Designing Interfaces and Abstract Classes

Estimated time needed: 20 minutes

In this lab, you will learn about the purpose of interfaces and abstract classes and the process of designing those.

You are currently viewing this lab in a Cloud based Integrated Development Environment (Cloud IDE). It is a fully-online integrated development environment that is pre-installed with JDK 21, allowing you to code, develop, and learn in one location.

Learning Objectives

After completing this lab, you will be able to:

- Understand what abstract methods are
- Create abstract class and interface
- Use polymorphism with abstract class and interface
- Choose between abstract class and interface
- Explore multiple inheritance in Java with interfaces

Abstract methods

Abstract methods are methods which are defined but not implemented. By defining a method, you give it a name, specify the parameters it will take, specify the access and also the return type. But the body of the method is empty. Abstract methods can only be defined as class methods. Meaning, they can't be `static`. Any class containting an abstract method, should be defined as an abstract class. In this step you will define abstract method in a class.

1. Create a project directory by running the following command.

```
mkdir my_impl_proj
```

2. Run the following code to create the directory structure.

```
mkdir -p my_impl_proj/src
mkdir -p my_impl_proj/classes
mkdir -p my_impl_proj/test
cd my_impl_proj
```

3. Now create a file named `ClassWithAbstractMeth.java` inside the src directory.

```
touch /home/project/my_impl_proj/src/ClassWithAbstractMeth.java
```

4. Click the following button to open the file for editing.

Open **ClassWithAbstractMeth.java** in IDE

5. Read each statement in the following program and understand how there is an abstract method in the class and also a static main method to run the class. Paste the following content in `ClassWithAbstractMeth.java`.

```
public abstract class ClassWithAbstractMeth {
    public abstract  String absMeth1(int num);
    protected abstract boolean absMeth2(String str);
    abstract float absMeth3(int num, String str);
    public static void main(String[] args) {
        System.out.println("This is a class with abstract methods");
    }
}
```

`public abstract class ClassWithAbstractMeth` - This will declare a class as an abstract class.
`public abstract String absMeth1(int num);` - Defines an abstract method which is public, that takes an int as a parameter and returns a String, this can be overridden in any inheriting subclass.
`protected abstract boolean absMeth2(String str);` - Defines an abstract method which is protected, that takes a String as a parameter and returns a boolean. This can be overridden in subclasses within the package and in subclasses outside of the package.
`abstract float absMeth3(int num, String str);` - Defines an abstract method which is default (package private), that takes a String as a parameter and returns a boolean. This can be overridden only in inheriting subclasses within the package.

6. Compile the java program, specifying the destination directory as the `classes` directory that you created.

```
javac -d classes src/ClassWithAbstractMeth.java
```

This compilation will be successful and you will be able to run the class.

7. Change the content of `ClassWithAbstractMeth.java` to have the following code.

```
public abstract class ClassWithAbstractMeth {
    public abstract static int absMeth1(String s);
    private abstract String absMeth2();
    public abstract  String absMeth3(int num);
    protected abstract boolean absMeth4(String str);
    abstract float absMeth5(int num, String str);
    public static void main(String[] args) {
        System.out.println("This is a class with abstract methods");
    }
}
```

`public abstract static int absMeth1(String s)` - This definition is not possible. It will throw a compilation error as abstract methods cannot be static
`private abstract String absMeth2()` - This definition is not possible. It will throw a compilation error as abstract methods cannot be private. The purpose of these methods are for these to be implemented in the child class. Child classes do not have access to super class's private methods.

The other definitions of abstract methods as seen previously, are valid.

8. Run the following command to compile `ClassWithAbstractMeth.java` .

```
javac -d classes src/ClassWithAbstractMeth.java
```

9. This compilation will fail. Read the compilation errors to observe that the abstract methods can't be private or static.

```
ClassWithAbstractMeth.java:2: error: illegal combination of modifiers: abstract and static
        public abstract static int absMeth1(String s);
                                   ^
ClassWithAbstractMeth.java:3: error: illegal combination of modifiers: abstract and private
        private abstract String absMeth2();
```

10. Correct the class definition by removing the erroneous method definitions and compile it again.

Now the compilation goes through.

Abstract class definition

Please note that an abstract method has to be in an abstract class, but an abstract class doesn't have to have an abstract method. When you don't want a class to be `instantiated` and only be `inherited`, you define it as an abstract class. Consider the following example. There is `Animal` class. There are `Dog`,`Cat` and `Cow` classes which inherit. There is no point in creating an instance of Animal class. In such a case, you can define Animal class as an abstract class.

1. Create a file named `Animal.java` inside the src directory.

```
touch /home/project/my_impl_proj/src/Animal.java
```

2. Click the following button to open the file for editing.

Open **Animal.java** in IDE

3. Read each statement in the following program and understand you create the `Animal` class just as you had created `Animal` class in the previous lab on polymorphism.

```
public abstract class Animal {
    private String name;
    private String food;
    public Animal(String name) {
        this.name = name;
    }
    public String sound() {
        return null;
    }
    public void setFood(String food) {
        this.food = food;
    }
    public String getFood(){
        return this.food;
    }
    public String toString() {
        return name.concat(" says ").concat(sound().concat(" and eats ").concat(food));
    }
}
class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }
    public String sound() {
        return "Woof";
    }
}
class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }
    public String sound() {
        return "Meow";
    }
}
class Cow extends Animal {
    public Cow(String name) {
        super(name);
    }
    public String sound() {
        return "Moo";
    }
}
```

See that `Animal` class is the same as used before. The only difference is that it is now declared as abstract.

4. Create a file named `AnimalFarm.java` inside the src directory.

```
touch /home/project/my_impl_proj/src/AnimalFarm.java
```

5. Click the following button to open the file for editing.

Open **AnimalFarm.java** in IDE

6. Read each statement in the following program. Recollect how you created Animal objects from the command line tool.

```
import java.util.Scanner;
public class AnimalFarm {
    public static void main(String s[]) {
        Scanner scanner = new Scanner(System.in);
        Animal[] animals = new Animal[10];
        int anmlIdx = 0;
        while(true) {
            System.out.println( "Press 1 to view the animals, " +
                                "\n2 to add animals, "+
                                "\nany other key to exit");
            String userAction = scanner.nextLine();
            if (userAction.equals("1")) {
                for(int i=0;i<animals.length;i++) {
                    if(animals[i] != null) {
                        System.out.println(animals[i]);
                    }
                }
            } else if (userAction.equals("2")) {
                if(anmlIdx == 10) {
                    System.out.println("10 animals added already. Cannot add any more animals!");
                    continue;
                }
                System.out.println("Which animal do you want to create? \nPress 1 for dog,"+
                                    "\n2 for cat " +
                                    "\n3 for cow" );
                String animalChoice = scanner.nextLine();
                if (animalChoice.equals("1")) {
                    System.out.println("Enter the name of the dog");
                    String dogName = scanner.nextLine();
                    Animal anmlTmp = new Dog(dogName);
                    System.out.println("Enter what the dog eats");
                    String dogFood = scanner.nextLine();
                    anmlTmp.setFood(dogFood);
                    animals[anmlIdx++] = anmlTmp;
                } else if (animalChoice.equals("2")) {
                    System.out.println("Enter the name of the cat");
                    String catName = scanner.nextLine();
                    Animal anmlTmp = new Cat(catName);
                    System.out.println("Enter what the cat eats");
                    String catFood = scanner.nextLine();
                    anmlTmp.setFood(catFood);
                    animals[anmlIdx++] = anmlTmp;
                } else if (animalChoice.equals("3")) {
                    System.out.println("Enter the name of the cow");
                    String cowName = scanner.nextLine();
                    Animal anmlTmp = new Cow(cowName);
                    System.out.println("Enter what the cow eats");
                    String cowFood = scanner.nextLine();
                    anmlTmp.setFood(cowFood);
                    animals[anmlIdx++] = anmlTmp;
                }
            } else {
                break;
            }
        }
    }
}
```

As you can see, you are using `Animal` type to refer to the Dog, Cat and Cow objects. But you cannot create a new Animal such as, `new Animal()`. It will throw a compilation error.

7. Run the following command to compile `Animal.java` and `AnimalFarm.java`.

```
javac -d classes src/Animal*.java
```

8. Set the `CLASSPATH` variable.

```
export CLASSPATH=$CLASSPATH:/home/project/my_impl_proj/classes
```

9. Now, when you run the java program, it will run seamlessly as expected.

```
java AnimalFarm
```

One child class can only inherit from one super class. Java doesn't allow multiple inheritance, the ability for a child class to inherit from more than one super class.

Multiple Inheritance

Let's assume that there is a class called `Animal` and there is a class called `Mammal`. A dog is both an animal and a mammal. The expectation would be for class `Dog` to inherit from both class. But to achieve this in Java, you have special kind of structure called `Interfaces`. An interface is very similar to a class but cannot have any class attributes and can have only abstract methods. An example for interface is as follows.

```
public interface Mammal {
    public abstract void reproduce(int gest_period);
}
```

The class which is implementing this interface, will have to implement this method. You may recall implementing `Cloneable` interface to make a class cloneable in this lab.

Imagine you are designing a system for smart home devices. You have different types of devices, such as:

Switchable: Devices that can be turned on or off.

Adjustable: Devices whose settings (for example, brightness and volume) can be adjusted.

Connectable: Devices that can connect to a network.

A smart bulb, for example, can be switchable, adjustable, and connectable. Whereas a dimmable bulb can be switchable and adjustable. A regular bulb is only switchable. To model this, you can use multiple interfaces.

1. Create a file named `Bulb.java` inside the src directory.

```
touch /home/project/my_impl_proj/src/Bulb.java
```

2. Click the following button to open the file for editing.

Open **Bulb.java** in IDE

3. Read each statement in the following program. Paste the following code in `Bulb.java`.

```
// Interface for devices that can be turned on/off
interface Switchable {
    void turnOn();
    void turnOff();
}
// Interface for devices with adjustable settings
interface Adjustable {
    void increase();
    void decrease();
}
// Interface for devices that can connect to a network
interface Connectable {
    void connect();
    void disconnect();
}
```

```java
// SmartBulb class implementing all three interfaces
class SmartBulb implements Switchable, Adjustable, Connectable {
    private boolean isOn = false;
    private int brightness = 50; // Default brightness level
    private boolean isConnected = false;
    // Switchable methods
    @Override
    public void turnOn() {
        isOn = true;
        System.out.println("SmartBulb is turned ON.");
    }
    @Override
    public void turnOff() {
        isOn = false;
        System.out.println("SmartBulb is turned OFF.");
    }
    // Adjustable methods
    @Override
    public void increase() {
        if (brightness < 100) {
            brightness += 10;
            System.out.println("Brightness increased to " + brightness + "%.");
        } else {
            System.out.println("Brightness is already at maximum.");
        }
    }
    @Override
    public void decrease() {
        if (brightness > 0) {
            brightness -= 10;
            System.out.println("Brightness decreased to " + brightness + "%.");
        } else {
            System.out.println("Brightness is already at minimum.");
        }
    }
    // Connectable methods
    @Override
    public void connect() {
        isConnected = true;
        System.out.println("SmartBulb is connected to the network.");
    }
    @Override
    public void disconnect() {
        isConnected = false;
        System.out.println("SmartBulb is disconnected from the network.");
    }
}
// DimmableBulb class implementing two interfaces
class DimmableBulb implements Switchable, Adjustable {
    private boolean isOn = false;
    private int brightness = 50; // Default brightness level
    // Switchable methods
    @Override
    public void turnOn() {
        isOn = true;
        System.out.println("DimmableBulb is turned ON.");
    }
    @Override
    public void turnOff() {
        isOn = false;
        System.out.println("DimmableBulb is turned OFF.");
    }
    // Adjustable methods
    @Override
    public void increase() {
        if (brightness < 100) {
            brightness += 10;
            System.out.println("Brightness increased to " + brightness + "%.");
        } else {
            System.out.println("Brightness is already at maximum.");
        }
    }
    @Override
    public void decrease() {
        if (brightness > 0) {
            brightness -= 10;
            System.out.println("Brightness decreased to " + brightness + "%.");
        } else {
            System.out.println("Brightness is already at minimum.");
        }
    }
}
// RegularBulb class implementing one interface
class RegularBulb implements Switchable {
    private boolean isOn = false;
    // Switchable methods
    @Override
    public void turnOn() {
        isOn = true;
        System.out.println("RegularBulb is turned ON.");
    }
    @Override
    public void turnOff() {
        isOn = false;
        System.out.println("RegularBulb is turned OFF.");
    }
}
```

Interfaces:

- `Switchable` Defines methods for turning a device on or off.
- `Adjustable` Defines methods for increasing or decreasing settings.
- `Connectable` Defines methods for connecting or disconnecting a device.

SmartBulb Class:

Implements all three interfaces (Switchable, Adjustable, Connectable).

Dimmable Class:

Implements two interfaces (Switchable, Adjustable).

RegularBulb Class:

Implements one interface only (Switchable).

4. Create a file named `BulbOperate.java` inside the src directory.

```
touch /home/project/my_impl_proj/src/BulbOperate.java
```

5. Click the following button to open the file for editing.

Open **BulbOperate.java** in IDE

6. Read each statement and the code comments in the following program. Paste the following code in `BulbOperate.java`.

```
public class BulbOperate {
    public static void main(String s[]) {
        // Create an array of Switchable objects to hold different types of bulbs
        Switchable switchables[] = new Switchable[3];
        // Create instances of SmartBulb, DimmableBulb, and RegularBulb
        SmartBulb sb = new SmartBulb(); // SmartBulb implements Switchable, Adjustable, and Connectable
        DimmableBulb db = new DimmableBulb(); // DimmableBulb implements Switchable and Adjustable
        RegularBulb rb = new RegularBulb(); // RegularBulb implements only Switchable
        // Populate the array with the bulb instances
        switchables[0] = sb; // Add SmartBulb to the array
        switchables[1] = db; // Add DimmableBulb to the array
        switchables[2] = rb; // Add RegularBulb to the array
        // Loop through the array and turn each bulb on and off
        for (int i = 0; i < switchables.length; i++) {
            switchables[i].turnOn(); // Turn on the bulb
            switchables[i].turnOff(); // Turn off the bulb
        }
        // Increase brightness of the SmartBulb directly
        sb.increase(); // Calls the increase() method of SmartBulb
        // Decrease brightness of the DimmableBulb by casting to Adjustable
        ((Adjustable) switchables[1]).decrease(); // Cast switchables[1] (DimmableBulb) to Adjustable and call decrease()
        // Connect the SmartBulb to the network by casting to Connectable
        ((Connectable) switchables[0]).connect(); // Cast switchables[0] (SmartBulb) to Connectable and call connect()
    }
}
```

Polymorphism: The `Switchable` interface allows treating all bulb types uniformly in the array.

Type Casting: To access methods specific to Adjustable or Connectable, the objects are cast to those interfaces.

7. Compile the class, specifying the destination directory as the `classes` directory that you created.

```
javac -d classes src/Bulb*.java
```

This will compile both Bulb.java and BulbOperate.java.

8. Run the java program BulbOperate.

```
java BulbOperate
```

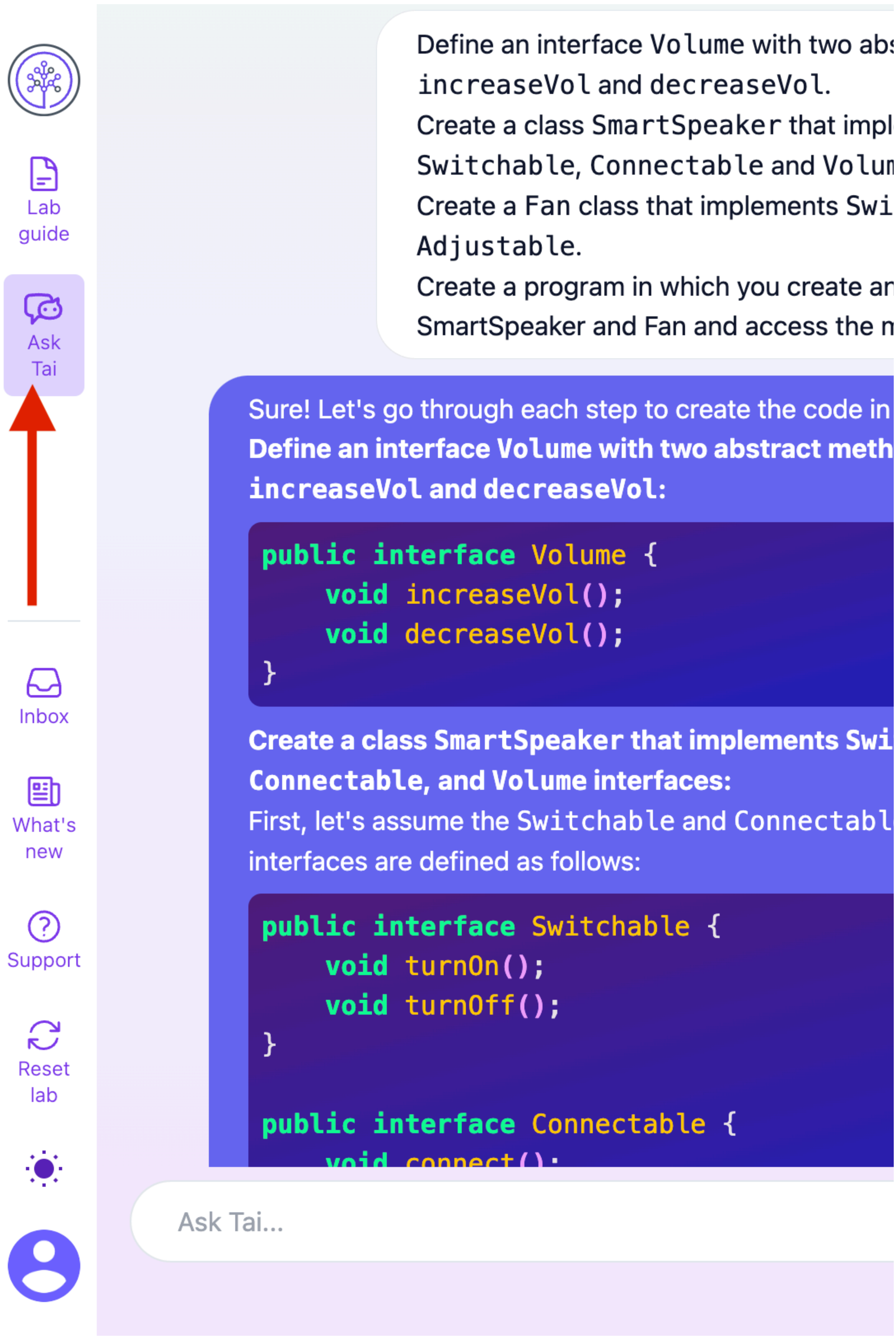
You will see the output as below:

```
SmartBulb is turned ON.
SmartBulb is turned OFF.
DimmableBulb is turned ON.
DimmableBulb is turned OFF.
RegularBulb is turned ON.
RegularBulb is turned OFF.
Brightness increased to 60%.
Brightness decreased to 40%.
SmartBulb is connected to the network.
```

Practice Exercise

1. Define an interface `Volume` with two abstract method, `increaseVol` and `decreaseVol`.
2. Create a class `SmartSpeaker` that implements `Switchable`, `Connectable` and `Volume` interfaces.
3. Create a `Fan` class that implements `Switchable` and `Adjustable`.
4. Create a program in which you create an object of SmartSpeaker and Fan and access the methods.

Hint: Use Generative AI avalable in the lab environment to complete this exercise.



Define an interface Volume with two abs
increaseVol and decreaseVol.
Create a class SmartSpeaker that impl
Switchable, Connectable and Volum
Create a Fan class that implements Swi
Adjustable.
Create a program in which you create ar
SmartSpeaker and Fan and access the m

Sure! Let's go through each step to create the code in
Define an interface Volume with two abstract meth
increaseVol and decreaseVol:

```
public interface Volume {
    void increaseVol();
    void decreaseVol();
}
```

Create a class SmartSpeaker that implements Swi
Connectable, and Volume interfaces:
First, let's assume the Switchable and Connectabl
interfaces are defined as follows:

```
public interface Switchable {
    void turnOn();
    void turnOff();
}

public interface Connectable {
    void connect();
```

▶ Click here for sample code

Conclusion

In this lab, you learned about abstract classes and interfaces and learned how to use them.

Author(s)

Lavanya