**Inheritance Introduction:**

Inheritance is one of the principles 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. The Inheritance tells the relationship between the Super class and sub class. The relationship is called as "**is-a**" relationship. 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.

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

class superclass-name

{

//bocy of super class

}

class subclass-name extends superclass-name

{

// body of sub class

}

**// program to create a class that extends another class**

class SuperClass

{

double length,breadth;

void setLB(double l, double b)

{

length = l;

breadth = b;

}

void disLB()

{

System.out.println("From Super Class...:");

System.out.println("Length: " + length + "\nBreadth: " + breadth);

}

void area()

{

System.out.println("The area from Sup Class area meth: " + length\*breadth);

}

}

class SubClass extends SuperClass

{

double height;

void setLBH(double l, double b, double h)

{

setLB(l,b);

height = h;

}

void disLBH()

{

System.out.println("From Sub Class...:");

disLB();

System.out.println("Height: " + height);

}

void volume()

{

System.out.println("The Volume from Sub class: " + length\*breadth\*height);

}

}

class InheritanceDemo1

{

public static void main(String args[])

{

SubClass subObj = new SubClass();

// subObj.setLBH(2.0,4.5,3.0);

subObj.length = 2;

subObj.breadth = 1;

subObj.height = 3;

subObj.disLB();

subObj.disLBH();

SuperClass supObj = new SuperClass();

supObj.length = 4;

supObj.breadth = 5;

supObj.disLB();

}

}

1:

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.

**// program to create a class that extends another class**

class SuperClass

{

double length,breadth;

void setLB(double l, double b)

{

length = l;

breadth = b;

}

void disLB()

{

System.out.println("From Super Class...:");

System.out.println("Length: " + length + "\nBreadth: " + breadth);

}

void area()

{

System.out.println("The area from Sup Class area meth: " + length\*breadth);

}

}

class SubClass extends SuperClass

{

double height;

void setLBH(double l, double b, double h)

{

setLB(l,b);

// length = l;

// breadth = b;

height = h;

}

void disLBH()

{

System.out.println("From Sub Class...:");

disLB();

System.out.println("Height: " + height);

}

void volume()

{

System.out.println("The Volume from Sub class: " + length\*breadth\*height);

}

}

class SubSubClass extends SubClass

{

double cost;

void disC()

{

System.out.println("The Cost from Sub Sub Class : " + cost);

}

}

class InheritanceDemo2

{

public static void main(String args[])

{

SuperClass supObj = new SuperClass();

supObj.length = 4;

supObj.breadth = 5;

supObj.disLB();

SubClass subObj = new SubClass();

// subObj.setLBH(2.0,4.5,3.0);

subObj.length = 2;

subObj.breadth = 1;

subObj.height = 3;

subObj.disLB();

subObj.disLBH();

SubSubClass subsubObj = new SubSubClass();

subsubObj.length = 22;

subsubObj.breadth = 11;

subsubObj.height = 33;

// subsubObj.setLBH(11.1,22.2,33.3);

subsubObj.cost = 35.98;

subsubObj.disLBH();

subsubObj.disC();

}

}

2: 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. (In a class hierarchy, private members remain private to their class.)

**/\* 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);

}

}

3: 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.

class SuperClass

{

double length,breadth;

void setLB(double l, double b)

{

length = l;

breadth = b;

}

void disLB()

{

System.out.println("From Super Class...: ");

System.out.println("Length: " + length + "\nBreadth: " + breadth);

}

void area()

{

System.out.println("The area from Sup Class area meth: " + length\*breadth);

}

}

class SubClass extends SuperClass

{

double height;

void setLBH(double l, double b, double h)

{

setLB(l,b);

// length = l;

// breadth = b;

height = h;

}

void disLBH()

{

System.out.println("From Sub Class...:");

disLB();

System.out.println("Height: " + height);

}

void volume()

{

System.out.println("The Volume from Sub class: " + length\*breadth\*height);

}

}

class InheritanceDemo3

{

public static void main(String args[])

{

SuperClass supObj;

SubClass subObj = new SubClass();

subObj.setLBH(2.0,4.5,3.0);

subObj.disLB();

subObj.disLBH();

supObj = subObj;

supObj.length = 3;

supObj.breadth = 4;

// supObj.height = 5; Error

supObj.disLB();

// supObj.disLBH(); Error Because it cannot have the knowledge of the sub class members(Can not find sysmbol)

}

}

Creating a Multilevel Hierarchy:

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.

class SuperClass

{

double length,breadth;

void setLB(double l, double b)

{

length = l;

breadth = b;

}

void disLB()

{

System.out.println("From Super Class...:");

System.out.println("Length: " + length + "\nBreadth: " + breadth);

}

void area()

{

System.out.println("The area from Sup Class area meth: " + length\*breadth);

}

}

class SubClass extends SuperClass

{

double height;

void setLBH(double l, double b, double h)

{

setLB(l,b);

// length = l;

// breadth = b;

height = h;

}

void disLBH()

{

System.out.println("From Sub Class...:");

disLB();

System.out.println("Height: " + height);

}

void volume()

{

System.out.println("The Volume from Sub class: " + length\*breadth\*height);

}

}

class SubSubClass extends SubClass

{

double cost;

void disC()

{

System.out.println("The Cost from Sub Sub Class : " + cost);

}

}

class InheritanceDemo2

{

public static void main(String args[])

{

SuperClass supObj = new SuperClass();

supObj.length = 4;

supObj.breadth = 5;

supObj.disLB();

SubClass subObj = new SubClass();

// subObj.setLBH(2.0,4.5,3.0);

subObj.length = 2;

subObj.breadth = 1;

subObj.height = 3;

subObj.disLB();

subObj.disLBH();

SubSubClass subsubObj = new SubSubClass();

subsubObj.length = 22;

subsubObj.breadth = 11;

subsubObj.height = 33;

// subsubObj.setLBH(11.1,22.2,33.3);

subsubObj.cost = 35.98;

subsubObj.disLBH();

subsubObj.disC();

}

}

**Constructors Calling:**

In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.

class FirstClass

{

double dVar;

FirstClass()

{

System.out.println("In FirstClass Constructor...");

dVar = 0;

}

FirstClass(double d)

{

dVar = d;

}

}

class SecondClass extends FirstClass

{

double dVarN;

SecondClass()

{

System.out.println("In SecondClass Constructor...");

dVarN = 0;

}

SecondClass(double d1, double d2)

{

dVar = d1;

dVarN = d2;

}

}

class ThirdClass extends SecondClass

{

double dVarN1;

ThirdClass()

{

System.out.println("In ThirdClass Constructor...");

dVarN1 = 0;

}

}

class InheritanceDemo4

{

public static void main(String args[])

{

// FirstClass fc = new FirstClass();

// SecondClass sc = new SecondClass();

ThirdClass tc = new ThirdClass();

}

}

**Using super:**

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.

**Using super to Call Superclass Constructors:**

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

super(arg-list);

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

**// super Method Demo**

class Box

{

private double length,width,height;

Box()

{

length = width = height = 1;

}

Box(double len)

{

length = width = height = len;

}

Box(double l,double w,double h)

{

this.length = l;

this.width = w;

this.height = h;

}

}

class BoxWeight extends Box

{

double weight;

BoxWeight()

{

super(); //To call Default Super class constructor

// Box(); Error: Can'n Find Symbol

// length = width = height = 1;

weight = 1;

}

BoxWeight(double len,double wt)

{

super(len);

// length = width = height = len;

weight = wt;

}

BoxWeight(double l,double w,double h,double wt)

{

super(l,w,h);

// length = l;

// width = w;

// height = h;

weight = wt;

}

void area()

{

System.out.println("Area: " + length\*width);

}

void volume()

{

System.out.println("Volume: " + length\*width\*height);

}

void showWeight()

{

System.out.println("The Weight is : " + weight);

}

}

class SuperMethodDemo

{

public static void main(String args[])

{

BoxWeight defaultBW = new BoxWeight();

System.out.println("The defaultBW object :");

defaultBW.area();

defaultBW.volume();

defaultBW.showWeight();

BoxWeight cube = new BoxWeight(2.4,7.0);

System.out.println("The cube object :");

cube.area();

cube.volume();

cube.showWeight();

BoxWeight bw = new BoxWeight(2.2,3.3,4.4,52.9);

System.out.println("The bw object :");

bw.area();

bw.volume();

bw.showWeight();

}

}

**A Second Use for 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.

class ExampleOne

{

int i,j;

ExampleOne()

{

i = 20;

j = 15;

}

}

class ExampleTwo extends ExampleOne

{

int i;

ExampleTwo()

{

i = 50;

}

void changeSuper(int ival)

{

super.i = ival;

}

void show()

{

System.out.println("The variable i in Super class: " + super.i + "\nThe variable j in Super class: " + super.j + "\nThe variable i in Sub class: " + i);

}

}

class SuperDemo

{

public static void main(String arg[])

{

ExampleTwo ex21 = new ExampleTwo();

ex21.show();

System.out.println("The Values After Changing:");

ex21.i = 50;

ex21.changeSuper(10);

ex21.j = 25;

ex21.show();

}

}

**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.

Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

class FirstClass

{

int i , j;

FirstClass()

{

i = j = 1;

}

void show()

{

System.out.println("The i and j values of First Class: " + i + " , " + j);

}

}

class SecondClass extends FirstClass

{

int k;

SecondClass()

{

i = 2; j = 2; k =1;

}

// This is overrided show method from super class

void show()

{

super.show();

System.out.println("The k value in B: " + k);

}

// This is overloaded show method in second class

void show(String o)

{

System.out.println(o + " : " + k);

}

}

class MethodOverriding

{

public static void main(String args[])

{

FirstClass ob1 = new FirstClass();

ob1.show();

SecondClass ob = new SecondClass();

ob.show();

ob.show("From SecondClass Show Method...");

}

}

**// program to illustrate the method overriding concept with wide example...**

class TwoDFigure

{

double dim1, dim2;

TwoDFigure()

{

dim1 = -1;

dim2 = -1;

}

TwoDFigure(double d1, double d2)

{

dim1 = d1;

dim2 = d2;

}

double area()

{

System.out.println("From Figure area Method: ");

return 0;

}

}

class RectangleFigure extends TwoDFigure

{

RectangleFigure()

{

super();

}

RectangleFigure(double d1, double d2)

{

super(d1,d2);

}

double area()

{

System.out.println("From RectangleFigure area Method: ");

return dim1\*dim2;

}

}

class TriangleFigure extends TwoDFigure

{

TriangleFigure()

{

super();

}

TriangleFigure(double d1, double d2)

{

super(d1, d2);

}

double area()

{

System.out.println("From TriangleFigure area Method: ");

return (1/2)\*dim1\*dim2;

}

}

class MethodOverridingExample

{

public static void main(String args[])

{

TwoDFigure tdf = new TwoDFigure();

System.out.println(tdf.area());

RectangleFigure rf = new RectangleFigure();

System.out.println(rf.area());

TriangleFigure tf = new TriangleFigure();

System.out.println(tf.area());

}

}

**Dynamic Method Dispatch:**

Method overriding forms the basis for one of Java’s most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Super class reference can refer to the sub class object.

We can assign the sub class object to the super class reference (object variable).

Using the super class reference we can call the overridden method in sub class. But, which version of the method to be executed will be decided at the run time based on the type of the object (the reference of the object) it is called by.

// Dynamic Method Dispatch

class A

{

void callme()

{

System.out.println("Inside A's callme method");

}

}

class B extends A

{

// override callme()

void callme()

{

System.out.println("Inside B's callme method");

}

}

class C extends A

{

// override callme()

void callme()

{

System.out.println("Inside C's callme method");

}

}

class Dispatch

{

public static void main(String args[])

{

A a = new A(); // object of type A

B b = new B(); // object of type B

C c = new C(); // object of type C

A r; // obtain a reference of type Ar = a; // r refers to an A object

r.callme(); // calls A's version of callme

r = b; // r refers to a B object

r.callme(); // calls B's version of callme

r = c; // r refers to a C object

r.callme(); // calls C's version of callme

}

}

**Using Abstract Keyword with Classes:**

Sometimes you will want to create a super class that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.

You can require that certain methods be overridden by subclasses by specifying the abstract type modifier. A subclass must override them—it cannot simply use the version defined in the super class. To declare an abstract method, use this general form:

**abstract type name(parameter-list);**

As you can see, no method body is present. Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.

abstract class A

{

abstract void callme();

// concrete methods are still allowed in abstract classes

void callmetoo()

{

System.out.println("This is a concrete method.");

}

}

class B extends A

{

void callme()

{

System.out.println("B's implementation of callme.");

}

}

class AbstractDemo

{

public static void main(String args[])

{

B b = new B();

b.callme();

b.callmetoo();

}

}

Example:

abstract class AbstractFigure

{

double length, width, height;

AbstractFigure()

{

length = width = height = -1;

}

abstract void area();

abstract void volume();

}

class Rectangle extends AbstractFigure

{

Rectangle(double l,double w,double h)

{

length = l;

width = w;

height = h;

}

void area()

{

System.out.println("The area of Rectangle is: " + length\*width);

}

void volume()

{

System.out.println("The volume of the Rectangle is: " + length\*width\*height);

}

}

class AbstactMethodDemo2

{

public static void main(String args[])

{

Rectangle r = new Rectangle(2.5,4.5,6.5);

r.area();

r.volume();

}

}

You cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

abstract class AbstractFigure

{

double length, width, height;

AbstractFigure()

{

length = width = height = -1;

}

abstract void area();

abstract void volume();

}

abstract class Rectangle extends AbstractFigure

{

Rectangle(double l,double w)

{

length = l;

width = w;

}

void area()

{

System.out.println("The area of Rectangle is: " + length\*width);

}

abstract void volume();

/\* {

// System.out.println("The volume of the Rectangle is: " + length\*width\*height);

}\*/

}

class AbstractMethodDemo

{

public static void main(String args[])

{

Rectangle r = new Rectangle(2.5,4.5);

r.area();

}

}

**Final keyword with the Inheritance:**

The keyword final has three uses. First, it can be used to create the equivalent of a named constant.

**final to Prevent Overriding**

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.

class A

{

int i, j;

A()

{

i = j = -1;

}

final void setValues(int ival,int jval)

{

i = ival;

j = jval;

}

}

class B extends A

{

int i,j;

/\* void setValues(int ii,int jj)

{

i = ii;

j = jj;

}

\*/ // setValues method cannot be overridden because setValues in the super class is "final"

void showIJ()

{

System.out.println("The i and j values are: " + i + ", " + j);

}

}

**final to Prevent Inheritance**

The classes that are said to be final cannot be "inherited". To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final.

final class A

{

int i, j;

A()

{

i = j = -1;

}

final void setValues(int ival, int jval)

{

i = ival;

j = jval;

}

}

class B /\*extends A\*/ // B can not inherit the Class A, because A is "final"

{

int i,j;

/\* void setValues(int ii,int jj)

{

i = ii;

j = jj;

}

\*/ // setValues method can not be overridden because setValues in the super class is "final"

void showIJ()

{

System.out.println("The i and j values are: " + i + ", " + j);

}

}

**The Object Class:**

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a super class of all other classes. This means that a reference variable of type Object can refer to an object of any other class.

Object defines the following methods, which means that they are available in every object.

Method Purpose

**Object clone( )** Creates a new object that is the same as the object being cloned.

**boolean equals(Object object )** Determines whether one object is equal to another.

**void finalize( )** Called before an unused object is recycled.

**Class getClass( )** Obtains the class of an object at run time.

**int hashCode( )** Returns the hash code associated with the invoking object.

**void notify( )** Resumes execution of a thread waiting on the invoking object.

**void notifyAll( )** Resumes execution of all threads waiting on the invoking object.

**String toString( )** Returns a string that describes the object.

**void wait( )**

**void wait(long milliseconds)**

**void wait(long milliseconds,**

**int nanoseconds )**

Waits on another thread of execution.

The methods getClass( ) , notify( ) , notifyAll( ), and wait( ) are declared as final.

The **equals( )** method compares the

c ontents of two objects. It returns true if the objects are equivalent and false otherwise.

The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using println( ).