5. PACKAGES AND INTERFACES

❖ Packages:

- A package is a group of classes and interfaces.
- Package can be categorized in two form:
    - built-in package such as java, lang, awt, javax, swing, net, io, util, sql etc.
    - user-defined package which is created by user.
- **Advantages of Packages:**
    - Package is used to categorize the classes and interfaces so that they can be easily maintained.
    - Package provides access protection.
    - Package removes naming collision.
- **Naming rules for packages:**
    - Package names are written in all lowercase to avoid conflict with the names of classes or interfaces.
    - The directory name must be same as the name of package that is created using "**package"** keyword in the source file.
    - Before running a program, the class path must be picked up till the main directory (or package) that is used in the program.
    - If we are not including any package in our java source file then the source file automatically goes to the default package.
    - In general, we start a package name begins with the order from top to bottom level.
    - In case of the internet domain, the name of the domain is treated in reverse (prefix) order.

❖ How to create a Package?

- To create a package, you choose a name for the package and put a package statement with that name at the top of every source file that contains the types (classes, and interfaces) that you want to include in the package.
- The package statement (example-**package graphics;**) must be the first line in the source file.
- There can be only one package statement in each source file, and it applies to all types in the file.
- **Steps to crate packages:**
    - Declare the package at the beginning of a file using the from
      package packagename;
    - Define the class that is to be put in the package and declare it public.
    - Create a subdirectory under the directory where the main source files are stored.
    - Store the listing as the classname.java file in the subdirectory created.
    - Compile the file. This creates Class file in the subdirectory.
- You can create a hierarchy of packages.
- To do so, simply separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here:
    - **package pkg1[.pkg2[.pkg3]];**
- A package hierarchy must be reflected in the file system of your Java development system.
- For example, a package declared as

- package java.awt.image;
- Needs to be stored in **java\awt\image** in a Windows environment.
  - You cannot rename a package without renaming the directory in which the classes are stored
- **Example :**
  - Save program in your directory(e.g.-**D:\Java_prog\Package\A.java**)
  - Compile using **D:\>Java_prog\Package\javac –d . A.java**
  - Then **pack** directory has been generate with A.class file.
  - Then run program **D:\>Java_prog\Package\java pack.A**

```
package pack;
public class A
{
        public static void main(String args[])
        {
                System.out.println("B. S. Patel Polytechnic");
        }
}
```

Finding Packages and CLASSPATH

- As just explained, packages are mirrored by directories.
- This raises an important question:
- How does the Java run-time system know where to look for packages that you create?
- The answer has three parts.
  - First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
  - Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
  - Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.
- For example, consider the following package specification:
  - package pk1
- In order for a program to find pk1**,** one of three things must be true. Either the program can be executed from a directory immediately above pk1, or the **CLASSPATH** must be set to include the path to pk1, or the **-classpath** option must specify the path to pk1 when the program is run via **java**.
- When the second two options are used, the class path must not include pk1, itself.It must simply specify the path to pk1.
- For example, in a Windows environment, if the path to **PK1** is
  - C:\MyPrograms\Java\ pk1
- Then the class path to **pk1** is
  - C:\MyPrograms\Java

Access Specifiers(Visibility Modifiers) OR Access protection in packages:

Access protection	**Description**
No modifier (default)	The classes and members specified in the same package are accessible to all the classes inside the same package.
public	The classes, methods and member variables under this specifier can be accessed from anywhere.

protected	The classes, methods and member variables under this modifier are accessible by all subclasses, and accessible by code in same package.
private	The methods and member variables are accessible only inside the class.

- **Access to fields in Java at a look:**

Access By	public	protected	default	private
The class itself	Yes	Yes	Yes	Yes
A subclass in same package	Yes	Yes	Yes	No
Non sub-class in the same package	Yes	Yes	Yes	No
A subclass in other package	Yes	Yes	No	No
Non subclass in other package	Yes	No	No	No

Importing package

- In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.
- This is the general form of the **import** statement:
  import pkg1[.pkg2].(classname|*);
- Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (**.**).
- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.
- Finally, you specify either an explicit classname or a star (*****), which indicates that the Java compiler should import the entire package.
- This code fragment shows both forms in use:
  - import java.util.Date;
  - import java.io.*;
- **Example:**

```
package pack;
public class A
{
        int a;
        int b;
        A(int i,int j)
        {
                a=i;
                b=j;
        }
        public void display()
        {
                System.out.println(a + "&" + b);
        }
}
```

- compile your program like :- **D:\>Java_prog\Package\javac –d . A.java**
- This source file should be named A.java and stored in the sub directory pack.
- Now compile this java file.
- The resultant A.class will be stored in the same sub directory.

```
import pack.A;
class B
{
        public static void main(String args[])
        {
                A ob= new A(5,9);
                ob.display();
        }
}
```

- The source file should be saved as B.java and then compiled.
- The source file and the compiled file would be saved in the directory of which pack was a subdirectory.
    - **D:\>Java_prog\Package\javac B. java**
    - **D:\>Java_prog\Package\java B**
    - **Output:- 5&9**

Interfaces

- Interface is just like a class.
- Interface declares with interface keyword.
- Interface does not have any implementation of method.
- In Java Interface defines the methods but does not implement them.
- **Interface is collection of methods and variables.**
- In interface variable consider as a constant and method consider as abstract.
- More than one class can be implement interface.
- Interface extends two or more interfaces.
- It means using interface we can create multiple inheritance.
- **Syntax:**

```
[access] interface InterfaceName [extends OtherInterface]
{
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

- **Implementing Interfaces**
    - Once an interface has been defined, one or more classes can implement that interface.
    - To implement an interface, include the implements keyword in a class definition, and then create the methods defined by the interface.
    - The general form of a class that includes the implements clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]]
{
    // class-body
}
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared public.

➢ **Example:-1**

```
interface I1
{
        public int a=10;
        public int b=20;
        public void sum();
}
class one implements I1
{
        public void sum()
        {
                System.out.print("Sum-->"+(a+b));
        }
}
class  two
{
        public static void main(String[] args)
        {
                one ob1=new one();
                ob1.sum();
        }
}
```

➢ **Example:-2**

```
interface I1
{
        public void show();
}
interface I2
{
        public void show1();
}
class A implements I1,I2
{
        public void show() {
                System.out.println("i am the implentation of interface first");   }
        public void show1() {
                System.out.println("I am the implentation of interface second");  }
}
class  interface1
{
        public static void main(String[] args)  {
                A a=new A();
                a.show();
                a.show1();
        }
 }
```