Inheritance

In the last unit, we 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.

Objectives

Upon completion of this topic, we will be able to:

- Explain the various types inheritance and its implementation

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.

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

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 behaviours 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 changes the behaviour automatically.
- Extend the functionality of an existing class by adding more methods to the subclass.

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 of its subclasses.

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 a 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.

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.

Example showing the default access

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

Chart showing the accessibility

Access Specifier - Class Y - Class Z
accessMe () is declared as protectedAccessible, 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 the code that is 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 would want to create a superclass that keeps the details of its implementation to itself (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 the 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 of its uses are examined here.

Using super to Call Superclass Constructors

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

```
super(parameter-list);
```

Here, parameter-list specifies any of the parameters required 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 by 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, a member can either be a method or an instance variable.

This second form of super is most applicable to situations in which the member names of a subclass hides the 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 would notice, super can also be used to call methods that are hidden by a subclass.

The abstract class

An abstract class defines common properties and behaviours 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 the 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.

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

The abstract keyword

The abstract keyword is used to declare classes that define common properties and the behaviour 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.