Core Java Interview Notes by Sahil Shahare

🗂️ Table of Contents (Topics 1-12)

🧩 Topic 1 – JVM, JDK, JRE & JIT (Q1–3)

1️⃣ What is JVM, JDK, JRE?

- 🧠 **Short & Crisp:**
  - **JVM** – Executes Java **bytecode**, providing platform independence.
  - **JRE** – Environment to **run** Java apps (JVM + libraries).
  - **JDK** – Full toolkit to **develop and run** Java (JRE + compiler + tools).
- 📘 **Detailed Explanation:**
  - **JVM (Java Virtual Machine):** Converts bytecode to machine code at runtime. Handles memory, class loading, and garbage collection.
  - **JRE (Java Runtime Environment):** Includes JVM and core class libraries needed to run Java programs.
  - **JDK (Java Development Kit):** Includes JRE + development tools like javac, debugger, and jar.
- 👉 **Relation:** JDK = JRE + Development Tools → JRE = JVM + Libraries.

2️⃣ What is JIT Compiler?

- 🧠 **Short & Crisp:**
  - The **JIT** (Just-In-Time) compiler converts bytecode into **native machine code** at runtime for faster execution.
- 📘 **Detailed Explanation:**
  - JVM first interprets bytecode line-by-line (slow).

    - JIT identifies frequently executed code (**hotspots**) and compiles it into machine code.
    - Improves speed while keeping portability.
- 💻 **Example:** Loops in Java are optimized by JIT for faster repeated execution.

3️⃣ Is Java interpreted or compiled language?

- 🧠 **Short & Crisp:**
    - Java is **both compiled and interpreted**.
    - **Compiled** → **.class** bytecode; **Interpreted** → JVM executes bytecode.
- 📘 **Detailed Explanation:**
    - javac compiles **.java** → **.class** (bytecode).
    - JVM interprets or JIT-compiles bytecode → machine code at runtime.
    - This makes Java **"Write Once, Run Anywhere."**
- 💻 **Pipeline:** Source (.java) → Bytecode (.class) → JVM (JIT) → Machine Code

---

🧩 Topic 2 – Data Types & Casting (Q4–8)

4️⃣ Data types in Java

- 🧠 **Short & Crisp:**
    - Java has **8 primitive types** and several **non-primitive (reference)** types.
- 📘 **Detailed Explanation:**
    - **Primitive:** byte, short, int, long, float, double, char, boolean.
    - **Non-Primitive:** String, Arrays, Classes, Interfaces, Enums.
    - Primitives store actual values; Non-primitives store memory references.

5️⃣ Primitive vs Non-Primitive

- 🧠 **Short & Crisp:**
    - Primitives store **actual values** in **stack**; non-primitives store **references** in **heap**.
- 📘 **Detailed Explanation:**

Feature	**Primitive**	**Non-Primitive**
Definition	Predefined by Java	Created by users
Memory	Stored in stack memory	Stored in heap memory

Speed	Faster	Slower (due to reference handling)
Example	int, float	String, Array

6️⃣ Difference between double and float

- 🧠 **Short & Crisp:**
  - **float** → **4 bytes**, 6–7 decimal precision.
  - **double** → **8 bytes**, 15–16 decimal precision.
- 📘 **Detailed Explanation:**
  - float f = 3.14f; (**must add f**)
  - double d = 3.14; (default type for decimals)
- 💡 **Use float for less precision (graphics), double for high accuracy (scientific).**

7️⃣ What is type casting and types in Java?

- 🧠 **Short & Crisp:**
  - Type casting converts one data type to another (compatible type).
- 📘 **Detailed Explanation:**
  - Two types:
    - **Widening (Implicit)** → small → large (int → double).
    - **Narrowing (Explicit)** → large → small (double → int).
- 💻 **Example:**

Java
```
int a = 10;
double b = a;      // widening
int c = (int) b;   // narrowing
```

8️⃣ What is narrowing and widening?

- 🧠 **Short & Crisp:**
  - **Widening:** Automatic conversion (no data loss).
  - **Narrowing:** Manual conversion (possible data loss).
- 📘 **Detailed Explanation:**

Type	Example	Cast Required	Risk
Widening	int → double	No	Safe

Narrowing	double → int	Yes	May lose data

🧩 Topic 3 – Arrays (Q9–10)

9️⃣ What are arrays in Java?

- 🧠 **Short & Crisp:**
  - An array is a container object that stores **multiple elements of the same type** in a contiguous memory location.
- 📘 **Detailed Explanation:**
  - Arrays are **fixed in size** once created.
  - Index starts from **0**.
- 💻 **Example:**

Java
```
int[] arr = {10, 20, 30};
System.out.println(arr[1]); // 20
```

🔟 Types of arrays in Java

- 🧠 **Short & Crisp:**
  - **1D**, **2D** (Matrix), and **Jagged Arrays**.
- 📘 **Detailed Explanation:**
  - **1D:** int[] a = new int[5];
  - **2D:** int[][] b = new int[3][3];
  - **Jagged:** Array of arrays with **different lengths**.
- 💻 **Example:**

Java
```
int[][] jagged = { {1,2}, {3,4,5} };
```

🧩 Topic 4 – Basic Programs & Strings (Q11–16)

1️⃣1️⃣ W.A.P to check if a number is prime

- 🧠 **Short & Crisp:**
  - A prime number is divisible only by **1 and itself**.
- 📘 **Detailed Explanation:**
  - To check primality, divide the number from 2 → $\sqrt{n}$.
- 💻 **Example:**

Java
```
int n = 7;
```

```
boolean prime = true;
for(int i = 2; i <= Math.sqrt(n); i++) {
    if(n % i == 0) {
        prime = false;
        break;
    }
}
System.out.println(prime ? "Prime" : "Not Prime");
```

1️⃣2️⃣ W.A.P to swap two numbers without the third variable

- 🧠 **Short & Crisp:**
  - Use **arithmetic** or **XOR** to swap two variables.
- 📘 **Detailed Explanation:**
- 💻 **Example 1 (Arithmetic):**

Java
```
int a = 5, b = 10;
a = a + b;
b = a - b;
a = a - b;
```

- 💻 **Example 2 (XOR):**

Java
```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

- Both methods work without using a third variable.

1️⃣3️⃣ What is String in Java?

- 🧠 **Short & Crisp:**
  - **String** is a sequence of characters stored as an object of java.lang.String.
- 📘 **Detailed Explanation:**
  - Strings are **immutable** (cannot change once created).
  - Can be created via **String literal** or **new keyword**:

Java
```
String s1 = "Hello";
String s2 = new String("Hello");
```

-
  - Stored in **String Constant Pool (SCP)** for memory efficiency.

1️⃣4️⃣ What is String Constant Pool (SCP)?

- 🧠 **Short & Crisp:**
  - **SCP** is a special memory area in the heap where **string literals** are stored to **avoid duplicates**.
- 📘 **Detailed Explanation:**
  - Literals are stored in SCP. If another variable has the same literal, it references the same memory.
- 💻 **Example:**

Java
String s1 = "Java";
String s2 = "Java";
System.out.println(s1 == s2); // true
// But using new String() creates a new object outside SCP.

1️⃣5️⃣ String vs StringBuffer vs StringBuilder

- 🧠 **Short & Crisp:**

Type	Mutability	Thread-safe	Performance
String	Immutable	Yes	Slow
StringBuffer	Mutable	Yes	Moderate
StringBuilder	Mutable	No	Fastest

- 📘 **Detailed Explanation:**
  - **String:** Every modification creates a new object.
  - **StringBuffer:** Mutable and **synchronized** (thread-safe).
  - **StringBuilder:** Mutable and **not synchronized** (faster).
- 💻 **Example:**

Java
StringBuilder sb = new StringBuilder("Java");
sb.append(" Rocks");
System.out.println(sb); // Java Rocks

1️⃣6️⃣ Explain String immutability

- 🧠 **Short & Crisp:**
  - Once created, a String object **cannot be modified**; any change creates a **new object**.
- 📘 **Detailed Explanation:**
  - Ensures security, thread safety, and caching.
- 💻 **Example:**

Java
String s = "Java";
s.concat(" Rocks");
System.out.println(s); // Java
// Here, "Rocks" creates a new String — original remains unchanged.

🧩 Topic 5 – Constructors (Q17–20)

1️⃣7️⃣ What are constructors in Java?

- 🧠 **Short & Crisp:**
  - A constructor is a **special method** used to **initialize objects** when they are created.
- 📘 **Detailed Explanation:**
  - Same name as class, **no return type**.
  - Called automatically when object is created.
- 💻 **Example:**

Java
class Demo {
  Demo() { System.out.println("Constructor called"); }
}

1️⃣8️⃣ Types of constructors in Java

- 🧠 **Short & Crisp:**
  - **Default Constructor**
  - **Parameterized Constructor**
- 📘 **Detailed Explanation:**
- 💻 **Example:**

Java
class Student {
  Student() { } // default (non-parameterized)
  Student(String name) { } // parameterized
}
// If no constructor is defined, Java automatically provides a default one.

1️⃣9️⃣ What is default constructor?

- 🧠 **Short & Crisp:**
  - The **compiler adds a no-argument constructor** if no constructor is explicitly defined.
- 📘 **Detailed Explanation:**
  - Initializes object with default values (0, null, false).
- 💻 **Example:**

Java
```
class Test {
   int a;
   Test() { }  // implicit if not written
}
```

2️⃣0️⃣ Param vs Non-Param constructors

- 🧠 **Short & Crisp:**
  - **Non-param:** No arguments → default initialization.
  - **Param:** Accepts values → customized initialization.
- 📘 **Detailed Explanation:**
- 💻 **Example:**

Java
```
class Car {
   String brand;
   Car() { brand = "Unknown"; } // Non-Param
   Car(String b) { brand = b; } // Param
}
// Calling new Car("BMW") sets the brand explicitly.
```

🧩 Topic 6 – this Keyword & Constructor Chaining (Q21–30)

2️⃣1️⃣ Explain the role of this keyword in Java

- 🧠 **Short & Crisp:**
  - this refers to the **current class object** — used to differentiate instance variables from parameters or call other constructors/methods.
- 📘 **Detailed Explanation:**
  - Commonly used when local variable names are same as instance variables (shadowing).
- 💻 **Example:**

Java

```
class Student {
   String name;
   Student(String name) {
      this.name = name;  // distinguishes instance var
   }
}
```

2️⃣2️⃣ Relation between this keyword and instance variables

- 🧠 **Short & Crisp:**
  - this gives access to the **instance variables** of the current object within methods or constructors.
- 📘 **Detailed Explanation:**
  - Resolves naming conflicts between instance and local variables.
  - Represents object's own data.
- 💻 **Example:**

Java
```
class Emp {
   int id;
   Emp(int id) { this.id = id; }
}
```

2️⃣3️⃣ What is CI in Java?

- 🧠 **Short & Crisp:**
  - **CI** stands for **Constructor Injection** — injecting values via constructors (a form of dependency injection).
- 📘 **Detailed Explanation:**
  - Promotes **immutability** and **loose coupling**.
- 💻 **Example:**

Java
```
class Engine { }
class Car {
   Engine e;
   Car(Engine e) { this.e = e; }  // constructor injection
}
```

2️⃣4️⃣ How do you make it compulsory to provide values during object creation?

- 🧠 **Short & Crisp:**
  - By defining a **parameterized constructor** (and **not** defining a default constructor).
- 📘 **Detailed Explanation:**

    - Without a no-arg constructor, users **must** pass parameters to create an object.
- 💻 **Example:**

Java
```
class User {
   String name;
   User(String name) { this.name = name; }
}
// User u = new User(); // Error
User u = new User("Sahil"); // must pass value
```

2️⃣5️⃣ Use of encapsulation in data hiding and security

- 🧠 **Short & Crisp:**
    - **Encapsulation** bundles data (variables) and methods together, restricting access via **private** fields + **public getters/setters**.
- 📘 **Detailed Explanation:**
    - Prevents unauthorized, direct modification of internal state.
    - Improves maintainability.
- 💻 **Example:**

Java
```
class Account {
   private double balance;
   public double getBalance() { return balance; }
   public void setBalance(double b) {
      if (b > 0) balance = b; // Validation/Control point
   }
}
```

2️⃣6️⃣ Bypassing privates with the help of setters

- 🧠 **Short & Crisp:**
    - **Private** variables can be modified **indirectly** using public **setters**.
- 📘 **Detailed Explanation:**
    - Setters act as **controlled access points**, allowing you to validate or add logic before modifying the private data.
- 💻 **Example:**
    - `acc.setBalance(5000); // indirectly modify private var`

2️⃣7️⃣ Why you should not apply CI to private instance variables

- 🧠 **Short & Crisp:**
    - CI should be used to initialize **dependencies**, not simple **hidden private data** that the class is responsible for managing internally.
- 📘 **Detailed Explanation:**

- If a variable is **private** and not a dependency (like a counter or a flag), it should typically be managed by the class's logic or initialized via a no-arg constructor/static block. Using CI for every private field can overcomplicate the class's API.

2️⃣8️⃣ Explain difference between this() and this call

- 🧠 **Short & Crisp:**
  - **this()** → calls **another constructor** in the same class (Constructor Chaining).
  - **this** → refers to the **current object** (instance).
- 📘 **Detailed Explanation:**
- 💻 **Example:**

Java
```
class A {
   A() { this(10); } // this() - calls parameterized constructor
   A(int x) { System.out.println("Value: " + x); }
}
// this() must be the first line in a constructor.
```

2️⃣9️⃣ What is constructor jumping?

- 🧠 **Short & Crisp:**
  - Constructor jumping (or **chaining**) is when one constructor calls another within the same class or superclass using **this()** or **super()**.
- 📘 **Detailed Explanation:**
  - Enables code reuse and avoids duplication of initialization logic across constructors.
- 💻 **Example:**

Java
```
class A {
   A() { this(10); } // Jumps to the second constructor
   A(int x) { System.out.println(x); }
}
// Control jumps from default to parameterized constructor.
```

3️⃣0️⃣ How do you track the number of instances of a given class?

- 🧠 **Short & Crisp:**
  - Use a **static counter variable** incremented inside the constructor.
- 📘 **Detailed Explanation:**
  - The static variable belongs to the class, not the object, so it tracks the total count across all instances.
- 💻 **Example:**

Java

```
class Counter {
   static int count = 0;
   Counter() { count++; } // Incremented for every new object
   static void showCount() {
      System.out.println("Objects: " + count);
   }
}
// Every object increases the count; showCount() prints total instances.
```

🧩 Topic 7 – super, Inheritance, and Polymorphism (Q31–40)

3️⃣1️⃣ Explain the super call in the context of constructor, method, and variable.

- 🧠 **Short & Crisp:**
  - super refers to the **parent class** — used to access parent variables, methods, or constructors.
- 📘 **Detailed Explanation:**
  - super.variable → accesses parent class variable (useful for shadowed variables).
  - super.method() → calls parent class method (useful before or after overriding).
  - super() → calls parent constructor (must be **first line**).
- 💻 **Example:**

Java
```
class Parent {
   int a = 10;
   void show() { System.out.println("Parent method"); }
}
class Child extends Parent {
   int a = 20;
   Child() { super(); } // calls Parent constructor
   void display() {
      System.out.println(super.a); // 10 (parent var)
      super.show();              // Parent method
   }
}
```

3️⃣2️⃣ Types of inheritance in Java.

- 🧠 **Short & Crisp:**
  - Java supports: **Single**, **Multilevel**, **Hierarchical**, and **Hybrid** inheritance (combination using interfaces) — but **not multiple** (using classes).
- 📘 **Detailed Explanation:**

Type	Example	Description
Single	A → B	One class inherits another
Multilevel	A → B → C	Inherits through multiple levels
Hierarchical	A → B, A → C	Multiple subclasses of one parent
Hybrid	Combo using Interfaces	Mix of types (Interfaces allow multi-inheritance)

3️3️⃣ Why multiple inheritance is not supported (Diamond Problem)?

- 🧠 **Short & Crisp:**
  - Because it creates **ambiguity** when two parent classes have the **same method** — the compiler won't know which one to call.
- 📘 **Detailed Explanation:**
  - The ambiguity arises because a subclass cannot decide which superclass method implementation to inherit.
- 💻 **Example (Conceptual, Java Class Error):**

Java
class A { void show(){} }
class B { void show(){} }
class C extends A, B { } // ❌ Error

- ✅ **Solution:** Java uses **Interfaces** to achieve the benefits of multiple inheritance without the diamond problem.

3️4️⃣ Compile-time vs Run-time polymorphism.

- 🧠 **Short & Crisp:**
  - **Compile-time: Method Overloading**
  - **Run-time: Method Overriding**
- 📘 **Detailed Explanation:**
  - **Overloading:** Same method name, different parameters → resolved at **compile time** (Static Binding).
  - **Overriding:** Subclass redefines parent method → resolved at **runtime** via dynamic binding.

- 💻 **Example:**

```
Java
class A {
   void show(int x){ System.out.println("A"); }
}
class B extends A {
   void show(int x){ System.out.println("B"); } // Overriding
}
A obj = new B();  // Polymorphic reference
obj.show(5); // Output: B (runtime polymorphism)
```

3️⃣5️⃣ Why must the return type and signature be the same in overriding?

- 🧠 **Short & Crisp:**
  - To ensure the overridden method correctly **replaces** the parent version without changing the method's contract or breaking **runtime polymorphism**.
- 📘 **Detailed Explanation:**
  - JVM uses dynamic dispatch to call methods based on runtime type. Changing the signature or an incompatible return type breaks the overriding rules.
  - **Covariant Return:** Allowed — a subclass can return a **subtype** of the parent's return type (e.g., Parent returns Animal, Child can return Dog).

3️⃣6️⃣ W.A.P to demonstrate overriding vs runtime polymorphism.

- 🧠 **Short & Crisp:**
  - **Overriding** defines a method in child class with the same signature as parent — **Runtime polymorphism** ensures the child's version is executed.
- 📘 **Detailed Explanation:**
- 💻 **Example:**

```
Java
class Animal {
   void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
   void sound() { System.out.println("Dog barks"); } // Overriding
}
Animal a = new Dog(); // Runtime reference (Parent type, Child object)
a.sound(); // Output: Dog barks (Child's method is called at runtime)
```

3️⃣7️⃣ What are access modifiers in Java?

- 🧠 **Short & Crisp:**
  - Keywords that define the **visibility** (access level) of classes, methods, and variables.

- 📘 **Detailed Explanation:**
  - public – accessible everywhere
  - protected – accessible in same package + subclasses (even in different packages)
  - default (no modifier) – accessible only in the **same package**
  - private – accessible **only within the class**

3️⃣8️⃣ Types and order of access modifiers (less → more secure).

- 🧠 **Short & Crisp:**
  - Most Accessible → Least Accessible (Least Secure → Most Secure): public → protected → default → private
- 📘 **Detailed Explanation:**
- 💻 **Example:**

```
Java
public class A {
   public int x;
   protected int y;
   int z;       // default
   private int w;
}
// public = most accessible; private = most restricted.
```

3️⃣9️⃣ Protected vs Default.

- 🧠 **Short & Crisp:**
  - **Default:** accessible only within the **same package**.
  - **Protected:** accessible within the **same package** and **subclasses** (even in different packages).
- 📘 **Detailed Explanation:**
- 💻 **Example:**

```
Java
// Package p1
public class A { protected void show(){} }

// Package p2
class B extends A {
   void test(){
      // Works because B is a subclass of A, even though they are in different packages
      show();
   }
}
```

4️⃣0️⃣ Why can't you change the return type in overridden methods (primitive/void/String)?

- 🧠 **Short & Crisp:**
  - Because primitive and final types are **not covariant** — the return type must match **exactly**.
- 📘 **Detailed Explanation:**
  - **Covariant return types** (where the child method returns a subtype of the parent method's return type) are **only allowed for object types**. For primitives (int, void, float, etc.) and non-subclassable final types like String, the return type must be identical.
- 💻 **Example:**

Java
```
class A { int show(){ return 1; } }
class B extends A {
   // String show(){ return "Hi"; } ❌ Error: Incompatible return type
}
```

🧩 Topic 8 – Access Modifiers, Abstraction & Interfaces (Q41–50)

4️⃣1️⃣ What is the relation between access modifiers and overriding?

- 🧠 **Short & Crisp:**
  - A child class **cannot reduce the visibility** of an overridden method.
- 📘 **Detailed Explanation:**
  - You can make the method **more accessible** (protected → public), but not less (public → private). This preserves the contract promised by the parent class.
- 💻 **Example:**

Java
```
class Parent {
   protected void show() {}
}
class Child extends Parent {
   public void show() {}  // ✅ allowed (more visible)
   // private void show() {} // ❌ not allowed (less visible)
}
```

4️⃣2️⃣ What is abstraction?

- 🧠 **Short & Crisp:**
  - Abstraction **hides implementation details** and shows only **essential features** to the user.

- 📘 **Detailed Explanation:**
  - Achieved using **abstract classes** and **interfaces**.
  - Focuses on **"what"** an object does rather than **"how"** it does it.
- 💻 **Example:**

Java
```
abstract class Shape {
   abstract void draw(); // The 'what'
}
class Circle extends Shape {
   void draw(){ System.out.println("Drawing Circle"); } // The 'how'
}
```

4️⃣3️⃣ Prove that an abstract class's object is created indirectly.

- 🧠 **Short & Crisp:**
  - You can't instantiate an abstract class directly, but its **child object implicitly calls its constructor** as part of the inheritance chain.
- 📘 **Detailed Explanation:**
  - An abstract class cannot have a direct object (new AbstractClass()), but its instance is created as a **base part** of the concrete subclass object.
- 💻 **Example:**

Java
```
abstract class A { A(){ System.out.println("A created"); } } // Abstract constructor
class B extends A { B(){ System.out.println("B created"); } }
B obj = new B();  // indirectly calls A() constructor first
// Output:
// A created
// B created
```

4️⃣4️⃣ Why is it compulsory to override abstract methods in the child class?

- 🧠 **Short & Crisp:**
  - Because an abstract method has **no body**, the **subclass must provide an implementation** (a concrete body) to become a usable, non-abstract class.
- 📘 **Detailed Explanation:**
  - If the subclass does not override all abstract methods, it **must itself be declared abstract**.
- 💻 **Example:**

Java
```
abstract class Animal { abstract void sound(); }
class Dog extends Animal {
   void sound(){ System.out.println("Bark"); } // Provides the required body
}
```

4️⃣5️⃣ Create reference of abstract class using anonymous class.

- 🧠 **Short & Crisp:**
  - You can create **anonymous subclasses** (an inner class without a name) to instantiate an abstract class on the spot and provide implementation for its abstract methods.
- 📘 **Detailed Explanation:**
- 💻 **Example:**

Java
```
abstract class A { abstract void show(); }

A obj = new A() { // Anonymous class starts here
   void show(){ System.out.println("Anonymous class"); }
};
obj.show(); // Output: Anonymous class
```

4️⃣6️⃣ Difference between Abstract Class and Interface.

- 🧠 **Short & Crisp:**

Feature	**Abstract Class**	**Interface**
Methods	Both concrete & abstract	Abstract (till Java 7), Default/Static (Java 8+)
Variables	Can have instance variables	public static final only
Inheritance	Single (extends)	Multiple (implements)
Constructors	Yes (called by child)	No

- 📘 **Detailed Explanation:**
  - Use an **abstract class** when you need **partial implementation** or shared state (instance variables); use an **interface** for defining a **contract** between unrelated classes.

4️⃣7️⃣ Solve the diamond problem in Java using interfaces (default methods).

- 🧠 **Short & Crisp:**
  - Java 8's **default methods** allow multiple inheritance via interfaces, resolving conflicts by forcing the implementing class to **explicitly choose** which method to call.
- 📘 **Detailed Explanation:**
- 💻 **Example:**

Java
```
interface A { default void show(){ System.out.println("A"); } }
interface B { default void show(){ System.out.println("B"); } }

class C implements A, B {
   // Must override 'show' to resolve the conflict from A and B
   public void show() {
      A.super.show(); // Explicitly calls the A's default method
   }
}
```

4️⃣8️⃣ What is backward compatibility in Java 8?

- 🧠 **Short & Crisp:**
  - New features (like **default/static methods** in interfaces) were added **without breaking old code**, ensuring older programs still compile and run.
- 📘 **Detailed Explanation:**
  - Before Java 8, adding a new method to an interface would break all implementing classes. **Default methods** solved this, ensuring smooth migration from older Java versions.

4️⃣9️⃣ Default and static methods in Java 8 interfaces.

- 🧠 **Short & Crisp:**
  - **Default methods:** Defined in interface with the **default** keyword (can be overridden).
  - **Static methods:** Defined with the **static** keyword, accessed via **interface name** (cannot be overridden).
- 📘 **Detailed Explanation:**
- 💻 **Example:**

Java
```
interface A {
   default void show() { System.out.println("Default"); }
   static void print() { System.out.println("Static"); }
}
```

A.print(); // Accessing static method via interface name
// To call default method:
new A(){}.show();

5️⃣0️⃣ Overriding in interface with static and default methods.

- 🧠 **Short & Crisp:**
  - **Default methods** → **can be overridden** by the implementing class.
  - **Static methods** → **cannot be overridden** (they are hidden).
- 📘 **Detailed Explanation:**
- 💻 **Example:**

Java
interface A { default void show(){} static void print(){} }

class B implements A {
  public void show(){} // ✅ Override is allowed for default methods
  // public static void print(){} ❌ Not allowed (only hides the interface's static method)
}

🧩 Topic 9 – Static Methods, Functional Interfaces & Exception Handling (Q51–60)

5️⃣1️⃣ Explain method hiding with static methods in Java 8.

- 🧠 **Short & Crisp:**
  - Static methods are bound at **compile time**, so they **cannot be overridden**, only **hidden**.
- 📘 **Detailed Explanation:**
  - When a subclass defines a static method with the same signature as the parent, it **hides** the parent method. The version called depends on the **reference type**, not the object type (unlike overriding).
- 💻 **Example:**

Java
class A { static void show(){ System.out.println("A"); } }
class B extends A { static void show(){ System.out.println("B"); } }

A obj = new B();
obj.show(); // Output: A (compile-time binding, reference type is A)

5️⃣2️⃣ What is a functional interface?

- 🧠 **Short & Crisp:**
  - An interface with **exactly one abstract method**.

- 📘 **Detailed Explanation:**
  - Can have multiple default or static methods (Java 8+).
  - The sole purpose is to enable the usage of **Lambda Expressions**.
- 💻 **Example:**

Java
```
@FunctionalInterface
interface Sayable {
    void say(String msg); // Only one abstract method
}
// Examples: Runnable, Callable, Comparator.
```

5️⃣3️⃣ How do you invoke an interface method using an anonymous class?

- 🧠 **Short & Crisp:**
  - By creating an **anonymous class** that implements the interface on the spot (typically for a single-use implementation).
- 📘 **Detailed Explanation:**
- 💻 **Example:**

Java
```
interface A { void show(); }

A obj = new A() { // Creates an anonymous class implementing A
    public void show() { System.out.println("Hello"); }
};
obj.show(); // Output: Hello
// Used widely in GUI and event handling.
```

5️⃣4️⃣ Writing of lambda expressions.

- 🧠 **Short & Crisp:**
  - Lambda expressions provide a **concise way** to implement the single abstract method of a functional interface.
- 📘 **Detailed Explanation:**
  - Syntax: (parameters) -> expression or (parameters) -> { statements }
- 💻 **Example:**

Java
```
interface Add { int sum(int a, int b); }

Add add = (a, b) -> a + b; // Lambda implementation
System.out.println(add.sum(3, 4)); // 7
```

5️⃣5️⃣ Object class in Java.

- 🧠 **Short & Crisp:**
  - **Object** is the **root superclass** of all Java classes.
- 📘 **Detailed Explanation:**
  - Every class implicitly extends Object.
  - Provides common methods inherited by all classes: toString(), equals(), hashCode(), clone(), getClass(), and wait/notify methods.
- 💻 **Example:**

Java
class A {}
A obj = new A();
System.out.println(obj.toString()); // Inherited from Object

5️⃣6️⃣ toString() and getClass() methods.

- 🧠 **Short & Crisp:**
  - **toString()** → Returns a **string representation** of the object.
  - **getClass()** → Returns the **runtime class** of the object.
- 📘 **Detailed Explanation:**
  - It is best practice to **override toString()** to print meaningful output instead of the default class@hashcode.
- 💻 **Example:**

Java
class A {}
A obj = new A();
System.out.println(obj.toString()); // Output: A@<hashcode>
System.out.println(obj.getClass()); // Output: class A

5️⃣7️⃣ Hierarchy of exception handling.

- 🧠 **Short & Crisp:**
  - Throwable → Exception → RuntimeException (Unchecked) or IOException (Checked), etc.
- 📘 **Detailed Explanation:**
  - Throwable is the superclass for all exceptions and errors.
  - **Errors** (e.g., OutOfMemoryError) are severe, unrecoverable system problems.
  - **Exceptions** are problems that can be handled.
- 💻 **Hierarchy (Partial):**

Throwable
├── Exception (Catchable)
│ ├── IOException (Checked)
│ ├── SQLException (Checked)
│ └── RuntimeException (Unchecked)
└── Error (Uncatchable/Critical)

5️⃣8️⃣ Is it compulsory to write the catch block with try? Explain.

- 🧠 **Short & Crisp:**
  - **No** — you can use a try block with a **finally** block, even without a catch block.
- 📘 **Detailed Explanation:**
  - catch handles exceptions; finally executes cleanup code (always runs, regardless of exception). The **combination** of try must be with either catch or finally (or both).
- 💻 **Example:**

Java
```
try {
   // code that might throw an exception
} finally {
   System.out.println("Cleanup done (always runs)");
}
```

5️⃣9️⃣ Can we write consecutive try/finally blocks?

- 🧠 **Short & Crisp:**
  - **Yes**, you can write multiple, independent try–finally blocks without a catch in between, though this is less common.
- 📘 **Detailed Explanation:**
  - Each try–finally block executes independently.

6️⃣0️⃣ How can we write multiple catch blocks?

- 🧠 **Short & Crisp:**
  - By chaining multiple catch blocks after a try, each handling **specific exceptions**.
- 📘 **Detailed Explanation:**
  - Always put **specific catches first** (e.g., ArithmeticException), and the **general** catch (Exception) **last**.
- 💻 **Example:**

Java
```
try {
   int a = 5 / 0; // Throws ArithmeticException
} catch (ArithmeticException e) {
   System.out.println("Divide by zero");
} catch (Exception e) { // General catch
   System.out.println("Generic exception occurred");
}
```

🧩 Topic 10 – Advanced Exception Handling & Object Methods (Q61–70)

6️⃣1️⃣ Why is it compulsory to handle checked exceptions, but optional for unchecked?

- 🧠 **Short & Crisp:**
  - **Checked** exceptions are known at **compile-time**, so the compiler enforces handling; **unchecked** exceptions occur at **runtime**, so handling is optional.
- 📘 **Detailed Explanation:**
  - **Checked** (e.g., IOException) → Indicates issues with external resources that developers must plan for.
  - **Unchecked** (e.g., NullPointerException) → Indicates programming bugs or logic errors, which the program is not expected to recover from easily.

6️⃣2️⃣ Difference between compile-time and runtime exceptions.

- 🧠 **Short & Crisp:**
  - **Compile-time (Checked):** Detected by compiler.
  - **Runtime (Unchecked):** Occur during program execution.
- 📘 **Detailed Explanation:**

Type	Example	Handling
Checked	IOException, SQLException	**Mandatory** (try-catch or throws)
Unchecked	NullPointerException, ArithmeticException	**Optional** (Indicates a bug)

6️⃣3️⃣ When to use the throw keyword?

- 🧠 **Short & Crisp:**
  - throw is used to **explicitly throw** an exception object inside a method or block.
- 📘 **Detailed Explanation:**
  - Used to trigger **custom** or predefined exceptions manually when a specific condition is met (e.g., input validation).
- 💻 **Example:**

Java
```
if(age < 18)
   throw new ArithmeticException("Not eligible for this process");
```

6️⃣4️⃣ Can you throw multiple exceptions from a single try block?

- 🧠 **Short & Crisp:**
  - **Yes** — the block can **potentially** throw multiple different types of exceptions, which can then be handled by multiple catch blocks.
- 📘 **Detailed Explanation:**
  - Only **one** exception is actually thrown and processed per execution. The first one encountered terminates the try block, and control is passed to the relevant catch block.
- 💻 **Example:**

Java
```
try {
   if (Math.random() > 0.5) {
      int a = 5/0; // Throws ArithmeticException
   } else {
      String s = null;
      s.length(); // Throws NullPointerException
   }
} catch(ArithmeticException e) {
   System.out.println("Math error");
} catch(NullPointerException e) {
   System.out.println("Null error");
}
```

6️⃣5️⃣ Can we throw an Exception from a constructor?

- 🧠 **Short & Crisp:**
  - **Yes** — constructors can throw exceptions like any method, and you must handle or declare them.
- 📘 **Detailed Explanation:**
  - If a constructor throws a checked exception, the class that calls new must either use try-catch or also declare the exception with throws.
- 💻 **Example:**

Java
```
import java.io.IOException;

class Demo {
   Demo() throws IOException { // Must declare checked exception
      throw new IOException("Error during initialization");
   }
}
```

6️⃣6️⃣ What is a static block and its use?

- 🧠 **Short & Crisp:**
  - A static block is executed **once** when the class is **loaded** into the JVM — mainly used for static initialization.
- 📘 **Detailed Explanation:**
  - Used to initialize static variables that require complex logic (more than a simple assignment).
- 💻 **Example:**

Java
```
class Test {
   static int value;
   static {
      // Runs before any object is created or any static method is called
      System.out.println("Static block executed");
      value = 100;
   }
}
```

67 Custom compile-time vs custom runtime exceptions.

- 🧠 **Short & Crisp:**
  - Custom **Checked** (Compile-time): Extend **Exception**.
  - Custom **Unchecked** (Runtime): Extend **RuntimeException**.
- 📘 **Detailed Explanation:**
  - Use **Checked** for recoverable, anticipated errors (e.g., File not found).
  - Use **Unchecked** for programming logic errors or bugs (e.g., Invalid input argument).
- 💻 **Example:**

Java
```
class MyCheckedEx extends Exception {}
class MyRuntimeEx extends RuntimeException {}
```

68 What is hashCode() and its significance?

- 🧠 **Short & Crisp:**
  - hashCode() returns a numeric **hash** used for **efficient storage and retrieval** of objects in hash-based collections (HashMap, HashSet).
- 📘 **Detailed Explanation:**
  - It's the first step in locating an object in a hash-based structure. If objects are equal (via equals()), they **must** produce the same hash code.
- 💻 **Example:**

Java
```
@Override
public int hashCode() {
```

```
    return Objects.hash(id, name);
}
```

6️⃣9️⃣ What is .intern() method and String interning in Java?

- 🧠 **Short & Crisp:**
  - intern() ensures that identical string literals share a single memory reference in the **String Constant Pool (SCP)**.
- 📘 **Detailed Explanation:**
  - When called on a String object, if a string with the same content exists in the SCP, it returns the reference to the pooled string; otherwise, it adds the string to the pool and returns its reference.
- 💻 **Example:**

```
Java
String s1 = new String("Java"); // Object outside SCP
String s2 = s1.intern(); // References the one in SCP
String s3 = "Java"; // Also references the one in SCP
System.out.println(s2 == s3); // true (same reference in SCP)
```

7️⃣0️⃣ How can we override the hashCode() method?

- 🧠 **Short & Crisp:**
  - Override it to ensure **equal objects** produce the **same hash value**, maintaining the contract with equals().
- 📘 **Detailed Explanation:**
  - You should use the same fields for calculating hashCode() that you use for comparing equality in equals().
- 💻 **Example:**

```
Java
import java.util.Objects;
class Emp {
    int id; String name;
    // Must be overridden with equals()
    @Override
    public int hashCode() {
        return Objects.hash(id, name); // Utility for easy and correct hashing
    }
}
```

🧩 Topic 11 – Throw vs Throws, Exception Propagation, and Object Equality (Q71–80)

7️⃣1️⃣ String class and hashCode() method.

- 🧠 **Short & Crisp:**
  - In String, hashCode() is **overridden** to produce the **same hash** for identical sequences of characters, ensuring content-based hashing.
- 📘 **Detailed Explanation:**
  - The hash code is computed based on the characters in the string, making lookups in HashMap and HashSet efficient. It is also **cached** for performance since strings are immutable.
- 💻 **Example:**

Java
String s1 = "Java";
String s2 = "Java";
System.out.println(s1.hashCode() == s2.hashCode()); // true

7️⃣2️⃣ Throw vs Throws.

- 🧠 **Short & Crisp:**

Feature	throw	throws
Purpose	Used to explicitly **throw** an exception object.	Used to **declare** exceptions a method might throw.
Location	Inside method body.	In method signature.
Usage	Followed by an **instance** (new Exception()).	Followed by a **class name** (IOException).

- 📘 **Detailed Explanation:**

Java
// Declaration with throws
void test() throws IOException {
    // Throwing with throw
    throw new IOException("Error");
}

7️⃣3️⃣ Pipe operator (|) and its constraint in Java 7.

- 🧠 **Short & Crisp:**
  - The **multi-catch** operator (|) allows handling **multiple exceptions** in **one catch block** (Java 7+).
- 📘 **Detailed Explanation:**
  - **Constraint** → the caught exceptions **must not have a parent-child relationship**. For example, you cannot catch Exception | IOException because IOException is a child of Exception.
- 💻 **Example:**

Java
```
try {
   // code
} catch (IOException | SQLException e) { // Catches either I/O or SQL exceptions
   System.out.println("Handled I/O or SQL error: " + e);
}
```

7️⃣4️⃣ instanceof operator.

- 🧠 **Short & Crisp:**
  - Used to check if an object is an **instance** of a specific class, subclass, or interface.
- 📘 **Detailed Explanation:**
  - It returns true if the object is compatible with the type on the right-hand side.
- 💻 **Example:**

Java
```
Animal a = new Dog(); // Dog extends Animal
if(a instanceof Dog)
   System.out.println("a is a Dog"); // true
if(a instanceof Animal)
   System.out.println("a is an Animal"); // true
```

7️⃣5️⃣ Propagation of exception in Java to JVM.

- 🧠 **Short & Crisp:**
  - If an exception is **not handled** in a method, it **propagates up the call stack** to the calling method, until it reaches the **JVM**, which then terminates the program.
- 📘 **Detailed Explanation:**
  - This unwinding of the call stack is called **exception propagation**. If a method in the chain handles the exception, propagation stops.
- 💻 **Example (Propagation):**

Java
```
void m1(){ int x = 10/0; } // Throws ArithmeticException
```

```
void m2(){ m1(); } // Does not handle, propagates
void main(){ m2(); } // Does not handle, propagates to JVM
// JVM terminates and prints stack trace
```

7️⃣6️⃣ Cases when finally does not execute.

- 🧠 **Short & Crisp:**
  - When System.exit() is called.
  - When the JVM crashes (e.g., power failure, severe Error).
- 📘 **Detailed Explanation:**
  - The finally block is designed to *almost* always run, even after a return or a thrown exception. System.exit() causes the thread to halt abruptly, bypassing the finally block.
- 💻 **Example:**

```
Java
try {
   System.exit(0); // Exits the JVM
} finally {
   System.out.println("Won't print"); // Skipped
}
```

7️⃣7️⃣ Why should we not handle Error?

- 🧠 **Short & Crisp:**
  - Error indicates **serious system failures** (like OutOfMemoryError), which are **unrecoverable** programmatically.
- 📘 **Detailed Explanation:**
  - Errors are often caused by external issues or JVM limitations. Catching them suggests the program can recover, which is rarely true, and may hide the underlying problem.

7️⃣8️⃣ Try-with-resources.

- 🧠 **Short & Crisp:**
  - Introduced in **Java 7**, it automatically closes **AutoCloseable** resources (files, streams) without an explicit finally block.
- 📘 **Detailed Explanation:**
  - Resources declared in the parentheses of the try statement are guaranteed to be closed, even if exceptions occur.
- 💻 **Example:**

```
Java
import java.io.FileReader;
import java.io.IOException;
```

```
try(FileReader f = new FileReader("file.txt")) { // Resource managed here
    // use file
} catch(IOException e) {
    // Handle exception
}
// No need for finally block to close 'f'
```

7️⃣9️⃣ What is hashCode() and its significance? (revisited)

- 🧠 **Short & Crisp:**
  - It provides a **unique integer** for each object to optimize performance in **hash-based collections**.
- 📘 **Detailed Explanation:**
  - **Contract:** If two objects are equal (via equals()), they **must** have the same hashCode(). This is crucial for collections like HashMap and HashSet to function correctly.

8️⃣0️⃣ Default behavior of equals() in Java.

- 🧠 **Short & Crisp:**
  - By default, equals() in the Object class compares **memory addresses** (reference equality), not the objects' values.
- 📘 **Detailed Explanation:**
  - The default equals() method behaves exactly the same as the == operator. Classes like String and wrapper classes override it for **content comparison**.
- 💻 **Example:**

```
Java
Object o1 = new Object();
Object o2 = new Object();
System.out.println(o1.equals(o2)); // false (different memory addresses)
```

🧩 Topic 12 – equals()–hashCode(), HAS-A Relationship, Coupling (Q81–90)

8️⃣1️⃣ Default behavior of the equals() method in Java

- 🧠 **Short & Crisp:**
  - The default equals() in Object checks **reference equality** (memory address), not content.
- 📘 **Detailed Explanation:**
  - You must override equals() if you need to compare objects based on their **internal state** (values of fields) rather than their memory location.
- 💻 **Example:**

Java
```
class Demo { }
Demo d1 = new Demo();
Demo d2 = new Demo();
System.out.println(d1.equals(d2)); // false
```

82 Why does equals() behave differently in the String class?

- 🧠 **Short & Crisp:**
  - Because String **overrides** the default equals() method to compare the **sequence of characters** (content), not memory addresses.
- 📘 **Detailed Explanation:**
  - This is fundamental to how strings are used in Java, ensuring that two strings with the same text are considered equal, even if they are stored in different memory locations.
- 💻 **Example:**

Java
```
String s1 = new String("Java");
String s2 = new String("Java");
System.out.println(s1.equals(s2)); // true (content is the same)
```

83 What is the contract between equals() and hashCode()?

- 🧠 **Short & Crisp:**
  - **Rule 1:** If two objects are equal (equals() is true), they **must** have the same hashCode().
  - **Rule 2:** If two objects have the same hashCode(), they are **not guaranteed** to be equal (hash collision).
- 📘 **Detailed Explanation:**
  - Violating this contract breaks the behavior of hash-based collections (HashMap, HashSet), potentially leading to objects not being found when expected.

84 Define the contracts of equals() (Reflexivity, Symmetry, etc.)

- 🧠 **Short & Crisp:**
  - The equals() method must follow five principles to be reliable.
- 📘 **Detailed Explanation:**

Contract	Meaning
Reflexivity	a.equals(a) must be true.

Symmetry	If a.equals(b) is true, then b.equals(a) must also be true.
Transitivity	If a.equals(b) and b.equals(c) are true, then a.equals(c) must also be true.
Consistency	Multiple calls on the same objects must return the same result.
Non-nullity	a.equals(null) must always be false.

8️⃣5️⃣ Why override equals() and hashCode() together?

- 🧠 **Short & Crisp:**
  - Because failing to override **both** breaks the **HashMap/HashSet contract** — an equal object may produce a different hash code, making it undetectable in a collection.
- 📘 **Detailed Explanation:**
  - HashSet first checks hashCode(), and if they match, it then calls equals(). If you only override equals(), two logically equal objects might have different default hash codes and end up in different buckets in the hash structure.

8️⃣6️⃣ Best practices when overriding equals() and hashCode()

- 🧠 **Short & Crisp:**
  - Use the **same fields** in both methods and always maintain **consistency**.
- 📘 **Detailed Explanation:**
  - Use fields that are part of the object's logical state.
  - Use the Objects.hash() utility method (Java 7+) for safer, cleaner hash code generation.
  - Always include the @Override annotation.

8️⃣7️⃣ What is the HAS-A relationship in Java? Can you equate it to dependency injection?

- 🧠 **Short & Crisp:**
  - **HAS-A** means one class **contains a reference** to another class (Composition or Aggregation) — yes, Dependency Injection (DI) is a way of managing this relationship.
- 📘 **Detailed Explanation:**
  - **HAS-A** defines a structural relationship, often called **Composition**.

    - **DI** is a pattern where the contained object (the dependency) is provided **externally** (injected via constructor/setter) rather than created internally, making the relationship looser.
- 💻 **Example (HAS-A):**

Java
class Engine {}
class Car {
  Engine e = new Engine(); // Car HAS-A Engine
}

8️⃣8️⃣ Difference between reference and object in Java

- 🧠 **Short & Crisp:**
    - An **object** is a real instance taking up memory on the **heap**; a **reference** is a variable (stored on the stack) that **points** to that object.
- 📘 **Detailed Explanation:**
    - You can have multiple references pointing to the same object, but a reference can only point to one object at a time.
- 💻 **Example:**

Java
Car c1 = new Car();
// c1 → reference, new Car() → object
Car c2 = c1; // c2 is a second reference pointing to the SAME object

8️⃣9️⃣ Composition vs Aggregation

- 🧠 **Short & Crisp:**

Type	**Lifespan**	**Dependency**	**Relationship**
Composition	**Strong** (Child dies with parent)	**Dependent** (Part of a whole)	"A **is composed of** B"
Aggregation	**Weak** (Child can exist independently)	**Independent** (Uses a whole)	"A **uses** B"

- 📘 **Detailed Explanation:**

- **Composition:** A Human **is composed of** a Heart. The Heart cannot exist without the Human.
- **Aggregation:** A Teacher **uses** a Department. The Department continues to exist even if the Teacher leaves.

9️⃣0️⃣ Loose coupling vs Tight coupling

- 🧠 **Short & Crisp:**
  - **Loose coupling:** Classes are **independent** and communicate via **interfaces** (flexible, good).
  - **Tight coupling:** One class is **highly dependent** on another's specific implementation (inflexible, bad).
- 📘 **Detailed Explanation:**
  - **Loose coupling** improves testability, maintainability, and scalability. It's the goal of good OOP design.
- 💻 **Example (Loose Coupling):**

Java
```
// Loose
interface Notifier { void notify(); }
class EmailNotifier implements Notifier { public void notify(){} }
class User {
   Notifier n;
   User(Notifier n) { this.n = n; } // Dependency is the interface, not the concrete class
}
```