Packages In Java

Difficulty Level : Easy Last Updated : 28 Jun, 2021

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.
We can reuse existing classes from the packages as many time as we need it in our program.

How packages work?

Package names and directory structure are closely related. For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present *college*. Also, the directory *college* is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.
Package naming conventions : Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.

Subpackages: Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no

access privileges, i.e., they are considered as different package for protected and default access specifiers.

Example :

```
import java.util.*;
```

util is a subpackage created inside **java** package.

Accessing classes inside a package

Consider following two statements :

```
// import the Vector class from util package.
import java.util.vector;

// import all the classes from util package
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
- Second statement imports all the classes from **util** package.

```
// All the classes and interfaces of this package
// will be accessible but not subpackages.
import package.*;

// Only mentioned class of this package will be accessible.
import package.classname;

// Class name is generally used when two packages have the same
// class name. For example in below code both packages have
// date class so using a fully qualified name to avoid conflict
import java.util.Date;
import my.packag.Date;
```

```
// Java program to demonstrate accessing of members when
// corresponding classes are imported and not imported.
```

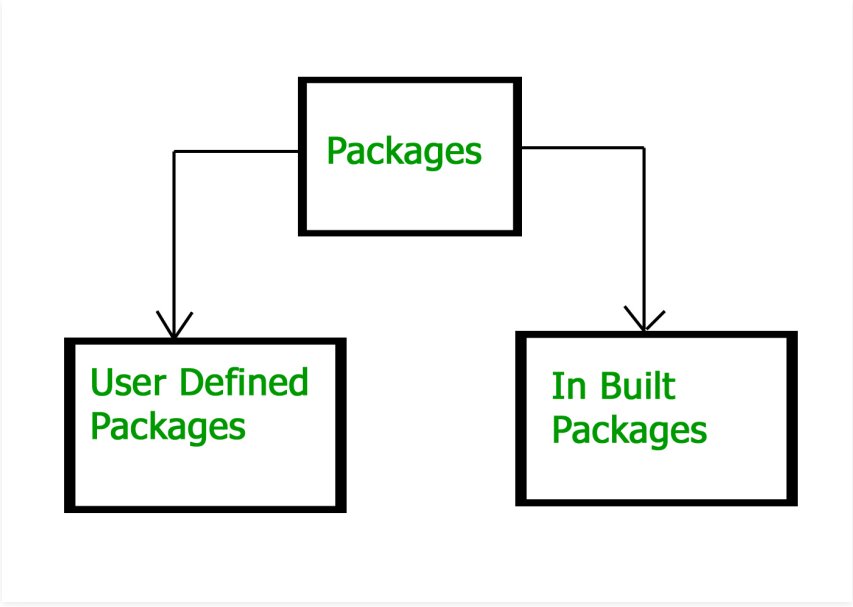
```
import java.util.Vector;

public class ImportDemo
{
   public ImportDemo()
   {
      // java.util.Vector is imported, hence we are
      // able to access directly in our code.
      Vector newVector = new Vector();

      // java.util.ArrayList is not imported, hence
      // we were referring to it using the complete
      // package.
      java.util.ArrayList newList = new java.util.ArrayList();
   }

   public static void main(String arg[])
   {
      new ImportDemo();
   }
}
```

Types of packages:



Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.

2) **java.io:** Contains classed for supporting input / output operations.

3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) **java.applet:** Contains classes for creating Applets.

5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

6) **java.net:** Contain classes for supporting networking operations.

User-defined packages

These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = "GeeksforGeeks";

      // Creating an instance of class MyClass in
      // the package.
```

```
        MyClass obj = new MyClass();

        obj.getNames(name);
    }
}
```

Note : MyClass.java must be saved inside the **myPackage** directory since it is a part of the package.

Using Static Import

Static import is a feature introduced in **Java** programming language (versions 5 and above) that allows members (fields and methods) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.
Following program demonstrates **static import** :

```
// Note static keyword after import.
import static java.lang.System.*;

class StaticImportDemo
{
   public static void main(String args[])
   {
        // We don't need to use 'System.out'
        // as imported using static.
        out.println("GeeksforGeeks");
   }
}
```

Output:

```
GeeksforGeeks
```

Handling name conflicts

The only time we need to pay attention to packages is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:

```
import java.util.*;
import java.sql.*;

//And then use Date class, then we will get a compile-time error :

Date today ; //ERROR-- java.util.Date or java.sql.Date?
```

The compiler will not be able to figure out which Date class do we want. This problem can be solved by using a specific import statement:

```
import java.util.Date;
import java.sql.*;
```

If we need both Date classes then, we need to use a full package name every time we declare a new object of that class.
For Example:

```
java.util.Date deadLine = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Directory structure

The package name is closely associated with the directory structure used to store the classes. The classes (and other entities) belonging to a specific package are stored together in the same directory. Furthermore, they are stored in a sub-directory structure specified by its package name. For example, the class Circle of package com.zzz.project1.subproject2 is stored as "$BASE_DIR\com\zzz\project1\subproject2\Circle.class", where $BASE_DIR denotes the base directory of the package. Clearly, the "dot" in the package name corresponds to a sub-directory of the file system.

The base directory ($BASE_DIR) could be located anywhere in the file system. Hence, the Java compiler and runtime must be informed about the location of the $BASE_DIR so as to locate the classes. This is accomplished by an environment variable called CLASSPATH. CLASSPATH is similar to another environment variable PATH, which is used by the command shell to search for the executable programs.

Setting CLASSPATH:
CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment: In Windows, choose control panel ?

System ? Advanced ? Environment Variables ? choose “System Variables” (for all the users) or “User Variables” (only the currently login user) ? choose “Edit” (if CLASSPATH already exists) or “New” ? Enter “CLASSPATH” as the variable name ? Enter the required directories and JAR files (separated by semicolons) as the value (e.g., “.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar”). Take note that you need to include the current working directory (denoted by ‘.’) in the CLASSPATH.
To check the current setting of the CLASSPATH, issue the following command:

```
> SET CLASSPATH
```

- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:

```
> SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
```

- Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,

```
> java -classpath c:\javaproject\classes com.abc.project1.subproject2.MyCl
```

Illustration of user-defined packages:
Creating our first package:
File name – ClassOne.java

```
package package_name;

public class ClassOne {
    public void methodClassOne() {
        System.out.println("Hello there its ClassOne");
    }
}
```

Creating our second package:
File name – ClassTwo.java

```
package package_one;
```

```
public class ClassTwo {
    public void methodClassTwo(){
        System.out.println("Hello there i am ClassTwo");
    }
}
```

Making use of both the created packages:
File name – Testing.java

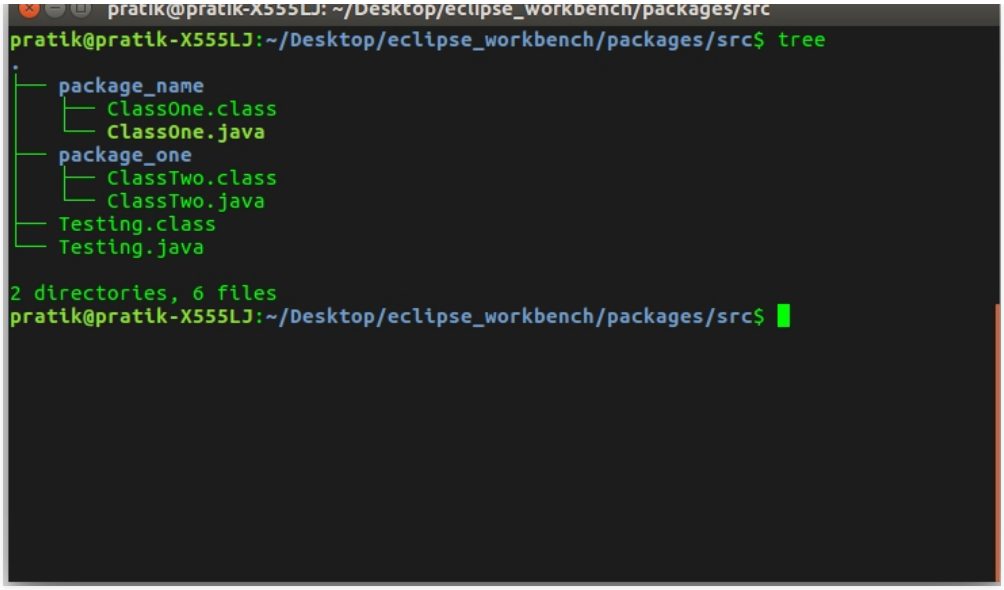
```
import package_one.ClassTwo;
import package_name.ClassOne;

public class Testing {
    public static void main(String[] args){
        ClassTwo a = new ClassTwo();
        ClassOne b = new ClassOne();
        a.methodClassTwo();
        b.methodClassOne();
    }
}
```

Output:

```
Hello there i am ClassTwo
Hello there its ClassOne
```

Now having a look at the directory structure of both the packages and the testing class file:



Important points:

1. Every class is part of some package.
2. If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).
3. All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
4. If package name is specified, the file must be in a subdirectory called name (i.e., the directory name must match the package name).
5. We can access public classes in another (named) package using: **package-name.class-name**

Related Article: Quiz on Packages in Java
Reference: http://pages.cs.wisc.edu/~hasti/cs368/JavaTutorial/NOTES/Packages.html
This article is contributed by **Nikhil Meherwal** and Prateek Agarwal. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.