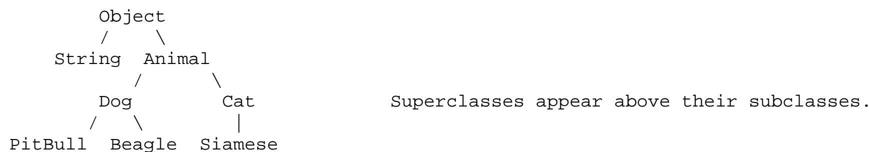
CS 4: Lecture 25
Monday, April 24, 2006

INHERITANCE (continued)
===========

Class Hierarchies

Subclasses can have subclasses. Subclassing is transitive: if Dog is a subclass of Animal, and Beagle is a subclass of Dog, then Beagle is a subclass of Animal. Furthermore, _every_ class is a subclass of the Object class (including Java's built-in classes like String and BufferedReader). Object is at the top of every class hierarchy.



Superclasses appear above their subclasses.

That's why the equals() method takes a parameter of type Object: you can use equals() to compare any object to any other object of any class. (You can't use equals() to compare with a primitive type, though.)

Inheritance and Constructors

What happens when we construct a Vector3D? As you would expect, Java executes a Vector3D() constructor, but first it executes the code in the Vector() constructor. The Vector3D() constructor should initialize fields unique to Vector3D(), like z; and it can also modify the work done by Vector() if appropriate.

```
  public class Vector3D extends Vector {
    [definitions from Lecture 24 here]

    public Vector3D() {
      // Vector() constructor called automatically
      System.out.println("Making new Vector3D.");
    }
```

The zero-parameter Vector() constructor is always called by default, regardless of the parameters passed to the Vector3D() constructor. To change this default behavior, the Vector3D constructor can explicitly call any constructor for its superclass by using the "super" keyword.

```
    public Vector3D(double xx, double yy, double zz) {
      super(xx, yy);
      z = zz;
    }
```

The call to "super()" must be the first statement in the constructor. If a constructor has no explicit call to "super", and its (nearest) superclass has no zero-parameter constructor, a compilation error occurs. There is no way to tell Java not to call a superclass constructor. You only have the power to choose which of the superclass constructors is called.

Invoking Overridden Methods

Sometimes you want to override a method, yet still be able to call the method implemented in the superclass. The following example shows how to do this. Suppose there is a method Vector.double() that multiplies a vector by two. We want Vector3D.double() to reuse the code in Vector.double(), but we also need to double the z-coordinate.

```
    public void double() {
      super.double();                         // Double the values of x and y.
      z = 2 * z;                              // Double the value of z.
    }
  }
```

Unlike superclass constructor invocations, ordinary superclass method invocations need not be the first statement in a method.

The "protected" Keyword

In the Vector class, we'd normally declare the fields "x" and "y" to be private. We usually don't want them to be public, because applications might corrupt or meddle with them, not using the proper interface. But if they're private, the Vector3D class can't read or change them directly, either.

So we declare "x" and "y" to be "protected", not "private".

```
  public class Vector {
    protected double x, y;

    ...
```

"protected" is a level of protection somewhere between "public" and "private". A "protected" field is visible to the declaring class and all its subclasses, but not to other classes. "private" fields aren't even visible to the subclasses.

If "x" and "y" are declared private, the method Vector3D.length() [see Lecture 24] can't access them and won't compile. If they're declared protected, Vector3D.length() can access them because Vector3D is a subclass of Vector.

When you write a class, if you think somebody might someday want to write a subclass of it, declare its vulnerable fields "protected", unless you have a reason for not wanting subclasses to see them. Helper methods often should be declared "protected" as well.

Subtleties of Inheritance
----

(1) Suppose we write a new method in the Cat class called meow(). We can't call meow() on an Animal. We can't even call meow() on a variable of type Animal that references a Cat.

```
  Cat c = new Cat();
  c.meow();                    // Groovy.
  Animal a = new Cat();        // Groovy--every Cat is an Animal.
  a.meow();                    // COMPILE-TIME ERROR.
```

Why? Because not every object of class Animal has a "meow()" method, so Java can't use dynamic method lookup to find the right "meow()" for the variable a.

But if we define meow() in Animal instead, the statements above compile and run without errors, even if no meow() method is defined in class Cat. (Cat inherits meow() from Animal.)

(2) You can't assign an Animal object to a Cat variable, because not every animal is a cat.

```
  Cat c = new Animal();        // COMPILE-TIME ERROR.
```

The rules are more complicated when you assign one variable to another.

```
  Animal a;
  Cat c = new Cat();
  a = c;                       // Groovy.
  c = a;                       // COMPILE-TIME ERROR.
  c = (Cat) a;                 // Groovy.
  a = new Animal();
  c = (Cat) a;                 // RUN-TIME ERROR:  ClassCastException.
```

Why does the compiler refuse "c = a", but accept "c = (Cat) a"? The cast in the latter statement is your way of reassuring the compiler that you've written the program to guarantee that the Animal "a" will always be a Cat.

If you're wrong, Java will find out when you run the program, and will crash with a "ClassCastException" error message. The error occurs only at run-time because Java cannot tell in advance what class of object "a" will reference.

(3) Java has an "instanceof" operator that tells you whether an object is of a specific class. WARNING: The "o" in "instanceof" is not capitalized.

```
  if (a instanceof Cat) {
    c = (Cat) a;
  }
```

This instanceof operation will return false if "a" is null or doesn't reference a Cat. It returns true if "a" references a Cat object--even if it's a subclass of Cat.