DIGITAL NOTES
ON
JAVA PROGRAMMING

FOR

COMPUTER ENGG. STUDENT

PROVIEDED BY: -

SHAKTI RAJ SINGH, PROGRAMMER

UNIT III

Inheritance and Polymorphism in Java

1. Inheritance

Definition:

Inheritance is an Object-Oriented Programming (OOP) concept where a class (subclass or child) derives properties and behaviors (fields and methods) from another class (superclass or parent). This allows the subclass to reuse code from the superclass and can also introduce its own additional features.

- In Java, inheritance is implemented using the extends keyword.

Example of Inheritance:

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

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

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog(); // Creating an object of the subclass
      dog.eat();  // Calling the inherited method
      dog.bark(); // Calling the method defined in the subclass
   }
}
```

In the example above, Dog inherits the eat() method from the Animal class, and it also defines its own bark() method.

Advantages of Inheritance:

1. **Code Reusability**: Instead of writing code in every class, inheritance allows you to write common methods and fields once in the superclass, which all subclasses can inherit. This reduces redundancy and leads to cleaner code.

Example: If multiple classes need a method like toString(), the superclass can define this method, and all subclasses can use it.

2. **Extensibility**: Inheritance allows you to extend functionality. A subclass can add additional features to the inherited methods or override them to meet its specific needs.

Example: A Bird class can inherit the general behavior from an Animal class but can also have additional methods, like fly().

3. **Ease of Maintenance**: When a change is made to the superclass, it is automatically reflected in all subclasses. This makes the code easier to maintain and less error-prone.

4. **Polymorphism Support**: Inheritance is essential for **polymorphism**. It allows methods to be overridden, which is a powerful mechanism for creating flexible systems.

Disadvantages of Inheritance:

1. **Tight Coupling**: Subclasses are dependent on their superclasses. If the superclass is modified, it may break the functionality of the subclasses that rely on it. This can lead to unintentional side effects.

Example: If you modify a method in the superclass, all subclasses that inherit that method might behave differently, which can cause bugs.

2. **Limited Flexibility**: Java does not support **multiple inheritance** (a class cannot inherit from more than one class), which can be a limitation. If you want a class to inherit from multiple classes, Java forces you to use **interfaces** or **abstract classes**.

3. **Complexity**: Excessive inheritance (deep inheritance trees) can create complex relationships that are hard to understand. This can lead to difficulties in tracing bugs and understanding the behavior of the program.

4. **Inheriting Unnecessary Functionality**: A subclass may inherit methods and properties that it doesn't need, leading to an increase in the size of the class and unwanted functionality.

2. Access Modifiers in Inheritance (Visibility of Data)

Java has different **access modifiers** that determine the visibility of class members (fields and methods) in subclasses. Understanding the use of access modifiers is important in inheritance because it controls what a subclass can inherit and access from a superclass.

Public:

- Members declared public are accessible from anywhere, including subclasses in other packages.

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

Protected:

- Members declared protected are accessible within the same package and by subclasses, even if they are in different packages.

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

Private:

- Members declared private are not accessible outside the class, even by subclasses.

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

Default (Package-Private):

- If no access modifier is specified, the member is accessible only within the same package.

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

3. Constructor Chaining

Definition:

Constructor chaining refers to the practice of calling one constructor from another within the same class or between a superclass and a subclass. This helps to reduce redundancy and manage complex initialization processes.

- **Within the Same Class**: A constructor can call another constructor within the same class using this().
- **Between Superclass and Subclass**: A subclass constructor can call a superclass constructor using super().

Example of Constructor Chaining:

```
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(); // Constructor chaining in action
   }
}
```

Output:

```
Animal constructor called
Dog constructor called
```

Advantages of Constructor Chaining:

- **Code Reusability**: Constructor chaining avoids redundant code by reusing constructors from the superclass.
- **Simplified Initialization**: It ensures that the parent class is properly initialized before the child class.

Disadvantages of Constructor Chaining:

- **Increased Complexity**: If chaining is not handled properly, it can lead to hard-to-understand code, especially when constructors are called multiple times in a hierarchy.
- **Error Propagation**: Constructor chaining may lead to errors propagating from the superclass if not handled carefully.

4. Types of Inheritance

Single Inheritance:

In single inheritance, a subclass inherits from only one superclass. This is the simplest form of inheritance.

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

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

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

Advantages:

- Simple and easy to implement.
- Clear hierarchy and understanding of class relationships.

Disadvantages:

- Limited when you need to inherit from multiple classes, as Java does not support multiple inheritance.

Multilevel Inheritance:

In multilevel inheritance, a class inherits from another class, and that class can be further inherited by another class.

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

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

class Puppy extends Dog {
   void play() {
      System.out.println("Playing...");
   }
}

public class Main {
   public static void main(String[] args) {
      Puppy puppy = new Puppy();
      puppy.eat();   // Inherited from Animal
      puppy.bark();  // Inherited from Dog
      puppy.play();  // Defined in Puppy
   }
}
```

Advantages:

- Provides a more detailed hierarchy.
- Each subclass can add specific behaviors.

Disadvantages:

- The inheritance chain can get long and complicated, leading to increased complexity.

Hierarchical Inheritance:

In hierarchical inheritance, one superclass is inherited by multiple subclasses.

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

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

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

public class Main {
```

```
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();   // Inherited from Animal
        dog.bark();  // Defined in Dog

        Cat cat = new Cat();
        cat.eat();   // Inherited from Animal
        cat.meow();  // Defined in Cat
    }
}
```

Advantages:

- Multiple classes can share common functionality.
- Code reuse is encouraged.

Disadvantages:

- Changes in the superclass may affect all subclasses.
- The system may become less modular and harder to maintain.

5. Polymorphism

Definition:

Polymorphism allows objects to be treated as instances of their parent class. The two types of polymorphism are:

1. **Compile-time Polymorphism (Method Overloading)**: Occurs when multiple methods in the same class have the same name but different parameters.
2. **Runtime Polymorphism (Method Overriding)**: Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

Method Overloading (Compile-time Polymorphism):

Method overloading allows a class to define multiple methods with the same name but different parameter lists.

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

    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10));      // Calls add(int, int)
        System.out.println(calc.add(5.5, 10.5));  // Calls add(double, double)
    }
}
```

Advantages:

- Allows the same method name to perform different tasks depending on the parameter types or number of parameters.

Disadvantages:

- Overuse can lead to confusion if methods with the same name perform too different tasks.

Method Overriding (Runtime Polymorphism):

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

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

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

public class Main {
   public static void main(String[] args) {
      Animal animal = new Dog();  // Upcasting (reference type is Animal)
      animal.sound();  // Calls the Dog's sound method (runtime polymorphism)
   }
}
```

Advantages:

- Supports dynamic method dispatch, allowing the system to be more flexible and extensible.
- Enables runtime flexibility by choosing the correct method based on the object's actual type.

Disadvantages:

- Can introduce unexpected behavior if not handled carefully, especially in large class hierarchies.