Package	**Creating and Using Packages in Java**	**Managing Source and Class Files In Java**	**The Static Import Statement**

Package

A namespace in computer science (sometimes also called a **name scope**), is an abstract container or environment created to hold a logical grouping of unique identifiers or symbols (i.e. names). A means of grouping logically related identifiers into corresponding namespaces, thereby making the system more modular

A Java **package** organizes related Java classes, interfaces and other types into namespaces.

All classes and other types of Java belong to a package, although that package need not be explicitly named. Code from other packages is accessed by prefixing the package name before the appropriate identifier, for example class String in package java.lang can be referred to as java.lang.String (this is known as the **fully qualified class name**).

Creating and Using Packages in Java

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

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 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.

If you put the graphics interface and classes 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 Rectangle.java file
package graphics;
public class Rectangle extends Graphic
   implements Draggable {
   . . .
}
//in the Point.java file
package graphics;
public class Point extends Graphic
```

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

In case we don’t use the package statement in the program, the classes defined in that program are considered by the Java Virtual Machine as classes belonging to the “default package”.

Going by the notion that every package is a directory, the current working directory is understood by the Java Virtual Machine as the default package.

- **Naming Conventions**

  - Package names are written in all lower case.
  - Companies use their reversed Internet domain name to begin their package names—for example, com.example.mypackage for a package named mypackage created by a programmer at example.com.
  - Packages in the Java language itself begin with java. or javax.

- **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 below.

- **Referring to a Package Member by Its Qualified Name**

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.

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

Fully qualified name

When a name is used repetitively, typing the name repeatedly becomes tedious and the code becomes difficult to read. So 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.

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();

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

Compiler error - you generally import only a single package member or an entire package.

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).

?

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.*;

Conceptually you can think of packages as being similar to different folders on your computer.

Subpackages: Packages that are inside another package are the **subpackages**. These are not imported by default when you import the parent package, they have to import explicitly.

Managing Source and Class Files in Java

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**.

- Package names and directory structure are closely related.
- For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is sub-directory of *staff* and *staff* is sub-directory of *college*.

- Also, the directory *college* is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH so that the Java compiler and runtime can locate it quickly. The idea is to make sure that classes are easy to locate.

Following are steps to **compile and run java program** with a package specifier :

1. Put the source code for a class, interface, enumeration, or annotation type in a text file with .java extension. For example:

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

2. 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

3. 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

4. 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 and class directories separately, as:

   ```
   <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. For example, if

<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.

Example of compilation java files with package specifier

Suppose following java source files are created in the path "src/com/heritage/cse/Person.java" and "src/com/heritage/cse/Start.java"

Person.java

```
package com.heritage.cse;
public class Person {
……
}
```

Start.java

```
package com.heritage.cse;
public class Start {
……..
}
```

Steps to compile and run java programs from command line:

"javac" is the java compiler available in bin folder of the jdk. During compilation we need to provide –classpath too because other classes may be referred in this class.

```
D:\cse>javac –d "classes"  "src/com/heritage/cse/Person.java"
D:\cse> javac –d "classes"  -classpath "classes"
                            "src/com/heritage/cse/Start.java"
D:\cse> java -classpath "classes"  com.heritage.cse.Start
```

- While running the program, we have to again specify that where all the class files exist with the help of parameter "**-classpath**"
- Meaning of parameters :

-classpath path : Specify where to find user class files. This class path overrides the user class path in the CLASSPATH environment variable. If neither CLASSPATH, -cp nor -classpath is specified, the user class path consists of the current directory.

-d directory : Set the destination directory for class files. The directory must already exist, javac will not create it.

If a class is part of a package, javac puts the class file in a subdirectory reflecting the package name, creating directories as needed. For example, if you specify -d C:\myclasses and the class is called com.mypackage.MyClass, then the class file is called C:\myclasses\com\mypackage\MyClass.class.
If -d is not specified, javac puts each class files in the same directory as the source file from which it was generated. Directory specified by -d is not automatically added to your user class path.

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
```

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. For example,

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

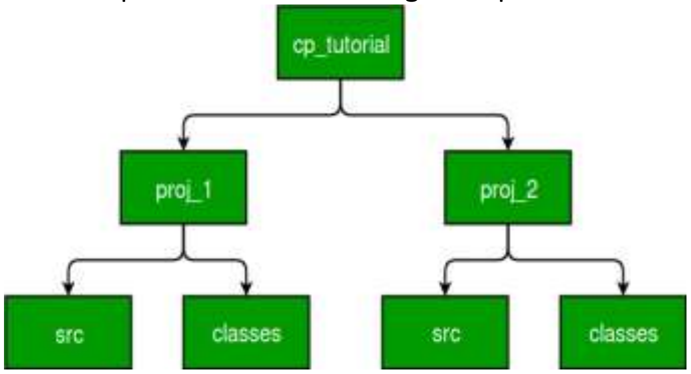
Note: Use static import very carefully. Overusing static import can result in code that is difficult to read and maintain, because readers of the code won't know which class defines a particular static object. Used properly, static import makes code more readable by removing class name repetition.

How to run java class file which is in different directory?

- Classpath is the location from where jvm starts execution of a program.
- The classpath tells Java where to look in the filesystem for files defining these classes. Variables and methods which are accessible and available at classpath are known as classpath variables. By default JVM always access the classpath classes while executing a program. JVM always go into the deep of classpath to search a class or resource.
- import keyword is used in Java to import classes from current project's classpath. You can import classes from different packages but from same classpath. It is to be remembered that packaging of a class starts from classpath. Suppose you have directory structure as follows:
  a > b > c > d > class A
  and your classpath starts from c, **then your class should be in package d not in a.b.c.d**.

Using classpath -cp option

- import keyword can import classes from current classpath, outside the classpath import can't be used. Now suppose you already have a project in which you have used some utility classes, which you need in your second project also. Then in this situation import keyword doesn't work because your first project is at another classpath. In that case you can use **-cp** command (stands for classpath)while compiling and executing your program.
- Let's proceed with following example. Create a directory structure as shown in figure below.



Here you have 2 projects proj1 and proj2. proj1 contains src and classes. In src directory, we will keep .java files that is source files and in classes directory we will keep .classes files that is files generated after compiling the project.

Following are the steps to run java class file which is in different directory:

Step 1 (Create utility class): Create A.java in src directory containing following code.

```
//java utility class
public class A
{   public void test()   {     System.out.println("Test() method of class A");   }
}
```

Step 2 (Compile utility class): Open terminal at proj1 location and execute following commands.

```
cp_tutorial/proj1>cd src              cp_tutorial/proj1/src>javac -d ../classes A.java
```

-d option: It is used to store the output to different directory. If we don't use this option then class file will be created in src directory. After -d option we provide the location of directory in which class files should be stored.

Step 3 (Check whether A.java is successfully compiled): Check in classes directory of proj1 whether class file is created or not. It will be certainly Yes if your program was Compiled successfully.

Step 4 (Write main class and compile it): Move to your proj2 directory. Here are also 2 directories for the same reasons. Create MainClass.java in src directory having following content and try to compile it.

```
//java class to execute program
public class MainClass{
   public static void main(String[] args){
      System.out.println("In main class");
      A a1 = new A();
      a1.test();
   }
}
```

```
cp_tutorial/proj2>cd src
cp_tutorial/proj2/src>javac -d ../classes MainClass.java
MainClass.java:4: error: cannot find symbol
            A a1 = new A();
location: class MainClass
```

You see, there is a compile time error that symbol A is not found. If, we want to use class A of proj1 then we have to use -cp option to include proj1's resources as shown in next step.

Step 5 (Compile with -cp option):

```
cp_tutorial/proj2>cd src
cp_tutorial/proj2/src>javac -d ../classes -cp  ../../proj1/classes MainClass.java
```

Now, your code will be compiled successfully and MainClass.class is created in classes directory. -cp stands for classpath and it includes the path given with current classpath and once it is included jvm recognizes the symbol A that it is a class and compiles the file successfully.

Step 6 (Execute the program): Try excuting the program.

```
cp_tutorial/proj2/src>cd ../classes
cp_tutorial/proj2/classes>java MainClass
```

Here too we got an error that class A is not found, because JVM is unable to locate class A in other project.

Step 7 (Execute with -cp option): We have to again provide the path of class A.

```
cp_tutorial/proj2/classes>java -cp ../../proj1/classes; MainClass
```

Now, you have successfuly run your program. Don't forget to include ; after provided classpath.