ENGINEER SERIES

IN JAVA LANGUAGE

3 CH

INHERITANCE AND POLYMORPHISM

ENG : Qusay Khudair

Creativity and Accuracy in Work

AUG PLUS .Edu

Chapter 11

Inheritance and Polymorphism

ENG : Qusay Khudair

Superclasses and Subclasses

Definition: A superclass (or parent class) is a class that is inherited by another class. A subclass (or child class) is a class that inherits from a superclass.

Example:

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

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

In this example, `Animal` is the superclass and `Dog` is the subclass.

Are Superclass's Constructor Inherited?

Explanation: Constructors of a superclass are not inherited by subclasses.

However, the subclass constructor can call the superclass constructor using the `super` keyword (They are invoked explicitly or implicitly) .

Example:

```
public class Animal {
    public Animal() {
        System.out.println("Animal
constructor");}}

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

Output:

```
Animal constructor

Dog constructor
```

Superclass's Constructor Is Always Invoked

Explanation: When a **subclass** object is created, its **superclass** constructor is invoked automatically. This ensures that the superclass is properly initialized before the subclass.

Example:

```
// Superclass

public class Animal {
    public Animal() {
System.out.println("Animal constructor");}}
```

```
// Subclass
public class Dog extends Animal {
    public Dog() {
        System.out.println("Dog constructor");
    }
}

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

Output:

```
Animal constructor

Dog constructor
```

Using the Keyword super

Definition: The `super` keyword is used to refer to the immediate superclass object. It can be used to access superclass methods and constructors.

-**Note** : You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Example:

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

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

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

Output:

Copy code

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

Constructor Chaining

Explanation: Constructor chaining is the process of calling one constructor from another constructor with respect to the current object.

One of the main use of constructor chaining is to avoid duplicate codes while having multiple constructor (by means of constructor overloading) and make code more readable.

Constructor chaining can be done in two ways:

- **Within same class**: It can be done using **this()** keyword for constructors in the same class
- **From base class:** by using **super()** keyword to call the constructor from the base class

Example:

```
public class Animal {
    public Animal() {
        System.out.println("Animal constructor");
}}
public class Dog extends Animal {
    public Dog() {
        this("Dog");
        System.out.println("Dog no-arg
constructor");
    }

    public Dog(String name) {
        super();
```

```
        System.out.println("Dog constructor with
name: " + name);
    }}
public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();}}
```

Output:

```
Animal constructor
Dog constructor with name: Dog
Dog no-arg constructor
```

Overriding Methods in the Superclass

Definition: Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

Example:

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

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

```
public class Test {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.sound(); // Dog's sound method is
called
    }
}
```

Output:

```
Dog barks
```

Overriding Vs Overloading

Overriding:

- Subclass provides a specific implementation for a method defined in the superclass.
- Method signature must be the same.
- Method overriding always needs inheritance.
- In method overriding, methods must have the same name and same signature.
- In method overriding, the return type must be the same or co-variant.
- Private and final methods can’t be overridden.
- The argument list should be the same in method overriding.

Overloading:

- Multiple methods with the same name but different parameters within the same class.
- Method overloading helps to increase the readability of the program.
- Method overloading may or may not require inheritance.
- Methods must have the same name and different signatures.
- The return type can or can not be the same, but we just have to change the parameter.

- Private and final methods can be overloaded.
- The argument list should be different while doing method overloading.

Example of Overriding:

```
import java.io.*;

// Base Class
class Animal {
    void eat() {
        System.out.println("eat() method of base class");
        System.out.println("eating.");
    }
}

// Inherited Class
class Dog extends Animal {
    void eat() {
        System.out.println("eat() method of derived
class");
        System.out.println("Dog is eating.");
    }

    // Method to call the base class method
    void eatAsAnimal() {
        super.eat();
    }
}

// Driver Class
class MethodOverridingEx {
    // Main Function
    public static void main(String args[]) {
        Dog d1 = new Dog();
        Animal a1 = new Animal();
```

```
        d1.eat();
        a1.eat();

        // Polymorphism: Animal reference pointing to Dog
object
        Animal animal = new Dog();

        // Calls the eat() method of Dog class
        animal.eat();

        // To call the base class method, you need to use
a Dog reference
        ((Dog) animal).eatAsAnimal();
    }
}
```

Output

```
add() with 2 parameters
10
add() with 3 parameters
17
```

Example of Overloading:

```
// Java Program to Implement
// Method Overloading
import java.io.*;

class MethodOverloadingEx {

    static int add(int a, int b) { return a + b; }

    static int add(int a, int b, int c)
    {
```

```
        return a + b + c;
    }

    // Main Function
    public static void main(String args[])
    {
        System.out.println("add() with 2 parameters");
        // Calling function with 2 parameters
        System.out.println(add(4, 6));

        System.out.println("add() with 3 parameters");
        // Calling function with 3 Parameters
        System.out.println(add(4, 6, 7));
    }
}
```

Output

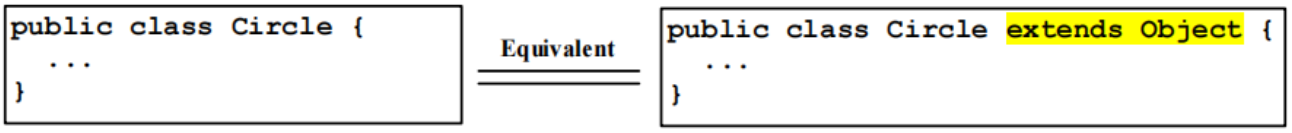
```
eat() method of derived class
Dog is eating.
eat() method of base class
eating.
eat() method of derived class
Dog is eating.
eat() method of base class
eating.
```

The Object Class and Its Methods

Overview: Every class in Java inherits from the `Object` class, which provides several methods such as `toString()`, `equals()`, and `hashCode()`.

Note : Every class in Java is descended from the java.lang.Object class. If no inheritance is

specified when a class is defined, the superclass of the class is Object.



The toString() Method in Object

Purpose: The `toString()` method returns a string representation of the object.

Example:

```
public class Dog {
    private String name;

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


    @Override
    public String toString() {
```

```
        return "Dog[name=" + name + "]";}

    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        System.out.println(dog);}}
```

Output:

```
Dog[name=Buddy]
```

Polymorphism, Dynamic Binding, and Generic Programming

Polymorphism:

- The ability of an object to take many forms.
- A parent class reference can refer to a child class object.

Types of Polymorphism:

1. **Compile-time (Static) Polymorphism:** Method overloading.
2. **Runtime (Dynamic) Polymorphism:** Method overriding.

Example:

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

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

class Cat extends Animal {
```

```
 @Override
 public void sound() {
System.out.println("Cat meows"); } }

public class TestPolymorphism {
public static void main(String[] args) {
 Animal a; a = new Dog(); a.sound(); // Outputs: Dog
barks a = new Cat(); a.sound(); // Outputs: Cat meows
} }
```

Method Matching Vs Binding

Method Matching:

- Method matching occurs at compile time and refers to determining which method to call based on the method signature. It involves selecting the correct method overload among multiple methods with the same name but different parameter lists.

- In this example, the compiler determines which `print` method to call based on the argument type.

Example:

```
public class MethodMatching {
    public void print(int a) {
        System.out.println("Integer: " + a);
    }

    public void print(String a) {
        System.out.println("String: " + a);
    }
```

```
    public static void main(String[] args) {
        MethodMatching obj = new MethodMatching();
        obj.print(10); // Calls print(int a)
        obj.print("Hello"); // Calls print(String a)
    }}
```

Binding:

- The method implementation to be called is determined at runtime (dynamic binding) or compile-time (static binding).

- **Static Binding (Early Binding):** Method calls are resolved at compile time. Typically applies to static, private, and final methods, which cannot be overridden.
- In this example, the `display` method is bound at compile time because it is private and cannot be overridden.

Example:

```
public class StaticBinding {
    private void display() {
        System.out.println("StaticBinding
display");
    }

    public static void main(String[] args) {
        StaticBinding obj = new StaticBinding();
        obj.display(); // Resolved at compile time
    }}
```

Casting Objects

Explanation: Casting is converting one type of object reference to another.

- **Upcasting (implicit)** is converting a subclass reference to a superclass reference.
- **Downcasting (explicit)** is converting a superclass reference to a subclass reference.

Example:

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

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

    public void bark() {
        System.out.println("Dog barks loudly");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        animal.sound();
```

```
        Dog dog = (Dog) animal; // Downcasting
        dog.bark();
    }
}
```

Output:

```
Dog barks

Dog barks loudly
```

Casting from Superclass to Subclass

Explanation: Downcasting is casting from a superclass reference to a subclass reference. It requires an explicit cast and can throw a `ClassCastException` if the object being cast is not actually an instance of the subclass.

Note : This type of casting may not always succeed.

Example:

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

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

```
public class TestCasting {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        animal.makeSound(); // Outputs: Dog barks

        // Downcasting
        if (animal instanceof Dog) {
            Dog dog = (Dog) animal; // Explicit downcast
            dog.bark(); // Outputs: Dog barks loudly
        }
    }}
```

The equals Method

Purpose: The `equals()` method is used to compare the contents of two objects for equality.

NOTE : The (==) comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The equals method is intended to test whether two objects have the same test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

Example:

```
public class Dog {
    private String name;

    public Dog(String name) {
        this.name = name;
```

```
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass())
return false;
        Dog dog = (Dog) obj;
        return name.equals(dog.name);
    }

    public static void main(String[] args) {
        Dog dog1 = new Dog("Buddy");
        Dog dog2 = new Dog("Buddy");
        System.out.println(dog1.equals(dog2)); // true
    }
}
```

Output:

```
true
```

The instanceof Operator

The `instanceof` operator is used to test whether an object is an instance of a specific class or a subclass of that class. It returns a **boolean value**: `true` if the object is an instance of the specified class or its subclass, and `false` otherwise.

Example:

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

```
    }}

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

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

public class TestInstanceOf {
    public static void main(String[] args) {
        // Creating objects of the classes
        Animal myAnimal = new Dog(); // Upcasting:
Converting Dog object to Animal type
        Animal anotherAnimal = new Cat(); // Upcasting:
Converting Cat object to Animal type

        // Using instanceof to check the type before
downcasting
        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal; // Downcasting:
Converting Animal object to Dog type
            myDog.bark(); // Calling the method specific
to Dog object
        }

        if (anotherAnimal instanceof Cat) {
```

```
            Cat myCat = (Cat) anotherAnimal; //
Downcasting: Converting Animal object to Cat type
            myCat.meow(); // Calling the method specific
to Cat object
        }

        // Demonstrating a failed instanceof check
        if (myAnimal instanceof Cat) {
            Cat myCat = (Cat) myAnimal; // This block will not
be executed
            myCat.meow();
        } else {
            System.out.println("myAnimal is not an instance of
Cat");}}}
```

Output :

```
Dog barks
Cat meows
myAnimal is not an instance of Cat
```

The protected Modifier

Definition: The `protected` modifier allows access to the member variables and methods within the same package and subclasses.

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-