

Java Interview Questions and Answers

Intern to senior level

Java interview questions and answers from beginner to senior: JDK vs JRE vs JVM, equals vs ==, String vs StringBuilder, abstract class vs interface, ArrayList vs LinkedList, how HashMap works, exceptions, Java 8 streams, JVM memory and multithreading.

12 questions Read online: <https://www.freedevtemplates.com/interview-questions/java>

1. What is the difference between JDK, JRE and JVM in Java?

Level: intern Java

The JVM runs Java bytecode, the JRE is the JVM plus the standard libraries needed to run Java programs, and the JDK is the JRE plus the development tools such as the compiler (javac).

You write source code in .java files, the compiler javac (part of the JDK) turns it into platform-independent bytecode in .class files, and the JVM on any operating system loads and executes that bytecode. That is how Java achieves "write once, run anywhere".

Since Java 11 Oracle and most vendors no longer ship a separate JRE download. You install a JDK, or build a trimmed runtime with jlink. The JVM itself also contains the class loader, the bytecode verifier, the JIT compiler that turns hot bytecode into native code, and the garbage collector.

Key points

JDK = JRE + development tools

JRE = JVM + class libraries

JVM executes bytecode and is platform specific

2. What is the difference between == and equals() in Java?

Level: intern Java

== compares primitives by value and objects by reference (whether they are the exact same object), while equals() compares the logical content of two objects when the class overrides it.

For String this surprises many people: two strings with the same text created with new are different objects, so == is false even though equals() is true. String literals may be equal under == only because of the string pool, which you should never rely on.

The default equals() inherited from Object behaves like ==. If you override equals() in your own class, you must also override hashCode() so that equal objects have equal hash codes, otherwise they will be lost in a HashMap or HashSet.

Key points

== compares references for objects

equals() compares content if overridden

Override hashCode() whenever you override equals()

```
String a = new String("java");
String b = new String("java");
System.out.println(a == b); // false - different objects
System.out.println(a.equals(b)); // true - same content

Integer x = 127, y = 127;
System.out.println(x == y); // true (cached small values) - do not rely on it
```

3. What is the difference between String, StringBuilder and StringBuffer in Java?

Level: intern Java

String is immutable, StringBuilder is a mutable and fast but not thread-safe buffer, and StringBuffer is a mutable buffer with synchronized methods that is thread-safe but slower.

Because String is immutable, every concatenation creates a new object, so building text in a loop with += is wasteful. Immutability also makes strings safe to share between threads, cache their hash code and use as HashMap keys, and it is what allows the string pool.

Use StringBuilder for building text in a single thread, which is almost always. StringBuffer is a legacy class from Java 1.0 and is rarely needed, since sharing a builder across threads is uncommon and you usually need your own higher-level synchronisation anyway.

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 5; i++) {
    sb.append(i).append(', ');
}
String result = sb.toString(); // "0,1,2,3,4,"
```

4. What is the difference between final, finally and finalize in Java?

Level: intern Java

final is a keyword that prevents reassigning a variable, overriding a method or extending a class; finally is a block that always runs after try/catch for cleanup; finalize() was a method called by the garbage collector and is now deprecated.

final on a variable means it can be assigned once (the object it refers to can still change). On a method it cannot be overridden, and on a class it cannot be subclassed (String is final). finally runs whether or not an exception was thrown, which is why it was used to close resources, but try-with-resources is now the preferred way.

finalize() was unreliable because nobody knows when, or whether, the garbage collector will call it. It has been deprecated for removal since Java 18. Use try-with-resources and AutoCloseable, or Cleaner, instead.

```
try (BufferedReader reader = new BufferedReader(new FileReader("data.txt"))) {
    System.out.println(reader.readLine());
} catch (IOException e) {
    log.error("read failed", e);
} // reader is closed automatically
```

5. What is the difference between an abstract class and an interface in Java?

Level: junior Java

An abstract class can have instance fields, constructors and a mix of implemented and abstract methods but supports only single inheritance, while an interface defines a contract that a class can implement multiple times and, since Java 8, can include default methods.

Choose an abstract class when subclasses share state and base behaviour (for example a BaseRepository with a connection field). Choose an interface when you want to define a capability that unrelated classes can have, such as Comparable or Runnable, and when you want callers to depend on an abstraction.

Interface fields are implicitly public static final. Since Java 8 an interface can have default and static methods, and since Java 9 private methods, which lets library authors add methods without breaking existing implementations. A class can implement many interfaces but extend only one class.

```
interface Payment {
    void pay(double amount);
    default String currency() { return "USD"; }
}

abstract class BasePayment implements Payment {
    protected final String accountId;
    BasePayment(String accountId) { this.accountId = accountId; }
}
```

6. What is the difference between ArrayList and LinkedList in Java?

Level: junior Java

ArrayList is backed by a resizable array with $O(1)$ index access and $O(n)$ insertion in the middle, while LinkedList is a doubly linked list with $O(n)$ index access and $O(1)$ insertion once you hold the node; in practice ArrayList is faster for most workloads.

ArrayList stores elements contiguously, which is cache-friendly, so iteration and `get(i)` are very fast. Adding at the end is amortised $O(1)$; inserting or removing in the middle shifts the following elements.

LinkedList stores each element in a node with previous and next pointers, which costs extra memory and scatters nodes around the heap. Finding the element at position `i` requires walking the list. It only wins when you repeatedly add or remove at the ends or through an iterator. For queue and deque behaviour, ArrayDeque is usually a better choice than LinkedList.

Key points

ArrayList: $O(1)$ get, amortised $O(1)$ add at end

LinkedList: $O(n)$ get, extra memory per element

Default to ArrayList; use ArrayDeque for queues

7. What is the difference between checked and unchecked exceptions in Java?

Level: junior Java

Checked exceptions (subclasses of Exception other than RuntimeException) must be caught or declared with throws, while unchecked exceptions (RuntimeException and Error) do not have to be handled and usually signal programming bugs.

Checked exceptions such as IOException and SQLException represent failures the caller can reasonably be expected to handle or recover from, so the compiler forces you to deal with them. Unchecked exceptions such as NullPointerException, IllegalArgumentException and IndexOutOfBoundsException indicate bugs, and fixing the code is better than catching them. Error (for example OutOfMemoryError) signals serious JVM problems you should not catch.

Good practice: never swallow an exception silently, catch the most specific type, add context when you wrap it, and keep the original as the cause (`new ServiceException("msg", e)`). Many modern frameworks, including Spring, prefer unchecked exceptions because checked ones clutter method signatures.

8. What are the main features of Java 8: lambdas, streams and Optional?

Level: junior Java

Java 8 introduced lambda expressions, functional interfaces, the Stream API for declarative collection processing, Optional to represent missing values, default methods in interfaces, method references and the java.time date and time API.

A lambda is a short way to write an instance of a functional interface (an interface with one abstract method), such as Comparator or Runnable. The Stream API lets you describe a pipeline: a source, zero or more

intermediate operations such as filter and map that are lazy, and a terminal operation such as collect that triggers the work.

Use Optional as a return type to make absence explicit instead of returning null, but not for fields or method parameters. Streams do not modify the source collection, and parallel streams should be used with care: they only help for large, CPU-bound, stateless operations.

Key points

Lambdas implement functional interfaces

Stream intermediate operations are lazy until a terminal operation runs

Optional is for return values, not fields or parameters

```
List<String> names = users.stream()
    .filter(u -> u.isActive())
    .map(User::getName)
    .sorted()
    .collect(Collectors.toList());

Optional<User> user = repository.findById(id);
String email = user.map(User::getEmail).orElse("unknown");
```

9. How does HashMap work internally in Java?

Level: mid Java

A HashMap stores entries in an array of buckets; it computes the key's hash code to choose a bucket, resolves collisions with a linked list that becomes a red-black tree when a bucket grows large, and resizes when the load factor of 0.75 is exceeded.

On put(key, value), HashMap calls key.hashCode(), spreads the bits, and uses hash & (capacity - 1) to pick a bucket index. If the bucket is empty, the entry is stored there. If not, it walks the bucket comparing keys with equals(): a matching key has its value replaced, otherwise the new entry is added to the end. That is why correct hashCode() and equals() implementations are essential for keys.

The default capacity is 16 and the load factor is 0.75, so when more than 12 entries are stored, the table doubles and entries are redistributed. Since Java 8, a bucket with 8 or more entries (when the table has at least 64 buckets) is converted from a linked list to a red-black tree, which keeps worst-case lookup at O(log n) instead of O(n). Average get and put are O(1). HashMap allows one null key, does not guarantee ordering and is not thread-safe.

Key points

Index = hash of key mapped onto the bucket array

Collisions: linked list, treeified at 8 entries

Resizes at load factor 0.75; not thread-safe

10. What is the difference between HashMap, Hashtable and ConcurrentHashMap in Java?

Level: mid Java

HashMap is not thread-safe, Hashtable is thread-safe by locking the whole map on every method, and ConcurrentHashMap is thread-safe with much finer-grained locking and lock-free reads, so it scales far better.

Using a plain HashMap from several threads can lose updates or, in older versions, even loop forever during a resize. Hashtable and Collections.synchronizedMap() fix this by synchronising every operation on a single lock, so threads queue up behind each other.

ConcurrentHashMap locks only the bucket being modified (using CAS and synchronized on the bucket head since Java 8) and allows reads without locking, so many threads can work in parallel. It does not allow null keys

or values. Its compound atomic methods such as `computeIfAbsent`, `merge` and `putIfAbsent` should be used instead of a separate `get` followed by `put`, which is a race condition.

```
ConcurrentHashMap<String, Integer> hits = new ConcurrentHashMap<>();
hits.merge("home", 1, Integer::sum); // atomic increment

// racy even on a ConcurrentHashMap:
// if (!map.containsKey(k)) map.put(k, v);
```

11. How does JVM memory management and garbage collection work in Java?

Level: mid Java

The JVM divides memory into a per-thread stack, a shared heap (split into young and old generations) and Metaspace; the garbage collector frees heap objects that are no longer reachable, collecting short-lived young objects frequently and old objects less often.

Each thread has its own stack holding method frames with local variables and references. All objects live on the heap. New objects are allocated in the young generation (Eden and two survivor spaces); objects that survive several minor collections are promoted to the old generation. Class metadata lives in Metaspace, which replaced PermGen in Java 8.

An object is garbage when it can no longer be reached from a GC root such as a thread stack, static field or JNI reference. G1 is the default collector since Java 9 and balances throughput with pause times; ZGC and Shenandoah target very low pauses on large heaps. Typical production problems are `OutOfMemoryError` from leaks (static collections, listeners, `ThreadLocals` not cleared) and long pauses from undersized or oversized heaps. Diagnose them with GC logs, heap dumps and tools such as VisualVM or Eclipse MAT.

Key points

Stack per thread, heap shared, Metaspace for class data

Generational collection: young then old

Leaks are objects still reachable by mistake

12. What is the difference between synchronized and volatile in Java, and how do you avoid deadlocks?

Level: senior Java

synchronized gives mutual exclusion and memory visibility for a block of code, while volatile only guarantees that reads and writes of one variable are visible to all threads and does not make compound operations such as `count++` atomic.

Without synchronisation a thread may keep a cached copy of a variable and never see another thread's update. `volatile` forces reads and writes to go through main memory (and establishes a happens-before relationship), so it is suitable for simple flags such as `volatile boolean running`. It does not protect `count++`, which is really read, add, write, and can interleave with another thread.

For compound actions use `synchronized`, a `ReentrantLock`, or atomic classes like `AtomicInteger` and `LongAdder`, which use compare-and-swap. A deadlock needs four conditions: mutual exclusion, hold and wait, no preemption and circular wait. You prevent it by always acquiring locks in a consistent global order, holding locks for as short a time as possible, using `tryLock` with a timeout, or avoiding shared mutable state altogether with immutable objects, message passing and higher-level utilities from `java.util.concurrent`. A thread dump (`jstack`) reports detected deadlocks.

Key points

`volatile`: visibility, not atomicity

`synchronized` / locks: mutual exclusion and visibility

Prevent deadlocks with consistent lock ordering and timeouts

```
private volatile boolean running = true; // visibility only
private final AtomicInteger counter = new AtomicInteger();

void work() {
while (running) {
counter.incrementAndGet(); // atomic, no lock
}
}
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/java>