Chapter : 5

Packages and Interfaces

5.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- Describe what is meant by a package
- Explain how to define a package
- State the way of importing packages in programs
- Describe what is an interface
- Explain how to define and implement an interface

5.1 INTRODUCTION

Packages are containers for classes which are used to keep the class name space compartmentalised. They are stored in a hierarchical manner and are explicitly imported into new class definitions. **Interfaces** can be used to specify a set of methods which can be implemented by one or more classes. The interface, itself does not define any implementation. An interface is similar to an abstract class, however a class can implement more than one interface. On the other hand, a class can only inherit a single superclass. Packages and interfaces are two of the basic components of a Java program.

In general a Java source file contains the following four internal parts :

- A single package statement (optional)

- Any number of import statements(optional)
- A single public class declaration (required)
- Any number of classes private to the package (optional)

Till now, we have only used the single **public** class declaration in all our programs. We shall see the remaining in this section.

5.2 PACKAGES

5.2.1 CLASSPATH

Before studying packages it is important to understand **CLASSPATH**. The specific location that the Java compiler will consider as the root of any package is controlled by **CLASSPATH**. Till this time, we have been storing all our classes in the same, unnamed default package. This allows you to compile the source code and run the Java interpreter on the compiled program by giving the name of the class on the command line. This works because the default current working directory is usually in the **CLASSPATH** evironmental variable. This variable is defined for the Java run time system by default.

Suppose you create a class **MyPack** in a package called pack. This means you have to create a directory called pack and put **MyPack.java** into this directory. pack should now be made the current directory to compile the program. The class file **MyPack.class** which will be created after compilation will also be stored in the pack directory. Now when you try to run **MyPack**, the Java interpreter reports an error. This is because the class is now stored in a package called **pack**. Thereofore you cannot refer to it only as **MyPack**. You should now refer to the class by enumerating its package hierarchy and separating the package by dots. Thus the class will now be called **MyPack.pack**. Now if you try using **MyPack.pack**, you will still receive an error message. This is because **CLASSPATH** sets the top of the class hierarchy. So there is no directory called pack, in the current working directory since you are working in pack itself.

To overcome this problem you have two alternatives :

i) change directories up one level and type java **MyPack.pack**

ii) add the top of your development class hierarchy to the **CLASSPATH** environment variable. Then you will be able to use java **MyPack.pack** from any directory and Java is able to find the correct **.class** file. eg. if you are working on source code in a directory called c:\student then you should set your **CLASSPATH** as :

.;C:\student;C:\java\classes

5.2.2 Defining Packages

You can define classes inside a package that are not accessible to the code outside the package. You can also define class members that are only exposed to other members of the same package. Thus the classes can have knowledge of

each other, but it is not exposed outside the package.

To create a package, you should include a package command as the first statement in a Java source file. Any classes defined within that file will belong to the specified package.

The general form for the package statement is :

package *pkg*;

where *pkg* is the name of the package. eg.

package pack

will create a package named pack.

Java uses the file system directories to store packages. Therefore any **.class** files for any classes declared to be a part of **pack**, must be stored in a directory called **pack**. The directory name must match the package name.

More than one file can be included the same package statement. You can also create a hierarchy of packages. The general form of a multilevel package statement is :

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

remember that the package hierarchy should be reflected in the file system of your Java development system. eg. package **java.awt.image**;

should be stored in j**ava\awt\image** on your Windows file system.

Let us create a small package with the following example :

```
package Pack;
class Student {
    String name;
    int roll;
    Student(String n, int r) {
        name = n;
        roll = r;
    }
    void display() {
        System.out.println("Name : " + name + "  Roll No. : " + roll);
    }
}
class Studrec {
    public static void main(String args[]) {
        Student S[] = new Student[3];
        S[0] = new Student("Vikram ", 20);
        S[1] = new Student("Jerry ",10);
        S[2] = new Student("Vishnu ",15);
```

```
            for(int i = 0; i<3; i++)
                S[i].display();
        }
    }
```

Save this file as **Studrec.java** and put in a directory called **Pack**. When you compile this file, make sure that the resulting **.class** file has also been stored in Pack. Then execute **Studrec** at the command line as follows :

java Pack.Studrec

Either set the **CLASSPATH** environment as described previously or go into the directory above Pack to execute the program. Remember that since **Studrec** is now a part of the package **Pack**, it cannot be executed by itself i.e. *a command line*

java Studrec

cannot be used.

5.2.1 & 5.2.2 Check Your Progress.

1. Write True or False.

a) A class can implement more than one interface.

b) You should include a package command as the first statement in a Java source file.

c) You can define classes inside a package that are not accessible to the code outside the package.

5.2.3 CATAGORIES OF VISIBILITY FOR CLASS MEMBERS WITH REFERENCE TO PACKAGES

Packages act as containers for classes and other subordinate packages. Java addresses four catagories of visibility for class members with reference to packages :

- subclasses in the same package
- non subclasses in the same package
- subclasses in different packages

	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

- Classes that are neither in the same package nor subclass.

The following table summarises the class member access

Anything declared **public** can be accessed from anywhere. Anything declared **private** is not accessible outside of its class. When a member does not have any specific access specification, it is visible to all 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 are the direct subclasses of your class, you have to declare that element as **protected**.

5.2.4 Importing packages

All of the built in Java classes are stored in packages. In order to make visible certain classes or entire packages, Java makes use of the **import** statement. Once imported, a class can be referred to directly using only its name. In a Java source file, the **import** statements occur immediately following the package statement (if it exists) and before any class definitions. The general form of the import statement is :

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

pkg1 is the name of the top level package, *pkg2* is the name of the subordinate package and is separated by a dot. Practically there is no limit to the depth of a package hierarchy.Then you can specify either an explicit *classname* or a star (*). A * indicates that the Java compiler should import the entire package. eg.

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

All the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package called **java.lang**. **java.lang** is implicitly imported by the compiler for all the programs because is provides a lot of functionality. This is equivalent to importing it as :

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

If a class with the same name exists in two different packages which you import using the * form, then compiler does not issue any warning unless you try to use one of the classes. In that case a compile time error occurs and you have to explicitly name the class and its package name.

Wherever you use a class name you can use its fully qualified name which will include its full package hierarchy. eg.

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

can be written without import as :

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

Only those items within the package declared as **public** will be available to non subclasses in the importing code.

5.2.3 & 5.2.4 Check Your Progress.

1. Fill in the blanks.

a) The basic language functions in Java are stored in a package called ..... .

b) To make visible certain classes or entire packages, Java provides the ..... statement.

c) To allow an element to be seen outside the current package, but only to classes that are the direct subclasses of your class, that element should be declared as ..... .

5.3 INTERFACES

5.3.1 Declaring an Interface

Using an interface you can specify what a class must do, but not how it does it. You can fully abstract a class' interface from its implementation. Interfaces do not have instance variables and their methods do not have any bodies. Once an interface is defined, any number of classes can implement an interface. Also one class can implement any number of interfaces. In order to implement an interface, a class must create a complete set of methods defined by the interface, but each class is free to implement its own details.

Interfaces are designed to support dynamic method resolution at run time. Usually, for a method to be called from one class to another class, both the classes need to be present at compile time. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.

The general form of an interface is :

access interface *name* {

return-type method-name1(*parameter-list*);

return-type method-name2(*parameter-list*);

return-type method-name2(*parameter-list*);

.....

return-type method-nameN(*parameter-list*);

type final-varnameN = value;

}

In this form, access is either **public** or not used. The default access applies when no specifier is used. It implies that the interface is only available to other members of the package in which it is declared. When the interface is declared **public**, the interface can be used by another code. *name* is the name of the interface which is a valid identifier. The methods in an interface are abstract methods. Each class which includes an interface must implement all the methods.

Variables can also be declared inside interface declarations. They are implicitly **final** and **static**. This means they cannot be changed by the classes which implement the interface. The variables must be initialised with constant values. If an interface is declared **public**, then all the methods and variables are implicitly declared **public**.

Let us see an example of interface declaration :

```
interface A {
    void meth1(int i);
}
```

5.3.1 Check Your Progress.

1. Fill in the blanks.

a) Variables declared in interfaces are implicitly ..... and .....

b) The ..... access applies when no specifier is used for an interface.

c) Interfaces do not have ..... variables.

5.3.2 Implementing an interface

To implement an interface include the **implements** clause in the class definition and then create the methods that are defined by the interface The general form of a class which includes the implements clause is :

access class *classname* [extends *superclass*]

[implements *interface* [,*interface....*]] {

class body

}

access is either **public** or not used. If a class implements more than one interface they should be separated by commas. The methods that implement an interface must be declared **public**. Also note that classes which implement interfaces, can have additional members of their own also.

The following example will demonstrate how to implement the above declared interface :

```
class One implements A {
    public void meth1(int i) {
        System.out.println("Implement interface A in class One");
        System.out.println("i is :" + i);
    }
}
```

You can also declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be stored in such a variable. When a method is called through any such references, the correct version based upon the actual instance of the interface

being referred to will be called.The method to be executed is looked up dynamically at run time. This allows classes to be created later than the code which calls methods on them.

Let us call the method **meth1()** in the above example with the help of an interface reference variable as shown :

```
class Iface {
        public static void main(String args[]) {
                A Ainter = new  One();
                Ainter.meth1(10);
        }
}
```

In this example, the variable AInter is of interface type A, but it is assigned an instance of class One. Ainter can access meth1(), but it cannot access any other members of the class One because the interface reference variable has only knowledge of the methods declared by that interface declaration. It cannot be used to access any other methods of the class.

If a class includes an interface but does not fully implement the methods defined by that interface, then the class must be declared **abstract**. eg

```
abstract class Two implements A {
        int p, q;
        void display() {
                System.out.println("p and q" + p + q);
        }
}
```

This class does not implement meth1() of the interface A therefore is declared **abstract**.

Interfaces can be extended i.e. one interface can inherit another by the use of the keyword **extends**. When a class implements an interface which inherits another interface, then it must provide implementations for all the methods defined within the interface inheritance chain.

5.3.3 Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialised to the desired values. When the interface is included in a class all those variables names will be in scope as **constants**. (This is similar to a header file in C/C++ which contains a number of **#define** constants or **const** declarations).

5.4 SUMMARY

Packages are containers for classes which are used to keep the class name space compartmentalized. Packages act as containers for classes and other subordinate packages.

Interfaces can be used to specify a set of methods which can be implemented by one or more classes.

5.5 CHECK YOUR PROGRESS - ANSWERS

5.2.1 & 5.2.2

1. a) True

b) True

c) True

5.2.3 & 5.2.4

1. a) java.lang

b) import

c) protected

5.3.1

1. a) static

b) default

c) instance

5.3.2 & 5.3.3

1. a) False

b) True

2. a) implements

b) abstract

5.6 QUESTIONS FOR SELF - STUDY

1. What is meant by packages and interfaces?
2. Explain with example what is CLASSPATH.
3. Explain the general form of declaring a package.
4. What do you understand by public, private and protected access, with relation to packages?
5. Write notes on :
   a) Declaring an Interface
   b) Implementing an Interface

5.7 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

Notes