Java Interview Preparation Guide — Ploceus Software Technology

Role: Junior Software Engineer

Contents: Core Java, Advanced Java, Collections, File I/O, Multithreading, Debugging & Eclipse, Coding Problems

Section I — Core Java (25 questions)

Q1. Explain the four pillars of OOP (Encapsulation, Inheritance, Polymorphism, Abstraction).

Answer: Object-oriented programming is based on four pillars: Encapsulation (bundling data and methods, controlling access via access modifiers), Inheritance (a class can extend another to reuse and specialize behavior), Polymorphism (ability to treat objects of different classes through a common interface — compile-time via overloading and run-time via overriding), Abstraction (exposing essential features while hiding implementation details).

Q2. What is the difference between JDK, JRE, and JVM?

Answer: JVM (Java Virtual Machine) executes bytecode on a platform. JRE (Java Runtime Environment) includes JVM and libraries needed to run Java applications. JDK (Java Development Kit) contains JRE plus development tools (compiler, debugger) required to develop Java applications.

Q3. Explain the difference between == and .equals() in Java.

Answer: '==' compares primitives by value and object references by identity (same memory address). '.equals()' is a method intended to compare object content/semantic equality — classes like String override equals to compare contents. Always consider null checks and contract of equals when overriding.

Q4. What is a Java Classloader and how does class loading work?

Answer: ClassLoaders are part of JVM that load class bytecode into the runtime. Bootstrap loads core classes, Extension loads optional libs, and System/Application loads application classes. Loading phases: Loading -> Linking (verify, prepare, resolve) -> Initialization (static initializers). Custom classloaders enable dynamic loading and isolation.

Q5. Describe the difference between static and instance members.

Answer: Static members belong to the class and are shared across all instances; accessed via ClassName.member. Instance members belong to each object instance and require an object reference. Static initializers run once when class is initialized.

Q6. What are Java primitive types and their default values?

Answer: Java primitives: byte(0), short(0), int(0), long(0L), float(0.0f), double(0.0d), char('\u0000'), boolean(false). Local variables are not initialized by default and must be assigned before use.

Q7. Explain autoboxing and unboxing with an example and pitfalls.

Answer: Autoboxing converts primitives to wrapper objects automatically (int to Integer). Unboxing converts wrapper to primitive. Pitfalls: may cause NullPointerException when unboxing a null wrapper; can cause performance overhead and unexpected identity comparisons when using '==' on wrappers.

Q8. What is the final, finally, and finalize in Java?

Answer: 'final' is a modifier for variables (constant), methods (non-overridable), classes (non-inheritable). 'finally' is a block executed after try/catch regardless of exceptions, often for cleanup. 'finalize()' was an Object method called by GC before reclaiming object — deprecated and unreliable, prefer try-with-resources/explicit cleanup.

Q9. Describe Java memory model: Heap vs Stack vs Method Area.

Answer: Stack stores primitive local variables and references; heap stores objects and arrays (shared). Method Area (or metaspace in modern JVMs) stores class metadata, static variables. GC manages heap; stack frames are thread-local and deallocated on method return.

Q10. Explain String immutability and String pool.

Answer: Strings are immutable; operations produce new String objects. Java maintains a String pool (intern pool) where string literals are interned to allow sharing. Using StringBuilder or StringBuffer is recommended for heavy mutable string operations.

Q11. Difference between StringBuilder and StringBuffer.

Answer: Both are mutable sequence of characters. StringBuffer is synchronized (thread-safe) and slower; StringBuilder is unsynchronized and faster — prefer StringBuilder in single-threaded contexts.

Q12. Explain exceptions hierarchy and checked vs unchecked exceptions.

Answer: Exceptions extend Throwable. Checked exceptions (extend Exception but not RuntimeException) must be declared/handled at compile time (e.g., IOException). Unchecked exceptions (RuntimeException and Error) need not be declared and often indicate programming errors (NullPointerException).

Q13. What is try-with-resources and why use it?

Answer: Introduced in Java 7, try-with-resources automatically closes resources implementing AutoCloseable at the end of the block, even if exceptions occur—reduces boilerplate and avoids resource leaks. Example: try (BufferedReader br = new BufferedReader(...)) { ... }

Q14. What are enums and advantages over constants?

Answer: Enums are special classes representing fixed sets of constants with type-safety and ability to add fields, methods, and implement interfaces. They avoid errors of integer/String constants and can have behavior per constant.

Q15. Explain Java annotations and common built-in ones.

Answer: Annotations provide metadata for classes, methods, fields. Built-in: @Override (compile-time checking), @Deprecated, @SuppressWarnings. Annotations can be processed at source, class, or runtime via reflection and custom annotation processors.

Q16. What is reflection and when to use it?

Answer: Reflection allows inspecting and manipulating classes, methods, fields at runtime (Class.forName, getMethods, invoke). Useful for frameworks, dependency injection, serializers, but has performance cost and breaks encapsulation.

Q17. Explain Serializable and transient keyword.

Answer: Serializable is a marker interface enabling object serialization. 'transient' marks fields that should not be serialized (e.g., passwords). Provide serialVersionUID for version control; implement custom readObject/writeObject for custom behavior.

Q18. What is garbage collection and can you force GC?

Answer: GC reclaims unused heap objects. You cannot force GC reliably; System.gc() is a hint. Use appropriate memory management, weak references, and tuning (GC algorithms) rather than relying on explicit GC calls.

Q19. Explain access modifiers (public, protected, default, private) and their scope.

Answer: public (accessible everywhere), protected (same package + subclasses), default/package-private (same package only), private (class-only). For nested classes, rules have additional considerations.

Q20. What is method overloading vs overriding?

Answer: Overloading — same method name different parameter lists within same class (compile-time polymorphism). Overriding — subclass provides specific implementation for a superclass method (same signature), enabling runtime polymorphism; use @Override for clarity.

Q21. Explain 'instanceof' and pattern matching (if available).

Answer: 'instanceof' checks runtime type compatibility (null-safe). Recent Java versions add pattern matching for instanceof to both test and cast in one expression: if (obj instanceof String s) { ... }

Q22. What is a functional interface and lambda expression?

Answer: A functional interface has a single abstract method (e.g., Runnable, Comparator). Lambda expressions provide concise implementation: (a, b) -> a + b. They enable functional-style programming and work with streams.

Q23. Explain Stream API basics (map, filter, reduce).

Answer: Streams provide declarative operations on collections: 'map' transforms elements, 'filter' selects, 'reduce' aggregates. Streams can be sequential or parallel. Avoid stateful operations and side-effects; close underlying resources if streams wrap IO.

Q24. Describe default and static methods in interfaces.

Answer: Java 8+ allows interfaces to have default methods with implementation and static methods. Default methods provide backward compatibility to extend interfaces without breaking implementors.

Q25. Explain volatile keyword and when to use it.

Answer: 'volatile' ensures visibility of writes to variables across threads and prevents instruction reordering for that variable. It does not provide atomicity for compound operations; use synchronized or atomic classes for atomicity.

Q26. What is synchronization and intrinsic locks (monitor)?

Answer: synchronized keyword acquires a monitor (intrinsic lock) on object/class to ensure mutual exclusion for critical sections. Methods or blocks can be synchronized. Be careful to avoid deadlocks and minimize lock scope.

Section II — Advanced Java (20 questions)

Q27. What is JDBC and how do you connect to a database in Java?

Answer: JDBC (Java Database Connectivity) is an API for interacting with relational DBs. Typical steps: load driver (modern drivers auto-register), obtain Connection via DriverManager.getConnection(url,user,pass), create PreparedStatement, execute queries, process ResultSet, and close resources (prefer try-with-resources).

Q28. Explain PreparedStatement vs Statement.

Answer: Statement builds SQL via concatenation — vulnerable to SQL injection and less efficient. PreparedStatement precompiles SQL with placeholders, allows parameter binding (safer), and can be reused with different parameters improving performance.

Q29. What are connection pools and why use them?

Answer: Connection pools reuse DB connections to avoid costly creation for each request. Pools (HikariCP, C3P0) improve throughput, manage max connections, and provide monitoring and timeout mechanisms.

Q30. Describe transactions and isolation levels.

Answer: Transactions ensure atomicity of multiple DB operations. Isolation levels (READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE) control visibility of changes across transactions affecting phenomena like dirty reads, non-repeatable reads, and phantom reads. Use appropriate isolation for correctness vs performance.

Q31. Explain Servlet lifecycle and when doGet vs doPost is used.

Answer: Servlet lifecycle: init -> service -> doGet/doPost/... -> destroy. doGet handles idempotent requests (read), doPost for state-changing operations (create/update). Always validate input and manage thread-safety in servlets.

Q32. What is JSP and difference from Servlet?

Answer: JSP (JavaServer Pages) allows embedding Java in HTML and is translated into a servlet at runtime. Use MVC patterns and keep logic in servlets/beans rather than JSP for maintainable code.

Q33. Explain Java EE vs Java SE.

Answer: Java SE is core platform (language, core libraries). Java EE (now Jakarta EE) builds on SE adding enterprise specs: Servlets, JSP, EJB, JPA, JMS, etc., for large-scale server-side apps.

Q34. What is JPA and ORM; differences between Hibernate and JDBC?

Answer: JPA (Java Persistence API) is a specification for object-relational mapping (ORM). Hibernate is a popular JPA implementation offering mapping between Java objects and DB tables, caching, lazy loading, and HQL. JDBC is low-level SQL-based interaction; ORM simplifies CRUD and models domain objects.

Q35. Explain connection between JPA entities and lazy vs eager fetching.

Answer: JPA entities can mark relationships as LAZY or EAGER. LAZY loads related entities on-demand (proxy) which helps performance but requires active session; EAGER fetches immediately which can cause N+1 query issues. Choose strategy based on use-case.

Q36. What are design patterns commonly used in Java?

Answer: Common patterns: Singleton, Factory, Builder, DAO, Repository, Observer, Strategy, Decorator. Knowledge of appropriate pattern selection improves maintainability and architecture.

Q37. Explain dependency injection and frameworks (Spring).

Answer: DI inverts control of object creation to a container (e.g., Spring) which injects dependencies, improving testability and decoupling. Spring provides annotations (@Component, @Autowired), configuration, and many modules for web, data, security.

Q38. What is Spring Boot and advantages?

Answer: Spring Boot simplifies creating production-ready Spring applications with embedded servers, auto-configuration, opinionated defaults, and fast startup—reducing boilerplate config.

Q39. Explain RESTful services and common HTTP status codes.

Answer: REST uses HTTP verbs (GET, POST, PUT, DELETE) and resource URIs. Common status codes: 200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Internal Server Error. Use appropriate codes for API semantics.

Q40. What is JWT and how is it used for authentication?

Answer: JWT (JSON Web Token) is a compact token format with header, payload, signature. Used for stateless authentication; server issues token after login, clients present token in Authorization header for subsequent requests; verify signature and claims on server.

Q41. Explain microservices basics and pros/cons compared to monolith.

Answer: Microservices split app into small services communicating via network (HTTP, gRPC). Pros: independent deployment, scalability, polyglot; cons: distributed complexity, service discovery, data consistency, and operational overhead.

Q42. What is WebSocket and when to use?

Answer: WebSocket provides full-duplex persistent connection for real-time bidirectional communication (chat, live updates) unlike request-response HTTP. Requires server and client support; handle connection lifecycle and scaling.

Q43. Explain classpath and module path (Java 9+) differences.

Answer: Classpath is traditional search path for classes/resources. Module system (Java 9) introduces module path with explicit module-info.java, strong encapsulation, and reliable configuration. Modules reduce runtime conflicts and improve security.

Q44. How do you profile and tune JVM performance?

Answer: Use tools like VisualVM, JFR (Java Flight Recorder), jstat, jmap to find GC issues, memory leaks, and CPU hotspots. Tune heap size, GC algorithm, thread pools, and analyze allocation patterns for performance improvements.

Q45. Explain serialization pitfalls and compatibility.

Answer: Serialization ties object structure to byte stream; changes break compatibility unless handled (serialVersionUID, custom readObject/writeObject, use of Externalizable or DTOs). Prefer version-tolerant formats (JSON, Protobuf) for long-term storage.

Q46. What is reactive programming (Project Reactor) vs traditional blocking IO?

Answer: Reactive programming uses non-blocking, asynchronous streams (Flux, Mono) to handle high concurrency with fewer threads. It requires reactive-enabled stacks (WebFlux) and different error handling; good for IO-bound workloads.

Section III — Collections Framework (15 questions)

Q47. Explain Collection, List, Set, and Map interfaces and their typical implementations.

Answer: Collection is root interface (except Map). List (ordered, allows duplicates) implementations: ArrayList (resizable array), LinkedList (doubly-linked). Set (no duplicates): HashSet (hashing), LinkedHashSet (insertion order), TreeSet (sorted). Map (key-value): HashMap, LinkedHashMap, TreeMap (sorted).

Q48. How does HashMap work internally? Explain capacity, load factor, and rehashing.

Answer: HashMap uses array of buckets; each bucket holds linked list or tree (after threshold). Hash code of key modulo capacity picks bucket. Load factor (default 0.75) triggers resizing when size exceeds capacity*loadFactor. Rehashing recalculates bucket positions—costly operation amortized over inserts.

Q49. What is difference between HashMap and ConcurrentHashMap?

Answer: HashMap is not thread-safe. ConcurrentHashMap supports concurrent access via segmented/bucket-level locking (in modern Java uses CAS and internal concurrency controls). It avoids locking whole map and allows high throughput concurrent reads/writes.

Q50. Explain fail-fast vs fail-safe iterators.

Answer: Fail-fast iterators (ArrayList, HashMap) detect structural modification during iteration and throw ConcurrentModificationException. Fail-safe iterators (CopyOnWriteArrayList, concurrent collections) iterate over a snapshot and do not throw but may not reflect recent changes.

Q51. How to iterate over a Map efficiently?

Answer: Use entrySet() to retrieve Map.Entry objects and iterate; avoids extra get() calls. Example: for (Map.Entry e : map.entrySet()) { ... }. In Java 8+, use map.forEach((k,v)->...).

Q52. When to use ArrayList vs LinkedList?

Answer: ArrayList provides random access and better cache locality—good for frequent reads. LinkedList excels in frequent insertions/removals at ends or when you need deque operations. For most use-cases ArrayList is preferred.

Q53. Explain TreeMap and how it orders keys.

Answer: TreeMap implements NavigableMap and stores entries in a Red-Black tree ordered by natural order or provided Comparator. It provides methods like subMap, headMap and operations in O(log n).

Q54. What are PriorityQueue's use-cases and complexity?

Answer: PriorityQueue is a heap-based queue providing O(log n) insert and remove-min operations. Use for scheduling, Dijkstra algorithm, or choosing top-k elements. Not thread-safe.

Q55. Explain difference between Comparable and Comparator.

Answer: Comparable defines natural ordering via compareTo within class. Comparator defines external ordering and can be implemented separately or provided as lambda; allows multiple sorting strategies.

Q56. What is EnumSet and EnumMap; advantages?

Answer: Specialized Set/Map implementations for enum keys with compact and fast bitset-based implementation. Much faster and memory-efficient compared to regular collections for enum types.

Q57. How does LinkedHashMap maintain order and how to implement LRU cache?

Answer: LinkedHashMap maintains insertion or access order via doubly-linked list. Override removeEldestEntry to implement LRU eviction: new LinkedHashMap(capacity, load, true) { protected boolean removeEldestEntry(Map.Entry e) { return size() > max; } }

Q58. Explain Collections.synchronizedList vs CopyOnWriteArrayList.

Answer: Collections.synchronizedList wraps list with synchronized methods — coarse-grained locking. CopyOnWriteArrayList creates snapshot copy on write, safe for concurrent reads with rare writes; good for observer lists.

Q59. What are BlockingQueue implementations and producer-consumer?

Answer: BlockingQueue supports blocking put/take operations. Implementations: ArrayBlockingQueue (fixed size), LinkedBlockingQueue (optionally bounded), PriorityBlockingQueue. Used for producer-consumer patterns with Executors and thread pools.

Q60. Explain identityHashMap use-case.

Answer: IdentityHashMap uses '==' for key equality instead of equals() — useful for low-level systems where identity semantics are required, e.g., object graph algorithms or maintaining distinct entries per instance.

Q61. How to choose between Collections and Streams for processing?

Answer: Use Collections API for mutating operations and index-based access; Streams for declarative, pipeline-style transformations, filtering, and parallel processing. Streams avoid explicit loops and can be parallelized but avoid when you need fine-grained control or mutation.

Section IV — File I/O (10 questions)

Q62. Explain InputStream vs Reader and OutputStream vs Writer.

Answer: InputStream/OutputStream handle byte-oriented IO; Reader/Writer handle character-oriented IO (respecting encoding). Use streams for binary data (images), readers/writers for text. Wrap streams with buffered variants for performance.

Q63. Explain FileInputStream, FileOutputStream, FileReader, FileWriter and Buffered variants.

Answer: FileInputStream/FileOutputStream read/write bytes. FileReader/FileWriter read/write chars. BufferedInputStream/BufferedReader wrap underlying to reduce system calls by buffering data—improves performance significantly.

Q64. How does FileChannel and NIO differ from classic IO?

Answer: NIO (New IO) introduces Channels, Buffers, Selectors for non-blocking IO and potentially better performance. FileChannel provides zero-copy transfers (transferTo/From) and memory-mapped files via MappedByteBuffer enabling efficient large-file access.

Q65. Explain serialization vs externalization and when to use them.

Answer: Serializable uses default serialization or custom readObject/writeObject; Externalizable requires implementing readExternal/writeExternal giving full control and potentially smaller output but more code. For long-term storage use explicit formats (JSON, protobuf) for compatibility.

Q66. How to read large files efficiently in Java?

Answer: Use BufferedReader with FileReader for text, use streaming (process line-by-line with Files.lines in Java 8), or use FileChannel with MappedByteBuffer for very large files. Avoid reading entire file into memory.

Q67. What is try-with-resources for file handling?

Answer: Ensures streams/readers are closed automatically; reduces resource leakage. Example: try (BufferedReader br = Files.newBufferedReader(path)) { ... }

Q68. Explain File vs Path (java.io vs java.nio.file).

Answer: java.io.File is older API; java.nio.file.Path (Java 7+) is richer, supports symbolic links, file attributes, and operations via Files utility methods. Prefer Path/Files for modern code.

Q69. How to write binary data and read it back (DataOutputStream/DataInputStream)?

Answer: DataOutputStream wraps OutputStream to write primitives in portable binary format (writeInt, writeDouble). DataInputStream reads them back in the same order. Use consistent encoding and handle EOF.

Q70. How to handle character encoding when reading/writing files?

Answer: Always specify charset (UTF-8) when converting bytes to chars: new InputStreamReader(stream, StandardCharsets.UTF_8) or Files.newBufferedReader(path, StandardCharsets.UTF_8). Avoid platform default encoding.

Q71. Explain WatchService API for file change notifications.

Answer: java.nio.file.WatchService allows registering directories to watch for create/delete/modify events. Useful for applications that monitor file system changes (auto-reload).

Section V — Multithreading & Concurrency (15 questions)

Q72. How do you create a thread in Java? Explain Runnable vs Thread subclass.

Answer: Implement Runnable and pass to new Thread(runnable) or extend Thread and override run(). Prefer Runnable (separation of task and thread) and use ExecutorService for managing thread pools.

Q73. Explain ExecutorService and advantages over raw threads.

Answer: ExecutorService manages a pool of threads and queues tasks, offering lifecycle management, configurable pools (fixed, cached), scheduling, and futures for async results. It centralizes thread management and avoids manual thread creation.

Q74. What are Future and CompletableFuture? Show example.

Answer: Future represents result of async computation; blocking get() waits for completion. CompletableFuture supports non-blocking composition, chaining, and combining async tasks using thenApply, thenCompose, exceptionally.

```
import java.util.concurrent.*;
public class CompletableFutureExample {
    public static void main(String[] args) {
        CompletableFuture<Integer> cf = CompletableFuture.supplyAsync(() -> {
            // simulating long computation
            return 2 + 3;
        }).thenApply(n -> n * 2)
          .exceptionally(ex -> { System.out.println("Error: " + ex); return -1; });
        try {
            System.out.println("Result: " + cf.get());
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

Q75. Explain synchronized vs ReentrantLock.

Answer: synchronized is simpler intrinsic locking. ReentrantLock (from java.util.concurrent.locks) provides advanced features: tryLock, lockInterruptibly, fairness policies, and Condition objects for fine-grained signaling. Use ReentrantLock when advanced control is needed.

Q76. What is a volatile variable? How is it different from atomic classes?

Answer: volatile ensures visibility across threads but not atomicity for compound operations. Atomic classes (AtomicInteger, AtomicReference) provide atomic read-modify-write operations via CAS ensuring thread-safety for specific atomic operations.

Q77. Explain CountDownLatch vs CyclicBarrier vs Phaser.

Answer: CountDownLatch waits for other threads to reach a count once (one-shot). CyclicBarrier waits until a fixed number of threads reach the barrier and can be reused. Phaser is more flexible and supports dynamic registration/deregistration of parties.

Q78. How to avoid deadlock and detect it?

Answer: Avoid nested locks, acquire locks in a consistent order, use tryLock with timeout, minimize lock scope. For detection use thread dumps (jstack), or tools like VisualVM; design to prevent deadlock rather than rely on detection.

Q79. Explain thread-safe collections in java.util.concurrent package.

Answer: Collections such as ConcurrentHashMap, CopyOnWriteArrayList, ConcurrentLinkedQueue, BlockingQueues are designed for concurrent access, using non-blocking algorithms or internal concurrency controls to provide thread-safety and high throughput.

Q80. What are atomic operations and CAS?

Answer: CAS (Compare-And-Swap) is an atomic primitive where a value is updated only if it matches an expected value. Used in Atomic* classes and lock-free algorithms to avoid locks and reduce contention; may suffer from ABA problem which can be mitigated with versioning.

Q81. Explain ThreadLocal and use-cases.

Answer: ThreadLocal provides per-thread variables — each thread has its own isolated copy. Useful for per-request context (e.g., SimpleDateFormat not thread-safe) or storing security/context info without passing parameters. Clean up values to avoid memory leaks in thread pools.

Q82. How do you implement producer-consumer pattern in Java?

Answer: Use BlockingQueue (e.g., ArrayBlockingQueue). Producers put() items; consumers take() items; the queue handles blocking behavior and capacity. This avoids manual wait/notify and reduces concurrency bugs.

Q83. Explain fork-join framework and parallelism.

Answer: ForkJoinPool supports divide-and-conquer parallelism using work-stealing. Tasks extend RecursiveTask/RecursiveAction and split work into subtasks; suitable for CPU-bound parallel computations where tasks can be recursively divided.

Q84. What is thread starvation and livelock?

Answer: Starvation occurs when a thread never gets CPU/locks due to resource scheduling; livelock is when threads are active but keep changing state to avoid conflict and make no progress (e.g., politeness loops). Use fair locks or different designs to mitigate.

Q85. How to safely publish objects between threads?

Answer: Use final fields, proper synchronization, volatile references, or safely-constructed objects before publishing (initialize fully before sharing). Avoid exposing 'this' during construction and use immutable objects when possible.

Q86. Explain memory consistency errors and happens-before relationship.

Answer: Java Memory Model defines happens-before relationships (synchronization actions) to ensure visibility and ordering across threads. Actions that happen-before others guarantee visibility of writes to subsequent reads; volatile, locks, thread start/join create happens-before edges.

Section VI — Debugging & Eclipse IDE (5 questions)

Q87. What is a typical debugging workflow in Java?

Answer: Reproduce the problem, add logs (use logging frameworks), run with debugger, set breakpoints, inspect stack traces and variable state, step through code, identify root cause, write tests, and fix. Logging and unit tests prevent regressions.

Q88. How to use breakpoints, watch expressions, and evaluate in Eclipse?

Answer: In Eclipse, open Debug perspective, set breakpoints by double-clicking gutter, use Variables view to inspect, add watch expressions, and use 'Display' or 'Inspect' to evaluate expressions at runtime. Conditional breakpoints reduce noise.

Q89. How to analyze a heap dump and thread dump?

Answer: Use tools like Eclipse MAT (Memory Analyzer), VisualVM, or jmap/jstack. Heap analysis finds memory leaks (largest objects, GC roots). Thread dumps show thread states and lock contention—use to diagnose deadlocks or blocked threads.

Q90. What JVM arguments help debug (e.g., -Xmx, -Xms, -XX:+HeapDumpOnOutOfMemoryError)?

Answer: -Xmx sets max heap, -Xms initial heap. -XX:+HeapDumpOnOutOfMemoryError generates heap dump on OOME. Use GC logging flags (-Xlog:gc*) and JFR for profiling. Tune GC and memory settings based on profiling.

Q91. How to set up remote debugging for a running JVM?

Answer: Start JVM with debug options: -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005 and connect from IDE to host:port. Use secure networks and avoid enabling in production without safeguards.

Section VII — Coding Problems (10 questions)

Q92. Reverse a string in Java (in-place for char array).

Answer: Convert to char array and swap characters from ends towards center. O(n) time, O(1) extra space. Example code below.

```
public class ReverseInPlace {
    public static void reverse(char[] s) {
        int i = 0, j = s.length - 1;
        while (i < j) {
            char tmp = s[i];
            s[i++] = s[j];
            s[j--] = tmp;
        }
    }
    public static void main(String[] args) {
        char[] data = "hello".toCharArray();
        reverse(data);
        System.out.println(new String(data)); // olleh
    }
}
```

Q93. Find first non-repeating character in a String.

Answer: Use LinkedHashMap to preserve order and counts; first entry with value 1 is answer. O(n) time and O(k) space where k is alphabet size. Example code below.

```
import java.util.*;
public class FirstNonRepeating {
    public static Character firstNonRepeating(String s) {
        Map<Character, Integer> map = new LinkedHashMap<>();
        for (char c : s.toCharArray()) map.put(c, map.getOrDefault(c,0)+1);
        for (Map.Entry<Character,Integer> e : map.entrySet()) if (e.getValue()==1) return e.getKey();
        return null;
    }
    public static void main(String[] args) {
        System.out.println(firstNonRepeating("swiss")); // w
    }
}
```

Q94. Implement a thread-safe LRU cache using LinkedHashMap.

Answer: Use LinkedHashMap with accessOrder=true and override removeEldestEntry; wrap with synchronization or use ConcurrentLinkedHashMap implementations for concurrency. Example code below.

```
import java.util.*;
public class LRUCache<K,V> extends LinkedHashMap<K,V> {
    private final int capacity;
    public LRUCache(int capacity) { super(capacity, 0.75f, true); this.capacity = capacity; }
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) { return size() > capacity; }
}
```

Q95. Read a large CSV file and compute aggregate (sum of column) using streaming.

Answer: Use BufferedReader and process line-by-line splitting by delimiter (or use CSV library), parse needed column, and accumulate. Avoid loading entire file into memory. Example snippet below.

```
import java.io.*;
public class SumCSV {
    public static long sumColumn(File f, int colIndex) throws IOException {
        long sum = 0;
        try (BufferedReader br = new BufferedReader(new FileReader(f))) {
            String line; while ((line = br.readLine())!=null) {
                String[] parts = line.split(",");
                sum += Long.parseLong(parts[colIndex]);
            }
        }
        return sum;
    }
}
```

Q96. Implement producer-consumer using BlockingQueue.

Answer: Use ArrayBlockingQueue and producer puts items while consumer takes items. Example below.

```
import java.util.concurrent.*;
class Producer implements Runnable {
    private final BlockingQueue<Integer> q;
    Producer(BlockingQueue<Integer> q){this.q=q;}
    public void run(){ try { for(int i=0;i<10;i++) q.put(i); q.put(-1); } catch(Exception e){} }
}
class Consumer implements Runnable {
    private final BlockingQueue<Integer> q;
    Consumer(BlockingQueue<Integer> q){this.q=q;}
    public void run(){ try{ int v; while((v=q.take())!=-1) System.out.println(v); }catch(Exception e){} }
}
public class ProdConsExample {
    public static void main(String[] args) {
        BlockingQueue<Integer> q = new ArrayBlockingQueue<>(5);
        new Thread(new Producer(q)).start();
        new Thread(new Consumer(q)).start();
    }
}
```

Q97. Given an array of integers, find two numbers that add to target (return indices).

Answer: Use a HashMap to store value->index while scanning; check complement target - nums[i]. O(n) time. Example code below.

```
import java.util.*;
public class TwoSum {
    public static int[] twoSum(int[] nums, int target) {
        Map<Integer,Integer> m = new HashMap<>();
        for (int i=0;i<nums.length;i++){
            int need = target - nums[i];
            if (m.containsKey(need)) return new int[]{m.get(need), i};
            m.put(nums[i], i);
        }
        return null;
    }
}
```

Q98. Implement merge sort for an int array (recursive).

Answer: Classic divide-and-conquer: split, sort halves, merge. O(n log n) time, O(n) auxiliary. Example below.

```
public class MergeSort {
    public static void mergeSort(int[] a) { if (a.length < 2) return; mergeSort(a,0,a.length-1); }
    private static void mergeSort(int[] a,int l,int r) {
        if (l>=r) return;
        int m=(l+r)/2;
        mergeSort(a,l,m); mergeSort(a,m+1,r);
        int[] tmp = new int[r-l+1]; int i=l,j=m+1,k=0;
        while(i<=m && j<=r) tmp[k++] = (a[i]<=a[j])?a[i++]:a[j++];
        while(i<=m) tmp[k++]=a[i++]; while(j<=r) tmp[k++]=a[j++];
        System.arraycopy(tmp,0,a,l,tmp.length);
    }
}
```

Q99. Write a Java method to detect cycle in a singly linked list.

Answer: Use Floyd's tortoise and hare algorithm — two pointers moving at different speeds; if they meet, a cycle exists. O(n) time, O(1) space. Example below.

```
class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
public class CycleDetect {
    public static boolean hasCycle(ListNode head) {
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next; fast = fast.next.next;
            if (slow == fast) return true;
        }
        return false;
    }
}
```

Q100. Serialize and deserialize a binary tree (preorder with null markers).

Answer: Use recursive preorder to write values and special marker for null; for deserialization use a queue of tokens and rebuild recursively. Example outline below.

```
// Outline (pseudocode)
// serialize(node):
//   if node==null: append "#,"
//   else: append node.val + ","; serialize(node.left); serialize(node.right);
// deserialize(tokens):
//   val = next token; if val=="#" return null; node = new Node(Integer.parseInt(val)); node.left = deserialize(tokens)
```

Appendix: Tips for Interview Preparation

• Practice writing code on a whiteboard and in an IDE; focus on clarity and space/time complexity.

• Be ready to explain trade-offs (why ArrayList over LinkedList, when to use synchronized vs concurrent collections).

• Prepare 2-3 projects to discuss—architecture, your contribution, challenges, and learnings.

• Revise core data structures and algorithms, and practice medium-level problems on arrays, strings, hashing, and recursion.

• Read up on Java 8+ features (streams, lambdas, Optional, new Date/Time API) and concurrency utilities.