🔥 1. INTRODUCTION TO OOPS - INTERVIEW Q&A, MCQs, CODING

🎯 THEORY QUESTIONS (with Interview-level Answers)

Q1. What is Object-Oriented Programming (OOP)?
A: OOP is a programming paradigm based on the concept of "objects" which contain data (fields) and methods (functions). It focuses on real-world modeling and includes principles like Encapsulation, Inheritance, Polymorphism, and Abstraction.
Q2. What are the 4 pillars of OOP?
A:

Encapsulation
Inheritance
Polymorphism
Abstraction

Q3. Why do we need OOP when procedural programming exists?
A: OOP allows for better code organization, reuse, maintainability, and modeling of complex systems as objects. Procedural programs become harder to scale and maintain over time due to tightly coupled logic.
Q4. Is Java a pure OOP language?
A: No. Java supports primitive types like int, char, etc., which are not objects, so it is not 100% pure. Languages like Smalltalk are considered pure OOP.
Q5. Can we implement OOP in C?
A: C is not an object-oriented language, but you can simulate OOP concepts like encapsulation using structs and function pointers. Full OOP is not supported natively.

🧠 TECHNICAL INTERVIEW QUESTIONS

Q6. Explain how OOP helps in real-life system design.
A: OOP maps real-world entities into code using classes. For example, in a banking system: Account, Customer, Transaction can be modeled as classes with relationships (inheritance/composition) and behaviors (methods).
Q7. What are the advantages of OOP?
A:

- Code Reusability (via Inheritance)
- Easy maintenance (via Encapsulation)
- Polymorphism makes code flexible and extensible
- Abstraction hides complex logic
- Real-world modeling

Q8. What are disadvantages of OOP?
A:

- Increased complexity
- Larger memory overhead
- Steeper learning curve
- Not ideal for small/straightforward scripts

✅ MCQs on OOP Introduction

Which of the following is not a pillar of OOP?
a) Encapsulation
b) Compilation ✅

c) Polymorphism
d) Inheritance
OOP helps in:
a) Faster execution
b) Better code organization ✅
c) Saving power
d) None of the above
Which programming language is a pure OOP language?
a) Java
b) Python
c) Smalltalk ✅
d) C++

👨‍💻 **CODING-BASED OOP UNDERSTANDING QUESTION**
Q. Create a simple OOP program that models a Car with basic behavior.

```
java
Copy code
class Car {
   String brand;
   int speed;
void start() {
      System.out.println(brand + " is starting...");
   }
void accelerate(int increment) {
      speed += increment;
      System.out.println("Speed increased to: " + speed);
   }
void display() {
      System.out.println("Brand: " + brand + ", Speed: " + speed);
   }
}
public class Main {
   public static void main(String[] args) {
      Car c = new Car();
      c.brand = "Honda";
      c.speed = 0;
      c.start();
      c.accelerate(30);
      c.display();
   }
}
```

Concepts tested: Class, Object, Method, State & Behavior — all core OOP.

==========

✅ 2. CLASS AND OBJECT – FULL INTERVIEW Q&A, MCQs, CODING

📘 THEORY QUESTIONS (WITH INTERVIEW-LEVEL ANSWERS)

Q1. What is a class in Java?
A:
A class is a blueprint or template for creating objects. It defines variables (fields) and methods (functions) that describe the behavior and state of the object.

Q2. What is an object in Java?
A:
An object is an instance of a class. It has its own memory and contains real values for the fields defined in the class.

Q3. How are class and object related?
A:
A class is like a design plan or prototype, while the object is the actual product created from that plan. You can create multiple objects from a single class.

Q4. How is memory allocated to objects in Java?
A:
Memory for objects is allocated on the **heap** at runtime using the new keyword.

Q5. What happens when you don't initialize object fields?
A:
Java initializes object fields with **default values**:

- int → 0
- boolean → false
- Object → null
- float/double → 0.0

Q6. Can you create an object without a class?
A:
No. In Java, you must have a class definition to create an object.

Q7. Can a class be empty?
A:
Yes. You can declare a class with no variables or methods. It's valid and sometimes used as a marker class.

Q8. Can you create objects without using the new keyword?
A:
Yes, in specific scenarios:

- Using cloning: Object.clone()
- Using reflection: Class.forName().newInstance()
- Deserialization

- Factory design pattern

Q9. What is this keyword in Java?
A:
this refers to the **current object** of the class. It is used to:
- Refer to current class variables
- Call current class methods
- Return current object from a method

Q10. Can a class have more than one object?
A:
Yes. You can create multiple objects (instances) from the same class, each with its own state.

🧠 TECHNICAL INTERVIEW QUESTIONS

Q11. Difference between class and structure in Java?
A:
Java doesn't have structures like C/C++. Everything is based on classes. In C/C++, structs only contain data; in Java, classes contain both data and behavior.

Q12. Can you instantiate an abstract class?
A:
No. Abstract classes cannot be directly instantiated. You must extend them and implement the abstract methods.

Q13. How do you create a class that cannot be extended?
A:
By marking the class as final:

java
Copy code
final class MyClass { }

Q14. What is object slicing?
A:
Object slicing happens when a subclass object is assigned to a superclass reference and the subclass-specific members are lost. Java doesn't have slicing like C++ due to its reference-based object model.

Q15. What happens if you try to access a member using a null object?
A:
You'll get a NullPointerException.

✅ MCQs on CLASS & OBJECT

1. What is the correct way to create an object in Java?
a) Car c = Car();
b) Car c = new Car;
c) Car c = new Car(); ✅
d) Car c();

2. Which of these is not a valid access modifier for a class?
a) public

b) private ✅
c) default
d) abstract

3. An object in Java is created at:
a) Compile time
b) Load time
c) Runtime ✅
d) Declaration time

4. Which keyword is used to refer to the current class object?
a) that
b) current
c) self
d) this ✅

👨‍💻 CODING QUESTIONS ON CLASS & OBJECT

Q1. Create a class Person with name and age fields, and a method to display details.

java
Copy code
```
class Person {
   String name;
   int age;
void display() {
      System.out.println("Name: " + name + ", Age: " + age);
   }
}
public class Main {
   public static void main(String[] args) {
      Person p1 = new Person();
      p1.name = "Alice";
      p1.age = 25;
      p1.display();
   }
}
```

Q2. Write a program to demonstrate multiple objects of a class.

java
Copy code
```
class Book {
   String title;
   double price;
void printDetails() {
      System.out.println(title + " costs ₹" + price);
   }
}
public class Main {
   public static void main(String[] args) {
      Book b1 = new Book();
```

```
        b1.title = "Java";
        b1.price = 399;
Book b2 = new Book();
        b2.title = "Python";
        b2.price = 499;
b1.printDetails();
        b2.printDetails();
    }
}
```

Q3. What will be the output?

```
java
Copy code
class Test {
    int x = 10;
void change(Test t) {
        t.x = 20;
    }
}
public class Main {
    public static void main(String[] args) {
        Test obj = new Test();
        obj.change(obj);
        System.out.println(obj.x);
    }
}
```

Answer:
20
Explanation: Objects are passed by reference in Java. So the change affects the original object.

Q4. Can you create a class without a main method and still run it?
A:
Yes, by calling it from another class with a main() method. Java needs an entry point (public static void main) only to start execution.

==========

COMPLETE OOPS INTERVIEW QUESTIONS AND ANSWERS

==========

This master document covers all possible interview questions, coding problems,
MCQs, and technical round questions for each topic under Object-Oriented Programming (OOP) in Java.
Answers are written as expected by top interviewers (Amazon, Google, Infosys, TCS, etc.).
Everything is formatted Notepad/OneNote-friendly.

TABLE OF CONTENTS

1. Introduction to OOP
2. Class and Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction
7. Constructor (incl. Constructor Overloading)
8. Static Keyword
9. this & super Keyword
10. Method Overloading vs Overriding
11. Final Keyword
12. instanceof Operator
13. Object Class
14. Access Modifiers
15. Aggregation vs Composition
16. Custom Exception Handling (OOP Design Focus)
17. Real-life Design Examples & UML Thinking
18. MCQs Section (Topic-wise)
19. Coding Questions (Topic-wise)
20. Conceptual Technical Interview Questions

NOTE: Each section includes theoretical, MCQ, and coding-based interview Q\&A.

\==========

1. INTRODUCTION TO OOPS
\==========

\-- THEORY QUESTIONS --
Q1: What is Object-Oriented Programming (OOP)?
Ans: OOP is a programming paradigm based on the concept of "objects", which can contain data (fields) and methods (functions). It emphasizes modularity, code reusability, abstraction, and maintainability.

Q2: What are the four pillars of OOP?
Ans:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Q3: What is the purpose of OOP in software development?
Ans: OOP allows for modeling real-world entities, improves maintainability, supports reusability of code, enhances security through encapsulation, and helps in large-scale application development.

Q4: What is the difference between OOP and POP (Procedural Oriented Programming)?
Ans:

* OOP is based on objects; POP is based on procedures.
* OOP promotes data hiding; POP exposes data globally.
* OOP supports inheritance and polymorphism; POP does not.

Q5: What are some real-life examples of OOP?
Ans:

* Car object: Properties - color, model; Behavior - start(), stop()
* Bank account: Properties - balance, accountNumber; Behavior - deposit(), withdraw()

Q6: What are access modifiers in OOP?
Ans: Access modifiers define the scope and visibility of classes, methods, and variables. Types include public, private, protected, and default.

Q7: Can OOP be achieved in C?
Ans: C is not object-oriented, but object-like behavior can be simulated using structures and function pointers. True OOP is best achieved using C++, Java, etc.

Q8: What is the difference between class and structure?
Ans: In Java, there is no structure. In C++, classes support OOP features like access modifiers and inheritance; structures have public access by default.

Q9: How does OOP help with scalability?
Ans: OOP allows teams to build modules independently, reuse code, and extend systems without rewriting existing components.

\-- MCQs --
Q1: Which is not a principle of OOP?
A. Encapsulation
B. Inheritance
C. Compilation
D. Abstraction
Ans: C. Compilation

Q2: Which feature of OOP is used to hide internal implementation?
A. Inheritance
B. Polymorphism
C. Encapsulation
D. Abstraction
Ans: C. Encapsulation

Q3: Which of the following best describes an object?
A. Instance of a method
B. Collection of functions
C. Instance of a class
D. Blueprint of a class
Ans: C. Instance of a class

Q4: OOP is based around:
A. Procedures
B. Functions
C. Objects

D. Variables
Ans: C. Objects

Q5: Which keyword is used to create an object in Java?
A. object
B. new
C. create
D. make
Ans: B. new

\-- CODING QUESTION 1 --
Q: Write a simple Java program demonstrating all 4 OOP pillars.
Ans:
class Animal {
private String name; // Encapsulation

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

public void makeSound() { // Polymorphism - base behavior
   System.out.println("Animal sound");
}
```

}

class Dog extends Animal { // Inheritance
public Dog(String name) {
super(name);
}

```
@Override
public void makeSound() { // Polymorphism - overriding
   System.out.println("Bark");
}
```

}

abstract class Vehicle { // Abstraction
abstract void start();
abstract void stop();
}

class Car extends Vehicle {
void start() {
System.out.println("Car started");
}
void stop() {

System.out.println("Car stopped");
}
}

public class OOPDemo {
public static void main(String\[] args) {
Animal a = new Dog("Tommy");
a.makeSound();

```
   Vehicle v = new Car();
   v.start();
   v.stop();
}
```

}

\-- CODING QUESTION 2 --
Q: Simulate a bank system using objects: Account, Customer, and Bank.
Ans:
class Customer {
private String name;
private String address;

```
public Customer(String name, String address) {
   this.name = name;
   this.address = address;
}

public String getName() { return name; }
public String getAddress() { return address; }
```

}

class Account {
private int accNo;
private double balance;
private Customer customer;

```
public Account(int accNo, double balance, Customer customer) {
   this.accNo = accNo;
   this.balance = balance;
   this.customer = customer;
}

public void deposit(double amount) {
   balance += amount;
}
```

```
public void withdraw(double amount) {
   if (amount <= balance)
      balance -= amount;
   else
      System.out.println("Insufficient balance");
}

public void display() {
   System.out.println("Account No: " + accNo);
   System.out.println("Balance: " + balance);
   System.out.println("Customer: " + customer.getName());
}
```

```
}

public class BankSystem {
public static void main(String\[] args) {
Customer c = new Customer("Alice", "Bangalore");
Account a = new Account(12345, 5000.0, c);
a.deposit(1000);
a.withdraw(700);
a.display();
}
}
```

==========

✅ 2. CLASS AND OBJECT – COMPLETE NOTES

💡 **THEORY QUESTIONS**

Q1: What is a class in Java?

➡️ A class is a **blueprint** or **template** for creating objects. It defines the structure (fields/variables) and behavior (methods/functions) that the objects of the class will have.

Q2: What is an object in Java?

➡️ An object is an **instance** of a class. It holds actual values and can invoke the class's methods.

Q3: How are classes and objects related?

➡️ A class defines the **structure**. Objects are **real-world instances** created from that class.

Q4: How do you create an object in Java?

```
java
CopyEdit
Student s = new Student();
```

Q5: What is object instantiation?

➡️ The process of **creating an object** using the new keyword and allocating memory on the heap.

Q6: Can you create multiple objects of the same class?

➡️ Yes. Each object gets **its own copy** of instance variables.

Q7: What happens in memory when an object is created?

➡️ The **object's data (fields)** are stored in the **heap**, and the reference is stored on the **stack**.

Q8: What are instance variables and methods?

➡️ Variables and methods that belong to a **specific object**. Each object has its own **separate copy**.

Q9: What are class variables and methods?

➡️ Declared with the **static** keyword. They **belong to the class**, not any object, and are **shared** by all objects.

Q10: Difference between object reference and object?

➡️ Object reference is a **pointer** to the memory location. Object is the **actual data** in memory.

Q11: Can instance variables be accessed without an object?

➡️ ❌ No. Only static variables/methods can be accessed without object creation.

Q12: How are objects destroyed in Java?

➡️ Objects are destroyed via **Garbage Collection** when they are no longer reachable. You can suggest for GC using System.gc() but cannot manually destroy an object.

❓ MCQs

Q1: Which of the following creates an object in Java?

A. Student s = Student();
B. Student s = new Student;
C. Student s = new Student(); ✅
D. Object.create(Student);

👉 **Correct:** C

Q2: Where are object instance variables stored?

A. Stack
B. Heap ✅
C. Register
D. Constant Pool

👉 **Correct:** B

Q3: What is the default value of an object reference?

A. 0
B. null ✅
C. undefined
D. empty

👉 **Correct:** B

Q4: Keyword used to access current object?

A. current
B. self
C. this ✅
D. class

👉 **Correct:** C

Q5: Which best defines a class?

A. Collection of methods
B. Blueprint of an object ✅
C. Memory instance
D. Code block

👉 **Correct:** B

💻 CODING QUESTIONS

Q1: Write a class Car with fields and create multiple objects.

java
CopyEdit

```
class Car {
   String make;
   String model;
   int mileage;
void display() {
      System.out.println("Make: " + make);
      System.out.println("Model: " + model);
      System.out.println("Mileage: " + mileage + " kmpl");
   }
}
public class CarTest {
   public static void main(String[] args) {
      Car c1 = new Car();
      c1.make = "Honda";
      c1.model = "City";
      c1.mileage = 18;
Car c2 = new Car();
      c2.make = "Hyundai";
      c2.model = "Verna";
      c2.mileage = 16;
c1.display();
      c2.display();
   }
}
```

Q2: Show how two objects of same class have separate memory.

```
java
CopyEdit
class Counter {
   int count = 0;
void increment() {
      count++;
   }
void display() {
      System.out.println("Count: " + count);
   }
}
public class CounterTest {
   public static void main(String[] args) {
      Counter c1 = new Counter();
      Counter c2 = new Counter();
c1.increment();
      c1.increment();
      c2.increment();
c1.display(); // Output: Count: 2
      c2.display(); // Output: Count: 1
   }
}
```

Q3: Class with private fields & public getters/setters (Encapsulation preview)

```
java
CopyEdit
class Book {
   private String title;
   private String author;
public String getTitle() {
      return title;
   }
public void setTitle(String t) {
      title = t;
   }
public String getAuthor() {
      return author;
   }
public void setAuthor(String a) {
      author = a;
   }
}
public class BookTest {
   public static void main(String[] args) {
      Book b = new Book();
      b.setTitle("Atomic Habits");
      b.setAuthor("James Clear");
System.out.println("Book: " + b.getTitle());
      System.out.println("Author: " + b.getAuthor());
   }
}
```

🧠 **TECHNICAL INTERVIEW-STYLE QUESTIONS**

Q1: What is the significance of the new keyword?

➡️ It allocates **heap memory**, invokes the **constructor**, and returns a reference to the object.

Q2: Explain the difference between stack and heap in context of objects.

➡️ Stack holds the **reference** (like s in Student s = new Student()), while heap stores the **actual object data**.

Q3: Can an object be created without using new?

➡️ ✅ Yes. Using methods like **cloning**, **deserialization**, or **reflection**.

Q4: How does Java know which object to destroy during GC?

➡️ When an object has **no references**, it's considered unreachable and eligible for GC.

Q5: Difference between static and non-static members in a class?

Feature	**Static**	**Non-static**
Belongs to	Class	Object
Memory	Once per class	Per object
Accessed via	Class name or object	Only via object
Example	Math.pow()	c1.display()

==========

🎯 OOPS MASTER INTERVIEW GUIDE — TOPIC: ENCAPSULATION (JAVA)

==========

🧠 THEORY QUESTIONS (With Interview-Expected Answers)

Q1: ❓ What is Encapsulation in Java?
Ans: ✅ Encapsulation is the process of wrapping data (variables) and methods (functions) together into a single unit — typically a class.
It is used to protect the internal state of the object from unauthorized access and modification.

Q2: ❓ Why is Encapsulation important?
Ans: ✅ It ensures:
- Data hiding (prevents external modification),
- Better maintainability,
- Flexibility to change internal code without impacting outside code,
- Improved control over the data.

Q3: ❓ How is Encapsulation implemented in Java?
Ans: ✅ By following 3 steps:
1. Declare variables `private` (restrict direct access)
2. Provide `public getter and setter methods` to read/write those variables
3. Access fields only through these getters/setters.

Q4: ❓ What is the difference between Encapsulation and Abstraction?
Ans: ✅
- Encapsulation: Hides the data/internal state (how data is stored).
- Abstraction: Hides implementation details (how functionalities are implemented).
🔁 Encapsulation = *How to protect data*, Abstraction = *How to expose behavior*.

Q5: ❓ Can a class be encapsulated without getter/setter methods?
Ans: ⚠️ Partially. If no getter/setter is provided, data is fully hidden (no access). But it defeats the purpose if we can't use the data.
We use getter/setters to control how variables are accessed.

Q6: ❓ What if we make fields public? Is it encapsulation?
Ans: ❌ No. Public fields break encapsulation. Anyone can modify them directly.

Q7: ❓ What are the advantages of using getters/setters over direct access?
Ans: ✅
- Add validation before assigning values,
- Restrict access (only get, not set),
- Log access for debugging,
- Change internal implementation later without breaking external code.

Q8: ❓ Is JavaBeans a form of encapsulation?
Ans: ✅ Yes. JavaBeans are classes that follow encapsulation rules: private fields + public getters and setters.

Q9: ❓ Can static variables be encapsulated?
Ans: ✅ Yes. Static fields can also be private and accessed via static getters/setters.

Q10: ❓ Can encapsulation help in making code thread-safe?
Ans: ⚠️ Not directly, but encapsulation can prevent race conditions by restricting direct access to shared data.

💉 MCQs ON ENCAPSULATION

Q1: Which keyword is used to restrict direct access to variables?
A. protected
B. final
C. private
D. static
✅ Ans: C. private

Q2: What is the best access modifier for encapsulation?
A. public
B. protected
C. default
D. private
✅ Ans: D. private

Q3: What is true about encapsulation?
A. It is the process of inheriting methods
B. It increases coupling
C. It hides data
D. It exposes internal structure
✅ Ans: C. It hides data

Q4: Which statement is correct about getter/setter methods?
A. They make fields public
B. They allow indirect access to private variables
C. They reduce security
D. They are private methods
✅ Ans: B. They allow indirect access to private variables

Q5: What is the output of accessing a private variable without a getter?
A. Runtime Error
B. Compilation Error
C. Value gets printed
D. Null
✅ Ans: B. Compilation Error

💻 CODING QUESTIONS (With Answers)

Q1: 🔧 Create a class `Employee` with private fields and public getter/setter methods. Show encapsulation.

```java
class Employee {
   private String name;
   private int salary;

   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }

   public int getSalary() {
      return salary;
   }
   public void setSalary(int salary) {
      if (salary > 0) {
         this.salary = salary;
      } else {
         System.out.println("Invalid Salary");
      }
   }
}

public class TestEmployee {
   public static void main(String[] args) {
      Employee emp = new Employee();
      emp.setName("Alice");
      emp.setSalary(50000);
      System.out.println(emp.getName());      // Alice
      System.out.println(emp.getSalary());    // 50000
   }
}
```

Q2: 🔧 Write a Java class that encapsulates student data and restricts invalid marks (>100 or <0).

java
Copy
Edit

```java
class Student {
   private String name;
   private int marks;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getMarks() {
      return marks;
```

```
    }

    public void setMarks(int marks) {
        if (marks >= 0 && marks <= 100) {
            this.marks = marks;
        } else {
            System.out.println("Invalid marks!");
        }
    }
}
```

Q3: 🔧 Create a class with only getter, no setter. What does it represent?

```
java
Copy
Edit
class FinalGrade {
    private String grade = "A+";

    public String getGrade() {
        return grade;
    }
}

// Read-only property. Cannot modify from outside (Immutable behavior).
```

Q4: 🔧 Encapsulate a bank account class where balance can't be negative and withdrawal is controlled.

```
java
Copy
Edit
class BankAccount {
    private double balance;

    public void deposit(double amount) {
        if (amount > 0) balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= balance) balance -= amount;
        else System.out.println("Insufficient funds.");
    }

    public double getBalance() {
        return balance;
    }
}
```

⚙️ SCENARIO / TECHNICAL INTERVIEW QUESTIONS

Q1: ❓ Can you enforce read-only or write-only encapsulation in Java?

Ans: ✅ Yes.

Read-only: Provide only getter

Write-only: Provide only setter

Q2: ❓ What if multiple threads access your object's data?
Ans: ✅ Encapsulation prevents direct access, but to ensure thread safety, use synchronized keyword on setters/getters or use Atomic classes.

Q3: ❓ Why is it a bad idea to use public fields in POJOs?
Ans: ✅ Public fields:

Break encapsulation

Allow unwanted modifications

Cannot control or validate data

Difficult to track changes or enforce security rules

Q4: ❓ Give an example where setter validation prevents logical errors.
Ans: ✅ For example, age of a person should not be negative. Validation in setter avoids bugs.

Q5: ❓ Can encapsulation be violated in any way in Java?
Ans: ⚠️ Reflection API can be used to access private fields, violating encapsulation. But it's generally discouraged unless for debugging.

📌 INTERVIEW TIP:
Encapsulation is not just about writing private fields and public methods — it's about thinking in terms of safety, validation, access control, and design clarity. Interviewers love when you explain both why and how.
==========

📘 4. INHERITANCE — MASTER NOTES
==========

🎓 THEORY QUESTIONS & ANSWERS

Q1: What is Inheritance in Java?
Ans: Inheritance is an OOP principle where one class (child/subclass) acquires the properties and behaviors (fields and methods) of another class (parent/superclass).
It promotes **code reusability** and **method overriding**.

Q2: What are the different types of inheritance in Java?
Ans:

Type	Description	Supported in Java?
Single Inheritance	One child inherits from one parent	✅ Yes
Multilevel	Child inherits from parent, grandchild from child	✅ Yes
Hierarchical	Multiple children from the same parent	✅ Yes
Multiple Inheritance (by class)	One class inherits from multiple classes	❌ Not supported directly
Hybrid	Combination of two or more types	✅ Supported via interface

Q3: Why does Java not support multiple inheritance with classes?
Ans: To avoid **ambiguity** caused by the **Diamond Problem** — where two superclasses have the same method and a subclass can't resolve which one to use. Java solves this using **interfaces**.

Q4: How is inheritance implemented in Java?
Ans: Using the extends keyword:

java
Copy code
class Parent { }
class Child extends Parent { }

Q5: What is the role of the super keyword?
Ans: super is used to:
Access parent class methods/constructors
Call parent class constructors using super()
Access parent class variables shadowed by child class

Q6: What is method overriding in inheritance?
Ans: When a child class provides its own version of a method already defined in the parent class.
Rules:
- Same method signature
- Access modifier should not be more restrictive
- Cannot override final, static, or private methods

Q7: Can constructors be inherited?
Ans: No. Constructors are **not inherited**. But the **child class must call** the constructor of the parent using super().

Q8: Can you inherit private members?
Ans: Private members are not accessible in the child class directly but still **exist** in the memory.

Q9: Is Java 100% single inheritance by class?
Ans: Yes, Java supports **only single inheritance** by class, but allows **multiple inheritance using interfaces**.

Q10: What is IS-A relationship?
Ans: IS-A is an inheritance relationship. For example, Dog IS-A Animal. It implies the child class is a type of the parent class.

❓ INTERVIEW QUESTIONS

Q1: What happens if the child class doesn't call super() explicitly in constructor?
Ans: Java inserts an implicit call to the **no-arg constructor** of the parent class. If the parent doesn't have it, compilation fails.

Q2: Can a class extend multiple interfaces and a class at the same time?
Ans: Yes. Java allows:

java
Copy code
class A extends B implements I1, I2 { }

Q3: What is object slicing in inheritance?
Ans: In C++-style languages, object slicing is when a subclass object is assigned to a superclass variable, and subclass-specific fields are "sliced" off. Java avoids this by always working with **references**, not copies.

Q4: When do we use inheritance in real-world design?
Ans: When multiple classes share common features but also need specialization. Example: Vehicle -> Car, Bike, Truck.

✅ MCQs with EXPLANATIONS

Q1: What keyword is used to inherit a class?
A. implement
B. super
C. extends
D. this
Ans: ✅ C. extends

Q2: Which type of inheritance is not supported in Java classes directly?
A. Single
B. Multilevel
C. Multiple
D. Hierarchical
Ans: ✅ C. Multiple

Q3: Which of the following is true about method overriding?
A. Return type must differ
B. Parameters must differ
C. Method must be private
D. Method must have same name and signature
Ans: ✅ D. Method must have same name and signature

Q4: Which keyword is used to access the parent class methods or variables?
A. this
B. super
C. parent
D. base
Ans: ✅ B. super

Q5: What is the relation between Dog and Animal if Dog extends Animal?
A. HAS-A
B. USES-A
C. IS-A
D. HOLDS-A
Ans: ✅ C. IS-A

💻 CODING QUESTIONS

Q1. Simple Inheritance Example

java

Copy code
```
class Animal {
   void sound() {
      System.out.println("Animal makes sound");
   }
}
class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks");
   }
}
public class Test {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.sound();  // inherited
      d.bark();   // own
   }
}
```

Q2. Multilevel Inheritance

java
Copy code
```
class Animal {
   void eat() {
      System.out.println("eating...");
   }
}
class Dog extends Animal {
   void bark() {
      System.out.println("barking...");
   }
}
class Puppy extends Dog {
   void weep() {
      System.out.println("weeping...");
   }
}
public class Test {
   public static void main(String[] args) {
      Puppy p = new Puppy();
      p.eat();
      p.bark();
      p.weep();
   }
}
```

Q3. Demonstrate method overriding

java
Copy code
```
class Vehicle {
```

```
    void run() {
        System.out.println("Vehicle is running");
    }
}
class Bike extends Vehicle {
    void run() {
        System.out.println("Bike is running fast");
    }
}
public class Test {
    public static void main(String[] args) {
        Vehicle v = new Bike(); // runtime polymorphism
        v.run();
    }
}
```

Q4. Using super keyword

```
java
Copy code
class Parent {
    String name = "Parent";
    void greet() {
        System.out.println("Hello from parent");
    }
}
class Child extends Parent {
    String name = "Child";
    void display() {
        System.out.println(super.name);  // access parent variable
        super.greet();                   // call parent method
    }
}
public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
    }
}
```

🧠 **TECHNICAL INTERVIEW–STYLE QUESTIONS**

Q1: How does inheritance affect memory and performance?
Ans: Inheritance doesn't duplicate code in memory for every object. Parent class methods are stored once in class metadata; reused by child instances → leads to efficient memory use. But too deep inheritance chains may reduce performance/readability.

Q2: How would you design a university hierarchy using inheritance?
Ans:

```
text
Copy code
```

Person (name, age)
↳ Student (rollNo, course)
↳ Teacher (subject, salary)
↳ Staff (role, department)

Q3: Explain how Java solves the diamond problem with interfaces.
Ans: Java allows a class to implement multiple interfaces. If two interfaces have same default method, class must override it explicitly.

java
Copy code
```
interface A {
   default void show() { System.out.println("A"); }
}
interface B {
   default void show() { System.out.println("B"); }
}
class C implements A, B {
   public void show() {
      A.super.show();  // choose explicitly
   }
}
```

✅ **INHERITANCE – All Possible Questions with Answers**
(Java OOP Interview Ready – Notepad/OneNote Friendly)

📘 **THEORY QUESTIONS (With Answers)**
Q1. What is Inheritance in Java?
Ans: Inheritance is an OOP concept that allows a class (child/subclass) to inherit fields and methods from another class (parent/superclass).
It promotes code reusability and method overriding.

Q2. What are the different types of inheritance in Java?
Ans:
- **Single Inheritance:** One class inherits from one superclass.
- **Multilevel Inheritance:** A class inherits from a class which itself inherits from another class.
- **Hierarchical Inheritance:** Multiple classes inherit from one superclass.
- **Multiple Inheritance (via Interfaces only):** Java doesn't support multiple inheritance with classes directly (to avoid ambiguity), but can be done using interfaces.

Q3. Why doesn't Java support multiple inheritance with classes?
Ans: To avoid **diamond problem** (ambiguity when two parent classes have the same method name). Java solves it using **interfaces** and **default methods** with clear resolution.

Q4. What is the syntax of inheritance in Java?

java
Copy code
```
class Parent {
  // fields and methods
}
class Child extends Parent {
```

```
  // inherited + new fields/methods
}
```

Q5. What is the use of the super keyword?
Ans:
- Refers to the **parent class**.
- Used to access parent class methods or constructor.
- Used to **call superclass constructor** from subclass constructor.

Q6. Can constructors be inherited?
Ans: No. Constructors are **not inherited**, but subclass can call them using super().

Q7. Can private members be inherited?
Ans: Yes, but **they are not accessible** in the subclass directly.
Access must be through **public/protected methods** of superclass.

Q8. What is method overriding?
Ans: When a subclass **redefines** a method of the superclass with **same name, return type, and parameters**.

Q9. What is dynamic method dispatch?
Ans: Process where a call to an overridden method is resolved at **runtime**, based on the object being referred to.

Q10. Can a class extend multiple classes in Java?
Ans: No, Java doesn't support multiple inheritance through classes.
Use **interfaces** for that purpose.

Q11. Can you override static methods?
Ans: No. Static methods belong to class, not instances.
They are **hidden**, not overridden.

Q12. Can we change the return type of an overridden method?
Ans: Yes, but only to a **covariant type** (a subclass of the original return type).

Q13. Is it mandatory to override all methods of the parent class?
Ans: No. Only if the parent class is abstract and the method is abstract.

Q14. What happens if a method is final in the parent class?
Ans: It **cannot be overridden** in the subclass.

Q15. What is object slicing?
Ans: In some languages (like C++), assigning a derived object to base object may "slice off" extra fields.
In Java, object slicing doesn't happen due to reference-based nature.

✅ **DEEP TECHNICAL INTERVIEW QUESTIONS**
Q16. How does Java internally resolve method calls in inheritance?
Ans: Java uses **dynamic dispatch** and **method tables (v-tables)**. At runtime, the JVM decides which method to invoke based on actual object.

Q17. How are access specifiers (private/protected/public/default) handled in inheritance?
Ans:
- **private** → not accessible in subclass

- **protected** → accessible in same package or through inheritance
- **public** → accessible everywhere
- **default** → package-level access only

Q18. Can abstract class have constructors?
Ans: Yes. Abstract classes can have constructors which are called when subclass objects are created.

Q19. Difference between "has-a" and "is-a" relationship?
Ans:
- **is-a** → Inheritance (Dog is-a Animal)
- **has-a** → Composition/Aggregation (Car has-a Engine)

Q20. What is the role of instanceof operator in inheritance?
Ans: To check whether an object is an instance of a subclass/superclass.

java
Copy code
if(obj instanceof Animal) { ... }

Q21. What are rules of method overriding?
- Must have same name and parameters.
- Return type must be same or covariant.
- Cannot override final, static, or private methods.
- Overriding method cannot reduce visibility.

💡 **MCQ QUESTIONS (With Answers)**
Q1. Which keyword is used to inherit a class in Java?
A. implements
B. extends
C. super
D. this
✅ Ans: **B. extends**

Q2. Which of the following is not a type of inheritance in Java?
A. Single
B. Multilevel
C. Multiple (through classes)
D. Hybrid
✅ Ans: **D. Hybrid** (Java does not support hybrid inheritance explicitly)

Q3. Which method can be overridden?
A. final method
B. static method
C. private method
D. public method
✅ Ans: **D. public method**

Q4. What is the default behavior if no constructor is defined in subclass?
A. It won't compile
B. Calls default constructor of superclass
C. No constructor is inherited
D. Object class constructor is called

✅ Ans: **B. Calls default constructor of superclass**

Q5. What happens when method with same signature exists in both parent and child?
A. Compile-time error
B. Parent method is called
C. Child method overrides parent
D. JVM exits
✅ Ans: **C. Child method overrides parent**

Q6. What is the output of following?

java
Copy code
```
class A { void show() { System.out.println("A"); } }
class B extends A { void show() { System.out.println("B"); } }
public class Main { public static void main(String[] args) {
  A a = new B(); a.show();
}}
```
✅ Ans: **B** – because of dynamic method dispatch.

💻 **CODING QUESTIONS**

Q1. Basic Inheritance Example

java
Copy code
```
class Animal {
 void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
 void sound() { System.out.println("Bark"); }
}
public class Test {
 public static void main(String[] args) {
   Animal a = new Dog();
   a.sound();  // Output: Bark
 }
}
```

Q2. Use of super keyword

java
Copy code
```
class Parent {
 int a = 10;
 void show() { System.out.println("Parent show()"); }
}
class Child extends Parent {
 int a = 20;
 void show() {
   System.out.println("Child show()");
   System.out.println("Parent a: " + super.a);
```

```
    super.show();
  }
}
```

Q3. Constructor chaining using super

```
java
Copy code
class A {
  A() { System.out.println("A's constructor"); }
}
class B extends A {
  B() {
    super(); // Optional
    System.out.println("B's constructor");
  }
}
```

Q4. Method Overriding with covariant return type

```
java
Copy code
class A {
  A get() { return this; }
}
class B extends A {
  @Override
  B get() { return this; }
}
```

Q5. Demonstrate private member inheritance

```
java
Copy code
class A {
  private int x = 5;
  protected int getX() { return x; }
}
class B extends A {
  void show() {
    System.out.println("x: " + getX());
  }
}
```

🧠 **SYSTEM DESIGN + ARCHITECTURE QUESTIONS (Senior level)**

- What design patterns use inheritance? (Ans: Template Pattern, Strategy, etc.)
- When would you choose composition over inheritance?
- How to achieve polymorphism without inheritance?
- What issues can arise with deep inheritance hierarchies?
- How does inheritance impact testability and code maintainability?

==========

==========

POLYMORPHISM IN JAVA – INTERVIEW MASTER NOTES

==========

🧠 THEORY QUESTIONS (DEEP & COMPLETE EXPLANATIONS)

Q1: What is Polymorphism in Java?
Ans: Polymorphism means "many forms". It allows objects to take on multiple forms depending on context. In Java, it means a method behaves differently based on the object that is invoking it. It promotes code reusability and flexibility.

Q2: What are the types of Polymorphism in Java?
Ans:
1. Compile-time Polymorphism (Static binding / Method Overloading)
2. Run-time Polymorphism (Dynamic binding / Method Overriding)

Q3: What is Method Overloading?
Ans: Method Overloading is when two or more methods in the same class have the same name but different parameters (number/type/order). It’s an example of compile-time polymorphism.

Q4: What is Method Overriding?
Ans: Method Overriding occurs when a subclass provides a specific implementation of a method already defined in its parent class. It’s resolved at runtime and is an example of runtime polymorphism.

Q5: Can a constructor be overloaded?
Ans: Yes. Constructors can be overloaded by changing the parameter list.

Q6: Can we override static methods?
Ans: No. Static methods belong to the class, not objects. They can be hidden but not overridden.

Q7: Can we override private or final methods?
Ans: No. Private methods are not inherited and final methods cannot be overridden.

Q8: What is the use of the `super` keyword in polymorphism?
Ans: `super` is used to call the parent class method or constructor from the child class. It's often used in method overriding to invoke the superclass version.

Q9: What is dynamic method dispatch?
Ans: It is the mechanism by which a call to an overridden method is resolved at runtime rather than compile time. It supports runtime polymorphism.

Q10: Is operator overloading supported in Java?
Ans: No, Java does not support user-defined operator overloading, unlike C++.

Q11: Can polymorphism be achieved without inheritance?
Ans: No. Runtime polymorphism depends on inheritance and method overriding.

Q12: Why is polymorphism important?

Ans: It allows us to write flexible, maintainable, and reusable code. It enables behavior to vary at runtime based on object types.

Q13: Difference between Overloading and Overriding?
Ans:

Aspect	Overloading	Overriding
Binding	Compile-time	Runtime
Inheritance	Not required	Required
Class	Same class	Parent-child class
Return type	Can be different	Must be same or covariant
Parameters	Must differ	Must be same

🧠 TECHNICAL INTERVIEW QUESTIONS

Q1: Can you overload methods by return type alone?
Ans: No. Return type alone is not enough to distinguish overloaded methods.

Q2: What is covariant return type in overriding?
Ans: It means the overriding method can return a subtype of the return type declared in the parent class method.

Q3: Can you override a method and reduce the visibility of the method?
Ans: No. You can only increase the visibility (e.g., from protected to public), not reduce it.

Q4: Is runtime polymorphism applicable to instance or static methods?
Ans: Only instance methods. Static methods are resolved at compile time.

Q5: What happens if you call an overridden method using a parent class reference?
Ans: The actual object type determines which version of the method is called (runtime polymorphism).

🧮 CODING QUESTIONS (WITH EXPLANATIONS)

Q1: Demonstrate method overloading with different parameter types.

```java
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}
public class TestOverload {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.add(2, 3));     // 5
        System.out.println(c.add(2.5, 3.5)); // 6.0
```

```
    }
}
```
Q2: Demonstrate method overriding using parent and child classes.

java
Copy code
```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
public class TestOverride {
    public static void main(String[] args) {
        Animal a = new Dog(); // Upcasting
        a.sound();           // Dog barks (runtime polymorphism)
    }
}
```
Q3: Show polymorphism using array of superclass references.

java
Copy code
```
class Shape {
    void draw() {
        System.out.println("Drawing shape");
    }
}
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing circle");
    }
}
class Square extends Shape {
    void draw() {
        System.out.println("Drawing square");
    }
}
public class TestShapes {
    public static void main(String[] args) {
        Shape[] shapes = { new Circle(), new Square() };
        for (Shape s : shapes) {
            s.draw(); // Executes subclass method
        }
    }
}
```
🎯 MCQ QUESTIONS (INTERVIEW LEVEL)
Q1: Which concept allows method behavior to change at runtime?
A. Abstraction

B. Polymorphism
C. Encapsulation
D. Inheritance
✅ Ans: B. Polymorphism

Q2: Which of the following is an example of method overloading?
A. Same method name, same parameters
B. Different method name, different parameters
C. Same method name, different parameters
D. Same method name, same class
✅ Ans: C. Same method name, different parameters

Q3: Method Overriding requires:
A. Same class
B. Different method names
C. Inheritance
D. Static methods
✅ Ans: C. Inheritance

Q4: Which keyword is used to access parent class method in child class?
A. this
B. super
C. base
D. parent
✅ Ans: B. super

Q5: Which is NOT true about polymorphism?
A. Enhances code flexibility
B. Occurs only at compile time
C. Supports method overriding
D. Reduces code duplication
✅ Ans: B. Occurs only at compile time

Q6: Can we overload the main method in Java?
A. No
B. Yes, but JVM only calls the standard one
C. Yes, and JVM calls the overloaded one
D. Only in classes with no constructor
✅ Ans: B. Yes, but JVM only calls the standard one

✅ REAL-TIME SCENARIO-BASED INTERVIEW QUESTIONS
Q1: How would you design a payment system where the method pay() behaves differently for CreditCard, PayPal, and UPI?
Ans: Use a superclass PaymentMethod with a pay() method and create subclasses for each payment type. Override the pay() method in each to provide specific behavior. Then use a reference of type PaymentMethod to point to specific implementations.

Q2: In your project, how did you use polymorphism?
Ans (example): In my School ERP system, I used polymorphism to handle different types of users (Admin, Teacher, Student). All users extend a base User class and override methods like login(), viewDashboard() to behave differently depending on the user type.

📌 SUMMARY:
Compile-time Polymorphism: Method Overloading

Runtime Polymorphism: Method Overriding

Achieved using inheritance + overriding

Promotes loose coupling, scalability

Supported via dynamic method dispatch

Static, final, and private methods can't be overridden

==========

==========

🔷 TOPIC: ABSTRACTION IN JAVA

==========

📘 THEORY CONCEPTS

🔹 Q1: What is Abstraction in Java?
Ans: Abstraction is the process of hiding internal implementation details and showing only the essential features to the user. It focuses on "what" an object does rather than "how" it does it.

🔹 Q2: Why is Abstraction important?
Ans: Abstraction helps to reduce complexity, improve code readability, enhance maintainability, and provide security by hiding internal logic.

🔹 Q3: How is abstraction achieved in Java?
Ans:
- Using **abstract classes** (0 to 100% abstraction)
- Using **interfaces** (100% abstraction prior to Java 8)

🔹 Q4: What is an abstract class?
Ans: A class declared with the `abstract` keyword. It can have abstract and non-abstract methods. It cannot be instantiated directly.

🔹 Q5: What is an abstract method?
Ans: A method without a body, declared using `abstract`. It must be implemented by subclasses unless the subclass is also abstract.

🔹 Q6: What is an interface?
Ans: An interface is a reference type in Java, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. All methods are implicitly `public` and `abstract` (except static and default methods from Java 8).

🔹 Q7: Difference between Abstract Class and Interface?

Feature	Abstract Class	Interface
Abstraction Level	Partial (0–100%)	Full (100% pre-Java 8)
Keywords	`abstract`	`interface`
Constructors	Allowed	Not allowed
Variables	Instance, static, final	public static final (by default)
Multiple Inheritance	Not supported	Supported via interfaces
Method Body	Can have both	Only from Java 8 (default/static)

🔹 Q8: Can you instantiate abstract classes?
Ans: No. Abstract classes cannot be instantiated directly.

🔹 Q9: Can constructors be defined in abstract classes?
Ans: Yes. Abstract classes can have constructors to initialize fields of the class.

🔹 Q10: Can abstract methods be static?
Ans: No. Abstract methods cannot be static, final, or private because they must be overridden.

🔹 Q11: Can an abstract class have a main method?
Ans: Yes. It can have a main method and be used to run a program.

🔹 Q12: Can you use access modifiers in interfaces?
Ans: Yes. Interface methods are implicitly `public`. From Java 9 onwards, you can also use `private` methods in interfaces.

🔹 Q13: What is the use of abstraction in real-world programming?
Ans: Helps define a common template or behavior while allowing flexibility to change implementations (e.g., payment systems, devices, APIs).

🧠 TECHNICAL INTERVIEW QUESTIONS

🔹 Q1: What is the main purpose of abstraction?
Ans: To hide implementation details and expose only necessary functionalities to reduce complexity and increase efficiency.

🔹 Q2: Can we declare a class abstract without having abstract methods?
Ans: Yes. A class can be abstract even if it doesn’t contain any abstract method.

🔹 Q3: Can an interface implement another interface?
Ans: Yes. Interfaces can extend one or more other interfaces.

🔹 Q4: What if a class does not implement all methods of an interface?
Ans: The class must be declared abstract; otherwise, a compilation error occurs.

🔹 Q5: What is the difference between abstraction and encapsulation?
Ans:
- Abstraction hides the **implementation**.

- Encapsulation hides the **data** using access modifiers.

🔹 Q6: How is abstraction achieved after Java 8?
Ans: With `default` and `static` methods in interfaces.

🔹 Q7: Can we create references of abstract class?
Ans: Yes, abstract class references can be used to refer to subclass objects.

💡 MCQ QUESTIONS

🔸 Q1: Which keyword is used to create abstract classes?
A. interface
B. class
C. abstract
D. extends
✅ Ans: C. abstract

🔸 Q2: What happens if an abstract method is not implemented in a subclass?
A. Nothing
B. Compile-time error
C. Runtime error
D. It gets skipped
✅ Ans: B. Compile-time error

🔸 Q3: Can we declare an abstract method as private?
A. Yes
B. No
✅ Ans: B. No

🔸 Q4: Which Java version introduced default methods in interfaces?
A. Java 7
B. Java 8
C. Java 9
D. Java 6
✅ Ans: B. Java 8

🔸 Q5: What is true about abstract classes?
A. Can be instantiated
B. Cannot have constructors
C. Cannot be final
D. Must contain at least one abstract method
✅ Ans: C. Cannot be final

👨‍💻 CODING QUESTIONS

🔹 Q1: Create an abstract class `Shape` with an abstract method `area()`. Implement it in `Circle` and `Rectangle`.

```java
abstract class Shape {
   abstract void area();
}

class Circle extends Shape {
   double radius = 5;

   void area() {
      System.out.println("Circle Area: " + (3.14 * radius * radius));
   }
}

class Rectangle extends Shape {
   int length = 4, breadth = 5;

   void area() {
      System.out.println("Rectangle Area: " + (length * breadth));
   }
}

public class Test {
   public static void main(String[] args) {
      Shape s1 = new Circle();
      Shape s2 = new Rectangle();
      s1.area();
      s2.area();
   }
}
```

- Q2: Demonstrate interface-based abstraction.

```
java
Copy code
interface Vehicle {
   void start();
   void stop();
}

class Car implements Vehicle {
   public void start() {
      System.out.println("Car started");
   }

   public void stop() {
      System.out.println("Car stopped");
   }
}

public class InterfaceTest {
   public static void main(String[] args) {
      Vehicle v = new Car();
```

```
        v.start();
        v.stop();
    }
}
```

🔹 Q3: Abstract class with constructor and concrete method.

```
java
Copy code
abstract class Animal {
    Animal() {
        System.out.println("Animal created");
    }

    void breathe() {
        System.out.println("Breathing...");
    }

    abstract void sound();
}

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

public class AnimalTest {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.breathe();
        a.sound();
    }
}
```

🧪 REAL-TIME EXAMPLE OF ABSTRACTION

🔸 ATM Machine:

You only interact with UI buttons like “Withdraw” or “Balance”.

Internal processing, like communication with the bank server, is hidden.

🔸 Java API:

We use classes like List, Map, or Scanner without knowing how they are implemented internally.

🏁 INTERVIEW TIP

Always mention real-world use cases and the benefits of abstraction in large enterprise systems during interviews. Highlight how it helps enforce contracts via interfaces and promotes loose coupling.

==========

✅ CONSTRUCTOR in Java (Including Constructor Overloading)

==========

📘 THEORETICAL CONCEPTS:

vbnet
Copy code
What is a Constructor?
- A constructor in Java is a special method that is used to initialize objects.
- It is called automatically when an object is created.
- It has the same name as the class and has no return type (not even void).
Types of Constructors:
1. Default Constructor (no parameters, compiler-provided if no constructor is defined)
2. No-arg Constructor (user-defined constructor with no parameters)
3. Parameterized Constructor (constructor with parameters)
4. Copy Constructor (Java doesn't have built-in copy constructor like C++, but we can mimic it manually)
Constructor Overloading:
- Multiple constructors with different parameter lists within the same class.
- Achieved through compile-time polymorphism.

❓ INTERVIEW QUESTIONS (with answers):

vbnet
Copy code
Q1: What is the purpose of a constructor in Java?
A1: To initialize the object when it is created. It sets initial values for object attributes.
Q2: Can a constructor be private in Java?
A2: Yes. Used in Singleton design pattern or for restricting object creation from outside the class.
Q3: How is constructor overloading achieved?
A3: By defining multiple constructors with different parameter lists in the same class.
Q4: What happens if you don’t define any constructor?
A4: The Java compiler automatically provides a default constructor.

Q5: Can a constructor call another constructor?
A5: Yes. Using `this()` keyword.
Q6: Can you call a constructor explicitly?
A6: No, you cannot call constructors directly like methods. But you can invoke one constructor from another using `this()`.
Q7: Can a constructor be static?
A7: No. Constructors are invoked during object creation, and static belongs to class level, not object level.
Q8: Can a constructor return a value?
A8: No. Constructors don’t have a return type, not even void.

Q9: What is constructor chaining?
A9: When one constructor calls another constructor in the same class using `this()`, or superclass constructor using `super()`.
Q10: Can we override constructors?
A10: No. Constructors are not inherited, so they cannot be overridden.

💻 CODING QUESTIONS (with explanations):

typescript

Copy code

Q1: Write a class with overloaded constructors and explain how each is called.

```
class Student {
   String name;
   int age;
Student() {
      System.out.println("Default constructor");
   }
Student(String name) {
      this.name = name;
      System.out.println("Name constructor: " + name);
   }
Student(String name, int age) {
      this.name = name;
      this.age = age;
      System.out.println("Name and age constructor: " + name + ", " + age);
   }
}
public class Main {
   public static void main(String[] args) {
      Student s1 = new Student();
      Student s2 = new Student("Alice");
      Student s3 = new Student("Bob", 22);
   }
}
```

Q2: Constructor chaining example:

```
class A {
   A() {
      this(10);
      System.out.println("No-arg constructor");
   }
A(int x) {
      System.out.println("Parameterized constructor: " + x);
   }
}
```

Q3: Singleton class using private constructor:

```
class Singleton {
   private static Singleton instance = null;
private Singleton() {
      System.out.println("Private Constructor");
   }
public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
}
```

✅ **MCQs with Answers:**

vbnet

Copy code

Q1: What is the return type of a constructor?
a) void
b) int
c) class type
d) none of the above
Ans: d) none of the above
Q2: Can a constructor be final?
a) Yes
b) No
Ans: b) No
Q3: Which keyword is used to call another constructor in the same class?
a) super
b) this
c) static
d) new
Ans: b) this
Q4: If no constructor is defined, what happens?
a) Object cannot be created
b) Compilation error
c) Java provides a default constructor
d) Runtime error
Ans: c) Java provides a default constructor
Q5: Which constructor is called automatically when object is created?
a) default
b) static
c) instance
d) main
Ans: a) default

🧠 **DEEP TECHNICAL INTERVIEW QUESTIONS:**

pgsql
Copy code
Q1: Explain how constructor overloading differs from method overloading.
A1: Constructor overloading involves multiple constructors in the same class with different parameters, while method overloading involves methods with same name but different parameters.
Q2: When would you use a private constructor?
A2: To restrict object creation outside the class. Mainly in Singleton or Factory patterns.
Q3: Why constructors can't be inherited?
A3: Because constructors are meant to initialize an object of a specific class; child class needs to define its own constructor.

Q4: What is the use of `super()` in constructors?
A4: It calls the constructor of the immediate parent class. It must be the first line in the constructor.
Q5: Can we use both `this()` and `super()` in the same constructor?
A5: No. Both must be the first statement, and only one first statement is allowed.
==========

==========

STATIC KEYWORD IN JAVA – MASTER INTERVIEW NOTES

==========

This section covers: Static variables, static methods, static blocks, static class, static import, memory model, best practices, and common pitfalls.

🔹 THEORY: STATIC KEYWORD IN JAVA

---

1. The `static` keyword in Java is used for memory management.
2. It can be applied to:
   - Variables (Static Variables)
   - Methods (Static Methods)
   - Blocks (Static Initialization Blocks)
   - Classes (Static Nested Classes)
3. Static members belong to the **class** rather than the instance.
4. Static content loads **once at class loading time**, stored in **Method Area**.
5. You can access static methods/variables **without creating an object**.

🔹 STATIC VARIABLE (a.k.a. Class Variable)

- Shared by all objects of the class.
- Memory is allocated only once during class loading.

Example:

```java
class Student {
   static String college = "IIT";
   int id;
   Student(int id) {
      this.id = id;
   }
}
```

🔹 STATIC METHOD

Belongs to class.

Can be called without an object.

Can access only static data directly.

java
Copy code

```
class Util {
   static void show() {
      System.out.println("Static method called");
   }
}
```

⚠️ Note: Static methods cannot use this or super keywords.

🔹 STATIC BLOCK

Used for static initialization of class.

Runs before main() and constructor.

java
Copy code
```
class Demo {
   static {
      System.out.println("Static Block Executed");
   }
}
```
🔹 STATIC CLASS
Only nested classes can be static.

Static nested classes cannot access non-static members of the outer class directly.

java
Copy code
```
class Outer {
   static class Inner {
      void display() {
         System.out.println("Static Nested Class");
      }
   }
}
```
🔹 STATIC IMPORT (RARELY ASKED)
java
Copy code
```
import static java.lang.Math.*;
System.out.println(sqrt(25));  // No Math. prefix needed
```
🔹 COMMON INTERVIEW QUESTIONS (WITH ANSWERS)
Q: Why is the main() method static in Java?
✅ A: So JVM can call it without creating an object. Since execution starts before any object is created, static makes sense.

Q: Can static methods be overridden?
✅ A: No, static methods are not part of object's runtime polymorphism. They can be hidden (method hiding), not overridden.

Q: Can we have a constructor as static?
❌ No. Constructors are instance-level and static belongs to class.

Q: Can static methods access non-static members?
❌ No. Only static context is accessible inside static methods.

Q: What is the memory location of static members?
✅ Method Area (a.k.a Class Area or MetaSpace in Java 8+)

Q: How many copies of static variables exist per class?
✅ One shared copy per class, regardless of how many objects are created.

Q: Can we make an entire class static?
✅ Only nested classes can be declared static. Top-level classes cannot.

🔹 MCQs (MULTIPLE CHOICE QUESTIONS)

Which statement is true about static variables?
A. Each object gets its own copy
B. Stored in heap
C. Shared by all instances ✅
D. None

Can static methods be overloaded?
A. Yes ✅
B. No
C. Only in abstract classes
D. Only in interfaces

What will happen if we try to use this inside a static method?
A. Compile-time error ✅
B. Runtime exception
C. It works
D. None

Can we call a static method using an object?
A. Yes ✅
B. No
C. Only if it's public
D. Only if class is final

Where is static data stored?
A. Heap
B. Stack
C. Method Area ✅
D. JVM Thread Stack

🔹 CODING QUESTIONS – STATIC KEYWORD

✅ Q1. Count the number of objects created for a class using static variable

java
Copy code
```
class Counter {
   static int count = 0;
   Counter() {
      count++;
      System.out.println("Object created. Count = " + count);
   }
}
```
✅ Q2. Demonstrate method hiding

java
Copy code
```
class A {
   static void show() {
      System.out.println("A's static method");
```

```
    }
}
class B extends A {
    static void show() {
        System.out.println("B's static method");
    }
}
```

✅ Q3. Static block execution before main

```
java
Copy code
class Test {
    static {
        System.out.println("Static block called");
    }
    public static void main(String[] args) {
        System.out.println("main method");
    }
}
```

✅ Q4. Use static nested class

```
java
Copy code
class Outer {
    static class Inner {
        void display() {
            System.out.println("Inside static nested class");
        }
    }
    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();
        obj.display();
    }
}
```

🔹 TECHNICAL INTERVIEW ROUND SCENARIOS

If two objects of the same class change the static variable, what will happen?

✅ Both will reflect the updated value. It's common memory.

What's the difference between static block and constructor?

✅ Static block runs once during class load; constructor runs every time an object is created.

What's method hiding?

✅ When a subclass defines a static method with same signature as in parent, it hides the parent's static method, not overrides it.

Can we serialize static variables?

❌ No. Static fields are not part of object state.

Why are utility classes filled with static methods?

✅ Because utility methods don't require object state; examples: Math, Collections.

✅ FINAL NOTES:
Don’t overuse static variables. It leads to tight coupling.

Singleton pattern and Utility classes use static members widely.

Static helps in memory efficiency but lacks flexibility.

==========

✅ this and super Keyword in Java – Complete Interview Notes

🔹 **THEORY – this Keyword**

The this keyword refers to the **current object** of the class.

Use Cases:

To differentiate between instance and local variables.
To invoke current class constructor.
To pass current object as a parameter.
To return the current object from a method.

✅ **Example 1: Differentiating Variables**

java
Copy code

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

✅ **Example 2: Calling Constructor (Constructor Chaining)**

java
Copy code

```
class Demo {
   Demo() {
      this(10); // calls parameterized constructor
      System.out.println("Default constructor");
   }
   Demo(int x) {
      System.out.println("Parameterized constructor: " + x);
   }
}
```

✅ **Example 3: Returning current object**

java
Copy code

```
class Demo {
   Demo getObj() {
      return this;
   }
}
```

🔹 **THEORY – super Keyword**

The super keyword is used to refer to the **immediate parent class**.

Use Cases:

Call the parent class constructor (must be first statement).

Access parent class methods/fields that are overridden or hidden.

✅ **Example 1: Calling Parent Constructor**

java
Copy code

```
class Parent {
   Parent() {
      System.out.println("Parent Constructor");
   }
}
class Child extends Parent {
   Child() {
      super(); // must be the first statement
      System.out.println("Child Constructor");
   }
}
```

✅ **Example 2: Calling Parent Method**

java
Copy code

```
class Parent {
   void display() {
      System.out.println("Parent Method");
   }
}
class Child extends Parent {
   void display() {
      super.display(); // calls parent method
      System.out.println("Child Method");
   }
}
```

🔹 **CODING INTERVIEW QUESTIONS**

Q1. Output of this code?

java
Copy code

```
class A {
   int x = 5;
}
class B extends A {
   int x = 10;
   void show() {
      System.out.println(super.x); // ?
   }
}
```

Answer:

5 (because super.x accesses parent's variable)

Q2. Constructor chaining using this and super

java
Copy code

```
class A {
   A() { System.out.println("A Constructor"); }
}
class B extends A {
   B() {
      this(10);
      System.out.println("B Default");
   }
   B(int x) {
      super();
      System.out.println("B Parameterized: " + x);
   }
}
```

Output:

less
Copy code

```
A Constructor
B Parameterized: 10
B Default
```

🔹 **TECHNICAL INTERVIEW QUESTIONS**

Q1. Can this() and super() be used in the same constructor?
Answer: No. Both must be the **first statement**, so only one can appear.
Q2. What happens if super() is not explicitly written in a subclass constructor?
Answer: The compiler adds super() by default if no other constructor call is made.
Q3. What if a parent class doesn't have a no-arg constructor and super() is not used?
Answer: Compilation error. You must call the parameterized constructor using super(args).

🔹 **MCQs**

1. What does this refer to?
A) Static variable
B) Current object
C) Parent class
D) Constructor
✅ **Answer: B**
2. Which of the following is true about super?
A) Calls subclass constructor
B) Refers to static variable
C) Calls parent class constructor/method
D) None
✅ **Answer: C**
3. Can super() be used in static context?
✅ **Answer:** No, because it refers to object context.

==========

Method Overloading vs Method Overriding in Java

==========

🔹 **THEORY – Method Overloading**

Definition:

Method overloading means **defining multiple methods with the same name but different parameters** (type, number, or order) **within the same class**.

📌 It is a **compile-time polymorphism** (static binding).

✅ **Valid Overloading:**

java
Copy code

```
class Calculator {
   int add(int a, int b) {
      return a + b;
   }
   double add(double a, double b) {
      return a + b;
   }
   int add(int a, int b, int c) {
      return a + b + c;
   }
}
```

🛑 Return type alone **cannot** be used to overload a method.

🔹 **THEORY – Method Overriding**

Definition:

Method overriding means **redefining a method of the parent class in the child class** with the **same name, parameters, and return type**.

📌 It is a **runtime polymorphism** (dynamic binding).

✅ **Example:**

java
Copy code

```
class Animal {
```

```
    void sound() {
        System.out.println("Animal makes sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

🔹 **Difference Between Overloading and Overriding**

Feature	**Method Overloading**	**Method Overriding**
Occurs in	Same class	Subclass (Inheritance)
Method signature	Must differ in parameters	Must be exactly same
Return type	Can differ	Must be same (or covariant)
Access modifier	Can vary	Must be same or more accessible
Static/Final methods	Can be overloaded	**Cannot** be overridden
Polymorphism type	Compile-time	Runtime
Binding	Static binding	Dynamic binding
Use case	Method flexibility	Behavioral change in child class

🔹 **TECHNICAL INTERVIEW QUESTIONS**

Q1. Can you overload a method by changing only the return type?

A: ❌ No. Overloading must differ in parameters.

Q2. Can static methods be overridden?

A: ❌ No. Static methods are class-level; they can be hidden, not overridden.

Q3. What if parent method is private or final?

A: ❌ Final methods **cannot** be overridden.

Private methods are not inherited, so can't be overridden.

Q4. Can you override a method with a more restrictive access modifier?

A: ❌ No. You can only keep it same or **more accessible** (e.g., protected → public).

🔹 **CODING OUTPUT QUESTIONS**

Q1:

java

Copy code

```
class A {
    void show(int a) {
        System.out.println("A: int");
    }
    void show(String a) {
        System.out.println("A: String");
    }
}
new A().show("Hello");
```

✅ **Output:** A: String

Q2:

java
Copy code
```
class A {
   void show() { System.out.println("A"); }
}
class B extends A {
   void show() { System.out.println("B"); }
}
```

java
Copy code
```
A obj = new B();
obj.show();
```
✅ **Output:** B
(Overriding – Runtime Polymorphism)

🔹 **MCQs**
1. Overloading is which type of polymorphism?
A) Runtime
B) Compile-time
✅ **Answer:** B
2. Which is true about method overriding?
A) Must change return type
B) Must change access modifier
C) Happens in same class
D) Must have same signature
✅ **Answer:** D
3. Which method can be overridden?
A) static
B) private
C) final
D) public
✅ **Answer:** D

🔹 **CODING QUESTION**
Q: Implement method overloading and overriding in the same program.

java
Copy code
```
class Parent {
   void show() {
      System.out.println("Parent Show");
   }
}
class Child extends Parent {
   @Override
   void show() {
      System.out.println("Child Show");
   }
```

```
void show(String msg) {
        System.out.println("Child Overloaded: " + msg);
    }
}
public class Main {
    public static void main(String[] args) {
        Parent obj1 = new Child();
        obj1.show(); // Overriding
Child obj2 = new Child();
        obj2.show("Hello"); // Overloading
    }
}
```

✅ **Output:**

```
sql
Copy code
Child Show
Child Overloaded: Hello
```

✅ final Keyword in Java

🔹 **THEORY – What is final in Java?**

The final keyword in Java is used to **restrict modification**. It can be applied to:

Variables
Methods
Classes

It ensures that:

- Final variables **cannot be reassigned**
- Final methods **cannot be overridden**
- Final classes **cannot be extended**

🔹 **1. Final Variables**

Once assigned, **cannot be changed**.

```
java
Copy code
final int x = 10;
x = 20; // ❌ Compile-time error
```

📌 You **must initialize** final variables either:

- at declaration
- in constructor
- in static block (for static finals)

🔹 **Final Instance Variable**

```
java
Copy code
class Demo {
```

```
    final int x;  // Blank final variable
Demo() {
        x = 100;  // Must initialize in constructor
    }
}
```

🔹 **Final Static Variable**

java
Copy code

```
class Test {
    static final int MAX;
static {
        MAX = 99;
    }
}
```

🔹 **2. Final Methods**

A final method **cannot be overridden** in subclass.

java
Copy code

```
class A {
    final void show() {
        System.out.println("A");
    }
}
class B extends A {
    void show() {  // ❌ Error
        System.out.println("B");
    }
}
```

✅ Useful when you want to **prevent altering core logic** in subclasses.

🔹 **3. Final Class**

A class declared final **cannot be inherited**.

java
Copy code

```
final class Shape { }
class Circle extends Shape { }  // ❌ Error
```

✅ Used for **security and immutability**.

🔹 **Final Parameters**

java
Copy code

```
void print(final int x) {
    x = 5;  // ❌ Error
}
```

✅ The method can **use** the parameter, but **cannot modify** it.

🔹 **Final with Objects**

java
Copy code
final Person p = new Person();
p.name = "John"; // ✅ Allowed (object state change)
p = new Person(); // ❌ Error (reassignment not allowed)
✅ You can **change the fields**, but **not the reference**.

🔹 **Final with Constructors**
- You **cannot** declare a constructor as final.

java
Copy code
final Demo() { } // ❌ Error

🔹 **Final + Static + Blank Variable**

java
Copy code
static final int RATE;
static {
  RATE = 50; // ✅ Must initialize in static block
}

✅ **INTERVIEW Q&A**

🔸 **Q1: Can final variable be uninitialized?**
A: Yes, but only if it's a **blank final variable** initialized in constructor or static block.

🔸 **Q2: What's the difference between final and const?**
A: Java uses only final. const is reserved but **not used**.

🔸 **Q3: Can a final method be overloaded?**
A: ✅ Yes. Overloading is allowed; overriding is not.

🔸 **Q4: Can we declare a constructor as final?**
A: ❌ No. Constructors cannot be final because they are not inherited.

🔸 **Q5: Can a static method be final?**
A: ✅ Yes. It cannot be overridden anyway, so it's valid.

🔸 **Q6: Can an abstract method be final?**
A: ❌ No. Abstract methods are meant to be overridden. Final prevents that.

🔸 **Q7: What happens if you don't initialize a final variable?**
A: ❌ Compile-time error – it must be initialized either at declaration, constructor, or block.

✅ **CODING OUTPUT QUESTIONS**
Q1:

java
Copy code
```
final int x = 10;
x = x + 1;
System.out.println(x);
```
✅ **Output:** Compilation error – cannot assign value to final variable

Q2:

java
Copy code
```
final int[] arr = {1, 2, 3};
arr[0] = 100;
System.out.println(arr[0]);
```
✅ **Output:** 100
(final reference allowed to change internal state)

Q3:

java
Copy code
```
class A {
   final void show() {
      System.out.println("A");
   }
}
class B extends A {
   void show() {
      System.out.println("B");
   }
}
```
✅ **Output:** Compilation Error – cannot override final method

Q4:

java
Copy code
```
final class Car {
   void start() {
      System.out.println("Car started");
   }
}
class Tesla extends Car {
   void start() {
      System.out.println("Tesla started");
   }
}
```
✅ **Output:** Compilation Error – cannot extend final class

✅ **MCQs**
1. Which of the following is true about final keyword?

A) Can be applied to variable only
B) Final class can be extended
C) Final method can't be overridden ✅
D) Final constructor is allowed
✅ **Answer:** C

2. What happens when a final object reference is reassigned?
A) Compiles but fails
B) Reassignment allowed
C) Compile-time error ✅
D) Runtime error
✅ **Answer:** C

3. Which can't be final in Java?
A) Class
B) Variable
C) Method
D) Constructor ✅
✅ **Answer:** D

4. What is blank final variable?
A) Unused variable
B) Final variable not initialized at declaration ✅
C) Static final variable
D) Local variable
✅ **Answer:** B

✅ **CODING PRACTICE: Final Keyword**
Q: Create a class where:
- final variable is initialized in constructor
- final method cannot be overridden
- final class can't be extended

java
Copy code
```
final class Vehicle {
   final int speed;
Vehicle(int speed) {
      this.speed = speed;
   }
final void run() {
      System.out.println("Running at speed: " + speed);
   }
}
// class Car extends Vehicle {}  // ❌ Not allowed
public class Main {
   public static void main(String[] args) {
      Vehicle v = new Vehicle(100);
      v.run();
   }
}
```

✅ **Output:**
Running at speed: 100
==========

instanceof Operator in Java – Complete Interview Guide

==========

🔷 **THEORY: What is instanceof?**
The instanceof operator in Java is used to test whether an object is an instance of a specific class or implements an interface.
Syntax:

java
Copy code
object instanceof ClassName
It returns true if the object is of the specified type, else false.

💡 **Use Cases of instanceof:**
- To **avoid ClassCastException** before casting
- For **dynamic type checking** in polymorphic code
- For **interface checking**
- To implement **custom behavior based on type**

✅ **CODE EXAMPLE 1: Basic usage**

java
Copy code
```
class Animal {}
class Dog extends Animal {}
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();
        System.out.println(a instanceof Dog);     // true
        System.out.println(a instanceof Animal);  // true
        System.out.println(a instanceof Object);  // true
    }
}
```

✅ **CODE EXAMPLE 2: Interface check**

java
Copy code
```
interface Vehicle {}
class Car implements Vehicle {}
public class Demo {
    public static void main(String[] args) {
        Vehicle v = new Car();
        System.out.println(v instanceof Car);      // true
        System.out.println(v instanceof Vehicle);   // true
    }
```

}

⚠️ **NOTE:**
instanceof **returns false if the reference is null.**

java
Copy code
String s = null;
System.out.println(s instanceof String); // false

❓ **INTERVIEW QUESTIONS – Theory Based**

What is the instanceof operator?
- It is a binary operator used to test whether an object is an instance of a specified class or interface.

When do you use instanceof?
- When performing safe downcasting or applying logic depending on runtime type.

Can instanceof be used with interfaces?
- Yes. You can check if an object implements a specific interface.

What happens if instanceof is used on a null reference?
- It returns false.

How does instanceof help with polymorphism?
- It checks the actual object type before casting, preventing ClassCastExceptions.

What's the difference between instanceof and .getClass()?
- instanceof checks for inheritance, getClass() checks for exact match of class type.

🧠 **TECHNICAL ROUND QUESTIONS**

Q: How would you handle type-specific behavior in a method taking Object as a parameter?

java
Copy code
public void process(Object obj) {
    if (obj instanceof String) {
        System.out.println("It's a string: " + obj);
    } else if (obj instanceof Integer) {
        System.out.println("It's an integer: " + obj);
    }
}

Q: Is instanceof considered bad design?
- It can be an **indicator of poor OO design** when overused. It's better to use **method overriding** or **polymorphism**.

💻 **CODING INTERVIEW QUESTIONS**

Q1. Safe downcasting using instanceof:

java
Copy code
Object obj = "Hello";
if (obj instanceof String) {
    String str = (String) obj;
    System.out.println(str.toUpperCase());
}

Q2. Custom implementation with interface check:

java
Copy code
```
interface Notification {}
class Email implements Notification {}
class SMS implements Notification {}
void notifyUser(Notification n) {
   if (n instanceof Email) {
      System.out.println("Send email notification.");
   } else if (n instanceof SMS) {
      System.out.println("Send SMS notification.");
   }
}
```

📝 **MCQs on instanceof Operator**

Q1. What is the result of null instanceof Object?

a) true
b) false
c) Compilation error
d) Runtime error

✅ **Answer: b) false**

Q2. Can instanceof be used with interfaces?

a) No
b) Yes

✅ **Answer: b) Yes**

Q3. If obj instanceof String is true, what does that imply?

a) obj is null
b) obj is of type String or subclass

✅ **Answer: b)**

Q4. What is the output?

java
Copy code
```
Object obj = new Integer(5);
System.out.println(obj instanceof Number);
```

a) true
b) false

✅ **Answer: a) true**

Q5. Which of the following is false?

a) instanceof can be used with interfaces
b) instanceof returns true if object is subclass
c) null instanceof Class returns true

✅ **Answer: c)**

==========

✅ Object Class in Java

(Includes: toString(), equals(), hashCode(), clone(), etc.)

🔷 What is the Object Class?

In Java, every class is a **child of java.lang.Object**, either directly or indirectly.
It is the **root of the class hierarchy**, providing basic methods common to all objects.

🔹 List of Common Methods in Object Class:

Method	Description
toString()	Returns string representation of the object
equals(Object obj)	Compares two objects for equality
hashCode()	Returns hash code value of object
getClass()	Returns runtime class of the object
clone()	Creates and returns a copy of the object
finalize()	Called by garbage collector before object is destroyed
wait()	Causes current thread to wait
notify()	Wakes up a thread waiting on this object
notifyAll()	Wakes up all threads waiting on this object

🔹 1. toString() Method

➤ **Purpose:**
Returns a **string representation** of the object.
➤ **Default:**

java
Copy code
ClassName@hashcode
➤ **Override Example:**

java
Copy code
```
public class Student {
   int id = 101;
   String name = "John";
public String toString() {
      return id + " - " + name;
   }
}
```

🔹 2. equals(Object obj) Method

➤ **Purpose:**
Compares **content equality** of two objects.
➤ **Default:**
Compares memory address (==).
➤ **Override Example:**

java
Copy code
```
public boolean equals(Object obj) {
```

```
    Student s = (Student)obj;
    return this.id == s.id && this.name.equals(s.name);
}
```

🔹 **3. hashCode() Method**

➤ **Purpose:**

Returns an **integer hash code** for the object, used in hashing structures like HashMap.

➤ **Contract:**

If two objects are equal (equals()), then **hashCode() must be same**.

➤ **Example:**

```
java
Copy code
@Override
public int hashCode() {
    return id * 31 + name.hashCode();
}
```

🔹 **4. getClass() Method**

➤ **Purpose:**

Returns the **Class object** that represents the runtime class.

➤ **Example:**

```
java
Copy code
Student s = new Student();
System.out.println(s.getClass().getName());
```

🔹 **5. clone() Method**

➤ **Purpose:**

Creates a **copy (clone)** of the object.

➤ **Requirements:**

- Class must implement Cloneable interface
- Must override clone()

➤ **Example:**

```
java
Copy code
class Student implements Cloneable {
    int id = 101;
public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

🔹 **6. finalize() Method**

➤ **Purpose:**

Called by GC **before object is garbage collected**.

➤ **Example:**

```
java
```

Copy code
```
protected void finalize() {
   System.out.println("Object is destroyed");
}
```
⚠️ Note: **Deprecated from Java 9 onwards**

🔹 **7. wait(), notify(), notifyAll() Methods**

These are used for **thread communication (synchronization)** and can be called only from synchronized blocks.

❓ **THEORY INTERVIEW QUESTIONS**

Why is Object class the root class in Java?

→ It provides common methods to every object in Java, enabling polymorphism.

What does toString() return by default?

→ ClassName@HexHashCode

What's the difference between == and equals()?

→ == checks reference; equals() can check content if overridden.

Can two equal objects have different hashCodes()?

→ No. If equals() returns true, hashCode() must be same.

What happens if you don't override hashCode() when overriding equals()?

→ May cause problems in HashMap, HashSet, etc.

Why is clone() protected?

→ It's meant to be used by subclasses and needs explicit permission to clone.

💻 **CODING INTERVIEW PRACTICE**

Q1. Override equals() and hashCode() properly:

java
Copy code
```
class Student {
   int id;
   String name;
public boolean equals(Object o) {
      if (this == o) return true;
      if (!(o instanceof Student)) return false;
      Student s = (Student) o;
      return id == s.id && name.equals(s.name);
   }
public int hashCode() {
      return id * 31 + name.hashCode();
   }
}
```
Q2. Implement a clone() method:

java
Copy code
```
class Employee implements Cloneable {
   int eid;
   public Object clone() throws CloneNotSupportedException {
      return super.clone();
   }
}
```

📝 **MCQs on Object Class**

Q1. Which method must be overridden to use objects as keys in HashMap?
a) equals() only
b) hashCode() only
c) equals() and hashCode() ✅
d) toString()

Q2. What is the return type of getClass()?
a) String
b) Object
c) Class ✅
d) void

Q3. What does toString() return if not overridden?
a) null
b) object ID
c) hash code in decimal
d) ClassName@hashcode ✅

Q4. What happens if you call clone() on an object not implementing Cloneable?
a) Compiles but fails at runtime ✅
b) Compiles and works
c) Compile-time error
d) No output

Q5. Which of these is not in Object class?
a) toString()
b) finalize()
c) notifyAll()
d) sleep() ✅

==========
==========
==========

🔒 14. ACCESS MODIFIERS IN JAVA – ALL INTERVIEW QUESTIONS

==========

🎓 **THEORY EXPLANATION**

Java provides **four access modifiers** to control visibility:

private – Accessible only within the same class.
default (no modifier) – Accessible within the same package.
protected – Accessible within the same package and subclasses (even outside the package).
public – Accessible from anywhere.

💡 **INTERVIEW QUESTIONS (THEORY)**

Q: What are access modifiers in Java?
A: Access modifiers are keywords used to define the scope or visibility of classes, constructors, methods, and variables. Java provides private, default, protected, and public.

Q: Explain the difference between private and protected.

A: private members are accessible only within the same class, while protected members are accessible within the same package and also in subclasses outside the package via inheritance.

Q: Can top-level classes be private in Java?

A: No. Top-level classes can only be declared public or have default (package-private) access. Inner classes can be private.

Q: What happens if you don't specify an access modifier?

A: It defaults to **package-private** (accessible within the same package only).

Q: Can we override a method and reduce its visibility?

A: No. Overridden methods must have the same or more accessible visibility, not less.

Q: Is it possible to access a protected member from a non-subclass outside the package?

A: No. protected allows access only to subclasses (outside package) or within the same package.

Q: What is the best practice regarding access modifiers in OOP design?

A: Use the **principle of least privilege**. Keep fields private, provide public methods only when necessary, and prefer protected or package-private access to limit misuse.

Q: How do access modifiers relate to encapsulation?

A: Access modifiers enable encapsulation by restricting direct access to data members and enforcing controlled access via getter/setter methods.

🧠 MCQs WITH ANSWERS

Which access modifier gives the **least** accessibility?

a) public
b) private
c) protected
d) default

✅ **Ans: b) private**

Which access modifier allows access from other packages but only in subclasses?

a) default
b) protected
c) public
d) private

✅ **Ans: b) protected**

A top-level class can have which of the following modifiers?

a) private
b) protected
c) default
d) public

✅ **Ans: c) default and d) public**

Which of the following statements is **false**?

a) private methods can be accessed outside class using inheritance
b) public methods can be accessed from anywhere
c) default access restricts usage to the same package
d) protected allows access to subclass even in different package

✅ **Ans: a) private methods can be accessed outside class using inheritance**

💻 CODING QUESTIONS WITH EXPLANATION

🔸 **Q1. Write a class with private fields and show how to access them using getter/setter methods.**

java
Copy code

```
public class Student {
   private String name;
```

```
    private int age;
public String getName() {
        return name;
    }
public void setName(String name) {
        this.name = name;
    }
public int getAge() {
        return age;
    }
public void setAge(int age) {
        this.age = age;
    }
}
```

✅ This demonstrates encapsulation using private access and public getters/setters.

🔸 **Q2. Show an example where a protected method is accessed in a subclass outside the package.**

java
Copy code

```
// In package com.base
package com.base;
public class Base {
    protected void show() {
        System.out.println("Protected method");
    }
}
// In package com.derived
package com.derived;
import com.base.Base;
public class Derived extends Base {
    public void display() {
        show(); // accessible due to inheritance
    }
}
```

✅ show() is protected, so it's accessible in Derived, which extends Base.

🔸 **Q3. What happens if a method is declared with more restrictive access during overriding?**

java
Copy code

```
class A {
    public void greet() {
        System.out.println("Hello from A");
    }
}
class B extends A {
    // compile-time error
    private void greet() {
        System.out.println("Hello from B");
    }
}
```

❌ This gives a **compile-time error** because the overridden method greet() in B cannot be more restrictive (private) than the original method in A (public).

🔸 **Q4. Create a class in default access and try using it from another package.**

java
Copy code
```
// package1
package package1;
class MyDefaultClass {
   void message() {
      System.out.println("Hello from default class");
   }
}
// package2
package package2;
import package1.MyDefaultClass; // error: not visible
```
❌ You'll get a compile-time error because default classes are **not visible outside the package**.

👨‍💼 **TECHNICAL INTERVIEW QUESTIONS**

Why are fields often declared as private in a class?
- To enforce encapsulation and prevent direct modification. Only controlled access through methods is allowed.

Can a subclass access private fields of its superclass?
- No. Even subclasses cannot access private members of the parent class.

How does protected differ from default in terms of inheritance?
- protected gives access in subclasses (even outside the package); default does not.

Why can't we use private for methods that are overridden?
- Private methods are not inherited, so they can't be overridden.

Why should constructor access levels be controlled?
- To control object creation – e.g., private constructors are used in Singleton patterns.

==========

✅ Aggregation vs Composition in Java (OOP)

🔍 DEFINITION & DIFFERENCE

Aspect	Aggregation	Composition
Definition	"Has-a" relationship where the child **can exist independently** of the parent	"Has-a" relationship where the child **cannot exist independently**
Lifespan	Independent lifespan	Dependent lifespan
Tight Coupling	Loosely coupled	Strongly coupled
Use Case	Student has Address (Address can exist independently)	Human has Heart (Heart cannot exist without Human)
Code Indicator	Uses regular object reference	Usually uses new keyword inside the class

🧠 THEORY EXPLANATION

- Both **Aggregation** and **Composition** are types of **Association** in OOP.
- **Association** means one class is connected with another class.
  - **Aggregation** is a *weaker form* of association.
  - **Composition** is a *stronger form* of association.

🧩 REAL-LIFE ANALOGY

- **Aggregation**: A Department has multiple Teachers. Even if the Department is deleted, teachers still exist.
- **Composition**: A House has Rooms. If the House is destroyed, Rooms don't exist.

💻 CODE EXAMPLES

✅ Aggregation Example

java
Copy code

```
class Address {
   String city, state;
   Address(String city, String state) {
      this.city = city;
      this.state = state;
   }
}
class Student {
   int id;
   String name;
   Address address; // Aggregation
Student(int id, String name, Address address) {
      this.id = id;
      this.name = name;
      this.address = address;
   }
void display() {
```

```
        System.out.println(name + " lives in " + address.city + ", " + address.state);
    }
}
public class Test {
    public static void main(String[] args) {
        Address addr = new Address("Bangalore", "Karnataka");
        Student s1 = new Student(101, "Amit", addr);
        s1.display();
    }
}
```

✅ **Composition Example**

java
Copy code

```
class Heart {
    void beat() {
        System.out.println("Heart is beating...");
    }
}
class Human {
    private Heart heart;
Human() {
        heart = new Heart(); // Composition
    }
void startLife() {
        heart.beat();
    }
}
public class Test {
    public static void main(String[] args) {
        Human h = new Human();
        h.startLife();
    }
}
```

🧠 **TECHNICAL INTERVIEW QUESTIONS**

Q1. What is aggregation in Java with example?

→ Aggregation is a "has-a" relationship where the child can exist independently. Example: Student has an Address.

Q2. What is composition in Java with example?

→ Composition is a "has-a" relationship where the child **cannot** exist independently. Example: Human has a Heart.

Q3. How are aggregation and composition implemented in Java?

→ Aggregation uses object references passed externally. Composition typically uses new inside the class to create the dependent object.

Q4. Which is preferred in system design: aggregation or composition?

→ Use composition when the part's lifecycle is tightly bound to the whole. Use aggregation when components are reusable.

Q5. Can you achieve aggregation using interfaces?

→ Yes. Aggregation allows flexibility, so interfaces can be used for polymorphic design.

🧠 UML DIAGRAM REPRESENTATION

vbnet
Copy code
AGGREGATION:
Student <>---- Address
  "Has a" but Address can exist independently
COMPOSITION:
Human ■ ---- Heart
  "Owns" and Heart is destroyed with Human
Legend:

- ■ ---- (filled diamond): Composition
- <>---- (empty diamond): Aggregation

💡 KEY DIFFERENCE IN LIFECYCLE (IN SHORT):

Parent Dies	Child in Aggregation	Child in Composition
Deleted	Continues to exist	Dies with parent

✅ MCQs on Aggregation vs Composition

1. What is the main difference between Aggregation and Composition?
A. Inheritance vs Interface
B. Tight vs Loose coupling
C. Dependent vs Independent object lifecycles
D. None
✅ **Answer: C**

2. Which one represents strong ownership?
A. Aggregation
B. Composition
C. Association
D. Inheritance
✅ **Answer: B**

3. Which is an example of composition?
A. Student has Address
B. Library has Books
C. Human has Heart
D. Company has Employees
✅ **Answer: C**

4. Which diagram symbol is used for aggregation in UML?
A. Filled diamond
B. Empty diamond
C. Arrow
D. Circle
✅ **Answer: B**

5. Which object is created inside the owner class in Composition?
A. In interface
B. Using external setter

C. Using new inside class
D. All of the above
✅ **Answer: C**
==========

Custom Exception Handling (OOP Design Focus)

==========

🔧 WHAT IS A CUSTOM EXCEPTION?

A **custom exception** is a **user-defined class** that extends either:

- Exception (for checked)
- or RuntimeException (for unchecked)

Use when **domain-specific** error handling is required.

✅ WHEN TO USE CUSTOM EXCEPTIONS?

- Business rule violations (e.g., "Minimum balance not maintained")
- More **readable**, **domain-driven** error messages
- Clean separation of concerns
- Easier debugging and logging

📐 HOW TO CREATE A CUSTOM EXCEPTION?

✅ Checked Exception

java
Copy code

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

✅ Unchecked Exception

java
Copy code

```
class InvalidInputException extends RuntimeException {
    public InvalidInputException(String message) {
        super(message);
    }
}
```

💻 CUSTOM EXCEPTION USAGE EXAMPLE

java
Copy code

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String msg) {
        super(msg);
    }
}
```

```
class Voter {
   void checkEligibility(int age) throws InvalidAgeException {
      if (age < 18) {
         throw new InvalidAgeException("Age must be >= 18 to vote");
      }
      System.out.println("Eligible to vote");
   }
}
public class Test {
   public static void main(String[] args) {
      Voter v = new Voter();
      try {
         v.checkEligibility(15);
      } catch (InvalidAgeException e) {
         System.out.println("Exception caught: " + e.getMessage());
      }
   }
}
```

🧠 INTERVIEW QUESTIONS

Q1. Why create a custom exception?
→ For specific domain-based error handling and more meaningful error messages.

Q2. What is the difference between throw and throws?

- throw: used to explicitly throw an exception.
- throws: used in method declaration to indicate possible exceptions.

Q3. What happens if exception is not caught?
→ The program terminates abnormally, and the JVM prints a stack trace.

Q4. Can a custom exception be unchecked?
→ Yes, by extending RuntimeException.

Q5. Can a constructor throw an exception?
→ Yes, constructors can throw both checked and unchecked exceptions.

✅ BEST PRACTICES IN EXCEPTION HANDLING

- **Never swallow** exceptions silently (catch(Exception e){} is bad).
- Log exceptions with meaningful messages.
- Create domain-specific custom exceptions.
- Use finally for clean-up tasks like closing files or DB connections.
- Do not use exceptions for flow control.

📦 COMMON JAVA EXCEPTIONS

Exception	Type	When it occurs
NullPointerException	Unchecked	Object reference is null
ArrayIndexOutOfBoundsException	Unchecked	Invalid index accessed
ArithmeticException	Unchecked	Division by zero
IOException	Checked	File or stream failure
ClassNotFoundException	Checked	Class not found during runtime
NumberFormatException	Unchecked	Invalid string to number conversion

✅ MCQs on Exception Handling

1. Which class is the superclass of all exceptions?
A. Error
B. Throwable
C. RuntimeException
D. Exception
✅ **Answer: B**

2. Which block is always executed?
A. try
B. catch
C. finally
D. throw
✅ **Answer: C**

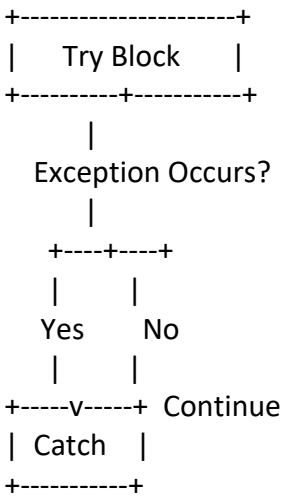
3. What does throws keyword do?
A. Catches exceptions
B. Ignores exception
C. Declares exceptions
D. Defines custom exceptions
✅ **Answer: C**

4. What happens if an exception is not caught?
A. Program runs fine
B. JVM terminates program
C. JVM recompiles code
D. Nothing
✅ **Answer: B**

5. Which of the following is a checked exception?
A. NullPointerException
B. ArrayIndexOutOfBoundsException
C. IOException
D. ArithmeticException
✅ **Answer: C**

✅ **DIAGRAM — Exception Handling Flow**

pgsql
Copy code



```
    |
+-----v-----+
|  Finally  |
+-----------+
    |
  Continue
```

==========

REAL-LIFE DESIGN EXAMPLES & UML THINKING (OOP DESIGN FOCUS)

==========

🔍 Overview:
This section helps interview candidates demonstrate **real-world modeling using OOP principles**, converting problem statements into **UML diagrams**, and explaining design rationale.

🎓 KEY UML & DESIGN CONCEPTS

- **Class Diagrams**: Represent classes, attributes, methods, and relationships.
- **Relationships**:
  - Association / Aggregation / Composition
  - Generalization (Inheritance) – “is-a”
  - Interface Realization – “implements”
- **Multiplicity**: 1..*, 0..1, etc.
- **Visibility indicators**: + public, – private, # protected
- **OOP Principles**: Encapsulation, Abstraction, Inheritance, Polymorphism

---

🧠 EXAMPLE 1: Library Management System

Classes:
- **Library**
- **Book**
- **Member**
- **Librarian**
- **Loan** (represents book issue and return)

Relationships:
- Library aggregates Books and Members
- A Member borrows multiple Books (Loan is association)
- Librarian manages Book inventory

UML Sketch:
Library

addBook()

registerMember()
–

1..*
├— Book
| + title
| + author
| + ISBN

1..*
└— Member
+ memberID
+ name

Loan

issueDate

returnDate

Member —< Loan >— Book
Librarian — manages — Library

markdown
Copy code

🧠 EXAMPLE 2: Online Shopping System

Classes:
• **User** (abstract)
• **Customer**, **Admin**
• **Product**
• **Order**
• **Cart**
• **Payment**, **CreditCardPayment**, **UPIPayment**
• **Inventory**

Design Highlights:
• Use **abstract class** `User` with overridden `login()`
• **Payment** is interface; implementations use polymorphism
• **Order** has composition with `OrderItem`

UML Sketch:
User (abstract)

login()
↓
Customer | Admin

Purchase 1..*

Cart — contains — Product ..

Order — has — OrderItem ..
Payment «interface»
↑ ↑
CreditCardPayment UPIPayment

typescript
Copy code

🧠 EXAMPLE 3: Vehicle Rental System

Key Classes:
• Vehicle (abstract) → Car, Bike
• Customer
• RentalOrder
• Payment (aggregation)
• Feedback

Design Choice:
• `Vehicle` is abstract with abstract `calculateFare()`
• `Customer` can own multiple `RentalOrder`
• Composition: `RentalOrder` creates `Vehicle` temporarily

UML Sketch:
Vehicle (abstract)
– calculateFare()
↑
Car | Bike

Customer — places — RentalOrder ..
RentalOrder — uses — Vehicle (1)
Payment — processes — RentalOrder
Feedback — associated with — Customer

vbnet
Copy code

🧠 INTERVIEW QUESTIONS – DESIGN THOUGHTS

Q1: How do you decide between using an abstract class vs interface?
Q2: When to use aggregation vs composition in your design?
Q3: Why is delegation preferred over inheritance in some cases?
Q4: How do you model one-to-many vs many-to-many relationships in UML?
Q5: How would you handle extensibility in product features (e.g. adding new payment methods)?
Q6: How to apply SOLID principles to the design of the system above?
Q7: How would you scale the design for millions of users or distributed systems?

✅ SUMMARY

- Start with identifying entities and relationships
- Use UML to visualize associations, multiplicities, and visibility
- Maintain OOP constructs: encapsulation, inheritance, polymorphism
- Strike balance between flexibility (interfaces) and reusability (abstract class)