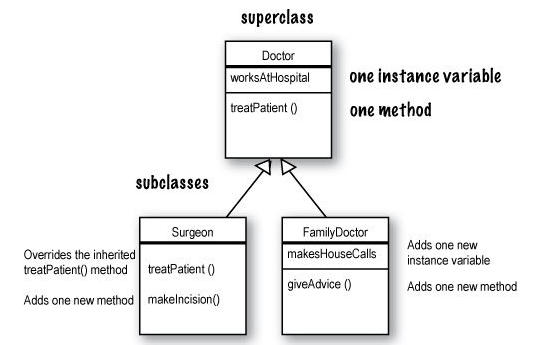
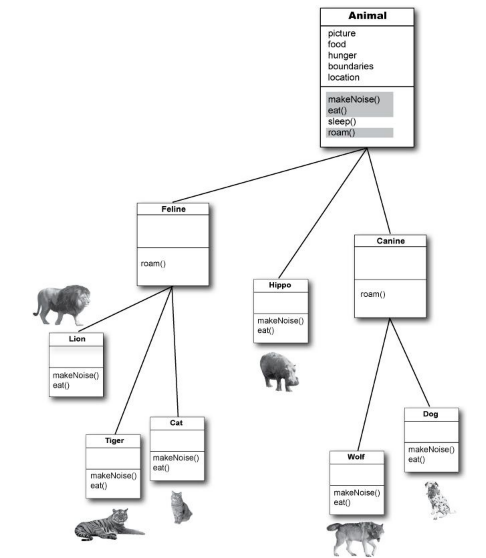
**B”H**

**Chapter 7**

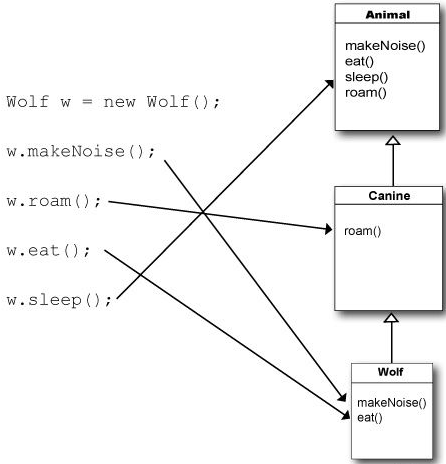
* A subclass extends the superclass.
* A subclass inherits the members of the superclass. I.e. the instance variables and methods (later we’ll look at other inherited members).
* A subclass can add new methods and instance variables of its own, and it can override the methods it inherits from the superclass
* Instance variables are not overridden because they don’t need to be. They don’t define any special behavior, so a subclass can give an inherited instance variable any value it chooses.



* One purpose of inheritance is to avoid duplicating code in subclasses

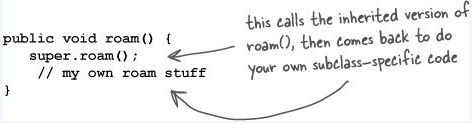


* When you call a method on an object reference, you’re calling the most specific version of the method for that object type. In other words, the lowest one wins!

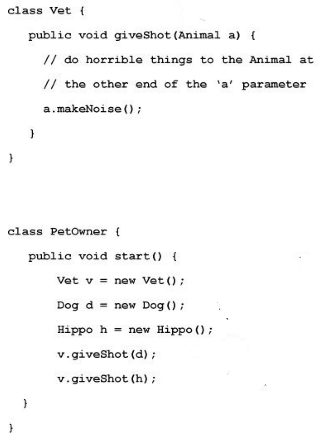


**Using IS-A and HAS-A**

* When you want to know if one thing should extend another, apply the IS-A test. For example:
  + Triangle IS-A Shape
  + Cat IS-A Feline
  + Etc.
* However Tub is **not** a Bathroom, rather “Bathroom HAS-A Tub” - I.e. Bathroom has a Tub instance variable (object reference).
* Remember if B extends class A and class C extends class B, class C passes the IS-A test for both B and A.
* 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.”
* Keep in mind that the inheritance IS-A relationship works in only one direction;
  + Hippo is an Animal, but not all Animals are Hippos.
* A superclass cannot use the subclass version of a method - children inherit from parents, not the other way around.
* In a subclass, you can use BOTH the superclass version and the overriding subclass version of a method. Hence the word “extends”:



* A superclass can choose whether or not it wants a subclass to inherit a particular member
  + **public** members are inherited and are as if the subclass defined the member itself
  + **private** members are not inherited
  + For **default** and **protected** see Chapter 17 and Appendix B .
* Don’t abuse inheritance - while inheritance is one of the key features of object-oriented programming, it’s not necessarily the best way to achieve behavior reuse – see Head First Design Patterns.
* In the Java API, most inheritance hierarchies are wide but not deep - 1 or 2 levels deep, although there are exceptions (especially in the GUI classes). You’ll come to realize that it usually makes more sense to keep your inheritance trees shallow.
* If you don’t have access to the source code for a class, but you want to change the way a method of that class works, you can extend the “bad” class and override the method with your own better code.
* So inheritance helps you:
  + Avoid writing duplicate code.
  + Guarantee that all classes grouped under a certain supertype have all the methods that the supertype has. I.e. you get to establish a contract.
  + Polymorphism - when you define a supertype for a group of classes, any subclass of that supertype can be substituted where the supertype is expected. This makes your code:
    - Flexible
    - Cleaner
    - Simpler
    - Easier to extend
* With polymorphism, the object reference variable’s type can be a superclass of the actual object type.
  + **Animal myDog = new Dog();**
* You can also have polymorphic arguments and return types.
* With polymorphism, you can write code that doesn’t have to change when you introduce new subclass types into the program.



* There’s no such thing as a private class, except in a very special case called an inner class, that we haven’t looked at yet. But there are three things that can prevent a class from being subclassed.

1. Access control. A non-public class can be subclassed only by classes in the same package as the class.
2. The keyword modifier final (for example the String class). A final class means that it’s the end of the inheritance line. Nobody, ever, can extend a final class. If you just want to protect a specific method from being overridden, mark the method the final.
3. If a class has only private constructors (see Chapter 9).

**Rules for overriding:**

1. Arguments must be the same, and return types must be compatible (either the same type or a subclass type).
2. The method can’t be less accessible; the access level must be the same or friendlier.
3. There’s also another rule about overriding related to exception handling (see chapter 11).

**Rules for overloading**: overloading is nothing more than having two methods with the same name but different argument lists. It has nothing to do with inheritance and polymorphism.

1. The return types can be different.
2. You can’t change ONLY the return type. To overload a method, you MUST change the argument list.
3. You can vary the access levels in any direction.