Unit-II-PART-III

Java Packages

Package is a collection of related classes. Java uses package to group related classes, interfaces and sub-packages in any Java project.

We can assume package as a folder or a directory that is used to store similar files.

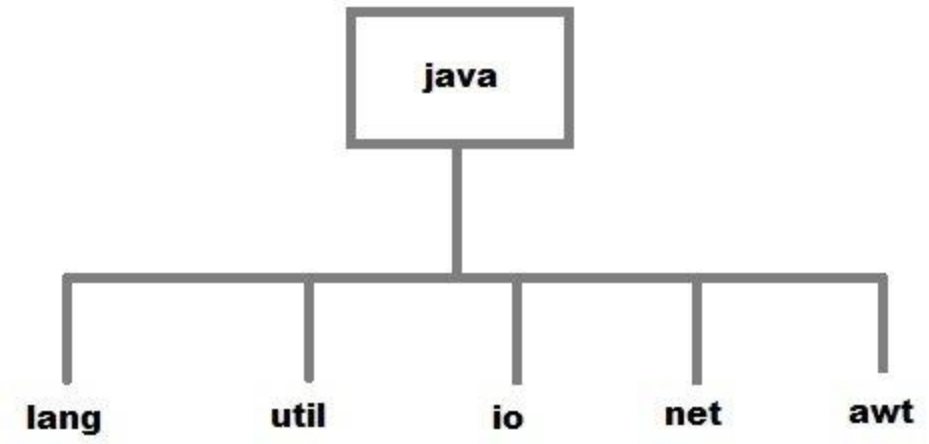
In Java, packages are used to avoid name conflicts and to control access of class, interface and enumeration etc. Using package it becomes easier to locate the related classes and it also provides a good structure for projects with hundreds of classes and other files.

Lets understand it by a simple example, Suppose, we have some math related classes and interfaces then to collect them into a simple place, we have to create a package.

Types Of Java Package

Package can be built-in and user-defined, Java provides rich set of built-in packages in form of API that stores related classes and sub-packages.

- **Built-in Package:** math, util, lang, i/o etc are the example of built-in packages.
- **User-defined-package:** Java package created by user to categorize their project's classes and interface are known as user-defined packages.



How to Create a Package

Creating a package in java is quite easy, simply include a package command followed by name of the package as the first statement in java source file.

```
package mypack;
public class employee
 {
    String empId;
    String name;
}
```

The above statement will create a package woth name **mypack** in the project directory.

Java uses file system directories to store packages.

For example the .java file for any class you define to be part of **mypack** package must be stored in a **directory** called **mypack**.

Additional points about package:

- Package statement must be first statement in the program even before the import statement.
- A package is always defined as a separate folder having the same name as the package name.
- Store all the classes in that package folder.
- All classes of the package which we wish to access outside the package must be declared public.
- All classes within the package must have the package statement as its first line.
- All classes of the package must be compiled before use.

Example of Java packages

Now lets understand package creation by an example, here we created a **leanjava** package that stores the FirstProgram class file.

```
//save as FirstProgram.java

package learnjava;

public class FirstProgram

{

   public static void main(String args[])

 {
```

```
        System.out.println("Welcome to package example");
    }
}
```

How to compile Java programs inside packages?

This is just like compiling a normal java program. If you are not using any IDE, you need to follow the steps given below to successfully compile your packages:

javac -d . FirstProgram.java

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

How to run Java package program?

To run the compiled class that we compiled using above command, we need to specify package name too. Use the below command to run the class file.

```
java learnjava.FirstProgram

Welcome to package example
```

After running the program, we will get "Welcome to package example" message to the console. You can tally that with print statement used in the program.

How to import Java Package

To import java package into a class, we need to use java **import** keyword which is used to access package and its classes into the java program.

Use import to access built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to any class that is present in a different package:

1. without import the package
2. import package with specified class
3. import package with all classes

Lets understand each one with the help of example.

Accessing package without import keyword

If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, other classes in the same package will not be accessible.

For this approach, there is no need to use the import statement. But you will have to use the fully qualified name every time you are accessing the class or the interface.

This is generally used when two packages have classes with same names. For example: java.util and java.sql packages contain Date class.

Example

In this example, we are creating a class A in package pack and in another class B, we are accessing it while creating object of class A.

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

Hello

Import the Specific Class

Package can have many classes but sometimes we want to access only specific class in our program in that case, Java allows us to specify class name along with package name. If we use
import packagename.classname statement then only the class with name classname in the package will be available for use.

Example:

In this example, we created a class Demo stored into pack package and in another class Test, we are accessing Demo class by importing package name with class name.

```
//save by Demo.java

package pack;

public class Demo {

   public void msg() {

      System.out.println("Hello");

   }

}
```

```
//save by Test.java
package mypack;
import pack.Demo;
class Test {
    public static void main(String args[]) {
        Demo obj = new Demo();
        obj.msg();
    }
}
```

```
Hello
```

Import all classes of the package

If we use **packagename.* statement**, then all the classes and interfaces of this package will be accessible but the classes and interface inside the sub-packages will not be available for use.

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

Example :

In this example, we created a class First in **learnjava** package that access it in another class Second by using import keyword.

```
//save by First.java

package learnjava;

public class First{

   public void msg() {

      System.out.println("Hello");

   }

}


//save by Second.java

package Java;
```

```
import learnjava.*;
class Second {
   public static void main(String args[]) {
      First obj = new First();
      obj.msg();
   }
}
```

Hello

Java Sub Package and Static Import

In this tutorial, we will learn about sub-packages in Java and also about the concept of static import and how it is different from normal import keyword.

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.

Note: The standard of defining package is **domain.company.package** e.g. LearnJava.full.io

Example:

In this example, we created a package **LearnJava** and a sub package **corejava** in the Simple.java file.

```
package LearnJava.corejava;

class Simple{

  public static void main(String args[]){

   System.out.println("Hello from subpackage");

  }

}
```

To compile the class, we can use the same command that we used for package. Command is given below.

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

To run the class stored into the created sub package, we can use below command.

```
java LearnJava.corejava.Simple
```

After successful compiling and executing, it will print the following output to the console.

```
Hello from subpackage
```

Static import in Java

static import is a feature that expands the capabilities of **import** keyword.

It is used to import static member of a class.

We all know that static member are referred in association with its class name outside the class.

Using **static import**, it is possible to refer to the static member directly without its class name. There are two general form of static import statement.

We can import single or multiple static members of any class.

To import single static member we can use statement like the below.

import static java.lang.Math.sqrt; //importing static method sqrt of Math class

The second form of static import statement, imports all the static member of a class.

```
import static java.lang.Math.*;   //importing all static member of Math class
```

Example without using static import

This is alternate way to use static member of the class. In this case, we don't need to use import statement rather we use direct qualified name of the class.

```
public class Test
{
   public static void main(String[] args)
   {
      System.out.println(Math.sqrt(144));
   }
}
12
```

Example using static import

In this example, we are using import statement to import static members of the class. Here, we are using * to import all the static members.

```
import static java.lang.Math.*;

public class Test

{

   public static void main(String[] args)

   {

      System.out.println(sqrt(144));

   }

}
```

O/P:-

Java Interfaces

Interface is a concept which is used to achieve abstraction in Java.

This is the only way by which we can achieve full abstraction. Interfaces are syntactically similar to classes, but you cannot create instance of an Interface and their methods are declared without any body.

It can have When you create an interface it defines what a class can do without saying anything about how the class will do it.

It can have only abstract methods and static fields.

When an interface inherits another interface extends keyword is used

whereas class use implements keyword to inherit an interface.

Advantages of Interface

- It Support multiple inheritance
- It helps to achieve abstraction
- It can be used to achieve loose coupling.

Syntax:

```
interface interface_name
{
// fields
// abstract/private/default methods
}
```

Interface Key Points

Methods inside interface must not be static, final, native or strictfp.

All variables declared inside interface are implicitly public, static and final.

All methods declared inside interfaces are implicitly public and abstract.

Interface can extend one or more other interface.

Interface cannot implement a class.

Interface can be nested inside another interface.

Let's take a simple code example and understand what interfaces are:

```
interface Moveable
{
    int AVERAGE-SPEED = 40;
    void move();
}
```

NOTE: Compiler automatically converts methods of Interface as public and abstract, and the data members as public, static and final by default.

Example of Interface implementation

In this example, we created an interface and implemented using a class. lets see how to implement the interface.

```
interface Moveable
{
    int AVG-SPEED = 40;
    void move();
}
class Vehicle implements Moveable
{
    public void move()
    {
    System .out. print in ("Average speed is"+AVG-SPEED");
    }
    public static void main (String[] arg)
    {
        Vehicle vc = new Vehicle();
        vc.move();
```

```
    }
}
```

Average speed is 40.

Interfaces supports Multiple Inheritance

Though classes in Java doesn't support multiple inheritance, but a class can implement more than one interfaces.

In this example, two interfaces are implemented by a class that show implementation of multiple inheritance.

```
interface Moveable
{
    boolean isMoveable();
}
interface Rollable
{
    boolean isRollable
}
```

```
class Tyre implements Moveable, Rollable
{
    int width;
    boolean isMoveable()
    {
        return true;
    }
    boolean isRollable()
    {
        return true;
    }

    public static void main(String args[])
    {
        Tyre tr = new Tyre();
        System.out.println(tr.isMoveable());
        System.out.println(tr.isRollable());
    }
```

```
}
```

o/p:-

true

true

Interface extends other Interface

Interface can inherit to another interface by using extends keyword.

But in this case, interface just inherit, does not provide implementation.

Implementation can be provided by a class only.

```
interface NewsPaper
{
    news();
}
interface Magazine extends NewsPaper
{
    colorful();
}
```

Difference between an interface and an abstract class?

Interface and abstract class both are used to implement abstraction ut

have some differences as well. Some of differences are listed below.

Abstract class is a class which contain one or more abstract methods, which has to be implemented by its sub classes.	Interface is a Java Object containing metho declaration but no implementation. The classes which implement the Interfaces mu provide the method definition for all the methods.
Abstract class is a Class prefix with an abstract keyword followed by Class definition.	Interface is a pure abstract class which star with interface keyword.
Abstract class can also contain concrete methods.	Whereas, Interface contains all abstract methods and final variable declarations.
Abstract classes are useful in a situation that Some general methods should be implemented and specialization behavior should be implemented by child classes.	Interfaces are useful in a situation that all properties should be implemented.

Default Methods in Interface

In Java 8 version a new feature is added to the interface, which was default method.

Default method is a method that can have its body.

It means default method is not abstract method, it is used to set some default functionality to the interface.

Java provides default keyword to create default method. Let's see an example:

```
interface Abc
{
   // Default method
   default void msg()
  {
      System.out.println("This is default method");
   }
   // Abstract method
   void greet(String msg);
```

```
}
public class Demo implements Abc
{
   public void greet(String msg)
  {       // implementing abstract method
      System.out.println(msg);
   }
   public static void main(String[] args)
  {
      Demo d = new Demo();
      d.msg();   // calling default method
      d.greet("Say Hi");  // calling abstract method
   }
}
```

```
This is default method
Say Hi
```

Static methods in Interface

From Java 8, Java allows to declare static methods into interface.

The purpose of static method is to add utility methods into the interface.

In the below example, we have created an interface Abc that contains a static method and an abstract method as well. See the below example.

```
interface Abc
{
   // static method
   static void msg()
  {
      System.out.println("This is static method");
   }
   // Abstract method
   void greet(String msg);
   }
```

```
public class Demo implements Abc
{
   public void greet(String msg)
  {       // implementing abstract method
       System.out.println(msg);
   }
   public static void main(String[] args)
   {
      Demo d = new Demo();
      Abc.msg();   // calling static method
      d.greet("Say Hi");  // calling abstract method
    }
}
```

o/p:-

```
This is static method
Say Hi
```