Packages

In the preceding section, the name of each example class was taken from the same name space. This means that a unique name has to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. Imagine a small group of programmers fighting over who gets to use the name "Foobar" as a class name. Or, imagine the entire Internet community arguing over ‘who first named a class Espresso’. Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the **package**. The package is both, a naming, and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Objectives

Upon completion of this topic, we will be able to:

- Define packages and understand CLASSPATH

Defining a Package

Creating a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. This is the reason why we had to worry about packages till now. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. This is the general form of the package statement: **package pkg;**

Here, pkg is the name of the package. For example, the following statement creates a package called **MyPackage**.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can be included in the same package statement. The package statement simply specifies which package the classes defined in a file belongs to. It does not exclude other classes in other files from being a part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by using a period. The general form of a multi-levelled package statement is shown here:

package pkg1 [.pkg2 [.pkg3]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as package **java.awt.image;** needs to be stored in java/awt/image, java\\awt\\image, or java:awt:image on your UNIX, Windows, or Macintosh file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Understanding CLASSPATH

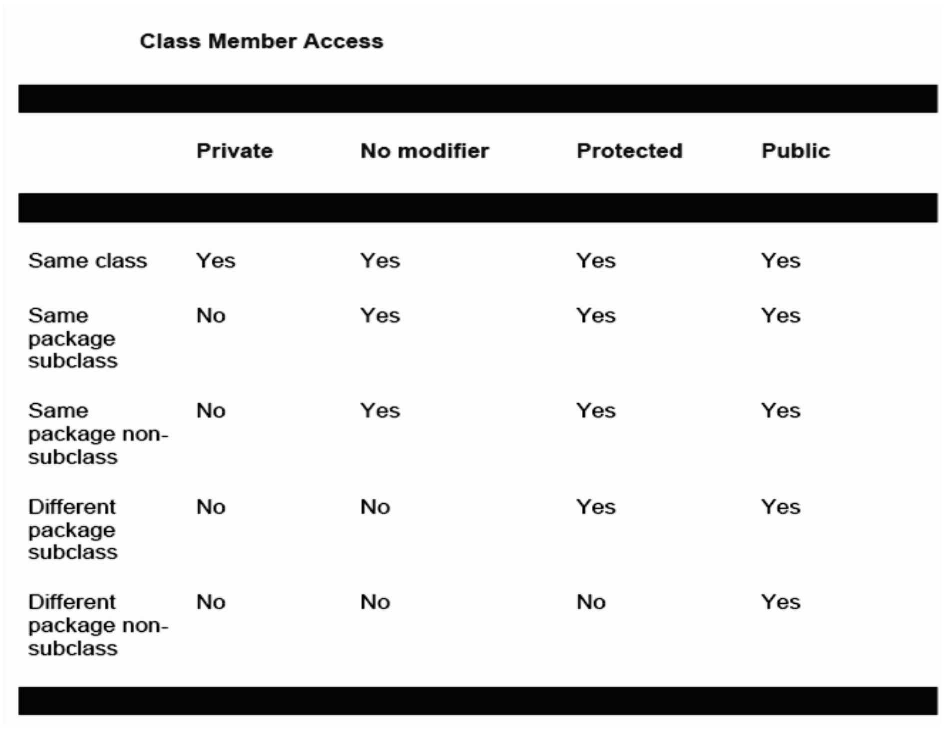
Before an example that uses a package is presented, a brief discussion of the CLASSPATH environmental variable is required. While packages solve many problems from an access control and name-space-collision perspective, they cause some curious difficulties when you compile and run programs. This is because the specific location that the Java compiler will consider as the root of any package hierarchy is controlled by CLASSPATH. Until now, you have been storing all of your classes in the same, unnamed default package. Doing so allowed you to simply compile the source code and run the Java interpreter on the result by naming the class on the command line. This worked because the default current working directory **(.)** is usually in the CLASSPATH environmental variable defined for the Java run-time system, by default. However, things are not so easy when packages are involved.

Assume that you create a class called **PackTest** in a package called test. Since your directory structure must match your packages, you can create a directory called test and put PackTest.java inside that directory. You then test the current directory and compile PackTest.java. This results in PackTest.class being stored in the test directory, as it should be. When you try to run PackTest, the Java interpreter reports an error message similar to "can't find class PackTest". This is because the class is now stored in a package called test. You can no longer refer to it simply as PackTest. You must refer to the class by enumerating its package hierarchy, separating the packages with dots. This class must now be called test.PackTest. However, if you try to use test.PackTest, you will still receive an error message similar to "can't find class test/PackTest".

The reason you still receive an error message is hidden in your CLASSPATH variable. Remember, CLASSPATH sets the top of the class hierarchy. The problem is that there's no test directory in the current working directory, because you are in the test directory, itself.

You have two choices at this point: change directories up one level and try java test.PackTest, or add the top of your development class hierarchy to the CLASSPATH environmental variable.

Table showing the Class Member Access



Class Member Access

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Summary

Here are the key takeaways:

- Package creates a family of classes.
- Package statement should be the first statement in a source code.