JAVA INTERVIEW CHEAT SHEET

Q&A for Technical Interviews

Popular questions

☕ PLATFORM INDEPENDENCE

Q: Is the Java compiler (javac) platform-dependent? If so, how does Java achieve platform independence with .class files?

The Java compiler (javac) itself is **platform-dependent**, meaning that the binary you run on Windows is different from the one you run on Linux or macOS, since it has to be compiled for that operating system.

However, the output of the compiler — the **.class file (Java bytecode)** — is **platform-independent**. Bytecode follows the Java Virtual Machine (JVM) specification, so the same .class file can run on any platform.

When you execute a .class file, the JVM of the target platform (Windows JVM, Linux JVM, etc.) interprets or JIT-compiles the bytecode into native machine code specific to that operating system and hardware.

In short:

- The **javac compiler binary** is platform-specific
- The **.class bytecode** it produces is platform-independent
- The **JVM** makes sure that the same bytecode runs on any platform by translating it into platform-specific instructions

🔄 CONCURRENCY

Q1: What is the difference between volatile and synchronized?

Volatile — Let's first understand the problem without volatile. When multiple threads share a variable, each thread has its own local CPU cache/working memory. When a thread writes to a variable, it first updates its local cache. Later (not immediately), it may flush the value to main memory. Another thread reading that variable may still see the old value from its own cache, leading to eventual consistency (delay in visibility).

Problem without volatile:

```
private static boolean running = true; // not volatile Main Thread: sets running = false (writes to its cache, flush later) Worker Thread: keeps reading running from its own cache (still true) Result → Worker thread may never stop, because it doesn't see the update
```

This is because **visibility is not guaranteed** across threads without synchronization.

Solution with volatile:

When we declare the variable as **volatile**, the Java Memory Model enforces:

- Every write goes directly to main memory
- Every read comes directly from main memory
- No caching of stale values is allowed

```
private static volatile boolean running = true; Main Thread: writes running = false → immediately flushed to main memory Worker Thread: always reads from main memory, sees the updated value Result → Worker thread stops correctly
```

Synchronized — when multiple threads update the same variable then race condition can occur

Problem: When two threads interleave, updates can be lost:

> Thread t1 reads counter = 0 Thread t2 also reads counter = 0 t1 increments → 1 t2 increments → also 1 (overwrites t1's update) ⚠️ Instead of 2, the counter is still 1. This is a lost update problem.

At first glance, you may think the result will always be 2000. But in reality, it is often less than 2000. Why? Because `counter++` is not an atomic operation. It's actually three steps:

1. Read the value of counter from memory
2. Increment the value
3. Write the new value back to memory

Solution with synchronized:

```
public synchronized void increment() { count++; } public synchronized int getCount() { return count; }
```

The `synchronized` keyword puts a lock on the object (counter).

When one thread enters the increment() method, the other thread must wait.

This ensures:

1. **Atomicity** → no interleaving of counter++
2. **Visibility** → updates are flushed to main memory

✅ Now the final result is always 2000.

💾 JAVA MEMORY MODEL

Q2: Explain Java Memory Model and happens-before relationship

The Java Memory Model defines:

- How threads interact through memory (variables, caches, main memory)
- What reads/writes are visible to other threads and when
- The rules for reordering instructions (by compiler or CPU) while still preserving correctness

Key concepts of JMM:

Each thread can cache variables in CPU registers or local memory. Updates to shared variables are not guaranteed to be immediately visible to other threads. Compiler/JVM/CPU may reorder instructions for optimization. `synchronized`, `volatile`, `final`, and java.util.concurrent classes (locks, atomics) establish rules so that threads see consistent values.

Happens-Before Relationship:

The happens-before relationship is the foundation of the JMM. It defines when one action's result is guaranteed to be visible to another action.

If A happens-before B:

- The result of action A is visible to action B
- Action A is ordered before action B (no reordering)

Happens-Before Rules:

1. Program order rule

Within a single thread, statements happen in the order they are written.

```
int a = 5; // happens-before int b = a+1; // this
```

2. Monitor lock rule

An unlock on a monitor (synchronized block exit) happens-before every subsequent lock on the same monitor.

```
synchronized(lock) { shared = 42; } // unlock happens-before synchronized(lock) { System.out.println(shared); }
// lock
```

3. Volatile Variable rule

A write to a volatile variable happens-before every subsequent read of that same variable.

```
volatile boolean flag = false; flag = true; // write happens-before if(flag) { ... } // read
```

4. Thread start rule

A call to Thread.start() on a thread happens-before any actions inside that thread.

```
Thread t = new Thread(() → { /* actions */ }); t.start(); // happens-before thread actions
```

5. Thread Termination Rule

All actions in a thread happen-before another thread detects its termination (via Thread.join() or Thread.isAlive()).

```
t.start(); t.join(); // happens-after t's actions
```

🎯 OPTIONAL IN JAVA 8

Q3: When do you use Optional in Java 8? What are the anti-patterns?

The main purpose of **Optional<T>** is to represent the possible absence of a value without using null. It makes code more readable and avoids NPE.

For method **getUserByName(String name)**, we are returning an **Optional<User>** which means it might return a user or empty result. Seeing the Optional return type from a method, the caller of the method is forced to handle it i.e. it documents the possibility of absence better than just returning null. Caller can use **ifPresentOrElse** method to handle both cases.

```
public Optional<User> getUserByName(String name) { if ("Anil".equals(name)) { return Optional.of(new User("Anil",
"anil@email.com", "123")); } else { return Optional.empty(); } } Optional<User> userOpt =
demo.getUserByName("Anil"); userOpt.ifPresentOrElse( user → System.out.println("User found: " + user.getName()),
() → System.out.println("User not found") );
```

Anti-patterns with Optional:

- ❌ Do not use optional in method argument, fields and constructor
- ❌ Do not directly use Optional.get() (may cause NoSuchElementException)
- ❌ Do not allow a getter method to return optional

🔗 EQUALS & HASHCODE

Q4: How does equals() and hashCode() contract affect collections like HashMap?

When working with Java collections like HashMap, HashSet, or Hashtable, two methods play a critical role in determining how objects are stored and retrieved: **equals()** and **hashCode()**.

The Contract Between equals() and hashCode():

- If two objects are equal according to equals(), they must have the same hashCode()
- If two objects have the same hashCode(), they may or may not be equal
- Different objects can collide in the same bucket

How HashMap Uses equals() and hashCode():

When you put an entry into a HashMap:

1. The **hashCode()** of the key is calculated → determines the bucket
2. If multiple keys map to the same bucket, **equals()** is used to resolve conflicts

```
Map<String, String> map = new HashMap<>(); map.put("apple", "fruit"); // Lookup process: String key = "apple"; int hash = key.hashCode(); // Step 1: bucket // Step 2: equals() check With existing keys in that bucket System.out.println(map.get("apple")); // "fruit"
```

The Danger of Mutable Keys:

A common pitfall is using a mutable object as a key.

```
class MutableKey { int id; MutableKey(int id) { this.id = id; } @Override public boolean equals(Object o) { return o instanceof MutableKey && this.id == ((MutableKey) o).id; } @Override public int hashCode() { return Objects.hash(id); } } Map<MutableKey, String> map = new HashMap<>(); MutableKey key = new MutableKey(1); map.put(key, "Value"); key.id = 2; // ❌ Mutation after insertion System.out.println(map.get(new MutableKey(1))); // null System.out.println(map.size()); // 1
```

Why does this happen?

- When key.id = 1, the entry is stored in the bucket based on hashCode(1)
- After changing id to 2, the key's hashCode() changes, but the entry remains in the original bucket
- When you look up with a new MutableKey(1), the map searches the correct bucket, but cannot find the mutated key

Common Anti-Patterns:

- ❌ Overriding equals() but not hashCode()
- ❌ Using mutable keys in hash-based collections
- ❌ Using business fields that frequently change inside equals() and hashCode()
- ❌ Returning random or dynamic values from hashCode()

🔍 MEMORY LEAKS

Q5: How would you prevent memory leaks in Java?

1. Event Listeners, Observers, and Callbacks

One of the most common sources of memory leaks in Java applications comes from event listeners, observers, and callbacks that are registered but never removed.

The Button class has a listener set on it. This listener is an anonymous inner class which implicitly holds a reference to **window.this**.

Even if we set **window=null**, the button still holds a reference to the listener, which holds a reference to Window. This prevents garbage collection of Window and causes memory leak.

Solution: Add a cleanup() method in Window class which should deregister listener.

```
void cleanup() { button.setListener(null); // ✅ Deregister listener } // Usage window.cleanup(); System.gc(); // Now Window can be collected
```

Or use static inner class which do not hold reference of outer class.

2. Inner Classes / Anonymous Classes

Non-static inner classes keep an implicit reference to their outer class. If the outer object is large and only the inner is referenced, the whole outer remains in memory.

3. ThreadLocal Misuse

If we don't call **remove()** after use, values may stick around in thread-local storage (especially in long-lived threads like thread pools). This is one of the most common causes in enterprise apps.

```
private static ThreadLocal<byte[]> threadLocal = ThreadLocal.withInitial(() → new byte[1024 * 1024]); // ❌ Problem: data remains in thread's internal map executorService.submit(() → { threadLocal.set(new byte[10 * 1024 * 1024]); System.out.println("Task executed"); // Missing cleanup! }); // ✅ Solution: always call remove() in finally block executorService.submit(() → { try { threadLocal.set(new byte[10 * 1024 * 1024]); System.out.println("Task executed"); } finally { threadLocal.remove(); // ✅ Ensure cleanup } });
```

A thread pool with 1 thread is created. Each task sets a 10MB array in ThreadLocal. Even after tasks finish, the data is still in the thread's internal map because we didn't call remove().

4. Using HashMap for Caching

If we use a normal HashMap for caching, keys and values will never get garbage-collected unless we manually remove them. That means the cache keeps growing → memory leak.

Problem: HashMap holds strong references to keys and values. Even though we set key to null, the map still holds its entry.

```
private static final Map<Object, byte[]> cache = new HashMap<>(); Object key = new Object(); cache.put(key, new byte[10 * 1024 * 1024]); // 10MB per key key = null; // Remove strong reference to key System.gc(); System.out.println("After GC: " + cache.size()); // 1 (entry retained!)
```

Solution: Use `WeakHashMap` which holds weak references to keys.

If a key is no longer referenced elsewhere (e.g., you set key = null or it goes out of scope), the entry is automatically eligible for garbage collection. The whole entry (key + value) is removed from the map.

```
private static final Map<Object, byte[]> cache = new WeakHashMap<>(); Object key = new Object(); cache.put(key, new byte[10 * 1024 * 1024]); // 10MB per key System.out.println("Before GC: " + cache.size()); // 1 key = null; // Remove strong reference to key System.gc(); Thread.sleep(1000); System.out.println("After GC: " + cache.size()); // 0 (entry cleared!)
```

By default, values in WeakHashMap are strong references. But you can make them weak by wrapping them in `WeakReference<T>`.

```
WeakHashMap<Object, WeakReference<String>> map = new WeakHashMap<>(); Object key = new Object(); String value = new String("Hello"); // force heap object map.put(key, new WeakReference<>(value)); System.out.println("Before null: " + map.get(key).get()); // Hello value = null; // drop strong reference to value System.gc(); Thread.sleep(1000); System.out.println("After GC: " + map.get(key)); // still has WeakReference System.out.println("After GC deref: " + map.get(key).get()); // null
```

The key is strongly referenced (still alive in main()), so the entry stays. But the value was weakly referenced, and since we set value = null, GC collects it. The map still holds the key, but the value inside the WeakReference is now null.

CALLABLE VS RUNNABLE

Q6: What's the difference between Callable and Runnable?

Let's discuss the basic difference between Callable and Runnable:

- Runnable has **no return type** (void)
- Callable has a **return type**
- Runnable **cannot throw checked exception** (must catch inside)
- Callable **can throw checked exception** (wrapped in ExecutionException on future.get())

Runnable example:

```
Runnable runnable = () -> { System.out.println("Running in thread: " + Thread.currentThread().getName()); // Must handle checked exceptions inside try { throw new IOException("An error occurred in Runnable"); } catch (IOException e) { throw new RuntimeException(e); } }; Thread thread = new Thread(runnable); thread.start();
```

Here you can see that the checked exception is handled inside the runnable only.

Callable example:

```
ExecutorService executor = Executors.newSingleThreadExecutor(); Future<Integer> submit = executor.submit(() -> { // Can throw checked exceptions if(true) { throw new IOException("An error in Callable"); } System.out.println(Thread.currentThread().getName()); return 42; }); try { System.out.println(submit.get()); } catch (ExecutionException e) { // Exception wrapped here throw new RuntimeException(e); } executor.shutdown();
```

Callable works with **ExecutorService + Future**. When we call **future.get()**, it catches the exception inside ExecutionException block.

Q7: ForkJoinPool in Java

ForkJoinPool is a specialized implementation of ExecutorService in Java 7. It is designed to efficiently execute **divide-and-conquer (recursive) tasks** in parallel.

Workflow:

1. A large task is submitted to the ForkJoinPool
2. The task is recursively split (forked) into smaller subtasks
3. The subtasks are executed in parallel by worker threads
4. The results of subtasks are combined (joined) to produce the final result

```
// Example: Sum numbers from 1 to 20 int[] numbers = new int[20]; for (int i = 0; i < numbers.length; i++) { numbers[i] = i + 1; } ForkJoinPool pool = new ForkJoinPool(); SumTask task = new SumTask(numbers, 0, numbers.length); Long result = pool.invoke(task); System.out.println("Sum of numbers: " + result);
```

```
public class SumTask extends RecursiveTask<Long> { private static final int THRESHOLD = 5; private int start, end; private int[] numbers; @Override protected Long compute() { if (end - start ⩽ THRESHOLD) { long sum = 0; for (int i = start; i < end; i++) { sum += numbers[i]; } return sum; } else { int mid = (start + end) / 2; SumTask left = new SumTask(numbers, start, mid); left.fork(); SumTask right = new SumTask(numbers, mid, end); long rightResult = right.compute(); long leftResult = left.join(); return leftResult + rightResult; } } }
```

Key Concepts:

- **invoke()** is a blocking call that waits until computation is finished
- **compute()** defines the divide-and-conquer logic
- **fork()** pushes task onto worker's deque
- **join()** waits for the result

Worker Deque Mechanism:

Each worker thread maintains its own **deque (double-ended queue)** to store tasks. When a worker thread **forks** a new task, it pushes that task onto the **tail** of its own deque.

The worker will later pop tasks from the tail (**LIFO order**), which makes sense because the most recently forked subtask is likely to be cached and executed quickly.

If a worker's deque becomes empty, it doesn't sit idle. Instead, it tries to **steal tasks from other workers' deques**. When stealing, it takes tasks from the **head** (**FIFO order**) to reduce contention — owner thread is working from the tail and thieves steal from the head, so rare chance of fighting for the same task.

DEADLOCK, LIVELOCK, STARVATION

Q8: Explain deadlock, livelock, and starvation

Deadlock — Deadlock happens when two or more threads are blocked forever, each waiting for a resource that another thread holds.

- Thread-1 holds lockA and waits for lockB
- Thread-2 holds lockB and waits for lockA
- Neither can proceed and both are stuck forever

Analogy: Two people meet in a narrow hallway. Each one insists the other move first. Both refuse → nobody moves.

```
Thread t1 = new Thread(() -> { synchronized (LOCK1) { System.out.println("Thread-1 acquired LOCK1");
Thread.sleep(100); synchronized (LOCK2) { System.out.println("Thread-1 acquired LOCK2"); } } }); Thread t2 = new
Thread(() -> { synchronized (LOCK2) { System.out.println("Thread-2 acquired LOCK2"); Thread.sleep(100);
synchronized (LOCK1) { System.out.println("Thread-2 acquired LOCK1"); } } });
```

Livelock — In livelock, threads are not blocked (they are active) but they keep responding to each other in a way that prevents progress.

Example: Two people meet in a narrow hallway. Both step aside to the left at the same time → still blocked. Both step aside to the right at the same time → still blocked. They keep moving but never make progress.

```
while (true) { if (res1.tryLock()) { System.out.println("Thread-1 locked Resource1"); if (!res2.tryLock()) {
System.out.println("Thread-1 releasing Resource1 to retry..."); res1.unlock(); Thread.sleep(50); } else {
System.out.println("Thread-1 locked Resource2"); break; } } }
```

Starvation — A thread suffers starvation when it waits indefinitely because higher-priority threads keep getting scheduled and it never gets CPU time or access to the resource.

In a priority-based scheduling system, if one low-priority thread needs a lock, but higher-priority threads keep preempting it, the low-priority thread may never acquire the lock.

Analogy: In a buffet, people keep cutting in line, and one person at the end never gets food.

```
Thread low = new Thread(lowPriorityTask); Thread high1 = new Thread(highPriorityTask); Thread high2 = new
Thread(highPriorityTask); low.setPriority(Thread.MIN_PRIORITY); // Priority = 1
high1.setPriority(Thread.MAX_PRIORITY); // Priority = 10 high2.setPriority(Thread.MAX_PRIORITY); // Priority = 10
```

THREAD-SAFE CACHE DESIGN

Q9: How would you design a thread-safe cache?

Caching is a critical concept in software development for improving performance by storing frequently accessed data in memory. In multi-threaded environments, designing a thread-safe cache is essential to avoid data inconsistencies, race conditions, and performance bottlenecks.

Problems without thread-safety:

- Two threads could try to write the same key simultaneously (race condition)
- A read operation could see inconsistent data if a write is halfway through
- Without eviction policies, a cache can grow unbounded

Key Features of Our Cache:

- **LRU Eviction** (Least Recently Used) — Removes the least recently accessed entry when cache exceeds capacity
- **TTL** (Time-to-Live) — Automatically expires entries after a set duration
- **Thread-Safe Access** — Multiple threads can safely read and write
- **Lazy Loading** — If a key is missing, the cache automatically computes or fetches the value

```
public class ThreadSafeCache<K, V> { private final long ttl; private final int capacity; private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock(); private final Map<K, CacheEntry<V>> cache; public ThreadSafeCache(long ttl, int capacity) { this.ttl = ttl; this.capacity = capacity; this.cache = new LinkedHashMap<>(capacity, 0.75f, true); } public V get(K key, Callable<V> loader) throws Exception { lock.readLock().lock(); try { CacheEntry<V> entry = cache.get(key); if (entry ≠ null && (System.currentTimeMillis() - entry.timestamp) < ttl) { return entry.value; } } finally { lock.readLock().unlock(); } V value = loader.call(); put(key, value); return value; } public void put(K key, V value) { lock.writeLock().lock(); try { if (cache.size() ⩾ capacity) { K oldestKey = cache.keySet().iterator().next(); cache.remove(oldestKey); } cache.put(key, new CacheEntry<>(value, System.currentTimeMillis())); } finally { lock.writeLock().unlock(); } } } class CacheEntry<V> { V value; long timestamp; CacheEntry(V value, long timestamp) { this.value = value; this.timestamp = timestamp; } }
```

Rules of ReadWriteLock

Write Lock:

To acquire a write lock:

- No other thread can hold a read or write lock
- If another thread holds a read or write lock → current thread waits
- If the same thread already holds the write lock — It can acquire it again immediately (reentrant)
- Hold count increases and must be matched with same number of unlocks

Read Lock:

- **Multiple threads** can hold read locks at the same time
- A thread can acquire a read lock only if no thread holds the write lock
- If a write lock is held by any thread → read lock acquisition waits

Reentrancy:

- The same thread can re-acquire a read lock or write lock it already holds
- But a thread holding a write lock cannot acquire a read lock using the same ReadWriteLock (to prevent upgrade/downgrade complications)

Usage example:

```
ThreadSafeCache<String, String> cache = new ThreadSafeCache<>(1000, 5); cache.put("key1", "value1"); cache.put("key2", "value2"); String result = cache.get("key3", () -> { System.out.println("Cache miss, loading value..."); return "loaded_value"; }); System.out.println(result); // loaded_value
```

This implementation uses:

- **LinkedHashMap** with **access-order** (true) for LRU
- **ReentrantReadWriteLock** for thread-safe reads and writes
- **TTL-based expiration** on cache entries
- **Lazy loading** with Callable<V> loader

Benefits:

- ✅ Multiple concurrent reads (high throughput)
- ✅ Exclusive writes (data consistency)
- ✅ Automatic eviction (memory management)
- ✅ TTL support (freshness guarantee)

MASTER JAVA INTERVIEWS

This cheat sheet covers:

Platform Independence • Concurrency • Memory Model

Optional • Collections • Memory Leaks

Threading • ForkJoinPool • Cache Design

Want to dive deeper?

Check out our comprehensive courses

suddo.io