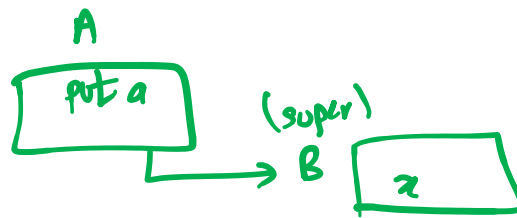


Inheritance

Using super



In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the width, height, and depth fields of **Box()**. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, 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. Each use is examined here.

Using super to Call Superclass Constructors

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

super(parameter-list);

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

To see how super() is used, consider this improved version of the BoxWeight() class:

Notice that **super()** is called with 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.

*Let's review the key concepts behind **super()**. When a subclass calls **super()**, it is calling the constructor of its immediate superclass. Thus, **super()** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super()** must always be the first statement executed inside a subclass constructor.*

A Second Use for super

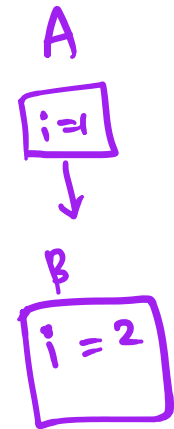
The second form of **super** acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.member

Here, member can be either a method or an instance variable.

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 defined in the superclass.

As you will see, **super** can also be used to call methods that are hidden by a subclass.

When Constructors Are Called

A
↓
B

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 parameterless constructor of each superclass will be executed.

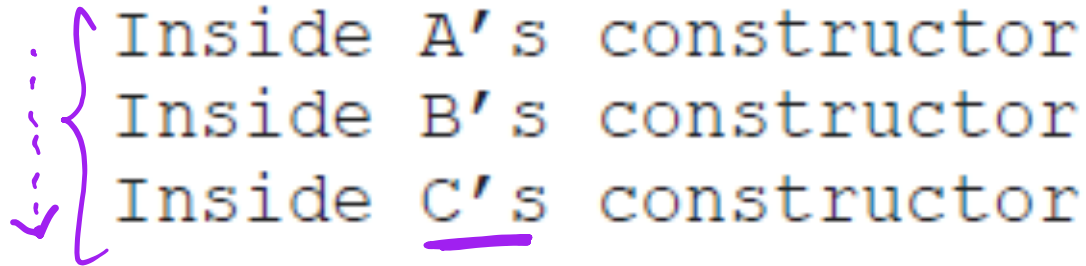
The following program illustrates when constructors are executed:


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

A
↓
B
↓
C

main

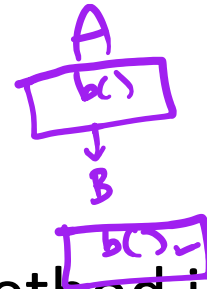
The output from this program is shown here:



```
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 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 be executed first.

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 1a, int 2b) {
```

```
i = a;
```

```
j = b;
```

```
}
```

```
// display i and j
```

```
void show() {
```

```
System.out.println("i and j: " + i + " " + j);
```

```
}
```

```
}
```

```
class B extends A {
```

sub Ob

i=1 j=2 k=3

A()

B()

show()

A Farmer

i

j

A()

show()

B (Darshan)

i

j

k

B()

A()

~~show()~~

show()

```

int k;
B(int 1a, int 2b, int 3c) {
    super(a, b);
    k = c;
}
// display k - this overrides show() in A
void show() {
    System.out.println("k: " + k); Super.show();
}
}
Main
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

i and j: 1 2

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 function, you can do so by using `super`. For example, in this 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()**.

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.*

For example, consider this modified version of the preceding example:

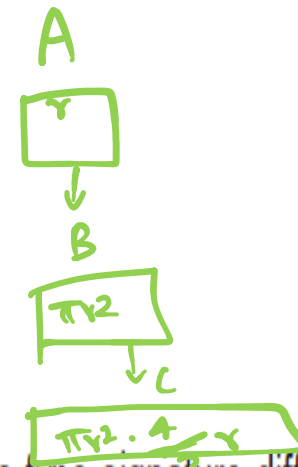

```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

```
// Create a subclass by extending class A.
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // overload show()
    void show(String msg) {
        System.out.println(msg + k);
    }
}
class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

The output produced by this program is shown here:

This is k: 3

i and j: 1 2



The version of `show()` in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place.

//WAP to calculate Volume of sphere using Multilevel Inheritance demonstrating Method Overriding. The base class method will accept the radius from the user. A class will be derived from the above mentioned class that will have a method to find and display the area of a circle and another class derived from this will have methods to calculate and display the volume of the sphere.

```
import java.io.*;
class Data{
    protected float r;
    public void read(float x)
    {
        r=x;
    }
}
class Area extends Data
{
    protected float area;
    public void calculate()
    {
        area=3.14f*r*r;
    }
    public void display()
    {
        System.out.println("Area="+area);
    }
}
```



```
class Volume extends Area{  
    private float volume;  
    public void compute()  
    {  
        volume=area*r*4/3;  
    }  
    public void display()  
    {  
        System.out.println("Volume="+volume); super.display();  
    }  
}
```

```
class Main{
public static void main (String args[]) throws IOException{
float x;
String str;
BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
System.out.println("Enter the radius:");
str=br.readLine();
x=Float.parseFloat(str);
Volume a=new Volume();
a.read(x);
a.calculate();
a.compute();
a.display();
}
}
```

Output:

Enter the radius:

10

Volume=4186.6665

Applying Method Overriding

Let's look at a more practical example that uses method overriding.

The following program creates a superclass called **Figure** that stores the dimensions of various two-dimensional objects. It also defines a method called `area()` that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides `area()` so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism.
class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
double area() {
System.out.println("Area for Figure is undefined.");
return 0;
}
}
```



```
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
```

```
class FindAreas {
public static void main(String args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
figref = f;
System.out.println("Area is " + figref.area());
}
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
```

```
Area is 45
```

```
Inside Area for Triangle.
```

```
Area is 40
```

```
Area for Figure is undefined.
```

```
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area()**. The interface to this operation is the same no matter what type of figure is being used.

Using final with Inheritance

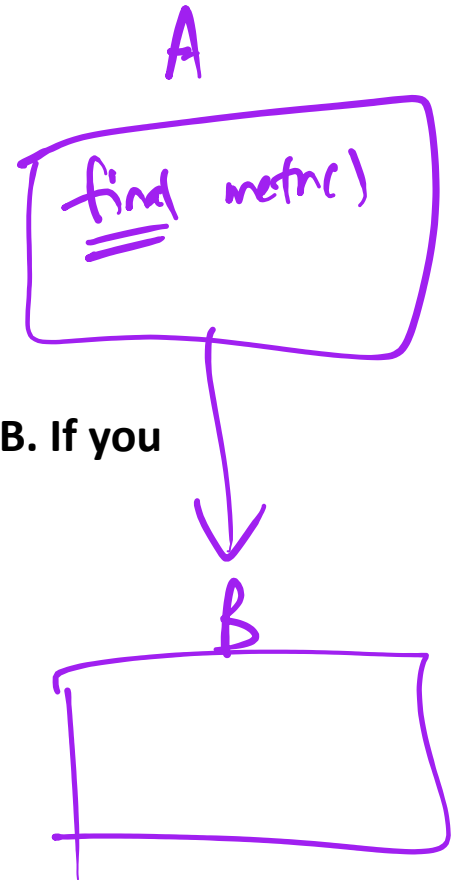
The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. 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. The following fragment illustrates **final**:

```
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.



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 only an option 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

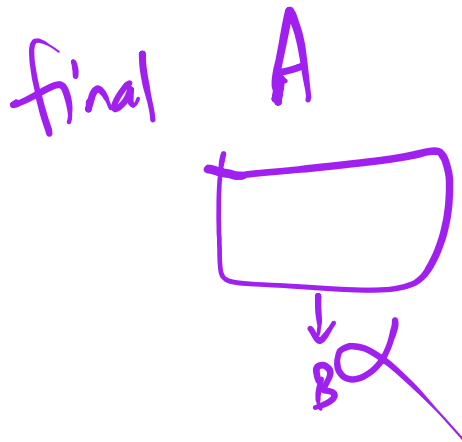
Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**.

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:

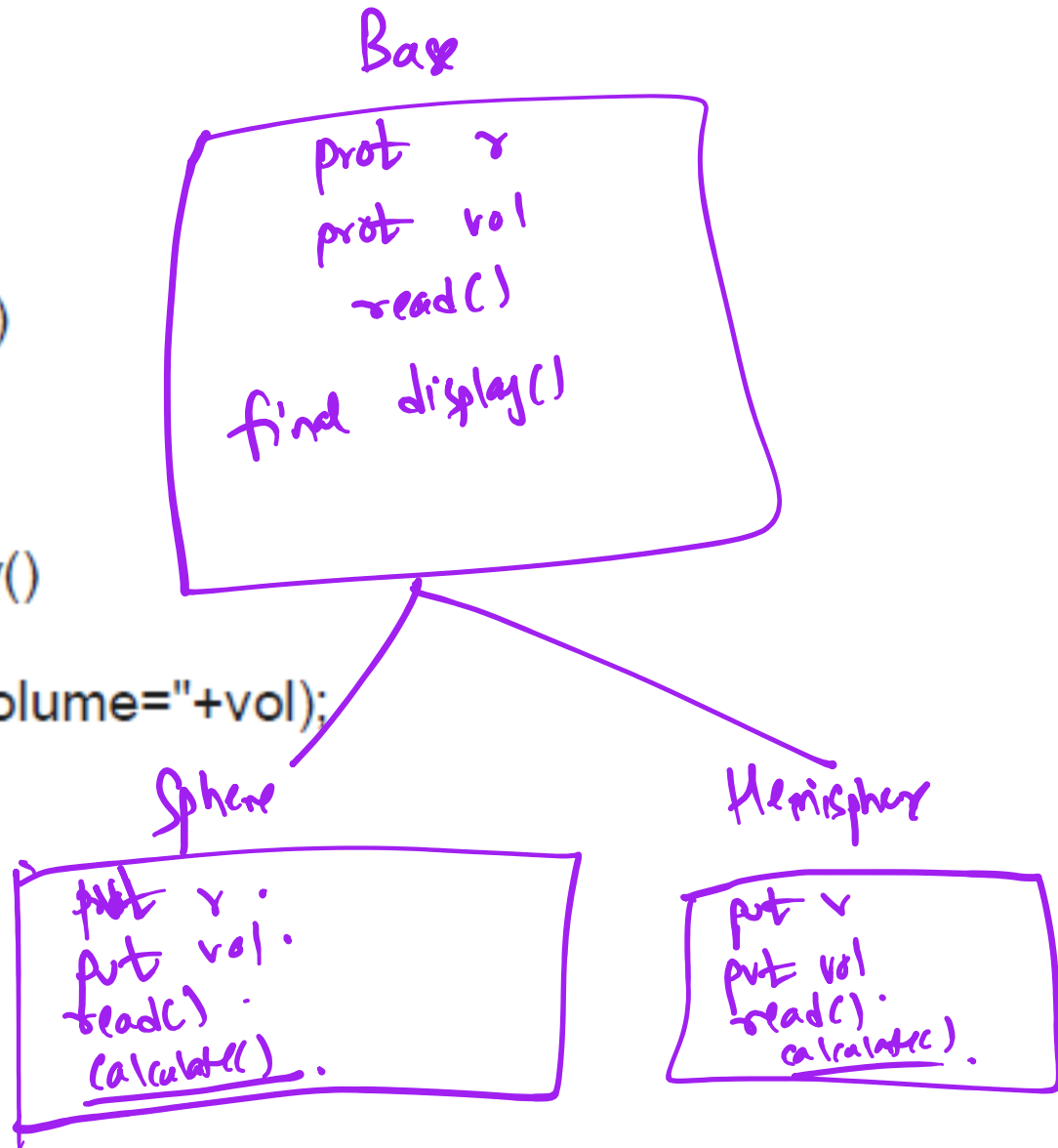
```
{ final class A {  
  // ...  
}  
// The following class is illegal.  
class B extends A { // ERROR! Can't subclass A  
  // ...  
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.



//WAP to display volume of the sphere and hemisphere. Make use of final method to display the volume.

```
import java.io.*;
class Base{
protected float r,vol;
public void read(float x)
{
    r=x;
}
final public void display()
{
    System.out.println("Volume="+vol);
}
}
```



```
class Sphere extends Base
{
    public void calculate()
    {
```

```
        vol=3.14f*r*r*r*4/3;
```

```
    }
```

```
}
```

```
class Hemisphere extends Base
{
    public void calculate()
    {
        vol=3.14f*r*r*r*2/3;
    }
}
```

```
class Main{
public static void main (String args[]) throws IOException{
float x;
String str;
BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
System.out.println("Enter the radius:");
str=br.readLine();
x=Float.parseFloat(str);
Sphere s=new Sphere();
s.read(x);
s.calculate();
System.out.println("Sphere:");
s.display();
Hemisphere h=new Hemisphere();
h.read(x);
h.calculate();
System.out.println("Hemisphere:");
h.display();
}
}
```

Output:

Enter the radius:

10

Sphere:

Volume=4186.6665

Hemisphere:

Volume=2093.3333

The implementation program is done with final method display() in base class and hence, cannot be again defined in subclasses Sphere and Hemisphere i.e. it cannot be overridden.

The base class —Base|| has the method display() declared as —final||, hence the derived class members cannot override this method.