UNIT 7: INTERFACE AND PACKAGES

UNIT STRUCTURE

7.1 Learning Objectives

7.2 Introduction

7.3 Introduction to Interface

- 7.3.1 Defining an Interface
- 7.3.2 Implementing an Interface
- 7.3.3 Nested Interface

7.4 Package

- 7.4.1 Defining a Package
- 7.4.2 Finding Packages and CLASSPATH
- 7.4.3 Access Protection in Packages
- 7.4.4 Importing Packages

7.5 Let Us Sum Up

7.6 Further Reading

7.7 Answer to Check Your Progress

7.8 Model Questions

7.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- learn about the concept of multiple inheritance through Interface
- acquire the process of defining and implementing Interface
- get the usefulness of a package
- learn the process of defining and using package

7.2 INTRODUCTION

This unit describes two of Java's most advanced features: packages and interfaces. There is no concept of multiple-inheritance in Java. Instead of this, Interface concept is included in Java which is unique to the language. Another unique feature in Java is Packages. Packages are containers for classes which are meant for keeping the class name space

compartmentalized. By organizing the user's classes into packages, they can be reused to write other classes. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions. For example, a package permits us to create a class named P1, which we can store in our own package.

7.3 INTRODUCTION TO INTERFACE

An interface in Java is essentially a special kind of class. Like classes, interfaces contain methods and variables. However, unlike classes, interfaces are always completely abstract. Through the use of the interface, Java allows us to fully abstract the interface from its implementation. Using interface, we can specify a set of methods that can be implemented by one or more classes. The interface, itself, does not actually define any implementation. They are similar to abstract classes, but interface has an advantage over an abstract class. A class can implement more than one interface wherein, a class can only inherit a single superclass (abstract or otherwise). Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time. Interfaces are designed to disconnect the definition of a method or set of methods from the inheritance hierarchy.

7.3.1 Defining an Interface

Similar to a class declaration, an interface is defined. The general form of an interface is:

```
 interface interface_name
{
type final-varname1 = value;
type final-varname2 = value;
//…
type final-varnameN = value;
return-type method-name1(parameter-list);
```

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

// ...

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

}

Although member elements are present in an interface, they are static and final, so they cannot be changed by the implementing class. They must be initialized. Again, methods are only declared in an interface; the code to implement the method is not written. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods. All methods and variables inside an interface are implicitly public. While classes can inherit from only one super class, interfaces are able to inherit from as many as interface one can provide.

After defining an interface, it is required to implement with classes which can be done with the following syntax:

```
class  ClassName [ extends SuperClassName ]
[ implements InterfaceName1 [, Interface Name2] )
 {
                class body
 }
```

7.3.2 Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, the implements clause is used in a class definition, and then create the methods defined by the interface.

The general form of a class that includes the implements clause looks like:

class classname[extends superclass][implements interface [,interface...]]

```
{
    // class-body
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

An example of an interface and its implementation is given below:

```
interface Resume
{
  void bioData( ) ; // By default it is an abstract
method.
            // when it is declared in an interface

}
class Teacher extends Employee implements Resume {
  // Assume Employee is a class defined somewhere
String Qualification;// A  member element

experience( ) {  …..  } // A member function
public void bioData( )
{
  // Code for this method for preparing the bio-
data
  of a teacher //
}
}
class Student implements Resume {
String Discipline;    // A member element.
```

```
result( ){ …… } // A member function
public void bioData ( ) {
// Code for the method for preparing the bio-data
   of a student //
}
}
```

Accessing Implementations through Interface References

We can 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 referred to by such a variable. When a method is called by using one of these references, the correct one will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is chosen dynamically at run time, allowing classes to be created rather than the code which calls methods on them. The calling code can dispatch through an interface without the knowledge of anything about the "callee."

Here is an example which calls a method named callback( ) via an interface reference variable:

```
class Test {
public static void main(String args[])
{
   Callback c = new Client();
   c.callback(60);
}
}
```

The output of this program will be:

callback called with 60

Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback( ) method, it cannot access any other members of the Client class. The object reference c cannot be used to access other methods defined by Client.

Partial Implementations:

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

For example:

```
abstract class Test1 implements Callback
{
  int x, y;
  void show()
  {
   System.out.println(x + " " + y);
  }
}
```

This example shows that the class Test1 does not implement callback( ) and must be declared as abstract. Any class that inherits Test1 must implement callback( ) or be declared abstract itself.

7.3.3 Nested Interfaces

An interface which can be declared a member of a class or another interface is called a member interface or a nested interface. A nested interface can be declared either as public, private, or protected. When a nested interface is used outside of its scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

The following example demonstrates the concept of a nested interface:

```
// A nested interface example.
// This class contains a member interface.
class A {
public interface NestedIF
{
```

```
  boolean negative(int a);
}
}
// B implements the nested interface.
class B implements A.NestedIF
{
    public boolean negative(int a)
    {
        return a < 0 ? false : true;
    }
}
class Main
{
   public static void main(String args[])
  {
   // use a nested interface reference
   A.NestedIF nif = new B();
   if(nif.negative(10))
   System.out.println("10 is not negative");
   if(nif.negative(-12))
   System.out.println("Yes, it is negative");
  }
}
```

Notice that A defines a member interface called NestedIF and that it is declared public. Next, B implements the nested interface by specifying

implements A.NestedIF

Moreover, the name is fully qualified by enclosing class name. Inside the main( ) method, an A.NestedIF reference named nif is created, and it is assigned a reference to a B object. Because B implements A.NestedIF, which is legal.

Applying Interfaces:

To understand the power of interfaces, let's consider another

practical example. We can consider a class called Stack that implements the functions of a simple fixed-size stack. No matter how the stack is implemented, the interface to the stack will remain the same where the methods push( ) and pop( ) define the interface to the stack independently of the details of the implementation.

```
// Define an integer stack interface.
interface IntStack
{
void push(int item); // store an item
int pop(); // retrieve an item
}
```

The following program creates a class called Stack that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class Stack implements IntStack
{
private int stck[];
private int tos;
// allocate and initialize stack
Stack(int size)
{
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item)
{
if(tos==stck.length-1) // use length member
System.out.println("Stack is full.");
else
```

```
stck[++tos] = item;
}
// Pop an item from the stack
public int pop()
{
   if(tos < 0) {
    System.out.println("Stack underflow.");
   return 0;
}
else
   return stck[tos--];
}
}
class IFTest
{
   public static void main(String args[])
  {
    Stack mys1 = new Stack(10);
    Stack mys2 = new Stack(15);
   // push some numbers onto the stack
   for(int i=0; i<10; i++)
   mys1.push(i);
   for(int i=0; i<15; i++)
   mys2.push(i);
// pop those numbers off the stack
   System.out.println("Stack in mystack1:");
   for(int i=0; i<10; i++)
   System.out.println(mys1.pop());
   System.out.println("Stack in mystack2:");
   for(int i=0; i<15; i++)
   System.out.println(mys2.pop());
  }
}
```

Interfaces Can Be Extended:

One interface can inherit another one by using the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following example explains it:

```
// One interface can extend another.
interface A
{
  void meth1();
  void meth2();
}
// B now includes meth1() and meth2()
interface B extends A
{
  void meth3();
}
// This class must implement all of A and B
class MyClass implements B
{
   public void meth1()
    {
     System.out.println("Implementing meth1().");
    }
    public void meth2()
    {
    System.out.println("Implementing meth2().");
 }
 public void meth3()
 {
    System.out.println("Implementing meth3().");
 }
}
```

```
class Interface_Extend
{
   public static void main(String arg[])
   {
      MyClass ob = new MyClass();
      ob.meth1();
      ob.meth2();
      ob.meth3();
  }
}

/* Example on the multiple inheritance with
interface with implementing two or mor interfaces*/
class A {
   protected int i = 100;
   public void print() { System.out.println("I am
from A "+i);
    } }
interface C {
    public static int j = 500;
    void printInterfaceC();
}
interface D {
    public static int k = 600;
    void printInterfaceD();
}
class B extends A implements C, D{
     public int aValue = 900;
     public void printInterfaceC() {
          System.out.println("I am from C " + j);
     }
     public void printInterfaceD() {
          System.out.println("I am from D " + k);
```

```
    }
    public void printB() {
        super.print();
        printInterfaceC();
        printInterfaceD();
    }
}
class Demonstration {
   public static void main (String[] args){
            B b = new B();
            b.printB();
   } }
```



CHECK YOUR PROGRESS

Q 1: Choose the correct options:

i) Which of these can be used to fully abstract a class from its implementation?

a) Objects b) Packages c) Interfaces

d) None of the Mentioned

ii) Which of these access specifiers can be used for an interface?

a) Public b) Protected c) private

d) All of the mentioned

iii) Which of the following is the correct way of implementing an interface A by class B?

a) class B extends A{} b) class B implements A{}

c) class B imports A{} d) None of the mentioned

iv) All methods must be implemented of an interface.

a) True b) False

7.4 PACKAGES

In the preceding examples, the name of each class was taken from the same name space. This means that a unique name has to be used for each class to avoid name collisions. Gratefully, Java includes a mechanism for partitioning the class name space into more manageable chunks. This is known as the package. The package is both a naming and a visibility control mechanism. We can define many classes inside a package that are not accessible by code outside that package. Moreover, we can declare class members that are only accessible to other members of the same package. Thus, it allows us classes to have intimate knowledge of each other, but not to expose that knowledge to the rest of the world.

There are two types of packages available in Java which are:

Built-in packages

These packages are already defined in java API, for example, java.util, java.io,java,lang, java.awt, java.applet, java.net, etc.

User defined packages

When user creates a package according to his/her need, then it is called user defined package.

7.4.1 Defining a Package

To create a package, we have to 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**. Actually, the package statement defines a name space in which classes are stored. In absence of package statement, the class names are put into the default **package**, which has no name.

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

package pkg_name;

Here, pkg_name is the name of the package.

For example, the following statement creates a package **MyPackage.**

package MyPackage;

Java uses file system directories to store packages. For example, the .class files for any classes declared to be part of MyPackage and must be stored in a directory called MyPackage. It is to be noted that case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply depicts 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. Most real-world packages are spread across many files.

Java allows us to create a hierarchy of packages. To order 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 in 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. It is to be noted that we cannot rename a package without renaming the directory in which the classes are stored.

7.4.2 Finding Packages and CLASSPATH

As explained above, packages are mirrored by directories. Now one important question is: How does the Java run-time system know where to look for packages that you create? For this query, there are three answers. First, by default, the Java run-time system uses the current working directory as its starting point.

Secondly, we can specify a directory path or paths by setting the CLASSPATH environmental variable. Lastly, Java allows us to use the - classpath option with Java and javac to specify the path to the classes.

For example, consider the following package specification:

package MyPackage

In order for a program to find MyPackage, one of three things must be true. Either the program can be executed from a directory immediately above MyPackage, or the CLASSPATH must be set to include the path to MyPackage, or the -classpath option must specify the path to MyPackage when the program is run via java.

For other two options, the class path must not include MyPackage itself. It is to specify simply the path to MyPackage. For example, in a Windows environment, if the path to MyPackage is

C:\MyPrograms\Java\MyPackage

Then the class path to MyPackage is

C:\MyPrograms\Java

This is the approach used in the following example.

A Short Package Example

```
// A simple package
package MyPackage;
class BalanceAmount
{
   String name;
   double bal;
   BalanceAmount(String n, double b)
   {
     name = n;
     bal = b;
   }
   void show()
   {
      if(bal<0)
      System.out.print("--> ");
      System.out.println(name + ": $" + bal);
   }
}
class Account_Balance
{
```

```
public static void main(String args[])
{
   BalanceAmount savings[]= new Balance[3];
  savings[0] = new BalanceAmount("Ramesh Murthi",
5460.67);
    savings[1] = new BalanceAmount("Williamson",
3257.02);
    savings[2] = new BalanceAmount("Amit Kohli",
456.78);
  for(int i=0; i<3; i++) savings[i].show();
 }
}
```

Now, we have to call this file Account_Balance.java and put it in a directory called MyPackage. Next, we have to compile the file. Make sure that the resulting .class file is also in the MyPackage directory. Then, try to execute the Account_Balance class, using the following command line:

java MyPackage.Account_Balance

7.4.3 Access Protection in Packages

It is clear from the above discussion that access to a private member of a class is granted only to other members of that class. Packages provide another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. 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 whereas Classes act as containers for data and code. Thus, the class is considered Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java provides 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

The three access specifiers namely private, public, and protected, provide various ways of access protection. Table 1 presents the summary of the class members accessibility. While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. 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 we want to allow an element to be seen outside the current package, but only to classes that subclass our class directly, then declare that element protected.

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.

Table 1. Accessibility of Class members

	Private	**Public**	**Protected**	**No modifier**
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	Yes	Yes	No
Different Package non-subclass	No	Yes	No	No

An Example of Access control

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages require to be stored in directories named after their respective packages-say, p1 and p2. The source for the first package p1 defines three classes: Protection, Derived, and SamePackage. The Protection class defines four int variables in each of the legal protection modes. The variable n is declared with the default protection, n_priv is private, n_prot is protected, and n_publ is public. Each subsequent class tries to access the variables as an instance of this class. The lines that cannot be compiled due to access restrictions are put as comments.

The second class, Derived, is a subclass of Protection in the same package, p1. This permits Derived class to access to every variable in Protection except for n_priv, 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 except n_priv.

This is the file **Protection.java:**

```
package p1;
public class Protection
{
int n = 1;
private int n_priv = 2;
protected int n_prot = 3;
public int n_publ = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_priv);
System.out.println("n_pro = " + n_prot);
```

```
System.out.println("n_pub = " + n_publ);
}
}
```

This is file **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_priv);
System.out.println("n_prot = " + n_prot);
System.out.println("n_publ = " + n_publ);
}
}
```

This is file **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_priv = " + p.n_priv);
System.out.println("n_prot = " + p.n_prot);
System.out.println("n_publ = " + p.n_publ);
}    }
```

Following is the source code for the second 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 allows to access to all of p1.Protection's variables except for n_priv and n, the variable declared with the default protection. It is to be mentioned that the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class OtherPackage has access to only one variable, n_publ, which was declared public.

This is file **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_priv);
System.out.println("n_prot = " + n_prot);
System.out.println("n_publ = " + n_publ);
}
}
```

This is file **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_priv = " + p.n_priv);
// class, subclass or package only
```

```
        // System.out.println("n_prot = " + p.n_prot);
        System.out.println("n_publ = " + p.n_publ);
        }
        }
```

If we want to use these two packages, here are two test files we can use. First one for p1 is:

```
        // 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();
        }
        }
```

The test file for p2 is shown next:

```
        // 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();
        }
        }
```

7.4.4 Importing Packages

There are no core Java classes in the unnamed default package. All of the standard classes are stored in some named package.

Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. Due to this, 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.

In a Java source file, 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|*);
```

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 limit on the depth of a package hierarchy, except that imposed by the file system. Finally, we 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.*;
```

All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in a package inside of the java package called java.lang. Normally, we have to import every package or class that we 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:

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

If a class with the same name exists in two different packages that we import using the * form, the compiler will remain silent, unless we try to use one of the classes. In that case, we will get a compile-time error and have to explicitly name the class specifying its package. It must be emphasized that the import statement is optional. Any place we use a class name, we can use its fully qualified

name, which includes its full package hierarchy. For example, the following program segment 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
{
}
```

In this version, Date is fully-qualified.

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 we want the BalanceAmount class of the package MyPackage shown earlier to be available as a stand-alone class for general use outside of MyPackage, then it is required to declare it as public and put it into its own file, as shown here:

```
package MyPackage;
public class BalanceAmount {
String name;
double bal;
public BalanceAmount (String n, double b) {
name = n;
bal = b;
}
public void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

Here, the BalanceAmount class is now public. Also, its constructor and its show( ) method are also public. This means that

they can be accessed by any type of code outside the MyPackage package. For example, here TestBalance imports MyPackage and is then able to make use of the BalanceAmount class:

```
import MyPackage.*;
class TestBalance {
public static void main(String args[]) {
/* Because Balance is public, you may use Balance
class and call its constructor. */
BalanceAmount test = new BalanceAmount("J. K.
Rashmi", 84.78);
test.show(); // you may also call show()
}
}
```



CHECK YOUR PROGRESS

Q 2: Select the correct option for the following questions:

i) Which of these keywords is used to define packages in Java?

a) pkg b) Pkg c) package

d) Package

ii) Which of these is a mechanism for naming and visibility control of a class and its content?

a) Object b) Packages c) Interfaces

d) None of the Mentioned

iii) Which of these access specifiers can be used for a class so that its members can be accessed by a different class in the same package?

a) Public b) Protected c) No Modifier

d) All of the mentioned

iv) What is the maximum number of Java classes that can be kept inside a Java package?

a) 8 b) 64 c) 128 d) unlimited



7.5 LET US SUM UP

- An interface in Java is essentially a special kind of class. Like classes, interfaces contain methods and variables. However, unlike classes, interfaces are always completely abstract.
- To define an interface, Java provides a keyword interface which is followed by interface name. Within an interface, variables and methods can be initialized. However, no method can be defined inside a package.
- All the variables inside an interface are declared as final and static.
- Once an interface has been defined, one or more classes can implement that interface. To implement an interface, the implements clause is used in a class definition, and then create the methods defined by the interface.
- Java includes a mechanism for partitioning the class name space into more manageable chunks. This is known as the package. The package is both a naming and a visibility control mechanism.
- A Package can group the similar type of classes, interfaces and sub-packages and provide access control. It organizes classes into single unit.
- There are two types of packages: Built-in and User defined.
- Packages provide another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. 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 whereas Classes act as containers for data and code.



7.6 FURTHER READINGS

1) Schildt, H., & Coward, D. (2014). Java: the complete reference. New York: McGraw-Hill Education.
2) Rao, N., Core and Advanced Java Black Book. dreamTech

7.7 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: i) c ii) d iii) b iv) a

Ans to Q No 2: i) c ii) b iii) b iv) d

7.8 MODEL QUESTIONS

Q 1: What is an interface? How to define it? Discuss with an example.

Q 2: How can we access an interface from another interface and how to access from a class?

Q 3: Explain extending interfaces with a proper example ?

Q 4: What is a Package? Explain the Packages with a suitable example and how to import packages?

Q 5: Describe the levels of access protection available for packages.

*** ***** ***