JB0270: JAVA OOP: PACKAGES*

Richard Baldwin

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License 3.0†

Abstract

This module explains the concept of packages and provides a sample program that illustrates the concept.

1 Table of Contents

- Preface (p. 1)
  - Viewing tip (p. 1)
    - Listings (p. 2)
- Introduction (p. 2)
- Classpath environment variable (p. 2)
- Developing your own packages (p. 3)
  - The package directive (p. 4)
  - The import directive (p. 5)
  - Compiling programs with the package directive (p. 5)
  - Sample program (p. 6)
- Run the program (p. 8)
- Looking ahead (p. 8)
- Miscellaneous (p. 9)

2 Preface

This module explains the concept of packages and provides a sample program that illustrates the concept.

2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

*Version 1.2: Nov 27, 2012 8:01 pm -0600
†http://creativecommons.org/licenses/by/3.0/

2.1.1 Listings

- Listing 1 (p. 6) . File: Package00.java.
- Listing 2 (p. 7) . File Package01.java.
- Listing 3 (p. 7) . File Package02.java.
- Listing 4 (p. 8) . File: CompileAndRun.bat.

3 Introduction

Before you can understand much about packages, you will need to understand the *classpath environment variable* , so that is where I will begin the discussion.

After learning about the classpath environment variable, you will learn how to create your own packages.

4 Classpath environment variable

The purpose of the *classpath environment variable* is to tell the JVM and other Java applications *(such as the javac compiler)* where to find class files and class libraries. This includes those class files that are part of the JDK and class files that you may create yourself.

I am assuming that you already have some familiarity with the use of environment variables in your operating system. All of the discussion in this module will be based on the use of a generic form of Windows. *(By generic, I mean not tied to any particular version of Windows.)* Hopefully you will be able to translate the information to your operating system if you are using a different operating system.

> NOTE: **In a nutshell** Environment variables provide information that the operating system uses to do its job.

There are usually a fairly large number of environment variables installed on a machine at any give time. If you would like to see what kind of environment variables are currently installed on your machine, bring up a command-line prompt and enter the command **set** . This should cause the names of several environment variables, along with their settings to be displayed on your screen.

While you are at it, see if any of those items begin with **CLASSPATH=** . If so, you already have a classpath environment variable set on your machine, but it may not contain everything that you need.

I am currently using a Windows 7 operating system and no classpath environment variable is set on it. I tend to use the **-cp** switch option *(see Listing 4 (p. 8))* in the JDK to set the classpath on a temporary basis when I need it to be set.

Rather than trying to explain all of the ramifications regarding the classpath, I will simply refer you to an Oracle document on the topic titled Setting the class path [1] .

I will also refer you to Java OOP: The Guzdial-Ericson Multimedia Class Library [2] where I discuss the use of the classpath environment variable with a Java multimedia class library.

Some rules

There are some rules that you must follow when working with the classpath variable, and if you fail to do so, things simply won't work.

For example, if your class files are in a jar file, the classpath must end with the name of that jar file.

On the other hand, if the class files are not in a jar file, the classpath must end with the name of the folder that contains the class files.

Your classpath must contain a fully-qualified path name for every folder that contains class files of interest, or for every jar file of interest. The paths should begin with the letter specifying the drive and end either with the name of the jar file or the name of the folder that contains the class files. .

[1] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html
[2] http://cnx.org/content/m44148

If you followed the default JDK installation procedure and are simply compiling and executing Java programs in the current directory you probably won't need to set the classpath. By default, the system already knows *(or can figure out)* how to allow you to compile and execute programs in the current directory and how to use the JDK classes that come as part of the JDK.

However, if you are using class libraries other than the standard Java library, are saving your class files in one or more different folders, or are ready to start creating your own packages, you will need to set the classpath so that the system can find the class files in your packages.

5 Developing your own packages

One of the problems with storing all of your class files in one or two folders is that you will likely experience name conflicts between class files.

Every Java program can consist of a large number of separate classes. A class file is created for each class that is defined in your program, even if they are all combined into a single source file.

It doesn't take very many programs to create a lot of class files, and it isn't long before you find yourself using the same class names over again. If you do, you will end up overwriting class files that were previously stored in the folder.

For me, it only takes two GUI programs to get in trouble because I tend to use the same class names in every program for certain standard operations such as closing a **Frame** or processing an **ActionEvent** . For the case of the **ActionEvent** , the body of the class varies from one application to the next so it doesn't make sense to turn it into a library class.

So we need a better way to organize our class files, and the Java package provides that better way.

The Java package approach allows us to store our class files in a hierarchy of folders *(or a jar file that represents that hierarchy)* while only requiring that the classpath variable point to the top of the hierarchy. The remaining hierarchy structure is encoded into our programs using package directives and import directives.

Now here is a little jewel of information that cost me about seven hours of effort to discover when I needed it badly.

When I first started developing my own packages, I spent about seven hours trying to determine why the compiler wouldn't recognize the top-level folder in my hierarchy of package folders.

I consulted numerous books by respected authors and none of them was any help at all. I finally found the following statement in the Java documentation *(when all else fails, read the documentation)* . By the way, a good portion of that wasted seven hours was spent trying to find this information in the documentation which is voluminous.

> NOTE: **The following text was extracted directly from the JDK 1.1 documentation**
> If you want the CLASSPATH to point to class files that belong to a package, you should specify a path name that includes the path to the directory one level above the directory having the name of your package.
>
> For example, suppose you want the Java interpreter to be able to find classes in the package mypackage. If the path to the mypackage directory is C:\java\MyClasses\mypackage, you would set the CLASSPATH variable as follows:
>
> set CLASSPATH=C:\java\MyClasses

If you didn't catch the significance of this, read it again. When you are creating a classpath variable to point to a folder containing classes, it must point to the folder. However, when you are creating a classpath variable to point to your package, it must point to the folder that is one level above the directory that is the top-level folder in your package.

Once I learned that I had to cause the classpath to point to the folder immediately above the first folder in the hierarchy that I was including in my package directives, everything started working.

5.1 The package directive

So, what is the purpose of a package directive, and what does it look like?

NOTE: **Purpose of a package directive** The purpose of the package directive is to identify a particular class *(or group of classes contained in a single source file (compilation unit))* as belonging to a specific package.

This is very important information for a variety of reasons, not the least of which is the fact that the entire access control system is wrapped around the concept of a class belonging to a specific package. For example, code in one package can only access public classes in a different package.

Stated in the simplest possible terms, a package is a group of class files contained in a single folder on your hard drive.

At compile time, a class can be identified as being part of a particular package by providing a package directive at the beginning of the source code..

A package directive, if it exists, must occur at the beginning of the source code *(ignoring comments and white space)* . No text other than comments and whitespace is allowed ahead of the package directive.

If your source code file does not contain a package directive, the classes in the source code file become part of the *default package* . With the exception of the default package, all packages have names, and those names are the same as the names of the folders that contain the packages. There is a one-to-one correspondence between folder names and package names. The default package doesn't have a name.

Some examples of package directives that you will see in upcoming sample programs follow:

NOTE: **Example package directives**

```
    package Combined.Java.p1;
package Combined.Java.p2;
```

Given the following as the classpath on my hypothetical machine,

CLASSPATH=.;c:\Baldwin\JavaProg

these two package directives indicate that the class files being governed by the package directives are contained in the following folders:

NOTE:

```
    c:\Baldwin\JavaProg\Combined\Java\p1
c:\Baldwin\JavaProg\Combined\Java\p2
```

Notice how I concatenated the package directive to the classpath setting and substituted the backslash character for the period when converting from the package directive to the fully-qualified path name.

Code in one package can refer to a class in another package *(if it is otherwise accessible)* by qualifying the class name with its **package name as follows** :

NOTE:

```
    Combined.Java.p2.Access02 obj02 =
       new Combined.Java.p2.Access02();
```

Obviously, if we had to do very much of that, it would become very burdensome due to the large amount of typing required. As you already know, the *import directive* is available to allow us to specify the package containing the class just once at the beginning of the source file and then refer only to the class name for the remainder of that source file.

5.2 The import directive

This discussion will be very brief because you have been using import directives since the very first module. Therefore, you already know what they look like and how to use them.

If you are interested in the nitty-gritty details *(such as what happens when you provide two import directives that point to two different packages containing the same class file name)* , you can consult the Java Language Reference by Mark Grand, or you can download the Java language specification from Oracle's Java website.

The purpose of the import directive is to help us avoid the burdensome typing requirement (p. 4) described in the previous section when referring to classes in a different package.

An import directive makes the definitions of classes from other packages available on the basis of their file names alone.

You can have any number of import directives in a source file. However, they must occur after the package directive *(if there is one)* and before any class or interface declarations.

There are essentially two different forms of the import directive, one with and the other without a wild card character *(*)* . These two forms are illustrated in the following box.

NOTE: **Two forms of import directives**

```
    import java.awt.*
import java.awt.event.ActionEvent
```

The first import directive makes all of the class files in the **java.awt** package available for use in the code in a different package by referring only to their file names.

The second import directive makes only the class file named **ActionEvent** in the **java.awt.event** package available by referring only to the file name.

5.3 Compiling programs with package directives

So, how do you compile programs containing package directives? There are probably several ways. I am going to describe one way that I have found to be successful.

First, you must create your folder hierarchy to match the package directive that you intend to use. Remember to construct this hierarchy downward relative to the folder specified at the end of your classpath setting. If you have forgotten the critical rule (p. 3) in this respect, go back and review it.

Next, place source code files in each of the folders where you intend for the class files associated with those source code files to reside. *(After you have compiled and tested the program, you can remove the source code files if you wish.)*

You can compile the source code files individually if you want to, but that isn't necessary.

One of the source code files will contain the *controlling class* . The controlling class is the class that contains the **main** method that will be executed when you run the program from the command line using the JVM.

Make the directory containing that source code file be the current directory. *(If you don't know what the current directory is, go out and get yourself a* ***DOS For Dummies*** *book and read it.)*

Each of the source code files must contain a package directive that specifies the package that will contain the compiled versions of all the class definitions in that source code file. Using the instructions that I am giving you, that package directive will also describe the folder that contains the source code file.

Any of the source code files containing code that refers to classes in a different package must also contain the appropriate import directives, or you must use fully-qualified package names to refer to those classes.

Then use the **javac** program with your favorite options to compile the source code file containing the controlling class. This will cause all of the other source code files containing classes that are linked to the code in the controlling class, either directly or indirectly, to be compiled also. At least an attempt will be

made to compile them all. You may experience a few compiler errors if your first attempt at compilation is anything like mine.

Once you eliminate all of the compiler errors, you can test the application by using the **java** program with your favorite options to execute the controlling class.

Once you are satisfied that everything works properly, you can copy the source code files over to an archive folder and remove them from the package folders if you want to do so.

Finally, you can also convert the entire hierarchy of package folders to a jar file if you want to, and distribute it to your client. If you don't remember how to install it relative to the classpath variable, go back and review that part of the module.

Once you have reached this point, how do you execute the program. I will show you how to execute the program from the command line in the sample program in the next section. *(Actually I will encapsulate command-line commands in a batch file and execute the batch file. That is a good way to guard against typing errors.)*

5.4 Sample program

The concept of packages can get very tedious in a hurry. Let's take a look at a sample program that is designed to pull all of this together.

This application consists of three separate source files located in three different packages. Together they illustrates the use of package and import directives, along with **javac** to build a standalone Java application consisting of classes in three separate packages.

(If you want to confirm that they are really in different packages, just make one of the classes referred to by the controlling class non-public and try to compile the program.)

In other words, in this sample program, we create our own package structure and populate it with a set of cooperating class files.

A folder named **jnk** is a child of the root folder on the M-drive.
A folder named **SampleCode** is a child of the folder named **jnk** .
A folder named **Combined** is a child of the folder named **SampleCode** .
A folder named **Java** is a child of the folder named **Combined** .
Folders named **p1** and **p2** are children of the folder named **Java** .
The file named **Package00.java** , shown in Listing 1 (p. 6) is stored in the folder named **Java** .

Listing 1: File: Package00.java.

```
    /*File Package00.java Copyright 1997, R.G.Baldwin
Illustrates use  of package and import directives to
build an application consisting of classes in three
separate packages.

The output from running the program follows:

Starting Package00
Instantiate obj of public classes in different packages
Constructing Package01 object in folder p1
Constructing Package02 object in folder p2
Back in main of Package00
*********************************************************/
package Combined.Java; //package directive

//Two import directives
import Combined.Java.p1.Package01;//specific form
```

```
import Combined.Java.p2.*; //wildcard form

class Package00{
  public static void main(String[] args){ //main method
    System.out.println("Starting Package00");

    System.out.println("Instantiate obj of public " +
                        "classes in different packages");
    new Package01();//Instantiate objects of two classes
    new Package02();// in different packages.

    System.out.println("Back in main of Package00");

  }//end main method
}//End Package00 class definition.
```

The file named **Package01.java**, shown in Listing 2 (p. 7) is stored in the folder named **p1**.

Listing 2: File Package01.java.

```
    /*File Package01.java Copyright 1997, R.G.Baldwin
See discussion in file Package00.java
*********************************************************/
package Combined.Java.p1;
public class Package01 {
  public Package01(){//constructor
    System.out.println(
             "Constructing Package01 object in folder p1");
  }//end constructor
}//End Package01 class definition.
```

The file named **Package02.java**, shown in Listing 3 (p. 7) is stored in the folder named **p2**.

Listing 3: File Package02.java.

```
    /*File Package02.java Copyright 1997, R.G.Baldwin
See discussion in file Package00.java
*********************************************************/
package Combined.Java.p2;
public class Package02 {
  public Package02(){//constructor
    System.out.println(
             "Constructing Package02 object in folder p2");
  }//end constructor
}//End Package02 class definition.
```

The file named **CompileAndRun**.bat, shown in Listing 4 (p. 8) is stored in the folder named **SampleCode**.

Listing 4: File: CompileAndRun.bat.

```
echo off
rem This file is located in folder named M:\SampleCode,
rem which is Parent of folder Combined.


del Combined\Java\*.class
del Combined\Java\p1\*.class
del Combined\Java\p2\*.class

javac -cp M:\jnk\SampleCode Combined\Java\Package00.java

java -cp M:\jnk\SampleCode Combined.Java.Package00

pause
```

The controlling class named **Package00** is stored in the package named **Combined.Java** , as declared in Listing 1 (p. 6) .

The class named **Package01** is stored in the package named **Combined.Java.p1** , as declared in Listing 2 (p. 7) .

The class named **Package02** is stored in the package named **Combined.Java.p2** , as declared in Listing 3 (p. 7) .

The controlling class named **Package00** imports **Combined.Java.p1.Package01** and **Combined.Java.p2.*** , as declared in Listing 1 (p. 6) .

Code in the **main** method of the controlling class in Listing 1 (p. 6) instantiates objects of the other two classes in different packages. The constructors for those two classes announce that they are being constructed.

The two classes being instantiated are **public** . Otherwise, it would not be possible to instantiate them from outside their respective packages.

This program was tested using JDK 7 under Windows by executing the batch file named **CompileAndRun.bat** .

The classpath is set to the parent folder of the folder named **Combined** *(M:\jnk\SampleCode)* by the **-cp** switch in the file named **CompileAndRun.bat** .

The output from running the program is shown in the comments at the beginning of Listing 1 (p. 6) .

6 Run the program

I encourage you to copy the code from Listing1 (p. 6) through Listing 4 (p. 8) into a properly constructed tree of folders. Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

7 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [3]

[3] http://cnx.org/content/m44148

- Java OOP: Objects and Encapsulation [4]

8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0270: Java OOP: Packages
- File: Jb0270.htm
- Originally published: 1997
- Published at cnx.org: November 27, 2012

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[4] http://cnx.org/content/m44153