OOPS Interview Questions

(140 Questions with Practical Answers)

🟩 **STAGE 1: OOP Fundamentals**
🟨 **STAGE 2: Encapsulation**
🟧 **STAGE 3: Inheritance**
🟦 **STAGE 4: Polymorphism**
🟪 **STAGE 5: Abstraction**
⬛ **STAGE 6: Object Relationships**
🟫 **STAGE 7: Important OOP Keywords**
🟧 **STAGE 8: Comparison-Based Topics**

STAGE 1: OOP Fundamentals

A. Class & Object Basics

---

1. What is a class in Java?
Answer : A class is a blueprint or template for creating objects. It defines state and behavior (methods) of objects. Think of it as a cookie cutter, while objects are the cookies.

```
class Car {
    String brand;
    int speed;

    void drive() {
        System.out.println(brand + " is driving at " + speed + " km/h");
    }
}
```

2. What is an object in Java?
Answer: An object is a real-world instance of a class. It has its own state stored in memory and can use the class's methods to perform behavior.

```
Car myCar = new Car();
myCar.brand = "BMW";
myCar.speed = 100;
myCar.drive(); // BMW is driving at 100 km/h
```

3. How do you create an object in Java?
Ans: Objects are created using the new keyword, which allocates memory & calls the constructor.

Car car1 = new Car(); // Object created

Car car2 = new Car(); // Another object

4. What is the difference between a class and an object?

Class	Object
Blueprint or template	Instance of a class
Defines state and behavior	Holds actual values for state
Exists in code	Exists in memory (heap)
Can't perform action by itself	Can perform actions using methods

Example: Car is the class, myCar is the object.

5. What is the purpose of the new keyword in Java?
Answer:

- Allocates memory on the heap for a new object.
- Calls the constructor to initialize the object.

Car car = new Car(); // 'new' allocates memory and calls constructor

6. What happens in memory when an object is created?
Answer:

1. Memory allocated on heap for instance variables.
2. The constructor initializes variables.
3. The reference variable points to the memory location in the stack.

Stack: car -> Heap: Car object {brand: null, speed: 0}

7. Can you have a class without any attributes or methods?
Answer: Yes, Java allows empty classes. Sometimes used as marker classes.

```
class Marker {}
```

8. What are instance variables and instance methods?
Answer:

Instance Variables

- **Definition:** Variables that belong to an **instance (object)** of a class.
- **Scope:** Each object of the class has its **own copy** of instance variables.
- **Declaration:** Declared inside the class but **outside any method**, usually without static.
- **Access:** Accessed through the **object reference**.

```java
class Car {
    String color;   // instance variable
    int speed;      // instance variable
}
```

If you create two objects:

Car car1 = new Car();

Car car2 = new Car();

car1.color = "Red";

car2.color = "Blue";

- car1.color is "Red" and car2.color is "Blue" → each object has its **own copy**.

Instance Methods

- **Definition:** Methods that belong to an instance (object) of a class.
- **Access:** Can access instance variables and other instance methods directly.
- **Declaration:** Normal methods without the static keyword.
- **Example:**

```java
class Car {
    String color;

    void displayColor() {  // instance method
        System.out.println("Car color is: " + color);
    }
}
```

9. What is a static variable and how is it different from an instance variable?
Answer:

- **Static variable:** A static variable (also called a class variable) is a variable that belongs to the class rather than to any specific object. It is declared using the static keyword. All objects of the class share the same copy of a static variable. It is mainly used to store common properties or values that are shared among all instances of the class.

```java
class Car {
    static int numberOfCars; // shared by all Car objects
}
```

Difference Between Static and Instance Variables

Feature	Instance Variable	Static Variable
Belongs to	Object (each object has its own copy)	Class (shared by all objects)
Memory Allocation	Allocated when object is created	Allocated only once when class is loaded
Access	Through object reference	Through class name (or object)
Purpose	Store object-specific data	Store class-level/shared data

10. Can static methods access instance variables? Why or why not?

Answer:

No, static methods belong to the class, not any object. They cannot access instance variables directly because instance variables belong to objects.

```
class Demo {
    int x = 10;
    static void printX() {
        // System.out.println(x); // ❌ Error
    }
}
```

B. Constructors & Initialization

---

11. What is a constructor in Java?

Answer: A constructor is a special block of code used to initialize objects.

- It has no return type (not even void).
- The name must match the class name.

```
class Car {
    String brand;
    int speed;

    Car(String brand, int speed) { // constructor
        this.brand = brand;
        this.speed = speed;
    }
}

Car myCar = new Car("BMW", 120);
```

Tip: Always mention constructors are implicitly called when new is used.

12. What is the difference between a constructor and a method?

Constructor	Method
No return type	Has return type (even void)
Name = Class name	Name = any valid identifier
Called automatically when object is created	Called explicitly
Used to initialize object	Used to perform operations

13: What are the types of constructors in Java?

1. **Default Constructor (No-Arg Constructor)**
    - **Definition: A constructor that takes no arguments.**
    - **Purpose: Initializes objects with default values.**

```
class Car {
    String color;
    Car() { // default constructor
        color = "White";
    }
}
Car c = new Car(); // color will be White
```

2. Parameterized Constructor

- **Definition:** A constructor that accepts arguments to initialize an object with specific values.
- **Purpose:** Provides flexibility to create objects with different states.

```
class Car {
    String color;
    Car(String c) { // parameterized constructor
        color = c;
    }
}
Car c1 = new Car("Red");
Car c2 = new Car("Blue");
```

3. Copy Constructor *(Optional in Java, user-defined)*

- **Definition:** A constructor that **creates a new object by copying values from another object**.
- **Purpose:** To duplicate an existing object.

```
class Car {
    String color;
    Car(Car c) { // copy constructor
        color = c.color;
    }
}
Car c1 = new Car("Red");
Car c2 = new Car(c1); // c2.color will be Red
```

14. What is a default constructor? When is it created by the compiler?
Answer:

- A default constructor is a no-argument constructor automatically provided by the compiler if no constructors are defined.
- If you define any constructor, the compiler does not create a default one.

class Car { } // compiler adds: Car() {}

15. Can a constructor be private? If yes, where is it useful?
Answer: Yes — private constructors are used for:

- Singleton design pattern
- Static utility classes (like Math)

```
class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() {} // private constructor

    public static Singleton getInstance() {
        return instance;
    }
}
```

16. Can a class have multiple constructors? How?
Answer: Yes — constructor overloading allows multiple constructors with different parameters.

```
class Car {
    String brand;
    int speed;

    Car() { // no-arg constructor
        brand = "Unknown";
        speed = 0;
    }

    Car(String brand) { // single-arg constructor
        this.brand = brand;
        speed = 0;
    }

    Car(String brand, int speed) { // two-arg constructor
        this.brand = brand;
        this.speed = speed;
    }
}
```

17. Can constructors be overloaded?
Answer: Yes — just like methods, constructors can have different parameter lists.

- Overloading allows creating objects in different ways.

```
Car c1 = new Car();
Car c2 = new Car("BMW");
Car c3 = new Car("Audi", 150);
```

18. What happens if you define a parameterized constructor but not a default one?
Answer:

- Compiler does not provide a default constructor.
- Trying to create an object with no arguments will cause a compile-time error.

```
class Car {
    Car(String brand) {}
}

// Car c = new Car(); // ❌ Error
```

19. Can a constructor call another constructor? How?
Answer: Yes — use this() to call another constructor in the same class.

- Must be the first statement in the constructor.

```
class Car {
    String brand;
    int speed;

    Car() {
        this("Unknown", 0); // calling parameterized constructor
    }

    Car(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }
}
```

20. What is constructor chaining?
Answer: Constructor chaining = calling one constructor from another (either within the same class using this() or from the parent class using super()).

- Helps avoid code duplication and initialize objects efficiently.

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

class Car extends Vehicle {
    Car() {
        super(); // parent constructor called
        System.out.println("Car created");
    }
}
```

21. Can we inherit constructors from the parent class?
Answer:

- No, constructors are not inherited.
- A subclass can call a parent constructor using super(), but cannot directly inherit it.

C. Static, this Keyword, Instance vs Class Variables, Getters & Setters

---

22. What is this keyword in Java?

Answer :

- this refers to the current object.
- Used to differentiate instance variables from parameters, call other constructors, or pass the current object.

```
class Person {
    String name;

    Person(String name) {
        this.name = name; // 'this.name' refers to instance variable
    }

    void printPerson() {
        System.out.println(this.name);
    }
}

Person p = new Person("Alice");
p.printPerson(); // Alice
```

23. What is a static variable in Java?

Answer:

- Belongs to the class, not individual objects.
- Shared by all instances of the class.
- Stored in method area (class memory), not heap.

```
class Counter {
    static int count = 0;

    Counter() {
        count++;
    }
}

Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println(Counter.count); // 2
```

24. What is a static method in Java?

Answer: It belongs to class, can be called without creating an object & can access only static variables/methods. It cannot use **this** because no instance is associated.

```
class Demo {
    static int x = 10;
    static void printX() {
        System.out.println(x);
    }
}

Demo.printX(); // 10
```

25. Difference between instance variables and static variables?

Instance Variable	Static Variable
Belongs to object	Belongs to class
Unique per object	Shared across all objects
Stored in heap	Stored in method area
Accessed via object reference	Accessed via class name

26. What are getters and setters? Why do we use them?

Answer :

- Methods used to access and modify private variables (encapsulation).
- Useful for validation and controlling access.

```
class Person {
    private String name;

    public String getName() { return name; }
    public void setName(String name) {
        if(name != null && !name.isEmpty()) this.name = name;
    }
}

Person p = new Person();
p.setName("Bob");
System.out.println(p.getName()); // Bob
```

27. Can a static method access instance variables?

Answer:

- ❌ No, because instance variables belong to objects, and static methods belong to the class.
- To access instance variables, pass an object reference.

```
class Demo {
    int x = 5;
    static void printX(Demo obj) {
        System.out.println(obj.x); // works
    }
}
```

28. Can this be used in a static method?
Answer:

- ❌ No, this refers to the current object, but static methods don't belong to any object.

```
class Demo {
    static void test() {
        // System.out.println(this); // ❌ Error
    }
}
```

29. What happens if you don't use getters/setters and make variables public?
Answer:

- Encapsulation is broken → anyone can modify directly → no validation/control → higher chance of bugs.
- Using getters/setters makes your class maintainable and secure.

30. Can static variables and methods be accessed via objects?
Answer:

- ✅ **Yes, but not recommended — should access via class name to indicate shared nature.**

```
class Demo {
    static int count = 0;
}

Demo obj = new Demo();
System.out.println(obj.count);      // works but not preferred
System.out.println(Demo.count);    // recommended
```

STAGE 2: Encapsulation

A. Core Concept

---

31. What is encapsulation in Java?
Answer : Encapsulation is the mechanism of wrapping data (variables) and behavior (methods) together and restricting direct access to some components.

- Ensures controlled access to the object's state.

```
class Person {
    private String name; // hidden
    public void setName(String name) { this.name = name; } // controlled access
    public String getName() { return name; }
}
```

32. Why is encapsulation important in OOP?

- Protects object state from unintended modifications.
- Makes code more maintainable and flexible.
- Supports modular programming and team collaboration.\

33. How does encapsulation differ from abstraction?

- **Encapsulation:** Hides implementation details at the object level using access modifiers.
- **Abstraction:** Hides complexity at class/method level, showing only what an object can do.

34. What is data hiding and how does Java achieve it?

Answers: Data hiding is a key part of encapsulation — it prevents outside classes from accessing the internal state directly. In Java, we achieve this using private access modifiers on variables and public getters/setters to provide controlled access.

- Data hiding is restricting access to object state.
- Achieved using private variables + public getters/setters.

```
class BankAccount {
    private double balance; // hidden

    public double getBalance() { return balance; } // controlled read
    public void deposit(double amount) {
        if(amount > 0) balance += amount;
    }
}
```

35. What are the key features that enable encapsulation in Java?

- **Private variables:** Restrict direct access.
- **Public/protected getters and setters:** Controlled access.
- **Final keyword:** Helps create immutable fields.
- **Access modifiers:** Determine visibility (public, private, protected, default).

36. Can encapsulation exist without access modifiers?

Answer: Not fully. While methods and variables are package-private by default, true encapsulation requires control over visibility, which is only possible with access modifiers like private or protected. Without them, other classes in the same package can modify variables directly, breaking encapsulation.

37. Is encapsulation only about making variables private?

Ans: No. While making variables private is a key part, encapsulation also involves:

- Providing controlled access through methods
- Possibly making fields immutable for better control

38. How does encapsulation improve code maintainability and flexibility?

Ans: Encapsulation allows you to change internal implementation without affecting code that uses the class. For example:

```
class Employee {
    private double salary;

    public void setSalary(double salary) {
        if(salary > 0) this.salary = salary;
    }

    public double getSalary() {
        return salary * 1.1; // internal logic can change
    }
}
```

39. What will happen if we make instance variables public?

Ans: If variables are public, any class can modify them directly. This can:

- Cause invalid states
- Break object integrity
- Lead to tightly coupled code

40. What are the disadvantages of not using encapsulation?

- Increased risk of invalid data
- Harder to maintain and refactor code
- Poor modularity and reusability
- Makes team collaboration more error-prone

B. Practical Implementation

---

41. What are getters and setters in Java?

- Getters: Methods that return the value of private variables.
- Setters: Methods that modify the value of private variables.
- Enable controlled access to fields.

42. Why do we use getters and setters instead of accessing variables directly?

- To validate input before setting values.
- To encapsulate data, hiding implementation.
- To allow future changes without affecting external code.

43. Can we have only a getter or only a setter in a class? Why?

Ans: ✅ Yes.

Getter-only → Read-only property (e.g., ID).
Setter-only → Write-only property (e.g., password).

```
class User {
    private String password;
    public void setPassword(String pwd) { this.password = pwd; } // write-only
}
```

44. Can you write a simple Java class that demonstrates encapsulation?

```
class Employee {
    private String name;
    private double salary;

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

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

45. How do access modifiers affect encapsulation?

- **Private → fully hidden, best for encapsulation**
- **Protected → accessible to subclasses, partial access**
- **Default → accessible within package**
- **Public → breaks encapsulation if applied to variables**

---

46. What is a POJO and how does it represent encapsulation?

Ans: POJO = Plain Old Java Object.

- Typically has private fields, public getters/setters, and no special annotations or logic.
- Represents encapsulation because fields are hidden and accessed only via controlled methods.

```
class Student {
    private String name;
    private int id;

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

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
}
```

47. How does encapsulation support immutability? Give an example.

Ans: Encapsulation allows you to hide fields and prevent setters, creating objects that cannot change state once created.

```
final class Person {
    private final String name;

    Person(String name) { this.name = name; }

    public String getName() { return name; } // no setter → immutable
}
```

48. How can you make a class immutable using encapsulation?

- Make class final
- Make all fields private final
- Provide no setters, only getters
- Ensure mutable fields are cloned before returning

49. How does encapsulation help in teams or large codebases?

- Limits who can modify object state
- Reduces bugs and unintended side-effects
- Allows modular development, one team can work on getters/setters without breaking others

50. Give a real-world example representing encapsulation.

- ATM machine: You don't see the internal code or money handling, you interact via buttons/screens only.
- Analogy: Object state is hidden; access controlled via methods (APIs).

STAGE 3: Inheritance

51. What is inheritance in Java?

Ans: Inheritance is an OOP concept where a class (child/subclass) acquires the properties and behaviors of another class (parent/superclass). It promotes code reuse, extensibility, and establishes a hierarchical relationship between classes.

```
class Animal {
    void eat() {
        System.out.println("Animal eats");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // inherited from Animal
        dog.bark(); // specific to Dog
    }
}
```

Here, Dog inherits eat() from Animal — no need to write it again.

---

52. Why do we use inheritance?

- Code Reuse: Avoid duplicating code across classes.
- Extensibility: New classes can be added with minimal changes.
- Polymorphism Support: Enables method overriding and runtime behavior changes.
- Logical Hierarchy: Helps model real-world relationships naturally.

Example: All vehicles share some features like start(). Instead of writing start() in each vehicle class, create a parent Vehicle class, and child classes inherit it.

53. What is the difference between IS-A and HAS-A relationships?

Type	Description	Example
IS-A	**Inheritance relationship; subclass is a type of superclass**	**Dog IS-A Animal**
HAS-A	**Composition relationship; class has an object of another class**	**Car HAS-A Engine**

```
class Engine {}
class Car {
    private Engine engine; // HAS-A relationship
}
```

- **IS-A → inheritance**
- **HAS-A → composition**

54. What different types of inheritance exist in Java? Explain each.

1. **Single Inheritance: One class inherits from another.**

```
class Parent {}
class Child extends Parent {}
```

2. **Multilevel Inheritance: A chain of inheritance.**

```
class GrandParent {}
class Parent extends GrandParent {}
class Child extends Parent {}
```

3. **Hierarchical Inheritance: Multiple subclasses inherit from a single superclass.**

```
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
```

4. **Multiple Inheritance (through interfaces):** One class implements multiple interfaces **(because Java classes cannot extend multiple classes).**

```
interface A {}
interface B {}
class C implements A, B {}
```

55. Why is multiple inheritance not allowed in Java using classes?

Ans: Java does not allow multiple inheritance with classes to avoid ambiguity and the "Diamond Problem".

Example of ambiguity:

```
Class A { void show() { System.out.println("A"); } }
Class B { void show() { System.out.println("B"); } }
Class C extends A, B {} // Which show() should C inherit?
```

Java avoids this problem by disallowing multiple class inheritance but allows interfaces for multiple inheritance of behavior.

56. How does Java support multiple inheritance using interfaces?

- A class can implement multiple interfaces, thus inheriting behavior from multiple sources.

- This allows multiple inheritance of type/behavior, but not implementation conflicts.

```
interface A { void methodA(); }
interface B { void methodB(); }

class C implements A, B {
    public void methodA() { System.out.println("A"); }
    public void methodB() { System.out.println("B"); }
}
```

Here, C inherits contracts from both interfaces and provides its implementation.

57. What is the super keyword? Give an example.

- super is a reference to the parent class.
- It is used to:
    1. Access parent class methods
    2. Access parent class fields
    3. Call parent class constructor

```
class Parent {
    int x = 10;
    void show() { System.out.println("Parent show"); }
}

class Child extends Parent {
    int x = 20;
    void show() {
        super.show(); // calls Parent's show()
        System.out.println("Child show");
        System.out.println(super.x); // access parent's x
    }
}
```

58. Can private members of a superclass be inherited? Why or why not?

Ans: Private members are not visible to subclasses, so they cannot be directly inherited. However, they can be accessed indirectly via public/protected getters and setters.

```java
class Parent {
    private int secret = 123;
    public int getSecret() { return secret; }
}

class Child extends Parent {
    void revealSecret() {
        System.out.println(getSecret()); // allowed
    }
}
```

---

59. Can constructors be inherited?

Ans: No, constructors are not inherited, because a constructor is specific to the class.

- Each class must define its own constructor.
- Subclasses can call superclass constructors using super().

60. What is constructor chaining in inheritance?

Ans:

- **Constructor chaining is the process of calling one constructor from another.**
- **In inheritance, a subclass constructor calls the parent constructor using super().**
- **Ensures the parent part of the object is initialized first.**

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

class Child extends Parent {
    Child() {
        super(); // call parent constructor
        System.out.println("Child constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

Output:

Parent constructor
Child constructor

B. Implementation / Coding Questions – Inheritance

---

61. Write a simple example of single inheritance (Parent → Child) in Java.

Ans: Single inheritance is when one class extends another. The child class inherits all accessible fields and methods from the parent class.

```
class Parent {
    void greet() {
        System.out.println("Hello from Parent");
    }
}

class Child extends Parent {
    void greetChild() {
        System.out.println("Hello from Child");
    }
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.greet();      // inherited from Parent
        c.greetChild(); // specific to Child
    }
}
```

Explanation:
The Child class inherits greet() from Parent. This avoids code duplication and demonstrates the IS-A relationship: Child IS-A Parent.

62. Write an example of multilevel inheritance (GrandParent → Parent → Child).

Ans: Multilevel inheritance is a chain where a class inherits from another, which in turn inherits from another.

```java
class GrandParent {
    void message() { System.out.println("GrandParent Message"); }
}

class Parent extends GrandParent {
    void messageParent() { System.out.println("Parent Message"); }
}

class Child extends Parent {
    void messageChild() { System.out.println("Child Message"); }
}

public class Test {
    public static void main(String[] args) {
        Child child = new Child();
        child.message();        // GrandParent
        child.messageParent();  // Parent
        child.messageChild();   // Child
    }
}
```

Explanation: Child inherits all methods from Parent and GrandParent, showing code reuse and logical hierarchy.

63. Show how to call the superclass constructor from a subclass.

Ans: In Java, a subclass can call a parent class constructor using super(). This is important for initializing parent class fields.

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

class Child extends Parent {
    Child() {
        super(); // call Parent constructor
        System.out.println("Child Constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        new Child();
    }
}
```

Output:
Parent Constructor
Child Constructor

64. Can we override a static method in Java? Explain with an example.

Ans: Static methods cannot be overridden. They belong to the class, not the instance. If you define a static method with the same signature in a subclass, it's called method hiding, not overriding.

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

class Child extends Parent {
    static void staticMethod() { System.out.println("Child static"); }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.staticMethod(); // calls Parent's method
    }
}
```

Output: Parent static

Explanation: The method is resolved at compile time, not runtime. True overriding requires instance methods.

65. What happens if a subclass defines a method with the same name but different parameters as the superclass?

Ans: This is called method overloading, not overriding.

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

class Child extends Parent {
    void display(String msg) { System.out.println("Child display: " + msg); }
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();          // Parent method
        c.display("Hello");   // Child method with parameter
    }
}
```

Explanation:
Overloaded methods are distinguished by parameters and are resolved at compile-time.

---

66. Can a subclass access protected members of the superclass in another package?

Ans: ✅ Yes, but only through inheritance. Protected members are accessible to subclasses even in different packages, but not through object references.

```
package pkg1;
public class Parent {
    protected int data = 100;
}

package pkg2;
import pkg1.Parent;

class Child extends Parent {
    void show() {
        System.out.println(data); // accessible
    }
}
```

Explanation: Protected allows subclass access across packages but maintains encapsulation.

---

67. Write an example demonstrating method overriding in Java.

Overriding happens when a subclass provides a new implementation for a method inherited from the parent.

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

class Child extends Parent {
    @Override
    void show() { System.out.println("Child show"); }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show(); // runtime polymorphism
    }
}
```

Output: Child show

Explanation: The child's method is called at runtime, demonstrating dynamic polymorphism.

68. Explain what happens when a subclass hides a superclass field with the same name.

Ans: Field hiding occurs when a subclass declares a field with the same name as in its parent.

```java
class Parent {
    int x = 10;
}

class Child extends Parent {
    int x = 20;
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        System.out.println(c.x);          // 20 (child)
        System.out.println(((Parent)c).x); // 10 (parent)
    }
}
```

Explanation: Field hiding does not involve polymorphism; the reference type determines which field is accessed.

69. How do you prevent a class from being subclassed?

Ans: Use the final keyword in the class.

```
final class Parent {}
class Child extends Parent {} // Compile-time error
```

Explanation: This ensures security and immutability in some cases (e.g., String class in Java).

---

70. Give a real-world example where inheritance improves code reuse and maintainability.

Ans: Consider a Vehicle superclass with start() and stop() methods. Subclasses like Car, Bike, and Bus inherit these methods.

- No need to write start() in each class → code reuse
- Adding new vehicle types is easy → maintainability
- Polymorphism allows treating all vehicles uniformly:

```
Vehicle[] vehicles = {new Car(), new Bike(), new Bus()};
for(Vehicle v : vehicles) v.start();
```

Explanation: This avoids repetitive code and demonstrates hierarchy, reuse, and polymorphism together.

STAGE 4: Polymorphism

71. What is polymorphism in Java?

Ans: Polymorphism literally means “many forms”. In OOP, it allows a single entity (method or object) to take multiple forms. It’s a way to design flexible and extensible code where one interface can work with different types of objects.

Example:

```java
class Animal {
    void sound() { System.out.println("Some sound"); }
}

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

class Cat extends Animal {
    void sound() { System.out.println("Meow"); }
}

public class Test {
    public static void main(String[] args) {
        Animal a1 = new Dog();
        Animal a2 = new Cat();

        a1.sound(); // Bark
        a2.sound(); // Meow
    }
}
```

Here, the same reference type Animal behaves differently depending on the actual object, demonstrating polymorphism.

72. Why is polymorphism important in OOP?

Ans: Polymorphism is crucial because it:

1. Reduces code complexity: One method/interface can handle multiple object types.
2. Enhances flexibility and maintainability: New types can be added without changing existing code.
3. Supports dynamic behavior: Subclass-specific behavior can replace superclass behavior at runtime.
4. Encourages loose coupling: Code depends on interfaces or parent classes rather than concrete implementations.

73. What are the two types of polymorphism in Java?

1. Compile-time (Static) Polymorphism
2. Runtime (Dynamic) Polymorphism

- Compile-time → resolved by the compiler (method overloading, operator overloading)
- Runtime → resolved at runtime (method overriding)

74. What is compile-time (static) polymorphism?

Ans: This occurs when the compiler decides which method to call at compile time. In Java, this is mainly achieved by method overloading**.**

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

public class Test {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2,3));     // int version
        System.out.println(calc.add(2.5,3.5)); // double version
    }
}
```

Explanation: The compiler selects the appropriate add method based on parameters — no runtime decision is needed.

75. What is runtime (dynamic) polymorphism?

Ans: Runtime polymorphism occurs when the method that gets executed is decided at runtime, not compile time. This is usually achieved using method overriding and inheritance.

```
class Animal {
    void sound() { System.out.println("Some 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(); // runtime decides: Dog's sound() is called
    }
}
```

The JVM determines at runtime which sound() method to call based on the actual object type (Dog).

76. How does method overloading achieve compile-time polymorphism?

Ans: By defining multiple methods with the same name but different parameters, the compiler can determine which method to call based on argument types and count.

```
class Printer {
    void print(int a) { System.out.println("Int: " + a); }
    void print(String s) { System.out.println("String: " + s); }
}

public class Test {
    public static void main(String[] args) {
        Printer p = new Printer();
        p.print(5);    // calls print(int)
        p.print("Hi"); // calls print(String)
    }
}
```

Explanation: The compiler resolves the method call during compilation → static polymorphism.

77. How does method overriding achieve runtime polymorphism?

Ans: When a subclass provides a specific implementation of a method inherited from the parent class, the JVM decides at runtime which method to execute.

```
class Vehicle {
    void start() { System.out.println("Vehicle starting"); }
}

class Car extends Vehicle {
    @Override
    void start() { System.out.println("Car starting"); }
}

public class Test {
    public static void main(String[] args) {
        Vehicle v = new Car();
        v.start(); // runtime decides: Car's start() executes
    }
}
```

This allows flexible behavior depending on the object.

78. Can constructors be overloaded? Is it polymorphism?

Ans: ✅ Yes, constructors can be overloaded (same name, different parameters).
❌ But constructor overloading is compile-time polymorphism, not runtime.

```
class Person {
    Person() { System.out.println("Default constructor"); }
    Person(String name) { System.out.println("Constructor with name: " + name); }
}
```

Explanation: The compiler decides which constructor to call based on parameters → static polymorphism.

79. Can private methods be overridden? Why or why not?

Ans: No. Private methods belong to the class itself, not the subclass. They are not visible outside the class, so overriding is impossible.

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

class Child extends Parent {
    private void show() { System.out.println("Child show"); }
}
```

Here, Child.show() is not overriding, it's a completely new method.

80. What is a covariant return type in method overriding?

Ans: Covariant return type allows the overridden method in subclass to return a subtype of the superclass method's return type.

```
class Animal {}
class Dog extends Animal {}

class Parent {
    Animal getAnimal() { return new Animal(); }
}

class Child extends Parent {
    @Override
    Dog getAnimal() { return new Dog(); } // covariant return
}
```

81. What is the difference between method overloading and method overriding?

Feature	Method Overloading	Method Overriding
Definition	Same method name but different parameters in the same class (or subclass).	Subclass provides a new implementation of a method from its superclass.
Polymorphism Type	Compile-time (static)	Runtime (dynamic)
Parameters	Must be different (number or type of parameters).	Must be same (name, parameters).
Return Type	Can vary (but if types are same, no issue).	Must be same or covariant return type.
Access Modifier	Any access modifier	Cannot reduce visibility (can increase).
Exceptions	Can throw any exception.	Must not throw broader checked exceptions than superclass.

Example – Overloading:

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

Example – Overriding:

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

82. Can private, final, or static methods be overridden? Why or why not?

Ans: Private methods → ❌ Cannot be overridden because they are not visible to subclasses. They belong to the class itself.

- Final methods → ❌ Cannot be overridden because final prevents modification.
- Static methods → ❌ Cannot be overridden. Defining a static method with the same name in a subclass hides the superclass method; it's method hiding, not overriding.

```
class Parent {
   private void privateMethod() {}
   final void finalMethod() {}
   static void staticMethod() {}
}
class Child extends Parent {
   // All above cannot be overridden
}
```

83. What is the difference between compile-time and runtime polymorphism in terms of method binding?

Ans: 1. Compile-time polymorphism → Early binding

- The compiler decides which method to call based on method signature (overloading).

2. **Runtime polymorphism** → Late binding

- JVM decides which method to call based on actual object type at runtime (overriding).

Example – Runtime (late binding):

```
Animal a = new Dog(); // actual type Dog
a.sound();            // JVM calls Dog's sound() at runtime
```

84. How does Java resolve which method to call at runtime in case of overriding?

- Java uses dynamic method dispatch: the JVM checks the actual object type (not reference type) at runtime and invokes the overridden method of the subclass.
- Reference type determines what methods are visible, but object type determines which implementation is executed.

```
Animal a = new Dog(); // Reference type: Animal, Object type: Dog
a.sound();          // JVM calls Dog's sound()
```

85. Can constructors participate in runtime polymorphism? Why or why not?

Ans: ❌ No. Constructors cannot be overridden, and hence cannot participate in runtime polymorphism.

- **Constructor is specific to the class it belongs to.**
- **JVM does not use dynamic method dispatch for constructors.**
- **Constructors are used only to initialize objects, not to provide polymorphic behavior.**

86. What is dynamic method dispatch in Java, and how is it related to polymorphism?

Ans: Dynamic Method Dispatch (DMD) is the mechanism by which Java resolves overridden method calls at runtime.

- It is the core of runtime polymorphism.
- JVM looks at actual object type and calls the appropriate method.

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

Animal a = new Dog();
a.sound(); // Dog's sound() executed
```

Explanation:

- Animal a reference → visible methods
- Dog object → actual method executed at runtime

87. How does polymorphism improve code maintainability and flexibility in large applications?

- Decouples code from implementation: Code depends on interfaces or superclass, not specific implementations.
- Easy to extend: New classes can be added without changing existing logic.
- Reduces code duplication: Same method calls can work for multiple object types.
- Supports runtime flexibility: Behavior can change dynamically based on object type.

Example – Payment System:

interface Payment { void pay(double amount); }

class CreditCard implements Payment { public void pay(double amt){
System.out.println("Paid via card"); } }

class UPI implements Payment { public void pay(double amt){ System.out.println("Paid via UPI"); } }

Payment p = new UPI();
p.pay(1000); // runtime decides which pay() method to execute

88. What is the difference between polymorphism in Java and C++?

Feature	Java	C++
Method Binding	Compile-time & Runtime	Compile-time & Runtime
Static Methods	Cannot be overridden	Can hide static methods (similar to Java)
Virtual Keyword	Not needed, all non-final instance methods are virtual	Required to mark a method as virtual for runtime polymorphism
Multiple Inheritance	Not allowed for classes, only via interfaces	Allowed for classes, can lead to diamond problem

Explanation: Java simplifies runtime polymorphism by making all instance methods virtual by default.

---

89. Can you override a superclass method and change its access modifier? What are the rules?

- Rules for overriding access modifiers:

    1. You cannot reduce visibility.
        - public → always public in subclass
        - protected → can be protected or public in subclass
        - default/package-private → can become protected or public
    2. You can increase visibility but not decrease it.

class Parent { protected void show() {} }
class Child extends Parent { public void show() {} } // ✅ allowed

90. Give a real-world analogy to explain polymorphism in Java.

Ans: Analogy – Remote Control for Devices:

- One remote control can work for TV, AC, Fan.
- Pressing the “Power” button sends the same signal, but each device responds differently: TV turns on, AC starts, Fan spins.

Explanation:

- The remote is like the reference type (superclass or interface)
- The device is like the actual object

- Dynamic method dispatch decides what happens when the “Power” button is pressed → runtime polymorphism.

STAGE 5: Abstraction

91. What is abstraction in Java?

Ans: Abstraction is the OOP concept of hiding implementation details and showing only essential functionality to the user. It allows you to focus on what an object does, not how it does it.

- **Example analogy:** When you drive a car, you just press the accelerator, brake, or steering; you don't need to know how the engine works internally.
- In Java, abstraction is implemented using **abstract classes** and **interfaces**.

92. How is abstraction different from encapsulation?

Feature	Abstraction	Encapsulation
Purpose	Hides implementation details, shows only functionality	Hides internal state and protects data from direct access
How implemented	Abstract classes, Interfaces	Access modifiers (private, protected) + getters/setters
Focus	"What an object does"	"How an object stores/manages its data"
Example	Vehicle class exposes start() but hides engine details	Car class makes speed private and provides getSpeed()

93. Why do we need abstraction in OOP?

- To **reduce complexity** by exposing only essential operations.
- To **increase code reusability**; different implementations can be used without changing client code.
- To **support polymorphism**; the same interface can represent multiple types.
- To **improve maintainability**; changes to implementation do not affect the client.

94. What is an abstract class in Java?

- A class declared with the abstract keyword.
- Can contain **abstract methods** (no body) and **concrete methods** (with body).
- Cannot be instantiated directly.
- Designed to be **extended by subclasses** that provide implementations for abstract methods.

95. Can an abstract class have a constructor? If yes, why is it useful?

Ans: ✅ Yes, abstract classes **can have constructors**.

- **Why:** Constructors are used to initialize **fields common to all subclasses**.
- **Example:** If Vehicle is abstract, its constructor can initialize speed and fuelType, and all subclasses like Car and Bike will inherit these initializations.

96. Can an abstract class have both abstract and non-abstract (concrete) methods?

✅ Yes.

- Abstract methods define **what subclasses must implement**.
- Concrete methods provide **default behavior** that can be reused.

Example: Vehicle has start() abstract, but stop() concrete. Subclasses implement start(), and inherit stop().

97. Can you instantiate an abstract class? Why or why not?

❌ No, you cannot directly create an object of an abstract class.

- **Reason:** Abstract class may have incomplete implementations (abstract methods), so it doesn't define a complete object.
- You **must extend it in a subclass** and implement abstract methods to instantiate.

98. What is an interface in Java?

- An interface is a **pure abstraction** mechanism that defines **a contract** — methods that a class must implement.
- From Java 8 onwards, interfaces can also have **default, static, and private methods**.
- A class can **implement multiple interfaces**, supporting multiple inheritance.

99. Can interfaces have method implementations in Java? If yes, how (default, static, private methods)?

Ans: ✅ Yes:

- **Default methods:** have a body, can be inherited by implementing classes, can be overridden.
- **Static methods:** belong to interface, cannot be overridden by implementing classes.
- **Private methods:** used inside interface to share code among default methods, not accessible outside.

100. What is the difference between an abstract class and an interface in Java?

Feature	Abstract Class	Interface
Multiple inheritance	❌ No	✅ Yes
Methods	Abstract + concrete	Default, static, abstract (all methods are public by default)
Variables	Instance variables allowed	Only public static final (constants)
Constructor	Allowed	❌ Not allowed
Access modifiers	Any	Methods: public by default, can have private (Java 9+)

101. Can a class extend multiple abstract classes? Why or why not?

Ans: ❌ No, Java does **not support multiple inheritance of classes** (abstract or concrete) to avoid **diamond problem**.

- **Solution:** Use **interfaces** to achieve multiple inheritance.

102. Can a class implement multiple interfaces? How is this different from multiple inheritance in classes?

Ans: ✅ Yes.

- A class can implement multiple interfaces.
- Unlike multiple class inheritance, **there's no diamond problem**, because interfaces contain no state (mostly abstract methods).

103. Can an interface extend another interface? Can it extend multiple interfaces?

Ans: ✅ Yes.

- One interface can extend multiple interfaces.
- This allows **combining multiple contracts into a single interface**.

104. Can an abstract class implement an interface? Explain.

Ans: ✅ Yes.

- An abstract class can **implement an interface partially** (provide some method implementations) and leave the rest to subclasses.

105. What is the purpose of default methods in interfaces?

- Introduced in Java 8 to allow **adding new methods to interfaces without breaking existing implementations**.
- Provides **concrete behavior** in interfaces.

106. Can static methods in interfaces be overridden in implementing classes? Why or why not?

Ans: ❌ No.

- Static methods belong to **interface itself**, not instances.
- They **cannot participate in polymorphism** or be overridden.

---

107. Can a private method exist in an interface? How is it used?

Ans: ✅ Yes, Java 9+ allows **private methods in interfaces**.

- Purpose: **reusable code among default and static methods**, keeping implementation hidden.

108. Can you declare variables in an interface? What are their default modifiers?

Ans: Yes, **all variables in an interface are implicitly:**

    - public
    - static
    - final (constant)
- They cannot be instance variables.

109. How does abstraction help achieve polymorphism in Java?

Ans:

- By exposing **common interfaces or abstract classes**, client code can **treat different concrete objects uniformly**.
- Example: Shape s = new Circle(); Shape s2 = new Rectangle(); s.draw(); → same method call, different behavior at runtime.

110. How does abstraction improve code maintainability, flexibility, and scalability in large projects?

- **Maintainability:** Changes in concrete implementations don't affect clients.
- **Flexibility:** New classes can implement interfaces without changing existing code.
- **Scalability:** Supports **polymorphism**, allowing a system to grow and integrate new modules easily.

STAGE 6: Object Relationships

111. What is object association in Java?

Ans: Association defines a **relationship between two separate classes** that are connected through their objects.
It represents a **“uses-a” or “works-with”** relationship.
Example: A Teacher teaches multiple Students. Both can exist independently.

112. What is aggregation? How is it different from association?

Ans: Aggregation is a special form of association that represents a “has-a” relationship with shared ownership.

- Both objects can exist independently.
- Example: A Library has Books. If the Library is deleted, the Books can still exist.

Difference:

- Association = General relationship
- Aggregation = Whole–part relationship with **independent lifecycles**

113. What is composition? How is it different from aggregation?

Ans: It is a strong form of aggregation where one object cannot exist without the other.

- Example: A House has Rooms. If the House is destroyed, the Rooms are too.
- **Lifecycle:** A child object’s lifecycle is bound to the parent.

Difference:

- Aggregation → weak relationship (independent lifecycles)
- Composition → strong relationship (dependent lifecycle)

114. Can you give real-world examples of association, aggregation, and composition?

Relationship Type	Real-World Example	Description
Association	Teacher ↔ Student	Both exist independently but interact
Aggregation	Library → Book	Books can exist even if library closes
Composition	House → Room	Rooms don’t exist without the house

115. What is the difference between HAS-A and IS-A relationships?

Type	Description	Example
IS-A	Inheritance relationship between classes	Dog **is a** Animal
HAS-A	Association or composition relationship	Car **has a** Engine

Key Difference:

- **IS-A:** established via extends (inheritance)
- **HAS-A:** established via object references (composition/aggregation)

116. When should you prefer composition over inheritance?

Ans: Prefer **composition** when:

- You want **flexibility** and **loose coupling**.
- Behavior should be **changed at runtime** (by replacing components).
- You don't want to expose all superclass behavior.
- Example: Instead of inheriting Engine, Car **has an** Engine object → you can easily swap engine types.

117. What is dependency in OOP?

Ans: Dependency means one class **depends on another** to perform its function.
It's a **temporary association** (used only when needed).
Example: Driver depends on Car to drive → if car changes, driver's behavior may change.

118. How does dependency differ from aggregation?

Concept	Dependency	Aggregation
Nature	Temporary (method-level)	Structural (class-level)
Lifetime	Exists during method execution	Exists as a class member
Example	PaymentService depends on BankAPI	Library aggregates Book objects

119. Can a class have multiple associations with another class?

Ans: ✅ Yes.
A class can have multiple associations with another class if they represent **different roles or relationships**.
Example:

- A Teacher may **teach** a Student (association 1)
- A Teacher may also **mentor** a Student (association 2)

So, multiple associations between the same classes are allowed if their **purpose differs**.

---

120. Can aggregation exist without a "has-a" relationship?

Ans: ❌ No.
Aggregation **always implies** a "has-a" relationship because it represents **ownership** or **whole-part** structure.
If two classes are only **loosely connected** without ownership, that's a simple **association**, not aggregation.

✅ **Summary Table for Quick Recall:**

Relationship	Type	Ownership	Lifecycle Dependency	Example
Association	Uses-a	No	Independent	Teacher–Student
Aggregation	Has-a (weak)	Shared	Independent	Library–Book
Composition	Has-a (strong)	Exclusive	Dependent	House–Room
Dependency	Uses temporarily	No	Temporary	Driver–Car

Stage 7: Important OOP Keywords

121. Can a constructor be declared final, static, or abstract? Why or why not?

Ans: No.

- final: Constructors are not inherited, so making them final makes no sense.
- static: Constructors are called to create objects, but static methods belong to the class, not objects.
- abstract: Constructors must have a body, while abstract methods don't.

Hence, none of these modifiers are valid for constructors.

122. Can we use this() and super() together in the same constructor? Why or why not?

Ans: No. Both this() and super() must be the first statement inside a constructor.
Java doesn't allow two first statements, so you can only call one — either another constructor of the same class (this()) or the parent constructor (super()).

123. What is the difference between a static block and an instance initializer block?

Feature	Static Block	Instance Initializer Block
When executed	**Once when class is loaded**	**Every time an object is created**
Access to instance members	**❌ No**	**✅ Yes**
Use case	**Initialize static variables**	**Common initialization code for all constructors**

Example:

```
class Demo {
   static { System.out.println("Static block"); }
   { System.out.println("Instance block"); }
}
```

Output (for 2 objects):

```
Static block
Instance block
Instance block
```

124. What happens if you declare a static variable inside a method?

Ans: It's not allowed in Java.
Static variables belong to a class, not to a specific method.
Local variables inside methods are stored in the stack, while static variables live in the method area (class memory) — hence they can't coexist in a local context.

125. Can a class be both abstract and final at the same time?

Ans: No.

- abstract → means "incomplete" and must be subclassed.
- final → means "cannot be subclassed."
  These are opposite in meaning, so Java doesn't allow both together.

126. Can a static method be abstract? Why or why not?

Ans: No.
Static methods belong to the class, while abstract methods are meant to be overridden by subclasses.
Since static methods cannot be overridden, combining static and abstract makes no sense.

127. Can a static method access non-static data members or methods?

Ans: No, because static methods belong to the class, not to any particular object.
They can only access static members directly.
To access instance variables, they must create an object first.

Example:

```
class Test {
    int x = 5;
    static void show() {
        // System.out.println(x); ❌ Not allowed
        Test obj = new Test();
        System.out.println(obj.x); // ✅ Allowed
    }
}
```

128. What happens if you call this() or super() from a normal method (not a constructor)?

Ans: You cannot call this() or super() from a normal method — it causes a compile-time error. These calls are only allowed inside constructors, and must be the first statement.

✅ Why?
Because this() and super() are used to control object construction — they determine which constructor is executed first in the chain. Once the object is constructed, there's no need (or logical sense) to invoke them again.

129. Can you override a method and make it static in the subclass?

Ans: No.
A static method cannot be overridden, because static methods are class-level — not tied to an object.
However, you can redeclare a static method in the subclass — this is called method hiding, not overriding.

Example:

```
class Parent {
    static void show() { System.out.println("Parent"); }
}
class Child extends Parent {
    static void show() { System.out.println("Child"); }
}
```

Here, calling Parent.show() prints *Parent*, and Child.show() prints *Child*,
but polymorphism does not apply — method resolution happens at compile time.

130. What happens if you make an interface variable non-final or non-static?

Ans: It's not allowed in Java.
All variables declared inside an interface are implicitly:(public, static, final)

That means they behave like constants shared by all implementing classes.
You cannot make them non-static or non-final, nor can you reassign their values in implementation classes.

✅ **Summary Table:**

Keyword	Meaning	Can combine with
this	Refers to current object	Used inside instance methods or constructors
super	Refers to parent class	Used to access parent methods/constructors
final	Prevent modification	Works with classes, methods, variables
static	Belongs to class, not instance	Variables, methods, blocks, nested classes
abstract	Incomplete element to be implemented	Classes, methods

STAGE 8: Comparison-Based Topics

131. Difference between Method Overloading and Method Overriding

Ans: Method Overloading happens when multiple methods in the same class have the same name but different parameters (type, number, or order).
Method Overriding occurs when a subclass provides a new implementation for a method that is already defined in its superclass.

Aspect	Overloading	Overriding
Binding Time	**Compile-time**	**Runtime**
Parameters	**Must differ**	**Must be same**
Return Type	**Can differ**	**Must be same or covariant**
Inheritance	**Not required**	**Requires inheritance**

132. Difference between Compile-time Polymorphism and Runtime Polymorphism

Ans:

1. Compile-time Polymorphism (Static Polymorphism)
Compile-time polymorphism occurs when the method to be executed is decided at compile time.How it is achieved:
✅ Achieved through Method Overloading (same method name, different parameter lists).Example:

```
class MathOperation {
    void add(int a, int b) {
        System.out.println(a + b);
    }
    void add(int a, int b, int c) {
        System.out.println(a + b + c);
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperation obj = new MathOperation();
        obj.add(5, 10);      // calls 2-arg version
        obj.add(5, 10, 15);  // calls 3-arg version
    }
}
```

2. Runtime Polymorphism (Dynamic Polymorphism)

Definition: Runtime polymorphism occurs when the method to be executed is decided at runtime.

How it is achieved:
✅ Achieved through Method Overriding (same method name and parameters in parent and child classes).

```java
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal obj = new Dog();  // Upcasting
        obj.sound();             // calls Dog's sound() at runtime
    }
}
```

Basis	Compile-time Polymorphism	Runtime Polymorphism
Definition	**Method call resolved at compile time**	**Method call resolved at runtime**
Achieved By	**Method Overloading**	**Method Overriding**
Binding Type	**Static Binding / Early Binding**	**Dynamic Binding / Late Binding**
Decision Made By	**Compiler**	**JVM (Java Virtual Machine)**
Speed	**Faster**	**Slightly Slower**
Inheritance Required	**Not required**	**Required (Parent–Child Relationship)**
Example	`void show(int a)`, `void show(double b)`	`Animal` → `Dog` overriding `sound()`
Flexibility	**Less flexible (fixed at compile time)**	**More flexible (decided at runtime)**

3. Difference between Static Binding and Dynamic Binding

Ans: Static Binding is determined at compile-time (used by static, private, and final methods).
Dynamic Binding is determined at runtime for overridden instance methods.

Aspect	Static Binding	Dynamic Binding
Resolved by	Compiler	JVM
Methods	Static, final, private	Overridden methods
Polymorphism	No	Yes

134. Difference between Constructor Overloading and Method Overloading

Ans: Both use the same name with different parameters, but constructors initialize objects while methods perform actions.

Aspect	Constructor Overloading	Method Overloading
Purpose	Initialize objects	Perform actions
Return Type	None	Can have any
Called by	new keyword	Directly invoked

135. Difference between Constructor and Method Overriding (and why constructors can’t be overridden)

Ans: Constructors cannot be overridden because they are not inherited. Each class has its own constructors that are called via super().

Aspect	Constructor	Method
Inherited	No	Yes
Overridable	No	Yes
Purpose	Object creation	Behavior definition

136. Difference between Instance Method and Static Method (in the context of polymorphism)

Ans: Instance Methods are tied to objects and participate in runtime polymorphism.
Static Methods are tied to the class and resolved at compile-time.

Aspect	Instance Method	Static Method
Belongs To	Object	Class
Polymorphism	Supported	Not supported
Binding	Dynamic	Static

137. Difference between Superclass Reference and Subclass Reference Behavior in Runtime Polymorphism

Ans: A superclass reference can refer to a subclass object, enabling runtime polymorphism.
At runtime, the actual object's overridden method is executed.

Example:

```
Parent p = new Child();
p.show(); // Executes Child's show()
```

Aspect	Superclass Reference	Subclass Reference
Access	Only superclass members	All subclass members
Method Execution	Depends on object type	Direct subclass methods

138. Difference between Abstract Class and Interface

Ans: Abstract Class provides partial abstraction (can have both abstract & concrete methods).
Interface provides full abstraction (only contracts).

Aspect	Abstract Class	Interface
Methods	Abstract + Concrete	Abstract (and default/static)
Constructors	Allowed	Not allowed
Multiple Inheritance	Not supported	Supported

139. Difference between Final Class and Abstract Class

Ans: Final Class cannot be extended — represents complete behavior.
Abstract Class must be extended — represents incomplete behavior.

Aspect	Final Class	Abstract Class
Inheritance	Not allowed	Required
Purpose	Restrict modification	Enforce implementation

140. Difference between Interface Inheritance and Class Inheritance

Ans: Interface Inheritance allows multiple inheritance and defines only contracts.
Class Inheritance allows single inheritance and includes data and behavior.

Aspect	Interface Inheritance	Class Inheritance
Multiple Inheritance	Allowed	Not allowed
Members	Methods only	Methods + Variables