Core Java Interview Questions For 3 Years Experience

Core Java Interview Questions For 3 Years Experience *Downloaded from old.oplm.com by Riley & Roberson*

MASTERING THE CORE JAVA INTERVIEW: A GUIDE FOR DEVELOPERS WITH 3 YEARS OF EXPERIENCE

After three years of working with Java, you have moved beyond the basics. You have built applications, debugged production issues, and collaborated on team projects. Interviewers now expect a deeper understanding of not just how to write code, but why it works the way it does. They want to assess your grasp of core principles, your ability to design robust systems, and your familiarity with the language's advanced features.

This article is crafted specifically for developers who have around three years of hands-on experience. We will explore the most important Core Java interview questions that frequently appear for this experience level. Instead of just listing questions with short answers, we will delve into concepts, common pitfalls, and best practices. Use this guide to strengthen your preparation and approach your next interview with confidence.

OBJECT-ORIENTED PROGRAMMING (OOP) IN DEPTH

While OOP is taught early, three years of experience should demonstrate that you apply these principles thoughtfully in real-world designs. Interviewers will probe beyond definitions.

Explain the SOLID principles with Java examples.

The SOLID principles are the foundation of maintainable object-oriented code. You should be able to discuss each one:

- **Single Responsibility Principle:** A class should have only one reason to change. For example, a *ReportGenerator* class should generate a report, but not also format it for PDF and send it via email. Those responsibilities belong to separate classes.
- **Open/Closed Principle:** Classes should be open for extension but closed for modification. Use inheritance or interfaces rather than modifying existing code. For instance, instead of adding an *if-else* block for a new payment type, implement a *PaymentStrategy* interface and add new implementations.
- **Liskov Substitution Principle:** Subtypes must be substitutable for their base types without altering correctness. A classic violation is a *Square* extending *Rectangle* and overriding setWidth/setHeight in a way that breaks expectations.
- **Interface Segregation Principle:** Clients should not be forced to depend on interfaces they do not use. Split large interfaces into smaller, more specific ones.
- **Dependency Inversion Principle:** High-level modules should not depend on low-level modules; both should depend on abstractions. Use dependency injection to decouple components.

How does Java handle multiple inheritance?

Java does not support multiple inheritance of classes to avoid the diamond problem. However, multiple inheritance of type is possible through interfaces. Since Java 8, interfaces can have default methods, which reintroduced a form of the diamond problem. Java resolves this by requiring an explicit override when a class implements two interfaces with the same default method, or by using the super keyword with the interface name (e.g., *InterfaceName.super.method()*).

COLLECTIONS FRAMEWORK – BEYOND THE BASICS

With three years of experience, you should not only know the differences between List, Set, and Map, but also understand internal workings and performance trade-offs.

How does HashMap work internally? What is the impact of load factor?

HashMap uses an array of buckets. Each bucket is a linked list or a tree (since Java 8). When you put a key-value pair, the hashCode of the key determines the bucket index. If multiple keys land in the same bucket, they are stored as nodes. Once the number of nodes exceeds a threshold (*TREEIFY_THRESHOLD = 8*), the list converts to a balanced tree (red-black tree) for faster search. The *load factor* (default 0.75) controls when the map resizes: when the number of entries exceeds (capacity * load factor), the capacity doubles and entries are rehashed. A lower load factor reduces collisions but wastes memory; a higher load factor saves memory but increases lookup time.

When would you use a ConcurrentHashMap over a Hashtable or synchronized map?

ConcurrentHashMap is designed for high concurrency. Unlike *Hashtable* which locks the entire map for every operation, *ConcurrentHashMap* uses lock striping (or CAS operations in Java 8+). Reads are mostly lock-free, and writes lock only specific segments or bins. This gives much better throughput in multi-threaded applications. Avoid *Hashtable* due to its global lock; use *ConcurrentHashMap* for thread-safe, high-performance scenarios.

Explain the difference between fail-fast and fail-safe iterators.

Fail-fast iterators (e.g., those in *ArrayList*, *HashMap*) throw *ConcurrentModificationException* if the collection is structurally modified after the iterator is created (except via the iterator's own *remove* method). They operate on the original collection. **Fail-safe iterators** (e.g., those in *ConcurrentHashMap*, *CopyOnWriteArrayList*) work on a snapshot of the collection, so they do not throw exceptions and allow concurrent modifications. However, they may not reflect the latest changes.

MULTITHREADING AND CONCURRENCY

Three years of experience often includes building multi-threaded components. Be prepared to discuss both theoretical and practical concurrency issues.

What is the difference between *synchronized* and *ReentrantLock*?

Both provide mutual exclusion, but *ReentrantLock* offers additional features: fairness policy, tryLock with timeout, ability to interrupt waiting threads, and explicit lock/unlock calls. Synchronized is simpler and built into the language syntax, but it cannot be interrupted or have fairness. *ReentrantLock* also supports multiple condition objects via *newCondition()*, whereas *synchronized* uses *wait* and *notify* on the same intrinsic lock. In modern Java, *synchronized* is optimized enough that it is often preferred unless you need advanced features.

Explain the concept of deadlock and how to prevent it.

A deadlock occurs when two or more threads are blocked forever, each waiting for a resource held by the other. Classic example: Thread1 locks resource A and waits for B; Thread2 locks B and waits for A. Prevention techniques include:

- **Lock ordering:** Always acquire locks in a fixed global order.
- **Lock timeout:** Use *tryLock* with a timeout and release all locks if acquisition fails.
- **Deadlock detection:** Use tools like jstack or ThreadMXBean to detect and recover.
- **Avoid nested locks** when possible.

What is the Java Memory Model (JMM) and why does it matter?

The JMM defines the rules for how threads interact through memory. It specifies when a write to a variable by one thread becomes visible to another thread. Key concepts: *happens-before* relationships, volatile, synchronized, and final fields. Without proper synchronization, you may encounter visibility issues (one thread sees stale data) and reordering issues. For example, using *volatile* ensures that reads and writes are directly from main memory, establishing a happens-before relationship.

EXCEPTION HANDLING – BEYOND TRY-CATCH

Checked vs Unchecked Exceptions – When to use each?

Checked exceptions (extending *Exception*, not *RuntimeException*) must be declared or caught. They represent recoverable conditions (e.g., *FileNotFoundException*). **Unchecked exceptions** (Extending *RuntimeException*) indicate programming errors (e.g., *NullPointerException*, *ArrayIndexOutOfBounds*). Best practice: Use checked exceptions for conditions that a caller can reasonably be expected to handle; use unchecked for faults that cannot be recovered from at runtime. Overuse of checked exceptions can clutter code.

Explain try-with-resources and how it works internally.

Try-with-resources (introduced in Java 7) automatically closes resources that implement *AutoCloseable*. The resource is declared in parentheses after *try*. The compiler generates a finally block that calls *close()* in the correct order, and any exceptions from close are suppressed if the try block also throws an exception. This reduces boilerplate and avoids resource leaks.

JAVA 8 FEATURES THAT MATTER

Functional programming is a significant part of modern Java. For a 3-year experienced developer, you should demonstrate comfortable use of lambdas and streams.

How would you use streams to process a list of objects?

Example: Given a list of *Employee* objects, find the top 3 highest paid employees from the sales department. Using streams:

```
List<Employee> topSales = employees.stream()
    .filter(e -> "Sales".equals(e.getDepartment()))
    .sorted(Comparator.comparingDouble(Employee::getSalary).reversed())
    .limit(3)
    .collect(Collectors.toList());
```

You should be able to explain intermediate operations (filter, sorted, limit) and terminal operations (collect). Also discuss parallel streams and when to avoid them (small datasets, shared mutable state).

What is the difference between map and flatMap?

map applies a function to each element and returns a stream of the results. **flatMap** is used when each element is itself a stream; it flattens the nested streams into a single stream. For example, given a list of sentences, *map(s -> s.split(" "))* gives a stream of arrays; *flatMap(s -> Arrays.stream(s.split(" ")))* gives a stream of words.

Explain Optional and how it helps avoid NullPointerException.

Optional is a container that may or may not contain a value. It encourages explicit handling of absent values instead of relying on null checks. For example, instead of *String name = getPerson().getName();*, you can use *Optional.ofNullable(getPerson()).map(Person::getName).orElse("Default");*. However, be careful not to use Optional for fields or method parameters (it is intended as a return type).

JVM INTERNALS AND MEMORY MANAGEMENT

Describe the memory structure of the JVM: Heap, Stack, Metaspace.

The **Heap** is where all objects and arrays are stored. It is divided into generations: Young Generation (Eden, Survivor S0, S1) and Old Generation. The **Stack** is per-thread and stores primitive values, object references, and method frames. The **Metaspace** (replaced PermGen in Java 8) stores class metadata, static variables, and method bytecode. Understanding garbage collection algorithms like G1GC, and tuning heap sizes, is often asked.

How does Garbage Collection work? What is the concept of stop-the-world?

Garbage collection automatically reclaims memory by removing objects that are no longer reachable. Different collectors have different pause behaviors. **Stop-the-world** means all application threads are paused while GC runs. Minor GC (young generation) pauses are usually short. Full GC (old generation) can cause longer pauses. For low-latency applications, choose collectors like *G1GC* or *ZGC*. Know the difference between serial, parallel, CMS, and G1 collectors.

COMMON DESIGN PATTERNS AND THEIR USE IN JAVA

When would you use the Singleton pattern in a 3-year experienced role? What are the thread-safe implementations?

Singleton ensures a class