# 7장. 상속

# Contents


- 1절. 상속 개념
- 2절. 클래스 상속(extends)
- 3절. 부모 생성자 호출(super(…))
- 4절. 메소드 재정의(Override)
- 5절. final 클래스와 final 메소드
- 6절. protected 접근 제한자
- 7절. 타입변환과 다형성(polymorphism)
- 8절. 추상 클래스(Abstract Class)


# 상속성(Inheritance)의 개요(1)

- 클래스 및 클래스 계통 개념
  - JAVA는 사용자 정의 타입(User Defined Type)을 만들 수 있는 클래스 개념 및 클래스 계통 개념을 가지고 있음.
  - 프로그래머가 구상한 개념과 응용 프로그램에서 쓰일 개념을 모형화 하는데 클래스(class)를 사용함.
- JAVA 에서 제공하는 상속성의 개념은 클래스들 사이의 계통관계를 표현하기 위한 장치
  - 계통관계는 곧 클래스들 사이의 공통 특성임.
    - (예) 원과 삼각형 사이에는 바로 "도형이다"라는 공통점이 존재.
      - 도형이란 개념을 공유하고 있다.
      - 원과 삼각형 개념을 프로그램에 넣는다고 할 때 이 '도형'이란 개념을 함께 포함시켜야만 제대로 표현이 가능하다.
- 상속성은 클래스들 사이의 공통점에 의한 관계를 표현하는 방법이다.

# 상속성의 개요(2)

- 상속성(Inheritance)
  - 하나의 객체가 다른 객체의 특성을 이어받을 수 있게 해 주는 과정
  - 자식 클래스는 부모클래스의 모든 특성을 상속 받고, 자기 자신의 특성을 추가시켜 정의 할 수 있다.

➢ 상위 클래스에 있는 것은 상속 받아쓰고 상위클래스에 없는 것은 새로 만들자

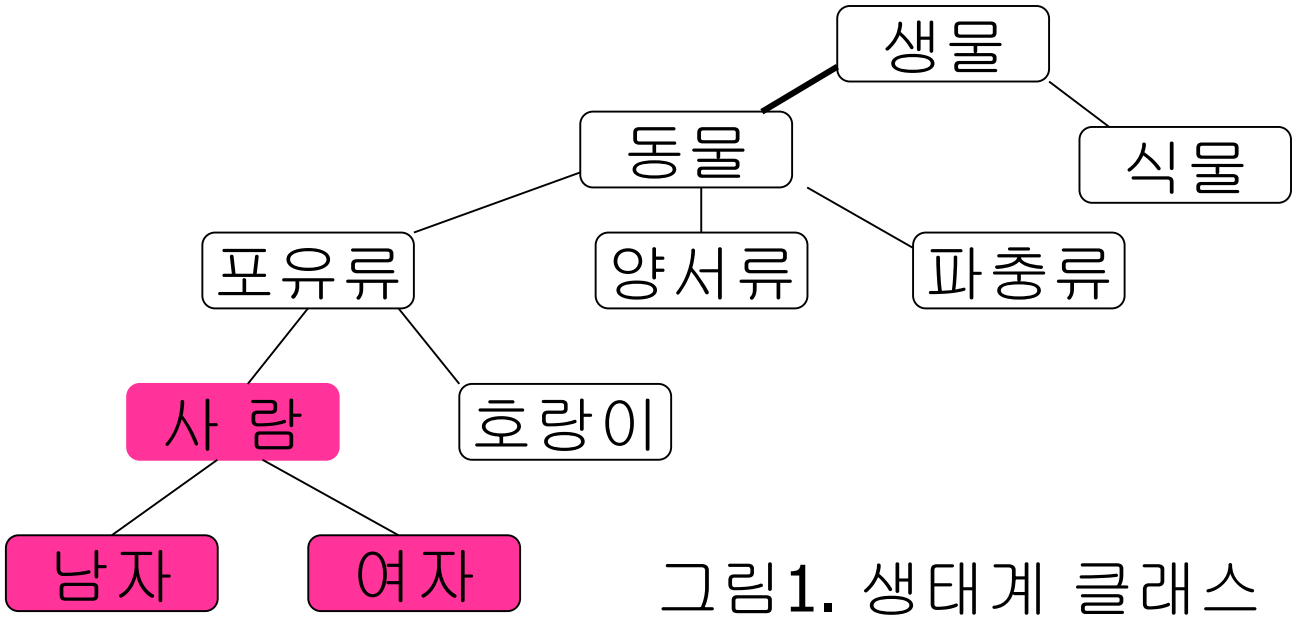



그림1. 생태계 클래스

# 상속성의 개요(3)

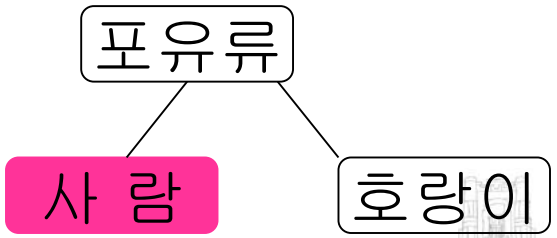
## ❑ 상속성

- 하나의 클래스가 다른 클래스의 특성을 이어받는 것

## ❑ 용어

- 기본(부모) 클래스(Base Class)
  - 일반적인 모든 성질을 정의한다
- 파생(자식) 클래스(Derived Class)
  - 기본 클래스의 특성을 이어 받는다
  - 이 클래스에서 지정하고자 하는 특성을 추가한다

❖ 코드의 중복 작업 최소화
❖ 소프트웨어의 재사용성 증대



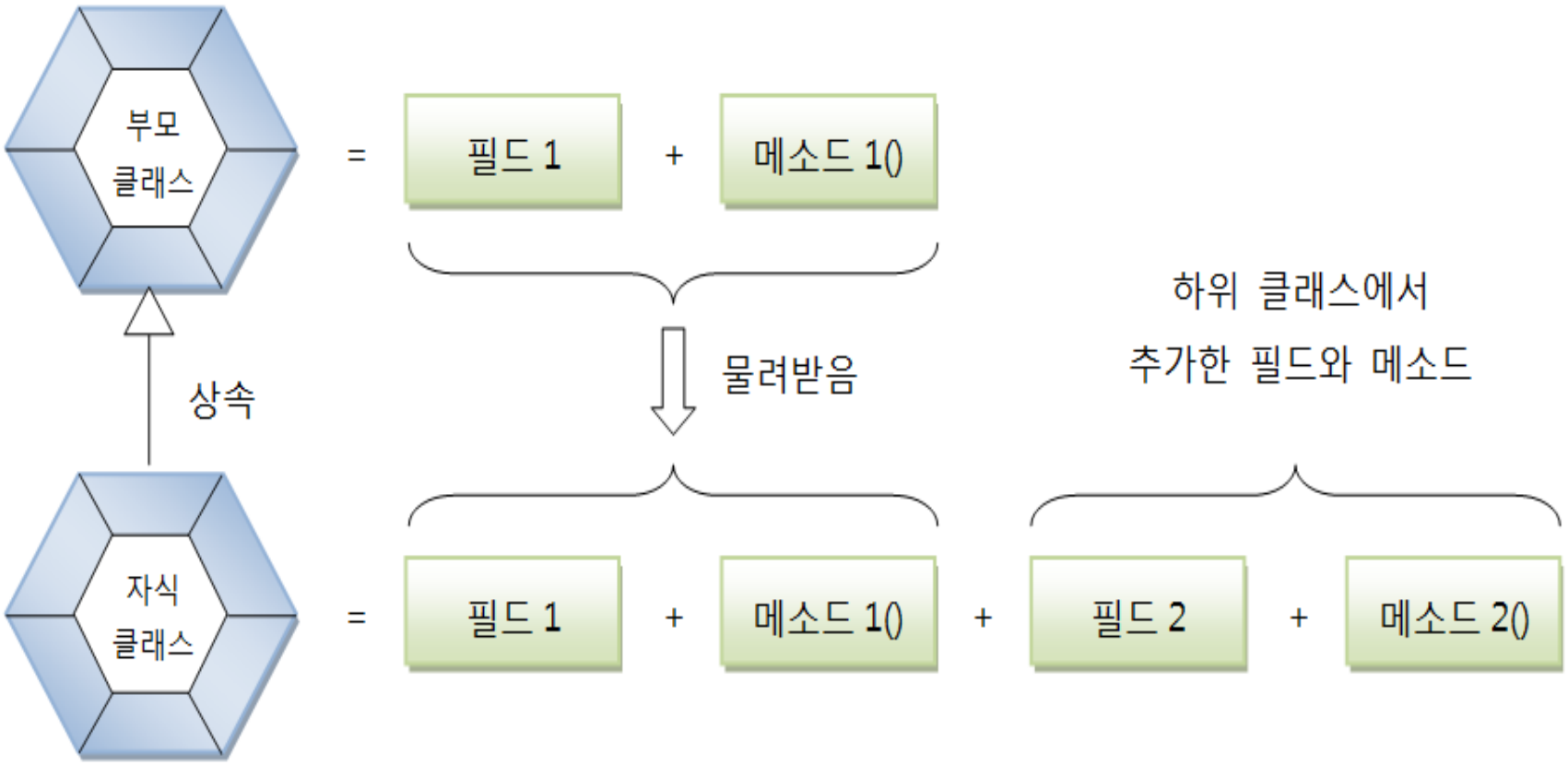
# 1절. 상속 개념

## ❖ 상속(Inheritance)이란?

- 현실 세계:
  - 부모가 자식에게 물려주는 행위
  - 부모가 자식을 선택해서 물려줌
- 객체 지향 프로그램:
  - 자식(하위, 파생) 클래스가 부모(상위) 클래스의 멤버를 물려받는 것
  - 자식이 부모를 선택해 물려받음
  - 상속 대상: 부모의 필드와 메소드

# 1절. 상속 개념

## ❖ 상속(Inheritance)이란?



# 1절. 상속 개념

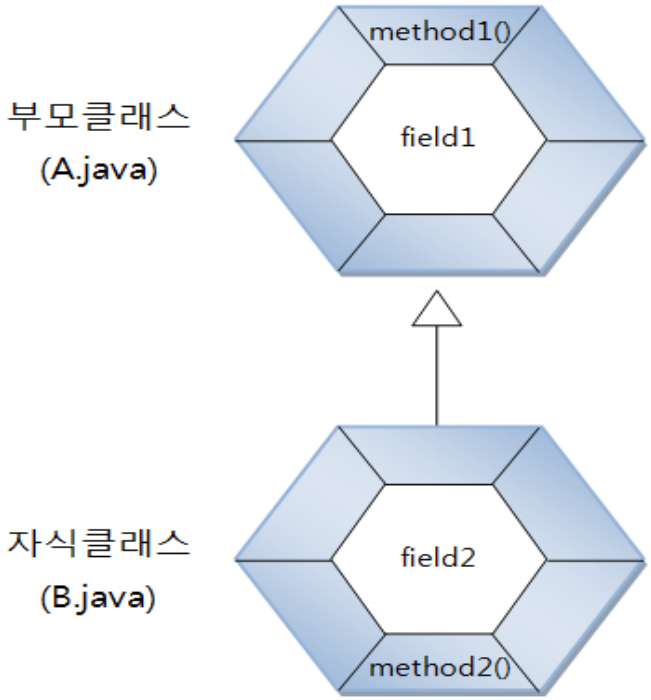
## ❖ 상속(Inheritance) 개념의 활용

- 상속의 효과
  - 부모 클래스 재사용해 자식 클래스 빨리 개발 가능
  - 반복된 코드 중복 줄임
  - 유지 보수 편리성 제공
  - 객체 다형성 구현 가능
- 상속 대상 제한
  - 부모 클래스의 private 접근 권한을 갖는 필드와 메소드 제외
  - 부모 클래스가 다른 패키지에 있을 경우, default 접근 갖는 필드와 메소드도 제외

# 2절. 클래스 상속(extends)

## ❖ extends 키워드

- 자식 클래스가 상속할 부모 클래스를 지정하는 키워드




```
public class A {
    int field1;
    void method1() { ... }
}
```

A 를 상속

```
public class B extends A {
    String field2;
    void method2() { ... }
}
```

- 자바는 단일 상속 – 부모 클래스 나열 불가

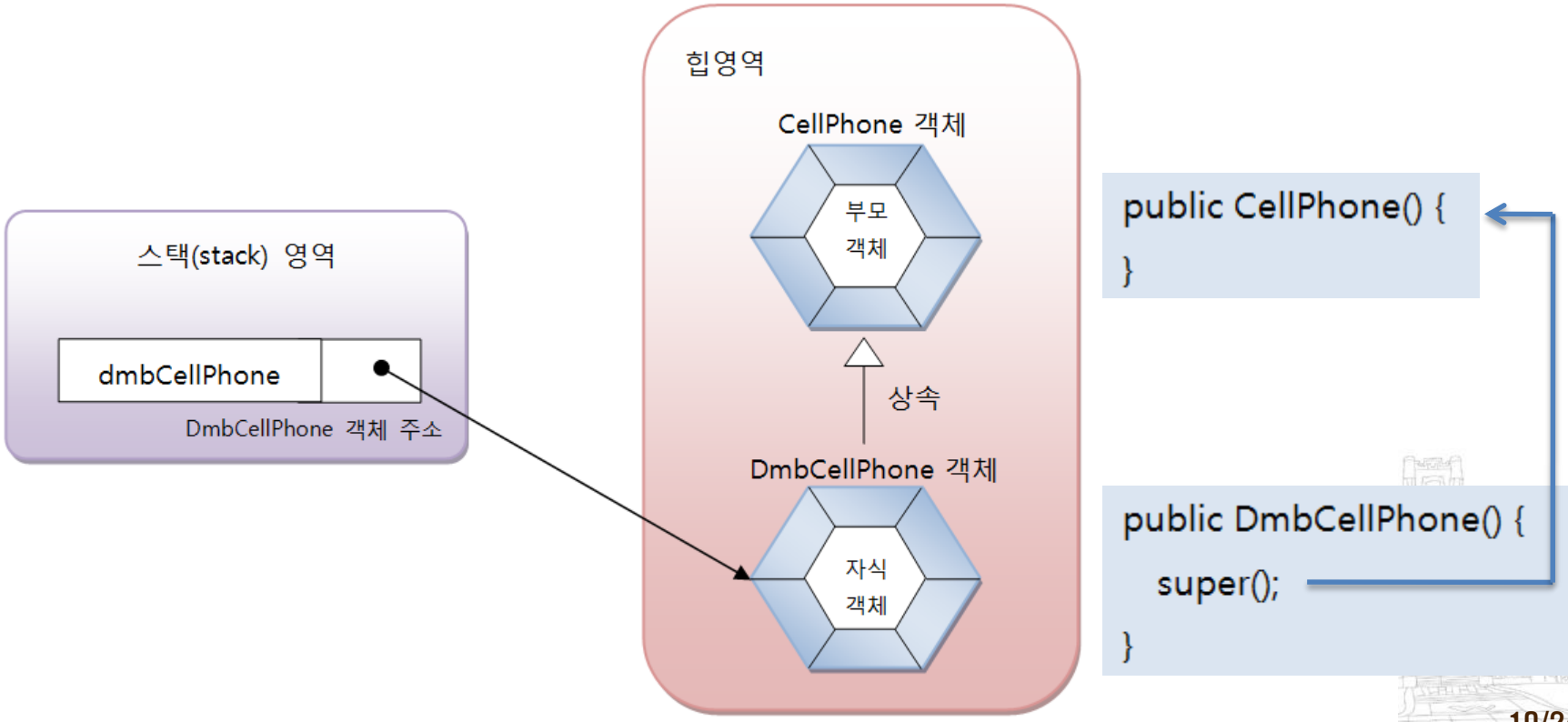
```
class 자식클래스 extends 부모클래스1, 부모클래스2 {
}
```

# 3절. 부모 생성자 호출(super(…))

## ❖ 자식 객체 생성하면 부모 객체도 생성되는가?

- 부모 없는 자식 없음
  - 자식 객체 생성할 때는 부모 객체부터 생성 후 자식 객체 생성
  - 부모 생성자 호출 완료 후 자식 생성자 호출 완료

```
DmbCellPhone dmbCellPhone = new DmbCellPhone();
```



# 3절. 부모 생성자 호출(super(…))

## ❖ 명시적인 부모 생성자 호출

- 부모 객체 생성할 때, 부모 생성자 선택해 호출

```
자식클래스( 매개변수선언, ... ) {
   super( 매개값, ... );
   ...
}
```

  - **super(매개값,…)**
    - 매개값과 동일한 타입, 개수, 순서 맞는 부모 생성자 호출
  - 부모 생성자 없다면 컴파일 오류 발생
  - 반드시 자식 생성자의 첫 줄에 위치
  - 부모 클래스에 기본(매개변수 없는) 생성자가 없다면 필수 작성

# 4절. 메소드 재정의(Override)

- 메소드 재정의(@Override)
  - 부모 클래스의 상속 메소드 수정해 자식 클래스에서 재정의 하는 것
  - 메소드 재정의 조건 (p.295~296)
    - 부모 클래스의 메소드와 동일한 시그너처 가져야
    - 접근 제한을 더 강하게 오버라이딩 불가
      - public을 default나 private으로 수정 불가
      - 반대로 default는 public 으로 수정 가능
    - 새로운 예외(Exception) throws 불가 (예외처리는 10장 참조)

# 4절. 메소드 재정의(Override)

- @Override 어노테이션
  - 컴파일러에게 부모 클래스의 메소드 선언부와 동일한지 검사 지시
  - 정확한 메소드 재정의 위해 붙여주면 OK

- 메소드 재정의 효과
  - 부모 메소드는 숨겨지는 효과 발생
    - 재정의된 자식 메소드 실행

# 4절. 메소드 재정의(Override)

## ❖ 부모 메소드 사용(super)

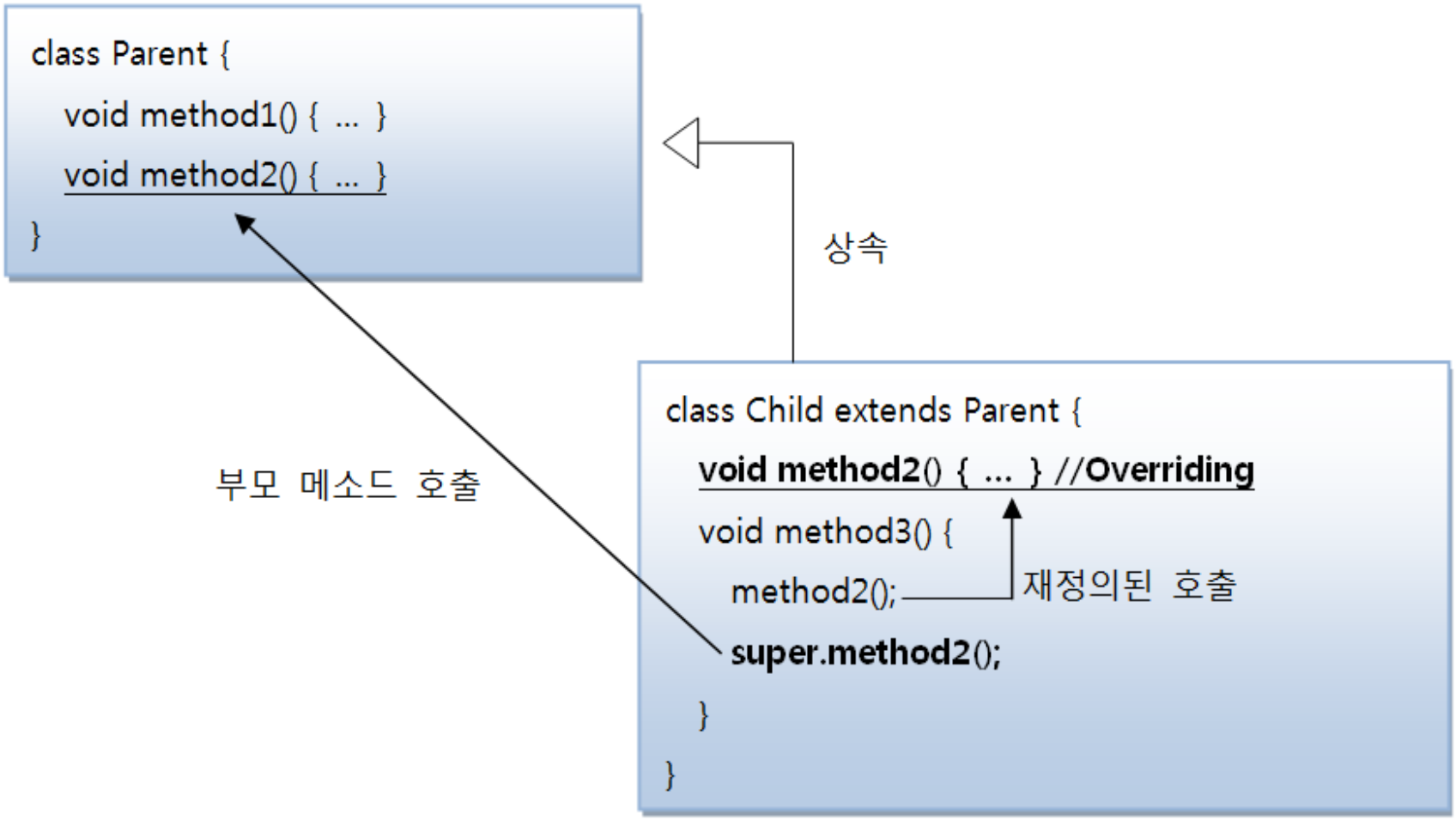
- 메소드 재정의는 부모 메소드 숨기는 효과 !!
  - 자식 클래스에서는 재정의된 메소드만 호출
- 자식 클래스에서 수정되기 전 부모 메소드 호출 – super 사용
  - super는 부모 객체 참조(참고: this는 자신 객체 참조)

```
super.부모메소드();
```

# 4절. 메소드 재정의(Override)

## ❖ 부모 메소드 사용(super)

super.부모메소드();



# 5절. final 클래스와 final 메소드

## ❖ final 키워드의 용도

- final 필드: 수정 불가 필드
- final 클래스: 부모로 사용 불가한 클래스
- final 메소드: 자식이 재정의할 수 없는 메소드

## ❖ 상속할 수 없는 final 클래스

- 자식 클래스 만들지 못하도록 final 클래스로 생성

```
public final class 클래스 { ... }
```

```
public final class String { .. }
```

```
public class NewString extends String { ... }
```

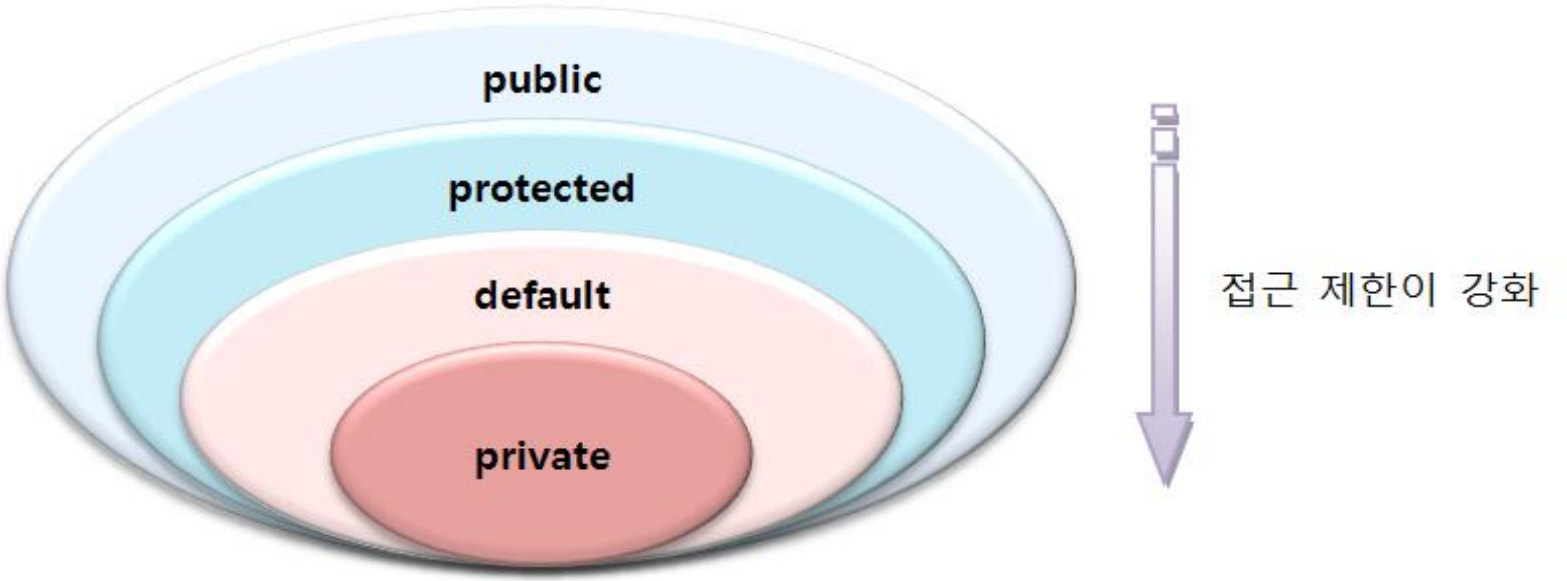
## ❖ 오버라이딩 불가한 final 메소드

- 자식 클래스가 재정의 못하도록 부모 클래스의 메소드를 final 로 생성

# 6절. protected 접근 제한자

## ❖ protected 접근 제한자

- 상속과 관련된 접근 제한자
  - 같은 패키지: default와 동일
  - 다른 패키지: 자식 클래스만 접근 허용 (p.303~305)



# 6절. protected 접근 제한자

## ❖ protected 접근 제한자

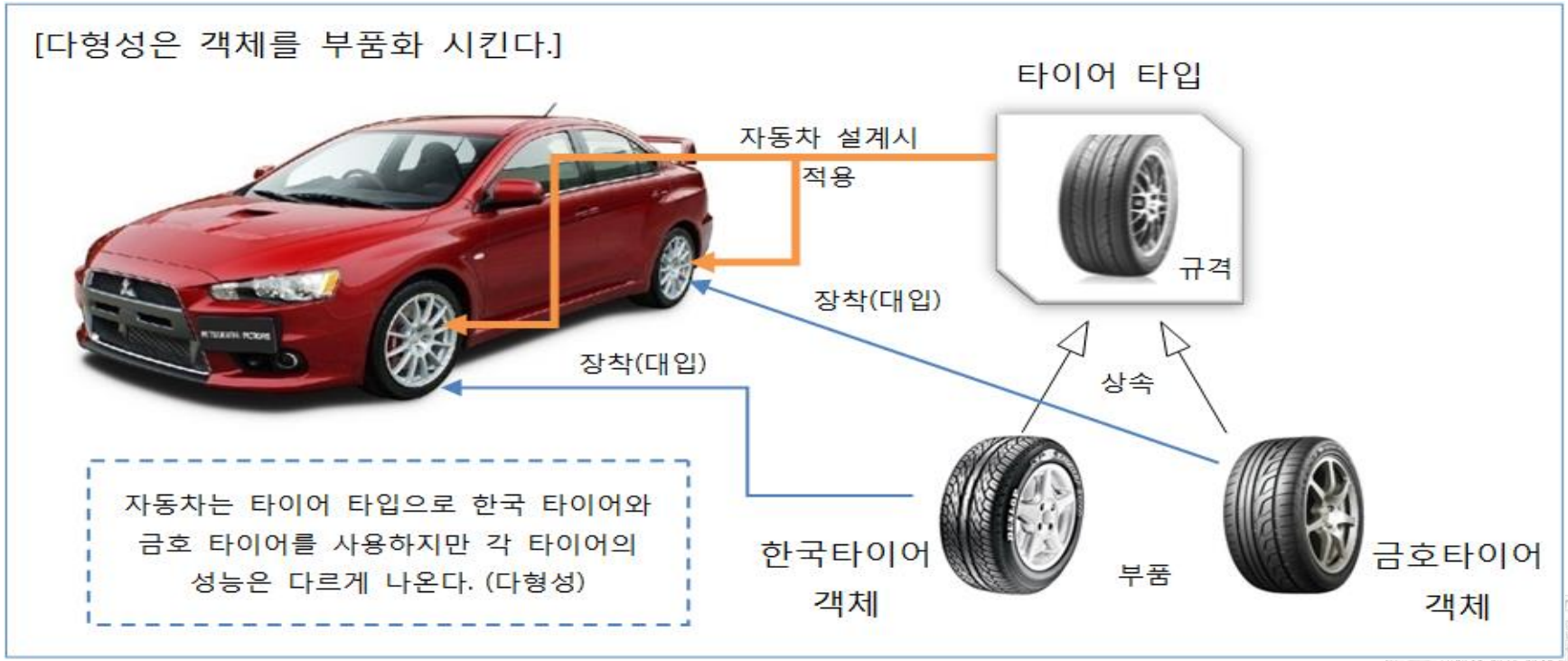
- 상속과 관련된 접근 제한자

| 접근 제한 | 적용할 내용 | 접근할 수 없는 클래스 |
|---|---|---|
| public | 클래스, 필드, 생성자, 메소드 | 없음 |
| protected | 필드, 생성자, 메소드 | 자식 클래스가 아닌 다른 패키지에 소속된 클래스 |
| default | 클래스, 필드, 생성자, 메소드 | 다른 패키지에 소속된 클래스 |
| private | 필드, 생성자, 메소드 | 모든 외부 클래스 |

# 7절. 타입변환과 다형성(polymorphism)

## ❖ 다형성 (多形性, Polymorphism)

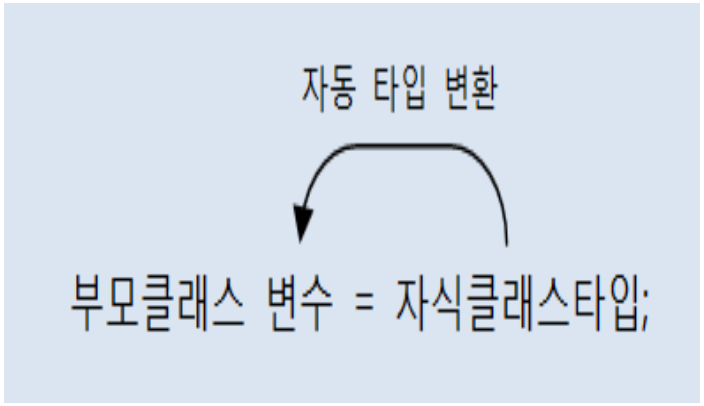
- 같은 타입이지만 실행 결과가 다양한 객체 대입(이용) 가능한 성질
  - 부모 타입에는 모든 자식 객체가 대입 가능
    - 자식 타입은 부모 타입으로 자동 타입 변환
  - 효과: 객체 부품화 가능

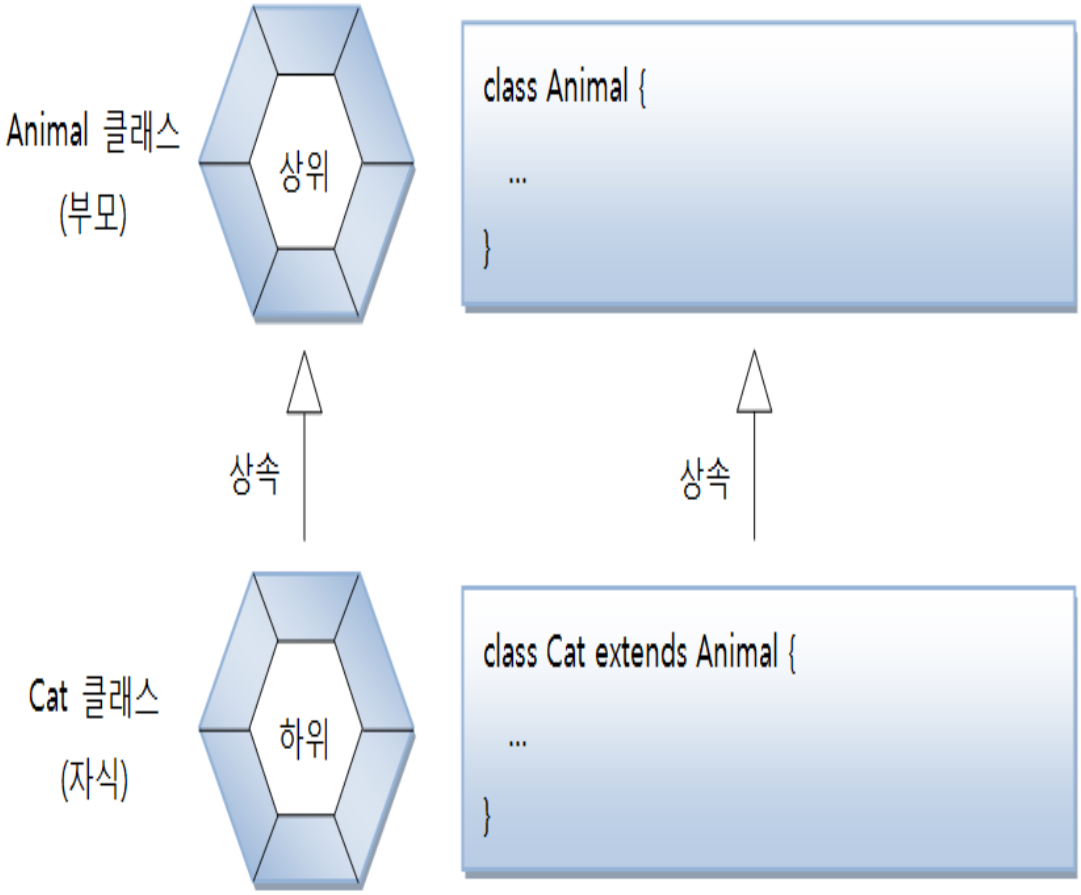


# 7절. 타입변환과 다형성(polymorphism)

## ❖ 자동 타입 변환(Promotion)

- 프로그램 실행 도중에 자동 타입 변환이 일어나는 것



# 7절. 타입변환과 다형성(polymorphism)

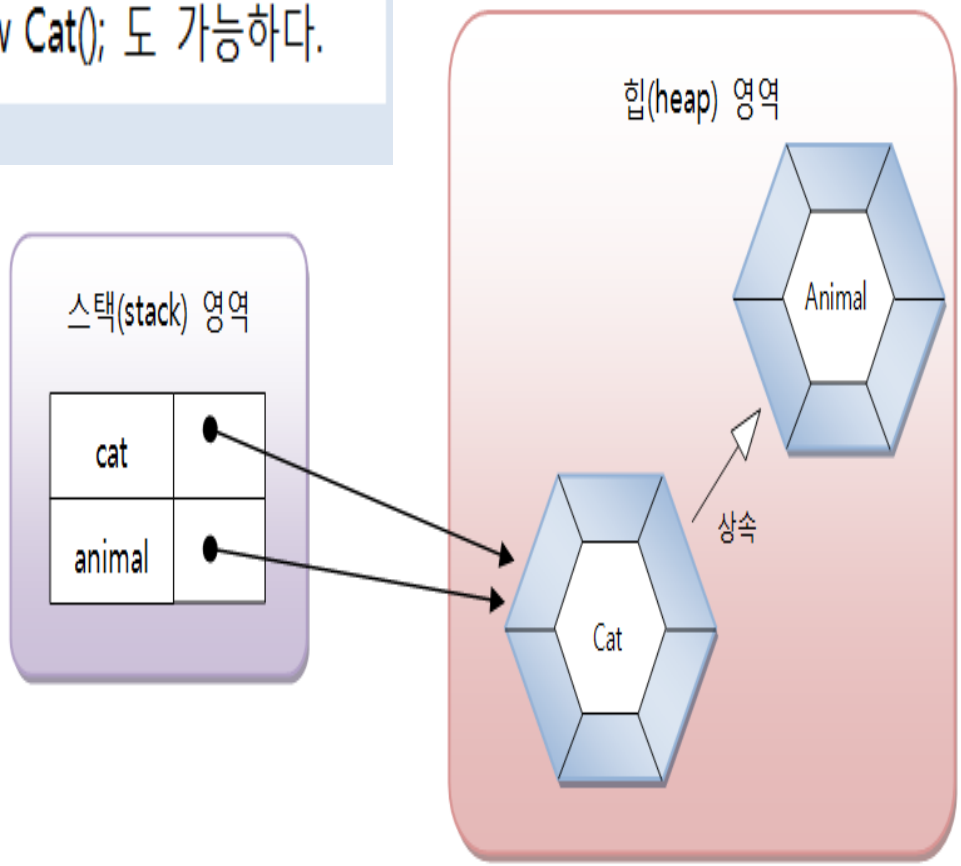
## ❖ 자동 타입 변환(Promotion)

- 프로그램 실행 도중에 자동 타입 변환이 일어나는 것

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

} Animal animal = new Cat(); 도 가능하다.

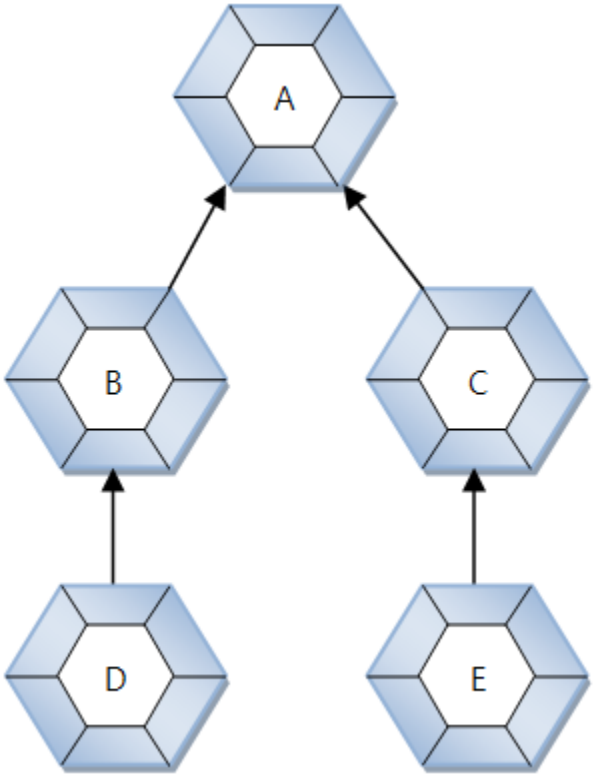
```
cat == animal    //true
```



# 7절. 타입변환과 다형성(polymorphism)

## ❖ 자동 타입 변환(Promotion)

- 바로 위의 부모가 아니더라도 상속 계층의 상위면 자동 타입 변환 가능
  - 변환 후에는 부모 클래스 멤버만 접근 가능 (p.308~310)



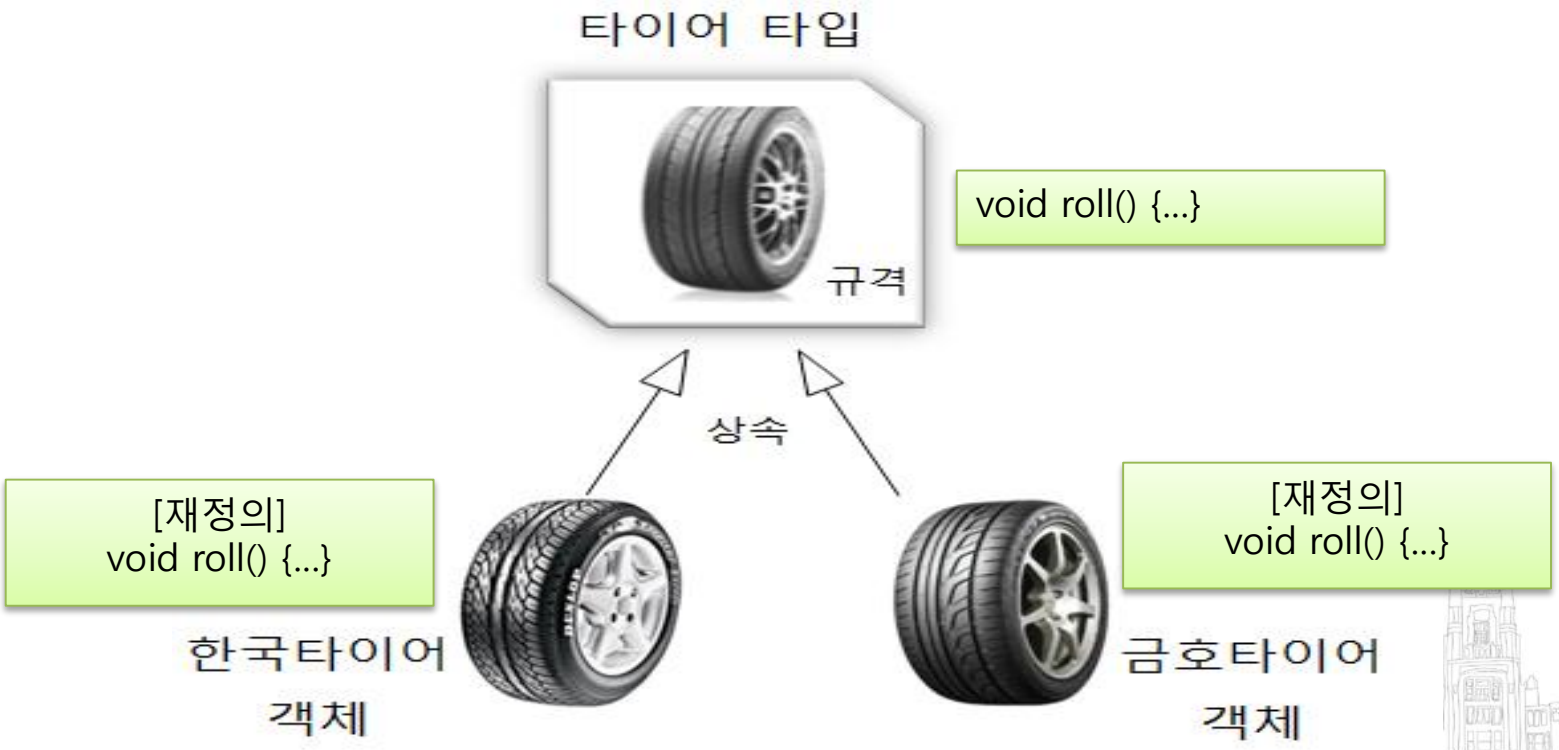

```
B b = new B();
C c = new C();
D d = new D();
E e = new E();
```

```
A a1 = b;   (가능)
A a2 = c;   (가능)
A a3 = d;   (가능)
A a4 = e;   (가능)

B b1 = d;   (가능)
C c1 = e;   (가능)

B b3 = e;   (불가능)
C c2 = d;   (불가능)
```

# 7절. 타입변환과 다형성(polymorphism)

## ❖ 필드의 다형성 (p.311~317)

- 다형성을 구현하는 기술적 방법
  - 부모 타입으로 자동 변환
  - 재정의된 메소드(오버라이딩)



# 7절. 타입변환과 다형성(polymorphism)

## ❖ 하나의 배열로 객체 관리 (p.318~320)

```
class Car {
    Tire frontLeftTire = new Tire("앞왼쪽", 6);
    Tire frontRightTire = new Tire("앞오른쪽", 2);
    Tire backLeftTire = new Tire("뒤왼쪽", 3);
    Tire backRightTire = new Tire("뒤오른쪽", 4);
}
```

```
class Car {
    Tire[] tires = {
        new Tire("앞왼쪽", 6),
        new Tire("앞오른쪽", 2),
        new Tire("뒤왼쪽", 3),
        new Tire("뒤오른쪽", 4)
    };
}
```

```
tires[1] = new KumhoTire("앞오른쪽", 13);
```

```
int run() {
        System.out.println("[자동차가 달립니다.]");
        for(int i=0; i<tires.length; i++) {
            if(tires[i].roll()==false) {
                stop();
                return (i+1);
            }
        }
        return 0;
}
```

# 7절. 타입변환과 다형성(polymorphism)

## ❖ 매개변수의 다형성 (p.321~323)

- 매개변수가 클래스 타입일 경우
  - 해당 클래스의 객체 대입이 원칙이나 자식 객체 대입하는 것도 허용
    - 자동 타입 변환
    - 매개변수의 다형성

# 7절. 타입변환과 다형성(polymorphism)

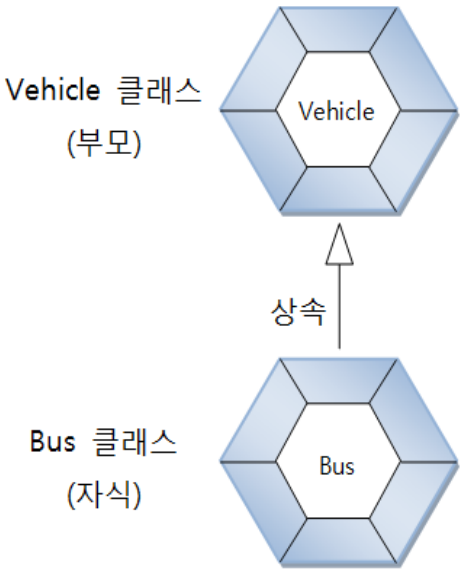
## ❖ 매개변수의 다형성 (p.321~323)

### ▪ 매개변수가 클래스 타입일 경우

```
class Driver {
    void drive(Vehicle vehicle) {
        vehicle.run();
    }
}
```

```
Driver driver = new Dirver();

Bus bus = new Bus();

driver.drive( bus );
```

자동 타입 변환 발생
Vehicle vehicle = bus;




```
void drive(Vehicle vehicle) {
    vehicle.run();
}
```

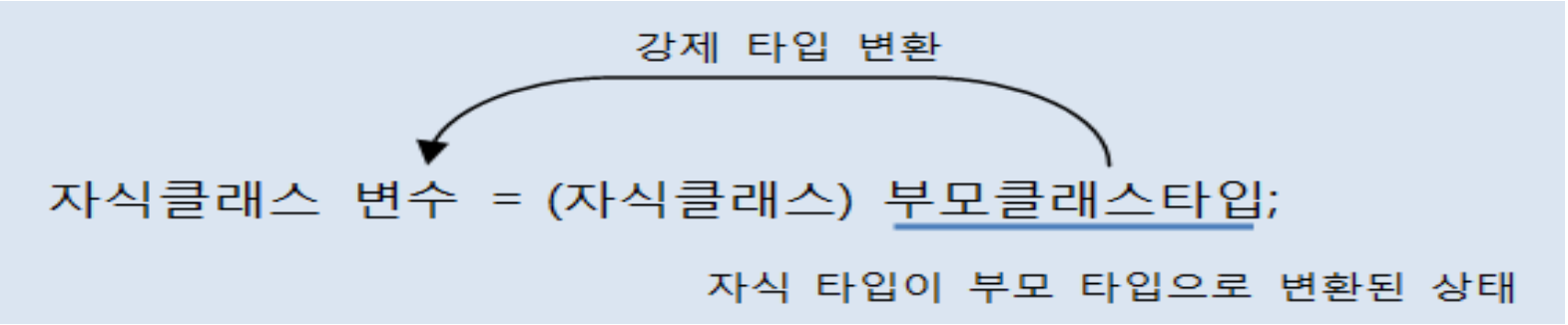
자식 객체

자식 객체가 재정의한 run() 메소드 실행

# 7절. 타입변환과 다형성(polymorphism)

## ❖ 강제 타입 변환(Casting) (p.324~325)

- 부모 타입을 자식 타입으로 변환하는 것




- 조건
  - 자식 타입을 부모 타입으로 자동 변환 후, 다시 자식 타입으로 변환할 때
- 강제 타입 변환이 필요한 경우
  - 자식 타입이 부모 타입으로 자동 변환
    - 부모 타입에 선언된 필드와 메소드만 사용 가능
  - 자식 타입에 선언된 필드와 메소드를 다시 사용해야 할 경우

# 7절. 타입변환과 다형성(polymorphism)

## ❖ 객체 타입 확인(instanceof) (p.326~329)

- 부모 타입이면 모두 자식 타입으로 강제 타입 변환할 수 있는 것 아님
  - ClassCastException 예외 발생 가능

```
Parent parent = new Parent();
Child child = (Child) parent;      //강제 타입 변환을 할 수 없다.
```

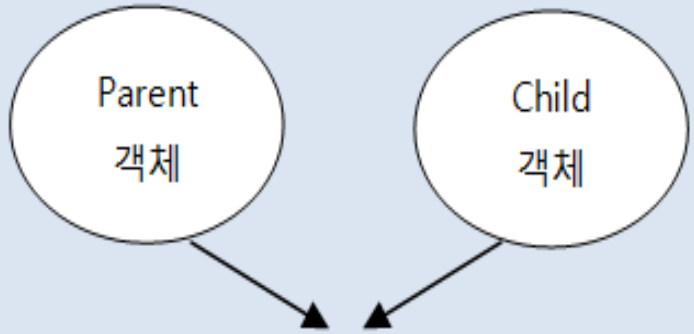
- 먼저 자식 타입인지 확인 후 강제 타입 실행해야 함

# 7절. 타입변환과 다형성(polymorphism)

## ❖ 객체 타입 확인(instanceof) (p.326~329)

- 먼저 자식 타입인지 확인 후 강제 타입 실행해야 함

```
boolean result = 좌항(객체) instanceof 우항(타입)
```




```
public void method(Parent parent) {
    if(parent instanceof Child) {
        Child child = (Child) parent;
    }
}
```

parnet 매개변수가 참조하는 객체가 Child 인지 조사

# 8절. 추상 클래스(Abstract Class)

## 추상 클래스 개념

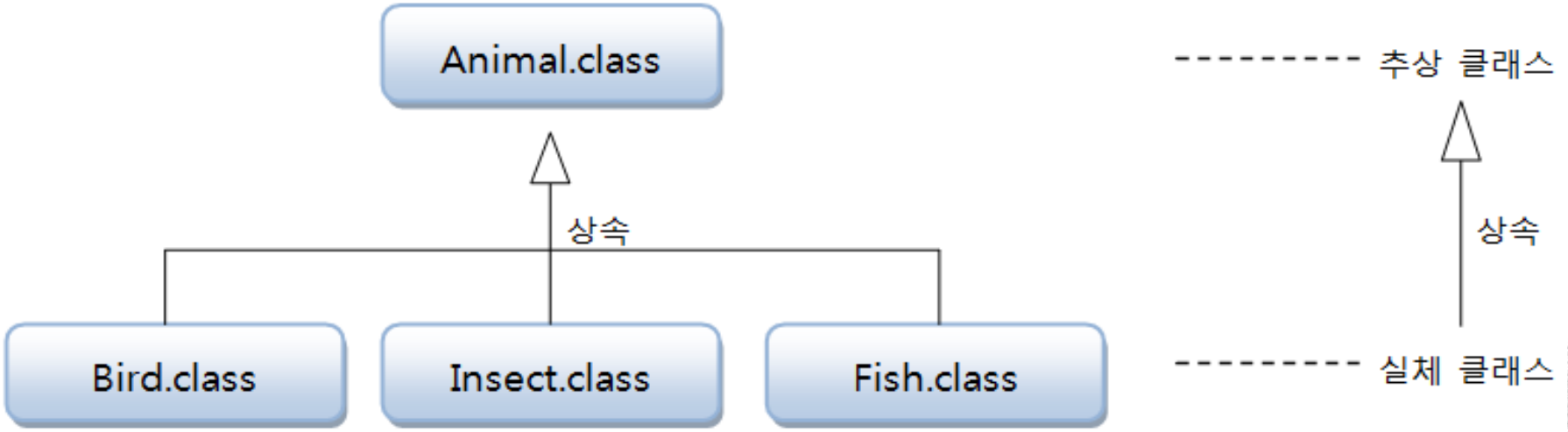
- 추상(abstract)
  - 실체들 간에 공통되는 특성을 추출한 것
    - 예1: 새, 곤충, 물고기→ 동물 (추상)
    - 예2: 삼성, 현대, LG → 회사 (추상)

# 8절. 추상 클래스(Abstract Class)

## ❖ 추상 클래스 개념

- 추상 클래스(abstract class)
  - 실체 클래스들의 공통되는 필드와 메소드를 정의한 클래스
  - 추상 클래스는 실체 클래스의 부모 클래스 역할
    - 추상 클래스는 단독으로 객체 생성(인스턴스화)이 불가능함.
    - 추상클래스로부터 상속받은 자식 클래스가 인스턴스화 될 때 먼저 한번 인스턴스화 될 뿐 임.

*실체 클래스: 객체를 만들어 사용할 수 있는 클래스



# 8절. 추상 클래스(Abstract Class)

- 추상 클래스의 용도
  - 실체 클래스의 공통된 필드와 메소드의 이름 통일할 목적
    - 실체 클래스를 설계자가 여러 사람일 경우,
    - 실체 클래스마다 필드와 메소드가 제각기 다른 이름을 가질 수 있음
  - 실체 클래스를 작성할 때 시간 절약
    - 실체 클래스는 추가적인 필드와 메소드만 선언
  - 실체 클래스 설계 규격을 만들고자 할 때
    - 실체 클래스가 가져야 할 필드와 메소드를 추상 클래스에 미리 정의
    - 실체 클래스는 추상 클래스를 무조건 상속 받아 작성

# 8절. 추상 클래스(Abstract Class)

- 추상 클래스 선언
  - 클래스 선언에 abstract 키워드 사용
    - new 연산자로 객체 생성하지 못하고 상속을 통해 자식 클래스만 생성 가능

```
public abstract class 클래스 {
    //필드
    //생성자
    //메소드
}
```

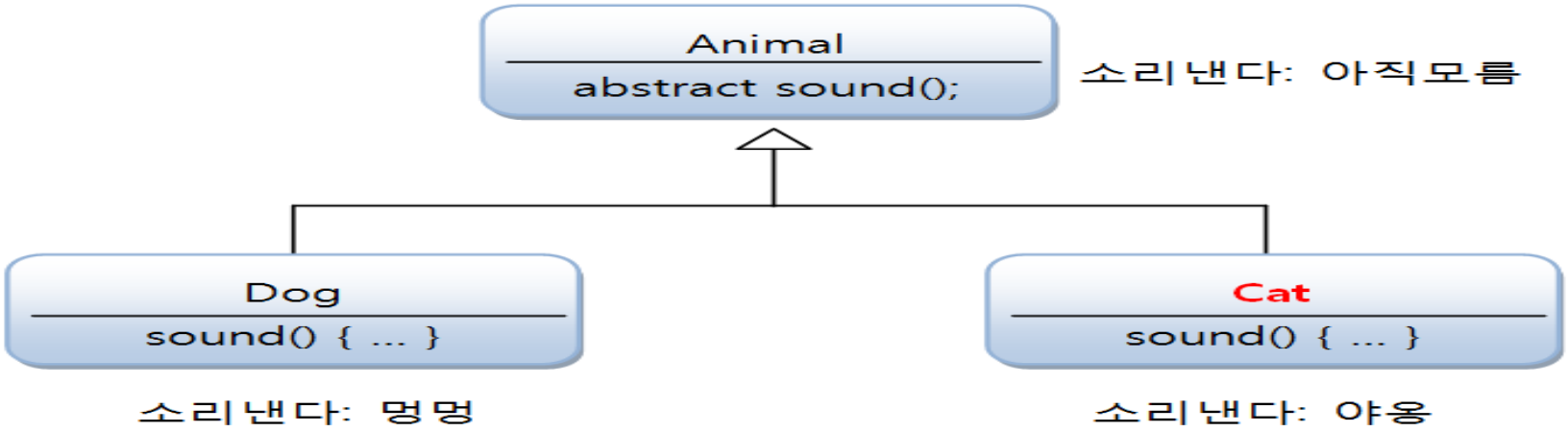
# 8절. 추상 클래스(Abstract Class)

## ❖ 추상 메소드와 오버라이딩(재정의)

- 메소드 이름 동일하지만, 실행 내용이 실체 클래스마다 다른 메소드
  - 예: 동물은 소리를 낸다. 하지만 실체 동물들의 소리는 제각기 다르다.
- 구현 방법
  - 추상 클래스에는 메소드의 선언부만 작성 (추상 메소드)
  - 실체 클래스에서 메소드의 실행 내용 작성(오버라이딩 (Overriding))

# 8절. 추상 클래스(Abstract Class)

## ❖ 추상 메소드와 오버라이딩(재정의)




```
public abstract class Animal {
    public abstract void sound();
}
```

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

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