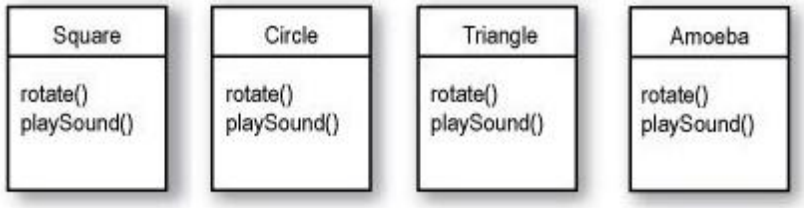
Inheritance and Polymorphism

Samkeun Kim <skim@hknu.ac.kr>

http://cyber.hankyong.ac.kr

Chair Wars Revisited...

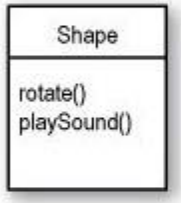


❶

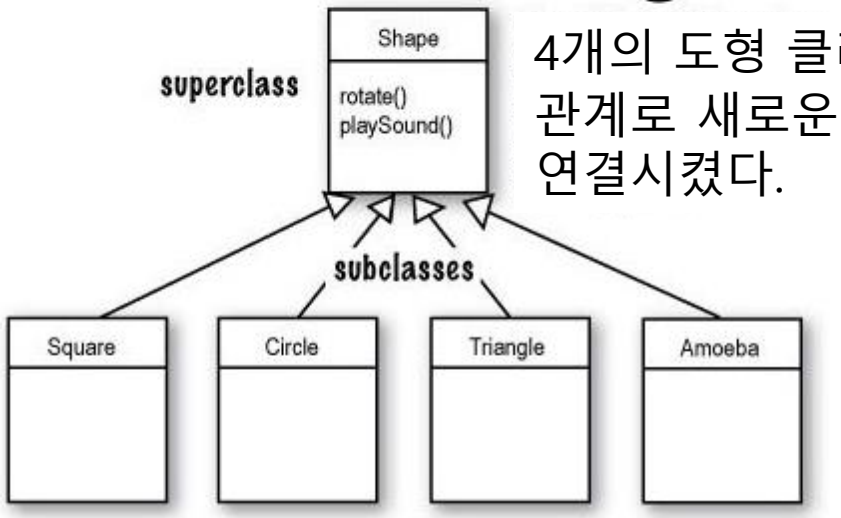
클래스 4개에 공통적으로 들어있는 것을 찾아내서

❷

Shape라는 새로운 클래스에 집어넣은 후



❸

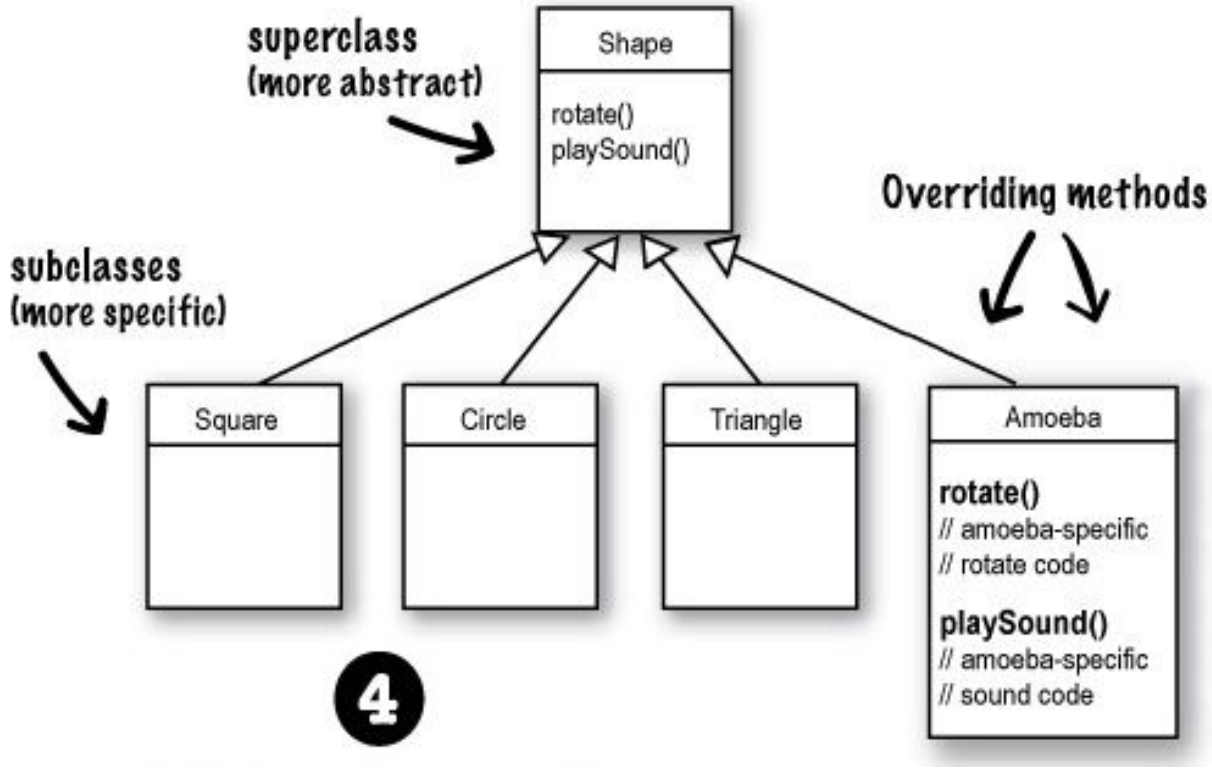


4개의 도형 클래스를 **상속**이라는 관계로 새로운 **Shape** 클래스에 연결시켰다.

You can read this as, "**Square inherits from Shape**", "**Circle inherits from Shape**", and so on.

The **Shape** class is called the **superclass** of the other four classes.

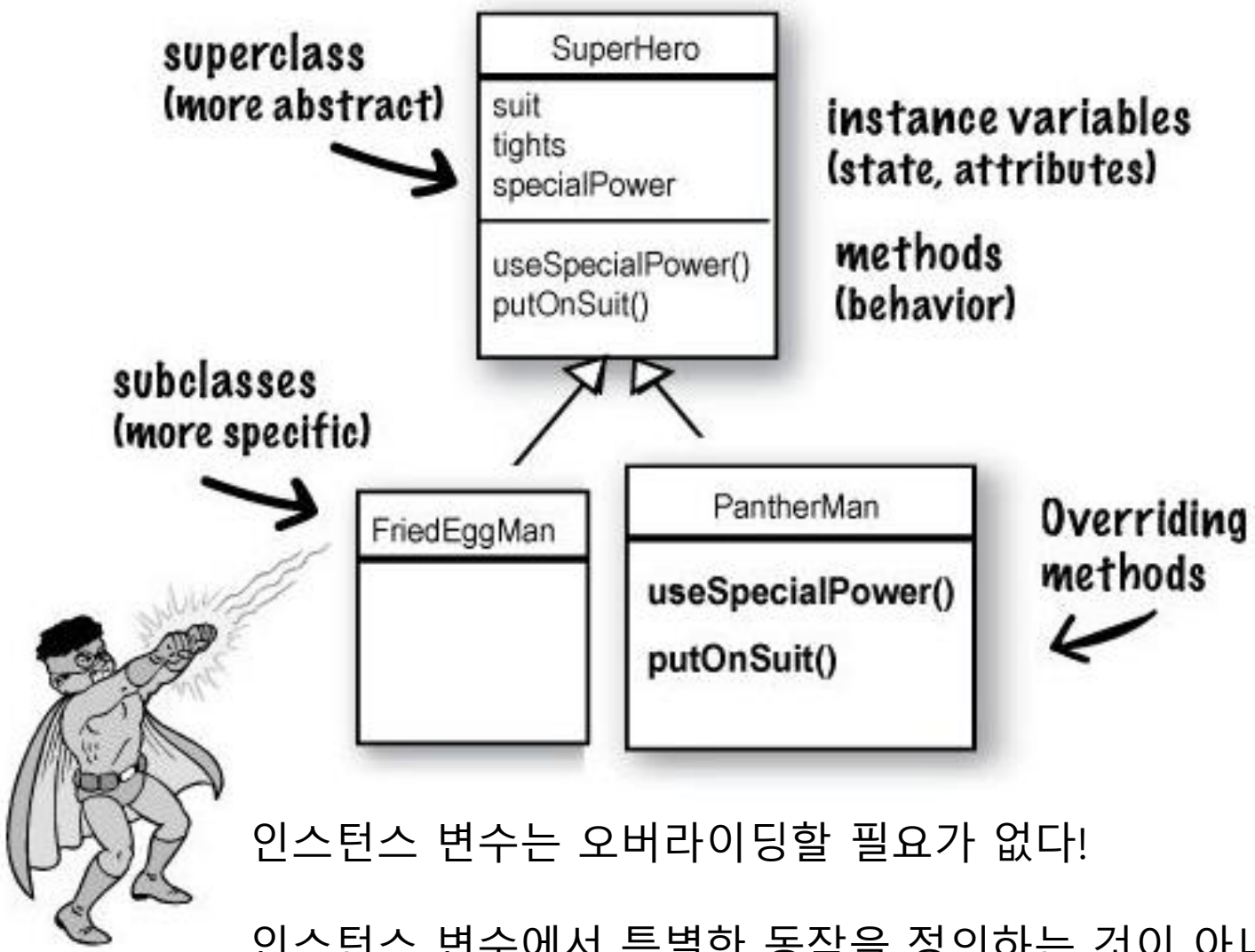
What about the Amoeba rotate()?



Amoeba 클래스에서 슈퍼클래스인 **Shape** 클래스의 **rotate()**와 **playSound()** 메소드를 오버라이딩 시켰다.

오버라이딩이란 서브클래스가 슈퍼클래스의 메소드를 상속받아서 재정의하는 것을 의미한다.

상속의 이해

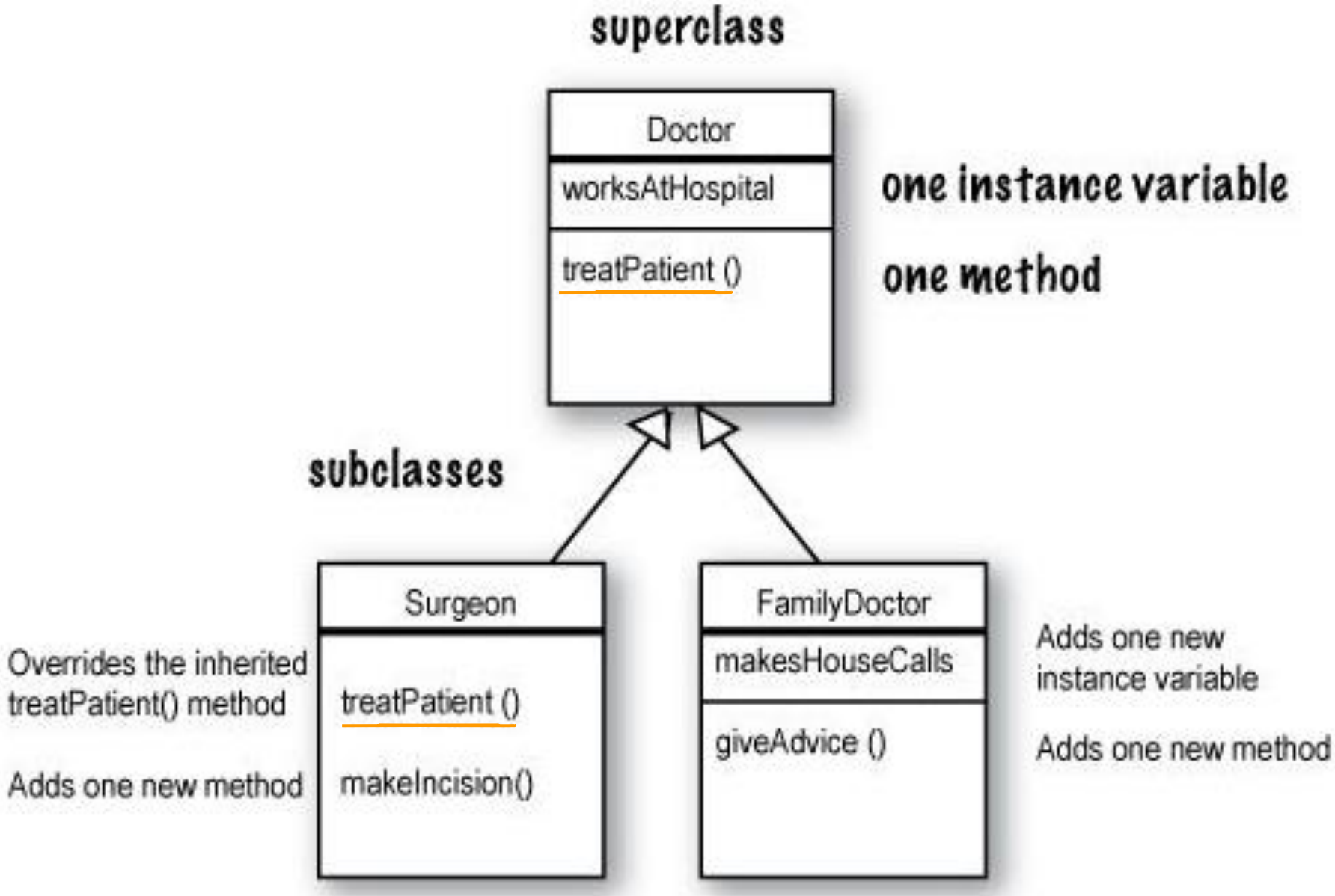


인스턴스 변수는 오버라이딩할 필요가 없다!

인스턴스 변수에서 특별한 동작을 정의하는 것이 아니기 때문에 상속받은 인스턴스 변수를 서브클래스에서 그대로 사용하고 적절한 값을 선택하면 된다.

상속 예제

```
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!)
   }
}
```


superclass
Doctor
worksAtHospital
treatPatient ()
one instance variable
one method
subclasses
Surgeon
treatPatient ()
makeIncision()
Overrides the inherited treatPatient() method
Adds one new method
FamilyDoctor
makesHouseCalls
giveAdvice ()
Adds one new instance variable
Adds one new method

동물 시뮬레이션 프로그램을 위한 상속 트리 설계

한 무더기의 서로 다른 동물들이 한 울타리에 들어 있을 때 어떤 일이 일어나는지를 살펴볼 수 있게 해주는 **시뮬레이션 프로그램**을 설계해보자.

프로그램에 들어갈 모든 동물의 목록은 아직 없고 **일부 동물의 목록**만 받은 상태이다.

각 동물은 객체로 표현될 수 있고 객체들은 주어진 환경에서 프로그램이 지정한대로 움직일 것이다.

그리고 다른 프로그래머들도 언제든지 프로그램에 새로운 종류의 동물을 추가할 수 있게 만들고 싶다.

먼저 모든 동물들이 가지는 공통적이고 추상적인 특성을 파악해서

그러한 특성을 모든 동물 클래스들이 상속받을 수 있도록 해주는 클래스를 만들어야 한다.

1. 공통적인 속성과 행동을 가지는 객체들을 찾아보자.

여섯 종류의 동물에서 어떤 공통적인 특성을 찾을 수 있을까?

=> 그 과정에서 행동을 추상화

각 타입은 어떻게 연관될까?

=> 상속 트리에서의 관계를 정의

상속을 이용하여 서브클래스에서 중복 코드를 피하는 법

5개의 **인스턴스 변수**를 가진다:

picture – the file name representing the JPEG of this animal

food – the type of food this animal eats.

hunger – an int representing the hunger level of the animal. It changes depending on when (and how much) the animal eats.

boundaries – values representing the height and width of the 'space' (for example, 640 x 480) that the animals will roam around in.

location – the X and Y coordinates for where the animal is in the space.

4개의 **메소드**를 가진다:

makeNoise() – behavior for when the animal is supposed to make noise.

eat() – behavior for when the animal encounters its preferred food source, meat or grass.

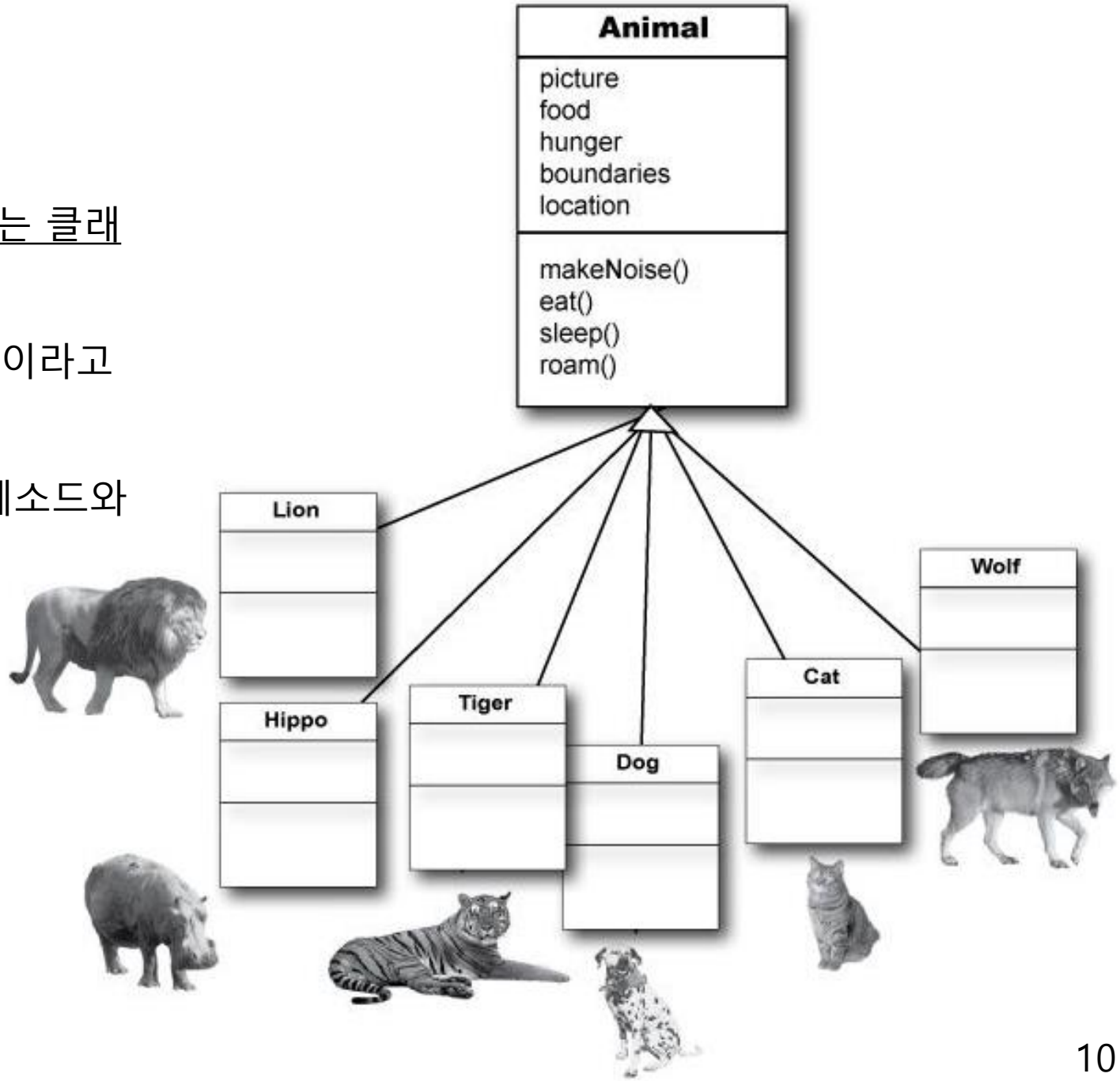
sleep() – behavior for when the animal is considered asleep.

roam() – behavior for when the animal is not eating or sleeping (probably just wandering around waiting to bump into a food source or a boundary).

2. 공통적인 **State**와 **Behavior**를 나타내는 클래스를 설계한다.

   이들 객체는 모두 동물이므로 **Animal**이라고 하는 슈퍼클래스를 만든다.

   그 안에는 모든 동물이 필요로 하는 메소드와 인스턴스 변수를 집어넣는다.

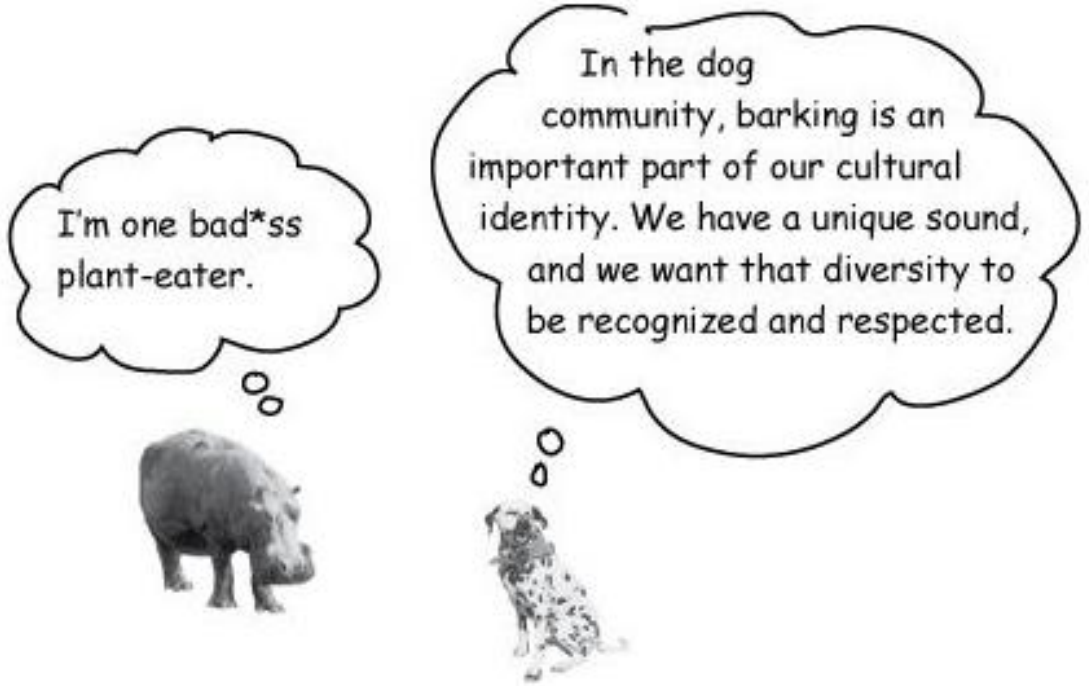


어떤 메소드를 오버라이딩해야 할까?

3. 특정 서브클래스에만 적용되는 Behavior (메소드 구현)가 필요한지 결정한다.

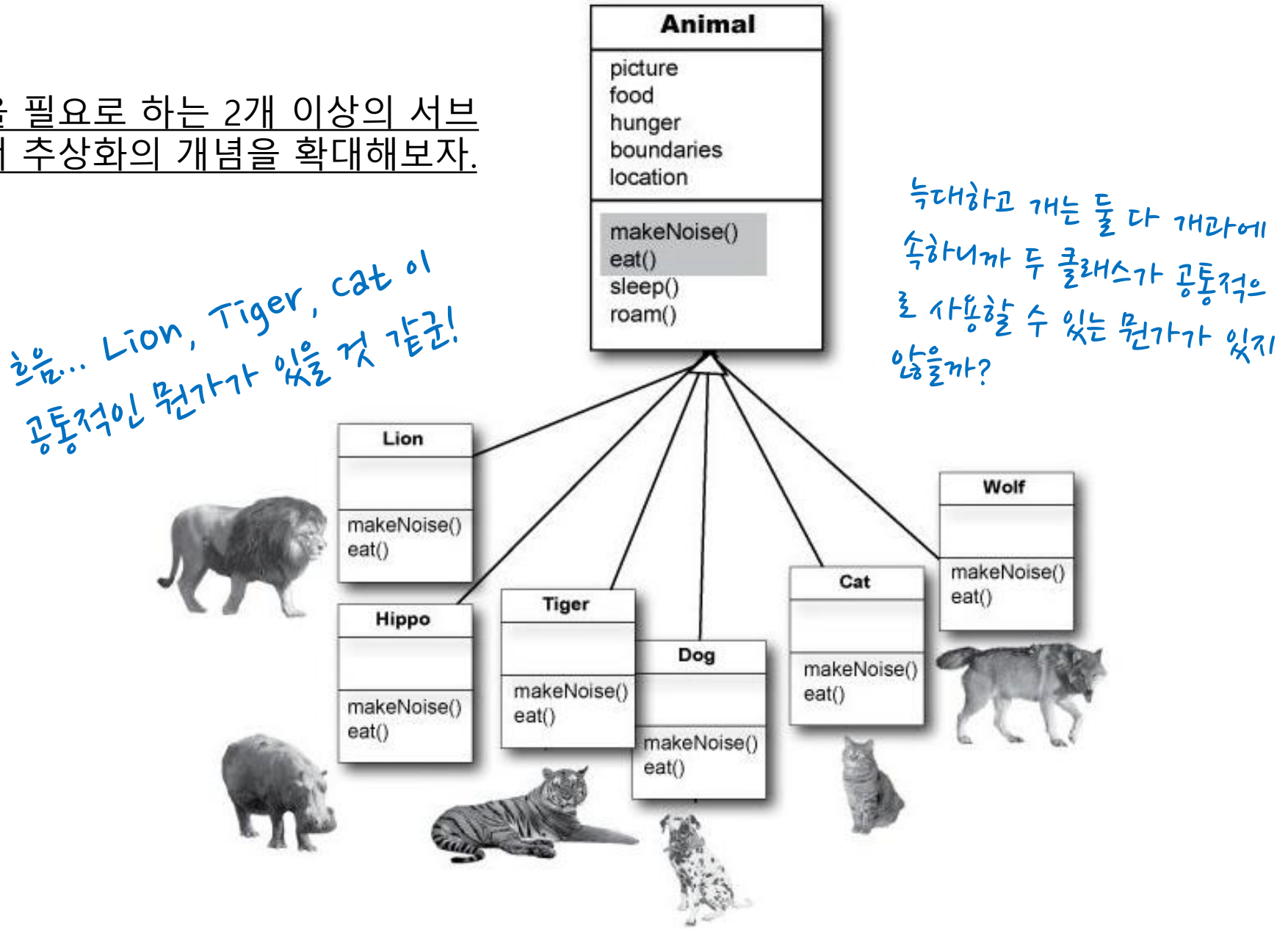
사자는 개와 같은 소음(noise)을 내는가?
고양이는 하마처럼 먹는가(eat)?

Animal 클래스에서 볼 때 **eat()**와 **makeNoise()** 메소드는 오버라이드하는 것이 좋겠다고 판단한다.



상속을 더 많이 활용하는 방법을 찾아보자

4. 공통적인 행동을 필요로 하는 2개 이상의 서브 클래스를 찾아서 추상화의 개념을 확대해보자.

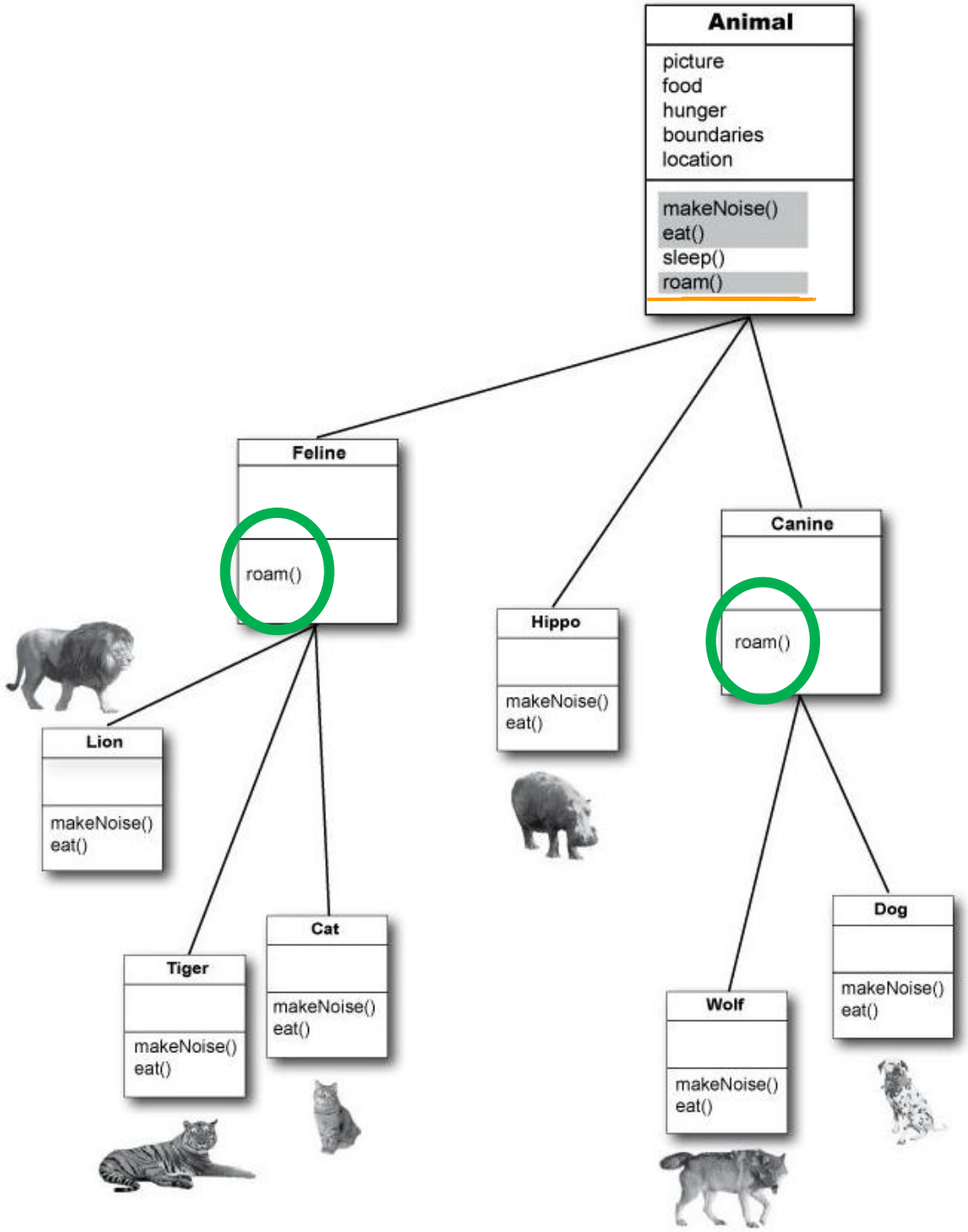


5. 클래스 계층 구조를 완성해보자.

개과 동물은 무리를 지어서 다니는 습성이 있으므로 **Canine** 클래스에서 공통 **roam()** 메소드를 사용할 수 있다.

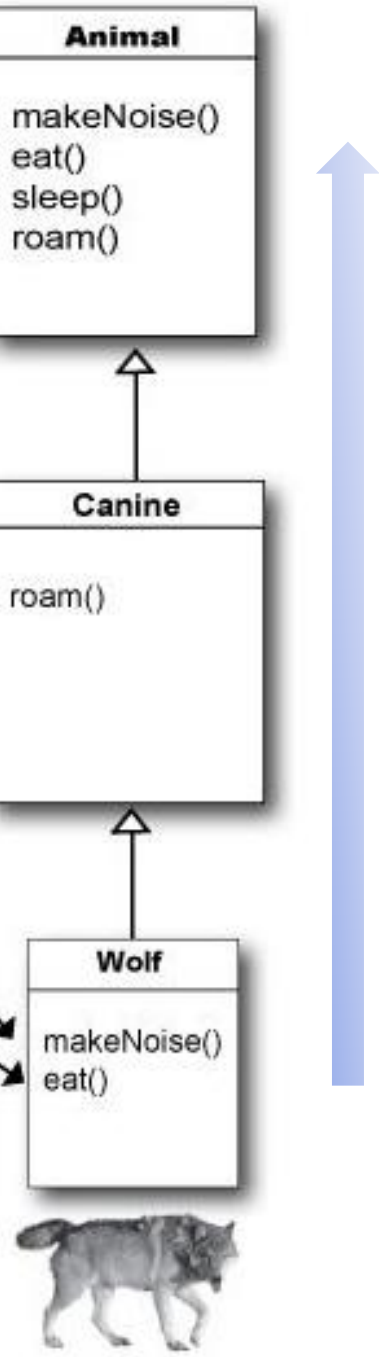
고양이과 동물은 같은 종류에 속하는 다른 동물들을 피하려는 습성이 있으므로 **Feline** 클래스에서 공통 **roam()** 메소드를 사용할 수 있다.

Hippo 클래스에서는 그냥 **Animal** 클래스에 있는 **roam()** 메소드를 사용한다.



어느 메소드가 호출될까?

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();`



객체 레퍼런스의 어떤 메소드를 호출하면 그 객체 형식의 메소드 중에서 가장 구체적인 버전이 호출된다.

즉 가장 아래에 있는 것이 호출된다!

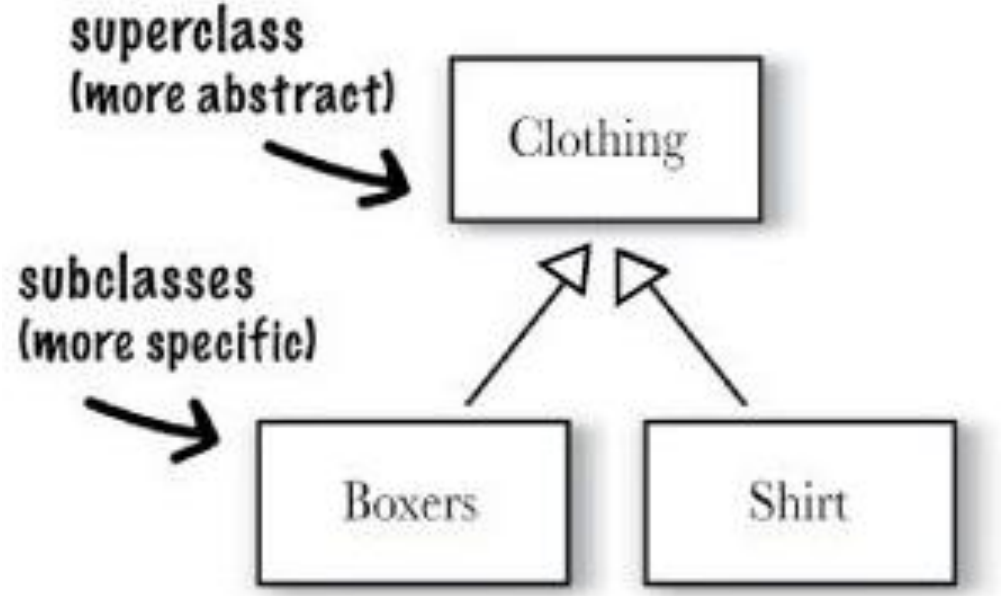
Wolf 객체에 대해 어떤 메소드가 호출되면 JVM에서는 일단 가장 아래쪽에 있는 **Wolf** 클래스에서부터 찾아본다. 없으면 찾을 때까지 계속 위로 찾아 올라간다.

상속 트리 설계

Designing an Inheritance Tree

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

Inheritance Table



Inheritance Class Diagram

IS-A와 HAS-A의 사용

한 클래스가 다른 클래스로부터 상속받을 때 **서브클래스**가 **슈퍼클래스**를 확장한다고 말한다.

한 사물이 다른 것을 확장하는지를 알고 싶을 때 **IS-A** 테스트를 적용하라!

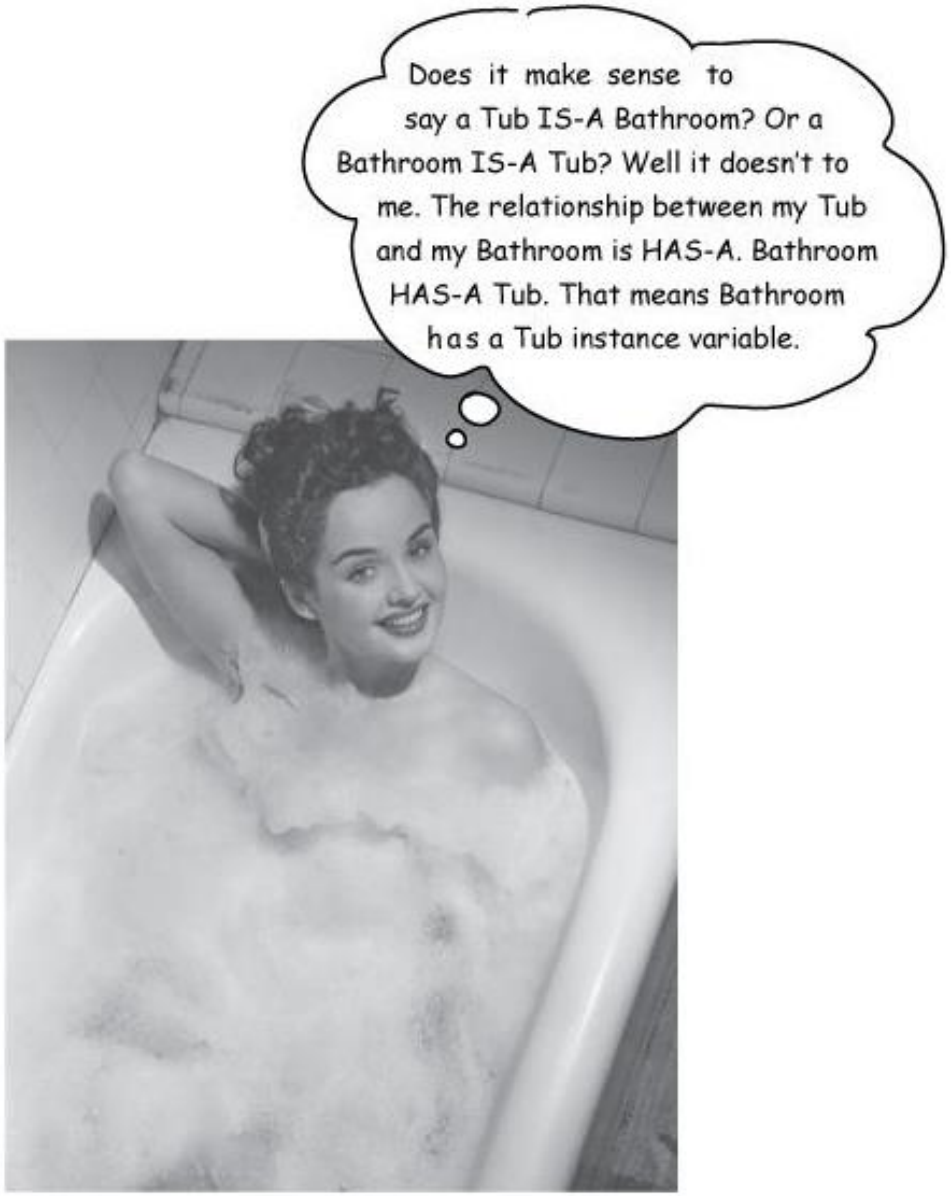
Triangle IS-A Shape, yeah, that works.
Cat IS-A Feline, that works too.
Surgeon IS-A Doctor, still good.

Tub extends Bathroom => 뭔가 맞는 말 같기도 한데...

IS-A 테스트를 적용해보기 전까지는.

Tub IS-A Bathroom? => **Definitely false**.

그렇다면 거꾸로 bathroom이 tub를 확장하는가? 여전히 맞지 않다.
Bathroom이 tub이다 (**IS-A**) => 여전히 이것도 맞지 않다.



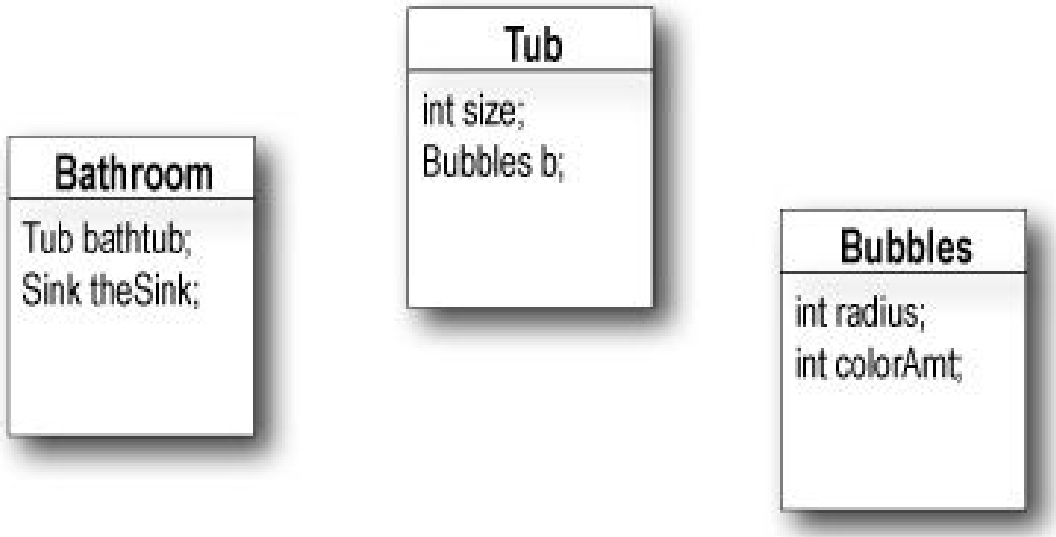
Tub와 bathroom은 연관관계는 있지만 상속관계는 아니다.

Tub와 bathroom은 **HAS-A** 관계로 엮여있다.

"**Bathroom HAS_A Tub**"? => **Valid**.

의미: bathroom은 tub 인스턴스 변수를 가진다.

Bathroom은 tub에 대한 레퍼런스를 가지지만 bathroom이 tub를 확장하지는 않는다. 이의 역도 마찬가지다.



잠깐! 약간 더 남아있다

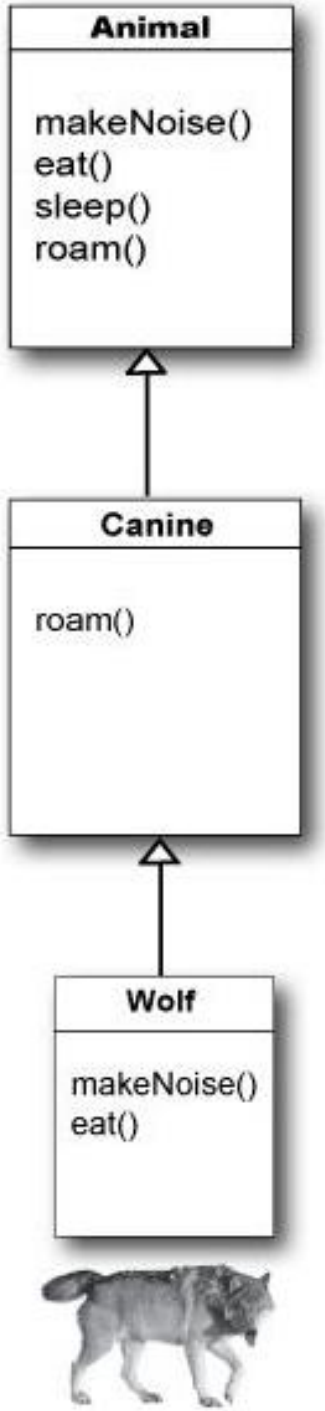
IS-A 테스트는 상속 트리의 모든 곳에서 성립한다.

If class B **extends** class A, class B **IS-A** class A.

If class C **extends** class B, class C **passes** the **IS-A** test for both B and A.

Canine extends **Animal**
Wolf extends **Canine**
Wolf extends **Animal**
Canine IS-A **Animal**
Wolf IS-A **Canine**
Wolf IS-A **Animal**

Animal이 상속 계층 구조에서 **Wolf** 위에 있기만 하면 '**Wolf**는 **Animal**이다' 라는 명제는 항상 참이다.



상속구조를 제대로 만들었는지 어떻게 알 수 있나

IS-A 상속 관계는 한 방향으로만 작동한다!

Triangle **IS-A** Shape => Valid!

Reverse: Shape **IS-A** Triangle => Invalid!

따라서 **Shape**은 **Triangle**을 확장하지 않는다!

NOTE If X **IS-A** Y, then X can do anything a Y can do (and possibly more).

Who gets the Porsche, who gets the porcelain?

(how to know what a subclass can inherit from its superclass)

무조건 상속?

서브클래스는 슈퍼클래스의 멤버들을 상속받는다.

멤버: **인스턴스 변수**와 **메소드**

슈퍼클래스에서는 접근 레벨을 통하여 서브클래스로 하여금 상속받을 수 있게 할지 아닐지를 선택할 수 있게 한다.

4가지 접근 레벨이 있다: 가장 왼쪽이 가장 제한적이다

private	default	protected	public

접근 레벨은 누가 무엇을 볼 수 있는지를 통제한다.

public members **are** inherited.

private members are **not** inherited.

상속을 활용하여 설계할 때의 주의해야 할 점

상속을 설계할 때 지켜야 할 몇 가지 규칙:

어떤 클래스가 슈퍼클래스의 더 구체화된 형식이라면 상속을 이용하라.

> 예를 들어, 버드나무(Willow)는 나무(Tree)를 구체화한 것이라고 볼 수 있으므로 버드나무가 나무를 확장했다고 하는 것이 의미가 있다.

같은 일반적인 형식에 속하는 여러 클래스에서 공유되어야 하는 어떤 행동(구현된 코드)이 있다면 상속을 활용하라.

> 예를 들어, **Square**, **Circle**, **Triangle**에는 모두 도형을 회전시키고 소리를 재생하는 메소드가 필요하다. 따라서 그런 기능을 **Shape**라는 슈퍼클래스에 집어넣는 것이 자연스럽고, 그렇게 하면 클래스의 관리와 확장이 용이해진다.

위의 두 가지 규칙 중에 하나라도 위배되고 단순히 다른 클래스로부터의 코드를 재사용하기 위해서는 상속을 사용하면 안된다.

> 예를 들어, **Alarm** 클래스에 특별한 '인쇄 코드'를 작성했고, 이제 **Piano** 클래스에서 그 '인쇄 코드'가 필요하다고 하자. 그러면 **Piano** 클래스가 '인쇄 코드'를 상속받을 수 있도록 'Piano extends Alarm'을 가져야 한다. 이건 말도 안되는 소리다! 피아노는 특수한 형태의 알람이 아니다.

서브클래스와 슈퍼클래스가 **IS-A** 테스트를 통과하지 못하면 상속을 사용하지 말라.

상속 기능을 활용하면 얻어지는 장점

상속을 이용하여 설계하면 **OO**(Object-Oriented, 객체지향)의 여러 이점을 얻을 수 있다.

일련의 클래스에서 공통적으로 필요한 행동을 뽑아서 그 코드를 슈퍼클래스에 집어넣으면 **코드가 중복되는 것을 방지**할 수 있다.

이렇게 되면 **코드를 수정할 필요가 있을 때 슈퍼클래스에서만 변경하면 된다!**

⇒ 변경된 내용은 그 행동을 상속받는 모든 클래스에서 자동으로 반영한다

⇒ 서브클래스는 전혀 건드릴 필요가 없다!!

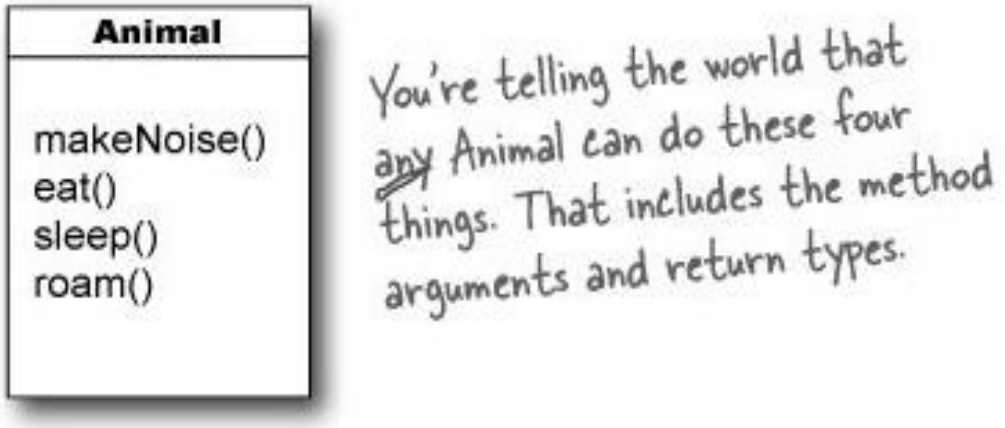
새로 컴파일한 슈퍼클래스를 기존의 자리에 넣기만 하면 그 클래스를 확장한 서브클래스에서는 자동으로 새로운 버전을 사용하게 된다!

1. 중복 코드를 피하라.
2. 클래스 그룹을 위한 **프로토콜**(protocol)을 정의하라.

프로토콜?

어떤 슈퍼 타입에서 파생된 모든 서브클래스는 슈퍼 타입이 가진 모든 메소드를 가지도록 보장받는다.
즉 연관된 클래스 집합에 대한 공통 규약을 **상속을 통하여** 정의한다.

아래 **Animal** 클래스에서는 **Animal**에 속하는 모든 서브클래스를 위한 **공통된 규약**을 확립해 놓은 것이다:



여기서 Any **Animal**이라고 하는 것은 **Animal**과 **Animal을 확장한 모든 클래스**를 의미한다.

- ✓ **Animal**을 확장한 모든 클래스: 상속 계층 구조에서 그 위 어딘가에 **Animal**이 있는 클래스를 의미한다

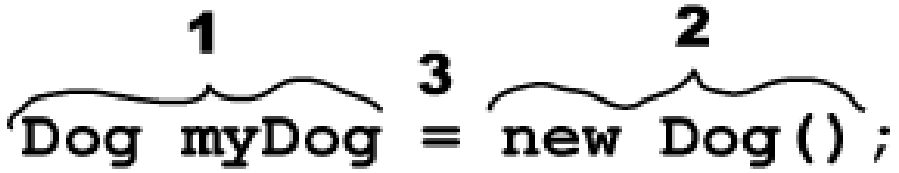
다형성(Polymorphism)

한 그룹의 클래스들을 위한 슈퍼타입을 정의할 때, 그 슈퍼타입의 어떤 서브클래스도 슈퍼타입이 들어가야 할 자리에 대신 들어갈 수 있다.

Say, What?

⇒ 슈퍼타입으로 선언된 레퍼런스를 사용하여 서브클래스 객체를 참조하게 한다는 뜻이다.

객체 선언, 생성과 대입의 3단계



1. 레퍼런스 변수 선언

   Dog myDog = new Dog();

   JVM에게 레퍼런스 변수를 저장할 공간을 할당해 달라고 요청

2. 객체 생성

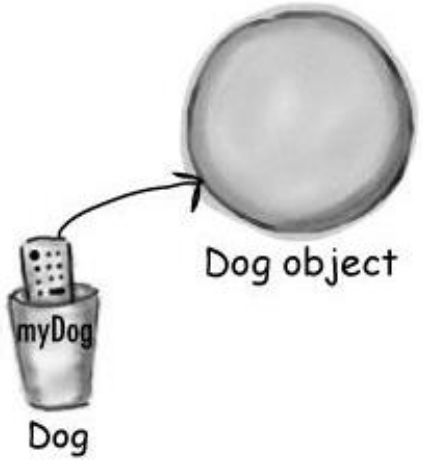
   Dog myDog = **new Dog();**

   JVM에게 힙에 새로운 Dog 객체를 위한 공간을 할당해 달라고 요청

3. 객체와 레퍼런스 연결

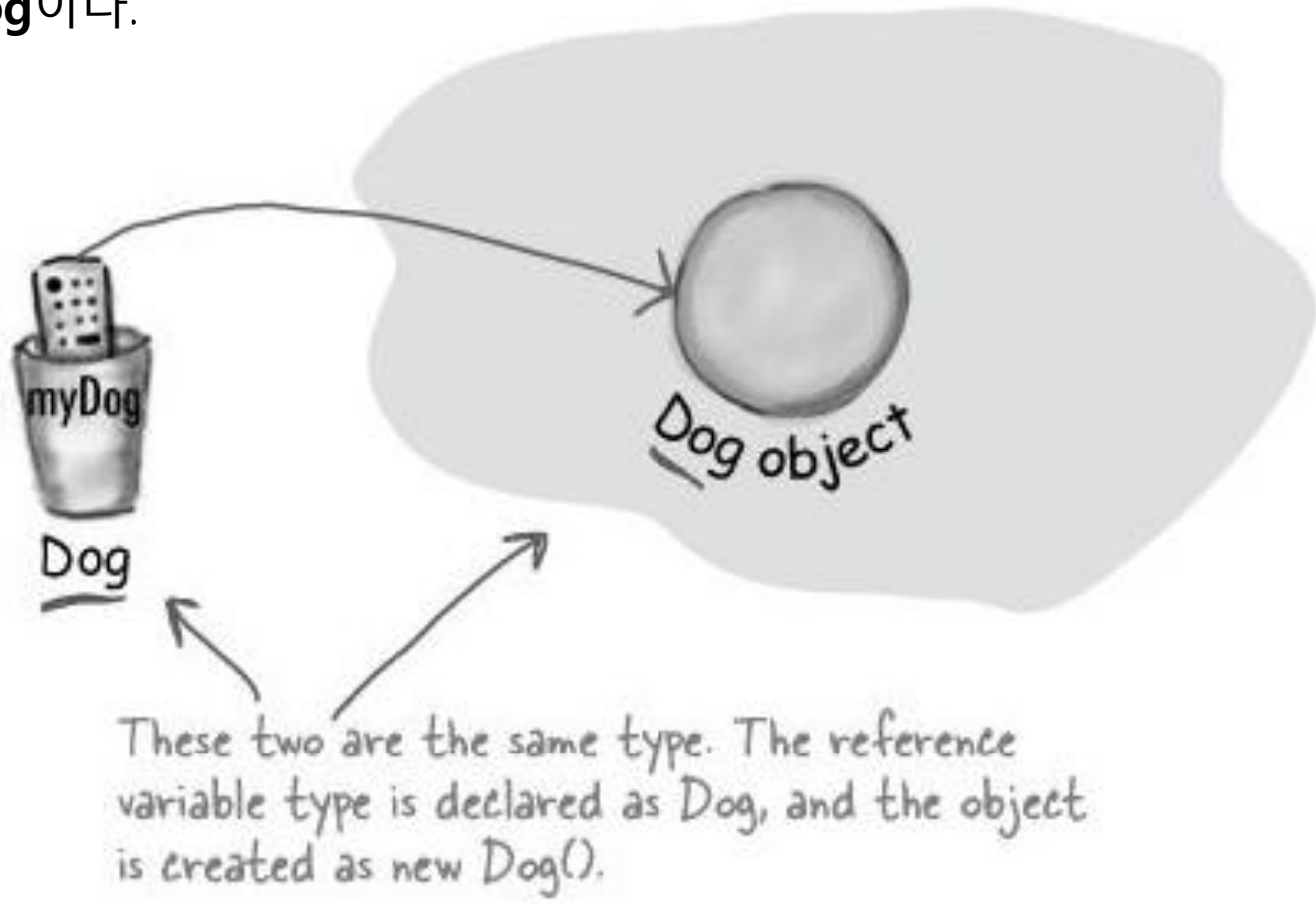
   Dog myDog **=** new Dog();

   새로운 Dog 객체를 myDog라는 레퍼런스 변수에 대입한다.



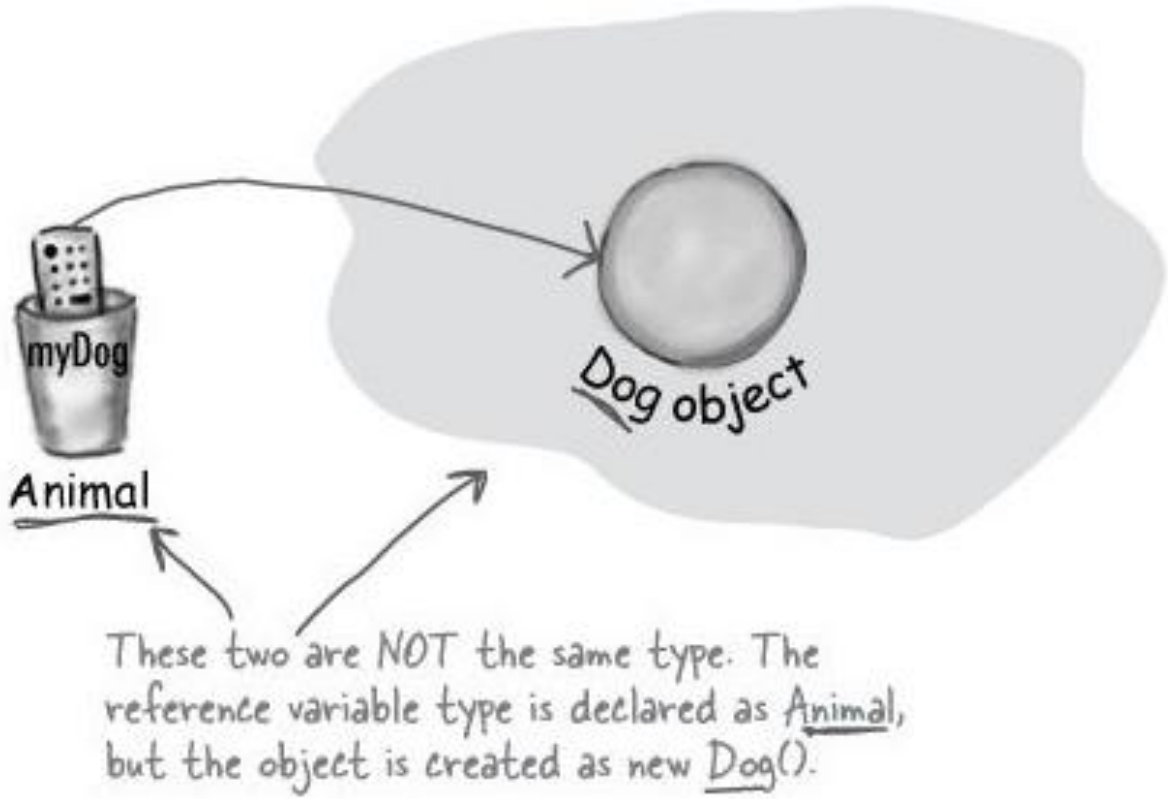
중요한 것은 레퍼런스 타입과 객체 타입이 똑같아야 한다는 점이다.

이 예에서 둘 다 **Dog**이다.



그러나 다형성을 이용하면 레퍼런스와 객체가 서로 다를 수도 있다!

Animal myDog = new Dog();



다형성에서는 레퍼런스 타입이 실제 객체 타입의 슈퍼클래스 타입이 될 수 있다!

OK, OK maybe an example will help.

```
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.

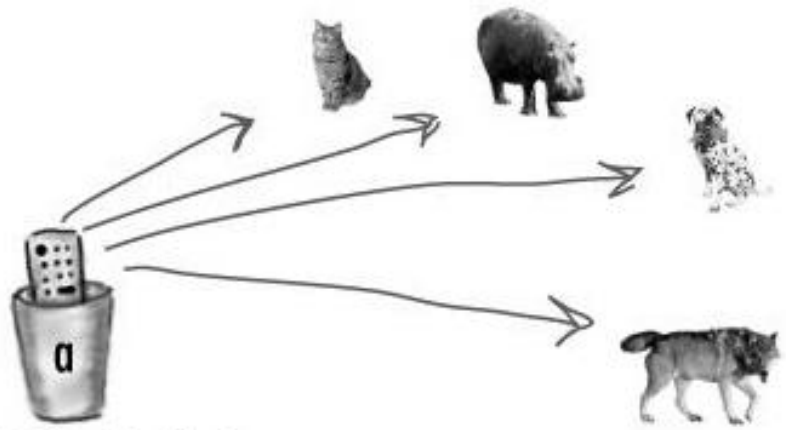
But look what you get to do... you can put ANY subclass of Animal in the Animal array!

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().

But wait! There's more!



아규먼트와 리턴 타입에 대해서도 다형성을 적용할 수 있다!

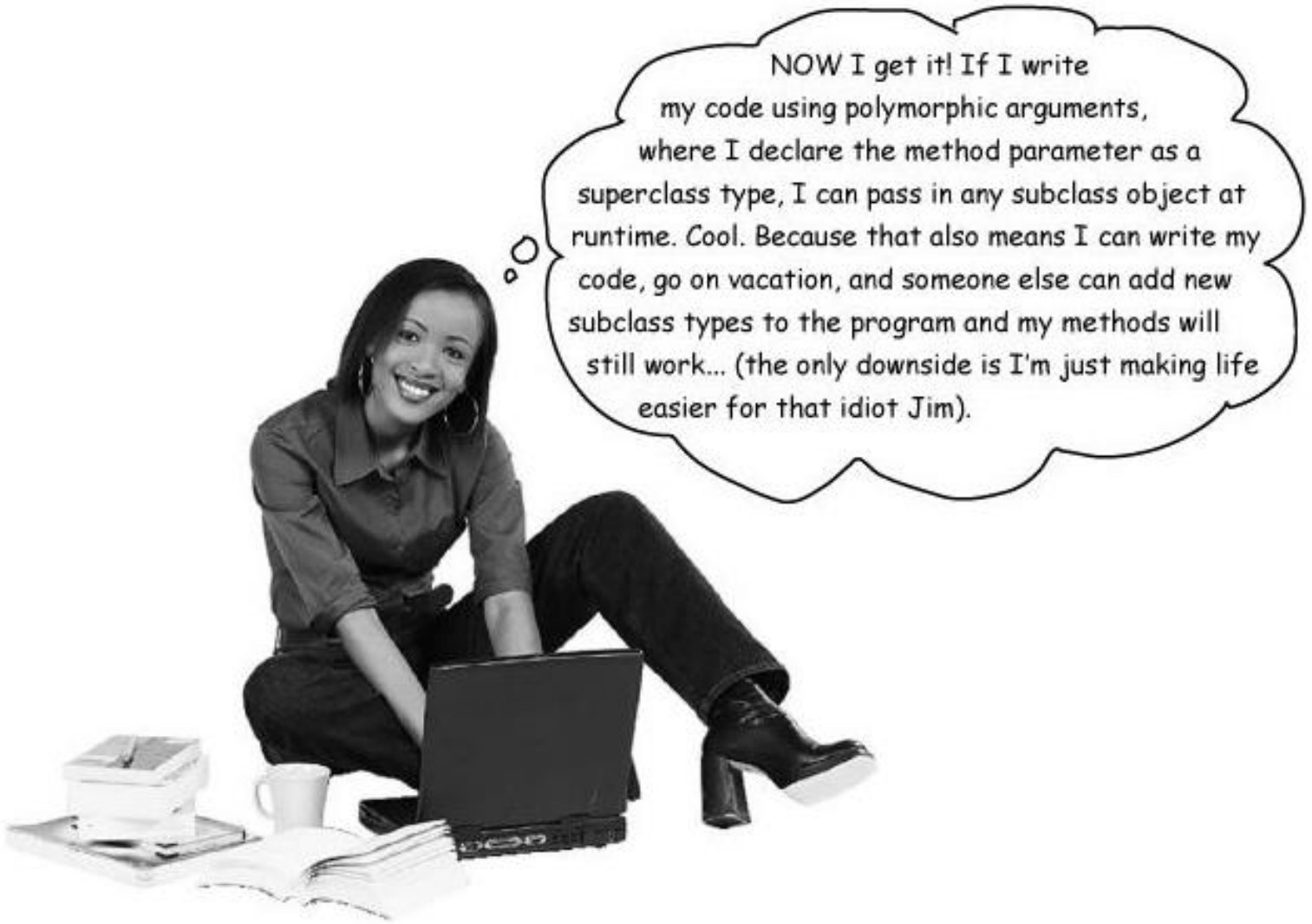
```
class Vet {
   public void giveShot(Animal a) {
      // do horrible things to the Animal at
      // the other end of the 'a' parameter
      a.makeNoise();
   }
}
```

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

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

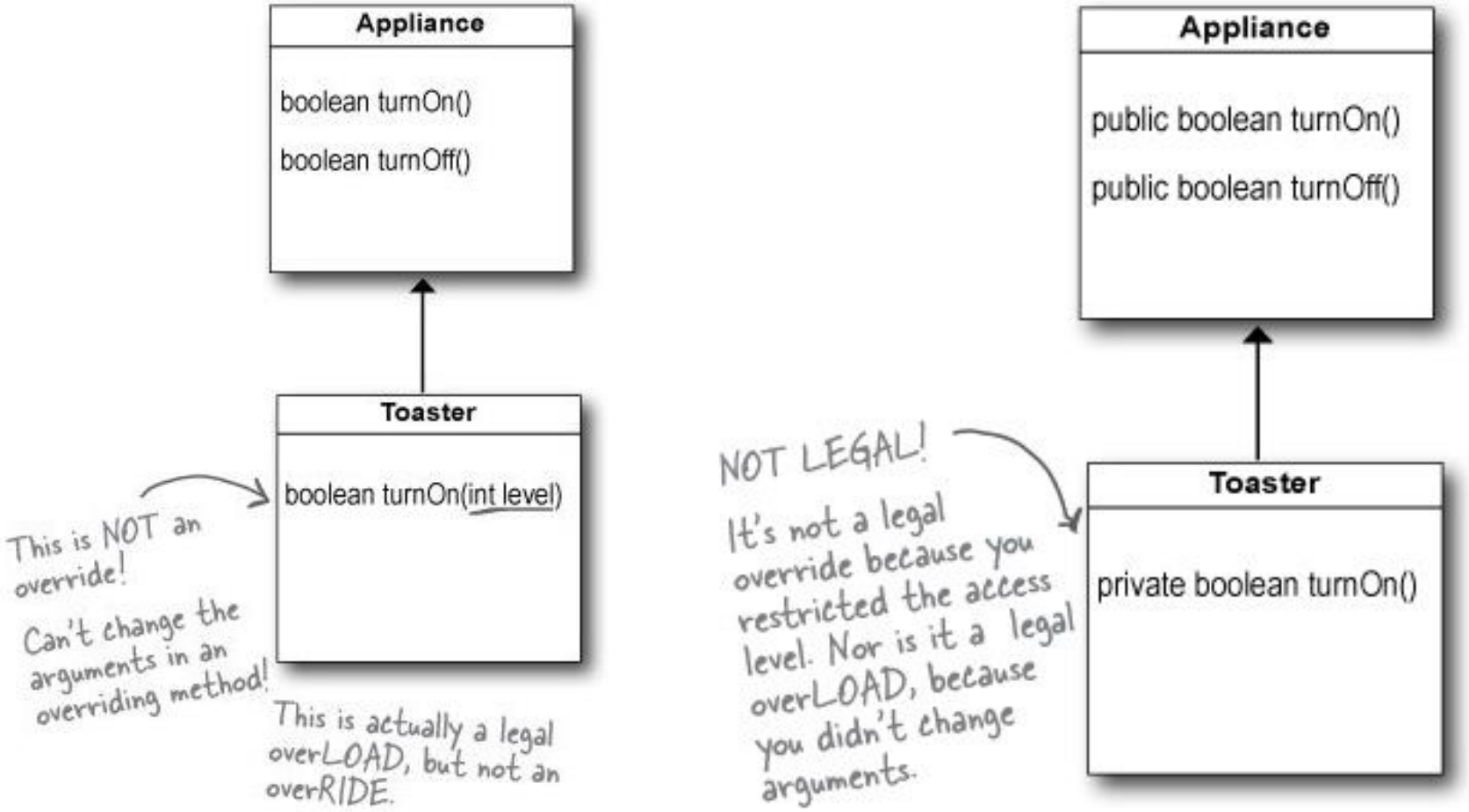


다형성을 이용하면 새로운 서브클래스 형식을 프로그램에 추가하더라도 굳이 코드를 변경할 필요가 없다!

Keeping the contract: rules for overriding

다형성이 동작하기 위해 오버라이딩된 Toaster 버전이 지켜야 할 규칙:

1. 아규먼트는 똑같아야 하고, 리턴 타입은 호환 가능해야 한다.
2. 메소드 접근 레벨은 더 제한적이지 않아야 한다.



메소드 오버로딩 (Overloading)

메소드 오버로딩: 단순히 두 메소드가 이름은 같지만 서로 다른 아규먼트를 가지는 것을 말한다.

1. 리턴 타입이 달라도 된다.
2. 리턴 타입만을 바꿀 수는 없다.
3. 접근 레벨을 마음대로 바꿀 수 있다.

오버로딩은 상속과 다형성과는 아무런 관련이 없다.

오버로딩된 메소드와 오버라이딩된 메소드는 결코 같지 않다.

Overloading a method

Legal examples 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);
   }
}
```

실습과제 7-1 BE the Compiler

```
public class MonsterTestDrive {
  public static void main(String [] args) {
    Monster [] ma = new Monster[3];
    ma[0] = new Vampire();
    ma[1] = new Dragon();
    ma[2] = new Monster();
    for(int x = 0; x < 3; x++) {
      ma[x].frighten(x);
    }
  }
}

class Monster {
  A
}

class Vampire extends Monster {
  B
}

class Dragon extends Monster {
  boolean frighten(int degree) {
    System.out.println("breath fire");
    return true;
  }
}
```

왼쪽에 있는 클래스에 삽입하면 오른쪽에 나열된 **A-B** 쌍의 메소드 중 어느 것이 컴파일되어 출력을 생성하는가? (**A** 메소드는 **Monster** 클래스에 삽입하고, **B** 메소드는 **Bampire** 클래스에 삽입한다)

```
File Edit Window Help SaveYourself
% java MonsterTestDrive
a bite?
breath fire
arrrgh
```

1

A
```
boolean frighten(int d) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
boolean frighten(int x) {
  System.out.println("a bite?");
  return false;
}
```

2

A
```
boolean frighten(int x) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
int frighten(int f) {
  System.out.println("a bite?");
  return 1;
}
```

3

A
```
boolean frighten(int x) {
  System.out.println("arrrgh");
  return false;
}
```

B
```
boolean scare(int x) {
  System.out.println("a bite?");
  return true;
}
```

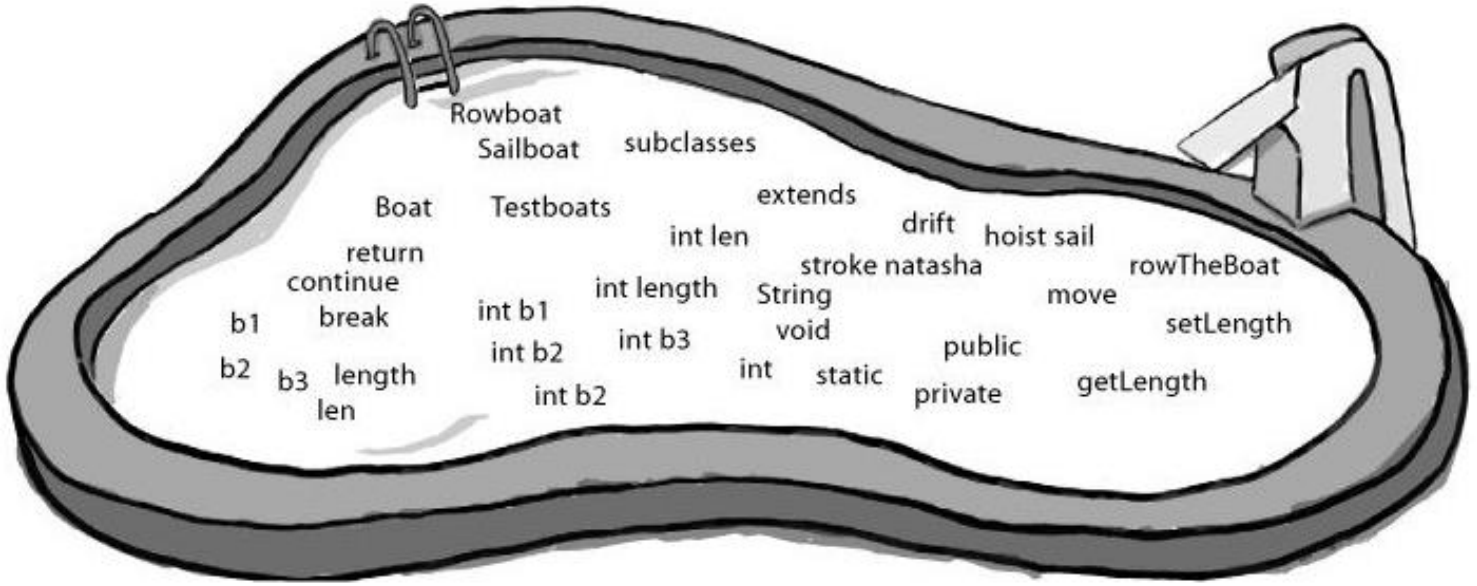
4

A
```
boolean frighten(int z) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
boolean frighten(byte b) {
  System.out.println("a bite?");
  return true;
}
```

실습과제 7-2 Pool Puzzle

OUTPUT: `drift drift hoist sail`




```
public class Rowboat _______ _______ {
   public __________ rowTheBoat() {
      System.out.print("stroke natasha");
   }
}

public class _______ {
   private int __________ ;
   _______ void _________ ( ______ ) {
      length = len;
   }
   public int getLength() {
      _______ _________ ;
   }
   public __________ move() {
      System.out.print("__________");
   }
}

public class TestBoats {
   ______ ______ _______ main(String[] args){
      _________ b1 = new Boat();
      Sailboat b2 = new __________();
      Rowboat _______ = new Rowboat();
      b2.setLength(32);
      b1.__________();
      b3.__________();
      _______.move();
   }
}

public class __________ _______ Boat {
   public _______ _________() {
      System.out.print("__________");
   }
}
```