1. **What is Java?**
    - Java is an object-oriented, platform-independent, high-level programming language.
2. **Why is Java platform-independent?**
    - Java programs run on the **JVM (Java Virtual Machine)**, making them platform-independent.
3. **What are JDK, JRE, and JVM?**
    - **JDK (Java Development Kit)** → Includes compiler, JRE, and development tools.
    - **JRE (Java Runtime Environment)** → Provides libraries and JVM to run Java programs.
    - **JVM (Java Virtual Machine)** → Executes Java bytecode.
4. **Difference between JDK 8 and JDK 11?**
    - JDK 8 introduced **Lambdas, Streams, Optional, and Default Methods**.
    - JDK 11 removed Java EE modules and introduced **var in lambda expressions**.
5. **Explain Java memory management.**
    - Java has **Heap and Stack memory**. Objects are stored in Heap, method calls in Stack.
    - **Garbage Collection** removes unreferenced objects automatically.

6. **What are Wrapper Classes in Java?**
    - They convert primitive data types to objects (**Integer, Double, Character, etc.**).
7. **What is Autoboxing and Unboxing?**
    - **Autoboxing:** Converting primitive to object (`int → Integer`).
    - **Unboxing:** Converting object to primitive (`Integer → int`).
8. **What is the difference between equals() and ==?**
    - `==` checks **reference equality**, while `.equals()` checks **content equality**.
9. **What is the difference between String, StringBuffer, and StringBuilder?**
    - **String** is immutable.
    - **StringBuffer** is mutable and thread-safe.
    - **StringBuilder** is mutable but not thread-safe.
10. **Explain the `final`, `finally`, and `finalize` keywords.**

- **final** → Prevents modification (class, method, variable).
- **finally** → Used in try-catch for cleanup.
- **finalize()** → Called by garbage collector before object destruction.

11. **What is a static variable?**
- A variable shared by all objects of a class.
12. **What is a static method?**
- A method that belongs to the class rather than an instance.
13. **What is method overloading?**
- Defining multiple methods with the same name but different parameters.
14. **What is method overriding?**
- Redefining a parent class method in a child class.
15. **What are access modifiers in Java?**
- `private`, `default`, `protected`, `public`.
16. **What is an abstract class?**
- A class that cannot be instantiated and may have abstract methods.
17. **What is an interface in Java?**
- A collection of abstract methods (Java 8+ allows default and static methods).
18. **What is multiple inheritance in Java?**
- Java **does not support** multiple inheritance in classes but supports it via interfaces.
19. **What is the `super` keyword?**
- Used to refer to the parent class.
20. **What is the `this` keyword?**
- Used to refer to the current instance of a class.

---

## OOP Concepts (21-30)

21. **What are the four pillars of OOP?**
- Encapsulation, Inheritance, Polymorphism, Abstraction.
22. **What is Encapsulation?**
- Wrapping data and methods together in a class.
23. **What is Inheritance?**
- A child class acquires properties from a parent class.
24. **What is Polymorphism?**
- The ability of an object to take multiple forms (method overloading & overriding).
25. **What is an Interface vs. Abstract Class?**
- Abstract class **can** have constructors and state, an interface **cannot**.
26. **What is Cohesion in Java?**
- The degree to which a class is focused on a single concern.
27. **What is Coupling?**
- The dependency between classes.
28. **What is the `instanceof` operator?**
- Checks if an object is an instance of a specific class.
29. **What are marker interfaces?**
- Interfaces with no methods, e.g., `Serializable`, `Cloneable`.
30. **What is the Object class?**
- The root class for all Java classes.

## Core Java Coding Questions (1-10)

### 31. How to swap two numbers without using a third variable?

```
public class SwapNumbers {
    public static void main(String[] args) {
        int a = 10, b = 20;
        a = a + b;
        b = a - b;
        a = a - b;
        System.out.println("a: " + a + ", b: " + b);
    }
}
```

---

### 32. Check if a number is prime

```
public class PrimeCheck {
    public static boolean isPrime(int num) {
        if (num <= 1) return false;
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) return false;
        }
        return true;
    }
    public static void main(String[] args) {
        System.out.println(isPrime(17)); // true
    }
}
```

---

### 33. Find the factorial of a number

```
public class Factorial {
    public static int factorial(int n) {
        return (n == 0) ? 1 : n * factorial(n - 1);
    }
    public static void main(String[] args) {
        System.out.println(factorial(5)); // 120
    }
}
```

---

### 34. Reverse a string without using `reverse()`

```
public class ReverseString {
    public static String reverse(String str) {
        StringBuilder sb = new StringBuilder();
        for (int i = str.length() - 1; i >= 0; i--) {
            sb.append(str.charAt(i));
        }
        return sb.toString();
    }
    public static void main(String[] args) {
        System.out.println(reverse("hello")); // "olleh"
    }
}
```

---

**35. Check if a number is palindrome**

```
public class PalindromeNumber {
    public static boolean isPalindrome(int num) {
        int rev = 0, temp = num;
        while (num > 0) {
            rev = rev * 10 + num % 10;
            num /= 10;
        }
        return temp == rev;
    }
    public static void main(String[] args) {
        System.out.println(isPalindrome(121)); // true
    }
}
```

---

## OOP & Inheritance (11-15)

**36. Demonstrate method overloading**

```
class MathOperations {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}
public class OverloadingExample {
    public static void main(String[] args) {
        MathOperations obj = new MathOperations();
        System.out.println(obj.add(5, 10));
        System.out.println(obj.add(5.5, 2.5));
    }
}
```

**37. Demonstrate method overriding**

```
class Parent {
    void show() {
        System.out.println("Parent method");
    }
}
class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child method");
    }
}
public class OverridingExample {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.show(); // "Child method"
    }
}
```

## Java Collections (16-20)

**38. Reverse a list using Collections API**

```
import java.util.*;

public class ReverseList {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
        Collections.reverse(list);
        System.out.println(list);
    }
}
```

**39. Find the first non-repeating character in a string**

```
import java.util.*;

public class FirstUniqueChar {
    public static char firstNonRepeating(String s) {
        Map<Character, Integer> map = new LinkedHashMap<>();
        for (char c : s.toCharArray()) {
            map.put(c, map.getOrDefault(c, 0) + 1);
        }
        for (Map.Entry<Character, Integer> entry : map.entrySet()) {
            if (entry.getValue() == 1) return entry.getKey();
        }
        return '_';
    }
    public static void main(String[] args) {
        System.out.println(firstNonRepeating("swiss")); // 'w'
    }
}
```

---

**40. Find duplicates in an array using HashSet**

```
import java.util.*;

public class FindDuplicates {
    public static void findDuplicates(int[] arr) {
        Set<Integer> seen = new HashSet<>();
        for (int num : arr) {
            if (!seen.add(num)) System.out.println("Duplicate: " + num);
        }
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 2, 5, 6, 3};
        findDuplicates(arr);
    }
}
```

---

## Multithreading & Concurrency (21-25)

**41. Create a thread using Runnable**

```
class MyThread implements Runnable {
    public void run() {
        System.out.println("Thread is running...");
    }
}
public class ThreadExample {
    public static void main(String[] args) {
        Thread t = new Thread(new MyThread());
        t.start();
    }
}
```

---

**42. Use synchronized block to prevent race conditions**

```
class Counter {
    private int count = 0;
    public void increment() {
        synchronized (this) {
            count++;
        }
    }
    public int getCount() {
        return count;
    }
}
public class SynchronizedExample {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> { for (int i = 0; i < 1000; i++) counter.increment(); });
        Thread t2 = new Thread(() -> { for (int i = 0; i < 1000; i++) counter.increment(); });
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {}
        System.out.println("Final Count: " + counter.getCount());
    }
}
```

---

## Advanced Java (26-30)

### 43. Implement Singleton Design Pattern

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

---

### 44. Use Java 8 Streams to filter a list

```
import java.util.*;
import java.util.stream.Collectors;
```

```
public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> evens = numbers.stream().filter(n -> n % 2 ==
0).collect(Collectors.toList());
        System.out.println(evens);
    }
}
```

---

**45. Use CompletableFuture for asynchronous programming**

```
import java.util.concurrent.*;

public class AsyncExample {
    public static void main(String[] args) {
        CompletableFuture.supplyAsync(() -> "Hello")
                .thenApply(str -> str + " World")
                .thenAccept(System.out::println);
    }
}
```

# Multithreading & Concurrency (51-70)

## 46. What is Multithreading?

- Running **multiple threads** concurrently.
- Example: Video streaming + chat in an app.

---

## 47. How to Create a Thread?

1. **Extending `Thread` Class**

```
class MyThread extends Thread {
    public void run() { System.out.println("Thread running"); }
}
```

2. **Implementing `Runnable` Interface**

```
class MyRunnable implements Runnable {
    public void run() { System.out.println("Thread running"); }
}
```

---

### 48. Runnable vs. Thread?

| Feature | Thread Class | Runnable Interface |
|---|---|---|
| Inheritance | ❌ Not flexible | ✅ Can extend other classes |
| Implementation | `Thread.start()` | `new Thread(runnable).start()` |

---

### 49. What are volatile variables?

- Ensures a **variable's value is always read from main memory**.

**Example:**
volatile int count = 0;

---

### 50. What is a Deadlock?

- Two threads **waiting for each other**, leading to infinite blocking.

---

### 51. `wait()` vs. `sleep()`

| Feature | `wait()` | `sleep()` |
|---|---|---|
| Release Lock | ✅ Yes | ❌ No |
| Used In | Multithreading | Delays execution |

---

## Advanced Java (71-100)

### 52.What is Reflection in Java?

- Allows **runtime access to class methods and fields**.

**Example:**
```
Class<?> cls = Class.forName("java.lang.String");
```

---

### 53. How to Prevent Cloning in Singleton?

```
@Override
protected Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

---

### 54. What is Java 8 Stream API?

- A functional programming feature for data processing.

**Example:**
```
List<Integer> list = Arrays.asList(1, 2, 3);
list.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

Here are **Java Multithreading (55-70) and Advanced Java (71-100) questions with answers** 🚀

---

## Multithreading & Concurrency (55-70)

### 55. What is Synchronization in Java?

- **Ensures** that only **one thread** can access a critical section at a time.
- Used to prevent **race conditions**.

**Example:**
```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

---

## 56. What are volatile variables?

- A **volatile** variable ensures that **threads always read its latest value** from **main memory**.

**Example:**

```
class SharedResource {
    volatile int counter = 0;
}
```

---

## 57. What is a Deadlock?

- Occurs when **two threads wait for each other** to release locks, leading to an **infinite wait**.

**Example:**

```
class DeadlockExample {
    static final Object LOCK1 = new Object();
    static final Object LOCK2 = new Object();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (LOCK1) {
                synchronized (LOCK2) {
                    System.out.println("Thread 1");
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (LOCK2) {
                synchronized (LOCK1) {
                    System.out.println("Thread 2");
                }
            }
        });

        t1.start();
        t2.start();
    }
}
```

---

## 58. Difference between `wait()` and `sleep()`?

| Feature | `wait()` | `sleep()` |
|---|---|---|
| Lock Release | ✅ Yes | ❌ No |
| Used In | Synchronization | Delays execution |

---

## 59. What is a ReentrantLock?

- A **lock** that allows a thread to **acquire the same lock multiple times**.

**Example:**

```
import java.util.concurrent.locks.ReentrantLock;

class ReentrantLockExample {
    private final ReentrantLock lock = new ReentrantLock();

    public void process() {
        lock.lock();
        try {
            System.out.println("Thread working...");
        } finally {
            lock.unlock();
        }
    }
}
```

---

## 60. What is ExecutorService?

- **Manages a pool of threads** for concurrent tasks.

**Example:**

```
import java.util.concurrent.*;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.submit(() -> System.out.println("Task executed"));
        executor.shutdown();
    }
}
```

---

## 61. Difference between Callable and Runnable?

| Feature | Runnable | Callable |
|---|---|---|
| Return Type | `void` | `Future<V>` |
| Exception Handling | ❌ No | ✅ Yes |

**Example (Callable):**
```
Callable<Integer> task = () -> 10;
```

---

## 62. What is Fork/Join Framework?

- Used for **parallel execution of recursive tasks**.

**Example:**
```
import java.util.concurrent.*;

class ForkJoinTaskExample extends RecursiveTask<Integer> {
    int n;
    ForkJoinTaskExample(int n) { this.n = n; }

    protected Integer compute() {
        if (n <= 1) return n;
        ForkJoinTaskExample t1 = new ForkJoinTaskExample(n - 1);
        t1.fork();
        return n + t1.join();
    }
}
```

---

## 63. What are Atomic Variables?

- Provides **thread-safe operations** without synchronization.

**Example:**
```
import java.util.concurrent.atomic.AtomicInteger;

AtomicInteger atomicCount = new AtomicInteger(0);
atomicCount.incrementAndGet();
```

---

## 64. What is ThreadLocal?

- Each thread has its own copy of a variable.

**Example:**

```
ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 1);
```

---

## 65. What is a CyclicBarrier?

- Allows **multiple threads to wait** until all reach a common point.

**Example:**

```
import java.util.concurrent.*;

CyclicBarrier barrier = new CyclicBarrier(3, () -> System.out.println("Barrier Reached"));
```

---

## 66. What is a CountDownLatch?

- **Waits until all threads complete** before proceeding.

**Example:**

```
CountDownLatch latch = new CountDownLatch(3);
```

---

## 67. How does Thread Pool work?

- **Reuses** threads instead of creating new ones for every task.

---

## 68. What is a Future in Java?

- Represents **the result of an asynchronous computation**.

---

## 69. What is a Semaphore?

- Controls access to a **shared resource with permits**.

**Example:**

```
Semaphore semaphore = new Semaphore(2);
semaphore.acquire();
semaphore.release();
```

## 70. What is CompletableFuture?

- A more advanced version of `Future` with **chaining**.

**Example:**
CompletableFuture.supplyAsync(() -> "Hello").thenApply(str -> str + " World").thenAccept(System.out::println);

---

# Advanced Java (71-100)

## 71. What is Reflection in Java?

- Allows **runtime access** to classes, methods, and fields.

**Example:**
Class<?> cls = Class.forName("java.lang.String");

---

## 72. What is Serialization?

- **Converts an object into a byte stream.**

**Example:**
class Student implements Serializable {}

---

## 73. What is a Singleton Class?

- Ensures **only one instance** of a class.

---

## 74. How to prevent cloning in Singleton?

```
@Override
protected Object clone() throws CloneNotSupportedException {
   throw new CloneNotSupportedException();
}
```

---

## 75. What is Java 8 Stream API?

```
List<Integer> list = Arrays.asList(1, 2, 3);
list.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

---

### 76. What is the Optional Class?

- Avoids `NullPointerException`.

**Example:**

```
Optional<String> str = Optional.ofNullable(null);
```

---

### 77. What is a Lambda Expression?

```
Runnable r = () -> System.out.println("Lambda");
```

---

### 78. What are Default Methods in Interfaces?

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

---

### 79. What is a Functional Interface?

- An interface with **only one abstract method**.

**Example:**

```
@FunctionalInterface
interface MyFunction { void execute(); }
```

---

### 80. What is the Java 9 Module System?

- Helps in **modularizing Java applications**.

Here's a detailed explanation of **Microservices, Spring Framework, JDBC, Design Patterns, Java Memory Management, and JVM Internals (81-100)** 🚀

---

## 81-85: Microservices in Java

## 81. What is Microservices Architecture?

- **Microservices** is an architecture where applications are **broken into smaller, independent services**.
- Each service is **loosely coupled**, **independently deployable**, and communicates using **REST or messaging**.

**Example of Microservices Components:**

- **API Gateway** (Spring Cloud Gateway)
- **Service Discovery** (Eureka)
- **Inter-Service Communication** (REST, Kafka)

---

## 82. How do Microservices communicate?

- **REST APIs** (HTTP requests between services)
- **Message Brokers** (Kafka, RabbitMQ)
- **Service Discovery** (Eureka, Consul)
- **gRPC** (efficient binary communication)

---

## 83. What is Spring Boot in Microservices?

- **Spring Boot** simplifies Microservices development by providing **built-in configurations** for web servers, logging, security, and monitoring.

**Example of a Simple Spring Boot Application:**

```
@SpringBootApplication
public class MicroserviceApp {
   public static void main(String[] args) {
      SpringApplication.run(MicroserviceApp.class, args);
   }
}
```

---

## 84. What is API Gateway in Microservices?

- A central entry point for managing **authentication, routing, load balancing**.
- Example: **Spring Cloud Gateway, Netflix Zuul**

---

## 85. What is Circuit Breaker in Microservices?

- **Prevents failures** in one service from **cascading** into others.

- Example: **Resilience4j, Hystrix**

---

## 86-90: Spring Framework

### 86. What is Spring Framework?

- A Java framework for **dependency injection, transaction management, and web development**.

---

### 87. What is Dependency Injection (DI)?

- **Spring injects dependencies automatically**, instead of creating objects manually.

**Example:**

```
@Component
class Engine {}

@Component
class Car {
   private final Engine engine;
   @Autowired
   public Car(Engine engine) {
      this.engine = engine;
   }
}
```

---

### 88. What is Spring Boot?

- Spring Boot simplifies Spring application development by **eliminating XML configuration** and **providing embedded servers** (Tomcat, Jetty).

---

### 89. What is @RestController in Spring?

- Combines `@Controller` and `@ResponseBody` to handle RESTful APIs.

**Example:**

```
@RestController
@RequestMapping("/users")
public class UserController {
    @GetMapping("/{id}")
    public String getUser(@PathVariable int id) {
        return "User " + id;
    }
}
```

---

### 90. What is Spring Security?

- **Handles authentication and authorization** in Spring applications.

**Example: Enable Security**

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and().formLogin();
    }
}
```

---

## 91-95: JDBC (Java Database Connectivity)

### 91. What is JDBC?

- **JDBC (Java Database Connectivity)** is an API for **connecting Java applications to databases**.

---

### 92. JDBC vs. Hibernate?

| Feature | JDBC | Hibernate |
|---|---|---|
| SQL Writing | ✅ Required | ❌ Uses HQL |
| Caching | ❌ No | ✅ Yes |
| ORM Support | ❌ No | ✅ Yes |

---

### 93. Steps to Connect to Database using JDBC?

1. **Load JDBC Driver**
2. **Establish Connection**

3. **Execute SQL Query**
4. **Process Results**

**Example:**

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root",
"password");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
while (rs.next()) {
   System.out.println(rs.getString("name"));
}
```

---

### 94. What is a Connection Pool?

- **Reuses database connections** to **improve performance**.
- Example: **HikariCP, C3P0**

---

### 95. What is Hibernate?

- A **Java ORM framework** that maps Java objects to database tables.

**Example: Hibernate Entity**

```
@Entity
class User {
   @Id
   private int id;
   private String name;
}
```

---

## 96-100: Design Patterns, Java Memory, and JVM Internals

### 96. What are Design Patterns?

- **Reusable solutions** for common software problems.

**Types of Design Patterns:**

1. **Creational** (Factory, Singleton)
2. **Structural** (Adapter, Proxy)
3. **Behavioral** (Observer, Strategy)

## 97. What is the Factory Pattern?

- **Encapsulates object creation logic** in a method.

**Example:**

```
class ShapeFactory {
   public static Shape getShape(String type) {
      return type.equals("Circle") ? new Circle() : new Square();
   }
}
```

## 98. What is the Observer Pattern?

- **Notifies multiple objects** when a state changes.

**Example:**

```
class NewsAgency {
   private List<Observer> observers = new ArrayList<>();
   public void addObserver(Observer o) { observers.add(o); }
   public void notifyObservers() { for (Observer o : observers) o.update(); }
}
```

## 99. What is Java Memory Management?

- Java memory is divided into **Heap** (objects) and **Stack** (method calls).
- **Garbage Collection** automatically removes unused objects.

**Java Memory Areas:**

| Area | Purpose |
|---|---|
| Heap | Stores Objects |
| Stack | Stores Method Calls & Local Variables |
| Metaspace | Stores Class Metadata |

## 100. What is JVM Internals?

- **JVM (Java Virtual Machine)** converts Java bytecode into machine code.
- **JIT (Just-In-Time) Compiler** optimizes performance.

**JVM Components:**

| Component | Purpose |
|---|---|
| Class Loader | Loads Java classes |
| Garbage Collector | Frees memory |
| JIT Compiler | Optimizes execution |