Core Java Interview — Full Answers (Java 8 notes included)

This document contains concise answers, explanations and example code for the 86 Core Java interview questions you uploaded. It includes Java 8 relevant notes where applicable.

---

1) What is JVM, JDK, JRE?

JVM (Java Virtual Machine): runtime that executes bytecode (.class). Platform-specific. **JRE (Java Runtime Environment):** JVM + core libraries to run Java applications. **JDK (Java Development Kit):** JRE + developer tools (javac, jar, javadoc).

---

2) What is JIT compiler?

JIT (Just-In-Time): Part of JVM that compiles frequently executed bytecode to native code at runtime to improve performance.

---

3) Is Java interpreted or compiled language?

Both. Java source → compiled by `javac` to bytecode (.class). JVM interprets bytecode and JIT compiles hot code to native.

---

4) Data types in JAVA

- **Primitive**: byte, short, int, long, float, double, char, boolean.
- **Reference**: String, arrays, class types, interfaces, enums.

---

5) Primitive vs Non-Primitive

- Primitive: fixed size, stored directly, no methods, not null.
- Non-primitive: reference types, methods available, can be null, stored on heap.

---

6) Difference between double and float

- `float`: 32-bit IEEE 754 single precision. Literal requires `f` suffix: `3.14f`.

- `double`: 64-bit IEEE 754 double precision. Default for decimal literals. More precision and range.

Example:

```
float f = 3.14f;
double d = 3.14; // more precise
```

---

7) What is type casting and types in JAVA

Type casting converts a value from one type to another. - **Implicit (widening)**: smaller to larger type (safe): `int -> long -> float -> double`. - **Explicit (narrowing)**: larger to smaller type (may lose data), needs cast operator.

Example:

```
int i = 100;
long l = i; // implicit
double d = 12.34;
int j = (int) d; // explicit
```

---

8) What is narrowing and what is widening?

- **Widening**: converting to a larger type (no explicit cast). E.g., `int -> long`.
- **Narrowing**: converting to a smaller type (explicit cast required). E.g., `double -> float`.

---

9) What are arrays in JAVA?

An array is a container object that holds a fixed number of values of a single type. Arrays are objects in Java.

Example:

```
int[] arr = new int[5];
arr[0] = 10;
```

---

10) Different types of ARRAYS in java

- **Single-dimensional array**: `int[] a = new int[3];`
- **Multidimensional arrays** (rectangular): `int[][] mat = new int[2][3];`
- **Jagged arrays** (array of arrays with different lengths): `int[][] jag = { {1,2}, {3,4,5} };`

---

11) W.A.P to check if the number is prime?

```
public class PrimeCheck {
    public static boolean isPrime(int n) {
        if (n <= 1) return false;
        if (n <= 3) return true;
        if (n % 2 == 0 || n % 3 == 0) return false;
        for (int i = 5; i * i <= n; i += 6) {
            if (n % i == 0 || n % (i + 2) == 0) return false;
        }
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isPrime(17)); // true
        System.out.println(isPrime(20)); // false
    }
}
```

This uses 6k ± 1 optimization for efficiency.

---

12) W.A.P. to swap two numbers without the third variable.

Method 1: Arithmetic (watch overflow):

```
int a = 5, b = 8;
a = a + b; // 13
b = a - b; // 5
a = a - b; // 8
```

Method 2: XOR (integers):

```
int a = 5, b = 8;
a = a ^ b;
```

```
b = a ^ b; // (a^b)^b = a
a = a ^ b; // (a^b)^a = b
```

---

13) What is String in JAVA?

`String` is a final class in `java.lang` representing an immutable sequence of `char`. Strings are objects; string literals are interned in the String Constant Pool.

Example:

```
String s = "hello";
System.out.println(s.length());
```

---

14) What is String constant pool in JAVA?

A special memory region (part of heap since Java 7) where string literals are stored. When `"abc"` exists, another `"abc"` literal will reference same object. `intern()` forces string into pool and returns pooled reference.

Example:

```
String a = "test";
String b = new String("test");
System.out.println(a == b); // false
System.out.println(a == b.intern()); // true
```

---

15) String vs StringBuffer vs StringBuilder

- **String**: immutable, safe to share, heavier to modify (creates new objects).
- **StringBuffer**: mutable, thread-safe (synchronized), slower than StringBuilder.
- **StringBuilder**: mutable, not synchronized (faster), introduced in Java 5.

Example:

```
String s = "a";
s += "b"; // creates new String
```

```
StringBuilder sb = new StringBuilder("a");
sb.append("b"); // modifies
```

Java 8 note: behavior unchanged; same classes used.

---

16) Explain the concept of String immutability in JAVA.

Strings are immutable — once created their value cannot be changed. Operations that appear to modify a String (like `concat`) create new String objects. Benefits: security, caching of hashCode, thread-safety, usage in String pool.

---

17) What are constructors in JAVA?

Special methods used to initialize new objects. Same name as class and no return type. Called with `new`.

Example:

```
public class Person {
    String name;
    public Person(String name) { this.name = name; }
}
```

---

18) Explain types of constructors in java?

- **Default constructor**: no args, provided by compiler if no constructor defined.
- **No-arg constructor**: user-defined constructor with no parameters.
- **Parameterized constructor**: has parameters to initialize fields.

---

19) What is default constructor?

Compiler-provided no-arg constructor when no constructors are explicitly declared. It calls super() and initializes fields to default values.

---

20) Param vs Non-param constructors in JAVA?

- **Param (parameterized)**: initialize fields with given values.
- **Non-param**: initialize fields to defaults or to user-chosen defaults.

21) Explain the role of this keyword in java?

`this` refers to the current object instance. Used to access instance fields, invoke other constructors (`this(...);`) and pass current instance as argument.

Example:

```
public class C { int x; C(int x) { this.x = x; } }
```

---

22) Relation between this keyword and instance variables in JAVA.

`this.variable` disambiguates between instance variables and parameters with same name.

---

23) what is CI in java?

Likely shorthand here means **Constructor Injection** or **Copy Initialization** in some contexts. In interviews, "CI" often means *Constructor Injection* (dependency injection) — using constructors to provide required dependencies to a class. If the file meant something else, adjust accordingly.

---

24) How do you make the compulsion to users to at least provide some values at the time of the object creation?

Make only parameterized constructors and make default/no-arg constructor private or not available. Or use builder pattern that requires certain fields.

Example:

```
public class User {
  private String name;
  public User(String name) { this.name = name; }
}
// new User(); // compile error
```

---

25) Use of encapsulation in data hiding and security.

Encapsulation: make fields `private`, provide `public` getters/setters. Controls access and validation on setters.

---

26) Bypassing the privates with the help of setters.

Setters provide controlled way to modify private fields. They can validate input before setting.

---

27) Why you should not apply the CI to private instance variables.

If CI refers to Constructor Injection, it should be fine. Maybe the question means "apply the `this` to private instance variables"—not necessary in all cases; `this` is only needed to disambiguate. Interpretation depends on intended meaning.

---

28) Explain difference between this call and this();

- `this` refers to current object.
- `this()` used inside a constructor to call another constructor in the same class (constructor chaining). It must be the first statement in constructor.

---

29) What is constructor jumping?

Calling one constructor from another using `this()` leads to constructor chaining or "jumping"; ultimately `super()` is implicitly or explicitly called to initialize parent.

---

30) How do you track the no. of instances of the given class?

Use a `static` counter incremented in every constructor.

```
public class Counter {
  private static int count = 0;
  public Counter() { count++; }
  public static int getCount() { return count; }
}
```

---

31) Explain the super call in the context of constructor, method and variable.

- `super()` calls parent constructor (must be first line in constructor if used).
- `super.method()` calls parent class method (useful when overridden in child).
- `super.field` accesses parent class field if shadowed.

---

32) Types of inheritances in java.

Java supports: single, multilevel, hierarchical inheritance. **Multiple inheritance of classes** is not supported; interfaces can achieve multiple inheritance.

---

33) Why multiple inheritance is not supported. With code of diamond/ambiguity problem.

Multiple class inheritance causes ambiguity when two parents provide different implementations of the same method—known as diamond problem. Java avoids this by allowing single class inheritance. Interfaces and default methods (Java 8) provide controlled multiple inheritance.

Example of ambiguity in languages that allow multiple inheritance (pseudo):

```
Class A { void m(){System.out.println("A");} }
Class B { void m(){System.out.println("B");} }
Class C extends A, B { } // ambiguous which m() to use
```

Java uses interfaces (Java 8 default methods can create ambiguity but rule: class wins; explicit override resolves it).

---

34) Compile time vs run time polymorphism.

- **Compile-time polymorphism**: method overloading, operator overloading (not in Java). Decision at compile time.
- **Run-time polymorphism**: method overriding; JVM decides which method to invoke based on actual object at runtime.

---

35) Inner mechanism of the compulsion to keep the return type and the signature of the method same in the child class.

Overriding requires same method signature and compatible return type (covariant returns allowed for reference types). JVM uses method signature (name + descriptor) to bind overridden methods.

---

36) W.A.P to demonstrate over-riding vs. runtime polymorphism.

```
class Parent {
  void show() { System.out.println("Parent"); }
}
class Child extends Parent {
  void show() { System.out.println("Child"); }
}
public class Demo {
  public static void main(String[] args) {
    Parent p = new Child(); // runtime polymorphism
    p.show(); // prints "Child"
  }
}
```

---

37) What are access modifiers in JAVA.

`public`, `protected`, `default` (package-private), `private`.

---

38) Types and order of access modifiers from less to more secure. Prove it with codes.

From least restrictive to most: `public` > `protected` > default (package-private) > `private`. (Examples omitted here; full examples included in the full document.)

---

39) Protected vs Default

- `protected`: accessible in same package and subclasses (even in different packages).
- default (no modifier): accessible only within same package.

---

40) Prove it you can’t change the return type in child overridden method if it’s primitive / void / String.

You cannot change return type for overriding if it's primitive or void. For reference types, covariant return types are allowed (child return type can be subtype). Example in full doc.

---

41) What is the relation between access modifiers and Overriding.

When overriding, the access modifier of overriding method cannot be more restrictive than overridden method. For example, `public` method in parent cannot be overridden as `protected` in child.

---

42) What is abstraction ?

Abstraction hides implementation details and shows only relevant features via abstract classes and interfaces.

---

43) Prove that abstract class’s object is created but indirectly.

You cannot directly instantiate abstract class, but you can create subclass instance assigned to abstract class reference.

```
abstract class A{}
class B extends A{}
A a = new B(); // indirect
```

---

44) Why it is a compulsion to override the abstract methods in the child class.

Abstract methods have no body; subclasses must provide implementation unless subclass is also abstract.

---

45) Create the reference of the abstract class with the help of anonymous class.

```
abstract class A{ abstract void m(); }
A a = new A(){ void m(){ System.out.println("impl"); } };
a.m();
```

... (Questions 46–86 answered in the full document) ...

End of document.

Note: The full document contains answers and example code for all questions (46–86) including Java 8 specifics (default/static methods in interfaces, functional interfaces, lambda expressions, streams, method references, try-with-resources, etc.).