

# COMP217

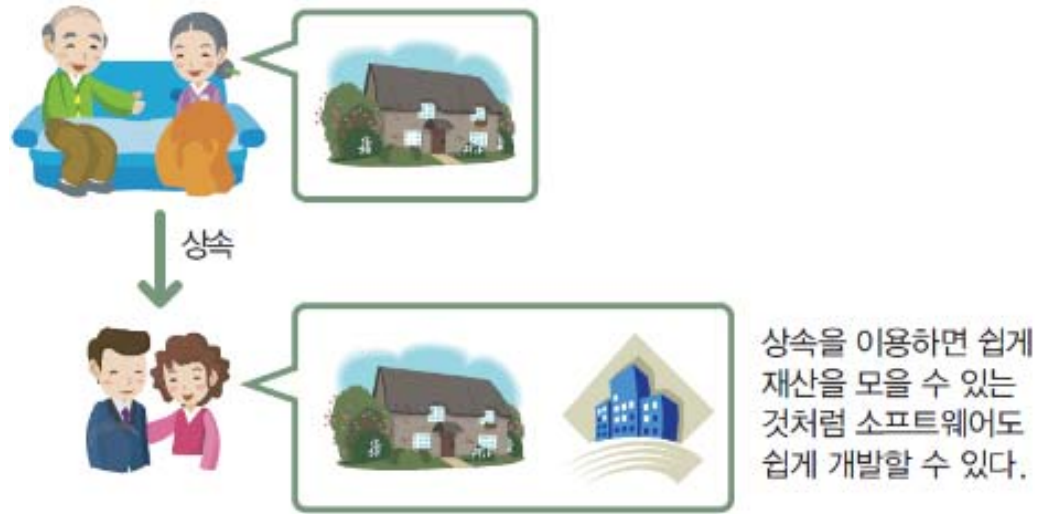
## JAVA Programming

### Spring 2021

*Inheritance, object class, polymorphism*

# Goals

- 상속이란?
- 상속의 사용
- 메소드 재정의
- 접근 지정자
- 상속과 생성자
- Object 클래스
- 종단 클래스
- 다형성



# Concept of Inheritance

Receiving useful features from parents

# 상속의 장점

- 상속의 장점
  - 상속을 통하여 기존 클래스의 필드와 메소드를 재사용
  - 신뢰성 있는 소프트웨어를 손쉽게 개발, 유지 보수
  - 상속은 이미 작성된 검증된 소프트웨어를 재사용
  - 코드의 중복을 줄일 수 있다.
  - 기존 클래스의 일부 변경도 가능

# 상속

```
class SubClass extends SuperClass
{
    ...// 추가된 메소드와 필드
}
```

상속을 의미한다. 슈퍼 클래스를 확장하여  
서브 클래스를 작성한다는 의미이다.

슈퍼 클래스==부모 클래스

```
class Car
{
    ...
}
class SportsCar extends Car
{
    ...
}
```



슈퍼 클래스(superclass)

서브 클래스(subclass)

# 상속

수퍼 클래스	서브 클래스
Animal(동물)	Lion(사자), Dog(개), Cat(고양이)
Bike(자전거)	MountainBike(산악자전거)
Vehicle(탈것)	Car(자동차), Bus(버스), Truck(트럭), Boat(보트), Motorcycle(오토바이), Bicycle(자전거)
Student(학생)	GraduateStudent(대학원생), UnderGraduate(학부생)
Employee(직원)	Manager(관리자)
Shape(도형)	Rectangle(사각형), Triangle(삼각형), Circle(원)

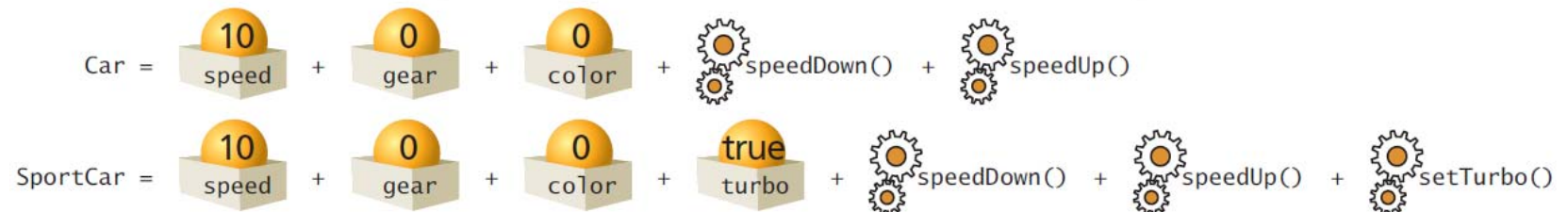
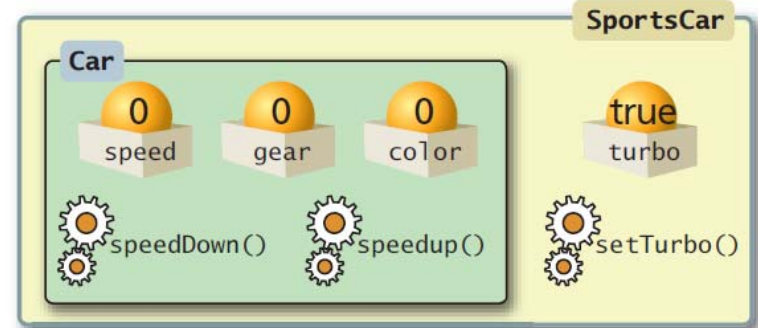
1. 컴퓨터, 데스크탑, 노트북, 태블릿 사이의 상속 관계를 결정하여 보자,
2. 상속의 장점은 무엇인가?

# 상속 사용

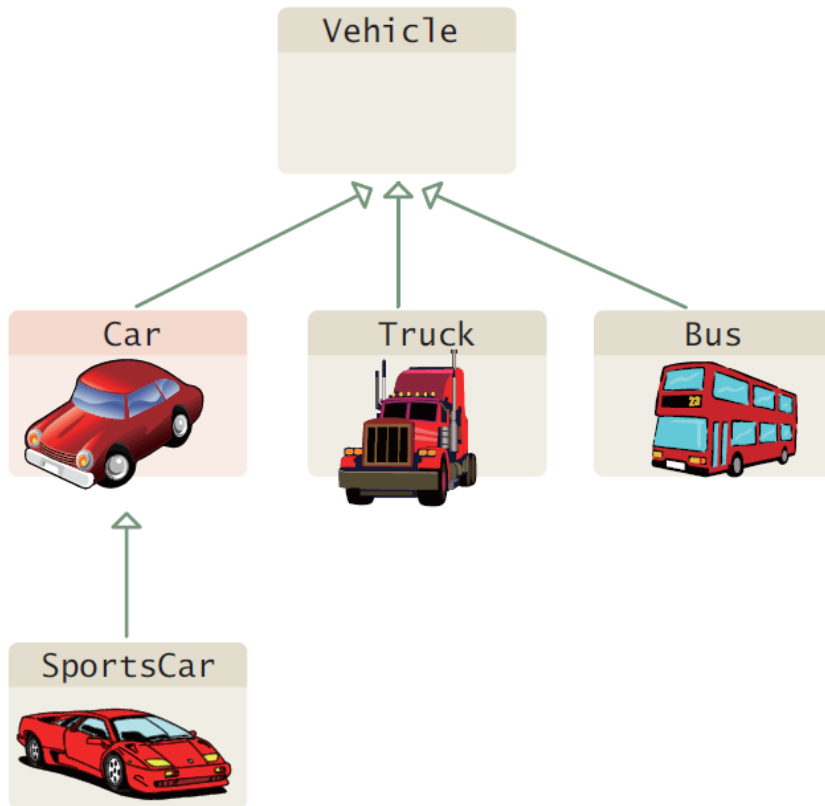
```
public class Car {  
    // 3개의 필드 선언  
    int speed;           // 속도  
    int gear;            // 기어  
    public String color; // 색상, 테스트를 위하여 공용 필드로 만들자.  
  
    public void speedUp(int increment) { // 속도 증가  
        speed += increment;  
    }  
    public void speedDown(int decrement) { // 속도 감소  
        speed -= decrement;  
    }  
}
```

```
public class Test {  
    // 서브 클래스 객체 생성  
    public static void main(String[] args) {  
        SportsCar c = new SportsCar();  
        c.color = "Red";  
        c.speedUp(100); // 슈퍼 클래스 필드와 메소드 사용  
        c.speedDown(30);  
        c.setTurbo(true); // 자체 메소드 사용  
    }  
}
```

```
class SportsCar extends Car { // Car를 상속받는다.  
    boolean turbo;  
  
    public void setTurbo(boolean newValue) { // 터보 모드 설정 메소드  
        turbo = newValue;  
    }  
}
```



# 상속의 계층 구조



```
class Vehicle { ... }  
class Car extends Vehicle { ... }  
class Truck extends Vehicle { ... }  
class Bus extends Vehicle { ... }  
class SportsCar extends Car { ... }
```

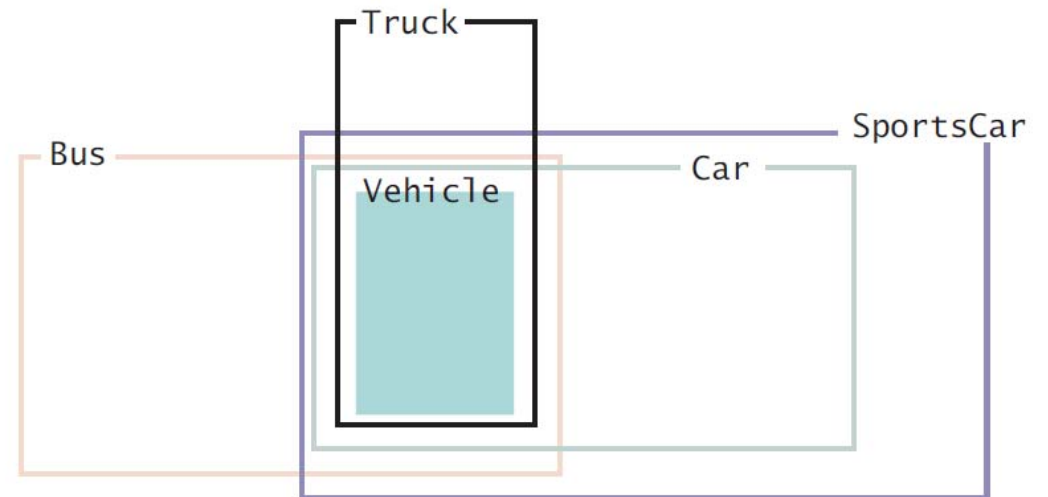


그림11-3. 상속 계층 구조도



# 상속은 중복을 줄인다.

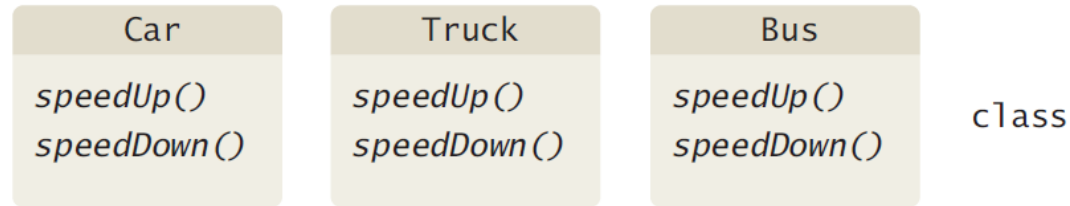


그림11-5. 각 클래스에 코드가 중복된다.

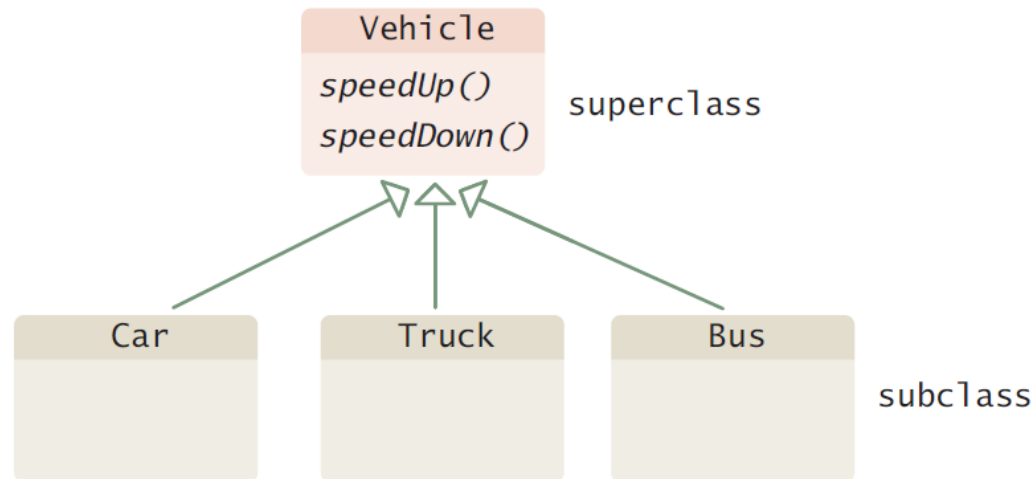


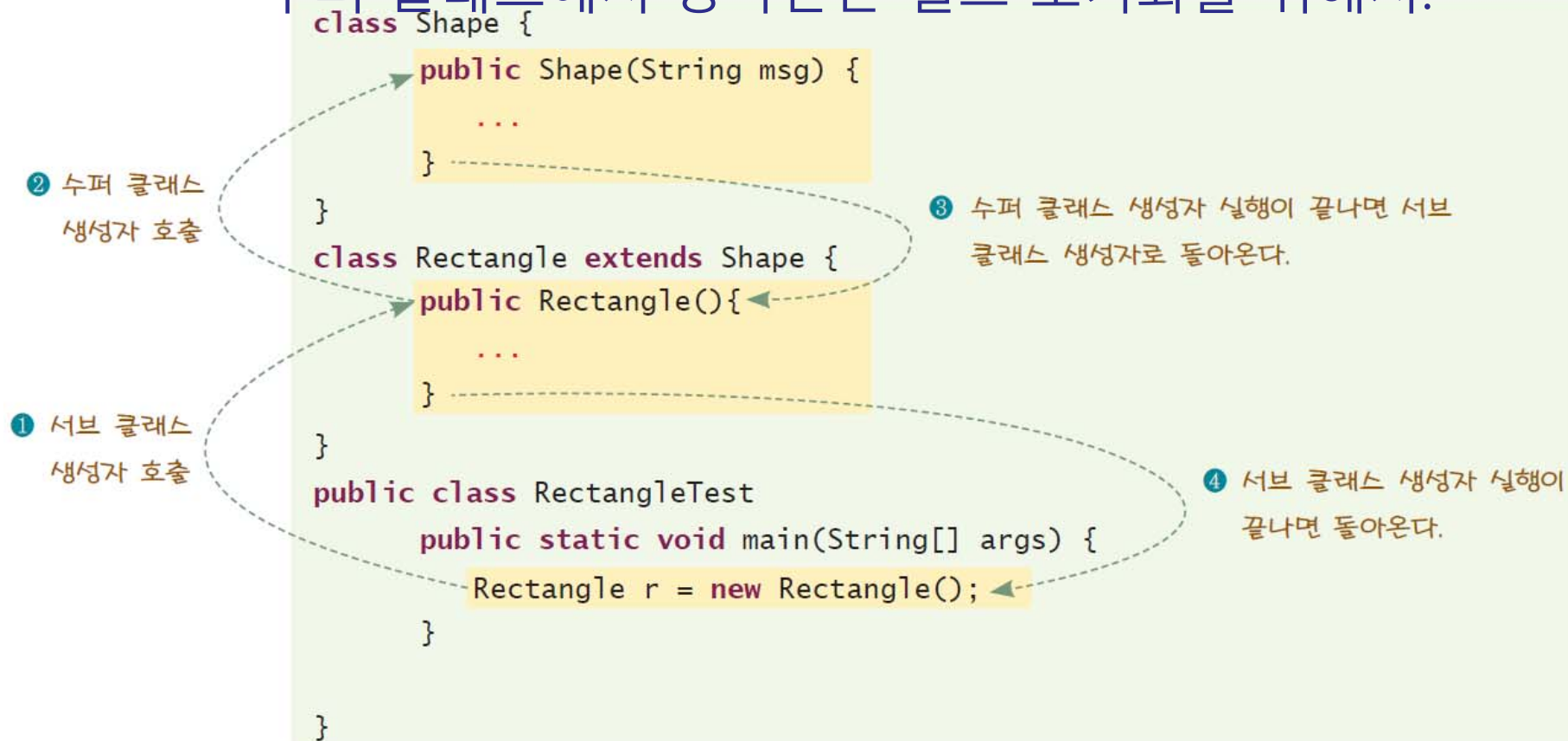
그림11-6. 중복되는 코드는 수퍼 클래스에 모은다.

# 중간점검

1. Animal 클래스 (sleep(), eat()), Dog 클래스(sleep(), eat(), bark()), Cat 클래스 (sleep(), eat(), play())를 상속을 이용하여서 표현하면 어떻게 코드가 간결해지는가?
2. 일반적인 상자(box)를 클래스 Box로 표현하고, Box를 상속받는 서브 클래스인 ColorBox(컬러 박스) 클래스를 정의하여 보자. 적절한 필드(길이, 폭, 높이)와 메소드(부피 계산)를 정의한다.

# 상속과 생성자

- 서브 클래스의 객체가 생성될 때
  - 서브 클래스의 생성자만 호출될까?
  - 수퍼 클래스의 생성자도 호출되는가?
  - 수퍼 클래스에서 상속받은 필드 초기화를 위해서.



# 명시적인 호출

- `super`를 이용하여서 명시적으로 수퍼 클래스의 생성자 호출

```
class Shape {  
    public Shape(String msg) {  
        System.out.println("Shape 생성자() " + msg);  
    }  
}
```

`new Rectangle();`

```
public class Rectangle extends Shape {  
    public Rectangle() {  
        super("from Rectangle");  
        System.out.println("Rectangle 생성자()");  
    }  
}
```

실행결과

// 명시적인 호출

## 실행결과

```
Shape 생성자 from Rectangle  
Rectangle 생성자
```

# 묵시적인 호출

```
class Shape {  
    public Shape() {  
        System.out.println("Shape 생성자()");  
    }  
}  
class Rectangle extends Shape {  
    public Rectangle() {  
        super();  
        System.out.println("Rectangle 생성자()");  
    }  
}
```

**new Rectangle();**

컴파일러가 Shape();을 자동적으로 넣어준다고 생각하라.

## 실행결과

Shape 생성자  
Rectangle 생성자

# 묵시적인 호출

```
class Shape {  
    Shape() {} // OK! 자동으로 디폴트 생성자 추가!  
}
```

**new Rectangle();**

```
public class Rectangle extends Shape {  
    public Rectangle() {  
        super();  
        System.out.println("Rectangle 생성자()");  
    }  
}
```

Shape();을 자동적으로  
넣어준다고 생각하라.

실행결과

Rectangle 생성자()

```
class Shape {  
    public Shape(String msg) { // 디폴트 생성자는 없음! new Rectangle();  
        System.out.println("Shape 생성자()" + msg);  
    }  
}
```

디폴트 생성자 Shape()을 호출할 수 없기  
때문에 컴파일 오류가 발생한다.

```
public class Rectangle extends Shape {  
    public Rectangle() { super(); // 오류: Shape()를 호출할 수 없음!  
        System.out.println("Rectangle 생성자()");  
    }  
}
```

실행결과

Implicit super constructor Shape() is undefined. Must explicitly invoke another constructor

# Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as *constructor chaining*.

```
public class Faculty extends Employee {
    public static void main(String[] args) { new Faculty(); }

    public Faculty() {
        System.out.println("(1) Faculty's no-arg constructor is invoked");
    }
}

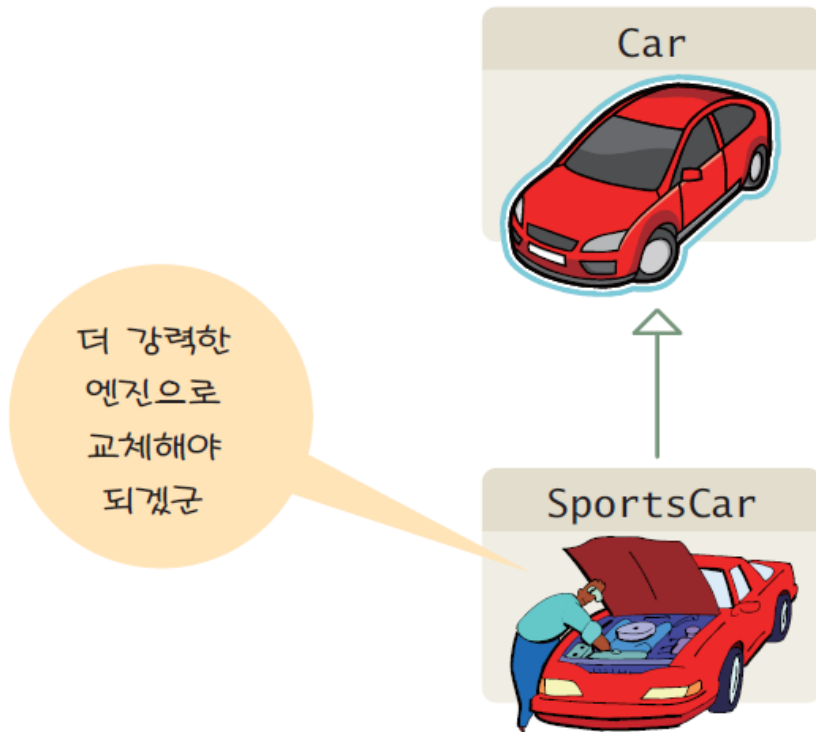
class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) { System.out.println(s); }
}

class Person {
    public Person() {
        System.out.println("(4) Person's no-arg constructor is invoked");
    }
}
```

# 메소드 재정의

- 메소드 재정의(method overriding): 서브 클래스가 필요에 따라 상속된 메소드를 다시 정의하는 것





# 메소드 재정의의 예

```
class Animal {  
    public void sound()  
    {  
        // 아직 특정한 동물이 지정되지 않았으므로 몸체는 비어 있다.  
    }  
}
```

```
class Dog extends Animal {  
    public void sound()  
    {  
        System.out.println("멍멍!");  
    }  
}  
  
public class DogTest {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        d.sound();  
    }  
}
```

메소드 재정의

재정의된 메소드가 호출된다.

실행결과

멍멍!

# @Override

```
class Dog extends Animal {
```

```
    void saund() {  
        System.out.println("멍멍!");  
    }
```

```
}
```

재정의할 의도하였으나 이름을 잘못 입력하였기  
때문에 컴파일러는 새로운 메소드 정의로 간주한다.

```
class Dog extends Animal {
```

```
    @Override  
    void saund() {                // 오류 발생!  
        System.out.println("멍멍!");  
    }
```

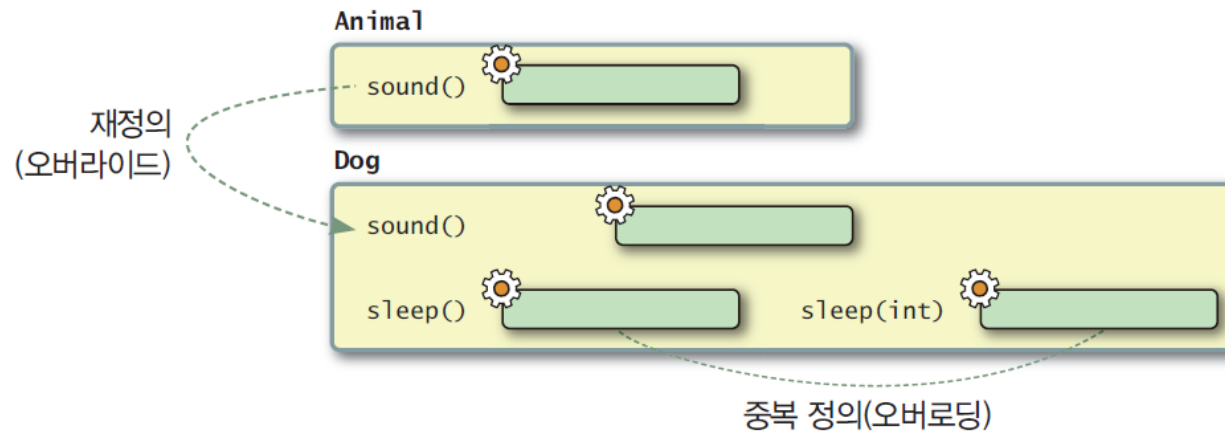
```
}
```

재정의할 의도하였다는 것을 확실하게  
컴파일러에게 전달하여 오류를 막는다.

## 실행결과

The method saund() of type Dog must override or implement a supertype method

# 중복 정의와 재정의의 차이



```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

# Super

```
class ParentClass {  
    int data=100;  
    public void print() {  
        System.out.println("수퍼 클래스의 print() 메소드");  
    }  
}
```

```
public class ChildClass extends ParentClass {  
    int data=200;  
    public void print() {    //메소드 재정의  
        super.print(); ← 수퍼 클래스의 메소드 호출  
        System.out.println("서브 클래스의 print() 메소드 ");  
        System.out.println(this.data);  
        System.out.println(super.data); ← 수퍼 클래스의 필드 접근  
    }  
    public static void main(String[] args) {  
        ChildClass obj = new ChildClass();  
        obj.print();  
    }  
}
```

## 실행결과

```
수퍼 클래스의 print() 메소드  
서브 클래스의 print() 메소드  
200  
100
```

# 접근 지정자

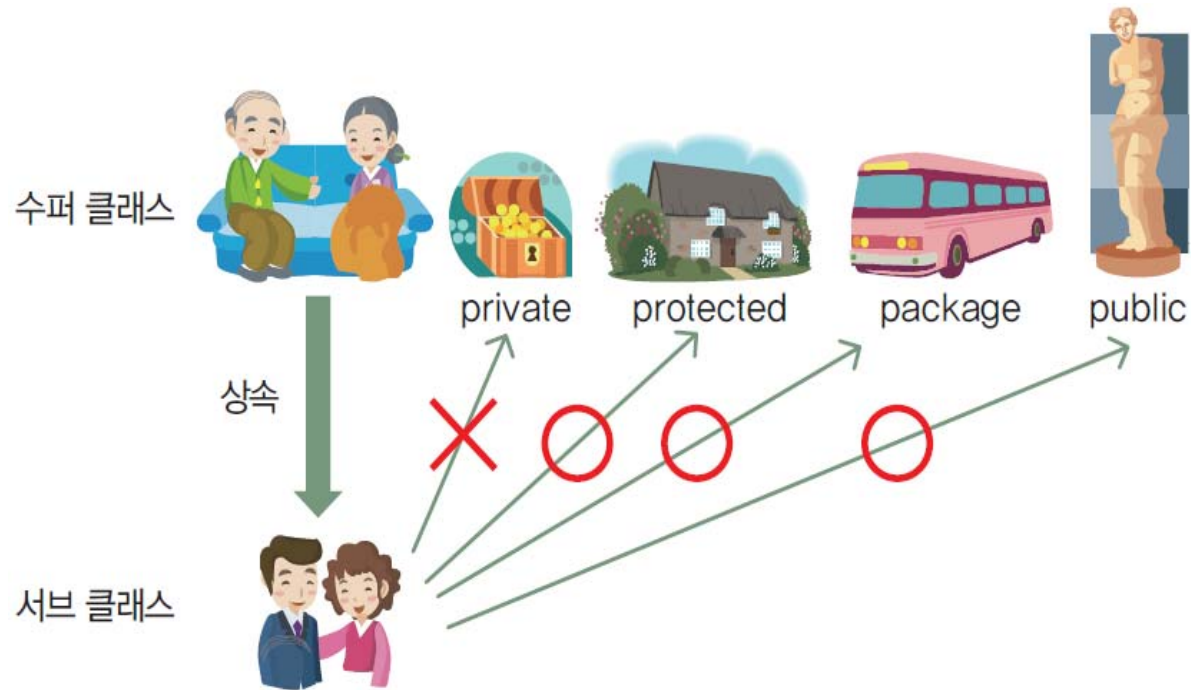


그림11-7. 상속에서 접근 지정자

## 예제

```

01 class Employee {
02     public String name;        // 이름: 공용 멤버
03     String address;           // 주소: 패키지 멤버
04     protected int salary;      // 월급: 보호 멤버
05     private int RRN;           // 주민등록번호: 전용 멤버
06
07     public String toString() {
08         return name + ", " + address + ", " + RRN + ", " + salary;
09     }
10 }
11
12 class Manager extends Employee {
13     private int bonus;
14
15     public void printSalary() {
16         System.out.println(name + "(" + address + "):" + (salary + bonus));
17     }
18
19     public void printRRN() {
20         System.out.println(RRN);
21     }
22 }
23 public class ManagerTest {
24     public static void main(String[] args) {
25         Manager m = new Manager();
26         m.printRRN();
27     }
28 }

```

수퍼클래스에서 private로 정의된 멤버는 서브 클래스에서 접근할 수 없다.

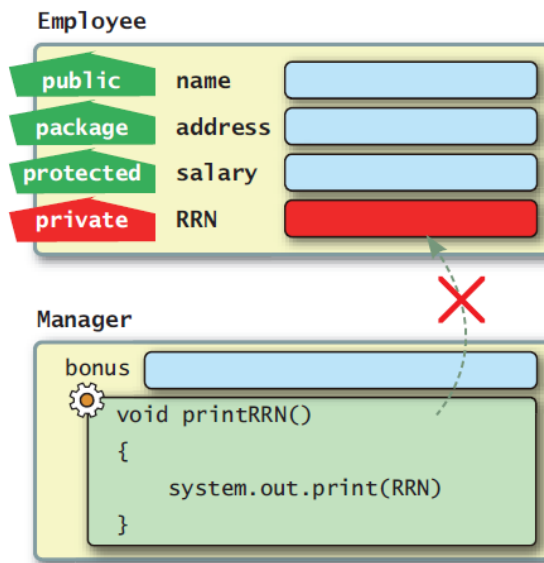
수퍼클래스의 private 멤버를 제외한 모든 멤버 접근 가능

오류! private는 서브 클래스에서 접근 못함!

**실행결과**

The field Employee.RRN is not visible

ManagerTest.java:20: error: RRN has private access in Employee  
System.out.println(RRN);



# Object class

Origin of all classes

# Object 클래스

- Object 클래스는 java.lang 패키지에 들어 있으며 자바 클래스 계층 구조에서 맨 위에 위치하는 클래스

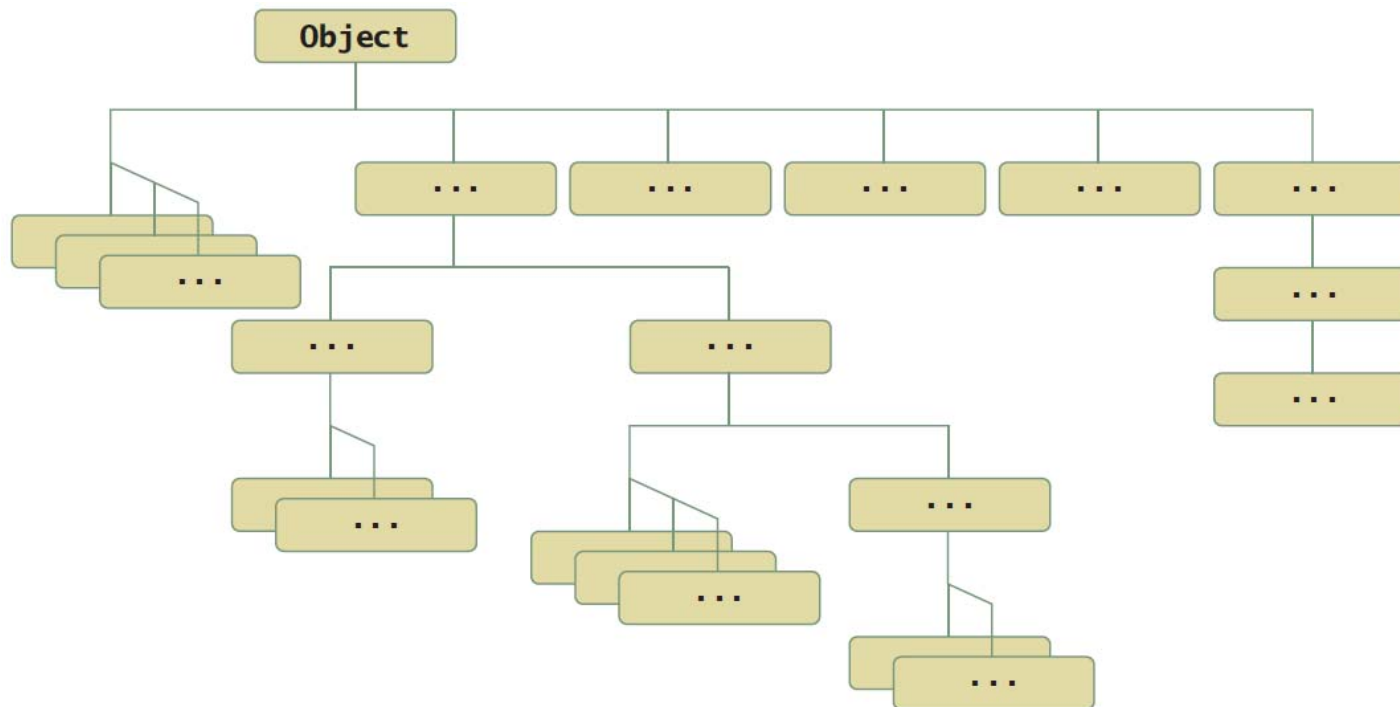


그림11-9.Object 클래스는 상속 계층 구조의 맨 위에 있다(출처: java.sun.com)



# Object의 메소드

- **protected Object clone()**
    - 객체 자신의 복사본을 생성하여 반환한다.
  - **public boolean equals(Object obj)**
    - obj가 이 객체와 같은지를 나타낸다.
  - **protected void finalize()**
    - 가비지 콜렉터에 의하여 호출된다.
  - **public final Class getClass()**
    - 객체를 생성한 클래스 정보를 반환한다.
  - **public int hashCode()**
    - 객체에 대한 해쉬 코드를 반환한다.
  - **public String toString()**
    - 객체의 문자열 표현을 반환한다.
- equals(), finalize(), toString() 은 보통 child class 에서 override 되어야 한다.  
추가되는 member에 대한 처리를 해주어야 하기 때문.**

# getClass()

```
class Car {  
    ...  
}  
  
public class CarTest {  
    public static void main(String[] args) {  
        Car obj = new Car();  
        System.out.println("obj is of type " + obj.getClass().getName());  
    }  
}
```

객체를 생성한 클래스 이름을 반환한다.

## 실행결과

obj is of type Car

# equals() 메소드

```
class Car {  
    private String model;  
    public Car(String model) {        this.model= model;    }  
    public boolean equals(Object obj) {  
        if (obj instanceof Car)  
            return model.equals(((Car) obj).model);  
        else  
            return false;  
    }  
}
```

← equals()를 재정의한다. String의 equals()를 호출하여서 문자열이 동일한지를 검사한다.

```
public class CarTest {  
    public static void main(String[] args) {  
        Car firstCar = new Car("BMW520");  
        Car secondCar = new Car("BMW520");  
        if (firstCar.equals(secondCar)) {  
            System.out.println("동일한 종류의 자동차입니다.");  
        }  
        else {  
            System.out.println("동일한 종류의 자동차가 아닙니다.");  
        }  
    }  
}
```

이 equals() 메소드를 사용하여 검사하는 다음과 같은 코드를 가정할 있다.

## 실행결과

동일한 종류의 자동차입니다.

# method equals()

```
1 class Car {
2     private String model;
3     public Car(String model){
4         this.model= model;
5     }
6     public boolean equals(Object obj) {
7         if (obj instanceof Car)
8             return model.equals(((Car) obj).model);
9         else
10            return false;
11    }
12 }
13 class Bike{
14     private int numOfWeek;
15     Bike(int i){
16         numOfWeek = i;
17     }
18 }
19 public class CarTest {
20     public static void main(String[] args) {
21         Car firstCar = new Car("AVANTE");
22         Car secondCar = new Car("AVANTE");
23         Bike bike = new Bike(4);
24
25         if (firstCar.equals(bike)) { //if (firstCar.equals(secondCar)) {
26             System.out.println("동일한 종류의 자동차입니다.");
27         } else {
28             System.out.println("동일한 종류의 자동차가 아닙니다.");
29         }
30     }
31 }
```

# toString() 메소드

- Object 클래스의 toString() 메소드는 객체의 문자열 표현을 반환

```
System.out.println(firstCar.toString());
```

## 실행결과

모델 HMW 520...

```
class Car {  
    private String model;  
    public Car(String model) {        this.model= model  
    public boolean equals(Object obj) {  
        if (obj instanceof Car)  
            return model.equals(((Car) obj).model);  
        else  
            return false;  
    }  
}
```

```
public String toString(){  
    return "모델"+this.model+"...";  
}
```

# 종단 클래스와 종단 메소드

- Class에 키워드 `final`을 붙이면 더 이상 상속할 수 없다.
- 메서드에 키워드 `final`을 붙이면 메서드를 재정의할 수 없다.

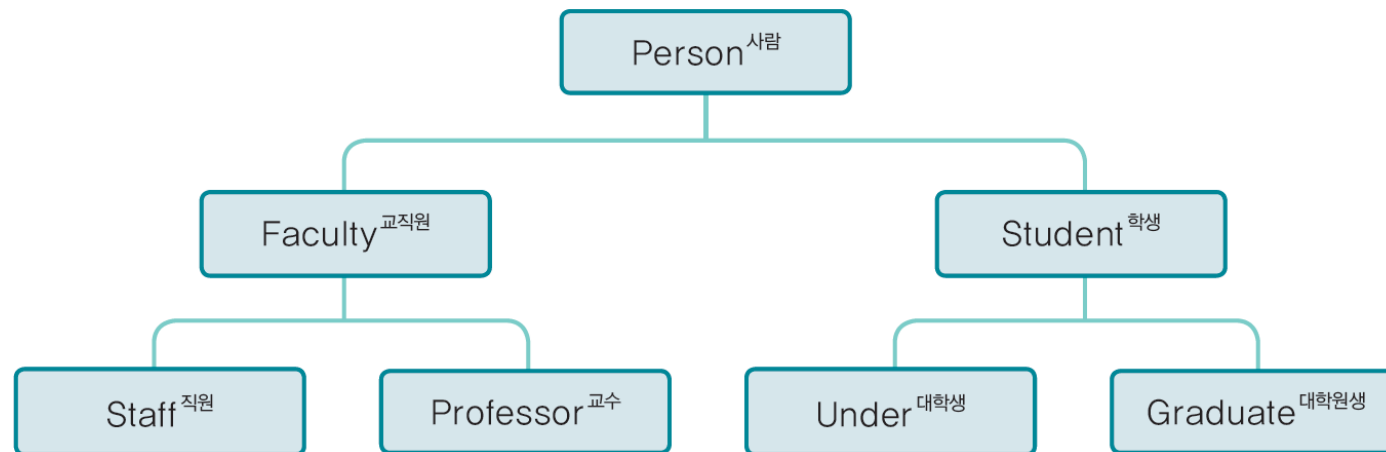
```
final class String {  
    ...  
}
```

```
class Baduk {  
    enum BadukPlayer { WHITE, BLACK }  
    ...  
    final BadukPlayer getFirstPlayer() {  
        return BadukPlayer.BLACK;  
    }  
}
```

서브 클래스에서 재정의할 수  
없도록 `final`로 지정한다.

# 클래스 Person

- 학교의 교직원과 학생 정보를 표현하기 위한 계층도



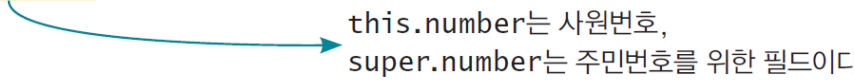
Person.java

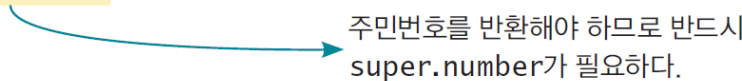
```
01 package inheritance.typecast;
02
03 public class Person {
04     public String name;    //이름
05     public long number;    //주민번호
06
07     public Person(String name, long number) {
08         super();           //생략 가능
09         this.name = name;
10         this.number = number;
11     }
12 }
```

# 교직원과 직원 클래스

## Faculty.java

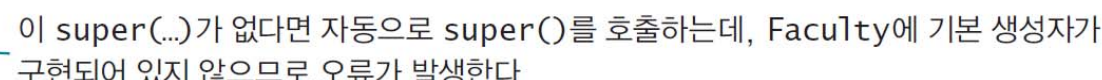
```
01 package inheritance.typecast;
02
03 public class Faculty extends Person {
04     public String univ;
05     public long number;
06
07     public Faculty(String name, long number, String univ, long idNumber) {
08         super(name, number);
09         this.univ = univ;
10         this.number = idNumber;
11     }
12
13     public long getSNumber() {
14         return super.number;
15     }
16 }
```

 this.number는 사원번호,  
super.number는 주민번호를 위한 필드이다.

 주민번호를 반환해야 하므로 반드시  
super.number가 필요하다.

## Staff.java

```
01 package inheritance.typecast;
02
03 public class Staff extends Faculty {
04     public String division;
05
06     public Staff(String name, long number, String univ, long idNumber) {
07         super(name, number, univ, idNumber);
08     }
09 }
```

 이 super(...)가 없다면 자동으로 super()를 호출하는데, Faculty에 기본 생성자가  
구현되어 있지 않으므로 오류가 발생한다.

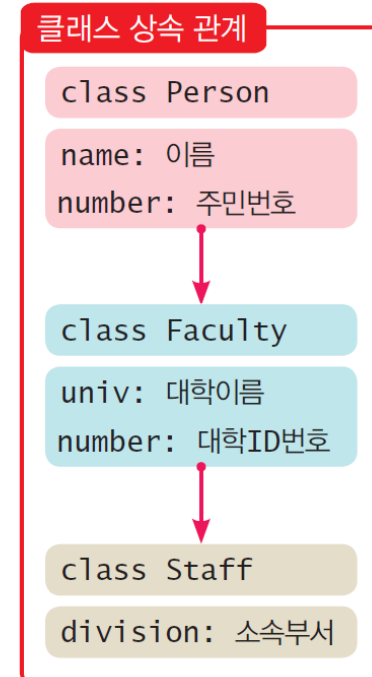
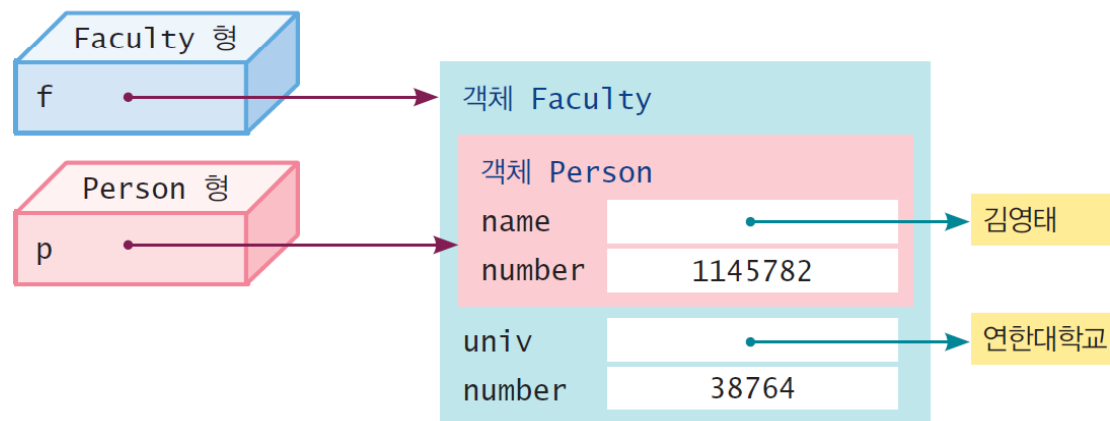


# 업 캐스팅

- 업 캐스팅
  - 하위 객체는 상위 클래스형 변수에 대입이 가능, 상위로의 자료형 변환
    - 업 캐스팅은 하위인 **교직원**은 상위인 **사람**이라는 개념이 성립
- 업 캐스팅의 제약
  - 업 캐스팅된 변수로는 하위 객체의 멤버를 참조할 수 없는 제약
    - Faculty** 형 변수 **f**로는 접근 지정자만 허용하면 모든 멤버를 접근 가능
    - Person** 형 변수 **p**로는 **Person**의 멤버인 **name**과 **number**만 접근이 가능

클래스자료형 변수 = 하위\_클래스\_자료형의\_객체\_또는\_변수;

```
Faculty f = new Faculty("김영태", 1145782, "연한대학교", 38764);  
Person p = f;
```



# 캐스팅(예제)

```
class Pet{...}
```

```
class Dog extends Pet{...} //강아지 is a 애완동물
```

```
class Cat extends Pet{...} //고양이 is a 애완동물
```

```
Pet p1 = new Pet(); // 가능
```

```
Pet p2 = new Dog(); // 강아지도 애완동물이기 때문에 가능
```

```
Pet p3 = new Cat(); // 고양이도 애완동물이기 때문에 가능
```

```
Cat p4 = new Pet(); // ?
```

# 업 캐스팅 예제

UpCasting.java

```
01 package inheritance.typecast;
02
03 public class UpCasting {
04     public static void main(String[] args) {
05         Person she = new Person("이소라", 2056432);
06         System.out.println(she.name + " " + she.number);
07
08         Faculty f = new Faculty("김영태", 1145782, "연한대학교", 38764);
09         Person p = f;           //업캐스팅
10         System.out.print(p.name + " " + p.number + " ");
11         //System.out.print(p.univ); //참조 불가능
12         System.out.println(f.name + " " + ((Person) f).number);
13         System.out.println(f.univ + " " + f.number);
14
15         Staff s = new Staff("김상기", 1187543, "강서대학교", 3456);
16         s.division = "교학처";
17         Person pn = s;           //업캐스팅
18         Faculty ft = s;          //업캐스팅
19         System.out.print(pn.name + " " + pn.number + " ");
20         System.out.print(ft.univ + " " + ft.number + " ");
21         System.out.println(s.division);
22     }
23 }
```

결과

이소라 2056432

김영태 1145782 연한대학교 38764

김상기 1187543 강서대학교 3456 교학처

# 다운 캐스팅

- 상위 클래스 형을 하위 클래스 형으로 변환
- 다운 캐스팅은 반드시 명시적인 형변환 연산자 (하위 클래스)가 필요
  - 만일 형변환 연산자가 없으면 컴파일 시간에 오류 발생

클래스자료형 변수 = (클래스자료형) 상위\_클래스\_자료형의\_객체\_또는\_변수;

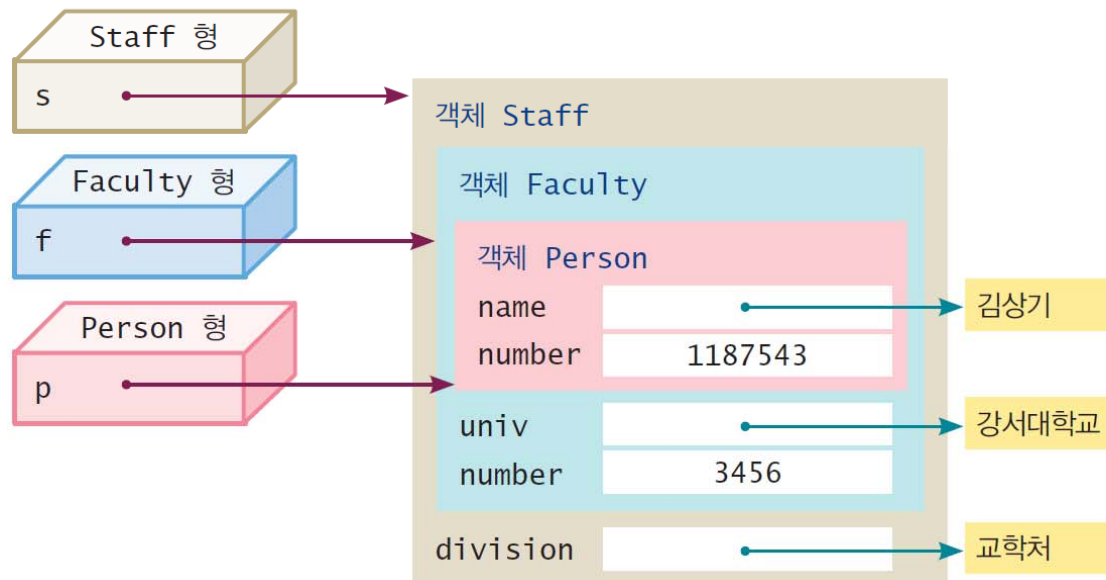
```
Person p = new Staff("김상기", 1187543, "강서대학교", 3456);
```

```
Staff s = (Staff) p;
```

```
Staff s = p;
```

Type mismatch cannot convert from Person to Staff

```
s.division = "교학처";
```



## 클래스 개층 정보

```
class Person
```

name: 이름

number: 주민번호

```
class Faculty
```

univ: 대학 이름

number: 대학 ID 번호

```
class Staff
```

division: 소속부서

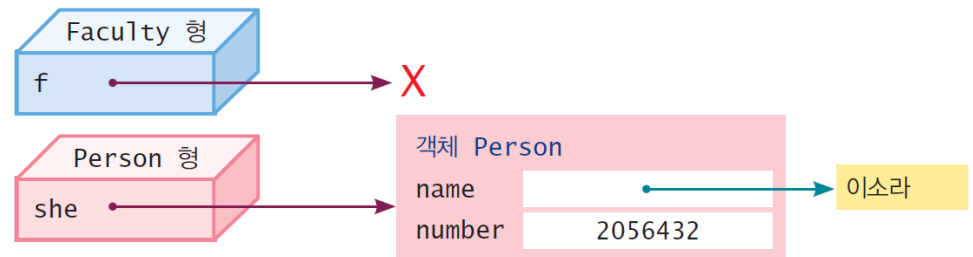
# 다운 캐스팅의 실행 오류

- 컴파일 시간
  - 상속 관계만 성립하면 다운 캐스팅은 가능
- 실행 시간
  - 실제 객체가 할당되지 않았다면 실행 시간에 오류가 발생

## DownCasting.java

```
01 package inheritance.typecast;
02
03 public class DownCasting {
04     public static void main(String[] args) {
05         Person she = new Person("이소라", 2056432);
06         System.out.println(she.name + " " + she.number);
07         //Faculty f = she;           //컴파일 오류
08         //Faculty f1 = (Faculty) she; //실행 오류
09
10         Person p = new Staff("김상기", 1187543, "강서대학교", 3456);
11         //Staff s = p;               //컴파일 오류
12         Staff s = (Staff) p;
13         s.division = "교학처";
14         System.out.print(p.name + " " + p.number + " ");
15         System.out.print(s.univ + " " + s.number + " ");
16         System.out.println(s.division);
17     }
18 }
```

```
Person she = new Person("이소라", 2056432);
//Faculty f = she;           //컴파일 오류
//Faculty f1 = (Faculty) she; //실행 오류
```



## 결과

이소라 2056432

김상기 1187543 강서대학교 3456 교학처

# 객체 확인 연산자 instanceof

## InstanceOf.java

```
01 package inheritance.typecast;
02
03 public class Instanceof {
04     public static void main(String[] args) {
05         Person she = new Person("이소라", 2056432);
06         if (she instanceof Staff) {
07             Staff st1 = (Staff) she;
08         } else {
09             System.out.println("she는 Staff 객체가 아닙니다.");
10         }
11
12         Person p = new Staff("김상기", 1187543, "강서대학교", 3456);
13         if (p instanceof Staff) {
14             System.out.println("p는 Staff 객체입니다.");
15             Staff st2 = (Staff) p;
16         }
17     }
18 }
```

```
Person she = new Person("이소라", 2056432);
if (she instanceof Staff) {
    Staff st1 = (Staff) she;
} else {
    System.out.print("she는 Staff 객체가 아닙니다. ");
}
```

사용법: 객체변수 instanceof 클래스이름

상속관계가 없는데,  
instanceof 연산자를 사용할  
경우 컴파일 에러

**결과** she는 Staff 객체가 아닙니다.  
p는 Staff 객체입니다.

# 다형성(polymorphism)

- 하나의 객체가 여러 개의 자료 타입을 가질 수 있는 것
- 객체들의 타입이 다르면 똑같은 메시지가 전달되더라도 서로 다른 동작을 하도록 하는데 사용됨
- 메소드의 매개 변수로 수퍼 클래스 참조 변수를 이용한다.

-> 다형성을 이용하는 전형적인 방법

매개변수를 공으로 설정  
공을 상속한 야구공, 축구공,  
골프공 모두 인자로 받을 수  
있음.



다형성을 이용하면  
뭐든지 받을 수 있지

매개변수를 동물로 둔 경우  
동물을 상속받은 고양이,  
강아지 모두 speak 메서드의  
인자로 받을 수 있다.



그림12-5. 다형성의 개념



# 예제

```
01 class Shape {
02     protected int x, y;
03     public void draw() {
04         System.out.println("Shape Draw");
05     }
06 }
07
08 class Rectangle extends Shape {
09     private int width, height;
10     public void draw() {
11         System.out.println("Rectangle Draw");
12     }
13 }
14
15 class Triangle extends Shape {
16     private int base, height;
17     public void draw() {
18         System.out.println("Triangle Draw");
19     }
20 }
21
```

각 도형들은 2차원 공간에서 도형의 위치를 나타내는 기준 점 (x, y)을 가진다. 이것은 모든 도형에 공통적인 속성이므로 부모 클래스인 Shape에 저장한다. 또한 각 도형들을 화면에 그리는 멤버 함수 draw()가 필요하다. 이것도 모든 도형에 필요한 기능이므로 부모 클래스 Shape에 정의하도록 하자. 하지만 아직 특정한 도형이 결정되지 않았으므로 draw()에서 하는 일은 없다.

이어서 Shape에서 상속받아서 사각형을 나타내는 클래스 Rectangle을 정의하여 보자. Rectangle은 추가적으로 width와 height 변수를 가진다. Shape 클래스의 draw()를 사각형을 그리도록 재정의한다. 물론 실제 그래픽은 아직까지 사용할 수 없으므로 화면에 사각형을 그린다는 메시지만을 출력한다.

서브 클래스인 Triangle을 Shape 클래스에서 상속받아 만든다.



# 예제

```
22 class Circle extends Shape {
23     private int radius;
24     public void draw() {
25         System.out.println("Circle Draw");
26     }
27 }
```

서브 클래스인 Rectangle을 Shape 클래스에서 상속받아 만든다.

```
28
29 public class ShapeTest {
30     private static Shape arrayOfShapes[];
31     public static void main(String arg[]) {
32         init();
33         drawAll();
34     }
```

클래스 Shape의 배열 arrayOfShapes[]를 선언한다.

```
35
36     public static void init() {
37         arrayOfShapes = new Shape[3];
38         arrayOfShapes[0] = new Rectangle();
39         arrayOfShapes[1] = new Triangle();
40         arrayOfShapes[2] = new Circle();
41     }
42 }
```

배열 arrayOfShapes의 각 원소에 객체를 만들어 대입한다. 다형성에 의하여 Shape 객체 배열에 모든 타입의 객체를 저장할 수 있다.

# 예제

```
43     public static void drawAll() {  
44         for (int i = 0; i < arrayOfShapes.length i++) {  
45             arrayOfShapes[i].draw();  
46         }  
47     }  
48 }
```

배열 arrayOfShapes[] 길이만큼 루프를 돌면서 각 배열 원소를 사용하여 draw() 메소드를 호출해본다. 어떤 draw()가 호출될까? 각 원소가 실제로 가리키고 있는 객체에 따라 서로 다른 draw()가 호출된다.

## 실행결과

Rectangle Draw  
Triangle Draw  
Circle Draw

# 다형성의 장점

- 만약 새로운 도형 클래스를 작성하여 추가한다고 해보자.

```
class Cylinder extends Shape {  
    public void draw(){  
        System.out.println("Cylinder Draw");  
    }  
};
```

- drawAll() 메소드는 수정할 필요가 없다.

```
public static void drawAll() {  
    for (int i = 0; i < arrayOfShapes.length; i++) {  
        arrayOfShapes[i].draw();  
    }  
}
```

# 동적 바인딩

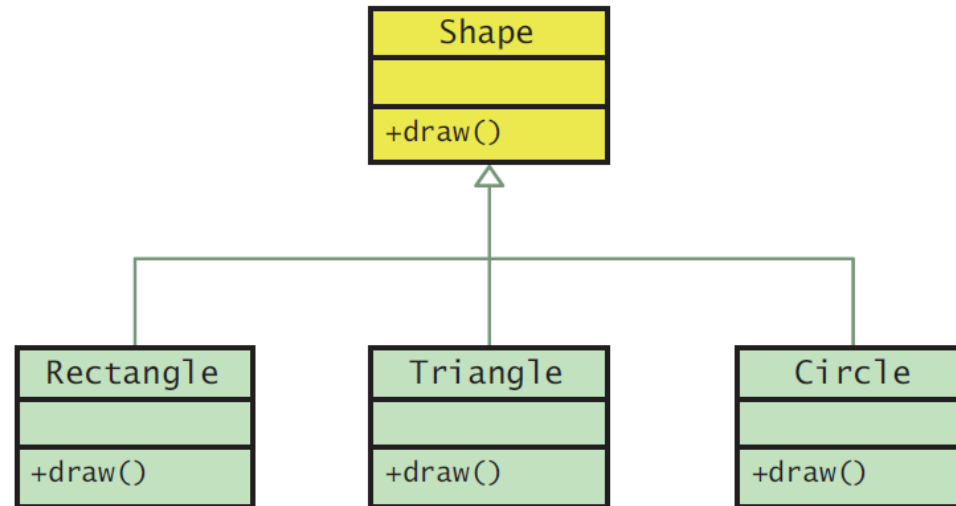


그림12-9. 도형의 UML

```
Shape s = new Rectangle(); // OK!
s.draw();                  // 어떤 draw()가 호출되는가?
```

Shape의 `draw()`가 호출되는 것이 아니라 **Rectangle**의 `draw()`가 호출된다. `s`의 타입은 **Shape**이지만 `s`가 실제로 가리키고 있는 객체의 타입이 **Rectangle**이기 때문이다.

# 중간 점검

1. 수퍼 클래스 참조 변수는 서브 클래스 객체를 참조할 수 있는가? 역은 성립하는가?
2. `instanceof` 연산자가 하는 연산은 무엇인가?
3. 다형성은 어떤 경우에 유용한가?
4. 어떤 타입의 객체라도 받을 수도 있게 하려면 메소드의 매개변수를 어떤 타입으로 정의하는 것이 좋은가?

- 과일, 사과, 배, 포도를 표현한 클래스를 만들고, 이들 간의 관계를 고려하여 하나의 클래스를 추상 클래스로 만들어 메소드 print()를 구현하고

```
Fruit f[] = {new Grape(), new Apple(), new Pear()}  
for(Fruit fr:f) fr.print();
```

의 결과가

나는 포도

나는 사과

나는 배

가 되도록 클래스를 작성하라.