Inheritance and Polymorphism

Vertebrates

animals with backbones

Mammals

suckle young (feed babies milk)
skin covered by hair/fur
breathe air
give birth to fully formed young

Land Mammals

Mammalia

outer ears
four limbs (arms/legs)

Marsupials

care for and feed their young in a pouch

Primates

well developed hands/feet with fingers/toes
can judge distance
very intelligent
social animals/ form bonds with family and friends

Rodents

gnawing animal
large incisor teeth (two pairs)
use like chisels to gnaw on hard foods

Marine Mammals

Animals which grow in and live in the water
some only have sparse covering of hair

Flying Mammals

Use echolocation
nocturnal
roosts in trees or caves, under roofs

Fish

scales
fins and tails
breathe underwater using gills

Reptiles

dry skin and scales
lay eggs
breathe air

Birds

Aves

feathers
two wings
claws
lay eggs

Amphibia

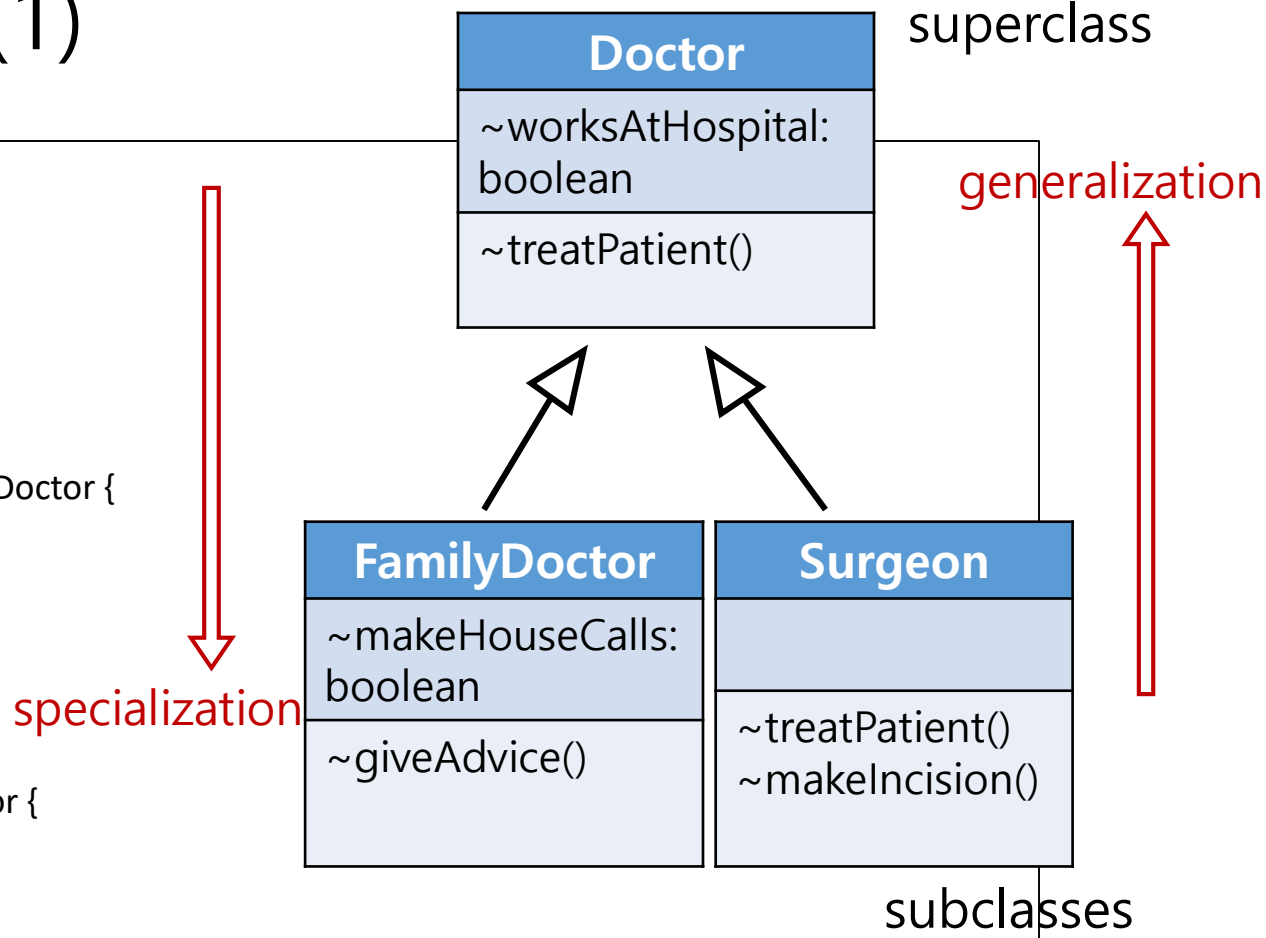
young live
water but
adults live
land
smooth we
slimy ski
lay eggs

Inheritance (1)

```
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() { // Override
    // perform surgery
  }
  void make Incision() {
    // make incision (yikes!)
  }
}
```



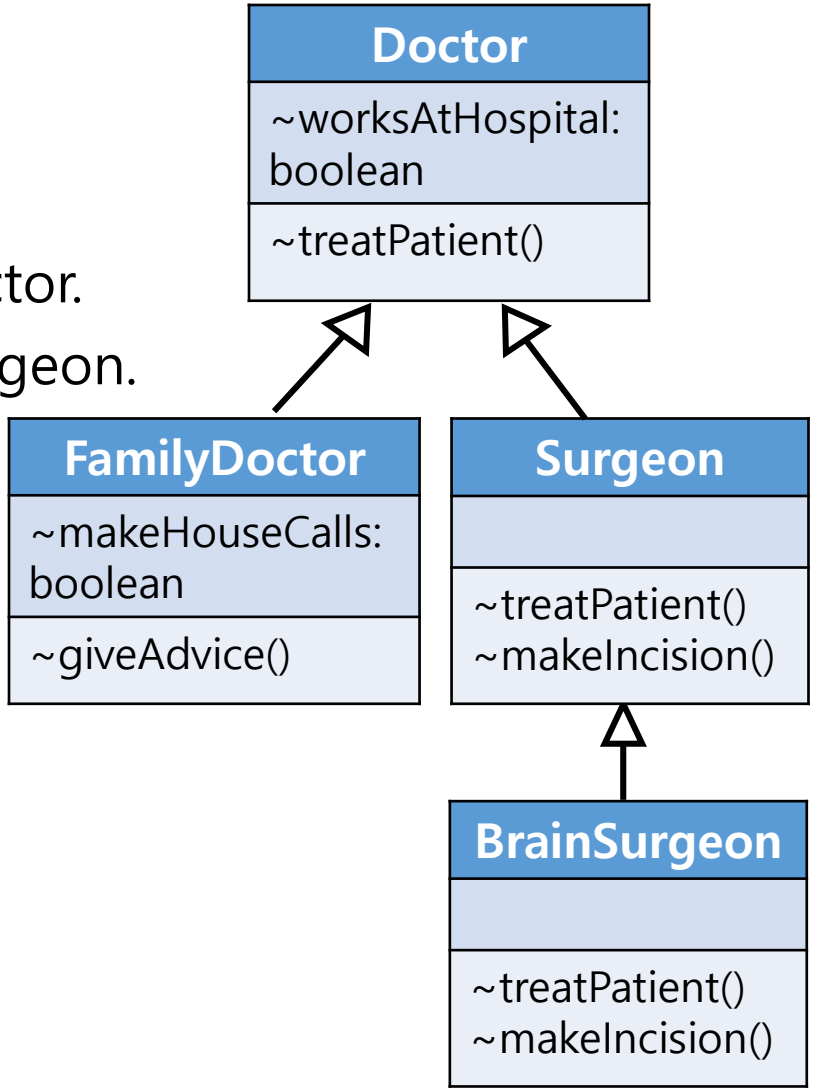
Inheritance

- A subclass inherits **(non-private) instance variables** and **methods** of its superclass.
  - In Java, we say that a subclass **extends** its superclass.
- A subclass can add new methods and instance variables of its own.
- A subclass can override a method that it inherits from the superclass.
- Instance variables are not overridden.

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Inheritance (2)

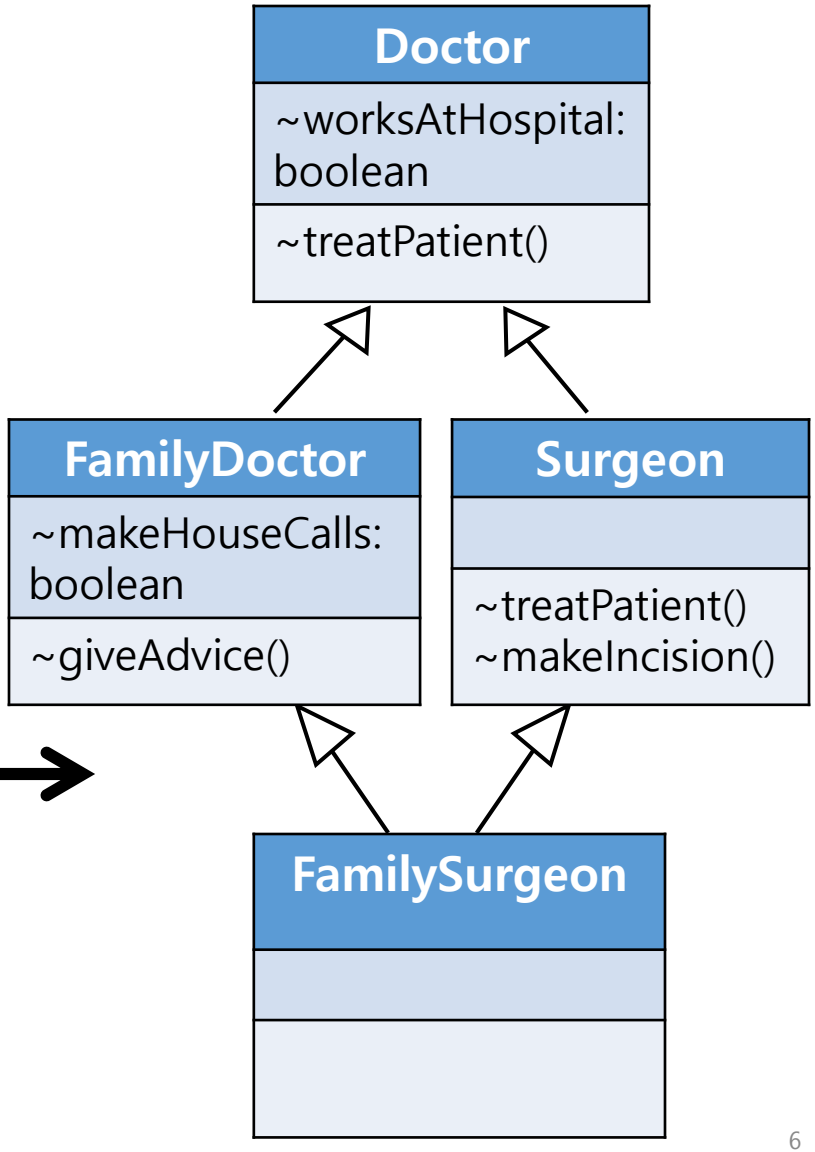
- Surgeon is a subclass of Doctor.
- Doctor is a superclass of Surgeon.



Multiple Inheritance

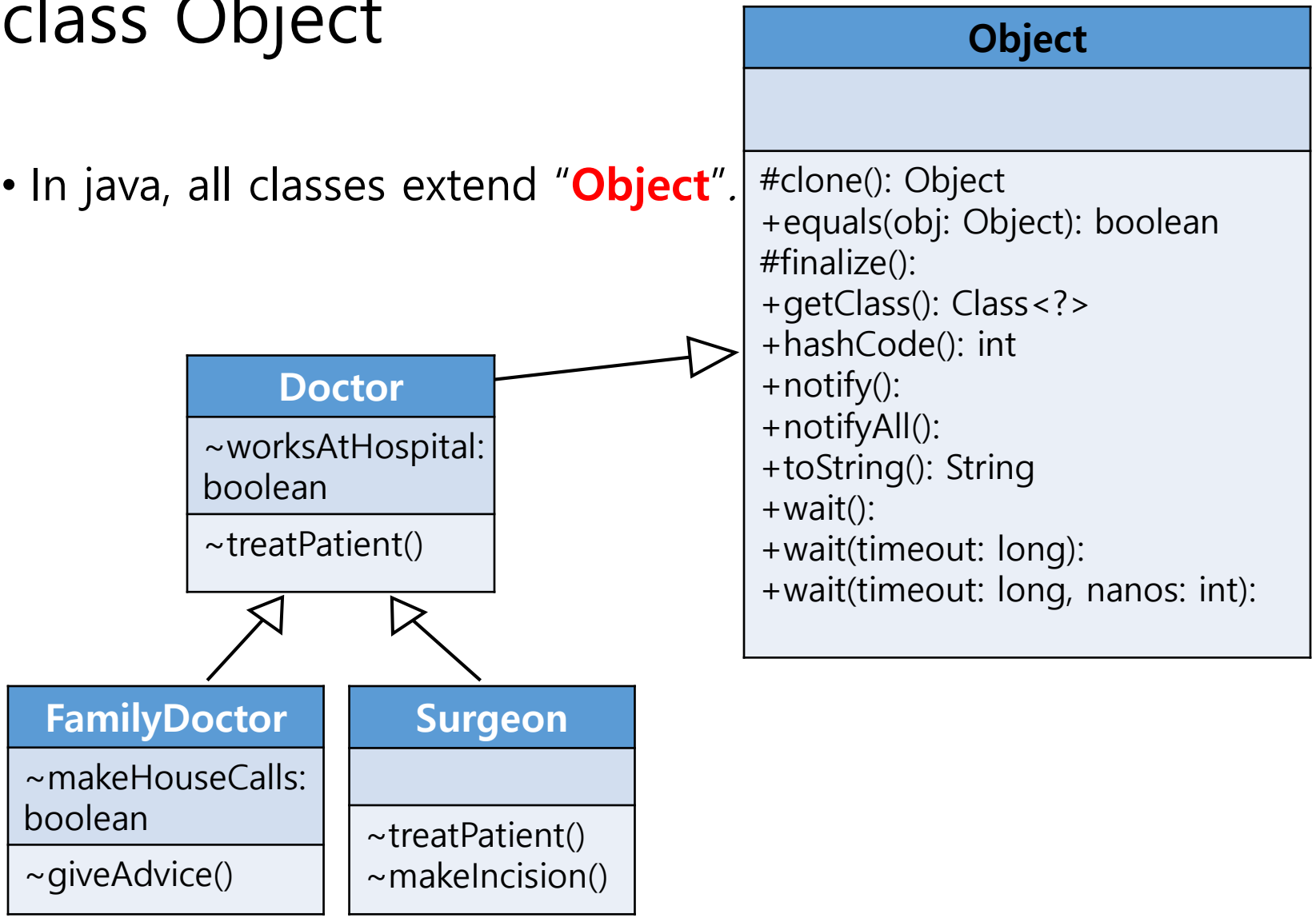
- It's *NOT* allowed in Java

Multiple inheritance →



class Object

- In java, all classes extend "**Object**".

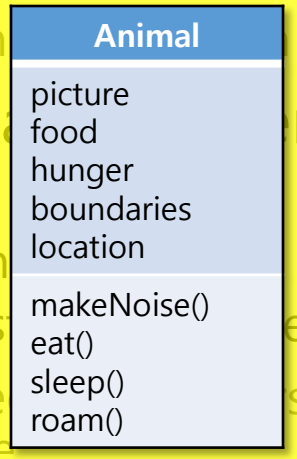


Inheritance Tree Design: Animal Simulation Program

1. Look for objects that have common attributes and behaviors.
   - Six objects: lion, hippo, tiger, dog, cat, wolf.
   - What do these six types have in common?
   - How are these types related?
2. Design a class that represents the common state and behavior.
3. Decide if a subclass needs behaviors that are specific to that particular subclass type.
4. Look for more opportunities to use abstraction, by finding two or more subclasses that might need common behaviors.
5. Finish the class hierarchy.

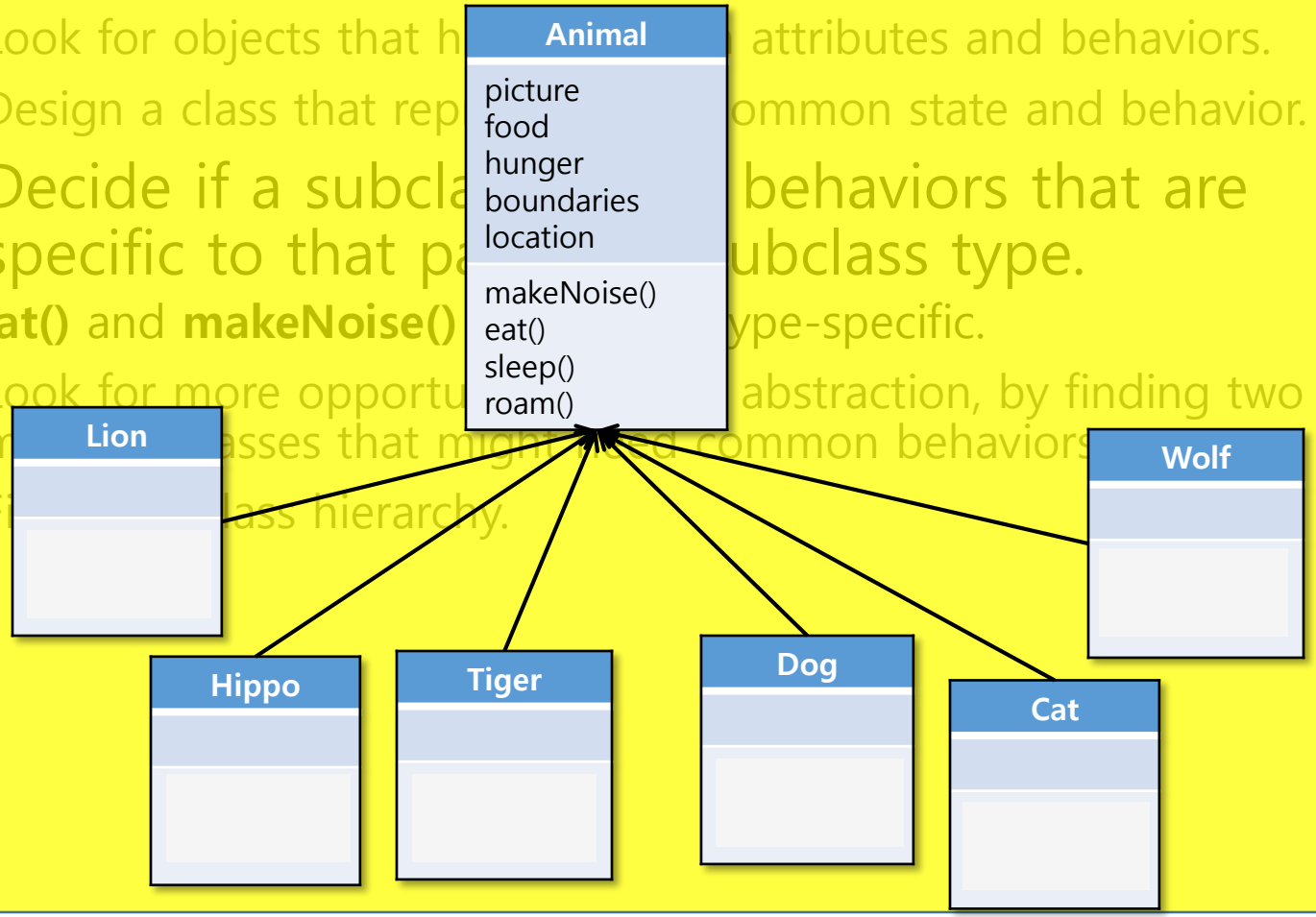
Inheritance Tree Design: Animal Simulation Program

1. Look for objects that h attributes and behaviors.
2. Design a class tha nts the common state and behavior.
   - These objects are all an
   - Put in methods and ins es that all animals might need.
3. Decide if a subclass ne s that are specific to that particular subclass type.
4. Look for more opportunities to use abstraction, by finding two or more subclasses that might need common behaviors.
5. Finish the class hierarchy.



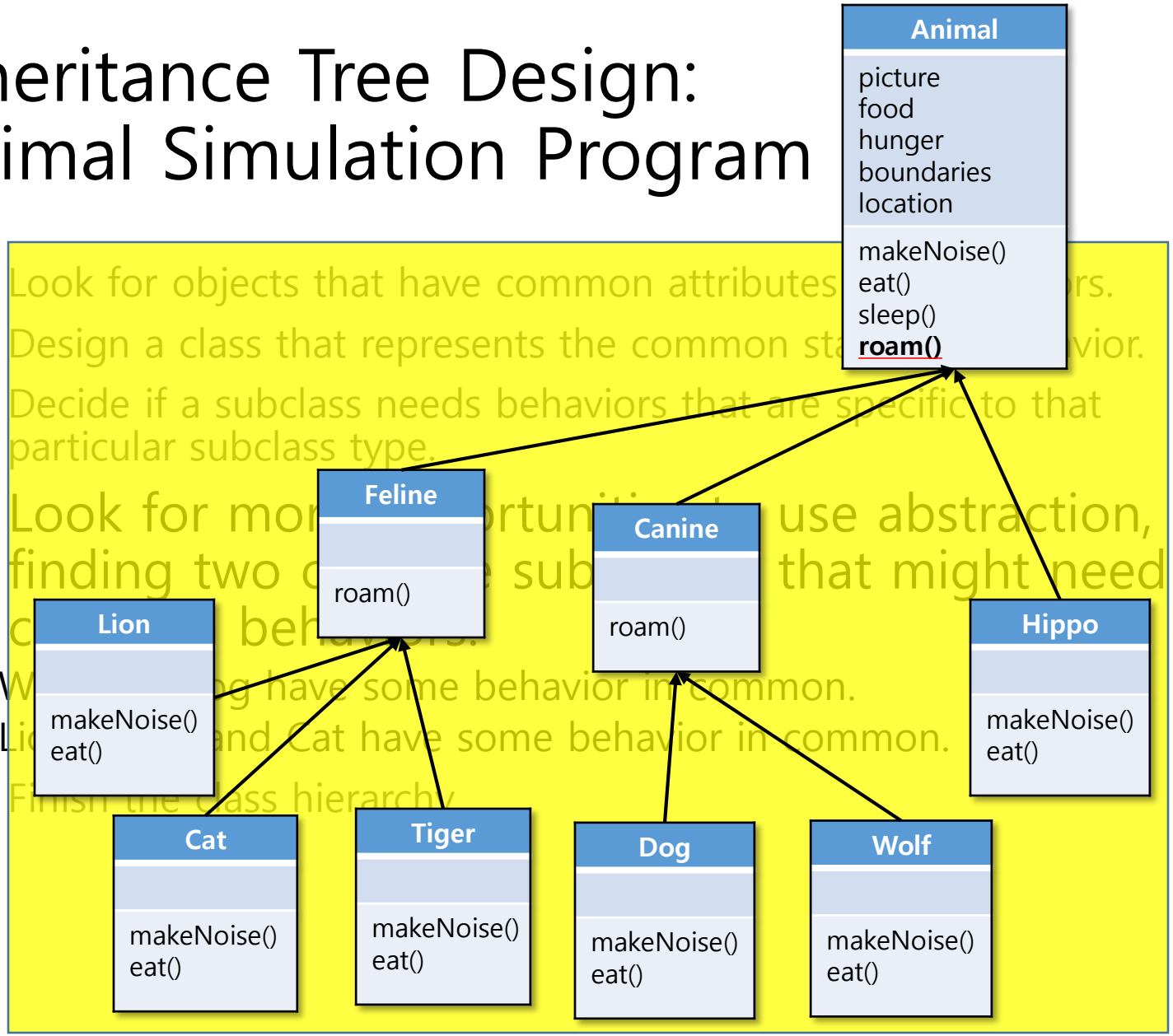
Inheritance Tree Design: Animal Simulation Program

1. Look for objects that h attributes and behaviors.
2. Design a class that rep ommon state and behavior.
3. Decide if a subcla behaviors that are specific to that pa ubclass type.
   - **eat()** and **makeNoise()** ype-specific.
4. Look for more opportu abstraction, by finding two or m asses that might share common behaviors
5. Fi lass hierarchy.



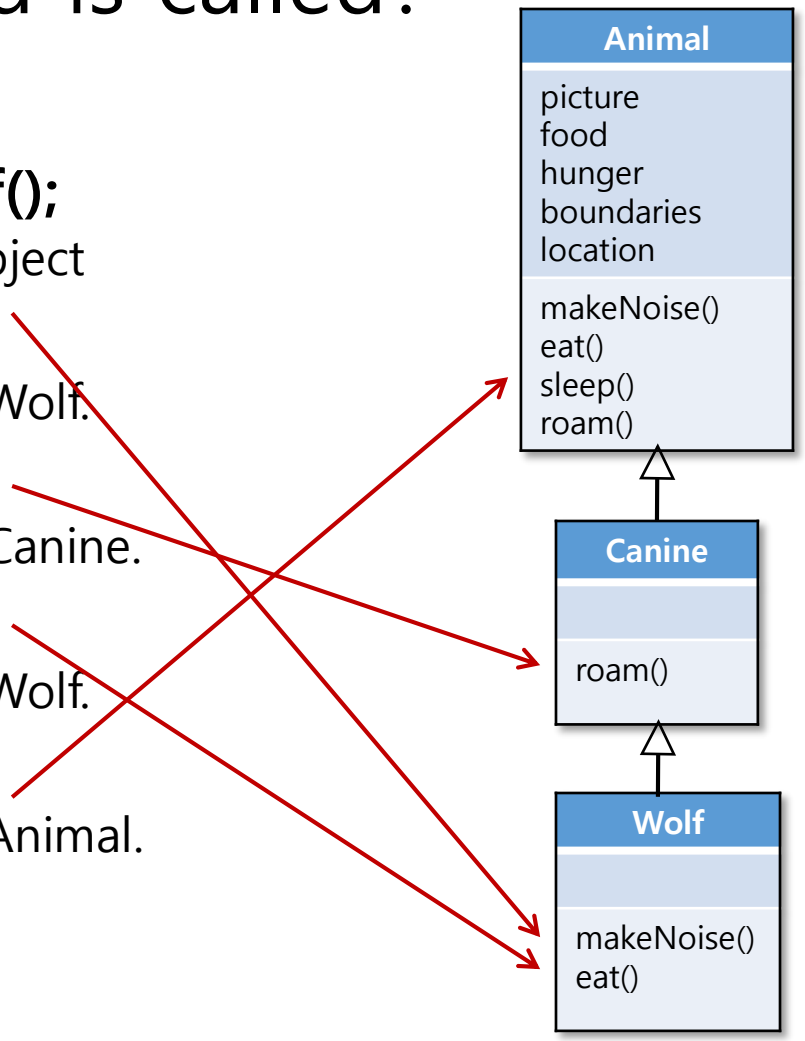
Inheritance Tree Design: Animal Simulation Program

1. Look for objects that have common attributes [illegible]rs.
2. Design a class that represents the common sta[illegible]vior.
3. Decide if a subclass needs behaviors that are specific to that particular subclass type.
4. Look for mor[illegible]ortun[illegible] use abstraction, by finding two c[illegible] sub[illegible] that might need c[illegible] behaviors.
   - W[illegible]g have some behavior in common.
   - Li[illegible]and Cat have some behavior in common.
5. Finish the class hierarchy.



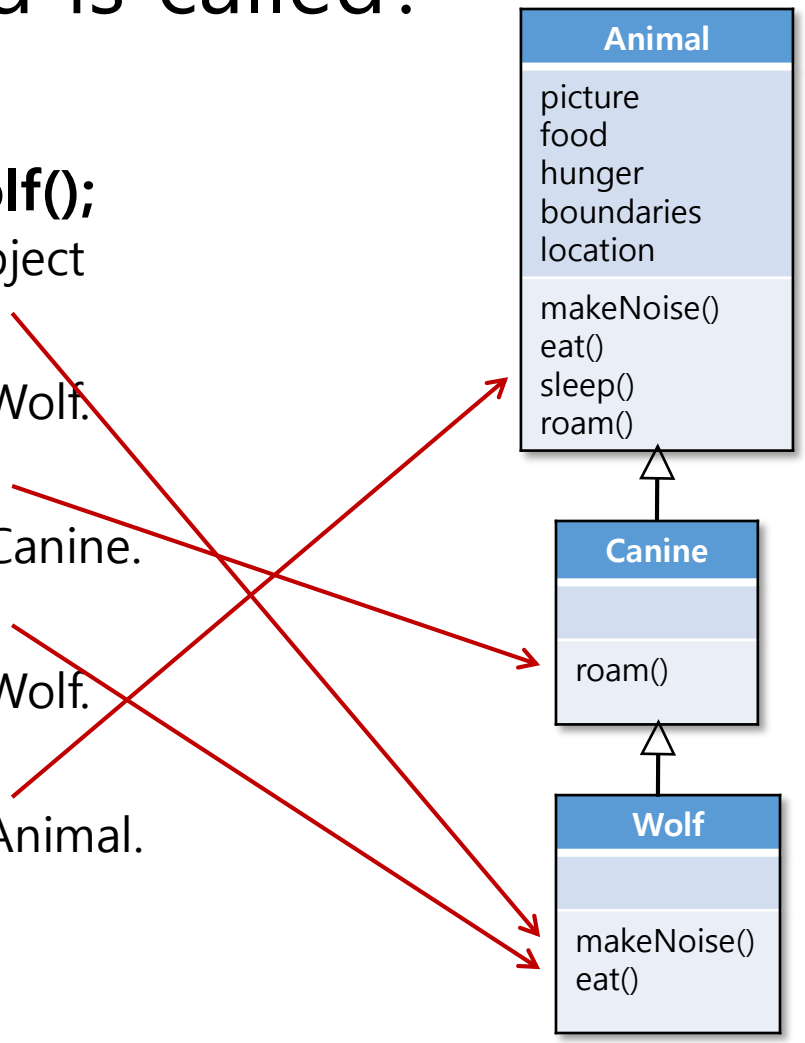
Which method is called?

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



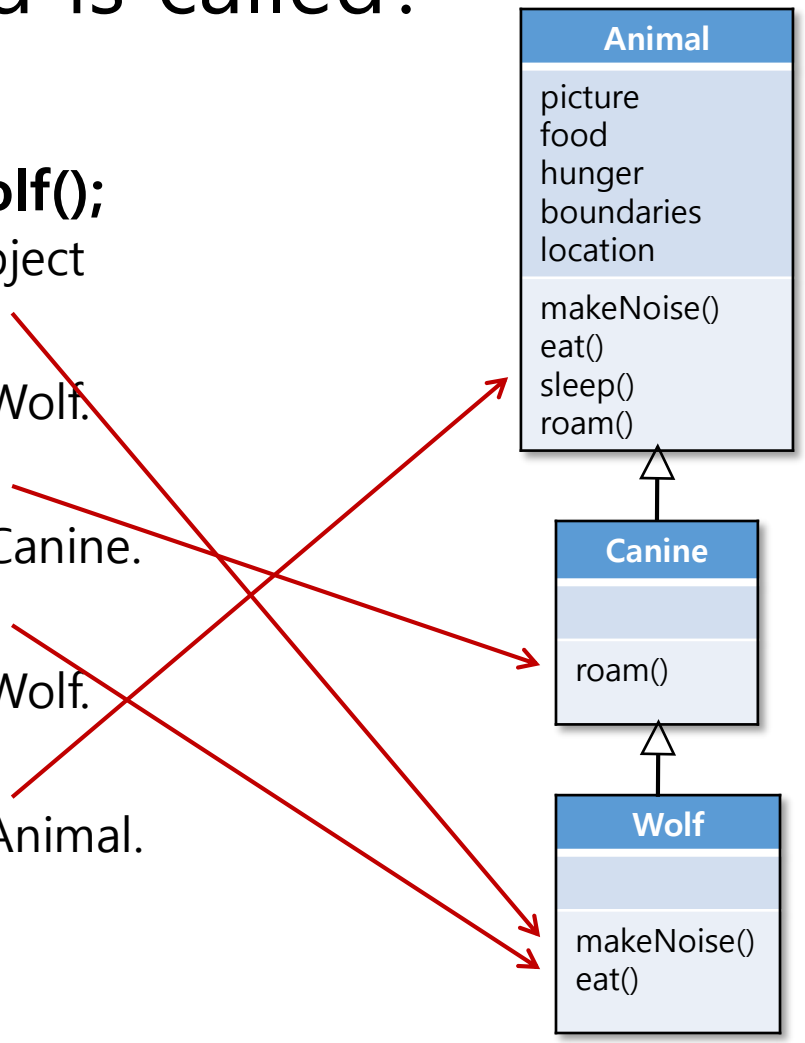
Which method is called?

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



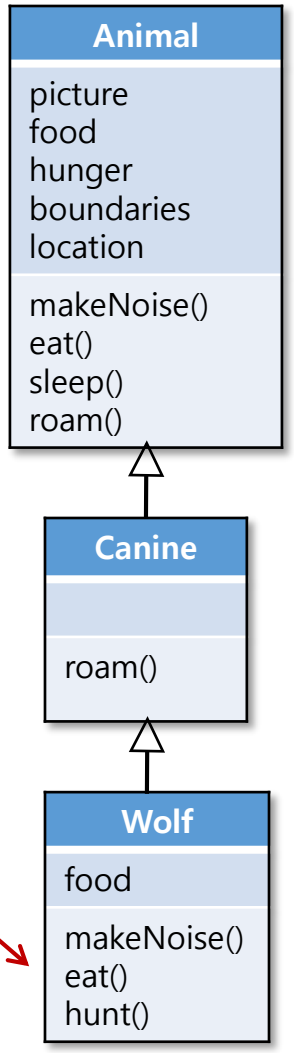
Which method is called?

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



Super

```
class Wolf extends Canine {
    ...
    void hunt() {
        ...
        super.eat();
        this.eat();
        ...
    }
}
```

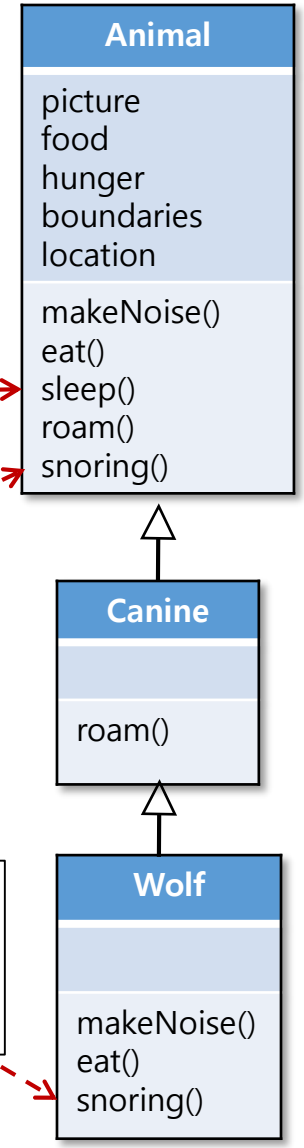


Which method is called?

- **Canine c = new Wolf();**
  - Make a new Wolf object
- **c.sleep();**
  - Calls the version in Animal.

```
public class Animal {
    public void sleep() {
        snoring();
        ...
    }
```

which one?

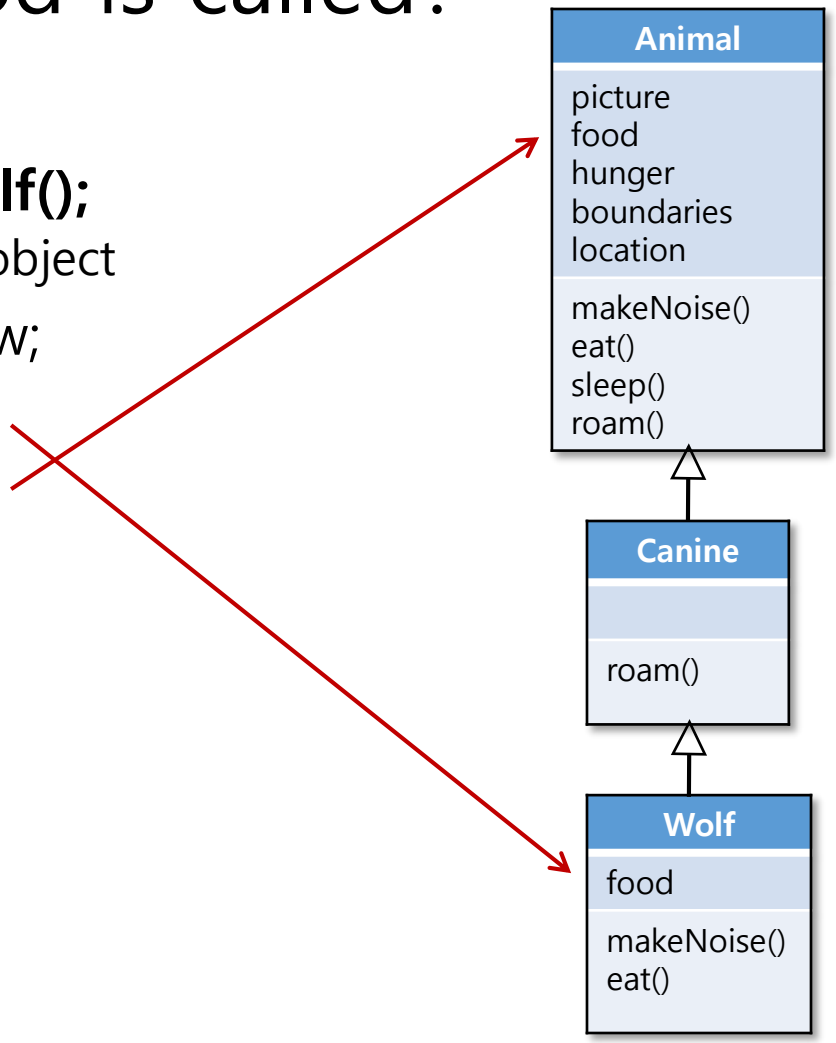


How to prevent snoring() from being overridden?

1. Make it **private** in Animal.
2. Make it **final** in Animal.

Which method is called?

- **Wolf w = new Wolf();**
  - Make a new Wolf object
- Animal a = (Wolf) w;
- w.food = “rabbit”;
- a.food = “squirrel”;
- w.eat();
- a.eat();



Example: ArrayList<E>

Object
#clone(): Object +equals(obj: Object): boolean #finalize(): +getClass(): Class<?> +hashCode(): int +notify(): +notifyAll(): +toString(): String +wait(): +wait(timeout: long): +wait(timeout: long, nanos: int):

```
public class Egg {
   private int age;
   public Egg(int a) {
      this.age = a;
   }
};

// ... Blah ... Blah...

ArrayList<Egg> eggList = new ArrayList<Egg>();
eggList.add(new Egg(5));
System.out.println(eggList.contains(new Egg(7)));    false
System.out.println(eggList.contains(new Egg(5)));    false

ArrayList<String> stringList = new ArrayList<String>();
stringList.add(new String("Alpha"));
System.out.println(stringList.contains(new String("Beta")));    false
System.out.println(stringList.contains(new String("Alpha")));   true
```

method: equals()

- Every class has the method equals() inherited from Object.
  - Object.equals() compares not the contents, but only the references.
- Class String has its own equals() comparing itself with other string object.

```
String a, b;
a.equals(b); //  a's text is compared with b's text.
```

- Class Egg does not have its own equals(), but uses the one inherited from Object.

```
Egg          a, b;
a.equals(b); // a's reference is compared with b's reference.
```

➔ Egg needs to have its own equals() to compare the contents.

Example - Override equals()

```
public class Egg {
   private int age;
   public Egg(int a) {
      this.age = a;
   }

   @Override // annotation
   public boolean equals(Object obj) {
      return (this.age == ((Egg)obj).age);
   }
};

// ... Blah ... Blah...

ArrayList<Egg> eggList = new ArrayList<Egg>();
eggList.add(new Egg(5));
System.out.println(eggList.contains(new Egg(7)));    false
System.out.println(eggList.contains(new Egg(5)));    true
```

IS-A and HAS-A

- IS-A
  - If class B extends class A, class B IS-A class A.
  - ex)

    A Rabbit is an Animal.

    ```
    public class Rabbit extends Animal { };
    ```
- HAS-A
  - If class B belongs to class A, class A HAS-A class B.
  - ex)

    A Rabbit has a Nose.

    ```
    public class Rabbit {
            Nose nose;
    };
    ```

Four Rules of Inheritance Design

1. Use inheritance when one class is a more specific type of a superclass.
2. Consider inheritance when you have behavior that should be shared among multiple classes of the same general type.
3. Do not use inheritance if the relationship between the superclass and subclass violate either of the above two rules.
4. Do not use inheritance if the subclass and superclass do not pass the IS-A test.

Q&A on Subclassing

- Are there any practical limits on the levels of subclassing?
  - No hard limit, but typically not so deep.
- Can you extend any class?
  - No private class.
  - Non-public class (without a ***public*** modifier) can be sub-classed only by classes in the same package.
  - A class with a ***final*** modifier, such as String, cannot be sub-classed.
  - A class with only private constructors cannot be sub-classed.
- Can you make only a method final?
  - It is possible.

Q&A on Subclassing

- Are there any practical limits on the levels of subclassing?
  - No hard limit,
- Can you extend a
  - No private clas
  - Non-public cla
    classed only b
  - A class with a
    classed.
  - A class with or
- Can you make only a method final?
  - It is possible.

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

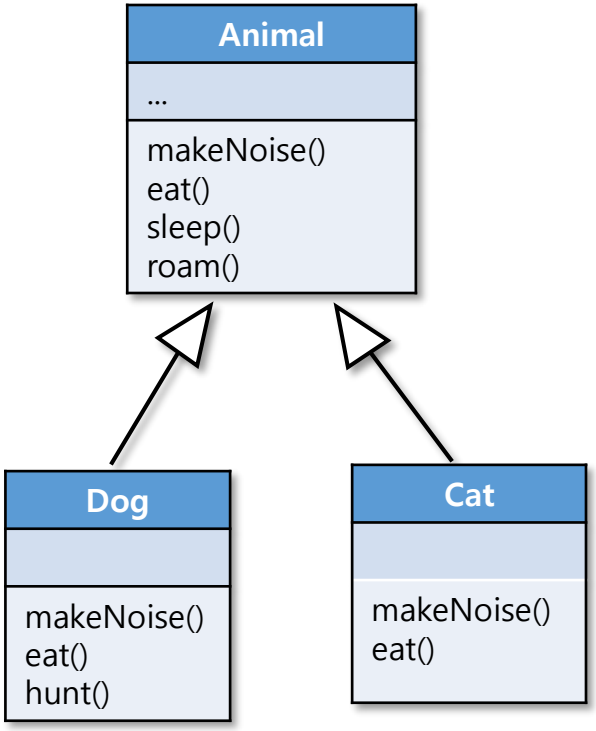
Polymorphism (1)

- Ability to create a variable, a function, or an object that has more than one form.
- Case that the reference type and the object type are **same**.

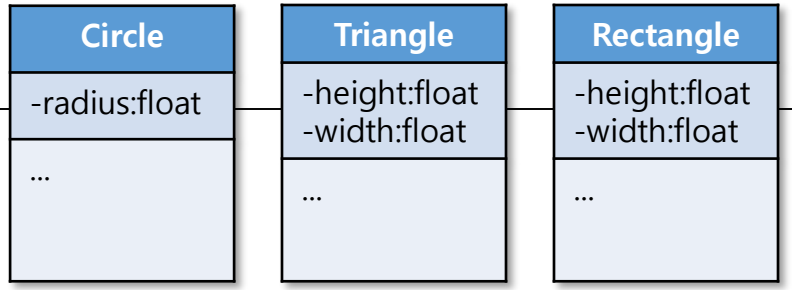
```
Dog terry = new Dog();
```

- With **polymorphism**,
  the reference and the object type
  can be **different**.

```
Animal terry = new Dog();
Animal tom = new Cat();
terry.makeNoise();
tom.makeNoise();
```

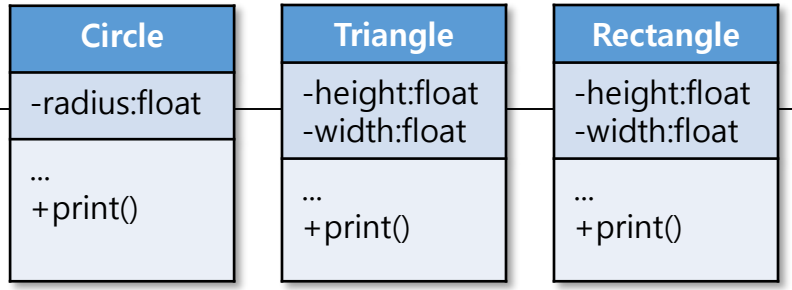


Polymorphism (1)



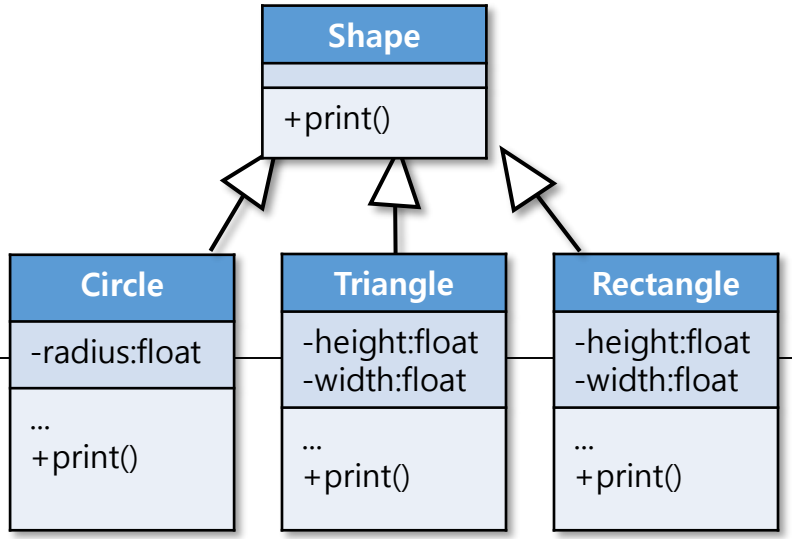

```
public class LinkedList {
    ...
   public void print() {
      System.out.println("List: "+mSize+" elements");
      for (Node node = mData; node != null; node = node.getNext()) {
         System.out.print("  (" + node.getKey()+",");
         Object obj = node.getObject();
         if (obj instanceof Triangle) {
            Triangle tri = (Triangle)obj;
            System.out.print("Triangle(width=" + tri.getWidth() + ",height="+tri.getHeight());
         } else if (obj instanceof Rectangle) {
            Rectangle rect = (Rectangle)obj;
            System.out.print("Rectangle(width=" + rect.getWidth() + ",height="+rect.getHeight());
         } else if (obj instanceof Circle) {
            Circle circ = (Circle)obj;
            System.out.print("Circle(radius=" + circ.getRadius());
         }
         System.out.println(")");
      }
   }
}
```

Polymorphism (1)



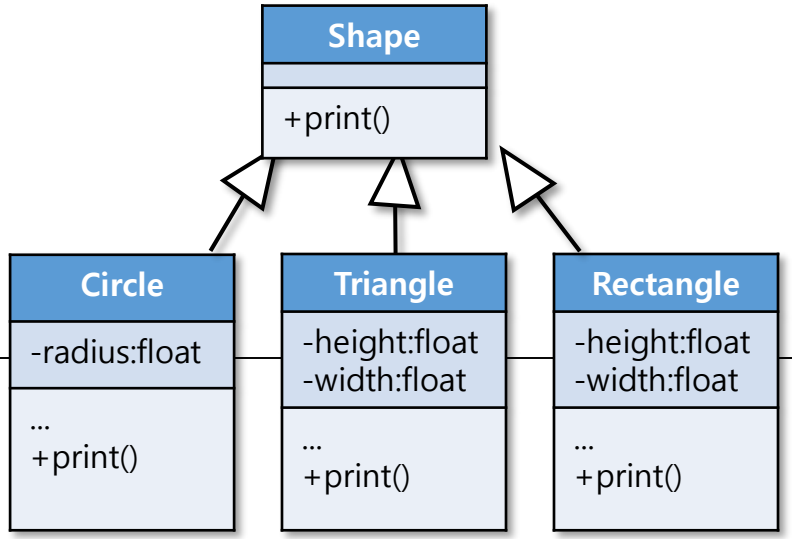

```
public class LinkedList {
    ...
   public void print() {
      System.out.println("List: "+mSize+" elements");
      for (Node node = mData; node != null; node = node.getNext()) {
         System.out.print("  (" + node.getKey()+",");
         Object obj = node.getObject();
         if (obj instanceof Triangle) {
            Triangle tri = (Triangle)obj;
            tri.print();
         } else if (obj instanceof Rectangle) {
            Rectangle rect = (Rectangle)obj;
            rect.print();
         } else if (obj instanceof Circle) {
            Circle circ = (Circle)obj;
            circ.print();
         }
         System.out.println(")");
      }
   }
}
```

Polymorphism (1)




```
public class LinkedList {
    ...
   public void print() {
      System.out.println("List: "+mSize+" elements");
      for (Node node = mData; node != null; node = node.getNext()) {
         System.out.print("  (" + node.getKey()+",");
         Object obj = node.getObject();
         if (obj instanceof Triangle) {
            Triangle tri = (Triangle)obj;
            tri.print();
         } else if (obj instanceof Rectangle) {
            Rectangle rect = (Rectangle)obj;
            rect.print();
         } else if (obj instanceof Circle) {
            Circle circ = (Circle)obj;
            circ.print();
         }
         System.out.println(")");
      }
   }
}
```

Polymorphism (1)




```
public class LinkedList {
    ...
   public void print() {
      System.out.println("List: "+mSize+" elements");
      for (Node node = mData; node != null; node = node.getNext()) {
         System.out.print("  (" + node.getKey()+",");
         Object obj = node.getObject();
         if (obj instanceof Shape      ) {
            Shape shape = (Shape)obj;
            shape.print();
         }
         System.out.println(")");
      }
   }
}
```

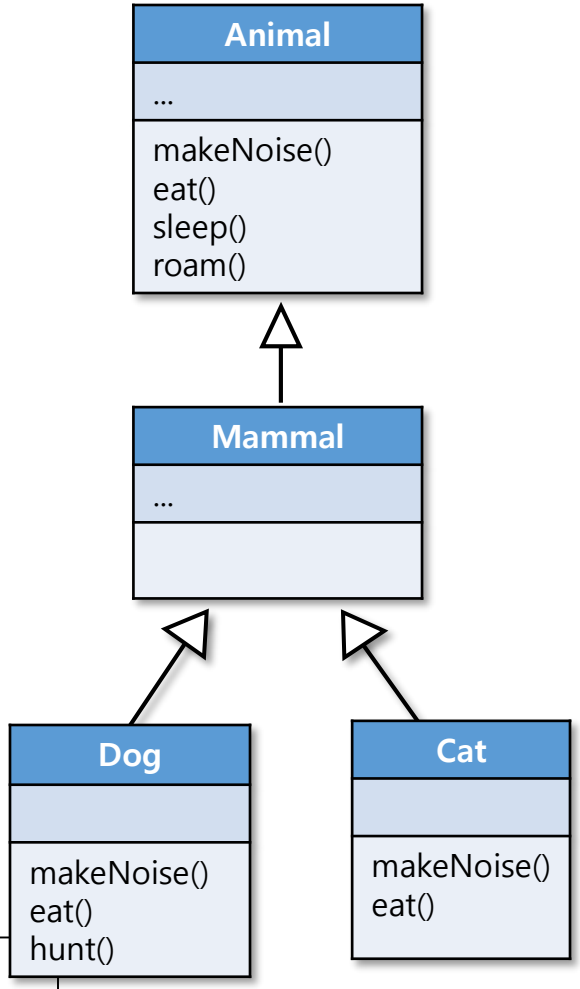
Polymorphism (2)

- With **polymorphism**,
  the reference and the object types
  can be **different**.

```
// The reference type should
// be the superclass or same
// class of the object type
Animal terry = new Dog();
Animal tom = new Cat();

Animal mammal = new Mammal();
```

```
Dog cannot = new Animal(); // ERROR
Cat notAllowed = new Mammal(); // ERROR
```

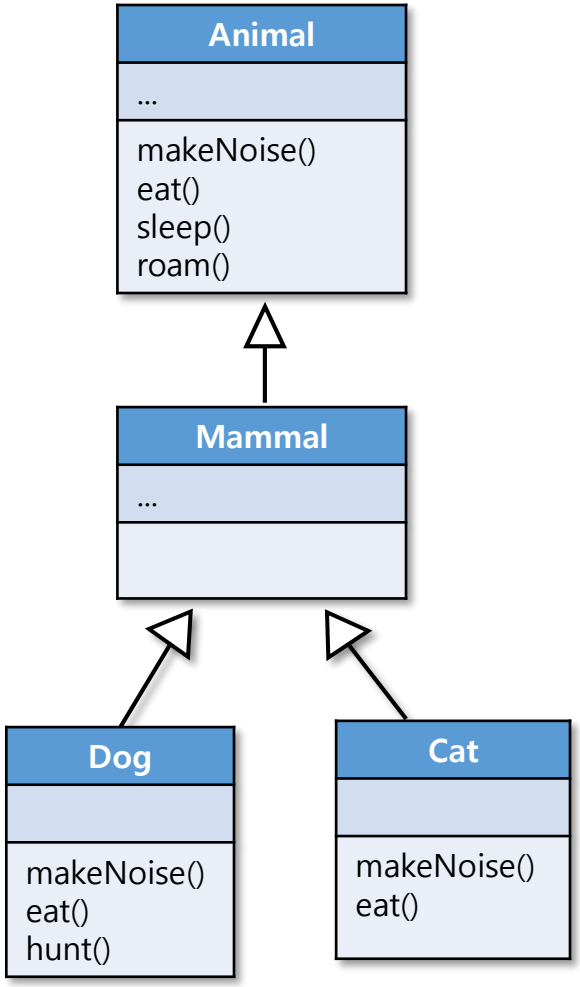


Polymorphism (2)

- With **polymorphism**,
  the reference and the object types
  can be **different**.

```
Animal terry = new Dog();
Animal tom = new Animal();

Dog dog;
dog = terry;  // ???
dog = (Dog)terry;  // ???
dog = (Dog)tom; // ???
```

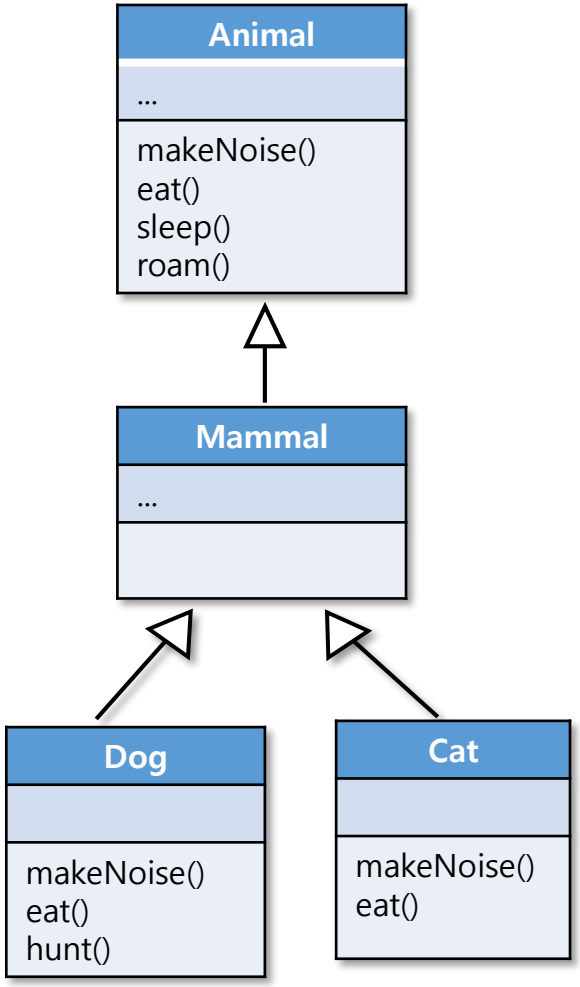


Polymorphism (3)

- You can have polymorphic arguments and return types.

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



Polymorphism (4): Example

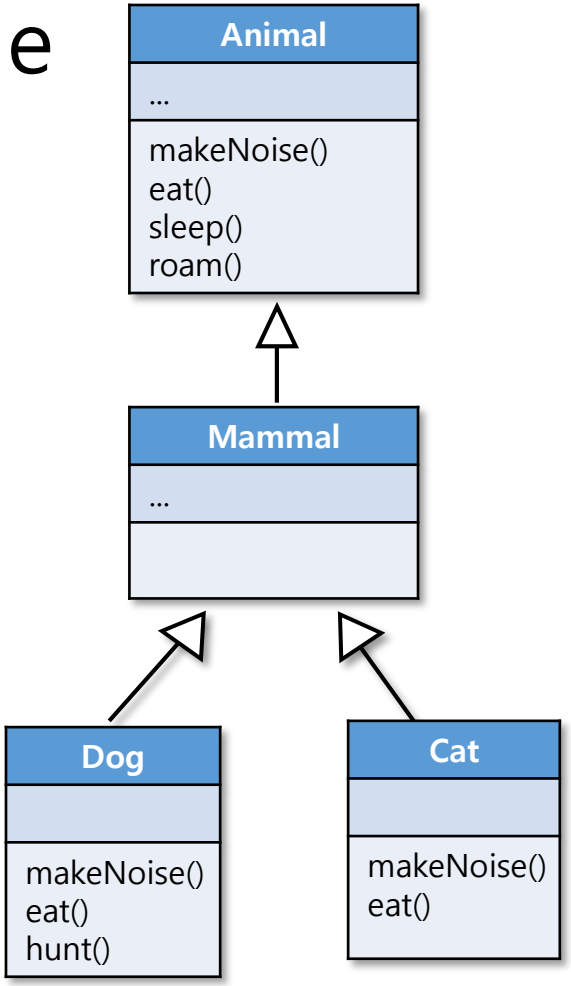
- Java Code

```
public class Animal {
   void makeNoise() { .. }
   void eat() { ... }
   void sleep() { ... }
   void roam() { ... }
}
```

```
public class Mammal extends Animal {
   ...
}
```

```
public class Dog extends Mammal {
   ...
}
```

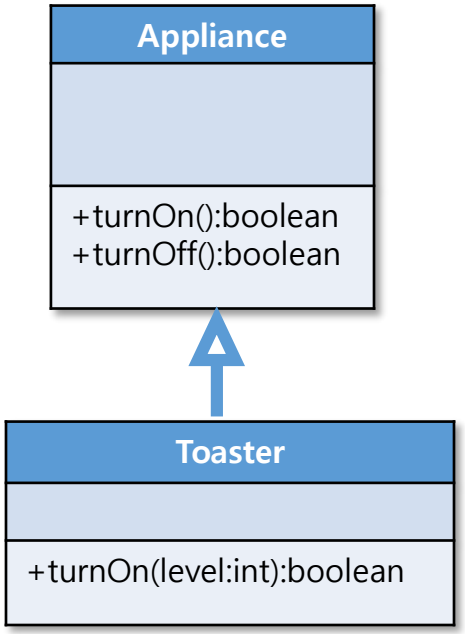
```
public class Cat extends Mammal {
   ...
}
```



Rules for Overriding

- The argument types and numbers must be same, and the return types must be comp atible.

- The method can't be less accessible.



Not overriding

Overloading

- Method overloading is a feature to allow more than one met hod with the same name but different arguments.

```
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;
   }
   public void setUniqueID(String theID) {
      this.uniqueID = theID;
   }
   public void setUniqueID(int ssNumber) {
      String numString = "" + ssNumber;
      setUniqueID(numString);
   }
}
```

Overloading

- Method overloading is a feature to allow more than one met hod with the same name but different arguments.

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


}
```

Overloading

- Method overloading is a feature to allow more than one met hod with the same name but different arguments.

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

Overloading: example

```
public class Canvas {
   public void draw(Image img) { // ...
   }
   public void draw(String text) { // ...
   }
   public void draw(Image img, int x, int y) { // ...
   }
   public void draw(Image img, Pos2 offset) { // ...
   }
}
```

Canvas
+draw(img: Image) +draw(text: String) +draw(img: Image, int x, int y) +draw(img: Image, Pos2 offset)

Rules for Overloading

- Arguments must be different.
- The return types can be different.
  - You can't change ONLY the return type.
- You can vary the access levels in any direction.

References

- Kathy Sierra and Bert Bates, *Head First Java*, O'Reilly, 2005.
- Java Platform, Standard Edition 7 API Specification
  - http://docs.oracle.com/javase/7/docs/api/

Q&A