🧠 **Java OOP Core Notes — Structured Summary**

---

1. Object Class vs Object Creation

🔸 **Object Class**

- Root of all Java classes (java.lang.Object).
- Every class **implicitly extends** Object.
- Common methods inherited by all:
  - toString(), equals(), hashCode(), getClass(), clone(), wait(), notify().
- Example:
- class Student extends Object { } // implied automatically

🔸 **Object Creation**

- Done using new keyword.
- Student s = new Student();
- Creates an **instance** (object) in heap memory.

🔸 **Connection Between Both**

- Whenever an object is created → it automatically inherits Object class methods.

🔸 **Difference**

Concept	Meaning
Object class	Blueprint giving base behavior to all classes.
Object creation	Actual instance creation for use in program.

---

2. Abstract Class

- Declared using **abstract** keyword.
- Cannot be instantiated directly.
- Used for **inheritance + partial implementation**.

Rules

1. Child class **must implement** all abstract methods.
2. If not implemented → child class itself **must be abstract**.
3. Abstract methods → **only declaration**, no body.
4. Abstract class **can have both** abstract and concrete methods.
5. Abstract methods **cannot be final** (since final = non-overridable).

Example

```
abstract class Animal {
   abstract void sound();  // no body
   void sleep() {
      System.out.println("Sleeping...");
   }
}

class Dog extends Animal {
   void sound() {
      System.out.println("Bark");
   }
}
```

🧱 3. Concrete Class

- A **normal class** with complete implementation.
- Can **create objects**.
- No abstract methods.

⚙️ 4. Interface

- **100% abstract (till Java 8)**.
- **After Java 8:** Interfaces became "partially abstract" — they can contain both abstract and concrete (default/static) methods.
- Supports **multiple inheritance** using implements.
- A class can implement **multiple interfaces**.

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

interface B { void display(); }


class Demo implements A, B {

   public void show() { }

   public void display() { }

}
```

⛔ 5. Final Keyword

- final class → cannot be inherited.
- final method → cannot be overridden.
- final variable → cannot be reassigned.
- So abstract method cannot be **final.**
- You can mark method parameters as final to prevent accidental changes inside method.

Blank Final Variables

- Declared but **not initialized** immediately.
- Must be initialized **once** in constructor or static block.
- Used for **runtime constants** — known only during object creation.

```
class Student {
   final int rollNo;
   Student(int roll) {
      rollNo = roll; // assigned once
   }
}
```

🔗 6. Constructor Chaining in Inheritance

🔸 Concept

When a child object is created → **parent constructor runs first**.

🔸 Rules

1. If no constructor in parent → compiler provides a default super().
2. If parent defines a parameterized constructor →
   Child **must call it explicitly** using super(args).

```
class Parent {
   Parent() { System.out.println("Parent"); }
}

class Child extends Parent {
```

```
    Child() {
        super();
        System.out.println("Child");
    }
}
```

Output:

Parent

Child

Case	Parent has constructor?	Child calls `super()`?	Output	Notes
1	✅ Yes	✅ Yes	Parent → Child	Explicit super
2	❌ No	❌ No	Child	Default parent constructor
3	✅ Yes	❌ No	Parent → Child	Implicit super added
4	❌ No	✅ Yes	Child	Default parent constructor (empty)

🧬 7. Inheritance Overview

- Enables **reusability** and **hierarchy**.
- Types:
    - Single
    - Multilevel
    - Hierarchical
- Java **does not support multiple inheritance with classes** (only via interfaces).

👉 **Interfaces** in Java **avoid the diamond problem** (unlike multiple inheritance in C++) because **you must explicitly resolve** any method ambiguity using

ParentInterfaceName.super.methodName();

🧭 **Why It's Safe:**

- Interfaces only allow **default methods**, not actual state (no fields that can conflict).
- If multiple interfaces define the **same default method**, Java **forces** you to choose one — it **won't guess**.
- That's why there's **no ambiguity or diamond problem** — you decide which path to take.

Polymorphism

Compile time polymorphism

Run time polymorphism

Java does NOT support Operator Overloading

➡️ **Main reason:**
👉 **To keep the language simple, clean, and less confusing for developers.**

**In languages like C++, you can redefine how +, -, * behave for your own classes —
but it often makes the code hard to read and error-prone because the same symbol can mean different things in different contexts.**

"

✅ **So how Java achieves the same thing?**

Through Method Overloading — using functions with the same name but different parameters.

That means instead of writing:

c3 = c1 + c2; // ❌ not allowed

We can write:

c3 = c1.add(c2); // ✅ clean, clear, and understandable

```java
class Complex {
    double real, imag;

    // Constructor
    Complex(double r, double i) {
        real = r;
        imag = i;
    }

    // Method to add two complex numbers
    Complex add(Complex c) {
        return new Complex(this.real + c.real, this.imag + c.imag);
    }

    // Overloaded add() - add real number only
    Complex add(double r) {
        return new Complex(this.real + r, this.imag);
    }

    // Display method
    void display() {
        System.out.println(real + " + " + imag + "i");
    }
}

public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(2.3, 4.5);
        Complex c2 = new Complex(3.4, 1.5);

        Complex c3 = c1.add(c2);  // adds two complex numbers
        Complex c4 = c1.add(2.0); // adds real number to complex number

        System.out.print("c3 = ");
        c3.display();
        System.out.print("c4 = ");
        c4.display();
    }
}
```

8. super Keyword

- **Uses**
  1. Access parent **variables**.
  2. Call parent **methods**.
  3. Invoke parent **constructors**.
- **Rules**
  - Must be **first line** in child constructor.
  - If not written, compiler adds super() automatically.
  - If parent has no no-arg constructor → compilation error.

```
class Parent {
   Parent() { System.out.println("Parent constructor"); }
}

class Child extends Parent {
   Child() {
      super();  // added automatically
      System.out.println("Child constructor");
   }
}
```

---

9. Method Overriding vs Variable Hiding

```
class Sample {
   int a = 10;
   void show() { System.out.println("Sample method"); }
}

class Main extends Sample {
   int a = 20;
   void show() { System.out.println("Main method"); }
}
```

🔸 **For Constructors**

```
Sample s = new Main();
Main m = new Main();
```

Output:

```
Sample constructor
Main constructor
Sample constructor
Main constructor
```

🔸 **For Methods (Overridden)**

- Decided by **runtime object (right side)**.

```
s.show();  // Main method
m.show();  // Main method
```

🔸 **For Variables (Hidden)**

- Decided by **reference type (left side)**.

```
System.out.println(s.a);  // 10 (Sample)
System.out.println(m.a);  // 20 (Main)
```

🎯 10. Key Points for MCQs

1. Abstract class can exist **without abstract methods**.
2. Abstract methods → must be implemented by subclass.

3. Abstract classes/interfaces → **cannot be instantiated**.
4. Concrete class → fully implemented, object can be created.
5. No multiple inheritance for **classes** (only interfaces).
6. super() → must be the first line in child constructor.

💡 **Golden Rule**

If both operands are int, division is integer.
Otherwise → automatically double.

System.out.println(5 / 2); // 2

System.out.println(5 / 2.0); // 2.5

🧩 **Null Matching (Concept Question)**

When null can match multiple reference types, **Java prefers the most specific type**, e.g.:

void m(Object o) {}

void m(String s) {}

m(null); // calls m(String) — more specific

☕ JAVA COMPONENTS — JDK, JRE, JVM

🧰 JDK (Java Development Kit)

→ The Developer's Toolbox

- **It has everything a developer needs to create, compile, and run Java programs.**
- **It includes:**
  - ✅ **Compiler (javac) – Converts .java → .class (bytecode)**
  - ✅ **Development Tools – For debugging, packaging, and documentation**
  - ✅ **JRE – To run the program**

In short:

JDK = JRE + Compiler + Developer Tools

Purpose: Used for developing and testing Java programs.

⚙️ JRE (Java Runtime Environment)

→ The Runner

- **It provides the environment to run Java programs.**
- **Contains:**
  - ✅ **JVM – To execute bytecode**
  - ✅ **Libraries – Core Java classes (like java.lang, java.io, java.util, etc.)**
  - ✅ **Runtime Files – Configuration and resources**

In short:

JRE = JVM + Libraries

Purpose: Used for running Java programs, not for developing them.

🧩 JVM (Java Virtual Machine)

→ The Heart of Java's Platform Independence

- **It is the engine that actually runs the Java bytecode.**
- **It loads .class files, verifies them, and executes them.**
- **Uses Just-In-Time (JIT) compiler to convert bytecode → machine code (CPU-understandable).**
- **Handles:**
  - ✅ **Memory management**
  - ✅ **Garbage collection**
  - ✅ **Security checks**
  - ✅ **Runtime optimization**

In short:

JVM = Executes Bytecode + Manages Memory + Ensures Security

Purpose: Enables Java's "Write Once, Run Anywhere" feature.

Step 1: Write code → Hello.java

Step 2: Compile using javac (JDK) → Hello.class (bytecode)

Step 3: Run using JVM (inside JRE)

Step 4: JVM → converts bytecode → machine code → executes

Essential Java Commands

Command	Description
javac filename.java	Compiles .java → .class (bytecode)
java classname	Runs the compiled bytecode (.class file)

🧠 Concept of **this** Keyword in Java

this keyword Enables clean and consistent code.

The main purpose of the this keyword is **to remove confusion between instance variables and local variables** when both have the same name, using same name improves readability and consistency in code

In Java, if the compiler finds both a **local variable** and an **instance variable** with the same name,
👉 it gives *priority* to the local variable.

So, when you write something like this 👇

```
class Human {
int age;

public void set(int age) {
age = age;   // ❌ looks right, but it's wrong
}
}
```

Here:

- The first age (on left side of =) → local variable (method parameter).
- The second age (on right side) → also local variable.

That means this line **just reassigns the local variable to itself**,
and the **instance variable of the class remains unchanged**.

Correct Way (using this)

To make sure the instance variable is assigned properly 👇

```
class Human {
   int age;

   public void set(int age) {
      this.age = age;   // ✅ now instance variable = local variable
   }
}
```

Here this.age clearly means **"the age belonging to the current object"**, this pointer refers to current object. (current instance of class)
and age on the right is the local variable (method parameter).

💡 **Why this was introduced**

Because without it, compiler behavior could be smisleading —
it always prioritizes **local variables over instance variables**,
which causes unexpected results while assigning values.

So, the this keyword tells Java:
👉 “Hey! Use the **current object’s variable**, not the local one.”

Static

static means **“belongs to the class, not to any object.”**
When you declare something as static,
it is **shared among all objects** of that class instead of each object having its own copy.

Keyword	Meaning
static variable	Shared among all objects, initialized once
static method	Belongs to class, can be called without object
static block	Runs once when class loads
static import	Allows direct access to static members without prefixing class name

☕ Why Strings in Java Are Immutable

💡 1️⃣ Security Reasons

Strings are widely used for **sensitive data** — like:

String url = "jdbc:mysql://localhost:3306/db";

String username = "root";

String password = "1234";

If Strings were mutable, a malicious code could alter their value after they are created — changing file paths, DB URLs, or credentials in memory.

✅ **Immutability ensures once created → can't be tampered with.**
Java's security features (like class loading and reflection) heavily rely on this guarantee.

⚙️ 2️⃣ String Pool (Memory Optimization)

Java maintains a **String Constant Pool (SCP)** in heap memory.

Example:

String a = "Hello";

String b = "Hello";

👉 Both a and b point to the **same** object in the pool.
If strings were mutable and you did a += "World", it would also change b — which is *unacceptable*.

✅ So immutability makes pooling safe and memory-efficient — multiple references can share the same object without risk

Interface	Implementation	Example
List	ArrayList, LinkedList	Ordered, allows duplicates
Set	HashSet, TreeSet	Unique elements
Map	HashMap, TreeMap	Key–value pairs