Module 3 - Introduction

Reusability is yet another concept of OOP paradigm. it is always nice if we could reuse something that already exist rather than creating the same all over again. Java supports this concept , Java classes can be reused in several ways by the principle of inheritance.

What is Inheritance

The process of creating new classes, by reusing the properties of existing ones. This mechanism of deriving a new class from an old one is called Inheritance.

The existing class is known as the **base class** or **super class** or **parent class** and the new one is called the **sub class** or **derived class** or **child class.**

The inheritance allows the subclasses to inherit all the fields and methods of their parent classes. inheritance may take different forms:

1. single inheritance (a sub class derived by only one super class)

2. Multiple inheritance (a sub class derived by several super classes)

3. Multilevel inheritance (a sub class derived from another subclass)

4. Hierarchal inheritance (several sub classes deriving one super class)

Note: Java does not support directly implement multiple inheritance. However this concept is implemented by the concept of **Interfaces**.

Defining a Subclass

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

```
class  SubClass-Name  extends  SuperClass-Name
{
     fields declarations;
     methods declarations;
}
```

The keyword **"extends"** signifies that the properties of the SuperClass-Name are extended to the SubClass-Name. The subclass will now contain its own fields and methods as well as those of the superclass.

Program to demonstrate single inheritance

```
class A                  //Base class
{
     int x;
     void setX()
     {
          x=10;
     }
void print()
     {
          System.out.println("Value of x:"+x);
     }
}

class B extends A         //Derived class
{
     int y;
     void setY()
     {
          y=20;
     }

     void display()
     {
          System.out.println("Value of x:"+x);
          System.out.println("Value of y:"+y);
          System.out.println("Value of x+y:"+(x+y));
     }
}

class DemoSingle
{
     public static void main(String args[])
     {
          B ob1=new B();
          B ob2=new B();

          System.out.println("Using ob1 of class B");
          ob1.setX();
          ob1.setY();
          ob1.print();
          ob1.display();

          System.out.println("Using ob2 of class B");
          ob2.x=50;            //accessing x  of base class A
          ob2.y=100;           //accessing y of sub class B
          ob2.display();
     }
}
```

```
// Another  simple example of inheritance.
// 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.

Member Access and Inheritance

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.

Constructors in the derived class (Constructors in extended classes)

Consider the following example:

```
class X
{
     int a;
     X(){ a=10; }
}
class Y extends X
{
     int b;
     Y(){ b=20; }
     void show()
     {
          System.out.println("a="+a+"and b="+b);
     }
}
class Test
{
     public static void main(String args[])
     {
          Y y1=new Y();
          y1.show();
     }
}
```

Notice Default constructors in both the base and derived classes

It is already seen that whenever an object is created for a class, the constructor of that class is invoked on its own. In the above program though an object is created for derived class, ***with the constructor order dependency***, first the constructor in base class is invoked and later the constructor of the derived class is executed, this is because both base class and derived class have default constructors.

Another example showing constructor invocation order

When a class hierarchy is created, in what order the constructors for the classses are invoked that makeup hierarchy? Here constructors are called in the order of their derivation from superclass to subclass.

```
class A
{
   A() {System.out.println("Inside class A");}
}

class B extends A
{
   B(){System.out.println("Inside class B");}
}
```

```
class C extends B
{
   C(){System.out.println("Inside class C");}
}
class DemoCons
{
   public static void main(String a[])
   {   C obj=new C();
   }
}
```

The output will be:

```
Inside class A
Inside class B
Inside class C
```

If a base class contains parameterized constructors ???

In case if the base class has a parameterized constructor, it becomes compulsory to have a parameterized constructor in derived class, thus when both base class and derived class has parameterized constructor, the constructor of the class for which object is created only that is invoked, that is during inheritance since objects are created only for derived class, so only derived class constructor is invoked.

But logically, the constructor of the derived class should not be executed at the beginning and only the constructors in their order of derivation should be executed from super class to subclass. because the super class fields remain uninitialized. if super class constructor is not invoked in the beginning.

To can be overcome by-

1. Intializing base class fields again in subclass -which makes duplication of code

2. use ***super*** keyword, -which calls immediate super class constructor

The super keyword

This is one of the keywords available in Java- super keyword can be used for three purposes-

1. ***To invoke super class (base class) constructors*** (during inheritance, when super class has parameterized constructor)

2. ***To invoke super class (base class) methods*** (during method overriding, when method of superclass is hidden)

3. ***To access super class (base class) fields*** (when both super class and subclass has same fields)

Using super to invoke super class (base class) constructors

when the base class and the derived class contains parameterized constructor, the constructors are not executed in their order of derivation. Instead the derived class constructor is invoked prior to the base class constructor. In order to invoke base class constructors first, we can use the super keyword.

General format:

super(arg-list)

Here, ***arg-list*** specifies any arguments needed by the constructor in the superclass. **super( )** statement must always be the first statement executed inside a subclass' constructor. consider the following program-

```
//super class
class A
{
     int x;
     A(int p)             //parameterized constructor
     {
          x=p;
     }
}
//derived class
class B extends A
{
     int y;
     B(int q, int r)      //parameterized constructor
     {
          super(q);       //invokes constructor of superclass A
          y=r;
     }
     void show()
     {
          System.out.println("a="+a+"and b="+b);
     }
}
class DemoSuper1
{
     public static void main()
     {
          B ob1=new B(25,50);
          ob1.show();
     }
}
```

A more practical example

```
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();
}
}
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method overriding.
class A
{
     int i, j;
     A(int a, int b)
     {
          i = a;
          j = b;
     }
     // display i and j
     void show()
     {
          System.out.println("i and j: " + i + " " + j);
     }
}
class B extends A
{
     int k;
     B(int a, int b, int c)
     {
          super(a, b);
          k = c;
     }
     // display k – this overrides show() in A
     void show()
     {
          System.out.println("k: " + k);
     }
}

class Override
{
     public static void main(String args[])
     {
          B subOb = new B(1, 2, 3);
          subOb.show(); // this calls show() in B
     }
}
```

The output produced by this program is shown here:

```
k: 3
```

When show( ) is invoked on an object of type B, the version of show( ) defined within B is used. That is, the version of show( ) inside B overrides the version declared in A. If you wish to access the superclass version of an overridden method, you can do so by using super. For example, in this version of B, the superclass version of show( ) is invoked within the subclass' version. This allows all instance variables to be displayed.

using super to invoke base class methods during overriding

General format**: super.methodname( );**

```
class B extends A
{
     int k;
     B(int a, int b, int c)
     {
          super(a, b);
          k = c;
     }
     void show()
     {
          super.show();  // this calls A's show()
          System.out.println("k: " + k);
     }
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

```
      i and j: 1 2
      k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**.

Using super to access base class fields

when ***super*** is used to access base class fields, it 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

Use of ***super*** to access base class fields 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:

program showing use of super to overcome name hiding

```
class X
{
     int a;
     X()
     {
          a=10;
     }
}
```

Notice : field 'a' being used in both base class and derived class

```
class Y extends X
{
     int a;          //this 'a' hides the 'a' in X
     Y()
     {
          a=20;
     }
     void show()
     {
```

gets 'a' of base class

```
          System.out.println("base class a="+super.a);
          System.out.println("derived class a="+a);
     }
}
class Test
{
     public static void main(String a[])
     {
          Y yobj=new Y();
          yobj.show();
     }
}
```

```
// Another example 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.

Multilevel Inheritance

Creating a Multilevel Hierarchy

Until now, we have been using single inheritance that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, C inherits all aspects of B and A.

To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass BoxWeight is used as a superclass to create the subclass called Shipment. Shipment inherits all of the traits of BoxWeight and Box, and adds a field called cost, which holds the cost of shipping such a parcel.

```
// Extend BoxWeight to include shipping costs.
// Start with Box.
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;
     }
}

// Add weight.
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;
        }
   }

   // Add shipping costs.
   class Shipment extends BoxWeight
   {
        double cost;

        // construct clone of an object
        Shipment(Shipment ob) // pass object to constructor
        {
             super(ob);
             cost = ob.cost;
        }
        // constructor when all parameters are specified
        Shipment(double w, double h, double d,double m, double c)
        {
             super(w, h, d, m); // call superclass constructor
             cost = c;
        }
        // default constructor
        Shipment()
        {
             super();
             cost = -1;
        }
        // constructor used when cube is created
        Shipment(double len, double m, double c)
        {
             super(len, m);
             cost = c;
        }
   }

   class DemoShipment
   {
        public static void main(String args[])
```

```
      {
           Shipment shipment1 =
           new Shipment(10, 20, 15, 10, 3.41);
           Shipment shipment2 =
           new Shipment(2, 3, 4, 0.76, 1.28);

           double vol;
           vol = shipment1.volume();
           System.out.println("Volume of shipment1 is " + vol);
           System.out.println("Weight of shipment1 is "
           + shipment1.weight);
      System.out.println("Shipping cost: $" + shipment1.cost);
      System.out.println();

           vol = shipment2.volume();
           System.out.println("Volume of shipment2 is " + vol);
           System.out.println("Weight of shipment2 is "
           + shipment2.weight);
      System.out.println("Shipping cost: $" + shipment2.cost);
      }
}
```

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

Using final with inheritance

The final keyword is another new keyword introduced in java. The keyword final has three uses.

1) First, it can be used to create the equivalent of a named constant.

The other two uses of final apply to inheritance.

2) To prevent method overriding.

3) To prevent or stop inheritance.

Using final to Prevent Overriding

While method overriding is one of Java’s most powerful features, there will be times when you will want to prevent it from occurring.

To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. **Methods declared as final cannot be overridden.**

The following fragment illustrates **final**:

```
class A
{
     final void meth()
     {
          System.out.println("This is a final method.");
     }
}
class B extends A
{
          void meth()  // ERROR! Can't override.
          {
               System.out.println("Illegal!");
          }
}
```

Because meth( ) is declared as final, it cannot be overridden in B. If you attempt to do so, a **compile-time error** will result.

Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with final methods.

Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called ***early binding***.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a final class:

```
final class A
{
     // ...
}
// The following class is illegal.
class B extends A
{    // ERROR! Can't subclass A
     // ...
}
```