JAVA

INTERVIEW CONCEPTS

25

MUST-KNOW TOPICS

FOR BACKEND ENGINEERS

1. volatile vs synchronized

`volatile` ensures visibility across threads any update to a volatile variable is immediately visible to other threads. However, it does not guarantee atomicity.

`synchronized` provides both visibility and atomicity by locking a method or block so only one thread can execute it at a time.

Use `volatile` for flags or simple variable updates, and `synchronized` for compound actions like incrementing a counter or read-modify-write operations.

2. equals() vs ==

`==` compares references whether two variables point to the same object in memory.

`equals()` checks for logical equality (i.e., whether the objects have the same content).

Override `equals()` and `hashCode()` together when creating value-based objects like entities or DTOs.

3. HashMap Internals

HashMap uses an array of buckets. Each bucket stores entries using chaining (linked list or tree after threshold).

Key placement is based on `hashCode()` and equality check via `equals()`.

Important interview areas: collision resolution, resizing, time complexity, and concurrent access behavior.

4. Thread vs Runnable vs Callable

`Thread` represents a thread of execution by extending and overriding `run()`.

`Runnable` is a functional interface used with `Thread` or executors.

`Callable` returns a result and can throw exceptions used with `Future`.

5. Exception Hierarchy

Java exceptions inherit from `Throwable`: Errors (like OutOfMemoryError), and Exceptions.

Checked exceptions must be handled; unchecked (RuntimeExceptions) are optional.

Understand `try-catch-finally`, custom exceptions, and `throw` vs `throws`.

6. JVM Memory Model (Stack vs Heap)

Stack stores method calls and local variables; it's thread-safe and short-lived.

Heap stores objects and class instances shared among threads.

Understanding memory areas is critical for avoiding leaks and optimizing performance.

7. Autoboxing Pitfalls

Java auto-converts primitives to wrapper types (and vice versa). This can introduce bugs and performance issues.

Example: `Integer i = null; int j = i;` will throw NullPointerException.

Autoboxing can also affect object identity checks with `==`.

8. Streams vs Loops

Streams provide a declarative way to process collections using functional-style operations (map, filter, reduce).

Loops are imperative and sometimes more performant.

Streams may introduce overhead and debugging complexity if overused.

9. Immutability vs final vs effectively final

`final` prevents reassignment of variables.

Immutability ensures object state cannot change.

Effectively final means a variable isn't reassigned required in lambdas and anonymous classes.

10. Object Creation Cost vs Caching

Creating many short-lived objects increases GC activity.

Use object pooling or caching for frequently used values (e.g., Integer cache from -128 to 127).

Immutable objects help avoid GC churn too.

11. Static vs Instance Methods

Static methods belong to the class and cant access instance variables unless passed as parameters.

Instance methods operate on object state.

Use static for utility behavior, instance for logic tied to object data.

12. Transient Keyword

Fields marked `transient` are skipped during serialization.

Useful for sensitive info like passwords or non-serializable objects in a Serializable class.

13. CompletableFuture vs Future

Future represents the result of an asynchronous computation, but its limited:

- Blocks with `get()`
- No chaining
- No built-in exception handling

CompletableFuture improves async programming by:

- Supporting non-blocking chaining via `thenApply`, `thenCombine`, etc.
- Handling exceptions with `exceptionally` or `handle`
- Running tasks asynchronously with `supplyAsync`, `runAsync`

Use CompletableFuture for building efficient concurrent flows in microservices, file I/O, and non-blocking DB calls.

14. Enum vs Constant Class

Enums are type-safe, can hold behavior, and prevent invalid values.

Better than public static final constants especially in switch-case and domain modeling.

15. String Pooling

Java stores string literals in a pool to save memory.

Use `intern()` to force string pooling.

Avoid `new String("...")` unless needed explicitly.

16. Functional Interfaces

An interface with a single abstract method.

Used in lambda expressions, method references, and Streams API.

Examples: Runnable, Callable, Comparator, Function.

17. Serialization vs Deserialization

Serialization = converting an object into a byte stream.

Deserialization = reconstructing an object from that stream.

Use `Serializable` interface and handle versioning with `serialVersionUID`.

18. Polymorphism

Object behavior depends on the runtime type, not reference type.

Enables method overriding and flexible APIs.

Key to designing extensible systems.

19. Abstraction vs Encapsulation

Abstraction hides implementation via interfaces or abstract classes.

Encapsulation binds data and logic, and restricts access using access modifiers (private, protected).

20. Java Collections Hierarchy

Root interfaces: List, Set, Queue, Map.

Implementations: ArrayList, LinkedList, HashSet, TreeMap, PriorityQueue, etc.

Understand time complexity and usage trade-offs.

21. WeakHashMap vs HashMap

WeakHashMap keys are weakly referenced they get GCd when no strong reference exists.

Great for caches.

HashMap keeps keys strongly referenced.

22. LinkedList vs ArrayList

ArrayList offers fast random access, but costly insertions/removals in the middle.

LinkedList is faster for frequent adds/removes but slower in access due to traversal.

23. try-with-resources

Introduced in Java 7 ensures resources implementing AutoCloseable are closed automatically.

Cleaner than finally blocks for I/O, DB connections, etc.

24. ReentrantLock vs synchronized

ReentrantLock allows more control: tryLock(), lockInterruptibly(), fairness, and multiple conditions.

Use when flexibility beyond synchronized is needed.

25. Method Overloading vs Overriding

Overloading = multiple methods with same name but different parameters (compile-time).

Overriding = redefining superclass method in subclass (runtime).

Key for polymorphism and behavior extension.