Packages

Creating and Using Packages

- To make types easier to find and use
- To avoid naming conflicts
- To control access...

programmers bundle groups of related types into packages.

Definition: A *package* is a grouping of related types providing access protection and name space management. Note that *types* refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so *types* are often referred to in this lesson simply as *classes* and *interfaces*.

The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in java.lang, classes for reading and writing (input and output) are in java.io, etc. Types go in packages aswell.

Example:
A group of classes that represent graphic objects, such as circles, rectangles, lines, and points.

You also write an interface, Draggable, that classes implement if they can be dragged with the mouse.

```
//in the Draggable.java file
public interface Draggable {
    ...
}

//in the Graphic.java file
public abstract class Graphic {
    ...
}

//in the Circle.java file
public class Circle extends Graphic
    implements Draggable {
    . . .
}

//in the Rectangle.java file
public class Rectangle extends Graphic
    implements Draggable {
    . . .
}

//in the Point.java file
public class Point extends Graphic
    implements Draggable {
    . . .
}

//in the Line.java file
public class Line extends Graphic
    implements Draggable {
    . . .
}
```

You should bundle these classes and the interface in a package for 7 reasons, including:

- You and other programmers can easily determine that these types are related.
- You and other programmers know where to find types that can provide graphics-related functions.
- The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Creating a Package

To create a package, you choose a name for the package and put a package statement with that name at the top of every source file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, package graphics;) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

---

note

If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file. For example, you can define public class Circle in the file Circle.java, define public interface Draggable in the file Draggable.java, define public enum Day in the file Day.java, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be *package private*.

If you put the graphics interface and classes listed in the preceding section in a package called graphics, you would need six source files, like this:

```
//in the Draggable.java file
package graphics;
public interface Draggable {
    . . .
}

//in the Graphic.java file
package graphics;
public abstract class Graphic {
    . . .
}

//in the Circle.java file
package graphics;
public class Circle extends Graphic
    implements Draggable {
    . . .
}

//in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic
    implements Draggable {
    . . .
}

//in the Point.java file
package graphics;
public class Point extends Graphic
    implements Draggable {
    . . .
}

//in the Line.java file
package graphics;
public class Line extends Graphic
    implements Draggable {
    . . .
}
```

If you do not use a *package* statement, your type ends up in an unnamed package.

An unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

Naming a Package

With programmers worldwide writing classes and interfaces using the Java programming language, it is likely that many programmers will use the same name for different types. Shown in previous example:

It defines a Rectangle class when there is already a Rectangle class in the java.awt package. Still, the compiler allows both classes to have the same name if they are in different packages. The fully qualified name of each Rectangle class includes the package name. That is, the fully qualified name of the Rectangle class in the graphics package is graphics.Rectangle, and the fully qualified name of the Rectangle class in the java.awt package is java.awt.Rectangle.

This works well unless two independent programmers use the same name for their packages. What prevents this problem? **Convention**.

Naming Conventions

Package names are written in all lower case to avoid conflict with the names of classes or interfaces.

Companies use their reversed Internet domain name to begin their package names

Example:

`com.example.mypackage` for a package named `mypackage` created by a programmer at `example.com`

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name

Example: `com.example.region.mypackage`

Packages in the Java language itself begin with java. or javax.

In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int". In this event, the suggested convention is to add an underscore.

Example

Legalizing Package Names

Domain Name	Package Name Prefix
hyphenated-name.example.org	org.example.hyphenated_name
example.int	int_.example
123name.example.com	com.example._123name

Using Package Members

The types that comprise a package are known as the *package members.*

Using a `public` package member from outside its package

- Refer to the member by its fully qualified name
- Import the package member
- Import the member's entire package

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

Referring to a Package Member by Its Qualified Name

So far, most of the examples in this tutorial have referred to types by their simple names, such as Rectangle and StackOfInts. You can use a package member's simple name if the code you are writing is in the same package as that member or if that member has been imported.

However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's fully qualified name, which includes the package name. Here is the fully qualified name for the Rectangle class declared in the graphics package in the previous example.

```
graphics.Rectangle
```

You could use this qualified name to create an instance of graphics.Rectangle:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Qualified names are all right for infrequent use. When a name is used repetitively, however, typing the name repeatedly becomes tedious and the code becomes difficult to read. As an alternative, you can import the member or its package and then use its simple name.

Importing a Package Member

To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one. Here's how you would import the Rectangle class from the graphics package created in the previous section.

```
import graphics.Rectangle;
```

Now you can refer to the Rectangle class by its simple name.

```
Rectangle myRectangle = new Rectangle();
```

This approach works well if you use just a few members from the graphics package. But if you use many types from a package, you should import the entire package.

Using Package Members

Importing an Entire Package

To import all the types contained in a particular package, use the import statement with the asterisk (*) wildcard character.

```
import graphics.*;
```

Now you can refer to any class or interface in the graphics package by its simple name.

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

The asterisk in the import statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package.
Example: the following does not match all the classes in the graphics package that begin with A.

```
// does not work
import graphics.A*;
```

Instead, it generates a compiler error. With the import statement, you generally import only a single package member or an entire package.

---

note

Another, less common form of import allows you to import the public nested classes of an enclosing class. For example, if the graphics.Rectangle class contained useful nested classes, such as Rectangle.DoubleWide and Rectangle.Square, you could import Rectangle and its nested classes by using the following two statements.

```
import graphics.Rectangle;
import graphics.Rectangle.*;
```

Be aware that the second import statement will **not** import Rectangle.

Another less common form of import, the static import statement, will be discussed at the end of this section.

For convenience, the Java compiler automatically imports two entire packages for each source file: (1) the java.lang package and (2) the current package (the package for the current file).

Using Package Members

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.*;
```

Name Ambiguities

If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. For example, the graphics package defined a class named Rectangle. The java.awt package also contains a Rectangle class. If both graphics and java.awt have been imported, the following is ambiguous.

```
Rectangle rect;
```

In such a situation, you have to use the member's fully qualified name to indicate exactly which Rectangle class you want.

Example

```
graphics.Rectangle rect;
```

Using Package Members

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.

The java.lang.Math class defines the PI constant and many static methods, including methods for calculating sines, cosines, tangents, square roots, maxima, minima, exponents, and many more. For example,

```
public static final double PI
    = 3.141592653589793;
public static double cos(double a)
{
    ...
}
```

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);
```

Obviously, you can write your own classes that contain constants and static methods that you use frequently, and then use the static import statement. Example:

```
import static mypackage.MyConstants.*;
```

note

Use static import very sparingly. Overusing static import can result in code that is difficult to read and maintain, because readers of the code won't know which class defines a particular static object. Used properly, static import makes code more readable by removing class name repetition.

Managing Source & Class Files

Strategy

Many implementations of the Java platform rely on hierarchical file systems to manage source and class files, although The Java Language Specification does not require this.

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is .java.
Example:

```
//in the Rectangle.java file
package graphics;
public class Rectangle {
   ...
}
```

Then, put the source file in a directory whose name reflects the name of the package to which the type belongs: `.....\graphics\Rectangle.java`

The qualified name of the package member and the path name to the file are parallel, assuming the Microsoft Windows file name separator backslash (for UNIX, use the forward slash).

- **class name** - graphics.Rectangle
- **pathname to file** - graphics\Rectangle.java

As you should recall, by convention a company uses its reversed Internet domain name for its package names. The Example company, whose Internet domain name is example.com, would precede all its package names with com.example. Each component of the package name corresponds to a subdirectory. So, if the Example company had a com.example.graphics package that contained a Rectangle.java source file, it would be contained in a series of subdirectories like this: `....\com\example\graphics\Rectangle.java`

When you compile a source file, the compiler creates a different output file for each type defined in it.
The base name of the output file is the name of the type, and its extension is .class.
For example, if the source file is like this

```
//in the Rectangle.java file
package com.example.graphics;
public class Rectangle {
      . . .
}

class Helper{
      . . .
}
```

then the compiled files will be located at:

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

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does **not** have to be the **same** as the path to the .java source files.

You can arrange your source & class directories separately

```
<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.

Example:

`<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.

Managing Source & Class Files

Setting the CLASSPATH System Variable

commands

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:

```
In Windows:    C:\> set CLASSPATH=
In UNIX:       % unset CLASSPATH; export CLASSPATH
```

To set the CLASSPATH variable:

```
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 that's 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.

Question 1:

Assume that you have written some classes.
Belatedly, you decide that they should be split into three packages, as listed in the table below.
Furthermore, assume that the classes are currently in the default package (they have no package statements).

Destination Packages

Package Name	Class Name
mygame.server	Server
mygame.shared	Utilities
mygame.client	Client

a. What line of code will you need to add to each source file to put each class in the right package?
Answer 1a: The first line of each file must specify the package:

```
In Client.java add:
package mygame.client;
In Server.java add:
package mygame.server;:
In Utilities.java add:
package mygame.shared;
```

b. To adhere to the directory structure, you will need to create some subdirectories in your development directory, and put source files in the correct subdirectories. What subdirectories must you create? Which subdirectory does each source file go in?
Answer 1b: Within the mygame directory, you need to create three subdirectories: client, server, and shared.

```
In mygame/client/ place:
Client.java
In mygame/server/ place:
Server.java
In mygame/shared/ place:
Utilities.java
```

c. Do you think you'll need to make any other changes to the source files to make them compile correctly? If so, what?
Answer 1c: Yes, you need to add import statements. Client.java and Server.java need to import the Utilities class, which they can do in one of two ways:

```
import mygame.shared.*;
       --or--
import mygame.shared.Utilities;
Also, Server.java needs to import the Client class:

import mygame.client.Client;
```