

5

상속

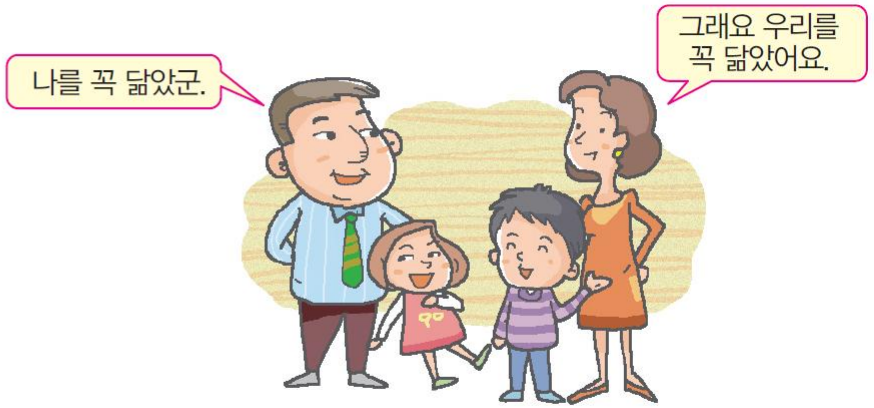
상속 (inheritance)



- 객체 지향의 상속
  - 부모클래스에 만들어진 필드, 메소드를 자식클래스가 물려받음
  - 부모의 생물학적 특성을 물려받는 유전과 유사



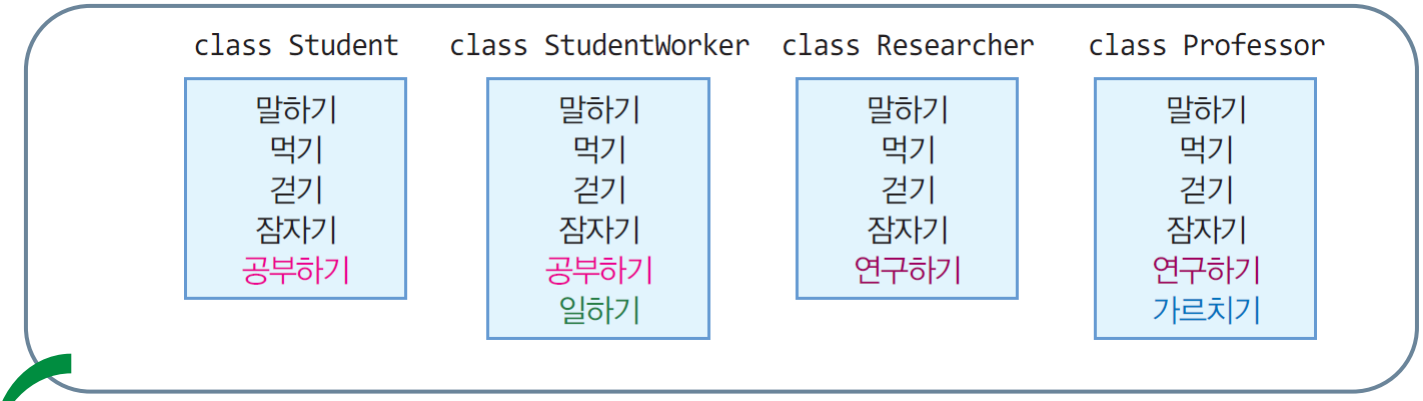
유산 상속



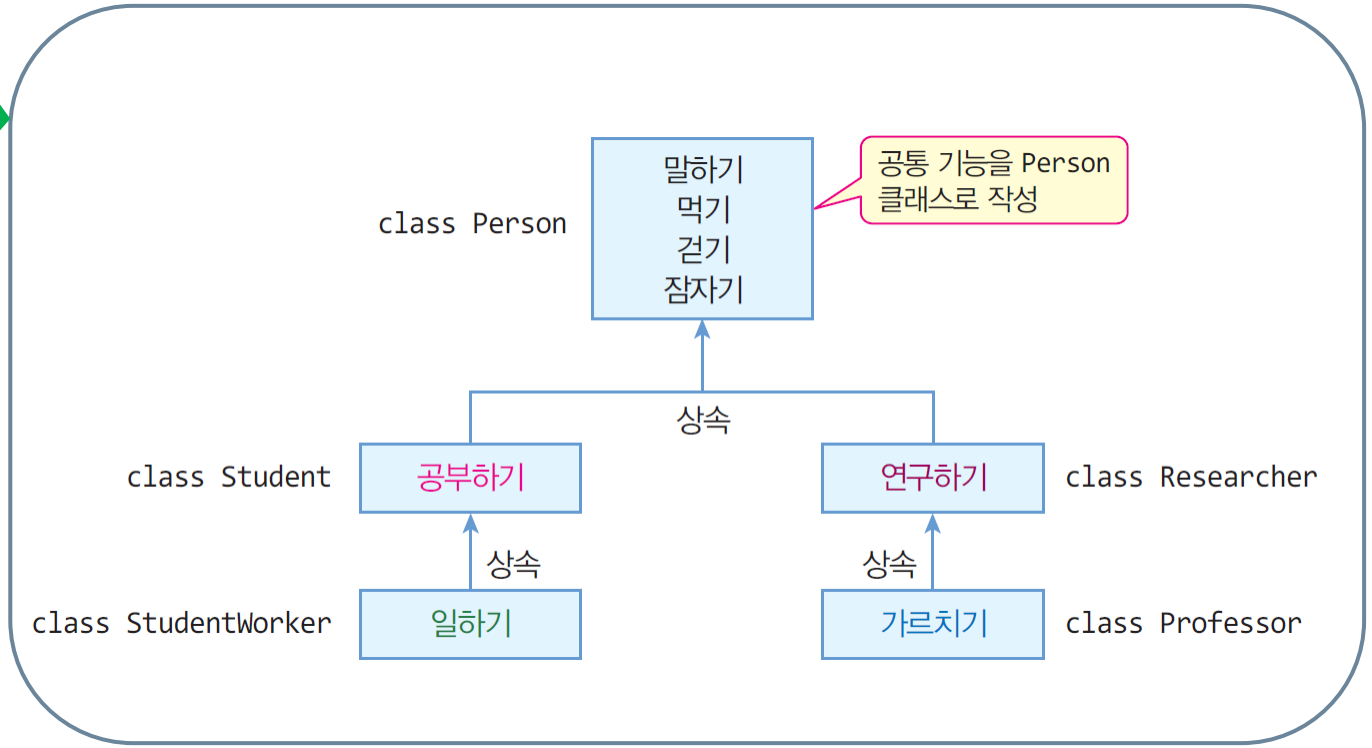
유전적 상속 : 객체 지향 상속

- 상속을 통해 간결한 자식 클래스 작성
  - 동일한 특성을 재정의할 필요가 없어 자식 클래스가 간결해짐

상속의 편리한 사례



상속이 없는 경우
중복된 멤버를 가진
4 개의 클래스



상속을 이용한
경우 중복이 제거되고
간결해진 클래스 구조

객체 지향에서 상속의 장점



- 클래스의 간결화
  - 멤버의 중복 작성 불필요
- 클래스 관리 용이
  - 클래스들의 계층적 분류
- 소프트웨어의 생산성 향상
  - 클래스 재사용과 확장 용이
  - 새로운 클래스의 작성 속도 빠름

클래스 상속과 객체



자바의 상속 선언

```
public class Person {
        …
}
public class Student extends Person { // Person을 상속받는 클래스 Student 선언
        …
}
public class StudentWorker extends Student { // Student를 상속받는 StudentWorker 선언
        …
}
```

- 부모 클래스 -> 슈퍼 클래스(super class)로 부름
- 자식 클래스 -> 서브 클래스(sub class)로 부름
- extends 키워드 사용
  - 슈퍼 클래스를 확장한다는 개념

예제 5-1 : 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 + ")");
    }
}
```

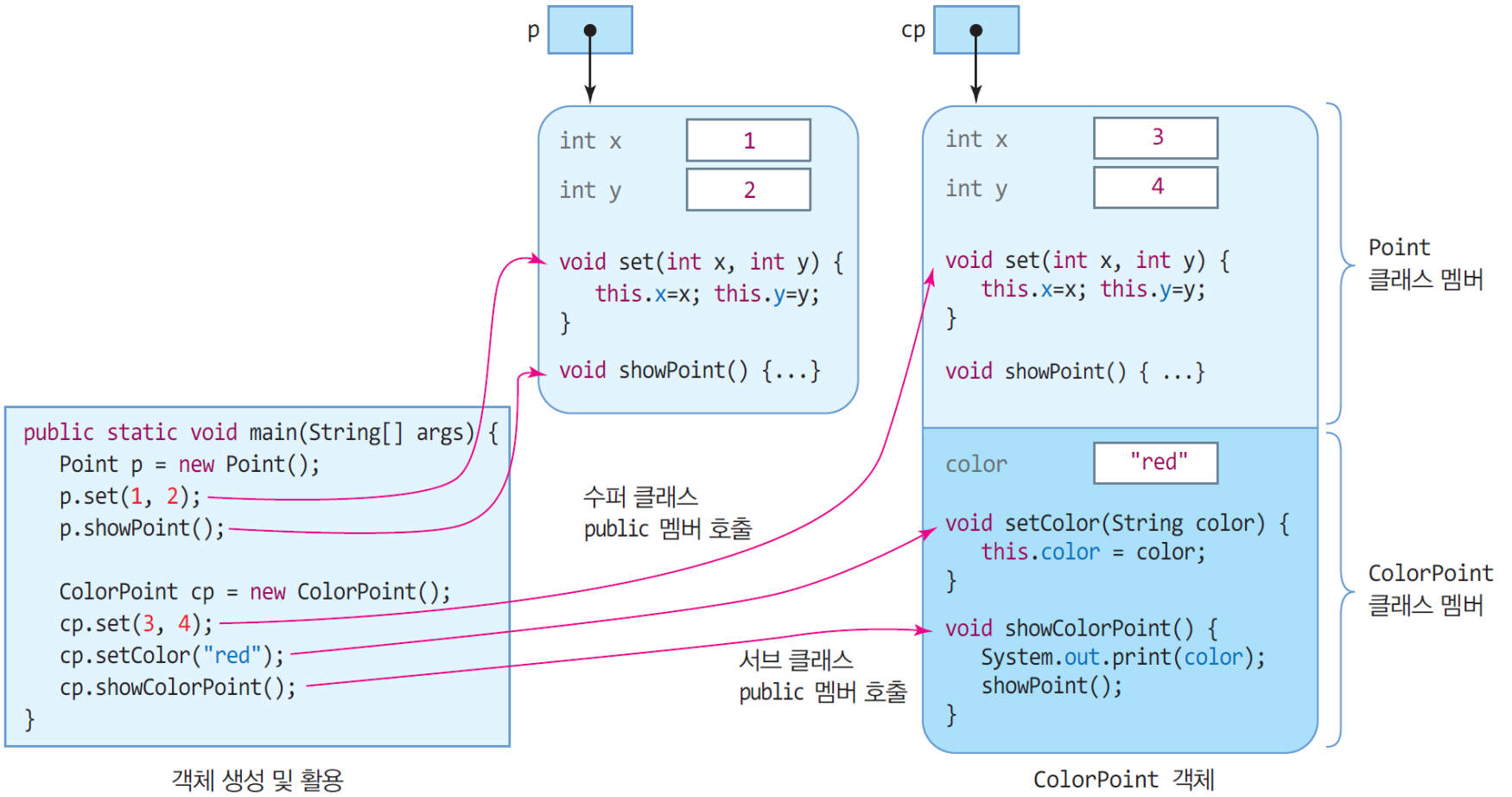
```
// Point를 상속받은 ColorPoint 선언
class ColorPoint extends Point {
    private String color; // 점의 색
    public void setColor(String color) {
        this.color = color;
    }
    public void showColorPoint() { // 컬러 점의 좌표 출력
        System.out.print(color);
        showPoint(); // Point 클래스의 showPoint()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

예제 5-1의 객체 생성

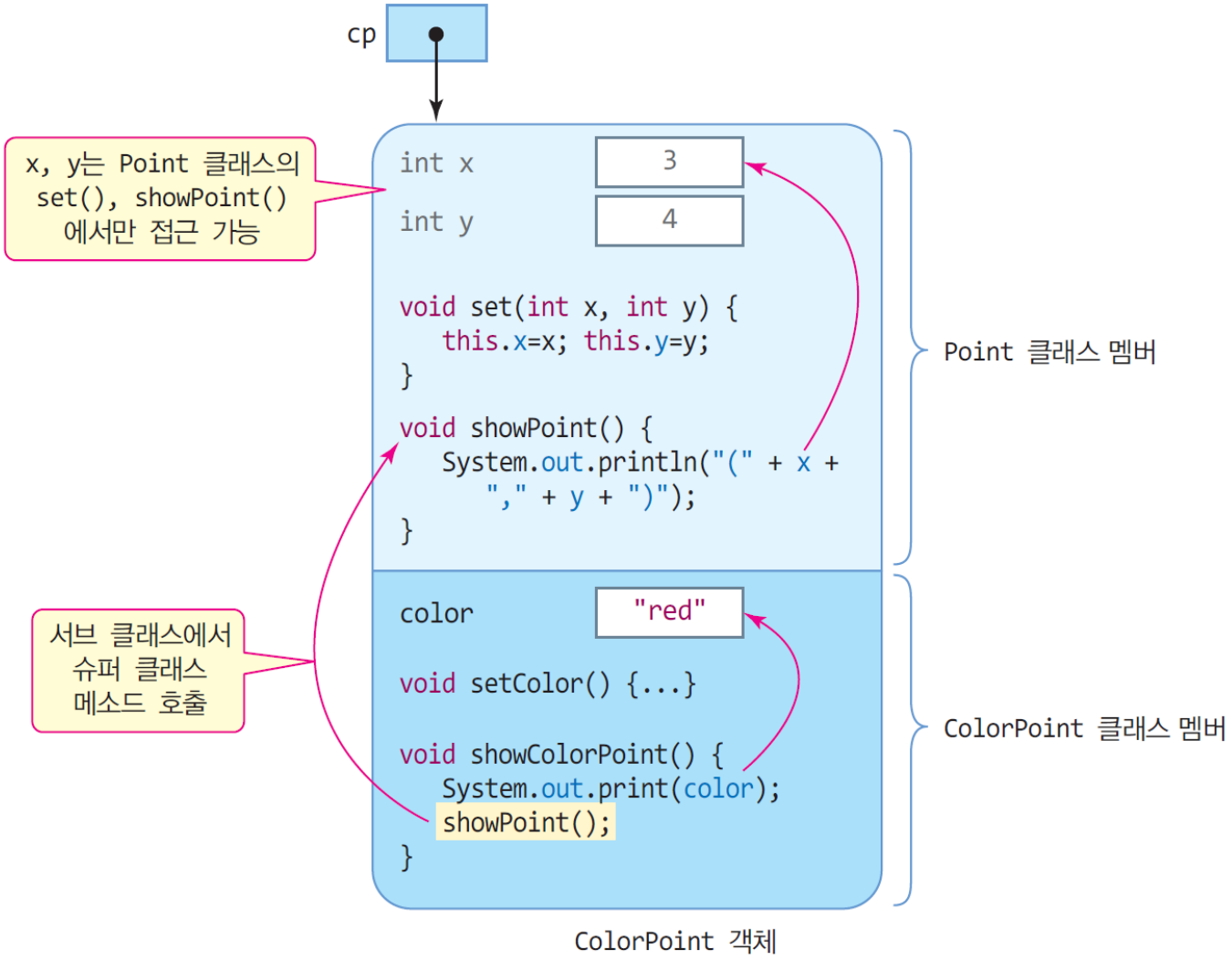




* new ColorPoint()에 의해 생긴 서브 클래스 객체에 주목

서브클래스에서 슈퍼 클래스의 멤버 접근

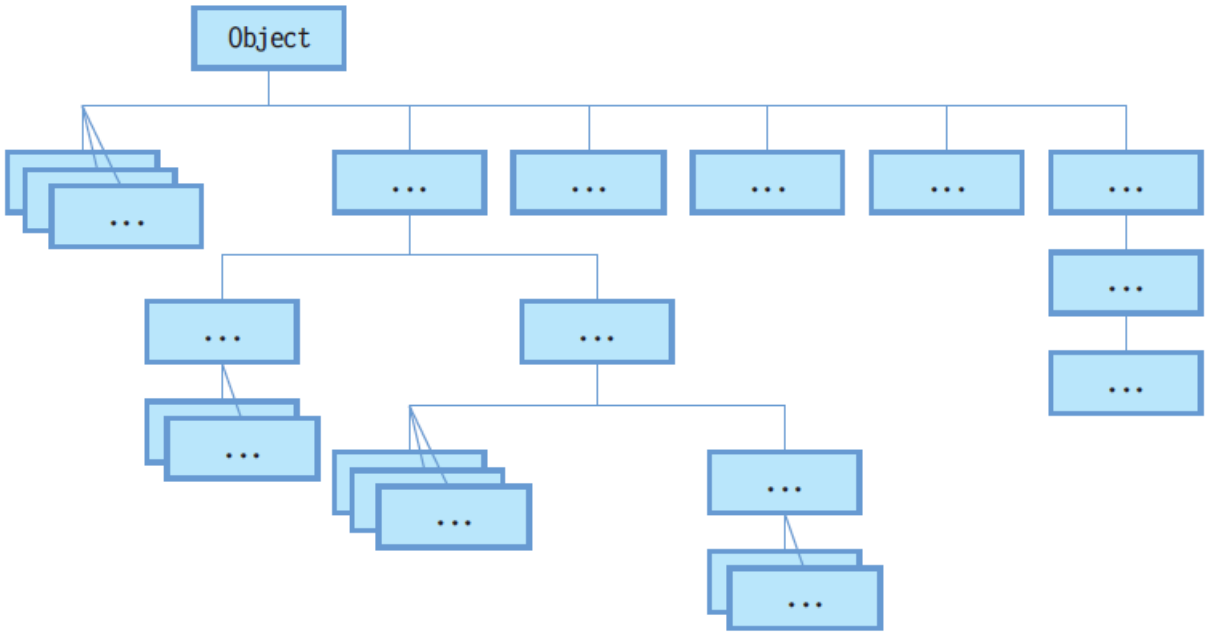




자바 상속의 특징



- 클래스의 다중 상속 지원하지 않음
- 상속 횟수 무제한
- 상속의 최상위 조상 클래스는 java.lang.Object 클래스
  - 모든 클래스는 자동으로 java.lang.Object를 상속받음
  - 자바 컴파일러에 의해 자동으로 이루어짐



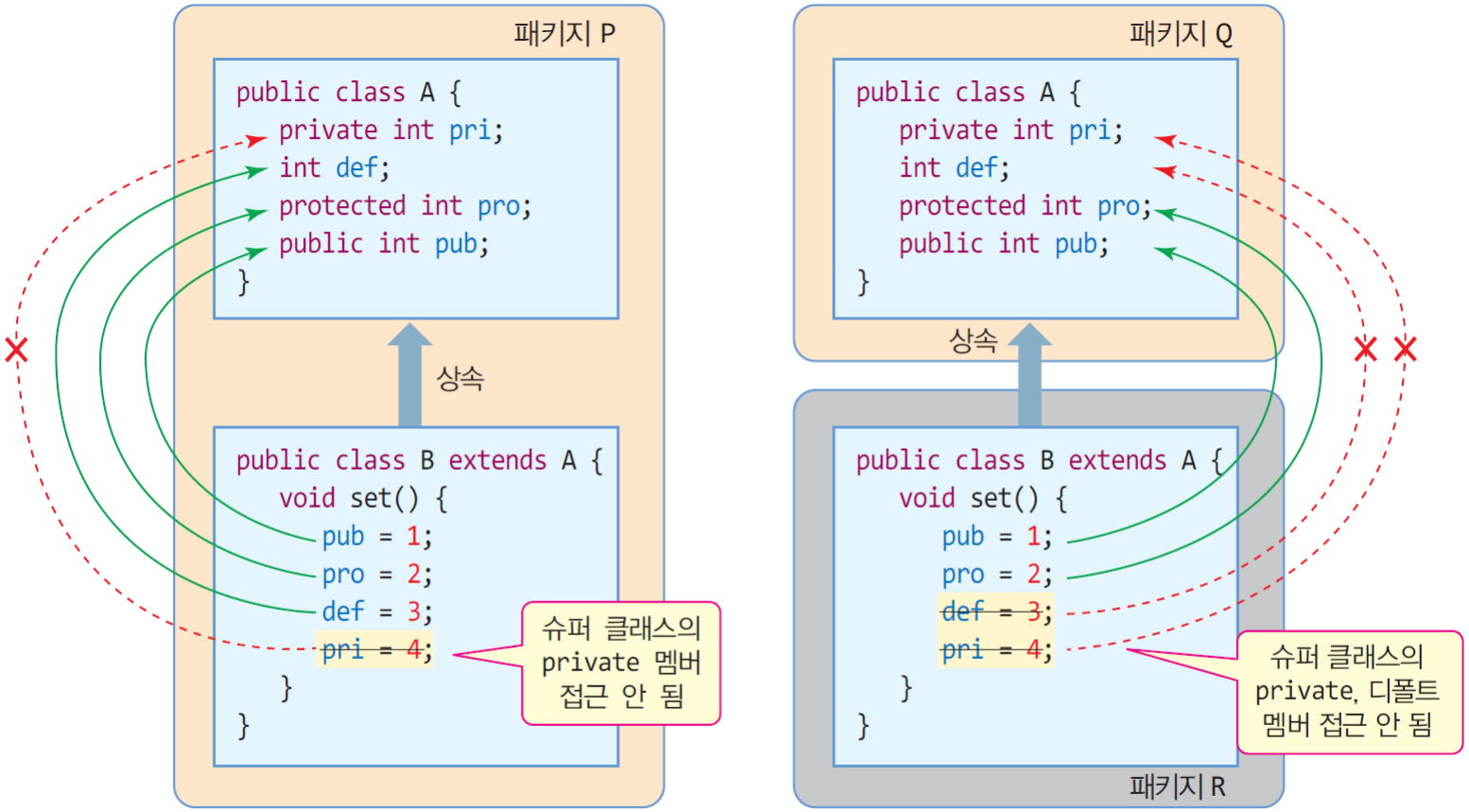
상속과 접근 지정자



- 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

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

예제 5-2: 상속 관계에 있는 클래스 간 멤버 접근



클래스 Person을 아래와 같은 멤버 필드를 갖도록 선언하고 클래스 Student는 클래스 Person을 상속받아 각 멤버 필드에 값을 저장하시오. 이 예제에서 Person 클래스의 private 필드인 weight는 Student 클래스에서는 접근이 불가능하여 슈퍼 클래스인 Person의 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();
    }
}
```

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.

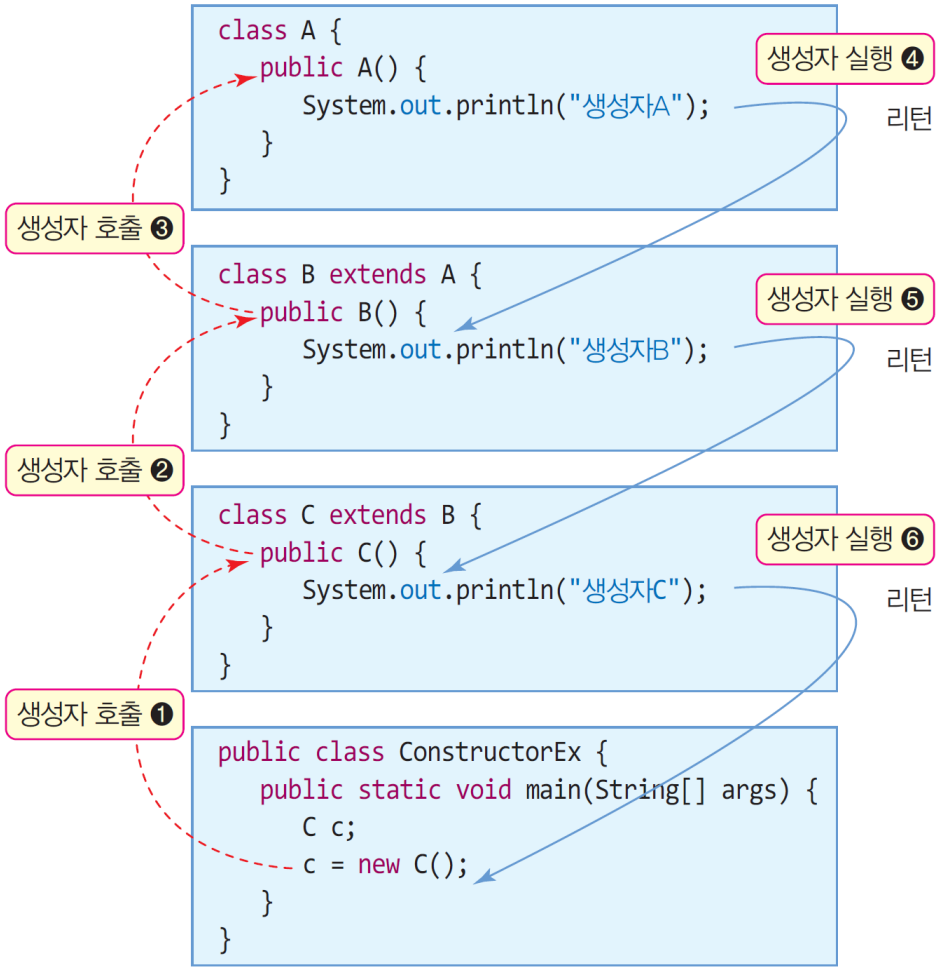
질문 2 서브 클래스의 생성자와 슈퍼 클래스의 생성자 중 누가 먼저 실행되는가?

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

슈퍼클래스와 서브 클래스의 생성자간의 호출 및 실행 관계





실행 결과

```
생성자A
생성자B
생성자C
```

서브 클래스에서 슈퍼 클래스의 생성자 선택



- 상속 관계에서의 생성자
  - 슈퍼 클래스와 서브 클래스 각각 각각 여러 생성자 작성 가능
- 서브 클래스 생성자 작성 원칙
  - 서버 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

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
        System.out.println("생성자B");
    }
}
```

오류

```
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 이용하여 명시적으로 슈퍼 클래스 생성자 선택



- super()
  - 서브 클래스에서 명시적으로 슈퍼 클래스의 생성자 선택 호출
    - super(parameter);
    - 인자를 이용하여 슈퍼 클래스의 적당한 생성자 호출
    - 반드시 서브 클래스 생성자 코드의 제일 첫 라인에 와야 함

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
```

예제 5-3 : 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();
    }
}
```

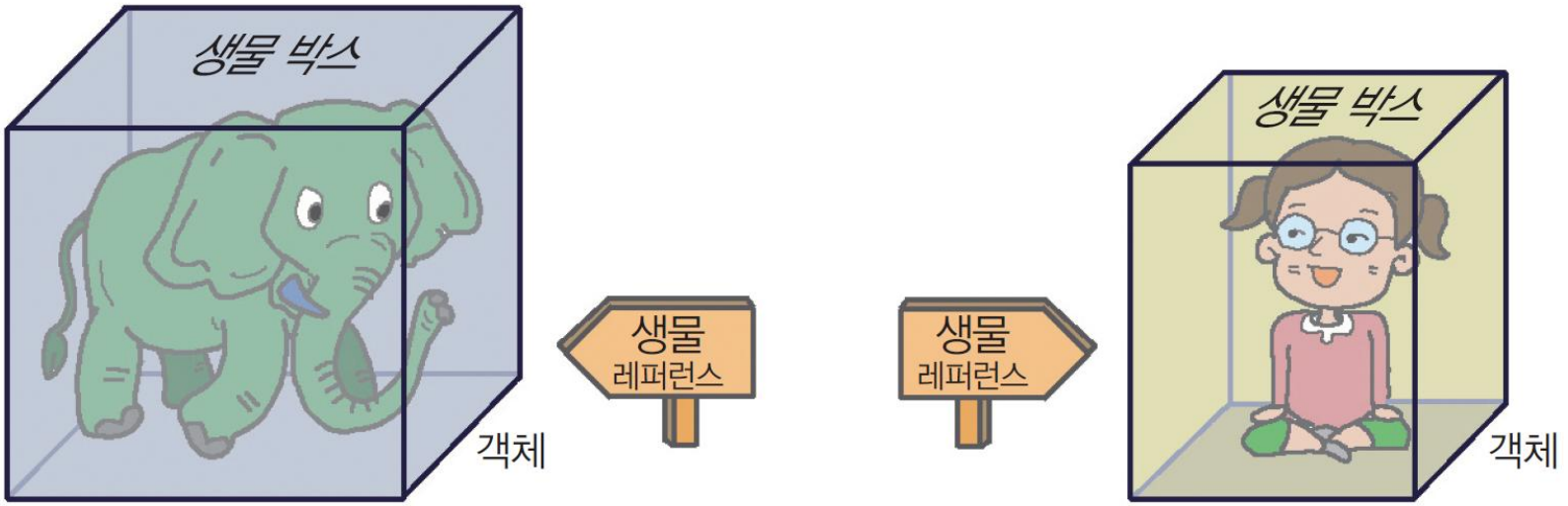
x=5, y=6,
color = "blue" 전달

```
blue(5,6)
```

사람은 생물이다.



생물을 넣는 박스에 코끼리나 사람을 넣고 박스 앞에 생물을 가리키는 팻말을 사용해도 무방하다. 왜냐하면, **사람은 생물을 상속 받았기 때문이다.**



업캐스팅(upcasting)



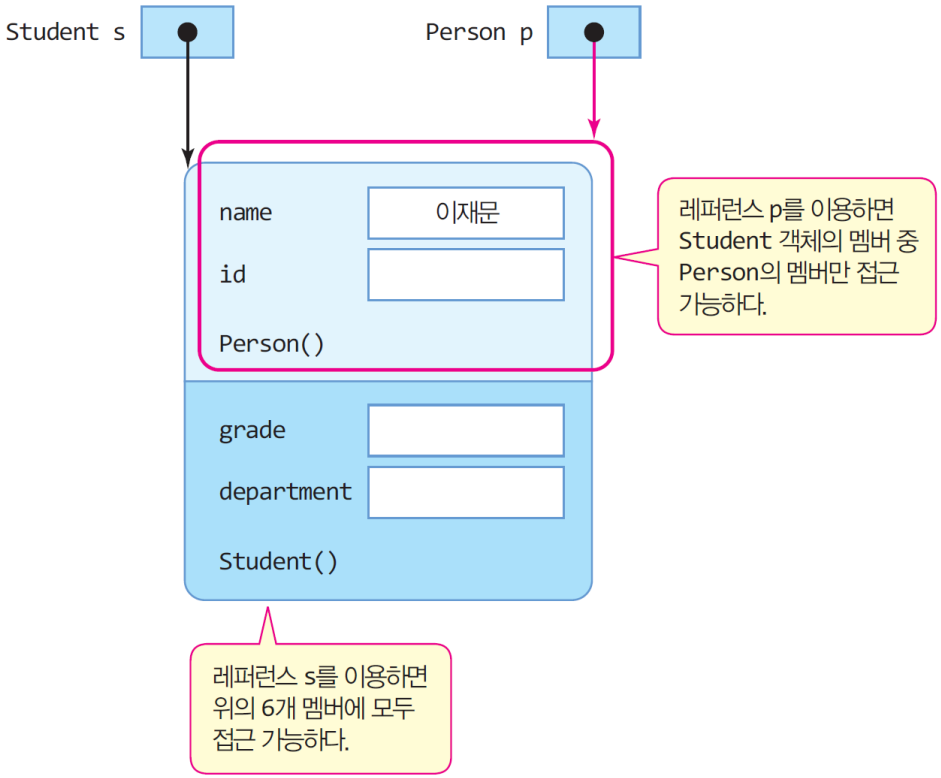
- 서브 클래스의 객체는...
  - 슈퍼 클래스의 멤버를 모두 가지고 있음
  - 슈퍼 클래스의 객체로 취급할 수 있음
    - '사람은 생물이다'의 논리와 같음
- 업캐스팅이란?
  - 서브 클래스 객체를 슈퍼 클래스 타입으로 타입 변환

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

Student s = new Student();
Person p = s; // 업캐스팅, 자동타입변환
```

- 업캐스팅된 레퍼런스
  - 객체 내에 슈퍼 클래스의 멤버만 접근 가능

업캐스팅 사례

```
class Person {
    String name;
    String id;

    public Person(String name) {
        this.name = name;
    }
}

class Student extends Person {
    String grade;
    String department;

    public Student(String name) {
        super(name);
    }
}

public class UpcastingEx {
    public static void main(String[] args) {
        Person p;
        Student s = new Student("이재문");
        p = s; // 업캐스팅

        System.out.println(p.name); // 오류 없음

오류    p.grade = "A"; // 컴파일 오류
        p.department = "Com"; // 컴파일 오류
    }
}
```



실행 결과

```
이재문
```

다운캐스팅(downcasting)



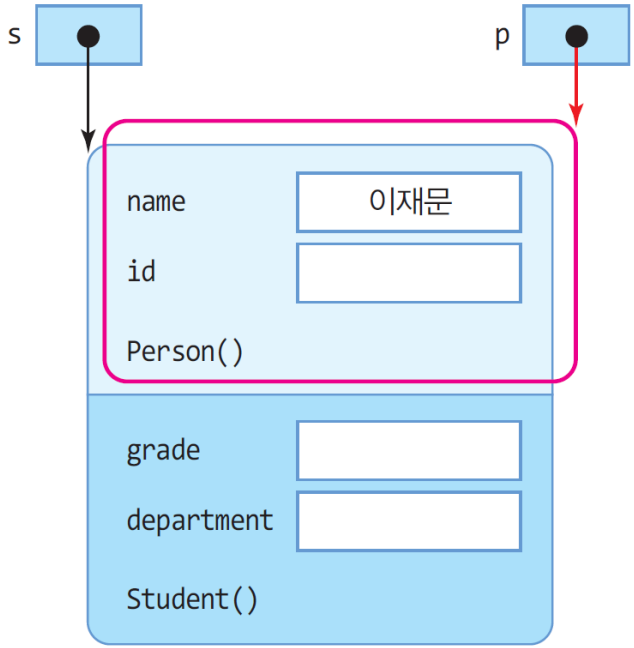
- 다운캐스팅이란?
  - 슈퍼 클래스 객체를 서브 클래스 타입으로 변환
  - 개발자의 명시적 타입 변환 필요

```
class Person { ... }
class Student extends Person { ... }
...
Person p = new Student("이재문"); // 업캐스팅
...
Student s = (Student)p; // 다운캐스팅, (Student)의 타입 변환 표시 필요
```

다운캐스팅 사례




```
public class DowncastingEx {
    public static void main(String[] args) {
        Person p = new Student("이재문");  // 업캐스팅
        Student s;

        s = (Student)p;  // 다운캐스팅

        System.out.println(s.name);  // 오류 없음
        s.grade = "A";  // 오류 없음
    }
}
```



실행 결과

```
이재문
```

instanceof 연산자와 객체의 타입 판단



- 업캐스팅된 레퍼런스로 객체의 타입 판단 어려움
  - 슈퍼 클래스는 여러 서브 클래스에 상속되기 때문
    - 예) '생물' 팻말(레퍼런스)이 가리키는 박스에 들어 있는 객체의 타입이 사람인지, 동물인지 팻말만 보고서는 알 수 없음

- instanceof 연산자
  - 레퍼런스가 가리키는 객체의 타입 식별을 위해 사용
  - 사용법

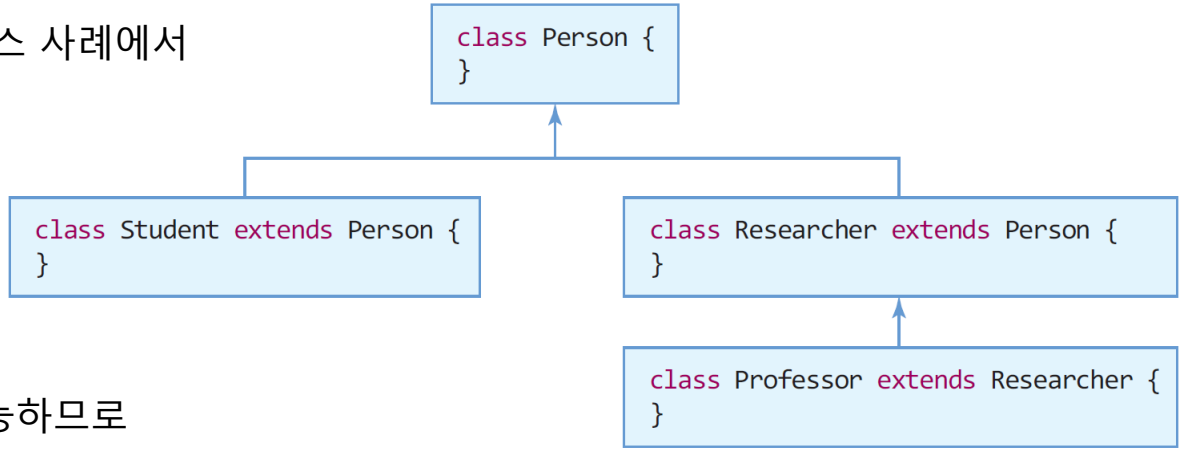
객체레퍼런스 **instanceof** 클래스타입

연산의 결과 : true/false의 불린 값

업캐스팅 레퍼런스가 가리키는 객체는?



1. 상속 관계에 있는 옆의 클래스 사례에서



2. 다음과 같은 업캐스팅이 가능하므로

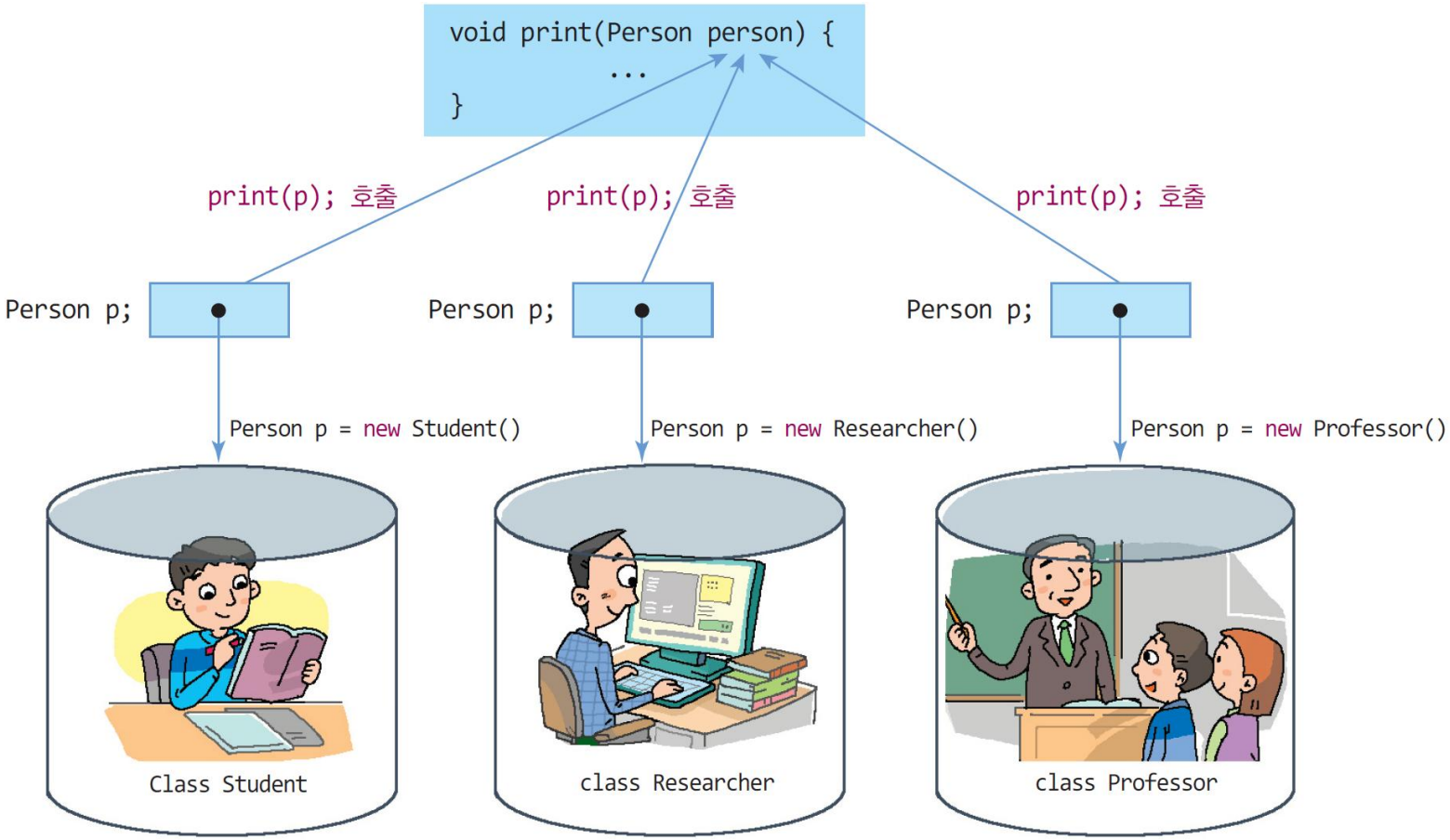
```
Person p = new Person();
Person p = new Student();      // 업캐스팅
Person p = new Researcher();   // 업캐스팅
Person p = new Professor();    // 업캐스팅
```

3. `print(p);` 를 호출하면,
print() 메소드에서 person이 어떤 객체를 가리키는지 알 수 없음

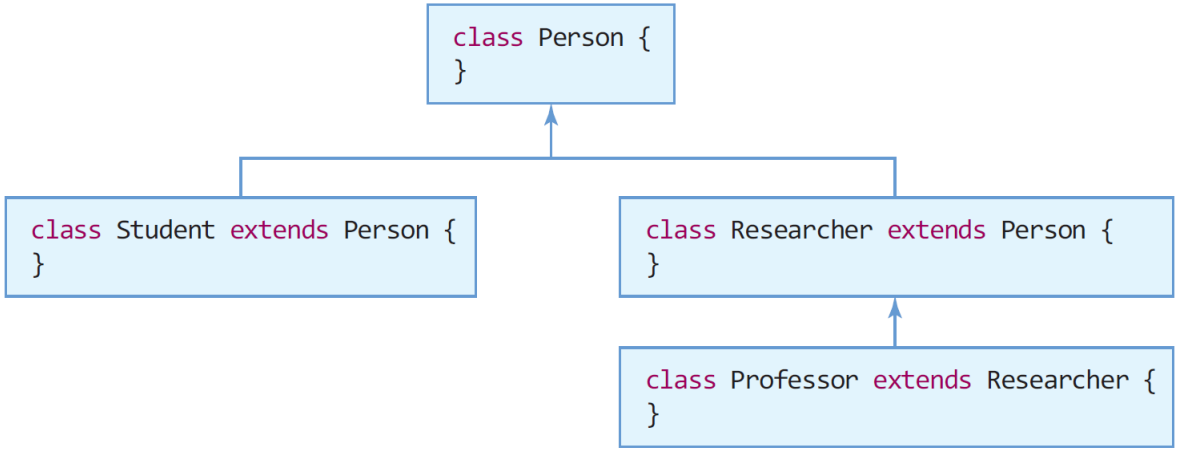
```
void print(Person person) {
   // person이 가리키는 객체가 Person 타입일 수도 있고,
   // Student, Researcher, 혹은 Professor 타입일 수도 있다.
   .....
}
```

Person 타입의 레퍼런스 person이 어떤 타입의 객체를 가리키는지 알 수 없음





instanceof 사용 예




```
Person jee= new Student();
Person kim = new Professor();
Person lee = new Researcher();
if (jee instanceof Person)      // jee는 Person 타입이므로 true
if (jee instanceof Student)     // jee는 Student 타입이므로 true
if (kim instanceof Student)     // kim은 Student 타입이 아니므로 false
if (kim instanceof Professor)   // kim은 Professor 타입이므로 true
if (kim instanceof Researcher) // Professor 객체는 Researcher 타입이기도 하므로 true
if (lee instanceof Professor)   // lee는 Professor 타입이 아니므로 false
```

오류

```
if(3 instanceof int) // 문법 오류. instanceof는 객체에 대한 레퍼런스만 사용
```

```
if("java" instanceof String) // true
```

예제 5-4 : instanceof 연산자 활용



instanceof 연산자를 이용하여 상속 관계에 따라 레퍼런스가 가리키는 객체의 타입을 알아본다.
실행
결과는 무엇인가?

```
class Person { }
class Student extends Person { }
class Researcher extends Person { }
class Professor extends Researcher { }

public class InstanceOfEx {
   static void print(Person p) {
      if(p instanceof Person)
         System.out.print("Person ");
      if(p instanceof Student)
         System.out.print("Student ");
      if(p instanceof Researcher)
         System.out.print("Researcher ");
      if(p instanceof Professor)
         System.out.print("Professor ");
      System.out.println();
   }
   public static void main(String[] args) {
      System.out.print("new Student() ->₩t"); print(new Student());
      System.out.print("new Researcher() ->₩t"); print(new Researcher());
      System.out.print("new Professor() ->₩t"); print(new Professor());
   }
}
```

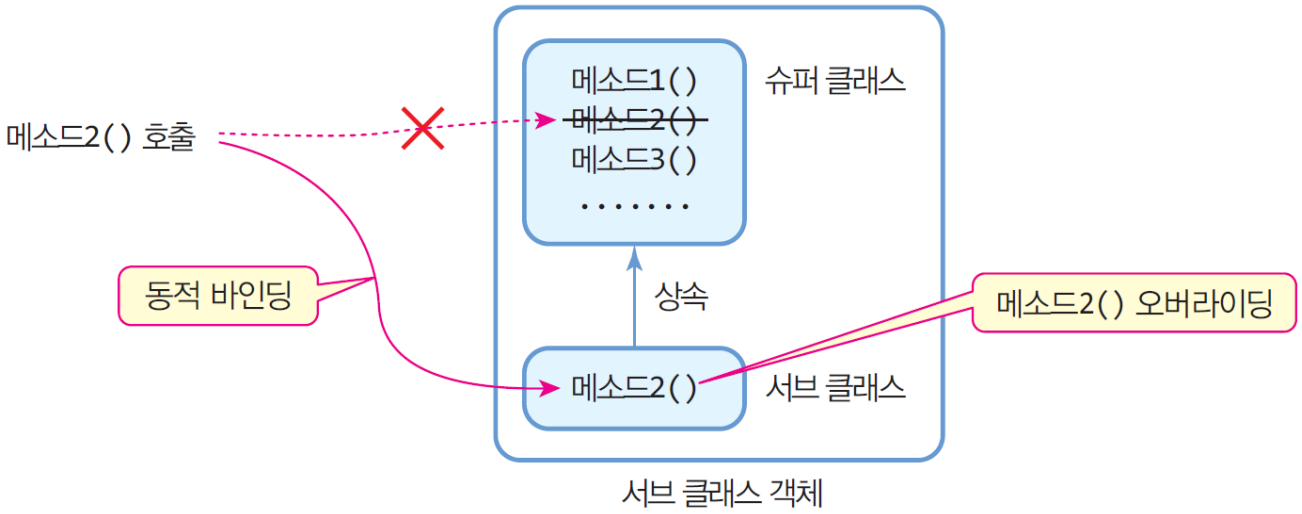
new Professor() 객체는
Person 타입이기도 하고,
Researcher, Professor 타입이기도 함

```
new Student() -> Person Student
new Researcher() -> Person Researcher
new Professor() -> Person Researcher Professor
```

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

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

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

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

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

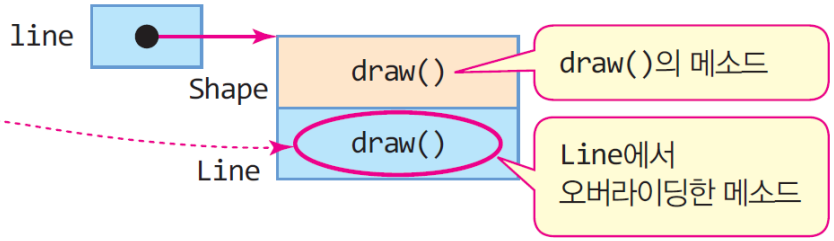
오버라이딩

오버라이딩에 의해 서브 클래스의 메소드 호출



(1) 서브 클래스 레퍼런스로 오버라이딩된 메소드 호출

```
Line line = new Line();
line.draw();
```

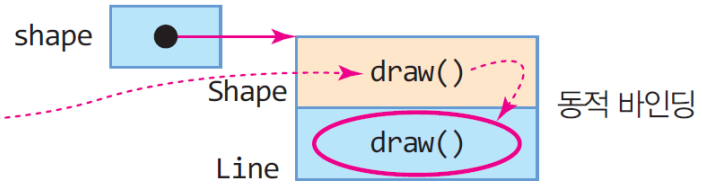


실행 결과

```
Line
```

(2) 업캐스팅에 의해 슈퍼 클래스 레퍼런스로 오버라이딩된 메소드 호출(동적 바인딩)

```
Shape shape = new Line();
shape.draw();
```



실행 결과

```
Line
```

오버라이딩의 목적, 다형성 실현



- 오버라이딩
  - 수퍼 클래스에 선언된 메소드를, 각 서브 클래스들이 자신만의 내용으로 새로 구현하는 기능
  - 상속을 통해 '하나의 인터페이스(같은 이름)에 서로 다른 내용 구현'이라는 객체 지향의 다형성 실현
    - Line 클래스에서 draw()는 선을 그리고
    - Circle 클래스에서 draw()는 원을 그리고
    - Rect 클래스에서 draw()는 사각형 그리고
- 오버라이딩은 실행 시간 다형성 실현
  - 동적 바인딩을 통해 실행 중에 다형성 실현
    - 오버로딩은 컴파일 타임 다형성 실현

예제 5-5 : 메소드 오버라이딩으로 다형성 실현



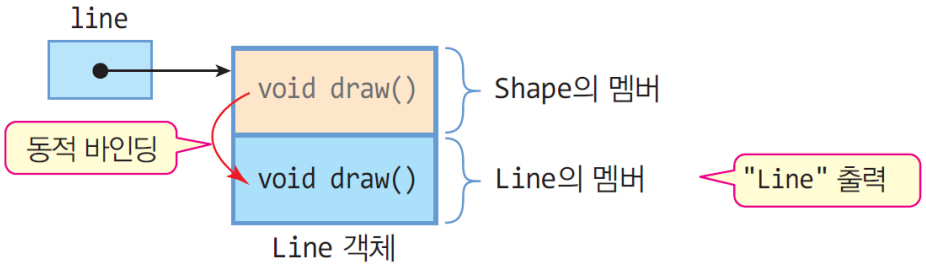

```
class Shape { // 슈퍼 클래스
    public Shape next;
    public Shape() { next = null; }

    public void draw() {
        System.out.println("Shape");
    }
}

class Line extends Shape {
    public void draw() { // 메소드 오버라이딩
        System.out.println("Line");
    }
}

class Rect extends Shape {
    public void draw() { // 메소드 오버라이딩
        System.out.println("Rect");
    }
}

class Circle extends Shape {
    public void draw() { // 메소드 오버라이딩
        System.out.println("Circle");
    }
}
```

```
public class MethodOverridingEx {
    static void paint(Shape p) {
        p.draw(); // p가 가리키는 객체 내에 오버라이딩된 draw() 호출.
                  // 동적 바인딩
    }

    public static void main(String[] args) {
        Line line = new Line();
        paint(line);
        paint(new Shape());
        paint(new Line());
        paint(new Rect());
        paint(new Circle());
    }
}
```

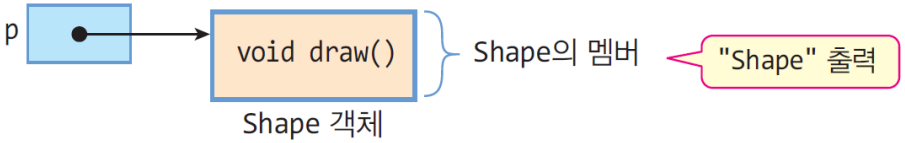
```
Line
Shape
Line
Rect
Circle
```

예제 실행 과정

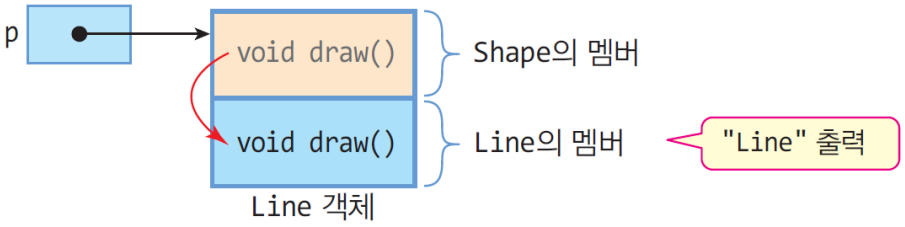
```
Line line = new Line()
paint(line);
```



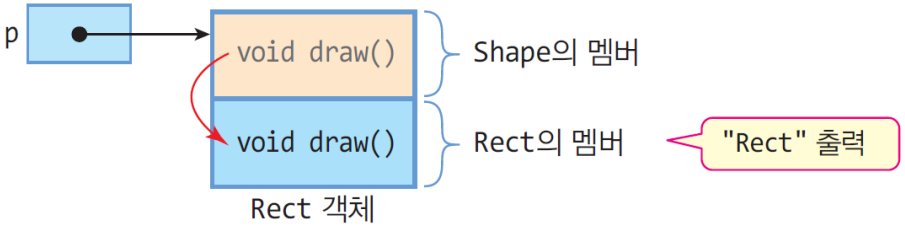

```
paint(new Shape());
```



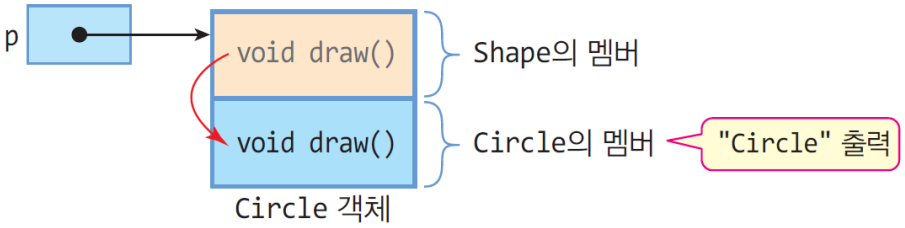

```
paint(new Line());
```




```
paint(new Rect());
```



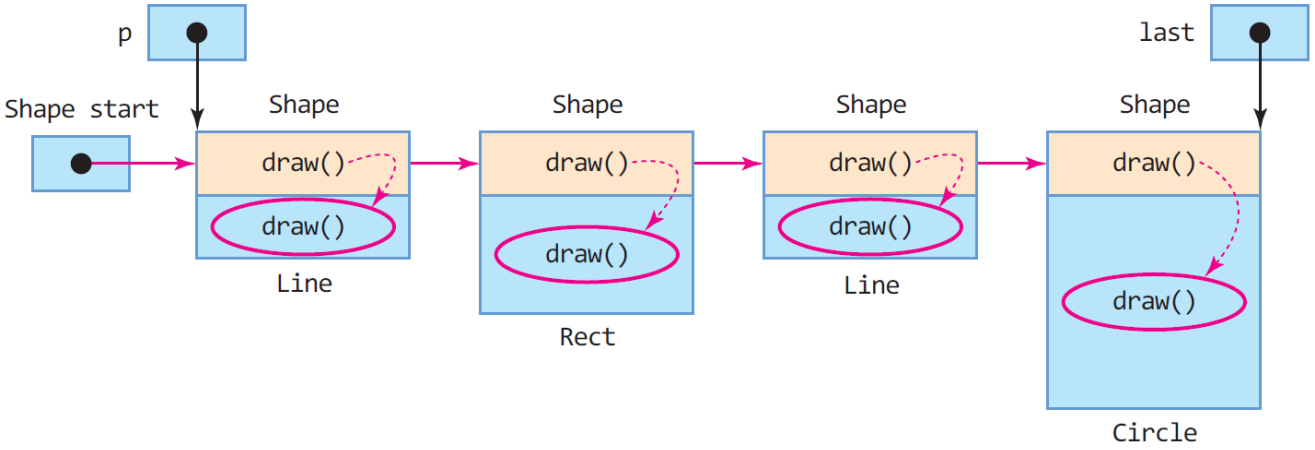

```
paint(new Circle());
```



오버라이딩 활용

```
// 예제 5-5의 Shape, Line, Rect, Circle 클래스 활용
public class UsingOverride {
    public static void main(String [] args) {
        Shape start, last, obj;
        // 링크드 리스트로 도형 생성하여 연결
        start = new Line(); // Line 객체 연결
        last = start;
        obj = new Rect();
        last.next = obj; // Rect 객체 연결
        last = obj;
        obj = new Line(); // Line 객체 연결
        last.next = obj;
        last = obj;
        obj = new Circle(); // Circle 객체 연결
        last.next = obj;
        // 모든 도형 출력
        Shape p = start;
        while(p != null) {
            p.draw();
            p = p.next;
        }
    }
}
```

Line
Rect
Line
Circle



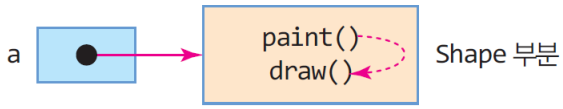
동적 바인딩

- 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
```

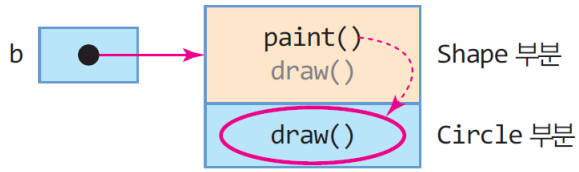



```
class Shape {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Shape");
    }
}
public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Circle");
    }
    public static void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

동적 바인딩

실행 결과

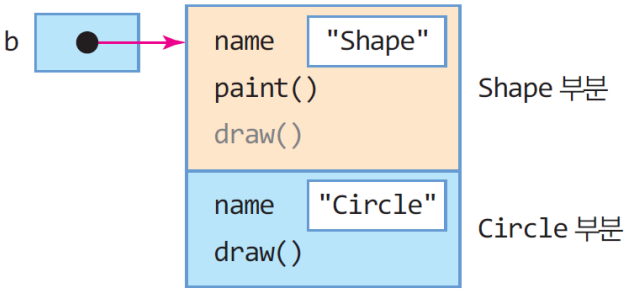
```
Circle
```



오버라이딩과 super 키워드



- super는 슈퍼 클래스의 멤버를 접근할 때 사용되는 레퍼런스
- 서브 클래스에서만 사용
- 슈퍼 클래스의 메소드 호출
- 컴파일러는 super의 접근을 **정적 바인딩**으로 처리




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

정적 바인딩

실행 결과

```
Shape
Circle
```

예제 5-6 : 메소드 오버라이딩



게임에서 무기를 표현하는 Weapon 클래스를 만들고 살상능력을 리턴하는 fire() 메소드를 작성하면 다음과 같다. fire()은 1을 리턴한다.

```
class Weapon {
    protected int fire() {
        return 1; // 무기는 기본적으로 한 명만 살상
    }
}
```

대포를 구현하기 위해 Weapon을 상속받는 Cannon 클래스를 작성하라. Cannon은 살상능력이 10이다. fire() 메소드를 이에 맞게 오버라이딩하라. main()을 작성하여 오버라이딩을 테스트하라.

```
class Cannon extends Weapon {
    @Override
    protected int fire() { // 오버라이딩
        return 10; // 대포는 한 번에 10명을 살상
    }
}
```

```
public class Overriding {
    public static void main(String[] args) {
        Weapon weapon;
        weapon = new Weapon();
        System.out.println("기본 무기의 살상 능력은 " +
                                    weapon.fire());
        weapon = new Cannon();
        System.out.println("대포의 살상 능력은 " +
                                    weapon.fire());
    }
}
```

```
기본 무기의 살상 능력은 1
대포의 살상 능력은 10
```

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

추상 메소드와 추상 클래스



- 추상 메소드(abstract method)
  - 선언되어 있으나 구현되어 있지 않은 메소드, abstract로 선언

```
public abstract String getName();
public abstract void setName(String s);
```

  - 추상 메소드는 서브 클래스에서 오버라이딩하여 구현해야 함

- 추상 클래스(abstract class)의 2종류
  1. 추상 메소드를 하나라도 가진 클래스
     - 클래스 앞에 반드시 abstract라고 선언해야 함
  2. 추상 메소드가 하나도 없지만 abstract로 선언된 클래스

2 가지 종류의 추상 클래스 사례




```
// 1. 추상 메소드를 포함하는 추상 클래스

abstract class Shape { // 추상 클래스 선언
    public Shape() { }
    public void paint() { draw(); }
    abstract public void draw(); // 추상 메소드
}
```

```
// 2. 추상 메소드 없는 추상 클래스

abstract class MyComponent { // 추상 클래스 선언
    String name;
    public void load(String name) {
        this.name = name;
    }
}
```

추상 클래스는 객체를 생성할 수 없다




```
abstract class Shape {
   …
}

public class AbstractError {
   public static void main(String [] args) {
      Shape shape;
      shape = new Shape(); // 컴파일 오류. 추상 클래스 Shape의 객체를 생성할 수 없다.
      …
   }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Cannot instantiate the type Shape

        at chap5.AbstractError.main(AbstractError.java:4)
```

추상 클래스의 상속



- 추상 클래스의 상속 2 가지 경우

1. 추상 클래스의 단순 상속
   - 추상 클래스를 상속받아, 추상 메소드를 구현하지 않으면 추상 클래스 됨
   - 서브 클래스도 abstract로 선언해야 함

```
abstract class Shape { // 추상 클래스
   public Shape() { }
   public void paint() { draw(); }
   abstract public void draw(); // 추상 메소드
}
abstract class Line extends Shape { // 추상 클래스. draw()를 상속받기 때문
   public String toString() { return "Line"; }
}
```

2. 추상 클래스 구현 상속
   - 서브 클래스에서 슈퍼 클래스의 추상 메소드 구현(오버라이딩)
   - 서브 클래스는 추상 클래스 아님

추상 클래스의 구현 및 활용 예





Line, Rect, Circle은 추상클래스 Shape를 상속받아 만든 서브 클래스들로서, draw()를 오버라이딩하여 구현한 사례입니다. 그러므로 Line, Rect, Circle은 추상 클래스가 아니며 이들의 인스턴스를 생성할 수 있습니다.

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

추상 클래스로 수정

```
abstract class Shape {
    public abstract void draw();
}
```

```
class Line extends Shape {
    @Override
    public void draw() {
        System.out.println("Line");
    }
}
```

```
class Rect extends Shape {
    @Override
    public void draw() {
        System.out.println("Rect");
    }
}
```

```
class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Circle");
    }
}
```

drow()라고 하면 컴파일 오류가 발생.
추상 메소드 draw()를 구현하지 않았기 때문

추상 클래스의 용도



- 설계와 구현 분리
  - 슈퍼 클래스에서는 개념 정의
    - 서브 클래스마다 다른 구현이 필요한 메소드는 추상 메소드로 선언
  - 각 서브 클래스에서 구체적 행위 구현
    - 서브 클래스마다 목적에 맞게 추상 메소드 다르게 구현
- 계층적 상속 관계를 갖는 클래스 구조를 만들 때

예제 5-7 : 추상 클래스의 구현 연습



다음 추상 클래스 Calculator를 상속받은 GoodCalc 클래스를 구현하라.

```
abstract class Calculator {
        public abstract int add(int a, int b);
        public abstract int subtract(int a, int b);
        public abstract double average(int[] a);
}
```

예제 5-7 정답

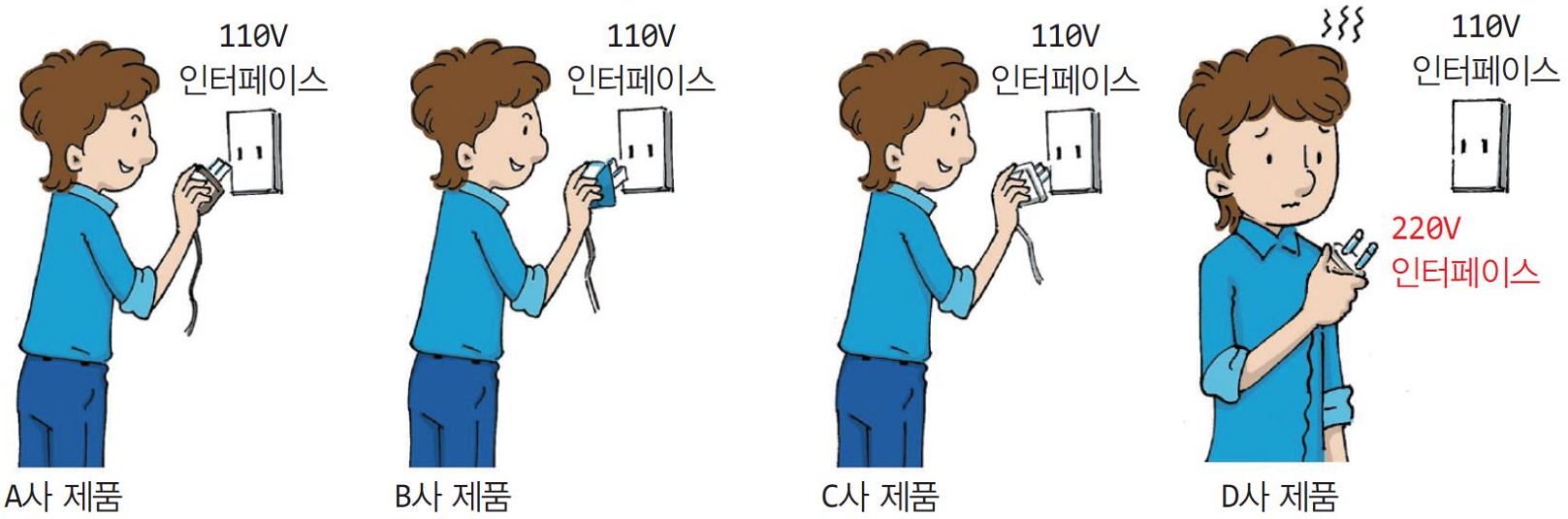



```
public class GoodCalc extends Calculator {
   @Override
   public int add(int a, int b) { // 추상 메소드 구현
      return a + b;
   }
   @Override
   public int subtract(int a, int b) { // 추상 메소드 구현
      return a - b;
   }
   @Override
   public double average(int[] a) { // 추상 메소드 구현
      double sum = 0;
      for (int i = 0; i <a.length; i++)
         sum += a[i];
      return sum/a.length;
   }

   public static void main(String [] args) {
      GoodCalc c = new GoodCalc();
      System.out.println(c.add(2,3));
      System.out.println(c.subtract(2,3));
      System.out.println(c.average(new int [] { 2,3,4 }));
   }
}
```

```
5
-1
3.0
```

실세계의 인터페이스





정해진 규격(인터페이스)에 맞기만 하면 연결 가능.
각 회사마다 구현 방법은 다름

정해진 규격(인터페이스)에 맞지 않으면 연결 불가

자바의 인터페이스



- 자바의 인터페이스
  - 클래스가 구현해야 할 메소드들이 선언되는 추상형
  - 인터페이스 선언
    - **interface** 키워드로 선언
    - Ex) public **interface** SerialDriver {…}
- 자바 인터페이스에 대한 변화
  - Java 7까지
    - 인터페이스는 상수와 추상 메소드로만 구성
  - Java 8부터
    - 상수와 추상메소드 포함
    - default 메소드 포함 (Java 8)
    - private 메소드 포함 (Java 9)
    - static 메소드 포함 (Java 9)
  - 여전히 인터페이스에는 **필드(멤버 변수) 선언 불가**

자바 인터페이스 사례




```
interface PhoneInterface { // 인터페이스 선언
    public static final int TIMEOUT = 10000; // 상수 필드 public static final 생략 가능
    public abstract void sendCall(); // 추상 메소드 public abstract 생략 가능
    public abstract void receiveCall();  // 추상 메소드 public abstract 생략 가능
    public default void printLogo() { // default 메소드 public 생략 가능
        System.out.println("** Phone **");
    };  // 디폴트 메소드
}
```

인터페이스의 구성 요소들의 특징



- 인터페이스의 구성 요소들
  - 상수
    - public만 허용, public static final 생략
  - 추상 메소드
    - public abstract 생략 가능
  - default 메소드
    - 인터페이스에 코드가 작성된 메소드
    - 인터페이스를 구현하는 클래스에 자동 상속
    - public 접근 지정만 허용. 생략 가능
  - private 메소드
    - 인터페이스 내에 메소드 코드가 작성되어야 함
    - 인터페이스 내에 있는 다른 메소드에 의해서만 호출 가능
  - static 메소드
    - public, private 모두 지정 가능. 생략하면 public

자바 인터페이스의 전체적인 특징



- 인터페이스의 객체 생성 불가

오류

```
new PhoneInterface(); // 오류. 인터페이스 PhoneInterface 객체 생성 불가
```

- 인터페이스 타입의 레퍼런스 변수 선언 가능

```
PhoneInterface galaxy; // galaxy는 인터페이스에 대한 레퍼런스 변수
```

- 인터페이스 구현
  - 인터페이스를 상속받는 클래스는 인터페이스의 모든 추상 메소드 반드시 구현
- 다른 인터페이스 상속 가능
- 인터페이스의 다중 상속 가능

인터페이스 구현



- 인터페이스의 추상 메소드를 모두 구현한 클래스 작성
  - implements 키워드 사용
  - 여러 개의 인터페이스 동시 구현 가능
- 인터페이스 구현 사례
  - PhoneInterface 인터페이스를 구현한 SamsungPhone 클래스

```
class SamsungPhone implements PhoneInterface { // 인터페이스 구현
   // PhoneInterface의 모든 메소드 구현
   public void sendCall() { System.out.println("띠리리리링"); }
   public void receiveCall() { System.out.println("전화가 왔습니다."); }

   // 메소드 추가 작성
   public void flash() { System.out.println("전화기에 불이 켜졌습니다."); }
}
```

  - SamsungPhone 클래스는 PhoneInterface의 default 메소드상속

예제 5-8 인터페이스 구현



PhoneInterface 인터페이스를 구현하고 flash() 메소드를 추가한 SamsungPhone 클래스를 작성하라.

```
** Phone **
띠리리리링
전화가 왔습니다.
전화기에 불이 켜졌습니다.
```

```
interface PhoneInterface { // 인터페이스 선언
    final int TIMEOUT = 10000; // 상수 필드 선언
    void sendCall(); // 추상 메소드
    void receiveCall(); // 추상 메소드
    default void printLogo() { // default 메소드
        System.out.println("** Phone **");
    }
}
class SamsungPhone implements PhoneInterface { // 인터페이스 구현
    // PhoneInterface의 모든 추상 메소드 구현
    @Override
    public void sendCall() {
        System.out.println("띠리리리링");
    }
    @Override
    public void receiveCall() {
        System.out.println("전화가 왔습니다.");
    }

    // 메소드 추가 작성
    public void flash() { System.out.println("전화기에 불이 켜졌습니다."); }
}
public class InterfaceEx {
    public static void main(String[] args) {
        SamsungPhone phone = new SamsungPhone();
        phone.printLogo();
        phone.sendCall();
        phone.receiveCall();
        phone.flash();
    }
}
```

인터페이스 상속



- 인터페이스가 다른 인터페이스 상속
  - extends 키워드 이용

```
interface MobilePhoneInterface extends PhoneInterface {
   void sendSMS();      // 새로운 추상 메소드 추가
   void receiveSMS();   // 새로운 추상 메소드 추가
}
```

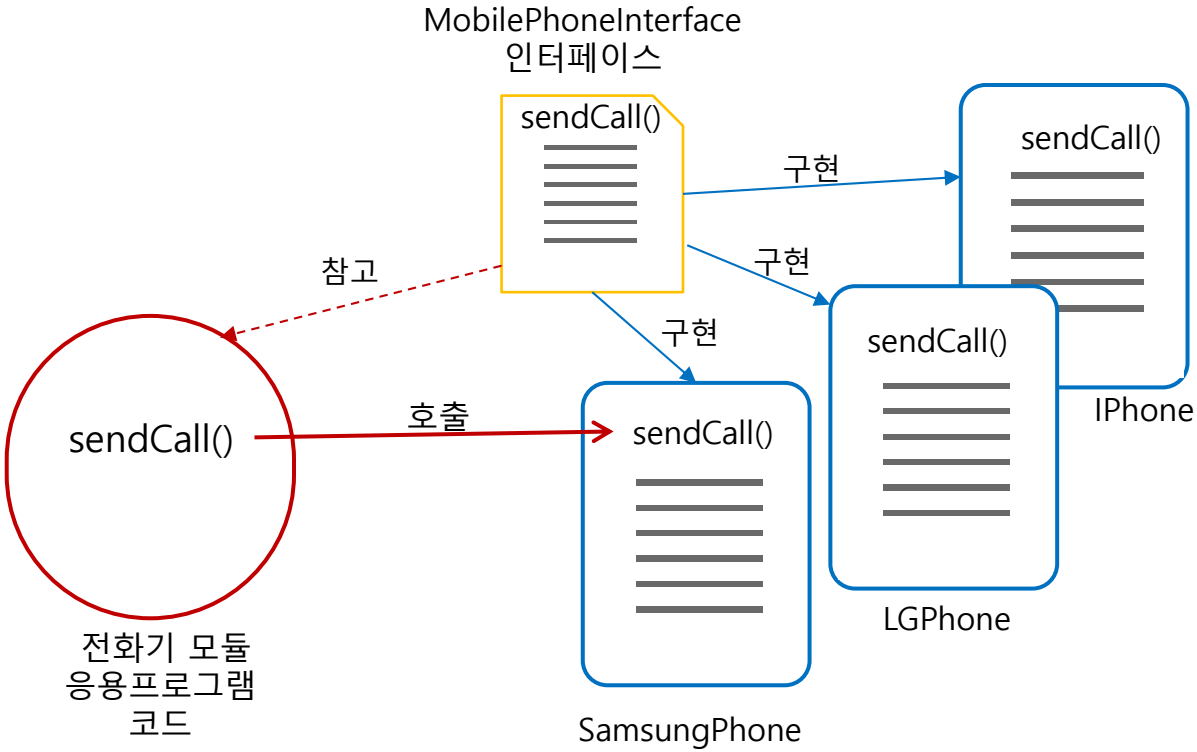
  - 다중 인터페이스 상속

```
interface MP3Interface {
   void play(); // 추상 메소드
   void stop(); // 추상 메소드
}

interface MusicPhoneInterface extends MobilePhoneInterface, MP3Interface {
   void playMP3RingTone(); // 새로운 추상 메소드 추가
}
```

인터페이스의 목적



인터페이스는 스펙을 주어 클래스들이 그 기능을 서로 다르게 구현할 수 있도록 하는 클래스의 규격 선언이며, 클래스의 다형성을 실현하는 도구이다



다중 인터페이스 구현



클래스는 하나 이상의 인터페이스를 구현할 수 있음

```
interface AIInterface {
   void recognizeSpeech(); // 음성 인식
   void synthesizeSpeech(); // 음성 합성
}

class AIPhone implements MobilePhoneInterface, AIInterface { // 인터페이스 구현
   // MobilePhoneInterface의 모든 메소드를 구현한다.
   public void sendCall() { ... }
   public void receiveCall() { ... }
   public void sendSMS() { ... }
   public void receiveSMS() { ... }

   // AIInterface의 모든 메소드를 구현한다.
   public void recognizeSpeech() { ... } // 음성 인식
   public void synthesizeSpeech() { ... } // 음성 합성

   // 추가적으로 다른 메소드를 작성할 수 있다.
   public int touch() { ... }
}
```

클래스에서 인터페이스의 메소드를 구현할 때 public을 생략하면 오류 발생

예제 5-9 : 인터페이스를 구현하고 동시에 클래스를 상속받는 사례




```
interface PhoneInterface { // 인터페이스 선언
    final int TIMEOUT = 10000; // 상수 필드 선언
    void sendCall(); // 추상 메소드
    void receiveCall(); // 추상 메소드
    default void printLogo() { // default 메소드
        System.out.println("** Phone **");
    }
}
interface MobilePhoneInterface extends PhoneInterface {
    void sendSMS();
    void receiveSMS();
}
interface MP3Interface { // 인터페이스 선언
    public void play();
    public void stop();
}
class PDA { // 클래스 작성
    public int calculate(int x, int y) {
        return x + y;
    }
}
// SmartPhone 클래스는 PDA를 상속받고,
// MobilePhoneInterface와 MP3Interface 인터페이스에 선언된 추상 메소드를 모두 구현한다.
class SmartPhone extends PDA implements
MobilePhoneInterface, MP3Interface {
    // MobilePhoneInterface의 추상 메소드 구현
    @Override
    public void sendCall() {
        System.out.println("따르릉따르릉~~");
    }
    @Override
    public void receiveCall() {
        System.out.println("전화 왔어요.");
    }
    @Override
    public void sendSMS() {
        System.out.println("문자갑니다.");
    }
    @Override
    public void receiveSMS() {
        System.out.println("문자왔어요.");
    }
    // MP3Interface의 추상 메소드 구현
    @Override
    public void play() {
        System.out.println("음악 연주합니다.");
    }
    @Override
    public void stop() {
        System.out.println("음악 중단합니다.");
    }
    // 추가로 작성한 메소드
    public void schedule() {
        System.out.println("일정 관리합니다.");
    }
}
public class InterfaceEx {
    public static void main(String [] args) {
    SmartPhone phone = new SmartPhone();
    phone.printLogo();
    phone.sendCall();
    phone.play();
    System.out.println("3과 5를 더하면 " +
        phone.calculate(3,5));
    phone.schedule();
    }
}
```

fe

```
** Phone **
따르릉따르릉~~
음악 연주합니다.
3과 5를 더하면 8
일정 관리합니다.
```

추상 클래스와 인터페이스 비교



- 유사점
  - 객체를 생성할 수 없고, 상속을 위한 슈퍼 클래스로만 사용
  - 클래스의 다형성을 실현하기 위한 목적
- 다른 점

비교	목적	구성
추상 클래스	추상 클래스는 서브 클래스에서 필요로 하는 대부분의 기능을 구현하여 두고 서브 클래스가 상속받아 활용할 수 있도록 하되, 서브 클래스에서 구현할 수밖에 없는 기능만을 추상 메소드로 선언하여, 서브 클래스에서 구현하도록 하는 목적(다형성)	• 추상 메소드와 일반 메소드 모두 포함 • 상수, 변수 필드 모두 포함
인터페이스	인터페이스는 객체의 기능을 모두 공개한 표준화 문서와 같은 것으로, 개발자에게 인터페이스를 상속받는 클래스의 목적에 따라 인터페이스의 모든 추상 메소드를 만들도록 하는 목적(다형성)	• 변수 필드(멤버 변수)는 포함하지 않음 • 상수, 추상 메소드, 일반 메소드, default 메소드, static 메소드 모두 포함 • protected 접근 지정 선언 불가 • 다중 상속 지원