Unit 5 Inheritance, Package and Interface

Structure:

5.1 Introduction

In the last unit, we have discussed arrays and strings in Java. In this unit, we will explain the concepts of Inheritance, Packages and Interfaces. **Inheritance** can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a ***superclass***. The class that does the inheriting is called a ***subclass***. Therefore, a subclass is a specialized version of a superclass. 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 only exposed to other members of the same package. Using the keyword

interface, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. Once defined then any number of classes can implement an interface.

Objectives:

After studying this unit, you should be able to:

- explain the various types inheritance and its implementation
- define packages and understand CLASSPATH
- define and give the uses of interfaces

5.2 Inheritance

Inheritance is one of the cornerstones of object-oriented programming, because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a ***superclass****.* The class that does the inheriting is called a ***subclass****.* Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements.

5.2.1 Types of Relationships

Relationships are classified as follows:

- A Kind-Of relationship.
- A Is-A relationship.
- A Part-Of-relationship.
- A Has-A relationship.

Consider for a moment the similarities and differences among the following objects/classes: Automobile, Ford, Porsche, Car and Engine. We can make the following observations:

- A truck is a kind of an automobile.
- A car is a (different) kind of an automobile.
- An engine is a part of an automobile.
- An automobile has an engine.
- The ford is a car.

A-Kind-Of Relationship

Taking the example of a human being and an elephant, both are ‘kind-of’ mammals. As human beings and elephants are ‘kind-of’ mammals, they share the attributes and behaviors of mammals. Human being and elephants are subset of the mammals class. The following figure depicts the relationship between the Mammals and Human Being classes:

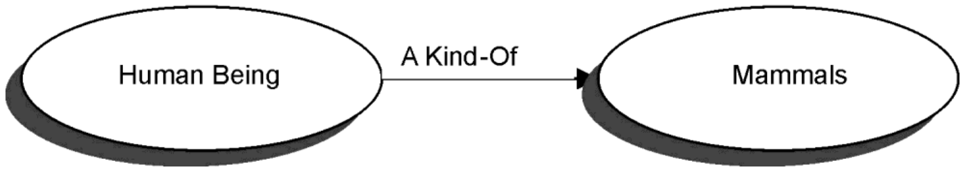


Figure 5.1: Example for A-Kind-Of relationship

Is-A Relationship

Let’s take an instance of the human being class – peter, who ‘is –a’ human being and, therefore, a mammal. The following figure depicts the ‘is –a’ relationship.

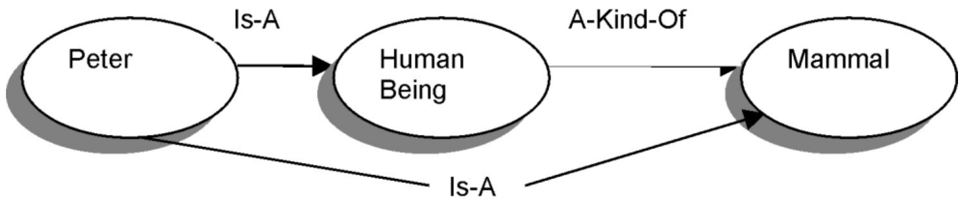


Figure 5.2: Example for Is-A relationship

Has-A Relationship/Part-Of Relationship

A human being has a heart. This represents **has-a** relationship. Heart is a part of the human being. This represents **part-of** relationship. The following figure depicts the relationship between a human being and a heart.

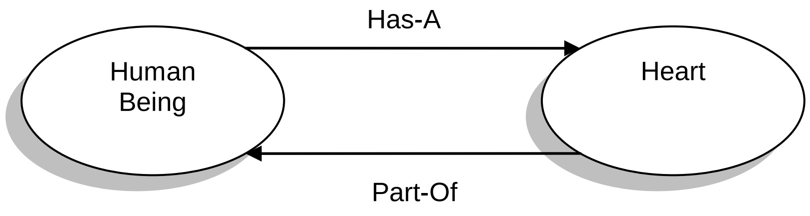


Figure 5.3: Example for Has-A/Part-Of Relationship

5.2.2 What is Inheritance?

The philosophy behind inheritance is to portray things as they exist in the real world. For instance, a child inherits properties from both the parents. Inheritance means that a class derives a set of attributes and related behaviors from another class.

Inheritance helps you to:

- Reduce redundancy in code. Code redundancy means writing the same code in different places, leading to unnecessary replication of code. Inheritance helps you to reuse code.
- Maintain code easily, as the code resides at one place (superclass). Any changes made to the superclass automatically change the behavior automatically.
- Extend the functionality of an existing class by adding more methods to the subclass.

5.2.3 Significance of Generalization

The most important reason for generalization is to make programs extensible. Consider a simple example from the programming world. Assume that you have a program that displays a string, after accepting it from the user. Now, suppose your requirement has changed and you want to display an integer.

Assume that you have Data as a superclass, which has a method to display its value. You also have Character and Float as subclasses of the Data class. To be able to display an integer, all you need to do is to create a class called Integer, which is a subclass of Data.

Adding a new subclass will not affect the existing classes. There is no need to write different methods to display each of these subclasses. Since the superclass has this method, the subclasses inherit the same from it.

Generalization helps in abstraction. A superclass has the attributes and methods, which are the bare essentials for that particular superclass, and are acquired by all its subclasses.

5.2.4 Implementing Inheritance in Java

The ***extends*** keyword is used to derive a class from a superclass, or in other words, extend the functionality of a superclass.

Syntax

public class <subclass_name> extends <superclass_name>

Example

```
public class Confirmed extends Ticket
{
}
```

Rules for Overriding Methods

- The method name and the order of arguments should be identical to that of the superclass method.
- The return type of both the methods must be the same.
- The overriding method cannot be less accessible than the method it overrides. For example, if the method to override is declared as public in the superclass, you cannot override it with the private keyword in the subclass.
- An overriding method cannot raise more exceptions than those raised by the superclass.

Example

```
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
```

```
class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij ( )**. Also, inside **sum ( )**, **i** and **j** can be referred to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a **class** declaration that inherits a superclass is shown here:

class *subclass-name* extends *superclass-name* {
// body of class
}

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. (This differs from C++, in which you can inherit multiple base classes.) You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass.

However, no class can be a superclass of itself.

5.2.5 Access Specifiers

An access specifier determines which features of a class (the class itself, the data members, and the methods) may be used by other classes. Java supports three *access specifiers.*

- public.
- private.
- protected.

The public Access Specifiers

All classes except the inner class (class within classes) can have the public access specifier. You can use a public class, a data member, or a method from any object in any Java program.

Example:

```
public class publicclass

{
        public int publicvaraible;

        public void publicmethod ()

        {

        }

}
```

The private Access Specifier

Only objects of the same class can access a private variable or method. You can declare only variables, methods, and inner classes as private.

Example:

private int privatevariable;

The protected Access Specifier

The variables, methods, and inner classes that are declared protected are accessible to the subclasses of the class in which they are declared.

Example:

protected int protectedvariable;

Default Access

If you do not specify any of the above access specifiers, the scope is friendly. A class, variable, or method that has friendly access is accessible to all the classes of a package.

Consider the following set of classes. Class Y and Z inherit from class X. Class Z belongs to a package different than that of classes X and Y.

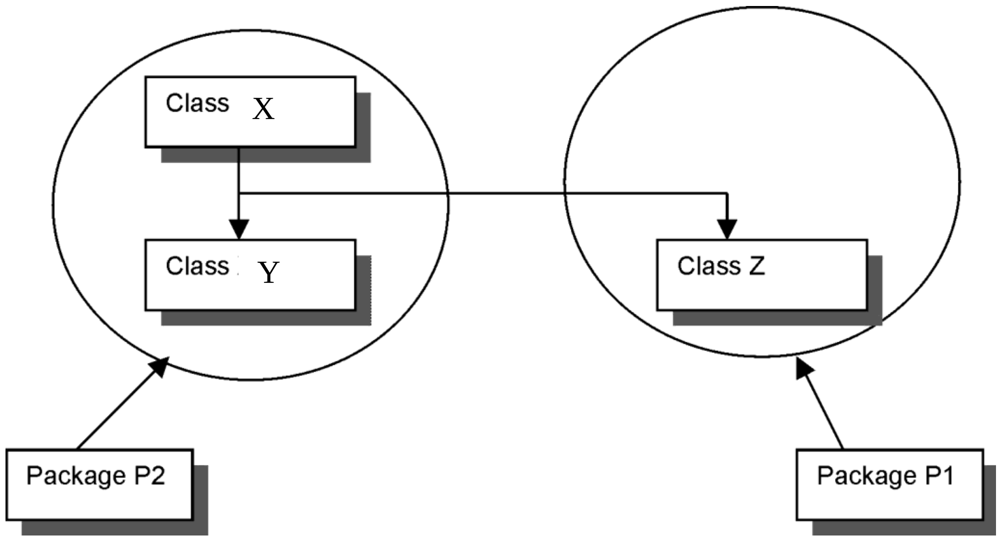


Figure 5.4: Example showing the default access

A method accessMe() has been declared in class X. The following table shows you the accessibility of the method accessMe() from classes Y and Z.

Table 5.1: Table showing the accessibility

Access Specifier	Class Y	Class Z
accessMe () is declared as protected	Accessible, as Y is a subclass	Accessible, as Z is a subclass (event if it is in another package)
accessMe () is declared without an access specifier (friendly)	Accessible, as it is in the same package	Not accessible, as it is not in the same package

You can access a non-private variable or method using an object of the class as shown below:

```
Someclass classobject = new someclass ();
Classobject.publicvariable;
Classobject.protectedmethod();
```

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain private to their class.
This program contains an error and will not compile. */
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

This program will not compile because the reference to **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

Note: A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A Superclass Variable can reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```
class RefDemo {
public static void main(String args[]) {
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
does not define a weight member. */
// System.out.println("Weight of plainbox is " +
plainbox.weight);
}
}
```

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

It is important to understand that it is the type of the reference variable – not the type of the object that it refers to – that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the **weight** field, because it does not define one.

Although the preceding may seem a bit esoteric, it has some important practical applications – two of which are discussed later in this chapter.

Using super

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box( )**. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors

A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

super(*parameter-list*);

Here, *parameter-list* specifies any parameters needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor. To see how **super( )** is used, consider this improved version of the **BoxWeight( )** class:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
double weight; // weight of box
// initialize width, height, and depth using super()
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
}
```

Here, **BoxWeight( )** calls **super( )** with the parameters **w**, **h**, and **d**. This causes the **Box( )** constructor to be called, which initializes **width**, **height**, and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super( )** was called with three arguments. Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that a box can be constructed. In each case, **super( )** is called using the appropriate arguments. Notice that **width**, **height**, and **depth** have been made private within **Box**.

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
```

```
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();
vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();
vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

Pay special attention to this constructor in **BoxWeight( )**:

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
```

Notice that **super( )** is called with an object of type **BoxWeight** – not of type **Box**. This still invokes the constructor **Box(Box ob)**. As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

Let's review the key concepts behind **super( )**. When a subclass calls **super( )**, it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the superclass immediately above the calling class. This is true even in a multi-leveled hierarchy. Also, **super( )** must always be the first statement executed inside a subclass constructor.

A Second Use of super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

5.2.6 The abstract class

An **abstract** class defines common properties and behaviors of other classes. An abstract class is used as a base class to derive specific classes of the same kind. It defines properties common to the classes derived from it. The abstract keyword is used to declare such a class. The classes declared using the abstract keyword cannot be instantiated.

Syntax:

```
abstract class <class_name>
{
}
```

You can also declare abstract methods. Abstract methods have public scope. The code below declares an abstract method for the class shape.

```
abstract class Shape
{
        public abstract float calculateArea ();
}
```

The abstract method calculateArea(), given above, is inherited by the subclasses of the **Shape** class. The subclasses Rectangle, Circle and Hexagon implement this method in different ways.

```
public class Circle extends Shape
{
        float radius;

        public float calculateArea ()
        {
                return radius*22/7;
        }
}
```

In the above example, the calculateArea () method has been overridden in the circle class. If the method is not overridden, the class will inherit the abstract method from the parent class. Any class that has an abstract method is abstract. Hence, you would not be able to create an object of the circle class. Therefore, it is necessary to override the calculateArea() method in the circle class.

The *final* Keyword

A class called password authenticates user login. You do not want anybody to change the functionality of the class by extending it. To prevent inheritance, use the final modifier.

Example:

```
final class password

{

}
```

You will also find final classes in JDK package. For example, the **java.lang.String** class has been declared final. This is done for security reasons. It ensures that any method that refers to the String class gets the actual String class and not a modified one.

Self Assessment Questions

1. ________ is one of the features of object oriented programming that allows the creation of hierarchical classifications.
2. A class that is inherited is called a ________.
3. The class that does the inheriting is called a ________.
4. Engine has a __________ relationship with automobile.
5. ________, ________ and ________ are the three types of access specifiers in Java.

5.3 Packages

In the preceding section, the name of each example 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. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name "Foobar" as a class name. Or, imagine the entire Internet community arguing over ‘who first named a class Espresso.’) Thankfully, 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 only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

5.3.1 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. Most of the time, you will define a package for your code. 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. 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. Most real-world packages are spread across many files.

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 multi-leveled 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**, **java\\awt\\image**, or **java:awt:image** on your **UNIX, Windows**, or **Macintosh** file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

5.3.2 Understanding CLASSPATH

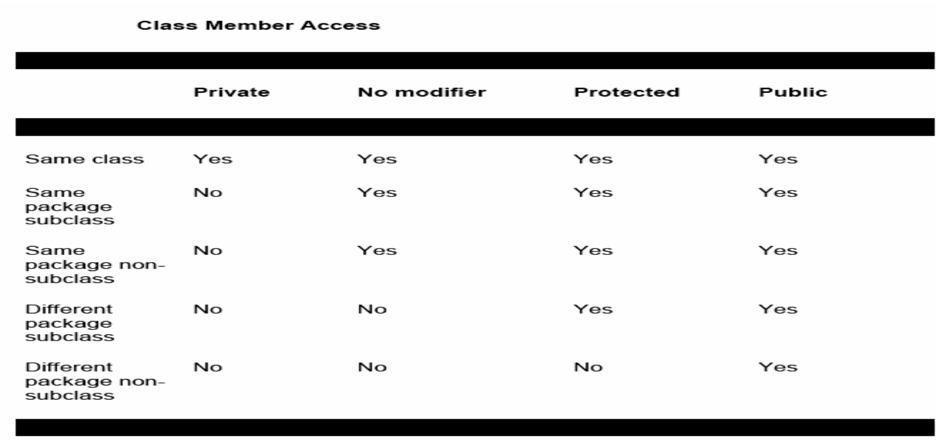
Before an example that uses a package is presented, a brief discussion of the **CLASSPATH** environmental variable is required. While packages solve many problems from an access control and name-space-collision perspective, they cause some curious difficulties when you compile and run programs. This is because the specific location that the Java compiler will consider as the root of any package hierarchy is controlled by **CLASSPATH**. Until now, you have been storing all of your classes in the same, unnamed default package. Doing so allowed you to simply compile the source code and run the Java interpreter on the result by naming the class on the command line. This worked because the default current working directory (**.**) is usually in the **CLASSPATH** environmental variable defined for the Java run-time system, by default. However, things are not so easy when packages are involved.

Assume that you create a class called **PackTest** in a package called **test**. Since your directory structure must match your packages, you create a directory called **test** and put **PackTest.java** inside that directory. You then make **test** the current directory and compile **PackTest.java**. This results in **PackTest.class** being stored in the **test** directory, as it should be. When you try to run **PackTest**, though, the Java interpreter reports an error message similar to "can't find class PackTest." This is because the class is now stored in a package called **test**. You can no longer refer to it simply as **PackTest**. You must refer to the class by enumerating its package hierarchy, separating the packages with dots. This class must now be called **test.PackTest**. However, if you try to use **test.PackTest**, you will still receive an error message similar to "can't find class test/PackTest."

The reason you still receive an error message is hidden in your **CLASSPATH** variable. Remember, **CLASSPATH** sets the top of the class hierarchy. The problem is that there's no **test** directory in the current working directory, because *you are in* the **test** directory, itself.

You have two choices at this point: change directories up one level and try **java test.PackTest**, or add the top of your development class hierarchy to the **CLASSPATH** environmental variable.

Table 5.2: Table showing the Class Member Access



Class Member Access

	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

Self Assessment Questions

6. ___________ is the mechanism in Java through which class name space is partitioned into more manageable chunks.
7. ___________ keyword is used to define a package.
8. In Java, multi-leveled packages can be created. (True / False)
9. The specific location that the Java compiler will consider as the root of any package hierarchy is controlled by ___________.
10. ________ symbol denotes the current working directory.

5.4 Interface

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**.

Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

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 so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and non-extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. 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. This is where the real power of interfaces is realized.

Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritances in a language such as C++.

5.4.1 Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
return-type method-name1 (parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN (parameter-list);
type final-varnameN = value;
}
```

Here, *access* is either **public** or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. ***name*** is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. 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.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Here is an example of an interface definition. It declares a simple interface which contains one method called **callback ( )** that takes a single integer parameter.

```
interface Callback {
void callback (int param);
}
```

5.4.2 Some uses of interfaces

- ✓ Interfaces are a way of saying, "You need to plug some code in here for this thing to fully work ". The interfaces specify the exact signatures of the methods that must be provided.
- ✓ Use the interface type as a parameter for a method. Inside the method you can invoke any of the methods promised by the interface parameter. When you actually call the method, you will have to provide an object that has all the methods promised by the interface. It might help to think of this as a bit like passing in a subclass as an argument and having the correct overriding methods called.
- ✓ Interfaces can be used to mix in generally useful constants. So, for example, if an interface defines a set of constants, and then multiple classes use those constants, the values of those constants could be globally changed without having to modify multiple classes. That is interfaces separate design from implementation.

5.4.3 Interfaces versus Abstract Classes

While an interface is used to specify the form that something must have, it does not actually provide the implementation for it. In this sense, an interface is a little like an abstract class that must be extended in exactly the manner that its abstract methods specify.

- An abstract class is an incomplete class that requires further specialization. An interface is just a specification or prescription for behavior.
- An interface does not have any overtones of specialization that are present with inheritance.
- A class can implement several interfaces at once, whereas a class can extend only one parent class.
- Interfaces can be used to support callbacks (inheritance does not help with this). This is a significant coding idiom. It essentially provides a pointer to a function, but in a type-safe way.

Self Assessment Questions

11. __________ is the mechanism through which you can specify what a class must do, but not how it does it.
12. Variables declared inside the interface declarations are implicitly __________ and __________.

5.5 Summary

In this unit, we have discussed the following:

Identifying Relationships

Relationships are of the following types:

- A Kind-Of relationship.
- A Is-A relationship.
- A Part-Of relationship.
- A Has-A relationship.

Implementing Inheritance in Java

The ***extends*** keyword is used to derive a class from a superclass, or in other words, extends the functionality of a superclass.

Identifying and Applying Constraints

An access specifier determines which features of a class (the class itself, the data members, and the methods) may be used by other classes. Java supports three access specifiers (also known as access modifiers).

- public.
- private.
- protected.

Using final and abstract classes

The abstract keyword

The abstract keyword is used to declare classes that define common properties and behavior of other classes. An abstract is used as a base class to derive specific classes of the same kind.

The final keyword

A class can be declared as final if you do not want the class to be sub-classed.

5.6 Terminal Questions

1. What are the different types of relationships?
2. How do you implement inheritance in Java?
3. What are the rules for overriding a method in Java?
4. What are the uses of interfaces?
5. What are the differences between an interface and an abstract class?

5.7 Answers

Self Assessment Questions

1. inheritance
2. superclass
3. subclass
4. part-of
5. public, private, protected
6. package
7. package
8. true
9. CLASSPATH
10. . (dot)
11. Interface
12. final, static

Terminal Questions

1. A-Kind-Of, Is-A, Part-Of, Has-A. (Refer section 5.2.1)
2. public class <subclass_name> extends <superclass_name>

   {

   } (Refer section 5.2.4)
3. **Rules for Overriding Methods**
   - ❖ The method name and the order of arguments should be identical to that of the superclass method.
   - ❖ The return type of both the methods must be the same.
   - ❖ The overriding method cannot be less accessible than the method it overrides. For example, if the method to override is declared as public in the superclass, you cannot override it with the private keyword in the subclass.
   - ❖ An overriding method cannot raise more exceptions than those raised by the superclass. (Refer section 5.2.4)
4. **Uses of Interfaces**
   - ✓ Interfaces can be used to mix in generally useful constants. If an interface defines a set of constants, and then multiple classes use those constants, the values of those constants could be globally changed without having to modify multiple classes. That is interfaces separate design from implementation. (Refer section 5.4.2)
5. **Interface vs Abstract Class**
   - ▪ An abstract class is an incomplete class that requires further specialization. An interface is just a specification or prescription for behavior.
   - ▪ An interface does not have any overtones of specialization that are present with inheritance. (Refer section 5.4.3)