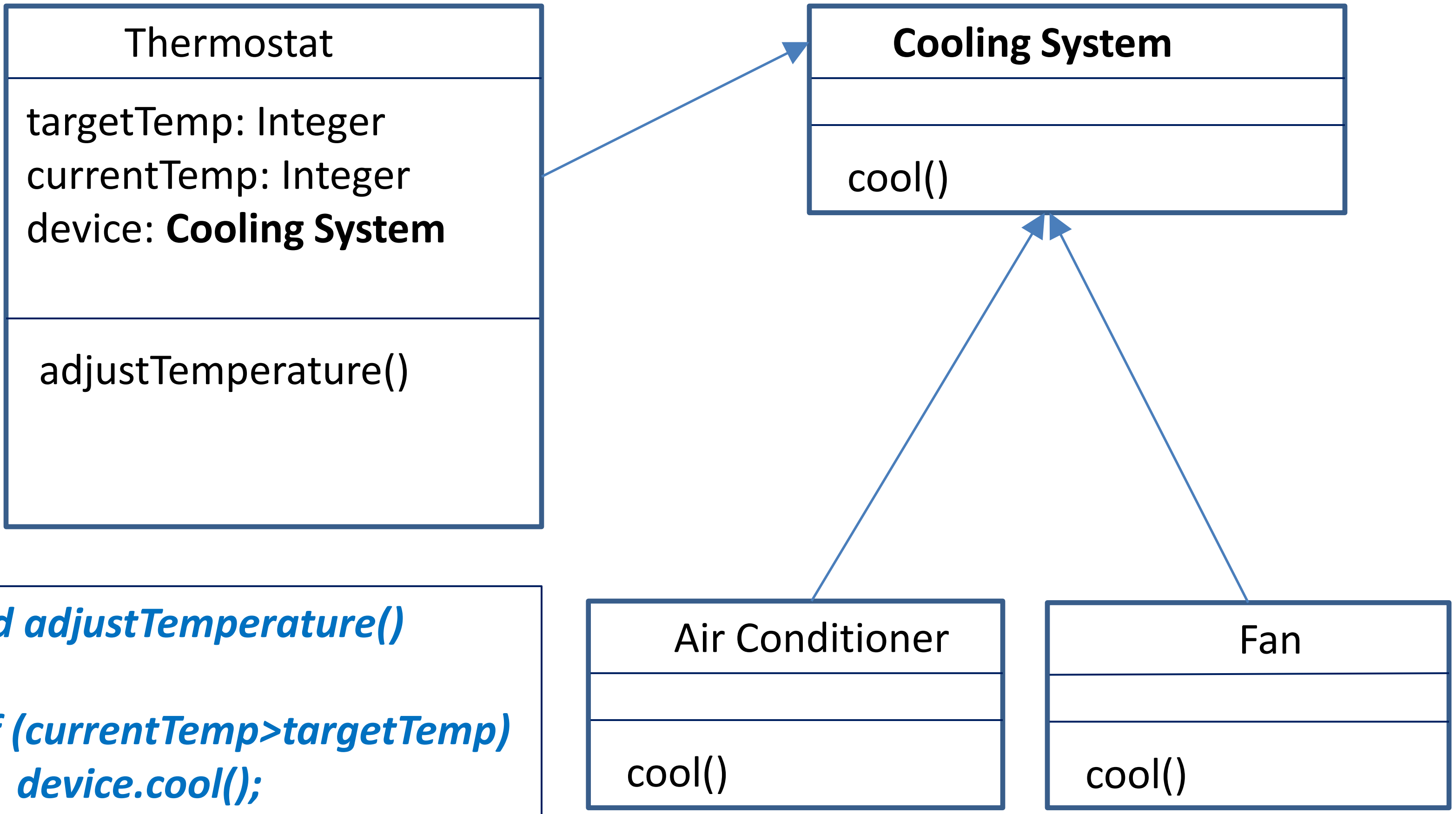
Polymorphism. Abstraction. Interface

Lecture 8

Is-a vs. ***is-like-a*** relationship

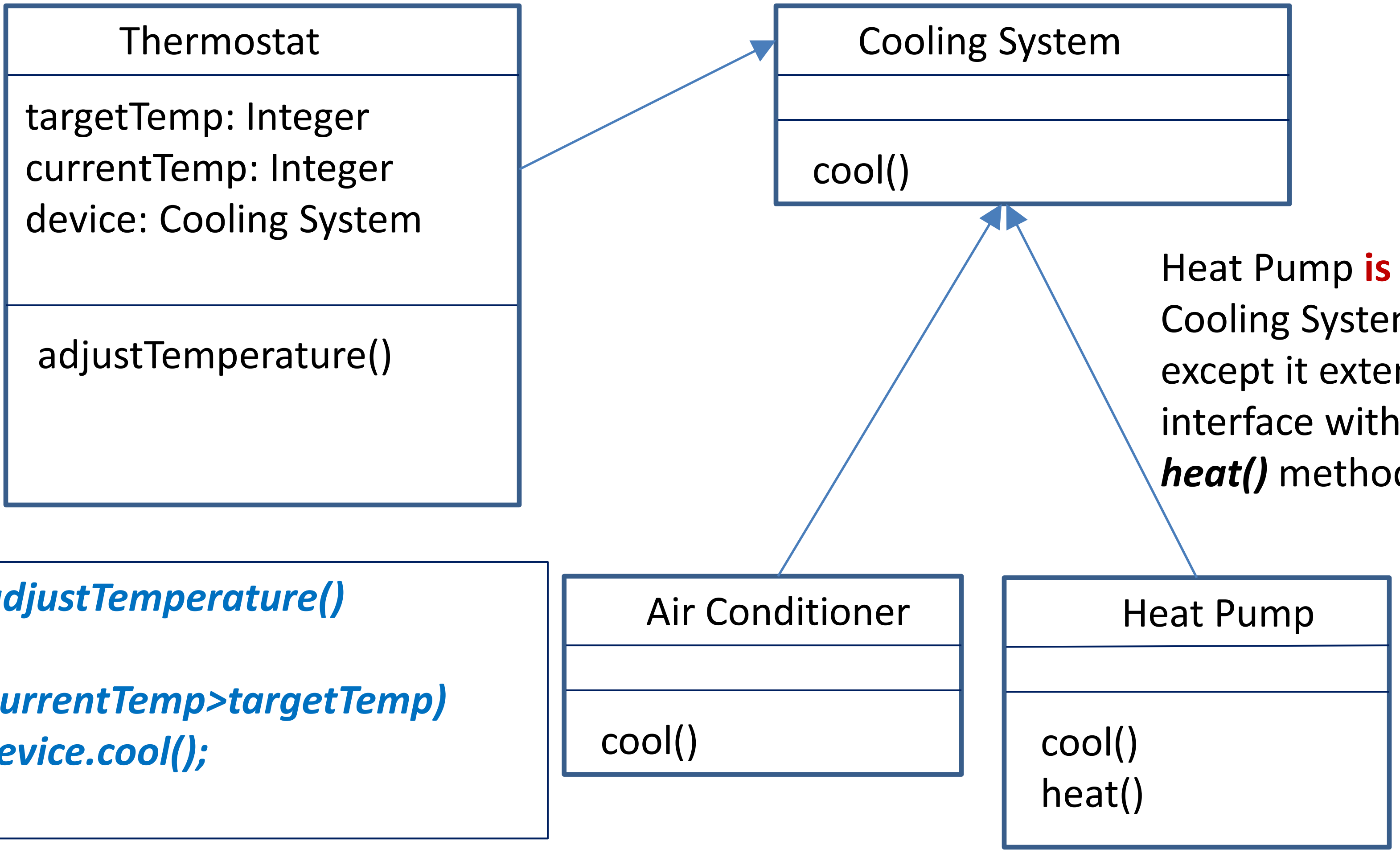
- A test for inheritance is to determine whether you can state the ***is-a*** relationship about the classes and have it make sense.
- There are times when you must add new interface elements to a derived type, thus **extending** the interface.
- The new type can still be substituted for the base type, but the substitution isn't perfect because your new methods are not accessible from the base type. This can be described as an ***is-like-a*** relationship.

Is-like-a example 1/4



```
void adjustTemperature()
{
   if (currentTemp>targetTemp)
      device.cool();
}
```

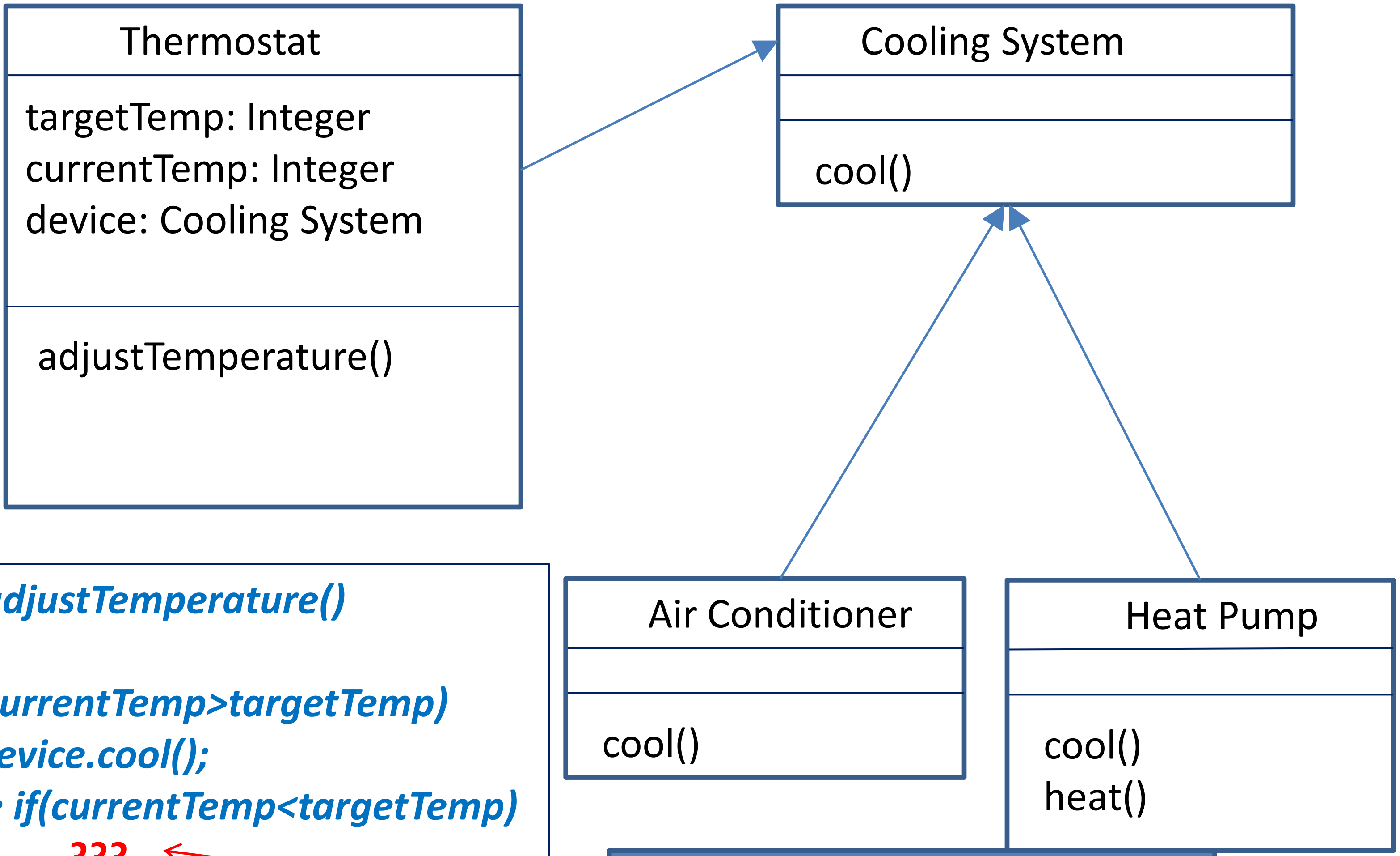
Is-like-a example 2/4



Heat Pump **is like a** Cooling System, except it extends its interface with ***heat()*** method

```
void adjustTemperature()
{
   if (currentTemp>targetTemp)
      device.cool();
}
```

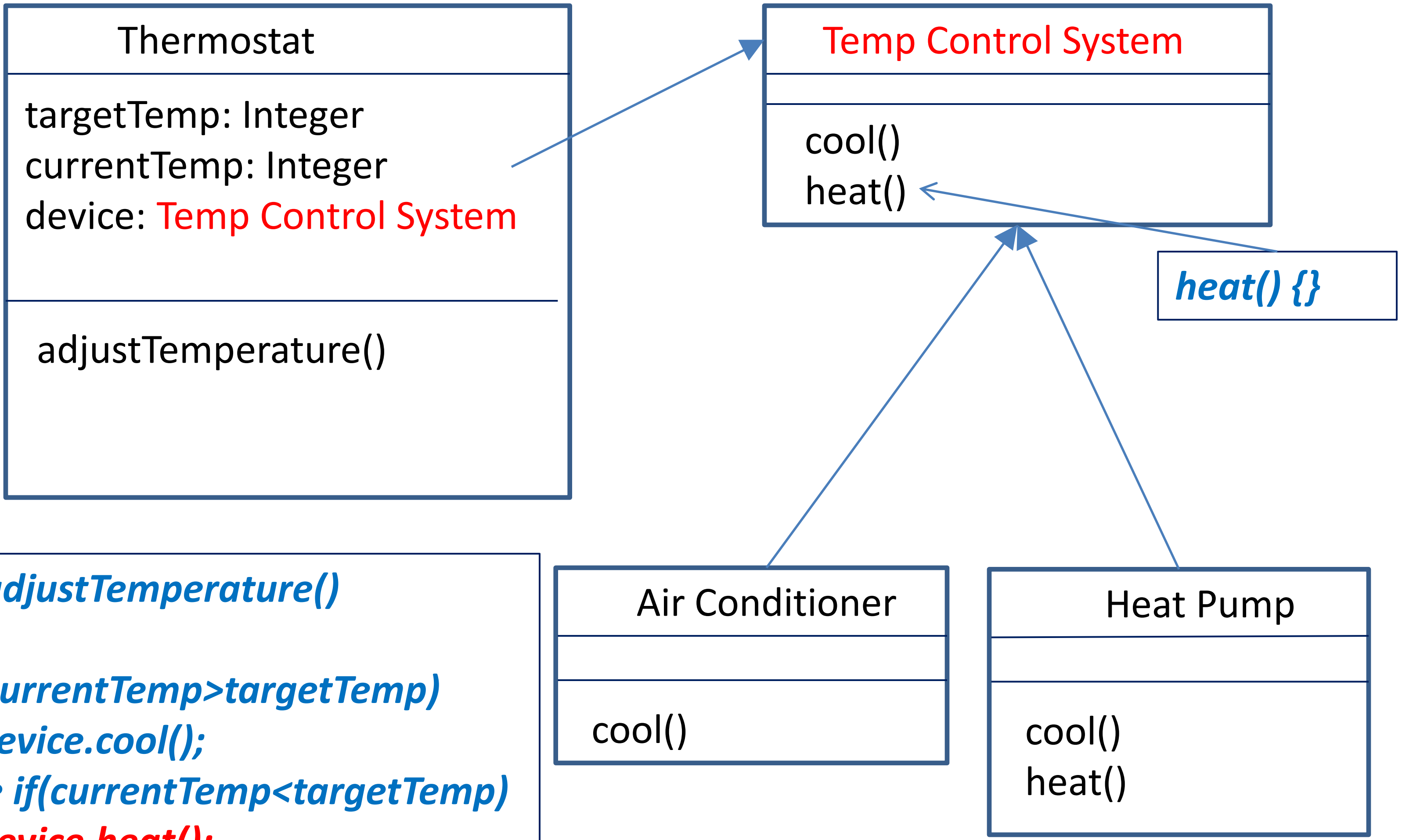
Is-like-a example 3/4



```
void adjustTemperature()
{
   if (currentTemp>targetTemp)
      device.cool();
   else if(currentTemp<targetTemp)
            ???
}
```

Cooling System base class does not have method *heat()*

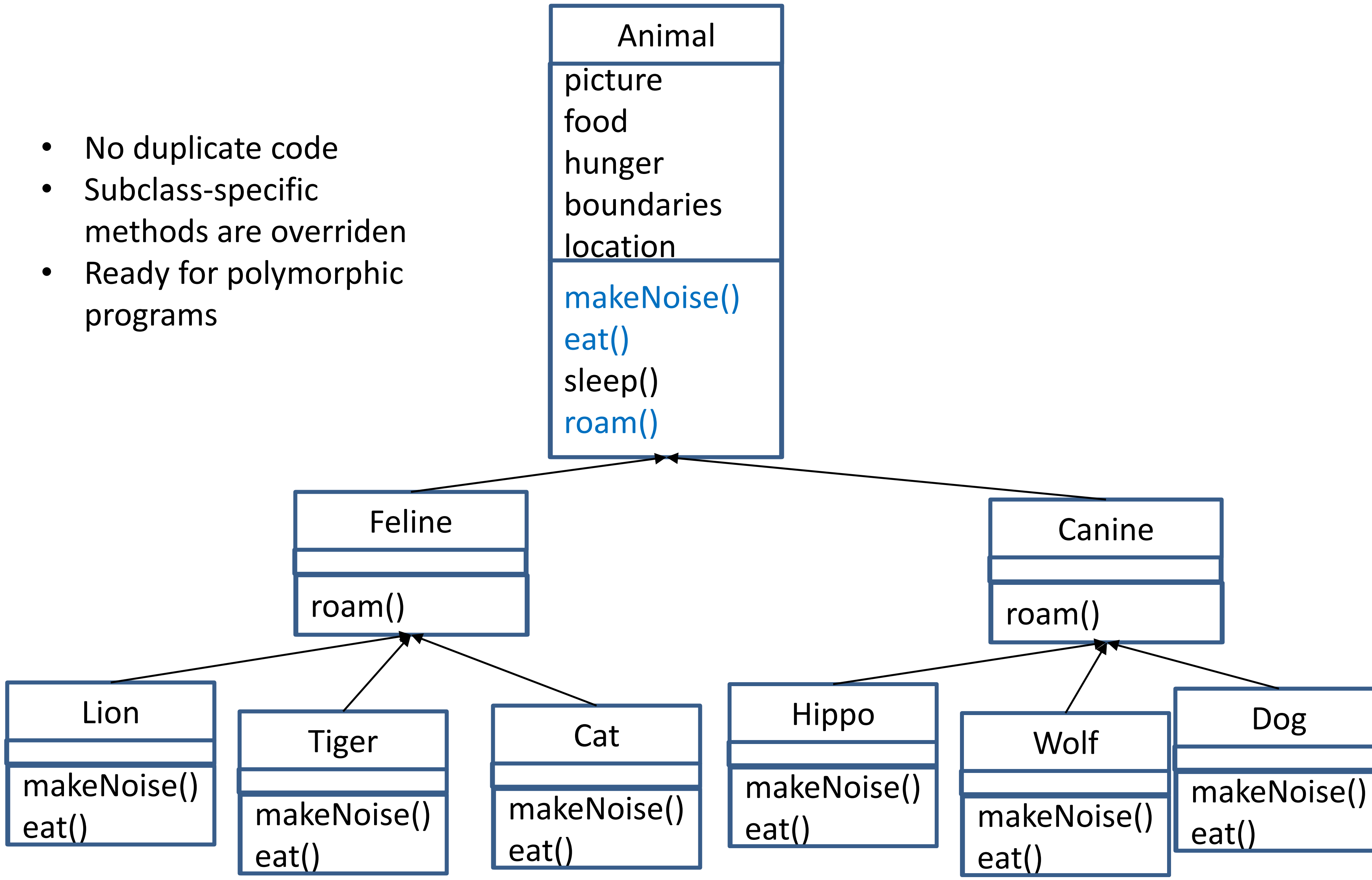
Is-like-a example: fix to is-a



```
void adjustTemperature()
{
   if (currentTemp>targetTemp)
      device.cool();
   else if(currentTemp<targetTemp)
      device.heat();
}
```

Back to Animal Simulation program design

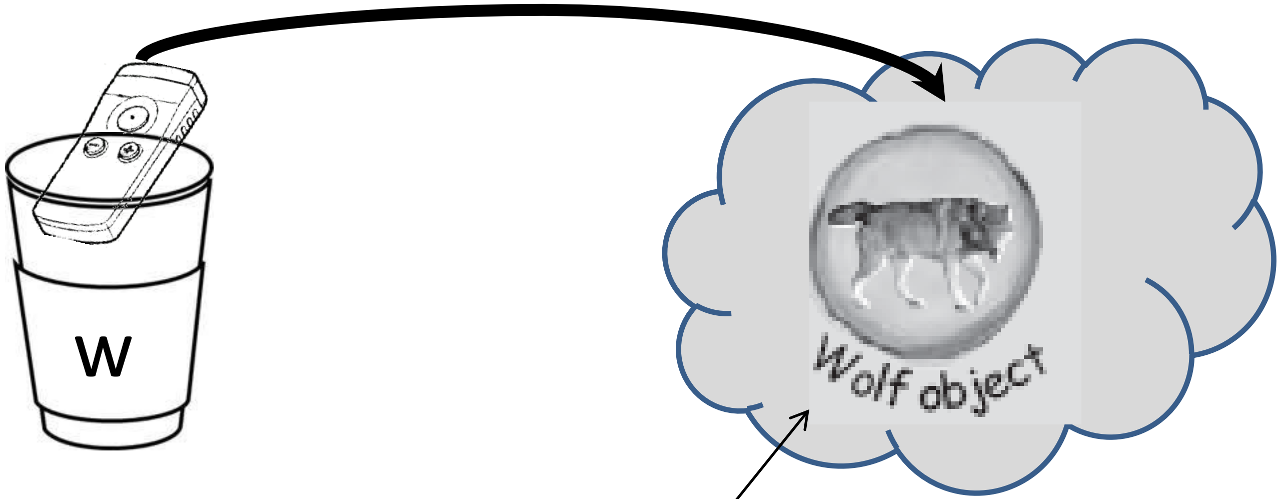
- No duplicate code
- Subclass-specific methods are overriden
- Ready for polymorphic programs



We know we can say:

Wolf w=new Wolf();

A Wolf reference to a Wolf object

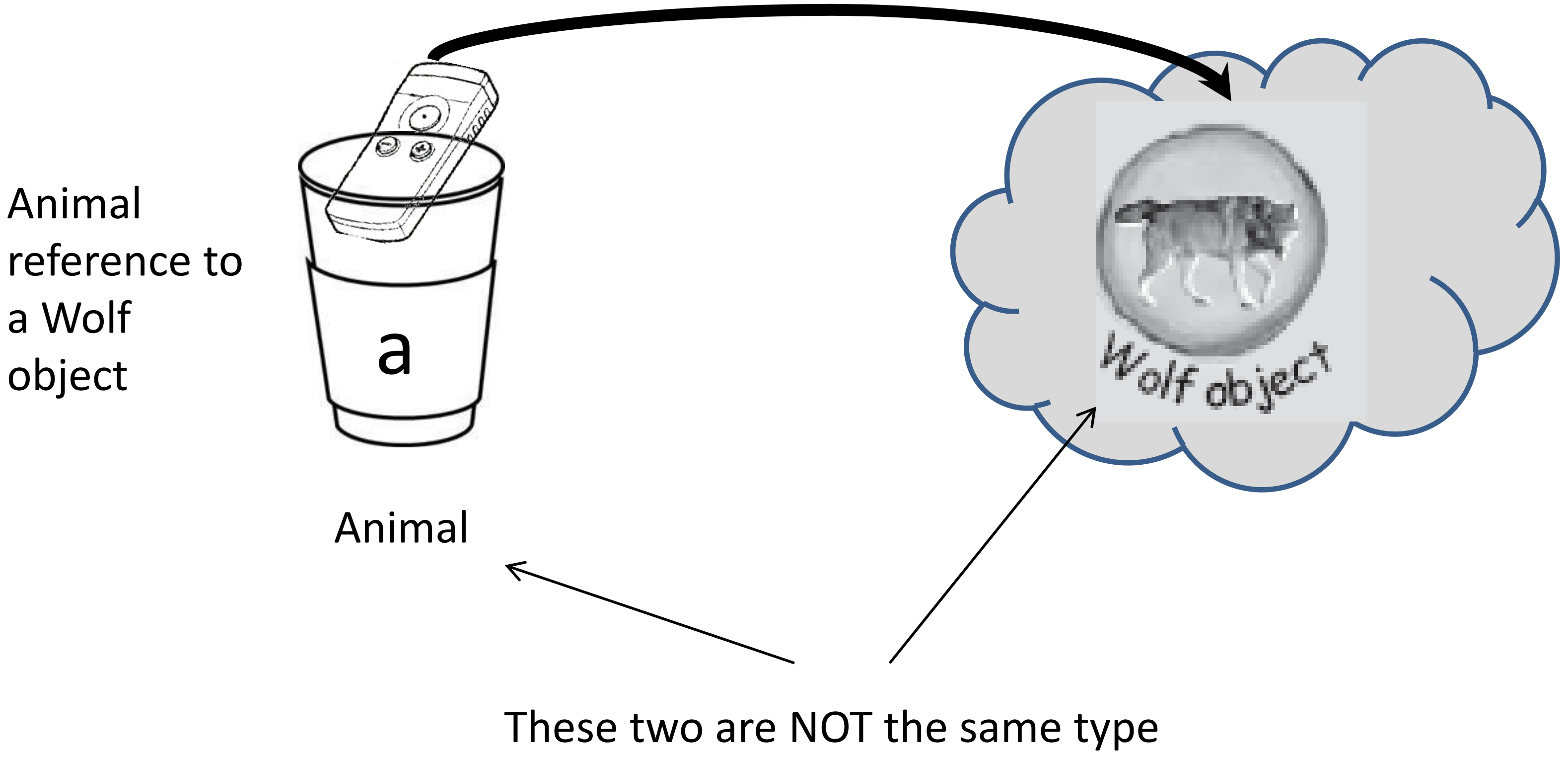


Wolf

These two are the same type

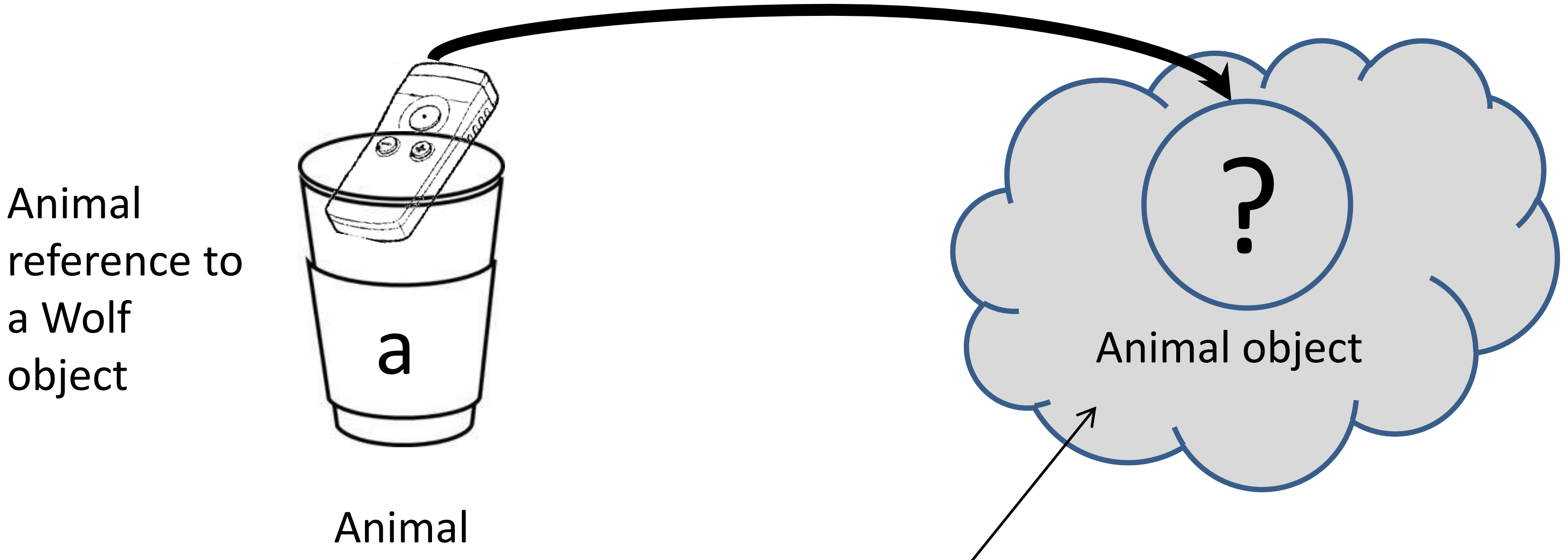
And we know we can say:

Animal a=new Wolf();


Animal reference to a Wolf object
a
Animal
Wolf object
These two are NOT the same type

But here's where it gets weird

Animal a=new Animal();



These two are the same type, but…
What does an Animal object look like?

What does an Animal object look like?

- What are the instance variable values?

What shape, what color, size, number of legs?

- We need Animal for inheritance and polymorphism, but we want to be able to make instances only of more concrete objects, not Animal objects

Solution: ***abstract*** classes

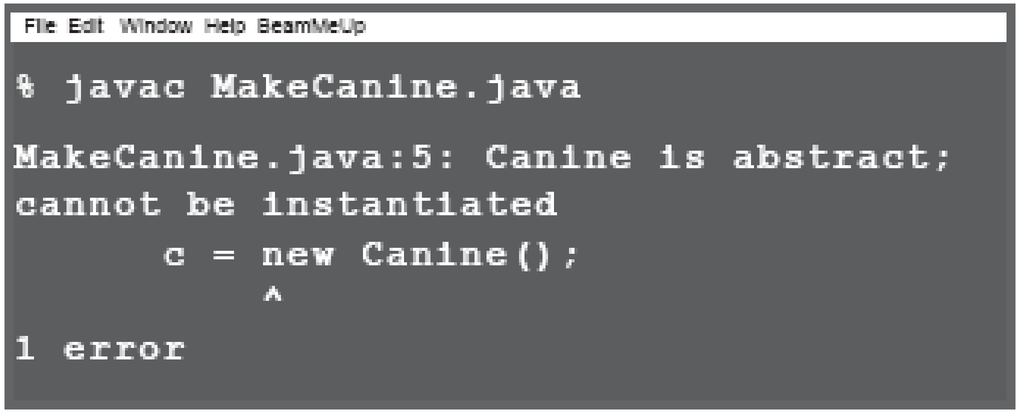
Abstract class declaration

```
abstract class Canine extends Animal{
        public void roam() {…}
}
```

You cannot create a new instance of an abstract class:

```
Canine c;
c=new Dog();
c=new Canine();
```

X



```
% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
        c = new Canine();
            ^
1 error
```

Abstract and concrete classes

- An abstract class has no use, no value, no purpose in life unless it is extended
- An abstract class means the class **must** be extended to be used
- A class that is not abstract (regular class) is called concrete
- A lot of abstract classes in Java GUI API: ***Component*** extended by ***Jbutton***, ***JTextArea*** etc.

Abstract methods

- We can mark **methods** as ***abstract***
- An abstract method means the method must be overriden by a subclass, to make a subclass concrete
- You make a method abstract if you can’t think of any generic implementation which could be useful for all subclasses

Abstract methods have no body

public abstract void eat();

- If you declare an abstract method, you **must** mark the class abstract as well
- You can mix abstract and non-abstract methods in an abstract class

It really sucks to be an abstract method. You don't have a body.

Purpose of abstract methods

- Not for reusing code (there is no code)
- To define a protocol common to all subclasses – to be used for polymorphism:

An abstract method says: all subclasses of this class have this method

You **must** implement all abstract methods

- The first concrete class in the inheritance tree must implement all abstract methods
- You must create a non-abstract method in your class with the same signature as an abstract method. It can even be empty.

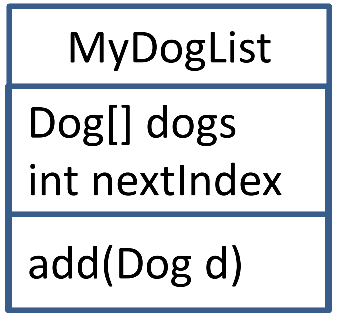
Abstract vs. Concrete

Concrete	Sample class	Abstract
Golf course simulation	Tree	Tree nursery application
	House	Architect application
	Book	
	Oven	
	Game unit	

Polymorphism in action

- List of Dogs

```
public class MyDogList {
        private Dog [] dogs = new Dog[5];
        private int nextIndex = 0;
        public void add(Dog d) {
                if (nextIndex < dogs.length) {
                        dogs[nextIndex] = d;
                        System.out.println("Dog added at " +
                                                        nextIndex);
                        nextIndex++;
                }
        }
}
```

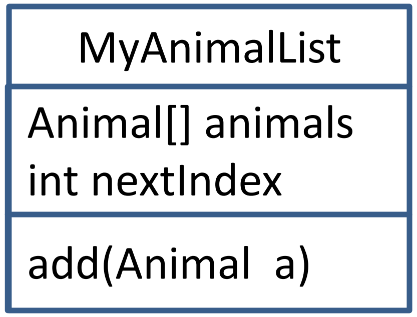


Polymorphism in action

- Now we need to keep Cats too

```
public class MyAnimalList {
        private Animal[] animals = new Animal[5];
        private int nextIndex = 0;
        public void add(Animal a) {
                if (nextIndex < animals.length) {
                        animals[nextIndex] = d;
                        System.out.println(“Animal added at “ +  nextIndex);
                        nextIndex++;
                }
        }
}

public class AnimalTestDrive{
        public static void main (String[] args) {
                MyAnimalList list = new MyAnimalList();
                Dog a = new Dog();
                Cat c = new Cat();
                list.add(a);
                list.add(c);
        }
}
```



Generic list

- What about non-animals? Why not to make list generic to take anything?
- We want to change the type of the array and the parameters of add() to something **above** Animal, something more abstract than Animal
- But we don't have a superclass for Animal

- Then again, maybe we do…

Object class

- Every class in Java extends class Object
- Class Object is the mother of all classes; it’s the superclass of *everything*
- Without a common superclass there is no way for the developers of Java to make useful libraries with methods which can take you custom types … types they never knew about when they wrote the library
- Implicitly:

public class Animal extends Object

So what’s in this ultra-super-megaclass **Object**?

Methods of class Object:

- boolean equals(Object b)
- Class getClass()
- int hashCode()
- String toString()
- …

Is class Object abstract?

- Object is **non-abstract class** because it has method implementations that all other classes can use out-of-the-box, without having to override them
- However, you can and must override such methods as ***equals()***, ***toString()*** and ***hashCode()*** in order to make your classes behave in a desirable manner
- Some of the methods (***getClass()***) cannot be overriden, they are marked as **final**
- You can create an instance of class Object, but this is very rare, and used mostly for thread synchronization

Why not to make all arguments and return types of class Object?

- This defeats the type-safety provided by Java
- The only methods you allowed to call on instances of class Object are the ones declared in class Object

```
Object o=new Ferrari();
o.goFast() ;
```

X

Method validity is based **on the reference type**, not the object type

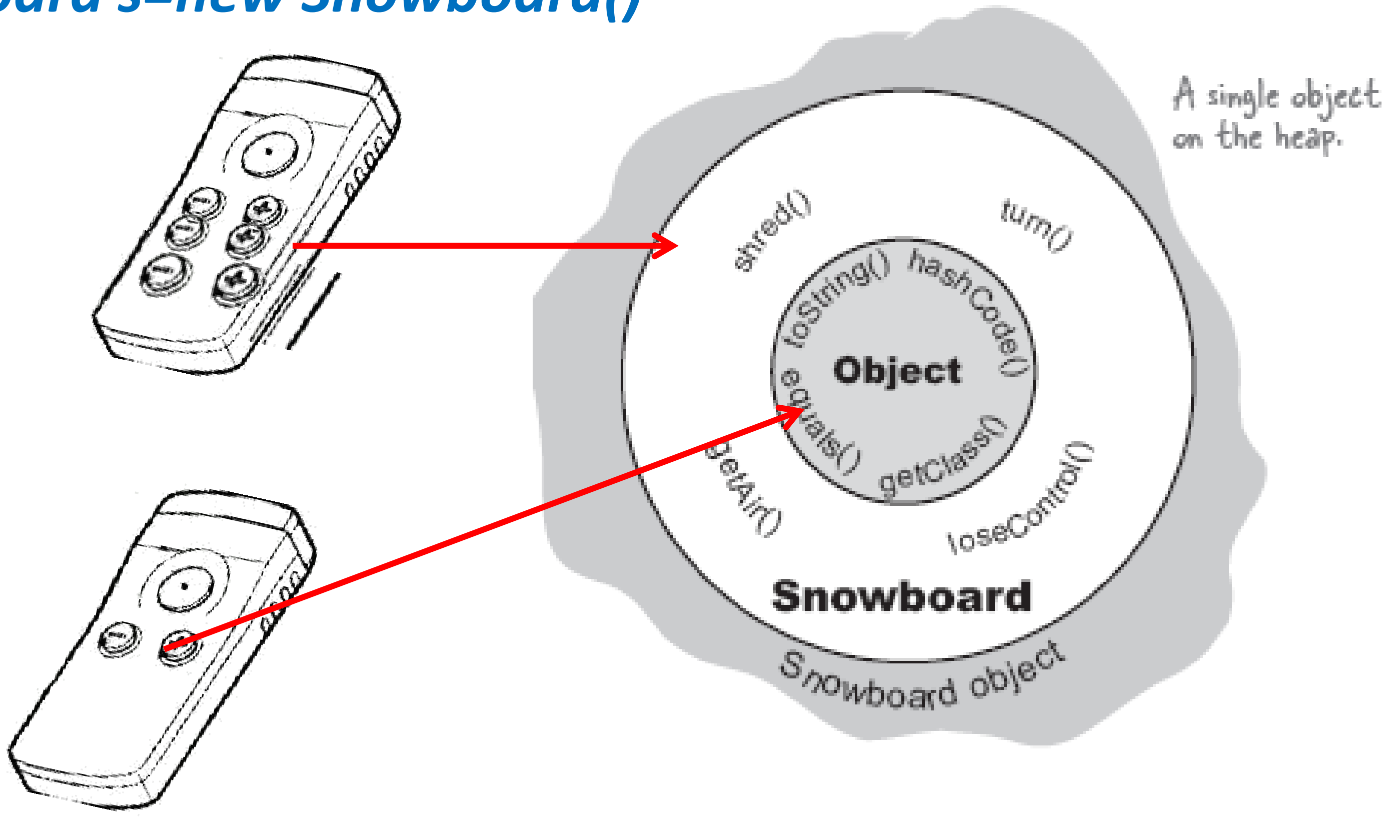
```
Object o=new Dog();
int i=o.hashCode();
o.bark() ; X
```

- You can cast it back to the Dog type in order to invoke the methods of class ***Dog***

```
((Dog)o).bark();
```

Each object contains all its superclasses

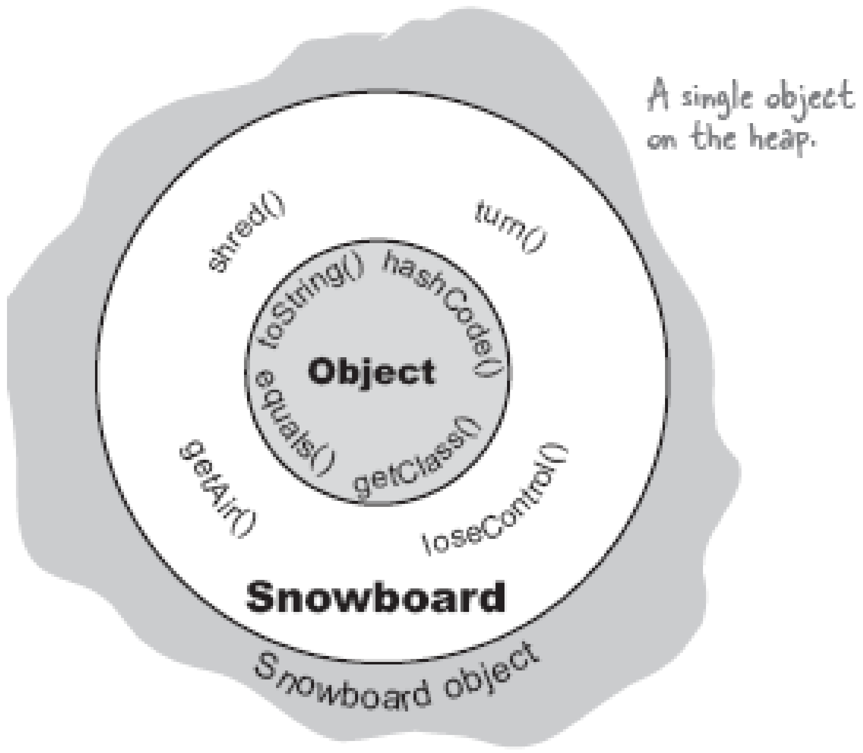
Snowboard s=new Snowboard()



Object o=s;

Polymorphism means *many forms*

- You can treat a Snowboard as a Snowboard or as an Object



Casting a reference back to its real type

```
If (o instanceof Snowboard)
{
    Snowboard s=(Snowboard) o;
}
```

Contract

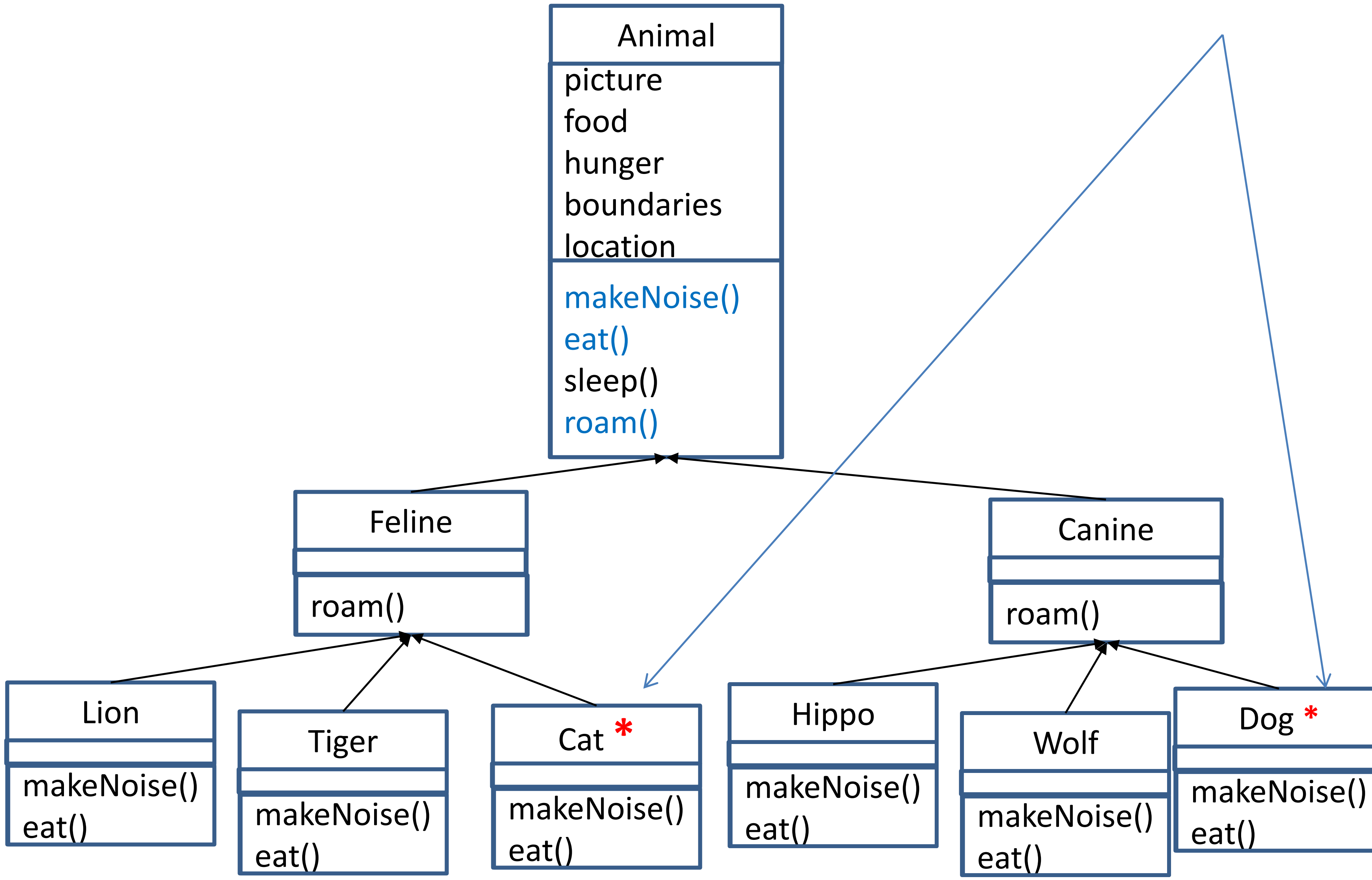
- Each class exposes public methods including methods of its superclass as a **contract**:

- The Dog class defines a contract:
  - Everything in class Canine is part of this contract
  - Everything in class Animal is part of this contract
  - Everything in class Object is part of this contract

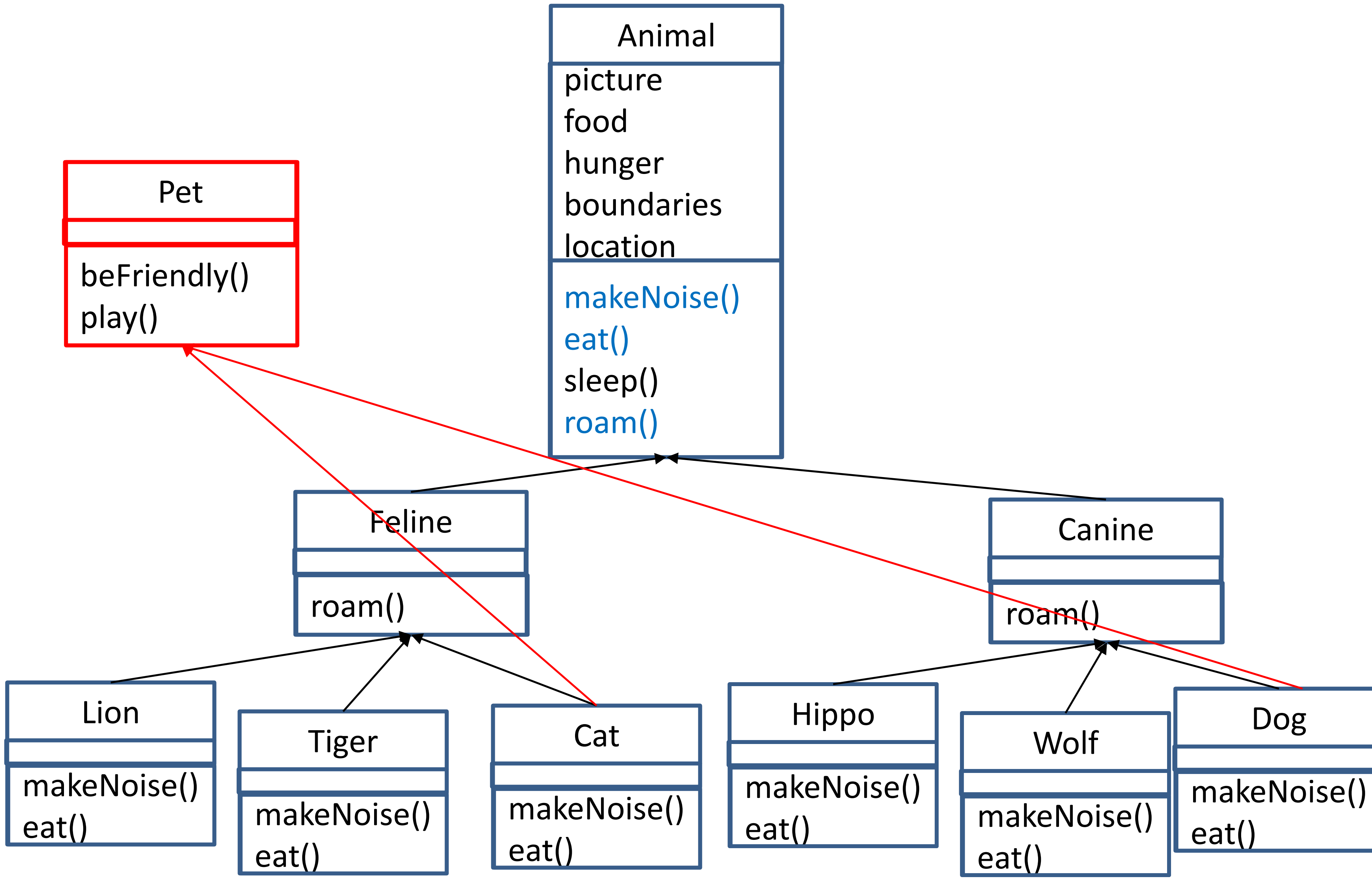
What if we need to change the contract?

- We want to use the same animal classes for the PetShop program
- How to add Pet behavior to some of the Animal classes?
- Should we add some new methods to a Dog class?
- Maybe the same methods to a Cat class? (duplicate code)

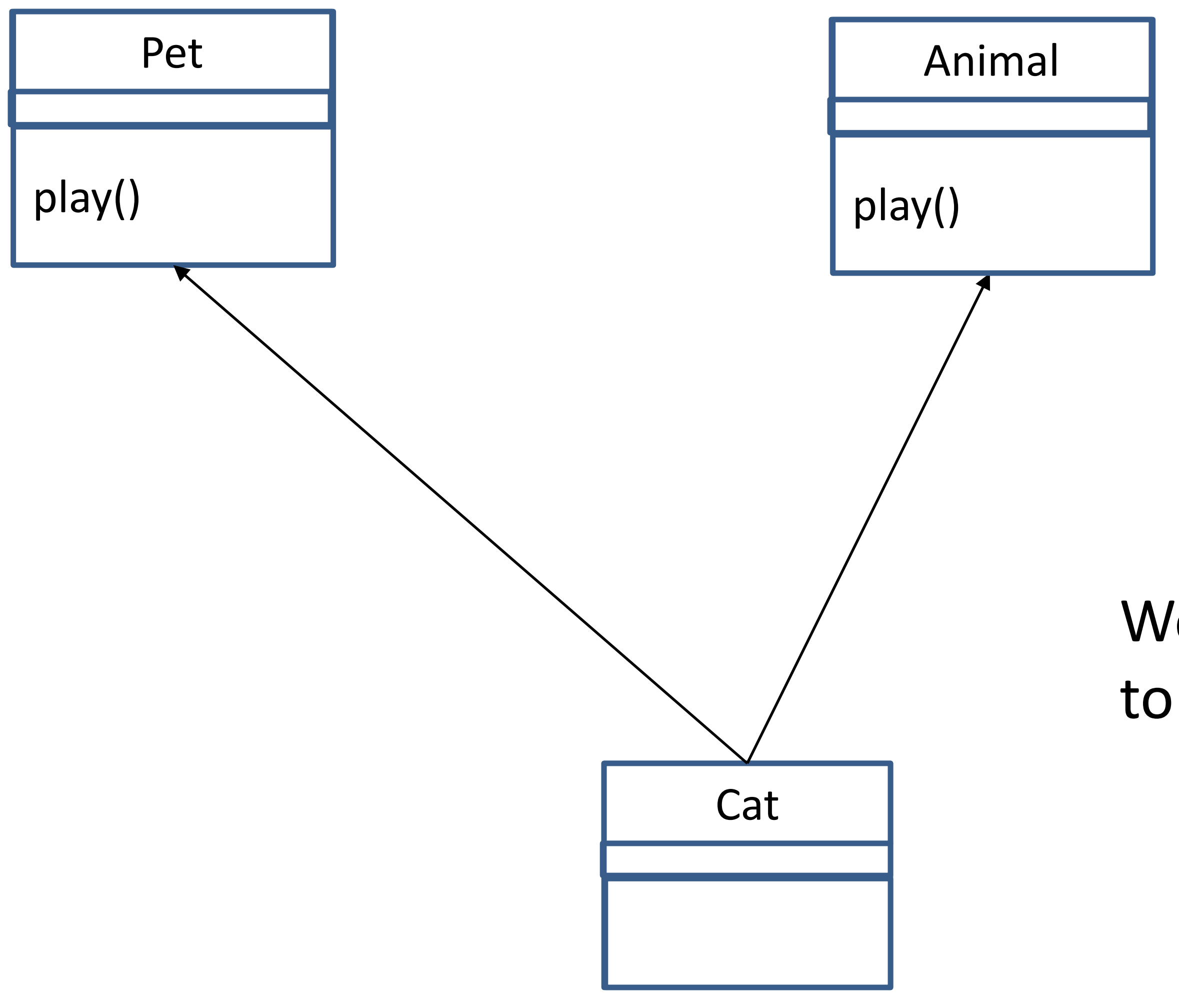
Add pet methods only to classes which can be pets



It looks that we need two superclasses at the top



Multiple inheritance can be a really bad thing



We need extra-rules to deal with collisions

Java solution: **Interface**

- Gives polymorphic benefits of multiple inheritance
- In the interface class **all the methods are abstract**
- Each class which implements an interface must to implement all the methods declared in the interface

public interface Pet {…}

public class Dog extends Canine implements Pet {…}

Importance of interfaces

- If you use interface as a type of arguments, you can pass any class which implements this interface
- A class does not have to come from one inheritance tree: another class can implement the same interface and come from a completely different inheritance tree
- We can treat an object by the role it plays rather than by the class type from which it was instantiated
- A class can implement multiple interfaces: play different roles

```
public class Dog extends Animal implements Pet,
                                     Saveable, Paintable { ... }
```

When to use interfaces

- Make a normal class that does not extend anything when your new class does not pass the IS-A test for any other type
- Make a subclass (extend the class) only when you need to make a more specific version of a superclass
- Use an abstract class when you want to define a template for a group of subclasses, and you have at least some code that all subclasses can use
- Use an interface when you want to define a role that other classes can play regardless of where the classes are in the inheritance tree

Partial method overriding

- If you want to use the code in a superclass's method, but extend it:

```
abstract class Report {
        void runReport() {
                // set-up report
        }
        void printReport() {
                // generic printing
        }
}
class BuzzwordsReport extends Report {
        void runReport() {
                super.runReport();
                buzzwordCompliance();
                printReport();
        }
        void buzzwordCompliance() {...}
}
```

Bullet points I

- When you don’t want a class to be instantiated – mark the class with **abstract** keyword
- An abstract class may have **both** abstract and non-abstract methods
- An abstract method has no body, only the declaration

Bullet points II

- Every class in Java is either a direct or indirect **subclass of class Object**
- Methods can be declared with ***Object*** arguments and return types
- You can call only the methods which are in the class used by **reference variable**

Bullet points III

- Multiple inheritance is not allowed in Java
- An **interface** is a 100% pure abstract class
- Your class can implement **multiple interfaces**
- A class that implements interface must implement all interface methods, since all interface methods are implicitly public and abstract