상속과 다형성

514760
2022년 봄학기
4/5/2022
박경신

Private Constructor

- Private 생성자
  - 정적 멤버만 포함하는 클래스에서 일반적으로 사용
  - 클래스가 인스턴스화 될 수 없음을 분명히 하기 위해 private constructor를 사용

```
public class Counter {
    private Counter() { }
    public static int currentCount;
    public static int IncrementCount() { return ++currentCount; }
}
class TestCounter {
    public static void main(String[] args) {
        //Counter aCounter = new Counter(); // Error
        Counter.currentCount = 100;
        Counter.IncrementCount();
        System.out.println("count="+Counter.currentCount);
    }
}
```

Protected Constructor

- Protected 생성자
  - Protected 생성자는 추상클래스(abstract class)에서 사용을 권함.
  - 추상클래스에는 protected 또는 default 생성자를 정의함.
  - 추상클래스를 상속받은 파생클래스에서 추상클래스의 protected 생성자를 호출하여 초기화 작업을 수행함.

```
abstract class Shape {
    protected Shape(String name) { this.name = name; }
    private String name;
    public void print() { System.out.println(this.name); }
}
class Triangle extends Shape {    public Triangle(string name) { super(name); } }
class Rectangle extends Shape {    public Rectangle(string name) { super(name); } }
public class ShapeTest { public static void main(String[] args) {
    //Shape s= new Shape("도형"); // error Shape is abstract: cannot be instantiated
    Shape s = new Triangle("삼각형");
    s.print(); // 삼각형
    s = new Rectangle("직사각형");
    s.print(); // 직사각형
} }
```

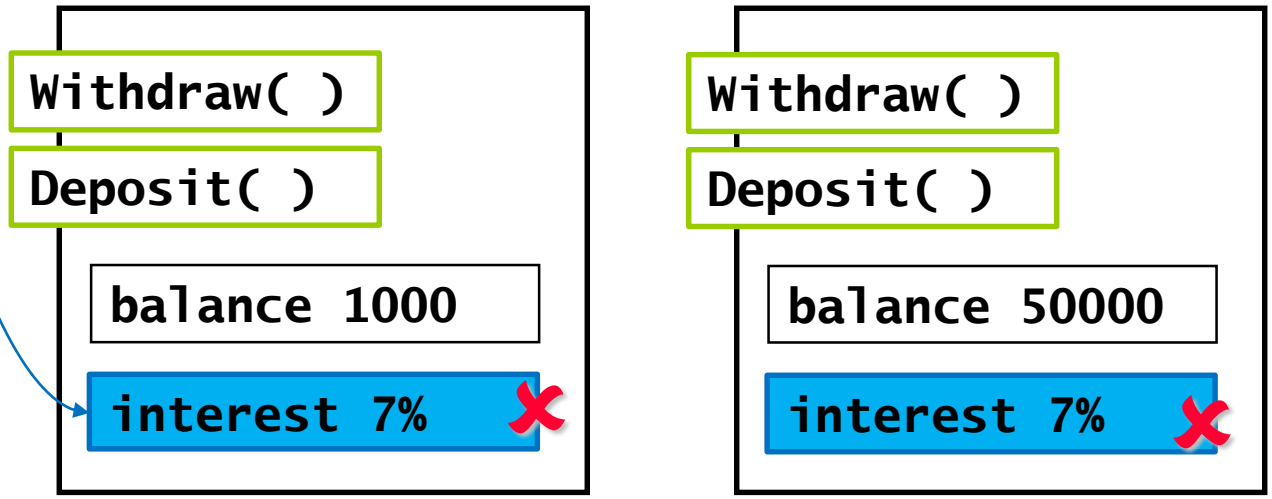
Static vs Instance Initializer Block

- Static Initializer Block
  - 클래스 로딩시 호출
  - instance 필드 메소드 사용 못함
  - static 변수 초기화에 사용
- Instance Initializer Block
  - 객체 생성시 호출
  - super 생성자 이후에 실행하고 생성자보다 먼저 실행
  - instance 필드 메소드에 접근가능
  - 모든 생성자의 공통 부분을 instance initializer block에 넣어줌

```
class StaticPoint {
      private static int[] data ;
      static    {
            data = new int[] { 1, 2, 3 };
      }
}
class InstancePoint {
     private int[] data ;
     {
          data = new int[] { 100, 200 };
     }
     public InstancePoint() {
          System.out.println("기본생성자");
     }
     public InstancePoint(int x) {
          System.out.println("생성자");
     }
}
```

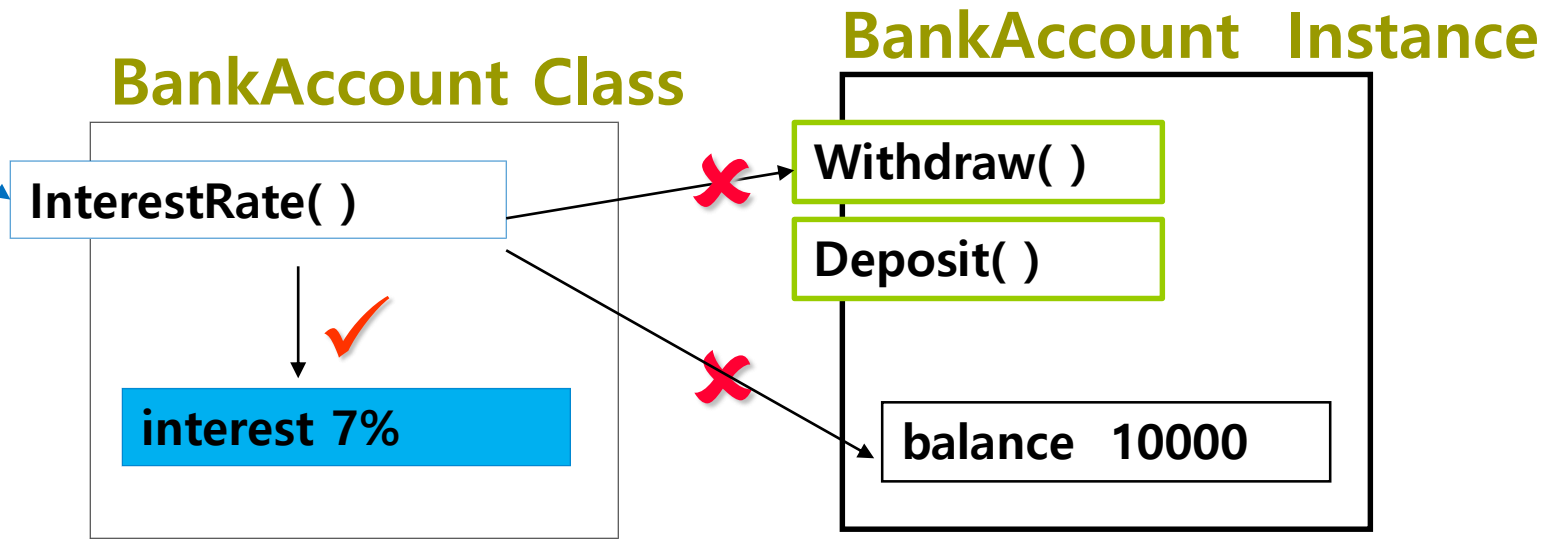
Instance vs Static Field

- Instance field
  - 객체의 현재상태를 저장할 수 있는 자료
  - 객체 생성시 메모리 공간 할당
- **Static(Class) field**
  - 전역 데이터, 공유 데이터로 클래스 로드 시 공간을 할당함
  - 클래스당 하나만 정적 필드가 할당
  - 객체의 개수를 카운트하거나 통계 값들을 저장하는 경우 사용함



Instance vs Static Method

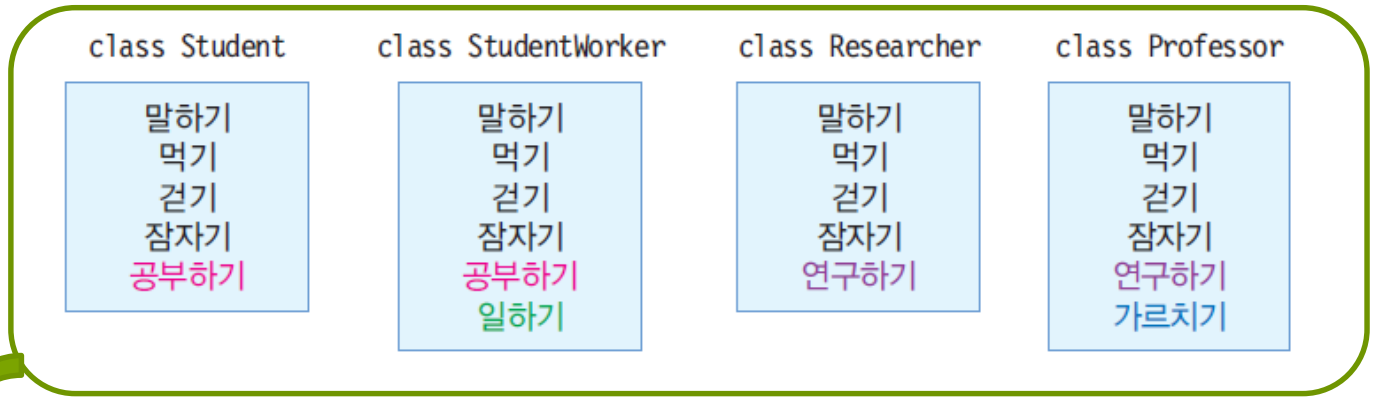
- Instance method
  - 객체 생성시 메모리 공간 할당
  - **정적 멤버(static field & method)도 사용 가능**
- **Static(Class) method**
  - 클래스 로딩시 할당, 객체 생성 없이 정적메소드 실행가능
  - 정적 멤버(static field & method)를 접근할 수 있는 메소드
  - **인스턴스 멤버(instance field & method) 사용 불가**
  - **this 사용 불가**



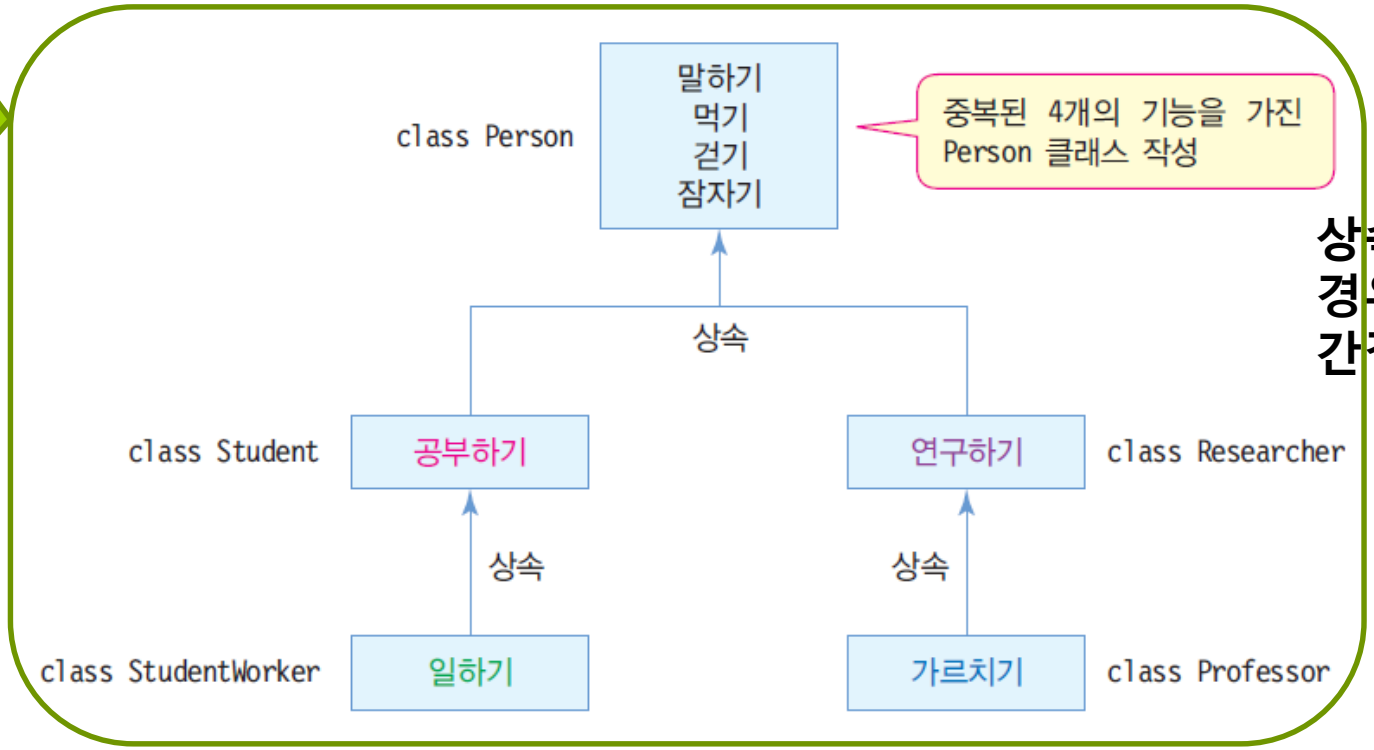
상속 (Inheritance)

- 상속
  - 상위 클래스의 특성 (필드, 메소드)을 하위 클래스에 물려주는 것
    - **상속**은 한 클래스가 기존 클래스의 **속성이나 기능(메소드)을 재사용(reuse)**하면서 자신의 새로운 속성이나 기능을 **추가해서 확장(extend)**하거나 기존 **기능을 정제하고 개선(refine)**하는 구현 방식
  - 부모클래스, 베이스클래스(base), 슈퍼클래스(superclass)
    - 특성을 물려주는 상위 클래스
  - 자식클래스, 파생클래스(derived), 서브클래스(subclass)
    - 특성을 물려 받는 하위 클래스
    - 슈퍼 클래스에 자신만의 특성(필드, 메소드) 추가
    - 슈퍼 클래스의 특성(메소드)을 수정 - 메소드오버라이딩(overriding)
  - 슈퍼 클래스에서 하위 클래스로 갈수록 구체적
    - 예) 폰 -> 모바일폰 -> 뮤직폰
  - 상속을 통해 간결한 서브 클래스 작성
    - 동일한 특성을 재정의할 필요가 없어 서브 클래스가 간결해짐

상속의 필요성



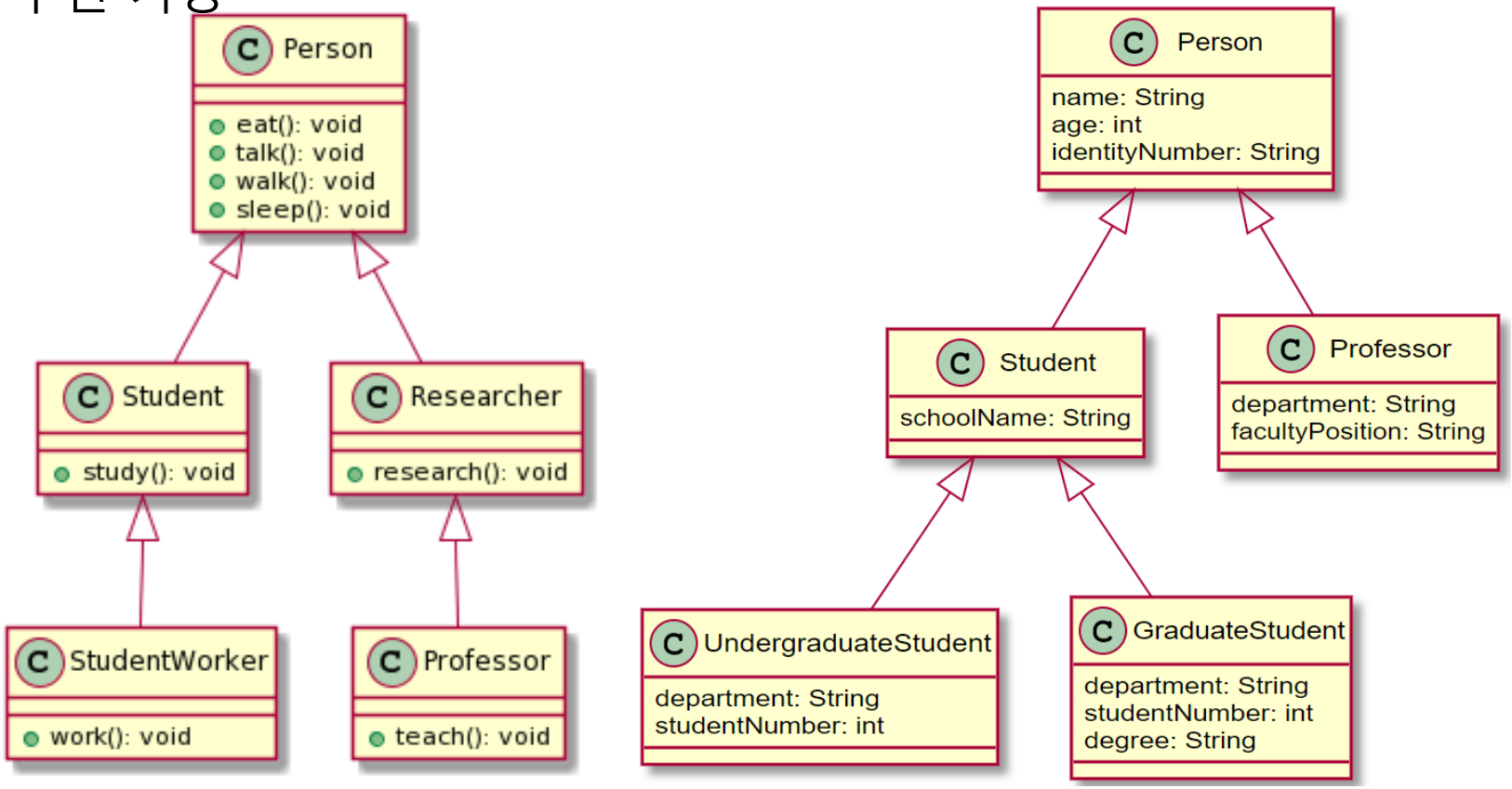
상속이 없는 경우
중복된 멤버를 가진
4 개의 클래스



상속을 이용한
경우 중복이 제거되고
간결해진 클래스 구조

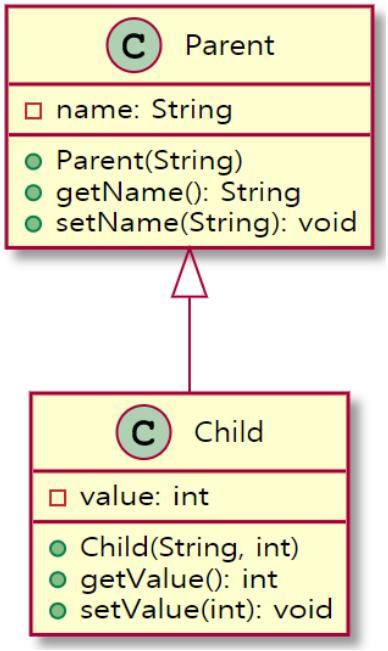
상속의 예

- 부모 클래스는 한 개 이상의 자식 클래스와 관계를 맺는 **계층 구조**로 표현 가능
  - 공통 부분을 부모 클래스에, 서로 다른 부분은 자식 클래스에 구현 가능



상속의 예

- Parent 클래스
  - "name"이라는 멤버필드
  - name을 초기화시키는 생성자
  - name에 접근할 수 있도록 구현된 getter/setter 메소드
- Child 클래스
  - parent의 자식 클래스
  - "value" 변수와 value를 초기화 시키는 생성자와 getter/setter 메소드
- Parent와 Child 클래스 객체가 생성된후 메모리 구조




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

Parent의 메모리 구조

String name
Parent(String n)
String getName()
void setName(String n)

Child의 메모리 구조

String name
Parent(String n)
String getName()
void setName(String n)
int value
Child(String n, int v)
int getValue()
void setValue(int v)

클래스 상속과 객체

- 자바 상속의 특징
  - **클래스 다중 상속 지원하지 않음**
    - 여러 개의 클래스를 상속받지 못함
  - 상속 횟수 무제한
  - 상속의 최상위 클래스는 java.lang.Object 클래스
    - 모든 클래스는 자동으로 java.lang.Object를 상속받음
- 자바 상속 구현 방법

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

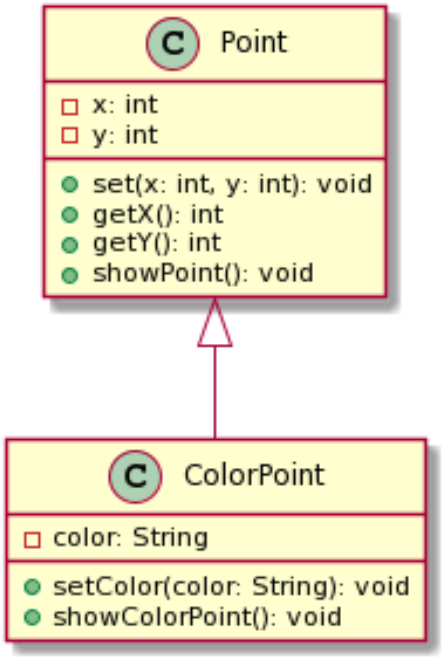
예제 : Point & ColorPoint 클래스

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

```
public 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 int getX() {
      return x;
   }
   public int getY() {
      return y;
   }
   public void showPoint() { // 점의 좌표 출력
      System.out.println("(" + x + "," + y + ")");
   }
}
```

예제: Point & ColorPoint 클래스

```
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)
```

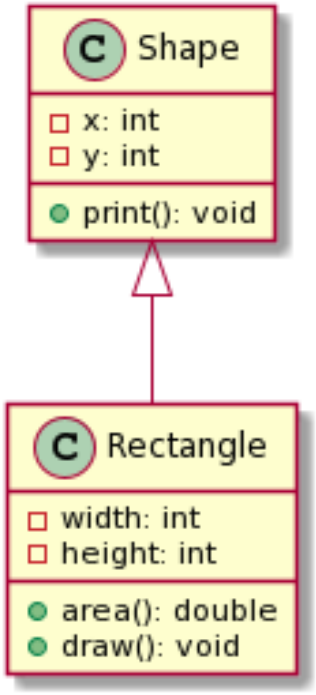
예제: Shape & Rectangle 클래스

```
public class Shape {
   private int x;
   private int y;
   public int getX() {
      return x;
   }
   public int getY() {
      return y;
   }
   void set(int x, int y) {
      this.x = x; this.y = y;
   }
   void print() { // x,y 좌표 출력
      System.out.println("(" + x +
     "," + y + ")");
   }
}
```

```
public class Rectangle extends Shape {
    private int width; // 사각형의 너비
    private int height; // 사각형의 높이
    public void getWidth() { return width;}
    public void getHeight() {return height; }
    public void setWidth(int width) {
       this.width = width; }
    public void setHeight(int height) {
       this.height = height; }
    public double area() { // 사각형의 넓이(영역)
       return (double)width * height);
   }
   void draw() { // x, y 출력
      System.out.println(" 사각형 " + getX()+ " x"
+ getY() + "의 영역은 " + area());
   }
}
```

예제 : Shape & Rectangle 클래스

```
public class RectangleTest {
   public static void main(String [] args) {
      Rectangle r1 = new Rectangle();
      Rectangle r2 = new Rectangle();
      r1.set(5,3); // Shape 클래스의 set() 메소드 호출
      r1.setWidth(10); // Rectangle 클래스의 setWidth() 메소드 호출
      r1.setHeight(20); // Rectangle 클래스의 setHeight() 메소드 호출
      r2.set(8,9); // Shape 클래스의 set() 메소드 호출
      r2.setWidth(10); // Rectangle 클래스의 setWidth() 메소드 호출
      r2.setHeight(20); // Rectangle 클래스의 setHeight() 메소드 호출

      r1.print(); // x,y좌표 출력
      r1.draw(); // x,y,width,height,area 출력
      r2.print(); // x,y좌표 출력
      r2.draw(); // x,y,width,height,area 출력
   }
}
```



서브 클래스의 객체와 멤버 사용

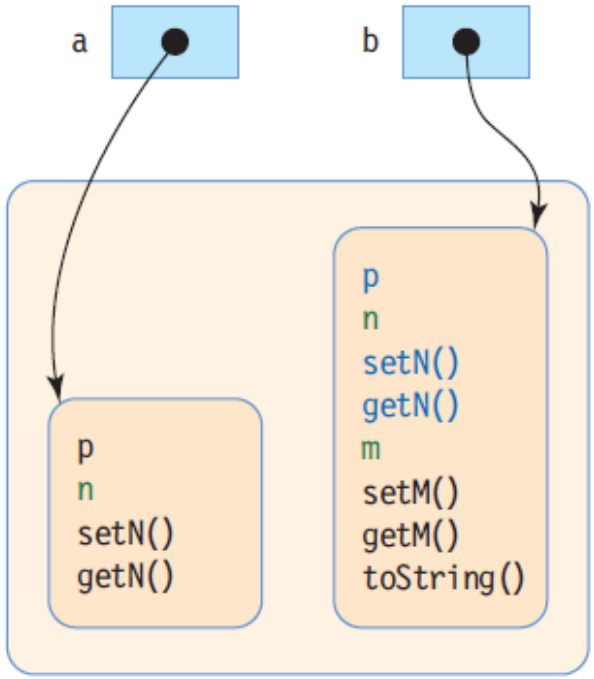
- 서브 클래스의 객체와 멤버 접근
  - 서브 클래스의 객체에는 슈퍼 클래스 멤버 포함
    - 슈퍼 클래스의 private 멤버는 상속되지만 서브 클래스에서 직접 접근 불가
    - 슈퍼 클래스의 private 멤버는 슈퍼 클래스의 public/protected 메소드를 통해 접근

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;
    }
}
```

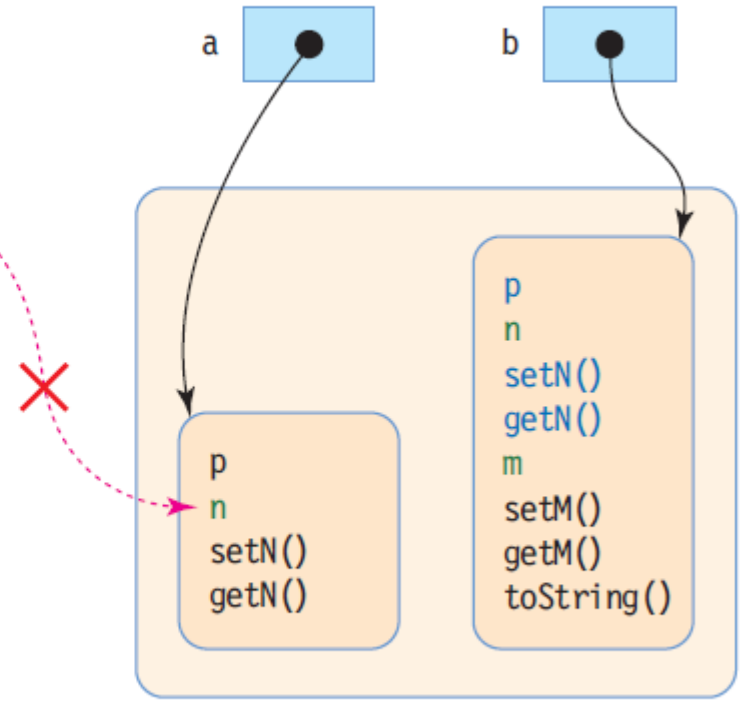
```
public static void main(String [] args) {
    A a = new A();
    B b = new B();
}
```



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 다른 모든 클래스에 접근 불허
- 슈퍼 클래스의 protected 멤버
  - 같은 패키지 내의 모든 클래스 접근 허용
  - 동일 패키지 여부와 상관없이 서브 클래스에서 슈퍼 클래스의 protected 멤버 접근 가능

슈퍼 클래스 멤버의 접근 지정자

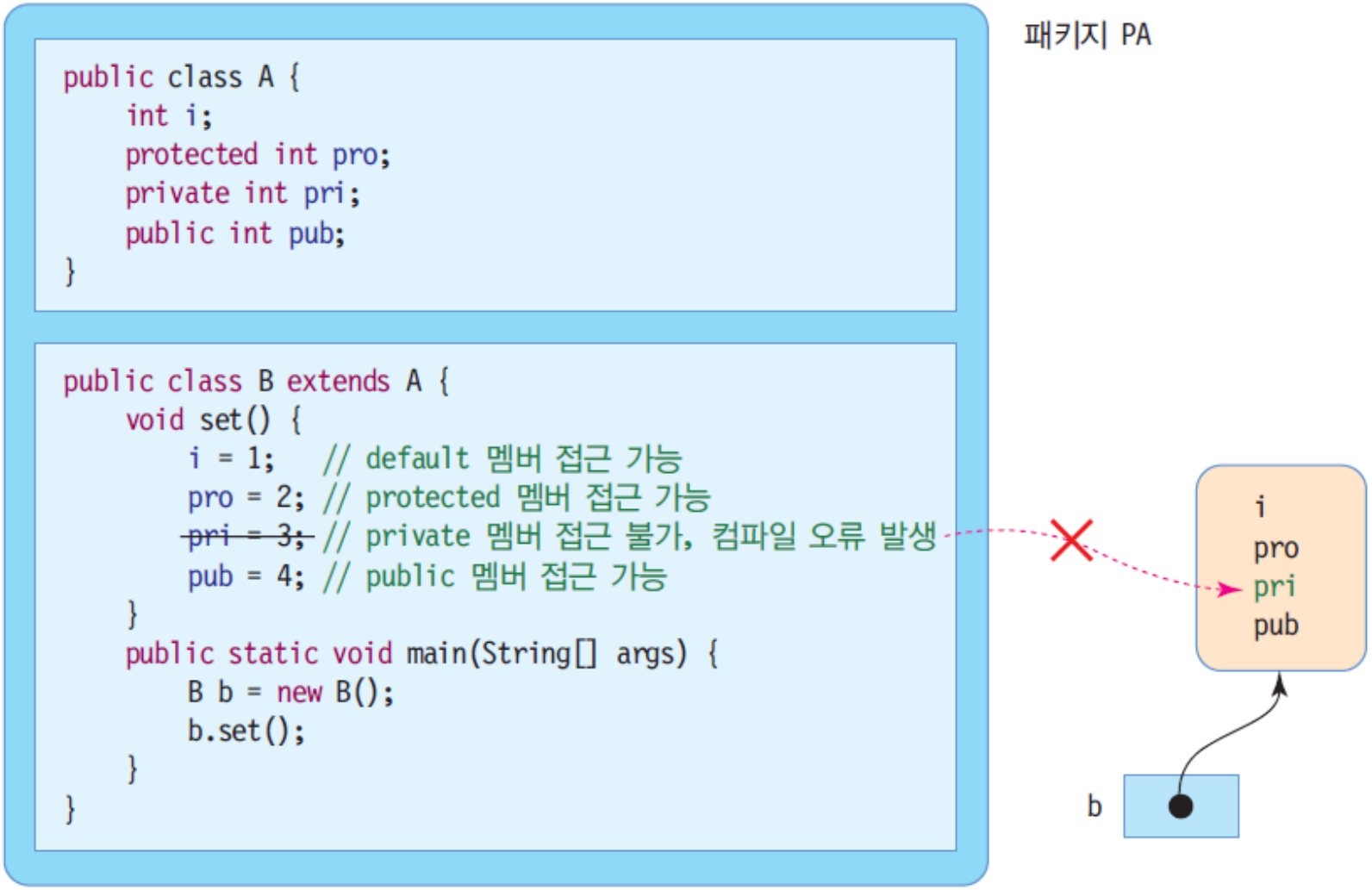
슈퍼 클래스 멤버에 접근하는 클래스 종류	슈퍼 클래스 멤버의 접근 지정자			
	default	private	protected	public
같은 패키지의 클래스	○	×	○	○
다른 패키지의 클래스	×	×	×	○
같은 패키지의 서브 클래스	○	×	○	○
다른 패키지의 서브 클래스	×	×	○	○

(○는 접근 가능함을, ×는 접근이 불가함을 뜻함)

슈퍼클래스와 서브클래스가 같은 패키지(같은 폴더, 디렉토리)에 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();
    }
}
```



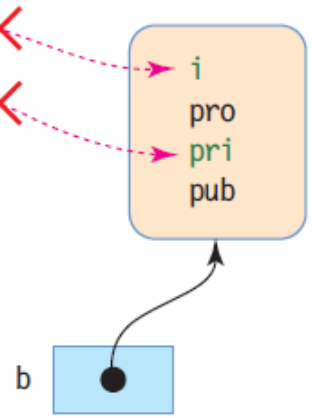
슈퍼클래스와 서브클래스가 서로 다른 패키지 (다른 폴더, 디렉토리) 에 있는 경우

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 : 상속 관계에 있는 클래스 간 멤버 접근

- Person 클래스
  - int age;
  - public String name;
  - protected int height;
  - private int weight;
- Student 클래스는 Person 클래스를 상속받아 각 멤버 필드에 값을 저장한다.
  - Person 클래스의 private 필드인 weight는 Student 클래스에서는 접근이 불가능하여 슈퍼 클래스인 Person의 get, set 메소드를 통해서만 조작이 가능하다.

```
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;
  }
}
```

예제 : 상속 관계에 있는 클래스 간 멤버 접근

```
public class Student extends Person {
   void set() {
      age = 30; // default 접근지정자는 같은 패키지에서 사용가능
      name = "홍길동";
      height = 175;
      setWeight(99);
   }

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

서브/슈퍼 클래스의 생성자 호출과 실행

- 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

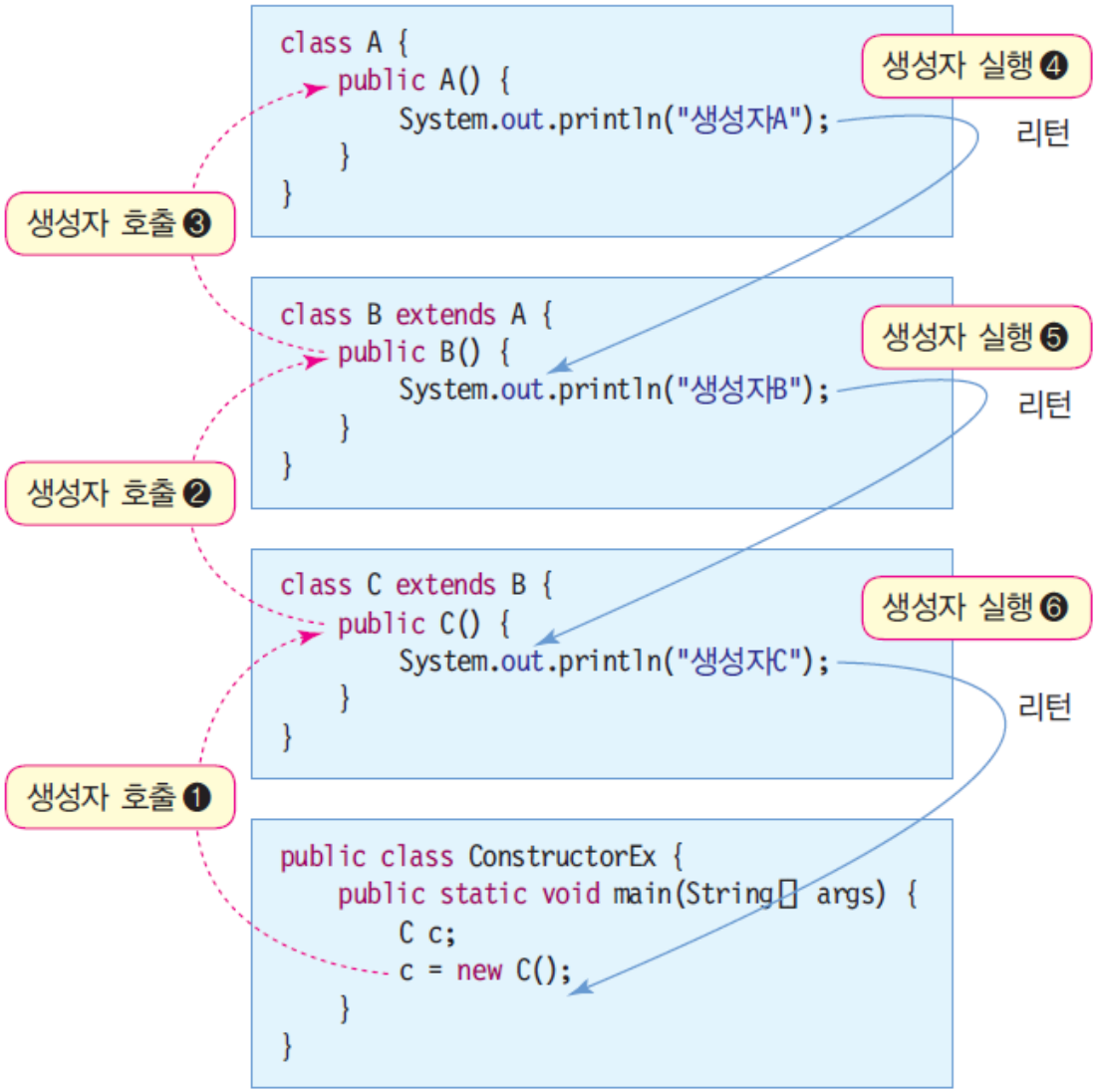
질문 1 서브 클래스의 인스턴스가 생성될 때 서브 클래스의 생성자와 슈퍼 클래스의 생성자가 모두 실행되는가? 아니면 서브 클래스의 생성자만 실행되는가?

답 둘 다 실행된다. 생성자는 인스턴스를 초기화할 목적으로 사용되므로 서브 클래스의 생성자는 서브 클래스 내의 멤버를 초기화하거나 필요한 초기화 작업을 수행할 필요가 있고, 슈퍼 클래스의 생성자는 슈퍼 클래스의 멤버를 초기화하거나 필요한 초기화 작업을 수행할 필요가 있기 때문이다.

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

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

서브/슈퍼 클래스의 생성자 호출 및 실행



생성자A
생성자B
생성자C

서브 클래스와 슈퍼 클래스의 생성자 짝 맞추기

- 슈퍼 클래스와 서브 클래스
  - 각각 여러 개의 생성자 작성 가능
- 슈퍼 클래스와 서브 클래스의 생성자 사이의 짝 맞추기
  - 서브 클래스와 슈퍼 클래스의 생성자 조합 4 가지

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

  - 서브 클래스에서 슈퍼 클래스의 생성자를 선택하지 않는 경우
    - 컴파일러가 자동으로 슈퍼 클래스의 기본 생성자 선택
  - 서브 클래스 개발자가 슈퍼 클래스의 생성자를 명시적으로 선택하는 경우
    - super() 키워드를 이용하여 선택

슈퍼 클래스의 기본 생성자 자동 호출 – 서브 클래스의 기본 생성자 경우

서브 클래스의 생성자가 슈퍼 클래스의 생성자를 선택하지 않은 경우

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

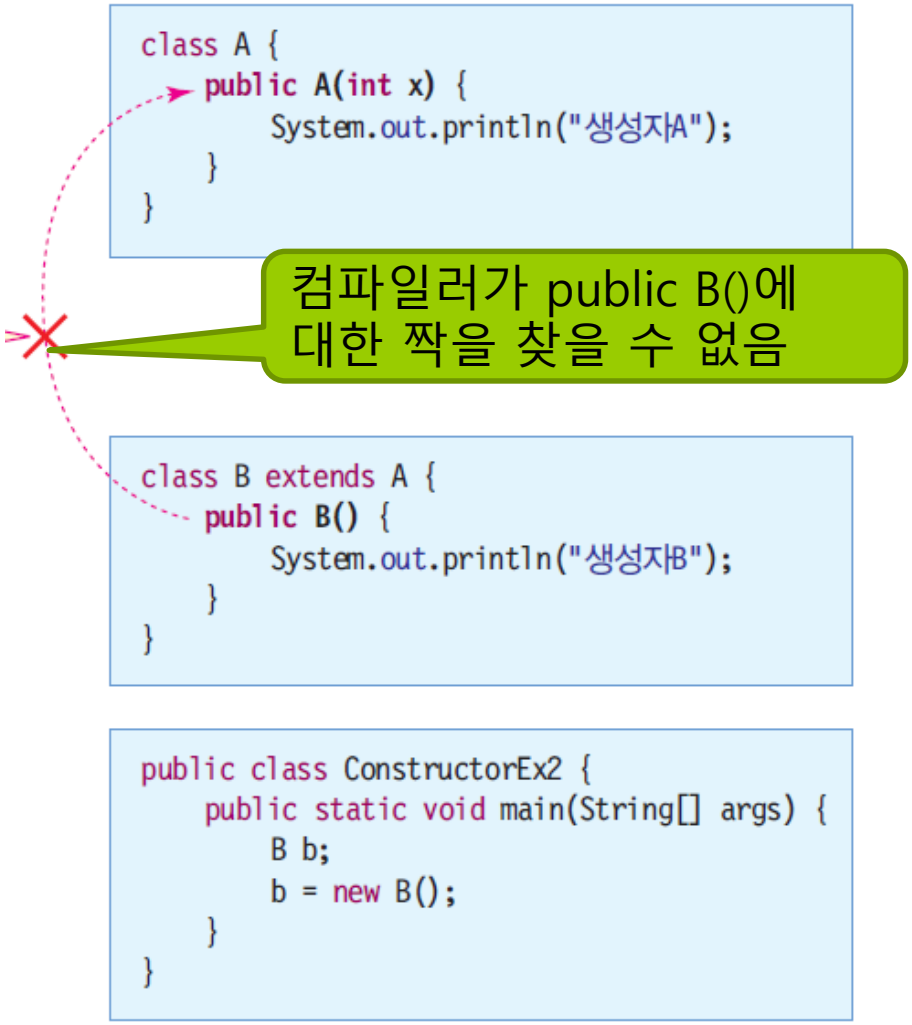
```
class B extends A {
    public B() {
        System.out.println("생성자B");
    }
}
```

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

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

생성자A
생성자B

슈퍼 클래스에 기본 생성자가 없어 오류 난 경우



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

슈퍼 클래스의 기본 생성자 자동 호출 – 서브 클래스의 매개 변수를 가진 생성자 경우

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);
    }
}
```

생성자A
매개변수생성자B

super()를 이용하여 슈퍼 클래스 생성자 선택

- **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

```
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);
    }
}
```

super(); 라고 하면 A() 호출

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

매개변수생성자A5
매개변수생성자B5

객체의 타입 변환

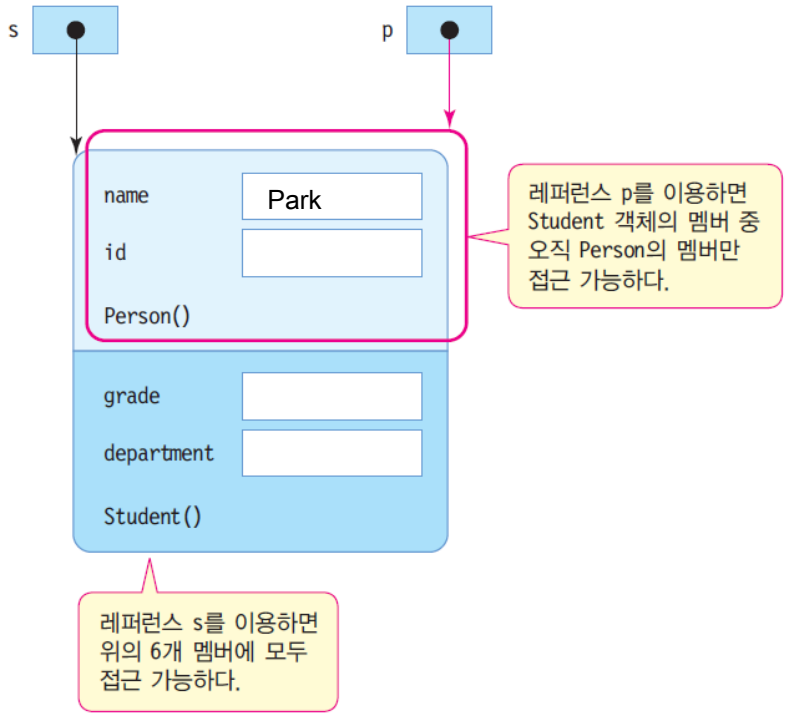
- 업캐스팅(upcasting)
  - 프로그램에서 이루어지는 자동 타입 변환
  - 서브 클래스의 레퍼런스 값을 슈퍼 클래스 레퍼런스에 대입
    - 슈퍼 클래스 레퍼런스가 서브 클래스 객체를 가리키게 되는 현상
    - 객체 내에 있는 모든 멤버를 접근할 수 없고 슈퍼 클래스의 멤버만 접근 가능

```
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("Park");
      p = s; // 업캐스팅 발생
      System.out.println(p.name); // 오류 없음
      //p.grade = "A"; // 컴파일 오류
      //p.department = "Com"; // 컴파일 오류
   }
}
```

Park



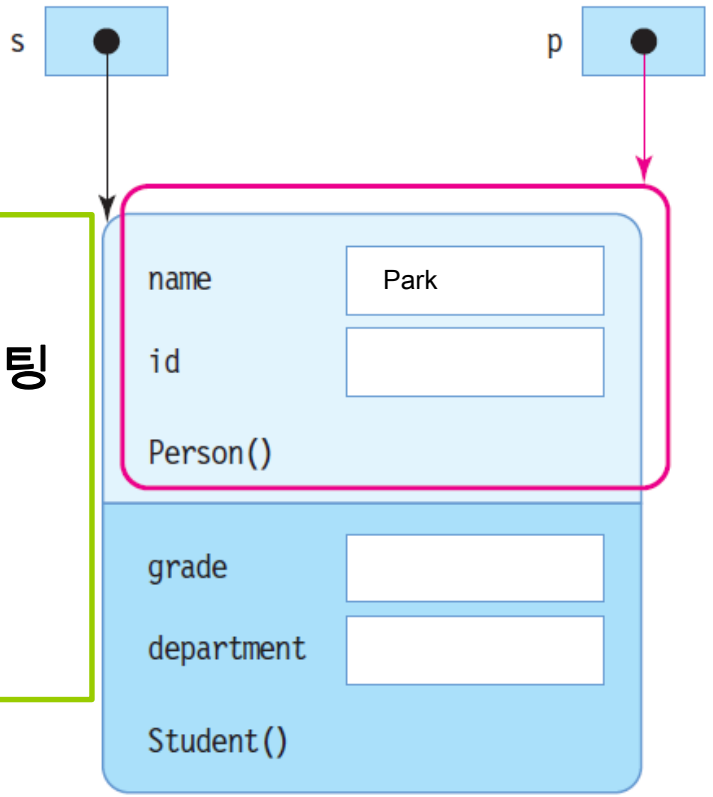
객체의 타입 변환

- 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

```
public class DowncastingEx {
   public static void main(String[] args) {
      Person  p = new Student("Park"); // 업캐스팅
      Student s = (Student)p; // 다운캐스팅
      System.out.println(s.name); // 오류 없음
      s.grade = "A"; // 오류 없음
   }
}
```



instanceof 연산자와 객체의 타입 구별

- 업캐스팅된 레퍼런스로는 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

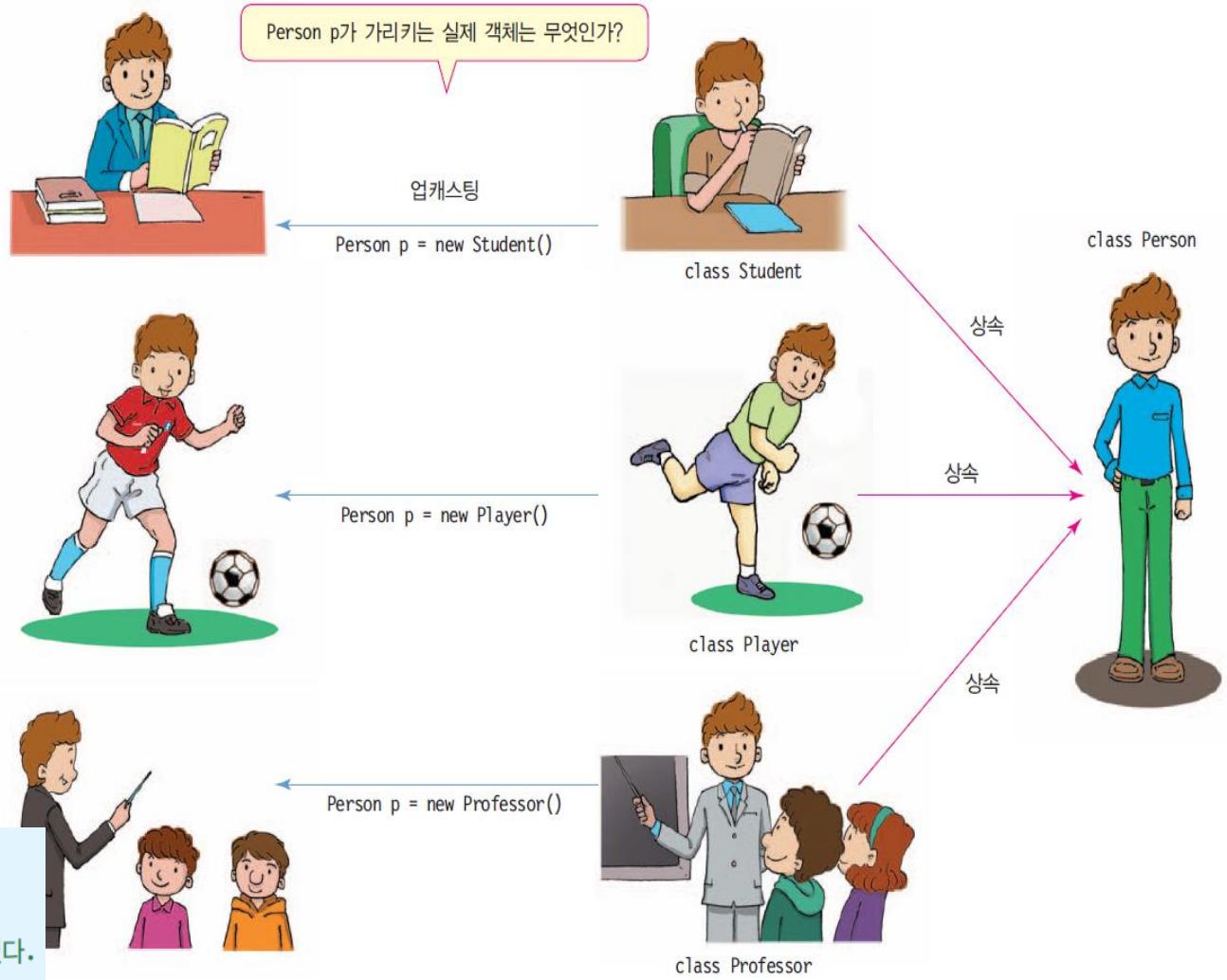
객체레퍼런스 **instanceof** 클래스타입

연산의 결과 : true/false의 불린 값

업캐스팅 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.
.....
}
```



예제 : 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
```

Method Overriding

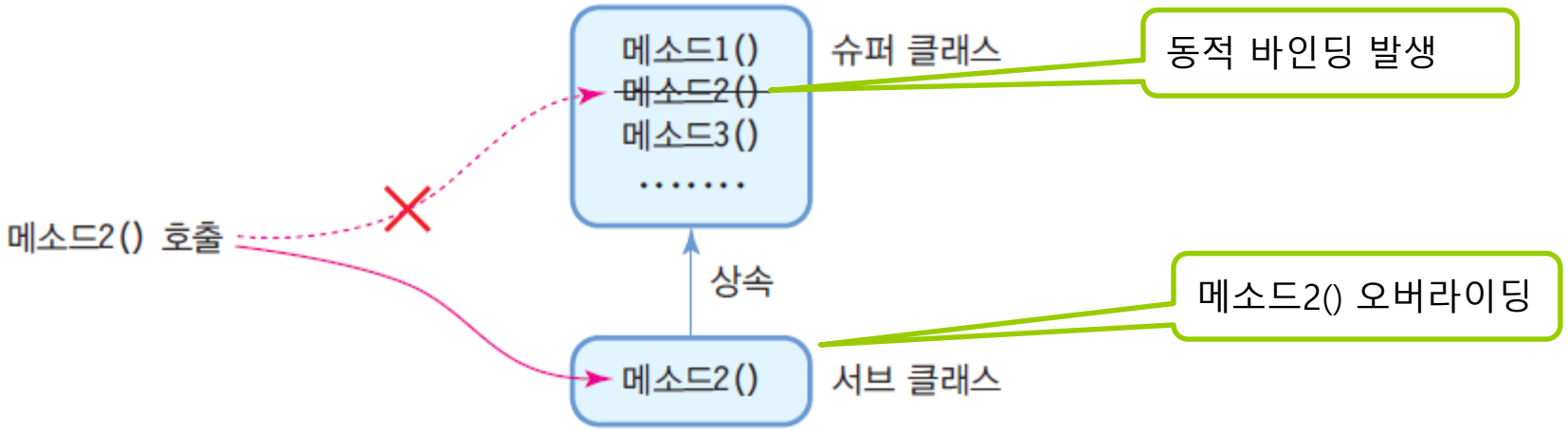
```
class Animal {
        public void sound() {
                System.out.println("동물소리!");
        }
}
```

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

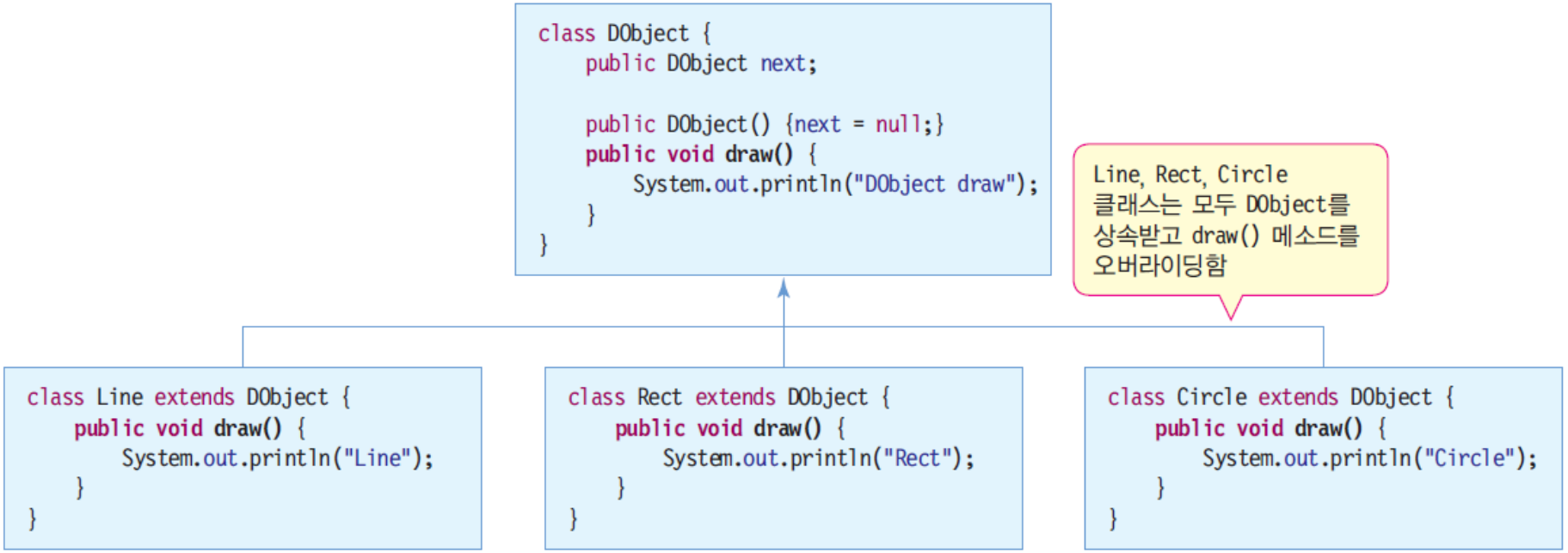
멍멍

Method Overriding

- 메소드 오버라이딩(Method Overriding)
  - 슈퍼 클래스의 메소드를 서브 클래스에서 재정의
    - 슈퍼 클래스의 메소드 이름, 메소드 인자 타입 및 개수, 리턴 타입 등 모든 것 동일하게 작성
      - 이 중 하나라도 다르면 메소드 오버라이딩 실패
  - 동적 바인딩 발생
    - 서브 클래스에 오버라이딩된 메소드가 무조건 실행되도록 동적 바인딩 됨

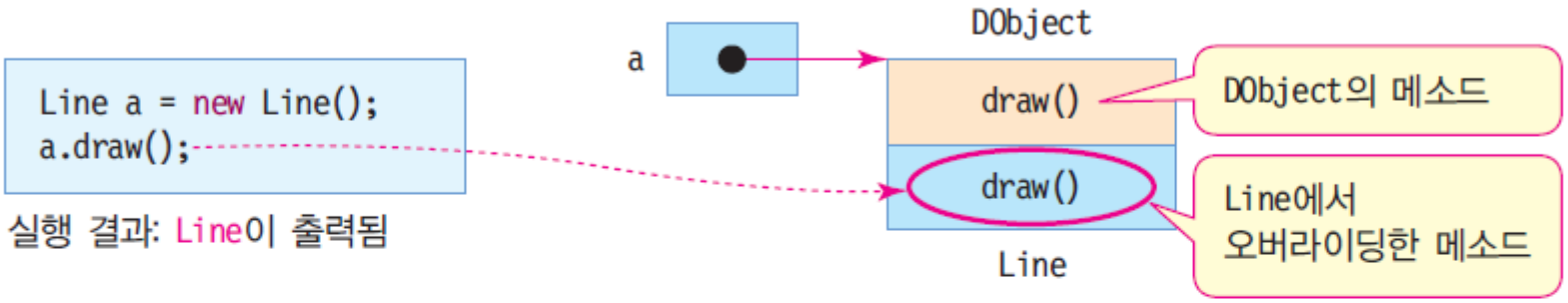


Method Overriding 사례

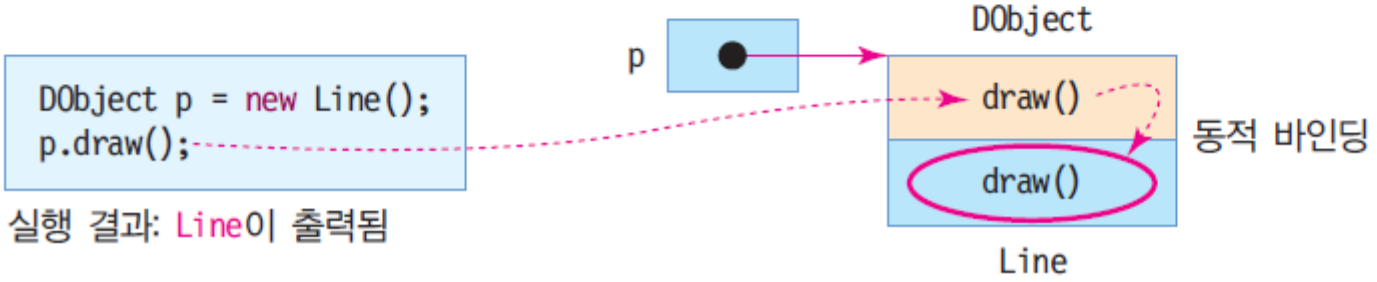


서브 클래스 객체와 오버라이딩된 메소드 호출

(1) 서브 클래스 레퍼런스로 오버라이딩된 메소드 호출



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



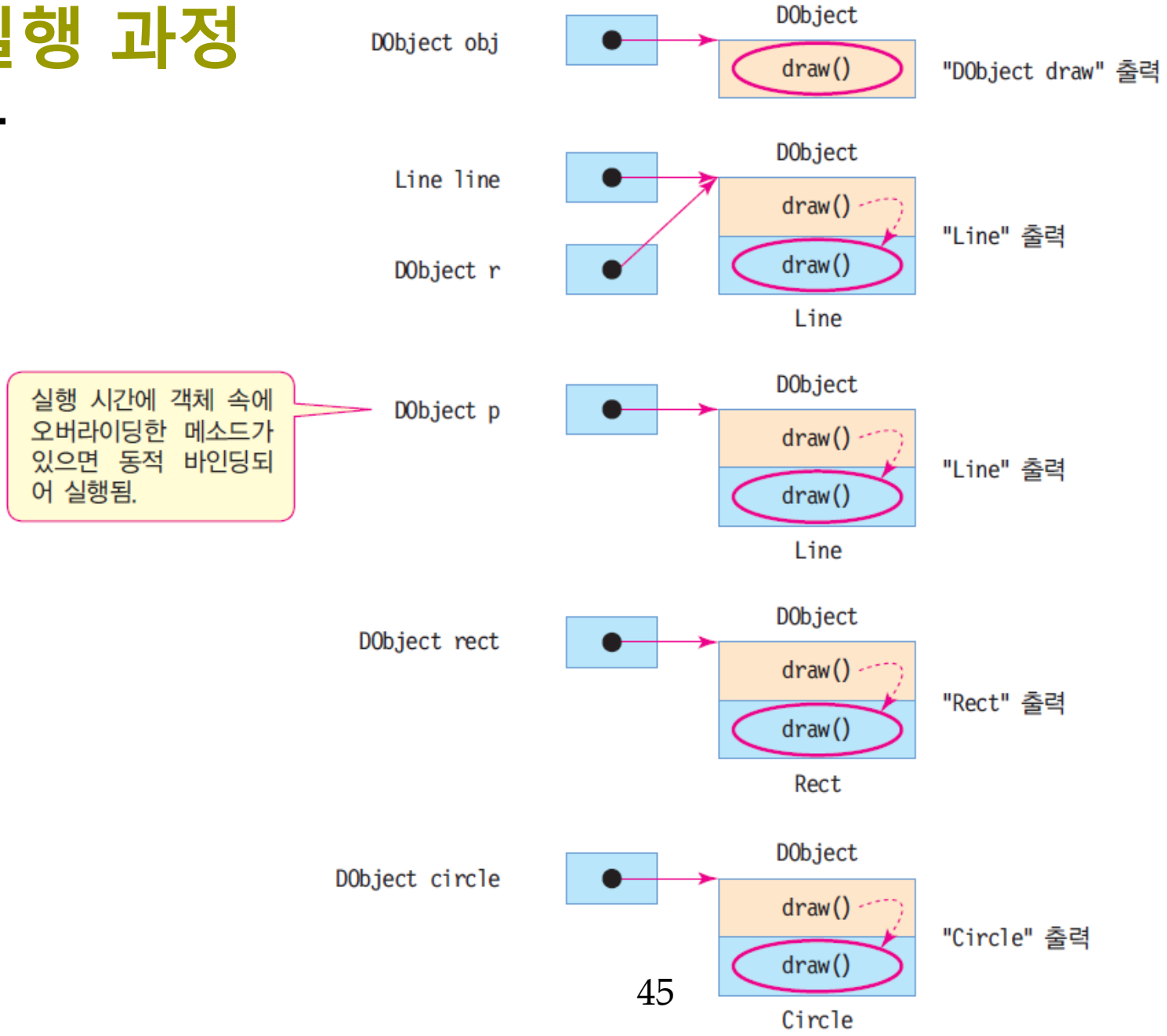
예제 : void draw() 메소드 오버라이딩 예

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

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

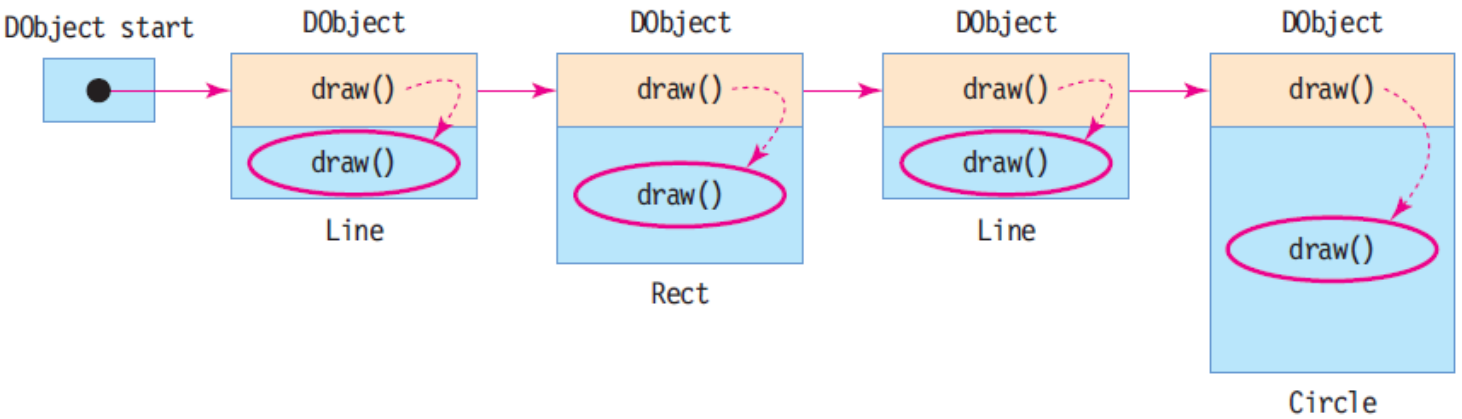
   public > protected > private 순으로 지정 범위가 좁아진다.
3. 반환 타입만 다르면 오류
4. private, final 메소드는 오버라이딩 될 수 없다.

```
class Person {
        String name;
        String phone;
        private int ID;
        public void setName(String s) { name = s; }
        public String getPhone() { return phone; }
        public final int getID() { return ID; }
}
class Professor extends Person {
        //protected void setName(String s) { // public을
        protected로 지정할 수 없음
        //}
        // 동일한 메소드명, 인자, 반환타입을 가짐
        public String getPhone() { return phone; }
        //public void getPhone(){ // 반환타입 다르면 오류
        //}
        //public int getID() { // final은 override할 수 없음
        // return ID // Person의 ID는 private이라서 오류
        //}
}
```

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

오버라이딩 메소드가 항상 호출된다.

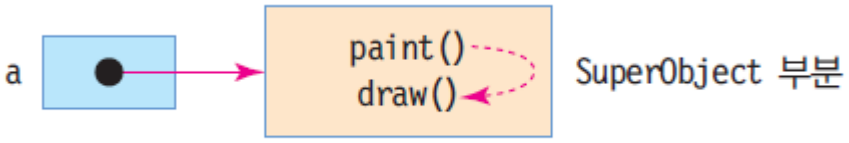
```
public class SuperObject {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Super Object");
    }
    public static void main(String [] args) {
        SuperObject a = new SuperObject();
        a.paint();
    }
}
```

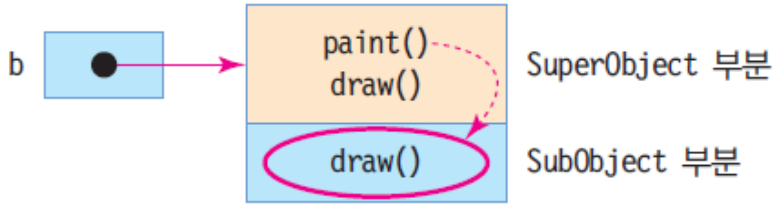
Super Object

```
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

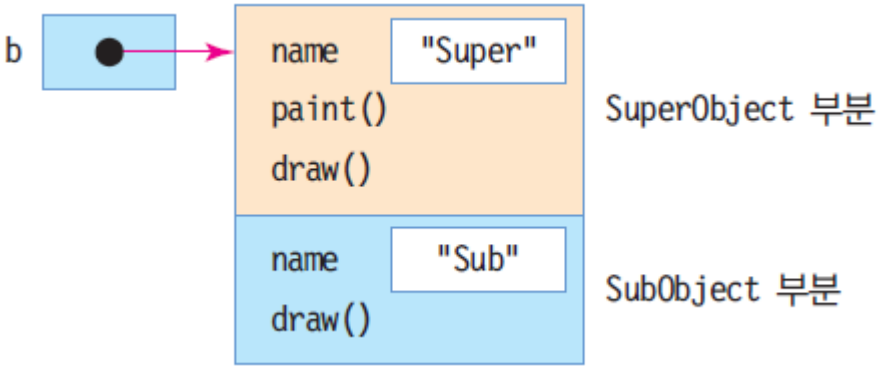
Sub Object





super 키워드

- **super** 키워드
  - super는 슈퍼 클래스의 멤버를 접근할 때 사용되는 레퍼런스
  - 서브 클래스에서만 사용
  - 슈퍼 클래스의 메소드 호출 시 사용
  - 컴파일러는 super 호출을 정적 바인딩으로 처리




```
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: Method Overriding

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
```

Method Overloading vs Overriding

비교 요소	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 편리성 향상	슈퍼 클래스에 구현된 메소드를 무시하고 서브 클래스에서 새로운 기능의 메소드를 재정의하고자 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
    - **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

추상 클래스

추상 메소드

```
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 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 상속

- 추상 클래스의 상속 2 가지 경우
  - 추상 클래스의 단순 상속
    - 추상 클래스를 상속받아, 추상 메소드를 구현하지 않으면 서브 클래스도 추상 클래스 됨
    - 서브 클래스도 abstract로 선언해야 함

```
abstract class DObject { // 추상 클래스
   public DObject next;
   public DObject() { next = null; }
   abstract public void draw(); // 추상 메소드
}
abstract class Line extends DObject { // draw()를 구현하지 않았기
때문에 추상 클래스
   public String toString() { return "Line";}
}
```

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

추상 클래스의 구현 및 활용 예

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

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

추상 클래스로 수정

```
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");
    }
}
```

추상 클래스를 상속받아
추상 메소드 draw()를
구현한 클래스

추상 클래스의 용도

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

예제: Calculator 추상 클래스

다음의 추상 클래스 Calculator를 상속받는 GoodCalc 클래스를 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);
}
```

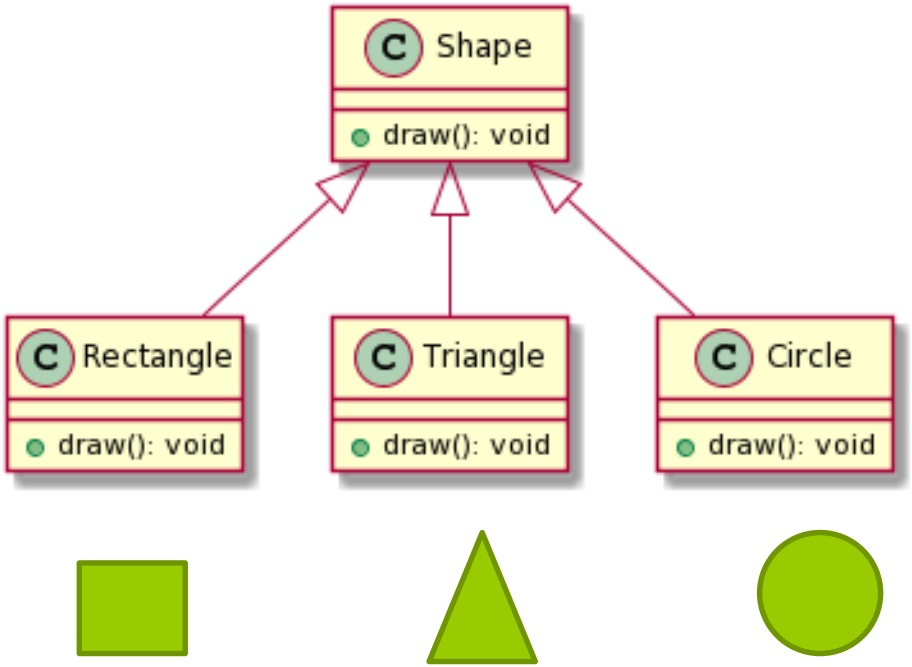
예제: GoodCalc 클래스

```
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
```

다형성(Polymorphism)

- 다형성(polymorphism)이란 객체들의 타입이 다르면 똑같은 메시지가 전달되더라도 각 객체의 타입에 따라서 서로 다른 동작을 하는 것(dynamic binding)



예제: 다형성 예시

```
class Shape {
  protected int x, y;
  public void draw() {
  System.out.println("Shape Draw");
  }
}
class Rectangle extends Shape {
  private int width, height;
  public void draw() {
  System.out.println("Rectangle Draw");
  }
}
class Triangle extends Shape {
  private int base, height;
  public void draw() {
  System.out.println("Triangle Draw");
  }
}
```

다형성 예시

```
class Circle extends Shape {
  private int radius;
  public void draw() {
  System.out.println("Circle Draw");
  }
}
public class ShapeTest {
  public static void main(String arg[]) {
  Shape s1 = new Shape();
  Shape s2 = new Rectangle();
  Shape s3 = new Triangle();
  Shape s4 = new Circle();
  s1.draw();
  s2.draw();
  s3.draw();
  s4.draw();
  }
}
```

```
Shape Draw
Rectangle Draw
Triangle Draw
Circle Draw
```

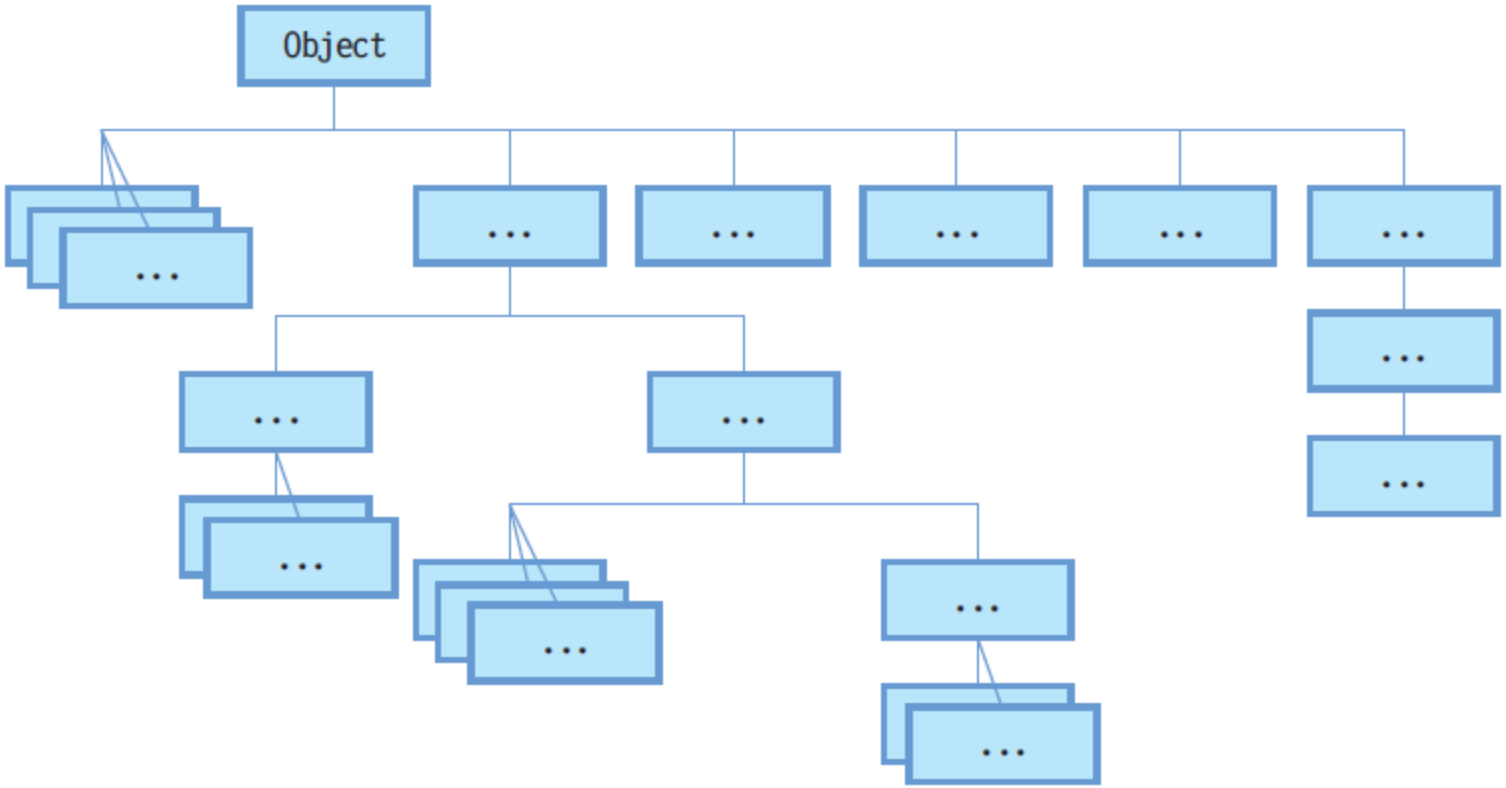
다형성 예시

```
public class ShapeTest2 {
    public static void main(String[] arg) {
        Shape[] arrayOfShapes = new Shape[4];
        arrayOfShapes[0] = new Shape();
        arrayOfShapes[1] = new Rectangle();
        arrayOfShapes[2] = new Triangle();
        arrayOfShapes[3] = new Circle();
        // 실행시 객체의 타입에 맞는 메소드를 호출하는 동적 바인딩 (dynamic
binding)
        for (int i = 0; i < arrayOfShapes.length; i++) {
            arrayOfShapes[i].draw();
        }
    }
}
```

```
Shape Draw
Rectangle Draw
Triangle Draw
Circle Draw
```

자바의 클래스 계층 구조

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



Object의 메소드

메소드	설명
protected Object clone()	현 객체와 똑같은 객체를 만들어 리턴
boolean equals(Object obj)	obj가 가리키는 객체와 현재 객체가 비교하여 같으면 true 리턴
Class getClass()	현 객체의 런타임 클래스를 리턴
int hashCode()	현 객체에 대한 해시 코드 값 리턴
String toString()	현 객체에 대한 스트링 표현을 리턴
void notify()	현 객체에 대해 대기하고 있는 하나의 스레드를 깨운다.
void notifyAll()	현 객체에 대해 대기하고 있는 모든 스레드를 깨운다.
void wait()	다른 스레드가 깨울 때까지 현재 스레드를 대기하게 한다.

Object의 메소드

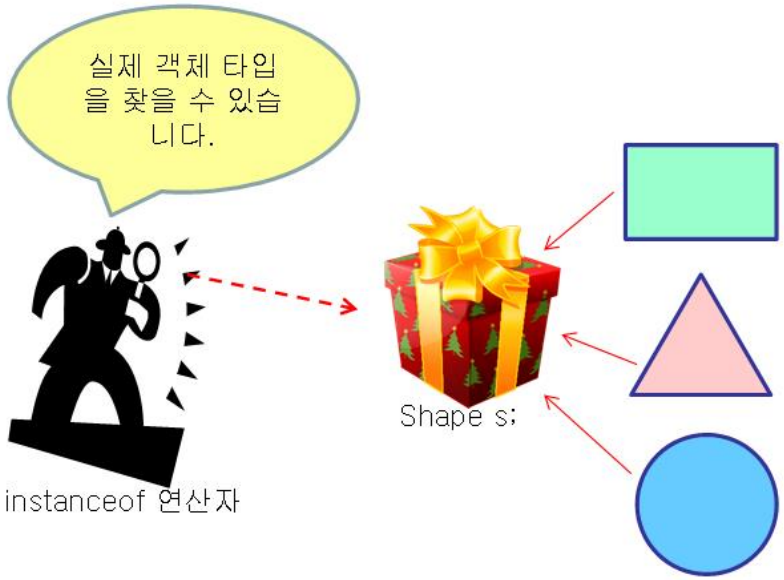
```
class Point {
   int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
public class ObjectProperty {
   public static void main(String [] args) {
      Point p = new Point(2,3);
      System.out.println(p.getClass().getName());
      System.out.println(p.hashCode());
      System.out.println(p.toString());
      System.out.println(p);
   }
}
```

```
Point
12677476
Point@c17164
Point@c17164
```

instanceof

- 객체의 실제타입인지 여부를 반환한다.

```
Shape s = getShape();
if (s instanceof Rectangle) {
  System.out.println("Rectangle이 생성되었습니다");
} else {
  System.out.println("Rectangle이 아닌 다른 객체가 생성되었습니다");
}
```



getClass() 메소드

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

```
obj is of type Car
```

equals() 메소드

```
class Car {
    private String model;
    public Car(String model) {          this.model= model;          }
    public boolean equals(Object obj) {
        if (obj instanceof Car)
                return model.equals(((Car) obj).model);
        else
                return false;
    }
}
public class CarTest {
    public static void main(String[] args) {
        Car firstCar = new Car("HMW520");
        Car secondCar = new Car("HMW520");

        if (firstCar.equals(secondCar)) {
                System.out.println("동일한 종류의 자동차입니다.");
        } else {
                System.out.println("동일한 종류의 자동차가 아닙니다.");
        }
    }
}
```

Object의 equals()를 재정의

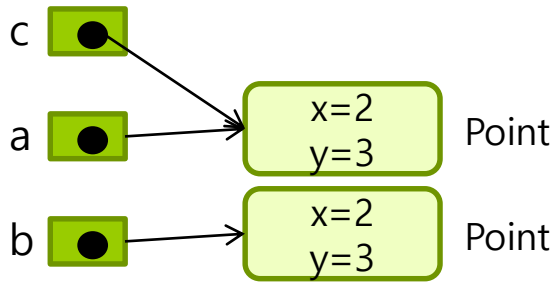
동일한 종류의 자동차입니다.

객체 비교(==과 equals())

- 객체 레퍼런스의 동일성 비교엔 == 연산자 이용
- 객체 내용 비교
  - 서로 다른 두 객체가 같은 내용물인지 비교
  - boolean equals(Object obj) 이용

```
class Point {
   int x, y;
   public Point(int x, int y) {
      this.x = x; this.y = y;
   }
}
```

```
Point a = new Point(2,3);
Point b = new Point(2,3);
Point c = a;
if(a == b) // false
   System.out.println("a==b");
if(a == c) // true
   System.out.println("a==c");
```



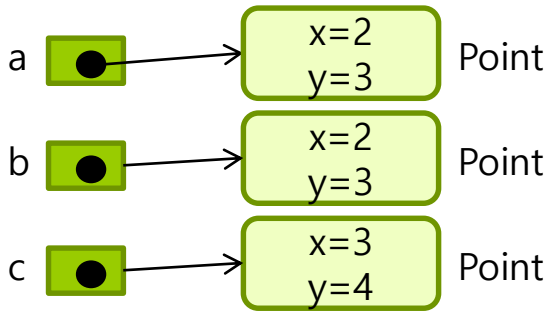
a==c

객체 비교(==과 equals())

```
class Point {
  int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  public boolean equals(Point p) {
    if(x == p.x && y == p.y)
      return true;
    else
      return false;
  }
}
```

```
Point a = new Point(2,3);
Point b = new Point(2,3);
Point c =  new Point(3,4);
if(a == b) // false
  System.out.println("a==b");
if(a.equals(b)) // true
  System.out.println("a is equal to b");
if(a.equals(c)) // false
  System.out.println("a is equal to c");
```

a is equal to b



예제: Rect 클래스 equals()

int 타입의 width, height의 필드를 가지는 Rect 클래스를 작성하고, 두 Rect 객체의 width, height 필드에 의해 구성되는 면적이 같으면 두 객체가 같은 것으로 판별하도록 equals()를 작성하라. Rect 생성자에서 width, height 필드를 인자로 받아 초기화한다.

```
class Rect {
   int width;
   int height;
   public Rect(int width, int height) {
      this.width = width;
      this.height = height;
   }
   public boolean equals(Rect p) {
      if (width*height == p.width*p.height)
         return true;
      else
         return false;
   }
}
```

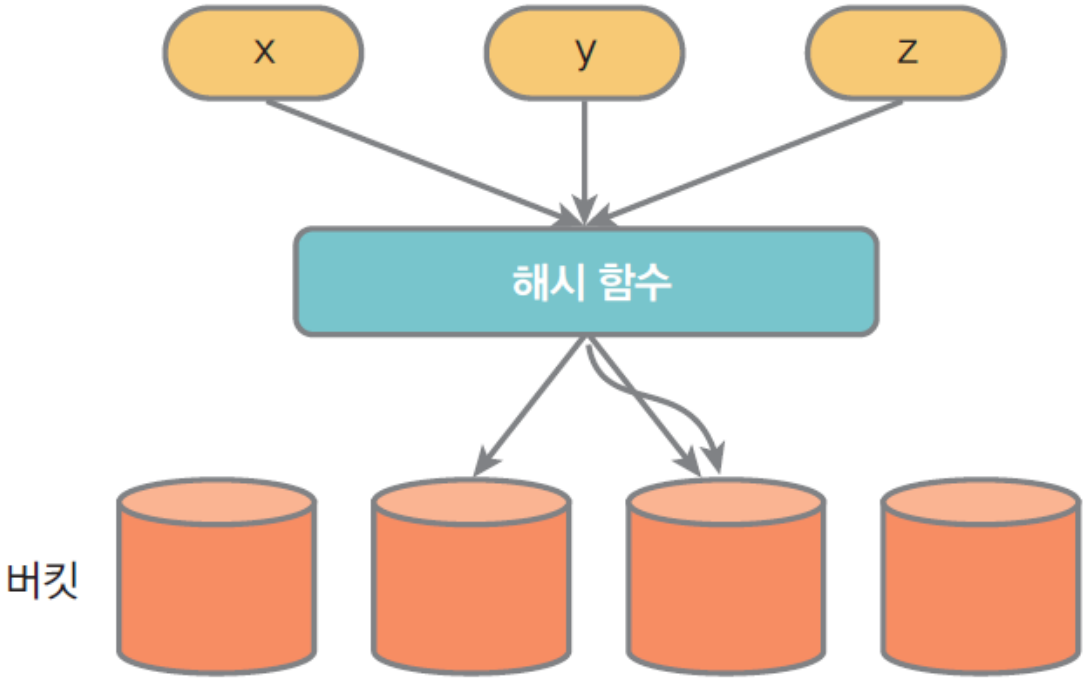
예제: Rect 클래스 equals()

```
public class EqualsEx {
  public static void main(String[] args) {
    Rect a = new Rect(2,3);
    Rect b = new Rect(3,2);
    Rect c = new Rect(3,4);
    if(a.equals(b))   System.out.println("a is equal to b");
    if(a.equals(c))   System.out.println("a is equal to c");
    if(b.equals(c))   System.out.println("b is equal to c");
  }
}
```

```
a is equal to b
```

hashCode() 메소드

- hashCode()는 해싱이라는 탐색 알고리즘에서 필요한 해시값을 생성하는 메소드이다.



toString() 메소드

- Object 클래스의 toString() 메소드는 객체의 문자열 표현을 반환한다.

```
class Point {
   int x, y;
   public Point(int x, int y) {
      this.x = x; this.y = y;
   }
   public String toString() {
      return "Point(" + x + "," + y+ ")";
   }
}
public class ObjectProperty {
   public static void main(String [] args) {
      Point a = new Point(2,3);
      System.out.println(a.toString());
   }
}
```

Object의 toString()를 재정의

System.out.println(a); 라고 해도 동일

Point(2,3)

toString() 메소드 사용

- String toString()
  - 객체를 문자열로 반환
  - Object 클래스에 구현된 toString()이 반환하는 문자열
    - 클래스 이름@객체의 hash code
  - 각 클래스는 toString()을 오버라이딩하여 자신만의 문자열 리턴 가능
- 컴파일러에 의한 자동 변환
  - '객체 + 문자열' -> '객체.toString() + 문자열'로 자동 변환

```
Point a = new Point(2,3);
String s = a + "점";
System.out.println(s);
```

변환

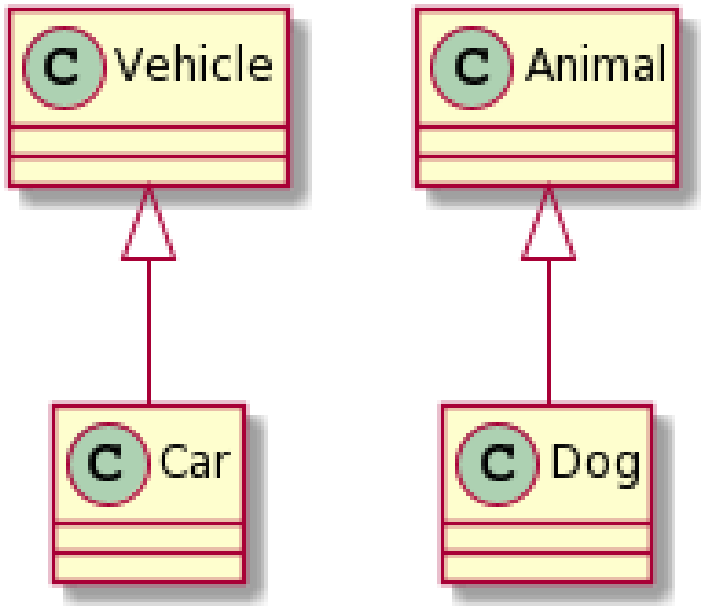
```
Point a = new Point(2,3);
String s = a.toString()+ "점";
System.out.println(s.toString());
```

Point@c17164점

IS-A 관계

- is-a 관계: "~은 ~이다"와 같은 관계
- **상속은 is-a 관계이다.**

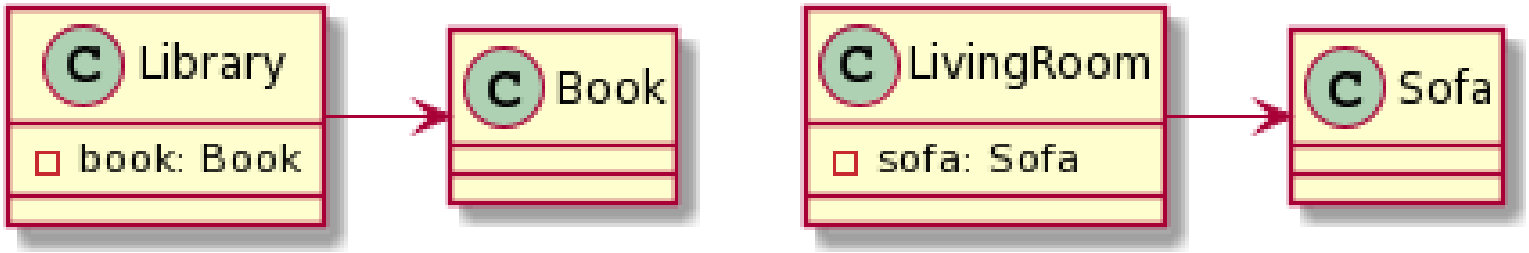
- 자동차는 탈것이다(Car is a Vehicle).
- 강아지는 동물이다(Dog is a animal).



HAS-A 관계

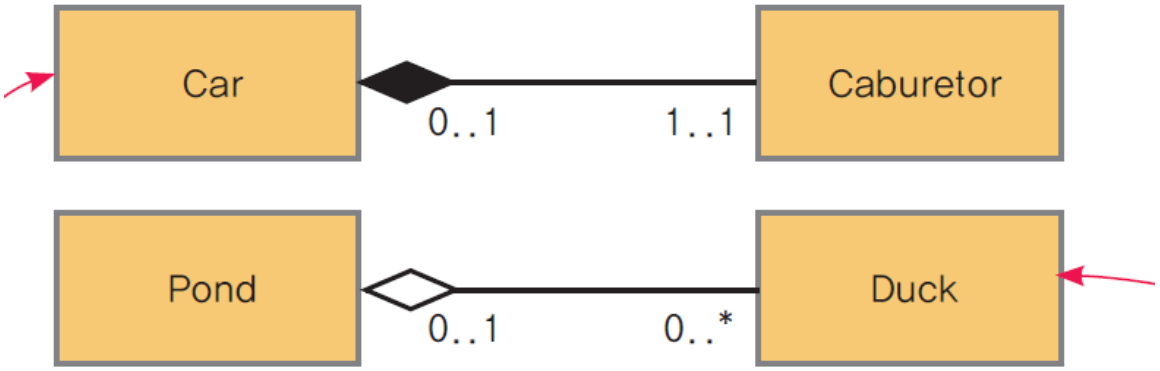
- has-a 관계: ~은 ~을 가지고 있다"와 같은 관계

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

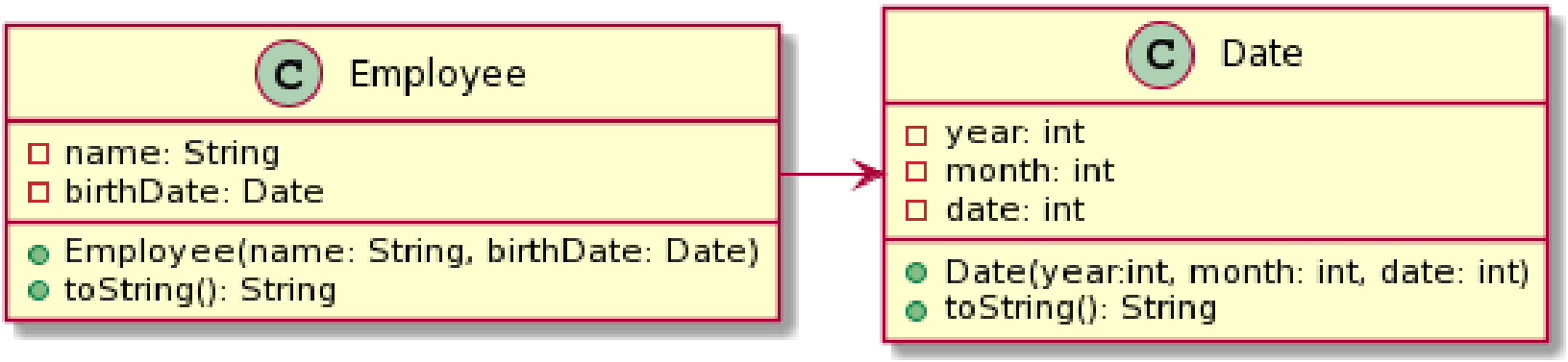


HAS-A 관계

- 객체 지향 프로그래밍에서 has-a 관계는 구성 관계(composition) 또는 집합 관계(aggregation)를 나타낸다.



HAS-A 관계의 예제


Employee
name: String
birthDate: Date
Employee(name: String, birthDate: Date)
toString(): String
Date
year: int
month: int
date: int
Date(year:int, month: int, date: int)
toString(): String

예시 : Date 클래스

```
public class Date {
  private int year;
  private int month;
  private int date;

  public Date(int year, int month, int date) {
  this.year = year;
  this.month = month;
  this.date = date;
  }
  @Override
  public String toString() {
  return "Date [year=" + year + ", month=" + month + ", date=" + date + "]";
  }
}
```

예시 : Employee 클래스 has-a Date 클래스

```
public class Employee {
  private String name;
  private Date birthDate;

  public Employee(String name, Date birthDate) {
  this.name = name;
  this.birthDate = birthDate;
  }

  @Override
  public String toString() {
  return "Employee [name=" + name + ", birthDate=" + birthDate + "]";
  }
}
```

예시 : Employee 클래스 has-a Date 클래스

```
public class EmployeeTest {
  public static void main(String[] args) {
  Date birth = new Date(1990, 1, 1);
  Employee employee = new Employee("홍길동", birth);
  System.out.println(employee);
  }
}
```

```
Employee [name=홍길동, birthDate=Date [year=1990, month=1, date=1]]
```

정적 메소드 오버라이딩

- 부모 클래스의 메소드 중에서 정적 메소드를 재정의(오버라이드)하면 부모 클래스 객체에서 호출되느냐 아니면 자식 클래스에서 호출되느냐에 따라서 호출되는 메소드가 달라진다.

```
public class Animal {
    public static void eat() {
        System.out.println("Animal의 정적 메소드 eat()");
    }
    public void sound() {
        System.out.println("Animal의 인스턴스 메소드 sound()");
    }
}
```

정적 메소드 오버라이딩

```
public class Cat extends Animal {
    public static void eat() {
        System.out.println("Cat의 정적 메소드 eat()");
    }
    public void sound() {
        System.out.println("Cat의 인스턴스 메소드 sound()");
    }
    public static void main(String[] args) {
        Cat myCat = new Cat();
        Cat.eat();
        Animal myAnimal = myCat;
        Animal.eat();
        myAnimal.sound();
    }
}
```

```
Cat의 정적 메소드 eat()
Animal의 정적 메소드 eat()
Cat의 인스턴스 메소드 sound()
```