Module: Core Java

Session 07: Packages and Access Modifiers

# Packages

* Working with Packages
* Naming Conventions
* Using Package Members
* Referring to a Package Member by its Qualified Name
* Importing a Packages
* Apparent Hierarchies of Packages
* The Static Import Statement
* Set CLASSPATH System Variable

**Objective:**

At the end of the chapter, you will be able to:

* Create and name Packages
* Understand Naming Conventions
* Build Package Members
* Learn apparent hierarchies of Packages
* Deal with Name Ambiguities
* Prepare yourself with Static Import Statement
* Manage Source and Class Files
* Set the CLASSPATH System variable

### Using Packages

Programmers cluster groups of related types into packages for making it easier to find and use types. Avoid naming conflicts and control access.

**Definition:** A package is a grouping of related types providing access protection and name space management.

Types that are part of the JAVA platform, are members of various packages that bundle classes by function. In java.lang you will find the fundamental classes. In java.io you will find the classes for reading and writing (i.e. input and output). Your types can also be put in your packages.

Classes, interfaces, enumerations, and annotation types are referred to by types. Note that types are often referred to simply as classes and interfaces, since respectively enumerations and annotation types are special kinds of classes and interfaces.

Suppose a group of classes like circles, rectangles, lines, and points i.e. classes that represent drawing objects are written. There is also an interface named Movable which other classes implement so that you can move them with the mouse:

//in the Movable.java file

public interface Movable {

. . .

}

//in the Drawing.java file

public abstract class Drawing {

. . .

}

//in the Circle.java file

public class Circle extends Drawing implements Movable {

. . .

}

//in the Rectangle.java file

public class Rectangle extends Drawing implements Movable {

. . .

}

//in the Point.java file

public class Point extends Drawing implements Movable {

. . .

}

//in the Line.java file

public class Line extends Drawing implements Movable {

. . .

}

These classes and the interfaces in a package can be bundled for several reasons:

* It can be easily determined by you and the other programmers that these types are related.
* This enables you and other programmers to know where to find types that can provide drawings related functions.
* Names of your types will not conflict with the type names in other packages, since a new namespace is created by the packages.
* Types can be allowed within the package to have unrestricted access to one another. Access for types outside the package can still be restricted.

### Creating a Package

If you want to create a package, you need to choose a name for the package and put a package statement with that name at the top of every source file which you want to be included in the package. Note that every source file contains types like classes, interfaces, enumerations and annotation types.

The first line in the source file must be package the statement (package drawings, for instance). Each source file can have only one package statement and this applies to all the types in the file.

**Note:** When multiple types are put in a single file, only one can be public and that must have the same name as the source file. For instance, the public class Circle can be defined in the file Circle.java, public interface Movable in the Movable.java, public enum Day in the file Day.java and so on.

Note that, non-public types in the same file can be included as a public type. But try to avoid this practice everywhere, except when the non-public types are small and closely related to the public type. However, from the outside of the package, only the public type will be accessible. Note that each and every top-level, non-public types will be *package private*.

You would need six source files if you put the drawing interface and classes listed in the preceding section in a package called drawings. Consider the following:

//in the Movable.java file

package drawings;

public interface Movable {

. . .

}

//in the Drawing.java file

package drawings;

public abstract class Drawing {

. . .

}

//in the Circle.java file

package drawings;

public class Circle extends Drawing implements Movable {

. . .

}

//in the Rectangle.java file

package drawings;

public class Rectangle extends Drawing implements Movable {

. . .

}

//in the Point.java file

package drawings;

public class Point extends Drawing implements Movable {

. . .

}

//in the Line.java file

package drawings;

public class Line extends Drawing implements Movable {

. . .

}

What happens if package statement is not used? Your type ends up in an unnamed package. Where are the unnamed packages used? They are used for small or temporary applications, or when the development process is just beginning. In other instances, classes and interfaces belong in named packages.

### Naming Packages

Given the proliferation of Java programmers throughout the world, it is a pretty likely scenario that many of them would use the same name for different types. Consider the previous example where a Rectangle class is defined while it is still in the java.awt package. Note that there is already a Rectangle class. Yet both the classes are allowed by the compiler to have the same name provided they are in different packages. The package name is included by the fully qualified name of each Rectangle class. The fully qualified name of the Rectangle class is drawings.Rectangle in the drawings package and java.awt.Rectangle in the java.awt package.

But what if the same name is used by two independent programmers in their packages? That is where convention comes in.

### Naming Conventions

How to avoid the conflict with the names of classes or interfaces? Write package names in all lower cases.

Reversed Internet domain names are used by the companies to begin their package names. Consider com.example.ortel. This is a package name Ortel, created by example.com.

Use convention within a company to handle name collisions that occur within a single company. There might be different ways to do this. For instance, you can consider including the region or the project name after the company name (for example, com.company.region.package).

Remember that packages in the Java language itself begin with java. or javax.

There might be cases when the Internet domain contains a hyphen or some other special character. The package name might begin with a digit or other character that is illegal to use at the beginning of a Java name. There might be a reserved Java keyword, like “int” contained in the package name. Note that in these cases the Internet domain name may not be a valid package name. In all these cases the convention is to add an underscore:

|  |  |
| --- | --- |
| **Legalizing Package Names** | |
| **Domain Name** | **Package Name Prefix** | |
| clipart-open.org | org.clipart\_open | |
| free.fonts.int | int\_.fonts.free | |
| Poetry.7days.com | com.\_7days.poetry | |
|  |  | |

### Using Package Members

Package members are the types that comprise a package.

If you need to use a package member from outside its package, then depending on the situation, one of the following steps has to be taken:

* Referring to the member by its fully qualified name
* Importing the package number
* Importing the entire package of the member

Each one is appropriate for different situations, as explained in the sections that follow.

### Referring to a Package Member by its Qualified Name

The simple name of a package member can be used if the code you are writing is in the same package as that of the member, or if that member has been imported. The problem with this arises when you try to use a member from a different package, which has not yet been imported. In this case the member’s fully qualified name, which includes the package name, must be used. Consider the following example of the fully qualified name for the Rectangle class that was declared in the drawings package in the previous example:

drawings.Rectangle

This qualified name could have been used to create an instance of drawings.Rectangle:

drawings.Rectangle myRect = new drawings.Rectangle();

For infrequent use, qualified names are fine. However, it becomes tedious when a name is used repetitively and you have to type it every time it comes. This might also make the code a bit complicated to read. What is the alternative? Importing the member or its package and then using its simple name.

### Importing a Package Member

Put an import statement at the beginning of the file to import a specific member into the current file. Note that you need to put this statement before any type definitions and after the package statement (if there is a package statement). Again consider the example of the Rectangle class from the drawings package created. To import it, write the following:

import drawings.Rectangle;

The Rectangle class can be referred by its simple name:

Rectangle myRectangle = new Rectangle();

Note that this works well when only a few members of the drawings package is used. Remember that the entire package needs to be imported if many types from a package are used.

### Importing an Entire Package

How would you do so- i.e. import all the types that are contained in a particular package? You need to use the import statement with the asterisk (\*) wildcard character:

import drawings.\*;

Now to any class or interface in the drawings package can be referred to its simple name. Consider the following:

Circle myCircle = new Circle();

Rectangle myRectangle = new Rectangle();

As demonstrated here, you can use the asterisk in the import statement only to specify all the classes within a package. Note that you cannot use it to match a subset of the classes in a package. Consider the following example where all the classes in the drawings package beginning with A are not matched:

import drawings.A\*; //does not work

What happens is that a compiler error is generated? Generally either only a single package member or an entire package is imported with the import statement.

**Note:** You can use a less common form of import to import the public nested classes of an enclosing class. Consider the case of the drawings.Rectangle class again. If this class had contained useful nested classes, like Rectangle.DoubleWide and Rectangle.Square, you could use the following two statements to import both the Rectangle and its nested classes:

import drawings.Rectangle;

import drawings.Rectangle.\*;

Note that, here the Rectangle is not imported by the second import statement.

Static import statement is one more among the less common forms of import.

To make things easy three entire packages are imported automatically by the Java compiler for each source file:

1. The package with no name,
2. The java.lang package,
3. The current package (the package for the current file).

### Apparent Hierarchies of Packages

Consider the case of the Java API including a java.awt package, a java.awt.color package, a java.awt.font package, and many others that begin with java.awt. You might have thought that there is a hierarchy among packages, but they are actually not so. Notice that the java.awt package doe not include the java.awt.color package, the java.awt.font package, and whatever other java.awt.xxxx packages are there. The prefix java.awt (the Java Abstract Window Toolkit) is used only for a number of related packages to make the relationship evident, but not to show inclusion.

What happens is that when you import the java.awt.\* that imports all of the types in the java.awt package. However, java.awt.color, java.awt.font, or any other java.awt.xxxx packages are not imported with this. You need to import both the java.awt.color and java.awt packages with all their files if you want use the classes and other types in both of them. You need to write the following:

import java.awt.\*;

import java.awt.color.\*;

### Name Ambiguities

What happens if the name of a member in one package is shared with a member in another package and both packages are imported? The answer is that then each member needs to be referred to by its qualified name. Consider the case of when the drawings package defines a class named Rectangle. A Rectangle class is also contained by the java.awt package. Now, if you import both the drawings and the java.awt, the following becomes ambiguous.

Rectangle rect;

How do you indicate, which Rectangle class you exactly want in this kind of a situation? You need to use the fully qualified name of the member:

drawings.Rectangle rect;

### The Static Import Statement

Consider a situation when frequent access to both static final fields (constants) and static methods from one or two classes are needed. Your code can become a bit cluttered if you prefix the name of these classes repetitively. Avoid prefixing and use the static import statement to import the constants as well as the static methods you need.

The PI constant and multiple static methods, like methods for calculating sines, cosines, tangents, square roots, maxima, minima, exponents, and many others are defined by the java.lang.Math class. Consider the following example:

public static final double PI 3.141592653589793

public static double cos(double a)

Typically, you will need to prefix the class name to use these objects from another class

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

The static import statement can be used to import the static members of java.lang.Math. Thus the class name, Math need not be prefixed, since the static members of Math can be imported either as individually or as a group:

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

as a group:

or as a group:

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

You can use the static members without qualification after you have imported them. Consider the previous code snippet. It will become:

double r = cos(PI \* theta);

You can write own classes containing constants and frequently used static methods and then use the static import statement. Consider the following:

import **static** mypackage.MyConstants.\*;

**Note:** You need to be cautious while using static import as overuse of it might result in codes thar are difficult for both reading and maintaining. For instance, readers will not know that a particular static object is defined by which class. The only way static import makes a more readable code is when it removes the class name repetition.

### Managing Source and Class Files

To manage source and file classes, many implementations on the Java platform rely on hierarchical file systems. However, it is not required by the Java Language specification. The strategy is to put the source code for an interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is .java. For instance:

// in the Rectangle.java file

package drawings;

public class Rectangle() {

. . .

}

After this, you need to put the source file in a directory whose name reflects the name of the package to which the type belongs:

.....\drawings\Rectangle.java

Assuming that the Microsoft Windows file name separator is backslash or the forward slash for Unix, the qualified name of the package member and the path name to the file are parallel.

|  |  |
| --- | --- |
| **class name** | drawings.Rectangle |
| **pathname to file** | drawings\Rectangle.java |

A company uses its reversed Internet domain name for its package names by using convention. In the example of the company whose Internet domain name was example.com, all its package names will be preceded by com.example. Each component of the package name is correspondent to a subdirectory. Suppose the company had a com.example.drawings package that contained a Rectangle.java source file. Now it will be contained in a series of subdirectories:

....\com\example\drawings\Rectangle.java

The compiler creates a different output file for each type when a source file is compiled. The name of the type is the base name of the output file while the extension is .class. If the source class is like, say:

// in the Rectangle.java file

package com.example.drawings;

public class Rectangle{

. . .

}

class Helper{

. . .

}

then the location of the compiled files will be at:

<path to the parent directory of the output files>\com\example\drawings\Rectangle.class

<path to the parent directory of the output files>\com\example\drawings\Helper.class

The compiled.class files, like the .java source files, should be in a series of directories, that reflect the package name. The path to the .class files does not necessarily need to be the same as the path to the .java source files. The source and class directories can be arranged separately:

<path\_one>\sources\com\example\drawings\Rectangle.java

<path\_two>\classes\com\example\drawings\Rectangle.class

The advantage of this is that the classes directory can be given to other programmers without revealing the sources. Also the Java Virtual Machine can find all the types used in your program if it manages source and class files in this manner.

The class path is the full path to the classes directory –i.e. path\_two>\classes. This is set with the set CLASSPATH system variable. The path to your .class files is constructed by both the compiler and the JVM. They do this by adding the package name to the class path. For instance, if

<path\_two>\classes

is your class path, and the package name is

com\example\drawings,

then the compiler and JVM look for .class files in

<path\_two>\classes\com\example\drawings.

Several paths, separated by semicolon (in Windows) or a colon (in Unix) may be included in a classpath. The current directory and the JAR file containing the Java platform classes are searched by the compiler and the JVM by default. The advantage of this is that these directories automatically come in your program classpath.

### 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

### Summary

* To create a package for a type, put a package statement as the first statement in the source file that contains the type (class, interface, enumeration, or annotation type).
* To use a public type, which is in a different package, you have three choices: (1) use the fully qualified name of the type, (2) import the type, or (3) import the entire package of which the type is a member.
* The path names for a package's source and class files mirror the name of the package.
* You might have to set your CLASSPATH so that the compiler and the JVM can find the .class files for your types.