Chapter 6.

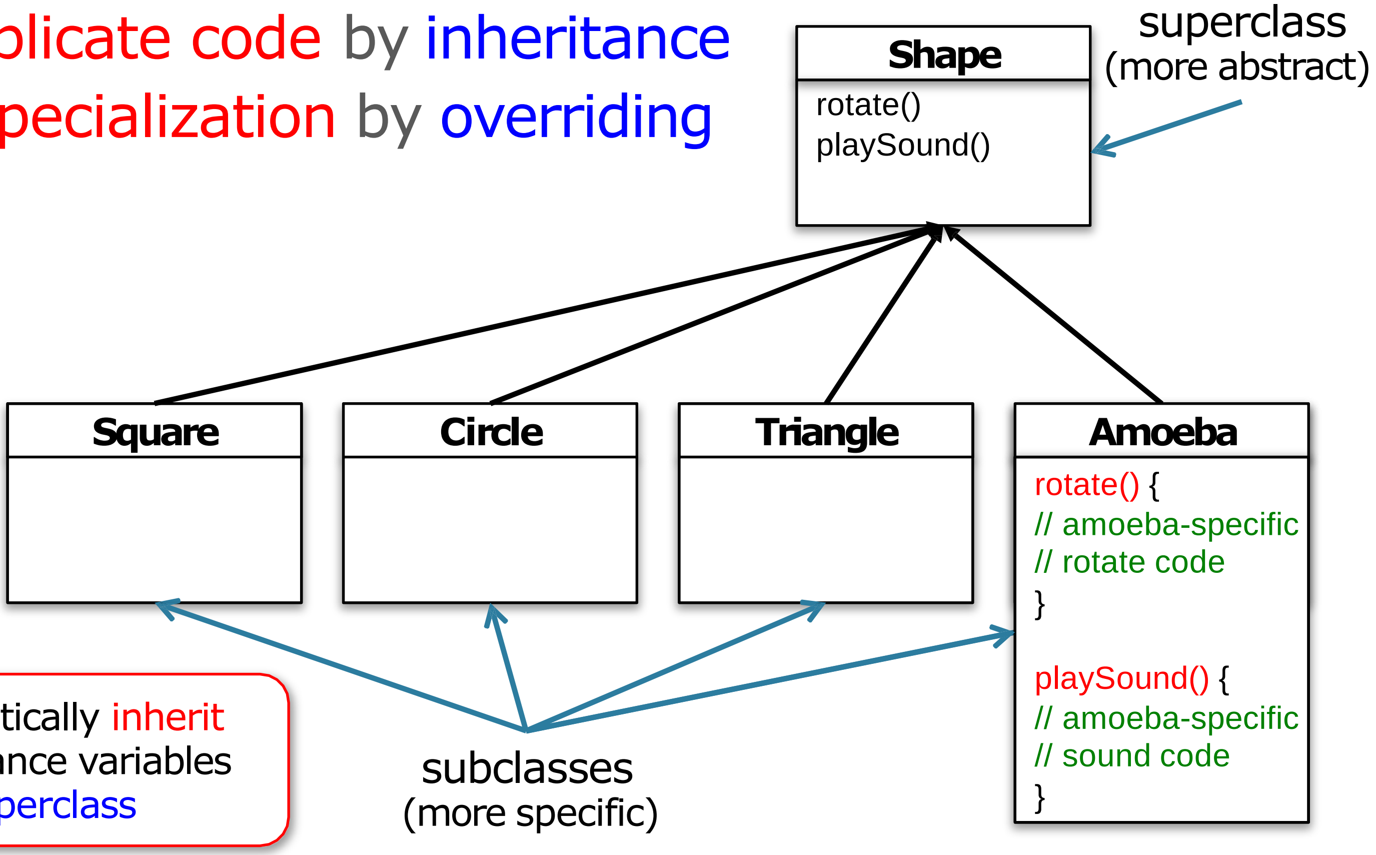
Inheritance and Polymorphism

2020-2021
COMP2396 Object-Oriented Programming and Java

Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

Shapes Example

- Recall the shapes example in Lecture 2, Ocean, the OO guy, wrote a class for each of the 4 shapes
  - Avoid duplicate code by inheritance
  - Handle specialization by overriding methods



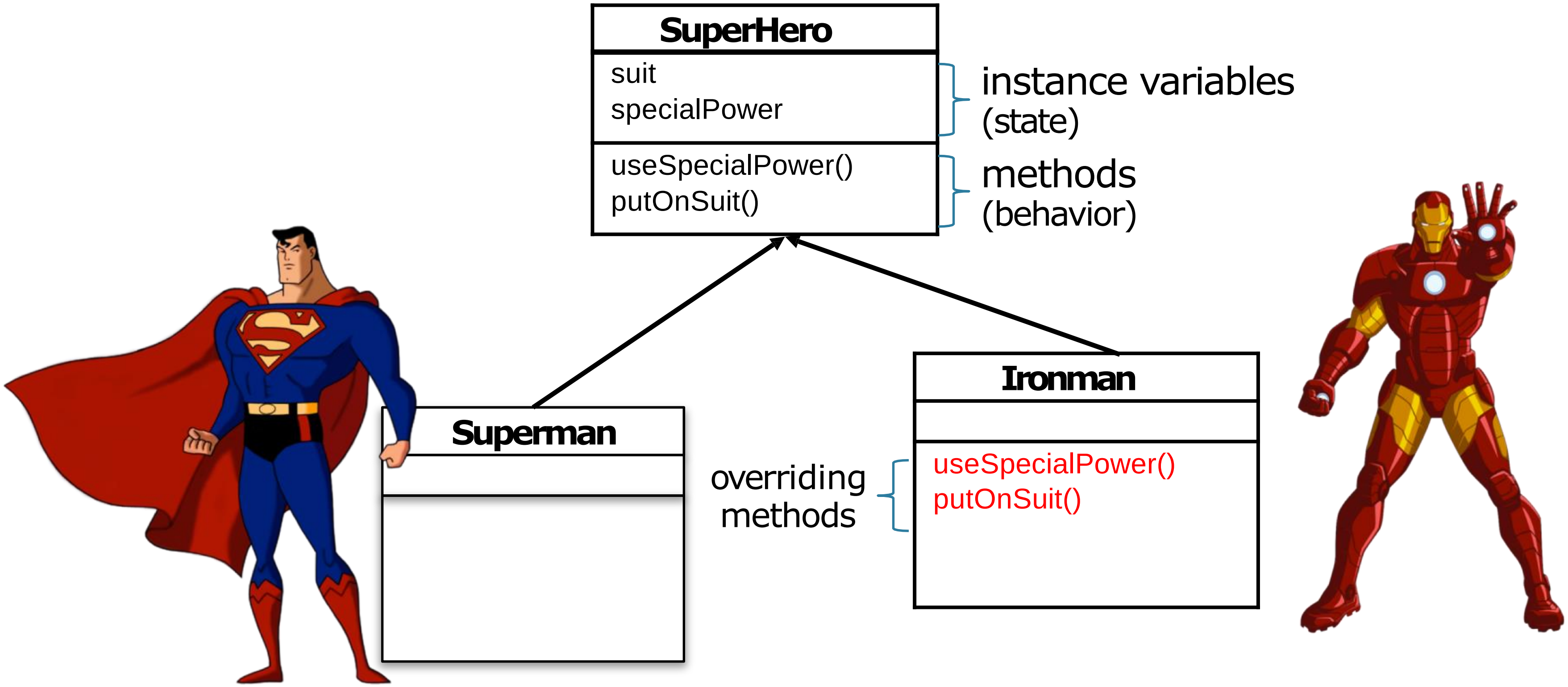
```
rotate() {
// amoeba-specific
// rotate code
}

playSound() {
// amoeba-specific
// sound code
}
```

A subclass will automatically inherit the members (i.e., instance variables and methods) of its superclass

Inheritance Overview

- When designing with inheritance, put common code in a class and make it the superclass of the other more specific classes (which then become its subclasses)
- In Java, we say that a subclass extends its superclass
- There exists an inheritance relationship between a subclass and its superclass where the subclass inherits the members (i.e., instance variables and methods) of its superclass
- A subclass can also add new instance variables and methods of its own, and can override the methods it inherits from its superclass (specialization!)
- Instance variables are not overridden because they don’t need to be (they don’t define any behavior). A subclass can give an inherited instance variable any value it chooses

Example: SuperHero



Superman doesn't need any behavior that's unique, so he doesn't override any method

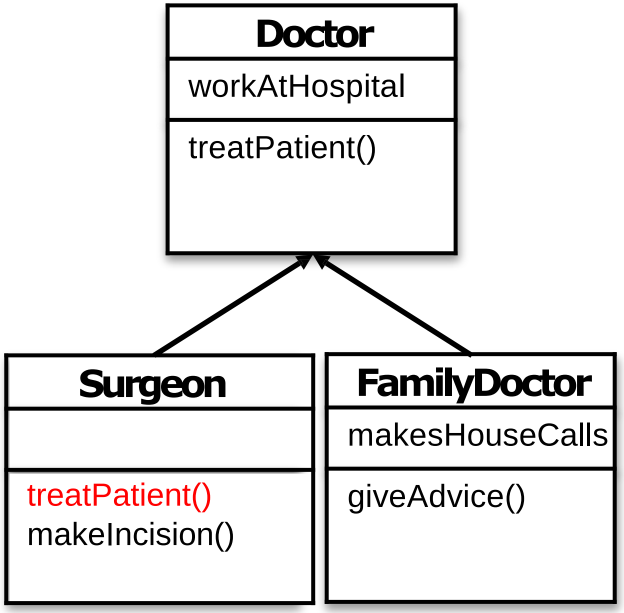
Ironman has specific requirements for his suit and special powers, so both useSpecialPower() and putOnSuit() are overridden in the Ironman class

Example: Doctor

```
public class Doctor {
   boolean workAtHospital;
   void treatPatient() { /* perform a checkup */ }
}
```

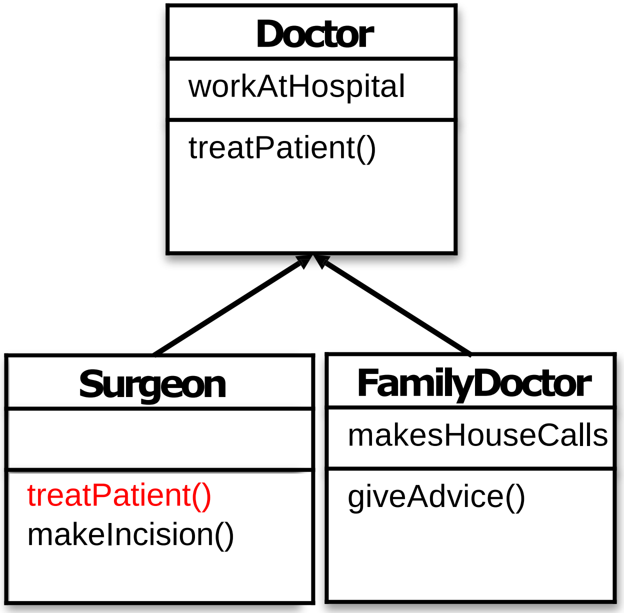
```
public class FamilyDoctor extends Doctor {
   boolean makesHouseCalls;
   void giveAdvice() { /* give homespun advice */ }
}
```

```
public class Surgeon extends Doctor {
   void treatPatient() { /* perform surgery */ }  void
   makeIncision() { /* make incision */ }
}
```



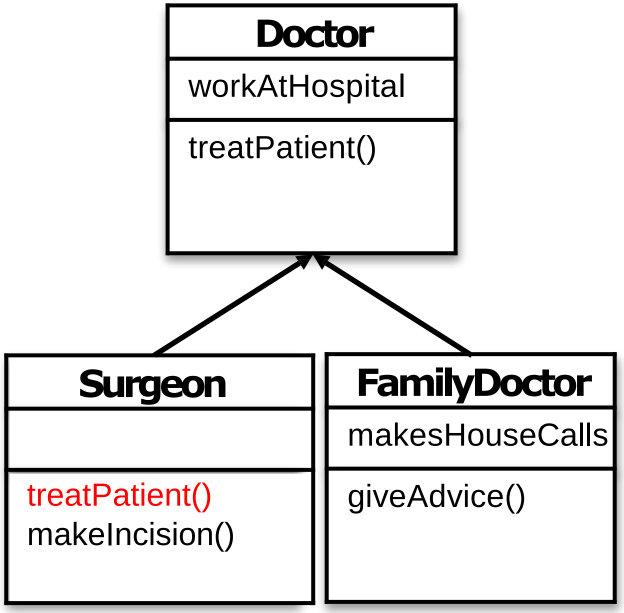
Example: Doctor

- How many instance variables does Surgeon have?
- How many instance variables does FamilyDoctor have?
- How many methods does Doctor have?
- How many methods does Surgeon have?
- How many methods does FamilyDoctor have?
- Can a FamilyDoctor do treatPatient()?
- Can a FamilyDoctor do makeIncision()?



Example: Doctor

- How many instance variables does Surgeon have? 1
- How many instance variables does FamilyDoctor have? 2
- How many methods does Doctor have? 1
- How many methods does Surgeon have? 2
- How many methods does FamilyDoctor have? 2
- Can a FamilyDoctor do treatPatient()? Yes
- Can a FamilyDoctor do makeIncision()? No



Designing an Inheritance Tree

- Ocean was asked to design a simulation program that lets user throw a bunch of different animals into an environment to see what happens
- Initially, the program should support 6 kinds of animals:

- New kinds of animals, however, may be added to the program at any time
- Ocean began by designing an inheritance tree for the animals

Designing an Inheritance Tree

1 Look for objects that have common attributes and behaviors

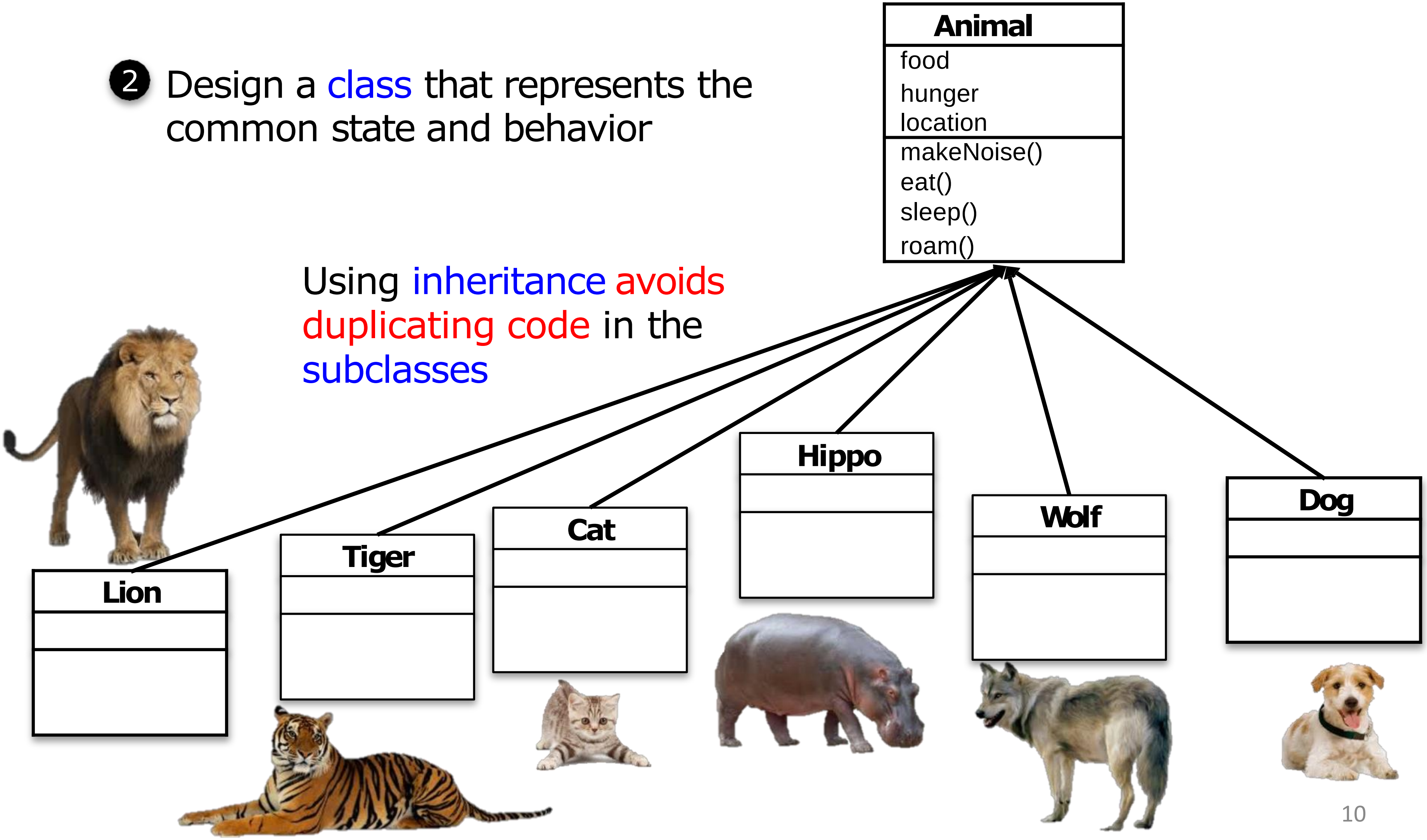
Common attributes:
food
hunger
location

Common behaviors:
makeNoise()
eat()
sleep()
roam()

Designing an Inheritance Tree

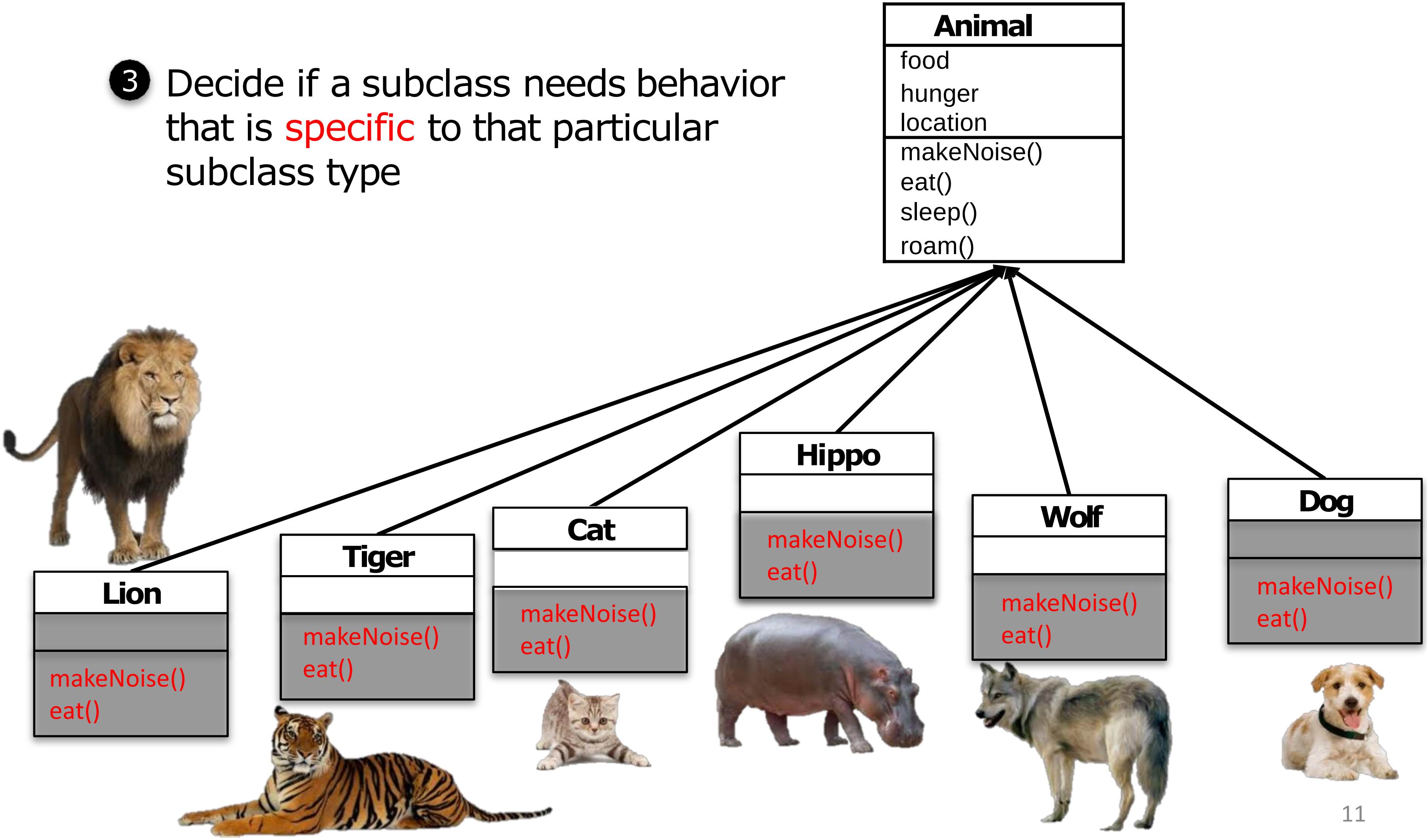
2 Design a class that represents the common state and behavior

Using inheritance avoids duplicating code in the subclasses



Designing an Inheritance Tree

3. Decide if a subclass needs behavior that is specific to that particular subclass type



Animal

food
hunger
location

makeNoise()
eat()
sleep()
roam()

Lion

makeNoise()
eat()

Tiger

makeNoise()
eat()

Cat

makeNoise()
eat()

Hippo

makeNoise()
eat()

Wolf

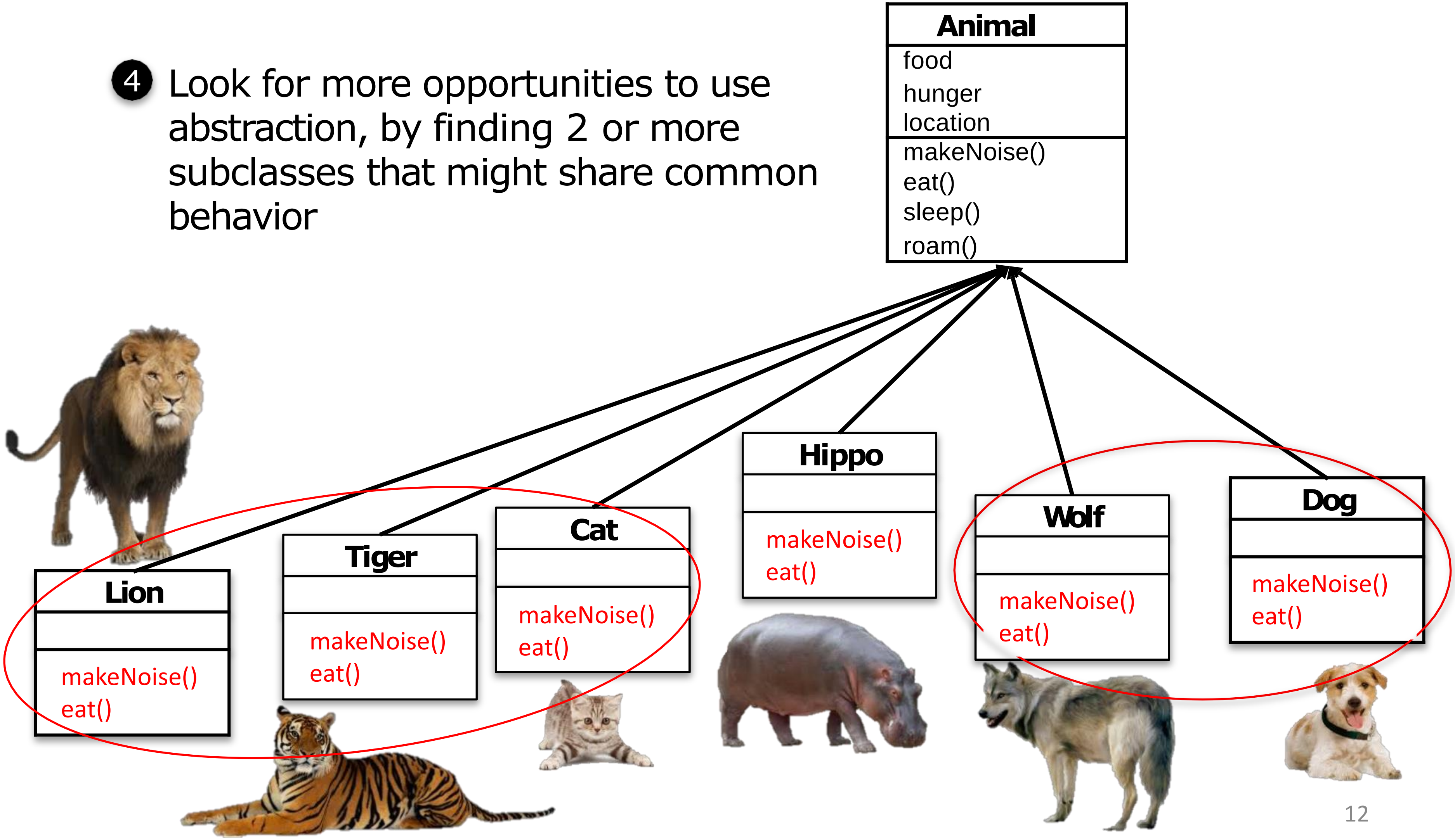
makeNoise()
eat()

Dog

makeNoise()
eat()

Designing an Inheritance Tree

4 Look for more opportunities to use abstraction, by finding 2 or more subclasses that might share common behavior



Animal
food
hunger
location
makeNoise()
eat()
sleep()
roam()

Lion
makeNoise()
eat()

Tiger
makeNoise()
eat()

Cat
makeNoise()
eat()

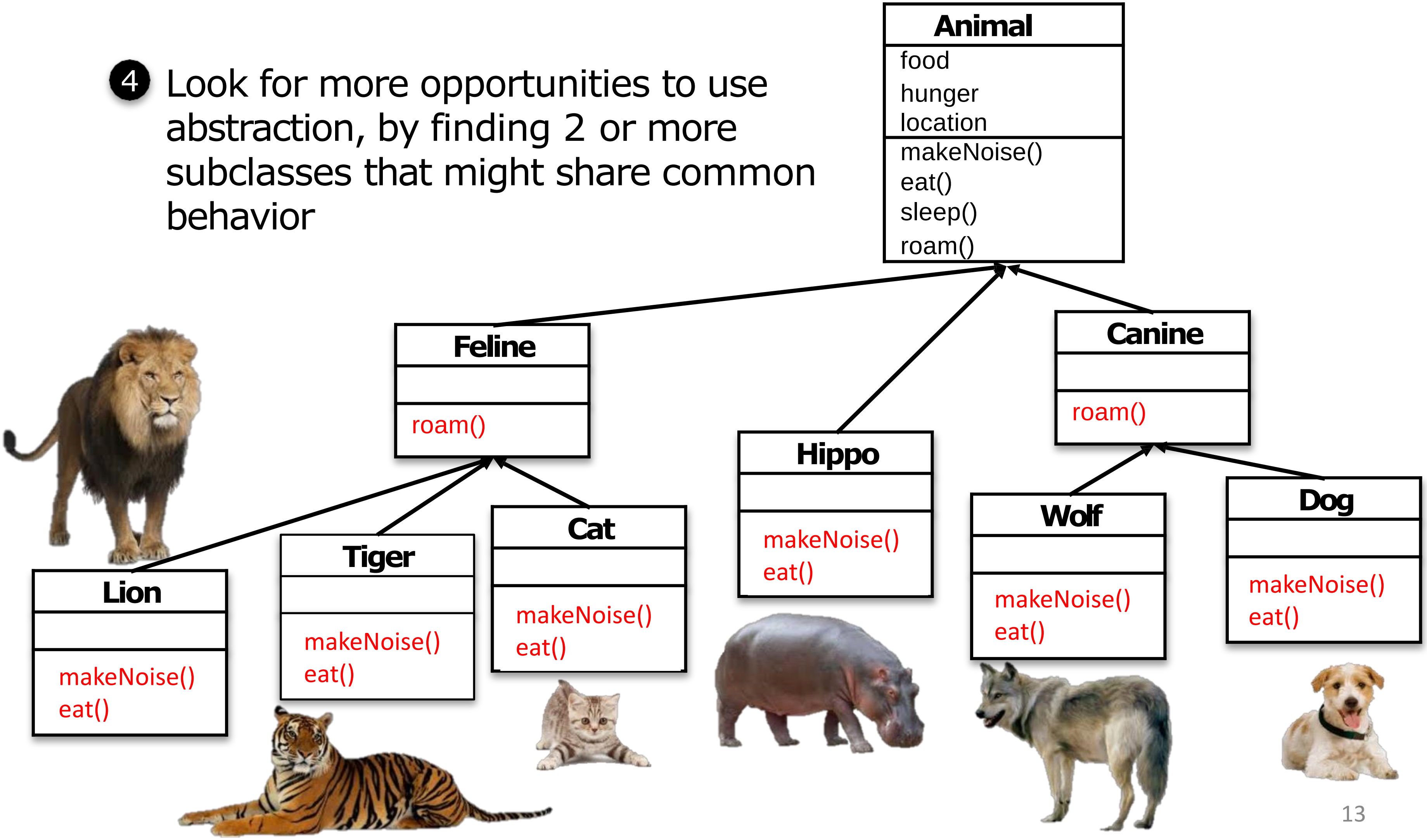
Hippo
makeNoise()
eat()

Wolf
makeNoise()
eat()

Dog
makeNoise()
eat()

Designing an Inheritance Tree

4 Look for more opportunities to use abstraction, by finding 2 or more subclasses that might share common behavior



Animal
food
hunger
location
makeNoise()
eat()
sleep()
roam()

Feline
roam()

Hippo
makeNoise()
eat()

Canine
roam()

Lion
makeNoise()
eat()

Tiger
makeNoise()
eat()

Cat
makeNoise()
eat()

Wolf
makeNoise()
eat()

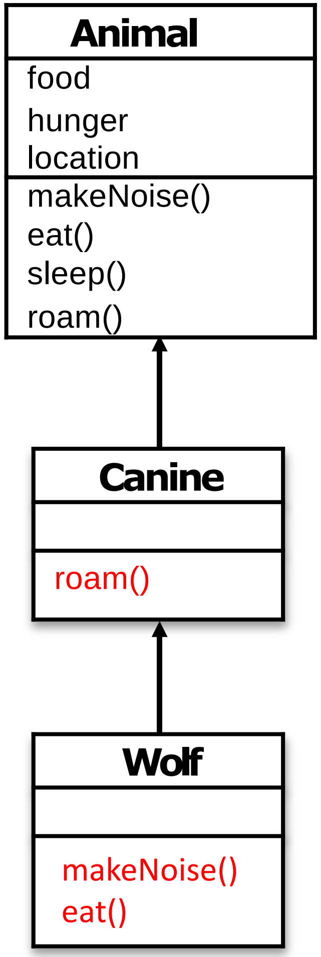
Dog
makeNoise()
eat()

Which method is called?

- The Wolf class has 4 methods: 1 inherited from Animal, 1 inherited from Canine, and 2 overridden in the Wolf class

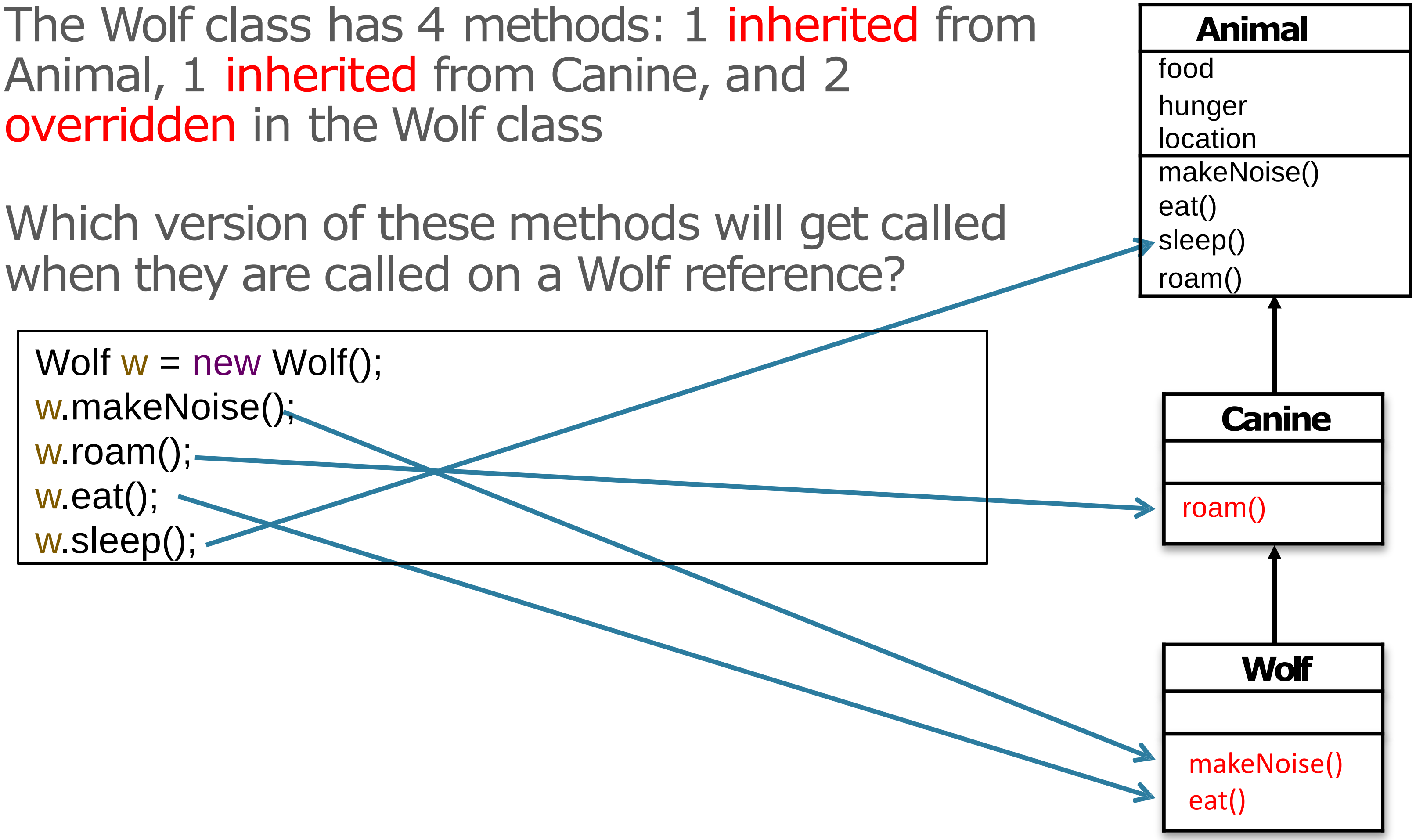
- Which version of these methods will get called when they are called on a Wolf reference?

```
Wolf w = new Wolf();
w.makeNoise();
w.roam();
w.eat();
w.sleep();
```



Which method is called?

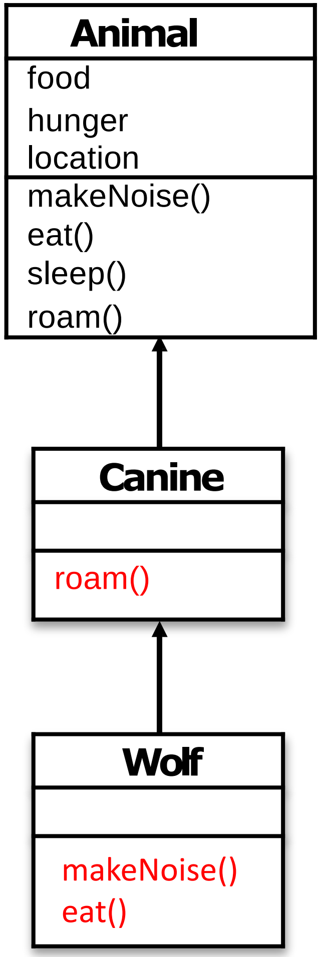
- The Wolf class has 4 methods: 1 inherited from Animal, 1 inherited from Canine, and 2 overridden in the Wolf class
- Which version of these methods will get called when they are called on a Wolf reference?



```
Wolf w = new Wolf();
w.makeNoise();
w.roam();
w.eat();
w.sleep();
```

Which method is called?

- When a method is called on an object reference, the most specific version of the method for that object type will be called
- In other words, the lowest one in the inheritance tree wins!
- In calling a method on a reference to a Wolf object, the JVM starts looking first in the Wolf class
- If the JVM doesn't find a version of the method in the Wolf class, it starts walking up the inheritance hierarchy until it finds a match

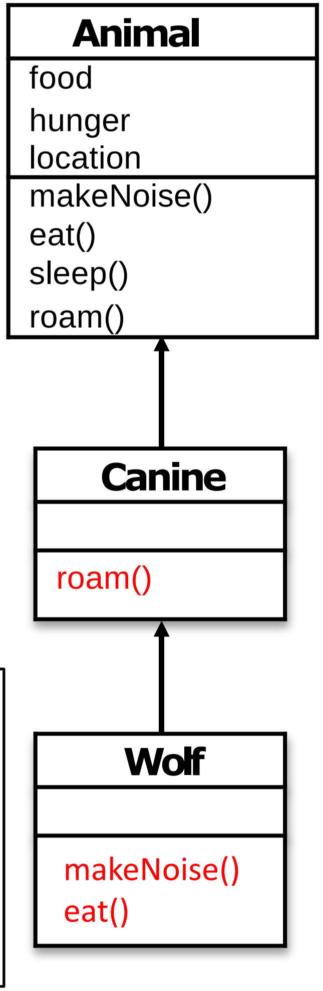


Which method is called?

- It is possible to call an overridden method of the superclass using the keyword super
- Example

```
public class Animal {
   public void roam() {
      System.out.println("Animal roams");
   }
   // ...
}
```

```
public class Canine extends Animal {
   public void roam() {
      super.roam(); // roam() in Animal class is called
      System.out.println("Canine roams");
   }
}
```



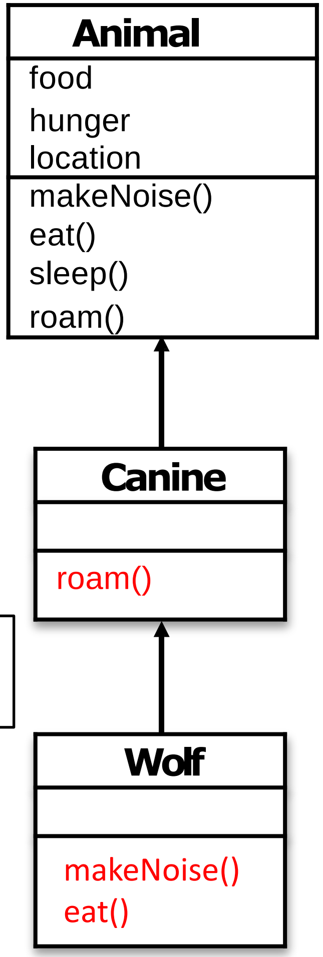
Which method is called?

Example

```
public class SuperTestDrive {
   public static void main(String[] args) {
      Canine c = new Canine();
      c.roam();
   }
}
```

Sample output

```
Animal roams
Canine roams
```



Access Control

- A superclass can choose whether or not it wants a subclass to inherit a particular member by the access level assigned to that particular member
  - public members are inherited
  - private members are not inherited

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

- A private instance variable of the superclass, which is not inherited, may still be accessed through the inherited public getter and public setter

Access Control

more restrictive

less restrictive

Access level	Access modifier	Class	Package	Subclass	World
private	private	Y	N	N	N
default	(none)	Y	Y	N	N
protected	protected	Y	Y	Y	N
public	public	Y	Y	Y	Y

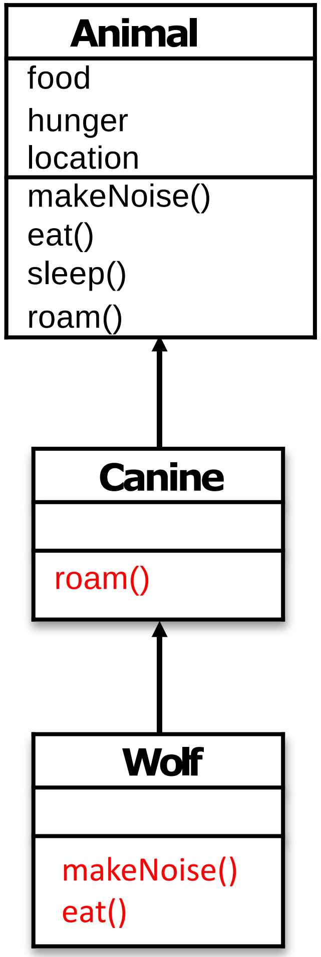
- private means that only code within the same class can access it
- default means that only code within the same package as the class with the default member can access it
- protected works just like default except it also allows subclasses outside the package to inherit the protected member
- public means any code anywhere can access it

IS-A Relationship

- When one class inherits from another, we say the subclass extends its superclass
- Recall that there exists a IS-A relationship between a subclass and its superclass: a subclass object IS-A superclass object (which means a subclass object can do anything that a superclass object can do)
- To check whether one thing, say X, should extend another, say Y, apply the IS-A test: check if it makes sense to say X IS-A Y?
- Examples:
  - Triangle IS-A Shape (Triangle extends Shape)
  - Surgeon IS-A Doctor (Surgeon extends Doctor)

IS-A Relationship

- The IS-A test works anywhere in the inheritance tree
- If class B extends class A, and class C extends class B, then class C passes the IS-A test for both class B and class A
- Example
  - Wolf IS-A Canine (Wolf extends Canine)
  - Canine IS-A(n) Animal (Canine extends Animal)
  - Wolf IS-A(n) Animal (Wolf extends Animal)
- Note that the IS-A relationship works in only one direction!



HAS-A Relationship

- What about “Tub extends Bathroom”?
- To check whether Tub should extend Bathroom, ask the question: “Does it make sense to say Tub IS-A Bathroom?”

- “Tub IS-A Bathroom” is definitely false, and therefore Tub should not extend Bathroom
- Tub and Bathroom are joined by a HAS-A relationship
- “Bathroom HAS-A TUB” means that Bathroom has a Tub instance variable

Rules in Inheritance Design

- When designing with inheritance
  - DO use inheritance when one class is a more specific type of a superclass
  - DO consider inheritance when common behavior should be shared among multiple classes of the same general type
  - DO NOT use inheritance just for reusing code from another class if the relationship between the subclass and superclass violates either of the above 2 rules
  - DO NOT use inheritance if the subclass and the superclass do not pass the IS-A test

Method Overriding

- Recall that a subclass object IS-A superclass object, and it can do anything that the superclass object can do
- When a subclass overrides a method inherited from its superclass, it must make sure that
  - Argument list must be the same
  - Return type must be compatible (i.e., either the same type or a subclass type)
  - The method cannot be less accessible (i.e., either the same or friendlier access level)

Method Overriding

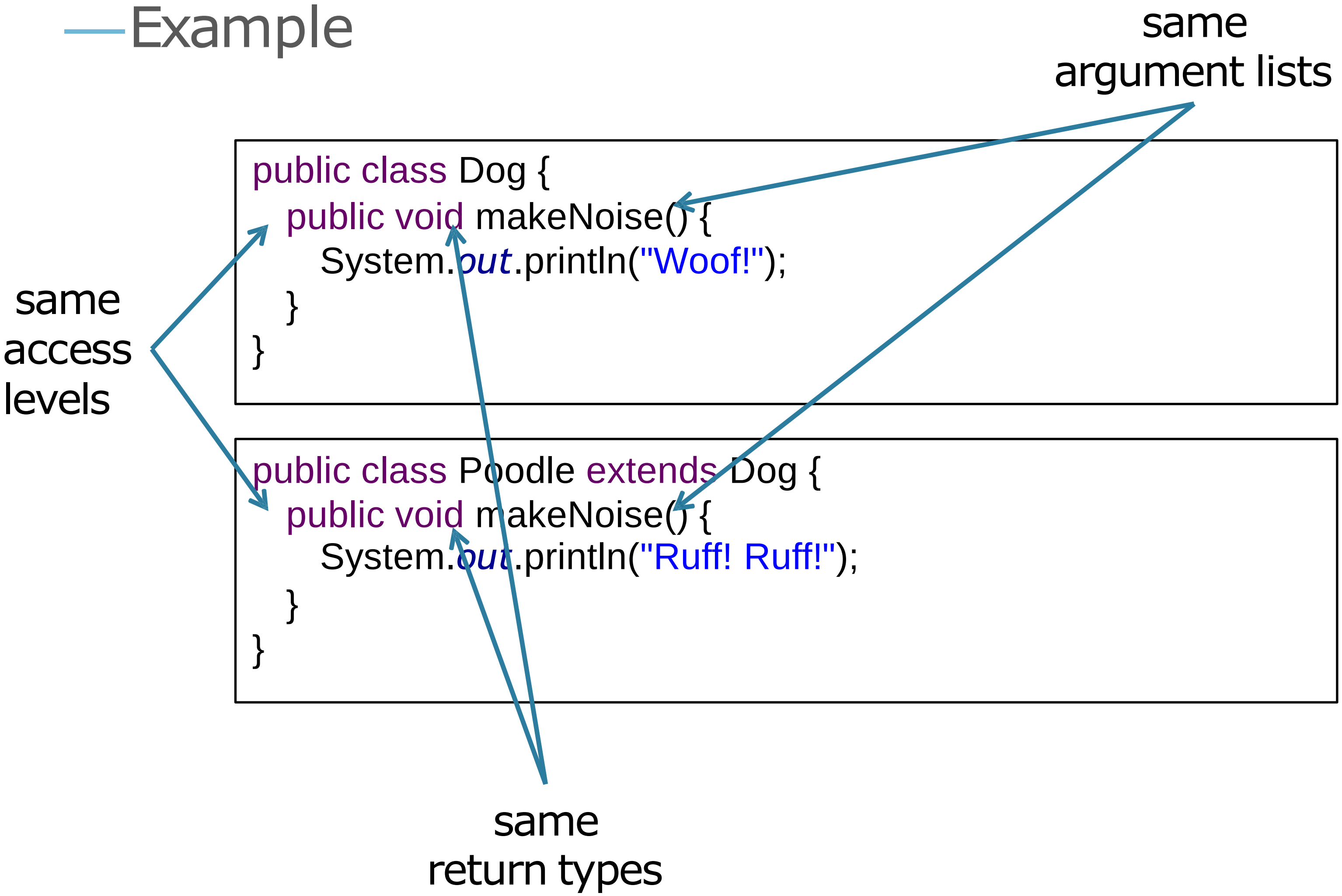
Example

```
public class Dog {
   public void makeNoise() {
      System.out.println("Woof!");
   }
}
```

```
public class Poodle extends Dog {
   public void makeNoise() {
      System.out.println("Ruff! Ruff!");
   }
}
```

Method Overriding

- Example



same argument lists

```
public class Dog {
   public void makeNoise() {
      System.out.println("Woof!");
   }
}
```

same access levels

```
public class Poodle extends Dog {
   public void makeNoise() {
      System.out.println("Ruff! Ruff!");
   }
}
```

same return types

Method Overloading

- Method overloading is nothing more than having 2 or more methods with the same name but different argument lists

- It has nothing to do with inheritance and polymorphism

- For overloaded methods
  - Argument lists must be different
  - Return types can be different
  - Access levels can be different

Method Overloading

- Example

```
public class Dog {
   public void makeNoise() {
      System.out.println("Woof!");
   }
}
```

```
public class Poodle extends Dog {
   public void makeNoise(int n) {
      for (int i = 0; i < n; i++) {
         System.out.println("Ruff! Ruff!");
      }
   }
}
```

Method Overloading

- Example

different argument lists

```
public class Dog {
   public void makeNoise() {
      System.out.println("Woof!");
   }
}
```

```
public class Poodle extends Dog {
   public void makeNoise(int n) {
      for (int i = 0; i < n; i++) {
         System.out.println("Ruff! Ruff!");
      }
   }
}
```

Benefits of Inheritance

- Get rid of duplicate code
  - Behaviors common to a group of classes are abstracted out and put in a superclass
  - Only one place to update when modifications of these common behaviors are needed
  - In Java, all classes that extend the superclass will automatically use the new version. Hence no need to touch the subclasses, not even recompiling them!

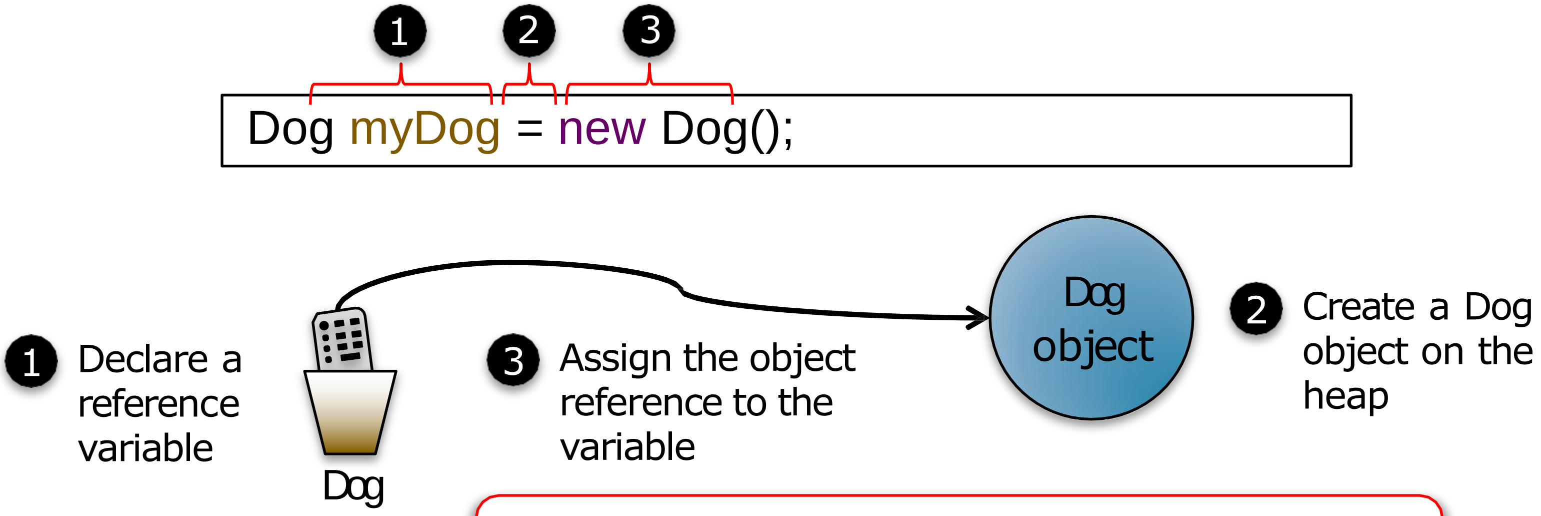
Benefits of Inheritance

- Define a common protocol for a group of classes
  - Establishes a contract which guarantees all classes that extend a superclass have all the inheritable methods of that superclass
  - Allows any subclass object be substituted where the superclass object is expected (i.e., polymorphism)
  - Example:

```
Animal animal1 = new Dog();
Animal animal2 = new Cat();
```

The Way Polymorphism Works

- The 3 steps of object declaration, creation and assignment
  1. Declare a reference variable
  2. Create an object on the heap
  3. Assign the object reference to the variable



```
Dog myDog = new Dog();
```

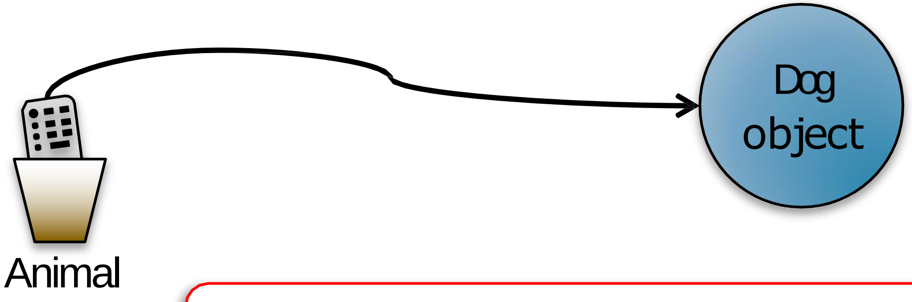
1
2
3
1 Declare a reference variable
Dog
3 Assign the object reference to the variable
Dog object
2 Create a Dog object on the heap

Here, both the reference type and the object type are the same (i.e., both are Dog)

The Way Polymorphism Works

- With polymorphism
  - The reference type can be a superclass of the actual object type
  - Any object that passes the IS-A test for the declared type of the reference variable can be assigned to that reference variable

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



The reference variable type is declared as Animal, but the object created is a Dog object

Benefits of Polymorphism

- Can do things like polymorphic arrays, polymorphic argument and polymorphic return types
- Makes it possible to write code that does not have to change when new subclass types are introduced
- Example

```
Animal[] animals = new Animal[6];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Wolf();
animals[3] = new Hippo();
animals[4] = new Lion();
animals[5] = new Tiger();

for (int i = 0; i < 6; i++) {
   animals[i].makeNoise ();
   animals[i].eat();
}
```

Calling the methods on the actual subclass objects in runtime

Benefits of Polymorphism

- Example

```
public class Vet {
   public void giveShot(Animal a) {
      // gives shot to the Animal
      a.makeNoise();
   }
}
```

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

This will work for future Animal subclasses as well

We are with you!

If you encounter any problems in understanding the materials in the lectures, **please feel free to contact me or my TAs**. **We are always with you!**

We wish you enjoy learning Java in this class. 🙂

Chapter 6.

End

2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong