Inheritance & Polymorphism

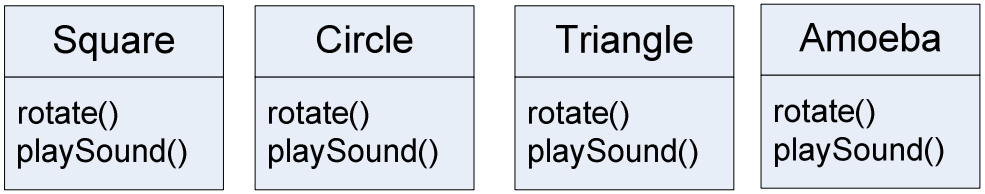
Object-Oriented Programming

Outline

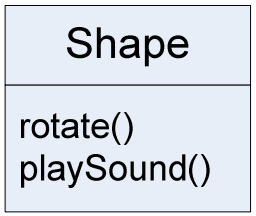
- Example
- Design an inheritance structure
- IS-A and HAS-A
- Polymorphism
- `protected` access level
- Rules for overriding
- the `Object` class

- Readings:
  - HFJ: Ch. 7.
  - GT: Ch. 7.

Inheritance – Example

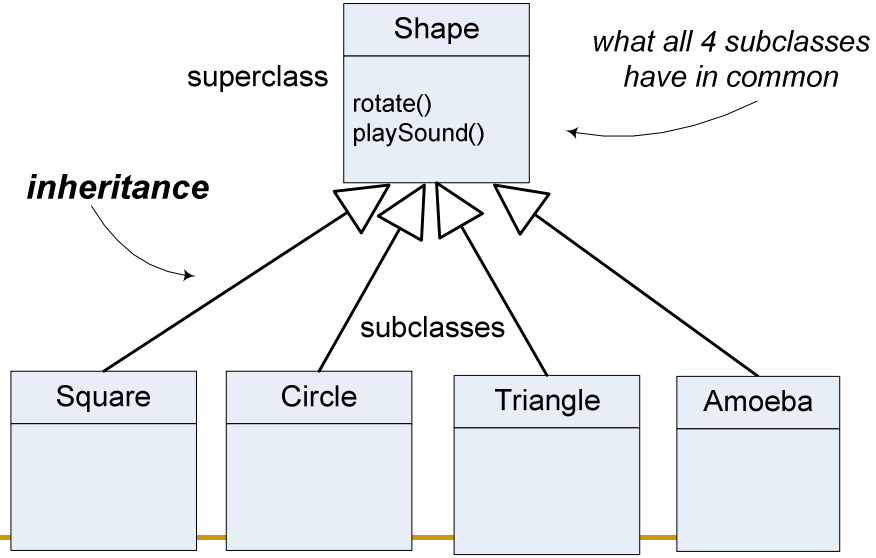


1. Look at what all four classes have in common.



2. They are Shapes, they all rotate and playSound, so we abstract out the common features and put them into a new class called Shape.

3. Then we link all four classes to the new Shape class in a relationship called inheritance.

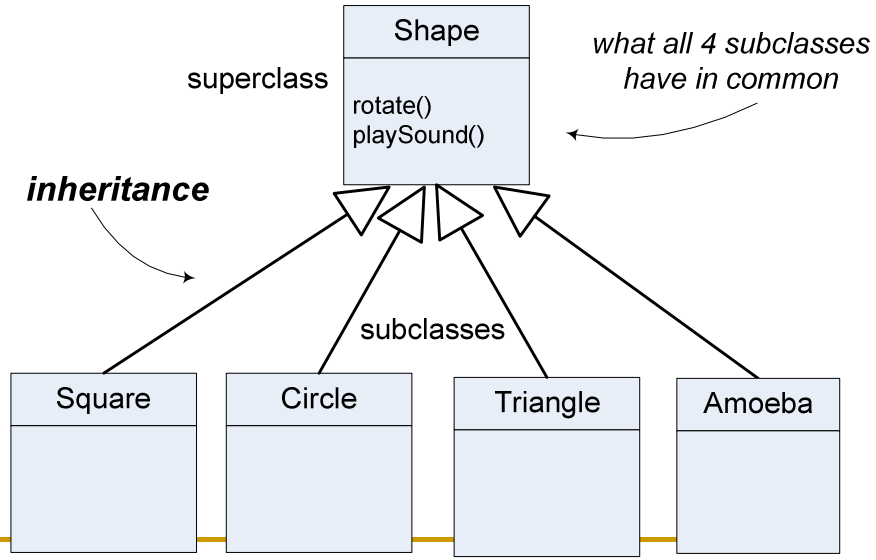


Inheritance – Example

We read this as…

- Square inherits from Shape. Circle inherits from Shape. …
- Shape is the superclass of Square, Circle, Triangle, Amoeba
- The other four are subclasses of Shape.

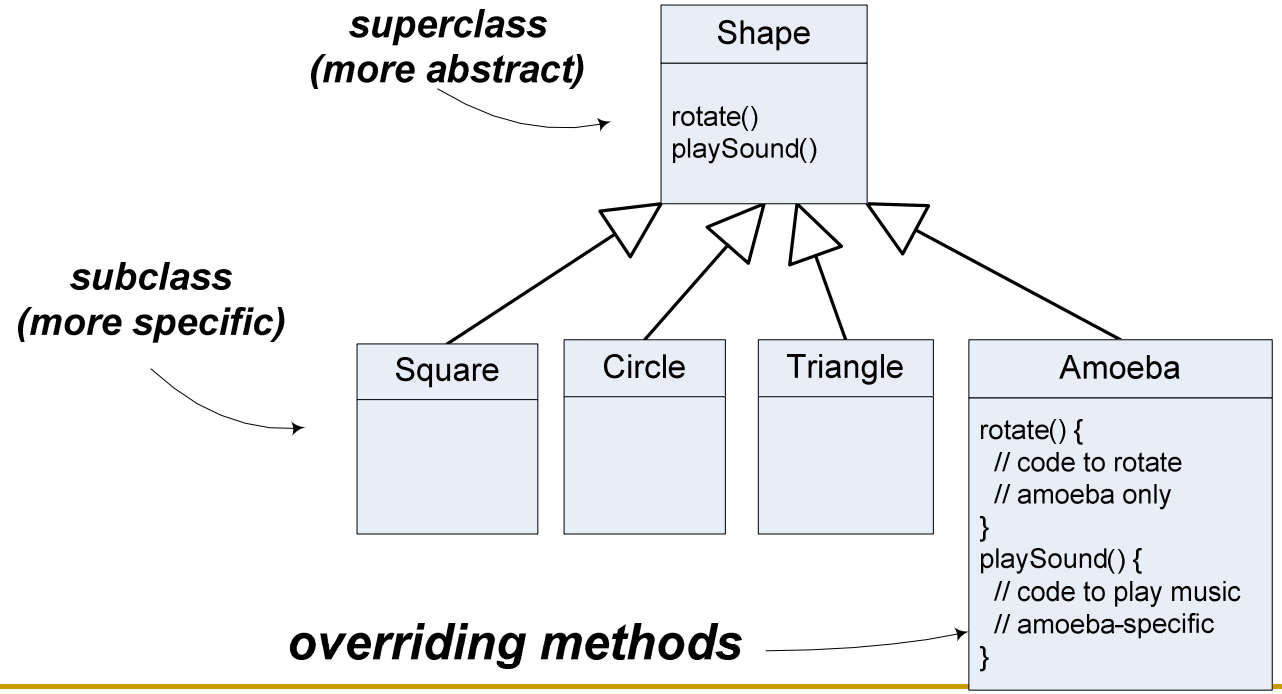
- if Shape has the functionality, then the subclasses automatically get the same functionality



Inheritance – Example

But… Amoeba *rotate* and *playSound* differently!

4. Let Amoeba **override** the inherited rotate() and playSound()



Food for thought

Tiger

HouseCat

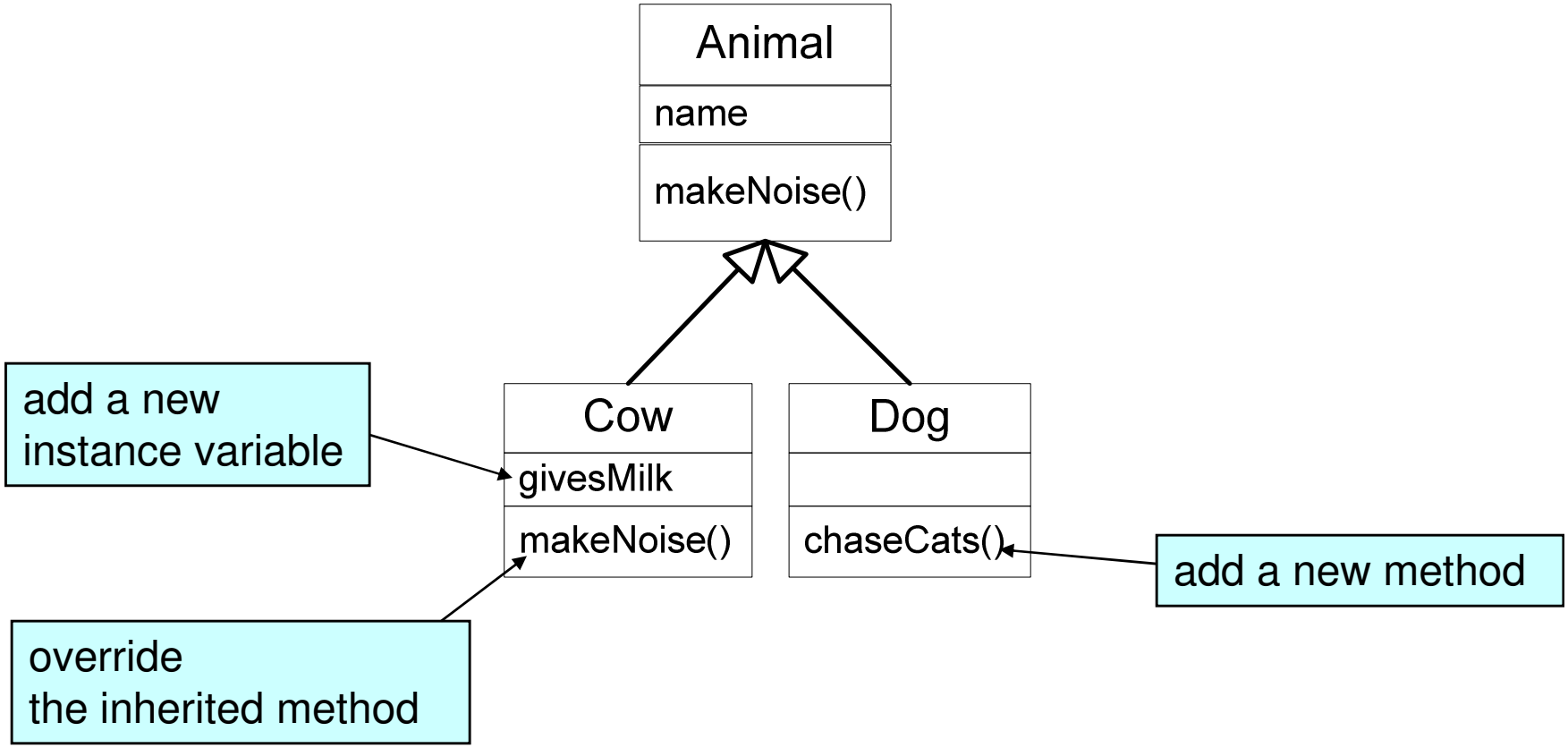
- Which one should be subclass/superclass?
- Or, should they both be subclasses to some *other* class?
- How should you design an inheritance structure?
- What should be overriden?

What is inheritance?

- The subclass **inherits** from the superclass,
  i.e, the subclass inherits members of the superclass:
  - instance variables and methods

- The subclass **specializes** the superclass:
  - it can add new variables and methods.
  - it can override inherited methods.

Example

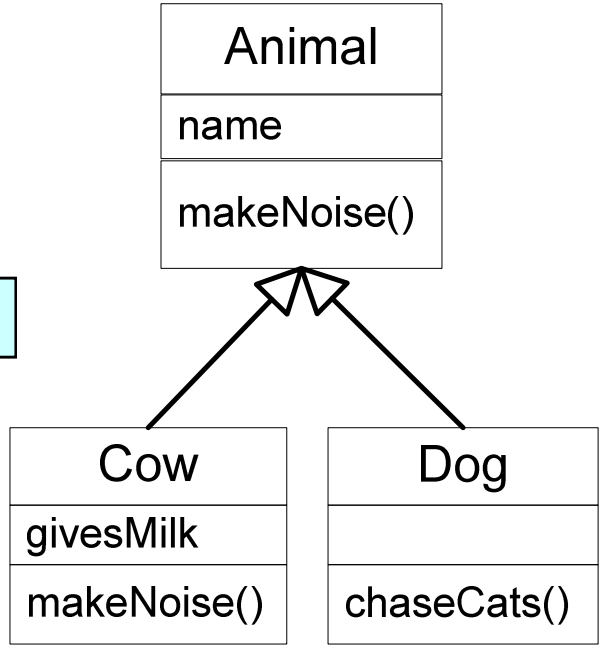

Animal
name
makeNoise()
Cow
givesMilk
makeNoise()
Dog
chaseCats()
add a new
instance variable
add a new method
override
the inherited method

```
class Animal {
  String name;
  void makeNoise() {
    System.out.print("Hmm");
  }
}

class Cow extends Animal {
  boolean givesMilk;
  void makeNoise() {
    System.out.print("Mooooooooo...");
  }
}

class Dog extends Animal {
  void chaseCats() {
    System.out.print("I'm coming, cat!");
  }
}
```

the overriden method

newly added members

the inherited method




```
Cow cow = new Cow();
cow.makeNoise();
cow.givesMilk = true;

Dog dog = new Dog();
dog.makeNoise();
dog.chaseCats();
```

Design an inheritance structure

- A program that simulates a number of animals of different species: tigers, lions, wolves, dogs, hippos, cats….
- We want other programmers to be able to add new kinds of animals to the program at any time.

Design an inheritance structure

Step 1:

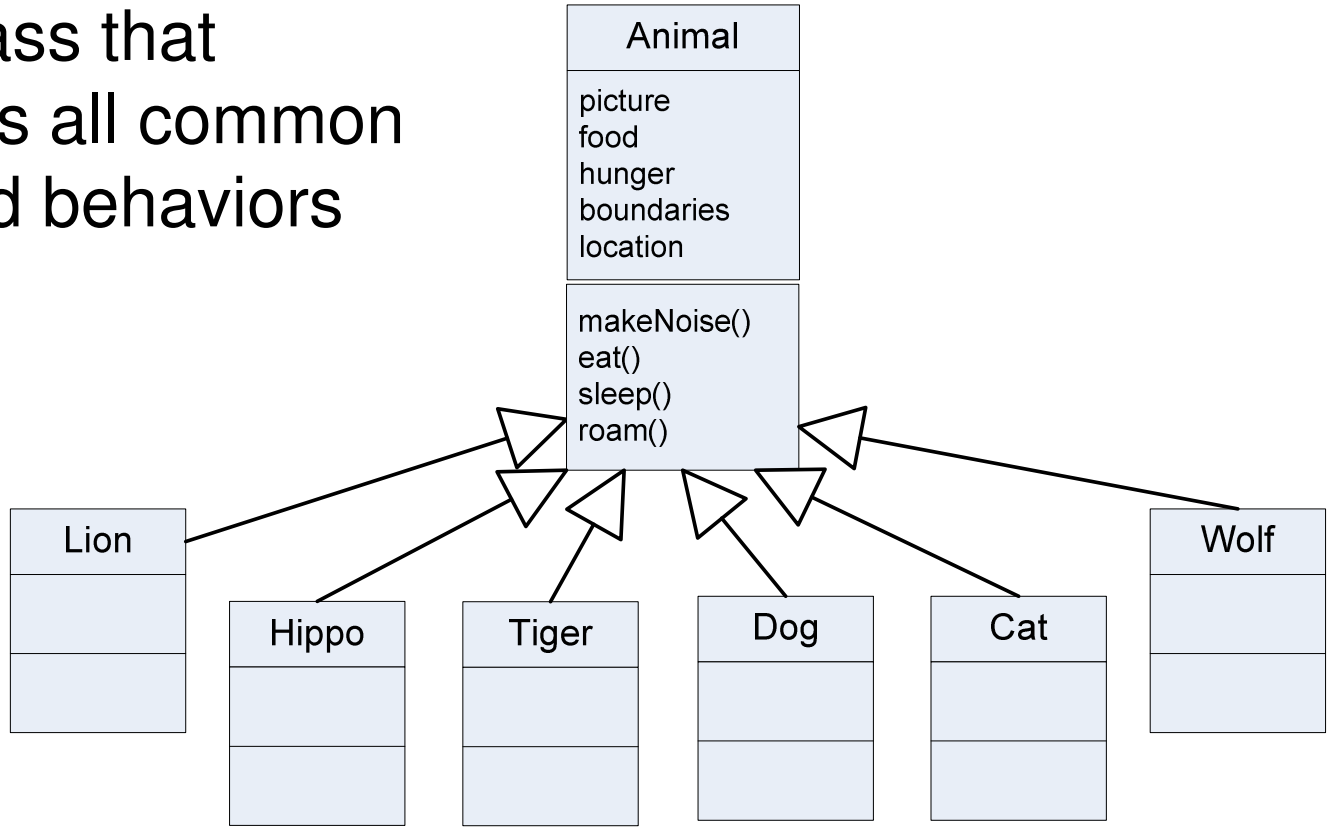
Figure out the common abstract characteristics that all animals have.

- instance variables
  - food
  - hunger
  - location
- methods
  - makeNoise()
  - eat()
  - sleep()
  - roam()

Design an inheritance structure

Step 2:

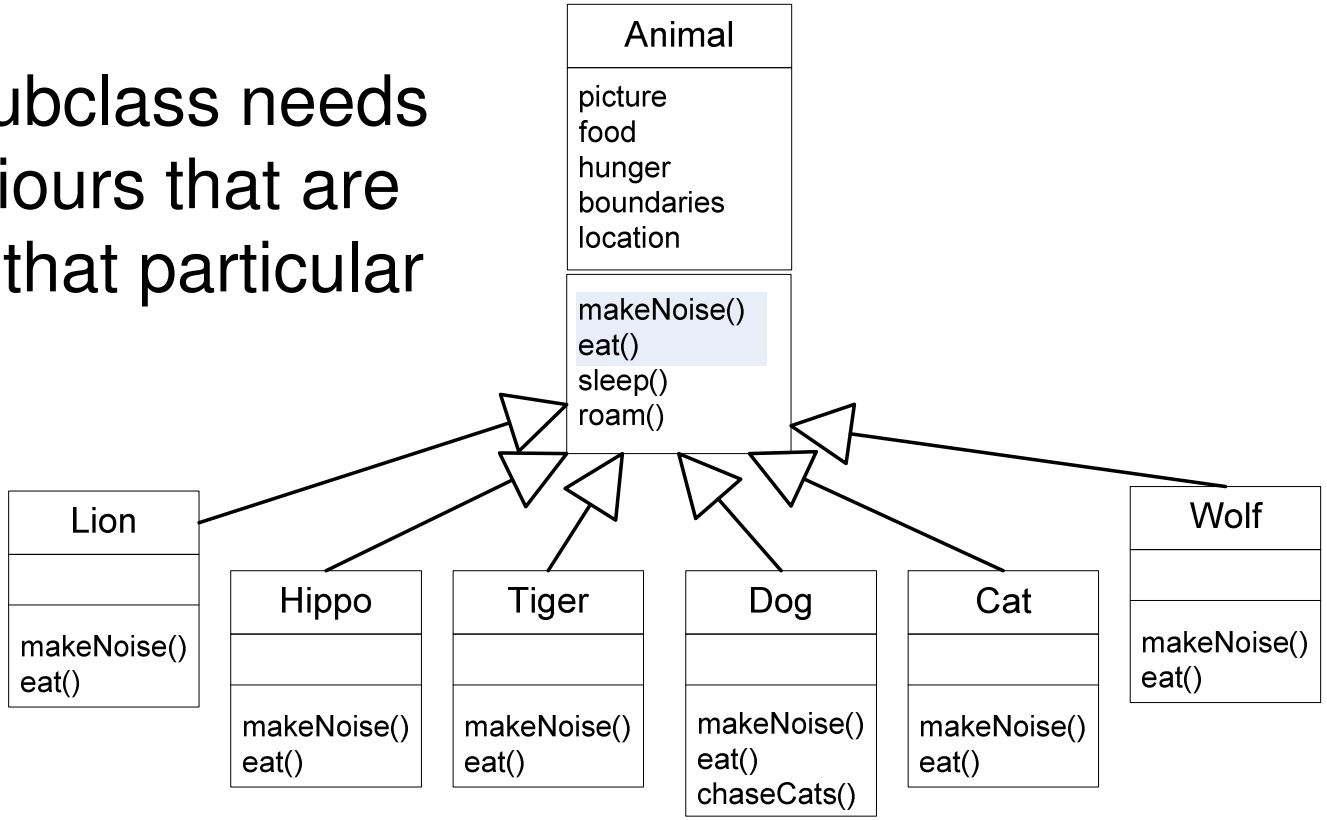
Design a class that represents all common states and behaviors



Design an inheritance structure

Step 3:

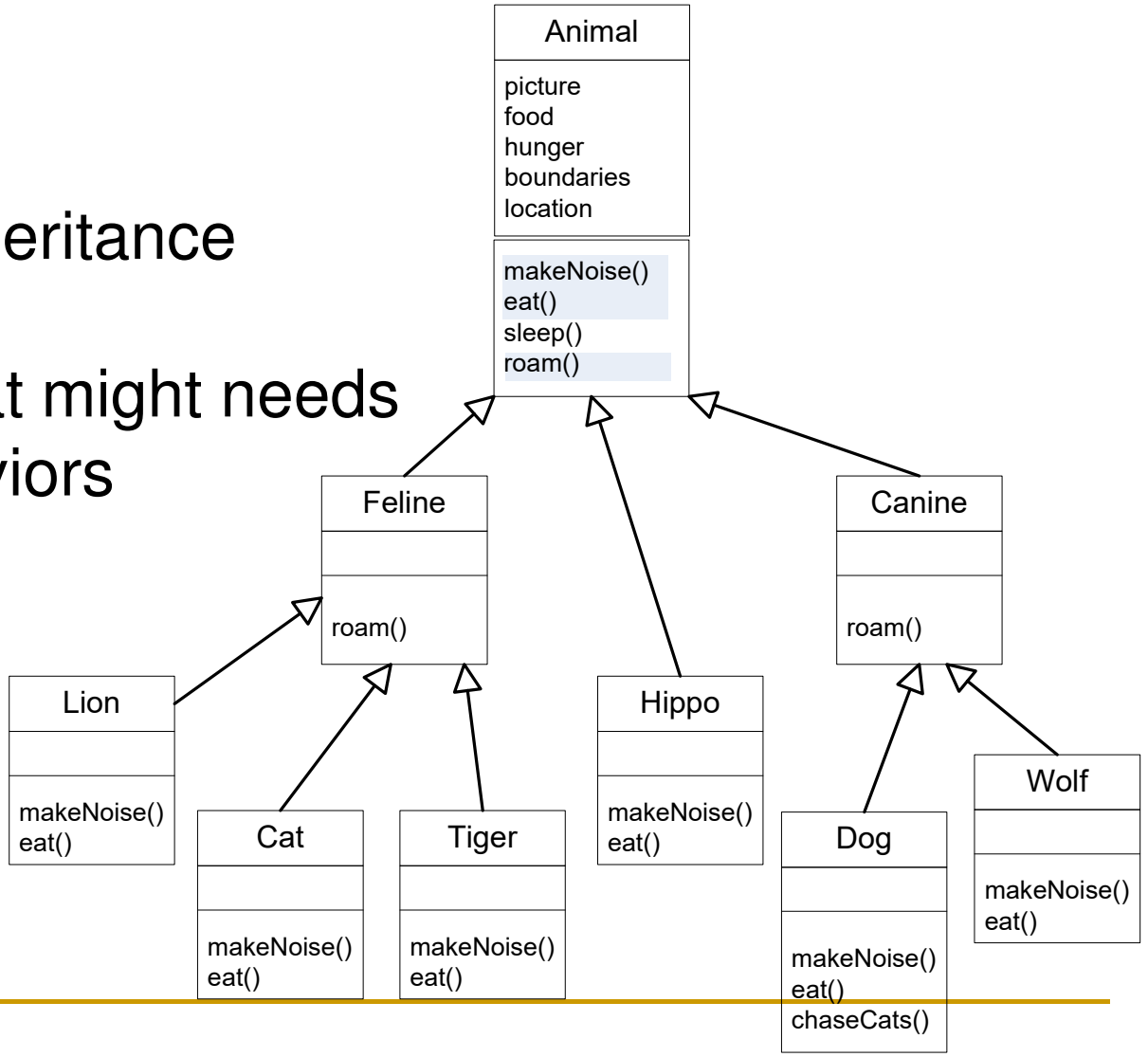
Decide if a subclass needs any behaviours that are specific to that particular subclass



Design an inheritance structure

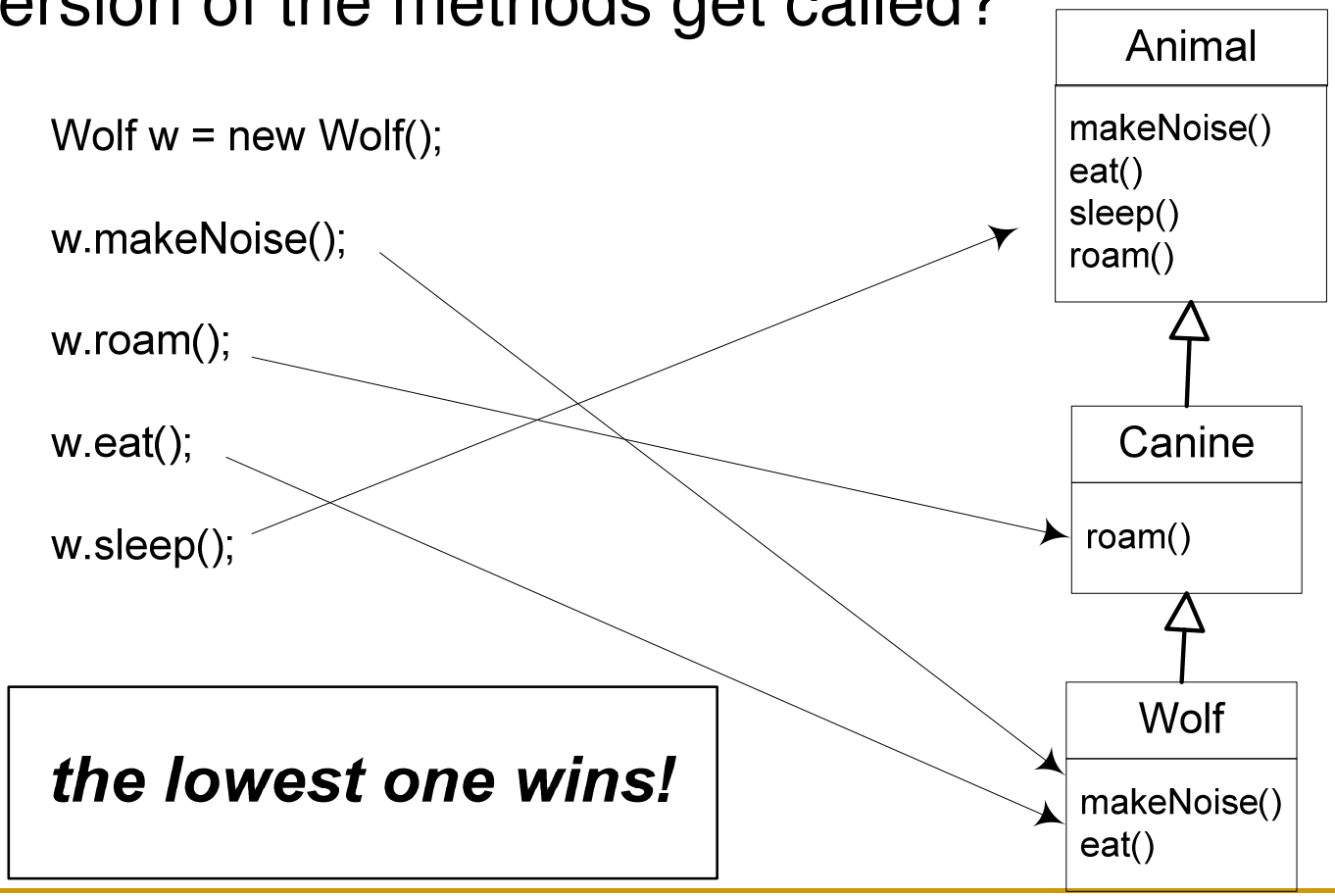
Step 4:

Look for more inheritance opportunities:
Subclasses that might needs common behaviors



Overriding - Which method is called?

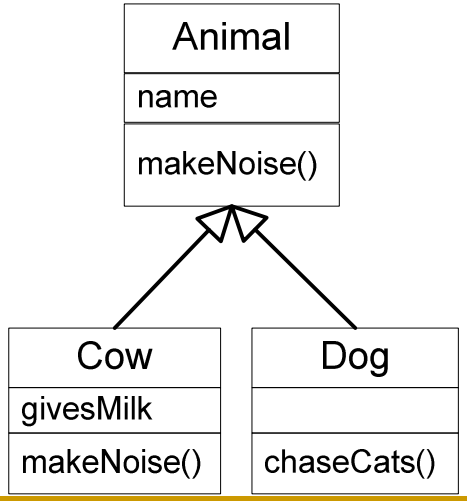
- Which version of the methods get called?



IS-A and HAS-A relationship

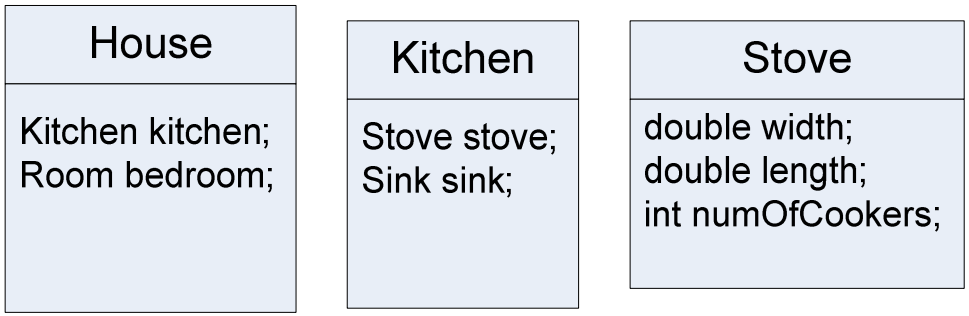
- Triangle IS-A Shape
- Cow IS-An Animal
- Dog IS-An Animal

➡ ***Inheritance***



- House HAS-A Kitchen
- Kitchen HAS-A Sink
- Kitchen HAS-A Stove

➡ ***Composition***



Code reuse

- Copy & paste
  - Manually ->Error-prone
- Composition – “HAS-A” relationship
  - the new class is composed of objects of existing classes.
  - reuse the functionality of the existing class, not its form
- Inheritance – “IS-A” relationship
  - create a new class as a *type of* an existing class
  - new class absorbs the existing class's members and extends them with new or modified capabilities

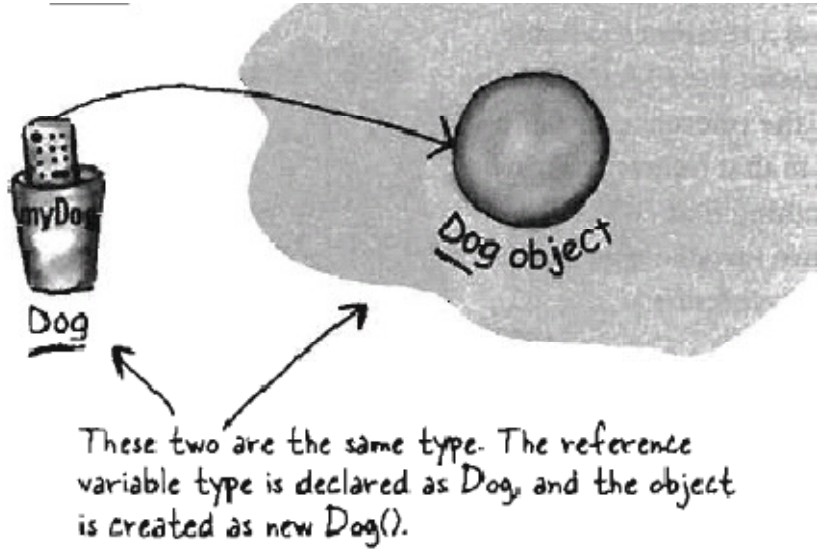
What does inheritance buy you?

1. You avoid duplicate code
   - Common features are put in one place
2. You define a common protocol for a group of classes
   - Objects of a subclass are guaranteed to have all features of the superclass.
   - Objects of a subclass can be treated as if they are objects of the superclass.
   - *Polymorphism*!

Polymorphism

- Normally,

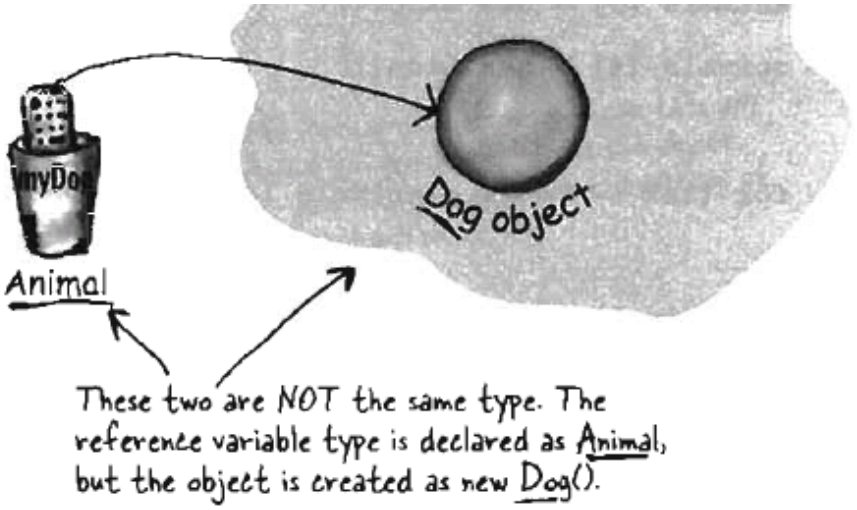
```
Dog dog = new Dog();
```



- With polymorphism:

```
Animal dog = new Dog();
```

The reference type can be a superclass of the actual object type.



Polymorphic arrays

- An array is declared of type Animal. It can hold objects of Animal's subclasses.

```
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].makeNoise();
}
```

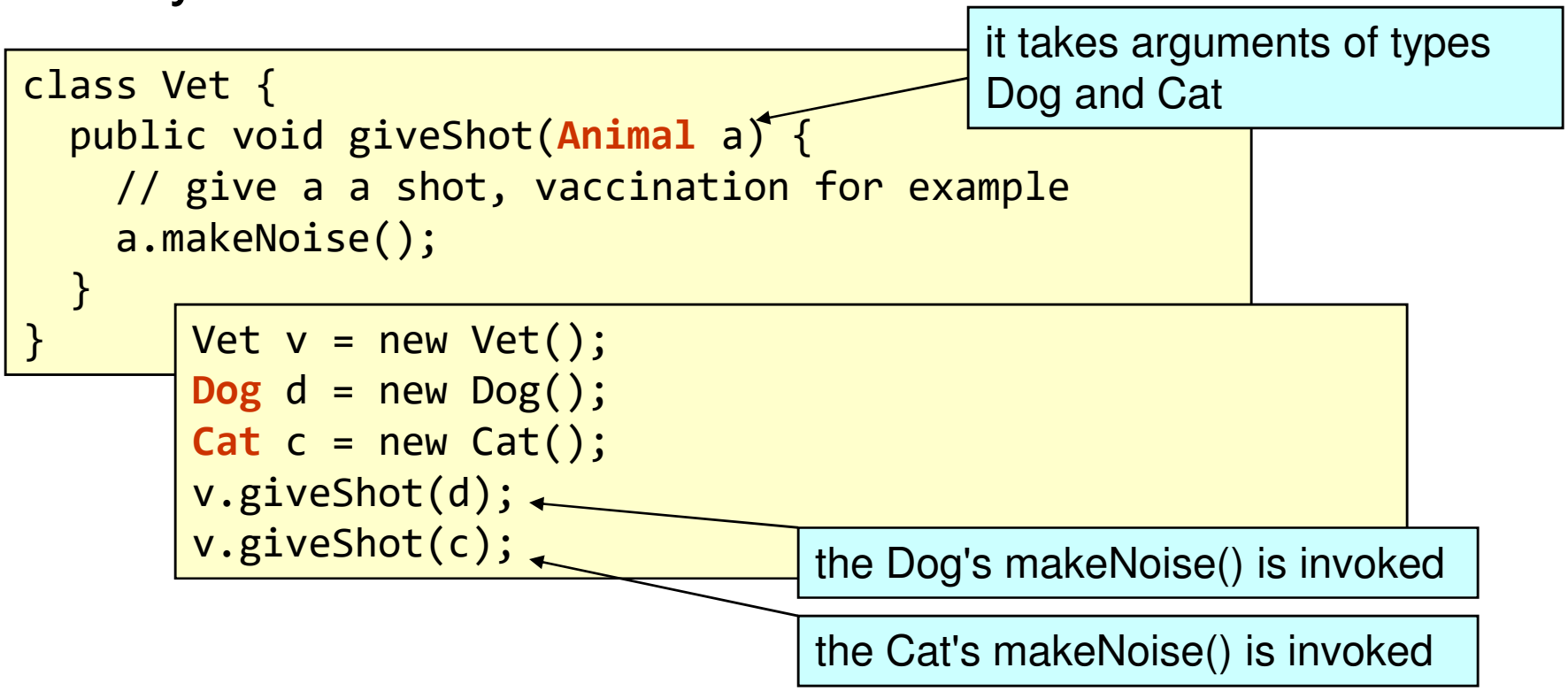
we put objects of any subclasses of Animal in the Animal array

we can loop through the array and call Animal-class methods, and every object does the right thing!

the cat runs Cat's version of makeNoise(), the dog runs Dog's version,…

Polymorphic arguments and return types

- Parameters of type Animal can take arguments of any subclasses of Animal.



```
class Animal {
   String name;
   ...
   public void makeNoise() {
      System.out.print ("Hmm.");
   }
   public void introduce() {
      makeNoise();
      System.out.println(" I'm " + name);
   }
}
class Cat extends Animal {
   ...
   public void makeNoise() {
      System.out.print("Meow...");
   }
}
class Cow extends Animal {
   ...
   public void makeNoise() {
      System.out.print("Moo...");
   }
}
```

Polymorphism: The same message "makeNoise" is interpreted differently, depending on the type of the owner object

```
Animal pet1 = new Cat("Tom Cat");
Animal pet2 = new Cow("Mini Cow");
pet1.introduce();
pet2.introduce();
```

Meow... I'm Tom Cat

Moo... I'm Mini Cow

What is polymorphism?

- Polymorphism: *exist in many forms*
- Object polymorphism:
  - Objects of subclasses can be treated as if they are all objects of the superclass.
    - A Dog object can be seen as an Animal object as well
  - Even when treated uniformly, objects of different subclasses interpret the same message differently
    - anAnimal.makeNoise() works differently depending on what kind of Animal anAnimal is currently refering to.

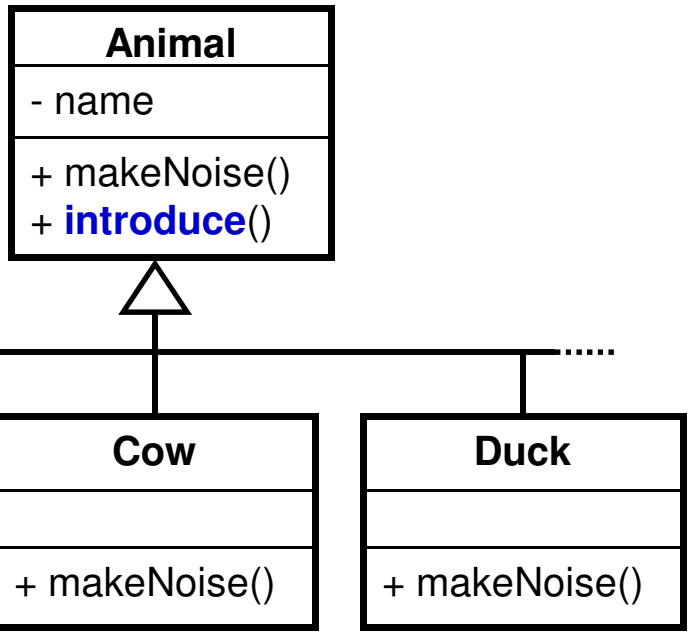
What polymorphism buy you?

- With polymorphism, you can write code that doesn't have to change when you introduce new subclass types into the program

```
Animal[] animals = new Animal[5];
...
for (int i = 0; i < animals.length; i++) {
  animals[i].makeNoise();
}
```

```
class Vet {
  public void giveShot(Animal a) {
    // give a a shot, vaccination for example
    a.makeNoise();
  }
}
```

```
class Animal {
   ...
   public void makeNoise() {
      System.out.print ("Hmm.");
   }
   public void introduce() {
      makeNoise();
      System.out.println(" I'm " + name);
   }
}
```




```
class Pig extends Animal {
   public void makeNoise() {
      System.out.print("Oi oi...");
   }
}
```

```
class Duck extends Animal {
   public void makeNoise() {
      System.out.print("Quack quack...");
   }
}
```

You can add as many new animal types as you want without having to modify the **`introduce()`** method !

⇨ Separate things that change from things that stay the same

`protected` access level

Modifier	accessible within			
	same class	**same package**	**subclasses**	**universe**
`private`	Yes			
`package` (*default*)	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

`protected` access level

- protected members of a superclass are directly accessible from inside its subclasses.

```
public class Person {
   protected String name;
   protected String birthday;
   ...
}
```

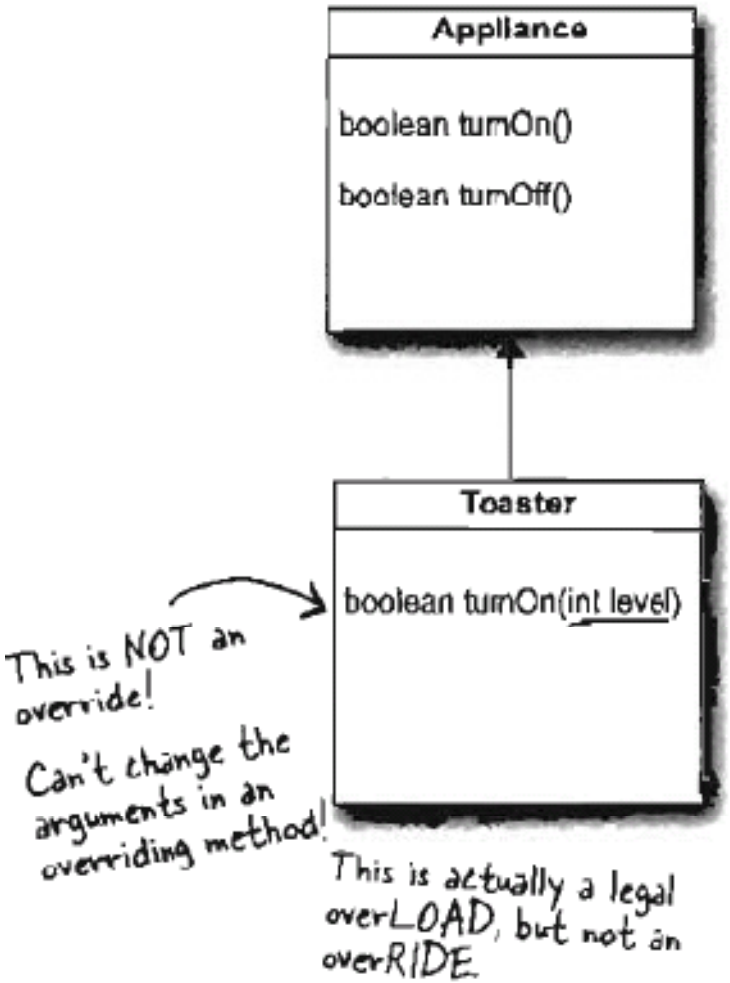
```
public class Employee extends Person {
   protected int salary;
   public String toString() {
      String s;
      s = name + "," + birthday;
      s += "," + salary;
      return s;
   }
}
```

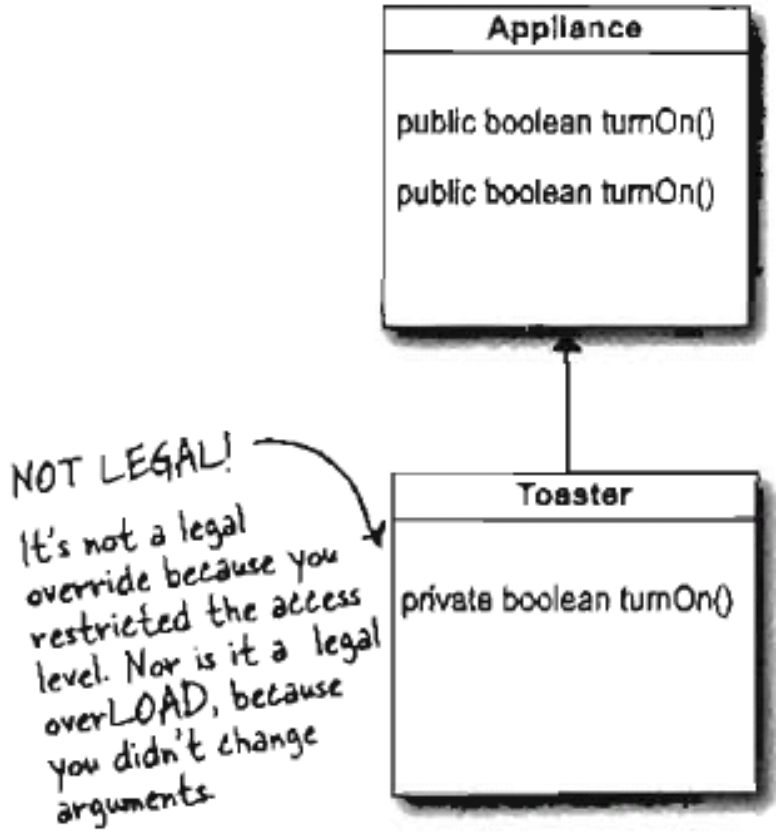
Subclass can directly access superclass's protected members

Rules for overriding

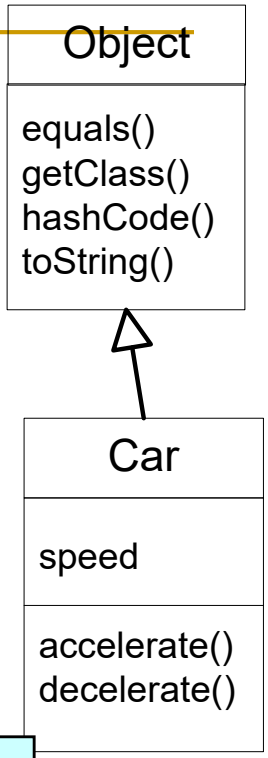
- The principle: the subclass must be able to do anything the superclass declares
- Therefore,
  - Parameter types must be the same
    - whatever the superclass takes as an argument, the subclass overriding the method must be able to take that same argument.
  - Return types must be compatible
    - whatever the superclass declares as return type, the subclass must return the same type or a subclass type.
  - The method can't be less accessible
    - a public method cannot be overriden by a private version

Wrong overriding





Object class



- All classes are subclasses to the class Object
- inherited methods:
  - Class getClass()
  - int hashCode()
  - boolean equals()
  - String toString()

equals() and toString() should be overriden to work properly

```
Car c1 = new Car();
Car c2 = new Car();

System.out.println(c1.equals(c2));
System.out.println(c1.getClass() + c1.hashCode());
System.out.println(c1.toString() + "," + c2);
```