CHAPTER 7

Better Living in Objectville(EXERCISE)

INHERITENCE

If class B extends class A, class B IS-A class A. This is true anywhere in the inheritance tree. If class C extends class B, class C passes the IS-A test for both B and A.

Canine extends Animal

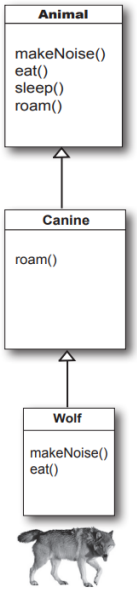
Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



With an inheritance tree like the one shown here, you're *always* allowed to say **"Wolf extends Animal"** or **"Wolf IS-A Animal."** It makes no difference if Animal is the superclass of the superclass of Wolf. In fact, **as long as Animal is *somewhere* in the inheritance hierarchy above Wolf, Wolf IS-A Animal will always be true.**

The structure of the Animal inheritance tree says to the world:

"Wolf IS-A Canine, so Wolf can do anything a Canine can do. And Wolf IS-A Animal, so Wolf can do anything an Animal can do."

It makes no difference if Wolf overrides some of the methods in Animal or Canine. As far as the world (of other code) is concerned, a Wolf can do those four methods. *How* he does them, or *in which class they're overridden,* makes no difference. A Wolf can makeNoise(), eat(), sleep(), and roam() because a Wolf extends from class Animal.

BULLET POINTS

- A subclass *extends* a superclass.
- A subclass inherits all *public* instance variables and methods of the superclass, but does not inherit the *private* instance variables and methods of the superclass.
- Inherited methods *can* be overridden; instance variables *cannot* be overridden (although they can be *redefined* in the subclass, but that's not the same thing, and there's almost never a need to do it.)
- Use the IS-A test to verify that your inheritance hierarchy is valid. If X *extends* Y, then X *IS-A* Y must make sense.
- The IS-A relationship works in only one direction. A Hippo is an Animal, but not all Animals are Hippos.
- When a method is overridden in a subclass, and that method is invoked on an instance of the subclass, the overridden version of the method is called. (*The lowest one wins.*)
- If class B extends A, and C extends B, class B IS-A class A, and class C IS-A class B, and class C also IS-A class A.

Overloading a method

Method overloading is nothing more than having two methods with the same name but different argument lists. Period. There's no polymorphism involved with overloaded methods!

Overloading lets you make multiple versions of a method, with different argument lists, for convenience to the callers. For example, if you have a method that takes only an int, the calling code has to convert, say, a double into an int before calling your method. But if you overloaded the method with another version that takes a double, then you've made things easier for the caller. You'll see more of this when we look into constructors in Chapter 9, *Life and Death of an Object*.

Since an overloading method isn't trying to fulfill the polymorphism contract defined by its superclass, overloaded methods have much more flexibility.

An overloaded method is just a different method that happens to have the same method name. It has nothing to do with inheritance and polymorphism. An overloaded method is NOT the same as a overridden method.

① The return types can be different.

You're free to change the return types in overloaded methods, as long as the argument lists are different.

② You can't change ONLY the return type.

If only the return type is different, it's not a valid over*load*—the compiler will assume you're trying to over*ride* the method. And even *that* won't be legal unless the return type is a subtype of the return type declared in the superclass. To overload a method, you MUST change the argument list, although you *can* change the return type to anything.

③ You *can* vary the access levels in any direction.

You're free to overload a method with a method that's more restrictive. It doesn't matter, since the new method isn't obligated to fulfill the contract of the overloaded method.

Legal examples of method overloading:

```
public class Overloads {
  String uniqueID;

  public int addNums(int a, int b) {
    return a + b;
  }

  public double addNums(double a, double
    return a + b;
  }

  public void setUniqueID(String theID)
    // lots of validation code, and then
    uniqueID = theID;
  }

  public void setUniqueID(int ssNumber)
    String numString = "" + ssNumber;
    setUniqueID(numString);
  }
}
```

EXERCISE

1. Mixed Messages

a) B's m1, A's m2, A's m3,
b) B's m1, A's m2, C's m3, 13
c) A's m1, A's m2, C's m3, 13
d) B's m1, A's m2, C's m3, 13

2. BE the Compiler

SET1

```
boolean frighten(int d) {
System.out.println("arrrgh");
return true;
}
boolean frighten(int x) {
System.out.println("a bite?");
return false;
}
```

3.POOL PUZZLE

```
public class Rowboat extends Boat {
    public void rowTheBoat() {
        System.out.print("stroke natasha");
    }
}
public class Boat{
    private int length;
    public void setLength( int len ) {
        length = len;
    }
    public int getLength() {
        return length;
    }
    public void move() {
        System.out.print("drift");
    }
}
public class TestBoats {
    public static void main(String[] args) {
        Boat b1 = new Boat();
        Sailboat b2 = new Sailboat();
        Rowboat b3 = new Rowboat();
```

```
        b2.setLength(32);
        b1.move();
        b3.move();
        b2.move();
    }
}
public class Sailboat extends Boat {
    public void move() {
        System.out.print("hoist sail “);
    }
}
```