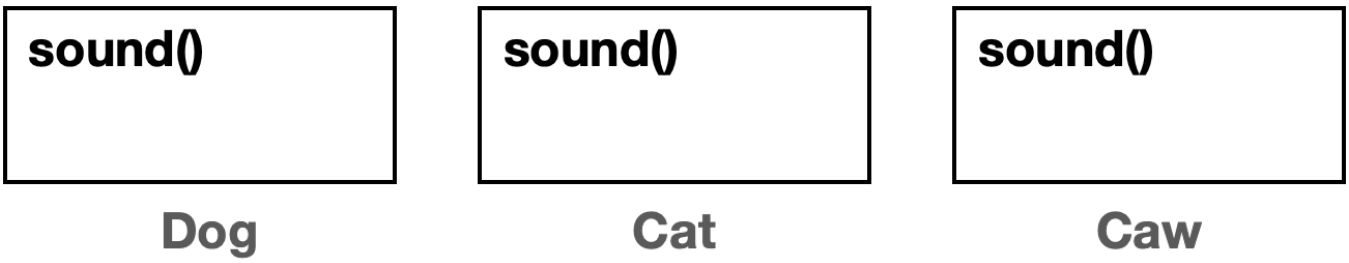
11. 다형성2

#1.인강/0.자바/2.자바-기본

다형성 활용1

지금까지 학습한 다형성을 왜 사용하는지, 그 장점을 알아보기 위해 우선 다형성을 사용하지 않고 프로그램을 개발한 다음에 다형성을 사용하도록 코드를 변경해보자.
아주 단순하고 전통적인 동물 소리 문제로 접근해보자.



개, 고양이, 소의 울음 소리를 테스트하는 프로그램을 작성해보자. 먼저 다형성을 사용하지 않고 코드를 작성해보자.

참고: `Caw`는 오타입니다. 대신에 `Cow`로 정정해야 맞습니다. 강의 내용과 영상 전반에 `Cow`를 너무 많이 사용해서, 고치면 코드를 따라갈 때 혼란스러울 수 있어서, 오타이지만 부득이하게 `Caw`로 유지하겠습니다.

예제1

```
package poly.ex1;

public class Dog {
    public void sound() {
        System.out.println("멍멍");
```

```
    }
}
```

```
package poly.ex1;

public class Cat {
    public void sound() {
        System.out.println("냐옹");
    }
}
```

```
package poly.ex1;

public class Caw {
    public void sound() {
        System.out.println("음매");
    }
}
```

```
package poly.ex1;

public class AnimalSoundMain {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();

        System.out.println("동물 소리 테스트 시작");
        dog.sound();
        System.out.println("동물 소리 테스트 종료");

        System.out.println("동물 소리 테스트 시작");
        cat.sound();
        System.out.println("동물 소리 테스트 종료");

        System.out.println("동물 소리 테스트 시작");
        caw.sound();
        System.out.println("동물 소리 테스트 종료");
    }
}
```

실행 결과

```
동물 소리 테스트 시작
멍멍
동물 소리 테스트 종료

동물 소리 테스트 시작
냐옹
동물 소리 테스트 종료

동물 소리 테스트 시작
음매
동물 소리 테스트 종료
```

단순히 개, 고양이, 소 동물들의 울음 소리를 출력하는 프로그램이다. 만약 여기서 새로운 동물이 추가되면 어떻게 될까?
만약 기존 코드에 소가 없었다고 가정해보자, 소가 추가된다고 가정하면 `Caw` 클래스를 만들고 다음 코드도 추가해야 한다.

```
//Caw를 생성하는 코드
Caw caw = new Caw();

//Caw를 사용하는 코드
System.out.println("동물 소리 테스트 시작");
caw.sound();
System.out.println("동물 소리 테스트 종료");
```

`Caw`를 생성하는 부분은 당연히 필요하니 크게 상관이 없지만, `Dog`, `Cat`, `Caw`를 사용해서 출력하는 부분은 계속 중복이 증가한다.

중복 코드

```
System.out.println("동물 소리 테스트 시작");
dog.sound();
System.out.println("동물 소리 테스트 종료");

System.out.println("동물 소리 테스트 시작");
cat.sound();
System.out.println("동물 소리 테스트 종료");
...
```

이 부분의 중복을 제거할 수 있을까?

중복을 제거하기 위해서는 메서드를 사용하거나, 또는 배열과 `for` 문을 사용하면 된다.
그런데 `Dog`, `Cat`, `Caw`는 서로 완전히 다른 클래스다.

중복 제거 시도

메서드로 중복 제거 시도

메서드를 사용하면 다음과 같이 매개변수의 클래스를 `Caw`, `Dog`, `Cat` 중에 하나로 정해야 한다.

```
private static void soundCaw(Caw caw) {
    System.out.println("동물 소리 테스트 시작");
    caw.sound();
    System.out.println("동물 소리 테스트 종료");
}
```

따라서 이 메서드는 `Caw` 전용 메서드가 되고 `Dog`, `Cat`은 인수로 사용할 수 없다.
`Dog`, `Cat`, `Caw`의 타입(클래스)이 서로 다르기 때문에 `soundCaw` 메서드를 함께 사용하는 것은 불가능하다.

배열과 for문을 통한 중복 제거 시도

```
Caw[] cawArr = {cat, dog, caw}; //컴파일 오류 발생!
System.out.println("동물 소리 테스트 시작");
    for (Caw caw : cawArr) {
        cawArr.sound();
}
System.out.println("동물 소리 테스트 종료");
```

배열과 for문 사용해서 중복을 제거하려고 해도 배열의 타입을 `Dog`, `Cat`, `Caw` 중에 하나로 지정해야 한다. 같은 `Caw`들을 배열에 담아서 처리하는 것은 가능하지만 타입이 서로 다른 `Dog`, `Cat`, `Caw`을 하나의 배열에 담는 것은 불가능하다.
결과적으로 지금 상황에서는 해결 방법이 없다. 새로운 동물이 추가될 때 마다 더 많은 중복 코드를 작성해야 한다.

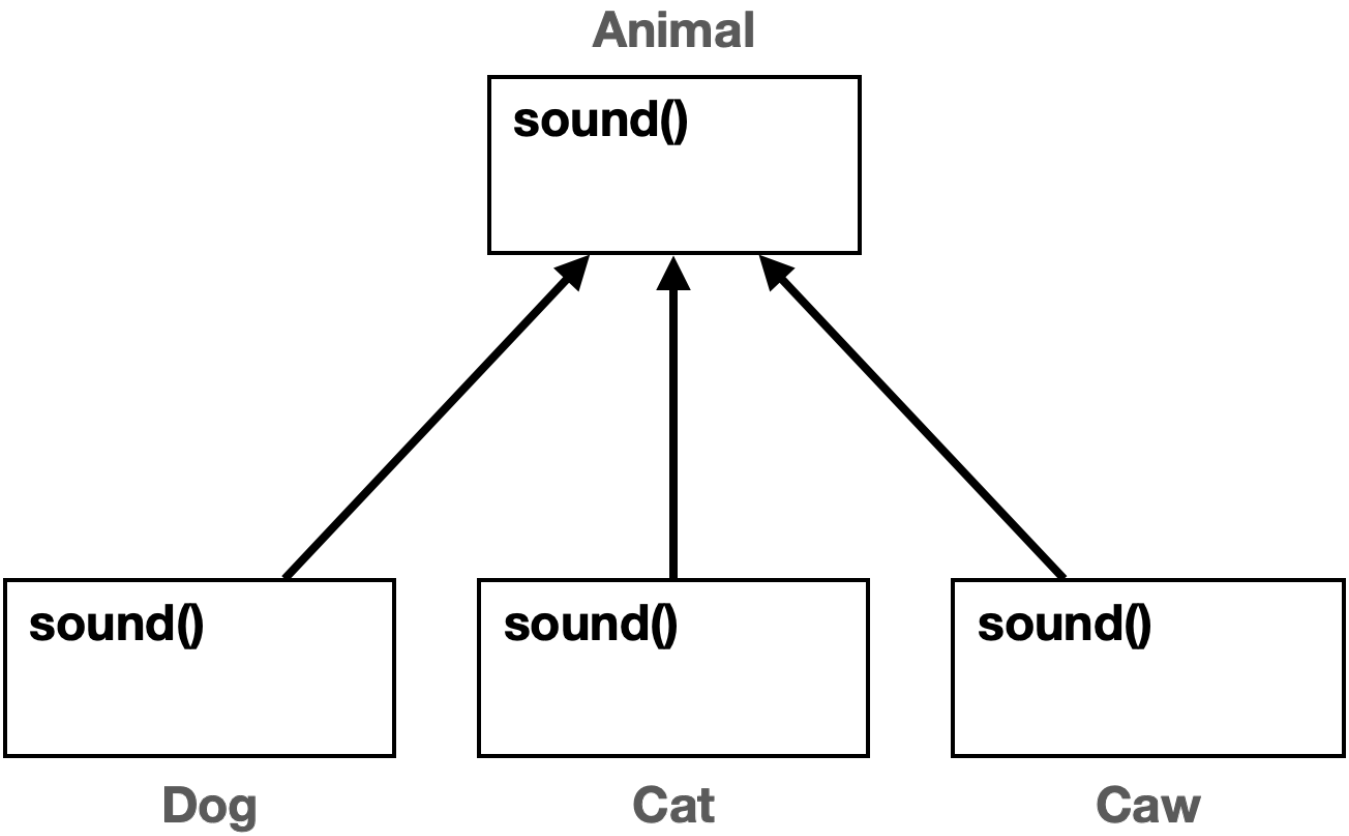
지금까지 설명한 모든 중복 제거 시도가 `Dog`, `Cat`, `Caw`의 타입이 서로 다르기 때문에 불가능하다. **문제의 핵심은 바로 타입이 다르다는 점**이다. 반대로 이야기하면 `Dog`, `Cat`, `Caw`가 모두 같은 타입을 사용할 수 있는 방법이 있다면 메서드와 배열을 활용해서 코드의 중복을 제거할 수 있다는 것이다.

다형성의 핵심은 다형적 참조와 메서드 오버라이딩이다. 이 둘을 활용하면 `Dog`, `Cat`, `Caw`가 모두 같은 타입을 사용하고, 각자 자신의 메서드도 호출할 수 있다.

다형성 활용2

이번에는 앞서 설명한 예제를 다형성을 사용하도록 변경해보자.

예제2



다형성을 사용하기 위해 여기서는 상속 관계를 사용한다. `Animal`(동물) 이라는 부모 클래스를 만들고 `sound()` 메서드를 정의한다. 이 메서드는 자식 클래스에서 오버라이딩 할 목적으로 만들었다.
`Dog`, `Cat`, `Caw`는 `Animal` 클래스를 상속받았다. 그리고 각각 부모의 `sound()` 메서드를 오버라이딩 한다.

기존 코드를 유지하기 위해 새로운 패키지를 만들고 새로 코드를 작성하자.
주의! 패키지 이름에 주의하자 `import`를 사용해서 다른 패키지에 있는 같은 이름의 클래스를 사용하면 안된다.

```
package poly.ex2;

public class Animal {
    public void sound() {
        System.out.println("동물 울음 소리");
    }
}
```

```
package poly.ex2;

public class Dog extends Animal {
```

```
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
}
```

```
package poly.ex2;

public class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("냐옹");
    }
}
```

```
package poly.ex2;

public class Caw extends Animal{
    @Override
    public void sound() {
        System.out.println("음매");
    }
}
```

```
package poly.ex2;

public class AnimalPolyMain1 {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();

        soundAnimal(dog);
        soundAnimal(cat);
        soundAnimal(caw);
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void soundAnimal(Animal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
```

```
        System.out.println("동물 소리 테스트 종료");
    }
}
```

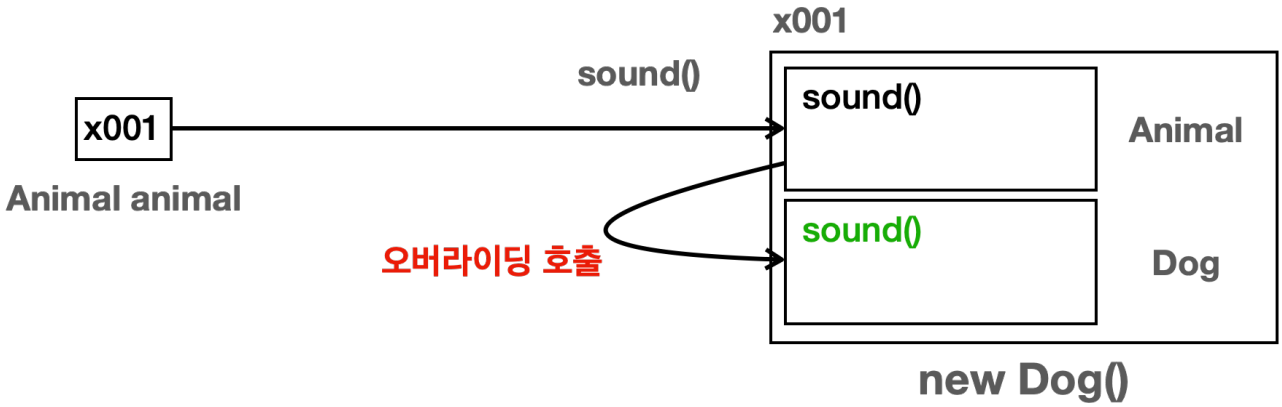
실행 결과

```
동물 소리 테스트 시작
멍멍
동물 소리 테스트 종료

동물 소리 테스트 시작
냐옹
동물 소리 테스트 종료

동물 소리 테스트 시작
음매
동물 소리 테스트 종료
```

실행 결과는 기존 코드와 같다.

코드를 분석해보자.

- `soundAnimal(dog)`을 호출하면
- `soundAnimal(Animal animal)`에 `Dog` 인스턴스가 전달된다.
  - `Animal animal = dog`로 이해하면 된다. 부모는 자식을 담을 수 있다. `Animal`은 `Dog`의 부모다.
- 메서드 안에서 `animal.sound()` 메서드를 호출한다.



- `animal` 변수의 타입은 `Animal`이므로 `Dog` 인스턴스에 있는 `Animal` 클래스 부분을 찾아서 `sound()` 메서드를 실행한다. 그런데 하위 클래스인 `Dog`에서 `sound()` 메서드를 오버라이딩 했다. 따라서 오버라이딩한 메서드가 우선권을 가진다.
- `Dog` 클래스에 있는 `sound()` 메서드가 호출되므로 "멍멍"이 출력된다.

이 코드의 핵심은 `Animal animal` 부분이다.

- **다형적 참조** 덕분에 `animal` 변수는 자식인 `Dog`, `Cat`, `Caw`의 인스턴스를 참조할 수 있다. (부모는 자식을 담

을 수 있다)

- **메서드 오버라이딩** 덕분에 `animal.sound()`를 호출해도 `Dog.sound()`, `Cat.sound()`, `Caw.sound()`와 같이 각 인스턴스의 메서드를 호출할 수 있다. 만약 자바에 메서드 오버라이딩이 없었다면 모두 `Animal`의 `sound()`가 호출되었을 것이다.

다형성 덕분에 이후에 새로운 동물을 추가해도 다음 코드를 그대로 재사용 할 수 있다. 물론 다형성을 사용하기 위해 새로운 동물은 `Animal`을 상속 받아야 한다.

```
private static void soundAnimal(Animal animal) {
    System.out.println("동물 소리 테스트 시작");
    animal.sound();
    System.out.println("동물 소리 테스트 종료");
}
```

다형성 활용3

이번에는 배열과 for문을 사용해서 중복을 제거해보자.

```
package poly.ex2;

public class AnimalPolyMain2 {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();
        Animal[] animalArr = {dog, cat, caw};

        //변하지 않는 부분
        for (Animal animal : animalArr) {
            System.out.println("동물 소리 테스트 시작");
            animal.sound();
            System.out.println("동물 소리 테스트 종료");
        }
    }
}
```

실행 결과

```
동물 소리 테스트 시작
```

```
멍멍
동물 소리 테스트 종료
동물 소리 테스트 시작
냐옹
동물 소리 테스트 종료
동물 소리 테스트 시작
음매
동물 소리 테스트 종료
```

배열은 같은 타입의 데이터를 나열할 수 있다.
`Dog`, `Cat`, `Caw`는 모두 `Animal`의 자식이므로 `Animal` 타입이다.

`Animal` 타입의 배열을 만들고 다형적 참조를 사용하면 된다.

```
//둘은 같은 코드이다.
Animal[] animalArr = new Animal[]{dog, cat, caw};
Animal[] animalArr = {dog, cat, caw}
```

다형적 참조 덕분에 `Dog`, `Cat`, `Caw`의 부모 타입인 `Animal` 타입으로 배열을 만들고, 각각을 배열에 포함했다.

이제 배열을 for문을 사용해서 반복하면 된다.

```
//변하지 않는 부분
for (Animal animal : animalArr) {
    System.out.println("동물 소리 테스트 시작");
    animal.sound();
    System.out.println("동물 소리 테스트 종료");
}
```

`animal.sound()`를 호출하지만 배열에는 `Dog`, `Cat`, `Caw`의 인스턴스가 들어있다. 메서드 오버라이딩에 의해 각 인스턴스의 오버라이딩 된 `sound()` 메서드가 호출된다.

조금 더 개선

이번에는 배열과 메서드 모두 활용해서 기존 코드를 완성해보자.

```
package poly.ex2;

public class AnimalPolyMain3 {

    public static void main(String[] args) {
        Animal[] animalArr = {new Dog(), new Cat(), new Caw()};
        for (Animal animal : animalArr) {
```

```
            soundAnimal(animal);
        }
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void soundAnimal(Animal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }
}
```

- `Animal[] animalArr` 를 통해서 배열을 사용한다.
- `soundAnimal(Animal animal)`
  - 하나의 동물을 받아서 로직을 처리한다.

새로운 동물이 추가되어도 `soundAnimal(..)` 메서드는 코드 변경 없이 유지할 수 있다. 이렇게 할 수 있는 이유는 이 메서드는 `Dog`, `Cat`, `Caw` 같은 구체적인 클래스를 참조하는 것이 아니라 `Animal` 이라는 추상적인 부모를 참조하기 때문이다. 따라서 `Animal` 을 상속 받은 새로운 동물이 추가되어도 이 메서드의 코드는 변경 없이 유지할 수 있다.

여기서 잘 보면 새로운 동물이 추가되었을 때 코드가 변하는 부분과 변하지 않는 부분이 있다.
`main()` 은 코드가 변하는 부분이다. 새로운 동물을 생성하고 필요한 메서드를 호출한다.
`soundAnimal(..)` 는 코드가 변하지 않는 부분이다.

새로운 기능이 추가되었을 때 변하는 부분을 최소화 하는 것이 잘 작성된 코드이다. 이렇게 하기 위해서는 코드에서 변하는 부분과 변하지 않는 부분을 명확하게 구분하는 것이 좋다.

남은 문제

지금까지 설명한 코드에는 사실 2가지 문제가 있다.

- `Animal` 클래스를 생성할 수 있는 문제
- `Animal` 클래스를 상속 받는 곳에서 `sound()` 메서드 오버라이딩을 하지 않을 가능성

Animal 클래스를 생성할 수 있는 문제

`Animal` 클래스는 동물이라는 클래스이다. 이 클래스를 다음과 같이 직접 생성해서 사용할 일이 있을까?

```
Animal animal = new Animal();
```

개, 고양이, 소가 실제 존재하는 것은 당연하지만, 동물이라는 추상적인 개념이 실제로 존재하는 것은 이상하다. 사실 이 클래스는 다형성을 위해서 필요한 것이지 직접 인스턴스를 생성해서 사용할 일은 없다.

하지만 `Animal`도 클래스이기 때문에 인스턴스를 생성하고 사용하는데 아무런 제약이 없다. 누군가 실수로 `new Animal()`을 사용해서 `Animal`의 인스턴스를 생성할 수 있다는 것이다. 이렇게 생성된 인스턴스는 작동은 하지만 제대로된 기능을 수행하지는 않는다.

Animal 클래스를 상속 받는 곳에서 `sound()` 메서드 오버라이딩을 하지 않을 가능성
예를 들어서 `Animal`을 상속 받은 `Pig` 클래스를 만든다고 가정해보자. 우리가 기대하는 것은 `Pig` 클래스가 `sound()` 메서드를 오버라이딩 해서 "꿀꿀" 이라는 소리가 나도록 하는 것이다. 그런데 개발자가 실수로 `sound()` 메서드를 오버라이딩 하는 것을 빠트릴 수 있다. 이렇게 되면 부모의 기능을 상속 받는다. 따라서 코드상 아무런 문제가 발생하지 않는다. 물론 프로그램을 실행하면 기대와 다르게 "꿀꿀"이 아니라 부모 클래스에 있는 `Animal.sound()`가 호출될 것이다.

좋은 프로그램은 제약이 있는 프로그램이다. 추상 클래스와 추상 메서드를 사용하면 이런 문제를 한번에 해결할 수 있다.

추상 클래스1

추상 클래스
동물(`Animal`)과 같이 부모 클래스는 제공하지만, 실제 생성되면 안되는 클래스를 추상 클래스라 한다.
추상 클래스는 이름 그대로 추상적인 개념을 제공하는 클래스이다. 따라서 실체인 인스턴스가 존재하지 않는다. 대신에 상속을 목적으로 사용되고, 부모 클래스 역할을 담당한다.

```
abstract class AbstractAnimal {...}
```

- 추상 클래스는 클래스를 선언할 때 앞에 추상이라는 의미의 `abstract` 키워드를 붙여주면 된다.
- 추상 클래스는 기존 클래스와 완전히 같다. 다만 `new AbstractAnimal()`와 같이 직접 인스턴스를 생성하지 못하는 제약이 추가된 것이다.

추상 메서드
부모 클래스를 상속 받는 자식 클래스가 반드시 오버라이딩 해야 하는 메서드를 부모 클래스에 정의할 수 있다. 이것을 추상 메서드라 한다. 추상 메서드는 이름 그대로 추상적인 개념을 제공하는 메서드이다. 따라서 실체가 존재하지 않고, 메서드 바디가 없다.

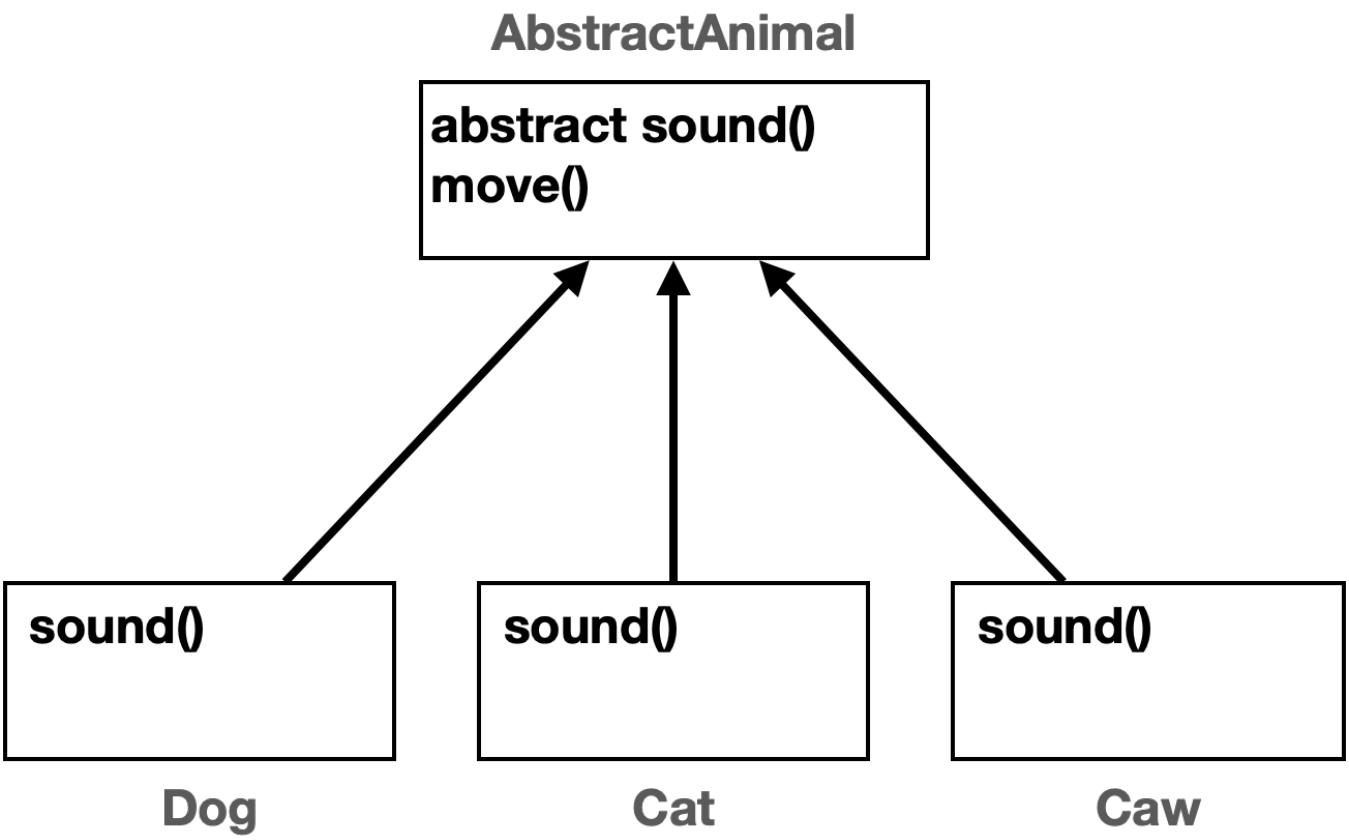
```
public abstract void sound();
```

- 추상 메서드는 선언할 때 메서드 앞에 추상이라는 의미의 `abstract` 키워드를 붙여주면 된다.
- **추상 메서드가 하나라도 있는 클래스는 추상 클래스로 선언해야 한다.**

    - 그렇지 않으면 컴파일 오류가 발생한다.
    - 추상 메서드는 메서드 바디가 없다. 따라서 작동하지 않는 메서드를 가진 불완전한 클래스로 볼 수 있다. 따라서 직접 생성하지 못하도록 추상 클래스로 선언해야 한다.
- **추상 메서드는 상속 받는 자식 클래스가 반드시 오버라이딩 해서 사용해야 한다.**
    - 그렇지 않으면 컴파일 오류가 발생한다.
    - 추상 메서드는 자식 클래스가 반드시 오버라이딩 해야 하기 때문에 메서드 바디 부분이 없다. 바디 부분을 만들면 컴파일 오류가 발생한다.
    - 오버라이딩 하지 않으면 자식도 추상 클래스가 되어야 한다.
- 추상 메서드는 기존 메서드와 완전히 같다. 다만 메서드 바디가 없고, 자식 클래스가 해당 메서드를 반드시 오버라이딩 해야 한다는 제약이 추가된 것이다.

이제 추상 클래스와 추상 메서드를 사용해서 예제를 만들어보자.

예제3




```
package poly.ex3;

public abstract class AbstractAnimal {
    public abstract void sound();

    public void move() {
```

```
        System.out.println("동물이 움직입니다.");
    }
}
```

- `AbstractAnimal`은 `abstract`가 붙은 추상 클래스이다. 이 클래스는 직접 인스턴스를 생성할 수 없다.
- `sound()`는 `abstract`가 붙은 추상 메서드이다. 이 메서드는 자식이 반드시 오버라이딩 해야 한다.

이 클래스는 `move()`라는 메서드를 가지고 있는데, 이 메서드는 추상 메서드가 아니다. 따라서 자식 클래스가 오버라이딩 하지 않아도 된다.

참고로 추상 클래스라고 `AbstractAnimal`처럼 클래스 이름 앞에 꼭 `Abstract`를 써야하는 것은 아니다. 그냥 `Animal` 이라는 클래스 이름으로도 충분하다. 여기서는 예제 코드를 다른 예제 코드와 구분해서 설명하기 위해 앞에 `Abstract`를 붙였다.

```
package poly.ex3;

public class Dog extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
}
```

```
package poly.ex3;

public class Cat extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("냐옹");
    }
}
```

```
package poly.ex3;

public class Caw extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("음매");
    }
}
```

```java
package poly.ex3;

public class AbstractMain {
    public static void main(String[] args) {

        //추상클래스 생성 불가
        //AbstractAnimal animal = new AbstractAnimal();

        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();

        cat.sound();
        cat.move();

        soundAnimal(cat);
        soundAnimal(dog);
        soundAnimal(caw);
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void soundAnimal(AbstractAnimal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }
}
```

실행 결과

```
냐옹
동물이 움직입니다.

동물 소리 테스트 시작
냐옹
동물 소리 테스트 종료

동물 소리 테스트 시작
멍멍
동물 소리 테스트 종료

동물 소리 테스트 시작
```

```
음매
동물 소리 테스트 종료
```

추상 클래스는 생성이 불가능하다. 다음 코드의 주석을 풀고 실행하면 컴파일 오류가 발생한다.

```
//추상클래스 생성 불가
AbstractAnimal animal = new AbstractAnimal();
```

컴파일 오류 - 인스턴스 생성

```
java: poly.ex3.AbstractAnimal is abstract; cannot be instantiated
```

`AbstractAnimal`가 추상이어서 인스턴스 생성이 불가능하다는 뜻이다.

추상 메서드는 반드시 오버라이딩 해야 한다. 만약 자식에서 오버라이딩 메서드를 만들지 않으면 다음과 같이 컴파일 오류가 발생한다. `Dog`의 `sound()` 메서드를 잠시 주석처리해보자.

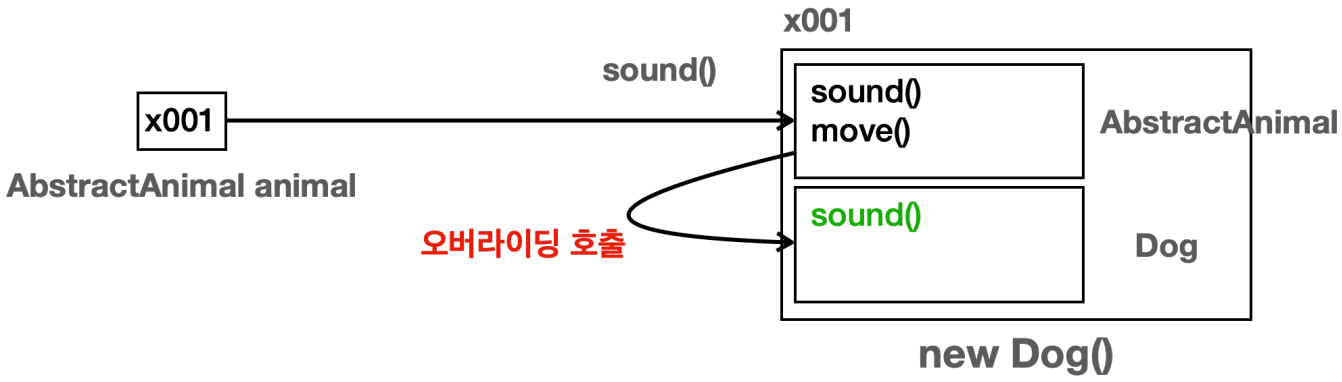
```
package poly.ex3;

public class Dog extends AbstractAnimal {
/*
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
*/
}
```

컴파일 오류 - 오버라이딩X

```
java: poly.ex3.Dog is not abstract and does not override abstract method sound()
in poly.ex3.AbstractAnimal
```

`Dog`는 추상클래스가 아닌데 `sound()`가 오버라이딩 되지 않았다는 뜻이다.

지금까지 설명한 제약을 제외하고 나머지는 모두 일반적인 클래스와 동일하다. 추상 클래스는 제약이 추가된 클래스일 뿐이다. 메모리 구조, 실행 결과 모두 동일하다.



정리

- 추상 클래스 덕분에 실수로 `Animal` 인스턴스를 생성할 문제를 근본적으로 방지해준다.
- 추상 메서드 덕분에 새로운 동물의 자식 클래스를 만들때 실수로 `sound()` 를 오버라이딩 하지 않을 문제를 근본적으로 방지해준다.

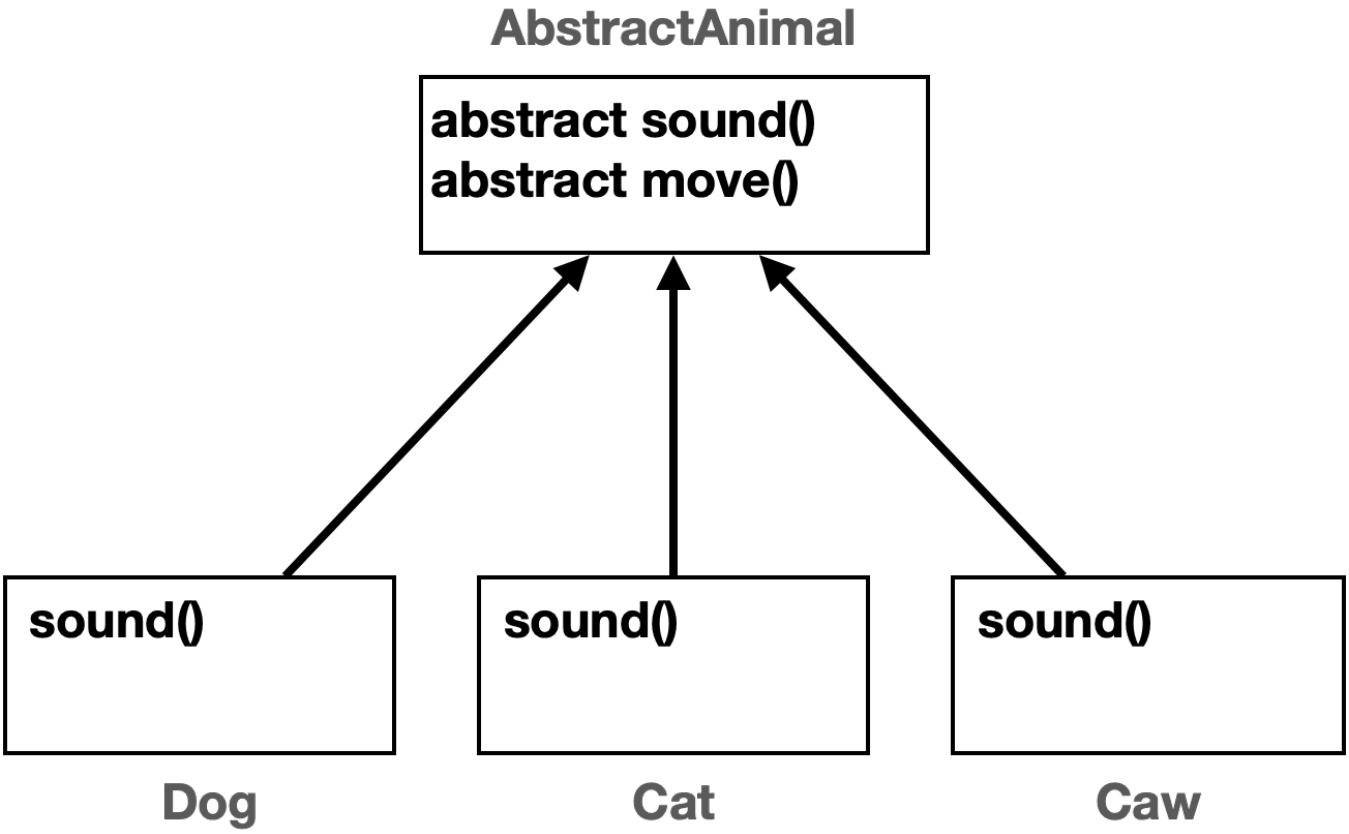
추상 클래스2

순수 추상 클래스: 모든 메서드가 추상 메서드인 추상 클래스

앞서 만든 예제에서 `move()` 도 추상 메서드로 만들어야 한다고 가정해보자.

이 경우 `AbstractAnimal` 클래스의 모든 메서드가 추상 메서드가 된다. 이런 클래스를 순수 추상 클래스라 한다.

`move()` 가 추상 메서드가 되었으니 자식들은 `AbstractAnimal` 의 모든 기능을 오버라이딩 해야 한다.



예제4

```
package poly.ex4;

public abstract class AbstractAnimal {
    public abstract void sound();
    public abstract void move();
}
```

```
package poly.ex4;

public class Dog extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("멍멍");
    }

    @Override
    public void move() {
        System.out.println("개 이동");
    }
}
```

```java
package poly.ex4;

public class Cat extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("냐옹");
    }

    @Override
    public void move() {
        System.out.println("고양이 이동");
    }
}
```

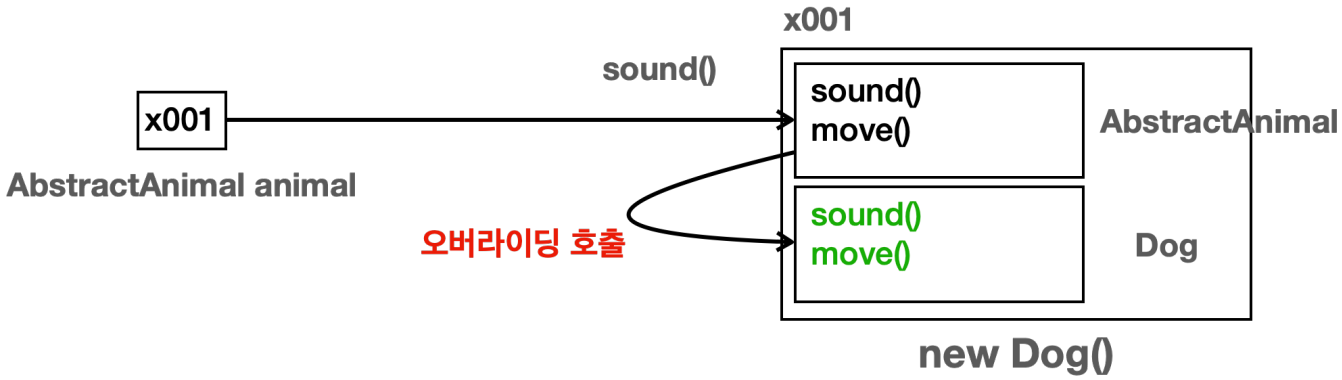
```java
package poly.ex4;

public class Caw extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("음매");
    }

    @Override
    public void move() {
        System.out.println("소 이동");
    }
}
```

```java
package poly.ex4;

public class AbstractMain {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();

        soundAnimal(cat);
        soundAnimal(dog);
        soundAnimal(caw);

        moveAnimal(cat);
        moveAnimal(dog);
```

```
        moveAnimal(caw);
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void soundAnimal(AbstractAnimal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void moveAnimal(AbstractAnimal animal) {
        System.out.println("동물 이동 테스트 시작");
        animal.move();
        System.out.println("동물 이동 테스트 종료");
    }
}
```

실행 결과

```
동물 소리 테스트 시작
냐옹
동물 소리 테스트 종료
...
동물 이동 테스트 시작
고양이 이동
동물 이동 테스트 종료
동물 이동 테스트 시작
개 이동
동물 이동 테스트 종료
동물 이동 테스트 시작
소 이동
동물 이동 테스트 종료
```



코드를 이해하는데 어려움은 없을 것이다.

순수 추상 클래스

모든 메서드가 추상 메서드인 순수 추상 클래스는 코드를 실행할 바디 부분이 전혀 없다.

```
public abstract class AbstractAnimal {
    public abstract void sound();
    public abstract void move();
}
```

이러한 순수 추상 클래스는 실행 로직을 전혀 가지고 있지 않다. 단지 다형성을 위한 부모 타입으로써 껍데기 역할만 제공할 뿐이다.

순수 추상 클래스는 다음과 같은 특징을 가진다.

- 인스턴스를 생성할 수 없다.
- 상속시 자식은 모든 메서드를 오버라이딩 해야 한다.
- 주로 다형성을 위해 사용된다.

상속하는 클래스는 모든 메서드를 구현해야 한다.

"상속시 자식은 모든 메서드를 오버라이딩 해야 한다."라는 특징은 상속 받는 클래스 입장에서 보면 부모의 모든 메서드를 구현해야 하는 것이다.
이런 특징을 잘 생각해보면 순수 추상 클래스는 마치 어떤 규격을 지켜서 구현해야 하는 것 처럼 느껴진다.
`AbstractAnimal`의 경우 `sound()`, `move()`라는 규격에 맞추어 구현을 해야 한다.

이것은 우리가 일반적으로 이야기하는 인터페이스와 같이 느껴진다. 예를 들어서 USB 인터페이스를 생각해보자. USB 인터페이스는 분명한 규격이 있다. 이 규격에 맞추어 제품을 개발해야 연결이 된다. 순수 추상 클래스가 USB 인터페이스 규격이라고 한다면 USB 인터페이스에 맞추어 마우스, 키보드 같은 연결 장치들을 구현할 수 있다.

이런 순수 추상 클래스의 개념은 프로그래밍에서 매우 자주 사용된다. 자바는 순수 추상 클래스를 더 편리하게 사용할 수 있도록 인터페이스라는 개념을 제공한다.

인터페이스

자바는 순수 추상 클래스를 더 편리하게 사용할 수 있는 인터페이스라는 기능을 제공한다.

순수 추상 클래스

```
public abstract class AbstractAnimal {
    public abstract void sound();
    public abstract void move();
}
```

인터페이스는 `class`가 아니라 `interface` 키워드를 사용하면 된다.

인터페이스

```
public interface InterfaceAnimal {
    public abstract void sound();
    public abstract void move();
}
```

인터페이스 - public abstract 키워드 생략 가능

```
public interface InterfaceAnimal {
    void sound();
    void move();
}
```

순수 추상 클래스는 다음과 같은 특징을 가진다.

- 인스턴스를 생성할 수 없다.
- 상속시 모든 메서드를 오버라이딩 해야 한다.
- 주로 다형성을 위해 사용된다.

인터페이스는 앞서 설명한 순수 추상 클래스와 같다. 여기에 약간의 편의 기능이 추가된다.

- 인터페이스의 메서드는 모두 `public`, `abstract`이다.
- 메서드에 `public abstract`를 생략할 수 있다. 참고로 생략이 권장된다.
- 인터페이스는 다중 구현(다중 상속)을 지원한다.

인터페이스와 멤버 변수

```
public interface InterfaceAnimal {
    public static final double MY_PI = 3.14;
}
```

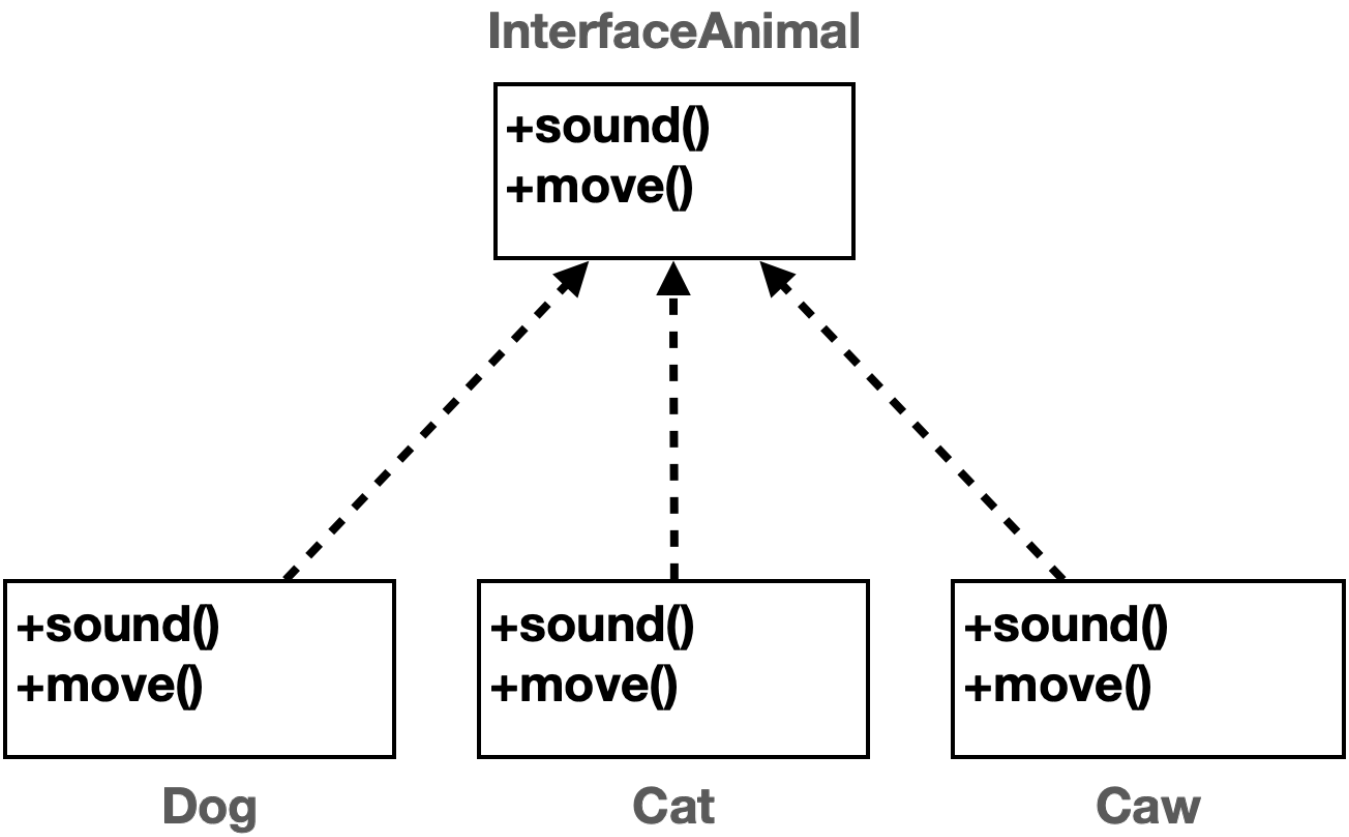
인터페이스에서 멤버 변수는 `public`, `static`, `final`이 모두 포함되었다고 간주된다. `final`은 변수의 값을 한 번 설정하면 수정할 수 없다는 뜻이다.

자바에서 `static final`을 사용해 정적이면서 고칠 수 없는 변수를 상수라 하고, 관례상 상수는 대문자에 언더스코어(`_`)로 구분한다.

해당 키워드는 다음과 같이 생략할 수 있다. (생략이 권장된다.)

```
public interface InterfaceAnimal {
    double MY_PI = 3.14;
}
```

예제5



클래스 상속 관계는 UML에서 실선을 사용하지만, 인터페이스 구현(상속) 관계는 UML에서 점선을 사용한다.

```
package poly.ex5;

public interface InterfaceAnimal {
    void sound();
    void move();
}
```

인터페이스는 `class` 대신에 `interface`로 선언하면 된다.

`sound()`, `move()`는 앞에 `public abstract`가 생략되어 있다. 따라서 상속 받는 곳에서 모든 메서드를 오버라이딩 해야 한다.

```java
package poly.ex5;

public class Dog implements InterfaceAnimal {
    @Override
    public void sound() {
        System.out.println("멍멍");
    }

    @Override
    public void move() {
        System.out.println("개 이동");
    }
}
```

인터페이스를 상속 받을 때는 `extends` 대신에 `implements` 라는 **구현**이라는 키워드를 사용해야 한다. 인터페이스는 그래서 상속이라 하지 않고 구현이라 한다.

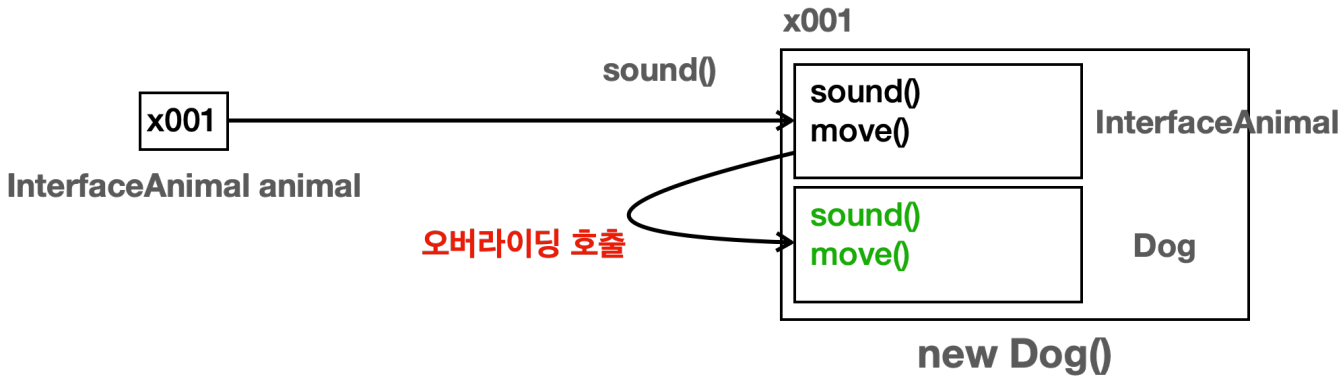
```java
package poly.ex5;

public class Cat implements InterfaceAnimal {
    @Override
    public void sound() {
        System.out.println("냐옹");
    }

    @Override
    public void move() {
        System.out.println("고양이 이동");
    }
}
```

```java
package poly.ex5;

public class Caw implements InterfaceAnimal {
    @Override
    public void sound() {
        System.out.println("음매");
    }

    @Override
    public void move() {
        System.out.println("소 이동");
    }
```

```
}
```

```
package poly.ex5;

public class InterfaceMain {
    public static void main(String[] args) {

        //인터페이스 생성 불가
        //InterfaceAnimal interfaceMain1 = new InterfaceAnimal();

        Cat cat = new Cat();
        Dog dog = new Dog();
        Caw caw = new Caw();

        soundAnimal(cat);
        soundAnimal(dog);
        soundAnimal(caw);
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void soundAnimal(InterfaceAnimal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }
}
```

```
동물 소리 테스트 시작
냐옹
동물 소리 테스트 종료

동물 소리 테스트 시작
멍멍
동물 소리 테스트 종료

동물 소리 테스트 시작
음매
동물 소리 테스트 종료
```



앞서 설명한 순수 추상 클래스 예제와 거의 유사하다. 순수 추상 클래스가 인터페이스가 되었을 뿐이다.

클래스, 추상 클래스, 인터페이스는 모두 똑같다.

- 클래스, 추상 클래스, 인터페이스는 프로그램 코드, 메모리 구조상 모두 똑같다. 모두 자바에서는 `.class`로 다루어진다. 인터페이스를 작성할 때도 `.java`에 인터페이스를 정의한다.
- 인터페이스는 순수 추상 클래스와 비슷하다고 생각하면 된다.

상속 vs 구현

부모 클래스의 기능을 자식 클래스가 상속 받을 때, 클래스는 상속 받는다고 표현하지만, 부모 인터페이스의 기능을 자식이 상속 받을 때는 인터페이스를 구현한다고 표현한다. 이렇게 서로 다르게 표현하는 이유는 알아보자. 상속은 이름 그대로 부모의 기능을 물려 받는 것이 목적이다. 하지만 인터페이스는 모든 메서드가 추상 메서드이다. 따라서 물려받을 수 있는 기능이 없고, 오히려 인터페이스에 정의한 모든 메서드를 자식이 오버라이딩 해서 기능을 구현해야 한다. 따라서 구현한다고 표현한다.

인터페이스는 메서드 이름만 있는 설계도이고, 이 설계도가 실제 어떻게 작동하는지는 하위 클래스에서 모두 구현해야 한다. 따라서 인터페이스의 경우 상속이 아니라 해당 인터페이스를 구현한다고 표현한다.

상속과 구현은 사람이 표현하는 단어만 다를 뿐이지 자바 입장에서는 똑같다. 일반 상속 구조와 동일하게 작동한다.

인터페이스를 사용해야 하는 이유

모든 메서드가 추상 메서드인 경우 순수 추상 클래스를 만들어도 되고, 인터페이스를 만들어도 된다. 그런대 왜 인터페이스를 사용해야 할까? 단순히 편리하다는 이유를 넘어서 다음과 같은 이유가 있다.

- **제약**: 인터페이스를 만드는 이유는 인터페이스를 구현하는 곳에서 인터페이스의 메서드를 반드시 구현해라는 규약(제약)을 주는 것이다. USB 인터페이스를 생각해보자. USB 인터페이스에 맞추어 키보드, 마우스를 개발하고 연결해야 한다. 그렇지 않으면 작동하지 않는다. 인터페이스의 규약(제약)은 반드시 구현해야 하는 것이다. 그런데 순수 추상 클래스의 경우 미래에 누군가 그곳에 실행 가능한 메서드를 끼워 넣을 수 있다. 이렇게 되면 추가된 기능을 자식 클래스에서 구현하지 않을 수도 있고, 또 더는 순수 추상 클래스가 아니게 된다. 인터페이스는 모든 메서드가 추상 메서드이다. 따라서 이런 문제를 원천 차단할 수 있다.
- **다중 구현**: 자바에서 클래스 상속은 부모를 하나만 지정할 수 있다. 반면에 인터페이스는 부모를 여러명 두는 다중 구현(다중 상속)이 가능하다.

좋은 프로그램은 제약이 있는 프로그램이다.

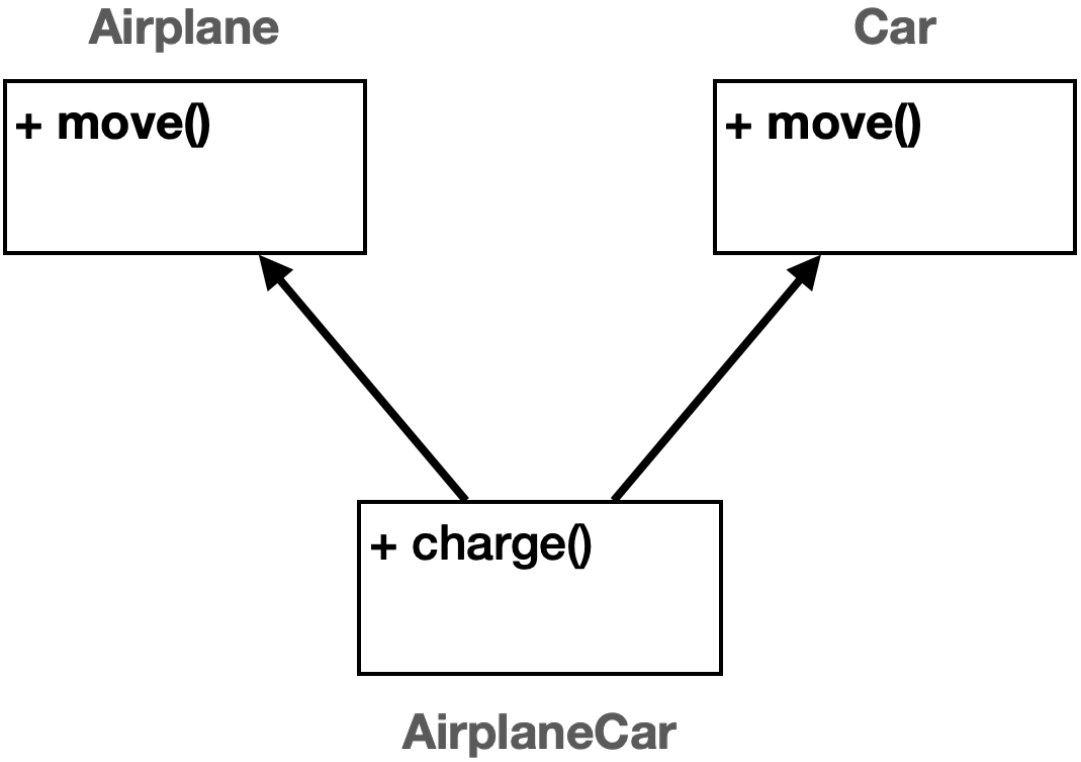
> **참고**
> 자바8에 등장한 `default` 메서드를 사용하면 인터페이스도 메서드를 구현할 수 있다. 하지만 이것은 예외적으로 아주 특별한 경우에만 사용해야 한다. 자바9에서 등장한 인터페이스의 `private` 메서드도 마찬가지이다. 지금 학습 단계에서는 이 부분들을 고려하지 않는 것이 좋다. 이 부분은 뒤에서 따로 다룬다.

인터페이스 - 다중 구현

자바가 다중 상속을 지원하지 않는 이유 - 복습

자바는 다중 상속을 지원하지 않는다. 그래서 `extends` 대상은 하나만 선택할 수 있다. 부모를 하나만 선택할 수 있다는 뜻이다. 물론 부모가 또 부모를 가지는 것은 괜찮다.

다중 상속 그림



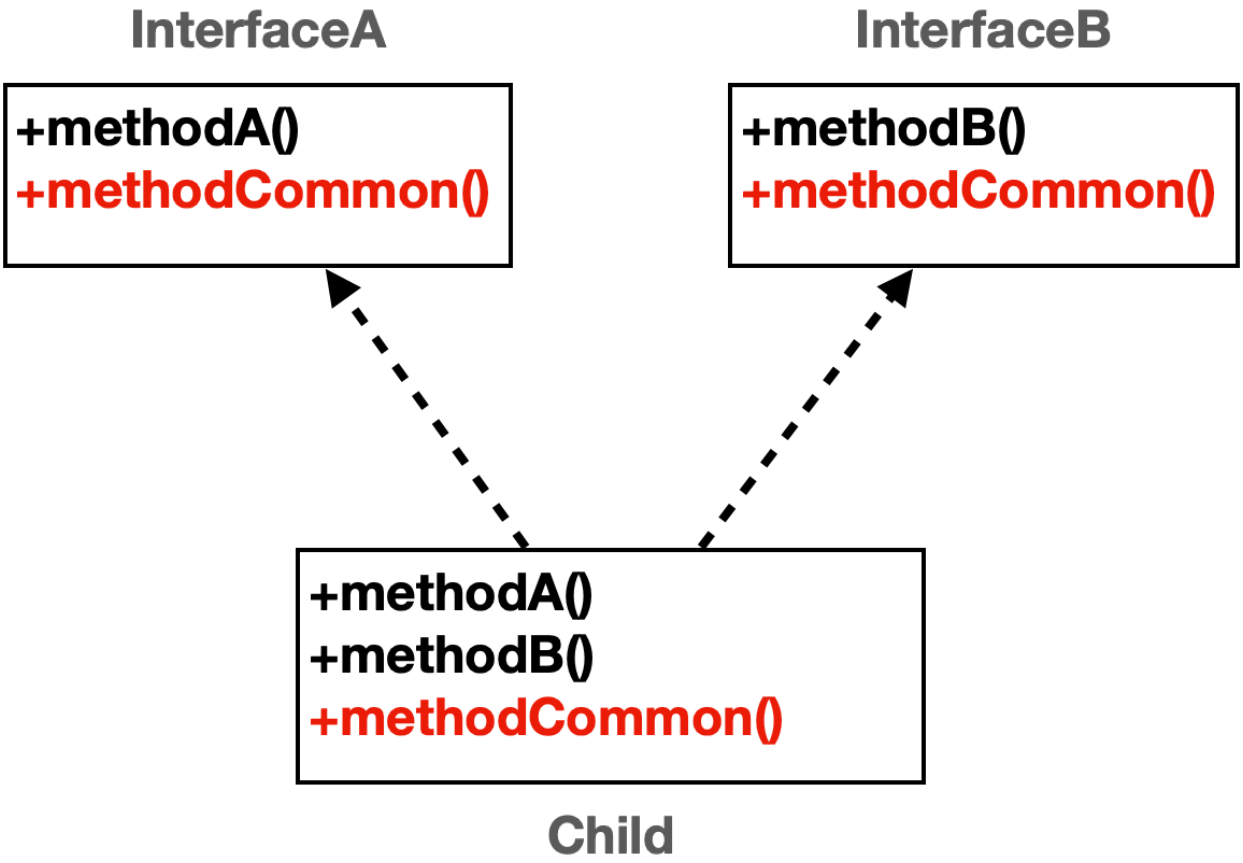
만약 비행기와 자동차를 상속 받아서 하늘을 나는 자동차를 만든다고 가정해보자. 만약 그림과 같이 다중 상속을 사용하게 되면 `AirplaneCar` 입장에서 `move()`를 호출할 때 어떤 부모의 `move()`를 사용해야 할지 애매한 문제가 발생한다. 이것을 다이아몬드 문제라 한다. 그리고 다중 상속을 사용하면 클래스 계층 구조가 매우 복잡해지 수 있다. 이런

문제점 때문에 자바는 클래스의 다중 상속을 허용하지 않는다. 대신에 인터페이스의 다중 구현을 허용하여 이러한 문제를 피한다.

클래스는 앞서 설명한 이유로 다중 상속이 안되는데, 인터페이스의 다중 구현은 허용한 이유는 뭘까?
인터페이스는 모두 추상 메서드로 이루어져 있기 때문이다.

다음 예제를 보자.

인터페이스 다중 구현 그림



`InterfaceA`, `InterfaceB`는 둘다 같은 `methodCommon()`을 가지고 있다. 그리고 `Child`는 두 인터페이스를 구현했다. 상속 관계의 경우 두 부모 중에 어떤 한 부모의 `methodCommon()`을 사용해야 할지 결정해야 하는 다이아몬드 문제가 발생한다.
하지만 인터페이스 자신은 구현을 가지지 않는다. 대신에 인터페이스를 구현하는 곳에서 해당 기능을 모두 구현해야 한다. 여기서 `InterfaceA`, `InterfaceB`는 같은 이름의 `methodCommon()`를 제공하지만 이것의 기능은 `Child`가 구현한다. 그리고 오버라이딩에 의해 어차피 `Child`에 있는 `methodCommon()`이 호출된다. 결과적으로 두 부모 중에 어떤 한 부모의 `methodCommon()`을 선택하는 것이 아니라 그냥 인터페이스들을 구현한 `Child`에 있는 `methodCommon()`이 사용된다. 이런 이유로 인터페이스는 다이아몬드 문제가 발생하지 않는다. 따라서 인터페이스의 경우 다중 구현을 허용한다.

예제를 코드로 작성해보자.

```java
package poly.diamond;

public interface InterfaceA {
    void methodA();
    void methodCommon();
}
```

```java
package poly.diamond;

public interface InterfaceB {
    void methodB();
    void methodCommon();
}
```

```java
package poly.diamond;

public class Child implements InterfaceA, InterfaceB {
    @Override
    public void methodA() {
        System.out.println("Child.methodA");
    }

    @Override
    public void methodB() {
        System.out.println("Child.methodB");
    }

    @Override
    public void methodCommon() {
        System.out.println("Child.methodCommon");
    }
}
```

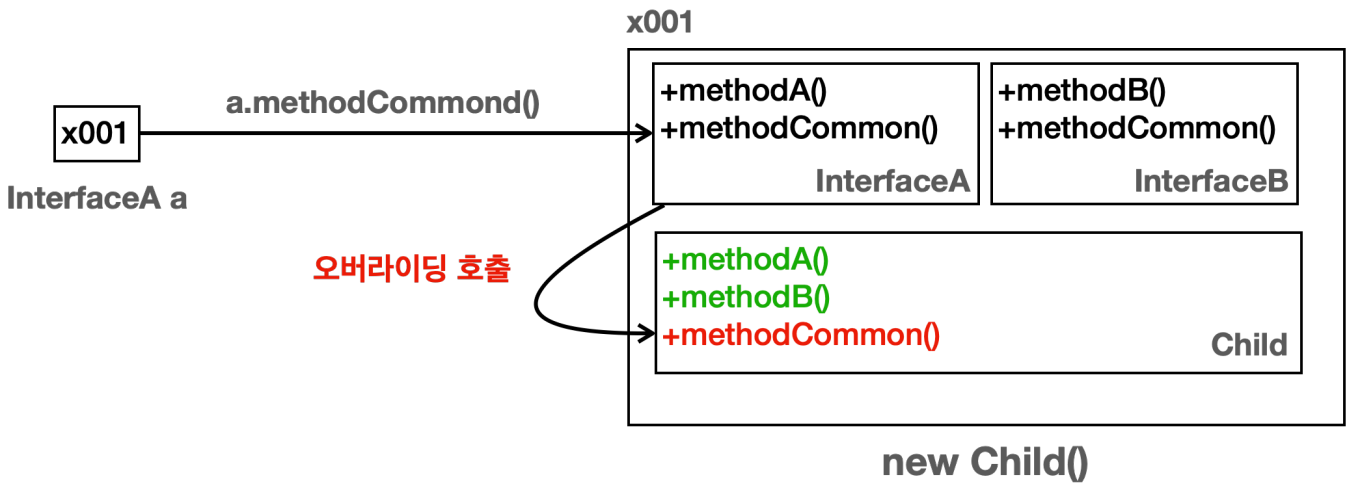
- `implements InterfaceA, InterfaceB`와 같이 다중 구현을 할 수 있다. `implements` 키워드 위에 `,`로 여러 인터페이스를 구분하면 된다.
- `methodCommon()`의 경우 양쪽 인터페이스에 다 있지만 같은 메서드이므로 구현은 하나만 하면 된다.

```java
package poly.diamond;
```

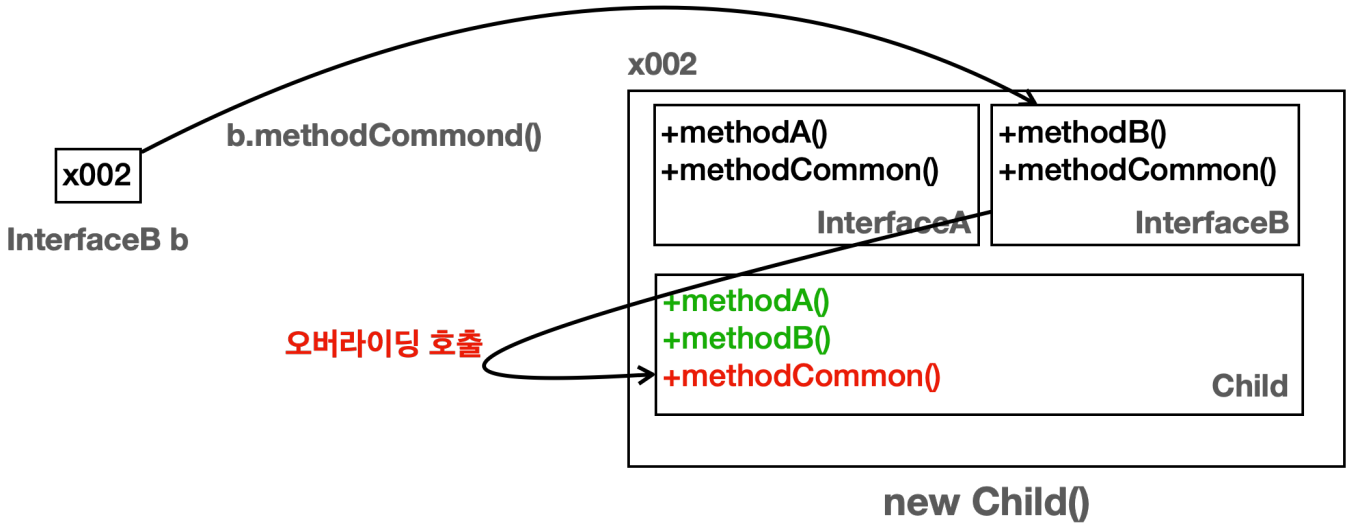
```java
//인터페이스 다중 구현
public class DiamondMain {
    public static void main(String[] args) {
        InterfaceA a = new Child();
        a.methodA();
        a.methodCommon();

        InterfaceB b = new Child();
        b.methodB();
        b.methodCommon();
    }
}
```

실행 결과

```
Child.methodA
Child.methodCommon
Child.methodB
Child.methodCommon
```



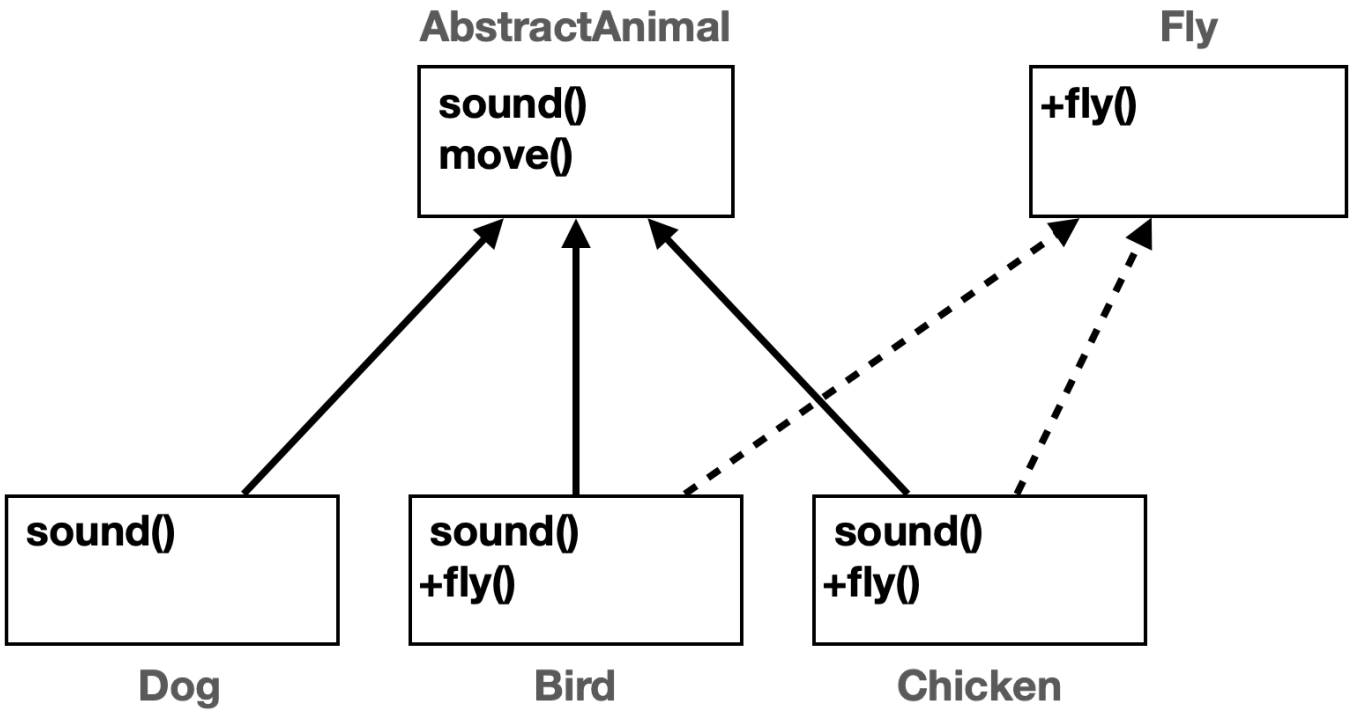
1. `a.methodCommon()`을 호출하면 먼저 `x001` `Child` 인스턴스를 찾는다.
2. 변수 `a`가 `InterfaceA` 타입이므로 해당 타입에서 `methodCommon()`을 찾는다.
3. `methodCommon()`은 하위 타입인 `Child`에서 오버라이딩 되어 있다. 따라서 `Child`의 `methodCommon()`이 호출된다.



4. `b.methodCommon()`을 호출하면 먼저 `x002` `Child` 인스턴스를 찾는다.
5. 변수 `b`가 `InterfaceB` 타입이므로 해당 타입에서 `methodCommon()`을 찾는다.
6. `methodCommon()`은 하위 타입인 `Child`에서 오버라이딩 되어 있다. 따라서 `Child`의 `methodCommon()`이 호출된다.

클래스와 인터페이스 활용

이번에는 클래스 상속과 인터페이스 구현을 함께 사용하는 예를 알아보자.



- `AbstractAnimal`은 추상 클래스다.

    - `sound()` : 동물의 소리를 내기 위한 `sound()` 추상 메서드를 제공한다.
    - `move()` : 동물의 이동을 표현하기 위한 메서드이다. 이 메서드는 추상 메서드가 아니다. 상속을 목적으로 사용된다.
- `Fly`는 인터페이스이다. 나는 동물은 이 인터페이스를 구현할 수 있다.
    - `Bird`, `Chicken`은 날 수 있는 동물이다. `fly()` 메서드를 구현해야 한다.

예제6

```
package poly.ex6;

public abstract class AbstractAnimal {
    public abstract void sound();
    public void move() {
        System.out.println("동물이 이동합니다.");
    }
}
```

```
package poly.ex6;

public interface Fly {
    void fly();
}
```

```
package poly.ex6;

public class Dog extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
}
```

`Dog`는 `AbstractAnimal`만 상속 받는다.

```
package poly.ex6;

public class Bird extends AbstractAnimal implements Fly {
    @Override
    public void sound() {
        System.out.println("짹짹");
```

```
    }

    @Override
    public void fly() {
        System.out.println("새 날기");
    }
}
```

`Bird`는 `AbstractAnimal` 클래스를 상속하고 `Fly` 인터페이스를 구현한다.

하나의 클래스 여러 인터페이스 예시

```
public class Bird extends AbstractAnimal implements Fly, Swim {
```

`extends`를 통한 상속은 하나만 할 수 있고 `implements`를 통한 인터페이스는 다중 구현 할 수 있기 때문에 둘이 함께 나온 경우 `extends`가 먼저 나와야 한다.

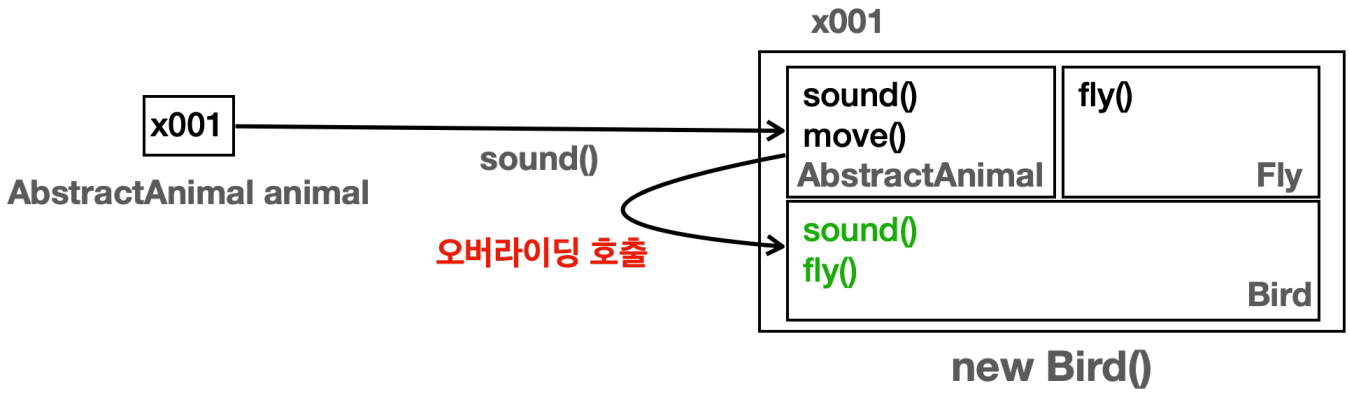
```
package poly.ex6;

public class Chicken extends AbstractAnimal implements Fly {
    @Override
    public void sound() {
        System.out.println("꼬끼오");
    }

    @Override
    public void fly() {
        System.out.println("닭 날기");
    }
}
```

```
package poly.ex6;

public class SoundFlyMain {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Bird bird = new Bird();
        Chicken chicken = new Chicken();

        soundAnimal(dog);
        soundAnimal(bird);
        soundAnimal(chicken);
```

```
        flyAnimal(bird);
        flyAnimal(chicken);
    }

    //AbstractAnimal 사용 가능
    private static void soundAnimal(AbstractAnimal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }

    //Fly 인터페이스가 있으면 사용 가능
    private static void flyAnimal(Fly fly) {
        System.out.println("날기 테스트 시작");
        fly.fly();
        System.out.println("날기 테스트 종료");
    }

}
```

실행 결과

```
동물 소리 테스트 시작
멍멍
동물 소리 테스트 종료

동물 소리 테스트 시작
짹짹
동물 소리 테스트 종료

동물 소리 테스트 시작
꼬끼오
동물 소리 테스트 종료

날기 테스트 시작
새 날기
날기 테스트 종료

날기 테스트 시작
닭 날기
날기 테스트 종료
```

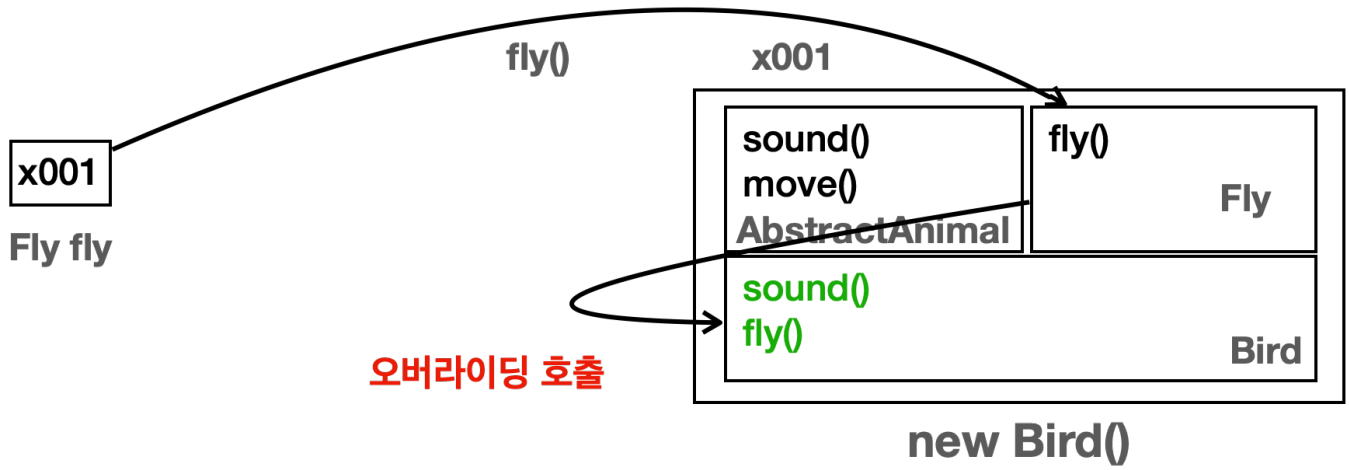


`soundAnimal(AbstractAnimal animal)`

`AbstractAnimal`를 상속한 `Dog`, `Bird`, `Chicken`을 전달해서 실행할 수 있다.

실행 과정

- `soundAnimal(bird)`를 호출한다고 가정하자.
- 메서드 안에서 `animal.sound()`를 호출하면 참조 대상인 `x001` `Bird` 인스턴스를 찾는다.
- 호출한 `animal` 변수는 `AbstractAnimal` 타입이다. 따라서 `AbstractAnimal.sound()`를 찾는다. 해당 메서드는 `Bird.sound()`에 오버라이딩 되어 있다.
- `Bird.sound()`가 호출된다.



`flyAnimal(Fly fly)`

`Fly` 인터페이스를 구현한 `Bird`, `Chicken`을 전달해서 실행할 수 있다.

실행 과정

- `fly(bird)`를 호출한다고 가정하자.
- 메서드 안에서 `fly.fly()`를 호출하면 참조 대상인 `x001` `Bird` 인스턴스를 찾는다.
- 호출한 `fly` 변수는 `Fly` 타입이다. 따라서 `Fly.fly()`를 찾는다. 해당 메서드는 `Bird.fly()`에 오버라이딩 되어 있다.

- `Bird.fly()`가 호출된다.

정리