6.1: Packages

IT1406 - Introduction to Programming

Level I - Semester 1

6.1. Packages

Introduction to package

- Packages are containers for classes.
- They are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

Introduction to package (contd.)

- Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.
- The package is both a naming and a visibility control mechanism.
- You can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are exposed only to other members of the same package.

Defining a Package

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications.

Defining a Package (contd.)

- This is the general form of the **package** statement:

  package *pkg*;

- Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:

  package MyPackage;

- Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.
- More than one file can include the same **package** statement.

Defining a Package (contd.)

- The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package.
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here:

  package *pkg1*[.*pkg2*[.*pkg3*]];

- A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

  package java.awt.image;

  needs to be stored in **java\awt\image** in a Windows environment.

Finding Packages and CLASSPATH

- Packages are mirrored by directories.
- How does the Java run-time system know where to look for packages that you create?
  - Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
  - You can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
  - You can use the **-classpath** option with **java** and **javac** to specify the path to your classes.
- For example, consider the following package specification:

```
package MyPack
```

Finding Packages and CLASSPATH (contd.)

- In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.
- When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path to **MyPack** is

  C:\MyPrograms\Java\MyPack

  then the class path to **MyPack** is

  C:\MyPrograms\Java

A Short Package Example

```
package MyPack;
        class Balance {
                String name;
                double bal;
                Balance(String n, double b) {
                        name = n;
                        bal = b;
                }
                void show() {
                        if(bal<0)
                        System.out.print("--> ");
                        System.out.println(name + ": $" + bal);
                }
        }
class AccountBalance {
        public static void main(String args[]) {
                Balance current[] = new Balance[3];
                current[0] = new Balance("K. J. Fielding", 123.23);
                current[1] = new Balance("Will Tell", 157.02);
                current[2] = new Balance("Tom Jackson", -12.33);
                for(int i=0; i<3; i++) current[i].show();
        }
}
```

A Short Package Example (contd.)

- Call this file **AccountBalance.java** and put it in a directory called **MyPack**.
- Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

- Remember, you will need to be in the directory above **MyPack** when you execute this command.
- **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

- **AccountBalance** must be qualified with its package name.

Access protection

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- ✓ Subclasses in the same package
- ✓ Non-subclasses in the same package
- ✓ Subclasses in different packages
- ✓ Classes that are neither in the same package nor subclasses

Access protection (contd.)

- The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. The following table sums up the interactions.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- Anything declared **public** can be accessed from anywhere.
- Anything declared **private** cannot be seen outside of its class.

Access protection (contd.)

- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.
- Above table applies only to members of classes.
- A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

Access Example

- This example has two packages and five classes.
- The classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.
- The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**.
- The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**.

Access Example (contd.)

- Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out.
- The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one.
- The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.

Protection.java:

```
package p1;
public class Protection {
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;

        public Protection() {
                System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

Derived.java:

```
package p1;
class Derived extends Protection {
        Derived() {
                System.out.println("derived constructor");
                System.out.println("n = " + n);
// class only
// System.out.println("n_pri = "4 + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}
```

SamePackage.java:

```
package p1;
class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);
    // class only
    // System.out.println("n_pri = " + p.n_pri);
        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Protection2.java:

- Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions that are affected by access control.
- The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection.
- Remember, the default only allows access from within the class or the package, not extrapackage subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

Protection2.java:

```
package p2;
class Protection2 extends p1.Protection {
    Protection2() {
        System.out.println("derived other package constructor");
        // class or package only
        // System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

OtherPackage.java:

```
package p2;
    class OtherPackage {
        OtherPackage() {
            p1.Protection p = new p1.Protection();
            System.out.println("other package constructor");
            // class or package only
            // System.out.println("n = " + p.n);
            // class only
            // System.out.println("n_pri = " + p.n_pri);
            // class, subclass or package only
            // System.out.println("n_pro = " + p.n_pro);
            System.out.println("n_pub = " + p.n_pub);
        }
}
```

Demo package p1

- To try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
    public static void main(String args[]) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}
```

Demo package p2

- The test file for **p2** :

```
// Demo package p2.
package p2;

// Instantiate the various classes in p2.
public class Demo {
        public static void main(String args[]) {
                Protection2 ob1 = new Protection2();
                OtherPackage ob2 = new OtherPackage();
        }
}
```

Importing Packages

- Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.
- The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.
- In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

  import *pkg1* [.*pkg2*].(*classname* | *);

Importing Packages (contd.)

- Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**).
- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.
- Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package.
- This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

Importing Packages (contd.)

- The basic language functions are stored in a package inside of the **java** package called **java.lang**.
- Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs.
- This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

Importing Packages (contd.)

- It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

- The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

Importing Packages (contd.)

- When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.
- For example, if you want the Balance class of the package MyPack shown earlier to be available as a standalone class for general use outside of MyPack, then you will need to declare it as public and put it into its own file, as shown here:

Importing Packages (contd.)

```
package MyPack;
// Now, the Balance class, its constructor, and its show() method are public.
public class Balance {
        String name;
        double bal;
        public Balance(String n, double b) {
                name = n;
                bal = b;
        }
public void show() {
        if(bal<0)
               System.out.print("--> ");
        System.out.println(name + ": $" + bal);
        }
}
```

Importing Packages (contd.)

- The **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;
class TestBalance {
        public static void main(String args[]) {
        /* Because Balance is public, you may use Balance
        class and call its constructor. */
        Balance test = new Balance("J. J. Jaspers", 99.88);
        test.show(); // you may also call show()
        }
}
```