中原大學
Chung Yuan Christian University

Chapter 7

inheritance and polymorphism

- What if you could write code that someone else could extend easily?
- If you could write code that was flexible, for those last-minute spec changes, would that be something you're interested in?
- *Inheritance* and *polymorphism* give you the design freedom and programming flexibility you deserve.

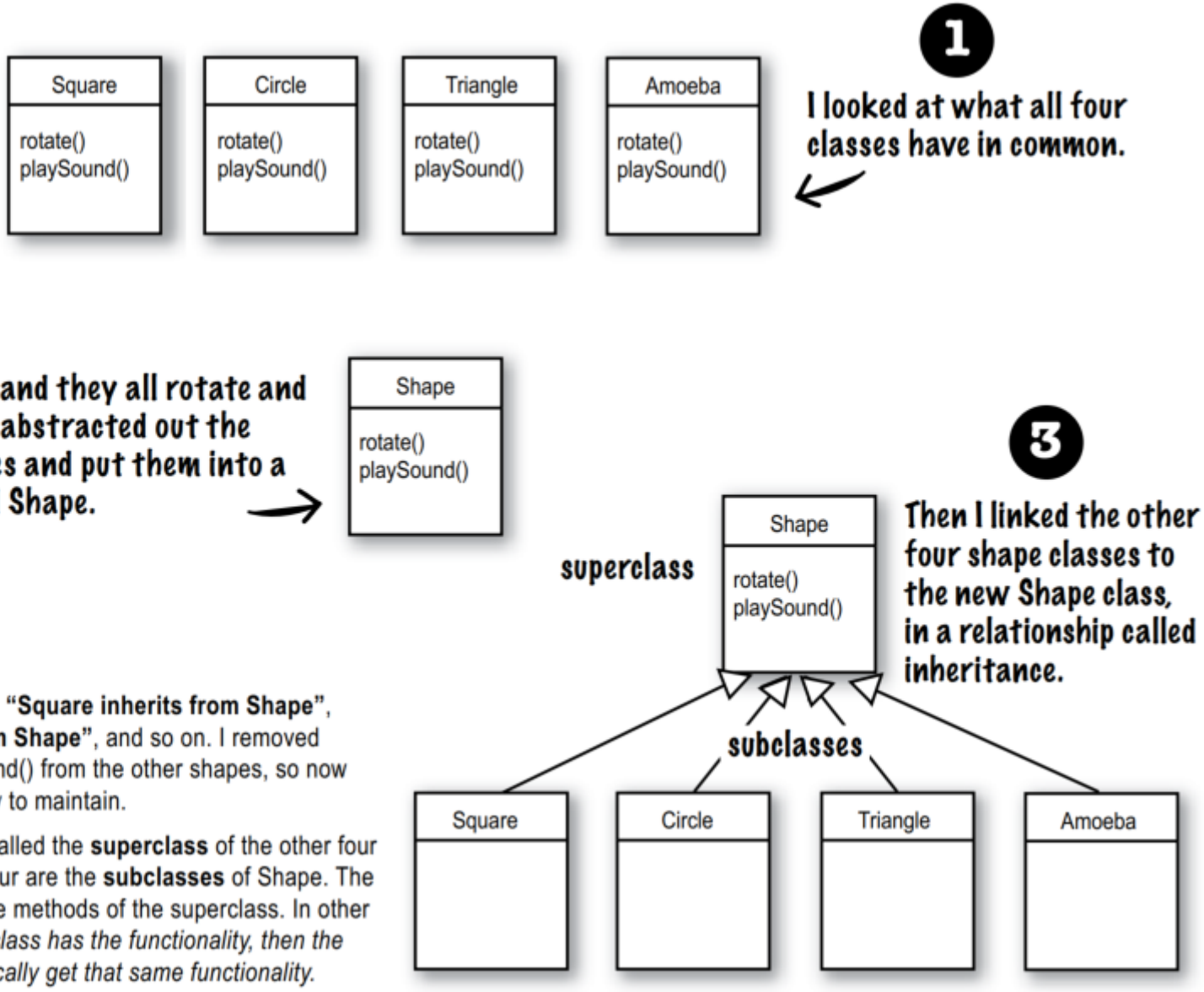
- Recall: the chair war story in chapter 2

LARRY: You've got duplicated code! The rotate procedure is in all four Shape things.

BRAD: It's a ***method***, not a procedure. And they're ***classes***, not things.

LARRY: Whatever. It's a stupid design. You have to maintain four different rotate "methods". How can that ever be good?

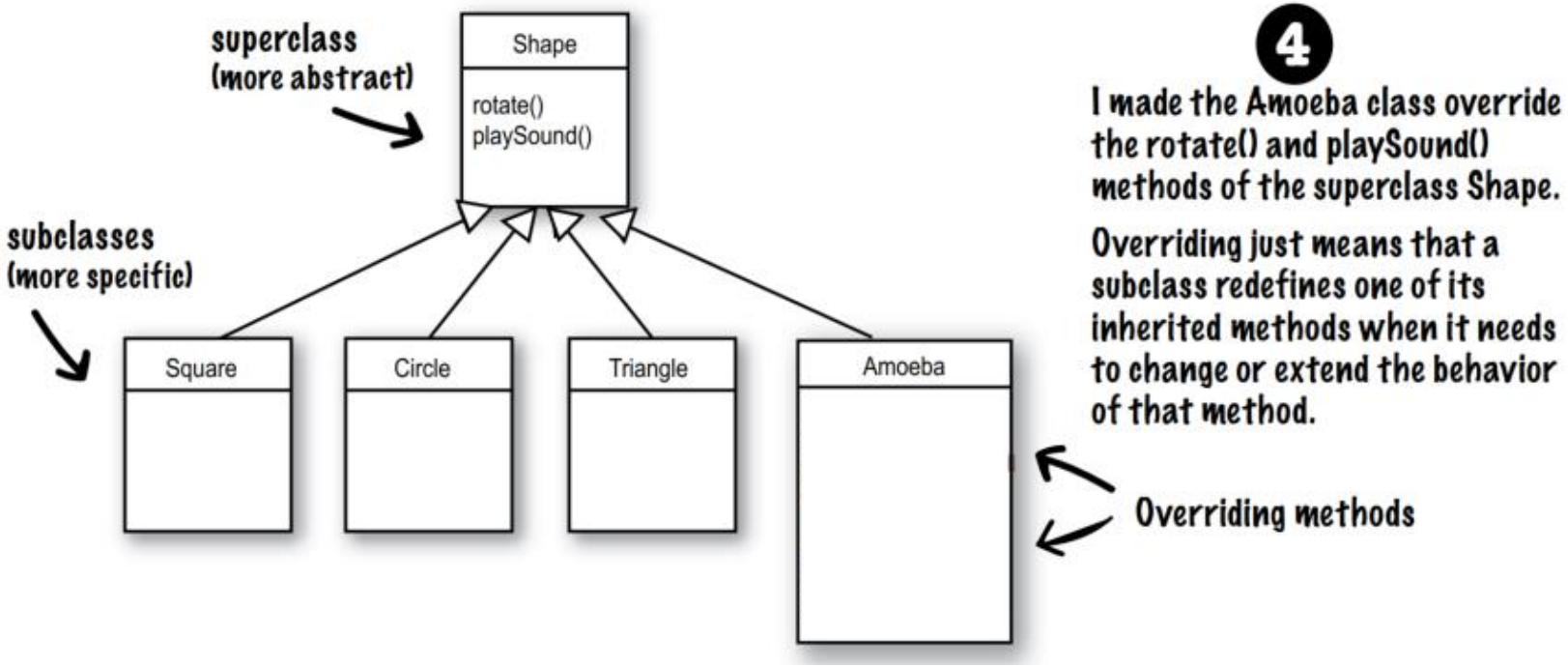
BRAD: Oh, I guess you didn't see the final design. Let me show you how OO **inheritance*** works, Larry



You can read this as, **"Square inherits from Shape"**, **"Circle inherits from Shape"**, and so on. I removed rotate() and playSound() from the other shapes, so now there's only one copy to maintain.

The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

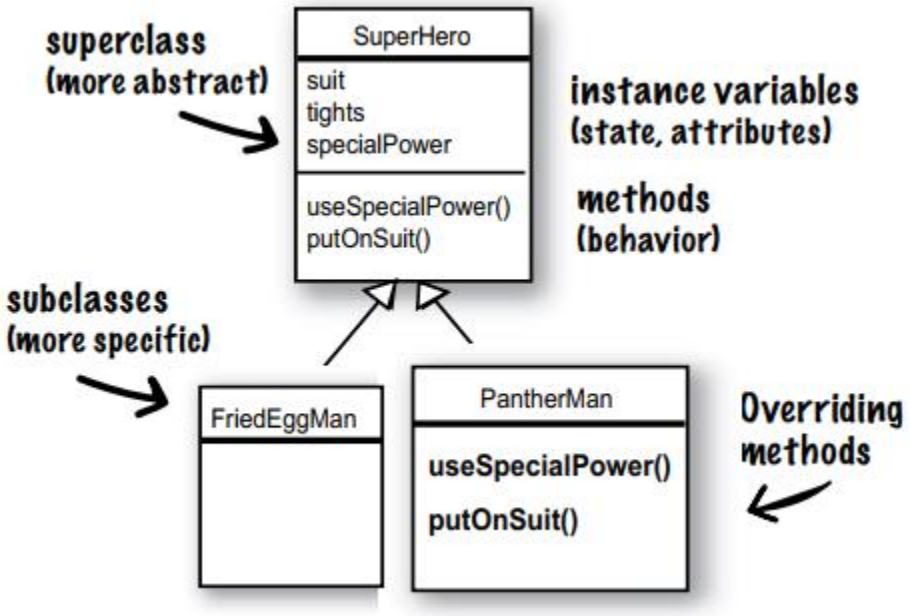
- What about the Amoeba rotate() and playSound()?
- How can amoeba do something different if it “inherits” its functionality from the Shape class?
- The Amoeba class ***overrides*** the methods of the Shape class. Then at runtime, the JVM knows exactly which rotate() method to run when someone tells the Amoeba to rotate



- Quiz
- How would you represent a house cat and a tiger, in an inheritance structure?
- Is a domestic cat a specialized version of a tiger? Which would be the subclass and which would be the superclass? Or are they both subclasses to some other class?
- How would you design an inheritance structure?

- Understanding Inheritance
- When you design with inheritance, you put ***common*** code in a class and then tell other more specific classes that the common (more abstract) class is their superclass. (This process is also called ***abstraction***)
- When one class inherits from another, the subclass inherits from the superclass.
- In Java, we say that the subclass ***extends*** the superclass.
- An inheritance relationship means that the subclass inherits the members of the superclass (member means instance variables and methods).

- For example, if PantherMan is a subclass of SuperHero, the PantherMan class automatically inherits the instance variables and methods common to all superheroes including suit, tights, specialPower, useSpecialPowers(), and so on.
- But the PantherMan subclass can add new methods and instance variables of its own, and it can override the methods it inherits from the superclass SuperHero.



- FriedEggMan doesn't need any behavior that's unique, so he doesn't override any methods. The methods and instance variables in SuperHero are sufficient.
- PantherMan, though, has specific requirements for his suit and special powers, so useSpecialPower() and putOnSuit() are both overridden in the PantherMan class
- Quiz: we can override methods, why we don't override instance variables?
- A: Instance variables are not overridden because they don't need to be. They don't define any special behavior, so a subclass can give an inherited instance variable any value it chooses.

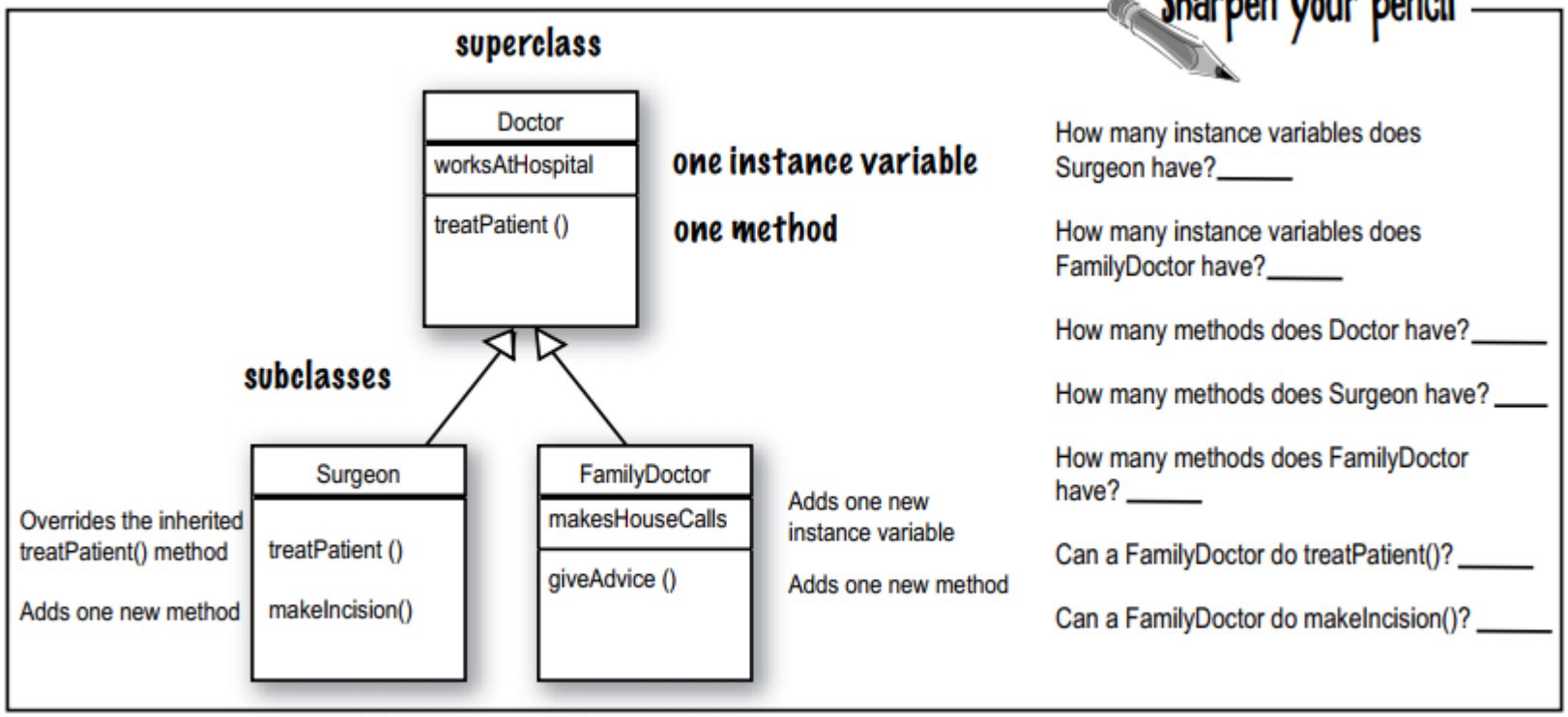
- An inheritance example:

```
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() {
      // perform surgery
   }

   void makeIncision() {
      // make incision (yikes!)
   }
}
```

Sharpen your pencil



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?____

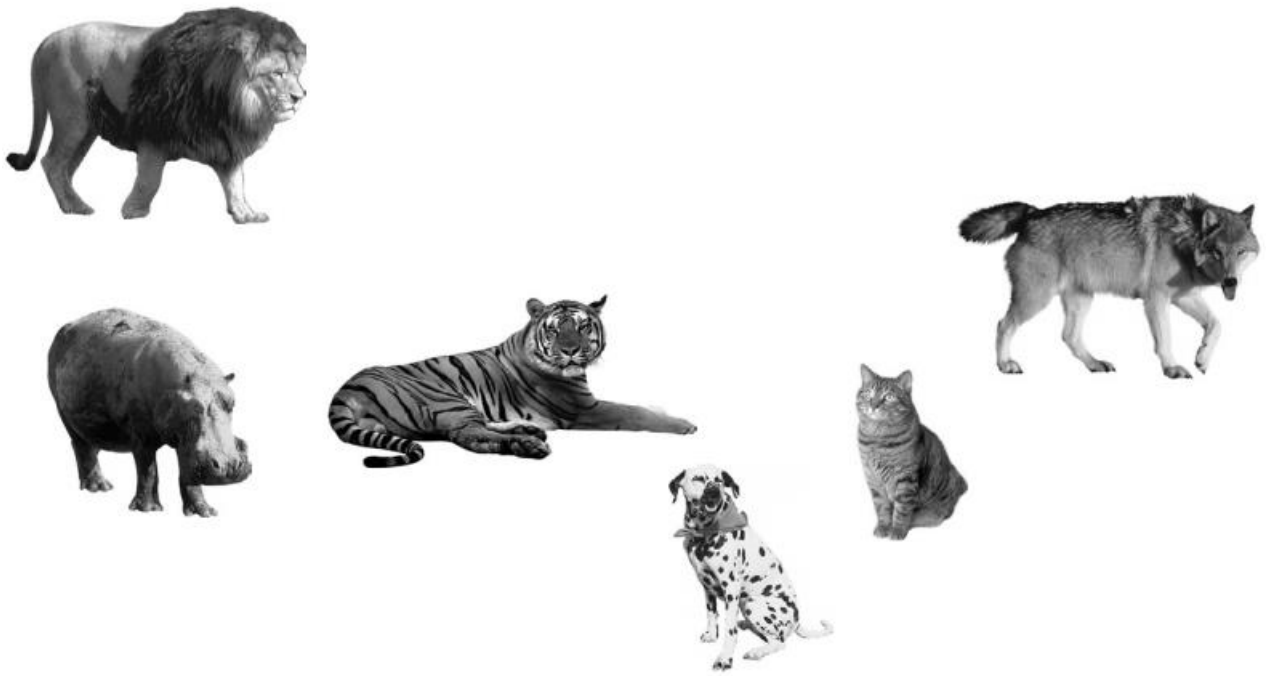
How many methods does FamilyDoctor have? _____

Can a FamilyDoctor do treatPatient()? _____

Can a FamilyDoctor do makeIncision()? _____

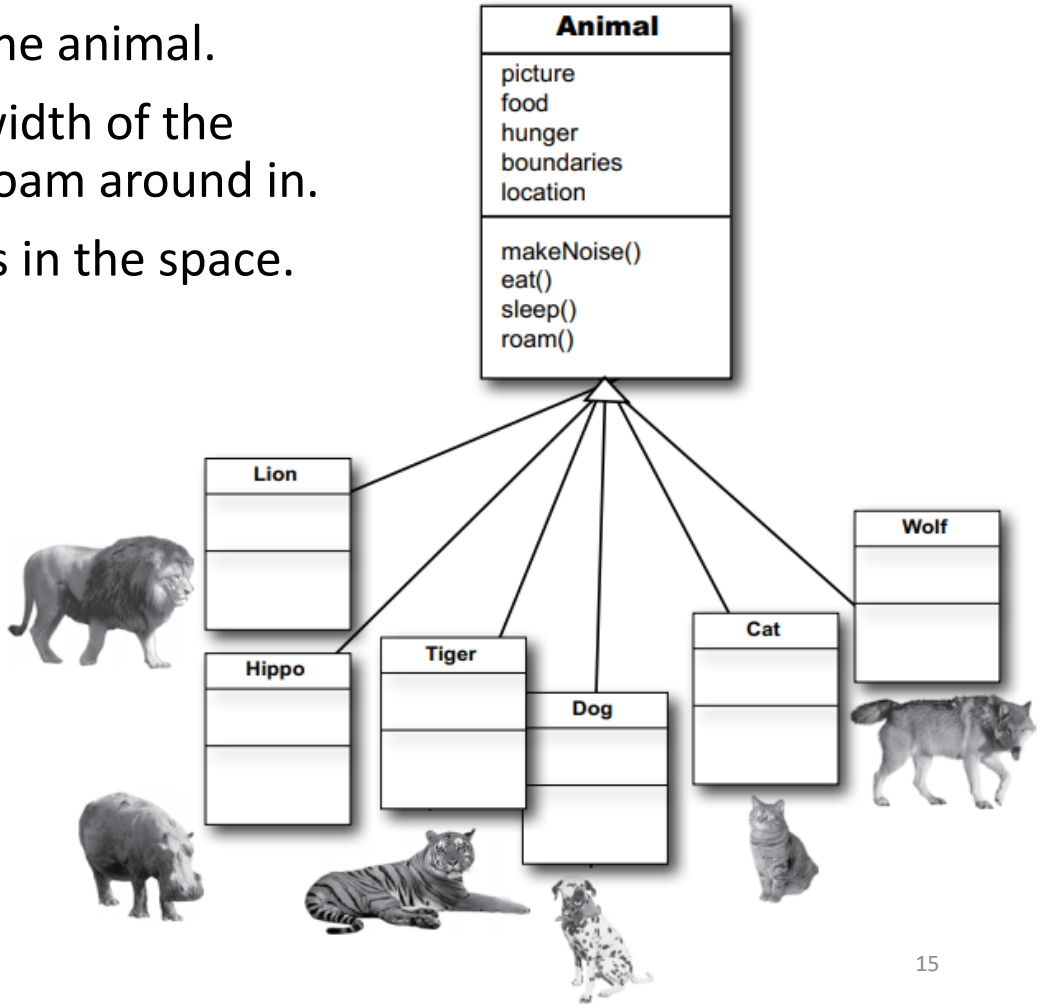
- Example
- Imagine you're asked to design a zoo simulation game that lets the user throw a bunch of different animals into an environment to see what happens.
- We've been given a list of some of the animals: lions, tigers, hippos, wolves that will be in the program, but not all.
- We want other programmers to be able to ***add new kinds of animals to the program at any time***.

- The starting point: look for objects that have common attributes and behaviors.

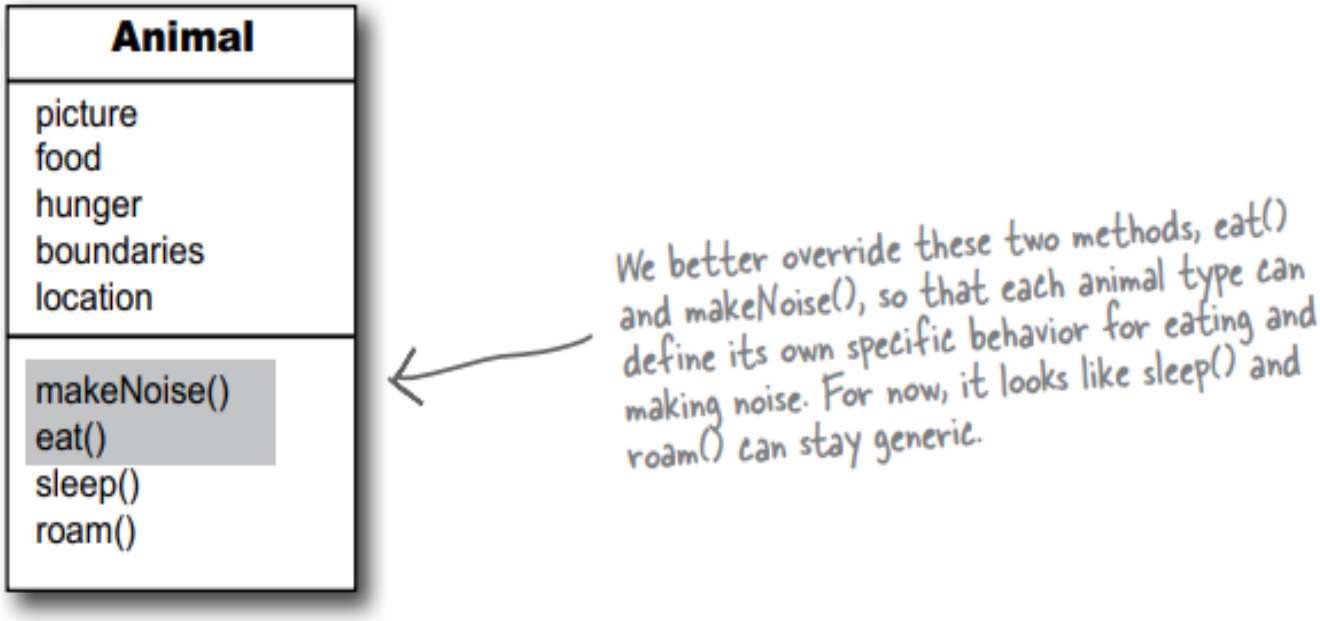


- First: figure out the common, abstract characteristics that all animals have, and build those characteristics into a class that all animal classes can extend.
- Second: design a class that represents the common state and behavior
- Third: decide if a subclass needs behaviors that are specific to that particular subclass type
- Fourth: Look for more opportunities to use abstraction, by finding ***two or more subclasses*** that might need common behavior
- Fifth: finish the class hierarchy

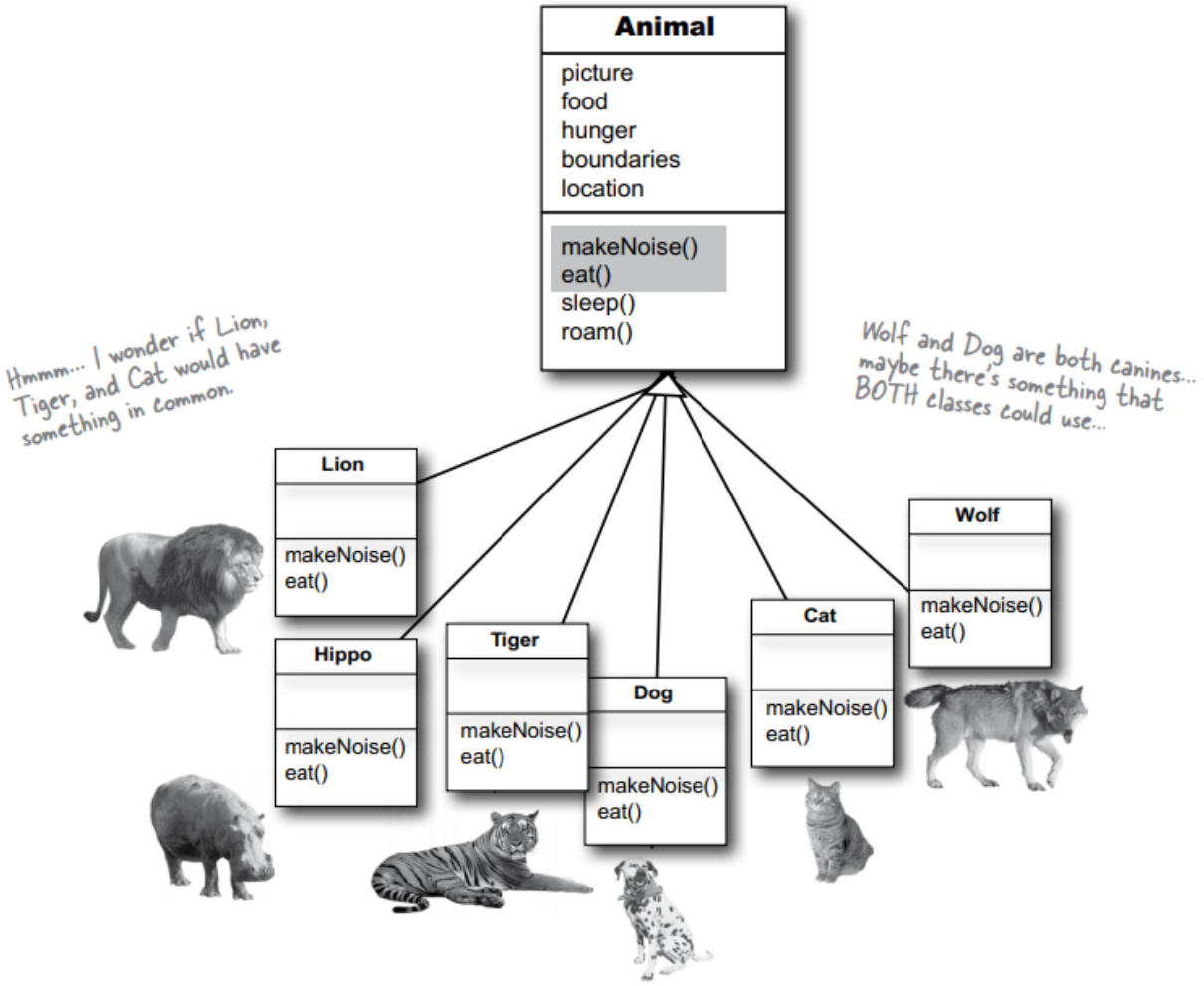
- Instance variables:
- picture – the file name
- food – the type of food this animal eats.
- hunger –the hunger level of the animal.
- boundaries –the height and width of the ‘space’ that the animals will roam around in.
- location – where the animal is in the space.
- Methods:
- makeNoise()
- eat()
- sleep()
- roam()



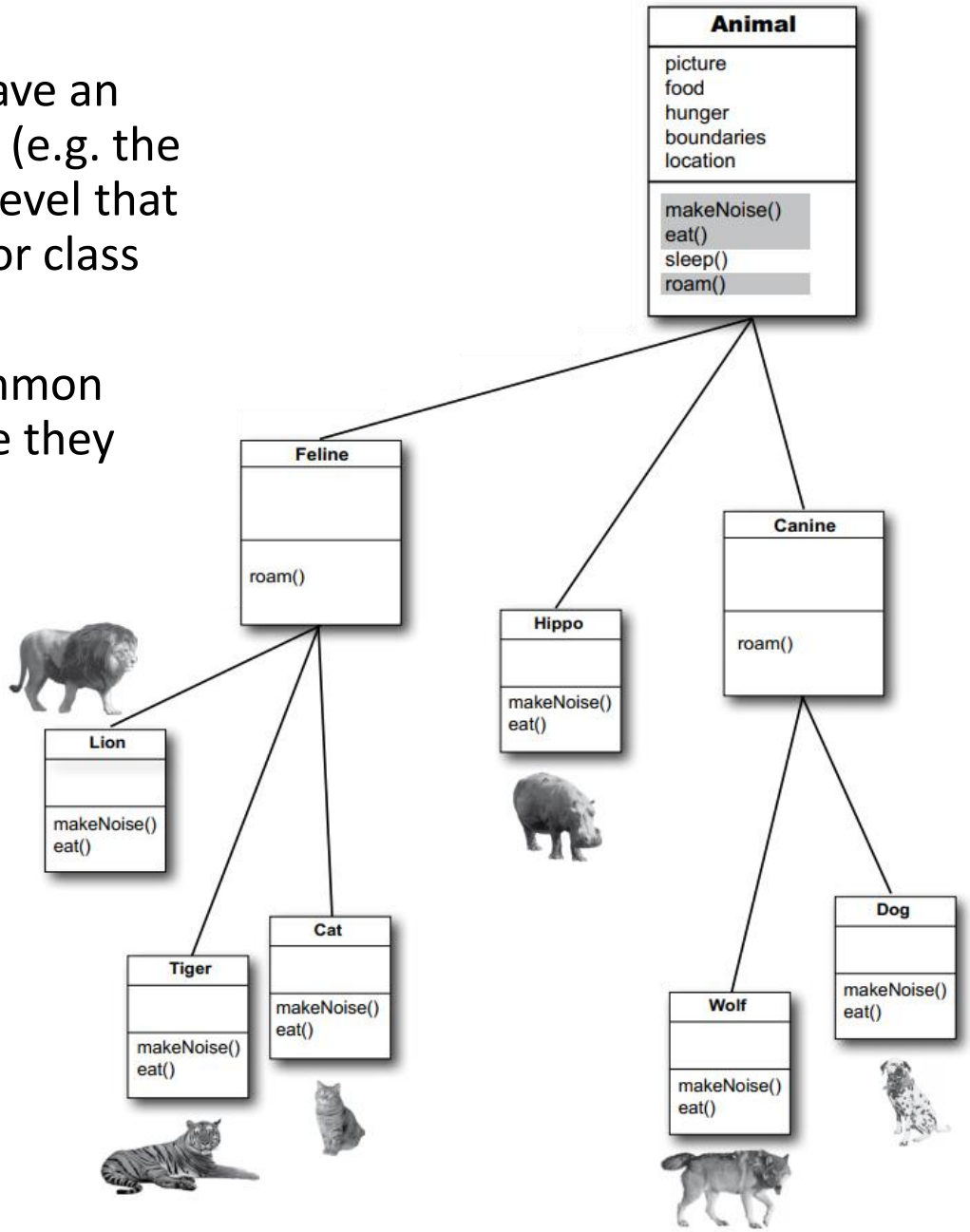
- Do all animals eat the same way?
- Does a lion make the same noise as a dog?



- We have to look at the subclasses of Animal, and see if two or more can be grouped together in some way, and given code that's common to only that new group.

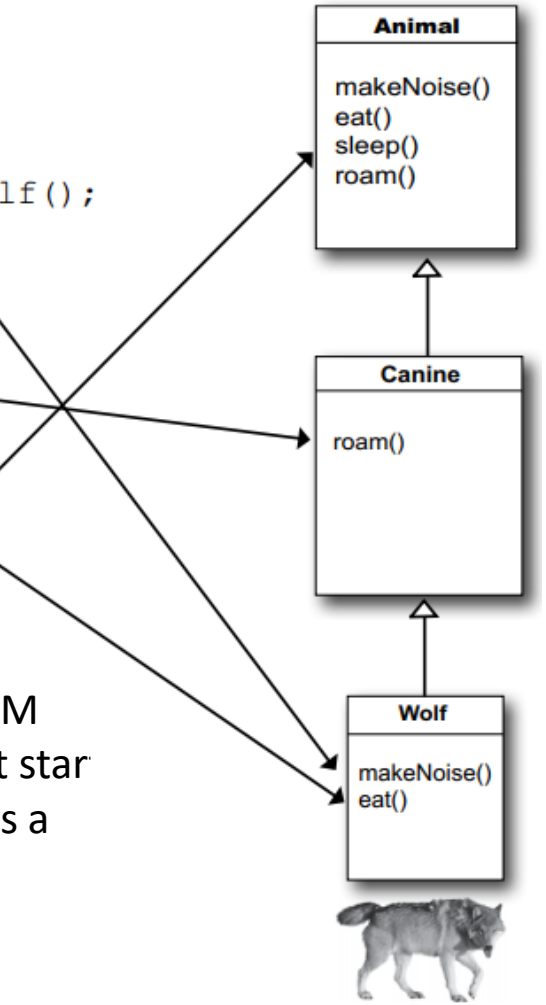


- Since animals already have an organizational hierarchy (e.g. the genus), we can use the level that makes the most sense for class design.
- Canines could use a common roam() method, because they tend to move in packs.



- For the Wolf class, which method is called?
- makeNoise(), eat() - overridden
- sleep() - inherited from Animal,
- roam() - inherited from Canine (which is actually an overridden version of a method in class Animal)

make a new Wolf object	`Wolf w = new Wolf();`
calls the version in Wolf	`w.makeNoise();`
calls the version in Canine	`w.roam();`
calls the version in Wolf	`w.eat();`
calls the version in Animal	`w.sleep();`



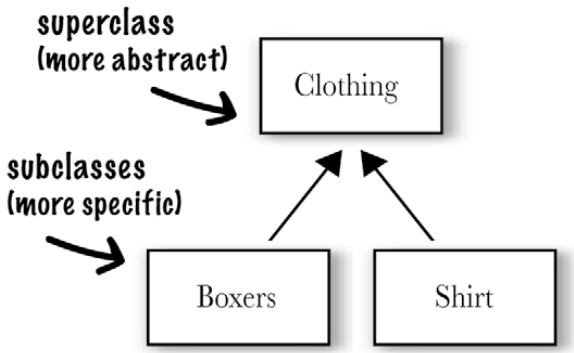
- The JVM starts looking first in the Wolf class. If the JVM doesn't find a version of the method the Wolf class, it star walking back up the inheritance hierarchy until it finds a match.

- What happens if the JVM doesn’t ever find a match in the inheritance tree?
- Remember that if a class inherits a method, it has the method. The complier will complain if it can’t find that particular method.

Designing an Inheritance Tree

Class	Superclasses	Subclasses
Clothing	---	Boxers, Shirt
Boxers	Clothing	
Shirt	Clothing	

Inheritance Table



Inheritance Class Diagram

Sharpen your pencil

Find the relationships that make sense. Fill in the last two columns

Class	Superclasses	Subclasses
Musician		
Rock Star		
Fan		
Bass Player		
Concert Pianist		

Hint: not everything can be connected to something else.
Hint: you're allowed to add to or change the classes listed.

- When one class inherits from another, we say that the subclass ***extends*** the superclass.
- When you want to know if one thing should extend another, apply the IS-A test.
- To know if you've designed your types correctly, ask, "Does it make sense to say type X IS-A type Y?" If it doesn't, you know there's something wrong with the design.
- Triangle IS-A Shape → O
- Cat IS-A Feline → O
- Surgeon IS-A Doctor → O

- What about the tub and bathroom?
- Tub IS-A bathroom →×
- Bathroom IS-A Tub →×
- Tub and Bathroom are related, but not through inheritance.
- Bathroom HAS-A Tub →○
- Bathroom has an instance variable of Tub class
- We'll discuss more about the relationship when we introduce UML.

- The IS-A test works anywhere in the inheritance tree. If your inheritance tree is well-designed, the IS-A test should make sense when you ask any subclass if it IS-A any of its supertypes.

Canine extends Animal

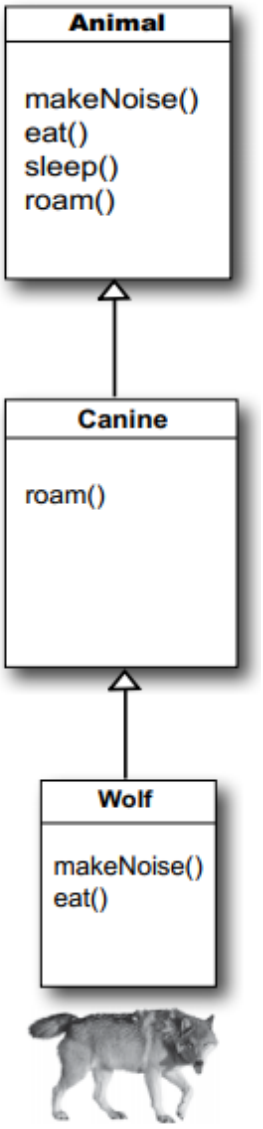
Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



- How do you know if you've got your inheritance right?
- Use the IS-A test
- the IS-A relationship implies that if X IS-A Y, then X can do anything a Y can do
- The inheritance IS-A relationship works in only ***one direction*** – the reverse will make no sense

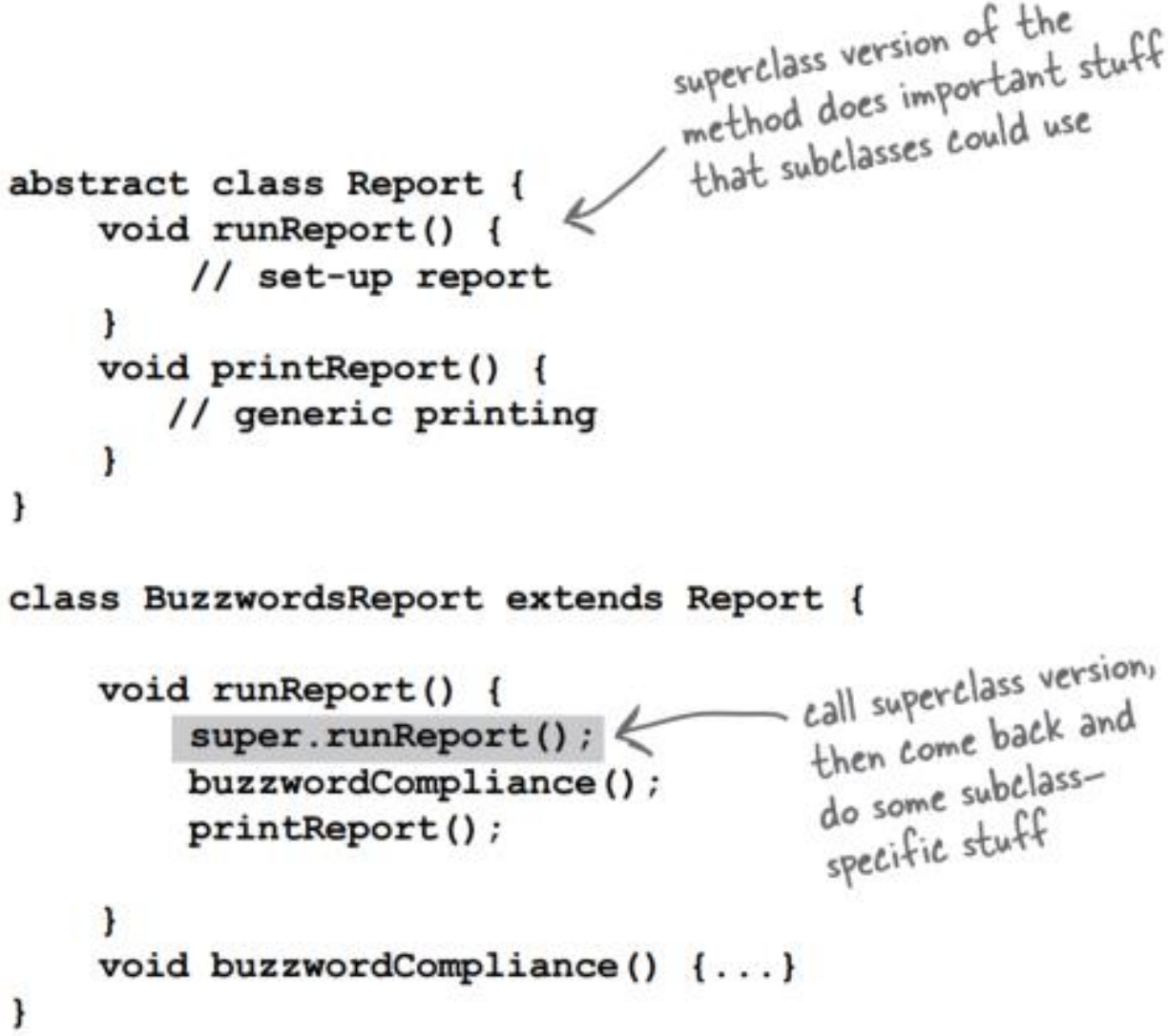
- Exercise
- Put a check next to the relationships that make sense.

☐ Oven extends Kitchen
☐ Guitar extends instrument
☐ Person extends Employee
☐ Ferrari extends Engine
☐ Fried-egg extends Food
☐ Beagle(小獵犬) extends Pet
☐ Container extends Jar
☐ Metal extends Titanium
☐ Beverage extends Bubble tea

- Q: We see how a subclass gets to inherit a superclass method, but what if the superclass wants to use the ***subclass version*** of the method?
- A: a superclass won't necessarily know about any of its subclasses. If you have to do so, it points to a possible design problem.
- Flaw 1: the method is in the wrong place (maybe it should be put in the superclass?)
- Flaw 2: the code which calls the method is in the wrong place. (should not be called in the superclass)
- Flaw 3: the subclass should not extend the superclass. (maybe use composition is a better idea)

- Q: Can I use part of the behavior of my superclass? I don't want to completely replace the superclass version, I just want to add more stuff to it.
- A: Yes you can! And it's an important design feature. Think about the keyword extends. "I want to extend the functionality of my superclass."
- You can design your superclass methods in such a way that they contain method implementations that will work for any subclass. In your subclass overriding method, you can call the superclass version using the keyword ***super***.

```
public void roam(){
      super.roam();
}
```

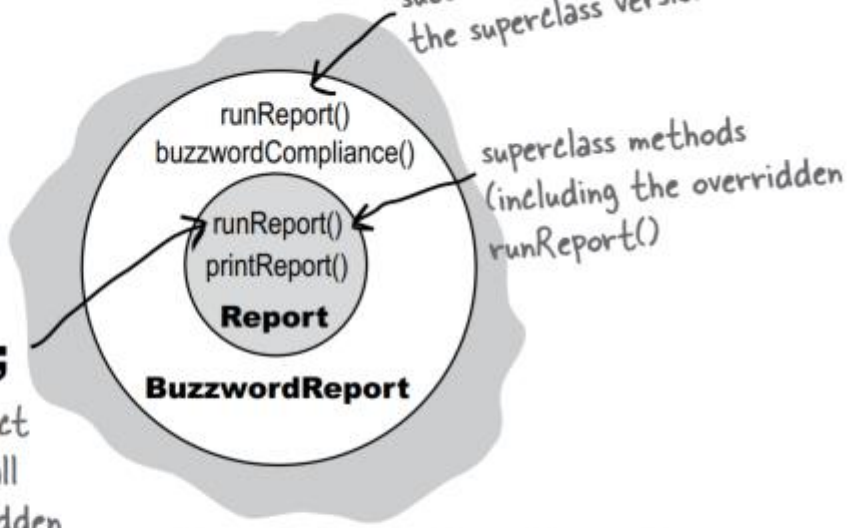



```
abstract class Report {
    void runReport() {
        // set-up report
    }
    void printReport() {
       // generic printing
    }
}

class BuzzwordsReport extends Report {

    void runReport() {
        super.runReport();
        buzzwordCompliance();
        printReport();

    }
    void buzzwordCompliance() {...}
}
```

If method code inside a BuzzwordReport subclass says:

```
super.runReport();
```

the runReport() method inside the superclass Report will run



super.runReport();

A reference to the subclass object (BuzzwordReport) will always call the subclass version of an overridden method. That's polymorphism. But the subclass code can call super.runReport() to invoke the superclass version.

The super keyword is really a reference to the superclass portion of an object. When subclass code uses super, as in super.runReport(), the superclass version of the method will run.

- Quiz: Does it matter if you put the super statement in different places?

```
class Animal
{
    public void foo()
    {
        System.out.println("animal foo");
    }
}
class Dog extends Animal
{
    public void foo()
    {
        super.foo();
        System.out.println("dog foo");
    }
}
```

What if you put the super.foo() here?

- A superclass can choose whether or not it wants a subclass to inherit a particular member by the level of access the particular member is given
- Recall: there are four access levels: public, protected, default, private
- public/protected members are inherited
- default members are inherited within same package
- private members are not inherited
- Note: when you're designing a new class, usually we declare every instance variable as private and then change it to public/protected if needed.

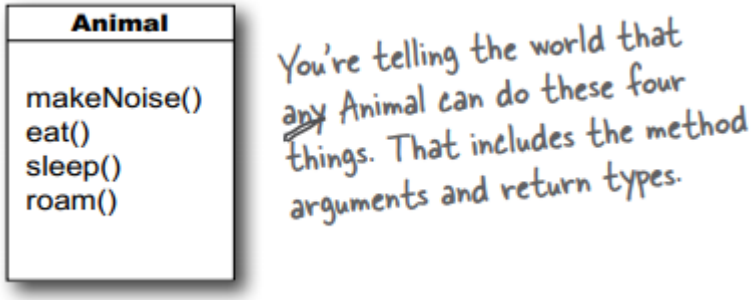
- Example
- Imaging that you're designing four different shapes: Circle, Rectangle, Triangle and Square.
- Now imagine that you have an instance variable named **centroid**. What will be your design? Which access level will you set for this instance variable?

- DO use inheritance when one class is a more specific type of a superclass. Example: Willow(柳樹) is a more specific type of Tree, so Willow extends Tree makes sense.
- DO consider inheritance when you have behavior (implemented code) that should be shared among multiple classes of the same general type.
- DO NOT use inheritance just so that you can reuse code from another class, if the relationship between the superclass and subclass violate either of the above two rules.
- DO NOT use inheritance if the subclass and superclass do not pass the IS-A test.

- Summary
- A subclass extends a superclass
- A subclass inherits all public instance variables and methods
- A subclass cannot inherits private instance variables and methods
- Inherited methods can be overridden. Inherited instance variables can be ***redefined***, not be overridden
- Use IS-A test to verify your inheritance logic
- The IS-A relationship works in only one direction
- When a method is overridden in a subclass, the lowest one wins.
- If B extends A and C extends B then C IS-A A

- You can get rid of duplicate code by abstracting out the behavior common to a group of classes, and sticking that code in a superclass.
- When you need to modify it, you have only one place to update, and the change is reflected in all the classes that inherit that behavior.

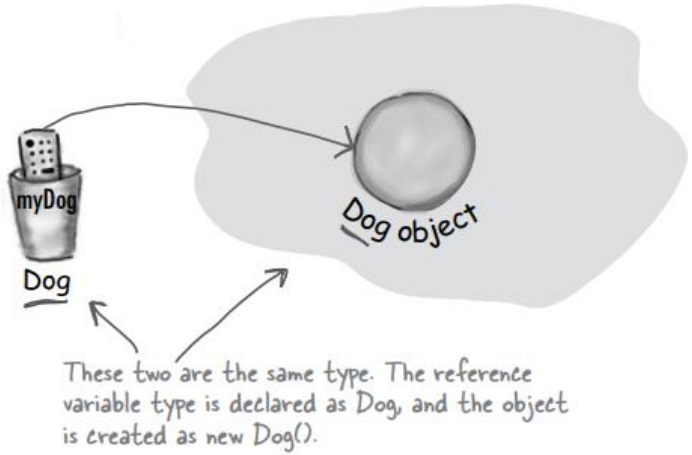
- Inheritance lets you guarantee that all classes grouped under a certain supertype have all the (inherited) methods that the supertype has
- In other words, you define a common **protocol** for a set of classes related through inheritance



- *Any subclass of that supertype can be substituted where the supertype is expected* – the polymorphism!

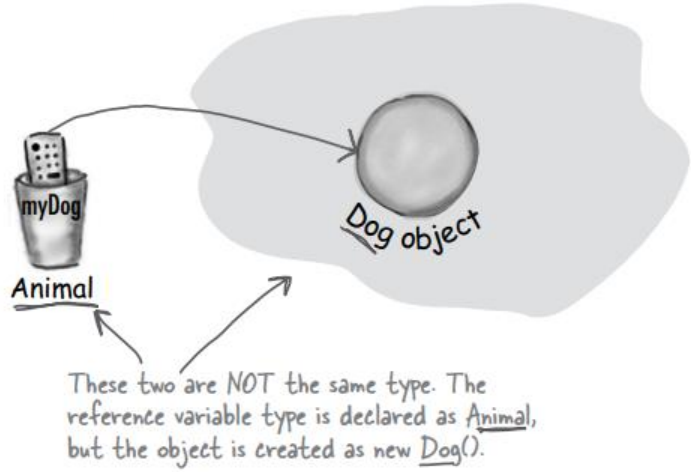
- Traditional way to declare an object

```
Dog d = new Dog();
```



- With polymorphism

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



- Anything that extends the declared reference variable type can be assigned to the reference variable.
- This lets you do things like make ***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();
}
```

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

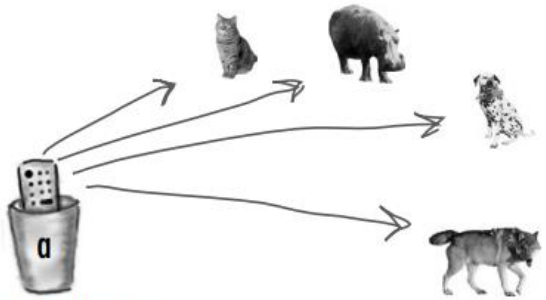
But look what you get to do... you can put ANY subclass of Animal in the Animal array!

And here's the best polymorphic part (the raison d'être for the whole example), you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

When 'i' is 0, a Dog is at index 0 in the array, so you get the Dog's eat() method. When 'i' is 1, you get the Cat's eat() method

Same with roam().

- There's more! 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();
   }
}
```

The 'a' parameter can take ANY Animal type as the argument. And when the Vet is done giving the shot, it tells the Animal to makeNoise(), and whatever Animal is really out there on the heap, that's whose makeNoise() method will run.

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

The Vet's giveShot() method can take any Animal you give it. As long as the object you pass in as the argument is a subclass of Animal, it will work.

Dog's makeNoise() runs

Hippo's makeNoise() runs

- What's the advantage of doing this?
- With polymorphism, you can write code that doesn't have to change when you introduce new subclass types into the program.
- You wrote a class CLA using arguments declared as a type ALC, your code can handle any ALC subclass, even though the CLA class was written without any knowledge of the new class extends from ALC.
- Quiz: why is polymorphism guaranteed to work this way? Why is it always safe to assume that any subclass type will have the methods you think you're calling on the superclass type?

- Quiz: Can this code work?

```
package test;

class Animal
{
    void animalFoo()
    {
        System.out.println("animalFoo");
    }
}

class Dog extends Animal
{
    void dogFoo()
    {
        System.out.println("dogFoo");
    }
}

public class test1
{
    public static void main (String args[])
    {
        Animal a = new Dog();
        a.dogFoo();
        System.out.println("finish");
    }
}
```

- Extension
- Upcasting has no problems, but be careful if you want to do downcasting

```
class Animal{}
class Dog extends Animal{}
class Cat extends Animal{}

public class test1
{

    public static void main(String[] args)
    {
        Animal a1 = new Animal();
        Animal a2 = new Dog();
        Animal a3 = new Cat();
        Dog a4 = (Dog)a1;  // runtime crash!
        Cat a5 = (Cat)a3;  // pass!
    }
}
```

- Are there any practical limits on the levels of subclassing? How deep can you go?
- In Java API, most inheritance hierarchies are wide but not deep.it usually makes more sense to keep your inheritance trees shallow, but there isn't a hard limit
- Can you extend any class? Or you can set access control to a particular class?
- There's no such thing as a private class. However, you have three ways to prevent a class be subclassed (by unknown class):
- A class can be non-public. A non-public class can be subclassed only by classes in the same package as the class.
- Second, use the keyword modifier final. A final class means that it's the end of the inheritance line. Nobody, ever, can extend a final class
- Third a class which has only private constructors (we'll look at constructors in chapter 9)

- What advantage would there be in preventing a class from being subclassed?
- If you want to ensure that the methods will always work the way that you wrote them (because they can't be overridden), a final class will give you that.
- A lot of classes in the Java API are final for that reason. (For example, the String class)
- Can you make a method final, without making the whole class final?
- If you want to protect a specific method from being overridden, mark the method with the final modifier. Mark the whole class as final if you want to guarantee that none of the methods in that class will ever be overridden.

- Quiz: can this compile?

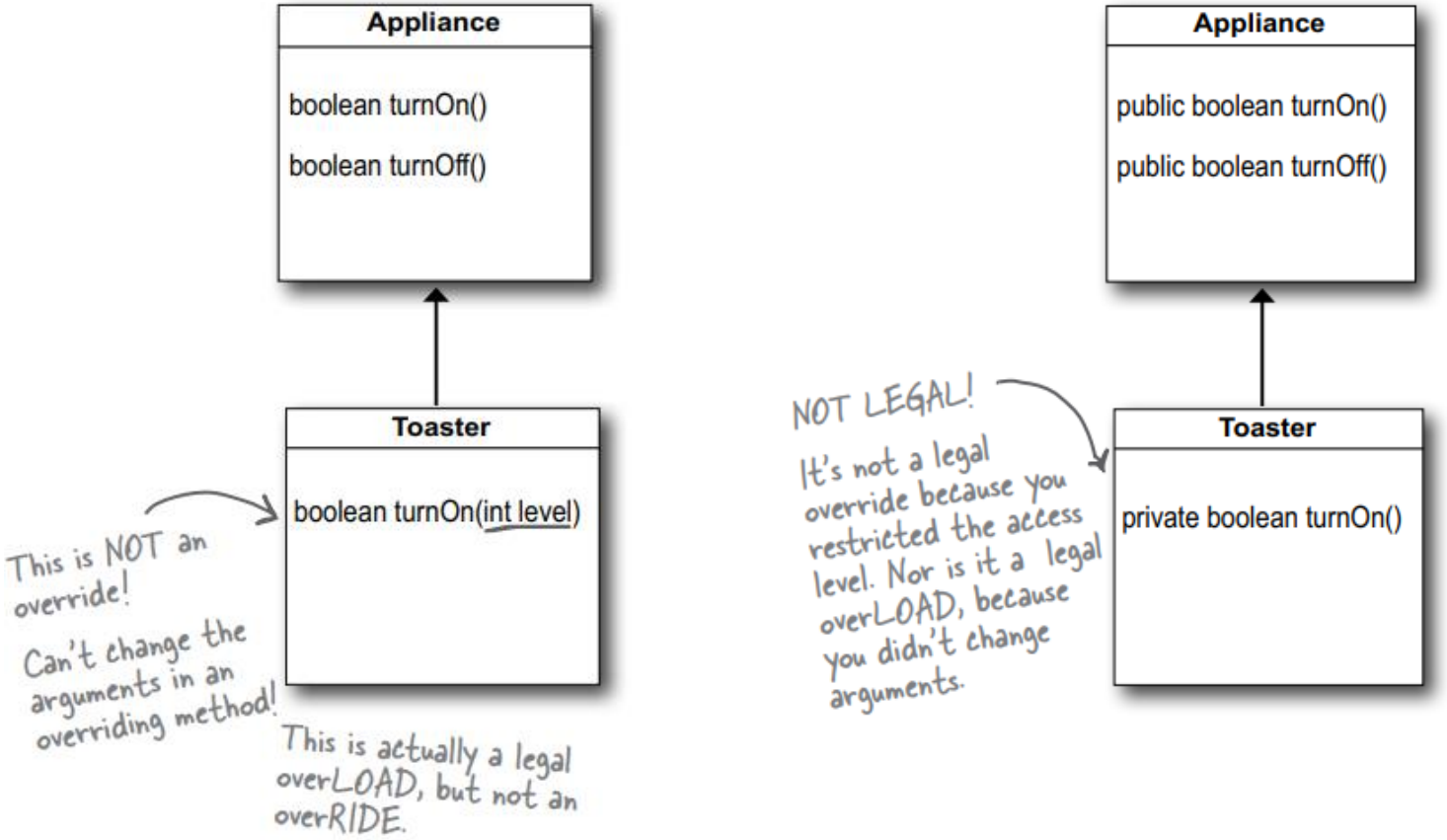
```
class Animal
{
    public int x=3;

    public int getX(){return x;}
}

class Dog extends Animal
{
    public int x=8;

    public int getX(){return x;}
}


public class test1
{

    public static void main(String[] args)
    {
        Animal a = new Dog();
        System.out.println(a.x);
        System.out.println(a.getX());

    }
}
```

- Variable overriding might break methods inherited from the parent if we change its type in the child class!

```
class Parent {
    int x;
    public int increment() {
        return ++x;
    }
    public int getX() {
        return x;
    }
}

class Child extends Parent {
    Object x;
    // Child is inherting increment(), getX() from Parent and both methods returns an int
    // But in child class type of x is Object, so increment(), getX() will fail to compil
e.
}
```

- Rules for overriding
- The arguments and return types of your overriding method must look to the outside world exactly like the overridden method in the superclass.
- If you change the arguments or return type, it is not an override anymore. (In fact, it is called overLOAD)

- Arguments must be the same, and return types must be compatible.
- The method can't be less accessible.



- Overloading a method
- An overloaded method is just a different method that happens to have the same method name.
- An overloaded method is NOT the same as an overridden method.
- Overloading lets you make multiple versions of a method, with different argument lists, for convenience to the callers.
- The return types can be different
- You can’t change ONLY the return type. (the compiler will assume you’re trying to override the method)
- You can vary the access levels in any direction.

- Legal example of method overloading

```
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) {
      // lots of validation code, and then:
      uniqueID = theID;
   }

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

the program:

```
class A {
  int ivar = 7;
  void m1() {
    System.out.print("A's m1, ");
  }
  void m2() {
    System.out.print("A's m2, ");
  }
  void m3() {
    System.out.print("A's m3, ");
  }
}

class B extends A {
  void  m1() {
    System.out.print("B's m1, ");
  }
}

class C extends B {
  void m3() {
    System.out.print("C's m3, "+(ivar + 6));
  }
}

public class Mixed2 {
  public static void main(String [] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();
    [                ]   <-- candidate code goes here (three lines)
  }
}
```

code candidates:

```
b.m1();
c.m2();
a.m3();
```

```
c.m1();
c.m2();
c.m3();
```

```
a.m1();
b.m2();
c.m3();
```

```
a2.m1();
a2.m2();
a2.m3();
```

output:

```
A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13
```

- Quiz: what is the result?

```
1.  public class SimpleCalc{
2.      public int value;
3.      public void calculate(){value += 7;}
4.  }
```

And:

```
1.  public class MultiCalc extends SimpleCalc{
2.      public void calculate(){value -= 3;}
3.      public void calculate(int multiplier){
4.          calculate();
5.          super.calculate();
6.          value *= multiplier;
7.      }
8.      public static void main(String[] args){
9.          MultiCalc calculator = new MultiCalc();
10.         calculator.calculate(2);
11.         System.out.println("Value is: " + calculator.value);
12.     }
13. }
```

- Quiz

Given:

```
10.  class One{
11.      void foo(){}
12.  }
13.  class Two extends One{
14.      //insert method here
15.  }
```

Which three methods, inserted individually at line 14, will correctly complete class Two? (Choose three.)

A. int foo(){/* more code here */}

B. void foo(){/* more code here */}

C. public void foo(){/* more code here */}

D. private void foo(){/* more code here */}

E. protected void foo(){/* more code here */}

- Quiz

```
1.  public class Blip{
2.      protected int blipvert(int x){return 0;}
3.  }
4.  class Vert extends Blip{
5.      //insert code here
6.  }
```

Which five methods, inserted independently at line 5, will compile? (Choose five.)

A. public int blipvert(int x){return 0;}

B. private int blipvert(int x){return 0;}

C. private int blipvert(long x){return 0;}

D. protected long blipvert(int x){return 0;}

E. protected int blipvert(long x){return 0;}

F. protected long blipvert(long x){return 0;}

G. protected long blipvert(int x, int y){return 0;}

- Quiz

```
1.    public class A{
2.        public void doit(){
3.        }
4.        public String doit(){
5.            return "a";
6.        }
7.        public double doit(int x){
8.            return 1.0;
9.        }
10.   }
```

What is the result?

A. An exception is thrown at runtime.

B. Compilation fails because of an error in line 7.

C. Compilation fails because of an error in line 4.

D. Compilation succeeds and no runtime errors with class A occur.

- Extension
- In chapter 4, we said that Java does not support default parameters. However, you can use method overloading to simulate the “default value” behavior

```
void foo(int a, int b) {
    //...
}

void foo(int a) {
    foo(a, 0); // here 0 is a default value for b
}

foo(3,4);
foo(3);
```