Lab 9

Interfaces

Objective:

Objective of this lab is to learn how to define Interface and implement Interfaces.
Students will also learn how to implement multiple interfaces in one class.

Activity Outcomes:

This lab teaches you the following topics:

- Interface.
- Method overriding with interfaces.
- Implementation of multiple interfaces in class.
- Use of interfaces with abstract classes

Instructor Note:

As pre-lab activity, read Chapter 13 from the text book "Introduction to Java Programming", Y. Daniel Liang, Pearson, 2019.

1) Useful Concepts

a. Interface

An interface is a type that groups together a number of different classes that all include method definitions for a common set of method headings.
Java interface specifies a set of methods that any class that implements the interface must have. An interface is itself a type, which allows you to define methods with parameters of an interface type and then have the code apply to all classes that implement the interface. One way to view an interface is as an extreme form of an abstract class. However, as you will see, an interface allows you to do more than an abstract class allows you to do. Interfaces are Java's way of approximating multiple inheritance. You cannot have multiple base classes in Java, but interfaces allow you to approximate the power of multiple base classes.

Defining interface :

```
public interface Ordered {
public boolean precedes(Object other);

}
```

Implement interface and its method:

```
public class Class_name implements Interface_Name
{
public boolean precedes(Object other){ }
}

public class implements SomeInterface, AnotherInterface{}
```

b. Defining Constants in Interfaces

The designers of Java often used the interface mechanism to take care of a number of miscellaneous details that do not really fit the spirit of what an interface is supposed to be. One example of this is the use of an interface as a way to name a group of defined constants. An interface can contain defined constants as well as method headings, or instead of method headings. When a method implements the interface, it automatically gets the defined constants. For example, the following interface defines constants for months:

```
public interface MonthNumbers {

        public static final int JANUARY = 1, FEBRUARY = 2,MARCH =
        3, APRIL = 4, MAY = 5,JUNE = 6, JULY = 7, AUGUST = 8,
        SEPTEMBER = 9, OCTOBER = 10, NOVEMBER = 11,
        DECEMBER = 12; }
```

Any class that implements the MonthNumbers interface will automatically have the 2 constants defined in the MonthNumbers interface. For example, consider the following toy class:

```
public class DemoMonthNumbers implements
MonthNumbers
{ public static void main(String[] args)          {
            System.out.println( "The number for January is " + JANUARY);          } }
```

c. The Comparable Interface

The Comparable interface is in the java.lang package and so is automatically available to your program. The Comparable interface has only the following method heading that must be given a definition for a class to implement the Comparable interface:

```
public int compareTo(Object other);
```

The method compareTo should return a negative number if the calling object "comes before" the parameter other, a zero if the calling object "equals" the parameter other, and a positive number if the calling object "comes after" the parameter other. The "comes before" ordering that underlies compareTo should be a total ordering. Most normal ordering, such as less-than ordering on numbers and lexicographic ordering on strings, is total ordering.

2) Solved Lab Activities (Allocated Time 50 Min.)

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	*15 mins*	*Medium*	*CLO-4*
Activity 2	*25 mins*	*Medium*	*CLO-4*

Activity 1:

Declare a Interface, RegisterForExams that contains single method register, implements the interface in two different classes (a) Student (b) Employee. Write down a Test Application to test the overridden method.

Solution:

```
public interface RegisterForExams {
public void register();
}
--------------------------------------------------------------------
public class EmplayeeTask implements RegisterForExams{

private String name; private String date; private int salary;

public EmplayeeTask()
{
name = null; date = null; salary = 0;
}

public EmplayeeTask(String name,String date,int salary)
{
this.name = name; this.date = date; this.salary = salary;
}

@Override
public void register() {
System.out.println("Employee is registered " + "Name " + name + "
salary " + salary + " date " + date);
}
}
------------------------------------------------------------------
public class StudentTask implements RegisterForExams{
private String name; private int age; private double gpa;

public StudentTask()
{

name = null; age = 0;
gpa = 0;
}
public StudentTask(String name,int age,double gpa)
{
this.name = name;
this.age = age;

this.gpa = gpa;

}
@Override
public void register() {
System.out.println("Student is registered  " + "Student name " + name
+ " gpa " + gpa);
}}
--------------------------------------------------------------------
------
public class Runner {
public static void main(String[] args) {

EmplayeeTask e = new EmplayeeTask("Ahmed","11,02,2001",20000);
StudentTask s = new StudentTask("Ali",22,3.5);
e.register();
s.register();
}  }
```

Output

```
Employee is registered Name Ahmed salary 20000 date 11,02,2001
Student is registered Student name Ali gpa 3.5
```

Activity 2:

An Example that shows How to create your own interface and implement it in abstract class

```
interface I1 {

void methodI1(); // public static by default
}
----------------------------------------------------------------------
interface I2 extends I1 {

void methodI2(); // public static by default
}
----------------------------------------------------------------------
class A1 {
public String methodA1() {
String strA1 = "I am in methodC1 of class A1"; return strA1;
}
public String toString() {
return "toString() method of class A1";
}
}
----------------------------------------------------------------------
class B1 extends A1 implements I2 {

public void methodI1() {
System.out.println("I am in methodI1 of class B1");
}
public void methodI2() {
System.out.println("I am in methodI2 of class B1");
}
}
----------------------------------------------------------------------
class C1 implements I2 {

public void methodI1() {
System.out.println("I am in methodI1 of class C1");
}
public void methodI2() {
```

```
System.out.println("I am in methodI2 of class C1");
}
}

// Note that the class is declared as abstract as it does not
// satisfy the interface contract

abstract class D1 implements I2 {

public void methodI1() {
}
// This class does not implement methodI2() hence declared abstract.
}
----------------------------------------------------------------------
public class InterFaceEx {
public static void main(String[] args) {

I1 i1 = new B1();
i1.methodI1(); // OK as methodI1 is present in B1
// i1.methodI2(); Compilation error as methodI2 not present in I1
// Casting to convert the type of the reference from type I1 to type
I2 ((I2) i1).methodI2();
I2 i2 = new B1();
i2.methodI1(); // OK
i2.methodI2(); // OK
// Does not Compile as methodA1() not present in interface reference
I1
// String var = i1.methodA1();
// Hence I1 requires a cast to invoke methodA1
String var2 = ((A1) i1).methodA1(); System.out.println("var2 : " +
var2);
String var3 = ((B1) i1).methodA1(); System.out.println("var3 : " +
var3);

String var4 = i1.toString();
System.out.println("var4 : " + var4);
String var5 = i2.toString();
```

```
System.out.println("var5 : " + var5);
I1 i3 = new C1();
String var6 = i3.toString();
System.out.println("var6 : " + var6); // It prints the Object
toString() method
Object o1 = new B1();
// o1.methodI1(); does not compile as Object class does not define
// methodI1()
// To solve the probelm we need to downcast o1 reference. We can do
it
// in the following 4 ways
((I1) o1).methodI1(); // 1
((I2) o1).methodI1(); // 2
((B1) o1).methodI1(); // 3
/*
*
*     B1 does not have any relationship with C1 except they are
"siblings".
*
*     Well, you can't cast siblings into one another.
*
*/
// ((C1)o1).methodI1(); Produces a ClassCastException
}
}
```

Output

```
I am in methodI1 of class B1
I am in methodI1 of class B1
I am in methodI2 of class B1
var2 : I am in methodC1 of class A1
var3 : I am in methodC1 of class A1
var4 : toString() method of class A1
```

```
var5 : toString() method of class A1
var6 : javaapplication12.C1@15db9742
I am in methodI1 of class B1
I am in methodI1 of class B1
I am in methodI1 of class B1
```

Graded Lab Tasks (Allocated Time 2 Hr.)

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

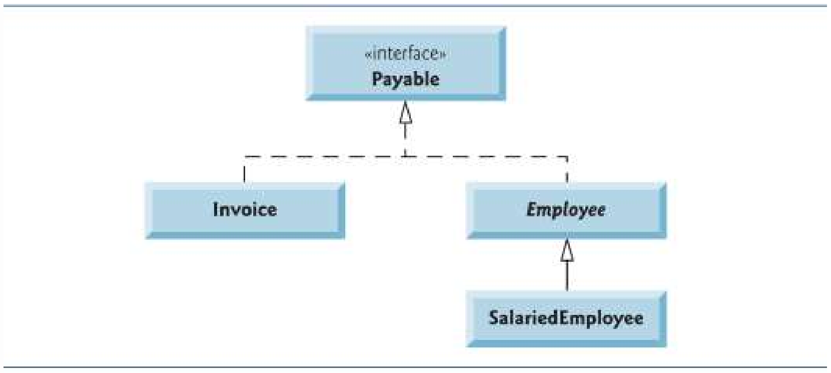
Lab Task 1

```
Public interface Shape
{
                double getArea();
}
```

Create two classes Circle and Rectangle. Both must implement the interface Shape.
Note: You can assume appropriate data members for circle and rectangle

Lab Task 2

Implement the following hierarchy



Payable:

Double getPaymenyAmount();

Invoice:

```
private String partNumber;
private String partDescription;
private int quantity;
private double pricePerItem;
```

Employee:

```
private String firstName;
private String lastName;
private String socialSecurityNumber;
```

Salaried Employee:

```
private double weeklySalary;
```

In the runner , call the getPaymentAmount() method polymorphically.

Lab Task 3

Below is the skeleton for a class named "InventoryItem" . Each inventory item has a name and a unique ID number:

```
class InventoryItem
{
private String name;
private int uniqueItemID;
}
```

Flesh out the class with appropriate accessors, constructors, and mutatators. This class will implement the following interface:

```
Public interface compare
{
        boolean compareObjects(Object o);
}
```

Lab Task 4

Below is a code skeleton for an interface called "Enumeration" and a class called "NameCollection ". Enumeration provides an interface to sequentially iterate through some type of collection. In this case, the collection will be the class NameCollection that simply stores a collection of names using an array of strings.

```
interface Enumeration
{
// return true if a value exists in the next index
public boolean hasNext(int index);

// returns the next element in the collection as an Object
public Object getNext(int index);

}
//NameCollection implements a collection of names using  a simple array.
 class NameCollection
{
  String[] names = new String[100];
 }
```

Create constructor and abstract methods of interface in the class NameCollection.
Then write a main method that creates a NamesCollection object with a sample array of strings, and then iterates through the enumeration outputting each name using the getNext() method.