LAB MANUAL

CS1421: Object Oriented Programming
LAB 08

Instructors
Safiyah Batool
Samia Bashir

Interfaces and Abstract Classes And Methods

Objective

After completing this lab, the students should be able to

- Understand what are interfaces
- Understand what are Abstract Classes and Abstract Methods
- Understand the usage of interfaces
- Create, implement and extend interfaces

Abstract Classes and Methods

- Abstract class in java is a class declared with **abstract** keyword.
- Abstract class may contain one or more abstract methods.
- An abstract method in java, is a method that does not have implementation (no body) and it is also declared with **abstract** keyword.
- A class that extends from abstract class must provide implementation (body) for the abstract methods.
- Abstract class cannot be instantiated.
- If there is any abstract method in a class, then that class must be abstract.

Syntax

```
access_modifier abstract class ClassName
{
  //Abstract Method
  access_modifier abstract return_type methodName(Parameters);
}
```

Interface

The interface keyword takes the abstract concept one step further. You could think of it as a "pure" abstract class. Each interface is compiled into a separate bytecode file, just like a regular class. An interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default. According to the Java Language Specification, it's proper style to declare an interface's abstract methods without keywords public and abstract, because they're redundant in interface-method declarations. Similarly, an interface's constants should be declared without keywords public, static and final, because they, too, are redundant.

An interface is often used when disparate classes—i.e., classes that are not related by a class hierarchy—need to share common methods and constants. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to the same method calls. You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality.

Syntax

```
modifier interface InterfaceName
 {
 // Constant declarations
// Abstract method signatures
 }
```

Using an Interface

To use an interface, a concrete class must specify that it **`implements`** the interface and must declare each method in the interface with the signature specified in the interface declaration. To specify that a class implements an interface, add the implements keyword and the name of the interface to the end of your class declaration's first line. A class that does not implement all the methods of the interface is an abstract class and must be declared abstract.
Implementing an interface is like signing a contract with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class abstract."
A Java class can implement multiple Java interfaces. In that case the class must implement all the methods declared in all the interfaces implemented. If the interfaces are not located in the same packages as the implementing class, you will also need to import the interfaces. Java interfaces are imported using the import instruction just like Java classes.

Interfaces and Inheritance

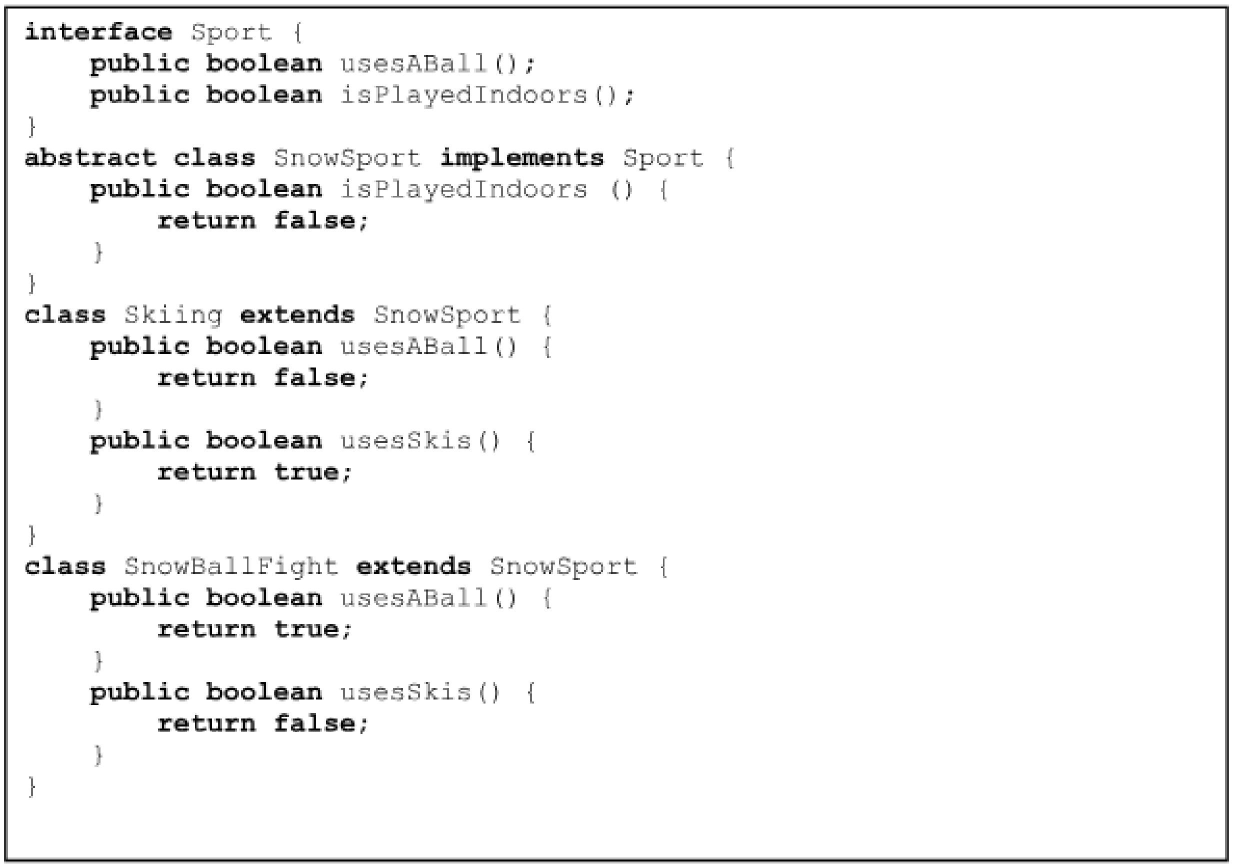
It is possible for a Java interface to inherit from another Java interface, just like classes can inherit from other classes. You specify inheritance using the `extends` keyword.

Example:

```
public interface a {
	public int add (int a, int b);
}


public abstract class ab {
	public abstract int sub(int a, int b);
}
```

```
public class imp extends ab implements a {

       public  int add(int a, int b)
       {

       return a + b;
       }

       public int sub(int c, int d)
       {
       return c-d;
       }

       public static void main(String[]args) {

       imp i= new imp();
       System.out.println(i.add(3,4));
       System.out.println(i.sub(3,4));
       }
}
```

Practice Tasks



```
interface Sport {
    public boolean usesABall();
    public boolean isPlayedIndoors();
}
abstract class SnowSport implements Sport {
    public boolean isPlayedIndoors () {
        return false;
    }
}
class Skiing extends SnowSport {
    public boolean usesABall() {
        return false;
    }
    public boolean usesSkis() {
        return true;
    }
}
class SnowBallFight extends SnowSport {
    public boolean usesABall() {
        return true;
    }
    public boolean usesSkis() {
        return false;
    }
}
```

```
a. Sport s = new Sport();
   System.out.println(s.usesABall());

b. SnowBallFight f = new SnowBallFight();
   System.out.println(f.usesABall());

c. SnowSport s = new Skiing();
   System.out.println(s.usesABall());

d. Sport s = new Skiing();
   System.out.println(s.usesSkis());
```

For each of these questions if the statement would compile indicate what value it would output. If

```
e. Sport s = new Skiing();
   SnowBallFight f = (SnowBallFight) s;
   System.out.println(f.usesSkis());
```

it would not compile, briefly explain why it would not compile.

Task 1:
Using interfaces, you can specify similar behaviors for possibly disparate classes. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like.
Many scientists blame these greenhouse gases for the phenomenon called global warming.

Create three small classes unrelated by inheritance—classes **Building, Car** and **Bicycle**.
Draw a UML diagram and give each class some unique appropriate attributes and behaviors that it does not have in common with other classes.

Write an interface **releaseCarbonFootprint** with a **getCarbonFootprint()** method. Have each of your classes implement that interface, so that its **getCarbonFootprint()** method calculates an appropriate carbon footprint for that class (surf the web and find out how to calculate different carbon footprints).
For Building: (Add Carbon footprints in the result of Natural Gas and Electricity consumption)

Due to Natural Gas Consumption :

Value of therms of natural gas the building consumes, multiply by a emission factor of 11.7

Due to Electricity Consumption:

Carbon foot print (in kg)**= [**consumption of energy] x emission factor **=**
Latest emission factor for electricity consumption is 0.82

For Car:

For Diesel: Fuel Consumption per Km in Litre x 2640 g/l / 100
For Petrol: Fuel Consumption per Km in Litre x 2392 g/l / 100
For LPG: Fuel Consumption per Km in Litre x 1665 g/l / 100 (per km)
For High Calorific CNG: Fuel Consumption per Km in Kg x 2252/100
For Low Calorific CNG: Fuel Consumption per Km in Kg x 2666/ 100

On average 271g of CO2 is released per KM

Bicycles do not require fuel in the same sense of cars and buses, so the ride does not release any more carbon emissions. Food intake, and the energy which it produces to help a cyclist propel their bike, is the final piece of a bike commute carbon footprint.

Cyclists on the average diet will add 16 g of CO2 per km ridden.

Write an application that creates objects of each of the three classes, places references to those objects in ArrayList, then iterates through the ArrayList, polymorphically invoking each object's **getCarbonFootprint()** method. For each object, print some identifying information and the object's carbon footprint.

Task 2:

Suppose the National Bank decides that every account type must have some monthly fees. Therefore, a `deductFees()` method should be added to the Account class:
public class Account { public void deductFees() { . . . } . . . }
But what should this method do? Of course, we could have the method do nothing. But then a programmer implementing a new subclass might simply forget to implement the deductFees method, and the new account would inherit the do-nothing method of the superclass. Also Bank does not open any generalize Account. So, class Account should not be instantiated. Implement a better way to do so: That forces the implementers of subclasses to specify concrete implementations of this method. (Of course, some subclasses might decide to implement a do-nothing method, but then that is their choice—not a silently inherited default.)

Design a class `Account` with `Account number, Account title and status (Active/ Closed)` and methods `closeAccount() and Display().`
Design it's subclasses `Current Account and Saving Account` with `balance` and methods `deductfee()`, Saving Account also has `profit`. For Current Account 500 Rs. are deducted from user account if the balance is greater than 15000. For Saving Account 1% of Profit is deducted from balance of the user. If the profit is greater than 3000.

Design a `Test class` that tests both types of account.

Task 3:

Design a `Student` class.
It holds data common to all students, (`id, name, yearOfAdmisison`) and methods to `calculateCGPA()` and `toString()` Display information, but does not hold all the data needed for students of specific majors. For example it can not calculate remaining credit hours for students. As the requirements might differ from major to major.

For example in CsStudent class we have required hours as follows.

```
MATH_HOURS = 20; // Math hours
CS_HOURS = 40;      // Comp sci hours
GEN_ED_HOURS = 60; //General hours
```

And hours earned information for every student.
```
mathHours, csHours, genEdHours;
```

Hours can not be negative or greater than the requirement.
Design a method to `calculateRemainingHours()`. Override `toString()` method to display information.

Design a test class to create an ArrayList of Cs Students and calculate remaining credit hours of those students.

2. You are asked to develop communication software such as an FTP (File Transfer Protocol) or Telnet program that uses a modem. Your program must work with a variety of different modems. Although all the modems provide the same functionality, their implementations are quite different.
Typically, there are open, close, read, and write operations that are invoked on a modem.
It would be inadvisable to create multiple versions of the application to interface with each of the modems available on the market because the code maintenance and application upgrades would be too much work.
Suggest and implement a solution in order to provide a uniform programming interface to all the modems so the application that uses a modem would not break even if the implementations in the methods of the modem change in future.

The two types of modems that use these common modem behaviours are **HuaweiModem** and **MindstickModem.**
Write an application that creates objects of **HuaweiModem** and **MindstickModem**, places references to those objects in ArrayList, then iterates through the ArrayList, polymorphically invoking each object's methods

3 'SmartKids! Pakistan' designs educational applications for children. They have hired you to design an application 'Animal Families' that teaches kids about different types of animals (based on animal families) and the difference in their behaviors etc. Basic hierarchy and description of the animals is given below.
Each animal has the following attributes:
nameOfFood, LivesIn

Every animal performs the following actions:
eat(), makeNoise(), sleep(), move()

a. Using information given above design some classes using the OOP principles to design the basic structure of application. Animal.java and Dog.java are given as a sample.
- *Your solution must inherit as much as possible from the super classes.*
- *Be sure to use abstract classes and abstract methods wherever appropriate*

b. Create a driver file in which
· Create an ArrayList named myAnimals of size ***n*** with Animal as declared type.
· The program should then take 'type of Animal' as input from the user ***n*** times and in each iteration, instantiate index of list with entered type.
· Call all the overridden methods for each index of the list.

4. You are required to design a class named Computer. Each computer has a **companyName, model, color, processorType** and **price**. A type of computer is DesktopComputer, that has **companyName, model, color, processorType, price** and **monitorDimensions**. Another type of computer is LaptopComputer that has **companyName, model, color, processorType, price, camera, size** and **weight**.

a. Identify abstract and concrete classes and design the above hierarchy using appropriate Object-Oriented Programming principles. Also use an abstract method to **displayData()**.

b. Create a driver file in which

· Create an ArrayList named computerList of size ***n*** with Computer as declared type.

· The program should then take 'type of Computer' as input from the user ***n*** times and in each iteration, instantiate index of list with entered type.

· Call the overridden method for each index of the list.