GENERAL

java coding interview questions write code

Published: September 3, 2026 | Index ID: #-

Introduction

Landing a role at a top tech company or a fast growing startup often hinges on how you perform in a**Java coding interview**. Recruiters look for candidates who can solve complex problems, write clean and efficient code, and explain their thought process clearly. To stand out, you need to master a set of **Java coding interview questions write code** that cover core language features, data structures, algorithms, and system design concepts.

In this article we'll dive deep into the most common Java interview questions, provide practical code examples, and share proven strategies for answering them with confidence. Whether you're a fresh graduate, a seasoned developer, or a senior engineer looking to sharpen your interview skills, this guide will equip you with the knowledge and tools you need to ace your next interview.

Why Java Coding Interview Questions Matter

Java remains one of the most widely used programming languages in enterprise, Android, and backend development. Interviewers use Java questions for several reasons:

- Language Familiarity: They want to see how comfortable you are with Java's syntax, standard library, and idiomatic patterns.
- Problem Solving Ability: Coding questions test your analytical thinking, algorithmic knowledge, and ability to translate requirements into code.
- Code Quality: Interviewers assess readability, maintainability, and adherence to best practices.
- Team Fit: Your approach to collaboration, debugging, and documentation reflects how you'll work in a real team.

Because of these factors, a solid grasp of **Java coding interview questions write code**can dramatically improve your chances of getting hired.

Common Themes in Java Coding Interviews

While every interview is unique, certain themes recur across companies:

1. Data Structures & Algorithms: Linked lists, trees, graphs, hash tables, and sorting/searching algorithms.
2. Object Oriented Design: Design patterns, SOLID principles, and class hierarchies.
3. Concurrency & Threading: Synchronization, deadlock, thread safety, and Java concurrency utilities.
4. Java Language Features: Streams, lambda expressions, generics, and exception handling.
5. System Design & Architecture: Scalable services, microservices, caching, and database design.
6. Testing & Debugging: Unit tests, mocking frameworks, and performance profiling.

Below we'll explore each theme with representative questions and code snippets.

Data Structures & Algorithms

Linked List Manipulation

Linked lists are a staple of Java interview questions. A classic problem is reversing a singly linked list.

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode nextTemp = head.next;
        head.next = prev;
        prev = head;
        head = nextTemp;
    }
    return prev;
}
```

This solution runs in O(n) time and O(1) space, which is the expected optimal answer. When explaining it, emphasize the pointer manipulation and the invariant that prev always points to the reversed part.

Tree Traversal

Binary tree traversals test your recursion skills. Consider printing a tree in inorder.

```
public void inorder(TreeNode root) {
    if (root == null) return;
    inorder(root.left);
    System.out.print(root.val + " ");
    inorder(root.right);
}
```

Discuss the call stack depth and how recursion naturally mirrors the tree structure.

Graph Algorithms

Depth First Search (DFS) and Breadth First Search (BFS) are common. For example, checking if a graph is a tree.

```
public boolean isTree(int n, List&lt;List&lt;Integer&gt;&gt; graph) {
    boolean[] visited = new boolean[n];
    return dfs(0, -1, graph, visited);
}

private boolean dfs(int node, int parent, List&lt;List&lt;Integer&gt;&gt; graph, boolean[] visited) {
    if (visited[node]) return false;
    visited[node] = true;
    for (int neighbor : graph.get(node)) {
        if (neighbor == parent) continue;
        if (!dfs(neighbor, node, graph, visited)) return false;
```

```
    }
    return true;
}
```

Explain cycle detection and the necessity of tracking the parent node to avoid false positives.

HashMap & Counting Problems

Counting duplicate words in a string is a frequent interview problem.

```
public Map&lt;String, Integer&gt; wordCount(String sentence) {
    Map&lt;String, Integer&gt; map = new HashMap();
    for (String word : sentence.split("\\s+")) {
        map.put(word, map.getOrDefault(word, 0) + 1);
    }
    return map;
}
```

Highlight the use of getOrDefault for concise code and discuss potential performance considerations with large inputs.

Sorting & Searching

Binary search on a sorted array is a must know. Here's an implementation that returns the index of a target or -1 if not found.

```
public int binarySearch(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left
```

Discuss time complexity and the importance of preventing integer overflow in the mid calculation.

Object Oriented Design

Design a Bank Account System

Interviewers often ask you to sketch a small domain model. A simple bank account system can demonstrate encapsulation, inheritance, and interfaces.

```
public interface Account {
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();
}

public class SavingsAccount implements Account {
    private double balance;
    private double interestRate;

    public SavingsAccount(double initial, double rate) {
        this.balance = initial;
```

```
        this.interestRate = rate;
    }

    @Override
    public void deposit(double amount) { balance += amount; }

    @Override
    public void withdraw(double amount) {
        if (balance
```

Explain why you chose an interface, how you enforce encapsulation, and how the applyInterest method demonstrates domain logic.

Implementing the Observer Pattern

Java’s java.util.Observer interface is deprecated, so you’ll often need to implement a custom observer.

```
public interface Observer {
    void update(String event);
}

public class EventSource {
    private List&lt;Observer&gt; observers = new ArrayList();

    public void subscribe(Observer o) { observers.add(o); }

    public void unsubscribe(Observer o) { observers.remove(o); }

    public void notifyObservers(String event) {
        for (Observer o : observers) {
            o.update(event);
        }
    }
}
```

Discuss the benefits of loose coupling and how this pattern scales with many observers.

Designing a Thread Safe Cache

Thread safety is critical in many real world applications. A simple LRU cache can illustrate synchronization.

```
public class LRUCache {
    private final int capacity;
    private final Map&lt;K, V&gt; map;
    private final LinkedList&lt;K&gt; order = new LinkedList();

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.map = new HashMap();
```

```
    }

    public synchronized V get(K key) {
        if (!map.containsKey(key)) return null;
        order.remove(key);
        order.addFirst(key);
        return map.get(key);
    }

    public synchronized void put(K key, V value) {
        if (map.containsKey(key)) {
            order.remove(key);
        } else if (map.size() == capacity) {
            K last = order.removeLast();
            map.remove(last);
        }
        order.addFirst(key);
        map.put(key, value);
    }
}
```

Explain the use of synchronized, the eviction policy, and potential