What are Packages in java and how to use them?

Chaitanya Singh | Filed Under: OOPs Concept | Updated: January 11, 2014

Packages in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages. Many implementations of Java use a hierarchical file system to manage source and class files. It is easy to organize class files into packages. All we need to do is put related class files in the same directory, give the directory a name that relates to the purpose of the classes, and add a line to the top of each class file that declares the package name, which is the same as the directory name where they reside.

In java there are already many predefined packages that we use while programming.

For example: `java.lang`, `java.io`, `java.util` etc.
However one of the most useful feature of java is that we can define our own packages

Advantages of using a package

Before discussing how to use them Let see why we should use packages.

- Reusability: Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
- Easy to locate the files.
- In real life situation there may arise scenarios where we need to define files of the same name. This may lead to “name-space collisions”. Packages are a way of avoiding “name-space collisions”.

Types of package:
1) User defined package: The package we create is called user-defined package.
2) Built-in package: The already defined package like java.io.*, java.lang.* etc are known as built-in packages.

Defining a Package:
This statement should be used in the beginning of the program to include that program in that particular package.
```
package  <package name>;
```

Example:

```
package tools;
public class Hammer {
   public void id ()
   {
       System.out.println ("Hammer");
   }
}
```

Points to remember:
1. At most one package declaration can appear in a source file.
2. The package declaration must be the first statement in the unit.

Naming conventions:
A global naming scheme has been proposed to use the internet domain names to uniquely identify packages. Companies use their reversed Internet domain name in their package names, like this:
`com.company.packageName`

How to Use a Package:

1. We can call it by its full name. For instance,

```
com.myPackage1.myPackage2 myNewClass = new com.myPackage1.myPackage2();
```

However this method is very time consuming. So normally we use the second method.

2. We use the “import” keyword to access packages. If you say import `com.myPackage1.myPackage2`, then every time we type “myPackage2”, the compiler will understand that we mean `com.myPackage1.myPackage2`. So we can do:

```
import com.myPackage1.myPackage2;
class myClass {
   myPackage2 myNewClass= new myPackage2 ();
   …
   …
   …
}
```

There are two ways of importing a package:
Importing only a single member of the package

```
//here ‘subclass’ is a java file in myPackage2
import com.myPackage1.myPackage2.subClass;
class myClass {
   subClass myNewClass= new subClass();
   …
   …
   …
}
```

Importing all members of a package.

```
import com.myPackage1.*;
import java.sql.* ;
```

Also, when we use *, only the classes in the package referred are imported, and not the classes in the sub package.

The Java runtime automatically imports two entire packages by default:
The java.lang package and the current package by default (the classes in the current folder/directory).

Points to remember:
1. Sometimes class name conflict may occur. For example:

There are two packages myPackage1 and myPackage2.Both of these packages contains a class with the same name, let it be `myClass.java`. Now both this packages are imported by some other class.

```
import myPackage1.*;
import myPackage2.*;
```

This will cause compiler error. To avoid these naming conflicts in such a situation, we have to be more specific and use the member's qualified name to indicate exactly which `myClass.java` class we want:

```
myPackage1.myClass myNewClass1 = new myPackage1.myClass ();
myPackage2.myClass myNewClass2 = new myPackage1.myClass ();
```

2. While creating a package, which needs some other packages to be imported, the package statement should be the first statement of the program, followed by the import statement.

Compiling packages in java:

The java compiler can place the byte codes in a directory that corresponds to the package declaration of the compilation unit. The java byte code for all the classes(and interfaces) specified in the source files `myClass1.java` and `myClass2.java` will be placed in the directory named `myPackage1/myPackage2` , as these sources have the following package declaration

```
package  myPackage1.myPackage2;
```

The absolute path of the myPackage1/myPackage2 directory is specified by using the –d (destination directory) option when compiling with the javac compiler.

Assume that the current directory is /packages/project and all the source files are to be found here,the command,

```
javac –d .file1.java file2.java
```

Issued in the working directory will create `./ myPackage1/myPackage2`(and any sub directories required) under the current directory, and place the java byte code for all the classes(and interfaces) in the directories corresponding to the package names. The dot (.) after the –d option denotes the current directory. Without the –d option, the default behaviors of the java compiler is to place all the class files in the current directory rather than the appropriate sub directories.

How do we run the program?
Since the current directory is /packages/project and we want to run `file1.java`,the fully qualified name of the file1 class must be specified in the java command,

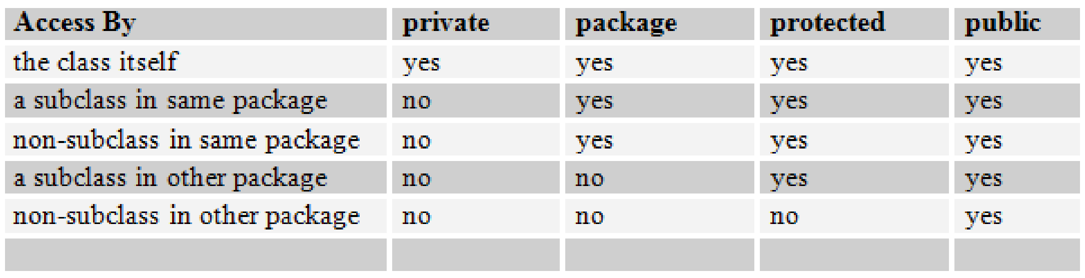
```
java myPackage1.myPackage2.file1
```

Classpath :
It is a environmental variable, which contains the path for the default-working directory (.).
The specific location that java compiler will consider, as the root of any package hierarchy is, controlled by Classpath

Access Specifiers

- `private`: accessible only in the class
- `no modifier`: so-called “package” access — accessible only in the same package
- `protected`: accessible (inherited) by subclasses, and accessible by code in same package
- `public`: accessible anywhere the class is accessible, and inherited by subclasses

Notice that `private protected` is not syntactically legal.



Access By	**private**	**package**	**protected**	**public**
the class itself	yes	yes	yes	yes
a subclass in same package	no	yes	yes	yes
non-subclass in same package	no	yes	yes	yes
a subclass in other package	no	no	yes	yes
non-subclass in other package	no	no	no	yes

References:
The Complete Reference- JAVA 2, Herbert Schildt
A Programmer Guide to Java Certification (Second Edition)