To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.

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. 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.

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.

A Superclass Variable Can Reference a Subclass 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.

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.

plainbox = weightbox;

(superclass) (subclass)

SUPERCLASS ref = new SUBCLASS(); // HERE ref can only access methods which are available in SUPERCLASS

Using super:

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.

```
BoxWeight(double w, double h, double d, double m) {
   super(w, h, d); // call superclass constructor
   weight = m;
}
```

Here, BoxWeight( ) calls super( ) with the arguments 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.

Thus, super( ) always refers to the superclass immediately above the calling class.

This is true even in a multileveled hierarchy.

```
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;
    }
}
```

```
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;
    }
}
```

Notice that super() is passed an object of type BoxWeight—not of type Box.This still invokes the constructor Box(Box ob).

NOTE: 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.

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.

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.

super( ) always refers to the constructor in the closest superclass. The super( ) in BoxPrice calls the constructor in

BoxWeight. The super( ) in BoxWeight calls the constructor in Box. In a class hierarchy, if a superclass constructor

requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a

subclass needs parameters of its own.

If you think about it, it makes sense that constructors complete their execution in order of derivation.

Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and

possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

NOTE: If super( ) is not used in subclass' constructor, then the default or parameterless constructor of each superclass

will be executed.

Using final with Inheritance:

The keyword final has three uses:

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

Using 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.

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 an option only 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.

NOTE: 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 & relies upon its subclasses to provide complete implementations.

NOTE: Although static methods can be inherited ,there is no point in overriding them in child classes because the

method in parent class will run always no matter from which object you call it. That is why static interface methods

cannot be inherited because these method will run from the parent interface and no matter if we were allowed to

override them, they will always run the method in parent interface.

That is why static interface method must have a body.

NOTE : Polymorphism does not apply to instance variables.