Inheritance and Polymorphism

Understanding Inheritance

When you design with inheritance, you put common code in a class and then tell other more specific classes that the common (more abstract) class is their superclass.

When one class inherits from another, **the subclass inherits from the superclass.**

In Java, we say the **subclass *extends* the superclass.**

Understanding Inheritance

An inheritance relationship means that the subclass inherits the **members** of the superclass.

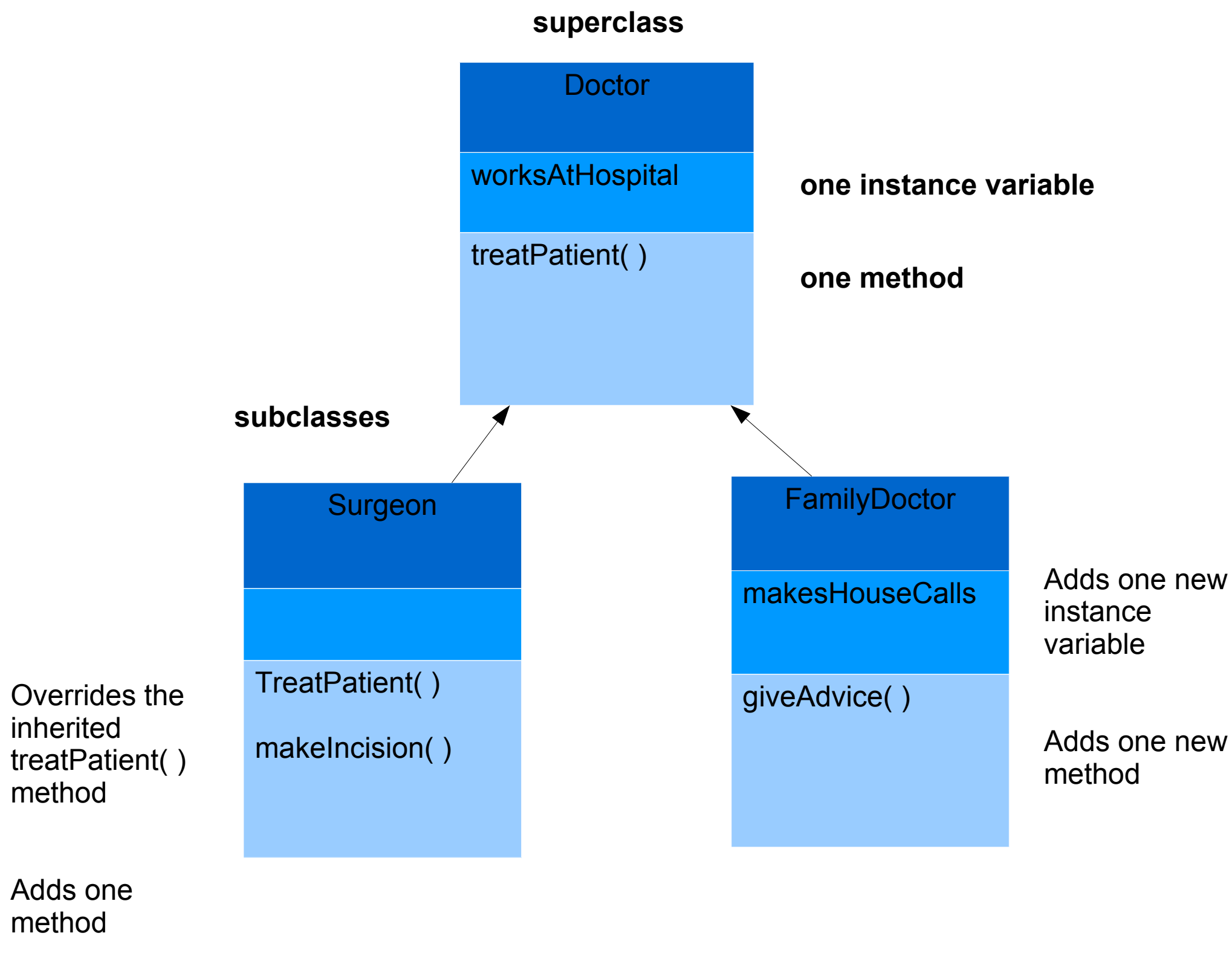
Members are the **instance variables and methods**

The **subclass can add new methods and instance variables** of its own, and it **can override methods it inherits from the superclass.**

Instance variables are not overriden because they don't need to be. They don't define any special behavior, so a subclasse can give an inherited instance variable any value it chooses.

An inheritance example:

```
public class Doctor {
        boolean worksAtHospital;

        void treatPatient( ) {
                perform a checkup
        }
}

public class FamilyDoctor extends Doctor {
        boolean makesHouseCalls;

        void giveAdvice( ) {
                give homespun advice
        }
}

public class Surgeon extends Doctor{
        void treatPatient( ) {
                perform surgery
        }

        void makeIncision( ) {
                make incision
        }
}
```


superclass
Doctor
worksAtHospital
treatPatient( )
one instance variable
one method
subclasses
Surgeon
TreatPatient( )
makeIncision( )
FamilyDoctor
makesHouseCalls
giveAdvice( )
Overrides the inherited treatPatient( ) method
Adds one method
Adds one new instance variable
Adds one new method

Designing for inheritance

We've been asked to design a program to catalog different animals. We want other programmers to be able to add new kinds of animals to the program at any time.

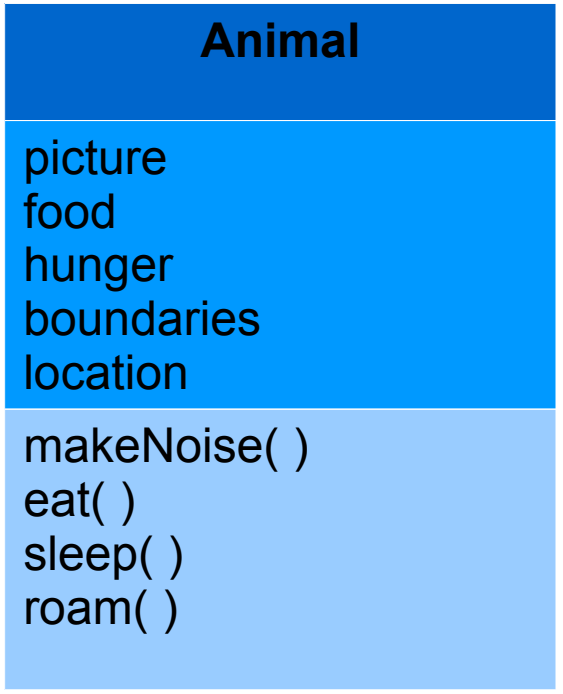
1) Look for objects that have common attributes and behaviors.

- What do the given animal types have in common? This helps abstract behaviors.
- How are these types related? This helps you to define the inheritance tree relationships.

Designing for inheritance

2) Design a class that represents the common state and behavior.

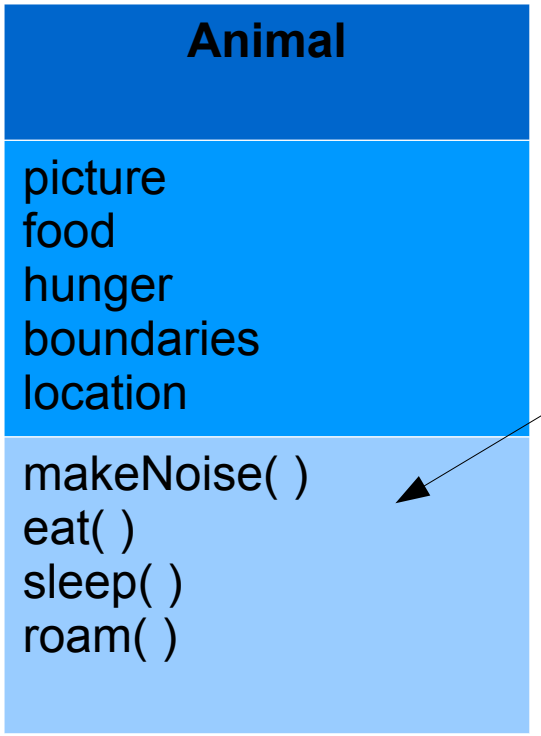
- These objects are all animals so we'll make a common superclass called Animal.
- We'll put in methods and instance variables that all animals might need.



Designing for inheritance

3) Decide if a subclass needs behaviors (method implementations) that are specific to that particular subclass type

- Looking at the Animal class, we decide that eat( ) and makeNoise( ) should be overridden by the individual subclasses.

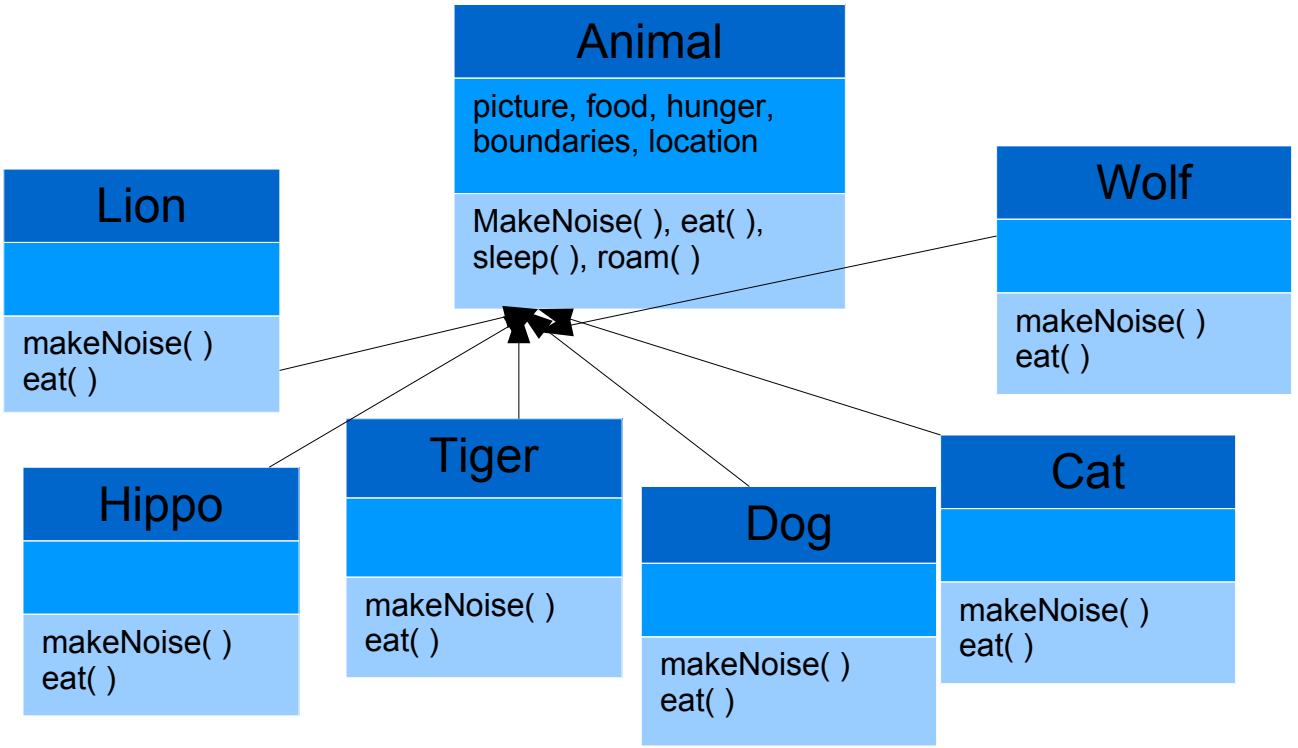


We should override the two methods, eat( ) and makeNoise( ), so that each animal type can define its own specific behavoir for eating and making noise.

Designing for inheritance

4) Look for more opportunities to use abstraction, by finding two or more subclasses that might need common behavior.

- We look at our classes and see that Wolf and Dog might have some behavior in common, and the same goes for Lion, Tiger, and Cat.

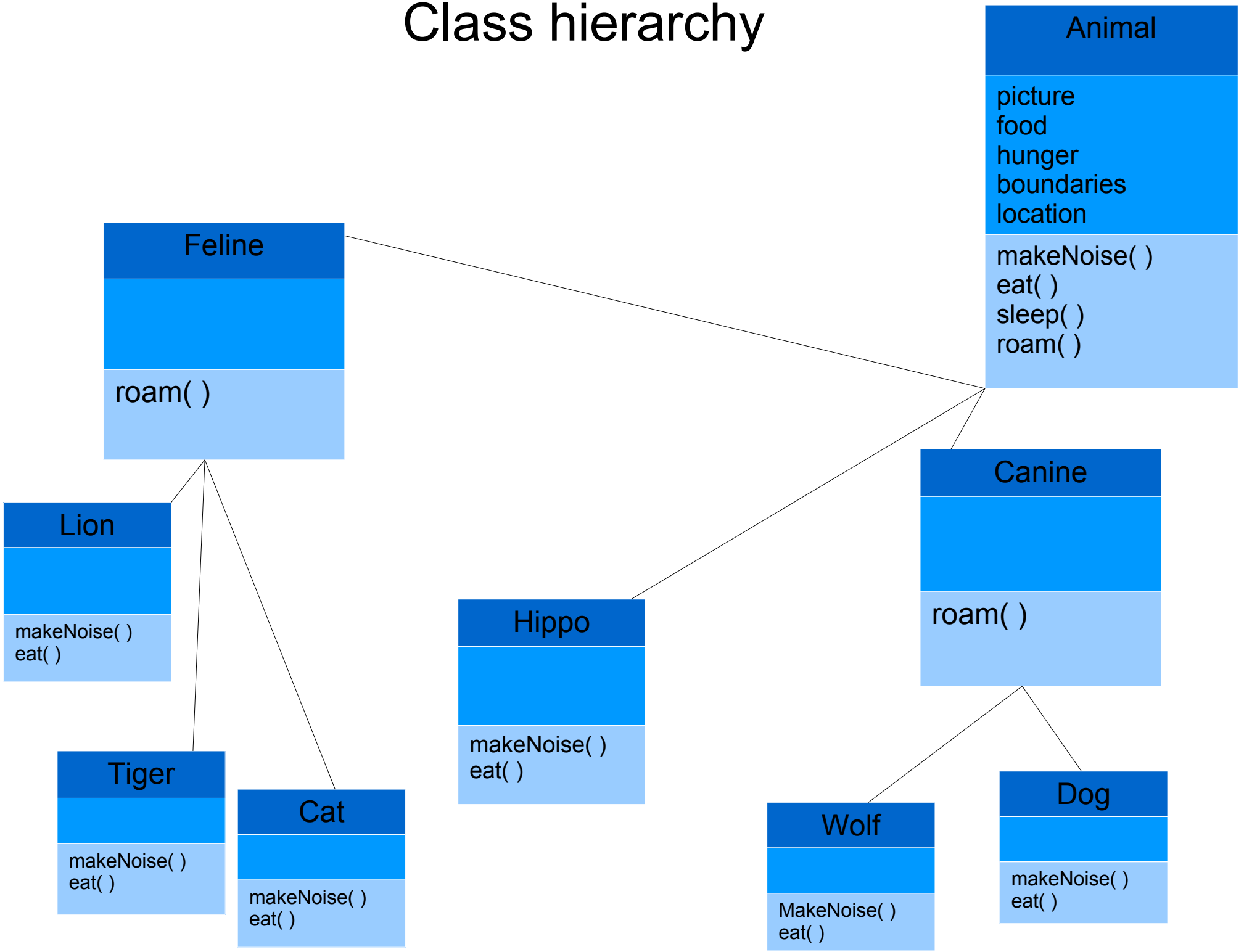


Designing for inheritance

5) Finish the class hierarchy.

- We decide that Canines could use a common roam( ) method. We also decide that Felines could use a common roam( ) method. Hippo will continue to use its inherited roam( ) method – the generic one it gets from the Animal class.

Class hierarchy



Designing for inheritance

Which method is called?

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!***

"Lowest" meaning lowest on the inheritance tree.

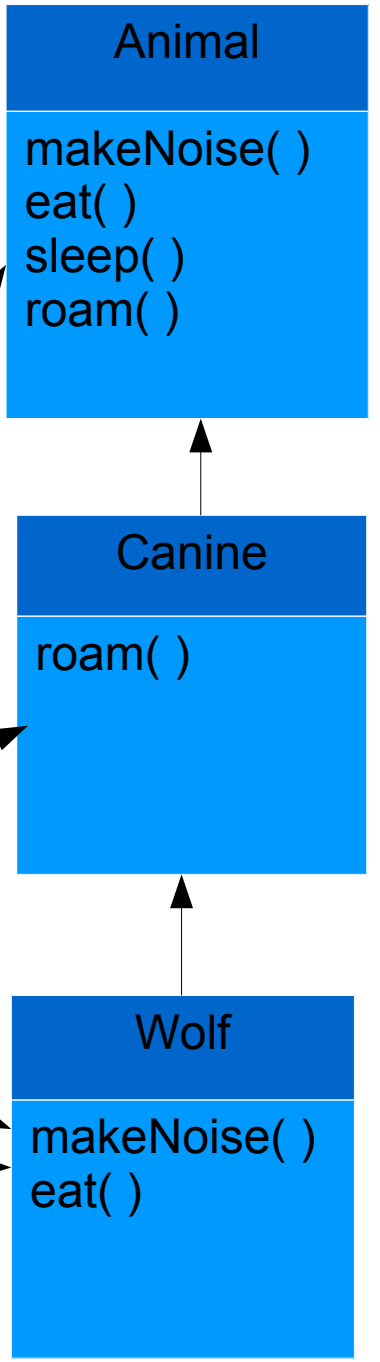
Make a new Wolf object Wolf w = new Wolf( );

Calls the version in Wolf w.makeNoise( );

Calls the version in Canine w.roam( );

Calls the version in Wolf w.eat( );

Calls the version in Animal w.sleep( );



Designing for inheritance

Using IS-A and HAS-A

When one class inherits from another, we say the subclass *extends* the superclass. When you want to know if one thing should extend another, apply the IS-A test.

- Triangle IS-A Shape
- Surgeon IS-A Doctor

Tub IS-A Bathroom?

False. Tub and Bathroom are related, but not through inheritance. Tub and Bathroom are joined by a HAS-A relationship. This means Bathroom has a Tub instance variable, or Bathroom has a *reference* to a Tub, but Bathroom does not *extend* Tub and vice-versa.

Designing for inheritance

If class B extends class A, calss 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

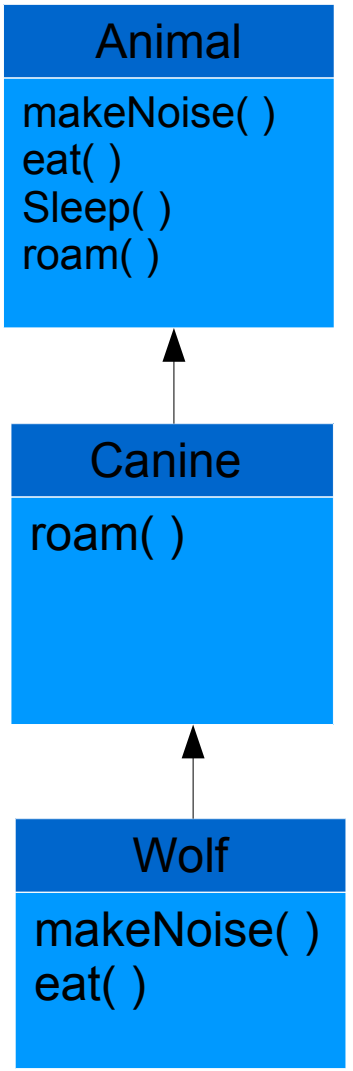
Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



You're always allowed to say "**Wolf extends Animal**" or "**Wolf IS-A Animal**" as long as Animal is somewhere in the inheritance hierarchy above Wolf, Wolf IS-A Animal will always be true.

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.

Designing for inheritance

Keep in mind that the inheritance IS-A relationship works in only *one* direction!

Triangle IS-A Shape makes sense, so yu can have Triangle extend Shape.

The reverse – Shape IS-A Triangle – does *not* make sense, so Shape should not extend Triangle. Remember, that the IS-A relationship implies that if X IS-A Y, then X can do anything a Y can do (and possibly more).

Designing for inheritance

How do you know what a subclass can inherit from its superclass?

A subclass inherits members of the superclass. Members include instance variables and methods.

A supercalss can shoose wheter or not it wants a subclass to inherit a particular member by the level of access the particular member is given.

There are four access levels. Moving from most restrictive to least, the four access levels are:

private **default** **protected** **public**

Designing for inheritance

Access levels control *who sees what*. For now we'll focus on public and private. The rules are simple for these two:

public members <u>are</u> inherited

private members are <u>not</u> inherited

When a subclass inherits a member, it is ***as if the subclass defined the member itself.***

Do's and don't s of inheritance

DO use inheritance when one class is a more specific type of a superclass.

DO consider inheritance when you have behavior (implemented code) that should be shared among multiple classes of the same general type.

DO NOT use inheritance just so that you can resue code from another class, if the relationship between the superclass and subclass violate either of the above two rules.

DO NOT use inheritance if the subclass and superclass do not pass the IS-A test. Always ask yourself if the subclass IS-A more specific type of the superclass.

Takeaways

- 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
- 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.

Why use inheritance?

1) **You avoid duplicate code.**

- Put common code in one place, and let the subclasses inherit that code from a superclass. When you want to change that behavior you have to modify it in only one place, and all the subclasses see the change.

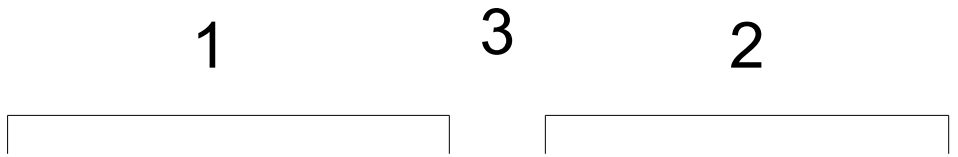
2) **You define a common protocol for a group of classes.**

- Inheritance lets you guarantee that all classes grouped under a certain supertype have all the methods that supertype has.

The way polymorphism works

To see how polymorphism works, we have to take a step back and look at the way we *normally* declare a reference and create and object...

The 3 steps of object declaration and assignment



1) Declare a reference variable

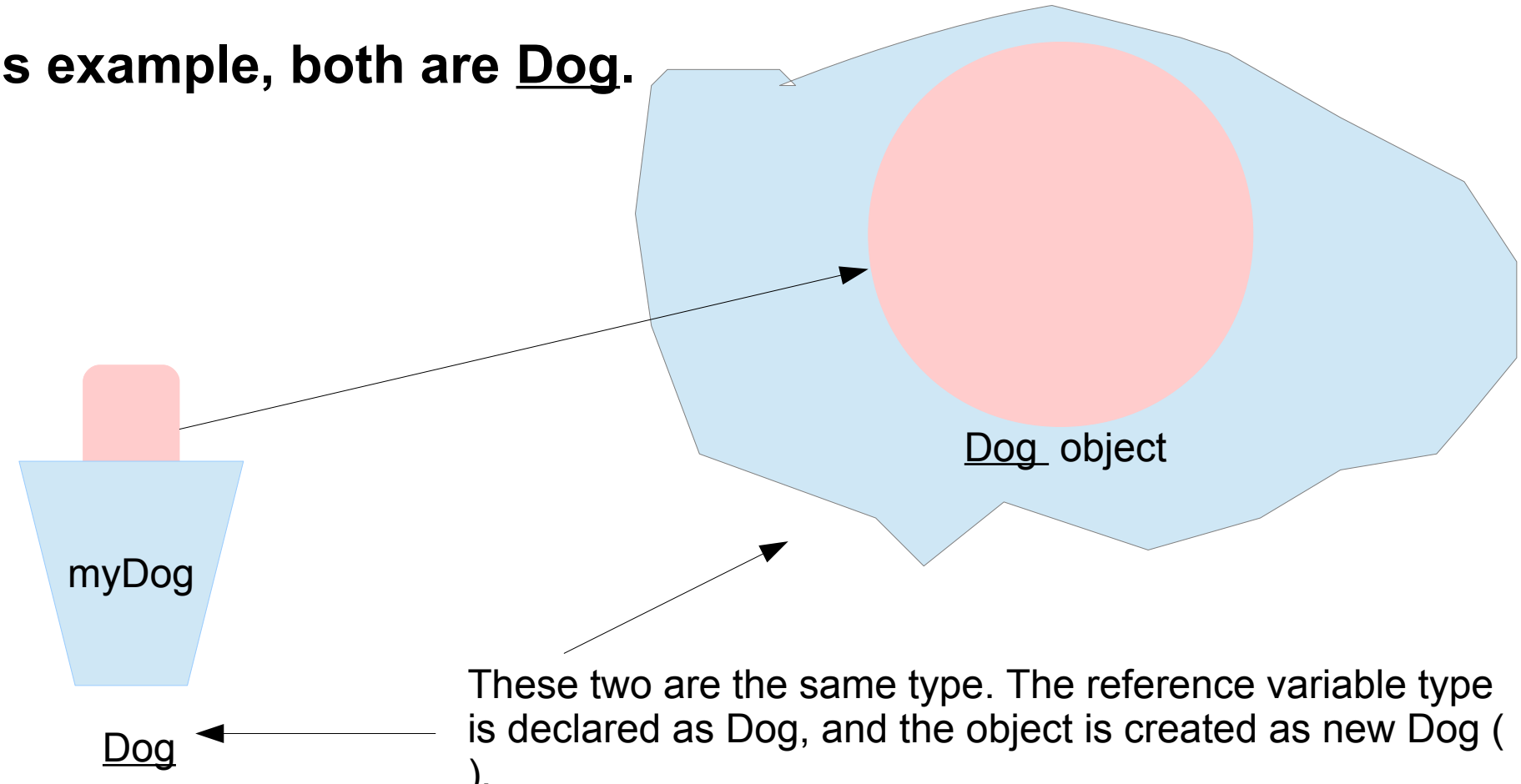
2) Create and object

3) Link the object and the reference

The way polymorphism works

The important point is that the reference type AND the object type are the same.

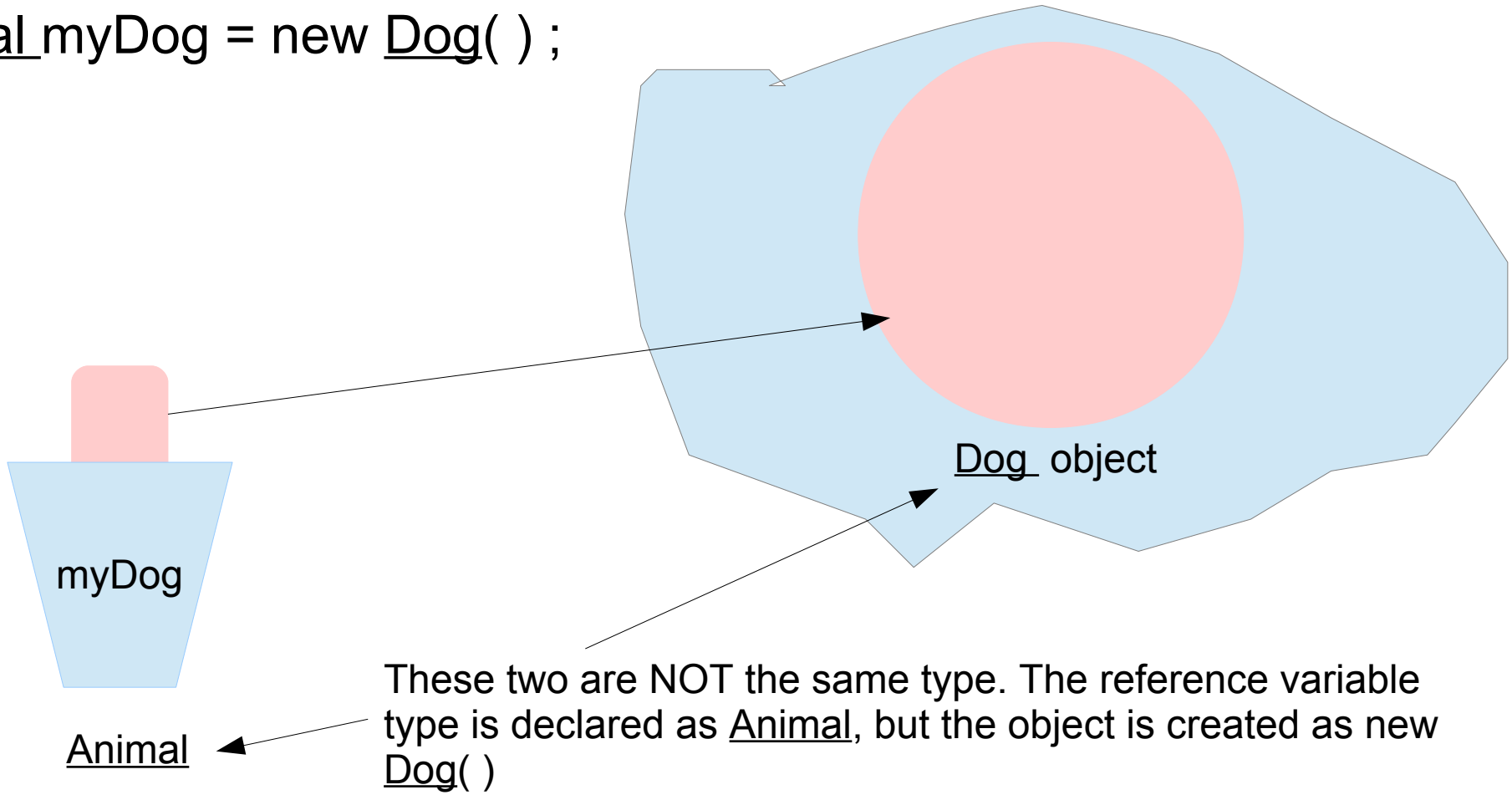
In this example, both are Dog.



The way polymorphism works

But with polymorphism, the reference and the object can be *different*.

Animal myDog = new Dog( ) ;



The way polymorphism works

With polymorphism, the reference type can be a superclass of the actual object type.

When you declare a reference variable, any object that passes the IS-A test for the reference variable can be assigned to that reference. In other words, anything that *extends* the declared reference variable type can be *assigned* to the reference variable. ***This lets you do thinks like make polymorphic arrays.***

The way polymorphism works

```
Animal [ ] animals = new Animal [5];
    animals [0] = new Dog( );
    animals [1] = new Cat( );
    animals [2] = new Wolf( );
    animals [3] = new Hipppo( );
    animals [4] = new Lion( );

    for (int i = 0; i < animals.length; i++) {
        animals[i].eat( );
        animals[i].roam( );
        }
```

Declare an array of type Animal (an array that will hold objects of the type Animal)

You can put ANY subclass of Animal in the Animal array.

You can loop through the array and call one of the Animal-class methods, and every object does the right thing.

The way polymorphism works

You can have polymorphic <u>arguments</u> and <u>return</u> types.

If you can declare a reference variable of a supertype, and assign a subclass object to it, think of how that might work when the reference is an argument to a method...

```
class Vet {
      public void giveShot(Animal a) {
            a.makeNoise( );
      }
}
```

The Animal parameter can take ANY Animal type as the argument. When the Vet is done giving the shot, it tells the Animal to makeNoise( ) and whichever Animal is instantiated is whose makeNoise( ) method will run.

```
class PetOwner {
      public void start ( ) {
            Vet v = new Vet( );
            Dog d = new Dog( );
            Hippo h = new Hippo( );
            v.giveShot(d);
            v.giveShot(h);
      }
}
```

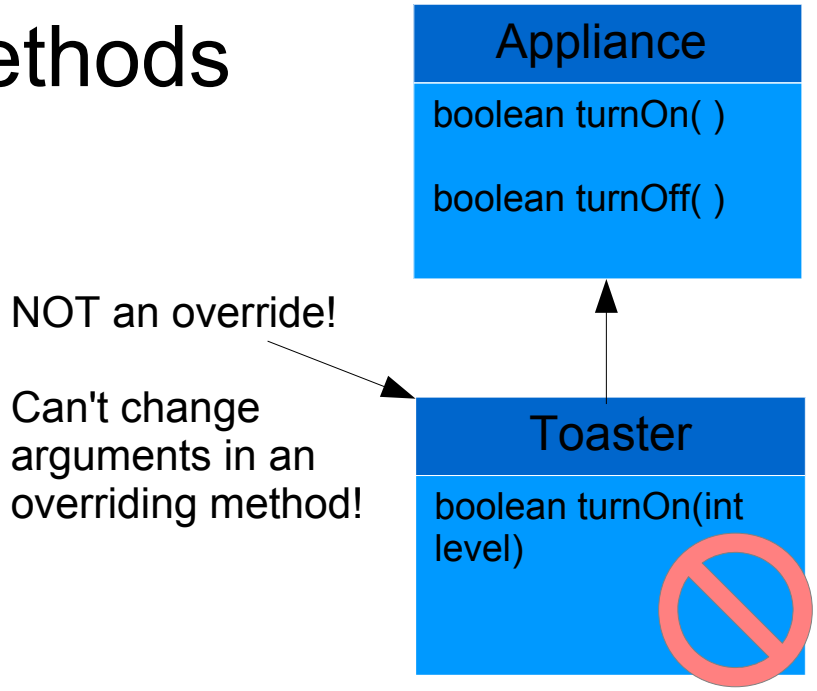
The Vet's giveShot( ) method can take any Animal you give it. As long as the object you pass in as the argument is a subclass of Animal, it will work.

Dog's makeNoise( ) runs

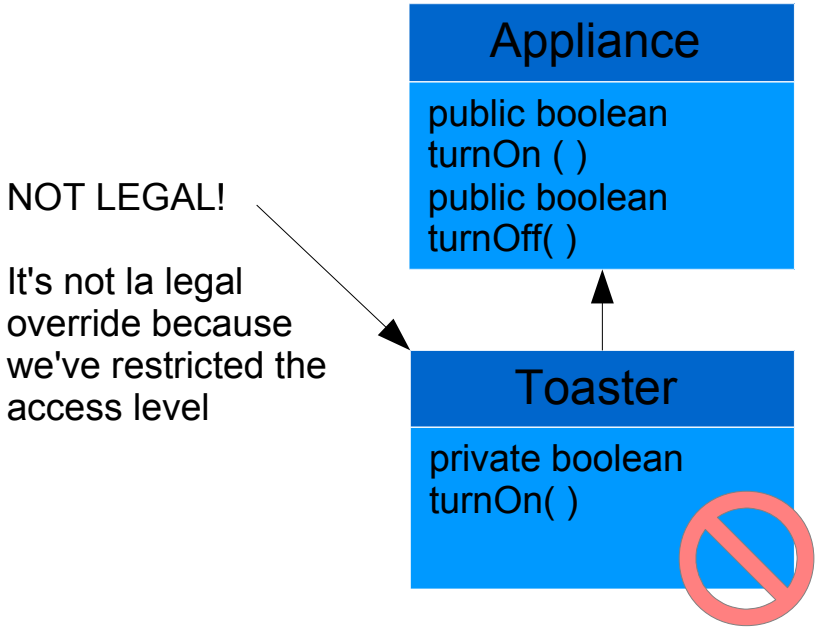
Hippo's makeNoise( ) runs

Overriding methods

1. Arguments must be the same, and return types must be compatible.
The superclass defines how other code can use a method. Whatever the superclass takes as an argument, the subclass overriding the method must use that same agrument. And whatever the superclass declares as a return type, the overriding method must declare eithe rthe same type, or a subclass type.



2. The method can't be less accessible.
The access level must be the same, or less restrictive. That means you can't override a public method and make it private.



Overloading a method

Method overloading is nothing more than having two methods with the same name but different argument lists. 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.

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

Overloading a 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 *overload* – the compiler will assume you're trying to *override* the method. To overload a method, you MUST change the argument list, although you can change the return type to anything you want.

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.

Overloading a method

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

Legal example of method overloading:

```
public class Overloads {
    String  uniqueID;

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

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