Java Programming 1

CHAPTER 5

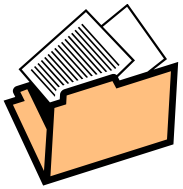
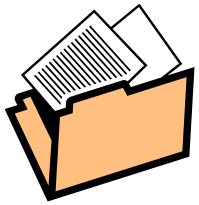
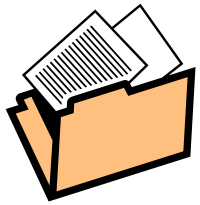
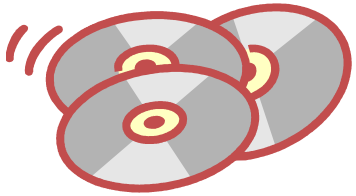
Packages

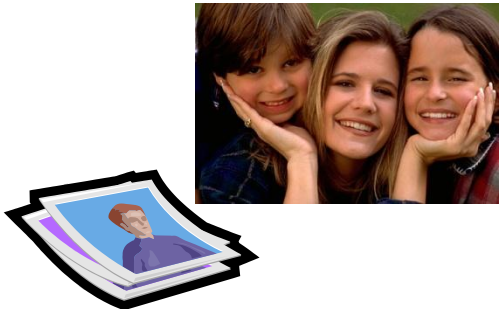
Contents

- What are Packages
- Creating and Using Packages
- Naming Conventions
- Using Package Members
- Apparent Hierarchies of Packages
- Managing Source and Class Files
- Example
- Summary of Creating and Using Packages

What is a Package?

- A package is a namespace that organizes a set of related classes and interfaces.
- Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another.

What are Packages

- In order to:
  - make types easier to find and use,
  - avoid naming conflicts, and
  - control access,

Programmers bundle groups of related types into packages.

What are 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 here as classes and interfaces.

What are Packages

- 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, and so on.
- You can put your types in packages too.

Creating and Using Packages

- Suppose you write 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 {
    . . .
}  // see the next slide
```

Creating and Using Packages

```
//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 {
    . . .
}
```

- Why should I bundle these classes and the interface in a package ?

Creating and Using Packages

- 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 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 and Using Packages

- To create a package:
  1. Choose a name for the package and
  2. 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.
  3. The package statement must be the first line in the source file.
  4. There can be only one package statement in each source file, and it applies to all types in the file.
- **Remember that:** 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.

Creating and Using Packages

- 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 {
    . . .
}   // see the next slide
```

Creating and Using Packages

```
//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 {
    . . .
}
```

Naming Conventions

- Package names are written in all lowercase 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.orion for a package named orion created by a programmer at example.com.
- Packages in the Java language itself begin with java. or javax.
- **Note:** using name conventions will help to avoid name conflict of packages written by different programmers.

Using Package Members

- To use a public package member from outside its package, you must do one of the following:
  1. Refer to the member by its fully qualified name
  2. Import the package member
  3. Import the member's entire package

Using Package Members

- Refer to the member by its fully qualified name
- Example:
  - Use just name e.g. Rectangle if your class in the graphics package (same class of Rectangle)
  - Use full name e.g. graphics.Rectangle if your class outside the graphics package

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

Using Package Members

- Import the package member
- Example:
  - import graphics.Rectangle;
  - Now you can refer to the Rectangle class by its simple name.

Rectangle myRectangle = new Rectangle();

Using Package Members

- Import the member's entire package
- Example:
  - 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();
```

Packages Imported Automatically

- The Java compiler automatically imports three entire packages for each source file (you do not need to specify the import statement):
  - the package with no name,
  - the java.lang package, and
  - the current package (the package for the current file).
- Members of these packages can be referred by their simple names.

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.
- 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.
- The following statement imports all of the types in the java.awt package, but it *does not import* java.awt.color, java.awt.font:
  - import java.awt.*;
- To use the members of the aforementioned packages, you have to write:

```
import java.awt.*;
import java.awt.color.*;
Import java.awt.font.*;
```

Managing Source and Class Files

- Many implementations of the Java platform rely on hierarchical file systems to manage source and class files.
- The strategy is as follows:
  - Put the source code for a class or interface in a text file whose extension is .java

```
// 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
```

Simple Example

```
// file ClassOne.java in the directory
//  /home/s111111/java/Ex08/demopackage
package demopackage;
public class  ClassOne {
   public void methodClassOne() {
      System.out.println("methodClassOne");
   }
}
// file ClassTwo.java in the directory
//  /home/s111111/java/Ex08/demopackage
package demopackage;
public class  ClassTwo {
   public void methodClassTwo() {
      System.out.println("methodClassTwo");
   }
}
```

- Compilation:

  javac *.java

```
// file UsageDemoPackage.java in
// the directory
//  /home/s111111/java/Ex08/
import demopackage.*;
class UsageDemoPackage {
  public static void main(String[] args) {
   ClassOne v1 = new ClassOne();
   ClassTwo v2 = new ClassTwo();
   v1.methodClassOne();
   v2.methodClassTwo();
  }
}
```

- Compilation:

  javac UsageDemoPackage.java
- Run:

  java UsageDemoPackage

Managing Source and Class Files

- The qualified name of the package member and the path name to the file are parallel:
  - **class name:** graphics.Rectangle
  - **pathname to file:** graphics\Rectangle.java
- 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

Managing Source and Class Files

- When you compile a source file, the compiler creates a different output file (its extension is .class) for each type defined in it
- the compiled files will be located at:

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

class Helper{
    . . .
}
```

<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

Managing Source and Class Files

- The full path to the classes directory is called the *class path*
- You may set the class path with the CLASSPATH system variable
- 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.

The class path is:
<path_two>/classes

Package name is:
com/example/graphics,

The compiler and JVM look for .class files in
<path_two>/classes/com/example/graphics

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 of Creating and Using Packages

- To create a package for a type, put a package statement as the first statement in the source file that contains the type (class or interface).
- To use a public type that is in a different package, you have three choices:
  - use the fully qualified name of the type,
  - import the type, or
  - 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.