

JAVA 상속





1 상속

1. 상속의 개요

2. 상속의 사용

3. 접근 지정자

4. 메소드의 재정의

5. 메소드와 생성자



상속의 개요

```
package exam_inheritance;

public class Employee{
    private String name;
    private int age;
    private String dept;

    public String getName() {
        return name;
    }
    public void setName
    (String name) {
        this.name = name;
    }
}
```

```
    public int getAge() {
        return age;
    }
    public void setAge
    (int age) {
        this.age = age;
    }

    public String getDept() {
        return dept;
    }
    public void setDept
    (String dept) {
        this.dept = dept;
    }
}
```




상속의 개요

```
package exam_inheritance;
```

```
public class Professor{  
    private String name;  
    private int age;  
    private String subject;  
  
    public String getName() {  
        return name;  
    }  
    public void setName  
    (String name) {  
        this.name = name;  
    }  
}
```

```
    public int getAge() {  
        return age;  
    }
```

```
    public void setAge  
    (int age) {  
        this.age = age;  
    }
```

```
    public String getSubject() {  
        return subject;  
    }
```

```
    public void setSubject  
    (String subject) {  
        this.subject = subject;  
    }
```

```
}
```



상속의 개요

```
package exam_inheritance;

public class Student {
    private String name;
    private int age;
    private String major;

    public String getName() {
        return name;
    }
    public void setName
    (String name) {
        this.name = name;
    }
}
```

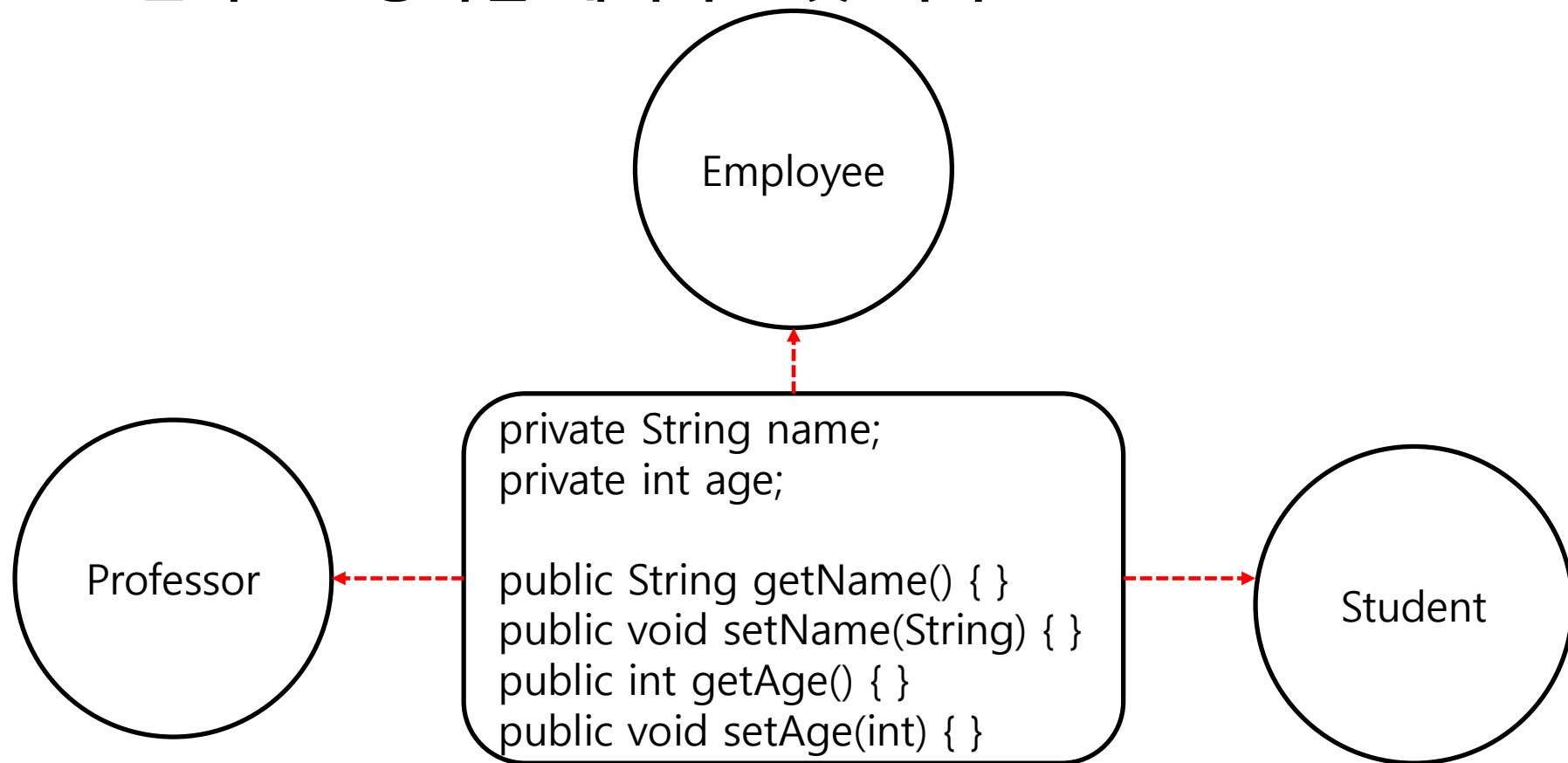
```
    public int getAge() {
        return age;
    }
    public void setAge
    (int age) {
        this.age = age;
    }

    public String getMajor() {
        return major;
    }

    public void setMajor
    (String major) {
        this.major = major;
    }
}
```

상속의 개요

객체지향 언어의 장점은 유지 보수성이 높은 시스템을 개발할 수 있다는 것이다. 유지보수성을 높이는 가장 기본적인 원칙은 "중복을 제거하는 것"이다.





상속의 개요

[중복 제거 조건1] 공통 멤버를 가지는 클래스들 간의 공통 점이 있는가?

Employee, Professor, Student 클래스의 공통점은 '사람 (Person)' 이라는 것이다.

[중복 제거 조건2] is-a 관계가 성립하는가?

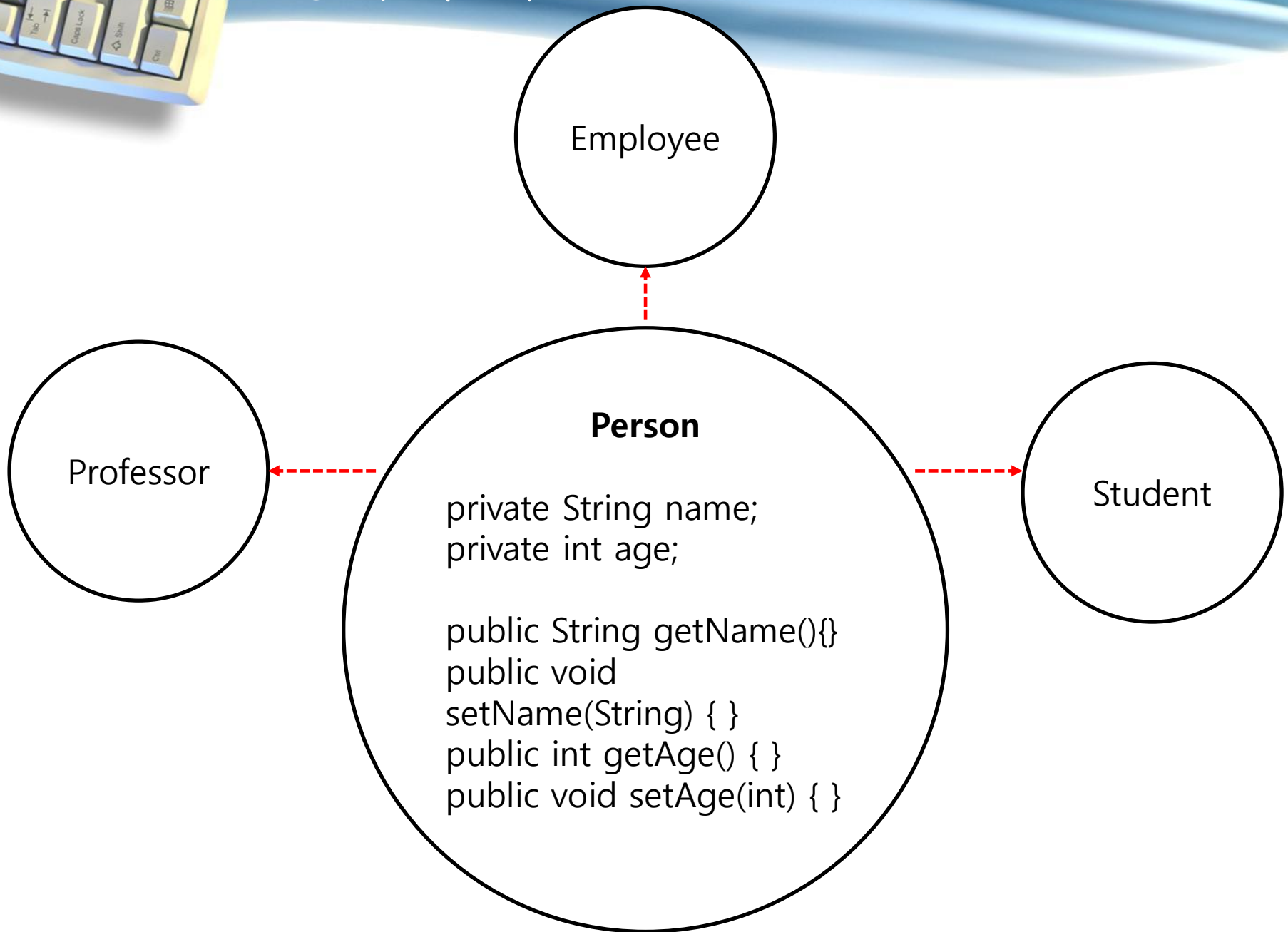
Employee is a Person.

Professor is a Person.

Student is a Person.

Person 클래스를 생성하고 Person에 각 클래스가 가지는 공통된 멤버를 선언하고 다음, Employee, Professor, Student 클래스에서 Person 클래스를 가져다 사용한다.

상속의 개요





상속의 개요

1) 상속의 개념

상속은 기존에 존재하는 유사한 클래스로부터 속성과 동작을 이어받고 자신이 필요한 기능을 추가하는 기법이다.

상속은 이미 작성된 검증된 소프트웨어를 재사용할 수 있어서 신뢰성 있는 소프트웨어를 손쉽게 개발, 유지 보수할 수 있게 해주는 중요한 기술이다.

또한 상속을 이용하면 코드의 중복을 줄일 수 있어서 전체적으로 코드의 크기가 작아진다.



상속의 개요

2) 상속

상속(inheritance)이란 이미 존재하는 클래스로부터 멤버들을 물려받는 것이다.

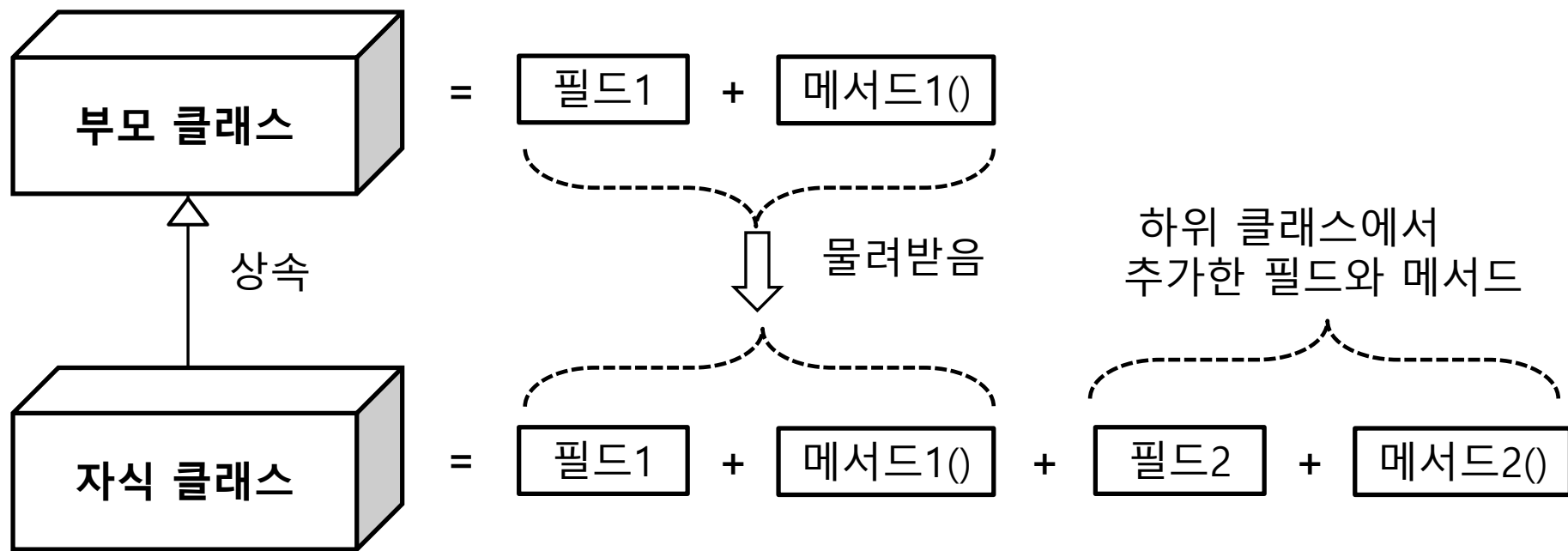
이미 존재하는 클래스(상속하는 클래스)를 슈퍼 클래스(super class), 부모 클래스(parent class), 베이스 클래스(base class)라고 한다.

상속을 받는 클래스를 서브 클래스(sub class), 자식 클래스(child class), 파생 클래스(derived class)라고 한다.

자바에서의 상속은 클래스 정의 다음에 **extends**를 써주고 슈퍼 클래스 이름을 적어주면 된다.

상속의 개요

- 자식(하위, 파생) 클래스가 부모(상위) 클래스의 멤버를 물려받는 것
- 자식이 부모를 선택해 물려받음
- 상속 대상: 부모의 필드와 메소드





상속의 개요

– 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 메서드도 제외



상속의 개요

- **extends**(사전적 의미 : 확장하다) 키워드
: 자식 클래스가 상속할 부모 클래스를 지정하는 키워드

슈퍼 클래스 == 부모 클래스

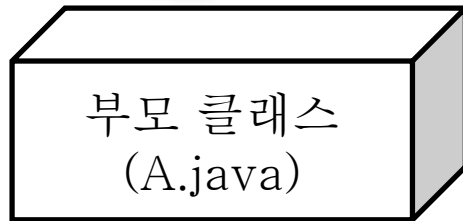
```
class SubClass extends SuperClass {  
    ... // 추가된 필드와 메소드 정의  
}
```

상속을 의미. 슈퍼 클래스를 확장하여
서브 클래스를 작성한다는 의미.

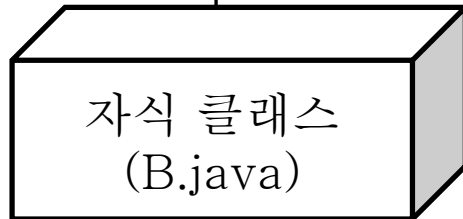
서브 클래스 == 자식 클래스

클래스 상속(extends)

❖ extends 키워드



상속



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



A를 상속

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

■ 자바는 단일 상속 - 부모 클래스 나열 불가

```
class 자식 클래스 extends 부모 클래스1, 부모 클래스2 {  
  
}
```



상속의 개요

상속 관계를 자바 코드로 구현하면 다음과 같다.

```
class Person { ... }  
class Employee extends Person { ... }  
class Professor extends Person { ... }  
class Student extends Person { ... }
```



상속의 개요

```
package exam_inheritance;

public class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName
(String name) {
        this.name = name;
    }
}
```

```
    public int getAge() {
        return age;
    }

    public void setAge
(int age) {
        this.age = age;
    }

    public String toString() {
        return name + ":" + age;
    }
}
```




상속의 개요

```
package exam_inheritance;
```

```
public class Employee extends Person {
```

```
    private String dept;
```

```
    public String getDept() {
```

```
        return dept;
```

```
    }
```

```
    public void setDept(String dept) {
```

```
        this.dept = dept;
```

```
    }
```

```
}
```



상속의 개요

```
package exam_inheritance;
```

```
public class Professor extends Person {  
    private String subject;  
  
    public String getSubject() {  
        return subject;  
    }  
  
    public void setSubject(String subject) {  
        this.subject = subject;  
    }  
}
```



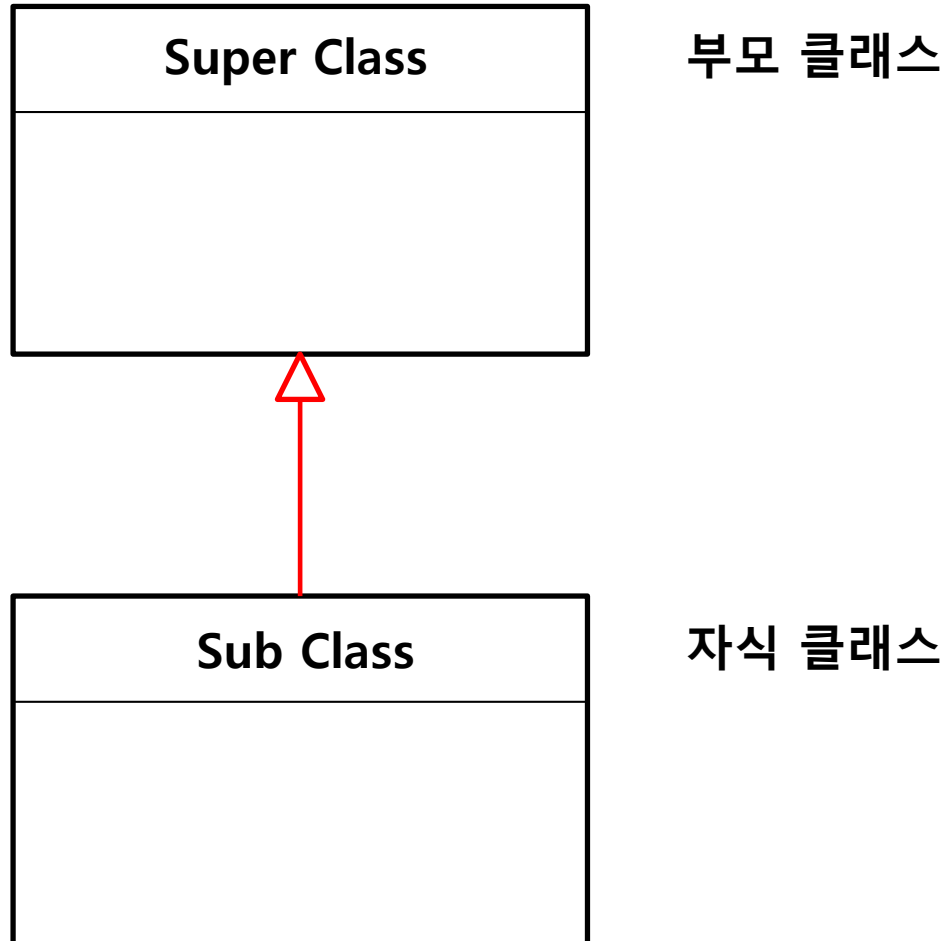
상속의 개요

```
package exam_inheritance;
```

```
public class Student extends Person {  
    private String major;  
  
    public String getMajor() {  
        return major;  
    }  
  
    public void setMajor(String major) {  
        this.major = major;  
    }  
  
}
```

상속의 개요

- 상위 클래스와 하위 클래스 간의 상속 관계를 나타내는 UML 표기법





상속의 개요

상속을 나타낼 때 extends(확장)라는 용어를 사용하는 이
유도 상속을 하게 되면 멤버가 증가하기 때문이다. 많이 등
장하는 수퍼 클래스와 서브 클래스의 예를 표로 정리하여
보면 다음과 같다. **대개 슈퍼 클래스는 추상적이고 서브 클
래스는 구체적이다.**

Animal(동물)	Dog(개), 고양이(Cat)
Vehicle(탈것)	Car(자동차), Bus(버스), Boat(보트), Motorcycle(오토 바이), Bicycle(자전거)
Student(학생)	GraduateStudent(대학원생), UnderGraduate(학부생)
Employee(직원)	Manager(관리자)
Shape(도형)	Rectangle(사각형), Triangle(삼각형), Circle(원)



상속의 개요

자동차를 예를 들어보면 스포츠카는 일반적인 자동차의 특징을 모두 가지고 있고 추가로 스포츠카에 필요한 장치가 추가되어 있다.

수퍼 클래스

```
class Car {  
}
```

```
class SportsCar extends Car {  
    ... // 추가된 필드와 메소드 정의  
}
```

서브 클래스

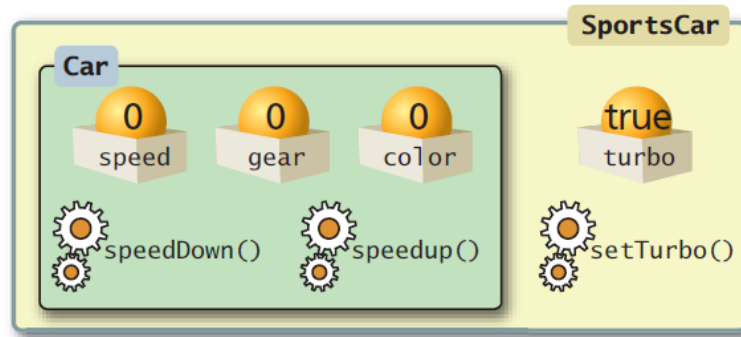


상속 사용








```
public class Car {  
    public int speed; // 속도  
    public int gear; // 기어  
    public String color; // 색상, 차  
  
    public void setGear(int newGear) {  
        gear = newGear;  
    }  
    public void speedUp(int increment)  
        speed += increment;  
    }  
    public void speedDown(int decrement) {  
        speed -= decrement;  
    }  
}
```

상속 사용

```
public class SportsCar extends Car {  
    public boolean turbo;  
  
    public void setTurbo(boolean newValue) {  
        turbo = newValue;  
    }  
}
```



Car =  speed +  gear +  color +  speedDown() +  speedUp()

SportCar =  speed +  gear +  color +  turbo +  speedDown() +  speedUp() +  setTurbo()



상속 사용

```
public class CarExtendsTest {  
    public static void main(String[] args) {  
        SportsCar c = new SportsCar();  
  
        c.color = "Red";           // 슈퍼 클래스 필드 접근  
        c.setGear(3);              // 슈퍼 클래스 메소드 호출  
        c.speedUp(100);            // 슈퍼 클래스 메소드 호출  
        c.speedDown(30);          // 슈퍼 클래스 메소드 호출  
        c.setTurbo(true);         // 자체 메소드 호출  
    }  
}
```

클래스 상속 만들기 - Point와 ColorPoint 클래스

(x, y)의 한 점을 표현하는 Point 클래스와 이를 상속받아 색을 가진 점을 표현하는 ColorPoint 클래스를 만들고 활용해보자.

```
class Point {  
    private int x, y;          // 한 점을 구성하는 x, y 좌표  
    public void set(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void showPoint() { // 점의 좌표 출력  
        System.out.println("(" + x + "," + y + ")");  
    }  
}
```

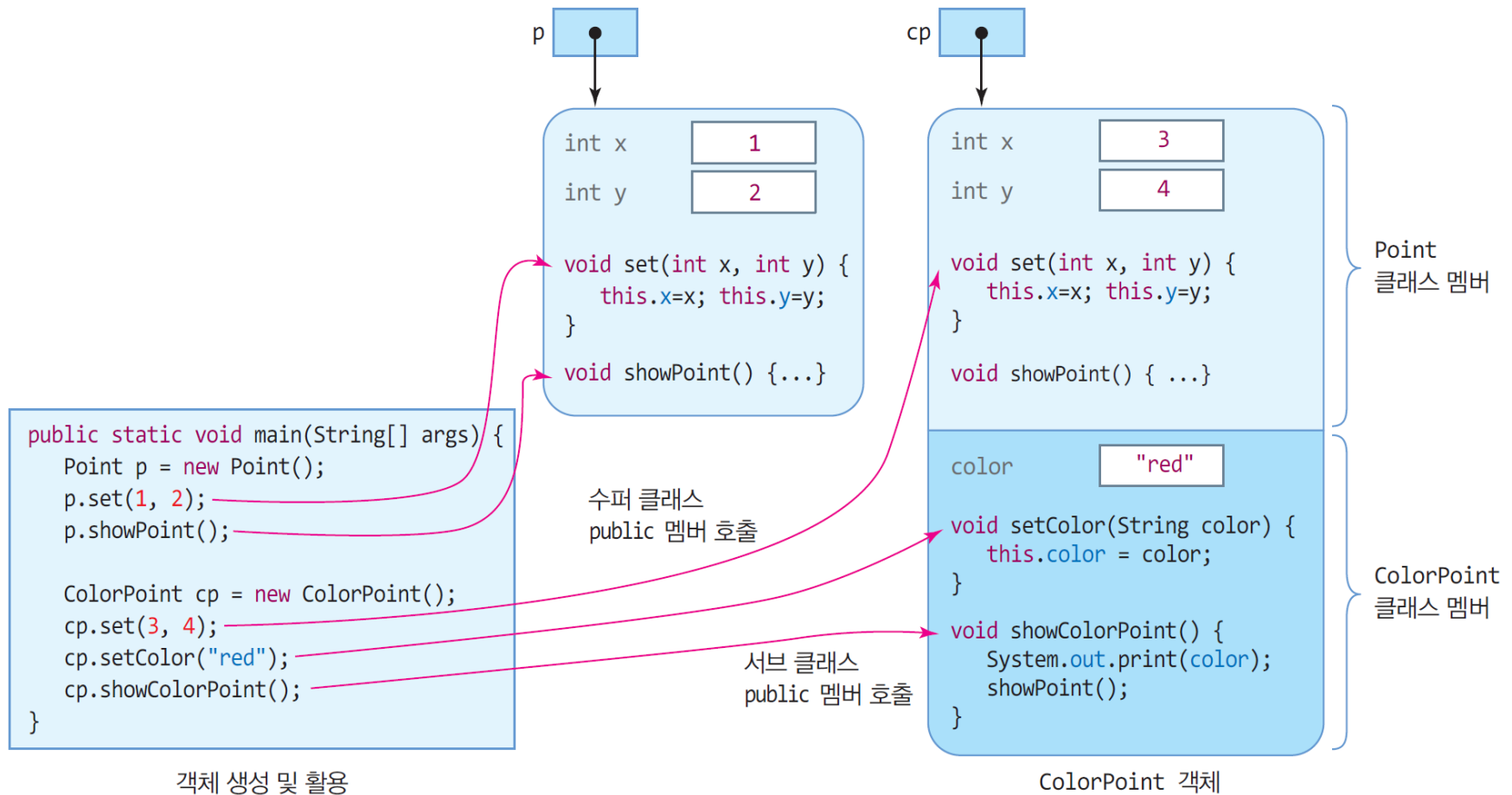
// Point를 상속받은 ColorPoint 선언

```
class ColorPoint extends Point {  
    private String color; // 점의 색  
    public void setColor(String color) {  
        this.color = color;  
    }  
    public void showColorPoint() { // 컬러 점의 좌표 출력  
        System.out.print(color);  
        showPoint(); // Point 클래스의 showPoint() 호출  
    }  
}
```

```
public class ColorPointEx {  
    public static void main(String [] args) {  
        Point p = new Point(); // Point 객체 생성  
        p.set(1, 2); // Point 클래스의 set() 호출  
        p.showPoint();  
  
        ColorPoint cp = new ColorPoint(); // ColorPoint 객체  
        cp.set(3, 4); // Point의 set() 호출  
        cp.setColor("red"); // ColorPoint의 setColor() 호출  
        cp.showColorPoint(); // 컬러와 좌표 출력  
    }  
}
```

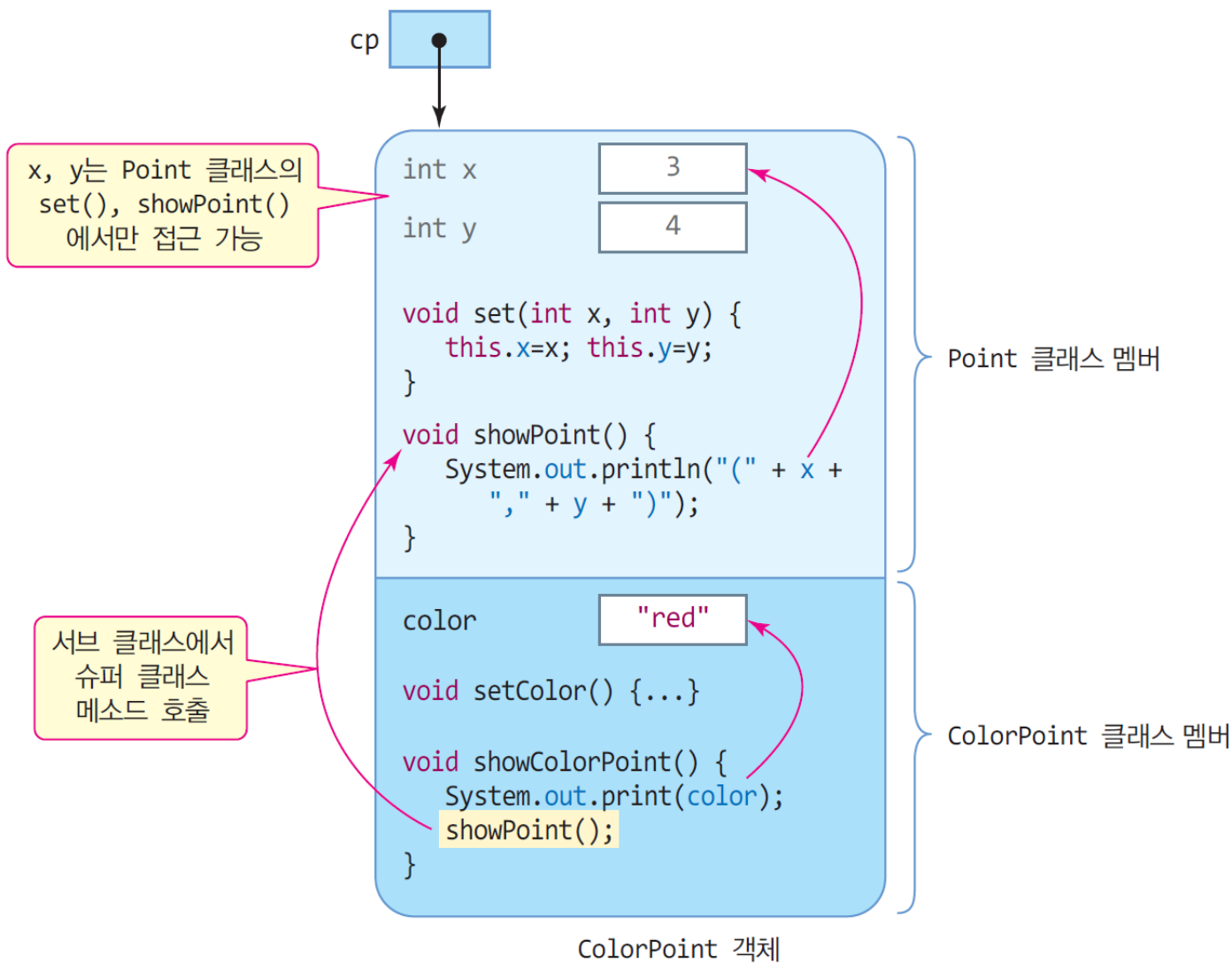
(1,2)
red(3,4)

객체 생성



* `new ColorPoint()`에 의해 생긴
서브 클래스 객체에 주목

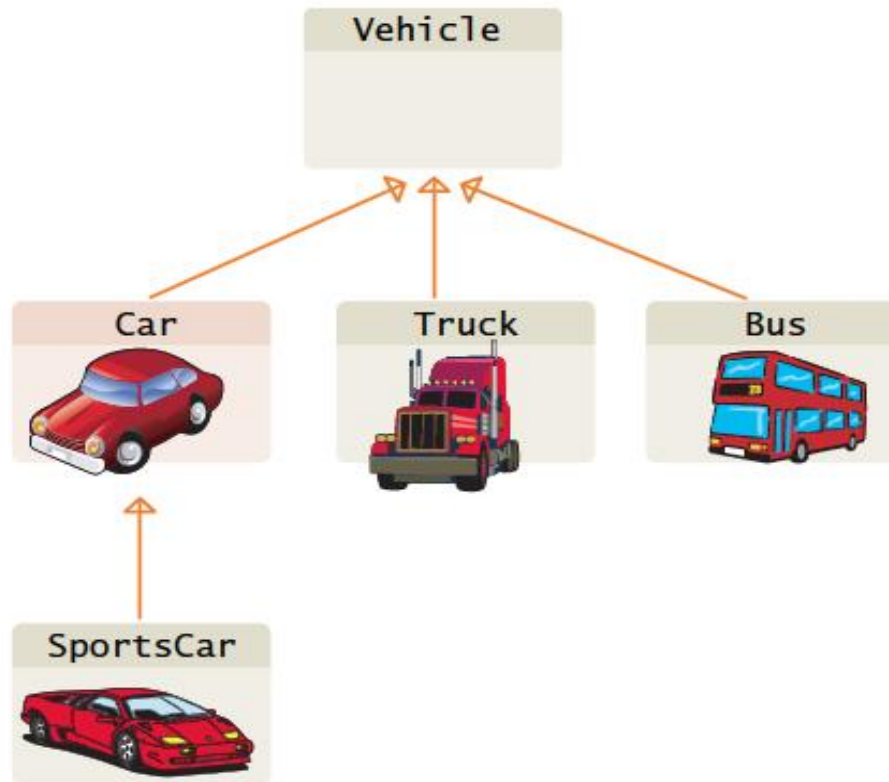
서브클래스에서 슈퍼 클래스의 멤버 접근



상속의 개요

3) 상속 계층 구조

상속 관계는 일반적으로 트리 모양의 계층 구조를 형성한다. 예를 들어서 일반적인 운송 수단을 의미하는 Vehicle 클래스를 시작으로 상속 계층 구조를 작성하여 보면 다음과 같다.





상속의 개요

클래스로 정의하면 다음과 같다.

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

슈퍼 클래스를 변경하면 서브 클래스에 영향을 준다.

하지만 서브 클래스를 변경하는 경우에는 슈퍼 클래스는
영향이 없다. 일반적으로 슈퍼 클래스는 서브 클래스들의
공통 부분이 된다.



상속의 개요

4) 상속은 중복을 줄인다.

상속을 사용하면 중복되는 코드를 줄일 수 있다.

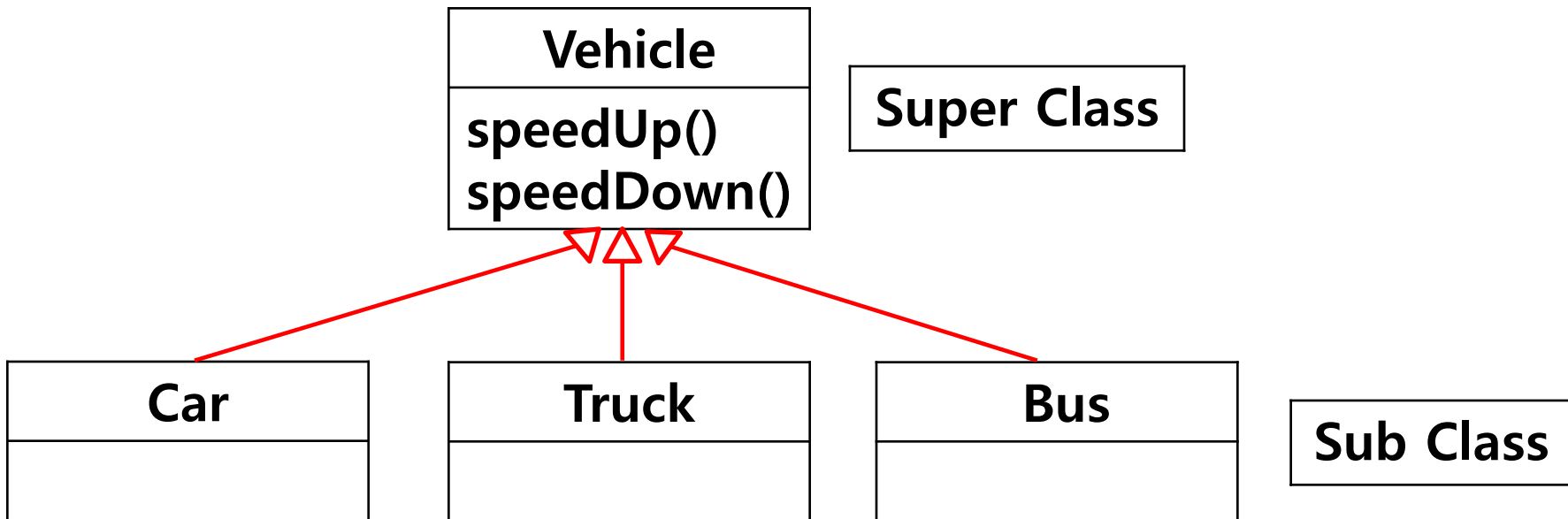
class Car
speedUp() speedDown()

class Truck
speedUp() speedDown()

class Bus
speedUp() speedDown()

상속의 개요

만약 여러 클래스에 공통적인 특징을 새로운 클래스 Vehicle로 만들고 Vehicle을 상속받는다면 중복되는 부분을 최소화할 수 있다.





상속의 개요(is-a관계)

5) 상속은 is-a관계

상속에서 서브 클래스와 수퍼 클래스는 "~은 ~이다"와 같은 is-a관계가 있다. 따라서 상속의 계층 구조를 올바르게 설계되었는지를 알려면 is-a관계가 성립하는지를 생각해 보면 된다.

- 자동차는 탈것이다. (Car is a Vehicle)
- 무선전화기는 일종의 전화기이다.
- 사자, 개, 고양이는 동물이다.
- 노트북은 일종의 컴퓨터이다

상속의 개요(has-a관계)

만약 "~은 ~을 가지고 있다"와 같은 has-a(포함)관계가 성립되면 이 관계는 상속으로 모델링을 하면 안된다.

- 도서관은 책을 가지고 있다. (Library has a book)
- 거실은 소파를 가지고 있다.

has-a관계가 성립되는 경우에는 상속을 이용하는 것이 아니라 하나의 클래스 안에 다른 클래스의 객체를 포함시키면 된다.

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

```
class Line {  
    private Point p1;  
    private Point p2;  
}
```





상속과 접근 지정자

자바의 접근 지정자 4 가지

- public, protected, 디폴트, private
 - 상속 관계에서 주의할 접근 지정자는 private와 protected

슈퍼 클래스의 private 멤버

- 슈퍼 클래스의 private 멤버는 다른 모든 클래스에 접근 불허
- 클래스내의 멤버들에게만 접근 허용

슈퍼 클래스의 디폴트 멤버

- 슈퍼 클래스의 디폴트 멤버는 패키지내 모든 클래스에 접근 허용

슈퍼 클래스의 public 멤버

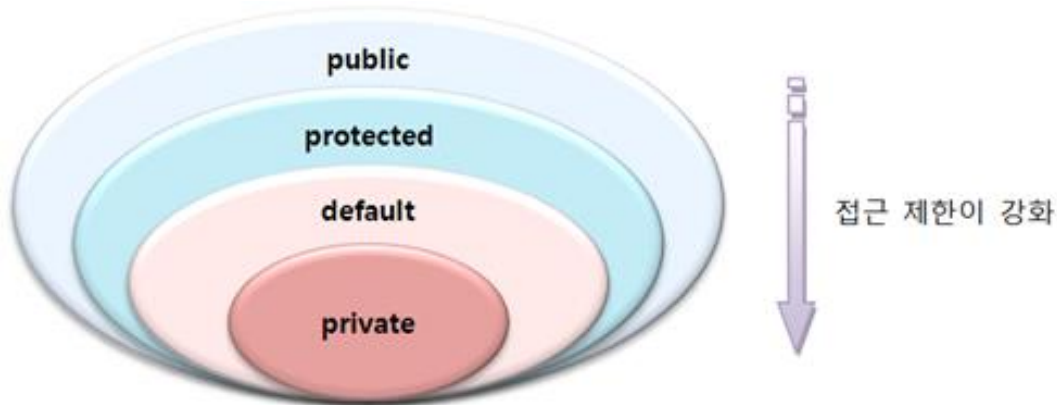
- 슈퍼 클래스의 public 멤버는 다른 모든 클래스에 접근 허용

슈퍼 클래스의 protected 멤버

- 같은 패키지 내의 모든 클래스 접근 허용
- 다른 패키지에 있어도 서브 클래스는 슈퍼 클래스의 protected 멤버 접근 가능

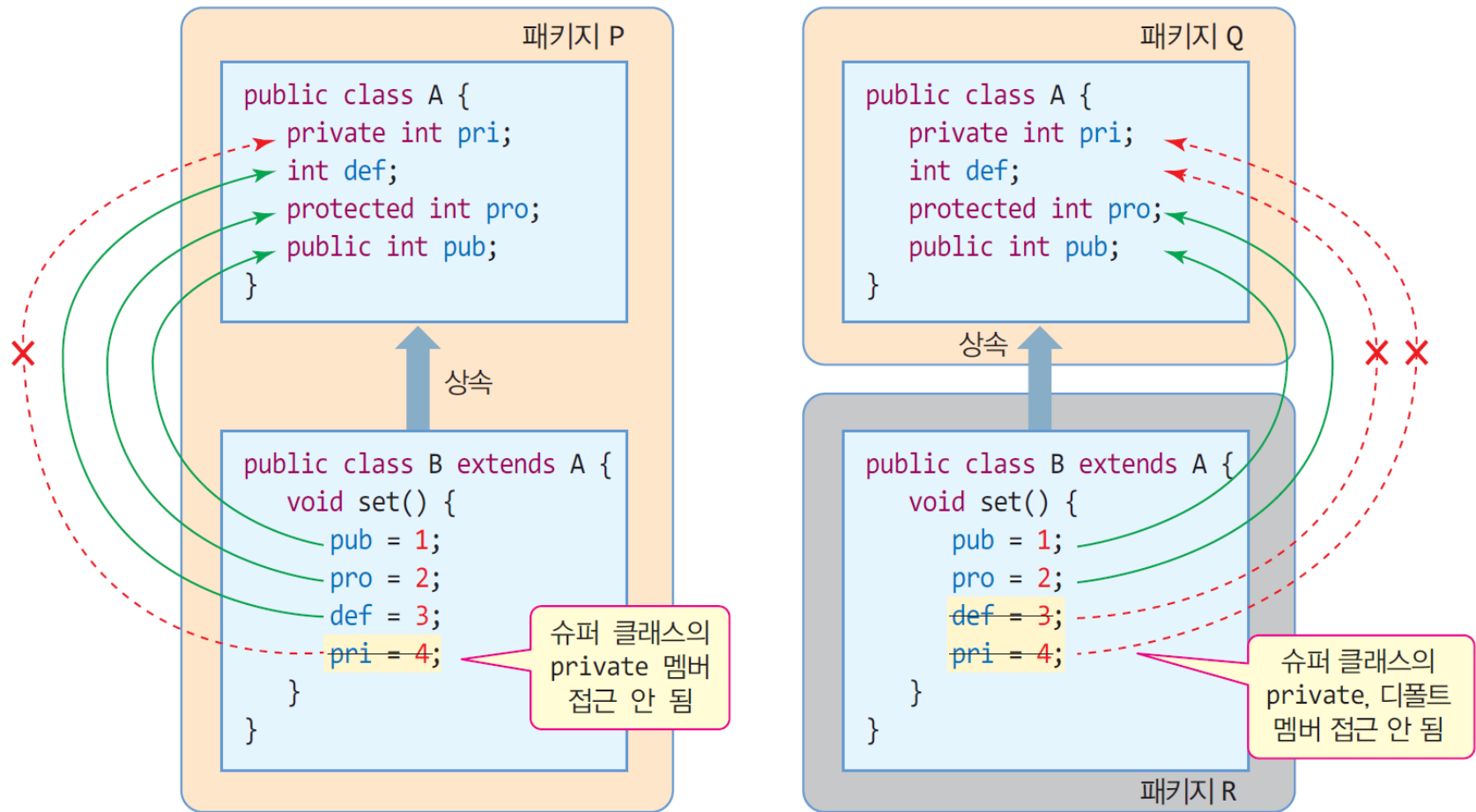
접근 제어자(제한자)

- 상속과 관련된 접근 제한자(protected)
같은 패키지: default와 동일
다른 패키지: 자식 클래스만 접근 허용



접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메서드	없음.
protected	필드, 생성자, 메서드	자식 클래스가 아닌 다른 패키지 에 소속된 클래스
default	클래스, 필드, 생성자, 메서드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메서드	모든 외부 클래스

슈퍼 클래스의 멤버에 대한 서브 클래스의 접근



(a) 슈퍼 클래스와 서브 클래스가 동일한 패키지에 있는 경우

(b) 슈퍼 클래스와 서브 클래스가 서로 다른 패키지에 있는 경우



접근 제어자(제한자)

```
package exam_protected.package1;
```

```
public class A {  
    protected String field;  
    protected A() {System.out.println("A클래스 생성자"); }  
    protected void method() {  
        System.out.println("A클래스의 메서드"); }  
}
```

```
package exam_protected.package1;
```

```
public class B {  
    public void method() {  
        A a = new A();  
        a.field = "value";  
        a.method();  
    }  
}
```

접근 제어자(제한자)

```
package exam_protected.package2;
```

```
import exam_protected.package1.A;
```

```
public class C {  
    public void method() {  
        A a = new A();  
        a.field = "value";  
        a.method();  
    }  
}
```

```
public class C {  
    public void method() {  
        A a = new A();  
        a.field = "value";  
        a.method();  
    }  
}
```



접근 제어자(제한자)

```
package exam_protected.package2;

import exam_protected.package1.A;

public class D extends A {
    public D() {
        super();
        field = "value";
        method();
    }
}
```



상속 관계에 있는 클래스 간 멤버 접근

클래스 **Person**을 아래와 같은 멤버 필드를 갖도록 선언하고 클래스 **Student**는 클래스 **Person**을 상속받아 각 멤버 필드에 값을 저장하시오. 이 예제에서 **Person** 클래스의 **private** 필드인 **weight**는 **Student** 클래스에서는 접근이 불가능하여 슈퍼 클래스인 **Person**의 **getXXX**, **setXXX** 메소드를 통해서만 조작이 가능하다.

- **private int weight;**
- **int age;**
- **protected int height;**
- **public String name;**

```
class Person {  
    private int weight;  
    int age;  
    protected int height;  
    public String name;  
  
    public void setWeight(int weight) {  
        this.weight = weight;  
    }  
    public int getWeight() {  
        return weight;  
    }  
}
```

```
class Student extends Person {  
    public void set() {  
        age = 30; // 슈퍼 클래스의 디폴트 멤버 접근 가능  
        name = "홍길동"; // 슈퍼 클래스의 public 멤버 접근 가능  
        height = 175; // 슈퍼 클래스의 protected 멤버 접근 가능  
        // weight = 99; // 오류. 슈퍼 클래스의 private 접근 불가  
        setWeight(99); // private 멤버 weight은 setWeight()으로 간접 접근  
    }  
}
```

```
public class InheritanceEx {  
    public static void main(String[] args) {  
        Student s = new Student();  
        s.set();  
    }  
}
```



super 참조변수

하위 클래스에서 상위 클래스로부터 상속받은 멤버를 참조할 때 사용하는 참조변수이다. 객체 자신을 참조하는 참조변수인 **this** 처럼 모든 하위 클래스의 객체에는 상위 클래스를 참조하는 참조변수 **super**가 있다.

super.속성명;	상위 클래스의 클래스 변수나 접근 제어에 의해 접근이 가능한 속성에 접근한다.
super.메서드();	상위 클래스의 메서드에 접근한다
super();	상위 클래스의 생성자에 접근한다.

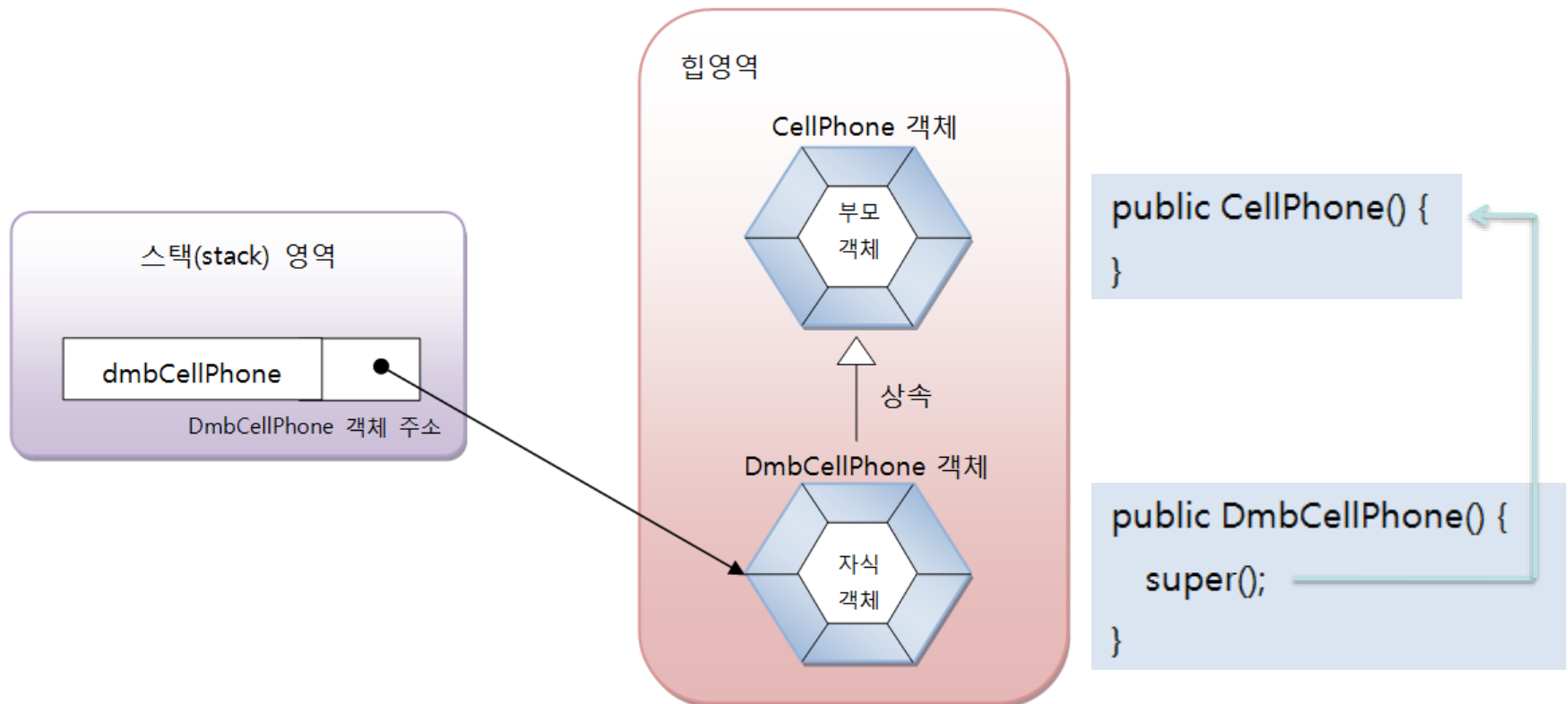
부모 생성자 호출(super(...))

❖ 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();
```





서브 클래스/슈퍼 클래스의 생성자 호출 및 실행

질문 1 서브 클래스 객체가 생성될 때 서브 클래스의 생성자와 슈퍼 클래스의 생성자가 모두 실행되는가? 아니면 서브 클래스의 생성자만 실행되는가?

답 둘 다 실행된다. 서브 클래스의 객체가 생성되면 이 객체 속에 서브 클래스와 멤버와 슈퍼 클래스의 멤버가 모두 들어 있다. 생성자의 목적은 객체 초기화에 있으므로, 서브 클래스의 생성자는 생성된 객체 속에 들어 있는 서브 클래스의 멤버 초기화나 필요한 초기화를 수행하고, 슈퍼 클래스의 생성자는 생성된 객체 속에 있는 슈퍼 클래스의 멤버 초기화나 필요한 초기화를 각각 수행한다.

질문 1 서브 클래스의 생성자와 슈퍼 클래스의 생성자 중 누가 먼저 실행되는가?

답 슈퍼 클래스의 생성자가 먼저 실행된 후 서브 클래스의 생성자가 실행된다.

new에 의해 서브 클래스의 객체가 생성될 때

- 슈퍼클래스 생성자와 서브 클래스 생성자 모두 실행됨
- 호출 순서
 - 서브 클래스의 생성자가 먼저 호출, 서브 클래스의 생성자는 실행 전 슈퍼 클래스 생성자 호출
- 실행 순서
 - 슈퍼 클래스의 생성자가 먼저 실행된 후 서브 클래스의 생성자 실행

상속과 생성자

서브 클래스의 객체가 생성될 때, 서브 클래스의 생성자만 호출될까? 아니면 수퍼 클래스의 생성자도 호출되는가?

```
class Shape {  
    ② public Shape() {  
        } ③  
}  
class Rectangle extends Shape {  
    public Rectangle() {  
        } ④  
}  
① class RectangleTest {  
    public static void main(String arg[]) {  
        Rectangle r = new Rectangle();  
    }  
}
```

The diagram illustrates the sequence of constructor calls during the execution of the provided Java code. Red dashed arrows and numbered circles (1-4) trace the execution flow:

- ①: The `main` method in `RectangleTest` is executed, leading to the creation of a new `Rectangle` object.
- ②: The `Rectangle` constructor is called, which then calls the `Shape` constructor.
- ③: The `Shape` constructor completes its execution and returns control to the `Rectangle` constructor.
- ④: The `Rectangle` constructor continues its execution after the `Shape` constructor has finished.



부모 생성자 호출(super(...))

❖ 명시적인 부모 생성자 호출

- 부모 객체 생성할 때, 부모 생성자 선택해 호출

```
자식 클래스(매개변수, 매개변수...){  
    super(매개변수, 매개변수);  
    ...  
}
```

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



상속과 생성자

1) 명시적인 호출

자바에서는 명시적으로 수퍼 클래스의 생성자를 호출할 수 있다. 이 때 `super`라는 키워드가 사용된다.



상속과 생성자

```
class SuperX{
    private int x;
    public SuperX(int x){
        this.x = x;
        System.out.println("x = " + x );
    }
}
class SubY extends SuperX{
    private int y;
    public SubY(int x, int y){
        super(x);
        this.y = y;
        System.out.println("y = " + y );
    }
}
public class Constructor{
    public static void main(String[] args){
        new SubY(100, 200);
    }
}
```




상속과 생성자

2) 묵시적인 호출

명시적으로 수퍼 클래스의 생성자를 호출해 주지 않아도 서브 클래스의 객체가 생성될 때 자동적으로 수퍼 클래스의 디폴트 생성자가 호출된다.



상속과 생성자

```
class SuperX{
    private int x;
    public SuperX(){
        x = 100;
        System.out.println("x = " + x );
    }
}
class SubY extends SuperX{
    private int y;
    public SubY(){
        y = 200;
        System.out.println("y = " + y );
    }
}
public class Constructor{
    public static void main(String[] args){
        new SubY();
    }
}
```

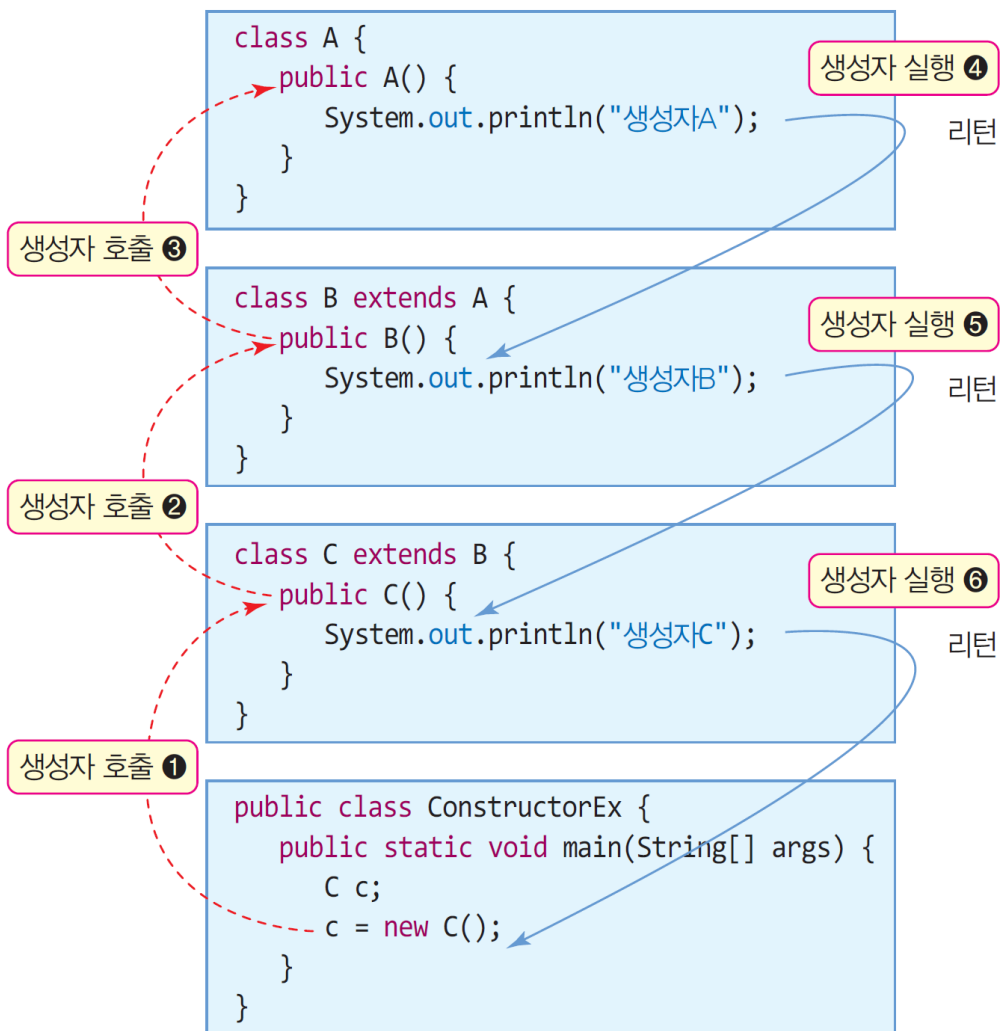


상속과 생성자

만약 수퍼 클래스에 매개 변수가 없는 생성자가 없는 경우에는 자바가 자동적으로 매개 변수가 없는 생성자(디폴트 생성자)를 추가하고 호출한다.

그러나 만약 생성자가 하나라도 정의되어 있는 경우에는 디폴트 생성자를 자동으로 추가하지 않는다.

슈퍼클래스와 서브 클래스의 생성자간의 호출 및 실행 관계



⇒ 실행 결과

생성자A
생성자B
생성자C

슈퍼 클래스의 기본 생성자가 자동 선택

서브 클래스의 생성자가
슈퍼 클래스의 생성자를
선택하지 않은 경우

컴파일러는
서브 클래스의 기본
생성자에 대해
자동으로 슈퍼 클래스의
기본 생성자와 짝을 맺음

```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        .....  
    }  
}
```

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
}
```

```
public class ConstructorEx2 {  
    public static void main(String[] args) {  
        B b;  
        b = new B();    // 생성자 호출  
    }  
}
```

⇒ 실행 결과

생성자A
생성자B

슈퍼 클래스에 기본 생성자가 없어 오류 난 경우

B()에 대한 짝,
A()를 찾을 수
없음

```
class A {  
    public A(int x) {  
        System.out.println("생성자A");  
    }  
}
```

```
class B extends A {  
    public B() { // 오류 발생 오류  
        System.out.println("생성자B");  
    }  
}
```

```
public class ConstructorEx2 {  
    public static void main(String[] args) {  
        B b;  
        b = new B();  
    }  
}
```

컴파일러에 의해 "Implicit super constructor A() is undefined. Must explicitly invoke another constructor" 오류 발생

서브 클래스에 매개변수를 가진 생성자

서브 클래스의 생성자가
슈퍼 클래스의 생성자를
선택하지 않은 경우

```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        System.out.println("매개변수생성자A");  
    }  
}
```

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
    public B(int x) {  
        System.out.println("매개변수생성자B");  
    }  
}
```

```
public class ConstructorEx3 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(5);  
    }  
}
```

⇒ 실행 결과

생성자A
매개변수생성자B

super()를 이용한 사례

```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        System.out.println("매개변수생성자A" + x);  
    }  
}
```

x에 5 전달

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
    public B(int x) {  
        super(x); // 첫 줄에 와야 함  
        System.out.println("매개변수생성자B" + x);  
    }  
}
```

x는 5

```
public class ConstructorEx4 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(5);  
    }  
}
```

⇒ 실행 결과

매개변수생성자A5
매개변수생성자B5

super()를 활용한 ColorPoint 작성

super()를 이용하여 ColorPoint 클래스의 생성자에서 슈퍼 클래스 Point의 생성자를 호출하는 예를 보인다.

```
class Point {
    private int x, y; // 한 점을 구성하는 x, y 좌표
    public Point() {
        this.x = this.y = 0;
    }
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
    public void showPoint() { // 점의 좌표 출력
        System.out.println("(" + x + "," + y + ")");
    }
}

class ColorPoint extends Point {
    private String color; // 점의 색
    public ColorPoint(int x, int y, String color) {
        super(x, y); // Point의 생성자 Point(x, y) 호출
        this.color = color;
    }
    public void showColorPoint() { // 컬러 점의 좌표 출력
        System.out.print(color);
        showPoint(); // Point 클래스의 showPoint() 호출
    }
}
```

x=5, y=6
전달

```
public class SuperEx {
    public static void main(String[] args) {
        ColorPoint cp = new ColorPoint(5, 6, "blue");
        cp.showColorPoint();
    }
}
```

blue(5,6)

x=5, y=6,
color = "blue" 전달



서브 클래스에서 슈퍼 클래스의 생성자 선택

상속 관계에서의 생성자

- 슈퍼 클래스와 서브 클래스 각각 각각 여러 생성자 작성 가능

서브 클래스 생성자 작성 원칙

- 서브 클래스 생성자에서 슈퍼 클래스 생성자 하나 선택

서브 클래스에서 슈퍼 클래스의 생성자를 선택하지 않는 경우

- 컴파일러가 자동으로 슈퍼 클래스의 기본 생성자 선택

서브 클래스에서 슈퍼 클래스의 생성자를 선택하는 방법

- `super()` 이용



상속과 생성자(정리)

- ① 생성자는 상속되지 않는다.
- ② 하위 클래스의 생성자를 호출하면 상위클래스로부터 상속받은 멤버의 생성과 초기화가 먼저 이루어지도록 상위 생성자를 호출한다. 이때 호출되는 상위 클래스의 생성자는 기본 생성자이다.
- ③ 상위 클래스의 생성자가 먼저 실행되고 하위 클래스의 생성자가 실행된다
- ④ 생성자 호출은 상위 클래스를 거슬러 올라가면서 연속적으로 이루어진다.



메소드 오버라이딩

1) 재정의(오버라이딩)의 개념

메소드 재정의(overriding method)란 서브 클래스가 필요에 따라 상속된 메소드를 다시 정의하여 사용하는 것을 의미한다. 이 기법을 사용하면 상속받은 메소드들을 자기 자신의 필요에 맞추어서 변경할 수 있다.



메소드 오버라이딩

Animal 클래스에서 sound()라는 메소드가 선언되어 있다고 가정하여 보자.

```
class Animal {  
    public void sound() {  
        System.out.println("소리를 낸다.");  
    }  
}
```



메소드 오버라이딩

Animal을 상속받아서 Dog클래스를 선언하였다면 sound() 메소드를 재정의한다.

```
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 오버라이딩

2) 메서드 오버라이딩(Method Overriding)

재정의는 메소드의 헤더는 그대로 두고 메소드의 몸체만을 교체하는 것이다. 따라서 메소드의 헤더 부분은 슈퍼 클래스의 헤더와 동일하여야 한다. 즉 슈퍼 클래스의 메소드와 동일한 시그니처를 가져야 한다(메소드의 이름, 반환형, 매개 변수의 개수와 데이터 타입이 일치하여야 한다.)

또한 슈퍼 클래스의 메소드를 재정의하려면 일반적으로 메소드가 public으로 선언되어 있어야 한다. private 메소드는 재정의할 수 없다. 또 접근 지정자의 경우 슈퍼 클래스의 메소드보다 더 좁은 범위로 변경할 수는 없다.



메소드 오버라이딩

- 정의

: 상속받은 자식 클래스에서 부모 클래스에 정의된 메소드를 재정의하는 것.

- 조건

- 메서드 오버라이딩은 **상속한 메서드의 본문만 변경할 수 있다.**
- 메서드 오버라이딩은 **상속한 메서드의 선언부를 변경할 수 없다.**
- 메서드 오버라이딩 할 때 접근 제한자는 부모의 메서드와 같거나 넓은 범위로만 변경할 수 있다.
 - ① public을 default나 private으로 수정 불가
 - ② 반대로 default는 public 으로 수정 가능
- 프로그램 실행 시 메서드 호출 순위는 오버라이딩한 메서드가 부모 클래스의 메서드보다 높다.



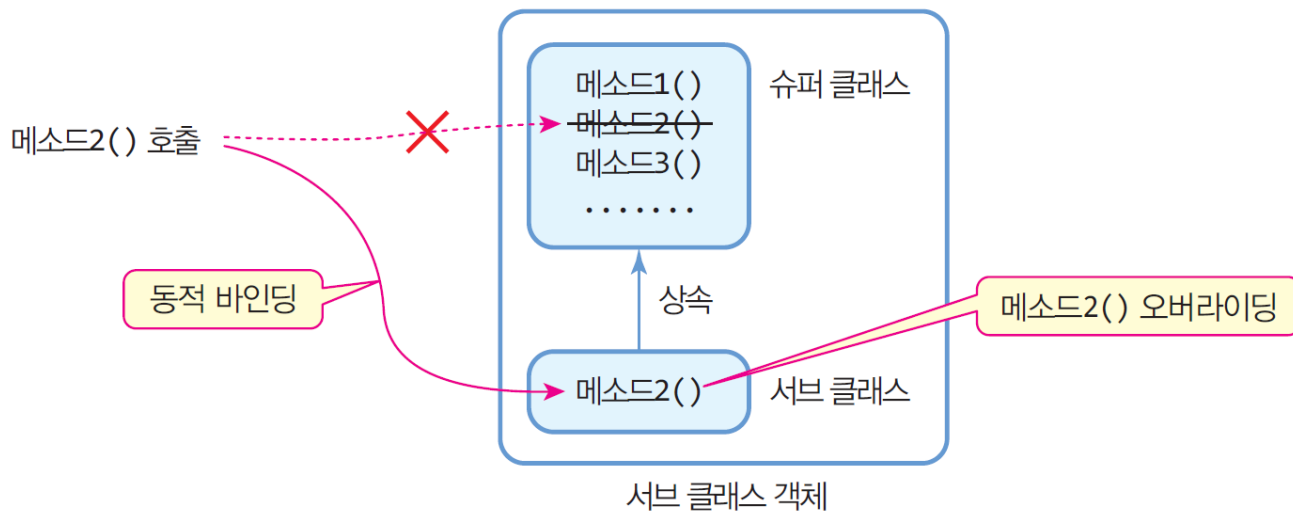
메소드 오버라이딩

- 오버라이딩의 특징은 다음과 같다
- ✓ 오버라이딩하고자 하는 메서드가 부모 클래스에 존재해야 한다.
- ✓ 부모 클래스에서 선언한 메서드가 자식 클래스에 사용할 때 메서드명이 반드시 같아야 한다
- ✓ 메서드의 파라미터 개수와 데이터형이 같아야 한다.
- ✓ 메서드의 리턴형이 같아야 한다.
- ✓ 부모 클래스의 메서드와 동일하거나 접근 범위가 넓은 접근 제한자를 자식 클래스에서 선언해야 한다.
- ✓ `static`, `final`, `private` 메서드는 오버라이딩을 할 수 없다.

메소드 오버라이딩

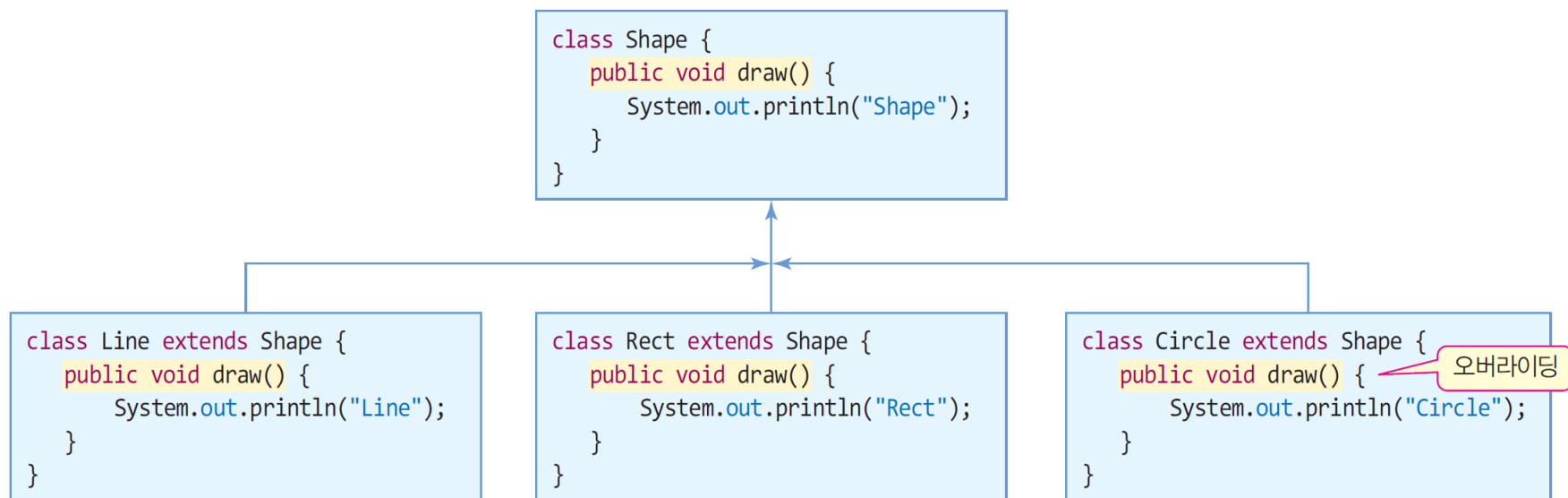
메소드 오버라이딩(Method Overriding)

- 슈퍼 클래스의 메소드를 서브 클래스에서 재정의
 - 슈퍼 클래스 메소드의 이름, 매개변수 타입 및 개수, 리턴 타입 등 모든 것 동일하게 작성
- 메소드 무시하기, 덮어쓰기로 번역되기도 함
- 동적 바인딩 발생
 - 서브 클래스에 오버라이딩된 메소드가 무조건 실행되는 동적 바인딩



메소드 오버라이딩 사례

Shape 클래스의 draw() 메소드를 Line, Rect, Circle 클래스에서 각각 오버라이딩한 사례





어노테이션(Annotation)

- 어노테이션(Annotation)이란?

프로그램에게 추가적인 정보를 제공해주는 메타데이터 (metadata:데이터에 관한 구조화된 데이터로, 다른 데이터를 설명해 주는 데이터)이다.

- 어노테이션 용도

- 컴파일러에게 코드 작성 문법 에러 체크하도록 정보 제공
- 소프트웨어 개발 툴이 빌드나 배치 시 코드를 자동 생성하게 정보 제공
- 실행 시(런타임시) 특정 기능 실행하도록 정보 제공



@Override

3) @Override

메소드 재정의에서 흔히 하는 실수는 메소드의 이름이 잘못되어서 자신이 재정의하였다고 믿고 있지만 실제로는 재정의되지 않은 경우이다.

예를 들어서 앞의 예제에서 `sound()`라고 하여야 할 것을 `saund()`라고 하였다면 메소드는 재정의되지 않으며 컴파일러는 `saund()`를 새로운 메소드로 인지하기 때문에 아무런 오류가 발생하지 않는다.

@Override 어노테이션

컴파일러에게 부모 클래스의 메소드 선언부와 동일한지 검사 지시



@Override

```
class Animal {  
    public void sound() {  
        System.out.println("소리를 낸다.");  
    }  
}
```

```
class Dog extends Animal {  
    public void saund() {  
        System.out.println("멍멍!");  
    }  
}
```

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



@Override

이 경우에는 @Override 어노테이션(annotation)을 사용하면 컴파일러에게 수퍼 클래스의 메소드를 오버라이드하려고 한다는 것을 알려줄 수 있다. 만약 컴파일러가 수퍼 클래스에 그러한 메소드가 존재하지 않는다는 것을 감지하면 오류가 발생하게 된다.

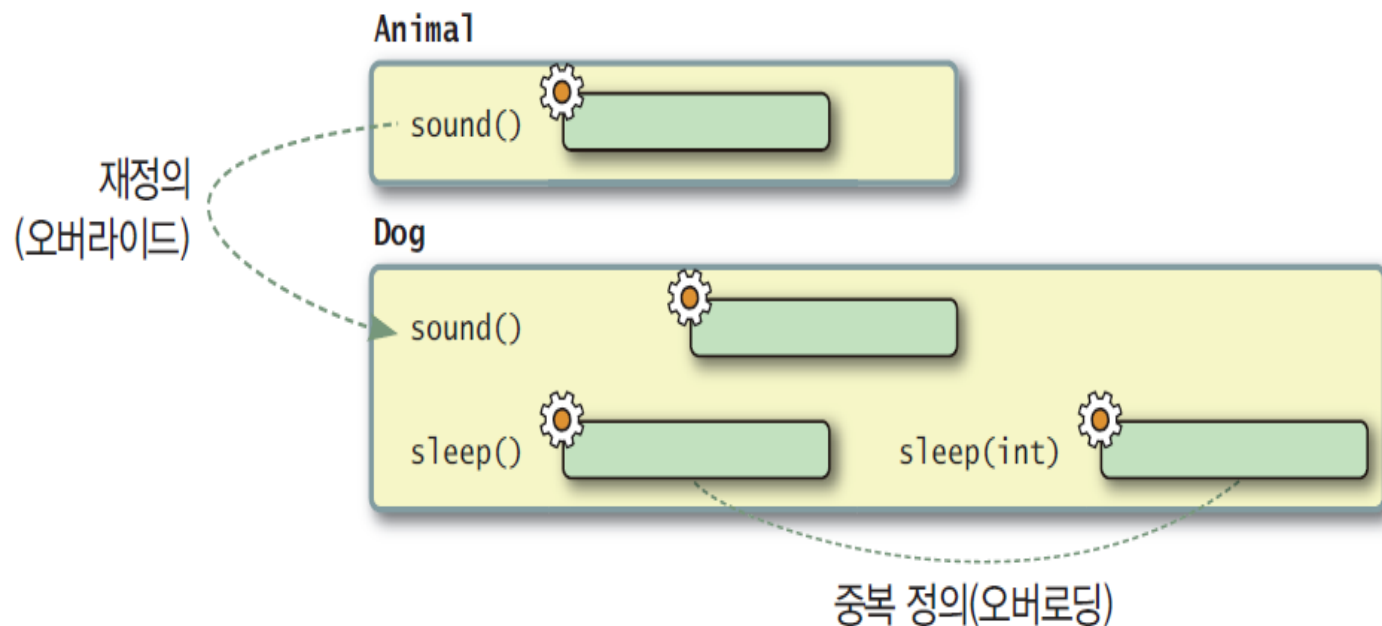
```
class Dog extends Animal {  
    @Override  
    public void saund() {  
        System.out.println("멍멍!");  
    }  
}
```

재정의의 의도하였다는 것을 확실하게 컴파일러에게 전달하여 오류를 방지한다.

오버라이딩과 오버로딩

4) 오버라이딩(Overriding)과 오버로딩(Overloading)

오버라이딩은 상속받은 메서드의 내용을 재정의하는 것이고, 오버로딩은 매개변수가 다른 동일한 이름의 메서드를 중복해서 정의하는 것이다





필드 재정의와 super

5) 필드 재정의와 super

만약 서브 클래스에서 수퍼 클래스와 동일한 이름의 필드를 정의하면 어떻게 되는가? 이런 경우에는 무조건 서브 클래스의 필드가 수퍼 클래스의 필드를 가리게 된다. 타입이 다르더라도 마찬가지이다. 이런 경우, 서브 클래스 안에서는 수퍼 클래스의 필드를 간단히 이름만으로는 참조할 수 없다. 대신 `super`라고 하는 키워드를 사용하여야 한다.



필드 재정의와 super

this는 현재 객체를 참조하기 위하여 사용된다.

super는 상속 관계에서 수퍼 클래스의 메소드나 필드를 참조하기 위하여 사용된다.

수퍼 클래스의 메소드를 완전히 대체하는 경우보다 내용을 추가하는 경우가 많다. 이런 경우에는 자신이 필요한 부분을 작성한 다음, **super** 키워드를 이용하여 수퍼 클래스의 메소드를 호출해주면 된다.

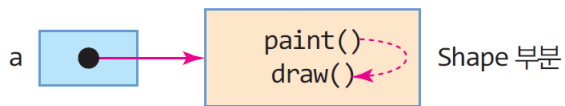
동적 바인딩

- 실행할 메소드를 실행 시(run time)에 결정
- 오버라이딩 메소드가 항상 호출

```
public class Shape {  
    protected String name;  
    public void paint() {  
        draw();  
    }  
    public void draw() {  
        System.out.println("Shape");  
    }  
    public static void main(String [] args) {  
        Shape a = new Shape();  
        a.paint();  
    }  
}
```

⇒ 실행 결과

Shape

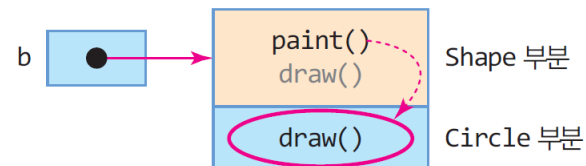


```
class Shape {  
    protected String name;  
    public void paint() {  
        draw();  
    }  
    public void draw() {  
        System.out.println("Shape");  
    }  
}  
public class Circle extends Shape {  
    @Override  
    public void draw() {  
        System.out.println("Circle");  
    }  
    public static void main(String [] args) {  
        Shape b = new Circle();  
        b.paint();  
    }  
}
```

동적 바인딩

⇒ 실행 결과

Circle





오버라이딩 vs. 오버로딩

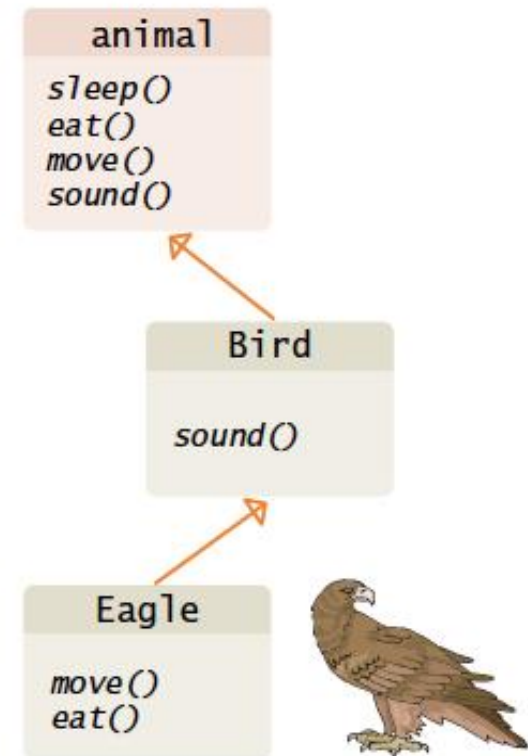
비교 요소	메소드 오버로딩	메소드 오버라이딩
선언	같은 클래스나 상속 관계에서 동일한 이름의 메소드 중복 작성	서브 클래스에서 슈퍼 클래스에 있는 메소드와 동일한 이름의 메소드 재작성
관계	동일한 클래스 내 혹은 상속 관계	상속 관계
목적	이름이 같은 여러 개의 메소드를 중복 작성하여 사용의 편리성 향상. 다형성 실현	슈퍼 클래스에 구현된 메소드를 무시하고 서브 클래스에서 새로운 기능의 메소드를 재정의하고자 함. 다형성 실현
조건	메소드 이름은 반드시 동일하고, 매개변수 타입이나 개수가 달라야 성립	메소드의 이름, 매개변수 타입과 개수, 리턴 타입이 모두 동일하여야 성립
바인딩	정적 바인딩. 호출될 메소드는 컴파일 시에 결정	동적 바인딩. 실행 시간에 오버라이딩된 메소드 찾아 호출

메소드 호출순서

6) 메소드 호출순서

다음과 같은 상속 계층 구조에서 만약 Eagle의 객체를 생성하여서 메소드를 호출하였다면 과연 메소드가 호출되는지를 알아보자.

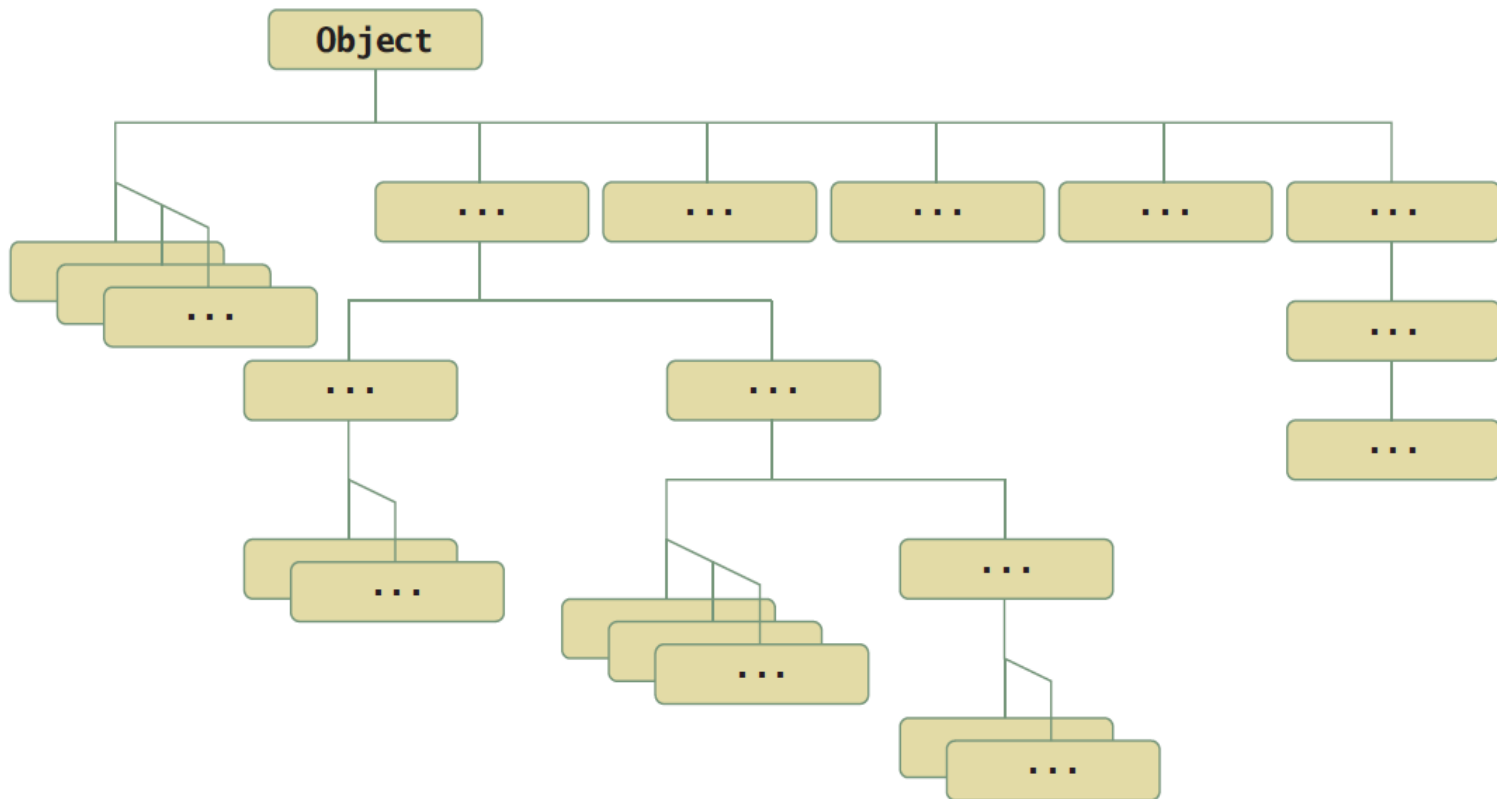
```
Eagle e = new Eagle();  
e.sleep();    // Animal의 sleep() 호출  
e.eat();      // Eagle의 eat() 호출  
e.sound();    // Bird의 sound() 호출
```



Object 클래스

자바에서는 클래스를 정의할 때 명시적으로 수퍼 클래스를 선언하지 않으면 Object 클래스가 수퍼 클래스가 된다.

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





Object 클래스

Object 안에 정의되어 있는 메소드는 다음과 같다. 만약 이들 메소드를 사용하기로 작성하였다면 사용자 자신의 용도에 맞도록 이들 메소드들을 재정의할 수 있다.

메서드	설명
<code>protected Object clone()</code>	객체 자신의 복사본을 생성하여 반환한다.
<code>public boolean equals(Object obj)</code>	<code>obj</code> 가 이 객체와 같은지를 나타낸다.
<code>protected void finalize()</code>	가비지 콜렉터에 의하여 호출된다.
<code>public final Class getClass()</code>	객체의 실행 클래스를 반환한다.
<code>public int hashCode()</code>	객체에 대한 해쉬 코드를 반환한다.
<code>public String toString()</code>	객체의 문자열 표현을 반환한다.



Object 클래스

```
public class Car {  
    private String model;  
    public Car(String model) {  
        this.model = model;  
    }  
    @Override  
    public String toString() {  
        return "자동차 모델명: " + model;  
    }  
}
```


Object 클래스

```
public class CarTest {  
    public static void main(String[] args){  
        Car myCar = new Car("쏘나타");  
        System.out.println(myCar.toString());  
    }  
}
```

실행결과 자동차 모델명: 쏘나타

만약 Car 클래스에 toString() 재정의
되어 있지 않다면 **Object의 toString()** 호출

```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

실행결과 exam_class_static.Car@15db9742

패키지명.클래스명@16진수값



종단 클래스와 종단 메소드

1) 클래스의 final

종단 클래스(final class)는 상속을 시킬 수 없는 클래스를 말한다. 대표적인 것이 String 클래스이다. String 클래스는 컴파일러에서 많이 쓰이기 때문에 종단 클래스로 선언되어 있다. 종단 클래스로 선언하려면 클래스의 선언 맨 앞에 final을 붙인다.

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

종단 클래스로 선언되면 그 클래스 내부의 메소드는 모두 재정의될 수 없다.

```
public final class String  
    implements java.io.Serializable, Comparable<String>, CharSequence {
```

종단 클래스

```
public final class Member {
```

```
.....
```

```
}
```

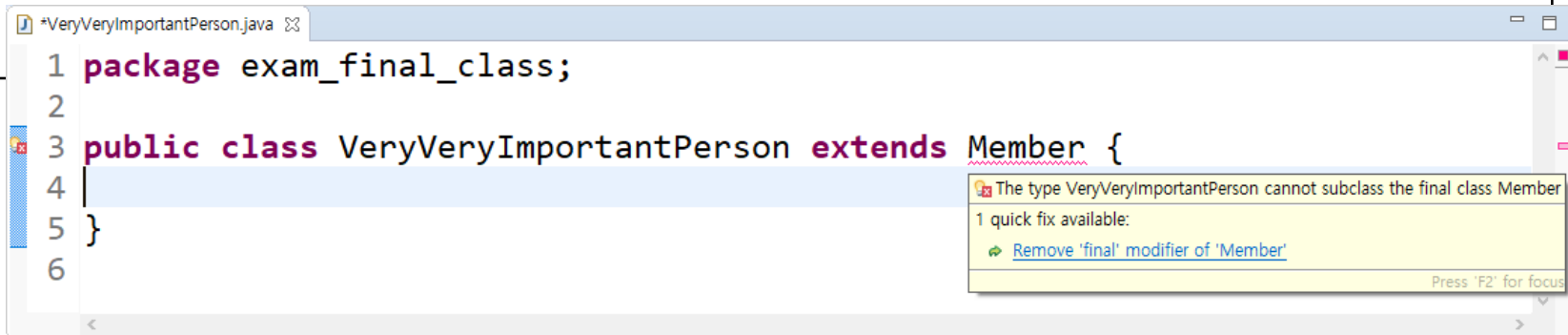
//아래의 클래스 생성이 가능한가요?

```
public class VeryVeryImportantPerson extends Member
```

```
{
```

```
.....
```

```
}
```



The screenshot shows an IDE window titled '*VeryVeryImportantPerson.java'. The code is as follows:

```
1 package exam_final_class;
2
3 public class VeryVeryImportantPerson extends Member {
4
5 }
6
```

A red squiggly line under 'Member' in line 3 indicates an error. A tooltip is displayed with the following text:

The type VeryVeryImportantPerson cannot subclass the final class Member

1 quick fix available:

- [Remove 'final' modifier of 'Member'](#)

Press 'F2' for focus



종단 클래스와 종단 메소드

2) 메소드의 final

일반 클래스에서 특정한 메소드만 재정의될 수 없게 만들려면 종단 메소드(final method)로 선언하면 된다.

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



종단 메소드

```
public class Car {  
    //필드  
    public int speed;  
    //메소드  
    public void speedUp() {  
        speed += 1;  
    }  
    //final 메소드  
    public final void stop() {  
        System.out.println("차를 멈춤");  
        speed = 0;  
    }  
}
```

종단 메소드

```
public class SportsCar extends Car {
    @Override
    public void speedUp() {
        speed += 10;
    }
    // 아래 stop() 메서드에 대해 생성이 가능한가?
    @Override
    public void stop() {
        System.out.println("스포츠카를 멈춤");
        speed = 0;
    }
    @Override
    public void stop() {
        System.out.println("스포츠카를 멈춤");
        speed = 0;
    }
}
```

3) 필드의 final

일반 클래스 내에 필드의 값을 고정하고자 할 때 필드 앞에 final 키워드를 선언하면 된다.

```
class Circle{  
    static final double PI=3.1415;  
    ...  
}
```




Thank You

