Lab 07: Packages & Interfaces in JAVA

Objective(s):

1. Packages
2. Interfaces

1: Packages

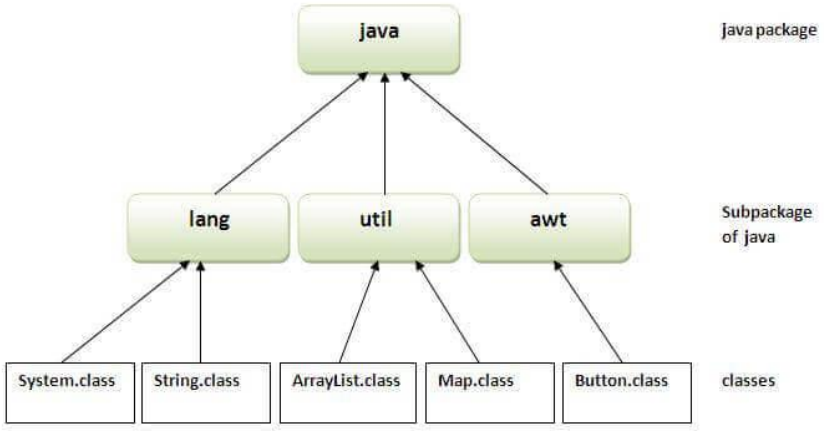
A java package is a group of similar types of classes, interfaces and sub-packages.

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.

Benefits of Java Package

1. Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2. Java package provides access protection.
3. Java package removes naming collision.



Example

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

Compile JAVA Package

If you are not using any IDE, you need to follow the syntax given below:

```
javac -d directory javafilename
```

Ex: `javac -d . Simple.java`

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

Run JAVA Package Program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: `javac -d . Simple.java`

To Run: `java mypack.Simple`

Access package from another package

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

```
1. import package.*;
2. import package.classname;
3. fully qualified name.
```

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:

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

2. **Using package.classname**

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

   Example:

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

3. **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.

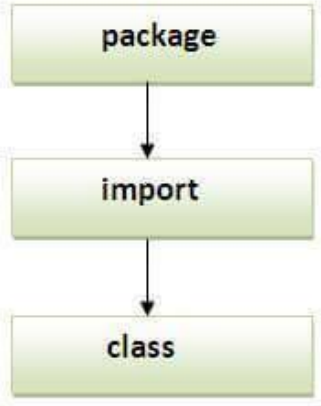
It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example:

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Sub Package in JAVA

Package inside the package is called the subpackage. It should be created to categorize the package further.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

```
package com.javatpoint.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

To Compile: `javac -d . Simple.java`

To Run: `java com.javatpoint.core.Simple`

Managing Source & Class Files

2: Interfaces

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

An interface in Java is a blueprint of a class. It has static constants and abstract methods.

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?

There are mainly three reasons to use interface:

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

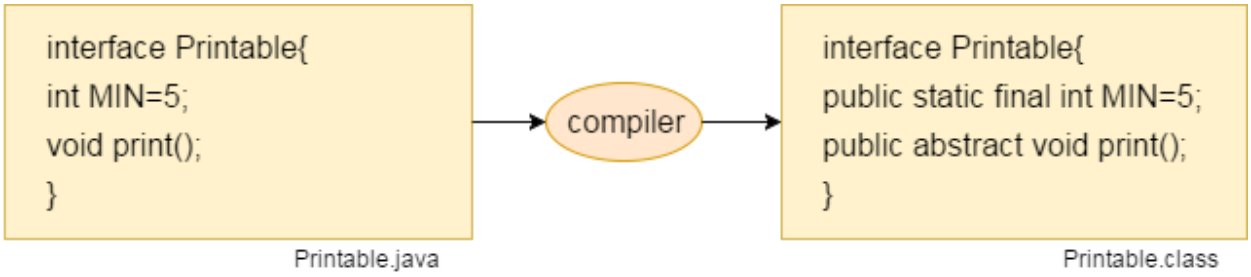
Declaration of an Interface

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:

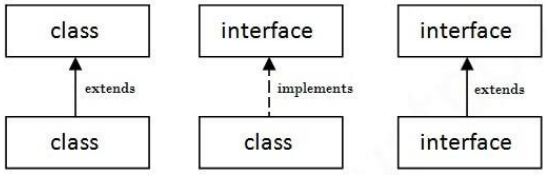
```
interface <interface_name>{

    // 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:

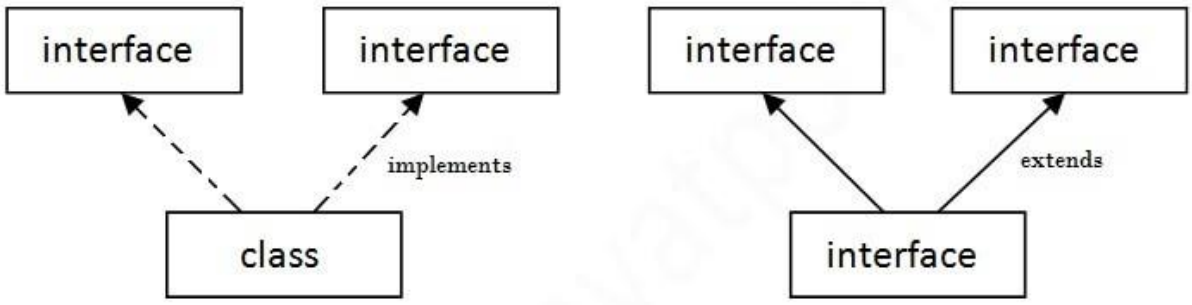
Let's consider the example of vehicles like bicycle, car, bike……, they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

```
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
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    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
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    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.



Multiple Inheritance in Java

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.

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 Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
   public void msg(){System.out.println("Hello nested interface");}

   public static void main(String args[]){
        Showable.Message message=new TestNestedInterface1();//upcasting here
        message.msg();
 }
}
```

Example of nested interface which is declared within the class

```
interface Showable{
  class A{
    interface Message{
    void msg();
  }
}
```

```
class TestNestedInterface2 implements A.Message{
  public void msg(){System.out.println("Hello nested interface");
}

  public static void main(String args[]){
     A.Message message=new TestNestedInterface2();//upcasting here
     message.msg();
 }
}
```

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:

Exercise 1 (Packages)

Assume you have written some classes. Belatedly, you decide they should be split into three packages, as listed in the following table. Furthermore, assume the classes are currently in the default package (they have no package statements).

Destination Packages	
Package Name	**Class Name**
mygame.server	Server
mygame.shared	Utilities
mygame.client	Client

4. Which line of code will you need to add to each source file to put each class in the right package?

5. To adhere to the directory structure, you will need to create some subdirectories in the development directory and put source files in the correct subdirectories. What subdirectories must you create? Which subdirectory does each source file go in?
6. Do you think you'll need to make any other changes to the source files to make them compile correctly? If so, what?

Exercises

Download the source files as listed here.

Client

Server

Utilities

1. Implement the changes you proposed in questions 1 through 3 using the source files you just downloaded.
2. Compile the revised source files. (Hint: If you're invoking the compiler from the command line (as opposed to using a builder), invoke the compiler from the directory that contains the mygame directory you just created.)

Exercise 2 (Interfaces)

1. What is wrong with the following interface?

```
public interface SomethingIsWrong {
    void aMethod(int aValue){
        System.out.println("Hi Mom");
    }
}
```

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

```
public interface Marker {
}
```

Exercises

1. Write a class that implements the CharSequence interface found in the java.lang package. Your implementation should return the string backwards. Select one of your sentence to use as the data. Write a small main method to test your class; make sure to call all four methods. Note: if you don't know the implementation of a particular method when you implement CharSequence interface then you add your own what you understand.

Following functions' implementations should be in your written class.

```
int fromEnd(int i);

char charAt(int i);
int length();
CharSequence subSequence(int start, int end);
String toString()
static int random(int max);
```

2. Suppose you have written a time server that periodically notifies its clients of the current date and time. Write an interface the server could use to enforce a particular protocol on its clients.

   Following methods should be in your interface:

```
setTime
setDate
setDateAndTime
getLocalDateTime
```

Exercise 3 (Interfaces: Queue)

A queue is an abstract data type for adding and removing elements. The first element added to a queue is the first element that is removed (first-in-first-out, FIFO). Queues can be used, for instance, to manage processes of an operating system: the first process added to the waiting queue is reactivated prior to all other processes (with the same priority).

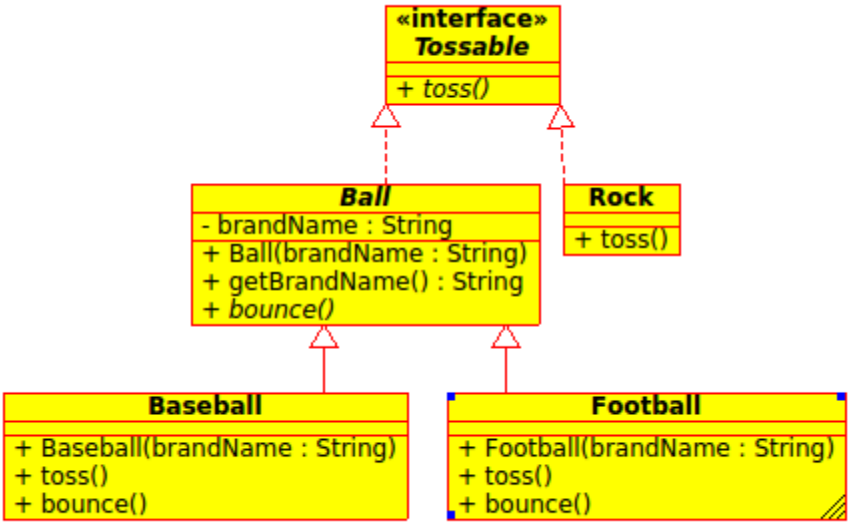
Design an interface Queue, with methods to add and remove elements (integers). Furthermore, a method to check whether the queue is empty or not should exist.

Implement the queue with an array. If the array becomes to small to hold all added elements, create a new larger (double the size of instance) array and copy all elements of the small array to the new one.

Thoroughly test your implementation with small and large queues.

Exercise 4 (Interfaces)

Implement the following class hierarchy. You do not need to fill in the method bodies with actual code for the **toss** or **bounce** methods. You can use string message or whatever suitable you want.



Exercise 5 (Interfaces)

Declare the Rectangle, SportCar, Manager classes as classes that implement the Printable interface and run the given application. The output of this application should be the details of each one of the objects that were instantiated.

```
interface Printable
{
    public void print();
}
public class PrintableDemo
{
     public static void main(String args[])
     {  Printable vec[] =
              {new Rectangle(110,80), new SportCar("Toyota", 989621),
               new Rectangle(34,32), new Manager("John", 40),
               new Rectangle(54,10), new SportCar("Audi", 2365644),
               new SportCar("Mazda", 4322343), new Manager("Joji", 22)};
       for(int index=0; index<vec.length; index++)
        {
              vec[index].print();
        }
      }
```

}**END**