**Packages**

* **Creating and Using Packages**
  + A package is a grouping of related types providing access protection and namespace management.
  + 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.
  + You should bundle these classes and interfaces 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 type names in other packages because the packages create a new namespace.
    - You can allow types within packages 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 the package statement with that name at the top of every source file that contains the types that you want to include in the package.
  + The package statement 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.
  + If you do not use a package statement, your type ends up in an unnamed package.
* **Naming a Package**
  + 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.mypackage
  + 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’s illegal to use as the beginning of java name.
  + Or if the package name contains reserved java keywords such as int.
  + In this event the suggested convention is to add an underscore.

|  |  |
| --- | --- |
| **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.
  + 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.
  + **Referring to a Package Member by its Qualified Name**
    - 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.
    - You could use this qualified name to create an instance of graphics.Rectangle

*graphics.Rectangle myReact = new graphics.Rectangle();*

* + **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 package statement.

*import graphics.Rectangle;*

* + - Now you can refer to the rectangle class by its simple name.

*Rectangle myRectangle = new Rectangle();*

* + **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();*

* + - It cannot be used to match a subset of the classes in a package.

import graphics.A\*; - It does not work

* + - For convenience, java compiler automatically imports two entire packages for each source file.
      * The java.lang package
      * The current package
  + **Apparent Hierarchies of Package**
    - 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.xxx packages.
    - If you plan to use the classes and other types 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.
    - You have to use the member’s fully qualified name to indicate exactly with Rectangle class you want.

graphics.Rectangle rect;

* + **The static Import Statement**
    - There are situations where you need frequent access to static final fields 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.

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.
    - 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.

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

* **Managing Source and Class files**
  + 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 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.
    - * Class name – graphics.Rectangle
      * Pathname to file – 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 file, the compiled .class files should be in a *serious of directories* that reflect the package name.
  + However the path to the .class files *does not have to be the same* as path to the .java source files.
  + You can arrange your source and class *directories separately*
    - *<path\_one>\sources\com\example\graphics\Rectangle.java*
    - *<path\_two>\classes\com\example\graphics\Rectangle.class*
  + The full path of the classes directory, <path\_two>\classes, is called the *class path*, and is set with *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
    - <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 path, separated by a semicolon(windows) or colon(Unix)
  + **Setting the CLASSPATH System Variable :**
    - To *display* the current CLASSPATH variable, use these commands in Windows and Unix
      * *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
      * *In windows: C:\> set CLASSPATH=C:\users\regu\java\classes*
      * *In Unix : % CLASSPATH=/home/regu/java/classes; export CLASSPATH*