

INHERITENCE IN JAVA

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass 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
```

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.

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.

Another Example which uses super()

```
// 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 mycloneis"+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();
    }
}

```

This program generates the following output:

```

Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3

```

Volume of mycube is 27.0
Weight of mycube is 2.0

When Constructors Are Called

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since **super()** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameter less constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are called.
// 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
```

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
```

Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. 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.

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

```

Abstract Class & Methods

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass 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.

```

// A Simple demonstration of abstract.
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();
    }
}

```

Using final to Prevent Overriding

```

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.

Using final to Prevent Inheritance

```

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

```