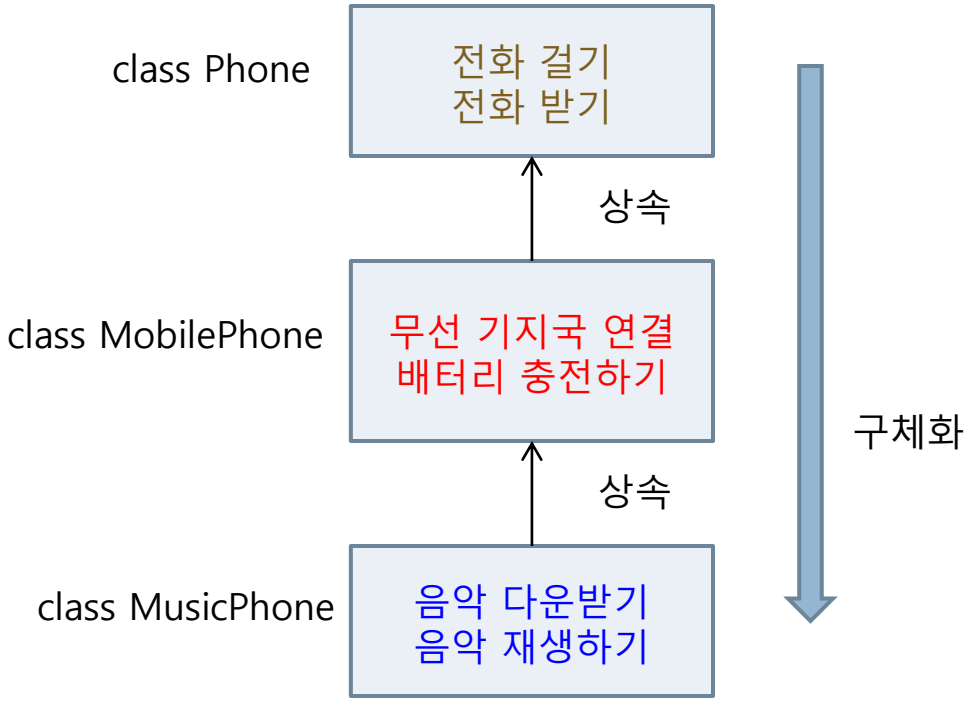
제 5 장 상속과 다형성

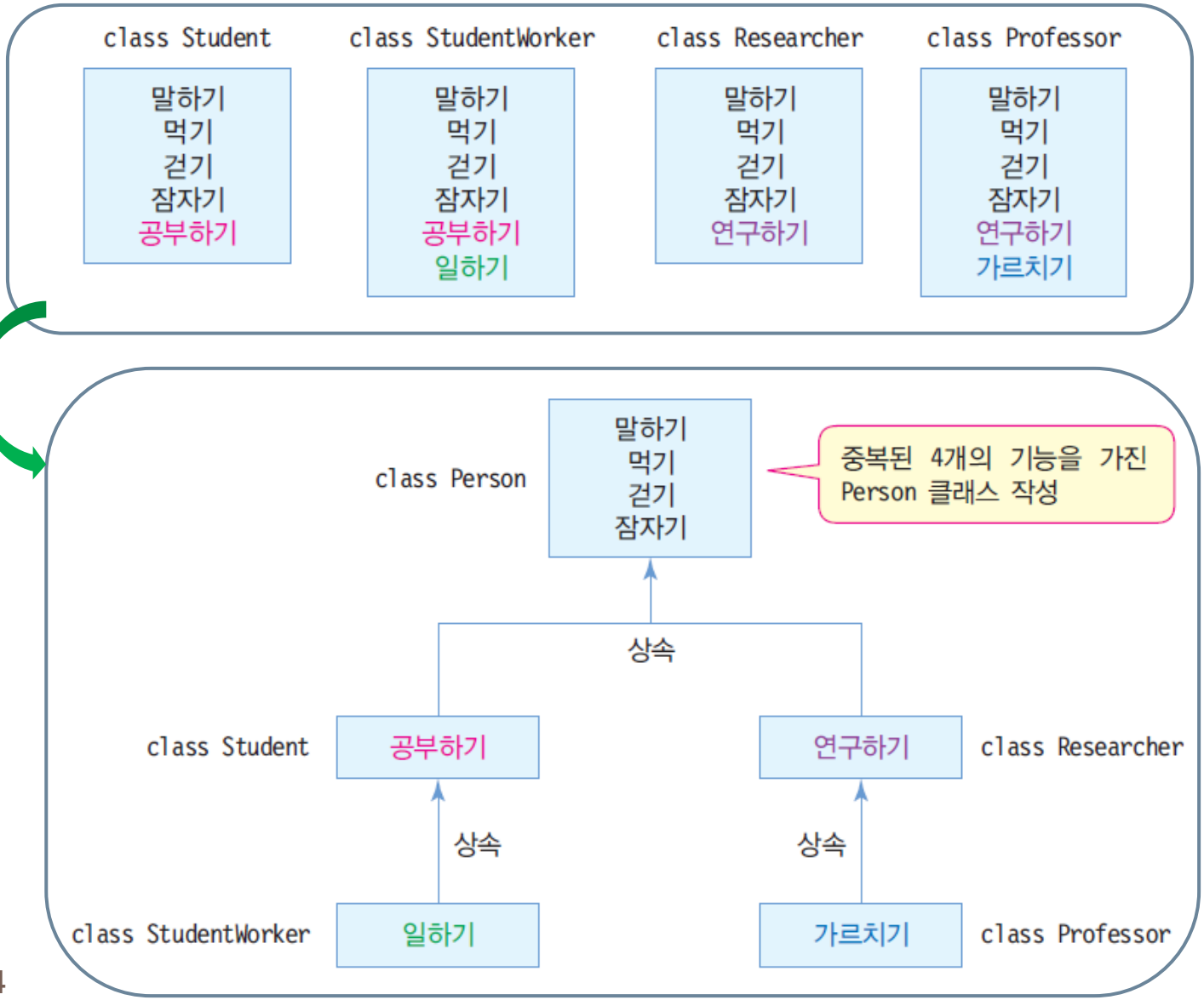
상속 (inheritance)

- 상속
  - 상위 클래스의 특성 (필드, 메소드)을 하위 클래스에 물려주는 것
  - 슈퍼 클래스 (superclass)
    - 특성을 물려주는 상위 클래스
  - 서브 클래스 (subclass)
    - 특성을 물려 받는 하위 클래스
    - 슈퍼 클래스에 자신만의 특성(필드, 메소드) 추가
    - 슈퍼 클래스의 특성(메소드)을 수정 : 구체적으로 오버라이딩이라고 부름
- 슈퍼 클래스에서 하위 클래스로 갈수록 구체적
  - 예) 폰 -> 모바일폰 -> 뮤직폰
- 상속을 통해 간결한 서브 클래스 작성
  - 동일한 특성을 재정의할 필요가 없어 서브 클래스가 간결해짐

상속 관계 예


class Phone
전화 걸기
전화 받기
상속
class MobilePhone
무선 기지국 연결
배터리 충전하기
상속
class MusicPhone
음악 다운받기
음악 재생하기
구체화

상속의 필요성



상속이 없는 경우
중복된 멤버를 가진
4 개의 클래스

상속을 이용한
경우 중복이 제거되고
간결해진 클래스 구조

클래스 상속과 객체

- 상속 선언

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

- 자바 상속의 특징
  - 다중 상속 지원하지 않음
    - 다수 개의 클래스를 상속받지 못함
  - 상속의 횟수는 무제한
  - 상속의 최상위 조상 클래스는 java.lang.Object 클래스
    - 모든 클래스는 자동으로 java.lang.Object를 상속받음

예제 5-1 : 클래스 상속 만들어 보기

(x,y)의 한 점을 표현하는 Point 클래스와 이를 상속받아 컬러 점을 표현하는 ColorPoint 클래스를 만들어보자.

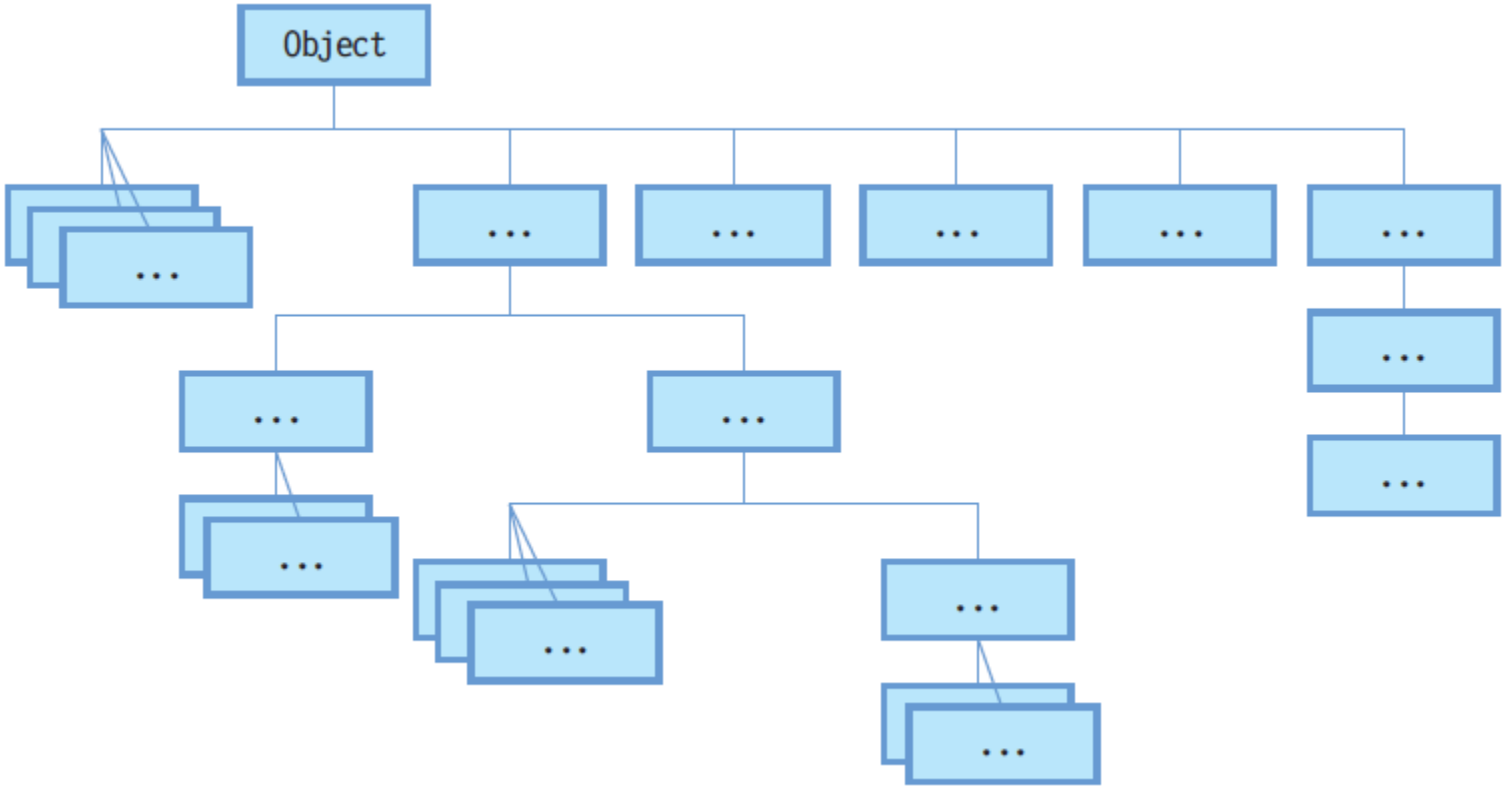
```
class Point {
   int x, y; // 한 점을 구성하는 x, y 좌표
   void set(int x, int y) {
      this.x = x; this.y = y;
   }
   void showPoint() { // 점의 좌표 출력
      System.out.println("(" + x + "," + y + ")");
   }
}
```

```
public class ColorPoint extends Point {
   // Point를 상속받은 ColorPoint 선언
   String color; // 점의 색
   void setColor(String color) {
      this.color = color;
   }
   void showColorPoint() { // 컬러 점의 좌표 출력
      System.out.print(color);
      showPoint(); // Point 클래스의 showPoint() 호출
   }
   public static void main(String [] args) {
      ColorPoint cp = new ColorPoint();
      cp.set(3,4); // Point 클래스의 set() 메소드 호출
      cp.setColor("red"); // 색 지정
      cp.showColorPoint(); // 컬러 점의 좌표 출력
   }
}
```

```
red(3,4)
```

자바의 클래스 계층 구조

자바에서는 모든 클래스는 반드시 java.lang.Object 클래스를 자동으로 상속받는다.



서브 클래스의 객체와 멤버 사용

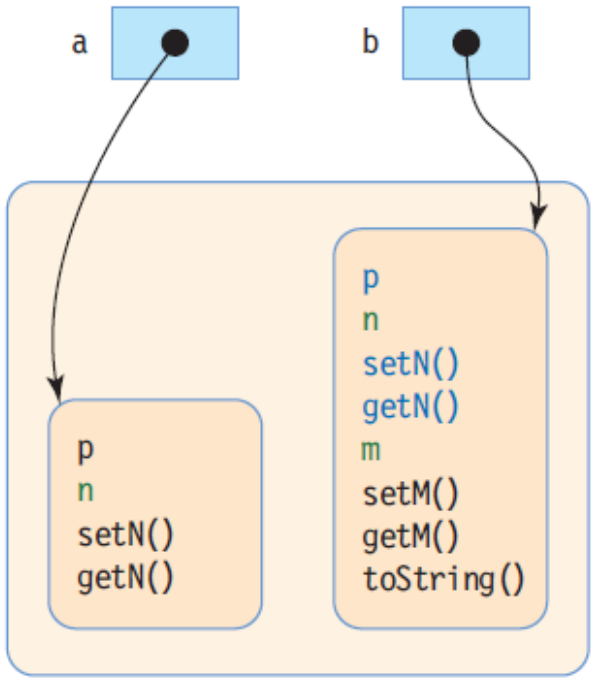
- 서브 클래스의 객체와 멤버 접근
  - 서브 클래스의 객체에는 슈퍼 클래스 멤버 포함
    - 슈퍼 클래스의 private 멤버는 상속되지 않음
      - 서브 클래스에서 직접 접근 불가
    - 슈퍼 클래스의 private 멤버는 슈퍼 클래스의 메소드를 통해 접근
  - 서브 클래스 객체에 슈퍼 클래스 멤버가 포함되므로 슈퍼 클래스 멤버의 접근은 서브 클래스 멤버 접근과 동일

슈퍼 클래스와 서브 클래스의 객체 관계


```
public class A {
    public int p;
    private int n;
    public void setN(int n) {
        this.n = n;
    }
    public int getN() {
        return n;
    }
}
```

```
public class B extends A {
    private int m;
    public void setM(int m) {
        this.m = m;
    }
    public int getM() {
        return m;
    }
    public String toString() {
        String s = getN() + " " + getM();
        return s;
    }
}
```

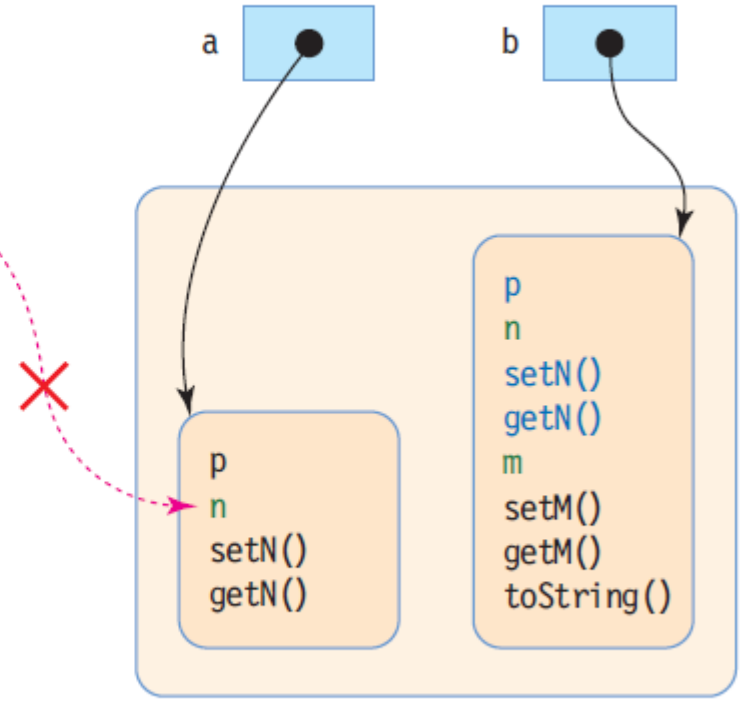
```
public static void main(String [] args) {
    A a = new A();
    B b = new B();
}
```



main() 실행 중 생성된 인스턴스

서브 클래스의 객체 멤버 접근


```
public class MemberAccessExample {
    public static void main(String [] args) {
        A a = new A();
        B b = new B();

        a.p = 5;
        a.n = 5; // n은 private 멤버, 컴파일 오류 발생

        b.p = 5;
        b.n = 5; // n은 private 멤버, 컴파일 오류 발생
        b.setN(10);
        int i = b.getN(); // i는 10

        b.m = 20; // m은 private 멤버, 컴파일 오류 발생
        b.setM(20);
        System.out.println(b.toString());
        // 화면에 10 200이 출력됨
    }
}
```



10 20

상속과 접근 지정자

- 자바의 접근 지정자 4 가지
  - public, protected, default, private
    - 상속 관계에서 주의할 접근 지정자는 private와 protected
- 슈퍼 클래스의 private 멤버
  - 슈퍼 클래스의 private 멤버는 모든 클래스에 접근 불허
- 슈퍼 클래스의 protected 멤버
  - 같은 패키지 내의 모든 클래스는 접근
  - 동일 패키지 여부와 상관없이 서브 클래스에서 슈퍼 클래스의 protected 멤버 접근 가능

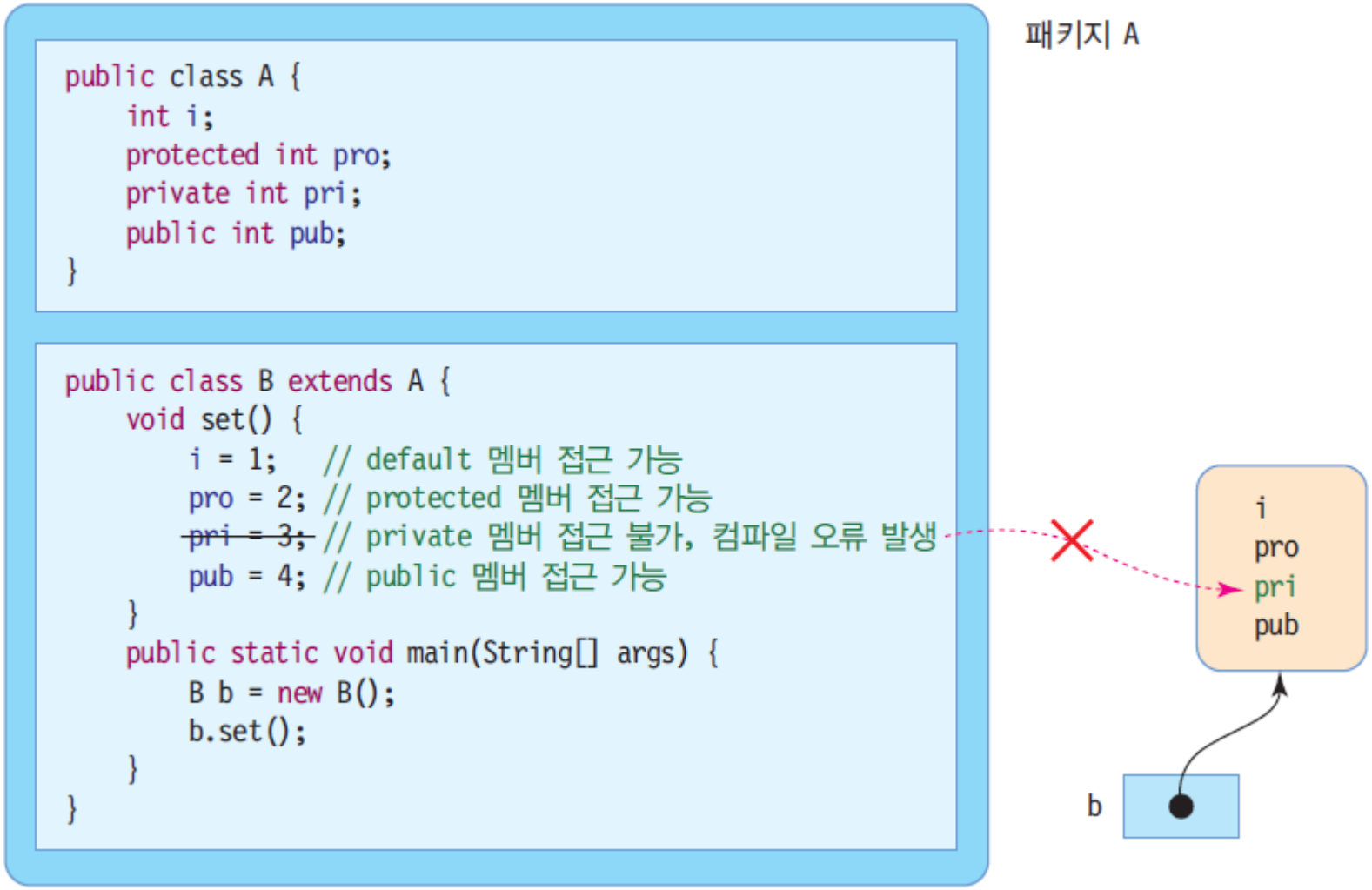
슈퍼 클래스 멤버의 접근 지정자

	default	private	protected	public
같은 패키지의 클래스	O	X	O	O
같은 패키지의 서브 클래스	O	X	O	O
다른 패키지의 클래스	X	X	X	O
다른 패키지의 서브 클래스	X	X	O	O

슈퍼클래스와 서브클래스가 같은 패키지에 있는 경우


```
public class A {
    int i;
    protected int pro;
    private int pri;
    public int pub;
}
```

```
public class B extends A {
    void set() {
        i = 1;   // default 멤버 접근 가능
        pro = 2; // protected 멤버 접근 가능
        pri = 3; // private 멤버 접근 불가, 컴파일 오류 발생
        pub = 4; // public 멤버 접근 가능
    }
    public static void main(String[] args) {
        B b = new B();
        b.set();
    }
}
```



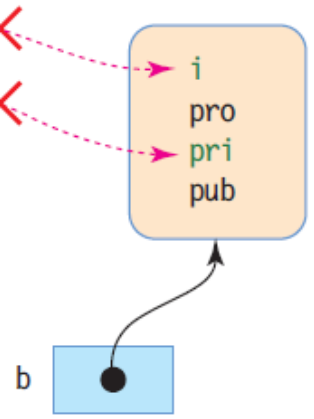
슈퍼클래스와 서브클래스가 서로 다른 패키지에 있는 경우

패키지 PA

```
public class A {
    int i;
    protected int pro;
    private int pri;
    public int pub;
}
```

패키지 PB

```
public class B extends A {
    void set() {
        i = 1;   // i는 default 멤버, 컴파일 오류 발생
        pro = 2; // protected 멤버 접근 가능
        pri = 3; // private 멤버 접근 불가, 컴파일 오류 발생
        pub = 4; // public 멤버 접근 가능
    }
    public static void main(String[] args) {
        B b = new B();
        b.set();
    }
}
```



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 getter와 setter를 통해서만 조작이 가능하다.

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

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

```
public class Student extends Person {
   void set() {
      age = 30;
      name = "홍길동";
      height = 175;
      setWeight(99);
   }
   public static void main(String[] args) {
      Student s = new Student();
      s.set();
   }
}
```

서브 클래스와 슈퍼 클래스의 생성자 호출 및 실행 관계

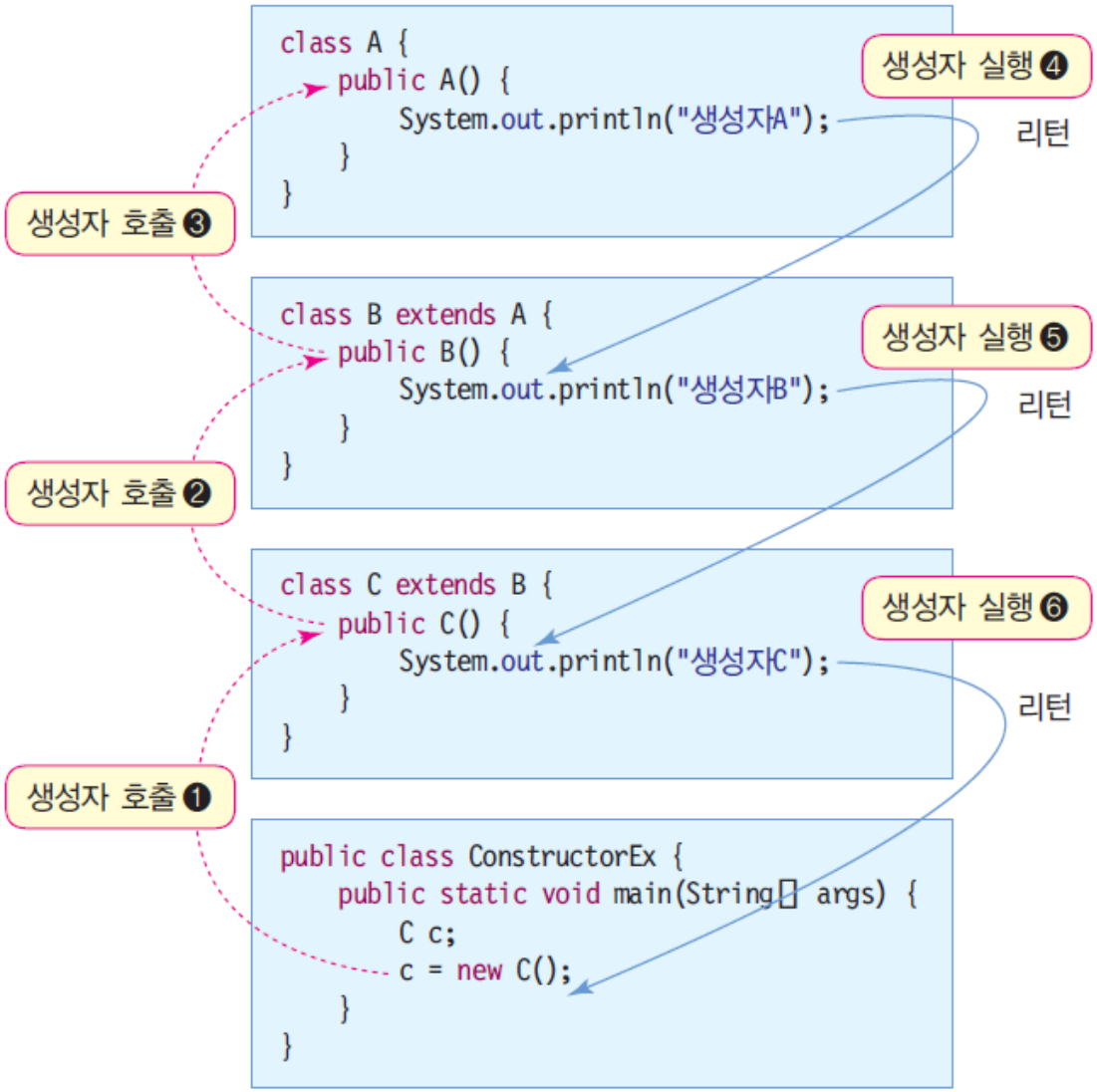
질문 1> 서브 클래스의 인스턴스가 생성될 때 서브 클래스의 생성자와 슈퍼 클래스의 생성자가 모두 실행되는가? 아니면 서브 클래스의 생성자만 실행되는가? 둘 다 실행된다.

질문 2> 서브 클래스의 인스턴스가 생성될 때 서브 클래스의 생성자와 슈퍼 클래스의 생성자의 실행 순서는 어떻게 되는가?

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
    - 서브클래스의 생성자가 먼저 호출되고, 서브 클래스의 생성자가 실행하기 전 슈퍼 클래스의 생성자 호출
  - 실행 순서
    - 슈퍼 클래스의 생성자가 먼저 실행된 후 서브 클래스의 생성자 실행

슈퍼클래스와 서브 클래스의 생성자간의 호출 및 실행 관계



예상 실행 결과는 ?

```
생성자A
생성자B
생성자C
```

위 코드는 모두 ConstructorEx.java 파일에 저장된다.

서브 클래스와 슈퍼 클래스의 생성자 짝 맞추기

- 슈퍼 클래스와 서브 클래스
  - 각각 여러 개의 생성자 가능
- 슈퍼 클래스와 서브 클래스의 생성자 사이의 짝 맞추기
  - 서브클래스의 객체 생성 시, 실행 가능한 슈퍼 클래스와 서브 클래스의 생성자 조합
    - 컴파일러는 서브 클래스의 생성자를 기준으로 아래 표와 같은 슈퍼 클래스의 생성자를 찾음
      - 경우 1, 3
    - 개발자가 서브 클래스의 생성자에 슈퍼 클래스의 짝을 지정하는 방법
      - 경우 2, 4
      - super() 키워드 이용

경우	1	2	3	4
서브 클래스	기본 생성자	기본 생성자	매개 변수를 가진 생성자	매개 변수를 가진 생성자
슈퍼 클래스	기본 생성자	매개 변수를 가진 생성자	기본 생성자	매개 변수를 가진 생성자

1: 슈퍼클래스(기본생성자),서브클래스(기본생성자)

아래 코드는 모두 ConstructorEx2.java 파일에 저장된다.

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

서브클래스의 생성자가 기본 생성자인 경우, 컴파일러는 자동으로 슈퍼클래스의 기본 생성자와 짝을 맺음

```
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
    }
}
```

생성자A
생성자B

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

컴파일러가 public B()에 대한 짝을 찾을 수 없음

```
class B extends A {
    public B() {
        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

3:서브 클래스에 매개변수 있는 생성자는 슈퍼클래스의기본생성자와 짝을 이룸

엮의 코드는 모두 ConstructorEx3.java 파일에 저장된다.

```
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);
    }
}
```

생성자A
매개변수생성자B

super()

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

super()를 이용한 사례

엮의 코드는 모두 ConstructorEx4.java 파일에 저장된다.

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

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

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

매개변수생성자A5
매개변수생성자B5

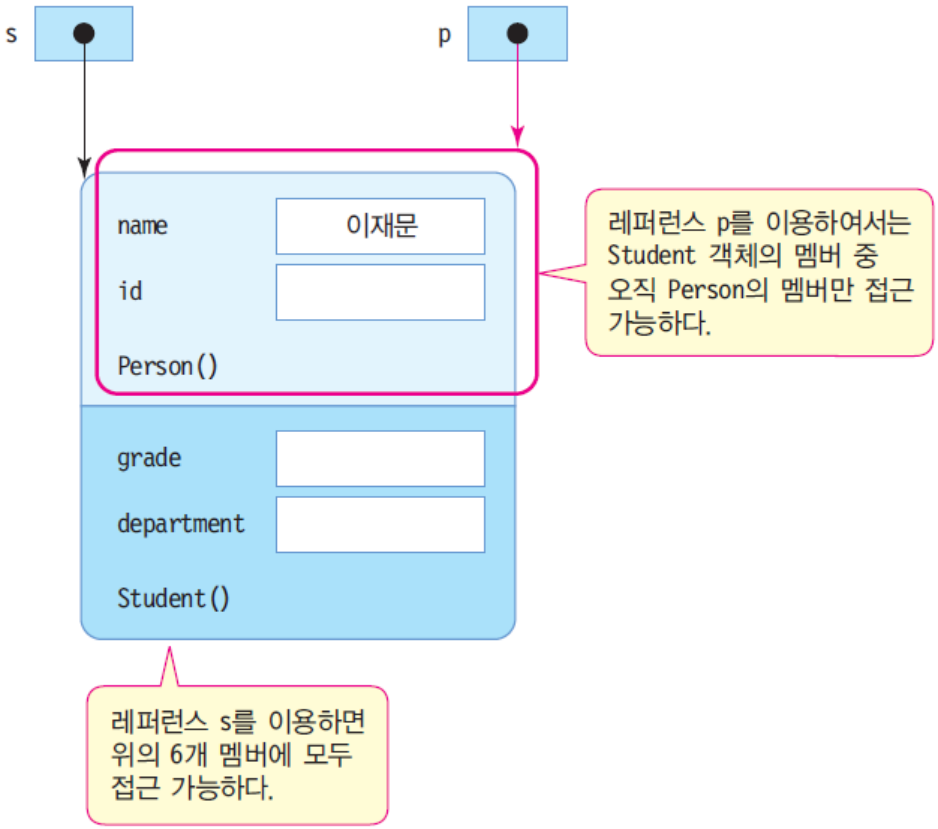
객체의 타입 변환

- 업캐스팅(upcasting)
  - 프로그램에서 이루어지는 자동 타입 변환
  - 서브 클래스의 레퍼런스 값을 슈퍼 클래스 레퍼런스에 대입
    - 슈퍼 클래스 레퍼런스가 서브 클래스 객체를 가리키게 되는 현상
    - 객체 내에 있는 모든 멤버를 접근할 수 없고 슈퍼 클래스의 멤버만 접근 가능

```
class Person {
}

class Student extends Person {
}

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

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 발생

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

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



이재문

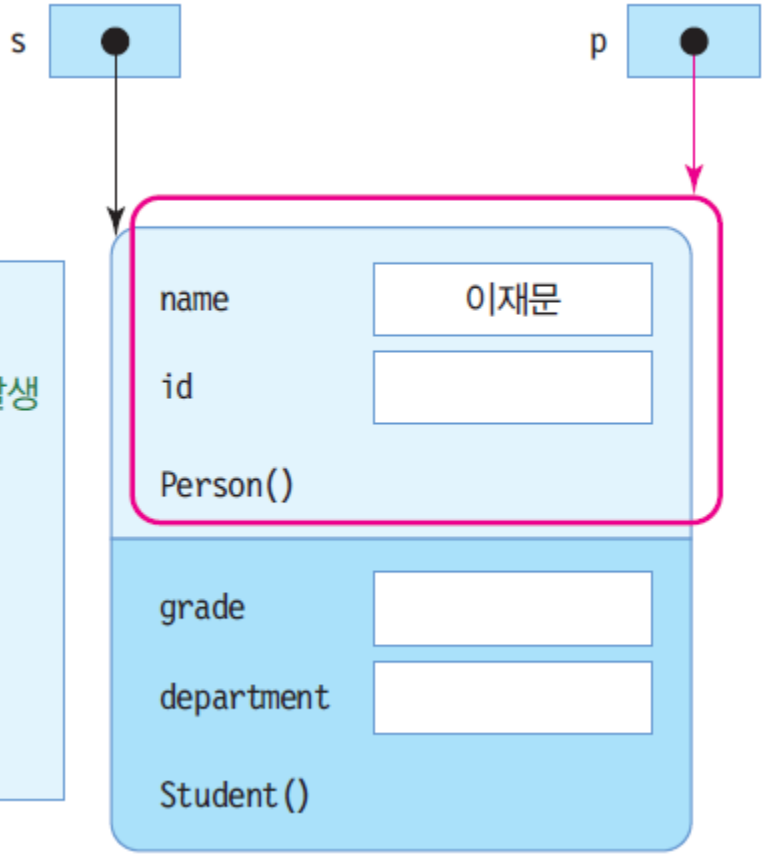
객체의 타입 변환

- 다운캐스팅(downcasting)
  - 슈퍼 클래스 레퍼런스를 서브 클래스 레퍼런스에 대입
  - 업캐스팅된 것을 다시 원래대로 되돌리는 것
  - 명시적으로 타입 지정

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

Student s = (Student)p; // 다운캐스팅, 강제타입변환
```

다운캐스팅 사례


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

        s = (Student)p; // 다운캐스팅

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



이재문

instanceof 연산자와 객체 구별

- 업캐스팅된 레퍼런스로 객체의 진짜 타입을 구분하기 어려움
  - 슈퍼 클래스는 여러 서브 클래스에 상속되기 때문
    - 슈퍼 클래스 레퍼런스로 서브 클래스 객체를 가리킬 수 있음
- instanceof 연산자
  - instanceof 연산자
    - 레퍼런스가 가리키는 객체의 진짜 타입 식별
  - 사용법

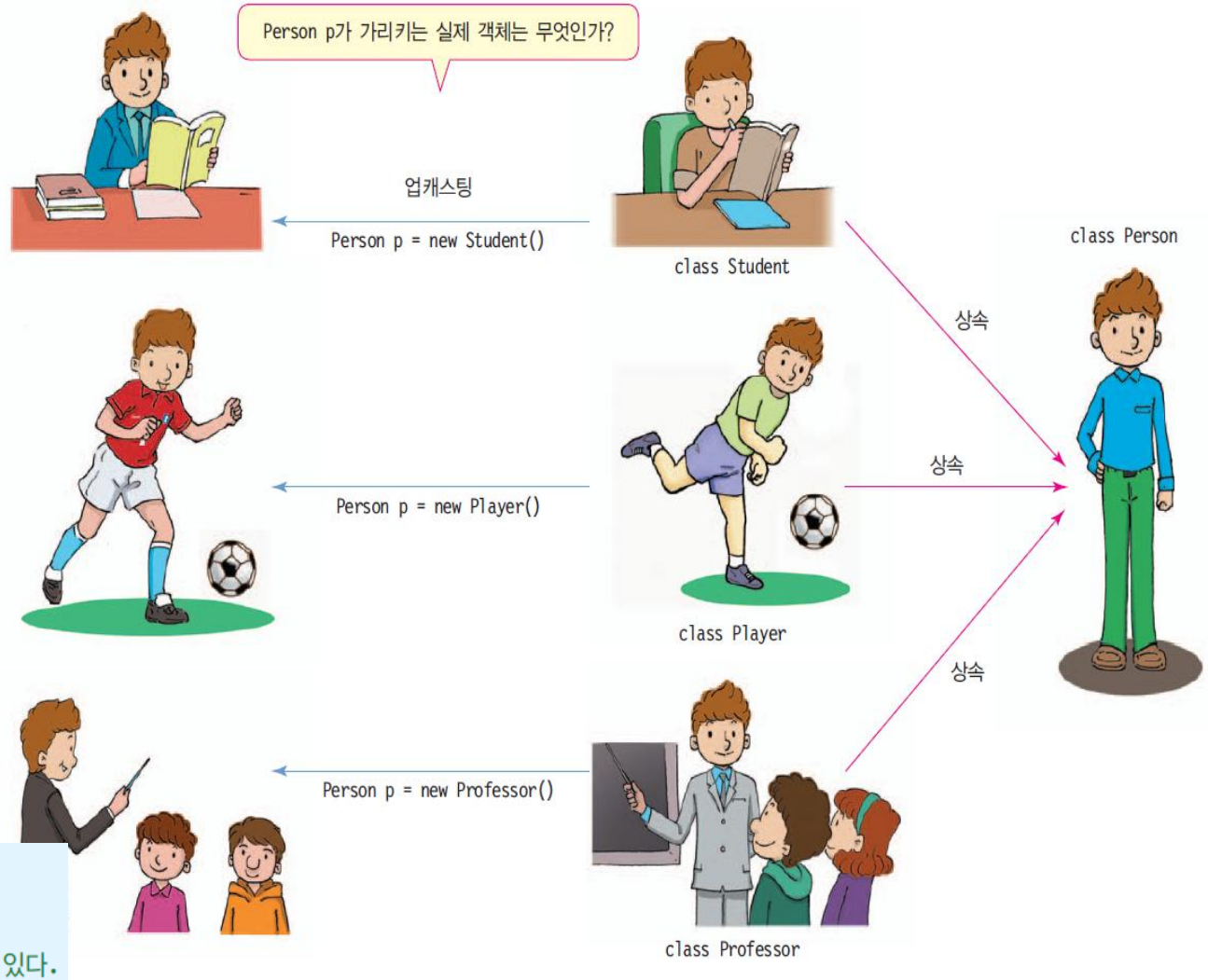
객체레퍼런스 **instanceof** 클래스타입

연산의 결과 : true/false의 불린 값

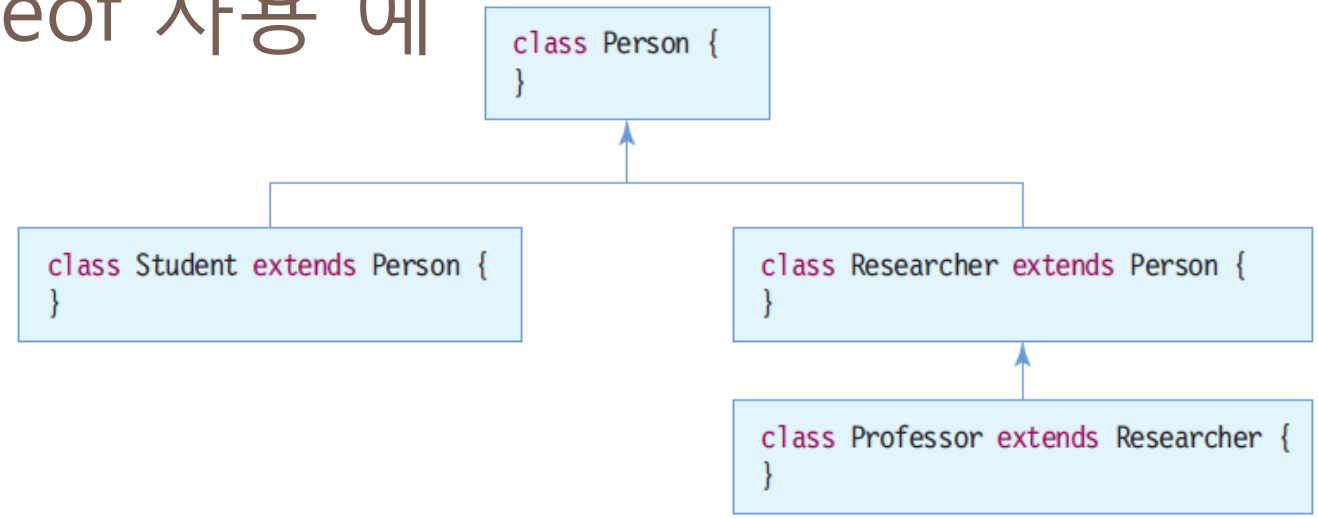
업캐스팅된 객체의 실제 타입은 무엇?

```
class Person {
.....
}
class Student extends Person {
.....
}
class Player extends Person {
.....
}
class Professor extends Person {
.....
}
Person p = new Person();
Person p = new Student();    // 업캐스팅
Person p = new Player();     // 업캐스팅
Person p = new Professor(); // 업캐스팅
```

```
void f(Person p) {
// p가 가리키는 객체가 Person 타입일 수도 있고,
// Student, Player, Professor 타입이 될 수도 있다.
.....
}
```



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)      // kim은 Researcher 타입이기도 하므로 true
if (lee instanceof Professor)       // lee는 Professor 타입이 아니므로 false
if ("java" instanceof String)       // "java"는 String 타입의 인스턴스이므로 true
if (3 instanceof int)      // 문법 오류. instanceof는 객체에 대한 레퍼런스에만 사용
```

예제 5-3 : instanceof를 이용한 객체 구별

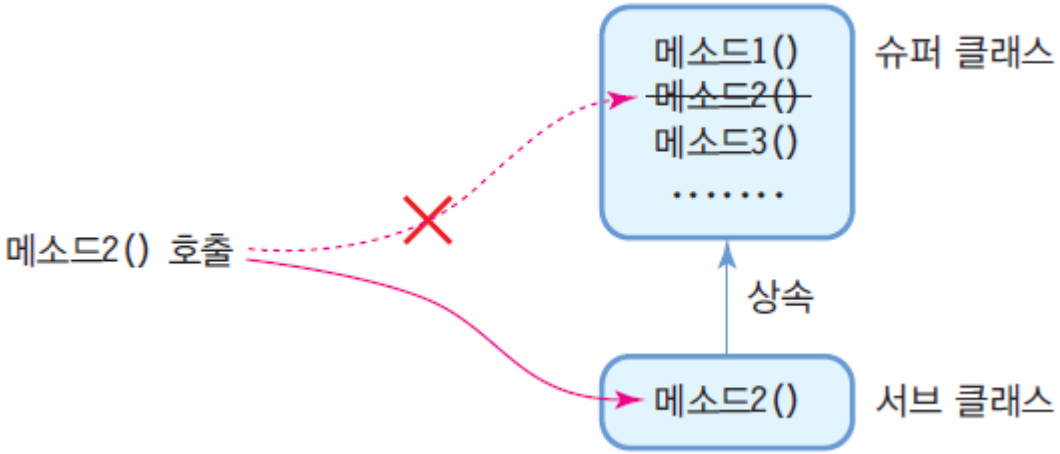
instanceof를 이용하여 객체의 타입을 구별하는 예를 만들어보자.

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

public class InstanceofExample {
   public static void main(String[] args) {
      Person jee= new Student();
      Person kim = new Professor();
      Person lee = new Researcher();
      if (jee instanceof Student) // jee는 Student 타입이므로 true
         System.out.println("jee는 Student 타입");
      if (jee instanceof Researcher) // jee는 Researcher 타입이 아니므로 false
         System.out.println("jee는 Researcher 타입");
      if (kim instanceof Student) // kim은 Student 타입이 아니므로 false
         System.out.println("kim은 Student 타입");
      if (kim instanceof Professor) // kim은 Professor 타입이므로 true
         System.out.println("kim은 Professor 타입");
      if (kim instanceof Researcher) // kim은 Researcher 타입이기도 하므로 true
         System.out.println("kim은 Researcher 타입");
      if (kim instanceof Person) // kim은 Person 타입이기도 하므로 true
         System.out.println("kim은 Person 타입");
      if (lee instanceof Professor) // lee는 Professor 타입이 아니므로 false
         System.out.println("lee는 Professor 타입");
      if ("java" instanceof String) // "java"는 String 타입의 인스턴스이므로 true
         System.out.println("₩"java₩"는 String 타입");
   }
}
```

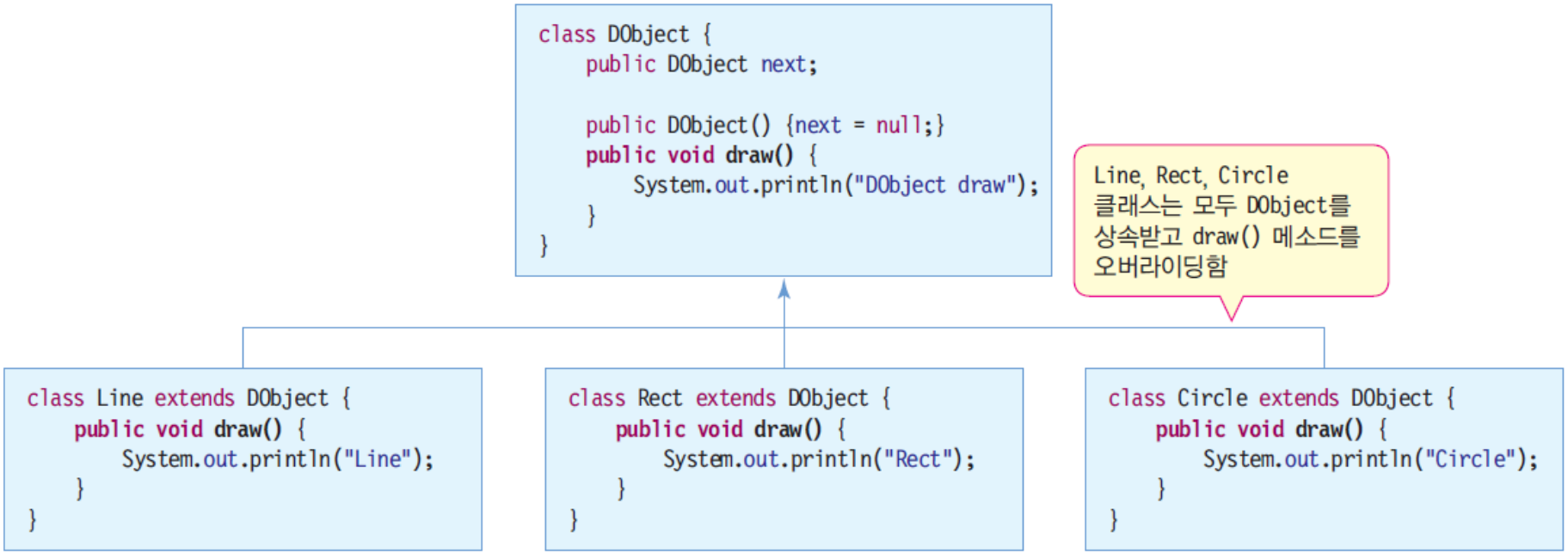
```
jee는 Student 타입
kim은 Professor 타입
kim은 Researcher 타입
kim은 Person 타입
"java"는 String 타입
```

메소드 오버라이딩

- 메소드 오버라이딩(Method Overriding)
  - 슈퍼 클래스의 메소드를 서브 클래스에서 재정의하는 것
    - 슈퍼 클래스의 메소드 이름, 메소드 인자 타입 및 개수, 리턴 타입 등 모든 것 동일하게 정의
      - 이 중 하나라도 다르면 메소드 오버라이딩 실패
  - "메소드 무시하기"로 번역되기도 함
  - 동적 바인딩 발생
    - 오버라이딩된 메소드가 무조건 실행되도록 동적 바인딩 됨

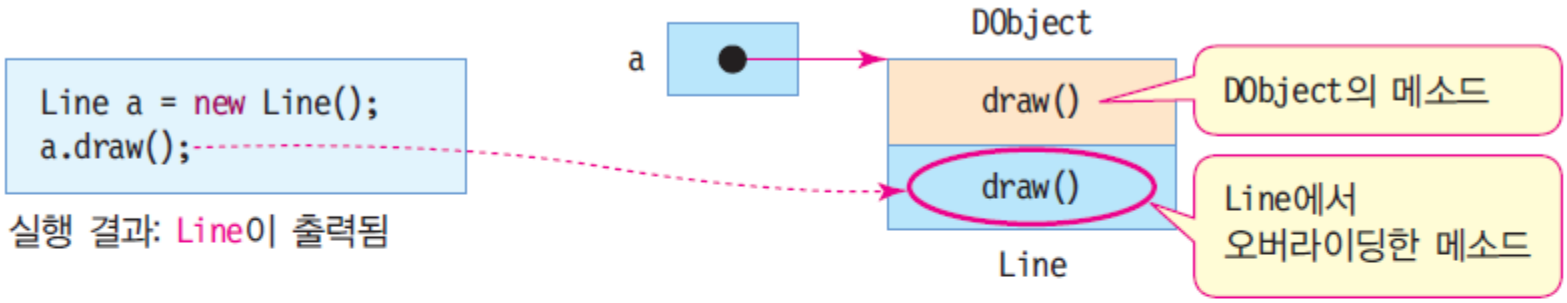


메소드 오버라이딩 사례

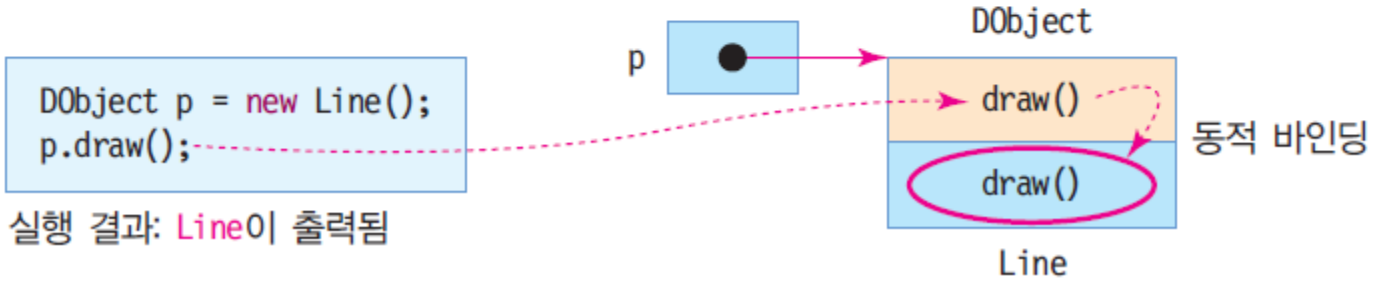

class DObject {
public DObject next;
public DObject() {next = null;}
public void draw() {
System.out.println("DObject draw");
}
}
Line, Rect, Circle 클래스는 모두 DObject를 상속받고 draw() 메소드를 오버라이딩함
class Line extends DObject {
public void draw() {
System.out.println("Line");
}
}
class Rect extends DObject {
public void draw() {
System.out.println("Rect");
}
}
class Circle extends DObject {
public void draw() {
System.out.println("Circle");
}
}

서브 클래스 객체와 오버라이딩된 메소드 호출

(1) 서브 클래스 레퍼런스로 오버라이딩된 메소드 호출



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



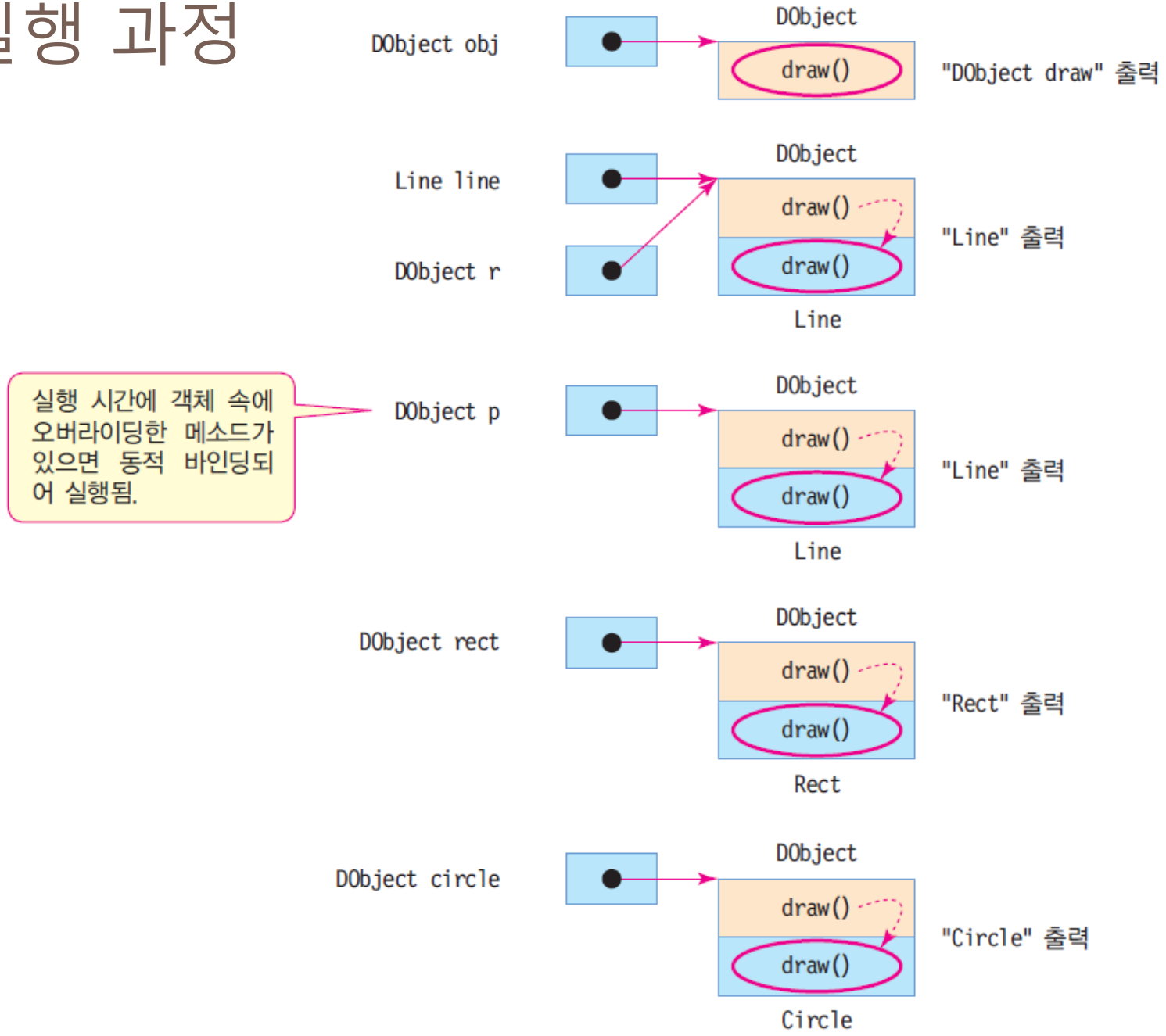
예제 5-4 : 메소드 오버라이딩 만들기


```
class DObject {
   public DObject next;

   public DObject() { next = null;}
    public void draw() {
      System.out.println("DObject draw");
   }
}

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

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

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

```
public class MethodOverringEx {
   public static void main(String[] args) {
      DObject obj = new DObject();
      Line line = new Line();
      DObject p = new Line();
      DObject r = line;

      obj.draw(); // DObject.draw() 메소드 실행. "DObject draw" 출력
      line.draw(); // Line.draw() 메소드 실행. "Line" 출력
      p.draw(); // 오버라이딩된 메소드 Line.draw() 실행, "Line" 출력
      r.draw(); // 오버라이딩된 메소드 Line.draw() 실행, "Line" 출력

      DObject rect = new Rect();
      DObject circle = new Circle();
      rect.draw(); // 오버라이딩된 메소드 Rect.draw() 실행, "Rect" 출력
      circle.draw(); // 오버라이딩된 메소드 Circle.draw() 실행, "Circle" 출력
   }
}
```

```
DObject draw
Line
Line
Line
Rect
Circle
```

예제 실행 과정


DObject obj
DObject
draw()
"DObject draw" 출력
Line line
DObject r
DObject
draw()
draw()
"Line" 출력
Line
실행 시간에 객체 속에 오버라이딩한 메소드가 있으면 동적 바인딩되어 실행됨.
DObject p
DObject
draw()
draw()
"Line" 출력
Line
DObject rect
DObject
draw()
draw()
"Rect" 출력
Rect
DObject circle
DObject
draw()
draw()
"Circle" 출력
Circle

메소드 오버라이딩 조건

1. 반드시 슈퍼 클래스 메소드와 동일한 이름, 동일한 호출 인자, 반환 타입을 가져야 한다.
2. 오버라이딩된 메소드의 접근 지정자는 슈퍼 클래스의 메소드의 접근 지정자 보다 좁아질 수 없다.

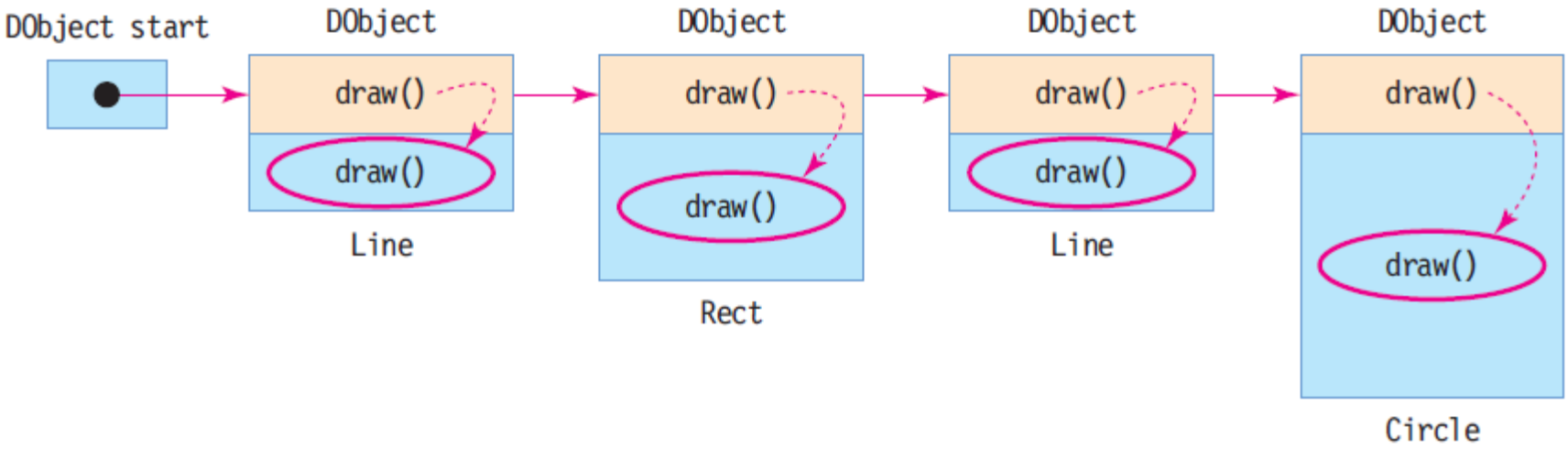
   public > protected > private 순으로 지정 범위가 좁아진다.
3. 반환 타입만 다르면 오류
4. static, private, 또는 final 메소드는 오버라이딩 될 수 없다.

```
class Person {
        String name;
        String phone;
        static int ID;

        public void setName(String s) {
                name = s;
        }
        public String getPhone() {
                return phone;
        }
        public static int getID() {
                return ID;
        }
}
class Professor extends Person {
        protected void setName(String s) {// 2번 조건위배
        }
        public String getPhone() {          // 1번 조건 성공
                return phone;
        }
        public void getPhone(){             // 3번 조건 위배
        }
        public int getID() {// 4번 조건 위배
        }
}
```

오버라이딩 활용

```
public static void main(String [] args) {
    DObject start, n, obj;

    // 링크드 리스트로 도형 생성하여 연결하기
    start = new Line(); //Line 객체 연결
    n = start;
    obj = new Rect();
    n.next = obj; //Rect객체 연결
    n = obj;
    obj = new Line(); // Line 객체 연결
    n.next = obj;
    n = obj;
    obj = new Circle(); // Circle 객체 연결
    n.next = obj;

    // 모든 도형 출력하기
    while(start != null) {
        start.draw();
        start = start.next;
    }
}
```

```
Line
Rect
Line
Circle
```



동적 바인딩

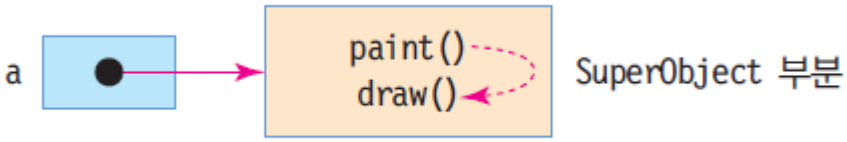
```
public class SuperObject {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Super Object");
    }
    public static void main(String [] args) {
        SuperObject a = new SuperObject();
        a.paint();
    }
}
```

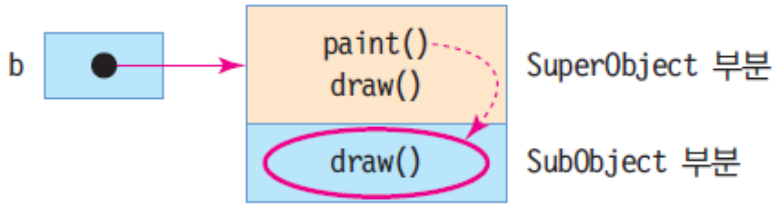
Super Object

```
class SuperObject {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Super Object");
    }
}
public class SubObject extends SuperObject {
    public void draw() {
        System.out.println("Sub Object");
    }
    public static void main(String [] args) {
        SuperObject b = new SubObject();
        b.paint();
    }
}
```

동적바인딩

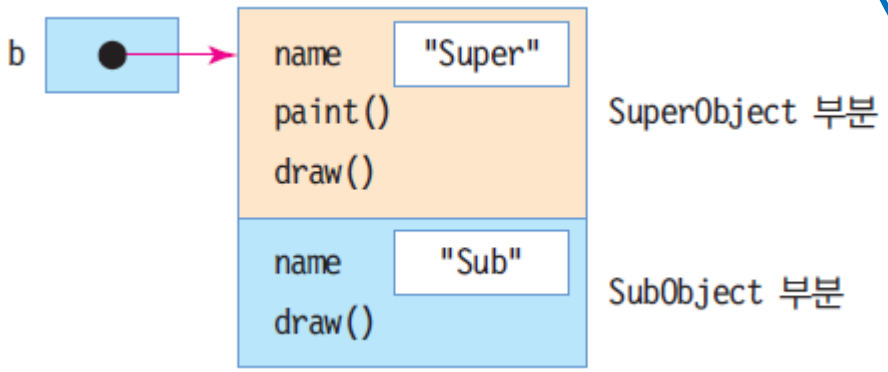
Sub Object





super 키워드

- super는 서브클래스에서 슈퍼 클래스의 멤버를 접근할 때 사용되는 슈퍼클래스 타입의 레퍼런스.
- 상속관계에 있는 서브 클래스에서만 사용됨
- 오버라이딩된 슈퍼 클래스의 메소드 호출 시 사용




```
class SuperObject {
   protected String name;
   public void paint() {
      draw();
   }
   public void draw() {
      System.out.println(name);
   }
}
public class SubObject extends SuperObject {
   protected String name;
   public void draw() {
      name = "Sub";
      super.name = "Super";
      super.draw();
      System.out.println(name);
   }
   public static void main(String [] args) {
      SuperObject b = new SubObject();
      b.paint();
   }
}
```

Super
Sub

예제 5-5 : 메소드 오버라이딩

Person을 상속받는 Professor라는 새로운 클래스를 만들고 Professor 클래스에서 getPhone() 메소드를 재정의하라. 그리고 이 메소드에서 슈퍼 클래스의 메소드를 호출하도록 작성하라.

```
class Person {
   String phone;
   public void setPhone(String phone) {
      this.phone = phone;
   }
   public String getPhone() {
      return phone;
   }
}

class Professor extends Person {
   public String getPhone() {
      return "Professor : " + super.getPhone();
   }
}
```

super.getPhone()은 아래 p.getPhone()과 달리 동적 바인딩이 일어나지 않는다.

```
public class Overriding {
   public static void main(String[] args) {
      Professor a = new Professor();
      a.setPhone("011-123-1234");
      System.out.println(a.getPhone());
      Person p = a;
      System.out.println(p.getPhone());
   }
}
```

동적 바인딩에 의해 Professor의 getPhone() 호출.

```
Professor : 011-123-1234
Professor : 011-123-1234
```

오버라이딩 vs. 오버로딩

비교요소	메소드 오버로딩	메소드 오버라이딩
정의	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 정의하여 사용의 편리성을 향상	슈퍼 클래스에 구현된 메소드를 무시하고 서브 클래스에서 새로운 기능의 메소드를 재정의 하고자 함
조건	메소드 이름은 반드시 동일함. 메소드의 인자의 개수나 인자의 타입이 달라야 성립	메소드의 이름, 인자의 타입, 인자의 개수, 인자의 리턴 타입 등이 모두 동일하여야 성립
바인딩	정적 바인딩. 컴파일 시에 중복된 메소드 중 호출되는 메소드 결정	동적 바인딩. 실행 시간에 오버라이딩된 메소드 찾아 호출

추상 메소드와 추상 클래스

- 추상 메소드(abstract method)
  - 선언되어 있으나 구현되어 있지 않은 메소드
  - 추상 메소드 선언
    - abstract 키워드로 선언
    - ex) public abstract int getValue();
  - 추상 메소드는 서브 클래스에서 오버라이딩하여 구현
- 추상 클래스(abstract class)
  1. 추상 메소드를 하나라도 가진 클래스
     - 클래스 앞에 반드시 abstract라고 선언해야 함
  2. 추상 메소드가 하나도 없지만 클래스 앞에 abstract로 선언한 경우

```
abstract class DObject {
   public DObject next;

   public DObject() { next = null;}
   abstract public void draw() ;
}
```

2 가지 종류의 추상 클래스 사례


```
// 추상 메소드를 가진 추상 클래스

abstract class DObject { // 추상 클래스 선언
   public DObject next;
   public DObject() { next = null; }
   abstract public void draw(); // 추상 메소드 선언
}
```

```
// 추상 메소드 없는 추상 클래스

abstract class Person { // 추상 클래스 선언
   public String name;
   public Person(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

추상 클래스 특성

- 추상 클래스의 객체는 생성할 수 없다.
- 추상 클래스 필요성
  - 계층적 상속 관계를 갖는 클래스 구조를 만들 때
  - 설계와 구현 분리
    - 슈퍼 클래스에서는 개념적 특징 정의
    - 서브 클래스에서 구체적 행위 구현
- 추상 클래스의 상속
  - 추상 클래스를 상속받아, 추상 메소드를 구현하지 않으면 서브 클래스도 추상 클래스 됨.
    - 서브 클래스도 abstract로 선언할 필요
  - 서브 클래스에서 추상 메소드를 구현하면 서브 클래스는 추상 클래스가 되지 않음

추상 클래스의 인스턴스 생성 불가


```
abstract class DObject { // 추상 클래스 선언
   public DObject next;

   public DObject() { next = null; }
   abstract public void draw(); // 추상 메소드 선언
}

public class AbstractError {
   public static void main(String [] args) {
      DObject obj;
      obj = new DObject(); // 컴파일 오류, 추상 클래스 DObject의 인스턴스를 생성할 수 없다.
      obj.draw(); // 컴파일 오류
   }
}
```

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

        at chap5.AbstractError.main(AbstractError.java:11)
```

추상 클래스의 활용 예


```
class DObject {
    public DObject next;

    public DObject() { next = null;}
     public void draw() {
        System.out.println("DObject draw");
     }
}
```

추상 클래스로 수정

```
abstract class DObject {
    public DObject next;

    public DObject() { next = null;}
    abstract public void draw();
}
```

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

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

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

예제 5-6 : 추상 클래스의 구현

다음의 추상 클래스 Calculator를 상속받는 GoodCalc 클래스를 독자 임의로 작성하라.

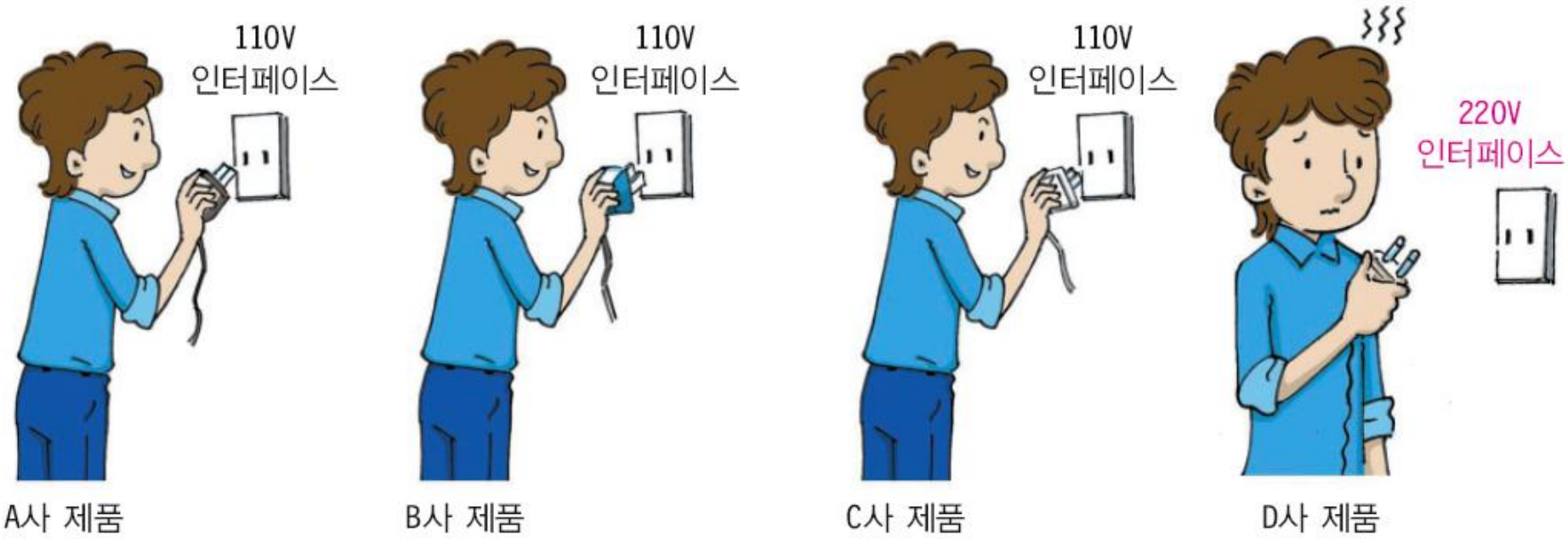
```
abstract class Calculator {
        public abstract int add(int a, int b);
        public abstract int subtract(int a, int b);
        public abstract double average(int[] a);
}
```

예제 5-6 정답


```
class GoodCalc extends Calculator {
	public int add(int a, int b) {
		return a+b;
	}
	public int subtract(int a, int b) {
		return a - b;
	}
	public double average(int[] a) {
		double sum = 0;
		for (int i = 0; i < a.length; i++)
			sum += a[i];
		return sum/a.length;
	}
	public static void main(String [] args) {
		Calculator 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 인터페이스와 인터페이스의 필요성



자바의 인터페이스

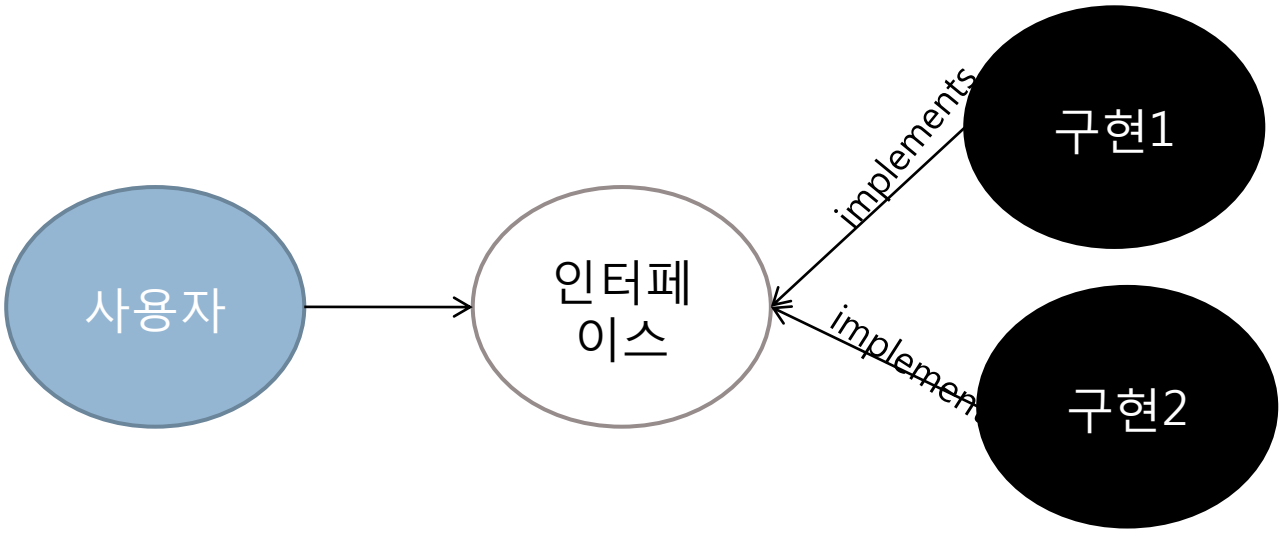
- 인터페이스(interface)
  - 모든 메소드가 추상 메소드인 클래스
  - 인터페이스는 상수와 메소드만 갖는다. 필드는 없음
- 인터페이스 선언
  - interface 키워드로 선언된 클래스
  - ex) public interface SerialDriver {...}
- 인터페이스의 특징
  - 메소드 선언에 abstract 키워드를 사용하지 않아도 됨
  - 인터페이스의 메소드 속성
    - public, static, final으로 가정되므로 키워드 생략 가능
  - 객체 생성 불가
  - 레퍼런스 변수 타입으로 사용 가능

자바 인터페이스 사례


```
public interface Clock {
   public static final int ONEDAY = 24; // 상수 필드 선언
   abstract public int getMinute();
   abstract public int getHour();
   abstract void setMinute(int i);
   abstract void setHour(int i);
}

public interface Car {
   int MAXIMUM_SPEED = 260; // 상수 필드 선언
   int moveHandle(int degree); // abstract 생략 가능
   int changeGear(int gear); // public 생략 가능
}
```

인터페이스의 필요성

- 인터페이스를 이용하여 다중 상속 구현
  - 클래스는 다중 상속 불가
- 인터페이스는 명세서와 같음
  - 구현은 블랙 박스와 같아 인터페이스의 사용자는 구현에 대해 알 필요가 없음
- 인터페이스만 정의하고 구현을 분리하여, 작업자가 다양한 구현을 할 수 있음



인터페이스 상속

- 인터페이스 간에도 상속 가능
  - 인터페이스 상속하여 확장된 인터페이스 작성 가능
- 다중 상속 허용

```
interface MobilePhone {
        public boolean sendCall();
        public boolean receiveCall();
        public boolean sendSMS();
        public boolean receiveSMS();
}

interface MP3 {
        public void play();
        public void stop();
}

interface MusicPhone extends MobilePhone, MP3 {
        public void playMP3RingTone();
}
```

인터페이스 구현

- 인터페이스 구현
  - implements 키워드 사용
  - 여러 개의 인터페이스 동시 구현 가능
  - 상속과 구현이 동시에 가능

```
interface USBMouseInterface {
   void mouseMove();
   void mouseClick();
}

public class MouseDriver implements USBMouseInterface { // 인터페이스 구현
   void mouseMove() { .... }
   void mouseClick() { ... }

   // 추가적으로 다른 메소드를 작성할 수 있다.
   int getStatus() { ... }
   int getButton() { ... }
}
```

인터페이스의 다중 구현


```
interface USBMouseInterface {
   void mouseMove();
   void mouseClick();
}

interface RollMouseInterface {
   void roll();
}

public class MouseDriver implements RollMouseInterface , USBMouseInterface {
   void mouseMove() { .... }
   void mouseClick() { ... }
   void roll() { ... }

   // 추가적으로 다른 메소드를 작성할 수 있다.
   int getStatus() { ... }
   int getButton() { ... }
}
```

추상 클래스와 인터페이스 비교

비교	내용
추상 클래스	• 일반 메소드 포함 가능 • 상수, 변수 포함 가능 • 모든 서브 클래스에 공통된 메소드가 있는 경우는 추상 클래스가 적합
인터페이스	• 모든 메소드가 추상 메소드 • 상수만 포함 가능 • 다중 상속 지원