Oops

1. What is Inheritance in Java .

Ans1. Inheritance in Java is a mechanism that allows a class to acquire the properties (fields) and behaviors (methods) of another class. It promotes code reusability and establishes a parent-child relationship between classes.

Key Concepts of Inheritance in Java

1. **Parent Class (Super Class)** – The class whose properties and methods are inherited.
2. **Child Class (Subclass)** – The class that inherits from another class.
3. **`extends` Keyword** – Used to inherit a class.

Types of Inheritance in Java

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

```
class Animal {

    void eat() {

        System.out.println("This animal eats food.");

    }

}
```

```
class Dog extends Animal {

    void bark() {

        System.out.println("The dog barks.");

    }

}



public class Main {

    public static void main(String[] args) {

        Dog myDog = new Dog();

        myDog.eat();  // Inherited method

        myDog.bark(); // Own method

    }
```

}

Multilevel Inheritance – A class inherits from another class, which in turn inherits from another class.

java

```
class Animal {

    void eat() {

        System.out.println("This animal eats food.");

    }

}




class Mammal extends Animal {

    void walk() {

        System.out.println("This mammal walks.");

    }
```

```
}

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited from Animal
        myDog.walk(); // Inherited from Mammal
```

```
        myDog.bark(); // Own method
    }
}
```

Hierarchical Inheritance – Multiple child classes inherit from a single parent class.

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

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

```
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
```

```
        dog.eat();

        dog.bark();

        cat.eat();

        cat.meow();

    }

}
```

Important Notes

- **Java does not support multiple inheritance** (i.e., a class cannot inherit from multiple classes) to avoid ambiguity.
- **The `super` keyword** is used to access parent class methods or constructors in the child class.
- **Method Overriding** allows a subclass to provide a specific implementation of a method already defined in the parent class.

2. What is superclass and subclass?

Ans2. **Superclass and Subclass in Java**

In Java, **inheritance** allows one class to derive properties and behaviors from another. This creates a parent-child relationship between classes.

1. Superclass (Parent Class)

A **superclass** (also called the **parent class** or **base class**) is the class from which another class

inherits. It contains common attributes and methods that can be shared by multiple subclasses.

Example:

```
class Animal {  // Superclass
    void eat() {
        System.out.println("This animal eats food.");
    }
}
```

2. Subclass (Child Class)

A subclass (also called the child class or derived class) is a class that inherits from another class. It can use or override the methods of the superclass and can also have its own additional methods and fields.

Example:

```
class Dog extends Animal {  // Subclass
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

3. Example of Superclass and Subclass in Action

```
class Animal {  // Superclass
    void eat() {
        System.out.println("This animal eats food.");
```

```
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited from superclass
        myDog.bark(); // Defined in subclass
    }
}
```

Output:

```
This animal eats food.
The dog barks.
```

Key Points

- **A superclass provides common properties and behaviors.**

- **A subclass extends a superclass and can use or override its methods.**
- **The `extends` keyword is used to define a subclass.**
- **A subclass inherits non-private members (fields and methods) from the superclass.**

3. How is Inheritance implemented/achieved in Java

Ans3. How Inheritance is Implemented in Java

Inheritance in Java is implemented using the `extends` keyword, which allows a subclass (child class) to inherit properties and methods from a superclass (parent class).

---

1. Using the `extends` Keyword

The `extends` keyword is used to define a subclass that inherits from a superclass.

Example:

```
class Animal {  // Superclass
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {  // Subclass
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited method
        myDog.bark(); // Own method
    }
}
```

Output:

```
This animal eats food.
The dog barks.
```

2. Using the `super` Keyword

The `super` keyword is used inside a subclass to access members (methods or constructors) of the superclass.

Example:

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

class Dog extends Animal {
```

```
    void makeSound() {
        super.makeSound();  // Calling superclass method
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

Output:

```
Animal makes a sound
Dog barks
```

3. Types of Inheritance in Java

Java supports the following types of inheritance:

a) Single Inheritance

One subclass inherits from one superclass.

```
class Parent {
```

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

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

b) Multilevel Inheritance

A class inherits from another class, which in turn inherits from another class.

```
class GrandParent {
    void display1() {
        System.out.println("Grandparent class method");
    }
}

class Parent extends GrandParent {
    void display2() {
        System.out.println("Parent class method");
    }
```

```
}

class Child extends Parent {
    void display3() {
        System.out.println("Child class method");
    }
}
```

c) Hierarchical Inheritance

Multiple classes inherit from a single parent class.

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

class Child1 extends Parent {
    void display1() {
        System.out.println("Child1 class method");
    }
}

class Child2 extends Parent {
```

```
    void display2() {
        System.out.println("Child2 class method");
    }
}
```

4. Why Multiple Inheritance is Not Supported

Java does not support multiple inheritance (one class inheriting from multiple classes) to avoid ambiguity problems.

Example of a potential issue:

```
class A {
    void show() {
        System.out.println("Class A");
    }
}

class B {
    void show() {
        System.out.println("Class B");
    }
}

// This would cause ambiguity if Java allowed multiple inheritance
// class C extends A, B {
```

```
//     // Error: Which show() method should be inherited?
// }
```

5. Implementing Inheritance Using Interfaces

If multiple inheritance is needed, Java allows it through interfaces.

Example:

```
interface Animal {
    void eat();
}

interface Mammal {
    void walk();
}

class Dog implements Animal, Mammal {
    public void eat() {
        System.out.println("Dog eats food");
    }

    public void walk() {
        System.out.println("Dog walks on four legs");
    }
}
```

```
public class Main {
   public static void main(String[] args) {
      Dog myDog = new Dog();
      myDog.eat();
      myDog.walk();
   }
}
```

Output:

Dog eats food

Dog walks on four legs

4. What is polymorphism.

Ans4. **Polymorphism in Java**

Polymorphism in Java is the ability of an object to take many forms. It allows the same method to have different implementations based on the object calling it. This makes code more flexible and reusable.

Types of Polymorphism

1. **Compile-time Polymorphism (Method Overloading)**
2. **Runtime Polymorphism (Method Overriding)**

---

1. Compile-time Polymorphism (Method Overloading)

This occurs when multiple methods in the same class have the **same name** but different **parameters** (different number, type, or order of parameters). The method that gets called is

determined at compile time.

Example: Method Overloading

```
class MathOperations {
    // Method with two int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with three int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method with two double parameters
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
```

```
        System.out.println(math.add(5, 10));         // Calls method with two int parameters
        System.out.println(math.add(5, 10, 15));     // Calls method with three int parameters
        System.out.println(math.add(5.5, 2.5));      // Calls method with two double parameters
    }
}
```

Output:

15

30

8.0

2. Runtime Polymorphism (Method Overriding)

This occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method that gets called is determined at runtime, based on the actual object.

Example: Method Overriding

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


class Dog extends Animal {
```

```
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // Calls Dog's makeSound()

        myAnimal = new Cat(); // Now referring to a Cat object
        myAnimal.makeSound(); // Calls Cat's makeSound()
    }
}
```

Output:

Dog barks

Cat meows

3. Polymorphism Using Interfaces

Since Java doesn’t support multiple inheritance, interfaces help achieve polymorphism.

Example: Polymorphism with Interfaces

```
interface Animal {
    void makeSound();
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat implements Animal {
    public void makeSound() {
        System.out.println("Cat meows");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.makeSound(); // Dog barks

        myAnimal = new Cat();
        myAnimal.makeSound(); // Cat meows
    }
}
```

Real-world Example of Polymorphism

Let's say we are developing a payment system where different payment methods (Credit Card, PayPal, UPI) have different implementations but use the same method name.

```
class Payment {
    void makePayment() {
        System.out.println("Processing payment...");
    }
}

class CreditCard extends Payment {
    @Override
```

```
    void makePayment() {
        System.out.println("Payment made using Credit Card.");
    }
}

class PayPal extends Payment {
    @Override
    void makePayment() {
        System.out.println("Payment made using PayPal.");
    }
}

public class Main {
    public static void main(String[] args) {
        Payment payment;

        payment = new CreditCard();
        payment.makePayment(); // Calls CreditCard's method

        payment = new PayPal();
        payment.makePayment(); // Calls PayPal's method
    }
```

}

Output:

Payment made using Credit Card.

Payment made using PayPal.

5. Differentiate between method overloading and overriding.

Ans5. **Difference Between Method Overloading and Method Overriding in Java**

Feature	Method Overloading	Method Overriding
Definition	Defining multiple methods with the **same name** but different parameters in the **same class**.	Redefining a method of the **superclass** in the **subclass** with the same signature.
Method Name	Same name in the same class	Same name in superclass and subclass
Parameters	Must be **different** (different number, type, or order)	Must be **exactly the same** as the parent method
Return Type	Can be **different**	Must be **same** (or covariant return type)
Access Modifier	No restrictions	Cannot reduce visibility (e.g., `public` cannot be changed to `private`)
When Resolved	**Compile-time** (Static binding)	**Runtime** (Dynamic binding)

Inheritance Required?	**No**, happens in the same class	**Yes**, requires a superclass and subclass
Use of `@Override` Annotation	**Not required**	**Required (recommended)**
Can it be applied to static methods?	**Yes**, static methods can be overloaded	**No**, static methods cannot be overridden (but can be hidden)
Polymorphism Type	Compile-time Polymorphism	Runtime Polymorphism

Example of Method Overloading

```
class MathOperations {
    // Method with two parameters
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with three parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }
```

```
    // Overloaded method with different parameter types
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        System.out.println(math.add(5, 10));         // Calls add(int, int)
        System.out.println(math.add(5, 10, 15));     // Calls add(int, int, int)
        System.out.println(math.add(5.5, 2.5));      // Calls add(double, double)
    }
}
```

Output:

```
15
30
8.0
```

Example of Method Overriding

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // Calls Dog's makeSound() (runtime polymorphism)
    }
}
```

Output:

Dog barks

6. What is an abstraction explained with an Example

Ans6. **Abstraction in Java**

Abstraction is the process of **hiding the implementation details** and **showing only the necessary features** of an object. It helps reduce complexity and increases reusability.

How is Abstraction Achieved in Java?

Java provides **two ways** to achieve abstraction:

1. **Abstract Classes (`abstract` keyword)**
2. **Interfaces (`interface` keyword)**

---

1. Abstraction Using Abstract Classes

An **abstract class** is a class that cannot be instantiated (you cannot create objects of it). It may contain **abstract methods** (methods without a body) that must be implemented by subclasses.

Example of Abstraction Using an Abstract Class

```
// Abstract class

abstract class Vehicle {

   abstract void start(); // Abstract method (no body)



   void fuelCapacity() { // Concrete method

      System.out.println("Fuel capacity is 50 liters.");

   }

}
```

```
// Subclass providing implementation for abstract method
class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car starts with a key.");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car(); // Upcasting
        myCar.start();  // Calls overridden method
        myCar.fuelCapacity();  // Calls concrete method from abstract class
    }
}
```

Output:

```
Car starts with a key.
Fuel capacity is 50 liters.
```

2. Abstraction Using Interfaces

An **interface** is a **fully abstract** type that only contains method **signatures** (without implementation). A class implementing an interface must provide implementations for all its methods.

Example of Abstraction Using an Interface

```
// Interface
interface Animal {
    void makeSound(); // Abstract method
}

// Implementing the interface
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound(); // Calls implemented method
    }
}
```

Output:

Dog barks.

Real-World Example: Abstraction in Banking System

Imagine a banking system where different types of accounts exist (Savings Account, Current Account), but every account must implement a **withdraw()** method.

Example Using Abstract Class

```
abstract class BankAccount {

    abstract void withdraw(double amount); // Abstract method


    void deposit(double amount) { // Concrete method

        System.out.println("Deposited: $" + amount);

    }

}


class SavingsAccount extends BankAccount {

    @Override

    void withdraw(double amount) {

        System.out.println("Withdrawal from Savings Account: $" + amount);

    }

}


public class Main {
```

```
    public static void main(String[] args) {
        BankAccount myAccount = new SavingsAccount();
        myAccount.deposit(500);
        myAccount.withdraw(200);
    }
}
```

Output:

Deposited: $500.0

Withdrawal from Savings Account: $200.0

7. What is the difference between an abstract method and final method in Java?Explain with an example

Ans 7. **Difference Between Abstract Method and Final Method in Java**

Feature	Abstract Method	Final Method
Definition	A method **without a body**, meant to be implemented in subclasses.	A method that **cannot be overridden** in subclasses.
Usage	Used in **abstract classes** to enforce implementation in subclasses.	Used to **prevent** subclasses from modifying the method's behavior.
Implementation	**Must be overridden** in a subclass.	**Cannot be overridden** in a subclass.
Keyword Used	`abstract`	`final`
Can it have a	**No**, it must be implemented by a	**Yes**, it has a complete

body?	subclass.	implementation.
Can it be static?	**No**, because it must be overridden.	**Yes**, but it cannot be overridden.

Example of an Abstract Method

Abstract methods **must** be overridden in a subclass.

```
// Abstract class
abstract class Animal {
    abstract void makeSound(); // Abstract method (no body)
}

// Subclass providing implementation
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
```

```
        myDog.makeSound(); // Calls overridden method
    }
}
```

Output:

Dog barks.

Example of a Final Method

Final methods **cannot** be overridden in a subclass.

```
class Vehicle {
    final void start() { // Final method
        System.out.println("Vehicle is starting...");
    }
}

// Subclass trying to override final method (will cause an error)
class Car extends Vehicle {
    // Uncommenting this will cause a compilation error
    // void start() {
    //     System.out.println("Car is starting...");
    // }
}
```

```
public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Vehicle();
        myCar.start(); // Calls final method from superclass
    }
}
```

Output:

Vehicle is starting...

8. What is the final class in Java

Ans8. **Final Class in Java**

A **final class** is a class that **cannot be extended (inherited)** by any other class. This means that no other class can create a subclass of a final class.

How to Declare a Final Class?

In Java, you use the `final` keyword to declare a class as final:

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

```
// This will cause an error because final classes cannot be extended
// class Car extends Vehicle {
//     void start() {
//         System.out.println("Car is starting...");
//     }
// }

public class Main {
    public static void main(String[] args) {
        Vehicle myVehicle = new Vehicle();
        myVehicle.start(); // Calls start() method from final class
    }
}
```

Output:

```
Vehicle is starting...
```

Example: Why Use a Final Class?

Imagine we are developing a banking system where the `BankAccount` class should not be modified by unauthorized subclasses.

```
final class BankAccount {
    private double balance = 1000;
```

```
    void showBalance() {
        System.out.println("Balance: $" + balance);
    }
}

// This will cause an error because final classes cannot be inherited
// class SavingsAccount extends BankAccount { }

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.showBalance();
    }
}
```

Output:

Balance: $1000.0

9. Differentiate between abstraction and encapsulation

Ans9. **Difference Between Abstraction and Encapsulation in Java**

Feature	Abstraction	Encapsulation

Definition	Hides the **implementation details** and only shows essential features.	Hides **data** by restricting direct access and allowing controlled access via methods.
Purpose	Focuses on **what an object does**, not how it does it.	Focuses on **protecting** an object's data from unintended access/modification.
How It's Achieved?	Using **abstract classes** and **interfaces**.	Using **private variables** and **public getter/setter methods**.
Access Control	Not related to access control but hides **implementation logic**.	Uses **access modifiers** (`private`, `public`, `protected`) to **restrict direct access**.
Implementation	Implemented using **abstract classes (`abstract` keyword) or interfaces**.	Implemented using **classes, private fields, and getters/setters**.
Example	**Hiding how a car engine works but providing a `startCar()` method**.	**Making variables private and providing `getBalance()` instead of allowing direct access to `balance`.**
Focus	**Hides implementation details.**	**Hides object data and protects it from direct access.**

Example of Abstraction (Using an Abstract Class)

```
// Abstract class

abstract class Vehicle {

   abstract void start(); // Abstract method (no body)

}
```

```
// Subclass providing implementation
class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car starts with a key.");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.start(); // Calls overridden method
    }
}
```

Output:

Car starts with a key.

Example of Encapsulation (Using Private Variables & Getters/Setters)

```
class BankAccount {
    private double balance; // Private variable (cannot be accessed directly)
```

```
    // Constructor
    public BankAccount(double balance) {
        this.balance = balance;
    }

    // Getter method (Controlled access)
    public double getBalance() {
        return balance;
    }

    // Setter method (Controlled modification)
    public void setBalance(double balance) {
        if (balance >= 0) {
            this.balance = balance;
        } else {
            System.out.println("Balance cannot be negative.");
        }
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        BankAccount account = new BankAccount(5000);
        System.out.println("Balance: $" + account.getBalance());

        account.setBalance(-100); // Trying to set a negative balance
        System.out.println("Balance after update: $" + account.getBalance());
    }
}
```

Output:

Balance: $5000.0

Balance cannot be negative.

Balance after update: $5000.0

Explanation:

- **Direct access to `balance` is restricted (`private`).**
- **Controlled access is provided via `getBalance()` and `setBalance()`.**
- **Prevents setting invalid values (e.g., negative balance).**

10.Difference between Runtime and compile time polymorphism explain with an example

Ans10. **Difference Between Runtime and Compile-Time Polymorphism in Java**

Polymorphism in Java is the ability of an object to take multiple forms. It is categorized into two

types:

Feature	Compile-Time Polymorphism (Method Overloading)	Runtime Polymorphism (Method Overriding)
Definition	Resolving method calls at **compile time**.	Resolving method calls at **runtime**.
Achieved Through	**Method Overloading** (Same method name, different parameters).	**Method Overriding** (Same method name, same parameters, different implementation in subclass).
Binding Type	**Early Binding** (Decided at compile time).	**Late Binding** (Decided at runtime).
Performance	Faster, as it is resolved at compile time.	Slightly slower, as it is resolved at runtime using dynamic method dispatch.
Flexibility	Less flexible, as method calls are determined at compile-time.	More flexible, allows different behaviors at runtime.
Usage	When multiple methods perform similar operations but with different inputs.	When a subclass provides a **specific implementation** for a method in the parent class.

Example of Compile-Time Polymorphism (Method Overloading)

Method Overloading allows multiple methods in the **same class** with the **same name but different parameters**.

class MathOperations {

```
    // Method with two int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with three int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with double parameters
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();

        System.out.println(math.add(5, 10));       // Calls first method
        System.out.println(math.add(5, 10, 15));   // Calls second method
```

```
        System.out.println(math.add(5.5, 10.5));   // Calls third method
    }
}
```

Output:

15

30

16.0

Explanation:

- The `add()` method is overloaded with **different parameter lists**.
- The compiler **determines** which method to call based on the **arguments**.
- **Resolved at compile-time → Compile-time polymorphism.**

Example of Runtime Polymorphism (Method Overriding)

Method Overriding allows a **subclass to provide a specific implementation** of a method that is already defined in its **parent class**.

```
// Parent class
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound.");
    }
}
```

```
// Subclass overrides the method
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks.");
    }
}

// Subclass overrides the method
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal;   // Reference variable of type Animal

        myAnimal = new Dog();
```

```
        myAnimal.makeSound(); // Calls overridden method in Dog


        myAnimal = new Cat();
        myAnimal.makeSound(); // Calls overridden method in Cat
    }
}
```

Output:

Dog barks.

Cat meows.