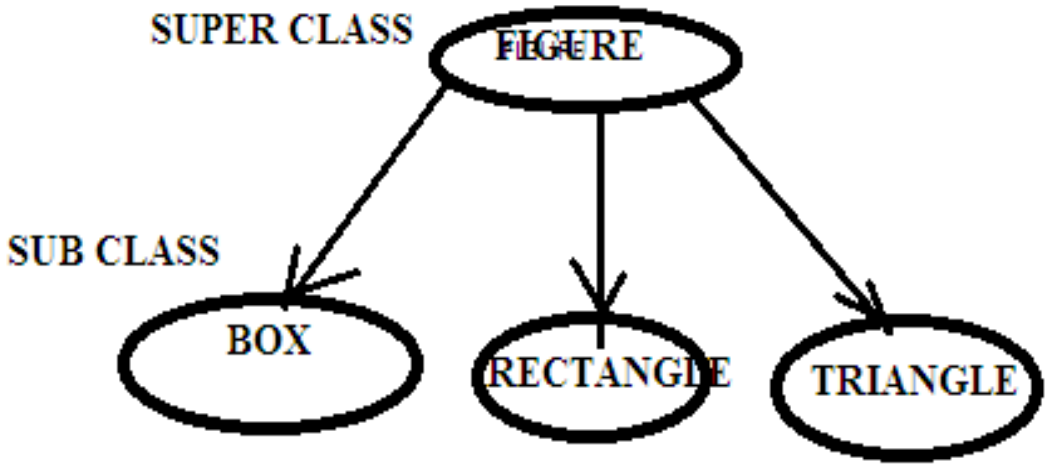
CHAPTER 7
INHERITANCE

INHERITANCE: EXTENDING CLASS

Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance supports the concept of hierarchical classification. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way.

Inheritance: A new class (derived class, child class or Super class) is derived from the existing class(base class, parent class or sub class).



Main uses of Inheritance:
1. Reusability
2. Abstraction

Syntax:

Class Sub-classname **extends** Super-classname
{
    Declaration of variables;
    Declaration of methods;
}

Super class: In Java a class that is inherited from is called a super class.
Sub class: The class that does the inheriting is called as subclass.

Therefore, a subclass is a specialized version of a super class. It inherits all of the instance variables and methods defined by the super class and add its own, unique elements.

The "**extends**" keyword indicates that the properties of the super class name are extended to the subclass name. The sub class now contain its own variables and methods as well those of the super class. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying the super class members.

To see how, let's begin with a short example. The following program creates a super class called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.

```
// A 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 super class, 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 super class for B, it is also a completely independent, stand-alone class. Being a super class for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a super class for another subclass.

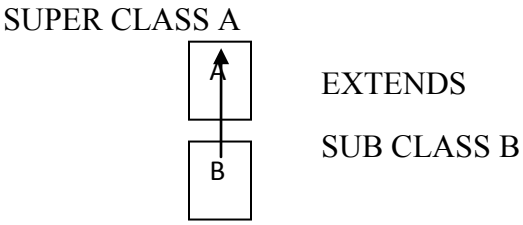
Types of Inheritance

Up to this point, we have been using simple class hierarchies that consist of only a super class 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 super class 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 super classes. In this case, C inherits all aspects of B and A.

Types of Inheritance are use to show the Hierarchical abstractions. They are:

- Single Inheritance
- Multiple Inheritance (with interface)
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

Single Inheritance: Simple Inheritance is also called as single Inheritance. Here One subclass is deriving from one super class.



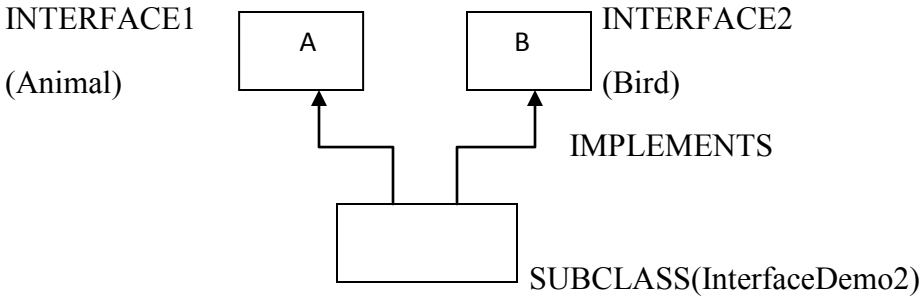
Example:

```
import java.io.*;
 abstract class A
{
   void display()
        {
           System.out.println(“Display version of Parent Class”);
       }
}
class B extends A
{
        void display()
        {
                System.out.println("hello");

        }
        public static void main(String args[])
        {
                B b=new B();
                b.display();
                super.display();

        }
}
```

Output:

Hello

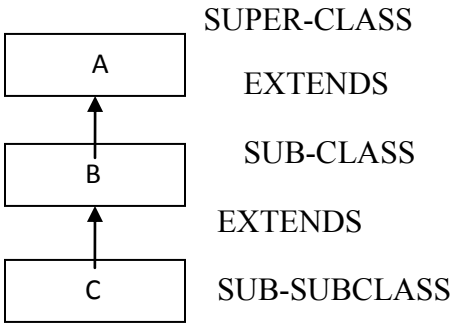
Multiple Inheritance: Deriving one subclass from more than one super classes is called multiple inheritance.



We know that in multiple inheritance, sub class is derived from multiple super classes. If two super classes have same names for their members then which member is inherited into the sub class is the main confusion in multiple inheritance. This is the reason, Java does not support the concept of multiple inheritance,. This confusion is reduced by using multiple interfaces to achieve multiple inheritance. The concept of interface learn in next chapter.

Multilevel Inheritance:

In multilevel inheritance the class is derived from the derived class.



Example: 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.

```
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
```

```
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

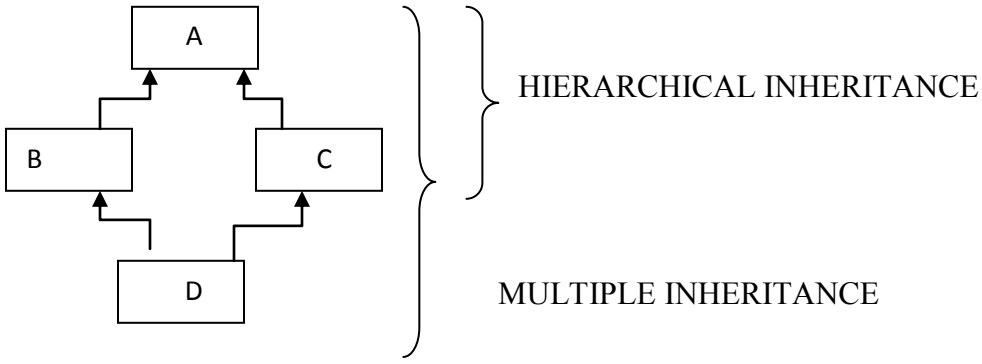
The output from this program is shown here:

Inside A's constructor Inside B's constructor Inside C's constructor

As you can see, the constructors are called in order of derivation. If you think about it, it makes sense that constructors are executed in order of derivation. Because a super class 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 be executed first.

Hybrid Inheritance:

It is a combination of multiple and hierarchical inheritance.



Subclass Constructor

A subclass constructor is used to construct the instance variable of both the subclass and the superclass. The subclass constructor uses the keyword **super** to invoke the constructor method of the superclass. **super** has the two general forms:

1. The first calls the superclass constructor. (super (args-list))
2. The second is used to access a member of the superclass that has been hidden by a member of a subclass. (super**.**member)

1. **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*);

Remember
The keyword **super** is used in the following conditions:

- Super may only be used within a subclass constructor method.
- The call to superclass constructor *super(args-list)* must appear as the first statement within the subclass constructor.
- The parameters in the super call must match the order and type of the instance variable declared in the superclass.
- Super class default constructor is available to sub class by default.
- First super class default constructor is executed then sub class default constructor is executed.

```
// 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();
}
}
```

```
// 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)**. 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.

2. A Second Use for super (Accessing the member of a super class)

Super.member;

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
```

Method Overriding

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() {
super.show();  // this calls a A's 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
}
}
```

Why Overridden Method

Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "**one interface, multiple methods**" aspect of polymorphism. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface.

Dynamic Method Dispatch

Dynamic method dispatch is an important mechanism in Java that is used to implement runtime polymorphism. In this mechanism, method overriding is resolved at runtime instead of compile time. That means, the choice of the version of the overridden method to be executed in response to a method call is done at runtime.

Dynamic dispatch is a mechanism by which a call to Overridden function is resolved at runtime rather than at Compile time , and this is how Java implements Run time Polymorphism. In dynamic method dispatch, super class refers to subclass object and implements method overriding.

Here is an example that illustrates dynamic method dispatch:

```
// 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 A
r = 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
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

Using Final with Inheritance

The keyword final has three uses:

- Create named constant
- To prevent overriding
- To Prevent Inheritance

Create Named Constant: This use was described in the previous chapter.

Using Final to Prevent Overriding

All method and variables can be overridden by default in subclasses. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as **final** using the keyword final as a modifier. In this way you will never be altered the method in any way.

Methods declared as final cannot be overridden. The following fragment illustrates final:

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

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

Using final to Prevent Inheritance
Sometimes we may like to prevent a class being further subclasses for security reasons. A class that cannot be subclassed is called a *final class*. This is achieved in Java using the keyword **final.**

Remember:
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
// ...
}
```