Inheritance

Lecture 7

Reading: chapters 9,10

Similarities and Differences

CSCI331Mobile, Van, Convertible

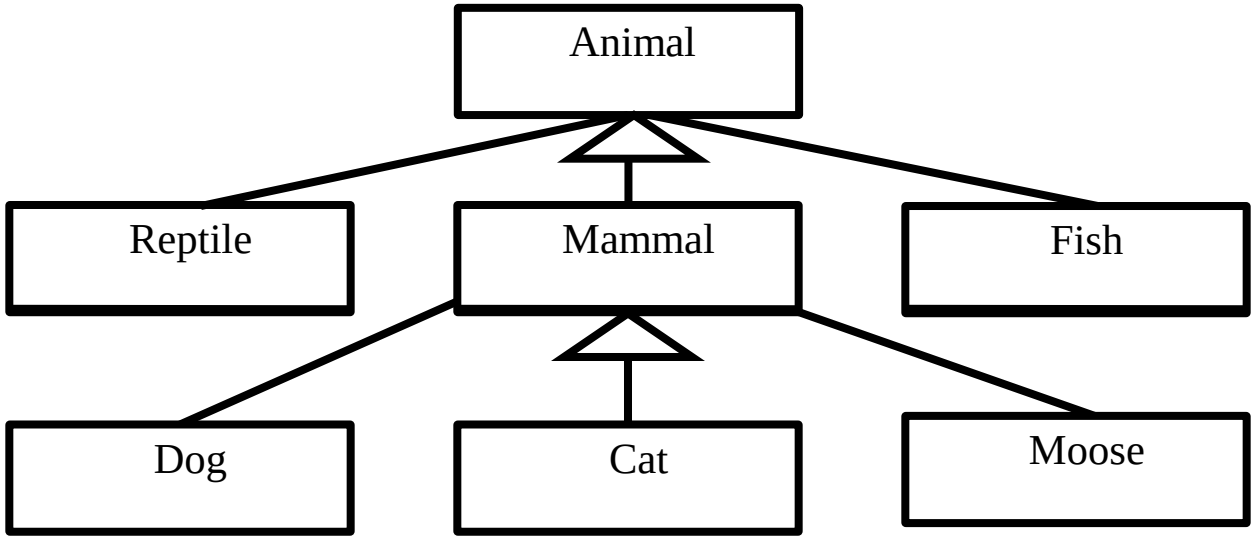
- What do these three automobiles have in common?
  - they're all vehicles!
    - all can move
    - all have an engine
    - all have doors
    - all have one driver
    - all hold a number of passengers

- What about these three vehicles is different?
  - **the sportscar:**
    - convertible top, 2 doors, moves really fast, holds small number of people
  - **the van:**
    - high top, 4 doors (two of which slide open), moves at moderate speed, holds large number of people
  - **the CSCI331Mobile:**
    - normal top, 2 doors, moves slowly, holds moderate number of people

Inheritance

- Inheritance models "*is a*" relationships
  - object "is an" other object if it can behave in the same way
  - inheritance uses *similarities and differences* to model groups of related objects

- Where there's inheritance, there's an *Inheritance Hierarchy* of classes



  - **`Mammal`** "*is an*" **`Animal`**
  - **`Cat`** "*is a*" **`Mammal`**
  - Transitive relationship: a **`Cat`** "*is an*" **`Animal`** too

- We can say:
  - **`Reptile, Mammal`** and **`Fish`** "*inherit from*" **`Animal`**
  - **`Dog, Cat,`** and **`Moose`** "*inherit from*" **`Mammal`**

Inheriting Capabilities and Properties

- Subclass *inherits* all **`public`** capabilities of its superclass
  - if **`Animal`**s eat and sleep, then **`Reptile`**s, **`Mammal`**s, and **`Fish`** eat and sleep
  - if **`Vehicle`**s move, then **`SportsCar`**s move!

- Subclass *specializes* its superclass
  - by *adding* new methods, *overriding* existing methods*, and defining* "abstract" methods declared by parent that have no code in them
  - we'll see these in a few slides!

- Superclass *factors out* capabilities common among its subclasses
  - subclasses are defined by their *differences* from their superclass

- As a general pattern, subclasses:
  - inherit **`public`** capabilities (methods)
  - inherit **`private`** properties (instance variables)
    - do not have **direct** access to them, only **indirect** access via inherited superclass methods that make use of them (including accessors/mutators)

→

Superclasses and Subclasses

- Inheritance is a way of:
  - organizing information
  - grouping similar classes
  - modeling similarities among classes
  - creating a taxonomy of objects

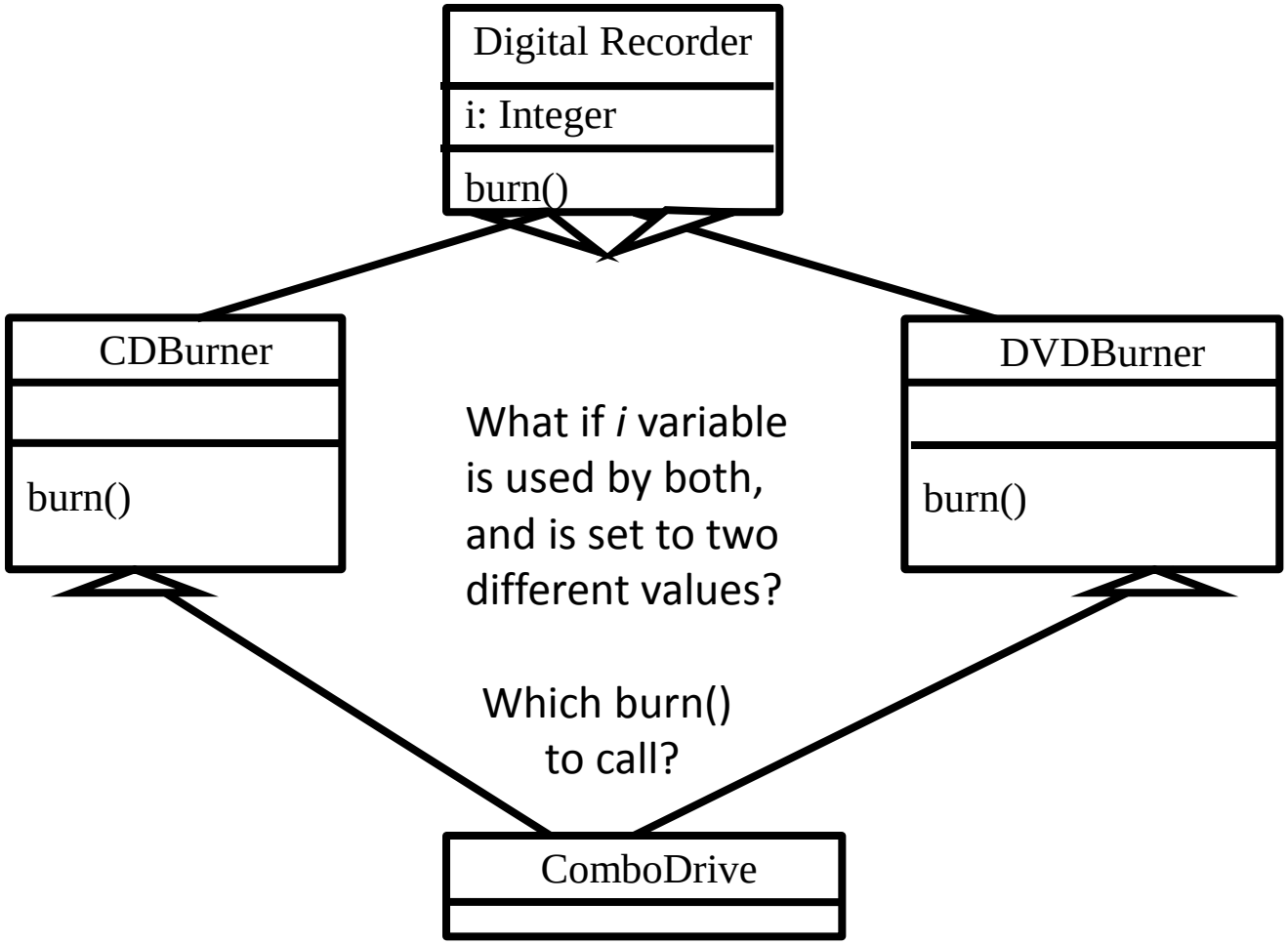
- **`Animal`** is called *superclass*
  - or *base class* or *parent class*
  - in our car example, **`Vehicle`** is called *superclass*

- **`Fish`** is called *subclass*
  - or *derived class* or *child class*
  - in our car example, **`SportsCar`** is *subclass*

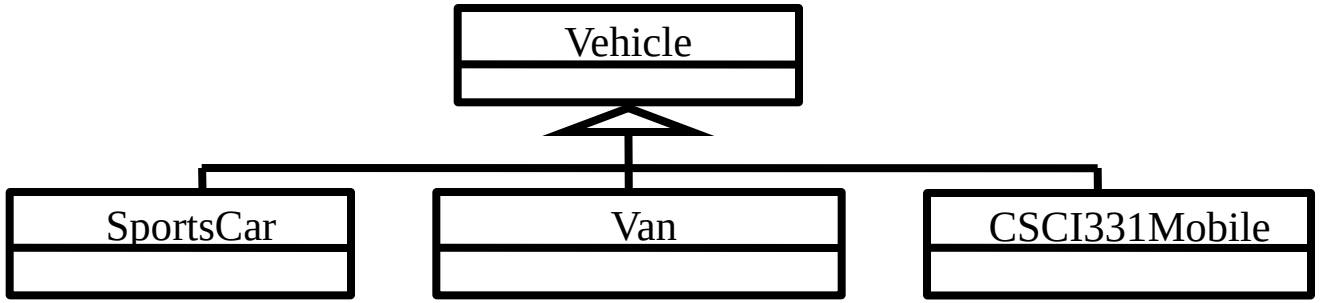
- Any class can be both at same time
  - e.g., **`Mammal`** is *superclass* of **`Moose`** and *subclass* of **`Animal`**

- Can inherit from only *one* superclass in Java
  - C++ allows a subclass to inherit from multiple superclasses, but this is prone to errors

Deadly Diamond of Death



Inheritance, Even with Vehicles!

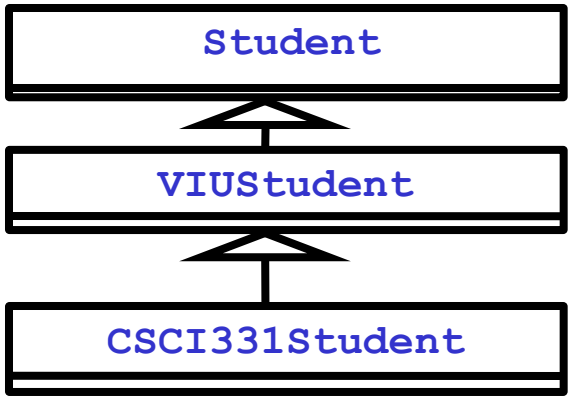


- a **`SportsCar`** "*is a*" **`Vehicle`**
- a **`CSCI331Mobile`** "*is a*" **`Vehicle`**
- you get the picture...

- We call this a *tree diagram*, with **`Vehicle`** as the "root" and **`SportsCar, CSCI331Mobile, Van`** as "leaves" (an upside-down tree)

Inheritance Example

- `Student` *inheritance hierarchy*:
  - `Student` is *base class*
  - `VIUStudent` is `Student`'s *subclass*
  - `CSCI331Student` is *subclass* of `VIUStudent`

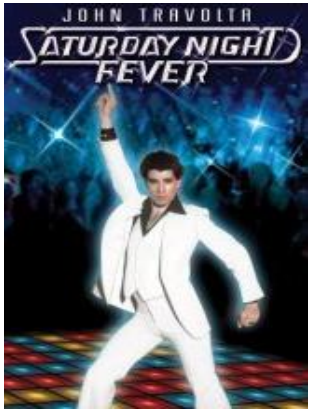


- `Student` has *a capability* (or *method*)
  - `study()` which works by:
    - going home, opening a book, and reading 50 pages.

Inheritance Example (cont.)

- `VIUStudent` "is a" `Student`, so it inherits the `study()` method
  - but it *overrides* the method by:
    - going to the library, reviewing lectures, and doing an assignment
  - note: it doesn't have to override this method!

- Finally, the `CSCI331Student` also knows how to `study()` (it `study()`s the same way a `VIUStudent` does)
  - however, the `CSCI331Student` subclass adds two capabilities: `gitDown()` and `gitFunky()`




```
public void gitDown()   {
   // Code to party
}
public void gitFunky()  {
   // Code to do awesome CSCI331 dance
}
```

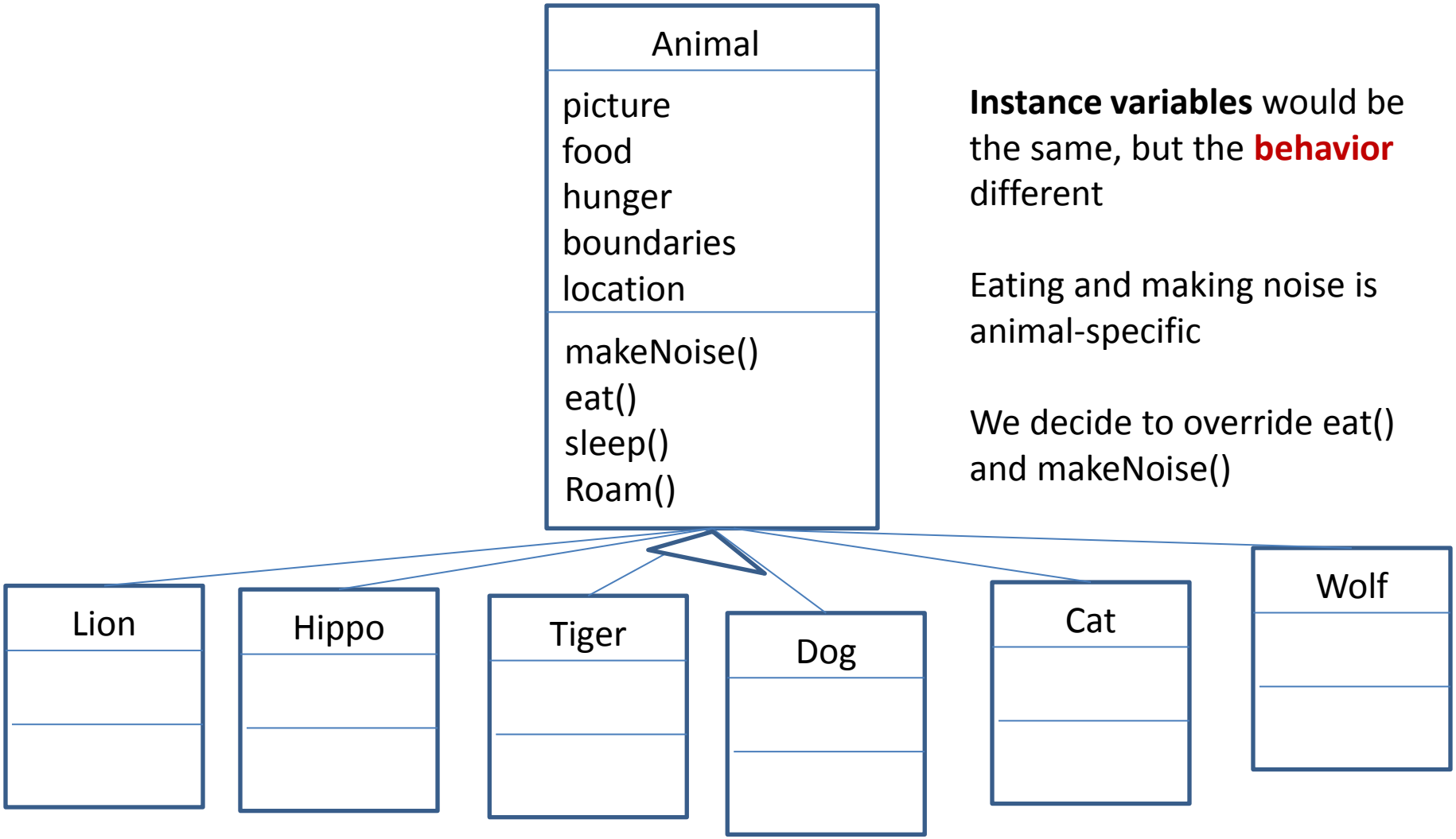
- Each subclass is a *specialization* of its superclass
  - `Student` knows how to `study()`, so all subclasses in hierarchy know how to `study()`
  - but the `VIUStudent` does not `study()` the same way a `Student` does
  - and the `CSCI331Student` has some capabilities that neither `Student` nor `VIUStudent` have (`gitDown()` and `gitFunky()`)

Abstract behaviour

- Superclass is too general to declare all behaviors, so each subclass adds its own behavior
- Superclass legislates an *abstract* behavior and therefore delegates implementation to its subclasses
- Superclass *specifies* behavior, subclasses *inherit* and *implement* behavior

Designing with Inheritance

Animal simulation program



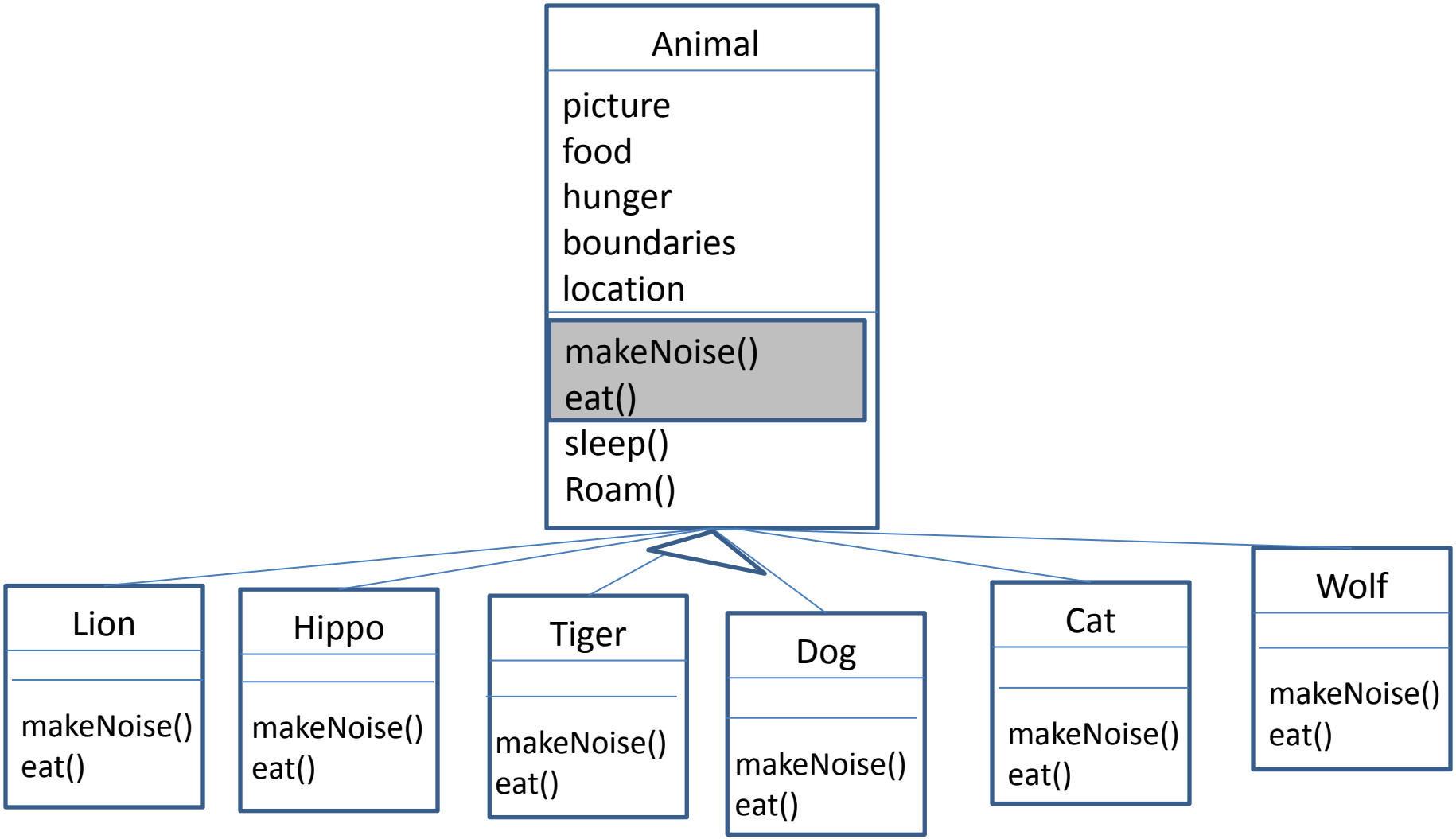
Instance variables would be the same, but the **behavior** different

Eating and making noise is animal-specific

We decide to override eat() and makeNoise()

Designing with Inheritance

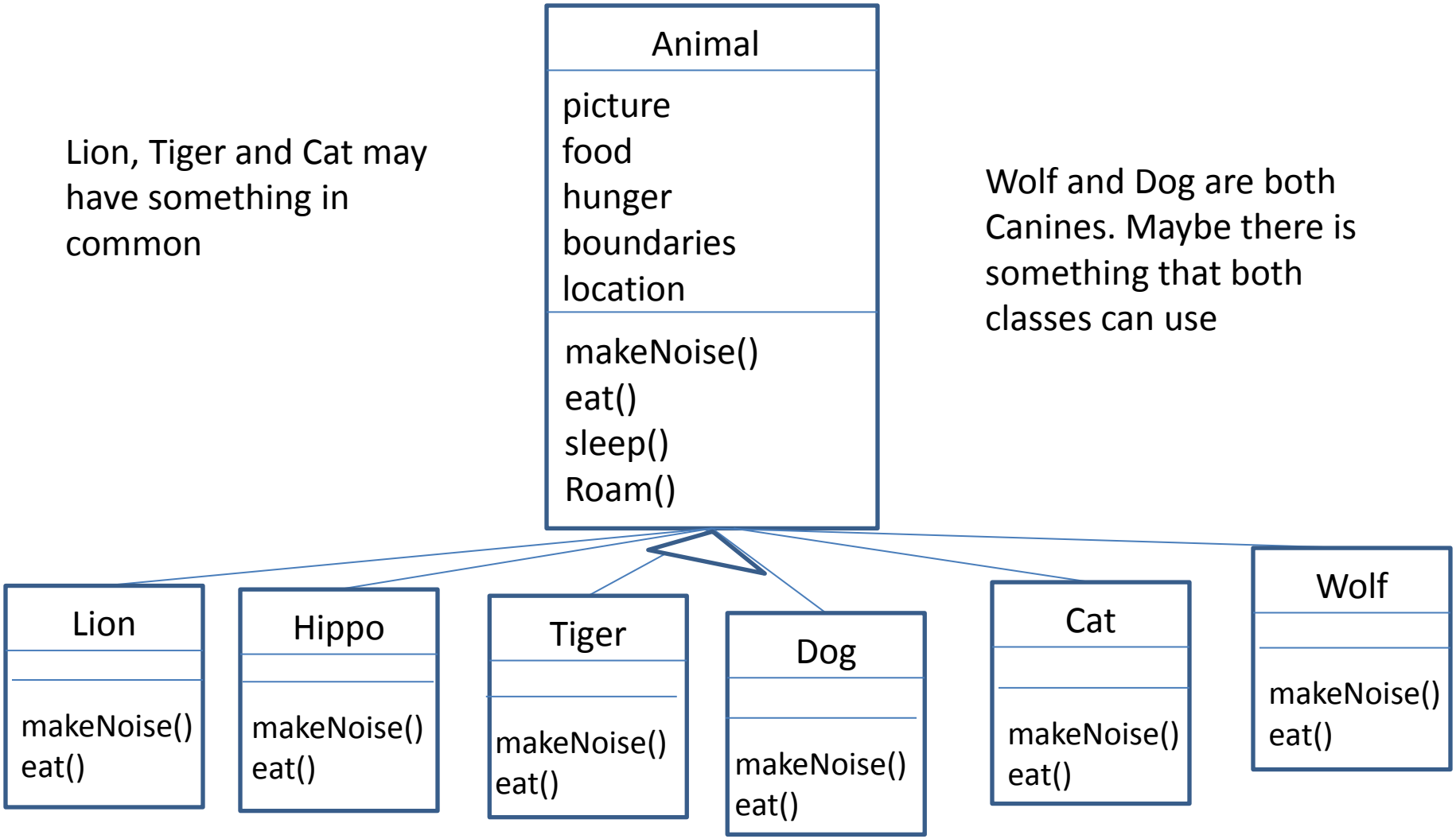
Animal simulation program



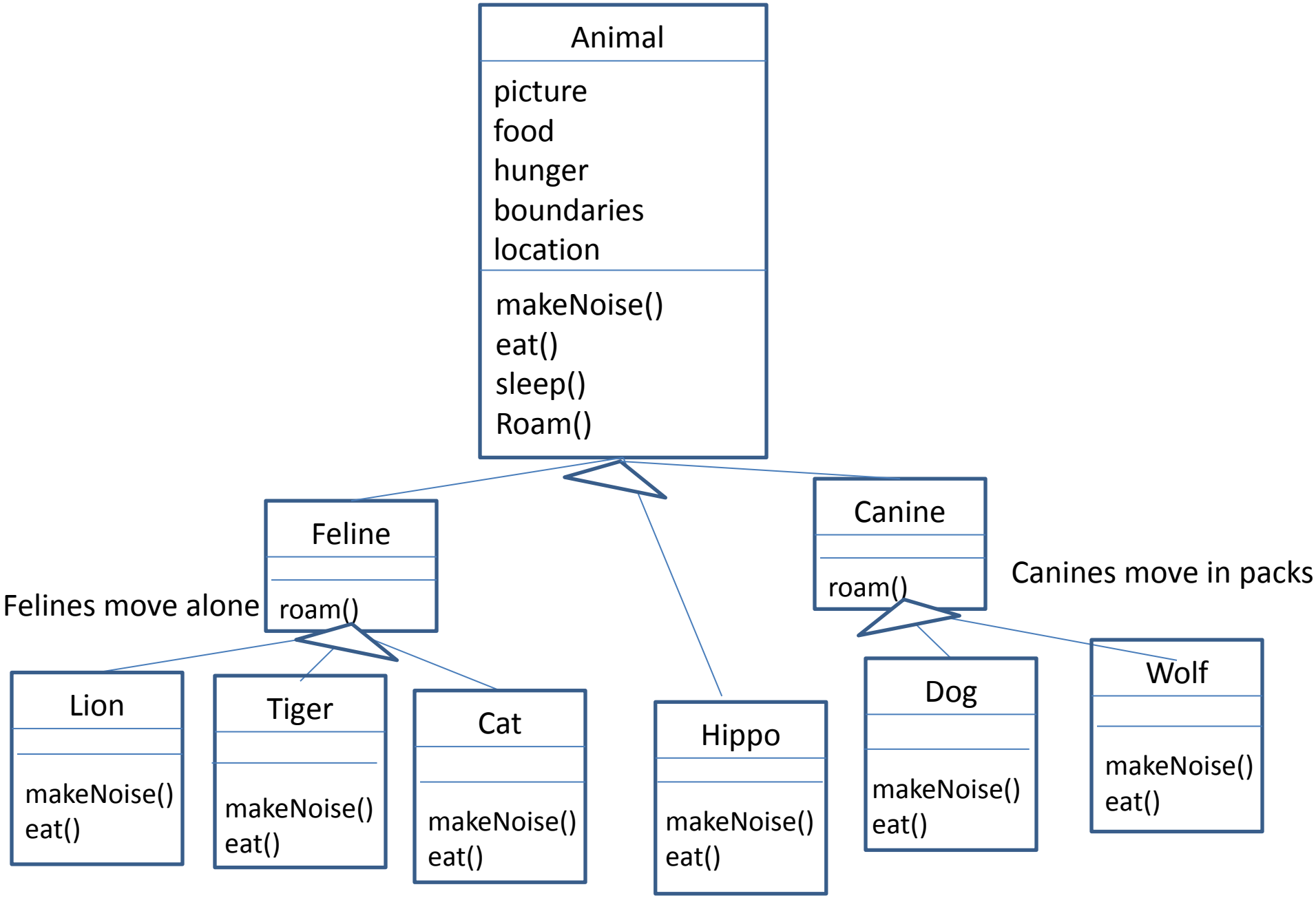
More inheritance opportunities

Lion, Tiger and Cat may have something in common

Wolf and Dog are both Canines. Maybe there is something that both classes can use



Final inheritance tree



Which method is called?

- Make a new Wolf object ***Wolf w=new Wolf();***
- Calls version in Wolf ***w.makeNoise();***
- Calls version in Canine ***w.roam();***
- Calls version in Wolf ***w.eat();***
- Calls version in Animal ***w.sleep();***

You are calling the most specific version of a method that exists for this class

Why use inheritance

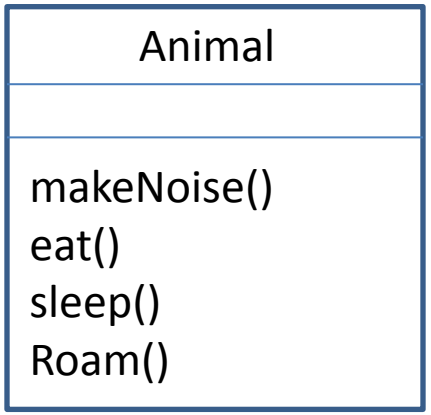
- Get rid of duplicate code by abstracting out the common behavior.
- Modify in one place, and the change is magically carried out to all subclasses
- Can add new subclasses easily, and they have some methods and properties right away

More important

1. Inheritance guarantees that all classes grouped under a certain supertype have all the methods that the superclass has:

We define *a common protocol* for a set of classes related through inheritance

Class *Animal* establishes a common protocol for all Animal subtypes



We are telling the world that any Animal can do this 4 things. That includes the method arguments and return types

2. When you define a supertype, any subclass can be substituted where the supertype is expected

This is called **Polymorphism**

Reference type and object type

Dog d=new Dog() ;

- Reference and object are of the same type

Animal a=new Dog();

- Reference and object are of the different type
- With polymorphism the reference type can be a superclass of the actual object type

Polymorphic arrays

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

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

And every object does the right thing

You can have polymorphic arguments and return types

```
class Vet {
public void giveShot(Animal a) {
                    // do horrible things to the Animal at
                    // the other end of the 'a' parameter
                    a.makeNoise();
          }
}

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

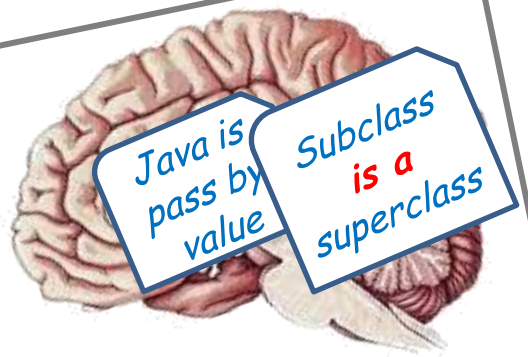
each animal makes a different noise

getShot() method can take any Animal you give it

Is-a vs. has-a

- When one class inherits from another, we say that subclasses *extend* the superclass.
- In order to test whether we need to use inheritance or composition, apply **is-a** test

- *Tub extends bathroom* sounds reasonable until you apply **is-a** test



Make it stick

Roses are red,
Violets are blue,
Square is a **Shape**
The reverse isn't true

True or False?

- Oven extends Kitchen
- Guitar extends Instrument
- Person extends Employee
- Ferrari extends Engine
- Hamster extends Pet
- Container extends Jar
- Metal extends Titanium
- Blonde extends Smart
- Beverage extends Martini

Hint: apply ***is a*** test

General guidelines for using inheritance

- DO use inheritance if one class is a more specific version of a superclass
- DO consider inheritance when you have behavior (code) that is shared among multiple classes of the same general type
- DO NOT use inheritance simply to reuse the code (for example, printing code for Alarm and for Piano) – create a Printer class that can be shared via composition by different objects
- DO NOT use inheritance if the subclass and superclass do not pass **is-a** test