Junior Level Java Interview Questions

1. **Name some characteristics of Object-Oriented Programming languages.**

Answer - *Encapsulation* – Wrapping data and methods into a single unit (class).

Abstraction – Hiding implementation details and exposing only relevant functionality.

Inheritance – Allowing a class to inherit properties and methods from another class.

Polymorphism – Enabling a single interface to represent different underlying forms.

2. **What are the access modifiers you know? What does each one do?**

Answer - *Private* – Accessible only within the same class.

Default (package-private) – Accessible within the same package.

Protected – Accessible within the same package and subclasses.

Public – Accessible from anywhere.

3. **What is the difference between overriding and overloading a method in Java?**

Answer - *Method Overriding* – Redefining a method in a subclass while maintaining the same signature.

Method Overloading – Defining multiple methods with the same name but different parameters.

4. **What's the difference between an Interface and an Abstract class?**

Answer - *Interface* – Defines a contract with abstract methods (default methods in Java 8+).

Abstract Class – Can have both abstract and concrete methods, as well as instance variables.

5. **Can an Interface extend another Interface?**

Answer - Yes, an interface can extend multiple other interfaces.

Interface A { void methodA(); }

Interface B extends A { void methodB(); }

6. **What does the static keyword mean in Java?**

Answer - Declares methods or variables that belong to the class rather than an instance.

Static members are shared across all instances of the class.

7. **Can a static method be overridden in Java?**

Answer - No, static methods are not overridden; they are hidden in the subclass if redefined.

8. **What is Polymorphism? What about Inheritance?**

Answer - *Polymorphism* – Allows a single interface to represent multiple types (method overriding and method overloading).

Inheritance – Enables a class to inherit fields and methods from another class.

9. **Can a constructor be inherited?**

Answer - No, constructors are not inherited, but a subclass constructor can invoke a superclass constructor using super().

10. **Do objects get passed by reference or value in Java?**

Answer - Java passes objects by value, meaning a copy of the reference is passed. Changes affect the original object, but reassigning the reference does not.

11. **What's the difference between using == and .equals() on a string?**

Answer- == checks reference equality.

.equals() checks content equality.

12. **What are hashCode() and equals() used for?**

Answer - They define how objects are compared in collections like HashMap and HashSet. Objects that are equal (equals()) must have the same hash code (hashCode()).

13. **What does the Serializable interface do? What about Parcelable in Android?**

Answer - *Serializable* – Used to convert objects into byte streams for storage or transfer.

Parcelable – Optimized for Android inter-process communication (IPC), faster than Serializable.

14. **Why are Array and ArrayList different? When would you use each?**

Answer - *Array* – Fixed size, faster, stores primitive and objects.

ArrayList – Dynamic size, easier to manipulate, only stores objects.

15. **What's the difference between an Integer and int?**

Answer - Int is a primitive type.

Integer is a wrapper class providing utility methods.

16. **What is a ThreadPool? Is it better than using several "simple" threads?**

Answer - A ThreadPool manages multiple worker threads efficiently, reducing overhead compared to creating new threads for every task.

17. **What's the difference between local, instance, and class variables?**

Answer -

Local – Declared inside a method, accessible only within that method.

Instance – Belongs to an object, defined at class level.

Class (static) – Shared across all instances of a class.

No

Mid-Level Java Interview Questions

1. **What is reflection?**

Answer: Reflection allows Java programs to inspect and manipulate classes, methods, and fields at runtime, even if their names are unknown at compile time. It's commonly used for:

Dependency injection frameworks (e.g., Spring uses reflection to inject dependencies without needing explicit object creation).

ORMs like Hibernate (which map Java objects to database tables dynamically).

Testing frameworks (Junit uses reflection to invoke test methods).

However, reflection is slower than direct method calls and violates encapsulation, so it should be used cautiously.

2. **What is dependency injection? Can you name a few libraries?**

Answer: Dependency Injection (DI) is a design pattern where object dependencies are provided externally rather than created within the class. This promotes loose coupling and easier testing.

Spring (Spring Boot): Uses @Autowired, constructor injection, and setter injection.

Google Guice: Lightweight DI framework.

Dagger (for Android): Compile-time DI for efficiency.

In my experience, constructor injection is preferable over field injection because it makes dependencies explicit and allows easier unit testing.

3. **What are strong, soft, and weak references in Java?**

Answer: *Strong Reference (default)* → Prevents object from being garbage collected.

Soft Reference → Eligible for GC only when JVM needs memory, useful for caching (e.g., SoftReference<T> in LinkedHashMap).

Weak Reference → Eligible for GC as soon as it's unreachable. Used in WeakHashMap (commonly for caching where memory must be freed aggressively).

Real-world example: Java caches class loaders using WeakReference to avoid memory leaks when unloading classes dynamically.

4. **What does the synchronized keyword mean?**

Answer: The synchronized keyword prevents race conditions by ensuring only one thread can access a block of code at a time. However, it has performance overhead due to blocking.

Better alternatives:

ReentrantLock: Provides better flexibility (e.g., tryLock(), fair locking).

Atomic variables (AtomicInteger): Useful for counters without full synchronization.

Concurrent collections (ConcurrentHashMap): Avoid full object locking.

Example: In a banking system, if multiple users withdraw money from an account, synchronization ensures correct balance updates. However, for high-performance apps, we prefer ReentrantLock or AtomicLong.

5. **Can you have memory leaks in Java?**

Answer: Yes, even though Java has garbage collection, memory leaks can still happen:

Unclosed resources (e.g., database connections, input streams).

Static references (objects referenced by static fields never get collected).

Listeners/Callbacks (anonymous inner classes holding strong references to outer class).

ThreadLocal variables (objects may persist if thread pool is used).

Solution: Always use try-with-resources for AutoCloseable objects and call remove() on ThreadLocal after use.

6. What does it mean that a String is immutable?

Answer: In Java, String objects cannot be changed after creation. This makes them:

Thread-safe (no synchronization needed).

Optimized for caching (string literals stored in the String Pool).

Efficient in HashMaps (immutable keys don’t change, preventing lookup failures).

However, modifying a String creates new objects, causing performance overhead.
Better alternative: Use StringBuilder or StringBuffer for frequent modifications.

Example:

```
String s = “Java”;

S += “ Interview”; // Creates a new object instead of modifying ‘s’
```

7. How does try {} finally {} work?

Answer: The finally block executes always, even if an exception is thrown (except in System.exit()). It’s commonly used for resource cleanup.

Better approach: Use try-with-resources instead of finally to close resources automatically:

```
Try (FileReader fr = new FileReader(“file.txt”)) {

   // File reading logic

} // FileReader is closed automatically
```

Try-with-resources is preferred over finally because it prevents accidental resource leaks.

8. Why are Generics used in Java?

Answer: Generics provide type safety and code reusability by enforcing compile-time type checks.

Example:

List<String> list = new ArrayList<>();

List.add("Java"); // Prevents adding non-string values

Without generics, Java collections used raw types, leading to ClassCastException at runtime.

In real projects, generics are useful in APIs, dependency injection, and collections (e.g., Map<K, V> in caching frameworks like Ehcache).

9. **Can you mention the design patterns you know? Which do you use?**

Answer:

Singleton → Ensures a class has only one instance (e.g., database connection).

Factory → Creates objects dynamically (used in Calendar.getInstance()).

Observer → Implements event-driven architecture (e.g., Listeners in GUI apps).

Builder → Simplifies complex object creation (StringBuilder).

Example: In my last project, I used Factory Pattern to create different payment gateways dynamically instead of if-else conditions.

10. **Can you mention some types of testing you know?**

Answer: *Unit Testing (Junit, Mockito)* → Tests single classes/methods.

Integration Testing → Tests communication between modules (TestContainers for DB testing).

Performance Testing (JMeter, Gatling) → Ensures scalability.

End-to-End (E2E) Testing (Selenium, Cypress) → Simulates user interactions.

Best practice: Write unit tests for all business logic and use Mockito for dependency mocking to isolate test cases.

Senior-Level Java Interview Answers

1. **How does Integer.parseInt() work?**

Answer: Integer.parseInt(String s) converts a numeric string into an integer. Internally, it:

1. Checks for null and validates characters (only digits, optional + or – at the start).
2. Converts characters to numbers using s.charAt(i) – ‘0’ (ASCII subtraction).
3. Multiplies the previous result by 10 to shift digits left and adds the new digit.

4. **Handles overflow by comparing against Integer.MAX_VALUE / 10.**

Answer - Performance Consideration: If parsing a large amount of data, Integer.parseInt() is faster than using regex or BigInteger.

Example:

System.out.println(Integer.parseInt(“1234”)); // Output: 1234

Optimization Tip: If parsing millions of numbers, consider parallel processing or primitive-based approaches like DataInputStream.readInt() for better performance.

2. **What is the “double-checked locking” problem?**

Answer: Double-checked locking is a technique to implement lazy initialization in a thread-safe manner. However, it was broken before Java 5 due to compiler reordering issues.

Incorrect Implementation (Before Java 5, not safe):

```
Public class Singleton {
   Private static Singleton instance;
   Public static Singleton getInstance() {
      If (instance == null) {  // First check
         Synchronized (Singleton.class) {
            If (instance == null) {  // Second check
               Instance = new Singleton();
            }
        }
    }
    Return instance;
  }
}
```

Why is it broken?

The JVM might reorder instructions, leading to a partially constructed object being accessed

Correct Approach (Java 5+ with volatile):

```
Public class Singleton {
  Private static volatile Singleton instance;
  Public static Singleton getInstance() {
    If (instance == null) {
      Synchronized (Singleton.class) {
        If (instance == null) {
          Instance = new Singleton();
        }
      }
    }
    Return instance;
  }
}
```

Real-world Use Case: Used in caching mechanisms like database connection pooling or configuration loading.

3. **Difference between StringBuffer and StringBuilder?**

Answer:

Feature	StringBuffer	StringBuilder
Thread-safe?	Yes (synchronized)	No
Performance	Slower (locks required)	Faster
Usage	Multi-threaded apps	Single-threaded apps

Example:

```
StringBuilder sb = new StringBuilder(“Java”);
Sb.append(“ Interview”);
```

When to Use?

Single-threaded → Use StringBuilder for better performance.

Multi-threaded (rare cases) → Use StringBuffer, but prefer concurrent utilities like ConcurrentLinkedQueue.

4. **How does Class.forName() work?**

Answer: Loads a class dynamically at runtime (used in frameworks, reflection-based libraries).

If the class is not found, it throws ClassNotFoundException.

If used with “Class.forName(name, true, loader)”, it also initializes the class.

Example:

```
Class<?> cls = Class.forName(“com.example.MyClass”);
Object obj = cls.getDeclaredConstructor().newInstance(); // Creates an object dynamically
```

Real-World Use Cases:

JDBC Drivers: Class.forName(“com.mysql.cj.jdbc.Driver”).

Spring Framework: Loads beans dynamically.

Optimization Tip: If frequently loading classes, use caching to avoid repeated lookups.

5. **What is Autoboxing and Unboxing?**

Answer: *Autoboxing*: Converting a primitive to its wrapper (int → Integer).

Unboxing: Converting a wrapper to primitive (Integer → int).

Example:

```
Integer obj = 10; // Autoboxing

Int num = obj;    // Unboxing
```

Performance Concern:

Autoboxing creates new objects, leading to higher memory usage.

== compares references in wrappers (Integer I = 128; Integer j = 128; I == j is false).

Use int for performance-sensitive tasks (e.g., loops, calculations).

6. **What is fail-fast vs. fail-safe in Java?**

Answer:

Feature	Fail-Fast	Fail-Safe
Behavior	Throws ConcurrentModificationException	Works even if modified concurrently
Example	ArrayList, HashMap	ConcurrentHashMap, CopyOnWriteArrayList
Use Case	Single-threaded	Multi-threaded

Example:

```
List<Integer> list = new ArrayList<>(List.of(1, 2, 3));

For (Integer num : list) {

   List.add(4); // Throws ConcurrentModificationException

}
```

Best Practice: For high-performance applications, use concurrent collections instead of locking mechanisms.

7. **What is the Java Heap?**

Answer: Java Heap is the memory where all objects are stored.

It is divided into:

Young Generation (Eden, Survivor) → Short-lived objects (garbage collected frequently).

Old Generation (Tenured) → Long-lived objects.

Metaspace → Stores class metadata.

Tuning Heap Size (JVM Options):

```
Java -Xms512m -Xmx4g MyApp
```

Performance Tip: For applications with high object churn, tune GC settings (G1GC, ZGC) for better throughput.

8. **What is a daemon thread?**

Answer: Daemon threads run in the background (e.g., garbage collector, monitoring threads).

They do not prevent the JVM from exiting.

Example:

```
Thread daemonThread = new Thread(() -> {
   While (true) {
      System.out.println("Daemon running…");
   }
});
daemonThread.setDaemon(true);
daemonThread.start();
```

If the main thread terminates, the daemon stops automatically.

Real-world Example: Background tasks in web servers, Kafka consumers, log processors.

9. **Can a dead thread be restarted?**

Answer: No, once a thread completes or is terminated, it cannot be restarted.

Example:

Thread t = new Thread(() -> System.out.println(“Running…”));

t.start();

t.join();

t.start(); // Throws IllegalThreadStateException

Solution: Instead of restarting threads, use:

Thread pools (ExecutorService) for reusing threads.

Message queues (Kafka, RabbitMQ) to handle tasks asynchronously.