**Apparent Hierarchies of Packages**

At first, packages appear to be hierarchical, but they are not. For example, the Java API includes a java.awt package, a java.awt.color package, a java.awt.font package, and many others that begin with java.awt. However, the java.awt.color package, the java.awt.font package, and other java.awt.xxxx packages are *not included* in the java.awt package. The prefix java.awt (the Java Abstract Window Toolkit) is used for a number of related packages to make the relationship evident, but not to show inclusion.

Importing java.awt.\* imports all of the types in the java.awt package, but it *does not import* java.awt.color, java.awt.font, or any other java.awt.xxxx packages. If you plan to use the classes and other types in java.awt.color as well as those in java.awt, you must import both packages with all their files:

import java.awt.\*;

import java.awt.color.\*;

In reality there is no such thing as a sub-package in Java - each package is a completely separate entity, with the names being seemingly hierarchical only for convenience.

**The Static Import Statement**

There are situations where you need frequent access to static final fields (constants) and static methods from one or two classes. Prefixing the name of these classes over and over can result in cluttered code. The *static import* statement gives you a way to import the constants and static methods that you want to use so that you do not need to prefix the name of their class.

Ordinarily, to use these objects from another class, you prefix the class name, as follows.

double r = Math.cos(Math.PI \* theta);

You can use the static import statement to import the static members of java.lang.Math so that you don't need to prefix the class name, Math. The static members of Math can be imported either individually:

import **static** java.lang.Math.PI;

or as a group:

import **static** java.lang.Math.\*;

Once they have been imported, the static members can be used without qualification. For example, the previous code snippet would become:

double r = cos(PI \* theta);

**Managing Source and Class Files**

<path\_one>\**sources**\com\example\graphics\Rectangle.java

<path\_two>\**classes**\com\example\graphics\Rectangle.class

By doing this, you can 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. For example, if

<path\_two>\classes

is your class path, and the package name is

com.example.graphics,

then the compiler and JVM look for .class files in

<path\_two>\**classes**\com\example\graphics.

A class path may include several paths, 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 your class path.

**Setting the CLASSPATH System Variable**

To display the current CLASSPATH variable, use these 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 these commands:

In Windows:   C:\> set CLASSPATH=

In UNIX:      % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable, use these commands (for example):

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

In UNIX:      % CLASSPATH=/home/george/java/classes; export CLASSPATH