Part-B

Packages and Interfaces:-

1. **Defining**
2. **Creating and Accessing a Package**
3. **Understanding CLASSPATH**
4. **Importing Packages**

Packages:-

Defining:-

It is necessary in software development to create several classes and interfaces. After creating these classes and interfaces, it is better if they are divided into some groups depending on their relationship. Thus, the classes and interfaces which handle similar or same task are put into the same directory. This directory or folder is also called a package.

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

There are several following advantages of package concept:

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.

/*

- Packages are useful to arrange related classes and interfaces into a group. This makes all the classes and interfaces performing the same task to put together in the same package. For example, in Java, all the classes and interfaces which perform input and output operations are stored in java.io package.
- packages hide the classes and interfaces in a separate sub directory, so that accidental deletion of classes and interfaces will not take place.
- The classes and interfaces of a package are isolated from the classes and interfaces of another package. This means that we can use same names for classes of two different classes. For example, there is a Date class in java.util package and also there is another Date class available in java.sql package.
- A group of packages is called a library. The classes and interfaces of a package are like books in a library and can be reused several times. This reusability nature of packages makes programming easy. Just think, the packages in Java are created by JavaSoft people only once, and millions of programmers all over the world are daily by using them in various programs.

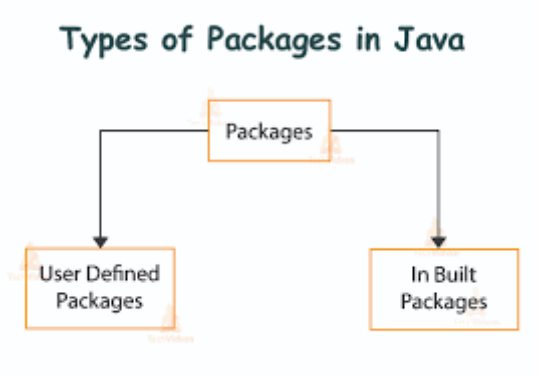
*/

Different Types of Packages

There are two different types of packages in Java. They are:

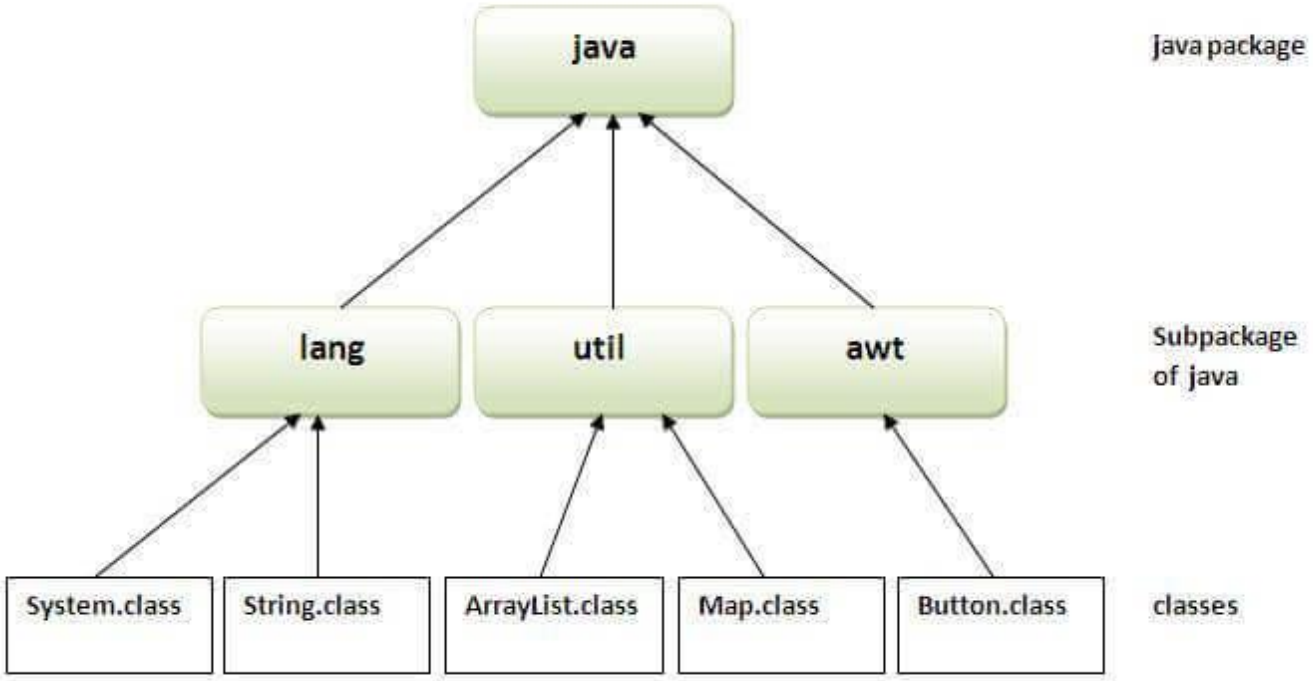
- Built-in packages
- User-defined packages

(or)



Built-in or Pre-defined Packages:-

The Package which are already created by java developer people are called pre-defined package.



Java.lang.* :-This package as primary classes and interfaces essential for Java language. It consists of Wrapper Classes . Ex: System, String, Integer, Character classes etc..,

Java.util.* :-This Package consist useful classes and interfaces like Stack, LinkedList, Vector, ArrayList, Date etc.,

java.io.* :-This packages handles Files and input, output related tasks. Ex:- File, FileInputStream, FilOutputStream, FileReader, FileWriters.

Java.awt.* :-awt means Abstract Window Toolkit. This packages helps to develop GUI. It consists of two important some packages.

1. Java.awt.*;

2. Java.awt.event.*;

3. Java.awt.image.*;

Javax.swing.* :-This package helps to develop GUI. Java.awt will have additional features than java.awt.*;

Java.net.* :-Client and Server programming can be done using packages it uses TCP/IP TCP means Transmission Control Protocol and IP means Internet Protocol.

Java.applet.* : - Applets are small intelligent program which travel from one place to another place on Internet and executed in client side.

Java.sql.*:-This package helps to connect to the database like Oracle, MsAccess etc..,

Java.beans.*:-Beans are software reusalble components in the network they can be develop this package.

Java.rmi.*:-rmi stands for Remote Method Invocation. This object which exists one computer in the network can be invoked from another computer and can be used.

Javax.servlet.* :-Servlets are Server Side Programming which handles clients.

Creating and Accessing a Package:-

User Defined Package:-

Creating a Package:-

the users of the Java language can also create their own packages. They are called user-defined packages. User defined packages can also be imported into other classes and used exactly in the same way as the Built-in packages.

Let us see how to create a package of our own and use it in any other class. To create a package the keyword package used as:

Syntax:-

1. Package packagename; //to create a package

2. Package packagename.subpackagename; //to create a sub package within a package

Simple example of java package:

The **package keyword** is used to create a package in java.

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

How to compile java package

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

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```

See the output to understand how to compile a Java program that contains a package. The –d option (switch) tells the Java compiler to create a separate sub directory and place the .class file there. The dot(.) after –d indicates that the package should be created in the current directory i.e., C:\. We have written as:

How to run java package program

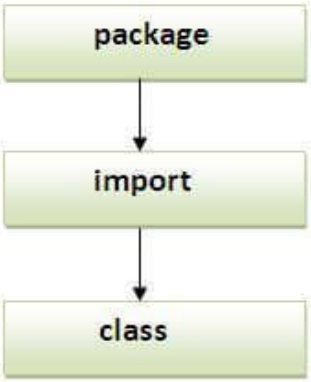
You need to use fully qualified name e.g. mypack.Simple etc to run the class.

Ex:-java mypack.Simple

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

Output:Welcome to package

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



Note: If you import a package, subpackages will not be imported.

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.

Subpackage 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.

Syntax:-

Package packagename.subpackagename; //to create a sub package within a package

Example of Subpackage

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

To Compile: javac -d . Simple.java

To Run: java high.sub.Simple

Output:Hello subpackage

Accessing a 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 packagename.*

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 of package that import the packagename.*

```
//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();
  }
}
```

To Compile: javac -d . B.java

To Run: java mypack.B

Output:Hello

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 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();
  }
}
```

To Compile: javac -d . B.java

To Run: java mypack.B

Output:Hello

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
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();
```

```
  }
}
```

To Compile: javac -d . B.java

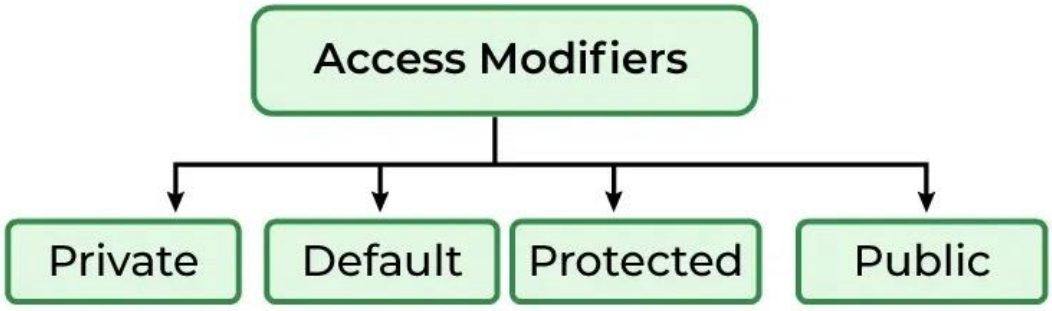
To Run: java mypack.B

Output:Hello

Accessing a Packages using Access Modifiers:-

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers using to access Packages:



1) Private

The access level of a private modifier is only within the class. It cannot be accessed from outside the 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 a compile-time error.

```
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
   }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

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

Note: A class cannot be private or protected except nested class.

2) Default

The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. 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 pack;
class A{
  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();//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

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.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack 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 pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

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

```
}
```

Output:Hello

4) Public

The **public access modifier** is accessible everywhere. It can be accessed from within the class, outside the class, within the package and outside the package. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
```

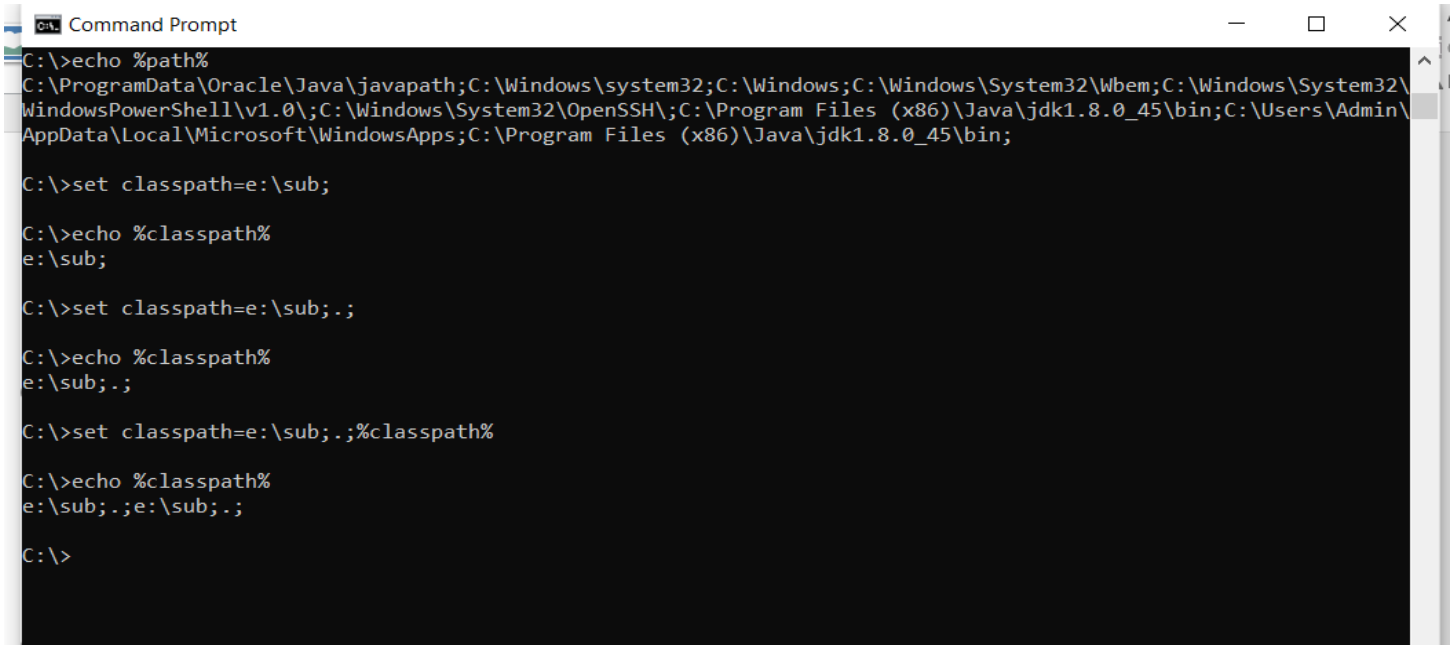
```
//save by B.java

package mypack;
import pack.*;

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

Output:Hello

Understanding CLASSPATH:-

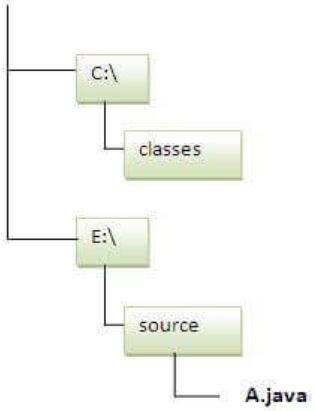
The CLASSPATH is an environment variable that tells the Java compiler where to look for class files import. CLASSPATH is generally set to a directory or a Jar(Java Archive) file.

To see what is there currently in the CLASSPATH variable in your system, you can type in Windows
C:\>echo %path% \\used to know the all the classpaths in the particular directory
C:\>echo %classpath% \\used to know the current classpath name



send the class file to another directory or drive?

There is a scenario, I am created one class A.java in a e directory folder source. I want to put the class file of A.java source file in classes folder of c: drive.




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

To Compile:

e:\sources> javac -d c:\classes A.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The –classpath or -cp switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

Output:Welcome to package

e:\sources> java -classpath c:\classes mypack.Simple

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

4.Importing Packages

There are three ways to import a 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 subpackages.

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

Syntax:-

```
import Packagename.*;

import packagename.subpackagename.*;
```

Example of package that import the packagename.*

```
//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();
  }
}
```

To Compile: javac -d . B.java

To Run: java mypack.B

Output:Hello

2) Using packagename.classname

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

Syntax:-

Import packagename.classname;

import packagename.subpackagename.classname;

Example of package by import package.classname

```
    //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();
  }
}
```

To Compile: javac -d . B.java

To Run: java mypack.B

Output:Hello

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
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();
  }
}
```

To Compile: javac -d . B.java
To Run: java mypack.B

Output:Hello

1.How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same.

For example:

//save as A.java

package p1;

public class A{}

//save as B.java

package p2;

public class B{}

2.How to put same class name in a packages?

If you want to put two classes names same so, create two different package names and create same class names.

save as A.java

package p1;

public class A{}

//save as A.java

package p2;

public class A{}

The JAR Files

A Java Archive file (JAR) is a file that contains compressed version of several .class files, audio files, image files or directories. JAR file is useful to bundle up several files related to a project and use them easily.

Let us see how to create a .jar file and related commands which help us to work with .jar files:

¬ To create .jar file, JavaSoft people have provided jar command, which can be used in the following way:

jar cf jarfilename inputfiles

¬ Here, cf represents create file. For example, assuming our package pack is available in C:\directory, to convert it into a jar file with the name pack.jar, we can given the command as:

C:\> jar cf pack.jar pack

Now, pack.jar file created.

¬ To view the contents of a .jar fie, we can use the jar command as:

jar tf jarfilename

Here, tf represents table view of file contents. For example, to view the contents f our pack.jar file, we can give the command:

C:\> jar tf pack.jar

Now, the contents of pack.jar are displayed as:

META-INF

META-INF/MANIFEXT.MF

pack/

pack/Addition.class

pack/Subtraction.class

The first two entries represent that there is a manifest file created and added to pack.jar file. The third entry represents the sub directory with the name pack and the last two represents the file names in the directory pack.

When we create a .jar file, it automatically receives the default manifest file. There can be only one manifest file in an archive, and it always has the pathname

META-INF/MANIFEXT.MF

This manifest file is useful to specify the information about other files which are packaged.

¬ To extract the files from a .jar file, we can use:

jar xf jarfilename

Here, xf represents extract files from the jar file. For example, to extract the contents of our pack.jar file, we can write: C:\> jar xf pack.jar

This will create the following directories in C:\

META-INF

Pack // in this directory, we can see Addition.class and Subtraction.class

Now, we know how to create a .jar file, let us see how it can be used. In software development, any package is converted into a .jar file and stored in a separate sub directory. For example, convert our package pack into pack.jar file and store it in a sub directory e:\temp

Now set the **CLASSPATH permanently** to the pack.jar file by following the procedure shown here:

◊First, go to start->settings -> Control panel

◊In Control Panel, select System and double click on it, System properties dialog box appears

◊In this, select Advanced tab and then click on Environment variables button

◊Go to User variables and click on New button

◊Set the CLASSPATH variable to pack.jar and also to the current directory, by typing at:

◊Variable name: CLASSPATH

◊Variable value : E:\temp\pack.jar;

◊Then click on OK button

◊Then click on OK in the Environment variables window and System Properties windows.

◊Close the Control Panel.

After setting the CLASSPATH permanently as shown in the preceding steps to the apck.jar file, it is available any were in that computer system. Our program (Use.java) which uses the package may present in any directory, it can be compiled and run without any problem.