objects created only for new keys (e.g., D)
- Existing nodes updated in-place for duplicates → minimal allocation
- Large maps → pre-sizing `HashMap` reduces resizing and GC overhead

7) Real-World Tie-ins

1. Spring / Hibernate:
    - Merge entity attribute maps or configurations
2. Payments / Fintech:
    - Aggregate transaction amounts per user ID or merchant

3. **Analytics / Metrics:**
    - **Merge metrics collected from multiple sources or shards**

8) Pitfalls & Refactors

1. **Pitfalls:**
    - **Mutable keys → hashCode changes break bucket mapping**
    - **Null keys/values → HashMap allows null key but merge() may throw NPE if remappingFunction returns null**
2. **Refactors / Improvements:**
    - **Use ConcurrentHashMap for thread-safe merges**
    - **Compose custom remapping function for complex aggregation: List.concat(), Set.addAll()**

9) Interview Follow-ups (1-Line Answers)

1. **Time complexity? → O(n + m) for merging two maps**
2. **Space complexity? → O(k) for total unique keys**
3. **Null handling? → merge() allows null values if remapping function handles them**
4. **Thread-safety? → Not thread-safe; use ConcurrentHashMap.merge() for multi-threaded merges**
5. **Real-world use case? → Aggregating logs, transactions, or counters from multiple sources**

📌 For Levels 4–5–6-7, you can download the complete in-depth document. I hope this helps you learn faster and grow deeper in your JVM journey. — CoVaib DeepLearn

★ PDF Guides for Every Java Topic

In-deapth java concepts explained at JVM Level

By CoVaib DeepLearn

Level up your Java mastery with complete, deep-dive PDFs for every essential Java concept.

20 Most Ask java Concepts with JVM Mastery Library

🧠 Core Java & OOP

1. Encapsulation Unwrapped

Understand how data hiding, access modifiers & controlled exposure power clean architecture.

— Explore What You'll Learn —

— Download Guide —

🎯 Encapsulation In-Depth Learning Kit (70+ Q&A)

- ✓ 70+ encapsulation-focused interview questions (Basic → Expert)
- ✓ Real-world scenarios: DTOs, entities, service boundaries & API design
- ✓ Code-based exercises on access modifiers, immutability & defensive copying
- ✓ GitHub repo with clean, refactored examples of good vs bad encapsulation
- ✓ Common pitfalls (anemic models, exposing internals, leaking invariants)
- ✓ Level-wise cheat sheets for all 7 interview stages (encapsulation patterns)
- ✓ Structured notes to revise before LLD/HLD + Java design rounds

— Explore Encapsulation In-Depth Kit —

— Access Encapsulation Interview Kit —

2. Inheritance Decoded

Master IS-A relationships, method overriding & class hierarchies the right way.

— Explore What You'll Learn —

— Download Guide —

🎯 Inheritance In-Depth Learning Kit (70+ Q&A)

- ✓ 70+ inheritance & hierarchy questions (from basics to edge cases)
- ✓ Scenarios: when to use inheritance vs composition in real systems
- ✓ Code drills on overriding, abstract classes, interfaces & default methods
- ✓ GitHub repo with hierarchy refactoring exercises & UML → code mapping
- ✓ Common pitfalls: fragile base class, deep inheritance trees, tight coupling
- ✓ Level-wise cheat sheets covering IS-A, LSP, and interface-based design
- ✓ Concise notes to revise before object-oriented design & refactoring rounds

— Explore Inheritance In-Depth Kit —

— Access Inheritance Interview Kit —

3. Polymorphism in Action

Run-time vs compile-time behavior, dynamic dispatch & interview-favorite concepts.

— Explore What You'll Learn —

— Download Guide —

🎯 Polymorphism In-Depth Learning Kit (70+ Q&A)

- ✓ 70+ polymorphism questions (overloading vs overriding, runtime dispatch)
- ✓ Tricky "output prediction" & binding questions used in real interviews
- ✓ Code-based exercises with upcasting, downcasting & interface polymorphism
- ✓ GitHub repo with polymorphic designs & refactoring from if-else to OOP
- ✓ Common pitfalls: wrong assumptions on compile-time vs runtime behavior
- ✓ Level-wise cheat sheets for 7 stages: from basics to strategy pattern usage
- ✓ Snapshot notes to revise before Java/OOP, pattern & system design rounds

— Explore Polymorphism In-Depth Kit —

— Access Polymorphism Interview Kit —

4. Aggregation, Composition & Association

Clear, visual explanation of HAS-A relationships every senior engineer must know.

— Explore What You'll Learn —

— Download Guide —

🎯 HAS-A Design In-Depth Learning Kit (70+ Q&A)

- ✓ 70+ questions on aggregation, composition & associations in real systems
- ✓ Domain modeling scenarios: entities, value objects, services & relationships
- ✓ Code exercises turning requirements → correct HAS-A modeling in Java
- ✓ GitHub repo with UML diagrams mapped to clean, production-style code
- ✓ Common pitfalls: wrong lifecycle modeling, overuse of bi-directional links
- ✓ Level-wise cheat sheets for 7 levels of system modeling & object design
- ✓ Summary notes for LLD/HLD, design review & architecture-focused interviews

— Explore HAS-A In-Depth Kit —

— Access HAS-A Interview Kit —

📚 Core Java Foundations

5. Immutability & Thread-Safe Design

Master defensive copying, safe object construction and memory safety rules.

— Explore What You'll Learn —

— Download Guide —

6. Java Final, Finally, Finalize

Deep clarity on final fields, exception cleanup & GC-level finalize behaviour.

— Explore What You'll Learn —

— Download Guide —

7. Covariance & Contravariance Explained

Wildcards, PECS rule & how generics behave during overriding.

— Explore What You'll Learn —

— Download Guide —

8. Constructor Chaining Masterclass

Learn this(), super() call sequencing & object initialization internals.

— Explore What You'll Learn —

— Download Guide —

JVM & Memory Internals

9. JVM Memory Model Deep Dive

The exact structure of Heap, Stack, Metaspace & GC paths explained visually.

— Explore What You'll Learn —

— Download Guide —

10. String Pool Internals

Understand how the JVM optimizes String memory & why interning matters.

— Explore What You'll Learn —

— Download Guide —

11. How the JVM Loads Classes

Class loaders, delegation model & bytecode loading pipeline.

— Explore What You'll Learn —

— Download Guide —

12. The static Keyword Demystified

How static fields/methods behave in memory + interview tricks.

— Explore What You'll Learn —

— Download Guide —

13. Singleton Pattern in Java

Thread-safe singletons, double-checked locking & enum-based design.

— Explore What You'll Learn —

— Download Guide —

🚀 Concurrency & Multithreading

14. Thread Lifecycle & Scheduler

Every thread state, OS scheduling & concurrency fundamentals made clear.

— Explore What You'll Learn —

— Download Guide —

15. ExecutorService, Future & CompletableFuture

Deep-dive into async pipelines, thread pools & real-world concurrency design.

— Explore What You'll Learn —

— Download Guide —

16. Atomic Variables, volatile & ThreadLocal

Low-level memory visibility, lock-free operations & isolation patterns.

— Explore What You'll Learn —

— Download Guide —

17. Synchronized Collections vs Concurrent Data Structures

Compare synchronized wrappers, ConcurrentHashMap & lock striping.

— Explore What You'll Learn —

— Download Guide —

📚 Collections, Streams & FP

18. Java Collections — Deep Dive

HashMap internals, resizing, load factor & common pitfalls explained.

— Explore What You'll Learn —

— Download Guide —

19. Stream API Explained with Internals

Pipelines, lazy evaluation & parallel stream architecture.

— Explore What You'll Learn —

— Download Guide —

20. Functional Programming in Java

Lambdas, method references, functional interfaces & FP-style design.

— Explore What You'll Learn —

— Download Guide —

🔥 NEW: Combo Pack (Highly Recommended)

★ 20 Most-Asked Java Topics — Complete Mastery Kit

All 20 topics combined into one beautifully structured Java interview kit (350+ pages). Perfect for interview prep, revision & deep learning.

GET THE COMBO KIT

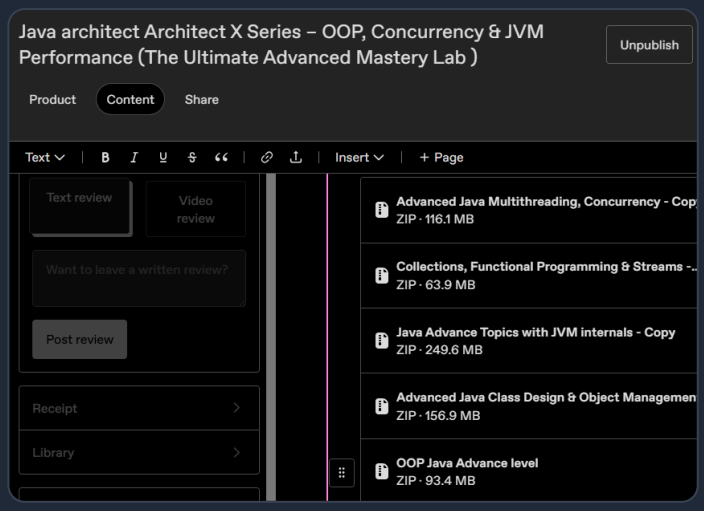
The CoVaib DeepLearn Architect Series

A collection of advanced masterclasses designed for senior and architect-level Java mastery.

1 Java Architect X Series

OOP, Concurrency & JVM Performance

The Ultimate Advanced Mastery Lab



🚀 Highlights:

- OOP Principles (Encapsulation, Inheritance, Polymorphism, Composition, Abstraction)
- JVM Internals: Class Loaders, Memory Areas, Execution Engine
- Concurrency & Performance Optimization
- Real-world Case Studies (Spring Boot, Async REST APIs, Thread Pool Designs)

🧠 Learn to think like the JVM — from object creation to thread scheduling.

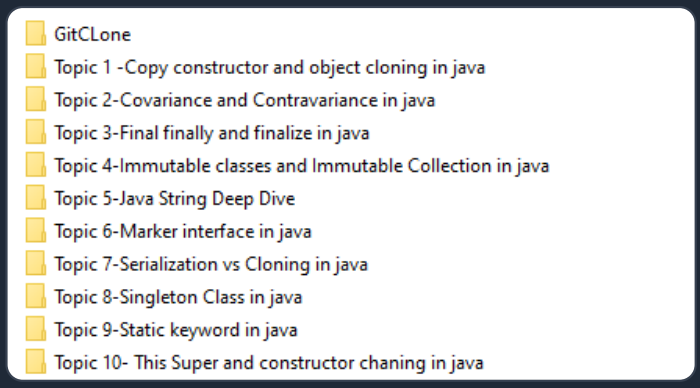
🔗 Get it on Gumroad

— Explore What You'll Learn —

2 Advanced Java Class Design & Object Management

Senior-Level Deep Dive

(Focus: Constructors, cloning, immutability, final/static behaviors, advanced object handling)



🧩 Highlights:

- **this**, **super** & Constructor Chaining – JVM init order, inheritance pitfalls
- Copy Constructor & Object Cloning – Deep vs shallow copies, custom cloning logic
- Singleton Patterns – Enum-based, Reflection & Serialization-safe implementations
- **final**, **finally**, **finalize** – JVM GC lifecycle, cleanup behavior
- Immutable Classes & Collections – Defensive copies, concurrency-safe designs

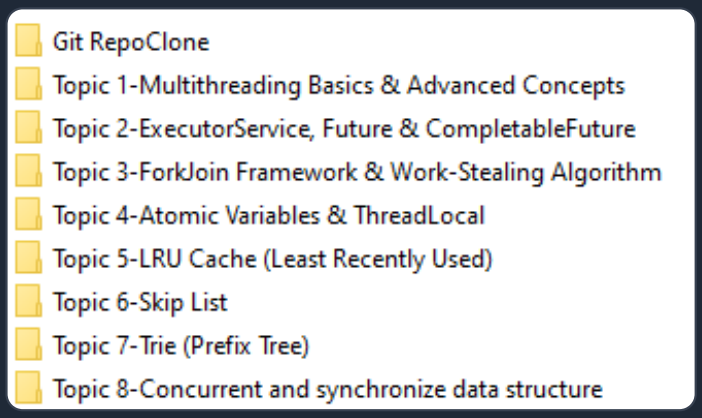
🧠 Master object lifecycle and design Java like a JVM engineer.

🔗 Get it on Gumroad

— Explore What You'll Learn —

3 Advanced Multithreading, Concurrency & High-Performance Data Structures

ExecutorService, CompletableFuture, ForkJoin, and lock-free structures



🧩 Highlights:

- Multithreading Foundations – Thread lifecycle, synchronization, locks, and **volatile**
- ExecutorService, Future & CompletableFuture – Async workflows & parallel composition
- ForkJoin Framework – Work-stealing, divide-and-conquer concurrency
- Atomic & ThreadLocal Utilities – Atomic variables, memory visibility, pitfalls
- ConcurrentSkipListMap & Set – Thread-safe sorted collections

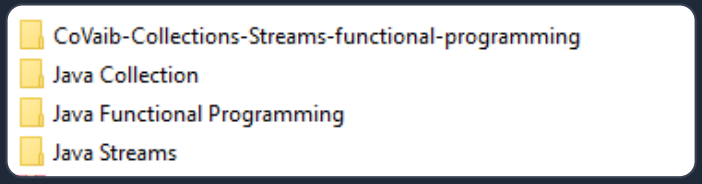
🧠 **Gain production-grade mastery of concurrency & lock-free architectures.**

🔗 **Get it on Gumroad**

— Explore What You'll Learn —

4 Java Collections, Functional Programming & Streams

Senior-Level Deep Dive



🧩 Highlights:

- Java Collections: Deep dive into List, Set, Map, Queue, Deque performance
- Thread-safe collections: ConcurrentHashMap, CopyOnWriteArrayList
- Functional Programming: Lambdas, method references, functional interfaces
- Java Streams: Stream transformations, parallel streams, lazy evaluation
- Collectors: groupingBy, partitioningBy, mapping, joining

🧠 **Master modern Java coding styles — clean, functional, and performant.**

🔗 Get it on Gumroad

— Explore What You'll Learn —

5️⃣ Advanced Java Masterclass

All Advance level Java concepts

All-in-One Masterclass

🧩 Highlights:

- JVM Internals & Performance, Class Loading & Linking
- GC Tuning & Log Analysis, JIT Compiler & HotSpot Optimizations
- Java Memory Model (JMM), High-Performance Data Structures
- NIO, Async I/O & Reactive Programming
- Advanced Concurrency & Virtual Threads (Java 21+)

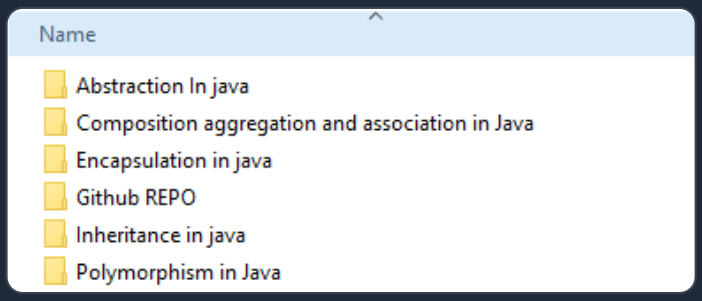
🧠 **The ultimate Java developer transformation — from code to architecture.**

🔗 Get it on Gumroad

— Explore What You'll Learn —

6 Java System-Level OOP

Real-world OOP modeling & architecture design



🧩 Highlights:

- Translating business problems into OOP architecture
- Design principles: SRP, OCP, DIP, LSP
- System-level modeling: entity relationships, composition, inheritance
- Integrating OOP with design patterns (Factory, Strategy, Observer)
- Applying OOP to real-world systems (banking, payments, open banking APIs)

🧠 **Think like a system designer — model scalable, modular architectures in Java.**

🔗 Get it on Gumroad

— Explore What You'll Learn —

"Every CoVaib DeepLearn course is built like an architect's lab — code, bytecode, GC, and real-world systems.

You don’t just learn Java. You master how Java thinks.
”

🔗 Explore the Full Architect Series Here

Level up in code. Level up in consciousness.

“Discipline builds depth. Every line you debug makes you wiser than yesterday.”

Every day, along with one advanced Java question, you’ll also receive a personal reflection quote — to sharpen both logic and life.

Ready to Dominate Your Interview?

Built for developers targeting ₹60LPA+ roles.

90% OFF!

Use Code at Checkout:

`JAVABOOM60BY2025`

Hurry! Offer valid till December!

Get Lifetime Access on Gumroad

- Lifetime Access
- 2000+ Pages of Notes
- 60+ Runnable Projects

Join the Java Architect Circle

Exclusive community for JVM mastery and high-level discussion.

💡 Daily advanced Level questions

🧠 JVM & Concurrency deep dives

💬 Live interview discussions

Join the Architect Circle

“Become the developer who thinks like the JVM.”

Mastery is not about knowing the code, but knowing the runtime.

🔵 ★ **Follow CoVaib DeepLearn — Daily JVM & Java Mastery**

Follow on LinkedIn

Daily JVM internals. Daily system-design-oriented Java. Daily interview mastery.

Stay consistent. Stay curious. Stay DeepLearned. ⚙️🔥