7장. 상속과 다형성

상속에 대해 알아봅니다.
상속과 클래스 계층구조
메서드 오버라이딩

객체마을에서의 더 나은 삶

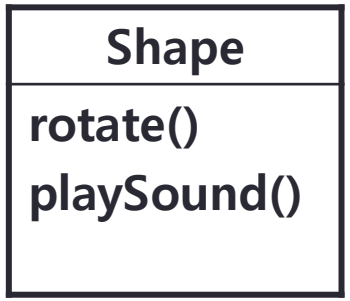
2장 상속 다시 보기

1. 네 클래스에 공통적으로 들어있는 것을 찾아낸다.

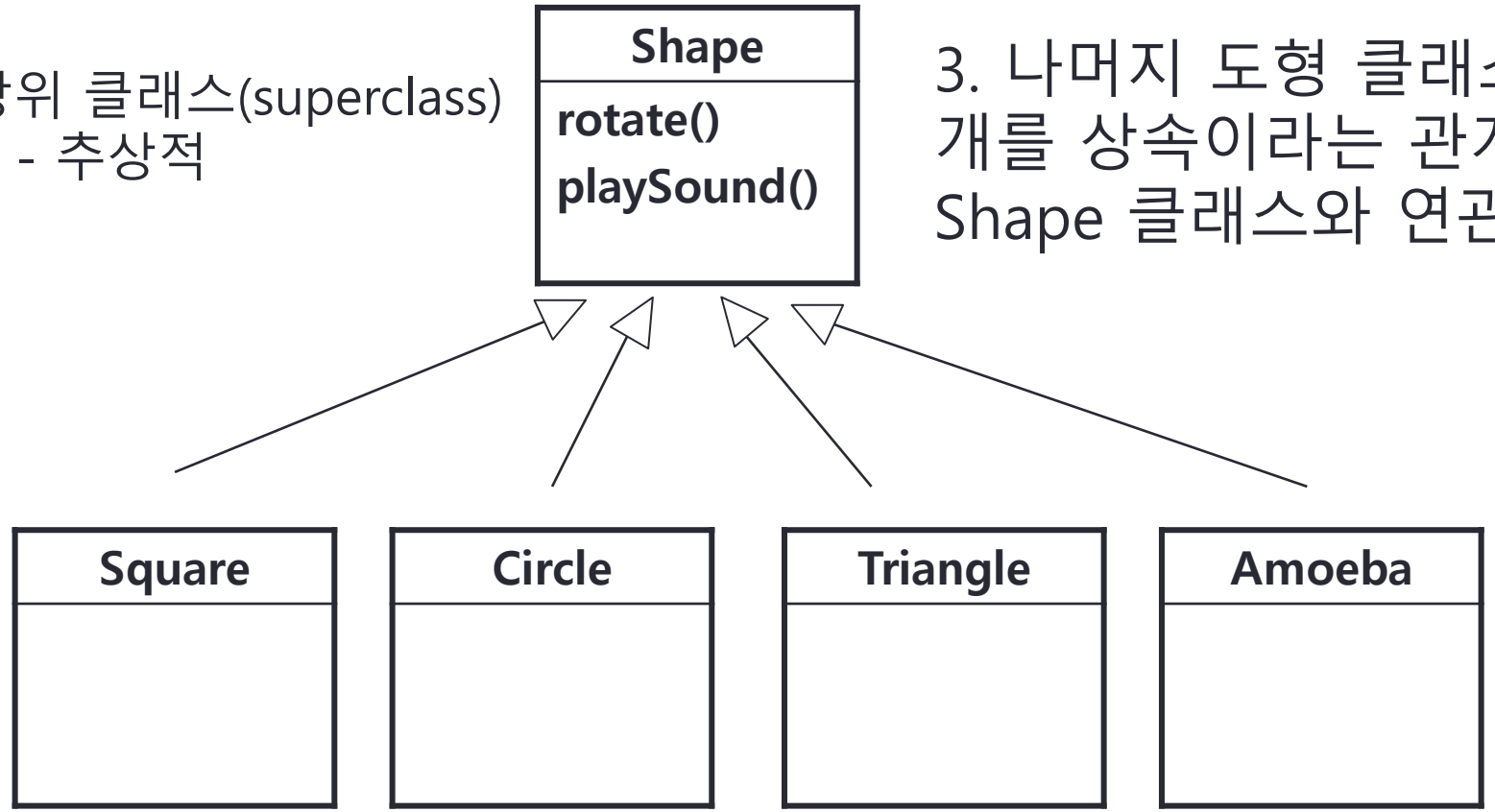
Square	Circle	Triangle	Amoeba
rotate() playSound()	rotate() playSound()	rotate() playSound()	rotate() playSound()

p.210

2. 모두 도형이고 회전(rotate()) 및 사운드 재생 (playSound()) 기능이 있으므로 공통적인 기능을 모두 뽑아서 Shape이라는 클래스를 만든다.



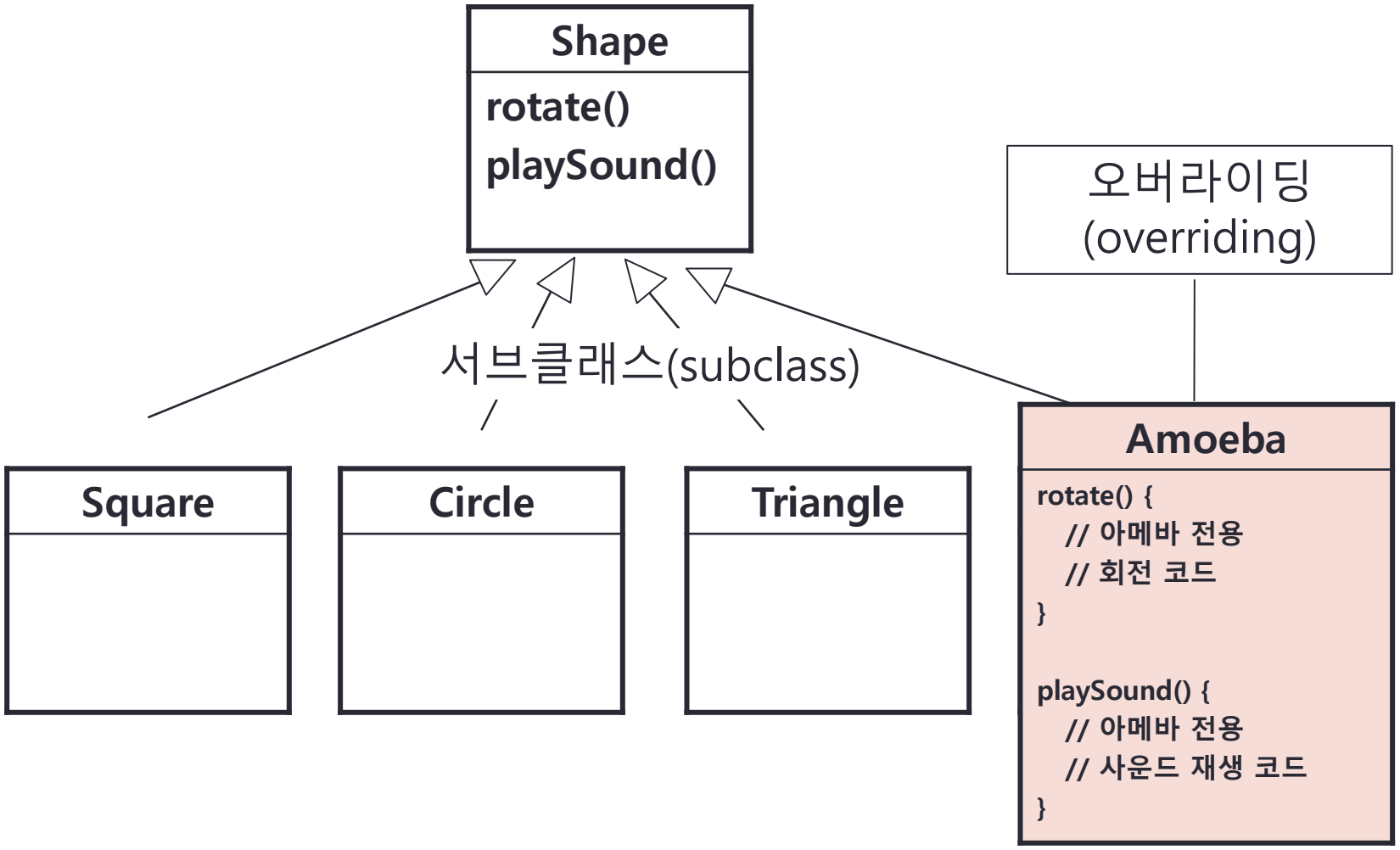
상위 클래스(superclass)
- 추상적



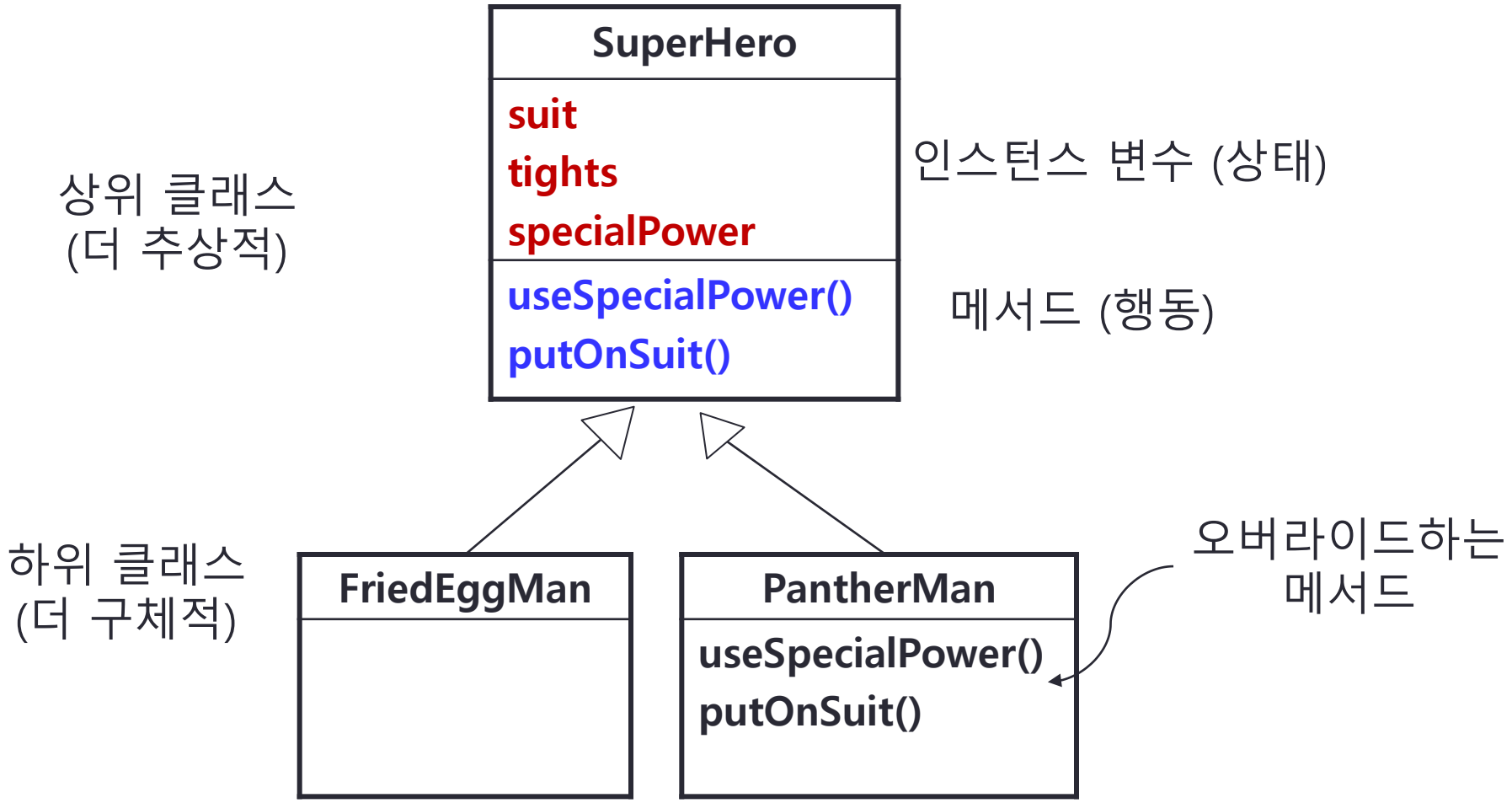
3. 나머지 도형 클래스 네 개를 상속이라는 관계로 Shape 클래스와 연관시킴

하위 클래스(subclass) - 구체적

아메바의 메서드는 어떻게 처리할 수 있을까요?

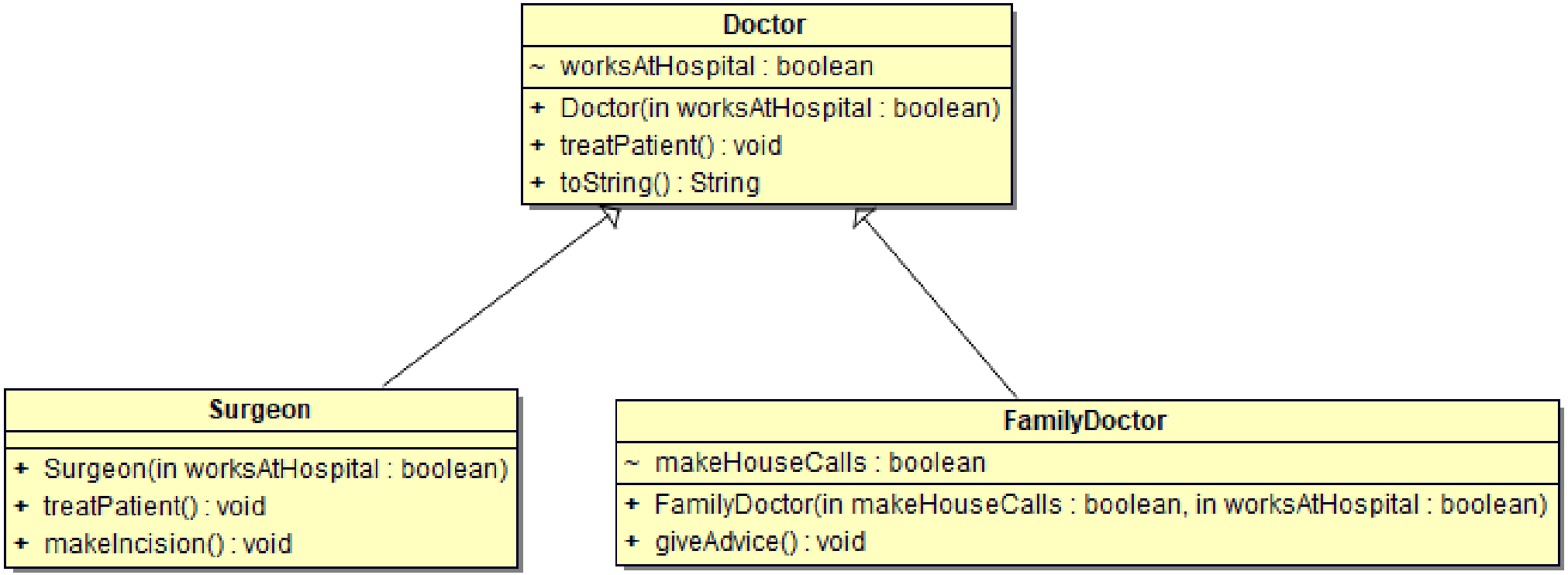


상속의 이해


SuperHero
suit
tights
specialPower
useSpecialPower()
putOnSuit()
인스턴스 변수 (상태)
메서드 (행동)
상위 클래스
(더 추상적)
하위 클래스
(더 구체적)
FriedEggMan
PantherMan
useSpecialPower()
putOnSuit()
오버라이드하는
메서드

p.212

상속 예제


Doctor
~ worksAtHospital : boolean
+ Doctor(in worksAtHospital : boolean)
+ treatPatient() : void
+ toString() : String
Surgeon
+ Surgeon(in worksAtHospital : boolean)
+ treatPatient() : void
+ makeIncision() : void
FamilyDoctor
~ makeHouseCalls : boolean
+ FamilyDoctor(in makeHouseCalls : boolean, in worksAtHospital : boolean)
+ giveAdvice() : void

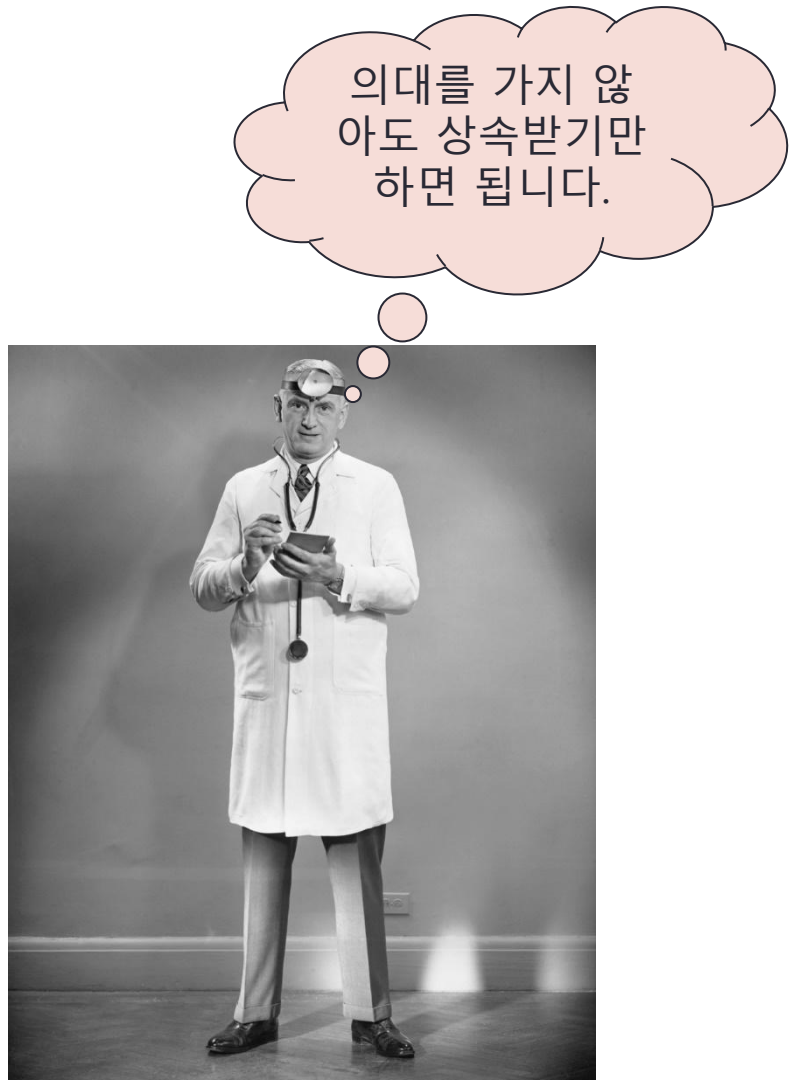
p.213

상속의 예

```
public class Doctor {
  boolean worksAtHospital;
  void treatPatient() {
    // 진료를 합니다.
  }
}
```

```
public class FamilyDoctor extends Doctor {
  boolean makeHouseCalls;
  void giveAdvice() {
    // 집에서 필요한 조언을 합니다.
  }
}
```

```
public class Surgeon extends Doctor {
  @Override
  void treatPatient() {
    // 외과 수술을 합니다.
  }
  void makeIncision() {
    // 살을 짭니다.
  }
}
```



p.213

실습 예제 01: Doctor.java

```
6   package cse.oop2.ch07.example01;
7
8   /**...4 lines */
    public class Doctor {
13
14      boolean worksAtHospital;
15
16      public Doctor(boolean worksAtHospital) {
17          this.worksAtHospital = worksAtHospital;
18      }
19
          public void treatPatient() {
21          System.out.println("진료를 합니다.");
22      }
```

```
24      @Override
        public String toString() {
            String result = null;
27
28          if (worksAtHospital) {
29              result = getClass().getName() +
30                      ": 병원에서 일하는 의사입니다.";
31          } else {
32              result = getClass().getSimpleName() +
33                      ": 집에서 일하는 의사입니다.";
34          }
35          return result;
36      }
37  }
```

실습 예제 01: FamilyDoctor.java

```
6   package cse.oop2.ch07.example01;
7
8   /**...4 lines */
12  public class FamilyDoctor extends Doctor {
13
14      boolean makeHouseCalls;
15
16      public FamilyDoctor(boolean makeHouseCalls, boolean worksAtHospital) {
17          super(worksAtHospital);      → Doctor(worksAtHospital)
18          this.makeHouseCalls = makeHouseCalls;
19      }
20
21      public void giveAdvice() {
22          System.out.println("집에서 필요한 조언을 합니다.");
23      }
24  }
```

실습 예제 01: Surgeon.java

```
6   package cse.oop2.ch07.example01;
7
8   /**...4 lines */
12  public class Surgeon extends Doctor {
13
14      public Surgeon(boolean worksAtHospital) {
15          super(worksAtHospital);
16      }
17
18      @Override
        public void treatPatient() {
20          System.out.println("외과 수술을 합니다.");
21      }
22
23      public void makeIncision() {
24          System.out.println("살을 짭니다.");
25      }
26  }
```

→ Doctor(worksAtHospital)

실습 예제 01: TestDrive.java

```
6   package cse.oop2.ch07.example01;
7
8   /**...4 lines */
12  public class TestDrive {
13
14      /**...3 lines */
17      public static void main(String[] args) {
18          // TODO code application logic here
19          Doctor doc = new Doctor(true);
20          System.out.println(doc);  // == doc.toString()
21          doc.treatPatient();
22          System.out.println();
23
24          FamilyDoctor familyDoctor = new FamilyDoctor(true, false);
25          System.out.println(familyDoctor);  // == doc.toString()
26          familyDoctor.treatPatient();
27          familyDoctor.giveAdvice();
28          System.out.println();
29
30          Surgeon surgeon = new Surgeon(true);
31          System.out.println(surgeon);  // == doc.toString()
32          surgeon.treatPatient();
33          surgeon.makeIncision();
34      }
35  }
```

실습 예제 01: 실행 결과

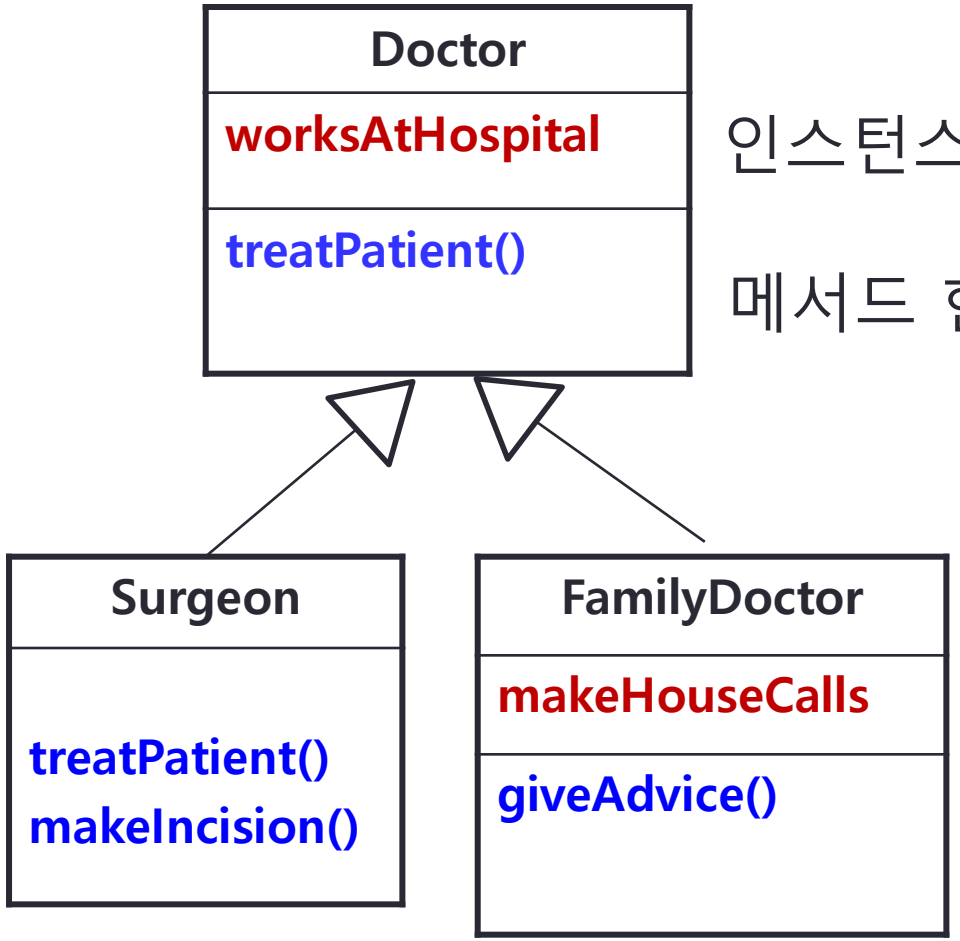
getClass().getName()

getClass().getSimpleName()

getClass().getName()

```
-----------------------< cse.ood:java_maven >-----------------------
Building java_maven 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:1.5.0:exec (default-cli) @ java_maven ---
cse.oop2.ch07.example01.Doctor: 병원에서 일하는 의사입니다.
진료를 합니다.

FamilyDoctor: 집에서 일하는 의사입니다.
진료를 합니다.
집에서 필요한 조언을 합니다.

cse.oop2.ch07.example01.Surgeon: 병원에서 일하는 의사입니다.
외과 수술을 합니다.
살을 짭니다.
```

연필을 깎으며



인스턴스 변수 한 개

메서드 한 개

Doctor에 있는 메서드 개수는?

Surgeon에 들어있는 인스턴스 변수의 개수는?

Surgeon에 들어있는 메서드 개수는?

FamilyDoctor에 들어있는 인스턴스 변수의 개수는?

FamilyDoctor에 들어있는 메서드 개수는?

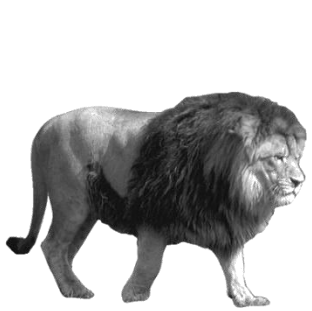
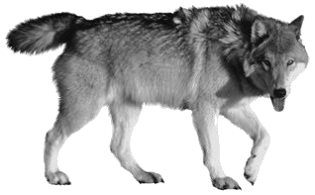
FamilyDoctor에서 treatPatient()를 실행할 수 있을까요?

FamilyDoctor에서 makeIncision()을 실행할 수 있을까요?

p.213

동물 시뮬레이션을 위한 상속 트리 설계

1. 공통적인 속성과 행동이 들어있는 객체를 찾아봅시다.

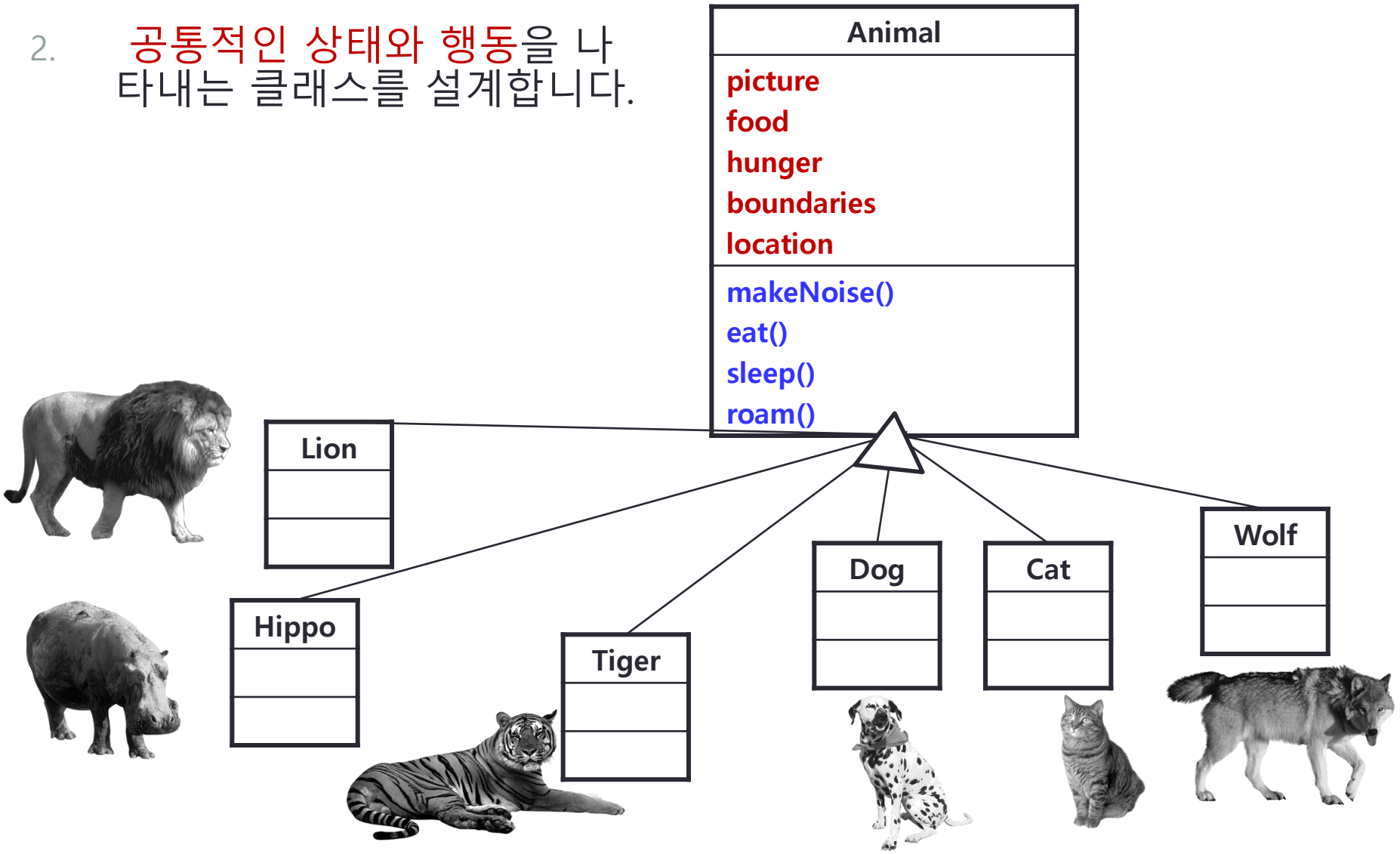





p.214

상속을 활용한 설계

2. 공통적인 상태와 행동을 나타내는 클래스를 설계합니다.

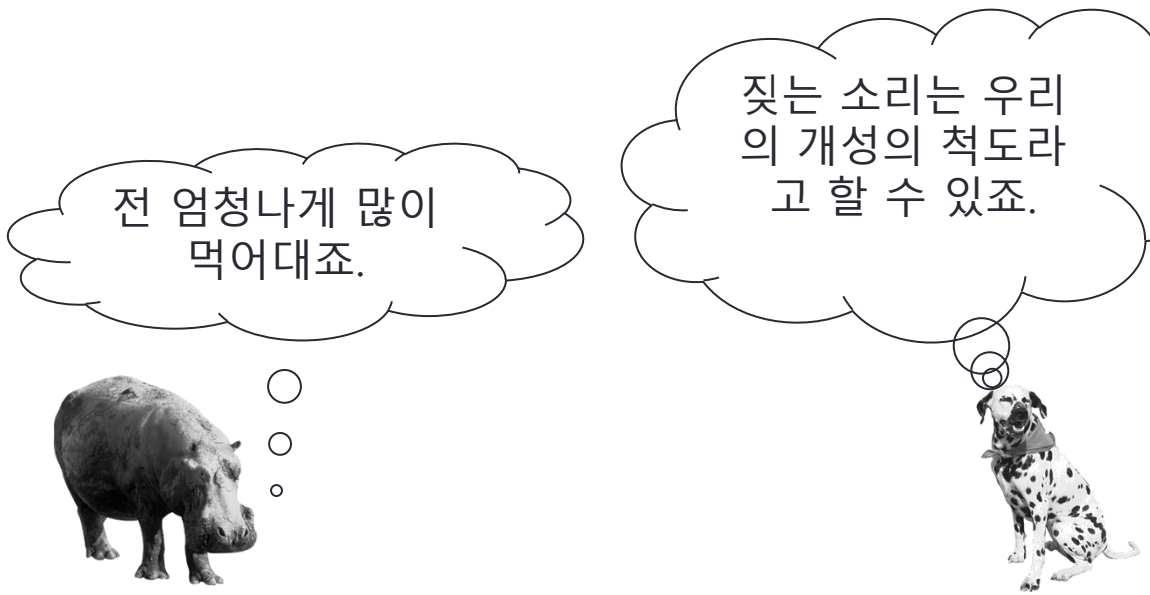


p.215

메서드 오버라이딩

3. 특정 하위클래스 유형에서만 적용되는 행동이 필요한지 결정합니다.

Animal 클래스에서는 각 하위클래스의 eat()과 makeNoise()를 오버라이드해야 한다는 결론을 내릴 수 있습니다.



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

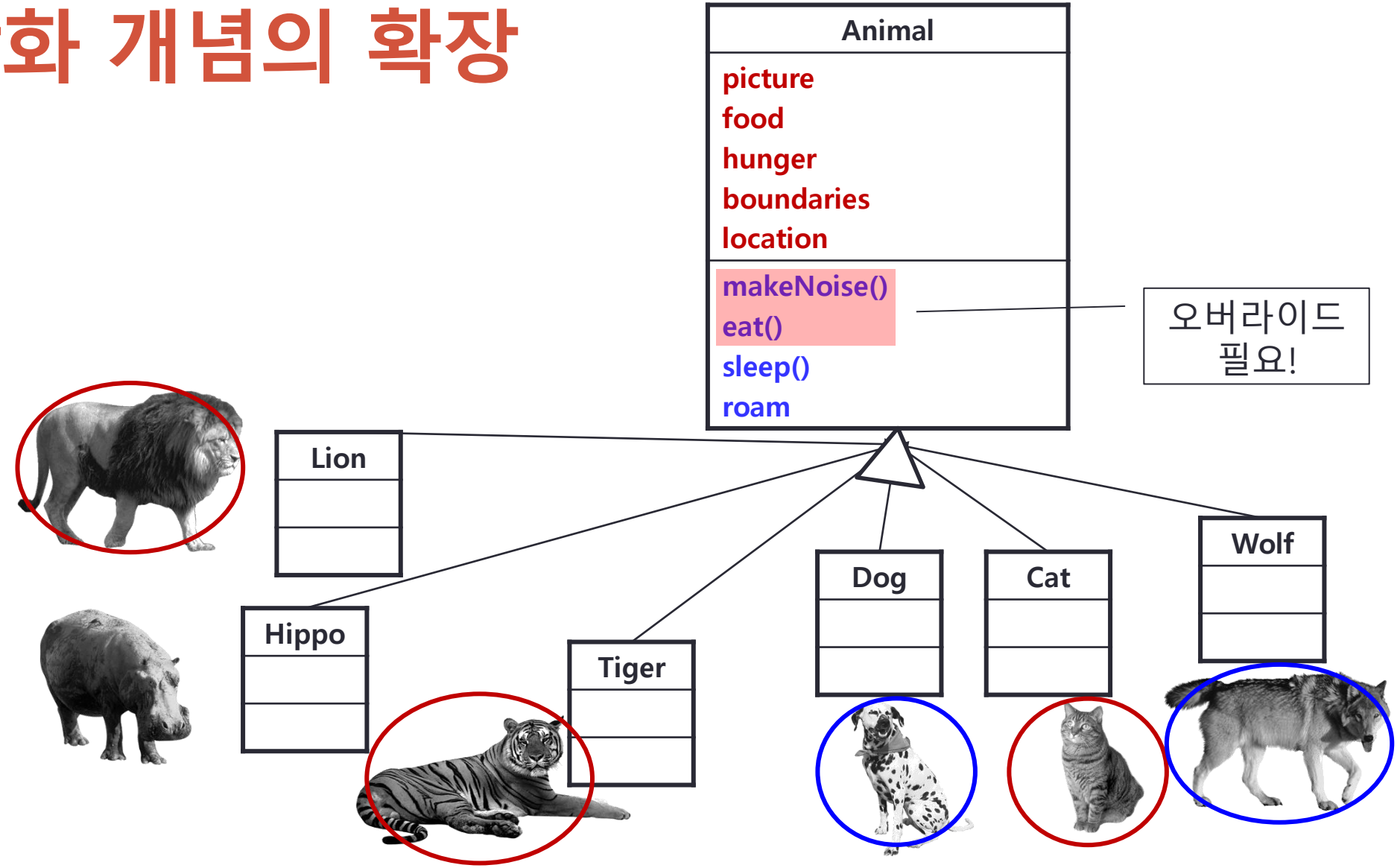
p.216

추상화 개념의 확장

4. 공통적인 행동이 필요한 하위클래스를 두 개 이상 찾아서 추상화의 개념을 더 폭넓게 활용할 수 있을지 찾아봅니다.

여러 클래스를 살펴보면 Wolf와 Dog에 공통적인 행동이 있고 Lion, Tiger, Cat에도 공통적인 행동이 있다는 것을 알 수 있습니다.

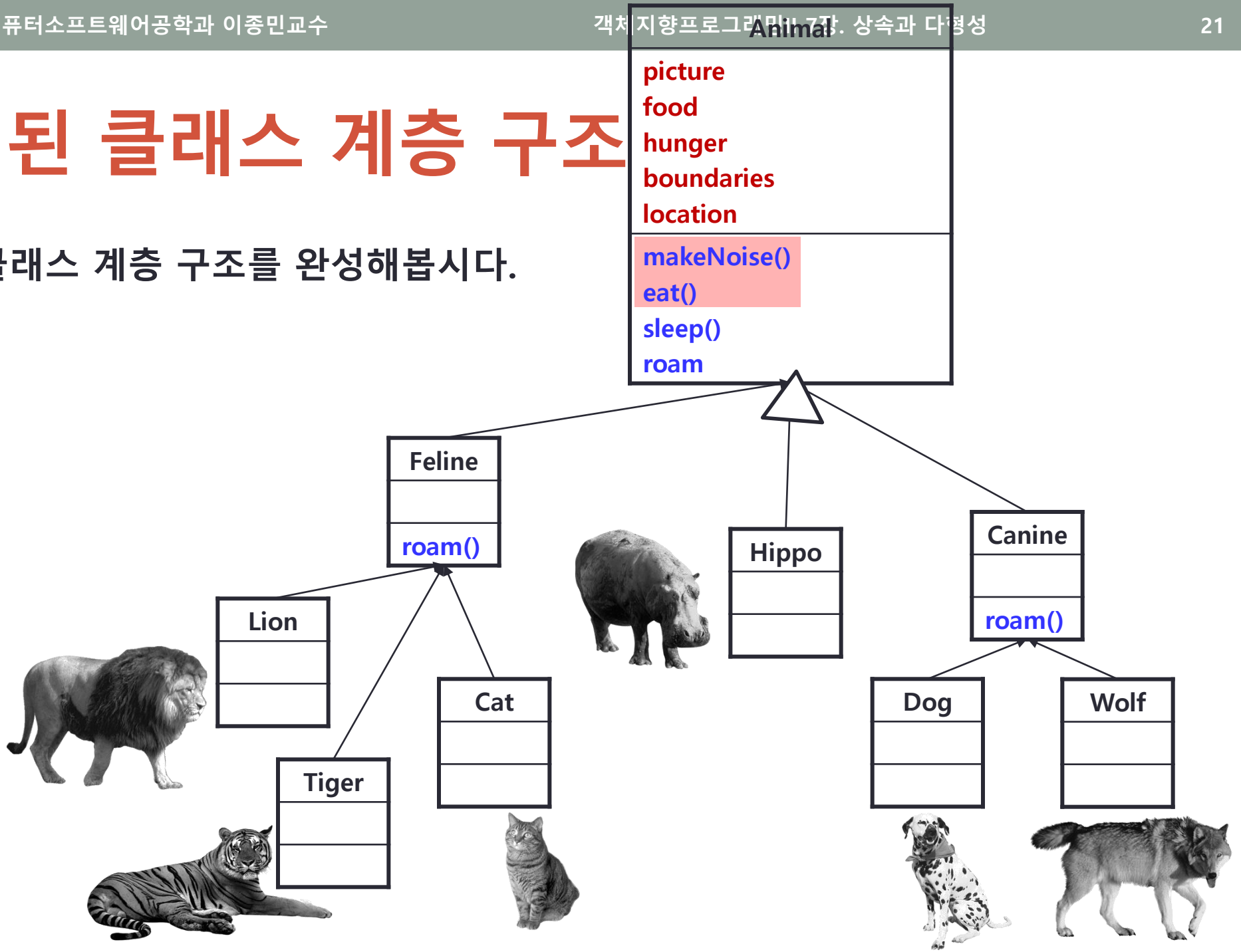
추상화 개념의 확장



p.217

완성된 클래스 계층 구조

5. 클래스 계층 구조를 완성해봅시다.



p.218

어떤 메서드가 호출될까요?

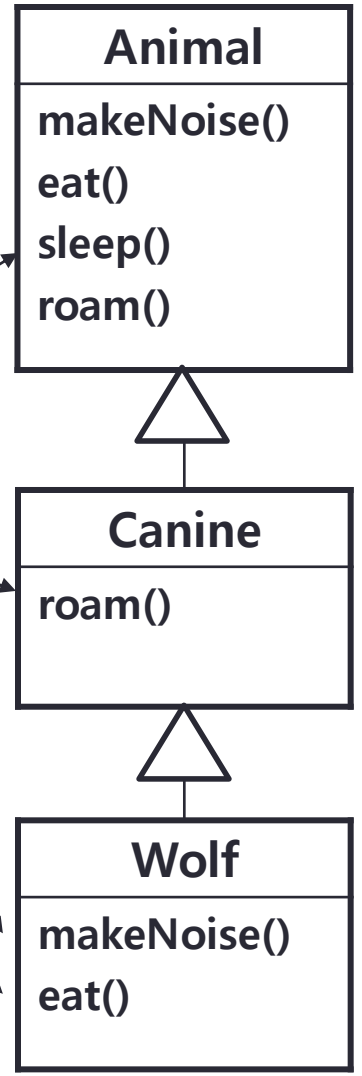
새로운 Wolf 객체를 만듭니다. Wolf w = new Wolf();

Wolf에 있는 버전을 호출합니다. w.makeNoise();

Canine에 있는 버전을 호출합니다. w.roam();

Wolf에 있는 버전을 호출합니다. w.eat();

Animal에 있는 버전을 호출합니다. w.sleep();



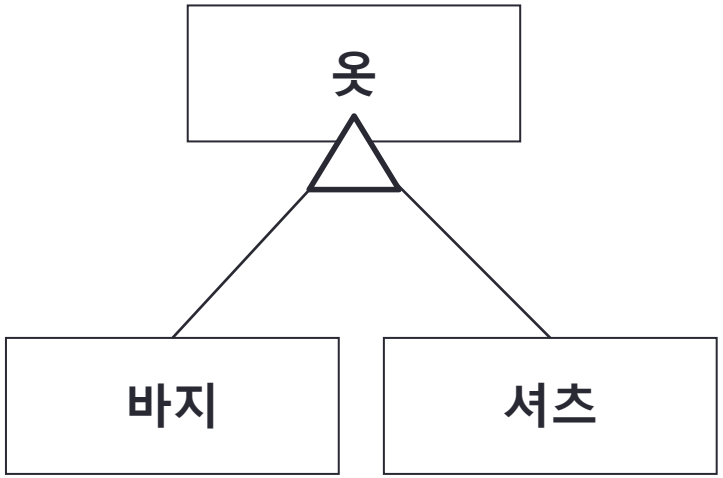
객체 레퍼런스에 대해 메서드를 호출하면 그 객체 유형의 메서드 중에서 가장 구체적인 버전이 호출됩니다.

p.219

상속 트리 설계

클래스	*상위클래스*	*하위클래스*
옷		바지, 셔츠
바지	옷	
셔츠	옷	

상속 테이블



상속 클래스 다이어그램

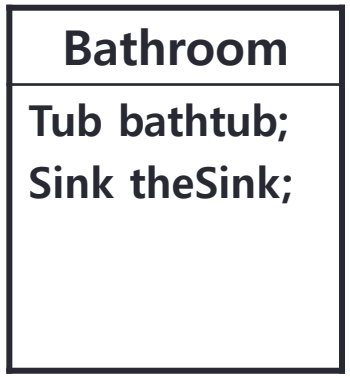
p.220

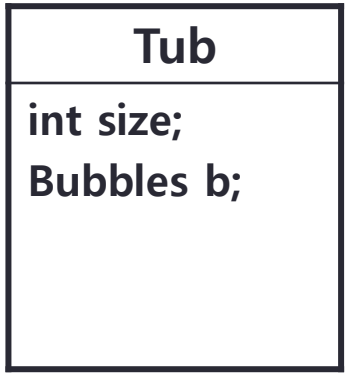
- 메서드를 호출한 클래스 유형에서 시작해서 상속 트리를 따라 올라간다고 했는데 JVM에서 매치되는 것을 전혀 찾을 수 없으면 어떻게 되나요?
  - 그런 문제는 일어나지 않습니다. 컴파일러에서 원천적으로 차단하기 때문이죠. 어떤 클래스에서 메서드를 상속한다면 그 메서드가 반드시 있는지 확인하고 넘어갑니다. JVM에서는 항상 올바른 메서드를 호출합니다.

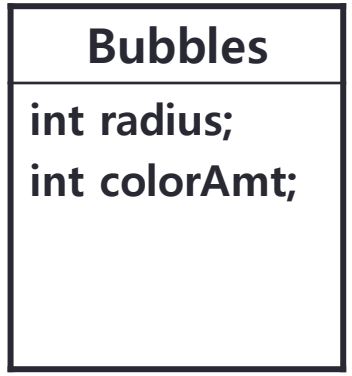
‘A는 B다’ 테스트

- 상속: “is-a” relationship
- 삼각형(Triangle)은 도형(Shape)이다.
- 고양이(Cat)는 고양이과(Feline)이다.
- 외과의사는 의사이다.

- 욕조는 화장실이다…?



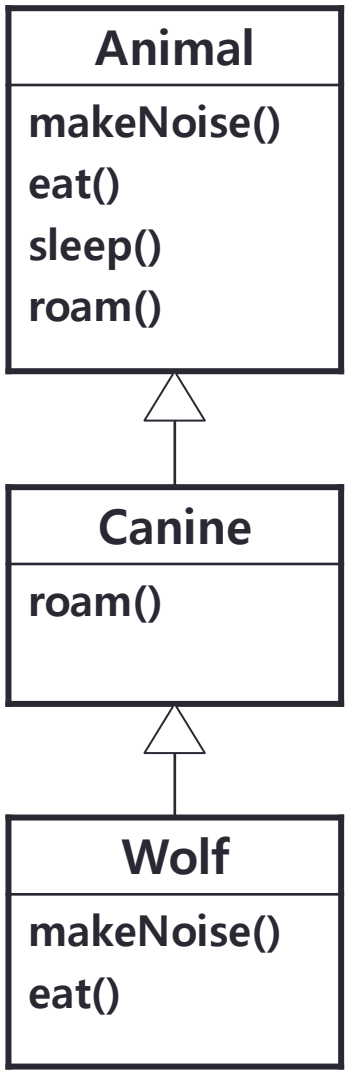




p.221

‘A는 B다’ 테스트

- 상속 트리를 제대로 설계했다면 어떤 하위클래스에 대해서도 ‘하위 클래스는 상위 클래스이다’라는 관계가 성립합니다.
- B라는 클래스가 A라는 클래스를 확장하면 B 클래스는 A 클래스입니다.
- C라는 클래스가 B라는 클래스를 확장하면 C 클래스는 B 클래스이며, 또한 A 클래스이기도 합니다.



p.222

- 상위클래스에서 하위클래스의 메서드를 쓰고 싶을 때는 어떻게 해야 하나요?
  - 상위클래스에서는 하위클래스에 대해 알 필요가 없습니다. 부모가 자식으로부터 상속을 받는 일이 없는 것처럼 상위클래스에서 하위클래스의 메서드를 만들 수는 없습니다.

- 하위클래스에서 상위클래스에 있는 버전의 메서드와 새로 오버라이드한 버전의 메서드를 모두 사용하고 싶다면 어떻게 해야 할까요?
  - super라는 키워드를 사용하면 됩니다.

```
public void roam() {
    super.roam();
    // 새로 추가할 내용
}
```

코드 예제: private 인스턴스 변수 사용

- package: cse.oop2.ch07

```
12  public class TestDrive {
13
14      /**...3 lines */
17      public static void main(String[] args) {
18          // TODO code application logic here
19          Derived d1 = new Derived();
20          System.out.println("1: number = " + d1.getNumber());
21
22          Derived d2 = new Derived();
23          d2.setNumber(10);
24          System.out.println("2: number = " + d2.getNumber());
25      }
26  }
```

```
    public class Base {
13      private int number;
14
15      public Base() {
16          this(1);
17      }
18      public Base(int number) {
19          this.number = number;
20      }
21
22      public int getNumber() {
23          return number;
24      }
25
26      public void setNumber(int number) {
27          this.number = number;
28      }
29  }
```

```
12  public class Derived  extends Base {
13  }
```

```
run:
1: number = 1
2: number = 10
```

상속 트리 설계시 주의할 점

- 어떤 클래스가 다른 클래스(상위클래스)를 더 구체화한 유형이라면 상속을 활용합니다.
  - 버드나무 – 나무
  - 자동차 – 승용차

- 같은 일반적인 유형에 속하는 여러 클래스에서 공유해야 하는 어떤 행동이 있다면 상속을 활용합니다.
  - Square, Circle, Triangle – Shape

p.225

핵심정리

- 하위클래스는 상위클래스를 확장합니다.
- 하위클래스는 상위클래스에 있는 모든 public으로 지정한 인스턴스 변수와 메서드를 상속합니다. **private으로 지정한 변수와 메서드는 상속하지 않습니다**.
- 메서드는 오버라이드할 수 있지만 **인스턴스 변수는 오버라이드할 수 없습니다**.
- **'A는 B다' 테스트**를 활용하여 상속 계층이 올바른지 확인합시다.
- 'A는 B다' 관계는 한 방향으로만 작동합니다.
- 하위클래스에서 메서드를 오버라이드하면, 그리고 하위클래스의 인스턴스에 대해 그 메서드를 호출하면 오버라이드된 버전의 메서드가 호출됩니다.
- B라는 클래스가 A라는 클래스를 확장하고 C는 B를 확장한다면 B는 A이고 C는 B이면서 또한 A가 됩니다.

- **상위 클래스는 하위 클래스를 대표하는 자료형: class B extends A**
  → B b = new B();(O) A b = new B();(O) A a = new A();(O) B a = new A();(X)

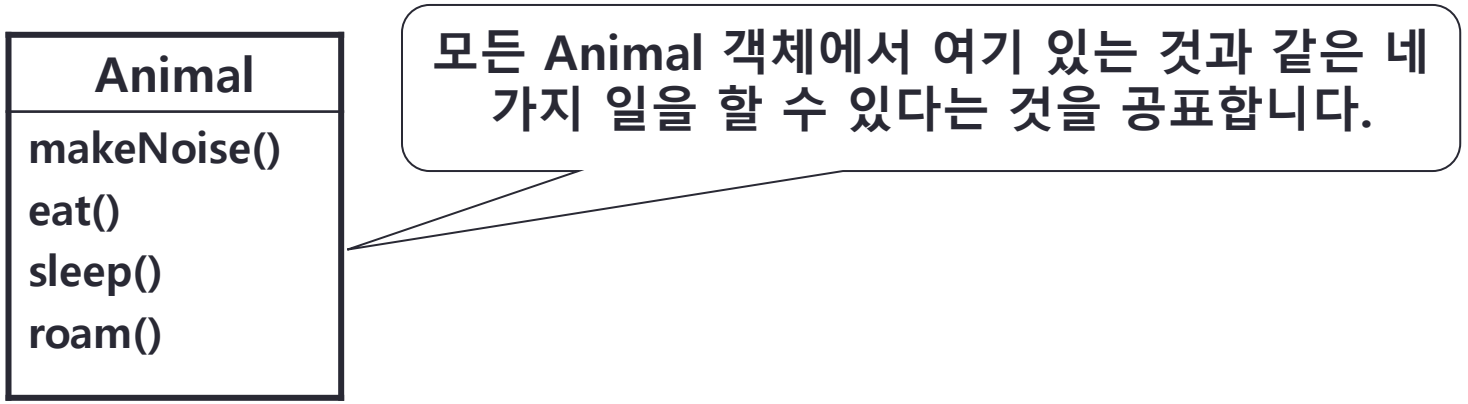
p.225

상속을 활용할 때의 장점

- **코드가 중복되는 것을 방지**할 수 있습니다.
  - 공통적인 코드를 한 군데 모아놓고 하위클래스에서 상위클래스로부터 상속을 받을 때 그 코드도 받게 합니다.
  - 행동을 변경하고 싶으면 한 군데만 변경하면 나머지 모든 하위클래스에서 변경된 기능을 활용할 수 있습니다.
- 일련의 클래스를 위한 **공통적인 규약(protocol)을 정의**합니다.

상속을 통한 규약 정의

- 상위클래스에서 메서드를 정의하면 그 메서드는 하위클래스로 상속될 수 있으며 그 메서드 정의는 일종의 규약이라고 할 수 있습니다.



p.227

다형성(polymorphism)

- 다형성
  - 서로 다른 유형의 개체에 대한 단일 인터페이스 제공 (Wikipedia)
  - (pp.228-229) 부모 클래스와 자식 클래스 사이의 다형성
    - 레퍼런스 유형을 실제 객체 유형의 상위 클래스 유형으로 지정 가능
    - 리스코프 대체 원리(Liskov Substition Princliple)
  - (p.231) 인자와 리턴 유형에 대한 다형성
  - (p.234) 오버라이딩 (overriding)
  - (p.235) 메서드 오버로딩 (method overloading)

객체 선언 및 대입 과정

Dog myDog = new Dog();

1. 레퍼런스 변수 선언

Dog myDog = new Dog();

Dog

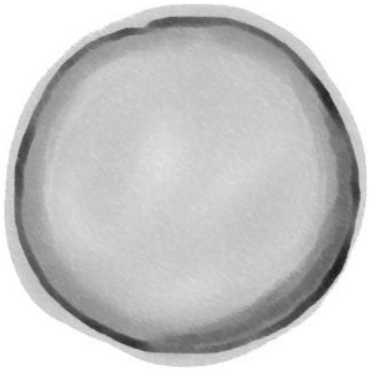
p.228

객체 선언 및 대입 과정

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

2. 객체 생성

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



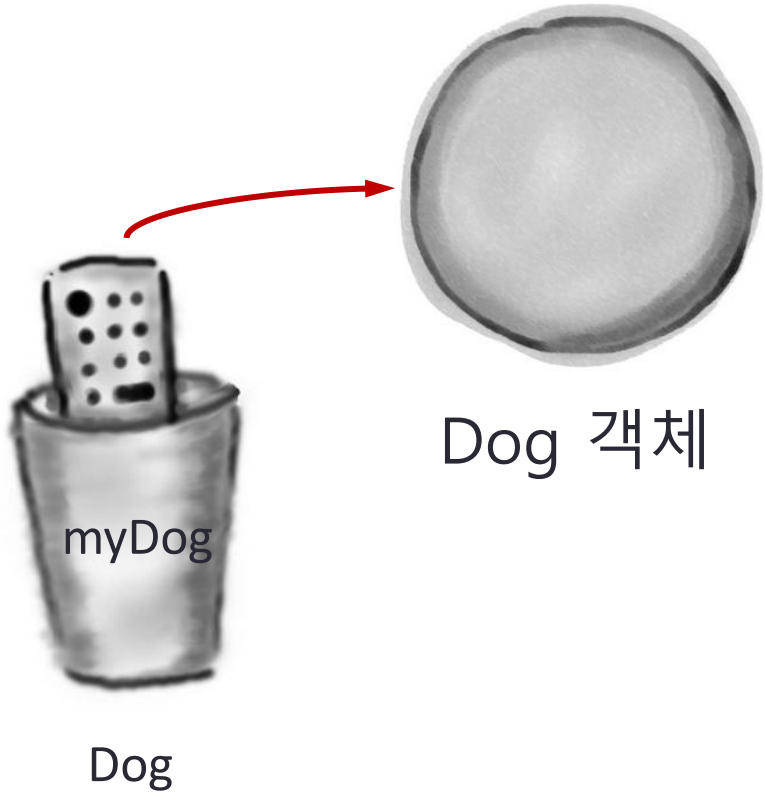
Dog 객체

객체 선언 및 대입 과정

Dog myDog = new Dog();

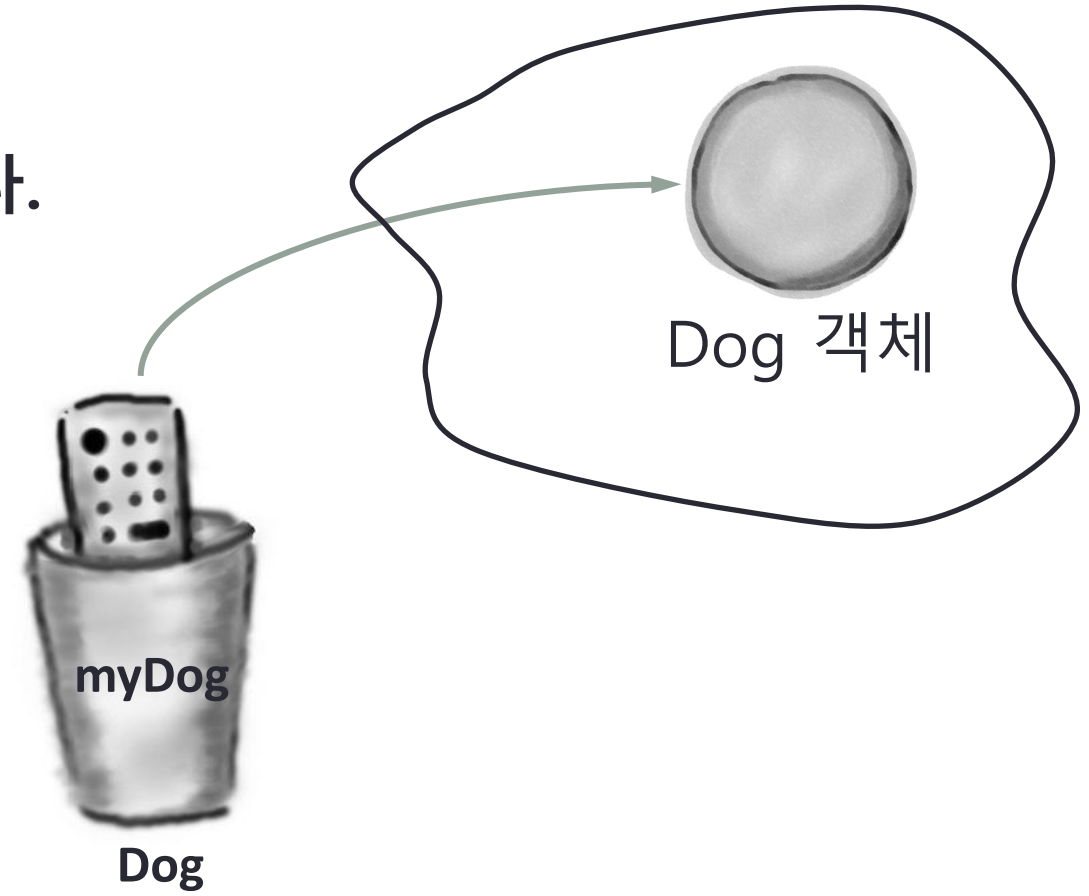
3. 객체와 레퍼런스 연결

Dog myDog = new Dog();



객체 선언 및 대입 과정

레퍼런스 유형과 객체 유형이 똑같아야 합니다.

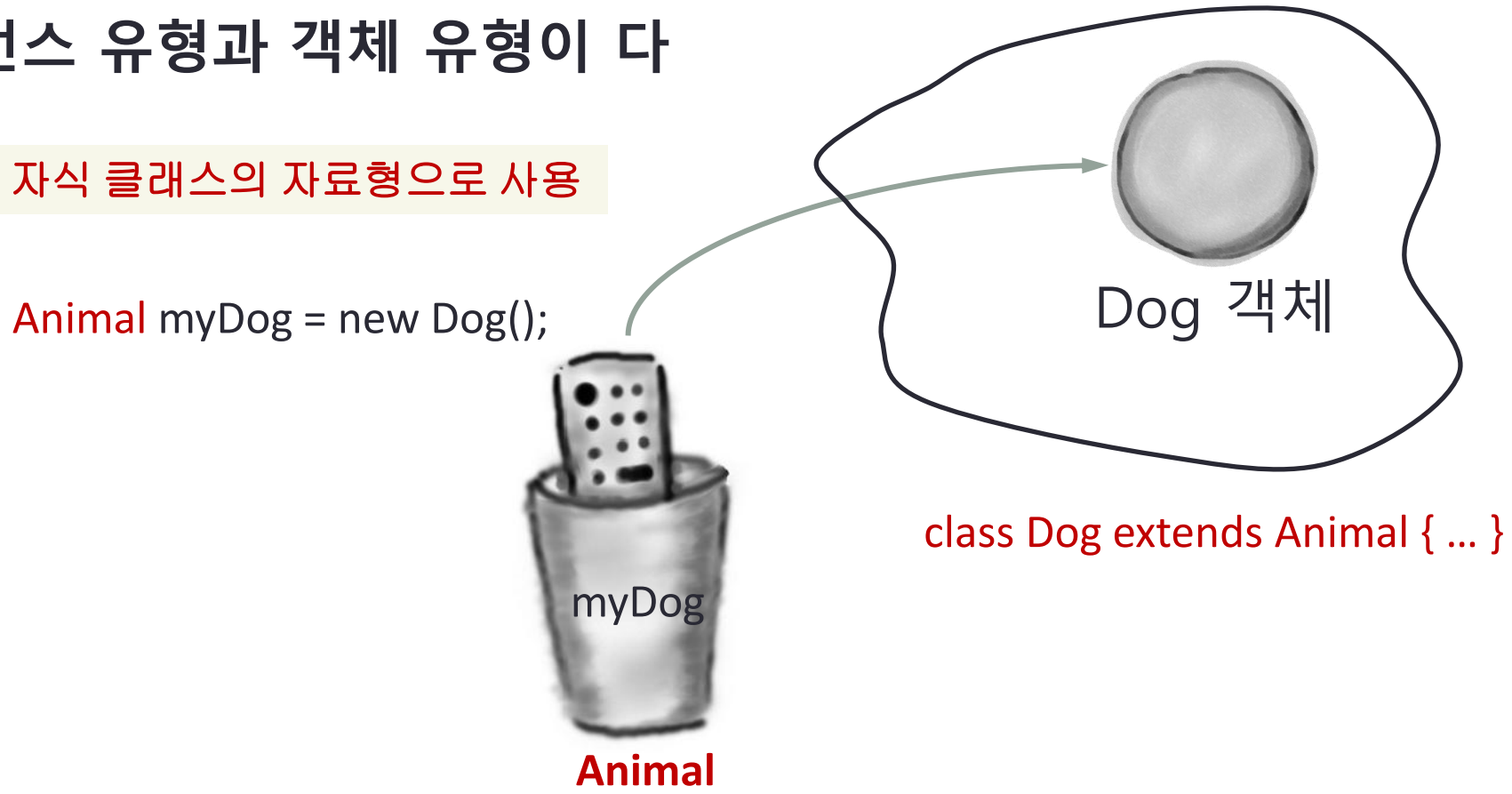


p.229

다형성 활용

다형성: 레퍼런스 유형과 객체 유형이 다른 것 허용

LSP: 부모 클래스를 자식 클래스의 자료형으로 사용



p.229

다형성 활용

- 다형성을 사용하면 레퍼런스 유형을 실제 객체 유형의 상위 클래스 유형으로 지정할 수 있습니다.

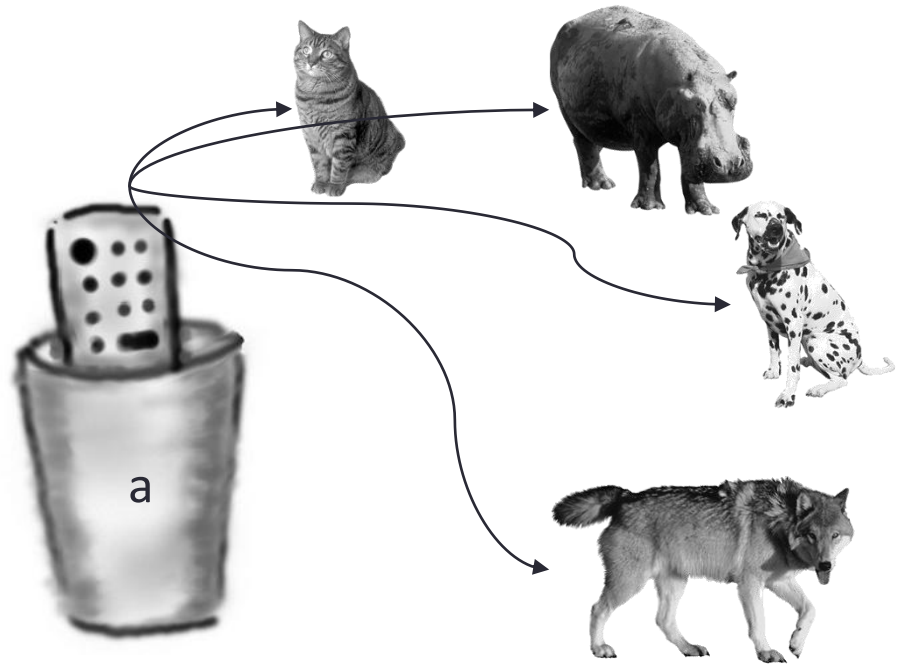
```
Animal[] animals = new Animal[5];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Wolf();
animals[3] = new Hippo();
animals[4] = new Lion();

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

p.230

다형적인 인자, 리턴 유형

- 인자와 리턴 유형에 대해서도 다형성을 적용할 수 있습니다.

```
class Vet {
  public void giveShot(Animal a) {
    // 'a' 매개변수가 가리키는 Animal 객체에
    // 주사를 놓습니다.
    a.makeNoise();
  }
}
```



p.231

다형적인 인자, 리턴 유형

- 하위클래스를 새로 만들더라도 코드를 바꿀 필요가 없습니다. (코드 재사용성 ↑)

```
class PetOwner {
  public void start() {
    Vet v = new Vet();
    Dog d = new Dog();
    Hippo h = new Hippo();
    v.giveShot(d);  // giveShot(Animal a)
    v.giveShot(h);  // Animal  상속받는 객체는 모두 대입 가능
  }
}
```

pp.231-232

바보 같은 질문은 없습니다.

- 하위클래스의 단계에 실질적인 제한이 있나요? 얼마나 깊이 들어갈 수 있어요?
  - 그런 제한이 따로 정해져 있지는 않습니다.
  - 프로그래밍을 하다 보면 상속 트리를 얕게 만드는 게 좋다는 것을 자연스럽게 깨달을 수 있을 것입니다. → (참고) 설계 패턴 (design patterns)

p.233

바보 같은 질문은 없습니다.

- 소스 코드를 직접 고칠 수 없는데 메서드 작동 방식을 바꾸고 싶을 때 하위클래스를 만들어서 그렇게 할 수 있나요?
  - 물론 가능합니다. 그리고 객체지향(OO; Object-Oriented)의 장점 가운데 하나라고 할 수 있습니다. 클래스를 완전히 새로 만들거나 누가 그 클래스를 처음 만들었는지 찾아내는 것보다는 훨씬 쉬운 방법이죠.

바보 같은 질문은 없습니다.

- 아무 클래스나 확장할 수 있나요? 아니면 클래스 멤버와 마찬가지로 클래스를 private로 지정하면 상속할 수 없다던가 하는 제한이 있나요?
  - 내부 클래스를 제외하면 private으로 지정하거나 하는 식으로 상속을 할 수 없는 클래스를 만드는 방법은 없습니다. 하지만 하위클래스를 만들 수 없도록 하는 방법은 있습니다.
  - 클래스를 public으로 선언하지 않는 방법
    → 같은 패키지 안에서만 상속 가능. 외부 패키지에서는 안 보임.
  - final 변경자를 쓰는 방법
    → final class MyClass와 같이 정의하면 더이상 수정 불가(상속 포함)를 의미
  - 클래스 생성자를 모두 private으로 지정하는 방법
    → 생성자가 안 보이므로 상속 받은 클래스를 객체로 만들 수 없음

바보 같은 질문은 없습니다.

- 굳이 하위클래스를 만들지 못하게 하는 이유가 있나요?
  - 보안상의 이유 때문에 종종 그렇게 하기도 합니다. 여러분이 직접 만든 클래스에 대해 final 변경자를 쓰는 일은 없겠지만 String 같은 클래스의 경우에는 보안상의 문제 때문에 final로 지정되어있습니다.

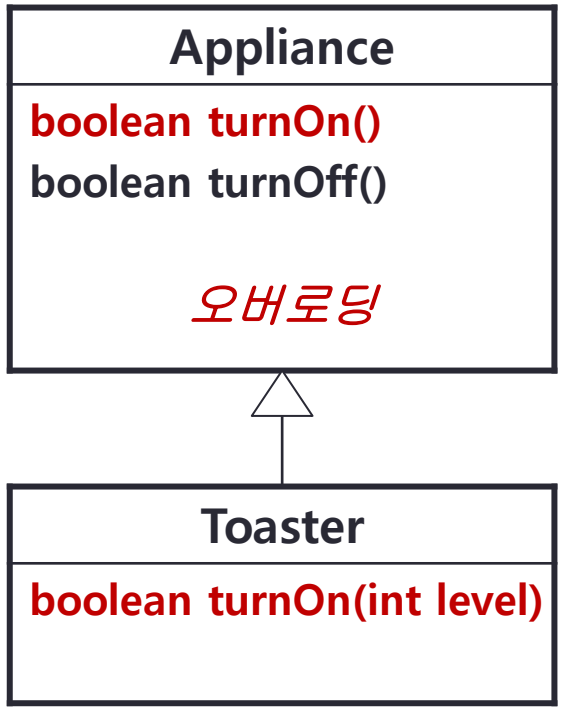
```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

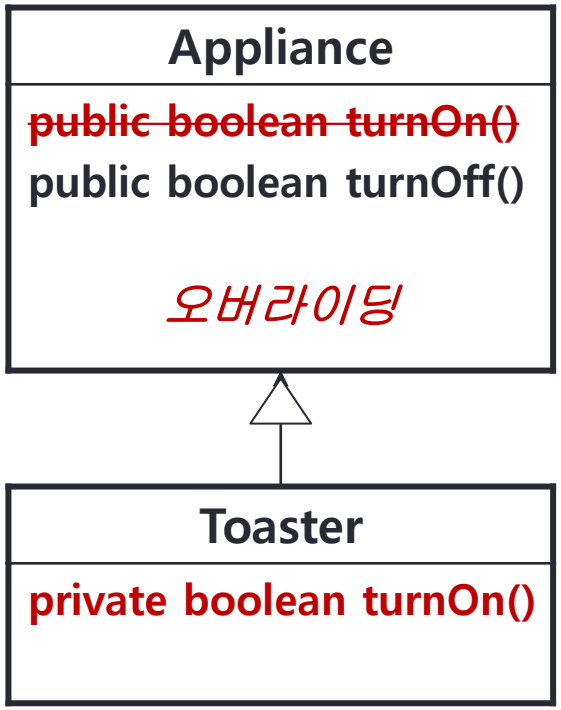
바보 같은 질문은 없습니다.

- 하위클래스는 만들 수 있도록 하면서 메서드만 오버라이드할 수 없도록 하는 방법도 있나요?
  - 메서드에만 final 변경자를 사용하면 됩니다. 클래스에 있는 모든 메서드를 오버라이드할 수 없도록 하고 싶다면 클래스 자체를 final로 지정해도 됩니다.

오버라이드(override) 규칙

- 오버라이드하는 메서드의 인자와 리턴 유형은 외부에서 보기에 상위클래스에 있는 메서드와 완벽하게 일치해야 합니다.
  - Overloading: 이름이 동일하지만 조금씩 다른 여러 개의 메서드를 정의
  - Overriding: 같은 이름의 메서드를 하나만 새로 정의





p.234

메서드 오버로딩

- 오버로딩 (Overloading)
  - 이름이 같고 인자 목록은 다른 메서드를 두 개 이상 만드는 것
  - 같은 메서드를 여러 다른 버전으로 만들 수 있습니다.

1. 리턴 유형이 달라도 됩니다.
2. 리턴 유형만 바꿀 수는 없습니다.
3. 접근 단계를 마음대로 바꿀 수 있습니다. (public, private, protected)

p.235

오버로딩의 예

```
public class Overloads {
  String uniqueID;

  public int addNums(int a, int b) {  // int  더하기
    return a + b;
  }

  public double addNums(double a, double b) {  // double  더하기
    return a + b;
  }

  public void setUniqueID(String theID) {
    // 여러 검증 과정 후에 다음을 실행
    uniqueID = theID;
  }

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

p.235

숙제

- 본문을 다시 한 번 꼼꼼히 읽어봅시다.
- 본문 및 맨 뒤에 나와있는 연습문제를 직접 풀어봅시다.

요약

- 상속(inheritance): UML에서는 일반화(generalization)이라고 함.
- 상속 찾는 방법: commonality analysis
- “is-a” relationship
- 상속이 코드 재사용 관점에서 무조건적으로 좋은 개념일까? (p.215)
- 상속의 장점
- 다형성 → overriding vs. overloading
- 객체 자료형으로 상위 클래스 유형 사용 가능: Liskov 대체 원리(LSP)
- Overriding: 메서드 재정의
- (Method) Overloading