Inheritance – Polymorphism

Table of Contents

---

1. The `this` Keyword

The `this` keyword is a reference variable in Java that refers to the current object. It can be used to call another constructor from within a constructor, a process known as constructor chaining.

Constructor Overloading

Constructor overloading allows multiple constructors to exist in a class, differing by the number or type of parameters.

Example: VIP Bank Customer

```
class Customer {
    private String name;
    private double balance;

    // Constructor with one parameter
    public Customer(String name) {
        this(name, 0.0); // Calls the other constructor
    }

    // Constructor with two parameters
    public Customer(String name, double balance) {
        this.name = name;
        this.balance = balance;
```

```
    }

    public void display() {
        System.out.println("Customer Name: " + name);
        System.out.println("Account Balance: " + balance);
    }
}

public class Main {
    public static void main(String[] args) {
        Customer customer1 = new Customer("Alice");
        Customer customer2 = new Customer("Bob", 1000.0);

        customer1.display();
        customer2.display();
    }
}
```

Note: The call to another constructor (`this(name, 0.0)`) must be the very first line within the constructor.

---

2. Good Programming Practices

Initializing Parameters Using Multiple Constructors

Using multiple constructors allows for flexibility in initializing objects with different parameters.

Constructor Chaining

Constructor chaining is the process of calling one constructor from another within the same class. This improves code readability and maintainability.

Avoiding Method Calls in Constructors

It is considered good practice not to call other methods (such as setters/getters) from within constructors. This prevents issues related to object state and ensures that the object is fully initialized before any methods are invoked.

---

3. Inheritance

Definition and Analogy

Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit properties and behaviors (methods) from another class. It promotes code reuse and establishes a hierarchical relationship between classes.

Analogy: Think of inheritance like a family tree. If a parent (superclass) has certain traits (attributes/methods), the child (subclass) inherits those traits.

Example: Animal Class

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

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

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

The `super` Keyword

The `super` keyword is used to refer to the immediate parent class. It can be used to access superclass methods and constructors.

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

class Dog extends Animal {
    void eat() {
        super.eat(); // Calls the eat method from Animal class
        System.out.println("Dog eats dog food.");
    }
}

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

---

4. Reference vs Object

- **Object**: An instance of a class that holds the state and behavior defined by that class.
- **Reference**: A variable that holds the memory address of the object. When you create an object, you can assign it to a reference variable.

```
class Dog {
    String name;

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy"); // myDog is a reference to the Dog object
        System.out.println(myDog.name); // Accessing the object's attribute
    }
}
```

---

5. Responsibility of Assigning Values

When using constructors, the responsibility of assigning values lies with the last constructor in the chain. This ensures that all parameters are properly initialized and that the object is in a valid state before use.

---

6. Compile Time vs Runtime Polymorphism

Compile Time Polymorphism

Compile-time polymorphism, also known as static polymorphism, occurs when method overloading is used. The method to be executed is determined at compile time based on the method signature (name and parameter types).

Example: Method Overloading

```
java
Copy code
class MathOperations {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
```

```
    // Method to add two doubles
    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 the first method
        System.out.println(math.add(5, 10, 15));      // Calls the second
method
        System.out.println(math.add(5.5, 10.5));      // Calls the third method
    }
}
```

Runtime Polymorphism

Runtime polymorphism, also known as dynamic polymorphism, occurs when method overriding is used. The method to be executed is determined at runtime based on the actual object type, rather than the reference type.

Example: Method Overriding

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal;

        myAnimal = new Dog(); // Dog object
        myAnimal.sound();     // Outputs: Dog barks.

        myAnimal = new Cat(); // Cat object
        myAnimal.sound();     // Outputs: Cat meows.
    }
}
```

7. Comprehensive Program Example

Below is a program that encompasses all the topics discussed:

```
class Customer {
    private String name;
    private double balance;

    // Constructor overloading
    public Customer(String name) {
        this(name, 0.0);
    }

    public Customer(String name, double balance) {
        this.name = name;
        this.balance = balance;
    }

    public void display() {
        System.out.println("Customer Name: " + name);
        System.out.println("Account Balance: " + balance);
    }
}

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

class Dog extends Animal {
    void eat() {
        super.eat(); // Calls the eat method from Animal class
        System.out.println("Dog eats dog food.");
    }

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

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

public class Main {
    public static void main(String[] args) {
        // Demonstrating Customer class
        Customer customer1 = new Customer("Alice");
        Customer customer2 = new Customer("Bob", 1000.0);
        customer1.display();
        customer2.display();

        // Demonstrating Animal class and inheritance
```

```
        Animal myAnimal;

        myAnimal = new Dog();
        myAnimal.eat();
        ((Dog) myAnimal).makeSound(); // Casting to Dog to call Dog's specific
method

        myAnimal = new Cat();
        myAnimal.eat();
        ((Cat) myAnimal).makeSound(); // Casting to Cat to call Cat's specific
method
    }
}
```

Comparing Both Example

BAD

```
class Rectangle {
   private int x;
   private int y;
   private int width;
   private int height;

   public Rectangle() {
       this.x = 0;
       this.y = 0;
       this.width = 0;
       this.height = 0;
   }

   public Rectangle(int width, int height) {
       this.x = 0;
       this.y = 0;
       this.width = width;
       this.height = height;
   }

   public Rectangle(int x, int y, int width, int height) {
       this.x = x;
       this.y = y;
       this.width = width;
       this.height = height;
   }
}
```

GOOD

```
class Rectangle {
   private int x;
   private int y;
   private int width;
   private int height;

   // 1st constructor
   public Rectangle() {
       this(0, 0); // calls 2nd constructor
   }

   // 2nd constructor
   public Rectangle(int width, int height) {
       this(0, 0, width, height); // calls 3rd constructor
   }

   // 3rd constructor
   public Rectangle(int x, int y, int width, int height) {
       // initialize variables
       this.x = x;
       this.y = y;
       this.width = width;
       this.height = height;
   }
}
```

- Responsibility of assigning values by the last constructors
- Avoids duplicate code and buggs.

super() call example

```java
class Shape {
   private int x;
   private int y;

   public Shape(int x, int y) {
       this.x = x;
       this.y = y;
   }
}

class Rectangle extends Shape {
   private int width;
   private int height;

   // 1st constructor
   public Rectangle(int x, int y) {
       this(x, y, 0, 0); // calls 2nd constructor
   }

   // 2nd constructor
   public Rectangle(int x, int y, int width, int height) {
       super(x, y); // calls constructor from parent (Shape)
       this.width = width;
       this.height = height;
   }
}
```

- In this example we have a class Shape with x,y variables and class Rectangle that extends Shape with variables width and height
- In Rectangle, the 1st constructor we are calling the 2nd constructor
- The 2nd constructor calls the parent constructor with parameters x and y
- The parent constructor will initialize x,y variables while the 2nd Rectangle constructor will initialize the width and height variables
- Here we have both **super()** and **this()** calls

Method Overriding vs Overloading

OVERRIDING

same name
same parameters

```java
class Dog {

   public void bark() {
       System.out.println("woof");
   }
}

class GermanShepherd extends Dog {

   @Override
   public void bark() {
       System.out.println("woof woof
woof");
   }
}
```

Seek

OVERLOADING

same name
different parameters

```java
class Dog {

   public void bark() {
       System.out.println("woof");
   }

   public void bark(int number) {
       for(int i = 0; i < number; i++) {
           System.out.println("woof");
       }
   }
}
```