CSE 11: Lecture 9

- Constructors, inheritance, this() and super()
- Instance methods, inheritance, and overriding
- Java's Object class
- Overriding vs. overloading
- Final classes
- Static members and inheritance
- Instance variables, inheritance, and variable hiding

(Reading: Savitch, Ch. 7)

Constructors, reviewed

- A constructor for a class is called whenever an object of the class is created
- A constructor is never called at any other time!
- A constructor has the job of initializing a newly created object
- Instance variables of the object being created are accessible within the body of the constructor
  - they can be referred to directly by name, or through the `this` variable and the dot operator
  - but note that private instance variables in ancestor classes are not accessible in this way!
- There can be several constructors for a class (constructor overloading); they must differ in the number and/or type of arguments that they take
- A constructor for a class must have the same name as the class, and no return type

Constructors and this()

- In a constructor for a class, you can call another constructor for the same class:
  - To do this, you use `this()` as the name of the constructor you're calling
  - If used, the call must be the first statement in the body of the constructor

- Often this is convenient to do:
  - A constructor that takes arguments can do complicated initialization of the new object's instance variables
  - A constructor that takes fewer arguments (for example, a default constructor) can just call the many-argument constructor to do the initialization, passing default values for arguments
  - This way, the default constructor does not need to duplicate the code in the body of the many-argument constructor... you get software reuse

Constructors and this(): an example

```
public class C {
   private int val;

   public C(int v) { val = v; }

   public C() { this(44); }

   public void setVal(int v) { val = v;}
   public int getVal() { return val; }
}
-------------------------------------------------------
public class Test {

   public static void main (String[] args) {
      C c1 = new C();
      C c2 = new C(7);

      System.out.println(c1.getVal());        // prints...
      System.out.println(c2.getVal());        // prints...
```

Constructors, inheritance, and super()

- Constructors of a base class are not inherited by a derived class

- However it is easy to call a base class constructor from a derived class constructor:
  - to do this, you use **`super()`** as the name of the base class constructor you're calling
  - if you do this, the call to the other constructor must be the first statement

    (so: within a derived class constructor, you cannot have both a call to a base class constructor, and a call to another constructor of the derived class...!)

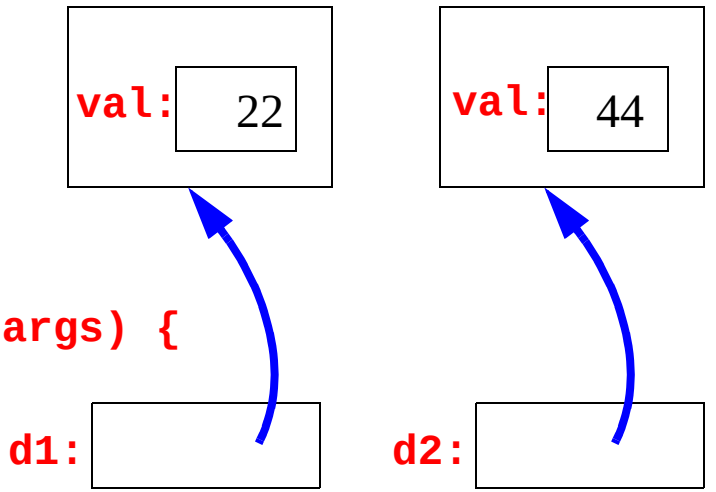
- In fact, it is not only easy, it is actually impossible to avoid it:
  - If you do not call the base class constructor (using **`super()`**) or another constructor of the derived class (using **`this()`**), then...

    Java will implicitly insert a call to the base class's default constructor **`super()`** before the first statement of the derived class constructor

- Java does this to make sure that initialization of the base class part of an object is complete before initialization of the derived class part begins

Constructors, this(), and super(): an example

```
public class C {
   private int val;
   public C(int v) { val = v; }
   public C() { this(44); }
   public void setVal(int v) {val = v;}
   public int getVal() {return val;}
}
-------------------------------
public class D extends C {
   public D() { super(22); }
   public D(int v) { } // strange!
-------------------------------
public class Test {
   public static void main (String[] args) {
      D d1 = new D();
      D d2 = new D(7);

      System.out.println(d1.getVal());        // prints...
      System.out.println(d2.getVal());        // prints...
```



Constructors and inheritance: another example

```
public class A {
      public A () { System.out.println(2); }
      public A (int x) { System.out.println(2*x); }
}
-----------------------------------------------------
public class B extends A {
      public B () { super(5); }
      public B (int x) { System.out.println(x+2); }
}
-----------------------------------------------------
public class Program {
   public static void main ( String args[] ) {
         A a_obj = new A();
         a_obj = new A(2);
         B b_obj = new B();
         b_obj = new B(4);
   }
}
```

- What is printed when this program runs? 2, 4, 10, 2, 6

Dynamic method binding reviewed

- Java always uses “dynamic binding” (also called “virtual” or “late binding”) of public instance method calls: the method called depends on the *type of the actual calling object* the pointer points to at runtime, not the type of the pointer

- But the method called must first be available in the class that is the type of the pointer, either by being defined in that class or by inheritance.

- The compiler will check this!

```
Object o = new Saxophone();  // okay, a Saxophone is-a Object
o.play();  // compile-time error: play() is not a method of Object
```

- To take advantage of dynamic binding at runtime, your program must first compile, and for it to compile, it must pass all compile-time type checks
- But good object-oriented design lets you take advantage of dynamic binding, which can be a very powerful programming technique
- We will look at an example of this from the Java standard library:
- The `System.out.print()` method is obviously very powerful: you can pass it anything as argument, and it will “do the right thing” and print it on the terminal screen. How does it do that?...

The print() methods of System.out

- Dynamic binding improves software reuse. The definition of `System.out.print()` shows a good example of this, using an object's `toString()` method
- If you look at the documentation or source code for the System class, you will see that `System.out` is an object of class PrintStream:

```
public final static PrintStream out;
```

- Then if you look at the documentation or source code for the PrintStream class, you will see lots of instance methods, in particular many versions of the `print()` method:

```
public void print(boolean b) // Print a boolean value.
public void print(char c) // Print a character.
public void print(char[] c) // Print an array of characters.
public void print(double d) // Print a double-precision f-p number.
public void print(float f) // Print a floating-point number.
public void print(int i) // Print an integer.
public void print(long l) // Print a long integer.
public void print(String s) //  Print a string.
public void print(Object o) // Print an object.
```

- So, System.out.print() can handle an argument of any primitive type (byte and short will automatically be converted to int) as well as char arrays and Strings.
- What about objects other than String? They are *all* handled by `print(Object)`

System.out.print(Object)

- If you look at the definition of **`print(Object)`** in the PrintStream class, you will see

```
public void print(Object o) {
   if (o == null) write("null");
   else write(o.toString());
}
```

(where **`write()`** is a low-level method that "knows how" to write Strings)

- This gives great software re-use... It is not necessary to create a new **`print()`** method for every class (many possible classes are not even invented yet!)
- Instead, this simple method will work for printing *any* object, because every Java object has a public **`toString()`** instance method
- Every object has a **`toString()`** method because that method is defined in the Object class, and in Java every object is-a Object!
- **`toString()`** can (and usually should) be overridden in user-defined subclasses (this trick would not work if **`toString()`** weren't defined in the Object class, or if Object weren't the ancestor of every class, or if Java didn't use dynamic binding)

- With dynamic method binding, the call to **`o.toString()`** does the "right thing"

The Object class

- There is one class in Java that is not a subclass of any other class: the class **`Object`**
- There is one class in Java that is an ancestor of every other class: the class **`Object`**
- So, in Java, every object is-a **`Object`**! So the public and protected instance methods of **`Object`** are inherited by every Java object
- Here are the headers of the methods of **`Object`**:

equals and toString are the most important for CSE 11. If you don't define your own versions in your classes, you'll inherit these. What do **`Object`**'s versions do? Try it!

```
public boolean equals(Object obj)
public String toString()
```

hashCode may be useful in CSE 12 when implementing hash tables

```
public native int hashCode()
```

getClass is final: you can't override it. Returns a Class object describing the class of the calling object

```
public final native Class getClass()
```

The Object class, cont'd

finalize is called automatically when an object is reclaimed by the garbage collector (this is as close as Java gets to having a destructor method)

```
protected void finalize() throws Throwable
```

clone is for creating a byte-by-byte copy of an object

```
protected native Object clone()
```

the remaining methods are for multithreaded synchronization

```
public final native void notify()
public final native void notifyAll()
public final native void wait()
public final native void wait(long millis)
public final native void wait(long millis, int nanos)
```

Overriding, overloading, and dynamic binding

- To take full advantage of dynamic binding, it is important to keep in mind these things:
- The difference between method overriding and method overloading:
  - *Overriding* is defining a method in a derived class that has exactly the same signature (name, number and type of arguments, return type, throws declaration) as a method that would otherwise be inherited from the base class
  - *Overloading* is defining a method in a class with the same name as another method in the class, but a different number and/or type of arguments
- The definition of dynamic binding of public instance methods: the method called depends on the *type of the actual calling object* the pointer points to at runtime, not the type of the pointer
  - But the method must exist in the class that is the type of the pointer for this to work, and
  - Dynamic binding applies only to the calling object in a method call; dynamic binding does not apply to objects passed to the method
- We will look closely at another example, the equals() method of Object

Overriding equals()

- Another example of software reuse potential is the equals() public instance method, introduced in the Object class
- equals() is defined in Object as follows:

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

- ... that is, the equals() method defined in Object just checks to see if two pointers point to the same Object. Consider:

```
public class A extends Object {
   public int i;
   public A(int j) {i = j;}
}
-------------------------
   public static void main(String args[]) {
      A a = new A(3);  A b = new A(3);
      System.out.println( a.equals(b) );   // prints...  false
      System.out.println( a.equals(a) );   // prints...  true
```

- Usually you will want a version of equals() that does something more sophisticated than this

Overloading equals()

- One way to do this is to overload equals() in the derived class:

```
public class A extends Object {
      public int i;
      public A(int j) {i = j;}
      public boolean equals(A other) {
         return this.i == other.i;
      }

   }
```

- (Note that this is not overriding, since the method signature has changed!)
- But this may not be what you want either:

```
   public static void main(String args[]) {
      A a = new A(3);  A b = new A(3);   A c = new A(4);
      System.out.println( a.equals(b) );   // prints...  true
      System.out.println( a.equals(c) );   // prints...  false

      Object o = new A(3);
      System.out.println( a.equals(o) );   // prints... false
      System.out.println( o.equals(a) );   // prints... false
```

Understanding overriding vs. overloading

- The reason for those results is that a method equals() taking an `A` argument doesn't exist in the `Object` class, and an equals() method taking an `Object` argument does exist in the `A` class (by inheritance from `Object`). In more detail:
- Consider `a.equals(o)`
  - The calling object pointer expression `a` is of type `A`, so the `equals()` method being called must exist in the class `A`.
  - But `equals()` has been overloaded and there are *two* versions there: `equals(Object)` inherited from the `Object` class, and `equals(A)` defined in the `A` class. Which one is being called?
  - The version of an overloaded method that will be called is the one with a formal parameter list that best fits the actual arguments. In this case the actual argument is `o`, which is a pointer expression of type `Object` (it happens to be pointing to an instance of `A`, but dynamic binding is not done for arguments), so `equals(Object)` will be called
- Consider `o.equals(a)`
  - The calling object pointer expression `o` is of type `Object`, so the `equals()` method being called must exist in the class `Object`.
  - There is only one version there, which takes an `Object` argument; the actual argument is of type `A`, which is is-a `Object`, so there is a match

A good way to override equals()

- To take maximum advantage of dynamic method binding, it is best to override equals(), not overload it

- When overriding equals(), it can be convenient to make use of downcasting, and the fact that a failed downcast throws a ClassCastException. Example:

```
public class A extends Object {
      public int i;
      public A(int j) {i = j;}

      public boolean equals(Object obj) {
         if (obj == null) return false;  // not equal to null!
         try {
               A other = (A) obj;  // downcast!
               return this.i == other.i;  // needed downcast to do

         } catch (ClassCastException e) {

            return false;  // obj does not point to an A object!
         }
      }
```

Another good way to override equals()

- Another approach is to use the **instanceof** operator, instead of doing a downcast that can fail with a ClassCastException
- The instanceof operator takes two arguments: a reference type pointer, and a class name. It returns true if the object pointed to by the pointer "is-a" instance of the class

```
public class A extends Object {
      public int i;
      public A(int j) {i = j;}

      public boolean equals(Object obj) {
         // the calling object is not equal to null
         if (obj == null) return false;
         // the calling object is not equal to any non-A object
         if (! (obj instanceof A) ) return false;

         A other = (A) obj;  // this downcast cannot fail
         return this.i == other.i;  // need downcast for access

      }
```

Overriding equals, cont'd

- Now equals() works this way, which is probably closer to what you wanted:

```
public static void main(String args[]) {

   A a = new A(3);  A b = new A(3);   A c = new A(4);
   System.out.println( a.equals(b) );   // prints...  true
   System.out.println( a.equals(c) );   // prints...  false

   Object o = new A(3);
   System.out.println( a.equals(o) );   // prints... true
   System.out.println( o.equals(a) );   // prints... true
   System.out.println( o.equals(c) );   // prints... false

   String s = new String("3");
   System.out.println( a.equals(s) );   // prints... false
```

Inheritance and overriding: another example

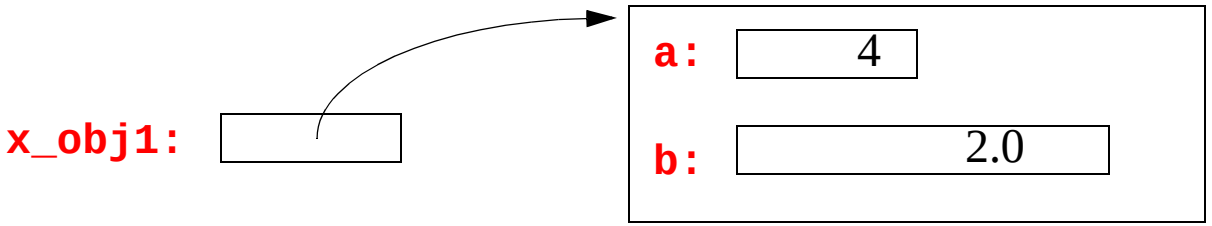
- Suppose classes X and Y are defined this way:

```
public class X {
        private int a;
        private double b;
        public X ( int num1, int num2 ) { a = num1; b = num2; }
        public double getVal () { return b; }
        public void print() { System.out.println( getVal() ); }
}

-------------------------------------------------------
public class Y extends X {
        private int c;
        public Y ( int num ) { super(2*num, 3);  c = num; }
        public double getVal () { return  c + super.getVal(); }
}
```

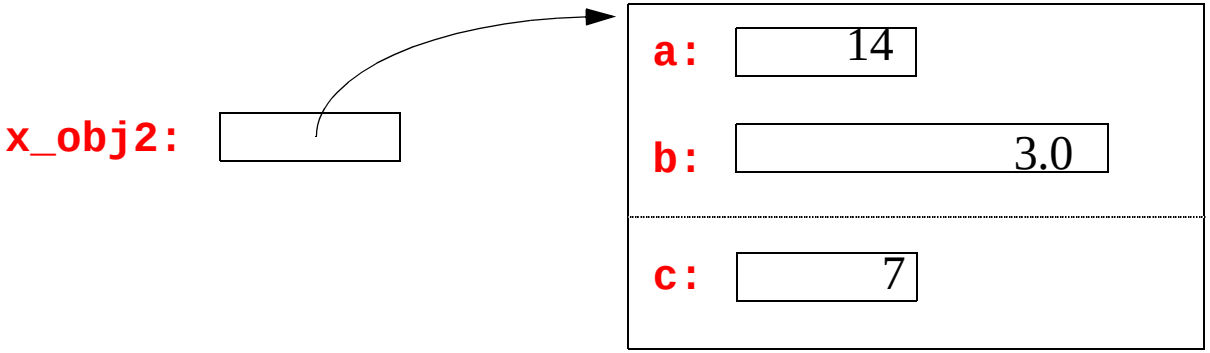
Derivation and instance variables

- An object of a class contains all the instance variables of that class, plus all the instance variables of all its superclasses... But superclass private variables are not directly accessible in the derived class!

```
X x_obj1 = new X(4,2);
```




```
X x_obj2 = new Y(7);
```



Inheritance and overriding: an example, cont'd

- Given the definitions of classes X, Y: what happens when this program runs?

```
-----------------------------------------------------

public class Tester {
public static void main ( String[] args ) {
   X x_obj1 = new X(4,2);
   X x_obj2 = new Y(7);

   System.out.println( x_obj1.getVal() );   // prints...  2.0
   System.out.println( x_obj2.getVal() );   // prints...  10.0

   x_obj1.print();      // prints...   2.0
   x_obj2.print();      // prints...   10.0
}
```

- ... remember that public instance methods are always dynamically bound in Java!

Classes that cannot be extended

- Sometimes you want to define a class that cannot be a base class for any derived class
- Why? If someone could use it as a base class, they might create a class that behaves very differently, though it would be treated as a subtype of the base class by the type system. With a system-critical class, this could be a problem

- To say that a class cannot be extended, use the keyword **`final`**:

```
public final class MyClass {  // MyClass cannot be extended

...
...
```

- Some classes in the standard Java libraries are final: **`Math`**, **`String`**, **`System`**, all the wrapper classes, etc.

Dynamic and static binding of methods

- As we have seen, Java uses dynamic binding of public instance method calls...
- However it is important to know that Java does not use dynamic binding for:
  - static methods
  - final instance methods (all methods in a final class are final)
  - private instance methods
- For static, final, or private methods, which method to call is determined at compile time
  - for static methods, this is because the method to call depends on the type of the pointer, not the object pointed to (if any)
  - for final methods, this is because they cannot be overridden
  - for private methods, this is because they are considered to be only for the specific private use of the other methods in the class in which they are defined

Static vs. instance methods

- A static method of a class is associated with the class itself, while an instance method is associated with each individual object that is an instance of the class
  - This means that the **`this`** variable is automatically defined in the body of every instance method, but not for any static methods

- From outside the class, you can refer to a visible static method by using an expression of the form **`<classname> . <methodname> (<arglist>)`**

- ... or, if you have a pointer expression of that class type, you can refer to a visible static method using an expression of the form

  `<pointerexpr> . <methodname> (<arglist>)`

  - only the compile-time *type of the pointer expression* is relevant here; its value can even be null, and this will still work

Static binding of static methods: example

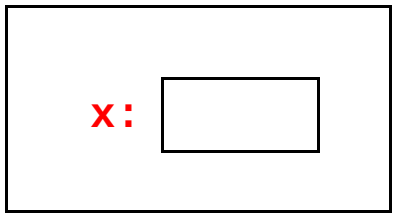
```
public class Base {
   public static void foo() {System.out.println(33);}
}
-------------------------------------------------------------
public class Derived extends Base {
   public static void foo() {System.out.println(44);}
}
-------------------------------------------------------------
public class Test {
   public static void main (String[] args) {
   Base.foo();          // prints...    33
   Derived.foo();       // prints...    44
   Base b1 = new Base();
   Base b2 = new Derived();
   Derived d1 = new Derived();
   Derived d2 = null;
   b1.foo();            // prints...    33
   b2.foo();            // prints...    33
   d1.foo();            // prints...    44
   d2.foo();            // prints...    44
```

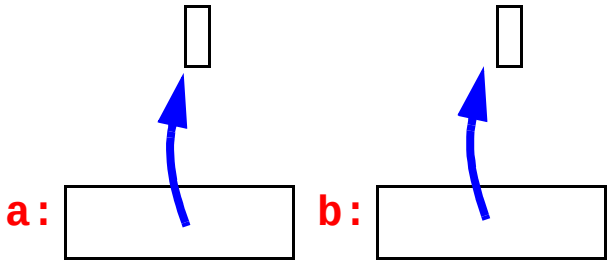
Static variables vs. instance variables

- A static variable of a class is associated with the class itself:
  - there is only one copy of it, associated with the class itself, no matter how many instances of the class have been created (even zero)
  - from outside the class, you can refer to a visible static variable by using an expression of the form **`<classname> . <variablename>`**
  - ... or, if you have a pointer expression of that class type, you can refer to a visible static variable using an expression of the form **`<pointerexpr>.<variablename>`**
    - only the compile-time type of the pointer expression is relevant in this case; its value can even be null, and this will still work
- An instance variable of a class is associated with each instance (object) of the class:
  - there are as many copies of it as there are instances of the class; each instance gets its own copy
  - if you have a pointer expression referencing an instance of the class, you can refer to a visible instance variable by using an expression of the form **`<pointerexpr> . <variablename>`**
- Let's look at an example...

Using a static variable...

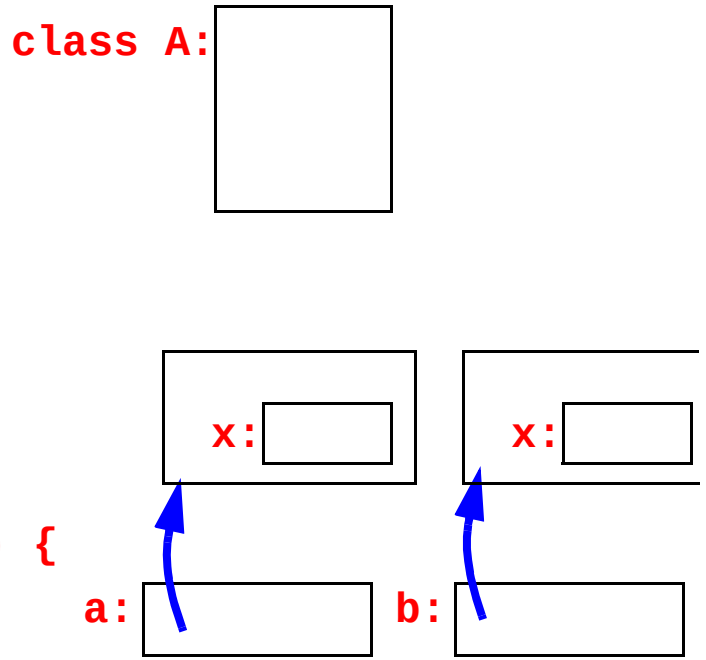
```
public class A {
   private static int x;

   public A(int val) { x = val; }

   public int getX() { return x; }

}

---------------------------------
public static void main(String args[]) {

   A a = new A(1);
   System.out.println(a.getX());   // prints... 1

   A b = new A(2);
   System.out.println(b.getX());   // prints... 2

   System.out.println(a.getX());   // prints... 2
```





Using an instance variable...

```
public class A {
   private int x;

   public A(int val) { x = val; }

   public int getX() { return x; }

}

-----------------------------------
public static void main(String args[]) {

   A a = new A(1);
   System.out.println(a.getX());   // prints...  1

   A b = new A(2);
   System.out.println(b.getX());   // prints...  2

   System.out.println(a.getX());   // prints...  1
```



Uses of static variables

- The main uses of static variables are:
  - as “class global” variables that are shared by all the static methods of a class (as when doing procedural programming in an OO language), or which can be exported from the class (public static final “named constants”)
  - as data that are shared by all instances of the class

- Example of the second use: a counter that keeps track of how many instances of the class have been created

```
public class MyClass {
   private static int numCreated = 0;

   public MyClass() {
      numCreated++;
      // other initialization goes here...
   }
   // any other ctors need to make sure count is incremented also
   ...
}
```

Inheritance and hiding of instance variables

- When a derived class public instance method has the same name and number and type of arguments as a base class instance method, the derived class method *overrides* the base class method
  - Dynamic method binding determines which method will be called
  - They type of the object pointed to (not the type of the pointer) determines which method is called at runtime

- When a derived class instance variable has the same name as a base class instance variable, the derived class variable *hides* or *shadows* the base class variable
  - Hiding is different from overriding!
  - Dynamic binding is NOT used in the case of variable hiding:
  - The type of the pointer (not the type of the object pointed to) determines which variable is accessed at runtime

- Since instance variables are almost always private, and so are inaccessible outside the class in which they are defined, this doesn't come up much in practice, but you should know how it works

Inheritance/hiding of instance variables: an example

- Consider these class definitions:

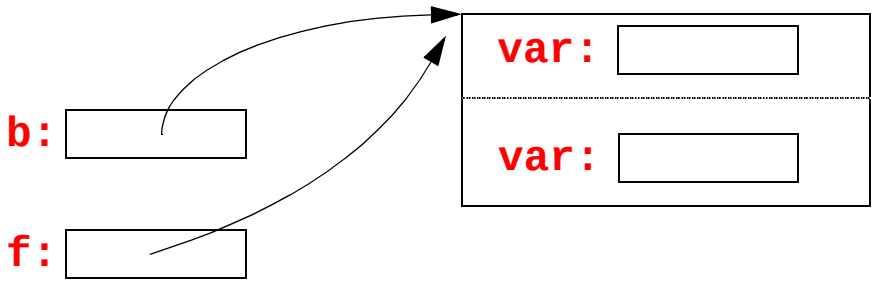
```
public class Foo {

  public double var;

  public void doIt() { System.out.println(var);}

}
-----------------------------------------------------
public class Bar extends Foo {

  public double var;

  public void doIt() { super.doIt(); System.out.println(var); }

}
```

- In the derived class, the declaration of `var` hides the declaration of `var` in the base class

Inheritance/hiding of instance variables: an example, cont'd

- Now what is printed when this program runs:

```
public class Test {
  public static void main(String args[])    {

      Foo f; Bar b;
      b = new Bar();
      f = b;

      f.var = 77.;
      b.var = 88.;

      f.doIt();
      b.doIt();
    }
}
```




```
77.0
88.0
77.0
88.0
```

Next time

- ✔ File I/O
- ✔ Text files, binary files
- ✔ Important classes in the java.io package
- ✔ The containment pattern
- ✔ Text file output: PrintWriter, FileWriter
- ✔ Text file input: BufferedReader, FileReader
- ✔ I/O streams
- ✔ Binary file I/O: DataInputStream, DataOutputStream
- ✔ Buffering

(Reading: Savitch, Ch. 9)