|  |  |
| --- | --- |
|  | **CSC-150 – Object Oriented Programming** |
| **Semester II (Fall 2021)**  **Course Instructor(s): Dr. Sher Muhammad, Sana Fatima** |

|  |
| --- |
| **Lab 07**  **Packages , Abstract classes and Interfaces in Java** |

**Objective(s):**

1. Packages
2. Abstract Class
3. Interfaces

|  |
| --- |
| **1: What is Package** |

A **package** is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer.

*You might keep HTML pages in one folder, images in another, and scripts or applications in yet another*.

Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

**Advantages of using Java Packages:**

1. **Better organization**: Java package is used to categorize the classes and interfaces so that they can be easily maintained. As in large java projects where we have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better and when you need something you can quickly locate it and use it, which improves the efficiency.
2. **Reusability**: While developing a project in java, we often feel that there are few things that we are writing again and again in our code. Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.
3. **Name conflicts**: Java package removes naming collision. For example; We can define two classes with the same name in different packages so to avoid name collision, we can use packages



**Types of packages in Java**

* *User defined package*: The package we create is called user-defined package.
* *Built-in package*: The already defined package like java.io.\*, java.lang.\* etc are known as built-in packages.

**Example**

A class Calculator is created inside a package name letmecalculate. To create a class inside a package, declare the package name in the first statement in your program.

A class can have only one package declaration.  
Calculator.java file created inside a package letmecalculate

package letmecalculate;

public class Calculator {

public int add(int a, int b){

return a+b;

}

}// save this file as Calculator.java. This class is used to create a package as well as to perform addition

Now lets see how to use this package in another program.

import letmecalculate.Calculator;

public class Demo{

public static void main(String args[]){

Calculator obj = new Calculator();

System.out.println(obj.add(100, 200));

}

}// save this file as Demo.java

**Steps for compiling and running java package**

1. Access path of the folder/directory where you have saved the Calculator file

For example : C:\Users\HP\Desktop\OOP, it suggests that Calculator file is on desktop in OOP folder

1. javac -d . Calculator.java \\ compile the package
2. javac Demo.java
3. java Demo

**Access package from another package**

There are three ways to access the package from outside the package.

1. Import package.classname;
2. Fully qualified name.
3. Import package.\*;
4. **Using package.classname**

If you import package.classname then only declared class of this package will be accessible.

**Example:**

import letmecalculate.Calculator;

1. **Using fully qualified name**

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

**It is generally used when two packages have same class name** e.g. java.util and java.sql packages contain Date class.

**Example:**

public class Demo{

public static void main(String args[]){

letmecalculate.Calculator obj = new letmecalculate.Calculator();

System.out.println(obj.add(100, 200));

}

}

1. **Using package.\***

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package

**Example:**

Import java.util.\*;

**Abstraction in Java**

Abstraction is a concept of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

There are two ways to achieve abstraction in java

1. Abstract class
2. Interface

|  |
| --- |
| **2: Abstract Class** |

In some situations, we don't want programmers to instantiate (create an instance of) a particular class, but rather we choose to create the class only so that it can serve as a generic or default superclass of other classes.

*For example*, we might want to have a class Shape to serve as superclass for a number of "real" subclasses, like Parallelogram or Rhombus: we want to have Parallelogram and Rhombus objects around, but not generic "Shape" objects.

An abstract class is a class that is declared ‘abstract’. It can, but does not have to, include abstract methods. Abstract classes cannot be instantiated but they can be subclassed using the extends keyword. An abstract method is a method that is declared without an implementation and it requires the abstract keyword.

If a class includes abstract methods, then the class itself must be declared as abstract.

When a child class extends an abstract class, it must either:

• Provide implementations for all abstract methods from its parent class, or

• Also be abstract

• Child classes can reference the constructor of abstract class by using super()

|  |
| --- |
| **abstract** **class** Shape{  **abstract** **void** draw();  }  //In real scenario, implementation is provided by others i.e. unknown by end user  **class** Rectangle **extends** Shape{  **void** draw(){System.out.println("drawing rectangle");}  }  **class** Circle1 **extends** Shape{  **void** draw(){System.out.println("drawing circle");}  }  //In real scenario, method is called by programmer or user  **class** TestAbstraction1{  **public** **static** **void** main(String args[]){  Shape s=**new** Circle1();  //In a real scenario, object is provided through method, e.g., getShape() method  s.draw();  }  } |

|  |
| --- |
| **3: Interfaces** |

An interface in Java is a blueprint of a class. The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also represents the **IS-A** relationship. It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default** and **static methods** in an interface.

Since Java 9, we can have **private** **methods** in an interface.

**Why interfaces?**

These are reasons to use interface:

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.

**Declaration of an Interface**

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

**Syntax:**

|  |
| --- |
| interface <interface\_name>{  // declare constant fields  // declare methods that abstract  // by default.  } |

Interface fields are public, static and final by default, and the methods are public and abstract.



**The relationship between classes and interfaces**

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.

**Example:**

Let’s consider the example of vehicles like bicycle, car, bike………, they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

|  |
| --- |
| import java.io.\*;  interface Vehicle {      // all are the abstract methods.      void changeGear(int a);      void speedUp(int a);      void applyBrakes(int a);  }    class Bicycle implements Vehicle{      int speed;      int gear;         // to change gear        public void changeGear(int newGear){            gear = newGear;      }        // to increase speed        public void speedUp(int increment){            speed = speed + increment;      }        // to decrease speed      public void applyBrakes(int decrement){            speed = speed - decrement;      }        public void printStates() {           System.out.println("speed: " + speed                + " gear: " + gear);      }  }    class Bike implements Vehicle {        int speed;      int gear;        // to change gear        public void changeGear(int newGear){            gear = newGear;      }        // to increase speed        public void speedUp(int increment){            speed = speed + increment;      }        // to decrease speed      public void applyBrakes(int decrement){            speed = speed - decrement;      }        public void printStates() {           System.out.println("speed: " + speed               + " gear: " + gear);      }    }  class GFG {        public static void main (String[] args) {          // creating an instance of Bicycle          // doing some operations          Bicycle bicycle = new Bicycle();          bicycle.changeGear(2);          bicycle.speedUp(3);          bicycle.applyBrakes(1);          System.out.println("Bicycle present state :");          bicycle.printStates();          // creating instance of the bike.          Bike bike = new Bike();          bike.changeGear(1);          bike.speedUp(4);          bike.applyBrakes(3);          System.out.println("Bike present state :");          bike.printStates();      }  } |

**Multiple inheritance in Java by interface**

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

|  |
| --- |
| //you can now inherit features of both interfaces  Class Test implements Interface1, Interface 2 |

**Default Method in Interface**

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

|  |
| --- |
| interface Drawable{  void draw();  default void msg(){System.out.println("default method");}  }  class Rectangle implements Drawable{  public void draw(){System.out.println("drawing rectangle");}  }  class TestInterfaceDefault{  public static void main(String args[]){  Drawable d=new Rectangle();  d.draw();  d.msg();  }} |

**Static Method in Interface**

|  |
| --- |
| interface Drawable{  void draw();  static int cube(int x){return x\*x\*x;}  }  class Rectangle implements Drawable{  public void draw(){System.out.println("drawing rectangle");}  }  class TestInterfaceStatic{  public static void main(String args[]){  Drawable d=new Rectangle();  d.draw();  System.out.println(Drawable.cube(3));  }  } |

**Nested Interface**

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

* Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
* Nested interfaces are declared static implicitly.

Nested interface which is **declared within the interface**

|  |
| --- |
| interface interface\_name{  ...  interface nested\_interface\_name{  ...  }  } |

Nested interface which is **declared within the class**

|  |
| --- |
| class class\_name{  ...  interface nested\_interface\_name{  ...  }  } |

Example of nested interface which is declared within the interface

|  |
| --- |
| interface OuterInterface {  void anymethod();  interface InnerInterface {    void print();  }  }    class NestedInterface implements OuterInterface.InnerInterface {  public void print() {  System.out.println("Print method of nested interface");  }  public static void main(String args []) {  NestedInterface obj = new NestedInterface();  obj.print();    }  } |

Example of nested interface which is declared within the class

|  |
| --- |
| class OuterClass {  interface InnerInterface {  int id = 20;  void print();  }  }    class NestedInterfaceDemo implements OuterClass.InnerInterface {  public void print() {  System.out.println("Print method of nested interface");  }  public static void main(String args []) {  NestedInterfaceDemo obj = new NestedInterfaceDemo();  obj.print();  System.out.println(obj.id);  // Assigning the object into nested interface type  OuterClass. InnerInterface obj2 = new NestedInterfaceDemo();  obj2.print();  }  } |

**Class inside the interface**

If we define a class inside the interface, java compiler creates a static nested class.

**Syntax:**

|  |
| --- |
| interface M{  class A{}  } |

**Lab Tasks: 10 marks**

Exercise 1 (Packages) Your creativity

1. Create three packages, think of them yourself
2. Put two different classes in each package
3. Import all three of these packages in class named PackagePractice, you will have access to 6 classes
4. Call methods of these 6 classes and use them in PackagePractice

Exercise 2 (Interfaces)

1. What is wrong with the following interface?

public interface SomethingIsWrong {

void aMethod(int aValue){

System.out.println("Hi Mom");

}

}

1. Fix the interface in question 1.
2. Is the following interface valid?

public interface Marker {

}

Exercise 3 (Interfaces)

1. Create the Animal interface.
2. Declare abstract method legs.
3. Declare an abstract method eat.
4. Create the Spider, Caterpillar and Cat class that implements animal interface.
5. All classes implement the Animal interface.
6. Implement the eat and legs method.

Exercise 4 (Abstract class)

We have to calculate the percentage of marks obtained in three subjects (each out of 100) by student A and in four subjects (each out of 100) by student B. Create an abstract class 'Marks' with an abstract method 'getPercentage'. It is inherited by two other classes 'A' and 'B' each having a method with the same name which returns the percentage of the students. The constructor of student A takes the marks in three subjects as its parameters and the marks in four subjects as its parameters for student B. Create an object for each of the two classes and print the percentage of marks for both the students.

**END**