Java Object-Oriented Programming Concepts

1 What is Inheritance in Java?

What is Inheritance in Java?

Inheritance is a fundamental concept in object-oriented programming (OOP) where a new class (subclass or child class) derives properties and behaviors (fields and methods) from an existing class (superclass or parent class). This promotes code reusability, hierarchy, and the "is-a" relationship. In Java, it is achieved using the `extends` keyword. The subclass inherits non-private members of the superclass and can add new members or override existing ones.

1.1 Real-life Example

Consider a vehicle hierarchy in a transportation system. A "Bicycle" class can inherit from a "Vehicle" class. The Vehicle class might have common attributes like speed and methods like `move()`, while Bicycle adds specific features like pedals. This avoids rewriting common code for different vehicles like cars or bikes.

1.2 Diagram

```
Vehicle (Superclass)
- speed: int
- move(): void

   |
   | (extends)
   v

Bicycle (Subclass)
- pedals: int
- ringBell(): void
```

2 Why Use the 'this' Keyword?

Why Use the 'this' Keyword?

The `this` keyword in Java refers to the current instance of the class. It is primarily used to:

- Distinguish between instance variables and local variables/parameters with the same name.
- Invoke another constructor in the same class (constructor chaining).
- Pass the current object as a parameter to another method.
- Access instance methods or variables from within the class.

Without `this`, ambiguity can arise in constructors or setters when parameter names match field names.

2.1 Real-life Example

In a banking application, an "Account" class has a constructor that sets the account balance. If the parameter is named "balance", `this.balance = balance;` clarifies that the left side is the instance variable. This is like labeling your own suitcase at an airport baggage claim to avoid confusion with others' identical ones.

3 What is Method Overriding vs Overloading?

What is Method Overriding vs Overloading?

- **Method Overloading:** This is compile-time polymorphism where multiple methods in the same class have the same name but different parameter lists (number, type, or order). It allows methods to perform similar actions with varying inputs. Signatures must differ, but return types can be the same or different.
- **Method Overriding:** This is runtime polymorphism where a subclass provides a specific implementation for a method already defined in its superclass. The method signature (name, parameters, return type) must be identical. It is used to change or extend behavior in subclasses.

3.1 Real-life Example for Overloading

In a calculator app, an "add" method is overloaded: `add(int a, int b)` for integers and `add(double a, double b)` for decimals. This is like a chef using the same "cook" recipe but with different ingredients (e.g., cook(vegetables) vs. cook(meat, spices)).

3.2 Real-life Example for Overriding

In an animal simulation, a superclass "Animal" has a `makeSound()` method saying "Generic sound". A subclass "Dog" overrides it to say "Bark". This is like a family business where the child inherits the parent's job but runs it differently (e.g., parent sells books traditionally, child adds online sales).

4 What is Object Instantiation?

What is Object Instantiation?

Object instantiation is the process of creating a new instance (object) of a class in memory using the `new` keyword. It allocates space for the object's fields and initializes them (often via a constructor). The class acts as a blueprint, and instantiation brings it to life as a usable entity.

4.1 Real-life Example

In a car manufacturing plant, the "Car" class is the design blueprint with specs like color and engine. Instantiation is like producing a specific car: `Car myCar = new Car("Red");`. This is similar to baking a cake—instantiation is mixing ingredients and baking based on a recipe (class) to get an edible cake (object).

5 Explain Single vs Multiple Inheritance

Explain Single vs Multiple Inheritance

- **Single Inheritance:** A class inherits from only one superclass. Java supports this directly via the `extends` keyword, creating a simple hierarchy. It is straightforward and avoids complexity.
- **Multiple Inheritance:** A class inherits from more than one superclass. Java does not support multiple inheritance for classes (to avoid the "diamond problem" where ambiguity arises from duplicate inherited members). However, it allows multiple inheritance through interfaces (using `implements`), as interfaces do not have implementation details.

5.1 Real-life Example for Single Inheritance

A "Smartphone" class inherits from a "Phone" class. Phone has basic call features, and Smartphone adds apps. This is like a child inheriting traits from one parent (e.g., eye color from mother).

5.2 Real-life Example for Multiple Inheritance (via Interfaces)

A "FlyingCar" class implements "Drivable" and "Flyable" interfaces. Drivable has `drive()`, Flyable has `fly()`. This is like a person inheriting skills from multiple mentors—one teaches cooking, another teaches painting—without conflicting physical traits.

5.3 Diagram

```
Drivable (Interface)    Flyable (Interface)
- drive(): void         - fly(): void

          \             /
           \           /
            v         v
          FlyingCar (Class)
          - Implements both
```

6 What is Encapsulation?

What is Encapsulation?

Encapsulation is an OOP principle that bundles data (fields) and methods that operate on the data into a single unit (class), while restricting direct access to the data using access modifiers (e.g., private fields with public getters/setters). It hides internal implementation details, promotes data security, and makes code maintainable.

6.1 Real-life Example

In a coffee machine class, the internal "waterLevel" is private, accessible only via public methods like `brewCoffee()` or `getWaterLevel()`. This protects against invalid changes. It is like a bank's ATM—users deposit/withdraw money via interfaces, but cannot directly access the vault (internal data).

7 What is Constructor Overloading?

What is Constructor Overloading?

Constructor overloading allows a class to have multiple constructors with the same name but different parameter lists. This enables creating objects in various ways, depending on the provided arguments. Like method overloading, it is resolved at compile time.

7.1 Real-life Example

In an "Employee" class, one constructor `Employee()` sets default values (e.g., salary=0), another `Employee(String name, int salary)` initializes with specifics. This is like ordering pizza: default (plain cheese) vs. customized (with toppings, size).

8 Can We Override Static Methods?

Can We Override Static Methods?

No, static methods cannot be overridden in Java because they belong to the class itself (not instances) and are resolved at compile time (static binding). A subclass can declare a static method with the same signature, but this is "method hiding," not overriding. Calling it depends on the reference type, not the object type.

8.1 Real-life Example

A utility class "MathUtils" has a static method `piValue()` returning 3.14. A subclass "AdvancedMath" declares the same static method returning 3.14159. It is like two branches of a store chain having the same "getStoreHours()" policy, but one hides/changes it locally—calls depend on which branch you reference, not dynamic choice.

9 What is Runtime Polymorphism?

What is Runtime Polymorphism?

Runtime polymorphism (or dynamic polymorphism) occurs when the method to invoke is determined at runtime based on the actual object type, not the reference type. It is achieved through method overriding, where a superclass reference points to a subclass object, and the JVM calls the overridden method.

9.1 Real-life Example

In a shape drawing app, a “Shape” reference can point to a “Circle” or “Square” object. Calling `draw()` on the reference invokes the specific subclass’s `draw()` at runtime. This is like a remote control (superclass reference) operating different TVs (subclass objects)—the action (turn on) behaves differently per TV model, decided when you press the button.

10 Difference Between Class and Object?

Difference Between Class and Object?

- **Class:** A blueprint or template that defines the structure (fields) and behavior (methods) for objects. It is logical and does not occupy memory until instantiated. Defined using the `class` keyword.
- **Object:** An instance of a class, created in memory with actual values for fields. Multiple objects can be created from one class, each with its own state.

10.1 Real-life Example

A “Recipe” is the class (instructions for making cookies: ingredients list, bake method). A “Cookie” object is the actual baked cookie (instantiated with specific flour amount, chocolate chips). You can bake multiple cookies (objects) from one recipe (class), each varying slightly (e.g., one with nuts, another plain).