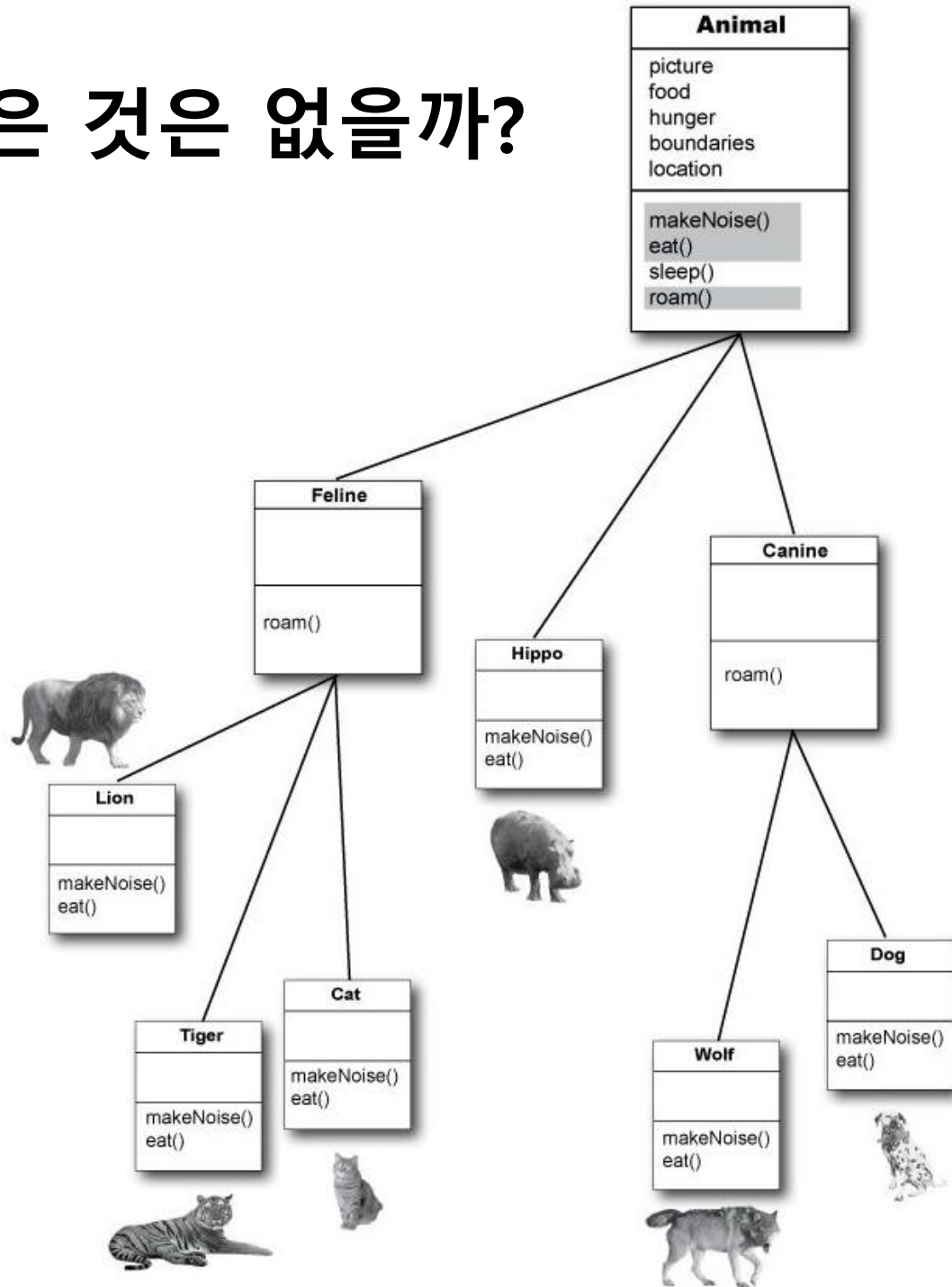


Interfaces and Abstract Classes: Serious Polymorphism

Samkeun Kim <skim@hknu.ac.kr>

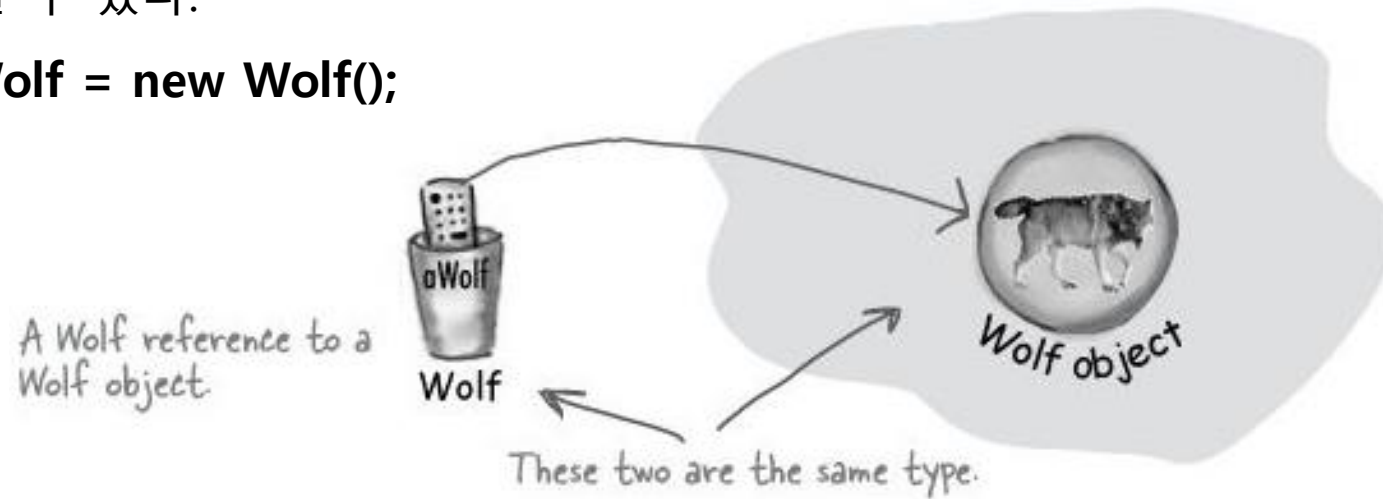
<http://cyber.hankyong.ac.kr>

이걸 설계할 때 뭔가 잊은 것은 없을까?



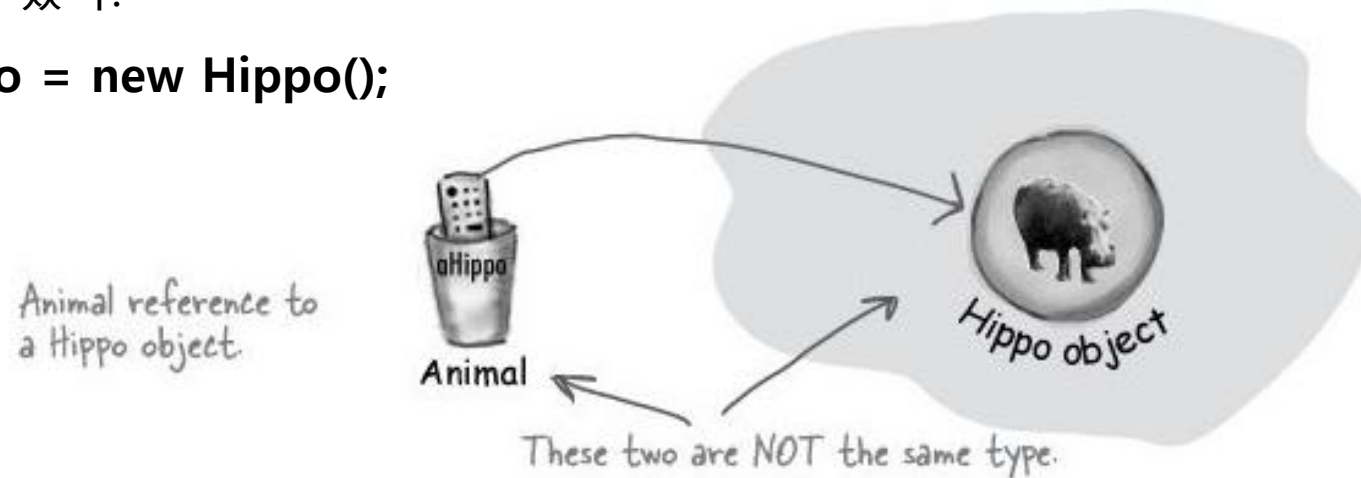
이렇게 말할 수 있다:

Wolf aWolf = new Wolf();



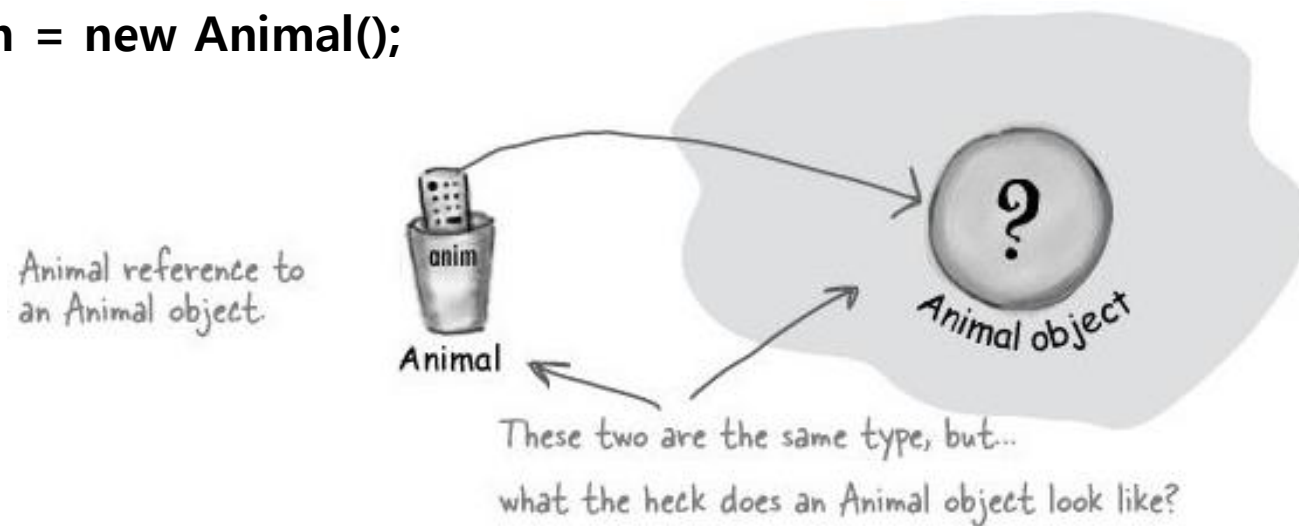
이렇게도 말할 수 있다:

Animal aHippo = new Hippo();

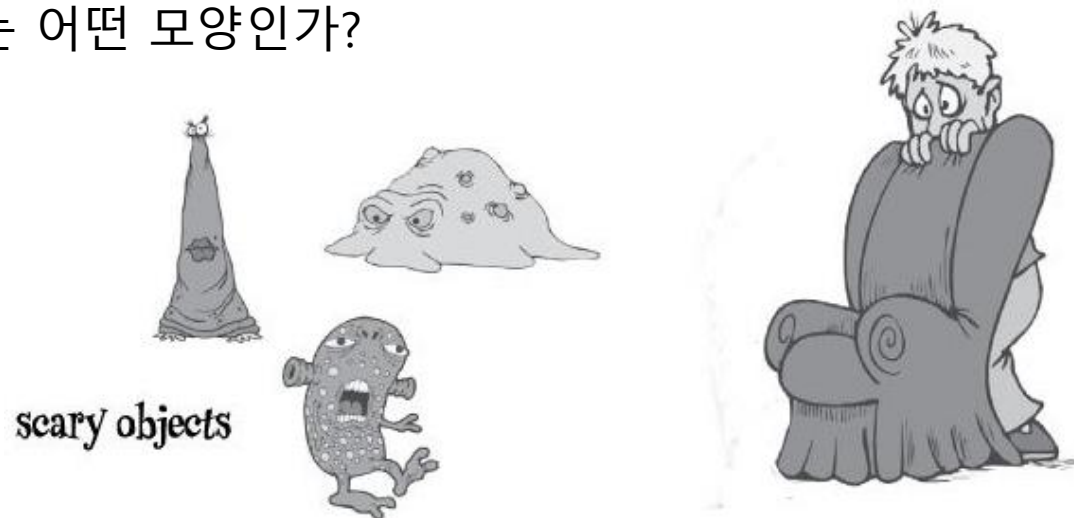


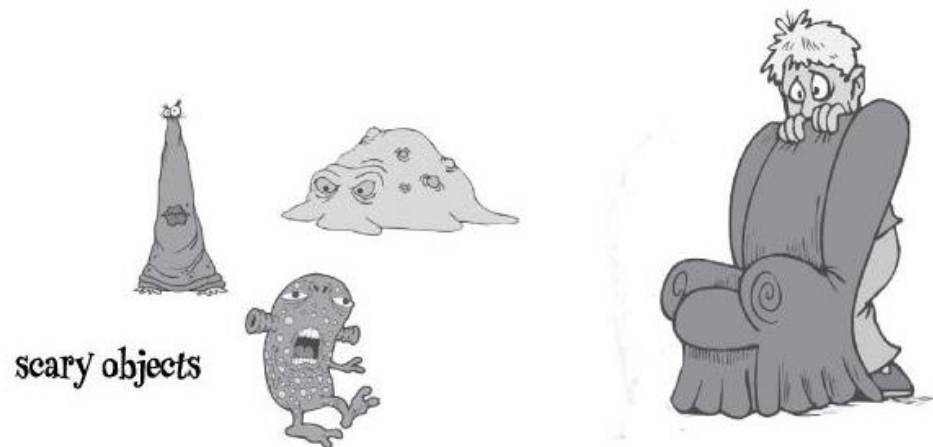
그런데 이렇게 하면 이상해진다:

Animal anim = new Animal();



새로운 **Animal()** 객체는 어떤 모양인가?





인스턴스 변수 값은 어떻게 되나?

클래스 중에는 인스턴스를 만들면 안 되는 것도 있다!

Wolf 객체, **Hippo** 객체, **Tiger** 객체를 생성하는 것은 의미가 있다.

그러나, 정확히 **Animal** 객체라는 것이 무엇인가?

어떤 모양인가? 색깔은? 사이즈는? 다리 개수는? ...

이 문제를 어떻게 처리할까?

우리는 상속과 다형성을 위해서는 **Animal** 클래스가 필요하다.

그러나 프로그래머는 **Animal** 객체 자체가 아니라 **Animal**의 덜 추상적인 서브클래스들을 객체화하고 싶어한다.

다행히도 어떤 클래스를 객체화하지 못하게 하는 방법이 있다.

✓ 즉, 그 타입에 "new"를 사용하지 못하도록 하는 방법.

⇒ 클래스를 "**abstract**"라고 마크한다:

```
abstract class Canine extends Animal {  
    public void roam() { }  
}
```

컴파일러는 추상 클래스를 객체화하지 않는다

```
abstract public class Canine extends Animal
{
    public void roam() { }
}
```

```
public class MakeCanine {
    public void go() {
        Canine c;
        c = new Dog();
        c = new Canine();
        c.roam();
    }
}
```

This is OK, because you can always assign a subclass object to a superclass reference, even if the superclass is abstract.

class Canine is marked abstract, so the compiler will NOT let you do this.

추상 클래스란 어느 누구도 그 클래스의 새로운 인스턴스를 만들 수 없다는 것을 의미한다.

```
File Edit Window Help BeamMeUp
% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
    c = new Canine();
          ^
1 error
```

추상 클래스는 확장되지 않는다면 거의 쓸모가 없고, 가치도 없고, 삶의 목적도 없다.
추상 클래스와 함께 런타임 때 실제로 일을 하는 녀석들은 바로 추상 클래스의 서브클래스 인스턴스(객체)들이다!

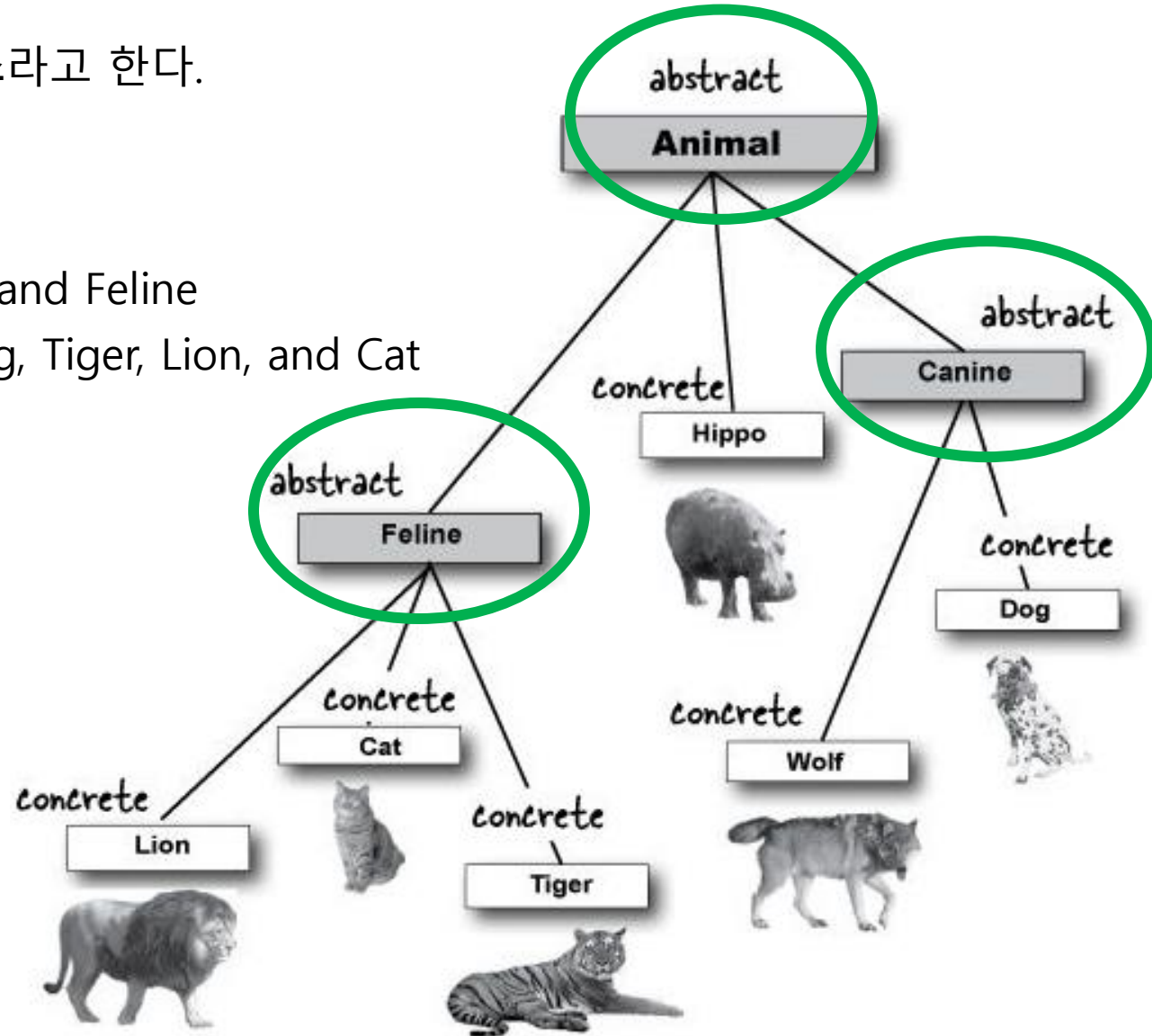
추상(Abstract) vs. 구상(Concrete)

추상이 아닌 클래스를 구상 클래스라고 한다.

Animal 상속 트리에서

추상 클래스: Animal, Canine, and Feline

구상 클래스: Hippo, Wolf, Dog, Tiger, Lion, and Cat



추상 메소드

클래스뿐만 아니라 메소드도 **abstract**로 만들 수 있다.

추상 메소드에는 몸통이 없다!

추상 메소드를 선언한다면 반드시 클래스도 추상으로 만들어야 한다.
즉, 추상 클래스가 아닌 클래스에 추상 메소드를 넣을 수는 없다.

```
public abstract void eat();
```

No method body!
End it with a semicolon.



모든 추상 메소드는 반드시 구현되어야 한다



추상 메소드를 구현하는 것은 마치 메소드를 오버라이딩하는 것과 흡사하다!

Polymorphism in action

ArrayList 클래스에 대해 모른다고 가정하고 **Dog** 객체를 집어넣기 위한 특별한 리스트 클래스를 직접 만들어야 한다고 하자.

- ✓ 추가되는 **Dog** 객체는 **Dog 배열** (Dog [])에 집어넣기로 하고 길이는 5로 한다.
- ✓ **Dog** 객체 5개가 다 찰 때까지 **add()** 메소드를 호출한다.
- ✓ 배열이 아직 가득 차지 않았다면 **Dog** 객체를 빈자리에 집어넣고 인덱스 번호를 1 증가시킨다.



```
public class MyDogList {
```

```
    private Dog [] dogs = new Dog[5];
```

```
    private int nextIndex = 0;
```

```
    public void add(Dog d) {
```

```
        if (nextIndex < dogs.length) {
```

```
            dogs[nextIndex] = d;
```

```
            System.out.println("Dog added at " + nextIndex);
```

```
            nextIndex++;
```

```
        }
```

```
    }
```

```
}
```

Use a plain old Dog array behind the scenes.

We'll increment this each time a new Dog is added.

If we're not already at the limit of the dogs array, add the Dog and print a message.

increment, to give us the next index to use

Cat 객체도 집어넣으려면 어떻게 해야 하나?

Cat 객체도 집어넣으려면?

몇 가지 옵션:

1. **MyCatList**라는 별도의 클래스를 만든다 => 너무 지저분하다.
2. 두 개의 다른 배열을 인스턴스 변수들로 유지하면서 두 개의 다른 **add()** 메소드 (**addCat(Cat c)** 와 **addDog(Dog d)**)를 가지는 **DogAndCatList**라는 하나의 클래스를 만든다.
3. 어떤 종류의 **Animal** 서브클래스도 받아들이는 이질적인 **AnimalList** 클래스를 만든다.
⇒ **Dog** 객체만이 아니라 모든 **Animal**을 받아들이는 포괄적인 클래스로 변경

Animal용 List 제작

version 2
MyAnimalList
Animal[] animals int nextIndex
add(Animal a)

```
public class MyAnimalList {  
  
    private Animal[] animals = new Animal[5];  
    private int nextIndex = 0;  
  
    public void add(Animal a) {  
        if (nextIndex < animals.length) {  
            animals[nextIndex] = a;  
            System.out.println("Animal added at " + nextIndex);  
            nextIndex++;  
        }  
    }  
}
```

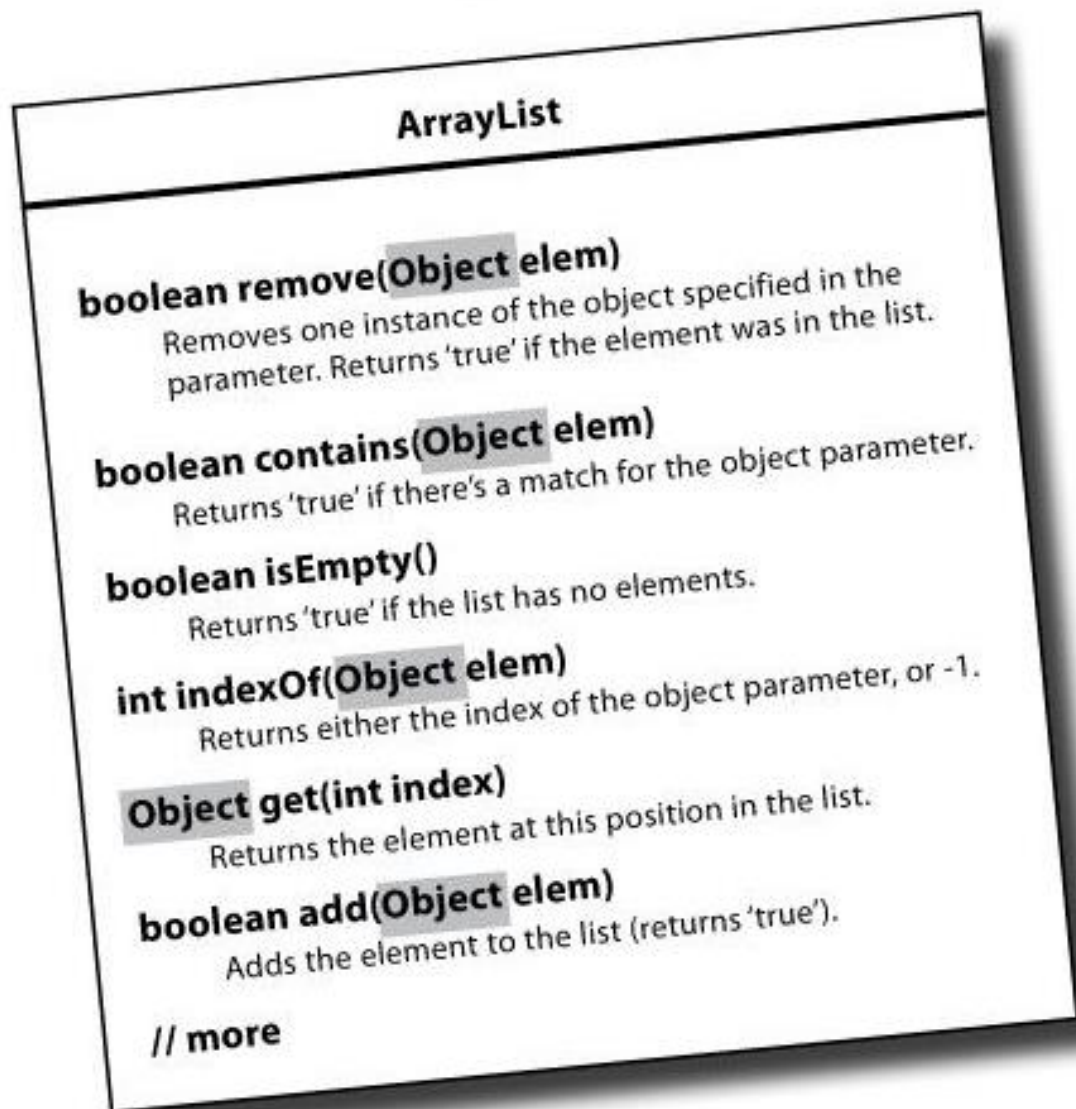
Animal 객체를 생성하는
것이 아니라 Animal이라는
새로운 array 객체를 만들고
있다.

```
public class AnimalTestDrive{  
    public static void main (String[] args) {  
        MyAnimalList list = new MyAnimalList();  
        Dog a = new Dog();  
        Cat c = new Cat();  
        list.add(a);  
        list.add(c);  
    }  
}
```

```
C:\Program Files\Java\jdk1.8.0_181\bin\java.exe ...  
Animal added at 0  
Animal added at 1  
  
Process finished with exit code 0
```

Animal이 아닌 객체는 어떻게 하나? 무엇이든 받아들이는 더 포괄적인 클래스를 만드는 것은 어떨까?

version
3



Remember those methods of **ArrayList**?

Look how the **remove**, **contains**, and **indexOf** method all use an object of type...

Object!

자바의 모든 클래스는 Object 클래스를 확장한다

자바에서 모든 클래스는 Object라는 클래스를 확장한 것이다.

- ✓ **Object** 클래스는 모든 클래스의 어머니, 즉 모든 것의 슈퍼클래스이다.
- ✓ 우리가 작성하는 모든 클래스도 **Object**를 확장한 것이다.

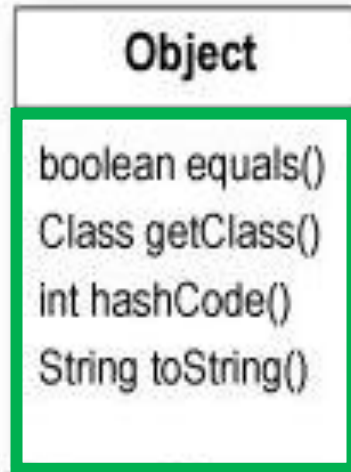
클래스를 만들 때 별도로 어떤 클래스를 확장하는지 지정하지 않으면 아래에 보이는 것처럼 자동으로 **Object**를 확장한 것으로 간주할 수 있다:

public class Dog extends Object { }

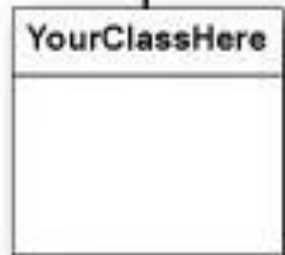
- ✓ **Dog**는 **Canine**을 확장한 클래스이다.
- ✓ 이 경우 컴파일러가 자동으로 **Canine**이 **Object**를 확장하게 해준다.
- ✓ 그런데 **Canine**은 또 **Animal**을 확장하고 있다.
- ✓ 별 상관없다. 컴파일러가 알아서 **Animal**이 **Object**를 확장하게 해준다.

“명시적으로 다른 클래스를 확장하지 않는 클래스는 자동으로 Object를 확장한 클래스로 정의한다.”

Object 클래스에는 무엇이 들어있을까?



*Just SOME of the methods
of class Object.*



*Every class you write inherits all the
methods of class Object. The classes
you've written inherited methods you
didn't even know you had.*

1. equals(Object o)

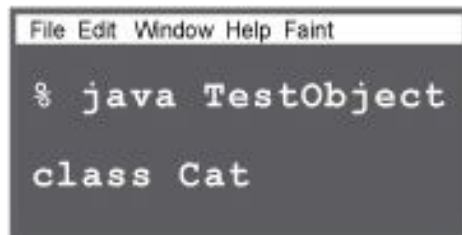
```
Dog a = new Dog();  
Cat c = new Cat();  
  
if (a.equals(c)) {  
    System.out.println("true");  
} else {  
    System.out.println("false");  
}
```



A terminal window with a menu bar (File, Edit, Window, Help, Stop) showing the output of the Java program. The prompt is '% java TestObject' and the output is 'false'.

2. getClass()

```
Cat c = new Cat();  
System.out.println(c.getClass());
```



A terminal window with a menu bar (File, Edit, Window, Help, Faint) showing the output of the Java program. The prompt is '% java TestObject' and the output is 'class Cat'.

3. hashCode()

```
Cat c = new Cat();  
System.out.println(c.hashCode());
```



A terminal window with a menu bar (File, Edit, Window, Help, Drop) showing the output of the Java program. The prompt is '% java TestObject' and the output is '8202111'.

4. toString()

```
Cat c = new Cat();  
System.out.println(c.toString());
```



A terminal window with a menu bar (File, Edit, Window, Help, LapseIntoComa) showing the output of the Java program. The prompt is '% java TestObject' and the output is 'Cat@7d277f'.

Object 타입의 다형적 레퍼런스 사용의 대가 ...

아규먼트/리턴 타입을 모두 Object로 만들면 엄청나게 유연한 메소드를 만들 수 있다.

- ✓ 그 전에 **Object** 타입의 레퍼런스를 사용할 때 생길 수 있는 **문제점**부터 살펴보자

아래처럼 **ArrayList<Dog>**에 **Dog** 객체를 집어넣는다면 나중에 꺼낼 때도 **Dog** 객체를 가져 올 수 있다:

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();  
Dog aDog = new Dog();  
myDogArrayList.add(aDog);  
Dog d = myDogArrayList.get(0);
```

← Make an ArrayList declared to hold Dog objects.

← Make a Dog.

← Add the Dog to the list.

← Assign the Dog from the list to a new Dog reference variable. (Think of it as though the get() method declares a Dog return type because you used ArrayList<Dog>.)

Dog 객체

그러나 아래처럼 ArrayList<Object>라고 선언했다면 어떨까?

이 경우 어떤 종류의 **Object** 객체도 다 집어넣을 수 있다:

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();  
Dog aDog = new Dog();  
myDogArrayList.add(aDog);
```

← Make an ArrayList declared to hold any type of Object.
← Make a Dog.
← Add the Dog to the list.
(These two steps are the same.)

위의 **ArrayList**에서 **Dog** 객체를 꺼내서 **Dog** 레퍼런스에 대입하려고 하면 어떻게 될까?

 `Dog d = myDogArrayList.get(0);`

NO!! Won't compile!!

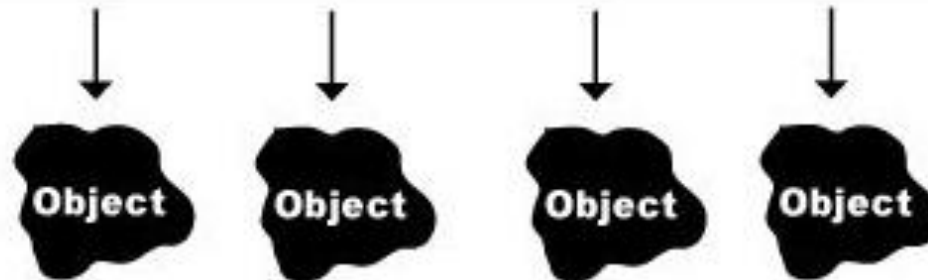
ArrayList<Object>에서 나오는 모든 것은 실제 객체가 무엇이든 간에 **Object** 타입으로 나오게 된다.

The objects go IN
as **SoccerBall**,
Fish, **Guitar**, and
Car.



ArrayList<Object>

But they come
OUT as though
they were of type
Object.



Objects come out of
an **ArrayList<Object>**
acting like they're
generic instances
of class **Object**. The
Compiler cannot
assume the object
that comes out is of
any type other than
Object.

개가 개처럼 행동하지 않으려고 할 때

모든 것을 다형적으로 **Object**로서 취급했을 때의 문제점:

- 객체의 본질적인 성질을 잃어버린 것처럼 보인다.
- The Dog appears to lose its dogness.



BAD
☹️

```
public void go() {  
    Dog aDog = new Dog();  
    Dog sameDog = getObject(aDog);  
}
```

```
public Object getObject(Object o) {  
    return o;  
}
```

↖ We're returning a reference to the same Dog, but as a return type of Object. This part is perfectly legal. Note: this is similar to how the get() method works when you have an ArrayList<Object> rather than an ArrayList<Dog>.

← This line won't work! Even though the method returned a reference to the very same Dog the argument referred to, the return type Object means the compiler won't let you assign the returned reference to anything but Object.

File Edit Window Help Remember

```
DogPolyTest.java:10: incompatible types  
found   : java.lang.Object  
required: Dog  
        Dog sameDog = getObject(aDog);  
1 error
```

← The compiler doesn't know that the thing returned from the method is actually a Dog, so it won't let you assign it to a Dog reference. (You'll see why on the next page.)

GOOD
😊

```
public void go() {  
    Dog aDog = new Dog();  
    Object sameDog = getObject(aDog);  
}
```

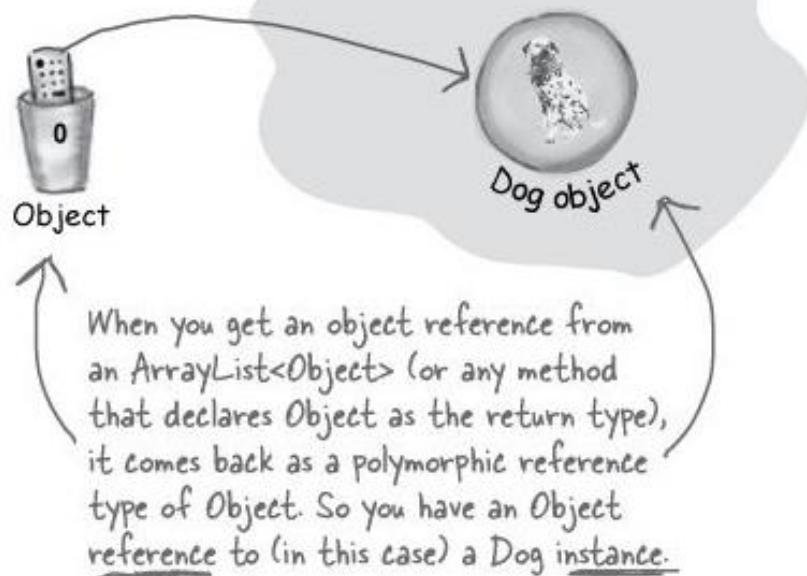
```
public Object getObject(Object o) {  
    return o;  
}
```

This works (although it may not be very useful, as you'll see in a moment) because you can assign **ANYTHING** to a reference of type `Object`, since every class passes the IS-A test for `Object`. Every object in Java is an instance of type `Object`, because every class in Java has `Object` at the top of its inheritance tree.

Dog 객체가 짚지 않는다!

객체가 **Object** 타입으로 선언된 변수에 의해 참조될 때 실제 객체 타입으로 선언된 변수(**sameDog**)에 할당될 수 없다!

⇒ 그렇다면 **Dog** 객체를 참조하기 위해 **Object** 레퍼런스 변수만 사용해야 하는가?



컴파일러에서는 `Object`라고 생각하
지만 실제로는 `Dog`인 객체에 대해
`Dog` 메소드를 호출해 보면 어떨까?

```
Object o = al.get(index);
```

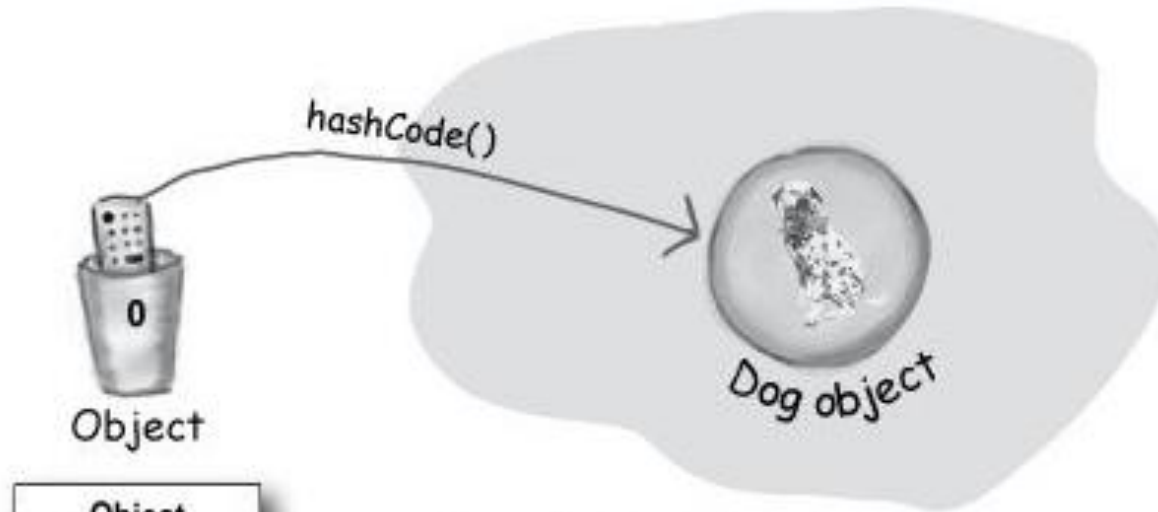
```
int i = o.hashCode();
```

This is fine. Class `Object` has a `hashCode()` method, so you can call that method on ANY object in Java.

Won't compile! → `o.bark();`

Can't do this!! The `Object` class has no idea what it means to `bark()`. Even though YOU know it's really a `Dog` at that index, the compiler doesn't.

컴파일러에서는 어떤 메소드를 호출할지를 결정할 때 객체 타입이 아니라 **레퍼런스 타입**의 클래스를 체크한다.



Object
<code>equals()</code>
<code>getClass()</code>
<code>hashCode()</code>
<code>toString()</code>

The method you're calling on a reference **MUST** be in the class of that reference type. Doesn't matter what the actual object is.

`o.hashCode()` ;

OK

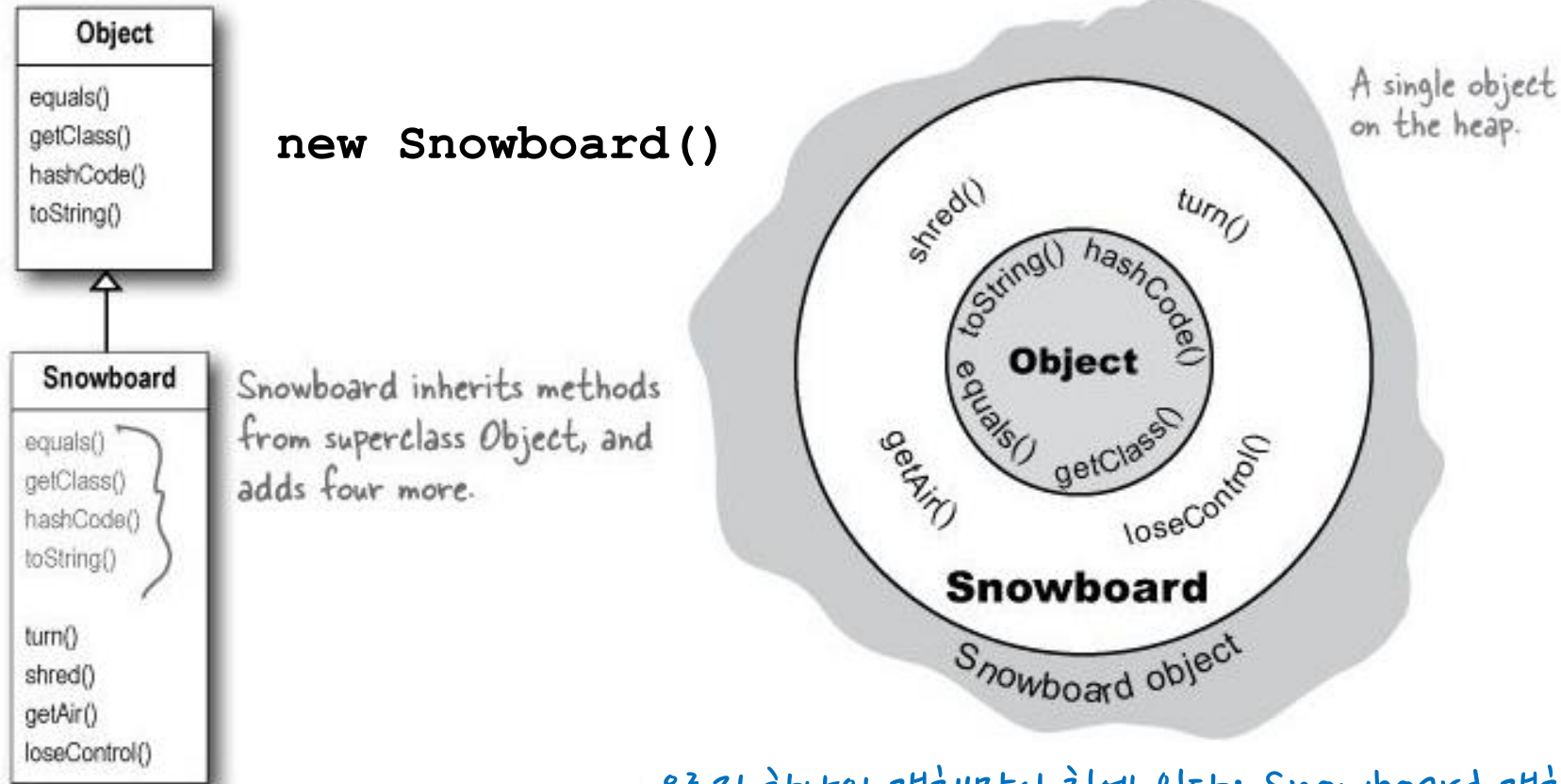
The "o" reference was declared as type `Object`, so you can call methods only if those methods are in class `Object`.



내부 Object를 만나보자

객체는 슈퍼클래스로부터 상속받은 모든 것을 포함한다.

즉 모든 객체는 또한 **Object** 클래스의 인스턴스이다.

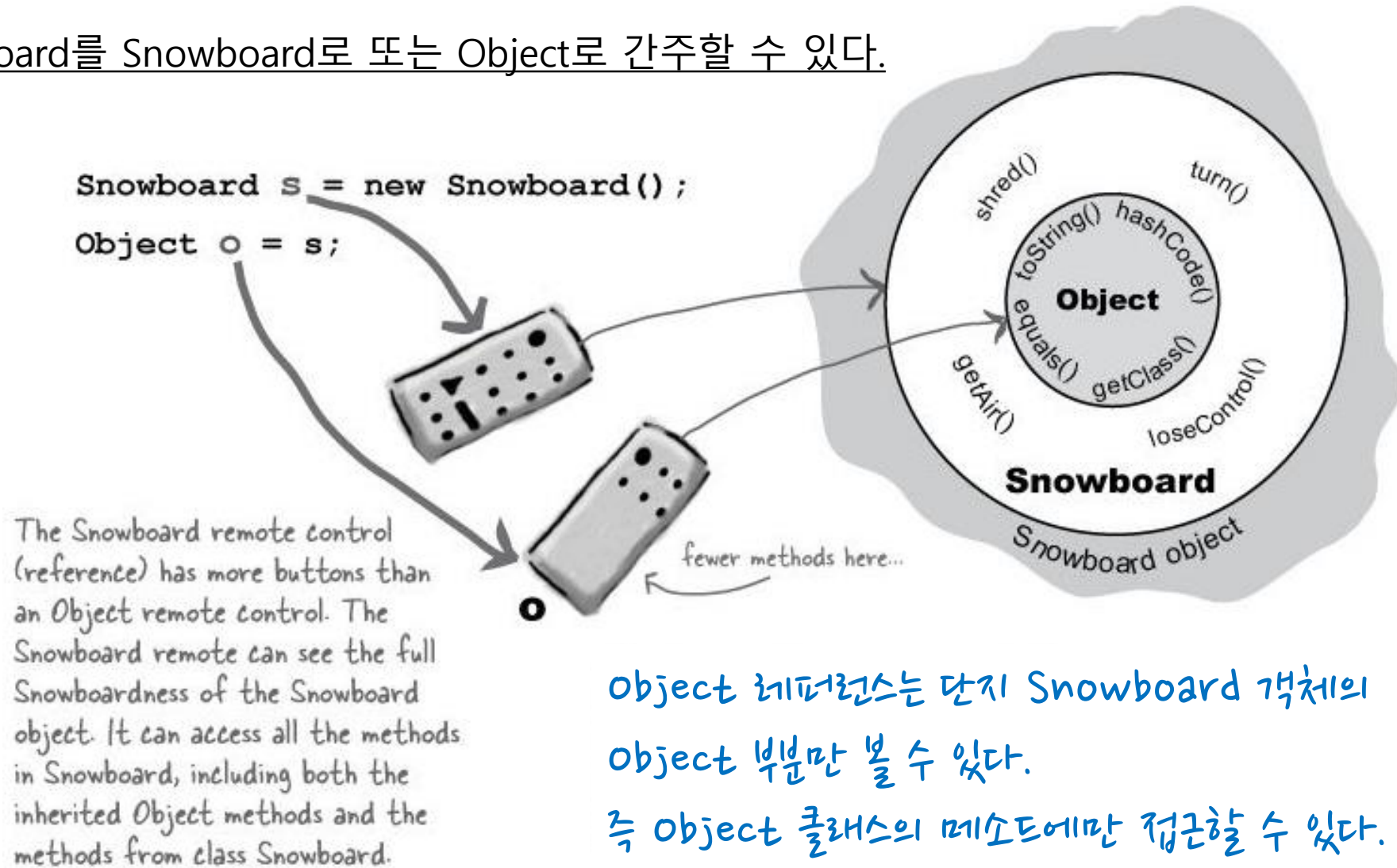


오로지 하나의 객체만이 힙에 있다: Snowboard 객체.

그러나 그 안에는 Snowboard 클래스 부분과 Object 클래스 부분이 모두 들어있다.

'Polymorphism' means 'many forms'

Snowboard를 Snowboard로 또는 Object로 간주할 수 있다.



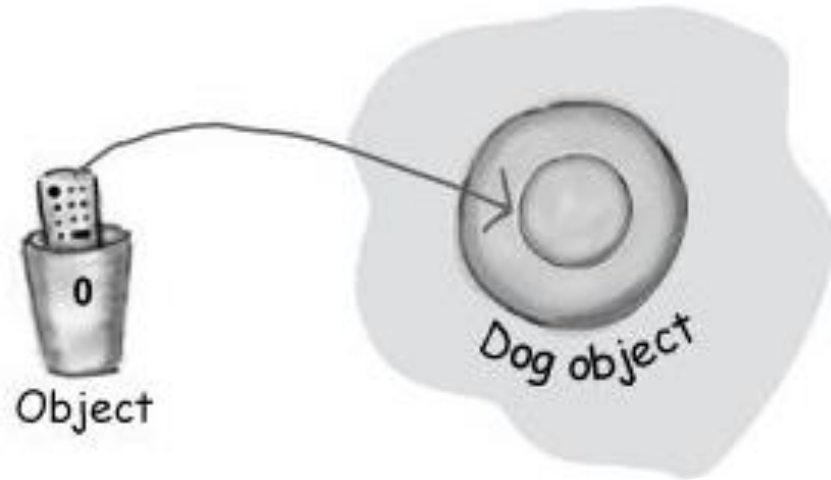


Wait a minute... what good is a Dog if it comes out of an `ArrayList<Object>` and it can't do any Dog things? There's gotta be a way to get the Dog back to a state of Dogness...

I hope it doesn't hurt. And what's so wrong with staying an Object? OK, I can't fetch, sure, but I can give you a real nice hashCode.

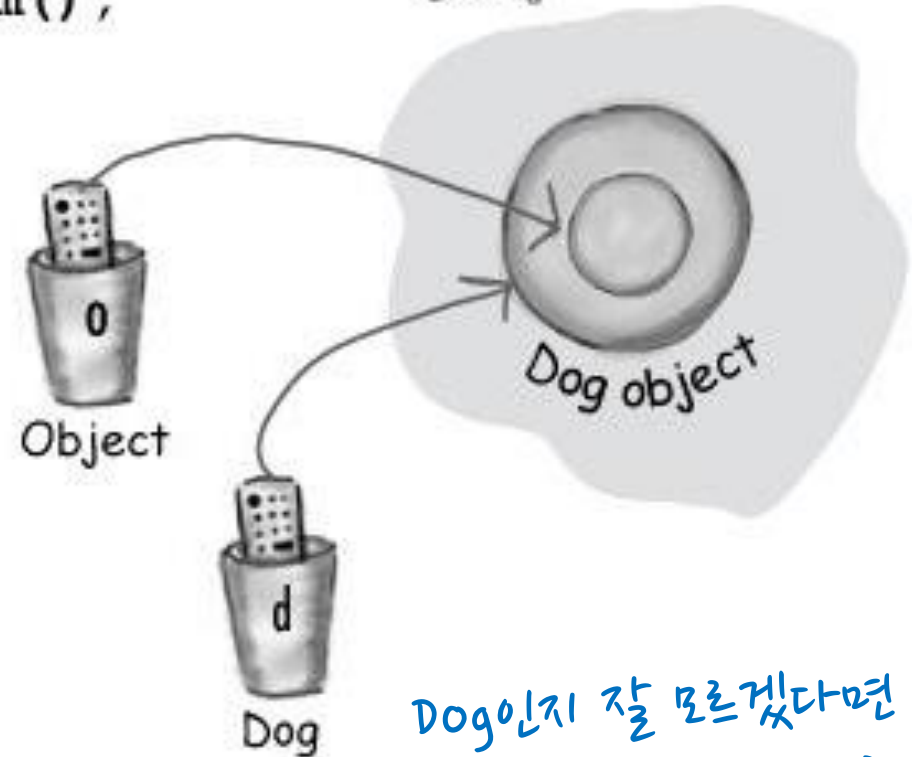
↑
Cast the so-called 'Object' (but we know he's actually a Dog) to type Dog, so that you can treat him like the Dog he really is.

객체 레퍼런스를 실제 타입으로 캐스팅하는 방법



Dog 객체이긴 한데 Object의
메소드에만 접근할 수 있다!

```
Object o = al.get(index);  
Dog d = (Dog) o; ← cast the Object back to  
d.roam();           a Dog we know is there.
```



Dog인지 잘 모르겠다면
instanceof 연산자를
써서 확인하면 된다.

```
if (o instanceof Dog) {  
    Dog d = (Dog) o;  
}
```

계약을 변경해야 한다면 어떻게 해야 할까?

Dog 클래스가 Dog가 뭔지를 정의해주는 유일한 계약은 아니다.

- 모든 슈퍼클래스로부터 메소드들을 상속받기 때문에.

Dog 클래스는 일종의 계약을 정의해준다(아래와 같은 계약들도 포함):

Canine 클래스의 모든 것이 계약에 포함된다.

Animal 클래스의 모든 것이 계약에 포함된다.

Object 클래스의 모든 것이 계약에 포함된다.

Dog 클래스를 애완동물 상점 프로그램(PetShop)에서 사용하고 싶다면 어떻게 될까?

- 현재의 **Dog** 클래스에는 애완동물의 행동에 대한 메소드가 없다:
beFriendly(), play() 등.

PetShop 프로그램에서 기존의 클래스를 재사용하기 위한 설계 방법

이제 **Dog** 클래스를 만드는 프로그래머 입장에서 생각해보자.

- 그냥 **Dog** 클래스에 새로운 메소드를 집어넣으면 되지 않을까?
- 다른 코드에서 **Dog** 객체를 호출하는 기존의 메소드는 건드리지 않으니까 별 문제 없을 것 같은데...

첫 번째 옵션

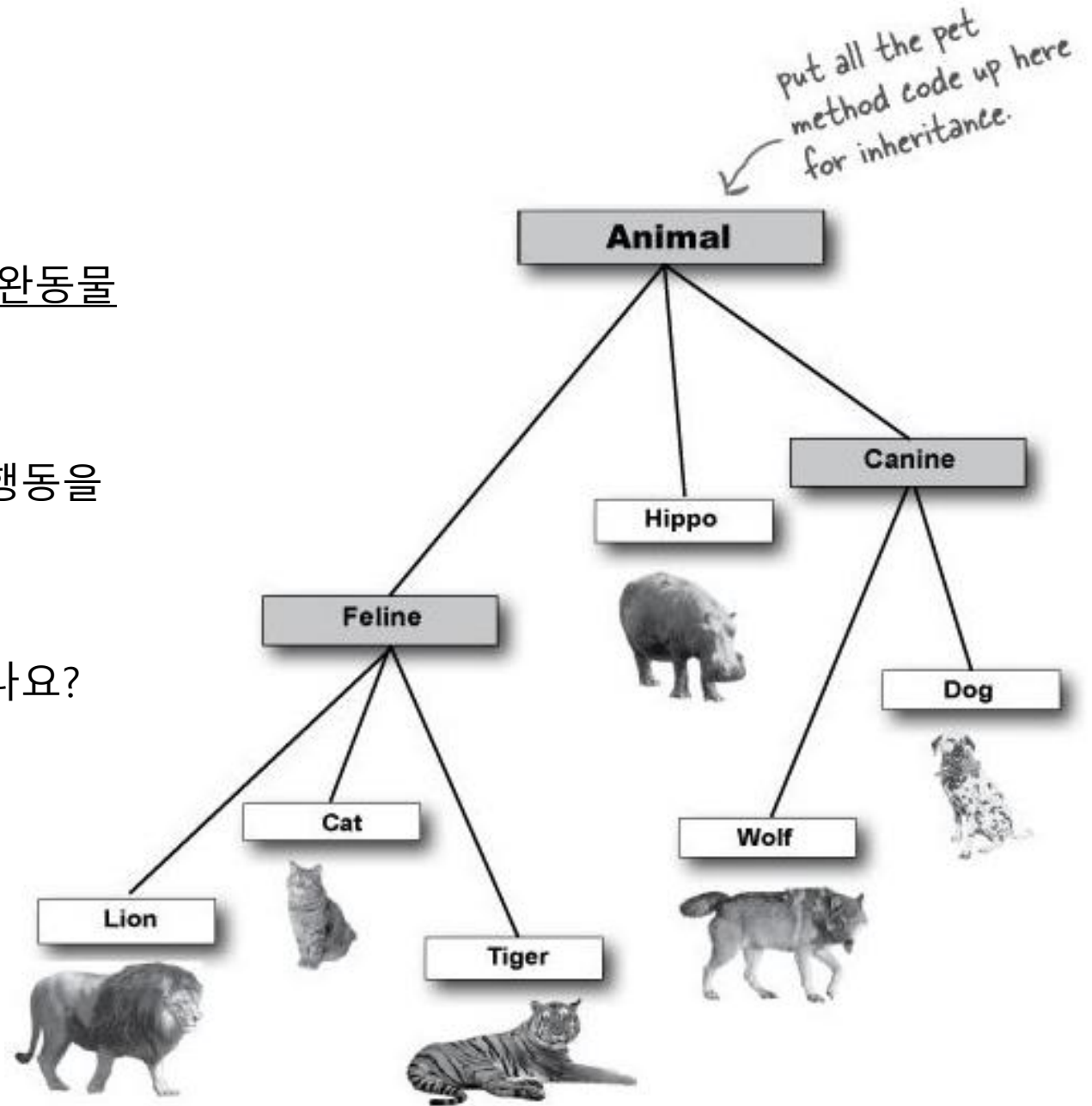
가장 간단한 방법으로 **Animal** 클래스에 애완동물이 하는 행동(메소드)을 집어넣는다.

장점:

모든 동물 객체들이 바로 애완동물의 행동을 상속받는다.

단점:

애완동물 상점에서 하마를 본 적이 있나요?
사자는? 늑대는?

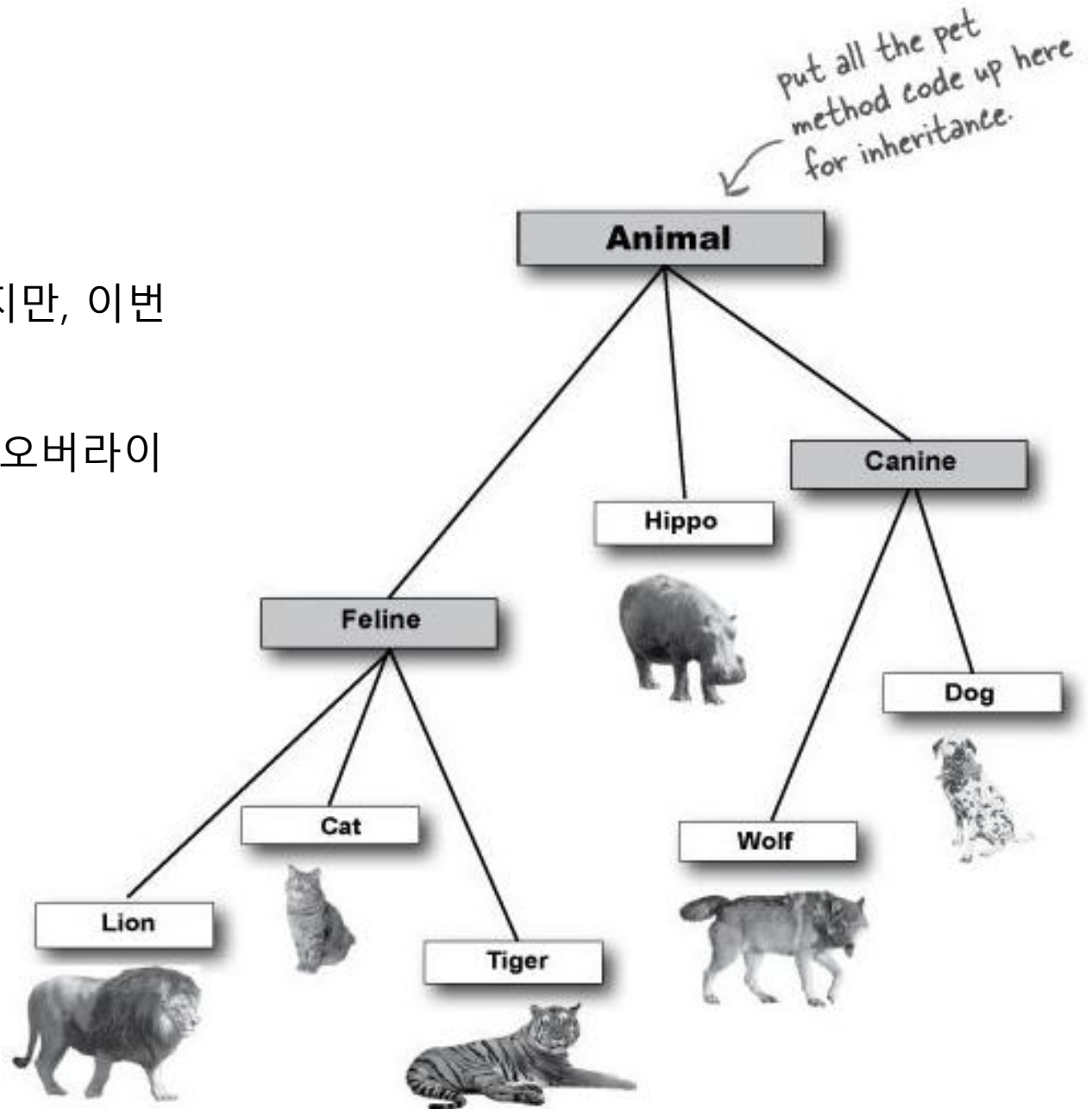


두 번째 옵션

Animal 클래스에 **Pet** 메소드들을 두긴 하지만, 이번에는 그 메소드들을 **abstract**으로 만든다.

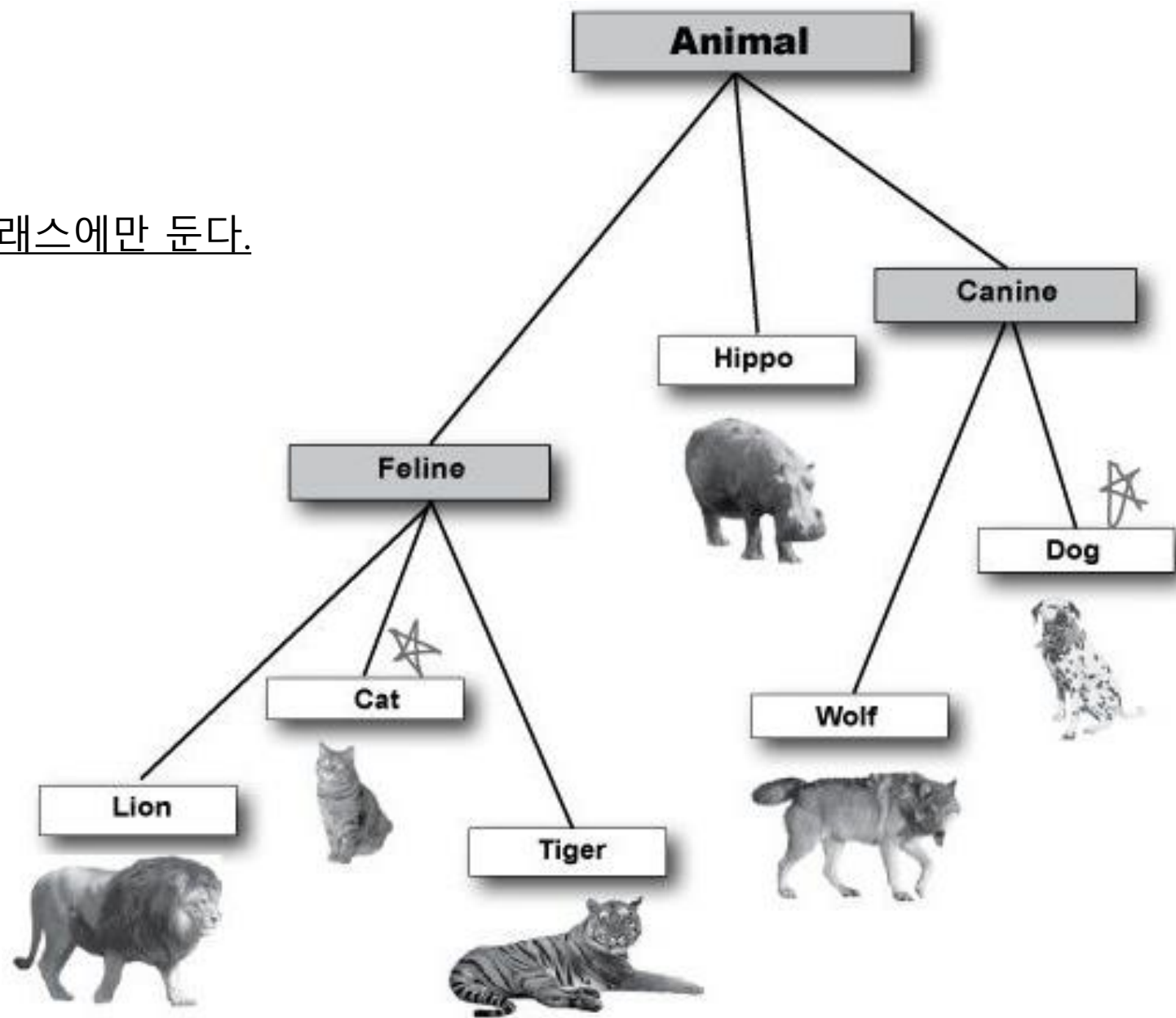
모든 **Animal** 서브클래스가 그들을 무조건 오버라이딩해야 한다!

What a waste of time!

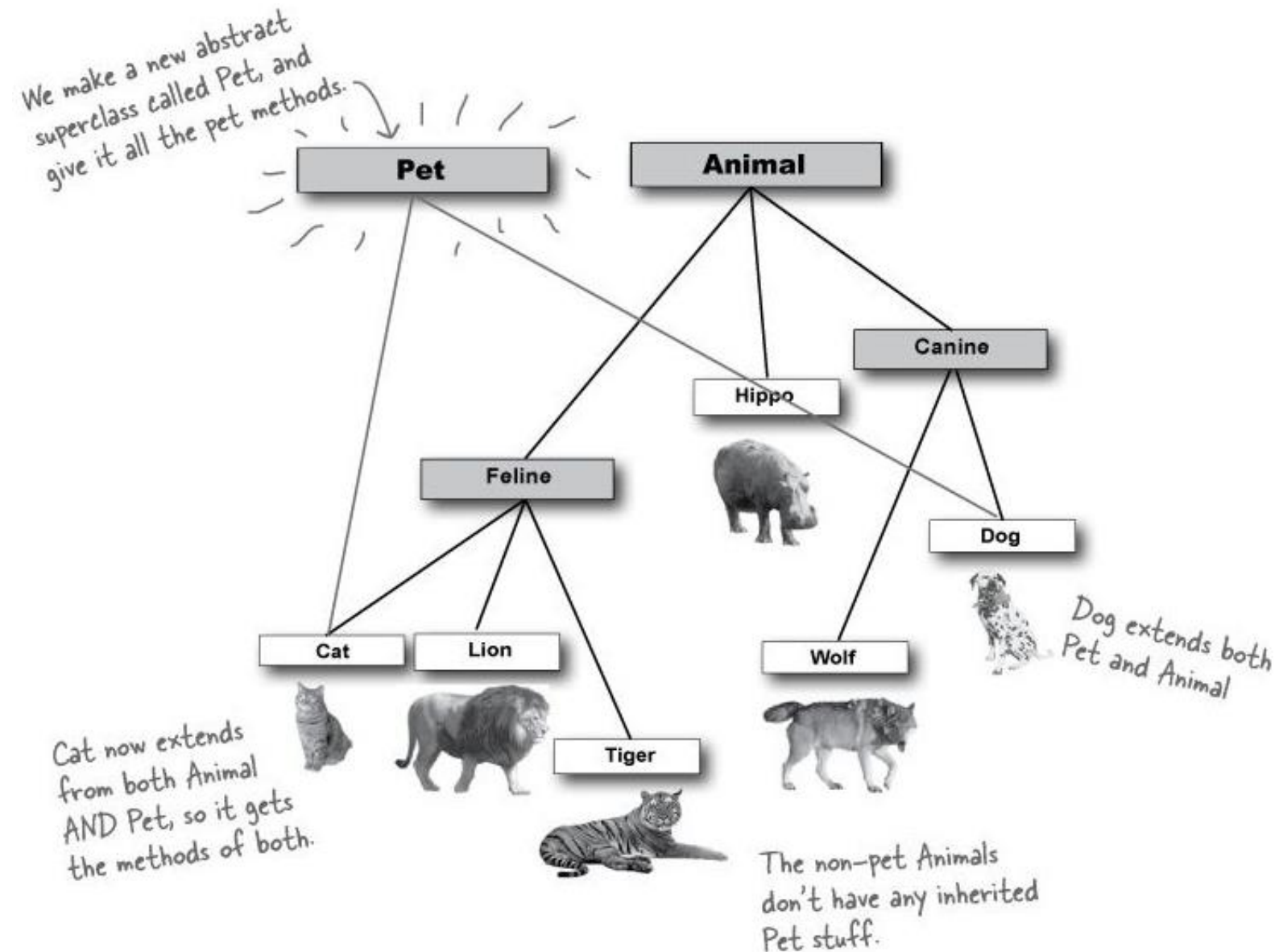


세 번째 옵션

Pet 메소드를 그걸 사용할 클래스에만 둔다.



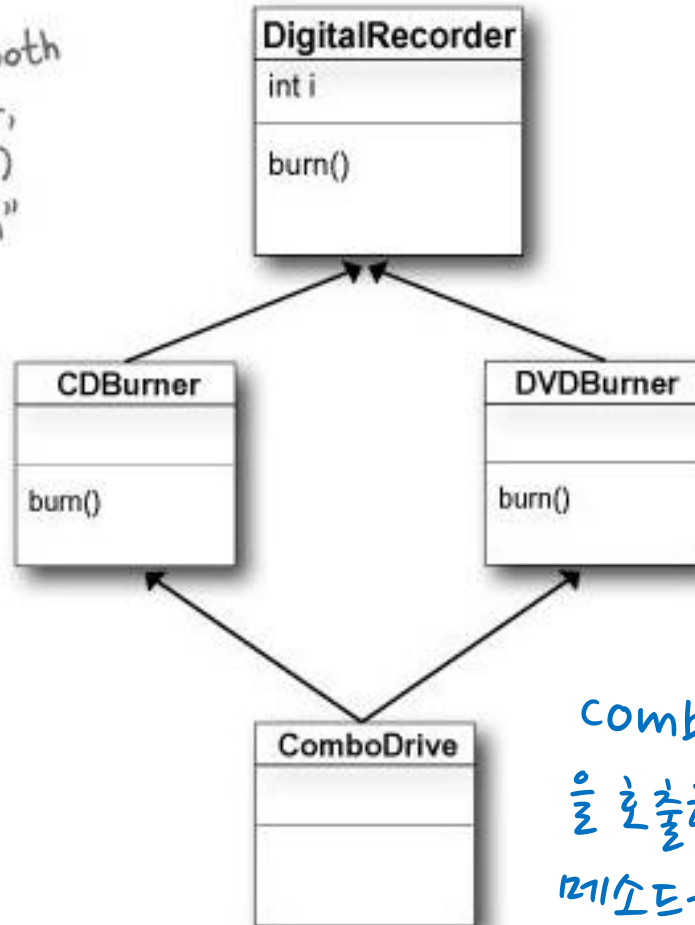
두 개의 슈퍼클래스가 필요한 것처럼 보인다



다중 상속?? 죽음의 다이아몬드!!

"multiple inheritance" => Deadly Diamond of Death

CDBurner and DVDBurner both inherit from DigitalRecorder, and both override the burn() method. Both inherit the "i" instance variable.



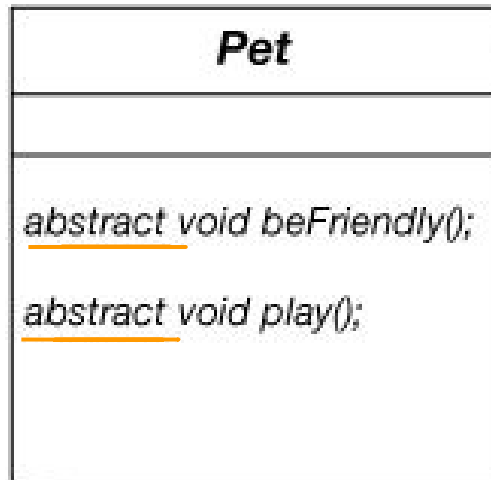
Imagine that the "i" instance variable is used by both CDBurner and DVDBurner, with different values. What happens if ComboDrive needs to use both values of "i"?

comboDrive에서 burn()
을 호출할 때 어느 burn()
메소드가 호출될까?

해결책은 인터페이스다!

자바의 인터페이스가 DDD 문제를 비켜가는 방법은 의외로 간단하다:

- 모든 메소드를 **abstract**로 만드는 것이다!
⇒ 서브클래스들은 그 추상 메소드를 구현해야만 한다.
(첫 번째로 만나는 구상 클래스에 의해 구현되어야 한다)



**A Java interface is like a
100% pure abstract class.**

*All methods in an interface are
abstract, so any class that IS-A
Pet MUST implement (i.e. override)
the methods of Pet.*

To **DEFINE** an interface:

```
public interface Pet {...}
```

↖ Use the keyword "interface"
instead of "class"

To **IMPLEMENT** an interface:

```
public class Dog extends Canine implements Pet {...}
```

↖ Use the keyword "implements" followed
by the interface name. Note that
when you implement an interface you
still get to extend a class

Making and Implementing the Pet interface

You say 'interface' instead of 'class' here

```
public interface Pet {  
    public abstract void beFriendly();  
    public abstract void play();  
}
```

All interface methods are abstract, so they **MUST** end in semicolons. Remember, they have no body!

Dog IS-A Animal
and Dog IS-A Pet

반드시 구현해야 한다!

```
public class Dog extends Canine implements Pet {
```

You say 'implements' followed by the name of the interface.

```
    public void beFriendly() {...}
```

```
    public void play() {...}
```

You SAID you are a Pet, so you **MUST** implement the Pet methods. It's your contract. Notice the curly braces instead of semicolons.

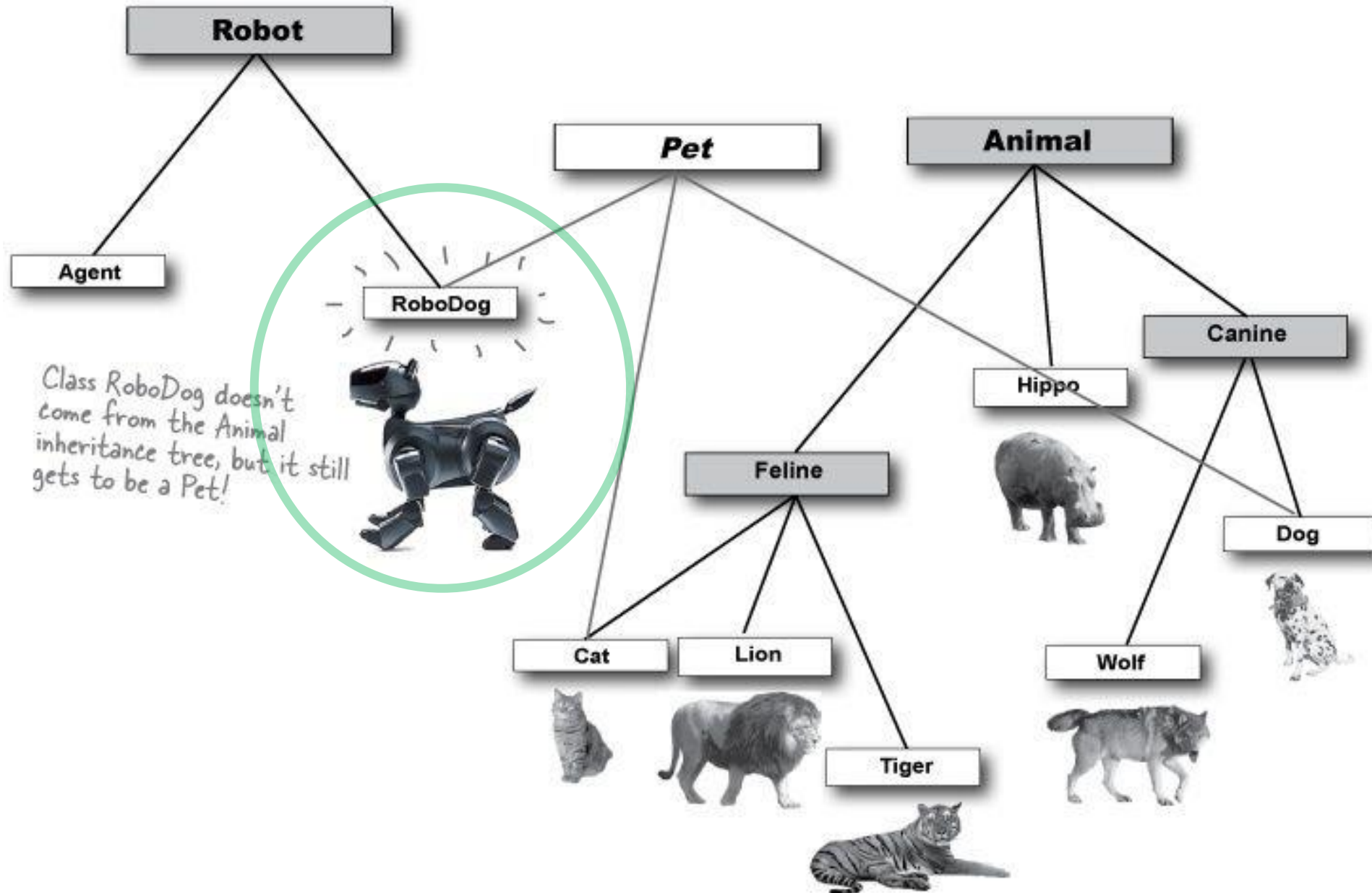
```
    public void roam() {...}
```

```
    public void eat() {...}
```

These are just normal overriding methods.

```
}
```


다른 상속 트리의 클래스에서도 인터페이스를 구현할 수 있다!



한 클래스에서 여러 개의 인터페이스를 구현할 수 있다

Dog object **IS-A Canine**, **IS-A Animal**, and **IS-A Object** => 모두 상속을 통해 성립된다.

Dog IS-A Pet

⇒ 인터페이스 구현을 통해 가능하다

⇒ **Dog**는 여러 개의 다른 인터페이스도 구현할 수 있다:

```
public class Dog extends Animal implements  
    Pet, Saveable, Paintable { ... }
```

슈퍼클래스에 있는 메소드를 호출하는 방법

```
abstract class Report {  
    void runReport() {  
        // set-up report  
    }  
    void printReport() {  
        // generic printing  
    }  
}  
  
class BuzzwordsReport extends Report {  
  
    void runReport() {  
        super.runReport();  
        buzzwordCompliance();  
        printReport();  
    }  
    void buzzwordCompliance() {...}  
}
```

← superclass version of the method does important stuff that subclasses could use

← call superclass version, then come back and do some subclass-specific stuff

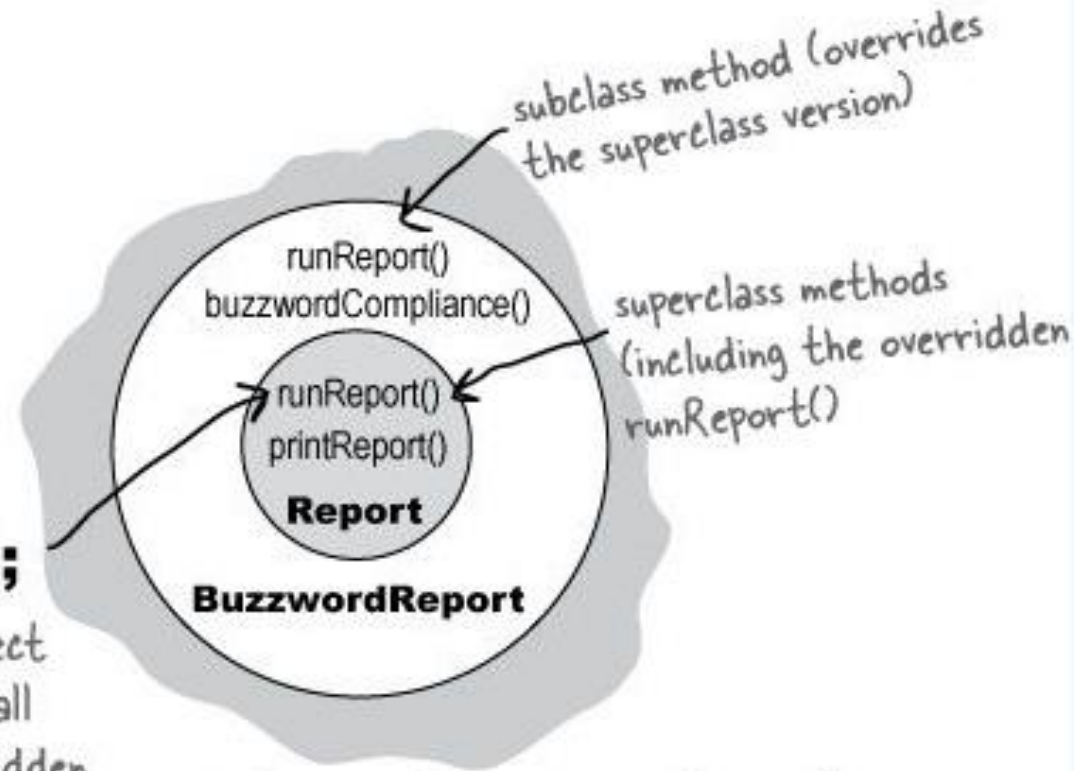
If method code inside a
BuzzwordReport subclass says:

super.runReport();

the runReport() method inside
the superclass Report will run

super.runReport();

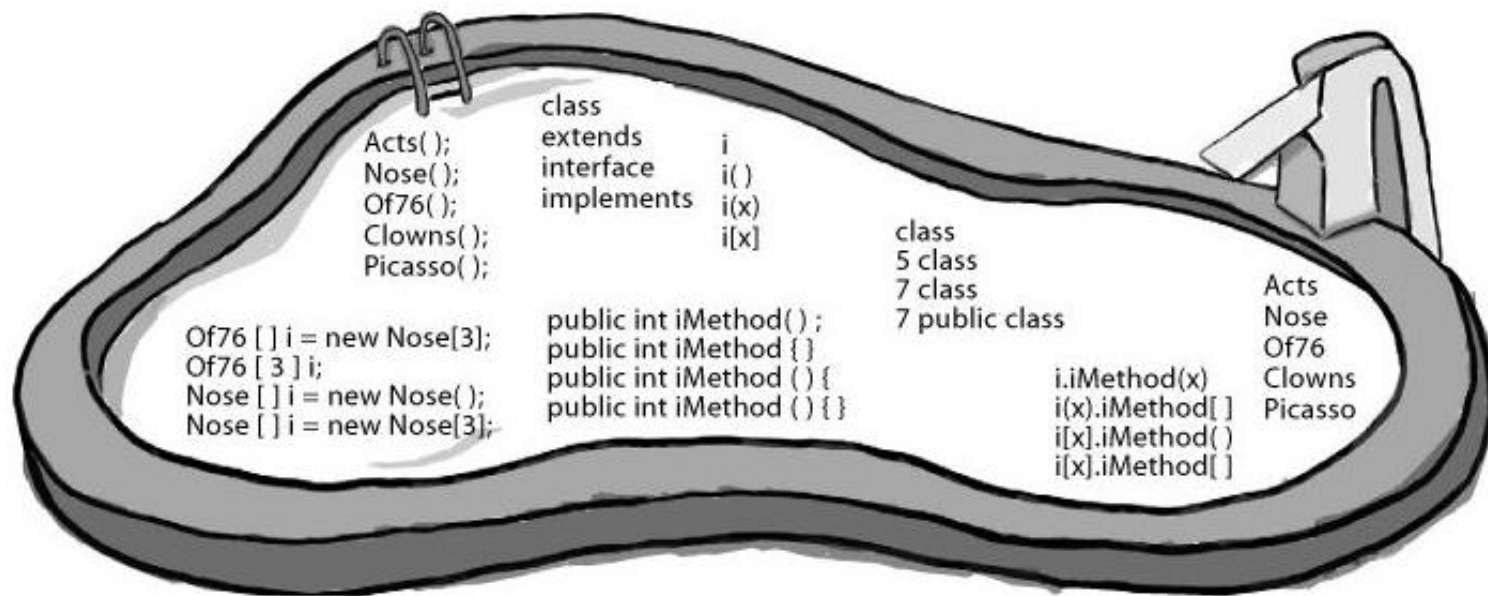
A reference to the subclass object
(BuzzwordReport) will always call
the subclass version of an overridden
method. That's polymorphism.
But the subclass code can call
super.runReport() to invoke the
superclass version.



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.

실습과제 8-1 Pool Puzzle

```
_____ Nose {  
    _____  
}  
  
abstract class Picasso implements _____ {  
    _____  
    return 7;  
}  
  
class _____ { }  
  
class _____ {  
    _____  
    return 5;  
}  
  
public _____ extends Clowns {  
  
    public static void main(String [] args) {  
        _____  
        i[0] = new _____  
        i[1] = new _____  
        i[2] = new _____  
        for(int x = 0; x < 3; x++) {  
            System.out.println(_____  
                + " " + _____.getClass( ) );  
        }  
    }  
}
```



Output

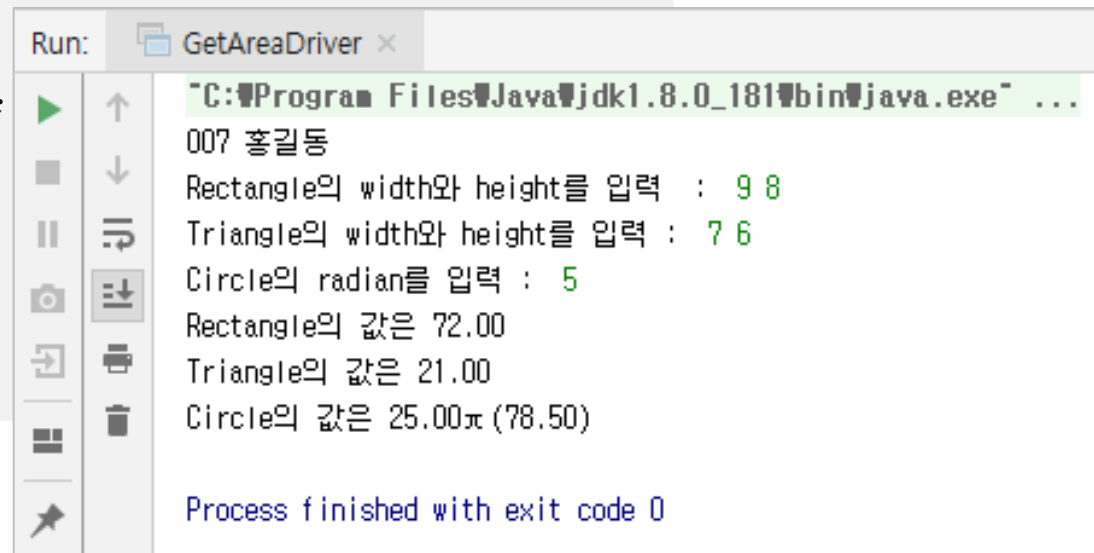
```
File Edit Window Help BeAfraid  
% java _____  
5 class Acts  
7 class Clowns  
_____ Of76
```

실습과제 8-2

다형성을 이용하여 **Rectangle**, **Triangle**, **Circle**의 면적(area)을 구하시오. 면적 계산에 요구되는 parameter 값들은 Scanner를 통해 읽어 들이시오. 예) **Rectangle**의 경우: width, height.
또한, nicely formatted output을 위해 toString 메소드를 사용하시오.

참고:

```
public static double calcArea(Shape s) {  
    double area = 0.0;  
    if (s instanceof Rectangle) {  
        int w = ((Rectangle) s).getWidth();  
        int h = ((Rectangle) s).getHeight();  
        area = (double) (w * h);  
    }  
    ... // 다른 도형들의 면적을 구한다.  
    return area;  
}
```



```
Run: GetAreaDriver x  
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...  
007 홍길동  
Rectangle의 width와 height를 입력 : 9 8  
Triangle의 width와 height를 입력 : 7 6  
Circle의 radian를 입력 : 5  
Rectangle의 값은 72.00  
Triangle의 값은 21.00  
Circle의 값은 25.00π (78.50)  
  
Process finished with exit code 0
```

참고자료: toString()

자바에서 **int**나 **char** 등의 기본형을 제외한 나머지 모든 것은 객체이다.

자바의 모든 객체(클래스)는 자동으로 **Object**로부터 상속받는다.

이 **Object** 클래스에는 모든 다른 객체들에게 공통적으로 필요한 메소드들이 정의되어 있고, 다른 모든 자바 클래스에서는 이 메소드를 오버라이딩하여 사용하도록 약속되어 있다.

String toString() 메소드도 그 중의 하나이다.

System.out.println과 같은 메소드의 인자로 객체를 넣으면 자동으로 **toString()** 메소드가 실행됨을 알 수 있다.

toString() 메소드를 오버라이딩 하지 않은 객체를 출력해 보면 클래스의 이름과 그 객체의 해쉬 코드 값뿐이다.

참고자료: toString()

```
package tostring;

class Object1 {
    private Object value1;

    public Object1(Object value1) { this.value1 = value1; }
}

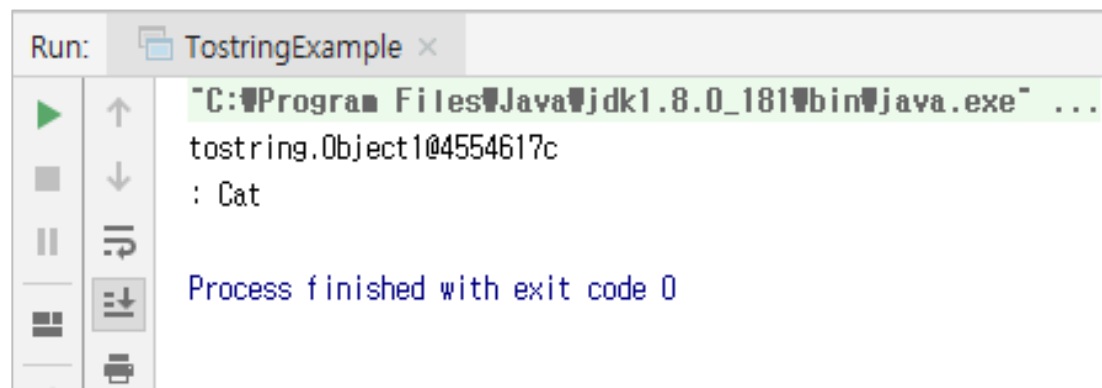
class Object2 {
    private Object value2;

    public Object2(Object value2) { this.value2 = value2; }

    public String toString() { return ":" + value2; }
}

public class TostringExample {

    public static void main(String[] args) {
        Object1 obj1 = new Object1( value1: "Dog");
        Object2 obj2 = new Object2( value2: "Cat");
        System.out.println(obj1.toString());
        System.out.println(obj2.toString());
    }
}
```



```
Run: TostringExample x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
tostring.Object1@4554617c
: Cat

Process finished with exit code 0
```

