

JAVA - PACKAGES

http://www.tutorialspoint.com/java/java_packages.htm

Copyright © tutorialspoint.com

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are::

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a package:

When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The **package** statement should 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.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

Example:

Let us look at an example that creates a package called **animals**. It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.

Put an interface in the package *animals*:

```
/* File name : Animal.java */
```

```
package animals;

interface Animal {
    public void eat();
    public void travel();
}
```

Now, put an implementation in the same package *animals*:

```
package animals;

/* File name : MammalInt.java */
public class MammalInt implements Animal{

    public void eat(){
        System.out.println("Mammal eats");
    }

    public void travel(){
        System.out.println("Mammal travels");
    }

    public int noOfLegs(){
        return 0;
    }

    public static void main(String args[]){
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}
```

Now, you compile these two files and put them in a sub-directory called **animals** and try to run as follows:

```
$ mkdir animals
$ cp Animal.class MammalInt.class animals
$ java animals/MammalInt
Mammal eats
Mammal travels
```

The import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

Example:

Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;

public class Boss
{
    public void payEmployee(Employee e)
    {
        e.mailCheck();
    }
}
```

What happens if Boss is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example:

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example:

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

The Directory Structure of Packages:

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java:

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**. For example:

```
// File Name : Car.java

package vehicle;

public class Car {
    // Class implementation.
```

```
}
```

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs:

```
...\vehicle\Car.java
```

Now, the qualified class name and pathname would be as below:

- Class name -> vehicle.Car
- Path name -> vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names. Example: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:

```
...\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**

For example:

```
// File Name: Dell.java

package com.apple.computers;
public class Dell{

}
class Ups{

}
```

Now, compile this file as follows using -d option:

```
$javac -d . Dell.java
```

This would put compiled files as follows:

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in `\com\apple\computers\` as

follows:

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:

```
<path-one>\sources\com\apple\computers\Dell.java
```

```
<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\computers.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable:

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell):

- In Windows -> C:\> set CLASSPATH
- In UNIX -> % echo \$CLASSPATH

To delete the current contents of the CLASSPATH variable, use :

- In Windows -> C:\> set CLASSPATH=
- In UNIX -> % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable:

- In Windows -> set CLASSPATH=C:\users\jack\java\classes

- In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH