Unit 4: Inheritance & Packaging [3 Hrs.]

1. Inheritance

Inheritance is a mechanism in Java that allows one class to inherit the properties (fields and methods) of another class. The class that inherits is called the **subclass** (or derived class), and the class being inherited from is called the **superclass** (or base class).

1.1 Using the `extends` Keyword

The `extends` keyword is used to create a subclass(child) that inherits from a superclass(parent).

Lab 1: Using the `extends` Keyword

```java
// 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 myDog = new Dog();
        myDog.eat();  // Inherited method
        myDog.bark(); // Subclass method
    }
}
```

Sample Output:

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

1.2 Subclasses and Superclasses

- **Superclass**: The class whose properties are inherited.
- **Subclass**: The class that inherits the properties of the superclass.

Lab 2: Subclasses and Superclasses

```java
class Vehicle {
    void run() {
        System.out.println("Vehicle is running.");
    }
}
```

```
class Car extends Vehicle {
    void accelerate() {
        System.out.println("Car is accelerating.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.run();        // Inherited method
        myCar.accelerate(); // Subclass method
    }
}
```

Sample Output:

```
Vehicle is running.
Car is accelerating.
```

1.3 `super` Keyword Usage

- The `super` keyword is used to refer to the parent class or superclass. It can be used in various ways:

1. **Accessing Parent Class Methods**

- If a subclass overrides a method, we can use `super` to call the method from the superclass.

Lab 3: Accessing Parent Class Methods

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

class Dog extends Animal {
    @Override
    void sound() {
        super.sound();  // Calling the superclass method
        System.out.println("Dog barks");
    }
}

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

Sample Output:

```
Animal makes a sound
Dog barks
```

2. **Accessing Parent Class Constructor**

- We can use `super()` to call the constructor of the superclass. This must be the first statement in the subclass constructor.

  Lab 4: Accessing Parent Class Constructor

```java
class Animal {
    Animal() {
        System.out.println("Animal Constructor");
    }
}

class Dog extends Animal {
    Dog() {
        super();  // Calling the parent class constructor
        System.out.println("Dog Constructor");
    }
}

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

Sample Output:

```
Animal Constructor
Dog Constructor
```

3. **Accessing Parent Class Fields**

- We can use `super` to access fields of the parent class, especially if they are hidden by the subclass.

  Lab 5: Accessing Parent Class Fields

```java
class Animal {
    String name = "Animal";
}

class Dog extends Animal {
    String name = "Dog";

    void printNames() {
        System.out.println(name);          // Prints the subclass field
        System.out.println(super.name);    // Prints the superclass field
    }
}
```

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

Sample Output:

```
Dog
Animal
```

1.4 Overriding Methods

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

Lab 6: Overriding Methods

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

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

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound(); // Calls the overridden method
    }
}
```

Sample Output:

```
Dog barks.
```

1.5 Dynamic Method Dispatch (also known as runtime polymorphism)

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime rather than compile time.

- **Animal class** has a method `sound()`.
- **Dog class** and **Cat class** both extend `Animal` and override the `sound()` method to provide their specific implementations.
- **Dynamic Method Dispatch** occurs when we assign instances of `Dog` and `Cat` to `Animal` references. At runtime, the actual method to be executed is determined

by the object type (either `Dog` or `Cat`), not by the reference type (`Animal`).

Lab 7: Dynamic Method Dispatch

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myAnimal.sound();
        myDog.sound();
        myCat.sound();
    }
}
```

Sample Output:

```
Animal makes a sound.
Dog barks.
Cat meows.
```

Explanation of Dynamic Method Dispatch:

1. `myAnimal.sound()`:
    - The reference type is `Animal`, and the object type is `Animal`. So, it calls the `sound()` method in the `Animal` class.
2. `myDog.sound()`:
    - The reference type is `Animal`, but the object type is `Dog`. Because the `Dog` class overrides the `sound()` method, it calls the `sound()` method in the `Dog` class.

3. `myCat.sound()`:
    - The reference type is `Animal`, but the object type is `Cat`. Since the `Cat` class overrides the `sound()` method, it calls the `sound()` method in the `Cat` class.

- Even though the reference variables (`myAnimal`, `myDog`, `myCat`) are all of type `Animal`, the actual method that gets called is determined at runtime based on the object's class. This is what makes dynamic method dispatch possible in Java.

1.6 The `Object` Class

The `Object` class is the root of the class hierarchy in Java. Every class in Java is directly or indirectly derived from the `Object` class.

Lab 8: The `Object` Class

```
class MyClass {
    // This class implicitly extends Object
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println(obj.toString()); // Calls the toString() method from Object class
    }
}
```

Sample Output:

```
MyClass@1b6d3586
```

1.7 Abstract and Final Classes

- **Abstract Class**: A class that cannot be instantiated and is meant to be subclassed. It can contain abstract methods (methods without a body).
- **Final Class**: A class that cannot be subclassed.

Lab 9: Abstract and Final Classes

```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle.");
    }
}

final class FinalClass {
    void display() {
```

```java
        System.out.println("This is a final class.");
    }
}

public class Main {
    public static void main(String[] args) {
        Circle myCircle = new Circle();
        myCircle.draw();

        FinalClass finalObj = new FinalClass();
        finalObj.display();
    }
}
```

Sample Output:

```
Drawing a circle.
This is a final class.
```

2. Packages

Packages are used to organize classes and interfaces into namespaces.

2.1 Defining a Package

A package is defined using the `package` keyword at the top of a Java file.

Lab 10: Defining a Package

```java
package com.example;

public class MyClass {
    public void display() {
        System.out.println("This is MyClass in com.example package.");
    }
}
```

2.2 Importing a Package

The `import` keyword is used to import classes and interfaces from other packages.

Lab 10: Importing a Package

```java
import com.example.MyClass;

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

Sample Output:

```
This is MyClass in com.example package.
```

2.3 Access Control

Java provides access modifiers to control the visibility of classes, methods, and variables.

- **public**: Accessible from any other class.
- **protected**: Accessible within the same package and subclasses.
- **default (no modifier)**: Accessible within the same package.
- **private**: Accessible only within the same class.

Lab 11: Access Control

```java
package com.example;

public class AccessExample {
    public int publicVar = 1;
    protected int protectedVar = 2;
    int defaultVar = 3; // default access
    private int privateVar = 4;

    public void display() {
        System.out.println("Public: " + publicVar);
        System.out.println("Protected: " + protectedVar);
        System.out.println("Default: " + defaultVar);
        System.out.println("Private: " + privateVar);
    }
}
```

Sample Output:

```
Public: 1
Protected: 2
Default: 3
Private: 4
```

3. Interfaces

An interface is a reference type in Java that contains abstract methods. It can also contain constants, default methods, and static methods.

3.1 Defining an Interface

An interface is defined using the `interface` keyword.

Example:

```java
interface Animal {
    void sound();
}
```

3.2 Implementing and Applying Interfaces

A class implements an interface using the `implements` keyword.

Lab 12: Implementing and Applying Interfaces

```
interface Animal {
    void sound();
}

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

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

Sample Output:

```
Dog barks.
```

3.3 Default and Static Methods in Interfaces

Java 8 introduced default and static methods in interfaces.

Lab 13: Default and Static Methods in Interfaces

```
interface Animal {
    void sound(); // Abstract method

    default void sleep() { // Default method
        System.out.println("This animal sleeps.");
    }

    static void info() { // Static method
        System.out.println("This is an Animal interface.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound();
        myDog.sleep();
        Animal.info();
    }
}
```

Sample Output:

```
Dog barks.
This animal sleeps.
This is an Animal interface.
```

What students have learned

In this unit, students learned key **object-oriented programming (OOP)** concepts in Java, focusing on **inheritance**, **packages**, and **interfaces**.

1. Inheritance: Promotes code reusability and establishes relationships between classes.

- **Purpose**: Allows a class (subclass) to inherit properties (fields and methods) from another class (superclass).
- **Key Concepts**:
  - `extends` keyword: Used to create a subclass.
  - `super` keyword: Refers to the superclass's methods, constructors, or variables.
  - **Method Overriding**: Subclass provides a specific implementation of a superclass method.
  - **Dynamic Method Dispatch**: Runtime polymorphism for overridden methods.
  - **Object Class**: All classes implicitly inherit from the `Object` class.
  - **Abstract Classes**: Cannot be instantiated; may contain abstract methods.
  - **Final Classes**: Cannot be subclassed.

---

2. Packages: Organize code and manage access control.

- **Purpose**: Organize classes and interfaces into namespaces to avoid naming conflicts.
- **Key Concepts**:
  - **Defining a Package**: Use the `package` keyword.
  - **Importing a Package**: Use the `import` keyword to access classes from other packages.
  - **Access Control**: Use access modifiers (`public`, `protected`, `private`, default) to control visibility.

---

3. Interfaces: Enable abstraction, polymorphism, and multiple inheritance.

- **Purpose**: Define a contract (set of methods) that classes must implement.
- **Key Concepts**:
  - **Defining an Interface**: Use the `interface` keyword.
  - **Implementing Interfaces**: Use the `implements` keyword to provide method implementations.
  - **Applying Interfaces**: Enable abstraction and multiple inheritance.
