Java Web Programming 입문 06

(Java #06)

Today's Keyword

- 생성자 (constructor)
- 변수의 초기화
  - 명시적 초기화 (Explicit Initialzation)
  - 생성자 (Constructor)
  - 초기화 블록 (Initialization Block)
- 상속 (Inheritance)
  - 상속 (Inheritance)
  - 포함 (Composite)
  - 단일상속 (Single Inheritance)
  - Object 클래스
  - 오버라이딩 (Overriding)
  - 오버로딩 (Overloading)과 오버라이딩 (Overriding)
  - super, super()

생성자 (Constructor)

- 생성자(Constructor)
  - 인스턴스 초기화 메소드
  - 클래스명과 동일한 이름
  - void를 명시하지 않는 void 메소드
  - 오버로딩(overloading)

```
Card myCard = new Card();
```

```
ClassName (DataType VarName, DataType VarName, ...){

    // Instance 생성 시 수행될 Code
    // 주로 Instance 변수의 초기와 Code를 작성

}
```

```
class card{

    // 매개변수가 없는 생성자
    Card(){
        //...
    }

    // 매개변수가 있는 생성자
    Card(String k, int num){
        //...
    }
    //...
}
```

생성자 (Constructor)

- 인스턴스 (Instance) 생성

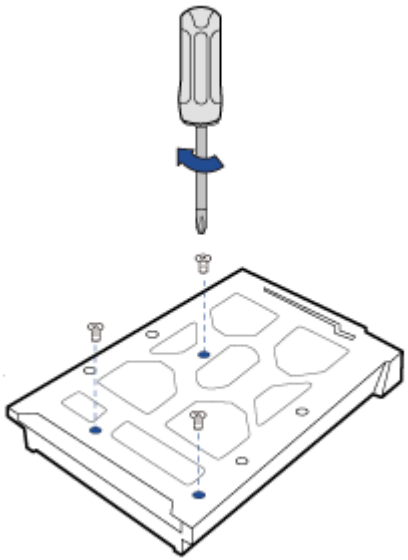
```
Card c = new Card();
```
3 1 2

  - 연산자 new에 의해 Memory에 Card Class의 Instance가 생성
  - 생성자 Card()가 호출되어 수행
  - 연산자 new의 결과로,
    - 생성된 Card의 Instance의 주소가 반환되어 참조변수 c에 저장

생성자 (Constructor)

- 기본 생성자(Default Constructor)
  - 반드시 하나의 생성자를 가지고 있어야 한다
  - 없다면 컴파일러가 생성해준다
  - 매개변수가 없다, 실행 코드도 없다

```
class Data1{
    int value;
}

class Data2{
    int value;
    Data2(int x){    // 매개변수가 있는 생성자
        value = x;
}

class ConstructorTest{
    public static void main(String[] args){
        Data1 d1 = new Data1();
        Data2 d2 = new Data2(); // compile error
    }
}
```



생성자 (Constructor)

- 오버로딩(overloading)

```
class ConstructorTest{
    public static void main(String[] args){
        Data1 d1 = new Data1();
        Data2 d2 = new Data2(); // compile error
    }
}


class Car{
    String color;          // 색상
    String gearType;       // 기어종류 auto/manual
    int door;              // 문의 개수

    Car() {} // 생성자
    Car(String c, String g, int d){ // 생성자
        color = c;
        gearType = g;
        door = d;
    }
}
```

```
Car c = new Car();
c.color = "white";
c.gearType = "auto";
c, door = 4;
```

```
Car c = new Car("white", "auto", 4);
```

생성자 (Constructor)

- this

```
class Car{
    String color;          // 색상
    String gearType;       // 기어종류 auto/manual
    int door;              // 문의 개수

    Car() {
        // Car(String color, String gearType, int door) 호출
        this("white", "auto", 4);
    }
    Car(String color){
        this(color, "auto", 4);
    }
    Car(String color, String gearType, int door){
        this.color = color;
        this.gearType = gearType;
        this.door = door;
    }
}

class CarTest2{
    public static void main(String[] args){
        Car c1 = new Car();
        Car c2 = new Car("blue");

        System.out.println("c1의 color=" + ", gearType=" +
                           c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + ", gearType=" +
                           c2.gearType + ", door=" + c2.door);
    }
}
```

- 생성자의 이름으로 클래스이름 대신 this를 사용한다.
- 한 생성자에서 다른 생성자를 호출할 때는 반드시 첫 줄에서만 호출이 가능하다.

생성자 (Constructor)

- this (self-calling)
  - Instance 자신을 가리키는 참조변수
  - Instance의 주소가 저장되어 있음
  - 모든 Instance Method에 지역변수
    - Hidden 상태로 존재
  - this(), this(매개변수)
    - 생성자
    - 동일 Class내 다른 생성자 호출 시 사용

```
Car c = new Car("white", "auto", 4);
```

```
Car c = new Car();
c.color = "white";
c.gearType = "auto";
c, door = 4;
```

```
Car(String c, String g, int d){
    color       = c ;
    gearType    = g ;
    door        = d ;
}
```

```
Car(String color, String gearType, int door){
    this.color      = color     ;
    this.gearType   = gearType  ;
    this.door       = door      ;
}
```

변수의 초기화

- 변수 종류에 따른 초기화

```
class InitTest {
    int x;      // 인스턴스 변수
    int y = x;  // 인스턴스 변수

    void method1(){
        int i;      // 지역변수
        int j = i;  // ??
    }
}
```

자료형	기본값
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형 변수	null

  - 멤버변수 (클래스/인스턴스) 와 배열의 초기화
    - 선택적
  - 지역변수
    - 반드시 사용 전 초기화가 선행되어야 한다

변수의 초기화

- 변수 값 대입 (revisit)

선언예	설 명
int i=10; int j=10;	int형 변수 i를 선언하고 10으로 초기화 한다. int형 변수 j를 선언하고 10으로 초기화 한다.
int i=10, j=10;	같은 타입의 변수는 콤마(,)를 사용해서 함께 선언하거나 초기화 할 수 있다.
int i=10, long j=0;	에러!!! 타입이 다른 변수는 함께 선언하거나 초기화할 수 없다.
int i=10; int j=i;	변수 i에 저장된 값으로 변수 j를 초기화 한다. 변수 j는 i의 값인 10으로 초기화 된다.
int j=i; int i=10;	에러!!! 변수 i가 선언되기 전에 i를 사용할 수 없다.

변수의 초기화

- 멤버변수의 초기화 방법
  - 명시적 초기화 (Explicit initialization)
  - 생성자 (Constructor) 를 이용
  - 초기화 블록 (Initialization Block)
    - Instance 초기화 블록 : Instance 변수 초기화에 사용
    - Class 초기화 블록 : Class 변수 초기화에 사용

변수의 초기화

- 명시적 초기화 (Explicit Initialization)

```
class Car{
    // 기본형(Primitive Type) 변수 초기화
    int door = 4;
    // 참조형(Reference Type) 변수 초기화
    Engine e = new Engine();

    //...
}
```

- 생성자 (Constructor)

변수의 초기화

- 초기화 블록(Initialization Block)
  - 클래스 초기화 블록
    - 클래스 변수의 복잡한 초기화에 사용
  - 인스턴스 초기화 블록
    - 인스턴스 변수의 복잡한 초기화에 사용

```
class InitBlock{
    static { /* 클래스 초기화 블록 */}

    {/* 인스턴스 초기화 블록 */}

    // ...
}
```

변수의 초기화

- 초기화 블록(Initialization Block)

```
class BlockTest{

    static {
        System.out.println("클래스 초기화 블록 수행");
    }

    {
        System.out.println("인스턴스 초기화 블록 수행");
    }

    public BlockTest(){
        System.out.println("생성자 수행");
    }

    public static void main(String args[]){
        System.out.println("BlockTest bt = new BlockTest(); ");
        BlockTest bt = new BlockTest();

        System.out.println("BlockTest bt = new BlockTest(); ");
        Blocktest bt2 = new BlockTest();
    }
}
```

변수의 초기화

- 초기화 블록(Initialization Block)
  - 클래스 변수의 초기화 시점
    - Class가 처음 Loading될 때 단 한번 초기화
  - 인스턴스 변수의 초기화 시점
    - Instance가 생성될 때마다
    - 각 Instance별로 초기화
  - 클래스 변수의 초기화 순서
    - 기본값 -> 명시적 초기화 -> 클래스 초기화 블록
  - 인스턴스 변수의 초기화 순서
    - 기본값 -> 명시적 초기화 -> 인스턴스 초기화 블록 -> 생성자

```
class InitTest{
    // 명시적 초기화
    static int cv = 1;
    int iv = 1;

    // 클래스 초기화 블록
    static {    cv = 2; }

    // 인스턴스 초기화 블록
    {   iv = 2; }

    // 생성자
    InitTest () {
        iv = 3;
    }
}
```

클래스 초기화			인스턴스 초기화			
기본값	명시적 초기화	클래스 초기화블럭	기본값	명시적 초기화	인스턴스 초기화블럭	생성자
cv 0	cv 1	cv 2	cv 2 iv 0	cv 2 iv 1	cv 2 iv 2	cv 2 iv 3
1	2	3	4	5	6	7

변수의 초기화

- 초기화 블록(Initialization Block) 예제 코드

```
class Product{
    static int count = 0;
    int serialNo;

    {
        ++count;
        serialNo = count;
    }

    public Proquct() {  }
}

class ProductTest{
    public static void main(String args[]){
        Product p1 = new Product();
        Product p2 = new Product();
        Product p3 = new Product();

        System.out.println("p1의 제품번호(serial no)" + p1.serialNo);
        System.out.println("p2의 제품번호(serial no)" + p2.serialNo);
        System.out.println("p3의 제품번호(serial no)" + p3.serialNo);
        System.out.println("생산된 제품의 수는 모두" + Product.count + "개 입니다.");
    }
}
```

```
p1의 제품번호(serial no)는 1
p2의 제품번호(serial no)는 2
p3의 제품번호(serial no)는 3
생산된 제품의 수는 모두 3개 입니다.
```

변수의 초기화

- 초기화 블록(Initialization Block) 예제 코드

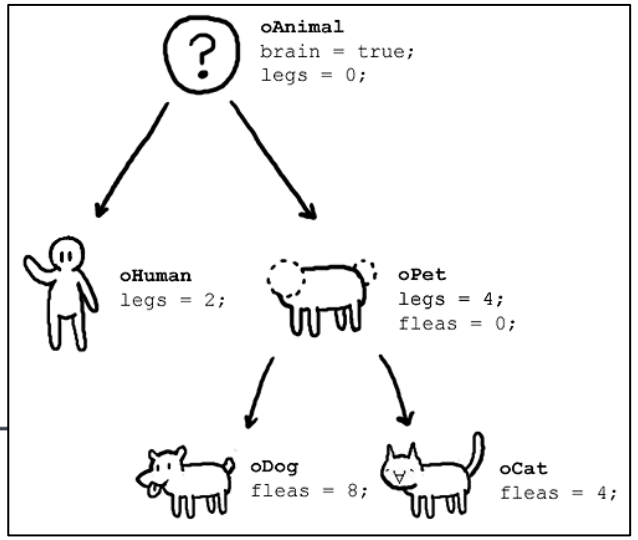
```
class Document{
    static int count = 0;
    String name;     // 문서명

    public Document(){
        this("제목없음" + ++count);
    }

    public Document(String name){
        this.name = name;
        System.out.println("문서 " + this.name + "가 생성되었습니다.");
    }
}

class DocumentTest{
    public static void main (String args[]){
        Document d1 = new Document();
        Document d2 = new Document("자바.txt");
        Document d3 = new Document();
        Document d4 = new Document();
    }
}
```

```
문서 제목없음1가 생성되었습니다.
문서 자바.txt가 생성되었습니다.
문서 제목없음2가 생성되었습니다.
문서 제목없음3가 생성되었습니다.
```

상속 (inheritance)

- 클래스의 한계
  - 변수와 메소드의 중복
  - 독립적인 요소들
  - 프로그래밍의 효율

- 상속 (Inheritance)
  - Class 재사용
  - 'is a' 관계
  - 변수와 메소드의 공유
  - 공통 코드 관리
  - 코드양의 감소
  - 객체지향 프로그래밍
    - Object Oriented Programming
  - 프로그래밍 효율
  - 코드 유지보수 용이 (추가/변경)



상속 相續 [발음 : 상속]

활용 : 상속만[상송만]

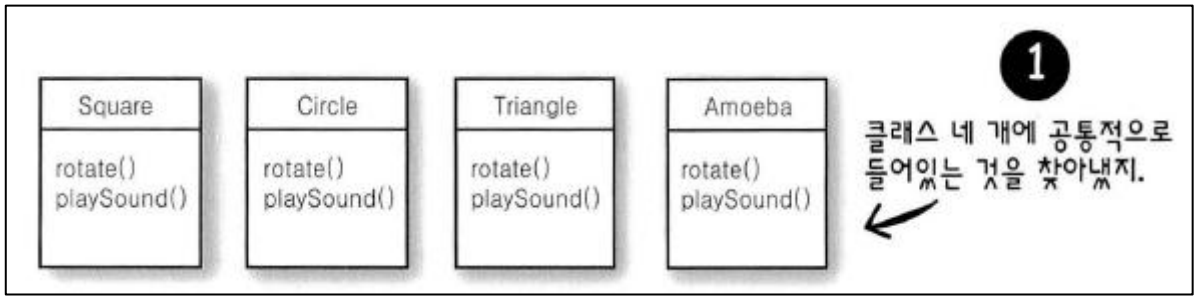
파생어 : 상속되다, 상속하다

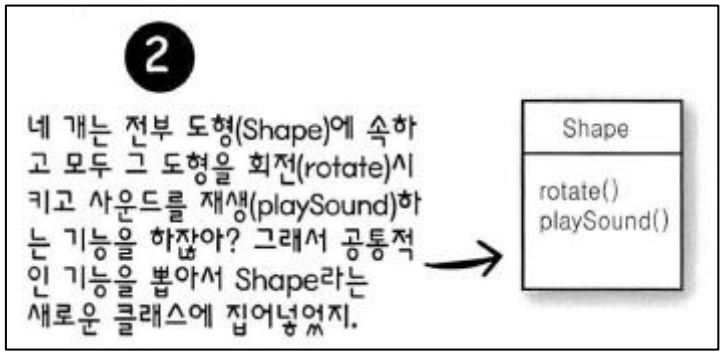
명사

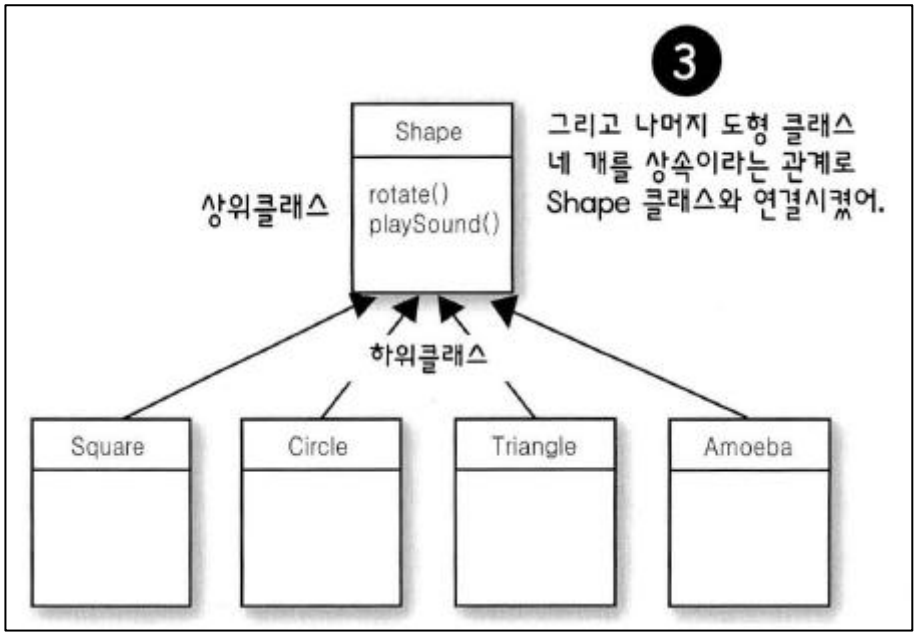
1. 다음 차례에 이어 주거나 이어받음.
2. <법률> 일정한 친족 관계가 있는 사람 사이에서, 한 사람의 사망으로 다른 사람이 재산에 관한 권리와 의무의 일체를 이어받는 일.
   - 상속을 받다
   - 그건 분명히 도인이 시골의 부모로부터 **상속** 삼아 받은 재산의 전부였다. 출처 : 김승옥, 60년대식

상속 (Inheritance)

- 상속, 객체지향 프로그래밍의 위대한 시작

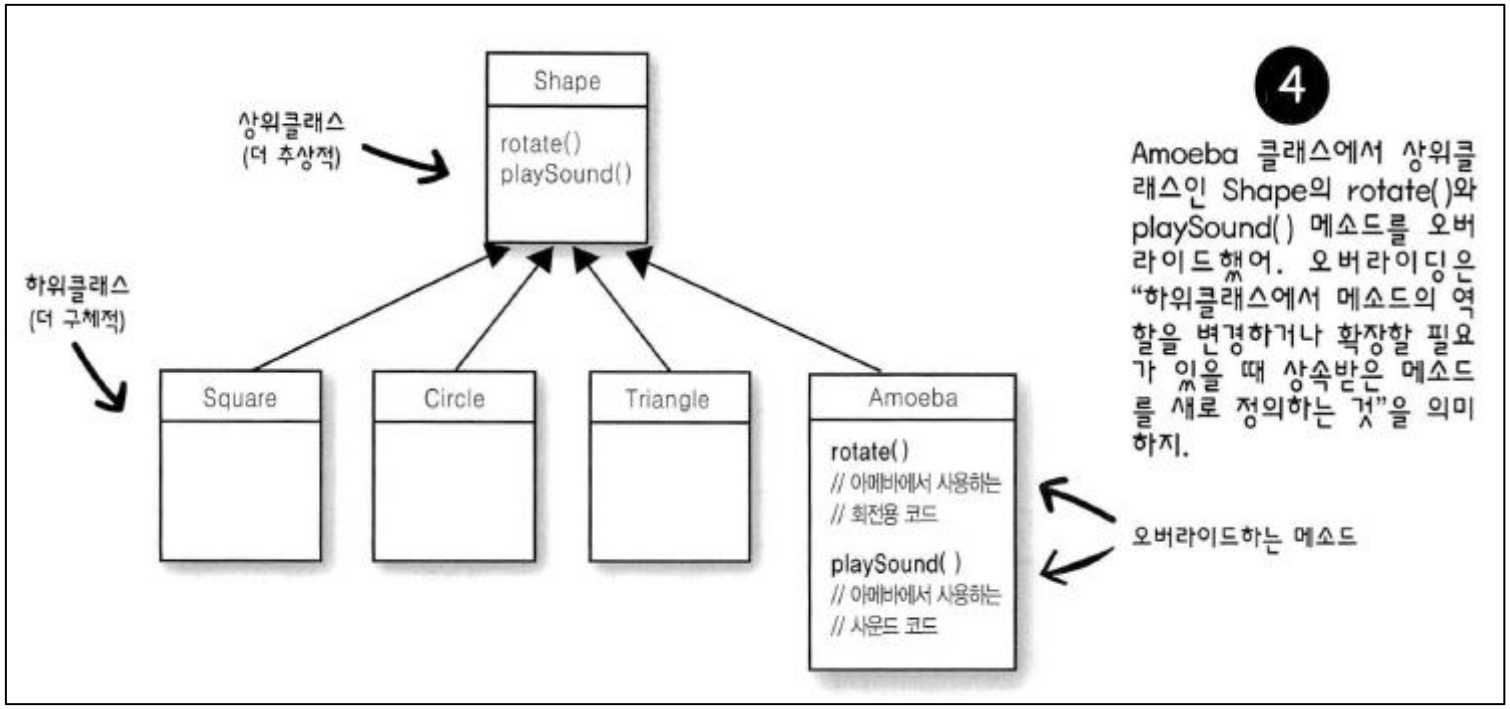






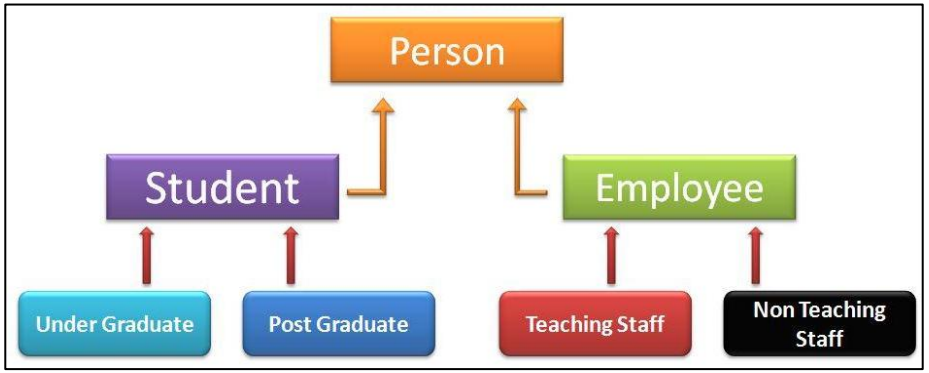
상속 (Inheritance)

- 상속, 객체지향 프로그래밍의 위대한 시작



상속 (inheritance)

- 주안점
  - 단일 상속(Single Inheritance)
  - 선언없이 멤버들을 재사용
  - 생성자, 초기화블록은 상속 X



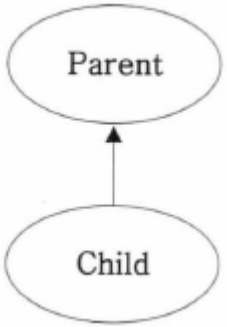
- 추상화(Abstraction), 일반화(Generalization)
  - 공통 요소를 찾음
  - 부모 클래스로 제작

- 구체화(materialization), 전문화(Specialization)
  - 더 자세하게 명시
  - 자식 클래스로 제작

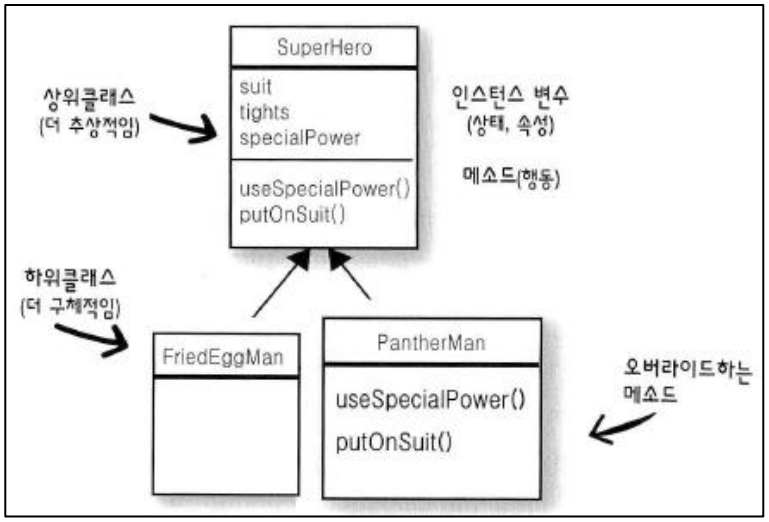
상속 (inheritance)

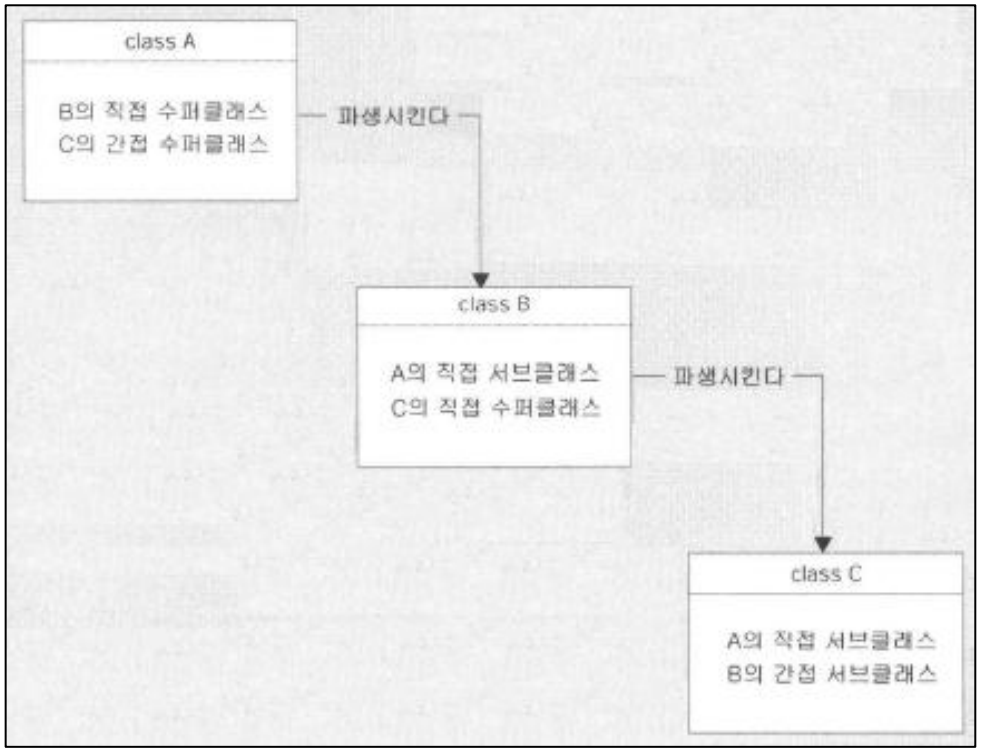
- 용어

조상 클래스 - 부모(parent)클래스, 상위(super)클래스, 기반(base)클래스
자손 클래스 - 자식(child)클래스, 하위(sub)클래스, 파생된(derived) 클래스



  - Parent – Child
  - Super – Sub





상속 (inheritance)

- example (from HeadFirst)

인스턴스 변수

picture – 그 동물의 모습을 보여주는 JPEG 파일명

food – 그 동물이 먹는 음식의 형식. 일단은 meat(고기)와 grass(풀), 이렇게 두 가지만 있다고 가정해봅시다.

hunger – 그 동물의 배고픈 정도를 나타내는 int 변수. 그 동물이 가장 최근에 언제 (그리고 얼마나) 먹었는지에 따라 달라집니다.

boundaries – 그 동물이 돌아다니는 '공간'의 높이와 너비(예를 들어, 640×480)를 나타내는 값

location – 공간 내에서 그 동물이 있는 위치를 나타내는 X와 Y 좌표

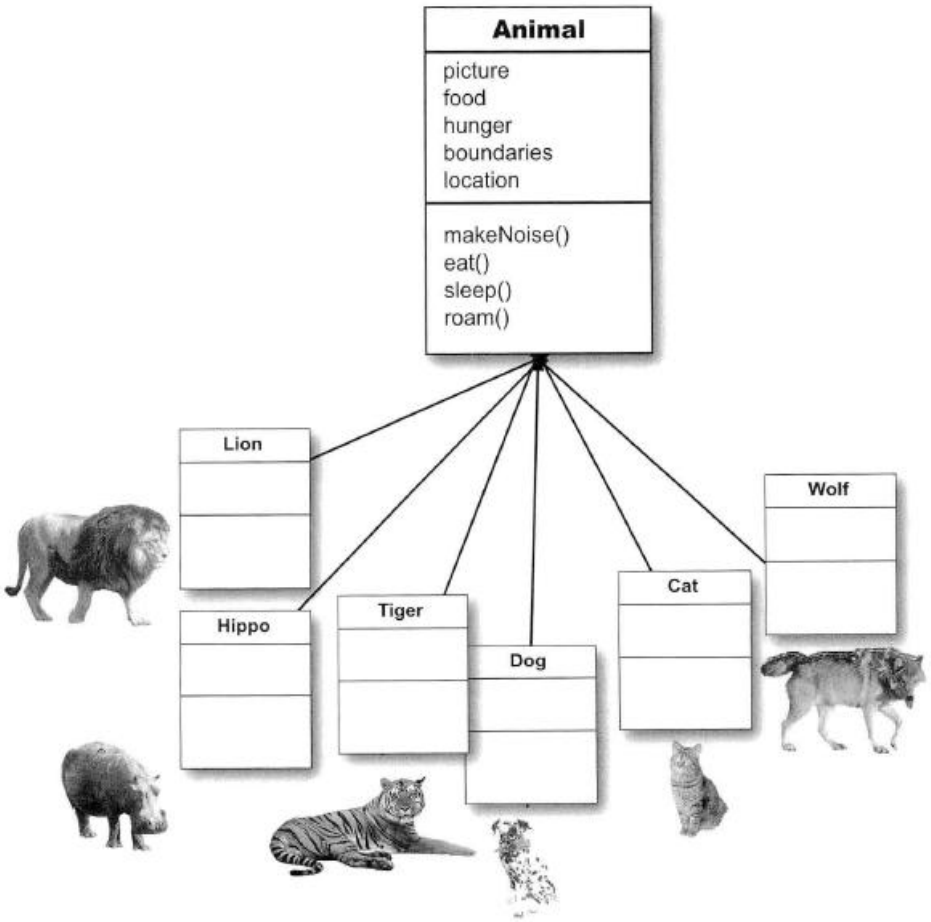
메소드

makeNoise() – 동물이 소리를 낼 때의 행동

eat() – 그 동물이 음식(meat 또는 grass)을 섭취했을 때의 행동

sleep() – 그 동물이 잠들어 있을 때의 행동

roam() – 그 동물이 먹거나 자고 있지 않을 때의 행동(그냥 먹이를 찾아 돌아다닐 때의 행동)

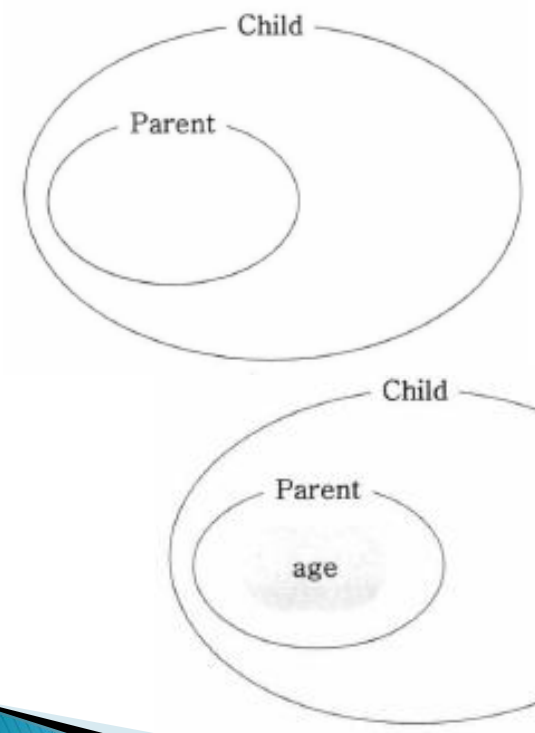


상속 (inheritance)

- How?

```
class Parent     {
    // . . .
}
```

```
class Child extends Parent {
    // . . .
}
```

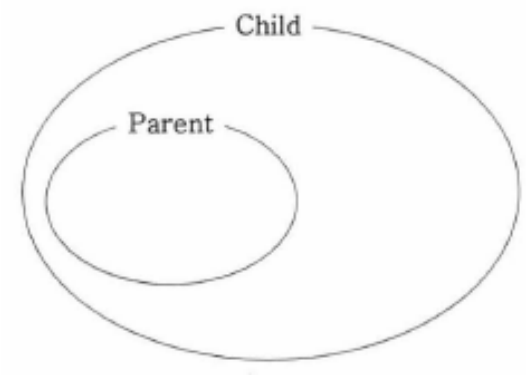



```
class Parent     {
    int age;
}

class Child extends Parent   {

}
```

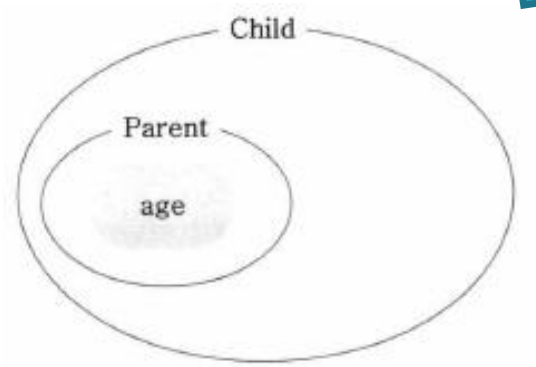
클래스 이름	클래스의 멤버
Parent	age
Child	age

상속 (Inheritance)

- 착각하면 안되는 부모(Parent) – 자식(Child) 관계




```
class Parent     {
    int age;
}

class Child extends Parent   {

}
```

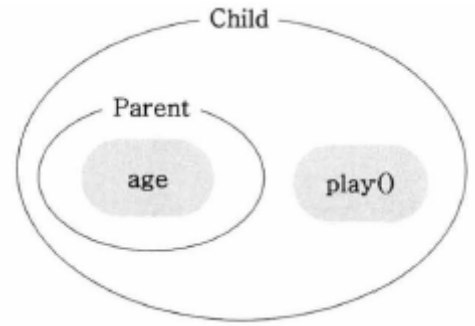


클래스 이름	클래스의 멤버
Parent	age
Child	age

상속 (Inheritance)

- 자식이(Child) 스스로 번 재산은 자기꺼

```
class Parent     {
     int age;
}

class Child extends Parent   {
     void play() {
          System.out.println("놀자~");
     }
}
```



클래스 이름	클래스의 멤버
Parent	age
Child	age, play()

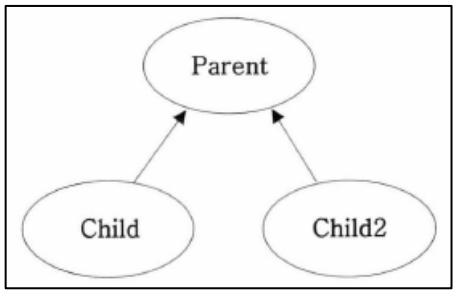
- 생성자와 초기화 블럭은 상속되지 않는다. 멤버만 상속된다.
- 자손 클래스의 멤버 개수는 조상 클래스보다 항상 같거나 많다.

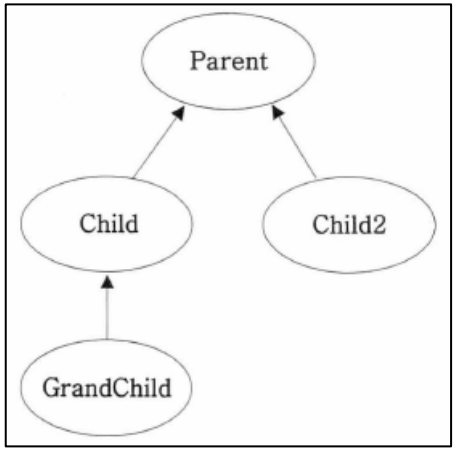
상속 (Inheritance)

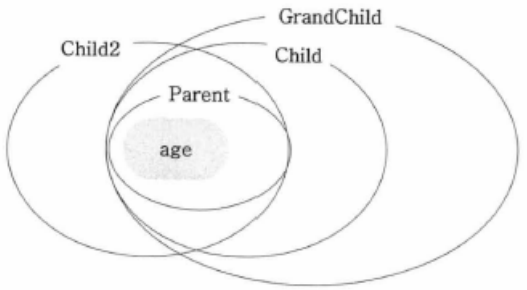
- 상속받은 부모 재산은 자기 자식에게 상속 가능

```
class Parent     {    }
class Child extends Parent   {    }
class Child2 extends Parent  {    }
```

```
class Parent     {    }
class Child extends Parent   {    }
class Child2 extends Parent  {    }
class GrandChild extends Child  {    }
```







```
class Parent     {
    int age;
}

class Child extends Parent   {    }
class Child2 extends Parent  {    }

class GrandChild extends Child   {    }
```

클래스 이름	클래스의 멤버
Parent	age
Child	age
Child2	age
GrandChild	age

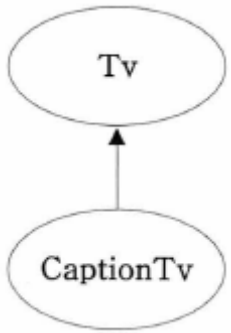
상속 (Inheritance)

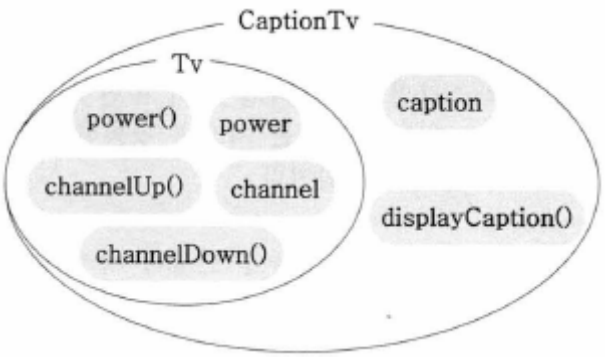
- example code

```
class Tv    {
    boolean power;  // 전원 on/off
    int channel;    // 채널

    void power()         {   power = !power; }
    void channelUp()     {   ++channel;      }
    void channelDown()   {   --channel;      }
}
```

```
class CaptionTv extends Tv  {
    boolean caption;    // 캡션상태 on/off
    void displayCaption (String text) {
        // 캡션상태 on (true) 일 때만 text 보여줌
        if (caption) {
            System.out.println(text);
        }
    }
}
```

```
class CaptionTvTest {
    public static void main(String args[]){
        CaptionTv ctv = new CaptionTv();
        ctv.channel = 10;   // 부모 클래스에게 상속 받음
        ctv.channelUp();    // 부모 클래스에게 상속 받음
        System.out.println(ctv.channel);
        ctv.displayCaption("Hello, World");
        ctv.caption = true; // 캡션 기능을 on
        ctv.displayCaption("Hello, world"); // 캡션이 화면에 출력
    }
}
```






```
11
Hello, World
```

자손 클래스의 인스턴스를 생성하면 조상 클래스의 멤버와 자손 클래스의 멤버가 합쳐진 하나의 인스턴스로 생성된다.

상속 (Inheritance)

- 클래스 관계 상속(Inheritance)이냐, 포함(Composite)이냐?

```
class Circle {
    Point c = new Point();
    int r;
}
```

```
class Circle extends Point {
    int r;
}
```

  - 원(Circle)은 점(Point)이다
    - Circle **is a** Point
  - 원(Circle)은 점(Point)을 가지고 있다
    - Circle **has a** Point

  - 상속관계
    - ~은 ~이다 (is a)
  - 포함관계
    - ~은 ~을 가지고 있다 (has a)

상속 (inheritance)

- 단일상속 (Single Inheritance)
  - 아빠나 엄마 둘 중 한 명만 선택해! 엄빠는 안돼!
  - 자바에서의 상속은 단일 상속뿐이다
  - 즉, 다중 상속은 사용할 수 없다

```
// 요런건 불가능하다!
class TVCR extends TV, VCR {
    // . . .
}
```

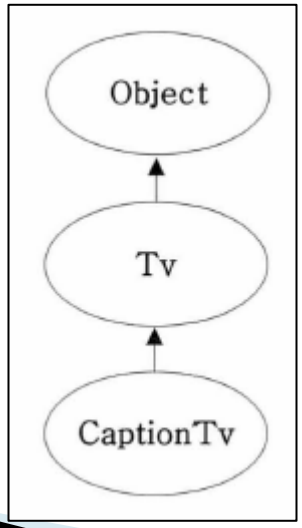
  - 굳이 복수의 클래스에 내용을 끌어쓰고 싶다면
    - 상속(Inheratance) + 포함(Composite) 조합
    - 인터페이스(Interface)

상속 (inheritance)

- Object Class
- 클래스의 선조 (최상위 조상님)

```
class Tv {
    // . . .
}
```

```
class Tv extends Object {
    // . . .
}
```




```
class Tv {
    // . . .
}

class CaptionTv extends Tv {
    // . . .
}
```

오버라이딩(Overriding)

- 면접에서도 많이 물어보고, 실제로도 많이 쓰인다
- 오버라이드의 사전적 의미

override [òuvəráid] *vt.* (*-rode* [-róud]; *-ridden* [-rídn], *-rid* [-rídl])
① (장소를) 말을 타고 지나다; 타고 넘다; ---위로 퍼지다.
② 짓밟다, 유린하다.
③ 무시하다; 거절하다.
④ (결정 따위를) 무효로 하다, 뒤엎다.
⑤ (말을) 지쳐 쓰러지도록 타다.
⑥ 〖의학〗 (부러진 뼈가 다른 뼈) 위에 겹쳐지다.
⑦ ---에 우월[우선]하다; (자동제어 따위를) 떼다, 분리하다.
⑧ 【미국】 (매상에 의하여) 커미션을 지불하다.
--------●~ one's commission 직권을 남용하다.④
♣~ a veto 【미국】 대통령이 거부한 법안을 재가결하다.

- 아버지에게 물려받은 차를 튜닝
- 어머니가 투자하시던 주식을 펀드로 돌린다

오버라이딩(Overriding)

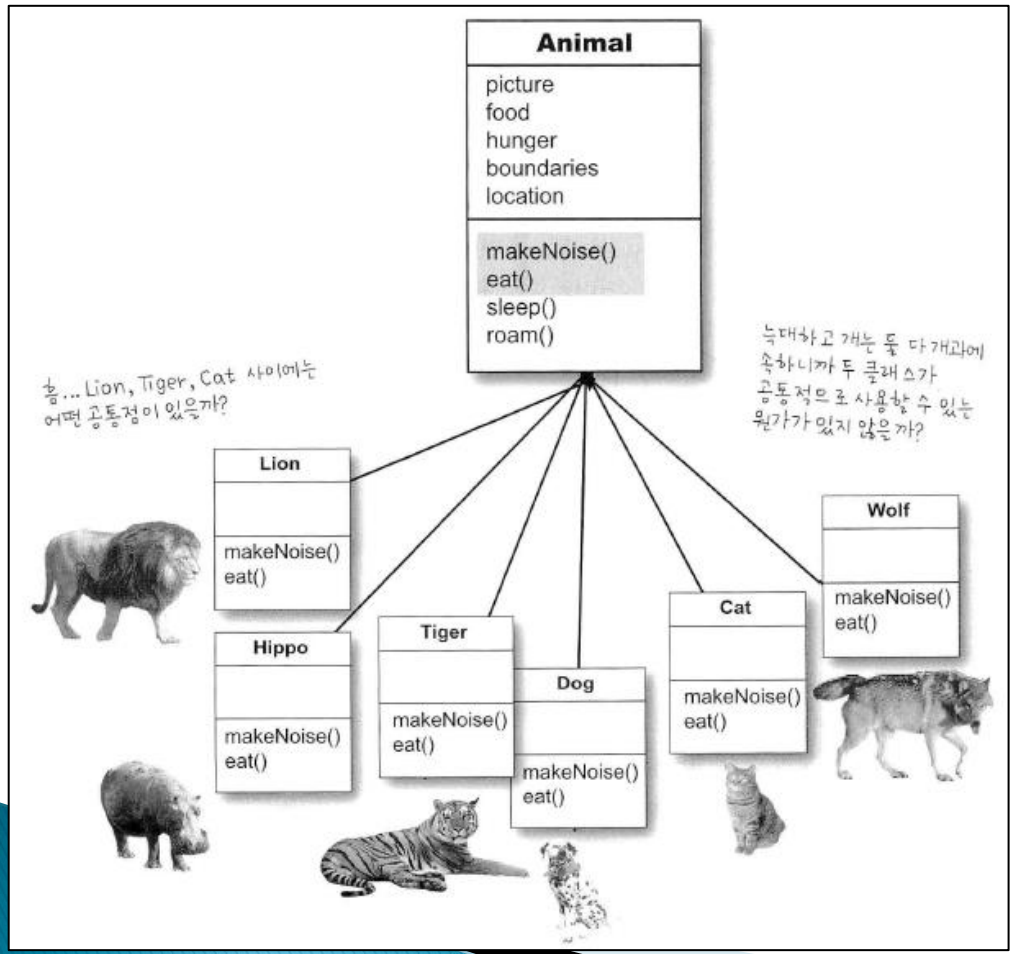
- How?
  - Child Class에서 Overriding하는 Method는
    - Parent Class의 Method와
      - Method 명 동일
      - Parameter 동일
      - Return Type 동일

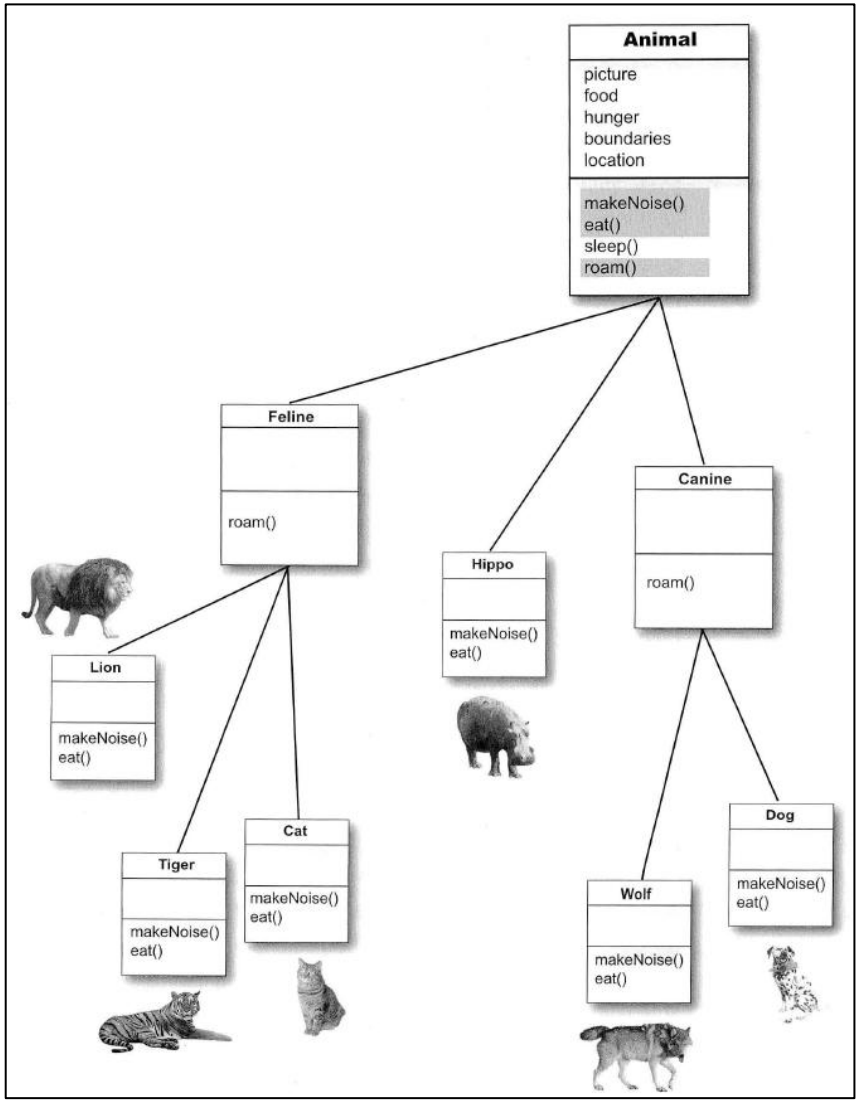
```
class Point {
    int x;
    int y;

    String getLocation(){
        return "x : " + x + ", " +
               "y : " + y;
    }
}
```

```
class Point3D extends Point {
    int x;
    String getLocation() {
        return "x : " + x + ", " +
               "y : " + y + ", " +
               "z : " + z;
    }
}
```

오버라이딩(Overriding)

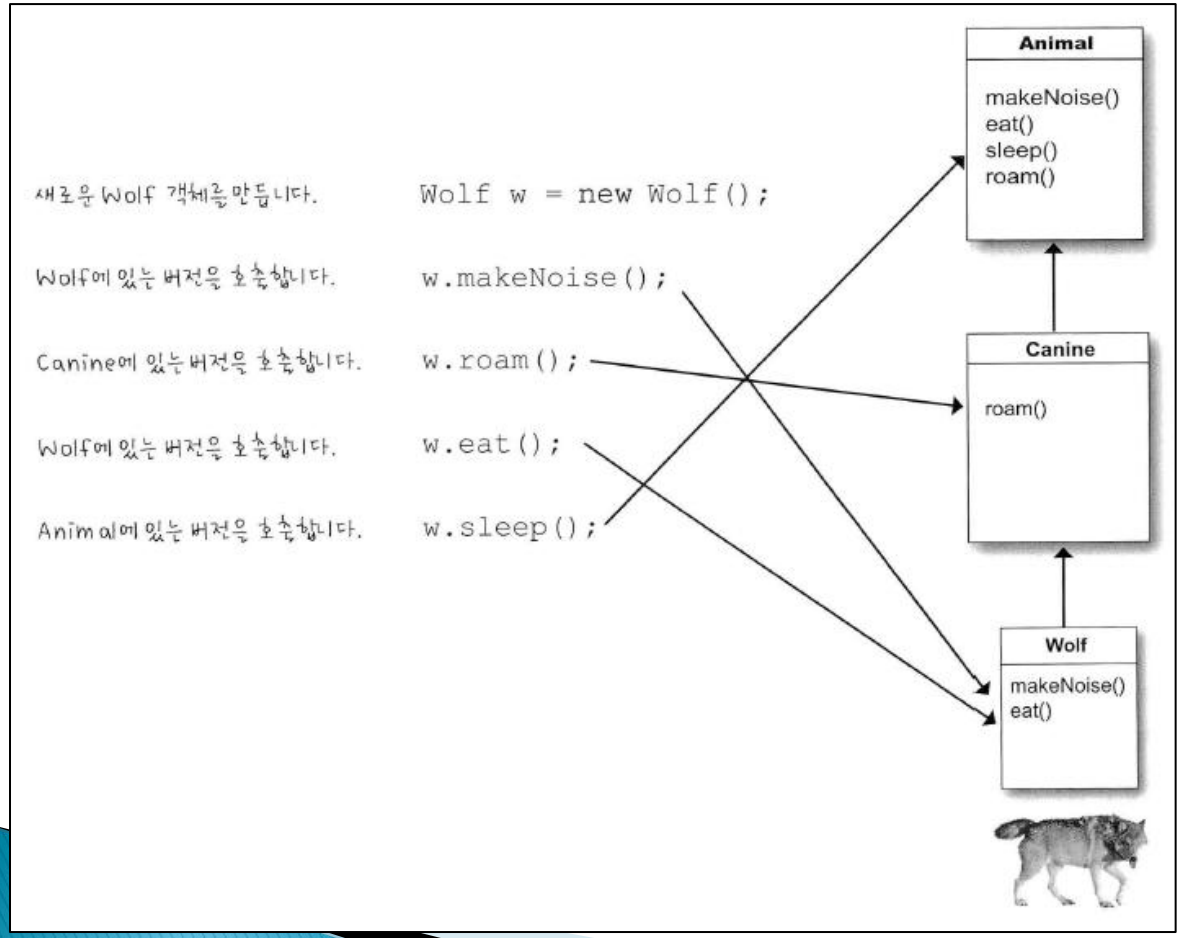
- example (from HeadFirst)





오버라이딩(Overriding)

- example (from HeadFirst)



오버라이딩(Overriding)

- overloading VS overriding

```
오버로딩(overloading)  - 기존에 없는 새로운 메서드를 정의하는 것(new)
오버라이딩(overriding) - 상속받은 메서드의 내용을 변경하는 것(change, modify)
```

```
class Parent {
    void parentMethod() {}
}

class Child extends Parent {
    void parentMethod() {}        // Overriding
    void parentMethod(int i) {} // Overloading

    void childMethod() {}
    void childMethod(int i) {}   // Overloading
    void childMethod() {}        // Error
}
```

오버라이딩(Overriding)

- Exception 범위에 따라

```
class Parent {
    void parentMethod() throws IOException,
                               SQLException {
        // . . .
    }
}

class Child extends Parent {
    void parentMethod() throws IOException {
        // . . .
    }
    // . . .
}
```

```
class Child extends Parent {
    void parentMethod() throws Exception {
        // . . .
    }
    // . . .
}
```

조상 클래스의 메서드를 자손 클래스에서 오버라이딩할 때

1. 접근 제어자를 조상 클래스의 메서드보다 좁은 범위로 변경할 수 없다.
2. 예외는 조상 클래스의 메서드보다 많이 선언할 수 없다.
3. 인스턴스메서드를 static메서드로 또는 그 반대로 변경할 수 없다.

오버로딩과 오버라이딩

- 면접 질문 단골 메뉴

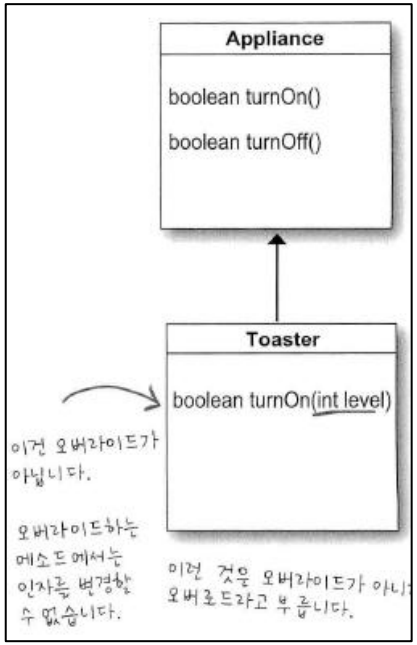
```
오버로딩(overloading) - 기존에 없는 새로운 메서드를 정의하는 것(new)
오버라이딩(overriding) - 상속받은 메서드의 내용을 변경하는 것(change, modify)
```

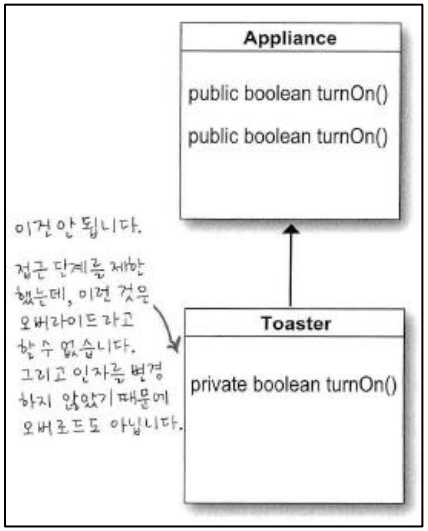
```
class Parent {
    void parentMethod() {}
}

class Child extends Parent {
    void parentMethod() {}        // Overriding
    void parentMethod(int i) {} // Overloading

    void childMethod() {}
    void childMethod(int i) {}   // Overloading
    void childMethod() {}        // Error
}
```

오버로딩과 오버라이딩

① **인자는 똑같아야 하고, 리턴 유형은 호환 가능해야 합니다.**

상위클래스에서 어떤 인자를 받아들이든 오버라이드하는 메소드에서는 똑같은 인자를 사용해야 합니다. 그리고 상위클래스에서 어떤 리턴 유형을 선언하든지 오버라이드하는 메소드에서는 똑같은 유형, 또는 하위클래스 유형을 리턴하는 것으로 선언해야 합니다. 이미 배웠듯이 하위클래스 객체에서는 상위클래스에서 선언한 것이면 무엇이든 할 수 있어야 하기 때문에 상위클래스 객체가 리턴될 자리에 하위클래스 객체를 리턴해도 문제가 되지 않겠죠.





② **메소드를 더 접근하기 어렵게 만들면 안 됩니다.**

상위클래스의 계약서에서는 다른 코드에서 어떻게 메소드를 사용할 수 있는지를 정의합니다. 즉, 접근 단계는 그대로 유지하거나 완화시켜야 합니다. 예를 들어, public 메소드를 오버라이드해서 private 메소드를 만들 수는 없습니다(컴파일할 때를 기준으로 하면). public 메소드라고 생각하고 호출했는데, 실행할 때 호출한 오버라이드하는 버전이 갑자기 private라고 하면서 JVM에서 접근을 금지하면 얼마나 황당하겠습니까?

지금까지는 두 가지 접근 단계(private와 public)에 대해 배웠습니다. 나머지 두 접근 단계는 17장과 부록 B에서 알아보겠습니다. 예외 처리와 관련된 오버라이딩 관련 규칙이 한 가지 더 있는데, 그 규칙에 대해서는 나중에 예외에 대한 장에서 살펴보도록 하겠습니다.

오버로딩과 오버라이딩

① 리턴 유형이 달라도 됩니다.

메소드를 오버로드할 때는 인자 목록만 다르면 리턴 유형을 마음대로 바꿔도 됩니다.

② 리턴 유형만 바꿀 수는 없습니다.

리턴 유형만 다르게 하는 것은 올바른 오버로딩이 아닙니다. 컴파일러에서는 프로그래머가 메소드를 오버라이드하려는 것으로 간주하게 됩니다. 게다가 리턴 형식이 상위클래스에서 선언된 리턴 유형하고 같거나 그 하위 유형이 아닌 경우에는 컴파일러에서 오류가 날 것입니다. 메소드를 오버로딩할 때는 리턴 유형하고는 무관하게 인자 목록을 반드시 변경해야 합니다.

③ 접근 단계를 마음대로 바꿀 수 있습니다.

메소드를 오버로드해서 더 제한이 심한 메소드를 만들어도 됩니다. 새로운 메소드가 오버로드된 메소드의 계약 조건을 이행해야 하는 것은 아니기 때문에 전혀 문제될 것이 없습니다.

메소드 오버로딩의 예:

```
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) {
    // 여러 가지 검증 과정을 거치고 나서 다음을 실행
    uniqueID = theID;
}

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

오늘 숙제

- 생성자 복습
  - 사용법은 '이거 어디서 봤던건데' 할 정도는 기억
  - 단, Class 에서 Object가 생성되는 과정에 대해서는 반드시 이해

- 변수의 초기화 블록 복습
  - 사용법보다는 실제 변수들의 값을 줄 때 실제 수행되는 순서를 이해

- p.34 우측 관계도 대로 클래스를 제작
  - 각 동물별 클래스를 상속관계를 정의하여 제작
  - 각 동물 클래스의 객체를 생성하여 메소드들을 실행하기 위한 테스트용 클래스 제작

- 5일차 숙제였던 구구단 메소드를 기반으로 제작

  - 메소드1을 오버라이딩하는 메소드4 : while 반복문을 사용
  - 메소드2를 오버라이딩하는 메소드5 : for 반목문을 사용
  - 메소드3을 오버라이딩하는 메소드6 : 홀수 인덱스를 가진 값을 정순으로 출력하되, 마지막 인덱스는 출력하지 않는 메소드

  - [실행]: 2~9까지 구구단을 짝수단일 경우 메소드4 사용, 홀수단일 경우 메소드5 사용하여, 반환받은 1차원 배열을 메소드6을 통해 출력