Chapter Three

Inheritance and Polymorphism

Mulugeta G. and Sufian K.

Inheritance

- Defined as the process where one class acquires the properties (methods and fields) of another.
- With the use of inheritance the information is made manageable in a hierarchical order.
- Allows one class to inherit the properties and behaviors (fields and methods) of another class.
- The class which inherits the properties of other is known as subclass (derived class, child class)
- The class whose properties are inherited is known as superclass (base class, parent class).

Extends keyword

- **extends is the keyword used to inherit the properties of a class.**
- **Syntax:**

```
class Super {
    .....
    .....
}
class Sub extends Super {
    .....
    .....
}
```

Note:

- A **subclass** inherits all the members (fields, methods, and nested classes) from its superclass.
- Constructors are **not members**, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Exercises**
  - Write a java program that a Car class inherits some functionality and properties of Vehicle class
  - Write a java program that a ComputerTeacher can inherit some fields and methods from the Teacher class
  - Write a java program that a ComputerStudent inherits all the fields and method of Student class

The Super keyword

- The super keyword is similar to this keyword.
- Following are the scenarios where the super keyword is used.
  - It is used to differentiate the members of superclass from the members of subclass, if they have same names.
  - It is used to invoke the superclass constructor from subclass.

Differentiating the members

- If a class is inheriting the properties of another class. And if the members of the superclass have **the same names** as the sub class, to differentiate these variables we use **super keyword** as shown below.
  - **super.variable**
  - **super.method();**
- In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named display() with different implementations, and a variable named num with different values. We are invoking display() method of both classes and printing the value of the variable num of both classes.
- Here you can observe that we have used **super keyword** to differentiate the members of superclass from subclass.

Con't

```
class Super_class {
int num = 20; // display method of superclass
public void display() {
System.out.println("This is the display method of superclass");
        }
}
public class Sub_class extends Super_class {
        int num = 10; // display method of sub class
    public void display() {
System.out.println("This is the display method of subclass");
}}
```

Con't

```
public void my_method() {
// Instantiating subclass
Sub_class sub = new Sub_class();
// Invoking the display() method of sub class
sub.display();
// Invoking the display() method of superclass
super.display();
// printing the value of variable num of subclass
System.out.println("value of the variable named num in sub class:"+
sub.num);
// printing the value of variable num of superclass
System.out.println("value of the variable named num in super
class:"+ super.num); }
 public static void main(String args[]) {
Sub_class obj = new Sub_class();
obj.my_method(); }}
```

Example2

```
class Parent{
        String name;
        }
public class Child extends Parent{
        String name;
   Public void details(){
    super.name;
    name;
System.out.println(super.name+" "+name);
        }
Public static void main(String args[]){
        Child c=new Child();
        c.details();
}
}
```

Invoking superclass constructor

- **If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass.**
- **But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.**
  - **super(values);**
- This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a string value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Con't

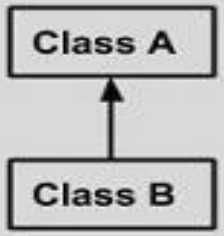
```
class Superclass {
 int age;
Superclass(int age) {
this.age = age; }
public void getAge() {
System.out.println("The value of the variable named age in
super class is: " +age); }
 }
public class Subclass extends Superclass {
Subclass(int age) {
super(age); }
 public static void main(String args[]) {
Subclass s = new Subclass(24);
s.getAge(); }}
```

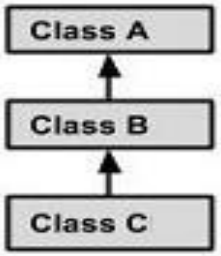
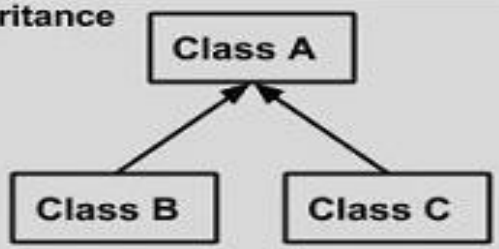
```
The value of the variable named age in super class is: 24
```

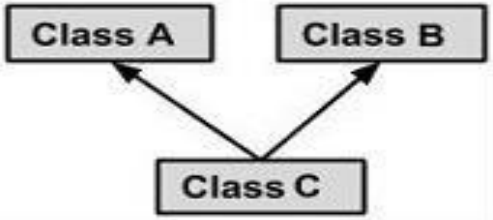
Types of Inheritance

- **Single Inheritance**
  - a subclass inherits from only one parent class.
- **Multilevel Inheritance**
  - a subclass inherits from a class, which in turn inherits from another class.
  - This forms a chain of inheritance.
- **Hierarchical Inheritance**
  - multiple subclasses inherit from a single parent class.
- **Multiple Inheritance**
  - a subclass can inherit from more than one parent class.

Types of Inheritance

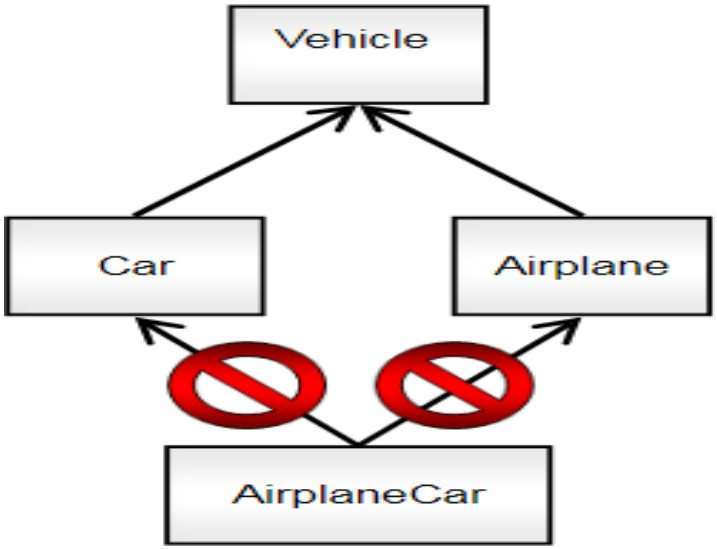
Single Inheritance 	public class A {
	.....
	}
	public class B **extends** A {
	.....
	}

Multi Level Inheritance 	public class A { .....} public class B **extends** A {.....} public class C **extends** B {..... }
Hierarchical Inheritance 	public class A { .....} public class B **extends** A {.....} public class C **extends** A {..... }

Multiple Inheritance 	public class A { .....}
	public class B {.....}
	public class C **extends** A,B {
	.....
	} // Java does not support mutiple Inheritance

Note:

- A very important fact to remember is that **Java does not support multiple inheritance**. This means that a class **cannot extend more than one class**.



Is-A Relationship

- **IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.**

```
public class Animal {
}

public class Mammal extends Animal {
}

public class Reptile extends Animal {
}

public class Dog extends Mammal {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true

Animal is the superclass of Mammal class.
Animal is the superclass of Reptile class.
Mammal and Reptile are subclasses of Animal class.
Dog is the subclass of both Mammal and Animal classes.

Con't

- Now, if we consider the IS-A relationship, we can say
  - Mammal IS-A Animal
  - Reptile IS-A Animal
  - Dog IS-A Mammal
  - Hence: Dog IS-A Animal as well
- With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.
- We can assure that Mammal is actually an Animal with the use of the instance operator.

Example

```
class Animal {}
class Mammal extends Animal {}
class Reptile extends Animal {}
public class Dog extends Mammal {
public static void main(String args[]) {
Animal a = new Animal();
Mammal m = new Mammal();
Dog d = new Dog();
System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal); }}
```

Con't

- Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.
- Generally, the **implements** keyword is used with classes to inherit the properties of an interface.
- Interfaces can **never be extended** by a class.

```
public interface Animal {}
public class Mammal implements Animal {}
public class Dog extends Mammal {
    }
```

The instanceof keyword

- Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

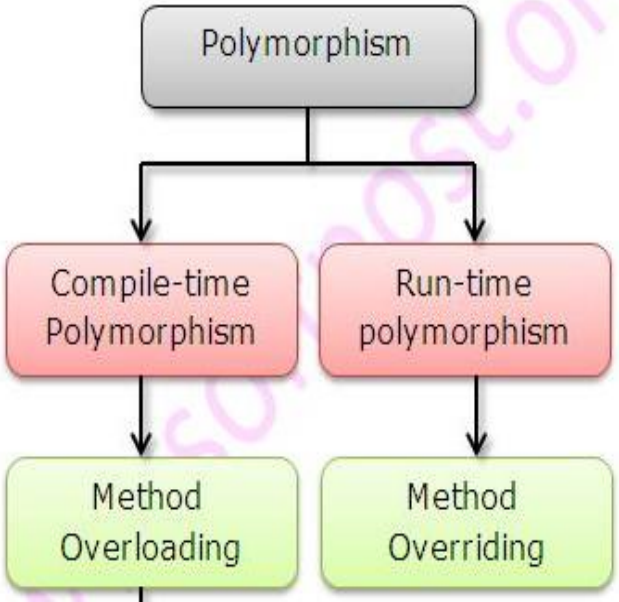
```
interface Animal{}
class Mammal implements Animal{}
public class Dog extends Mammal {
public static void main(String args[]) {
Mammal m = new Mammal();
Dog d = new Dog();
System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal); }}
```

Polymorphism

- **The ability of an object to take on many forms.**
- **Any Java object that can pass more than one ISA test is considered to be polymorphic.**
- **In Java, all Java objects are polymorphic since any object will pass the ISA test for their own type and for the class Object.**
- **polymorphism**' means '**having many forms**

Types of Java Polymorphism

- **Compile-Time Polymorphism**
- **Runtime Polymorphism**



Polymorphism

1. Compile-Time Polymorphism

- also known as static polymorphism
- Achieved by Method Overloading
- **NB:**
  - **Method Overloading**: This occurs when multiple methods with the same name exist in a class, but with different parameters (different number of parameters or parameter types).
  - **Constructor overloading:** a class can have any number of constructors that differ in parameter lists.

Example 1:

```
class Printer {
// Method Overloading (same method name, different parameters)
void print(int number) {
System.out.println("Printing integer: " + number);
 }
void print(String message) {
 System.out.println("Printing message: " + message);
}
public static void main(String[] args) {
Printer printer = new Printer();
printer.print(10);  // Calls print(int)
printer.print("Hello");  // Calls print(String)
 }
}
```

Example 2:

```
// Method overloading By using
// Different Types of Arguments
class Calculator {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {
        // Returns product of integer numbers
        return a * b;
    }
    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double a, double b)
    {
        // Returns product of double numbers
        return a * b;
    }
```

```
public static void main(String[] args)
{
 Calculator c = new Calculator();
        // Calling method by passing
        // input as in arguments
System.out.println(c.Multiply(2, 4));

System.out.println(c.Multiply(5.5,
6.3));
    }
}
```

<u>Output</u>

8

34.65

Polymorphism

2. Run-Time Polymorphism

- Also known as Dynamic Method Dispatch.
- It is a process in which a function call to the overridden method is resolved at Runtime.
- achieved by Method Overriding.

NB:

- **Method Overriding:**
  - occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
  - That superclass function is said to be overridden.

Example 1:

```
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 static void main(String[] args) {
Animal animal = new Dog();  // Animal reference but Dog object
animal.sound();  // Output: Dog barks
Animal animal = new Cat();  // Animal reference but Cat object
animal.sound();  // Output: Cat meows
        }
}
```

Example 2:

```
// Java Program for Method Overriding
// Class 1
class Parent {
    // Method of parent class
    void Print() {
        System.out.println("parent class");
    }
}
// Class 2
class subclass1 extends Parent {
    // Method
    void Print() {
      System.out.println("subclass1");
    } }
// Class 3
class subclass2 extends Parent {
    // Method
    void Print() {
        System.out.println("subclass2");}}
```

```
public static void main(String[] args) {
        // Creating object of class 1
        Parent a;
        // Now we will be calling print
methods
        // inside main() method
        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

<u>Output</u>
subclass1
subclass2

Polymorphism

- **Advantages of Polymorphism**
  - **Increases code reusability**
  - **Improves readability and maintainability of code**
  - **Supports dynamic binding, enabling the correct method to be called at runtime**
  - **Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.**
  - **Code Organization**

Polymorphism

- **Disadvantages of Polymorphism**
  - **Complexity**
    - Can make it more difficult to understand the behavior of an object, especially if the code is complex
  - **Performance Issues**
    - as polymorphic behavior may require additional computations at runtime.

Thank You