Unit -III

PACKAGES, INTERFACES

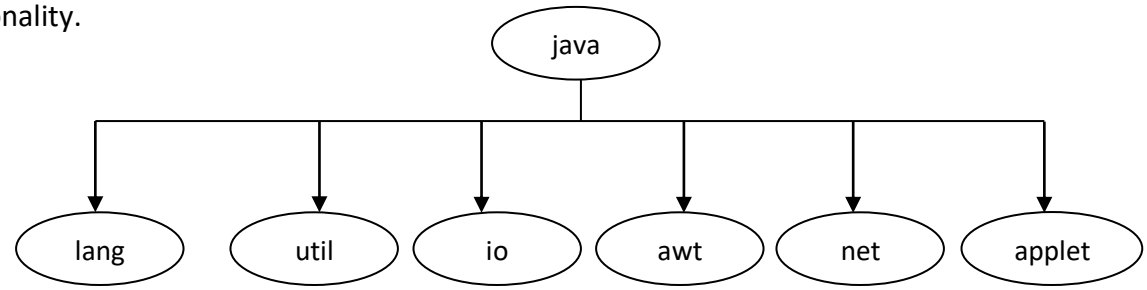
3.5 Define package:

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

Package in java can be categorized in two forms,

1. Built-in package and
2. User-defined package.

1. Java Built-in package (or) Java API packages (or) Java system packages: Java API [Application Programming Interface] provides a large no. of classes grouped into different packages according to functionality.



Frequently used java built-in packages are

Package name	Contents
java.lang	Language support classes. These are classes that java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.
java.util	Language utility classes such as vectors, Hash tables, random numbers , date, etc.
java.io	Input / output support classes. They provide facilities for the input and output of data.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	Classes for creating and implementing applets.

2. User defined packages: we can define our own packages. These user defined packages are created as follows:

We must first declare the name of the package using the 'package' keyword followed by a package name then we define a class as public, just as we normally define a class.

```
package   packagename;    // package declaration
public class classname        // class definition
{
----------
}
```

- Here the package name is name of the package.
- The class name is the name of the class.

CREATING PACKAGES: Creating our own package involves the following steps:

1. Declare the package at the beginning of file using the form

   package packagename;

2. Define the class that is to be put in the package and declare it **public.**
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the **.java** file in the subdirectory created.
5. Compile the .java file and this creates **.class** file in the subdirectory.

Advantage 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**.

3.6 Describe the concept of class path:

CLASS PATH: the CLASS PATH is an **environmental variable that tells the java compiler where to look for class files to import.** CLASS PATH is generally set a directory.

To set the CLASS PATH, the procedure is:

1. Right click **on my computer**
2. Select **system properties**. The system properties dialog box will appears.
3. In this **select advanced tab & then click on environmental variables** button
4. Go to **user variables & click on New button**
5. Type the following:

   Variable name: path

   Variable value: c:\program files\java\jdk1.5.0\bin;
6. Then **click on ok button**
7. Then click on ok button in the environmental variables window and system properties window

   C:\>**set path**=" c:\program files\java\jdk1.5.0\bin";

   C:\>**set classpath**=" c:\program files\java\jdk1.5.0\bin";

After this, class files will be available anywhere in that computer system. Our program which uses **the package may present in any directory, it can be compiled and run without any problem**

3.7 Describe the concept of access protection/access modifiers:

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specify accessibility **(scope) of a data member, method, constructor or class.**

There are 4 types of java access modifiers:

1. private 2. default 3. protected 4. public

There are many **non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.** Here, we will learn access modifiers.

Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

Modifier	**With in class**	**Within package**	**Outside package by sub class only**	**Outside package**
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

1) Private access modifier

The private access modifier is **accessible only within class.**

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
//save by Simple.java
class A
{
     private int data=40;
     private void msg()
     {
        System.out.println("Hello java");
     }
}
public class Simple
  {
     public static void main(String args[])
      {
            A obj=new A();
            System.out.println(obj.data);//Compile Time Error
            obj.msg();//Compile Time Error
      }
  }
```

2) Default access modifier

If you don't use any modifier, it is treated as **default** by default. The default modifier is **accessible only within package.**

Example of default access modifier

In this example, we have created two packages p1. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package p1;
class A{
  void msg()
 {
        System.out.println("Hello");
  }
 }
//save by B.java
import p1.*;
class B {
  public static void main(String args[]) {
    A obj = new A();//Compile Time Error
    obj.msg();//Compile Time Error
  }
 }
```

In the above example, **the scope of class A and its method msg() is default so it cannot be accessed from outside the package.**

3) Protected access modifier

The **protected access modifier** is accessible within package and **outside the package but through inheritance only.**

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example of protected access modifier

In this example, we have created the two packages p1. The A class of p1 package is public, so can be accessed from outside the package. But **msg method of this package is declared as protected**, so it can be accessed from outside the **class only through inheritance**.

```
//save by A.java
package p1;
public class A
{
  protected void msg()
  {
     System.out.println("Hello");
   }
 }
//save by B.java
import p1.A;
class B extends A
{
  public static void main(String args[])
{
   B obj = new B();
   obj.msg();
  }
}
```

Output: Hello

4) Public access modifier

The **public access modifier** is accessible everywhere. **It has the widest scope among all other modifiers.**

Example of public access modifier

```
//save by A.java
  package p1;
public class A {
        public void msg()
         {
              System.out.println("Hello");
          }
}
//save by B.java
import p1.*;
class B {
   public static void main(String args[]){
         A obj = new A();
         obj.msg();
     }
}
```

3.8 Using classes from another package/class:

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

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

Example of package that import the packagename.*

```
//save by A.java in p1 package
package p1;
public class A
{
    public void msg()
    {
        System.out.println("Hello");
    }
}
//save by B.java in p1 package
package p1;
public class B
{
    public void display()
    {
        System.out.println("HOW R U");
    }
}
//save by C.java
import p1.*;
class C {
    public static void main(String args[])
    {
        A obj = new A();
        B obj1=new B();
        obj.msg();
        obj1.display();
    }
}
```

Output: Hello
HOW R U

3.9 Appreciate the concept of importing packages.

Importing packages (Accessing packages):

The import statement can be used to search a list of packages for a particular class.

The general form of import statement for searching a class is as follows:

Syntax: import package1[.package2][.package3].classname;

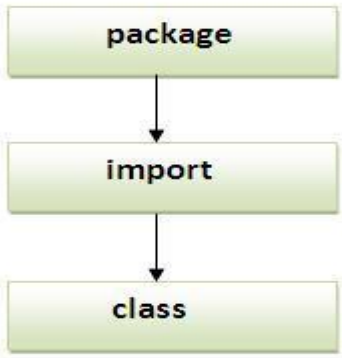
In the above syntax package1 is the name of the top level package, package2 is the name of the package that is inside the package1, and so on. We can have any number of packages in a package hierarchy. Finally the class name is specified.

Rules for importing packages:

1. The import statement must end with a semicolon.
2. The import statement should appear before any class definition in a sourced file.
3. Multiple import statements are allowed.

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 packagename.*

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

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

Example of package that import the packagename.*

```
//save by A.java in p1 package
package p1;
public class A
{
    public void msg()
    {
         System.out.println("Hello");
     }
}
//save by B.java in p1 package
package p1;
public class B
{
    public void display()
    {
         System.out.println("HOW ARE YOU");
     }
}
//save by C.java
import p1.*;
class C {
     public static void main(String args[]) {
          A obj = new A();
          B obj1=new B();
           obj.msg();
           obj1.display();
    }
}
```

Output: Hello

HOW ARE YOU

2) Using packagename.classname

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

Example of package by import package.classname

```
//save by A.java
package p1;
public class A
{
  public void msg()
   {
      System.out.println("Hello");
   }
}
//save by C.java
import p1.A;
class C
{
   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 of package by import fully qualified name

```
//save by A.java  in p1 package
package p1;
public class A
{
   public void msg()
    {
        System.out.println("Hello");
     }
}
//save by A.java  in p2 package
package p2;
public class A  {
   public void show()
    {
        System.out.println("HOW ARE U");
     }
}
```

```
//save by C.java
class C
{
    public static void main(String args[])
     {
          p1.A obj = new p1.A();//using fully qualified name
          p2.A obj1=new p2.A();//using fully qualified name
          obj.msg();
          obj1.show();
     }
}
```

Output: Hello
HOW ARE YOU

To Compile p1 class: javac -d . A.java

To compile current package class: javac C.java

To run program: java C

3.10 Explain concept of interfaces:

3.11 Define interface:

- An **interface in java** is a blueprint of a class. **It has static constants and abstract methods only.**
- The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body.
- It is used to **achieve fully abstraction and multiple inheritances in Java.**
- Java Interface also **represents IS-A relationship**.
- It cannot be instantiated just like abstract class.

Interface Syntax:

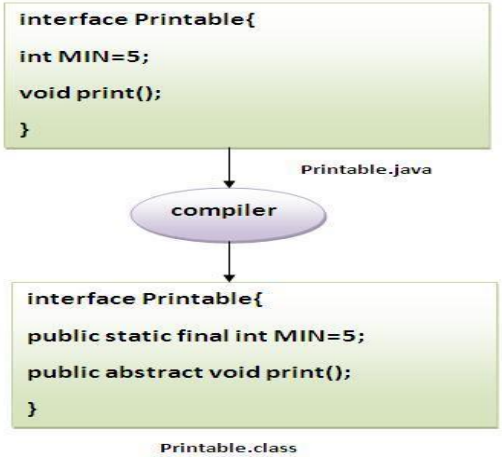
```
Access-specifier interface <interface-name>
{
     Declaration of variables;
     Declaration of methods;// no definition
}
```

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

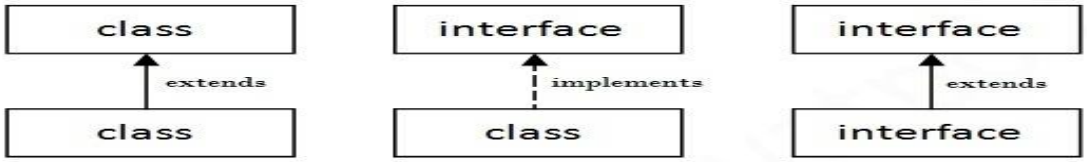
- It is used to **achieve fully abstraction**.
- By interface, we can support the **functionality of multiple inheritances**.
- It can be used to **achieve loose coupling.**

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



Understanding 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**.



Simple example of Java interface

In this example, Printable interface have only one method, its implementation is provided in the A class.

```
    interface printable
    {
        void print();
    }
class A implements printable
{
    public void print1()
    {
    System.out.println("Hello");
    }
}
Class B {
public static void main(String args[]) {
   A obj = new A();
   obj.print1();
  }
}
```

Output : Hello

3.12 Write the Difference between class and interface:

Property	Class	Interface
1.instantiation	1.can be instantiated	1.Cannot be instantiated
2.state	2.each object created will have its own state	2.each object created after implementing will have the same state
3.inheritance	3.a class can inherit only one class and can implement many interfaces	3.an interface cannot inherit any classes while it can extend many interfaces
4.variables	4.all variables are instance by a default unless otherwise specified	4.all the variables are the static, final by default and a value need to be assign at the time of definition
5.method	5.all the methods should be a having a definition unless decorated with an abstract keyword	5.all the methods are abstract by default and they will not have a definition

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods**. Abstract class and interface both can't be instantiated.**

3.13 HOW TO IMPLEMENT INTERFACES:

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the "**implements"** clause in a class definition & then create the methods defined by the interface.

The general form of a class to implement an interface:

class classname implements interface1, interface2……

{
// class body
}

e.g: class Use implements printable

```
{
   public void print1()
   {
     System.out.println("hello");
   }
}
```

Multiple Inheritance using Interfaces: In multiple inheritance, sub classes are derived from multiple super classes. By using multiple interfaces we can achieve multiple inheritances. Let us take 2- interfaces as:

```
    interface Printable
    {
          int a=10;
         void print1();
    }
    interface Showable
    {
           int b=20;
          void show();
    }
    class A7 implements Printable,Showable
    {
      public void print1()
      {
            System.out.println("printable a="+a);
      }
     public void show()
     {
               System.out.println("showable b="+b);
      }
    }
class Multiple
    {
        public static void main(String args[])
         {
              A7 obj = new A7();
```

```
            obj.print1();
            obj.show();
        }
    }
```

Extending interfaces: One interface can inherit another by use of the **keyword extends**. The syntax is the same as for inheriting classes.

When a class implements an interface that inherits another interface, it must provide implementation of all methods defined within the interface inheritance.

Note: Any class that implements an interface must implement all methods defined by that interface, including any that inherited from other interfaces.

```
interface f1
{
      void disp1();
}
interface f2 extends f1
{
     void disp2();
}
class A implements f2
{
      public void disp1()
      {
            System.out.println("This is display of i1");
      }
      public void disp2()
      {
           System.out.println("This is display of i2");
      }
}
public class Ext_iface
{
       public static void main(String args[])
            {
                  A obj = new A();
                  obj.disp1();
                  obj.disp2();
             }
}
```

Output: This is display of i1
This is display of i2

3.14 EXPLAIN THE SCOPE OF VARIABLES IN INTERFACES

- Interfaces can be **used to declare a set of shared constants that can be used in different classes**.
- This is similar to creating header files in C/C++ to contain a large number of constants.
- The constant values will be **available to any class that implements the interfaces.**
- Once declare any variable inside the interface these variables use inside package and outside package also because these members are public by default

Example:

```
interface A
{
      int  a=50;
      float b=30.8;
}
class B implements A
{
       int len=a;
       public float calculate()
       {
          return(len*b);
       }
}
Class C
{
    public static void main(String args[])
    {
          B b=new B():
          System.out.println("Result=" +(b.calculate()));
    }
}
```

Extra topic:

Difference between Abstarct class and interface:

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, **abstract class achieves partial abstraction** (0 to 100%) whereas **interface achieves fully abstraction (100%).**