100 Core Java Interview Questions

Mohammed Shahid

100 Core Java Questions

Author: Mohammed Shahid

Core Java

1. **Difference between JDK, JRE, and JVM?**

   - **JDK**: Java Development Kit, includes JRE + tools (e.g., `javac`).

   - **JRE**: Java Runtime Environment, includes JVM + libraries.

   - **JVM**: Java Virtual Machine, executes bytecode.
     No code example needed.

2. **What is JVM and is it platform independent?**
   JVM runs bytecode but is platform-dependent (specific to OS/hardware). Java code is platform-independent due to JVM.
   No code example needed.

3. **What is the difference between Stack and Heap Memory?**

   - **Stack**: Stores local variables, method calls.

   - **Heap**: Stores objects, managed by garbage collector.

   ```
   class Example {
       int instanceVar = 10; // Heap
       void method() {
           int localVar = 20; // Stack
       }
   }
   ```

4. **What is the difference between path and classpath?**

   - **Path**: OS variable for executables.

   - **Classpath**: Java variable for `.class` files/JARs.
     No code example needed.

5. **What is the difference between an instance variable and a local variable?**

   - **Instance variable**: Class-level, default values.

   - **Local variable**: Method-level, must be initialized.

   ```
   class Example {
       int instanceVar; // Instance, default 0
       void method() {
   ```

```java
        int localVar = 10; // Local, must initialize
        System.out.println(instanceVar + localVar);
    }
}
```

6. **Can we overload main method?**
   Yes, but only `public static void main(String[] args)` is the entry point.

```java
class MainOverload {
    public static void main(String[] args) {
        System.out.println("Main entry point");
    }
    public static void main(int x) {
        System.out.println("Overloaded main: " + x);
    }
}
```

7. **Can we have multiple public classes in a Java source file?**
   No, only one public class per file, matching file name.

```java
// File: Test.java
public class Test {}
class AnotherClass {} // Non-public is fine
```

8. **Difference between static block and instance block?**
   - **Static block**: Runs once at class loading.
   - **Instance block**: Runs per object creation.

```java
class Example {
    static { System.out.println("Static block"); }
    { System.out.println("Instance block"); }
}
```

9. **What is static import?**
   Imports static members directly.

```java
import static java.lang.Math.PI;
class Test {
    void show() {
        System.out.println(PI); // No Math.PI
    }
}
```

10. **What is the difference between import and static import?**
    - **Import**: Imports classes/packages.
    - **Static import**: Imports static members.

```java
import java.util.List;
import static java.lang.System.out;
class Test {
    void show() {
        out.println("Static import");
    }
}
```

Object-Oriented Concepts

11. **What is the difference between object-oriented and object-based programming language?**
    - **Object-oriented**: Supports inheritance, polymorphism (e.g., Java).
    - **Object-based**: Lacks inheritance/polymorphism (e.g., JavaScript).
      No code example needed.

12. **What is constructor chaining?**
    Calling one constructor from another using `this()` or `super()`.

```java
class Example {
    Example() { this(10); }
    Example(int x) { System.out.println("Value: " + x); }
}
```

13. **Can we override static methods?**
    No, static methods are hidden, not overridden.

```java
class Parent { static void show() { System.out.println("Parent"); } }
class Child extends Parent { static void show() { System.out.println("Child"); } }
```

14. **Can we override main method?**
    No, it's static.
    See Q6 for overloading example.

15. **Difference between Method Overloading and Method Overriding?**
    - **Overloading**: Same name, different parameters.
    - **Overriding**: Same signature in subclass.

```java
class Example {
    void show(int x) {} // Overloading
    void show(String s) {}
}
class Parent {
    void display() {}
}
class Child extends Parent {
    @Override
    void display() {} // Overriding
}
```

16. **Can we instantiate abstract class?**
    No, must be subclassed.

```java
abstract class Example {}
class Test extends Example {}
```

17. **Can we declare abstract methods as private?**
    No, must be `public` or `protected`.

```java
abstract class Example {
    abstract void show(); // Valid
    // private abstract void show(); // Invalid
}
```

18. **Can we use abstract and final together?**
    No, they're mutually exclusive.

No code example needed.

19. **Can we declare a class as abstract without having any abstract method?**
    Yes, to prevent instantiation.

```
abstract class Example {
    void show() { System.out.println("Concrete method"); }
}
```

20. **What is the use of abstract class?**
    Provides a blueprint for subclasses.

```
abstract class Shape {
    abstract void draw();
}
class Circle extends Shape {
    void draw() { System.out.println("Circle"); }
}
```

Interfaces and Inner Classes

21. **What is Marker interface?**
    Interface with no methods (e.g., `Serializable`).

```
class Example implements Serializable {}
```

22. **What is the difference between abstract class and interface?**
    - **Abstract class**: Can have concrete methods, fields.
    - **Interface**: Abstract methods, constants only (pre-Java 8).

```
abstract class A { void show() {} }
interface I { void show(); }
```

23. **Can we define a class inside an interface?**
    Yes, static nested class.

```
interface Example {
    class Inner {}
}
```

24. **Can we define interface inside a class?**
    Yes, nested interface.

```
class Example {
    interface InnerInterface {}
}
```

25. **What is nested interface?**
    Interface inside class/interface.

```
class Example {
    interface Nested { void show(); }
}
```

Exception Handling

26. **Can we override a method that throws runtime exception without throws clause?**
    Yes, runtime exceptions are unchecked.

```java
class Parent {
    void show() { throw new RuntimeException(); }
}
class Child extends Parent {
    void show() {} // No throws needed
}
```

27. **What is difference between final, finally, and finalize?**

    - **final**: Prevents modification/inheritance.
    - **finally**: Executes after try-catch.
    - **finalize**: GC cleanup (deprecated).

```java
final class Example {
    public void method() {
        try {} finally { System.out.println("Finally"); }
    }
}
```

28. **What is multi-catch block in exception handling?**
    Catches multiple exceptions in one block.

```java
try {
    // Code
} catch (IOException | SQLException e) {
    e.printStackTrace();
}
```

29. **What is try-with-resources in Java?**
    Auto-closes `AutoCloseable` resources.

```java
try (FileReader fr = new FileReader("file.txt")) {
    // Use fr
} catch (IOException e) {}
```

30. **What is the use of throw keyword?**
    Throws an exception explicitly.

```java
void check() {
    throw new RuntimeException("Error");
}
```

Threads and Concurrency

31. **What is the difference between throw and throws?**

    - **throw**: Throws exception.
    - **throws**: Declares exceptions.

```java
void method() throws IOException {
    throw new IOException("Error");
}
```

32. **Difference between wait and sleep?**

    - **wait**: Releases lock, needs `synchronized`.
    - **sleep**: Pauses thread, keeps lock.

```java
synchronized (obj) {
    obj.wait(); // Releases lock
}
Thread.sleep(1000); // Keeps lock
```

33. **Difference between notify and notifyAll?**

    - **notify**: Wakes one thread.

    - **notifyAll**: Wakes all waiting threads.

```java
synchronized (obj) {
    obj.notify(); // One thread
    obj.notifyAll(); // All threads
}
```

34. **Can we call run method directly to start a new thread?**
    Yes, but runs in current thread.

```java
Thread t = new Thread(() -> System.out.println("Run"));
t.run(); // Current thread
t.start(); // New thread
```

35. **Difference between start and run method?**

    - **start**: Creates new thread.

    - **run**: Executes in current thread.
      See Q34 example.

36. **Difference between Runnable and Callable interface?**

    - **Runnable**: `run()`, no return, no checked exceptions.

    - **Callable**: `call()`, returns value, throws exceptions.

```java
Runnable r = () -> System.out.println("Run");
Callable<String> c = () -> "Result";
```

37. **What is thread pool?**
    Reusable threads managed by `ExecutorService`.

```java
ExecutorService es = Executors.newFixedThreadPool(2);
es.submit(() -> System.out.println("Task"));
es.shutdown();
```

38. **What is the difference between user thread and daemon thread?**

    - **User thread**: Keeps JVM alive.

    - **Daemon thread**: JVM exits if only daemon threads remain.

```java
Thread t = new Thread();
t.setDaemon(true);
t.start();
```

39. **What is the purpose of ThreadLocal?**
    Thread-specific variables.

```java
ThreadLocal<Integer> tl = ThreadLocal.withInitial(() -> 1);
tl.set(2); // Thread-specific
```

40. **What is FutureTask?**
Wraps `Callable`, retrieves result via `Future`.

```
FutureTask<String> ft = new FutureTask<>(() -> "Done");
new Thread(ft).start();
System.out.println(ft.get());
```

41. **What is ExecutorService?**
Manages thread pools.
See Q37 example.

42. **What is CountDownLatch?**
Threads wait until count reaches zero.

```
CountDownLatch latch = new CountDownLatch(2);
new Thread(() -> { latch.countDown(); }).start();
latch.await(); // Waits
```

43. **What is CyclicBarrier?**
Threads wait until all reach barrier.

```
CyclicBarrier cb = new CyclicBarrier(2);
new Thread(() -> { try { cb.await(); } catch (Exception e) {} }).start();
```

44. **What is BlockingQueue?**
Queue that blocks on add/remove if full/empty.

```
BlockingQueue<Integer> q = new ArrayBlockingQueue<>(10);
q.put(1); // Blocks if full
```

45. **What is ConcurrentHashMap?**
Thread-safe `HashMap` with segmented locking.

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
map.put("key", 1);
```

Collections and Generics

46. **What is the difference between fail-fast and fail-safe iterator?**

    - **Fail-fast**: Throws `ConcurrentModificationException`.
    - **Fail-safe**: Works on copy, no exception.

```
List<String> list = new CopyOnWriteArrayList<>();
Iterator<String> it = list.iterator();
list.add("x"); // Fail-safe, no exception
```

47. **What is CopyOnWriteArrayList?**
Thread-safe `ArrayList`, creates copy on modification.
See Q46 example.

48. **What is difference between ArrayList and LinkedList?**

    - **ArrayList**: Array-based, fast access.
    - **LinkedList**: Node-based, fast insert/delete.

```
List<String> al = new ArrayList<>();
List<String> ll = new LinkedList<>();
```

49. **Difference between HashMap and Hashtable?**

- **HashMap**: Non-synchronized, allows null.
- **Hashtable**: Synchronized, no nulls.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put(null, 1); // Valid
```

50. **Difference between HashSet and TreeSet?**
    - **HashSet**: Unordered, faster.
    - **TreeSet**: Sorted, slower.

```
Set<String> hs = new HashSet<>();
Set<String> ts = new TreeSet<>();
```

51. **Difference between HashMap and TreeMap?**
    - **HashMap**: Unordered, faster.
    - **TreeMap**: Sorted, slower.

```
Map<String, Integer> hm = new HashMap<>();
Map<String, Integer> tm = new TreeMap<>();
```

52. **Difference between Iterator and ListIterator?**
    - **Iterator**: Forward-only, any collection.
    - **ListIterator**: Bidirectional, `List` only.

```
List<String> list = new ArrayList<>();
ListIterator<String> li = list.listIterator();
li.add("x");
```

53. **What is Type Erasure?**
    Generics removed at compile-time, replaced with `Object`/bounds.

```
List<String> list = new ArrayList<>();
// Compiled as List<Object>
```

54. **What is Heap Pollution?**
    Assigning raw type to parameterized type, causing type safety issues.

```
List list = new ArrayList<String>(); // Raw type
list.add(1); // Heap pollution
```

Java 8 Features

55. **What are default methods in interface?**
    Methods with implementation in interfaces.

```
interface Example {
    default void show() { System.out.println("Default"); }
}
```

56. **What is Functional Interface?**
    Interface with one abstract method.

```
@FunctionalInterface
interface Example {
    void show();
}
```

57. **What is lambda expression?**
    Anonymous function for functional interfaces.

```
Example e = () -> System.out.println("Lambda");
```

58. **What is method reference?**
    Shorthand for lambdas.

```
List<String> list = Arrays.asList("a", "b");
list.forEach(System.out::println);
```

59. **What is Stream API?**
    Processes collections functionally.

```
List<Integer> list = Arrays.asList(1, 2, 3);
list.stream().filter(n -> n > 1).forEach(System.out::println);
```

60. **What is Optional class?**
    Handles nullable values, avoids `NullPointerException`.

```
Optional<String> opt = Optional.ofNullable(null);
System.out.println(opt.orElse("Default"));
```

Miscellaneous

61. **What is tight coupling and loose coupling?**
    - **Tight**: Direct class dependencies.
    - **Loose**: Via interfaces.

```
interface Service { void execute(); }
class Client { Service s; Client(Service s) { this.s = s; } } // Loose
```

62. **What is Dependency Injection?**
    Providing dependencies externally.
    See Q61 example.

63. **What is Enum in Java?**
    Fixed set of constants.

```
enum Day { MON, TUE }
```

64. **What is Assertion in Java?**
    Tests assumptions, disabled in production.

```
assert x > 0 : "x must be positive";
```

65. **Can we overload or override static methods in Enum?**
    Overload yes, override no (static).

```
enum Example {
    ONE;
    static void show() {}
    static void show(int x) {} // Overload
}
```

66. **What is the purpose of transient keyword?**
    Skips variable in serialization.

```
class Example implements Serializable {
    transient int x;
}
```

67. **What is the purpose of volatile keyword?**
Ensures variable visibility across threads.

```
volatile boolean flag = false;
```

68. **Difference between Comparable and Comparator?**

- **Comparable**: `compareTo` in class.
- **Comparator**: Separate class, `compare`.

```
class Person implements Comparable<Person> {
    int age;
    public int compareTo(Person p) { return this.age - p.age; }
}
```

69. **What is Serialization and Deserialization?**

- **Serialization**: Object to byte stream.
- **Deserialization**: Byte stream to object.

```
class Example implements Serializable {}
```

70. **What is SerialVersionUID?**
Ensures serialization compatibility.

```
class Example implements Serializable {
    private static final long serialVersionUID = 1L;
}
```

71. **What is Externalizable interface?**
Custom serialization.

```
class Example implements Externalizable {
    public void writeExternal(ObjectOutput out) {}
    public void readExternal(ObjectInput in) {}
}
```

72. **Difference between checked and unchecked exception?**

- **Checked**: Must be handled/declared.
- **Unchecked**: Optional handling.

```
void method() throws IOException {} // Checked
void method2() throws RuntimeException {} // Unchecked
```

73. **What is the purpose of instanceof operator?**
Checks object type.

```
if (obj instanceof String) {}
```

74. **What is ClassLoader?**
Loads classes dynamically.

```
Class<?> c = ClassLoader.getSystemClassLoader().loadClass("Example");
```

75. **Can we have multiple catch blocks for a single try block?**
Yes.

```
try {} catch (IOException e) {} catch (SQLException e) {}
```

76. **What is finally block and when it is executed?**
Executes after try-catch.

```
try {} catch (Exception e) {} finally { System.out.println("Cleanup"); }
```

77. **Difference between final and effectively final?**

- **final**: Explicitly immutable.
- **Effectively final**: Not modified after initialization.

```
final int x = 10;
int y = 20; // Effectively final if not reassigned
```

78. **What is Reflection API?**
Inspects/modifies classes at runtime.

```
Class<?> c = Class.forName("java.lang.String");
```

79. **What is cloning and how to implement it?**
Creates object copy via `clone()`.

```
class Example implements Cloneable {
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

80. **Difference between shallow copy and deep copy?**

- **Shallow**: Copies references.
- **Deep**: Copies objects.

```
class Example implements Cloneable {
    int[] arr = {1, 2};
    @Override
    protected Object clone() throws CloneNotSupportedException {
        Example e = (Example) super.clone();
        e.arr = arr.clone(); // Deep copy
        return e;
    }
}
```

81. **What is Garbage Collection and how it works?**
Reclaims memory of unused objects (mark-and-sweep).

```
System.gc(); // Suggests GC
```

82. **What is OutOfMemoryError?**
Thrown when heap is full.

```
List<Object> list = new ArrayList<>();
while (true) list.add(new Object()); // May cause OutOfMemoryError
```

83. **Difference between == and equals?**

- `==`: Reference comparison.
- `equals`: Content comparison.

```java
String s1 = new String("a");
String s2 = new String("a");
System.out.println(s1 == s2); // False
System.out.println(s1.equals(s2)); // True
```

84. **What is hashCode and equals contract?**
    Equal objects must have same `hashCode`.

```java
class Example {
    @Override
    public boolean equals(Object o) { return true; }
    @Override
    public int hashCode() { return 1; }
}
```

85. **Can we override equals and not hashCode?**
    Yes, but breaks contract.
    See Q84 example.

86. **What is immutable class?**
    Objects can't be modified after creation.
    See Q87 example.

87. **How to create immutable class?**
    `final` class, `private final` fields, no setters.

```java
final class Immutable {
    private final int x;
    Immutable(int x) { this.x = x; }
    int getX() { return x; }
}
```

88. **What is Singleton class and how to implement it?**
    Single instance class.

```java
class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() { return INSTANCE; }
}
```

89. **How to break singleton class?**
    Via reflection/serialization/cloning.

```java
// Prevent via readResolve()
class Singleton implements Serializable {
    private Object readResolve() { return getInstance(); }
}
```

90. **What is double checked locking in Singleton?**
    Reduces synchronization overhead.

```java
class Singleton {
    private static volatile Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                    }
                }
            }
            return instance;
        }
    }
```

91. **What is Enum Singleton?**
    Thread-safe singleton using `enum`.

    ```
    enum Singleton {
        INSTANCE;
        void show() {}
    }
    ```

92. **What is Serialization Proxy Pattern?**
    Uses proxy for serialization.

    ```
    class Example implements Serializable {
        private Object writeReplace() { return new Proxy(); }
        private static class Proxy implements Serializable {}
    }
    ```

93. **What is Thread Safe Singleton?**
    Singleton safe for multi-threading.
    See Q90 example.

94. **Difference between static class and singleton class?**

    - **Static class**: No instance.

    - **Singleton**: Single instance.
      See Q88 example.

95. **What is Java Memory Model?**
    Defines thread-memory interactions.
    No code example needed.

96. **What is happens-before relationship?**
    Ensures visibility/ordering.

    ```
    volatile int x = 0; // Establishes happens-before
    ```

97. **What is escape analysis?**
    Allocates objects on stack if they don't escape.
    No code example needed.

98. **What is String pool?**
    Stores unique string literals in heap.

    ```
    String s1 = "hello";
    String s2 = "hello"; // Same reference
    ```

99. **What is String intern?**
    Adds string to pool.

    ```
    String s = new String("hello").intern();
    ```

100. **What is StringBuilder and StringBuffer?**
    - **StringBuilder**: Non-thread-safe, faster.
    - **StringBuffer**: Thread-safe, slower.

```java
StringBuilder sb = new StringBuilder("hello");
StringBuffer sbf = new StringBuffer("hello");
```