Inheritance in Java

Inheritance is one of the key features of Object-Oriented Programming (OOP). It allows a class to inherit properties and methods from another class, promoting code reusability and establishing a parent-child relationship between classes.

1. Inheritance Basics

In Java, inheritance is achieved using the extends keyword. When a class inherits from another class, it acquires all non-private members (fields and methods) of the parent class.

Terminology:

Superclass (Parent Class): The class whose features are inherited.

Subclass (Child Class): The class that inherits from another class.

Syntax:

```
class Subclass extends Superclass {
    // additional fields and methods
}
```

Example:

```
// Superclass
class Animal {
    void eat() {
        System.out.println("This animal is eating.");
    }
}
// Subclass
class Dog extends Animal {
    void bark() {
        System.out.println("The dog is barking.");
    }
```

```
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // Inherited method from Animal class
        dog.bark(); // Method of Dog class
    }
}
```

Output:

This animal is eating.

The dog is barking.

In the above example, the Dog class inherits the eat() method from the Animal class.

2. Member Access Rules

Private members: The members declared as private in the parent class are not accessible directly in the child class.

Protected members: Members declared as protected are accessible within the same package and also by subclasses.

Public members: Members declared as public are accessible from anywhere.

Default members: Members with no access modifier are accessible within the same package.

Example:

```
class Animal {
    private String type = "Animal";  // Private: not accessible in the subclass
    protected String sound = "Generic Sound";  // Protected: accessible in the subclass

    void displayType() {
        System.out.println("This is a generic animal.");
    }
}
```

```
class Dog extends Animal {
    void makeSound() {
        System.out.println("The dog makes: " + sound);  // Accessing protected member
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.displayType();
        dog.makeSound();
    }
}
```

Output:

This is a generic animal.

The dog makes: Generic Sound

In this example, sound is protected, so it is accessible in the subclass, while type is private and not accessible.

3. Super Keyword

The super keyword in Java refers to the superclass (parent class). It is used to access the superclass's members (fields and methods) and also to call the superclass constructor.

a. Accessing Superclass Methods

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

```
class Dog extends Animal {
    void eat() {
        System.out.println("Dog is eating.");
    }
    void display() {
        super.eat();  // Calling superclass method
        eat();        // Calling subclass method
    }
}

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

Output:

Animal is eating.

Dog is eating.

b. Calling Superclass Constructor

When creating a subclass object, the superclass constructor is called first. You can explicitly invoke it using super().

Example:

```
class Animal {
    Animal() {
```

```
        System.out.println("Animal constructor called.");
    }
}
class Dog extends Animal {
    Dog() {
        super();  // Calls the superclass constructor
        System.out.println("Dog constructor called.");
    }
}

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

Output:

Animal constructor called.

Dog constructor called.

4. Forms of Inheritance

Java supports different forms of inheritance, though it does not support multiple inheritance directly due to ambiguity problems. Instead, it uses interfaces to achieve that. The following forms of inheritance are possible in Java:

a. Single Inheritance

A subclass inherits from a single superclass.

Example:

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

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

b. Multilevel Inheritance

A subclass inherits from a class that is already a subclass of another class.

Example:

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

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

class C extends B {
```

```
    void print() {
        System.out.println("Class C");
    }
}

public class Main {
    public static void main(String[] args) {
        C c = new C();
        c.display(); // From Class A
        c.show();    // From Class B
        c.print();   // From Class C
    }
}
```

c. Hierarchical Inheritance

Multiple subclasses inherit from the same superclass.

Example:

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

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

```
class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();
        dog.bark();

        Cat cat = new Cat();
        cat.eat();
        cat.meow();
    }
}
```

5. Method Overriding

Method Overriding occurs when a subclass provides a specific implementation of a method already defined in its superclass. The overridden method must have the same name, return type, and parameters as the method in the parent class.

Example:

```
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) {
        Dog dog = new Dog();
        dog.makeSound();  // Calls the overridden method
    }
}
```

Output:

Dog barks.

6. Abstract Classes and Methods

An abstract class is a class that cannot be instantiated directly and is used to declare abstract methods that must be implemented by its subclasses. Abstract classes can have both abstract and concrete (regular) methods.

Abstract Method: A method declared without a body, which is meant to be implemented by subclasses.

Syntax:

java

```
abstract class ClassName {
```

```
    abstract void methodName();  // Abstract method
}
```

Example:

```
abstract class Animal {
    abstract void makeSound();  // Abstract method

    void sleep() {  // Concrete method
        System.out.println("This animal is sleeping.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound();  // Calls the overridden method
        dog.sleep();      // Calls the inherited method
    }
}
```

Output:

```
Dog barks.
This animal is sleeping.
```

Summary

Inheritance allows code reuse by letting a subclass inherit methods and fields from a superclass.

Member Access Rules determine whether members of a superclass are accessible in a subclass.

The super keyword is used to access superclass methods and constructors.

Forms of Inheritance include single, multilevel, and hierarchical inheritance.

Method Overriding allows a subclass to provide a specific implementation of a method that is already defined in the superclass.

Abstract Classes and methods define a blueprint for other classes to follow, ensuring specific methods are implemented.