**Java Programming**

**Package and Classpath**

This article is applicable to pre-Java 9. Java 9 introduces a higher level of structure called "module" on top of "package". Read "[Java 9 New Features](https://www3.ntu.edu.sg/home/ehchua/programming/java/JDK9_NewFeatures.html)".

**Package**

A *package* is *a collection of related Java entities* (such as classes, interfaces, exceptions, errors and enums). Packages are used for:

1. Resolving naming conflict of classes by prefixing the class name with a package name. For example, com.zzz.Circle and com.yyy.Circle are two distinct classes. Although they share the same class name Circle, but they belong to two different packages: com.zzz and com.yyy. These two classes can be used in the same program and distinguished using the *fully-qualified class name* - package name plus class name. This mechanism is called *Namespace Management*.
2. Access Control: Besides public and private, Java has two access control modifiers – protected and default – that are related to package. A protected entity is accessible by classes in the same package and its subclasses. An entity without access control modifier (i.e., default) is accessible by classes in the same package only.
3. For distributing a collection of reusable classes, usually in a format known as Java Archive (JAR) file.

**Package Naming Convention**

A package name is made up of the *reverse* of the Internet Domain Name (to ensure uniqueness) plus your own organization's internal project name, separated by dots '.'. Package names are in lowercase. For example, suppose that your Internet Domain Name is "zzz.com", you can name your package as "com.zzz.project1.subproject2".

The prefix "java" and "javax" are *reserved* for core Java packages and Java extensions, respectively.

**Package Name & the Directory Structure**

The package name is closely associated with the directory structure used to store the classes. 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.)

In writing GUI programs, we are often confused by two packages: java.awt and java.awt.event. They are two distinct packages sharing some common directory structures. The classes belonging to the package java.awt are stored in directory "$BASE\_DIR\java\awt\" while the classes of package java.awt.event are stored in directory "$BASE\_DIR\java\awt\event\". java.awt and java.awt.event are two distinct packages with common prefix and directory structure. There is no such concept of *sub-package* in Java (i.e., java.awt.event is not a sub-package of java.awt).

**Creating Packages**

To make a class as part of a package, you have to include the package statement as the first statement in the source file.

**Example 1**

We shall write a class called Circle in package com.yyy. It is a good practice to store the source files and the classes in separate directories, typically called "src" and "classes". This is to facilitate the distribution of classes without the source files.

Suppose that our base directory ($BASE\_DIR) is d:\myProject. Create two sub-directories "src" and "classes".

Write the Circle.java and save under "src\com\yyy", as follows:

// src\com\yyy\Circle.java

package com.yyy;

public class Circle {

private double radius;

public Circle(double radius) { this.radius = radius; }

public double getRadius() { return radius; }

public void setRadius(double radius) { this.radius = radius; }

public String toString() { return "Circle[radius=" + radius + "]"; }

}

To compile the source using JDK, we need to use the -d option to specify the base directory of the compiled class, i.e., "classes", as follows::

// Change directory to the package base directory

> **d:**

> **cd myProject**

// Compile

> **javac -d classes src/com/yyy/Circle.java**

// NOTE: you can use either forward slash or backward slash as directory separator in javac

// Show the directories/files

> **tree /f /a**

Folder PATH listing for volume winSystem

Volume serial number is 6A19-E18C

C:.

+---classes

| \---com

| \---yyy

| Circle.class

|

\---src

\---com

\---yyy

Circle.java

The generated class will be stored as "classes\com\yyy\Circle.class". Sub-directories "com" and "yyy" were created automatically with the -d option.

Let's write a test program to use this Circle class. Suppose that TestCircle.java (in default package) is saved in d:\myTest.

// d:\myTest\TestCircle.java

import com.yyy.Circle;

public class TestCircle {

public static void main(String[] args) {

Circle c1 = new Circle(1.23);

System.out.println(c1);

}

}

If we compile TestCircle.java from the directory d:\myTest, we will get a error message, as the compiler cannot find the com.yyy.Circle.class.

d:> **cd \myTest**

d:\myTest> **javac TestCircle.java**

TestCircle.java:2: error: package com.yyy does not exist

import com.yyy.Circle;

^

We need to use the -cp (or -classpath) option to specify the *base directory* of the package com.yyy, in order to locate com.yyy.Circle.class.

d:\myTest> **javac -cp d:\myProject\classes TestCircle.java**

To run the TestCircle, we again get a error, as JRE cannot find the com.yyy.Circle.

d:\myTest> **java TestCircle**

Exception in thread "main" java.lang.NoClassDefFoundError: com/yyy/Circle

Let include the base directory of the package com.yyy in the classpath (to locate com.yyy.Circle):

d:\myTest> **java -cp d:\myProject\classes TestCircle**

Error: Could not find or load main class TestCircle

Caused by: java.lang.ClassNotFoundException: TestCircle

But now, the JRE can't even find the TestCircle class, which is located in the current directory. This is because if CLASSPATH is not explicitly set, it is defaulted to the current directory. However, if CLASSPATH is explicitly set, it does not include the current directory unless the current directory is included. Hence, we need to include current directory (denoted as '.') in the CLASSPATH, together with the base directory of package com.yyy, separated by ';', as follows:

// For Windows

d:\myTest> **java -cp .;d:\myProject\classes TestCircle**

Circle[radius=1.23]

// For Unixes and MacOS X - Use ':' as path separator

$ **java -cp .:$BASE\_DIR TestCircle**

**Example 2**

Suppose that the TestCircle class in Example 1 in defined in a package com.abc, and save as d:\myTest\src\com\abc\TestCircle.java.

// d:\myTest\src\com\abc\TestCircle.java

package com.abc;

import com.yyy.Circle;

public class TestCircle {

public static void main(String[] args) {

Circle c1 = new Circle(1.23);

System.out.println(c1);

}

}

Suppose the compiled class is to be kept as d:\myTest\classes\com\abc\TestCircle.class.

// To compile TestCircle.java, set the current directory and use relative paths for TestCircle.

> **d:**

> **cd \myTest**

d:\myTest> **javac -d classes -cp d:\myProject\classes src\com\abc\TestCircle.java**

// To run TestCircle, need to include the base directory of TestCircle and Circle in classpath.

// Also need to use the fully-qualified name (package name plus class name) for TestCircle

d:\myTest> **java -cp classes;d:\myProject\classes com.abc.TestCircle**

Circle[radius=1.23]

Take note that you need to use fully-qualified name com.abc.TestCircle when running the program.

**Example 3**

In this example, we shall define two classes MyClass3 and MyClass4 in the same package com.zzz.project1.subproject2. The source file and the classes are kept in separate directories "src" and "classes" respectively. Suppose the base directory is "d:\myProject".

// d:\myProject\src\com\zzz\project1\subproject2\MyClass3.java

**package com.zzz.project1.subproject2;**

public class MyClass3 {

private MyClass4 myClass4;

public MyClass3 () { // constructor

System.out.println("MyClass3 constructed");

**myClass4 = new MyClass4();** // use MyClass4 in the same package

}

// main() included here for testing

public static void main(String[] args) {

new MyClass3();

}

}

// d:\myProject\src\com\zzz\project1\subproject2\MyClass4.java

**package com.zzz.project1.subproject2;**

public class MyClass4 { // constructor

public MyClass4() {

System.out.println("MyClass4 constructed");

}

}

// Change directory to the project base directory

> **d:**

> **cd \myProject**

// Compile all classes

> **javac -d classes src\com\zzz\project1\subproject2\MyClass3.java** **src\com\zzz\project1\subproject2\MyClass4.java**

// NOTE: wildcard \* does not work!

// Check the directory tree

> **tree /f /a**  
+---classes  
| \---com  
| \---zzz  
| \---project1  
| \---subproject2  
| MyClass3.class  
| MyClass4.class  
|  
\---src  
 \---com  
 \---zzz  
 \---project1  
 \---subproject2  
 MyClass3.java  
 MyClass4.java

// Run MyClass3

> **java -cp classes com.zzz.project1.subproject2.MyClass3**

MyClass3 constructed  
MyClass4 constructed

**Using IDE**

Managing packages and CLASSPATH yourself with obly JDK is troublesome. IDE such as Eclipses and NetBeans could manage the packages and CLASSPATH for you!!

**The Default Package**

Every Java class must belong to a package. You can explicitly name the package by providing a package statement in the beginning of the source file. If the package statement is omitted, the class belongs to the so-called *default package*, with no sub-directory structure. Use of default package is not recommended other than writing toy program and for quick testing.

**CLASSPATH - For Locating Classes**

CLASSPATH is an environment variable (i.e., global variables of the operating system available to all the processes) needed for the Java compiler and runtime to locate the Java packages/classes used in a Java program. (Why not call PACKAGEPATH?) This is similar to another environment variable PATH, which is used by the Command shell to find the executable programs.

CLASSPATH can be set in one of the following ways:

1. 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:\myProject\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

NOTE: For Unixes and Mac OS X: Use forward slash '/' as the directory separator and ':' as the path seperator, e.g., ".:/usr/local/myproject/classes:/usr/local/tomcat/lib/servlet-api.jar".

1. CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
2. > SET CLASSPATH=.;c:\myProject\classes;d:\tomcat\lib\servlet-api.jar
3. 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,
4. > java –classpath c:\myProject\classes com.abc.project1.subproject2.MyClass3

**How Classes are Found?**

(Read "*How classes are found*" at the JDK documentation's main page.)

The Java Virtual Machine (JVM) searches for and loads classes in this order:

1. *Bootstrap Classes*: include rt.jar (runtime), and other classes specified in the sun.boot.class.path system property, which could include i18n.jar (internationalization), sunrsasign.jar, jsse.jar, jce.jar, charsets.jar, and jre/classes.
2. *Extension Classes* via *Java Extension mechanism*: classes bundled as JAR file and kept in the "$JAVA\_HOME/jre/lib/ext" directory.
3. *User Classes*: located via –classpath or –cp command-line option or CLASSPATH environment variable.

Normal users need not concern about bootstrap and extension classes. User classes are found though the so-called *user class path* - a list of directories and JAR files which contain class files. The directories and JAR files in the user class path are separated with a semi-colon ';' for Windows systems, or colon ':' for UNIX systems. The user class path is kept in the System Property java.class.path. The value is obtained from:

1. The default value '.' or current working directory.
2. The value of the CLASSPATH environment variable, which overrides the default value.
3. The value of –classpath or -cp command-line option, which overrides both the default value and the CLASSPATH value.
4. The JAR files in the -jar command line option, which overrides all other values.

[How about java.lang, and classes in the same package? What is the order?]

**Java 9 Modules**

Java 9 introduces a higher level of structure called "module" on top of "package". Read "[Java 9 New Features](https://www3.ntu.edu.sg/home/ehchua/programming/java/JDK9_NewFeatures.html)".