Day 12

Name : DHANAPAL

Date : 09/08/2024

1. Implement Abstract Class with Overloading and Overriding:

Abstract Class: A class that cannot be instantiated and may contain abstract methods that must be implemented by its subclasses.

Overloading: Same method name but different parameters (either in number or type).

Overriding: A subclass provides a specific implementation of a method that is already defined in its superclass.

Code:

```
abstract class Animal {
  // Abstract method (no implementation)
  abstract void sound();

  // Overloaded method (same name, different parameters)
  void eat() {
    System.out.println("Animal is eating");
  }

  void eat(String food) {
    System.out.println("Animal is eating " + food);
  }
}
// Subclass of Animal
class Dog extends Animal {
```

```
    // Implementing the abstract method
    @Override
    void sound() {
        System.out.println("Dog barks");
    }

    // Overriding the eat method
    @Override
    void eat() {
        System.out.println("Dog is eating");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound();      // Output: Dog barks
        myDog.eat();        // Output: Dog is eating
        myDog.eat("bone");  // Output: Animal is eating bone
    }
}
```

Output:

```
Dog barks
Dog is eating
Animal is eating bone
```

2. Implement Multiple Inheritance with Interface

Multiple Inheritance in Java can be achieved using interfaces since Java doesn't allow extending more than one class.

Code

```
interface Flyable {
  void fly();
}

interface Runnable {
  void run();
}

// Class that implements both interfaces
class Bird implements Flyable, Runnable {
  @Override
  public void fly() {
    System.out.println("Bird is flying");
  }

  @Override
  public void run() {
    System.out.println("Bird is running");
  }
}

public class Main {
  public static void main(String[] args) {
    Bird sparrow = new Bird();
```

```
        sparrow.fly();  // Output: Bird is flying
        sparrow.run();  // Output: Bird is running
    }
}
```

Output:

```
Bird is flying
Bird is running
```

3. Show Final Methods in the Class that Can't Be Overridden

Final Methods: Methods that cannot be overridden by subclasses.

Code

```
class Vehicle {
    // Final method
    final void start() {
        System.out.println("Vehicle is starting");
    }

    void stop() {
        System.out.println("Vehicle is stopping");
    }
}
// Subclass of Vehicle
class Car extends Vehicle {
    // This method cannot be overridden because it's final
    /*
    void start() {
        // Error: Cannot override the final method from Vehicle
    }
```

```
  */

  @Override
  void stop() {
    System.out.println("Car is stopping");
  }
}
public class Main {
  public static void main(String[] args) {
    Vehicle myCar = new Car();
    myCar.start(); // Output: Vehicle is starting
    myCar.stop();  // Output: Car is stopping
  }
}
```

Output:
Vehicle is starting
Car is stopping

Loose Coupling Hint

In the examples above, particularly in the first example, notice how we use `Animal` as the reference type in `main`:

`Animal myDog = new Dog();`
This is an example of **loose coupling**. The code depends on the `Animal` base class rather than on a specific subclass (`Dog`), allowing flexibility and easier maintenance.