Java Package Tutorial

(Thai version here)

by Patrick Bouklee

Introduction

Many times when we get a chance to work on a small project, one thing we intend to do is to put all java files into one single directory. It is quick, easy and harmless. However if our small project gets bigger, and the number of files is increasing, putting all these files into the same directory would be a nightmare for us. In java we can avoid this sort of problem by using Packages.

Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to. An obvious example of packaging is the JDK package from SUN (java.xxx.yyy) as shown below:

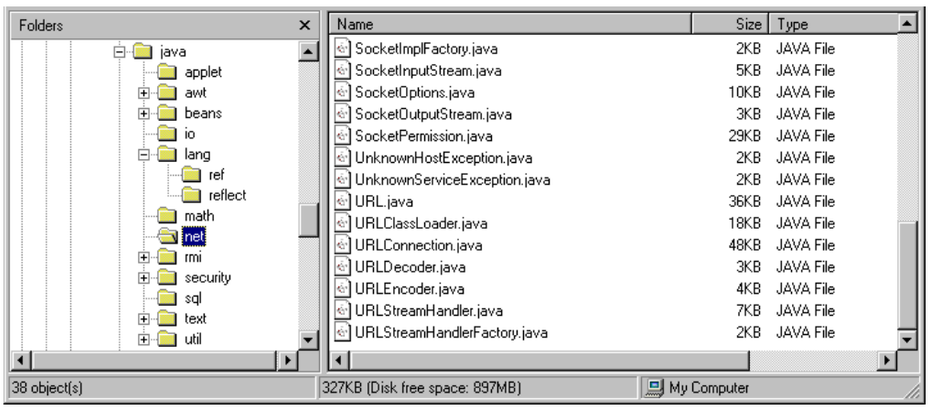

Figure 1. Basic structure of JDK package

Basically, files in one directory (or package) would have different functionality from those of another directory. For example, files in java.io package do something related to I/O, but files in java.net package give us the way to deal with the Network. In GUI applications, it's quite common for us to see a directory with a name "ui" (user interface), meaning that this directory keeps files related to the presentation part of the application. On the other hand, we would see a directory called "engine", which stores all files related to the core functionality of the application instead.

Packaging also help us to avoid class name collision when we use the same class name as that of others. For example, if we have a class name called "`Vector`", its name would crash with the `Vector` class from JDK. However, this never happens because JDK use `java.util` as a package name for the Vector class (`java.util.Vector`). So our `Vector` class can be named as "`Vector`" or we can put it into another package like `com.mycompany.Vector` without fighting with anyone. The benefits of using package reflect the ease of maintenance, organization, and increase collaboration among developers. Understanding the concept of package will also help us manage and use files stored in jar files in more efficient ways.

How to create a package

Suppose we have a file called `HelloWorld.java`, and we want to put this file in a package **`world`**. First thing we have to do is to specify the keyword **`package`** with the name of the package we want to use (**`world`** in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our HelloWorld class below:

```
// only comment can be here
package world;

public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierachy of the class. In our case, we have the **`world`** package, which requires only one directory. So, we create a directory **`world`** and put our HelloWorld.java into it.

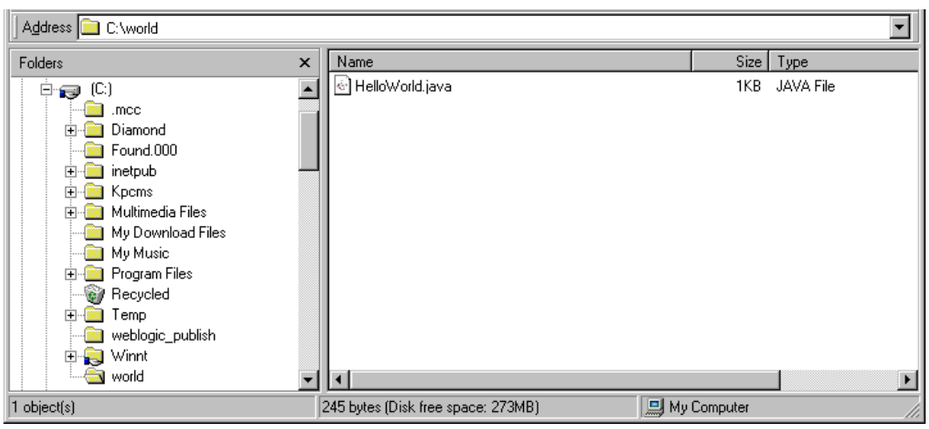

Figure 2. HelloWorld in world package (C:\world\HelloWorld.java)

That's it!!! Right now we have `HelloWorld` class inside `world` package. Next, we have to introduce the `world` package into our **`CLASSPATH`**.

Setting up the CLASSPATH

From figure 2 we put the package `world` under C:. So we just set our `CLASSPATH` as:

```
set CLASSPATH=.;C:\;
```

We set the `CLASSPATH` to point to 2 places, . (dot) and C:\ directory.
Note: If you used to play around with DOS or UNIX, you may be familiar with . (dot) and .. (dot dot). We use . as an alias for the current directory and .. for the parent directory. In our `CLASSPATH` we include this . for convenient reason. Java will find our class file not only from C: directory but from the current directory as well. Also, we use ; (semicolon) to separate the directory location in case we keep class files in many places.

When compiling `HelloWorld` class, we just go to the world directory and type the command:

```
C:\world\javac HelloWorld.java
```

If you try to run this `HelloWorld` using **`java HelloWorld`**, you will get the following error:

```
C:\world>java HelloWorld
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld (wrong name:
world/HelloWorld)
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:442)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:101)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
        at java.net.URLClassLoader.access$1(URLClassLoader.java:216)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:197)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:191)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:290)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:286)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
```

The reason is right now the `HelloWorld` class belongs to the package `world`. If we want to run it, we have to tell JVM about its **fully-qualified class name** `(world.HelloWorld)` instead of its plain class name `(HelloWorld)`.

```
C:\world>java world.HelloWorld
C:\world>Hello World
```

Note: **fully-qualified class name** is the name of the java class that includes its package name

To make this example more understandable, let's put the `HelloWorld` class along with its package (`world`) be under **C:\myclasses** directory instead. The new location of our `HelloWorld` should be as shown in Figure 3:

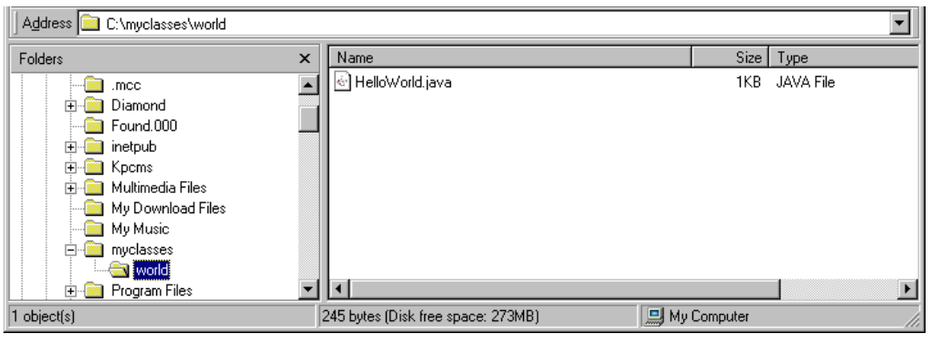

Figure 3. HelloWorld class (in world package) under **myclasses** directory

We just changed the location of the package from C:\world\HelloWorld.java to C:\myclasses\world\HelloWorld.java. Our `CLASSPATH` then needs to be changed to point to the new location of the package `world` accordingly.

```
set CLASSPATH=.;C:\myclasses;
```

Thus, Java will look for java classes from the current directory and **C:\myclasses** directory instead.

Someone may ask "Do we have to run the `HelloWorld` at the directory that we store its class file everytime?". The answer is NO. We can run the `HelloWorld` from anywhere as long as we still include the package `world` in the `CLASSPATH`. For example,

```
C:\>set CLASSPATH=.;C:\;

C:\>set CLASSPATH // see what we have in CLSSPATH
CLASSPATH=.;C:\;

C:\>cd world

C:\world>java world.HelloWorld
Hello World

C:\world>cd ..

C:\>java world.HelloWorld
Hello World
```

Subpackage (package inside another package)

Assume we have another file called **HelloMoon.java**. We want to store it in a subpackage **`"moon"`**, which stays inside package **`world`**. The `HelloMoon` class should look something like this:

```
package world.moon;

public class HelloMoon {
  private String holeName = "rabbit hole";

  public getHoleName() {
    return hole;
  }

  public setHole(String holeName) {
    this.holeName = holeName;
  }
}
```

If we store the package `world` under C: as before, the **HelloMoon.java** would be **c:\world\moon\HelloMoon.java** as shown in Figure 4 below:

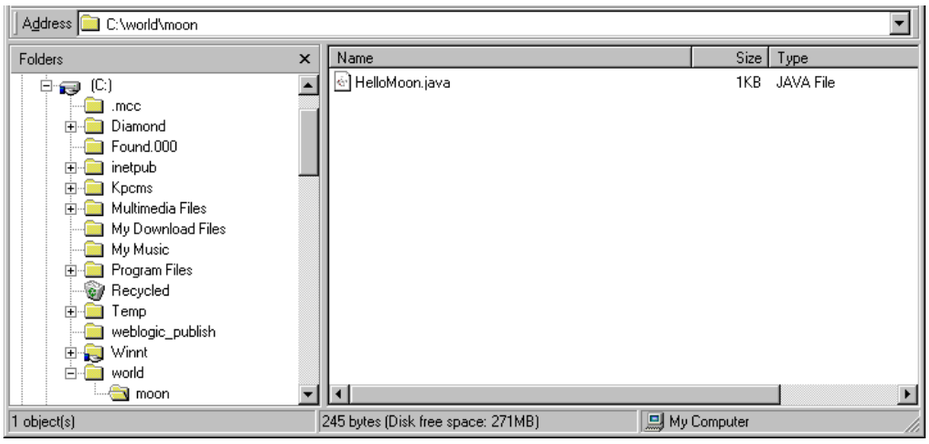

Figure 4. HelloMoon in world.moon package

Although we add a subpackage under package world, we still don't have to change anything in our `CLASSPATH`. However, when we want to reference to the `HelloMoon` class, we have to use `world.moon.HelloMoon` as its fully-qualified class name.

How to use package

There are 2 ways in order to use the public classes stored in package.
1. Declare the fully-qualified class name. For example,

```
...
world.HelloWorld helloWorld = new world.HelloWorld();
world.moon.HelloMoon helloMoon = new world.moon.HelloMoon();
String holeName = helloMoon.getHoleName();
...
```

2) Use an "`import`" keyword:

```
import world.*;  // we can call any public classes inside the world package
import world.moon.*;  // we can call any public classes inside the world.moon package
import java.util.*;  // import all public classes from java.util package
import java.util.Hashtable;  // import only Hashtable class (not all classes in java.util package)
```

Thus, the code that we use to call the HelloWorld and HelloMoon class should be

```
...
  HelloWorld helloWorld = new HelloWorld(); // don't have to explicitly specify world.HelloWorld anymore
  HelloMoon helloMoon = new HelloMoon(); // don't have to explicitly specify world.moon.HelloMoon anymore
...
```

Note that we can call public classes stored in the package level we do the import only. We can't use any classes that belong to the subpackage of the package we import. For example, if we import package `world`, we can use only the `HelloWorld` class, but not the `HelloMoon` class.

Using classes stored in jar file

Jar files are the place where we put a lot of files to be together. We compress these files and make them as a single bundle. Jar files may also include directories, subdirectories to represent class and package hierachy. Normally, we can see what is inside a jar file by using the command **`jar -tvf fileName.jar`** as shown in Figure 5:

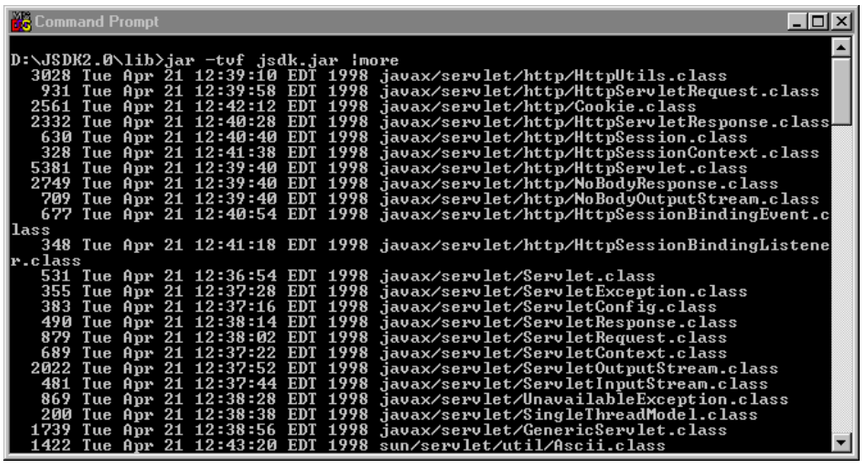

Figure 5. JSDK package viewed by **jar -tvf jsdk.jar** command

From figure 5 we see that there is a class called `javax.servlet.http.Cookie`. We can call this class by

```
import javax.servlet.http.Cookie; // import only Cookie class or
import javax.servlet.http.*; // import the whole javax.servlet.http package
```

But we have to include this package in the `CLASSPATH` as well.

```
set CLASSPATH=.;D:\JSDK2.0\lib\jsdk.jar;
```

Note that if the package is stored inside a jar file, we have to include the jar file with its extension (.jar) in the `CLASSPATH`. However, if the package is a plain directory, we just put the name of directory into the `CLASSPATH`.

Copyright (C) 2000-2001 http://www.jarticles.com/.