Packages

Understanding Packages

To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of rel

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

The types that are part of the Java platform are members of various packages that bundle classes by function: fundamer

Suppose you write a group of classes that represent graphic objects, such as circles, rectangles, lines, and points. You als

```
//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 several reasons, including the following:

- 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 will not conflict with the type names in other packages because the package creates a new r
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types o

Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put

The package statement (for example, `package graphics;`) must be the first line in the source file. There can be only one

> *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 sour*
>
> *You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public type*

If you put the graphics interface and classes listed in the preceding section in a package called `graphics`, you would need

```
1  //in the Draggable.java file
2  package graphics;
3  public interface Draggable {
4      . . .
5  }
6
7  //in the Graphic.java file
8  package graphics;
9  public abstract class Graphic {
10     . . .
11 }
12
13 //in the Circle.java file
14 package graphics;
15 public class Circle extends Graphic
16     implements Draggable {
17     . . .
18 }
19
20 //in the Rectangle.java file
21 package graphics;
22 public class Rectangle extends Graphic
23     implements Draggable {
24     . . .
25 }
26
27 //in the Point.java file
28 package graphics;
29 public class Point extends Graphic
30     implements Draggable {
31     . . .
32 }
33
34 //in the Line.java file
35 package graphics;
36 public class Line extends Graphic
37     implements Draggable {
38     . . .
39 }
```

If you do not use a `package` statement, your type ends up in an unnamed package. Generally speaking, an unnamed packa

Naming a Package and Naming Conventions

With programmers worldwide writing classes and interfaces using the Java programming language, it is likely that many

This works well unless two independent programmers use the same name for their packages. What prevents this problen

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—for example, `com.example.mypackag`

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by in

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

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.

To use a `public` package member from outside its package, you must do one of the following:

- 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 `StackOf`

However, if you are trying to use a member from a different package and that package has not been imported, you must u

```
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 be

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 defi

```
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 pa

Importing an Entire Package

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

```
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 canno

```
// 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 a

> *Note: Another, less common form of import allows you to import the public nested classes of an enclosing class. For exa*

```
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).

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 `ja`

Importing `java.awt.*` imports all of the types in the `java.awt` package, but it does not import `java.awt.color`, `java.awt`

```
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 mu

```
Rectangle rect;
```

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

```
graphics.Rectangle rect;
```

The Static Import Statement

There are situations where you need frequent access to static final fields (constants) and static methods from one or two

The `java.lang.Math` class defines the `PI` constant and many static methods, including methods for calculating sines, co

```
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 prefi

```
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 snip

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

Obviously, you can write your own classes that contain constants and static methods that you use frequently, and then us

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

> *Note: Use static import very sparingly. Overusing static import can result in code that is difficult to read and maintain, be*

Wrapping up Packages

To create a package for a type, put a `package` statement as the first statement in the source file that contains the type (cl

To use a public type that 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.

Last update: *September 14, 2021*

Back to Tutorial List