**Inheritance** is a mechanism in which one class inherit all the properties and behaviors of parent class. You can create new classes that are built upon existing classes. When you inherit from an existing class, you can **reuse** methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the ***IS-A relationship***, also known as parent-child relationship.

In the language 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 variables and methods defined by the superclass and add its own, unique elements.

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

**class** subclass-name **extends** superclass-name {

// body of class

}

The ***extends keyword*** indicates that you are making a new class that derives from an existing class.

In the terminology of Java, a class that is inherited is called a **super class**. The new class is called a **subclass**.

**Example**: The following program creates a superclass called A and a subclass called B.

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**));

}

}

## Types of inheritance

The types of inheritance:

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through **interface** only. We will learn about interfaces later.

**ClassA**

**ClassB**

**ClassC**

**ClassA**

**ClassB**

**ClassC**

**ClassA**

**ClassB**

**ClassC**

**ClassA**

**ClassB**

**ClassD**

**ClassC**

**ClassA**

**ClassB**

1. **Single**
2. **Multilevel**
3. **Hierarchical**
4. **Multiple**
5. **Hybrid**

**Q) Why multiple inheritance is not supported in java?**

To reduce the complexity and simplify the language, **hybrid** and **multiple** inheritances are not supported in java.

## 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:

This program contains an error and will not compile.

class A {

int i; // public by default

private int j; // private to A

void setij(int x, int y) {

i = x;

j = y;

}

}

//------------------------------------------------------

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.

**REMEMBER**: 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.

## Super

**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.

**Example:**

// This program uses inheritance to extend Box.

class Box {

double width;

double height;

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 uses super to initialize its Box attributes

Class BoxWeight extends Box {

double weight; // weight of box

// construct clone of an object

BoxWeight(BoxWeight ob) {

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 box1= new BoxWeight(10, 20, 15, 34.3);

BoxWeight box2 = new BoxWeight(2, 3, 4, 0.076);

BoxWeight box3 = new BoxWeight(); // default

BoxWeight cube = new BoxWeight(3, 2);

BoxWeight clone = new BoxWeight(box1);

doublevol;

vol = box1.volume();

System.out.println("Volume of box1 is " + vol);

System.out.println("Weight of box1:"+box1.weight);

vol = box2.volume();

System.out.println("Volume of box2 is " + vol);

System.out.println("Weight of box2:" +box2.weight);

vol = box3.volume();

System.out.println("Volume of box3 is " + vol);

System.out.println("Weight of box3:"+box3.weight);

vol = clone.volume();

System.out.println("Volume of clone is " + vol);

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

vol = cube.volume();

System.out.println("Volume of cube is " + vol);

System.out.println("Weight of cube:"+ cube.weight);

System.out.println();

}

}

This program generates the following output:

Volume of box1 is 3000.0

Weight of box1:34.3

Volume of box2 is 24.0

Weight of box2: 0.076

Volume of box3 is -1.0

Weight of box3:-1.0

Volume of clone is 3000.0

Weight clone:34.3

Volume of cube is 27.0

Weight of cube: 2.0

**A Second Use for super**

The second form of super acts somewhat like this, except that it always refers to the super class 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*** definedin the superclass. As you will see, super can also be used to call methods that are hidden by asubclass.

## 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 byusing **super**. For example, in the following version of ***B***, the superclass version of ***show( )*** is invoked within the subclass’ version. This allows all instance variables to be displayed.

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( ).