



## 7장. 상속

이것이 자바다 (<http://cafe.naver.com/thisjava>)

# 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)



# 1절. 상속 개념

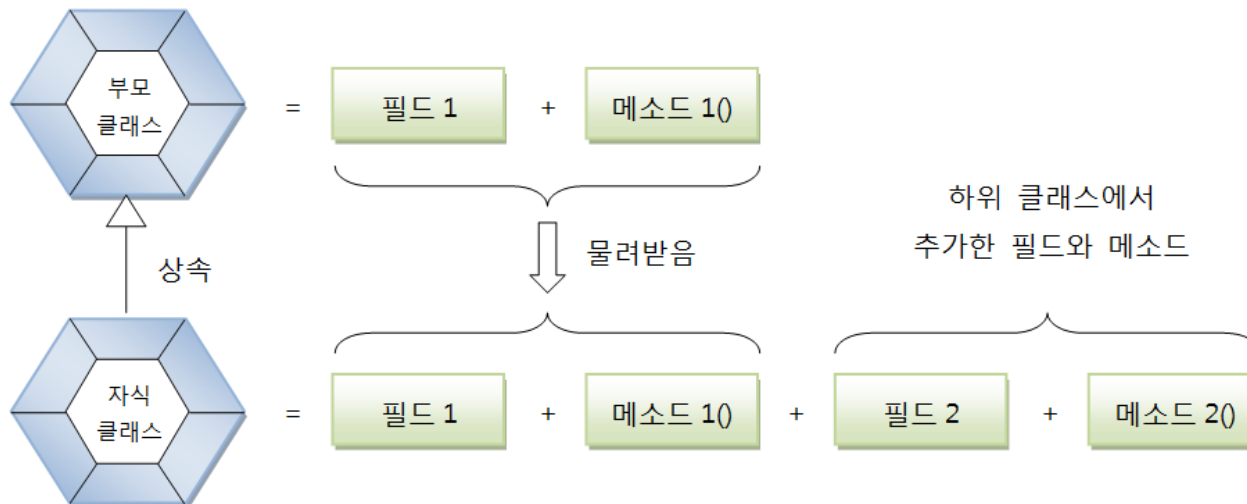
## ❖ 상속(Inheritance)이란?

### ■ 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) 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

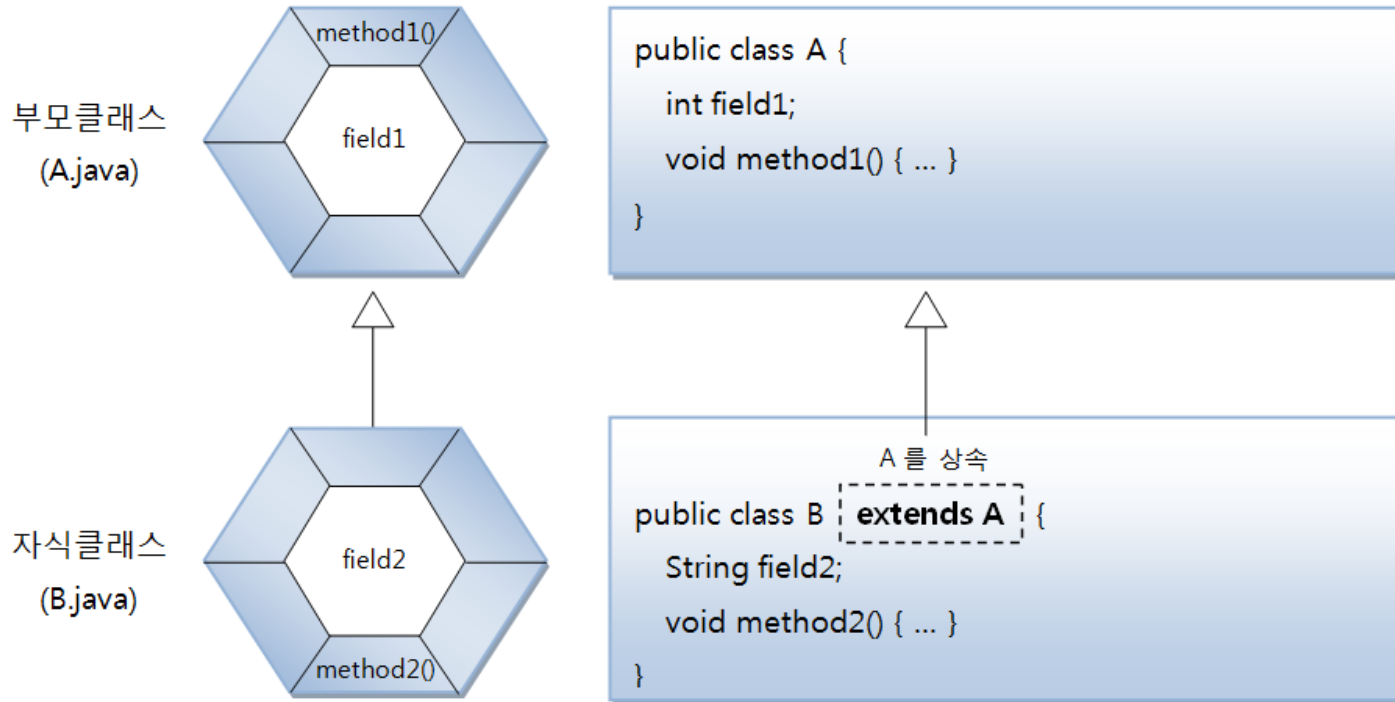
- 부모 클래스의 private 접근 갖는 필드와 메소드 제외
- 부모 클래스가 다른 패키지에 있을 경우, default 접근 갖는 필드와 메소드도 제외



## 2절. 클래스 상속(extends)

### ❖ extends 키워드

- 자식 클래스가 상속할 부모 클래스를 지정하는 키워드



- 자바는 단일 상속 - 부모 클래스 나열 불가

```
class 자식클래스 extends 부모클래스 1, 부모클래스 2 {  
}
```



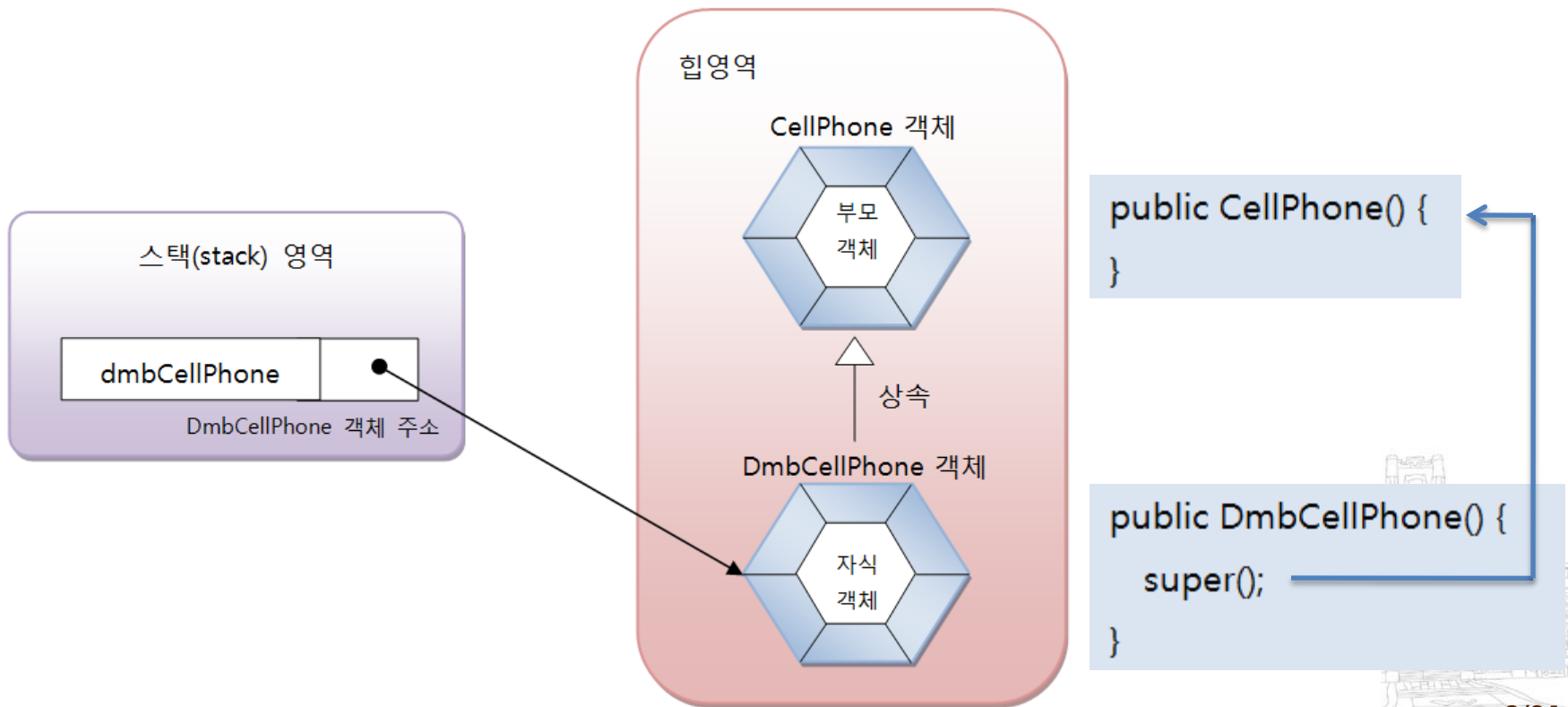
### 3절. 부모 생성자 호출(super(...))

#### ❖ 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 재정의하는 것
- 메소드 재정의 조건 (p.295~296)
  - 부모 클래스의 메소드와 동일한 시그니처 가져야
  - 접근 제한을 더 강하게 오버라이딩 불가
    - **public**을 **default**나 **private**으로 수정 불가
    - 반대로 **default**는 **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();
```

```
class Parent {  
    void method1() { ... }  
    void method2() { ... }  
}
```



상속

부모 메소드 호출

```
class Child extends Parent {  
    void method2() { ... } //Overriding  
    void method3() {  
        method2();  
        super.method2();  
    }  
}
```

재정의된 호출



## 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 final로 생성

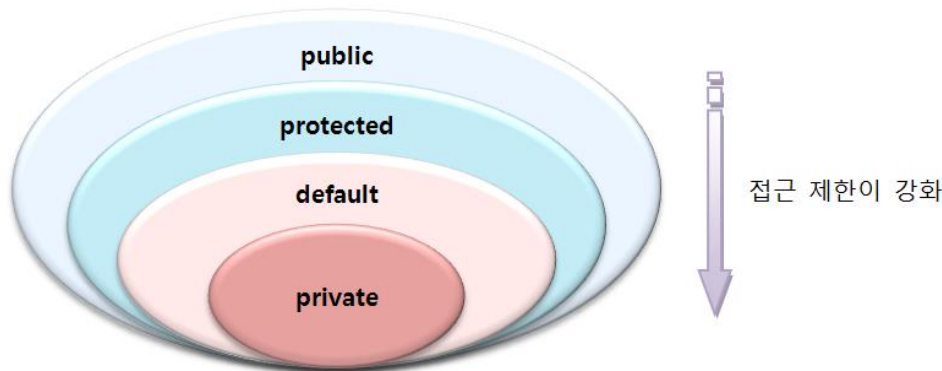


# 6절. protected 접근 제한자

## ❖ protected 접근 제한자

### ■ 상속과 관련된 접근 제한자

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



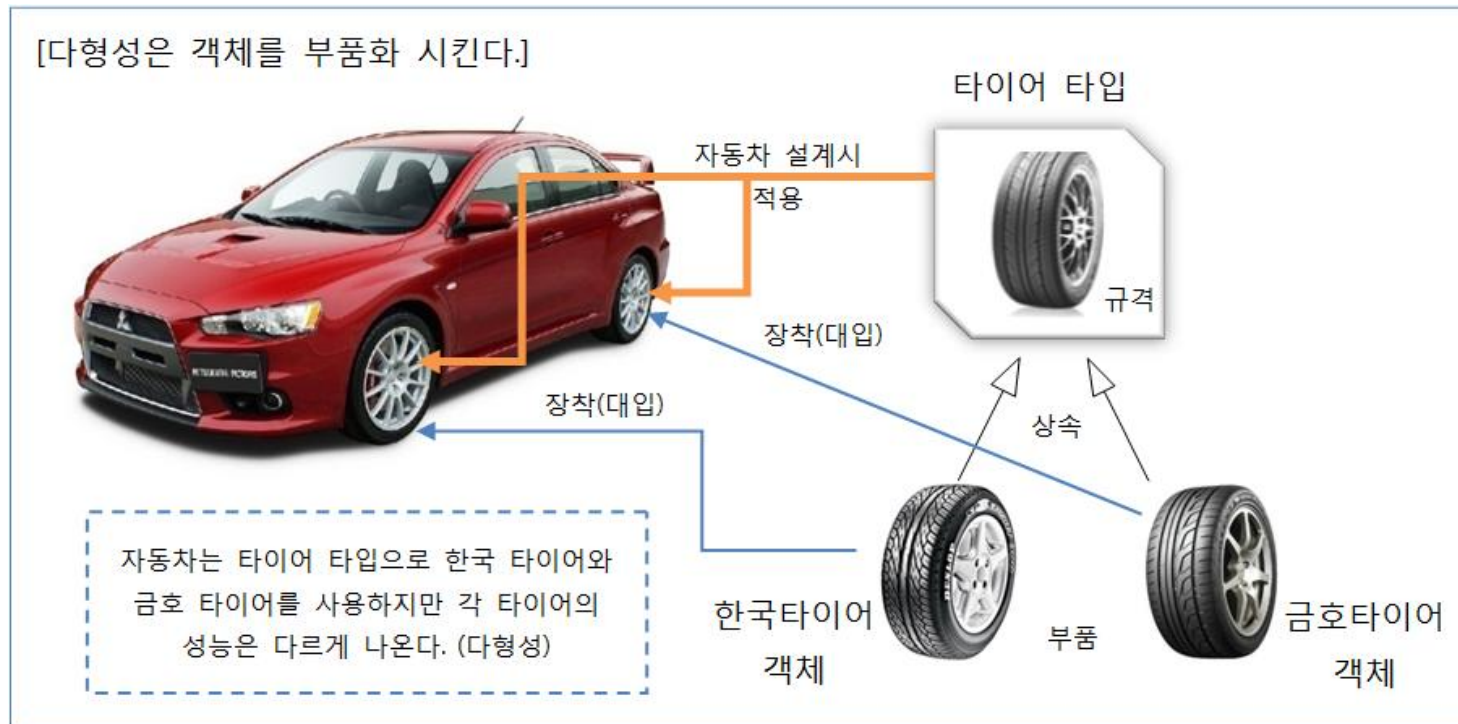
접근 제한	적용할 내용	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스



## 7절. 타입변환과 다형성(polymorphism)

### ❖ 다형성 (多形性, Polymorphism)

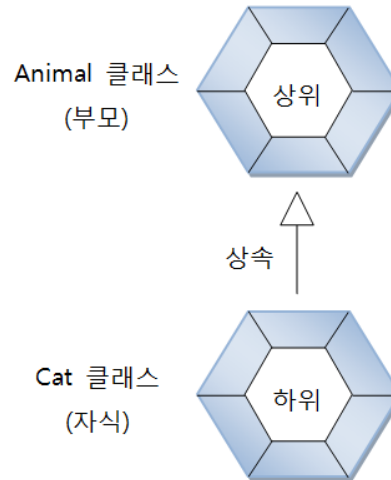
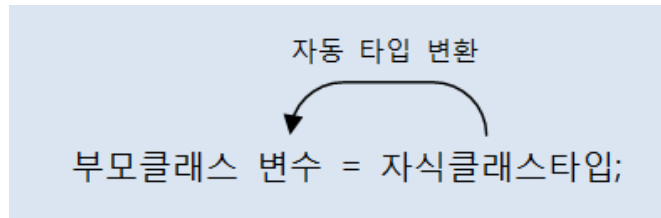
- 같은 타입이지만 실행 결과가 다양한 객체 대입(이용) 가능한 성질
  - 부모 타입에는 모든 자식 객체가 대입 가능
    - 자식 타입은 부모 타입으로 자동 타입 변환
- 효과: 객체 부품화 가능



# 7절. 타입변환과 다형성(polymorphism)

## ❖ 자동 타입 변환(Promotion)

- 프로그램 실행 도중에 자동 타입 변환이 일어나는 것



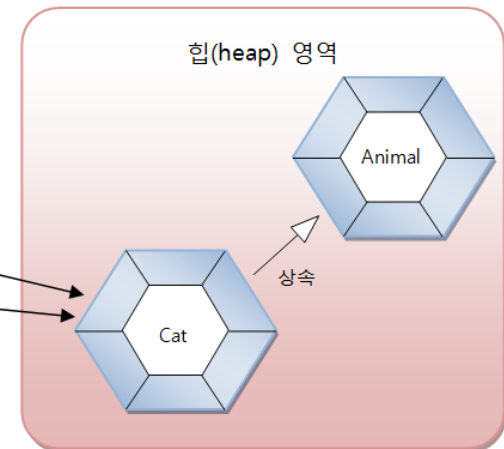
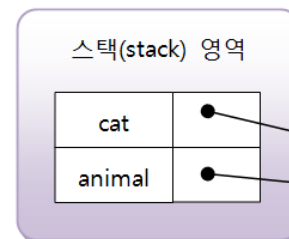
```
class Animal {  
    ...  
}
```

```
class Cat extends Animal {  
    ...  
}
```

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

Animal animal = new Cat(); 도 가능하다.

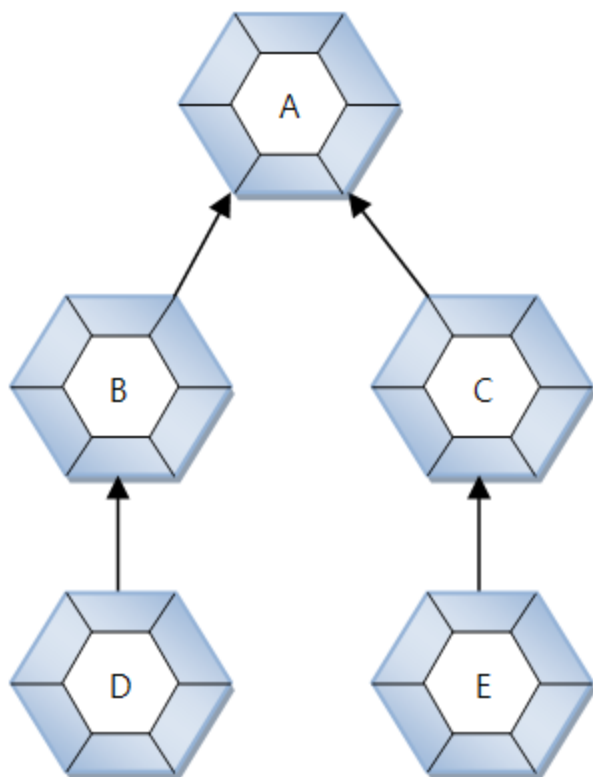
```
cat == animal //true
```



## 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)
```

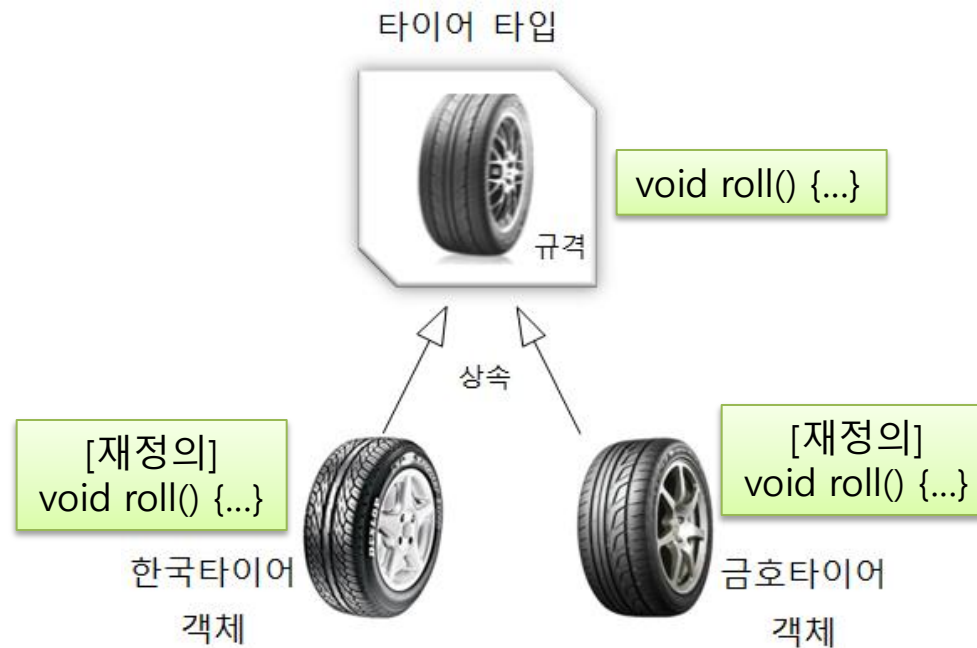


# 7절. 타입변환과 다형성(polymorphism)

## ❖ 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;  
}
```

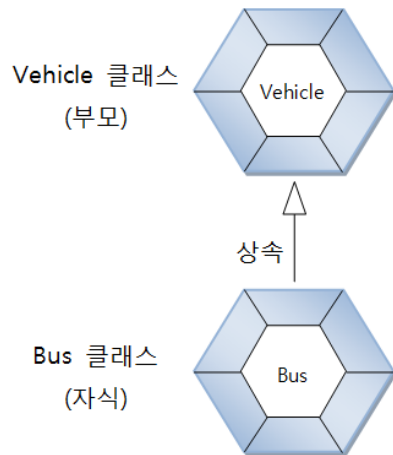
# 7절. 타입변환과 다형성(polymorphism)

## ❖ 매개변수의 다형성 (p.321~323)

### ■ 매개변수가 클래스 타입일 경우

- 해당 클래스의 객체 대입이 원칙이나 자식 객체 대입하는 것도 허용
  - 자동 타입 변환
  - 매개변수의 다형성

```
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;

자식 객체

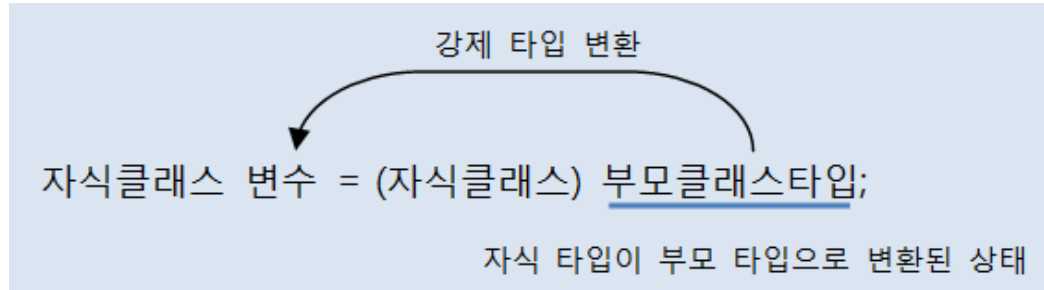
```
void drive(Vehicle vehicle) {  
    vehicle.run();  
}
```

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 변환 이 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

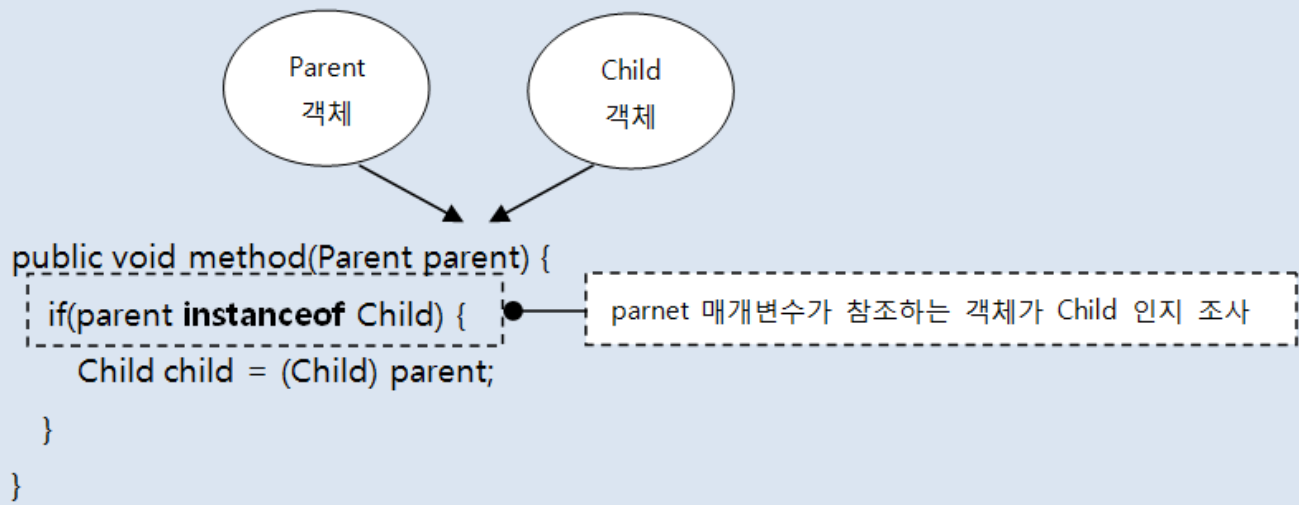
- **ClassCastException** 예외 발생 가능

```
Parent parent = new Parent();
```

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

- 먼저 자식 타입인지 확인 후 강제 타입 실행해야 함

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



# 8절. 추상 클래스(Abstract Class)

## ❖ 추상 클래스 개념

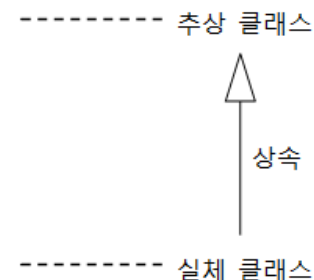
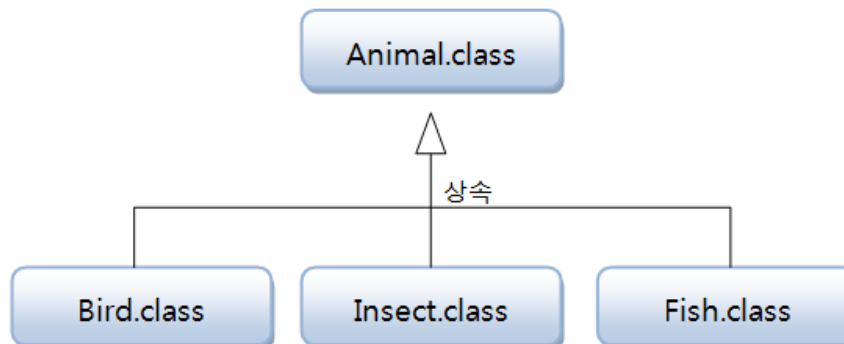
### ■ 추상(abstract)

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

### ■ 추상 클래스(abstract class)

- 실체 클래스들의 공통되는 필드와 메소드 정의한 클래스
- 추상 클래스는 실체 클래스의 부모 클래스 역할 (단독 객체 X)

\*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 **상속 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 실제 클래스마다 다른 메소드
- 예: 동물은 소리를 낸다. 하지만 실제 동물들의 소리는 제각기 다르다.
- 구현 방법
  - 추상 클래스에는 메소드의 선언부만 작성 (추상 메소드)
  - 실제 클래스에서 메소드의 실행 내용 작성(오버라이딩(Overriding))

