
UNIT 5 PACKAGES AND INTERFACES

Structure	Page Nos.
5.0 Introduction	67
5.1 Objectives	67
5.2 Creating Packages	67
5.3 Adding Classes to Existing Packages	70
5.4 Interfaces	71
5.5 Creating Interfaces	72
5.6 Exceptions	78
5.7 Summary	79
5.8 Solutions/Answers	79

5.0 INTRODUCTION

Java provides a powerful means of grouping related classes and interfaces together in a single unit – the package. In other words, **packages** are groups of related classes and interfaces. Packages provide a convenient mechanism for managing a large group of classes and interfaces, while avoiding potential naming conflicts. The Java API itself is implemented as a group of packages. In very simple terms, packages are the directory in your local machine from where the compiler imports a **class file**. For example, when we import a package, we actually send an instruction to the compiler that the **classes** we are using in our program are located in a particular directory, e.g.,

```
import java.awt.Frame
```

means that if I am using Frame class, then the **Frame.class** file is located in **java.awt** directory.

5.1 OBJECTIVES

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

- Understand the concept of packages;
- create new packages;
- add classes to pre-existing packages;
- understand the concept of interfaces;
- create new interfaces, and
- use interfaces in programs.

5.2 CREATING PACKAGES

Naming Conventions

Packages can be named using the standard Java naming rules. By convention, package names begin with lowercase letters, which makes it different from class names.

Declaring Packages

The name of the package must be preceded by the keyword **package** and this must be the first statement in a Java source file. This is followed by defining a class:

Example:

```
package myFirstPackage;
public class MyPackageClass
{
    (Body of the class)
}
```

Here the package name is **myFirstPackage**. The class **MyPackageClass** is now considered to be a part of this package. This file should be saved as **MyPackageClass.java** and compiled to create a **.class file**.

The **.class** file must be located in a 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.

A Java package file can have more than one class definition. In such cases, only one of the classes may be declared public and that class name with **.java** extension is the source file name. When a source file with more than one class definition is compiled, java creates independent **.class** files for those classes.

After declaring the package in the class file, the directory structure is not created automatically. To create a directory structure and place the **.class**, files we will compile our **.java** file with **-d** option.

```
C:\javac -d c:\ MyPackageClass.java
```

This will create a directory under the root directory by the name **myFirstPackage**.

Importing Packages

When it is time to use classes outside the package you are working in, you must use the import statement. The import statement enables you to import classes from other packages into a compilation unit. You can import individual classes, or entire packages of classes at the same time if you want. The syntax for the import statement is:

Import Identifier;

Identifier is the name of the class or package of classes you are importing.

```
import java.awt.Color;
import java.awt.*;
```

The first statement imports the specific class Color, which is located in the java.awt package. The second statement imports all the classes in the java.awt package. Note that the following statement doesn't work:

```
import java.*;
```

This statement doesn't work because you can't import nested packages with the * specification. This only works while importing all the classes in a particular package.

There is another way to import objects from other packages: *explicit package referencing*. By explicitly referencing the package name each time you use an object, you can avoid using an import statement. Using this technique, the declaration of the color member variable would look like this:

Explicitly referencing the package name for an external class is generally not required; it usually serves only to clutter up the class name and can make the code harder to read. The exception to this rule is when two packages have classes with the same name. In this case, you are required to explicitly use the package name with the class names.

Examples:

```
package pkg;
public class Eternity
{
    public void disp ()
    {
        System.out.println("Eternity");
    }
}
```

The class file is named Eternity.java and stored in the sub-directory pkg. The .java file is compiled into Eternity.class file and stored in the same sub-directory.

```
import pkg.Eternity;
class Testing
{
    public static void main(String args[ ])
    {
        Eternity objectA=new Eternity();
        ObjectA.disp ();
    }
}
```

This program imports the class Eternity from the package pkg. The source file of the program is stored as Testing.java and stored along with its compiled file in the parent directory of pkg. The compiler checks the Eternity.class file in the pkg directory but does not include it in the Testing.class file. When you run this program, Java loads the Testing.class file using a class loader. At this stage, the interpreter comes to know that it needs the Eternity.class file, and loads the same.

```
package pkgB;
public class EternityB
{
    protected int j =5;
    public void dispB()
    {
        System.out.println(EternityB");
        System.out.println("j=" + j);
    }
}
```

The source and compiled files of this package are located in the sub-directory pkgB.

```
import pkg.Eternity;
import pkgB.*;

class TestingB
```

```
{  
    public static void main(String args[ ]) {  
        {  
            Eternity objectA=new Eternity();  
            EternityB objectB=new EternityB();  
            ObjectA.disp();  
            ObjectB.dispB();  
        }  
    }  
}
```

This program uses classes from two packages. It is saved as TestingB.java, compiled and run. The output is as shown:

```
Eternity  
EternityB  
j= 5
```

You can subclass a class that you have imported from another package. If you want your new subclass to inherit members of the parent class, you must declare them as protected; for example, the variable “j” in the above program.

```
import pkgB.EternityB;  
class EternityC extends EternityB  
  
{  
    int m=10  
    void dispC()  
    {  
  
        System.out.println("EternityC");  
        System.out.println("j= "+j);  
        System.out.println("m="+m);  
    }  
  
}  
  
class TestingC  
{  
    public static void main(String args[ ]) {  
        {  
            EternityC objectC=new EternityC();  
            ObjectC.dispB();  
            ObjectC.dispC();  
        }  
    }  
}
```

5.3 ADDING CLASSES TO EXISTING PACKAGES

Suppose a package A contains a public class, Class A, and you want to add another public class, Class B to this package.

```
package packageA;  
public class ClassA;  
{  
    // (body of ClassA)  
}
```

Just define the class and make it public. Do not forget to add the package statement before the class definition, as shown:

```
package packageB;  
public class ClassB  
{  
    //(body of ClassB)  
}
```

Packages and Interfaces

Store the source and compiled files ClassB.java and ClassB.class in the directory **packageA**. You can add non-public classes also in this manner. When the package, **packageA**, is imported, both the classes **ClassA** and **ClassB** are imported, as they are contained in that package.

A **.java** file can have only one public class. So, if you want to create a package with multiple public classes in it, create the classes in separate source files and declare the package statement

```
package packagename;
```

at the top of each source file. Switch to the subdirectory created with the package name, and compile each source file. Now, the package contains **.class** files of all the source files.

5.4 INTERFACES

An *interface* is a prototype for a class, and is useful from a logical design perspective. Interfaces are abstract classes that are left completely unimplemented. *Completely unimplemented* in this case means that **no** methods in the class have been implemented. Additionally, interface member data is limited to static final variables, which means that they are constants.

The benefits of using interfaces are much the same as the benefits of using abstract classes. Interfaces provide a means to define the protocols for a class without having to worry about the implementation details. This seemingly simple benefit can make large projects much easier to manage. Once interfaces have been designed, the class development can take place without worrying about communication among classes.

Another important use of interfaces is the capacity for a class to implement multiple interfaces. The major difference between inheriting multiple interfaces and true multiple inheritance is that the interface approach enables you to inherit only **method descriptions**, not **implementations**. If a class implements multiple interfaces, that class must provide all the functionality for the methods defined in the interfaces. Although this approach is certainly more limiting than multiple inheritance, it is still a very useful feature. It is this feature of interfaces that separates them from abstract classes.

Limitations of Interfaces

Interfaces have one major limitation: they can define abstract methods and final fields, but they cannot specify any implementation for these methods. For methods, this means that while writing a method, the body is empty. The classes that implement the interface are responsible for specifying the implementation of these methods. This means that, unlike extending a class, where you implement a method, you *must* override every method in the interface.

☞ **Check Your Progress 1**

- 1) Explain what is Java Package?

.....
.....
.....

- 2) What is an interface?

.....
.....
.....

- 3) What is the syntax for creating an interface, and explain?

.....
.....
.....

5.5 CREATING INTERFACES

The syntax for creating an interface is extremely similar to that of creating a class. However, there are a few exceptions. The most significant difference is that none of the methods in your interface may have a body, nor can you declare any variables that will not serve as constants. Nevertheless, there are some important things that you may include in an interface definition.

Declaring Interfaces

The syntax for creating interfaces is:

```
interface Identifier
{
    InterfaceBody
}
```

Identifier is the name of the interface and *InterfaceBody* refers to the abstract methods and static final variables that make up the interface. Because it is assumed that all the methods in an interface are abstract, it isn't necessary to use the **abstract** keyword.

Implementing Interfaces

Because an interface is a prototype, or a template, for a class, you must implement an interface to arrive at a usable class. Implementing an interface is similar to deriving from a class, except that you are required to implement any method defined in the interface. To implement an interface, you have to use the **implements** keyword. The syntax for implementing a class from an interface follows:

```
class Identifier implements Interface
{
    //ClassBody
}
```

Identifier refers to the name of the new class, *Interface* is the name of the interface you are implementing, and *ClassBody* is the new class body. The following example includes an interface version of **packg** along with a **Test** class that implements the interface.

Packages and Interfaces

Example : The Eternity interface and Test class.

```
package packg;
import java.awt.Color;
interface Eternity
{
    abstract public void move();
    abstract public void move(int x, int y);
}

class Test implements Eternity
{

    protected Color color;

    public Test()
    {
        color=Color.green;
    }
    public Test(Color c)
    {
        color =c;
    }

    public void move()
    {
        // move an object of class Test
    }

    public void move(int x, int y)
    {
        // move the Test object to the position (x,y)
    }
}
```

Another Example:

Create an interface Product and save it as Product.java.

```
public interface Product
{
    static final String MAKER ="Cosmetics International Ltd.";
    static final String PHONE ="91-11-5556666";

    public int getPrice(int id);
}
```

Compile the file **Product.java** to get **Product.class**. Implement the interface in a class. Save the file as **Cosmetics.java**

```
import Product;
```

```
public class Cosmetics implements Product
{
    public int getPrice(int id)
    {
        if(id== 1)
        return(500);
        else
        return (100);
    }

    public String getMaker()
    {
        return(MAKER)
    }
}
```

Inherit the above class in the new class:

```
import Cosmetics;

public class CosmeticsStore
{
    static Cosmetics nailPolish;

    public static void main(String args[ ])
    {
        nailPolish = new Cosmetics();
        getInfo(nailPolish);
        orderInfo(nailPolish);
    }

    public static void getInfo(Cosmetics item)
    {
        System.out.println("This Product is made by "+item.MAKER);
        System.out.println("It costs $" + item.getPrice(1) + '\n');
    }

    public static void orderInfo(Product item)
    {
        System.out.println("To order from " + item.MAKER + " call " +
                           item.PHONE+ ".");
        System.out.println("Thankyou for Shopping with us. Please Visit" +
                           "again");
    }
}
```

Of course, the code in CosmeticsStore class can deal with functions other than those relating to the interface (such as the `getMaker()` method). But, in order to fulfil the requirements of implementing the Product interface, the class *must* override the `getPrice(int)` method. Declaring a method in an interface is a good practice. However, the method cannot be used until a class implements the interface and overrides the given method.

Naming Conventions

The rules for an interface name are identical to those for classes. The only requirements for the name are that it should begin with a letter, an underscore

character, or a dollar sign, contain only Unicode characters (basic letters and digits, as well as some other special characters); and not be the same as any Java keyword (such as *extends* or *int*). Again, like classes, it is a common practice to capitalise the first letter of any interface name.

Extending Other Interfaces

In keeping with the OOP practice of inheritance, Java interfaces may also extend other interfaces as a means of building larger interfaces upon previously developed code. The new sub-interface inherits all the methods and static constants of the super-interfaces just as subclasses inherit the properties of superclasses.

The one major rule that interfaces must obey while extending other interfaces is that they cannot define the body of the parent methods, any more than they can define the body of their own methods. Any class that implements the new interface must define the body of all of the methods for both the parent and child interfaces.

While classes *implement* interfaces to inherit their properties, interfaces *extend* other interfaces. When extending more than one interface, you have to separate each by a comma. This means that while classes cannot extend multiple classes, interfaces are allowed to extend multiple interfaces:

```
Interface MonitoredRunnable extends java.lang.Runnable.java.lang.Cloneable {
    Boolean isRunning()
    {
}
```

Methods in interfaces

The main purpose of interfaces is to declare abstract methods that will be defined in other classes. As a result, if you are dealing with a class that implements an interface, you can be assured that these methods will be defined in the class. While this process is not overly complicated, there is one important difference that should be noticed. The syntax for declaring a method in an interface is extremely similar to declaring a method in a class, but in contrast to methods declared in classes, methods declared in interfaces cannot possess bodies. An interface method consists of only declaration. For example, the following two methods are complete if they are defined in an interface.

```
public int getPrice(int id);
public void showState();
```

However, in a class, they would require method bodies:

```
public int getPrice(int id)
{
    if (id == 1)
        return(500);
    else
        return(100);
}

public void showState()
{
    System.out.println("Massachusetts");
}
```

The method declaration does not determine how a method will behave. It defines how it will be used by defining what information it needs, and what (if any) information will be returned. The method that is actually defined later in a class must have the same properties as you define in the interface. To make the best use of this fact, it is important to carefully consider factors like return type and parameter lists when defining the method in the interface. Method declarations in interfaces have the following syntax:

```
public return_value nameofmethod(parameters) throws ExceptionList;
```

Also note that unlike normal method declarations in classes, declarations in interfaces are immediately followed by a semicolon.

Variables in Interfaces

Although interfaces are generally employed to provide abstract implementation of methods, you may also define variables within them. Because you cannot place any code within the bodies of the methods, all variables declared in an interface must be global to the class.

Furthermore, regardless of the modifiers used when declaring the field, all fields declared in an interface are public, final, and static.

As seen in the Product interface, interface fields like final static fields in classes are used to define constants that can be accessed by all classes that implement the interface.

```
public interface Product  
{  
    //This variable is static and final.  
    Static final String MAKER = “My Corp”;  
  
    //This variable is also static and final by default,  
    //even though not stated explicitly.  
    String PHONE = “91-11-5556666”;  
    Public int getPrice(int id);  
}
```

Modifiers

Methods declared in interfaces are, by default, assigned the public level of access. Consequently, because you cannot override a method to be more private than it already is, all methods declared in interfaces and overridden in classes *must* be assigned the public access modifier, unless they are explicitly made less public in the interface.

Of the remaining modifiers that may be applied to methods, only native and abstract may be applied to methods originally declared in interfaces.

Parameter List

Interface methods define a set of parameters that must be passed to the method. Consequently, declaring a new method with the same name but with a different set of parameters than the method declared in your interface overloads the method, but does not override it.

While there is nothing wrong with overloading methods declared in interfaces, it is also important to implement the method declared in the interface. Therefore, unless you declare your class to be abstract, you must override each method, employing the same parameter signature as in your interface.

Example:

```
public class Runner implements Runnable
{
    //This method overloads the run() method;it does not
    //fulfill the requirements for Runnable.

    public void run(int max)
    {
        int count =0;
        while (count++<max)
        {
            try
            {
                Thread.sleep(500);
            }
            catch (Exception e) {}
        }
    }

    //This method fulfills the requirement for Runnable.
    //You must have this method.
    public void run()
    {
        while(true)
        {
            try
            {
                Thread.sleep(500);
            }
            catch (Exception e) {}
        }
    }
}
```

Body

When creating a class that implements an interface, one of your chief concerns will be creating bodies for the methods originally declared in the interface. Unless you decide to make the method native, it is necessary to create the body for every method originally declared in your interface, if you do not want to make your new class abstract.

The actual implementation and code of the body of your new method is entirely up to you. This is one of the good things about using interfaces. While the interface ensures that in a non-abstract class, its methods will be defined and it will return an appropriate data type, the interface places no further restrictions or limitations on the method bodies.

Using Interfaces from Other Classes

You have learnt how to create interfaces and build classes based on interfaces. However, interfaces are not useful unless you can develop classes that will either employ the derived classes, or the interface itself.

5.6 EXCEPTIONS

For any interface method to throw an exception, the exception type (or one of its superclasses) must be listed in the exception list for the method as defined in the interface. Here are the rules for overriding methods that throw exceptions:

The new exception list may only contain exceptions listed in the original exception list, or subclasses of the originally listed exceptions.

The new exception list does not need to contain any exceptions, regardless of the number listed in the original exception list. This is because the original list is inherently assigned to the new method.

The new method may throw any exception listed in the original exception list or derived from an exception in the original list, regardless of its own exception list.

In general, the exception list of the method, which is declared in the interface (*not* the re-declared method), determines which exceptions can and cannot be thrown. In other words, when a re-declared method changes the exception list, it cannot add any exception that is not included in the original interface declaration.

As an example, examine the interface and method declarations in the following listing:

Alternate Exception Lists

```
interface Example
{
    public int getPrice(int id) throws java.lang.RuntimeException;
}
class User implements Example
{
    public int getPrice(int id) throws java.awt.AWTException
    { //Illegal
        //java.awt.AWTException is not a subclass
        //ofjava.lang.RuntimeException
        //method body
    }
    public int getPrice(int id)
    {
        if (id == 6)
            throw new java.lang.IndexOutOfBoundsException();
        // Legal
        // IndexOutOfBoundsException is derived from      //
        // RuntimeException
    }
    ...
}
public int getPrice(int id) throws java.lang.IndexOutOfBoundsException
{
    //Legal
    //IndexOutOfBoundsException is derived from
    //RuntimeException
    if (id == 6)
```

```

throw new java.lang.ArrayIndexOutOfBoundsException();
    // Legal
    // ArrayIndexOutOfBoundsException is derived from      //
    IndexOutOfBoundsException

...
}

```

Check Your Progress 2

- 1) Define a structure for an interface.

.....

- 2) Describe a package and its relationship with classes.

.....

5.7 SUMMARY

In this Unit, you have learned how to create a package, how to add existing classes to a package, and how to create an interface.

A package is a collection of classes. It provides a convenient way to organise those classes. You can put classes that you developed in packages and distribute packages to others. You can think of packages as libraries to be shared by many users. Java language itself comes with a rich set of packages that you can use to build applications. You have already learned about java language package.

Packages are hierarchical and you can have packages within package. For example, `java.awt.Button` indicates that `Button` is a class in the package `awt` and `awt` is a package within the package `java`.

With interfaces, you can obtain the effect of multiple inheritance. An interface is treated like a special class in java. Each interface is compiled into a separate bytecode file just like a regular class. You cannot create an instance for the interface.

5.8 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) Java provides a powerful mechanism of grouping related classes and interfaces together in a single unit called a package. In other words, packages are groups of related classes and interfaces. Java provides a convenient mechanism for managing a large group of classes and interfaces while avoiding potential naming conflicts.

- 2) An *interface* is a prototype for a class and is useful from a logical design perspective. Interfaces are abstract classes that are left completely unimplemented. *Completely unimplemented* in this case means that **no** methods in the class have been implemented. Additionally, interface member data is limited to static final variables, which means that they are constants.

The benefits of using interfaces are much the same as the benefits of using abstract classes. Interfaces provide a means to define the protocols for a class without worrying about the implementation details.

- 3) The syntax for creating interfaces is as follows:

```
interface Identifier
{
    InterfaceBody
}
```

Identifier is the name of the interface, and *InterfaceBody* refers to the abstract methods and static final variables that make up the interface. Because it is assumed that all the methods in an interface are abstract, it is not necessary to use the *abstract* keyword.

Check Your Progress 2

- 1) The structure of a java interface is similar to that of an abstract class, in that you can have data and methods. The data, however must be constants, and the methods can have only declaration without implementations. The syntax to declare an interface is as follows:

```
modifier interface interfacename
{
    constants declarations;
    methods signatures:
}
```

- 2) A package is a structure for organising classes. JDK1.1 API has more than 477 classes and 22 interfaces that are organized into 25 packages. Programming in Java is essentially using these classes to build your projects.