Packages

Packages

Basic Java Packages

One of the main features of OOP is its ability to reuse the code already created. One way of achieving this is by using the classes from other programs, without physically copying them into the program you are currently +creating. This concept in Java is called as packages, while in other languages is called as class libraries. Packages are a way of grouping a variety of classes and/or interfaces together. The grouping is done according to the functionality. Therefore we can say packages act as containers for classes. By organizing our classes into packages we can achieve following benefits:

The classes of other programs contained in the packages can be easily used.
In packages, classes can be unique compared with classes in other packages i.e. two classes in two different packages can have the same name. They may be referred by their full name comprising of the package name and the class name.
Packages provide a way to hide classes.
Packages also provide a way for separating design from coding.

In Java packages are classified into two types:

1. Java System packages (examples below)
2. **User-defined packages (http://sites.google.com/site/introtojava35a/course-content/module-5/user-defined-packages)** (next section)

Java System packages: Java system provides a large number of classes grouped into different packages according to the functionality. In a Java program, we use the packages available with the Java system. Here are few commonly used packages:

Java Platform Package	**Description**
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images
java.beans	Contains classes related to Java Beans development

java.io	Provides for system input and output through data streams, serialization and the file system
java.lang	Provides classes that are fundamental to the design of the Java programming language
java.math	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal)
java.net	Provides the classes for implementing networking applications
java.rmi	Provides the RMI package
java.security	Provides the classes and interfaces for the security framework
java.sql	Provides the JDBC package
java.text	Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages
java.util	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

User Defined Packages

Creating User-Defined packages

We must first declare the name of the package using the keyword package, followed by the package name.

Naming conventions for a package

Packages can be name using the standard Java naming rules. By convention, however, packages begin with lowercase letters, in order to make it easy to distinguish package names from class names (all class names begin with an uppercase letter). Every package name must be unique to make the best use of packages. Duplicate names will cause run-time error. The statement, declaring the package should be the first statement in a Java source file. Then we define a class just like any other class.

Given below is an example:

```
//  package declaration
package MyPackage;

// class definition
public class MyClass
{
    .....
    .....
    (body of class)
    .....
```

```
    .....................
}
```

In this example, the package name is MyPackage. The class MyClass is now considered a part of this package i.e. MyPackage. The class would now be saved as a file called MyClass.java and located in a directory named MyPackage. When the source file is compiled, Java will create a .class file and store it in the same directory. The .class files must be located in the directory that has the same name as the package and this directory should be a sub directory of the directory where classes that will import the package are located. When a source file with more than one class definition is compiled, Java creates independent .class files for those classes. Given below are the steps for creating user-defined packages:

1. Declare the package at the beginning of a file using the notation: package packagename;
2. Define the class that is to be put in the package and declare it public.
3. Create a subdirectory under the directory, where the main source files are stored.
4. Store the listing as the classname.java file in the subdirectory created.
5. Compile the file. This creates .class file in the subdirectory.

Case is significant, therefore the sub directory name must match the package name exactly.
A Java package file can have more than one class definitions. In such cases, only one of the class may be declared public and that class name with .java extension is the source file name. Java supports the concept of package hierarchy. This is done by specifying multiple names in a package statement, separated by dots. For example,

```
package firstpackage.secondpackage;
```

This package is stored in a sub directory named firstpackage/secondpackage. This means we can group related classes into a package and then group related packages into a larger package.

Accessing a Package

A Java system package or a Java user-defined package can be accessed either by using a fully qualified class name or using a shortcut approach through the import statement. Given below are the examples:

```
import firstpackage.secondpackage.MyClass;
import firstpackage.secondpackage.*;
import packagename.*;
```

Here packagename may denote a single package or a hierarchy of packages. The asterisk (*) indicates that the compiler should search the entire package hierarchy, when it encounters a class name. This implies that we can access all the classes contained in the above package directly. The major draw back of this approach is that it is difficult to determine from which package a particular member came. Here is an example of a program, which uses the user-defined packages:

```
/* declaration of package package1 */
package package1;
public class ClassA
{
    public void displayA( )
```

```
    {
        System.out.println("This is class A");
    }
}
```

We have defined a package: package1. package1 contains a single class ClassA. This source file should be named as ClassA.java and should be stored in the subdirectory package1. Now compile this java file. The resultant ClassA.class will be stored in the same subdirectory.

```
/* declaration of package package2 */
package package2;
public class ClassB
{
    public void displayB( )
    {
        System.out.println("This is class B");
    }
}
```

Now we have defined another package package2. This package contains a single class ClassB. The source file ClassB.java is stored in a subdirectory package2 and is then compiled so that the class file ClassB.class is also stored in the same subdirectory.

```
/* package1 is imported to this program */
import package1.*;

/* ClassB has been imported from  package2 */
import package2.ClassB;
class PackageTest
{
    public static void main(String args[])
    {
        /* obj1 is declared as an object of ClassA */
        ClassA obj1 = new ClassA( );

        /* obj2 is declared as an object of ClassB */
        ClassB obj2 = new ClassB( );

        /* displayA( ) method is called here */
        obj1.displayA( );

        /* displayB( ) method is called here */
        obj2.displayB( );
```

```
    }
}
```

Given above is the code of the program that imports the classes from **package1** and class **ClassB** from **package2**. This source file should be saved as **PackageTest.java** in the directory of which **package1** and **package2** are the subdirectories. Then compile the source file **PackageTest.java**. During the compilation of **PackageTest.java**, the compiler checks for the file **ClassA.class** and **ClassB.class** in the directory **package1** and **package2** respectively. But it does not actually include the code from **ClassA.class** and **ClassB.class** in the **PackageTest.class**. When this program i.e. **PackageTest** is run, Java looks for the file **PackageTest.class** and loads it using class loader. Now the interpreter knows that it also needs the code in the files **ClassA.class** and **ClassB.class** and loads them as well.