

Class Inheritance

Byeongjoon Noh

Dept. of AI/Bigdata, SCH Univ.

powernoh@sch.ac.kr

Contents

1. Class inheritance
2. Casting
3. Method overriding
4. Abstraction
5. Interface

1. Class inheritance

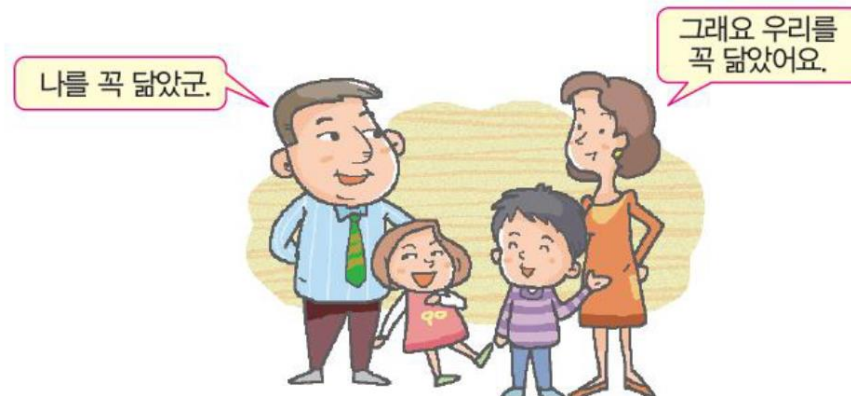
Inheritance

객체 지향 상속

- 자식이 부모 유전자를 물려 받는 것과 유사한 개념

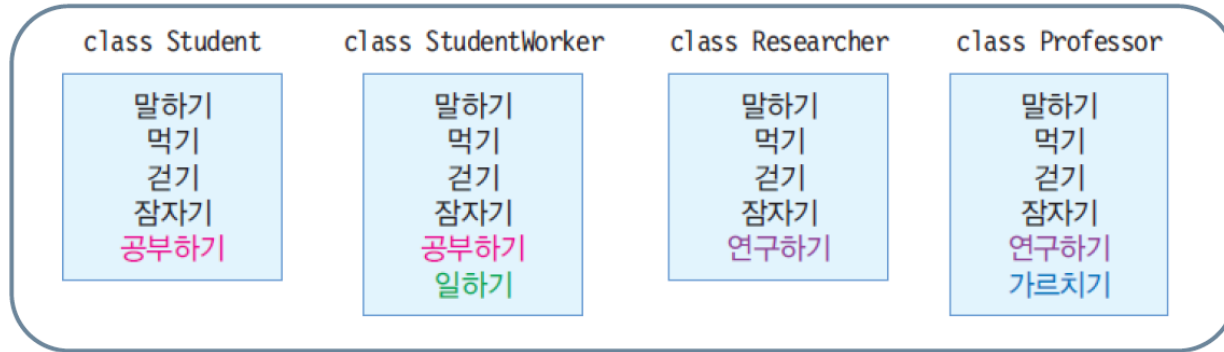


유산 상속

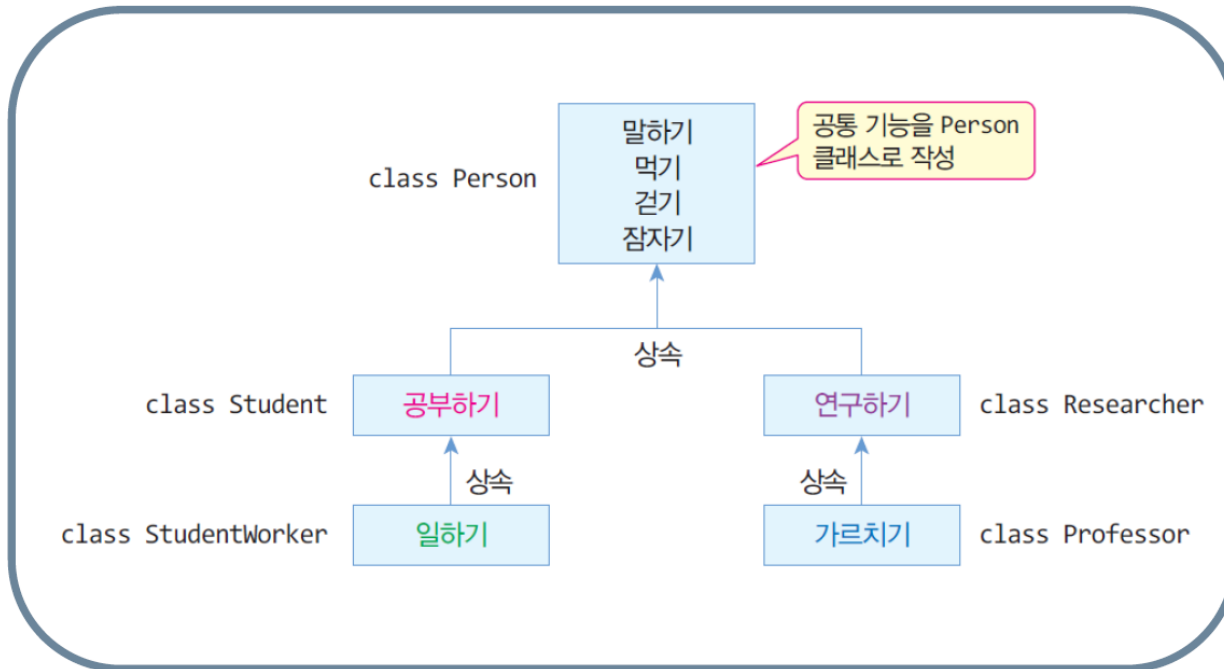


유전적 상속 : 객체 지향 상속

상속의 필요성



상속이 없는 경우
중복된 멤버를 가진
4 개의 클래스



상속을 이용한
경우 중복이 제거되고
간결해진 클래스 구조

Class inheritance

상속 선언

- extends 키워드를 활용
 - 부모 클래스를 물려받아 확장한다는 의미
- 부모 클래스 → 슈퍼 클래스 (super class)
- 자식 클래스 → 서브 클래스 (subclass)

```
class Point {  
    int x, y;  
    ...  
}  
class ColorPoint extends Point { // Point를 상속받는 ColorPoint 클래스 선언  
    ...  
}
```

서브 클래스 슈퍼 클래스

- 특징
 - ColorPoint는 Point를 물려 받으므로,
Point에 선언된 필드와 메소드를 별도의 선언 없이 활용 가능

Class inheritance

Example) (x, y)의 한 점을 표현하는 Point 클래스, 이를 상속받아 점에 색을 추가한 ColorPoint 선언 및 활용

```
class Point {  
    private 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 + ")");  
    }  
}  
  
// Point를 상속받은 ColorPoint 선언  
class ColorPoint extends Point {  
    private String color; // 점의 색  
    void setColor(String color) {  
        this.color = color;  
    }  
    void showColorPoint() { // 컬러 점의 좌표 출력  
        System.out.print(color);  
        showPoint(); // Point의 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();  
        cp.set(3, 4);          // Point 클래스의 set() 호출  
        cp.setColor("red");    // ColorPoint의 setColor() 호출  
        cp.showColorPoint();   // 컬러와 좌표 출력  
    }  
}
```

(1,2)
red(3,4)

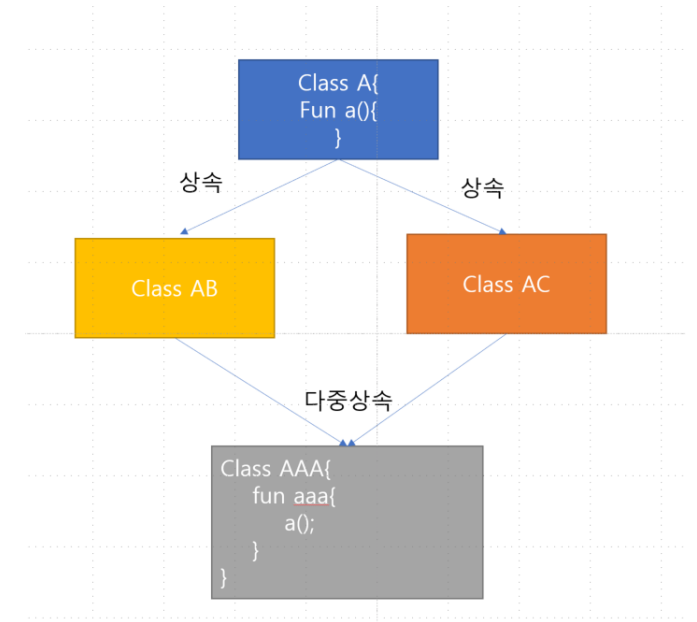
상속 특징

클래스 다중 상속 (multiple inheritance) 불가능

- 멤버 중복 생성 방지
- C++/Python은 다중 상속 가능
- Interface의 다중 상속은 허용

모든 Java 클래스는 묵시적으로 Object 클래스를 상속 받음

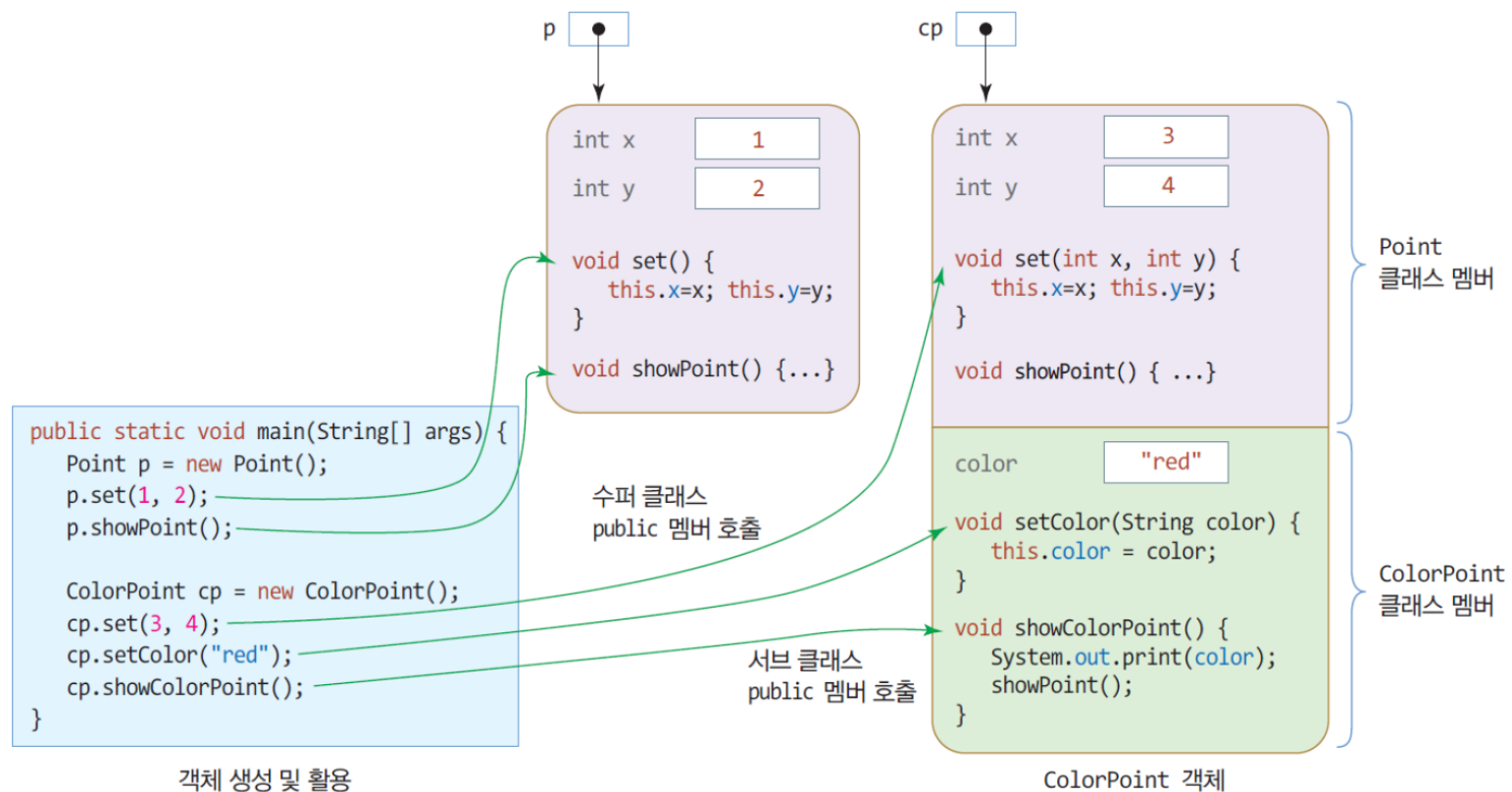
- java.lang.Object 클래스는 모든 클래스의 슈퍼 클래스



Subclass

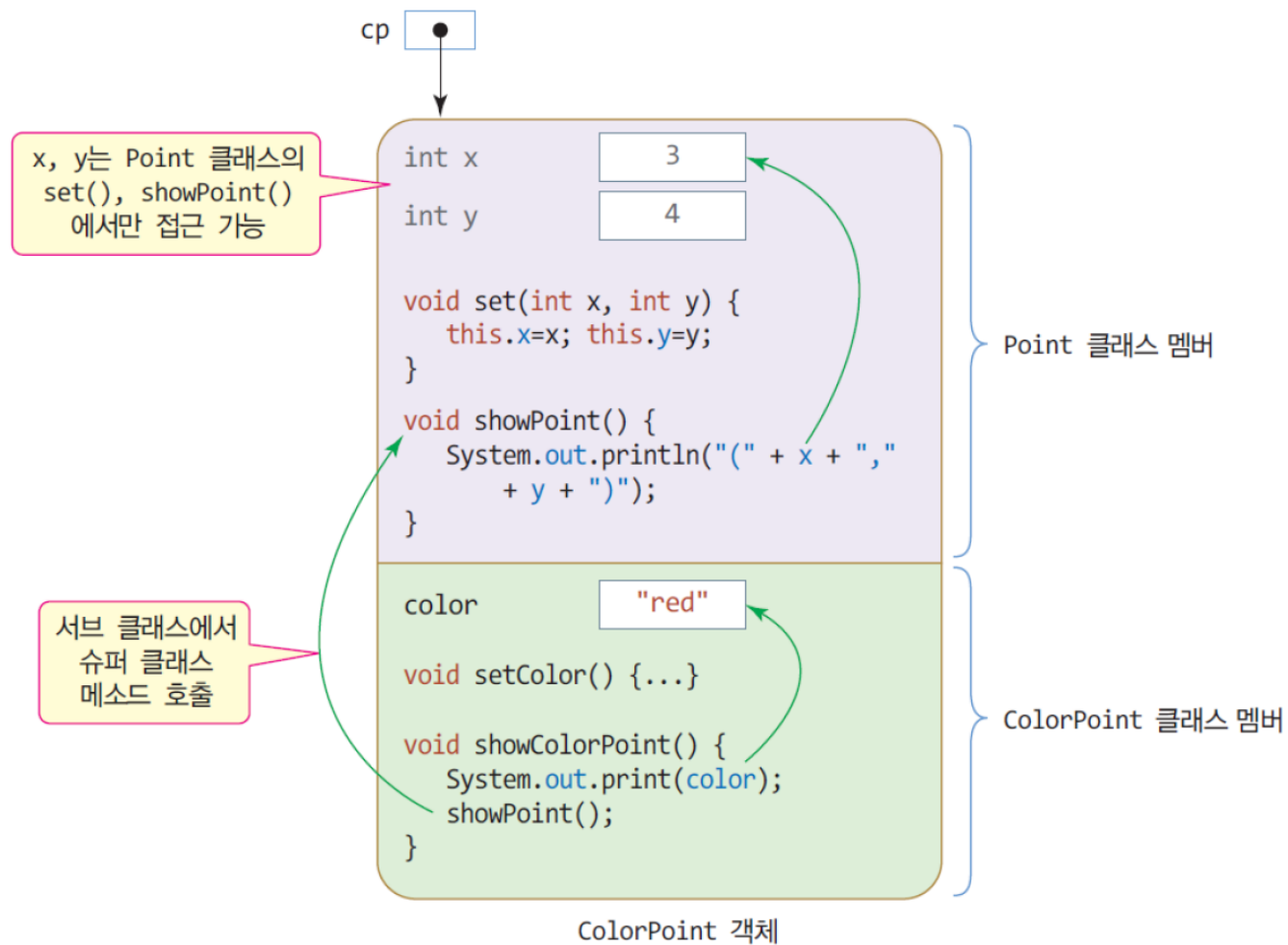
서브 클래스

- 슈퍼 클래스 객체와 서브 클래스 객체는 별개
- 서브 클래스 객체는 슈퍼 클래스 멤버를 포함함



슈퍼 클래스 멤버 접근

서브 클래스에서 슈퍼 클래스 멤버 접근



슈퍼 클래스 멤버 접근

슈퍼 클래스의 private 멤버

- 서브 클래스에서 접근 불가

슈퍼 클래스의 default 멤버

- 서브 클래스가 동일한 패키지에 있을 때 접근 가능

슈퍼 클래스의 public 멤버

- 서브 클래스는 항상 접근 가능

슈퍼 클래스의 protected 멤버

- 같은 패키지 내의 모든 클래스 접근 가능
- 동일 패키지 여부와 상관없이 서브 클래스는 접근 가능

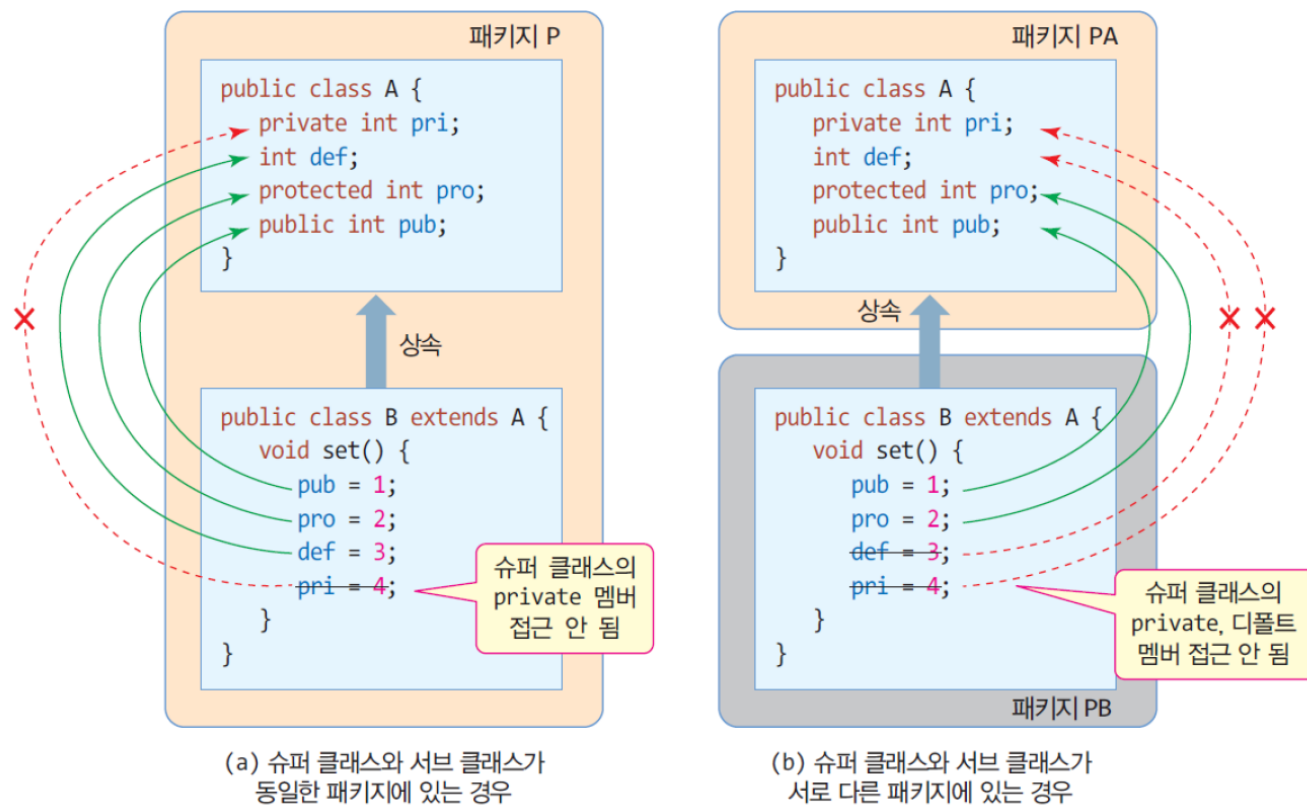
슈퍼 클래스 멤버에 접근하는 클래스 종류	슈퍼 클래스 멤버의 접근 지정자			
	private	디폴트	protected	public
같은 패키지의 클래스	×	○	○	○
다른 패키지의 클래스	×	×	×	○
같은 패키지의 서브 클래스	×	○	○	○
다른 패키지의 서브 클래스	×	×	○	○

(○는 접근 가능함을, ×는 접근 불가능함을 뜻함)

슈퍼 클래스 멤버 접근

protected 멤버에 대한 접근

- 같은 패키지의 모든 클래스에게 허용
- 서브 클래스는 패키지 동일 여부와 상관 없이 허용



슈퍼/서브 클래스 생성자

슈퍼 클래스 생성자가 실행되는 경우

- 슈퍼 클래스의 객체가 생성될 때
- 서브 클래스의 객체가 생성될 때
 - ➔ 슈퍼/서브 클래스의 생성자가 각각 실행되어 멤버 초기화 수행

서브 클래스 생성자가 실행되는 경우

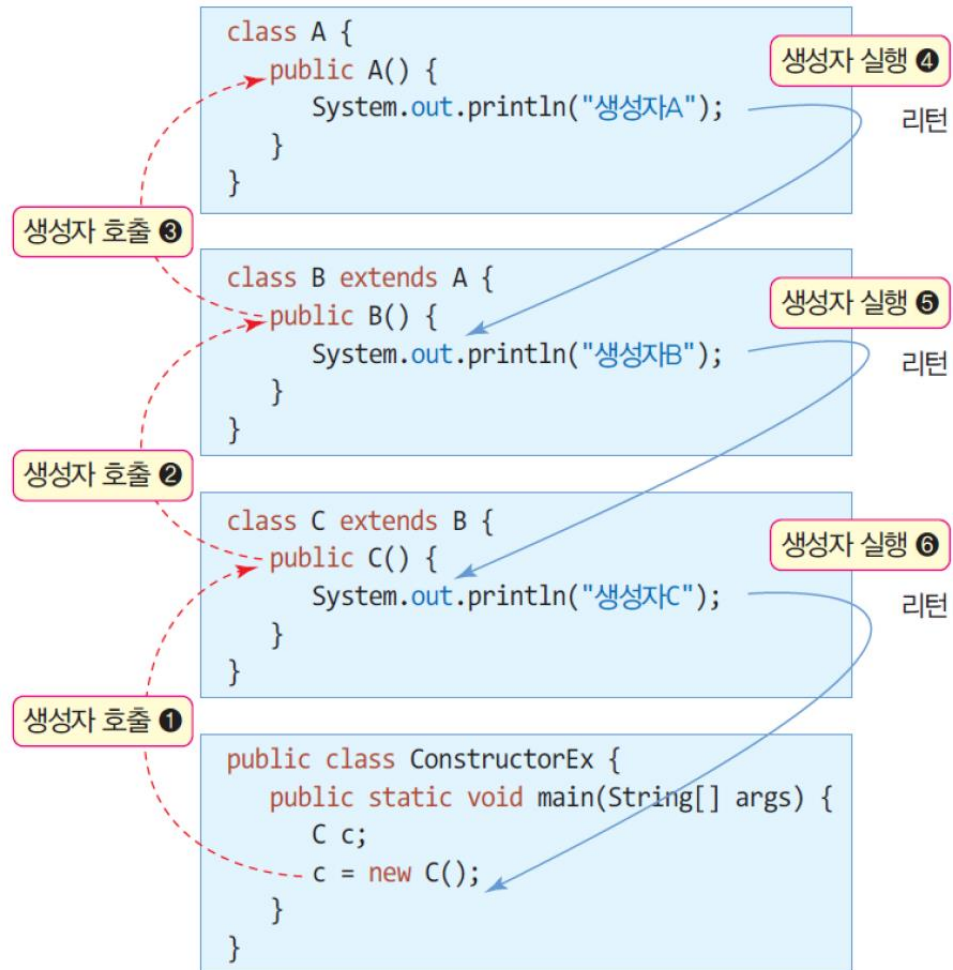
- 서브 클래스의 객체가 생성될 때

서브 클래스 생성 시 생성자 우선 순위

- 슈퍼 클래스 생성자가 먼저 실행 됨
- (호출 순서) 서브 클래스 생성 ➔ 서브 클래스 생성자 호출 ➔ 슈퍼 클래스 생성자 호출
 - ➔ 슈퍼 클래스 생성자 실행 ➔ 서브 클래스 생성자 실행

슈퍼/서브 클래스 생성자

Example) 슈퍼/서브 클래스의 생성자 호출 및 실행 관계 예제



생성자A
생성자B
생성자C

슈퍼/서브 클래스 생성자

슈퍼/서브 클래스 생성자 특징

- 슈퍼 클래스, 서브 클래스 각각 여러 개의 생성자 작성 가능
- 서브 클래스 객체 생성 시 ➔ 슈퍼 클래스 생성자 1개와 서브 클래스 생성자 1개가 실행됨
 - 결정방식
 - 1) 개발자의 명시적 선택
 - 서브 클래스 개발자가 슈퍼 클래스의 생성자 명시적 선택 (super() 키워드)
 - 2) 컴파일러가 기본 생성자 선택
 - 슈퍼 클래스 명시되어 있지 않을 때
 - ➔ 컴파일러가 자동으로 슈퍼 클래스의 기본 생성자 선택

슈퍼/서브 클래스 생성자

Example) 컴파일러에 의한 슈퍼 클래스 기본 생성자 호출 예제

- 개발자가 별도의 슈퍼 클래스 생성자 지정하지 않은 경우

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

```
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  
    }  
}
```

생성자A
생성자B

슈퍼/서브 클래스 생성자

Example) 컴파일러에 의한 슈퍼 클래스 기본 생성자 호출 예제

- 개발자가 별도의 슈퍼 클래스 생성자 지정하지 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");  
    }  
}
```

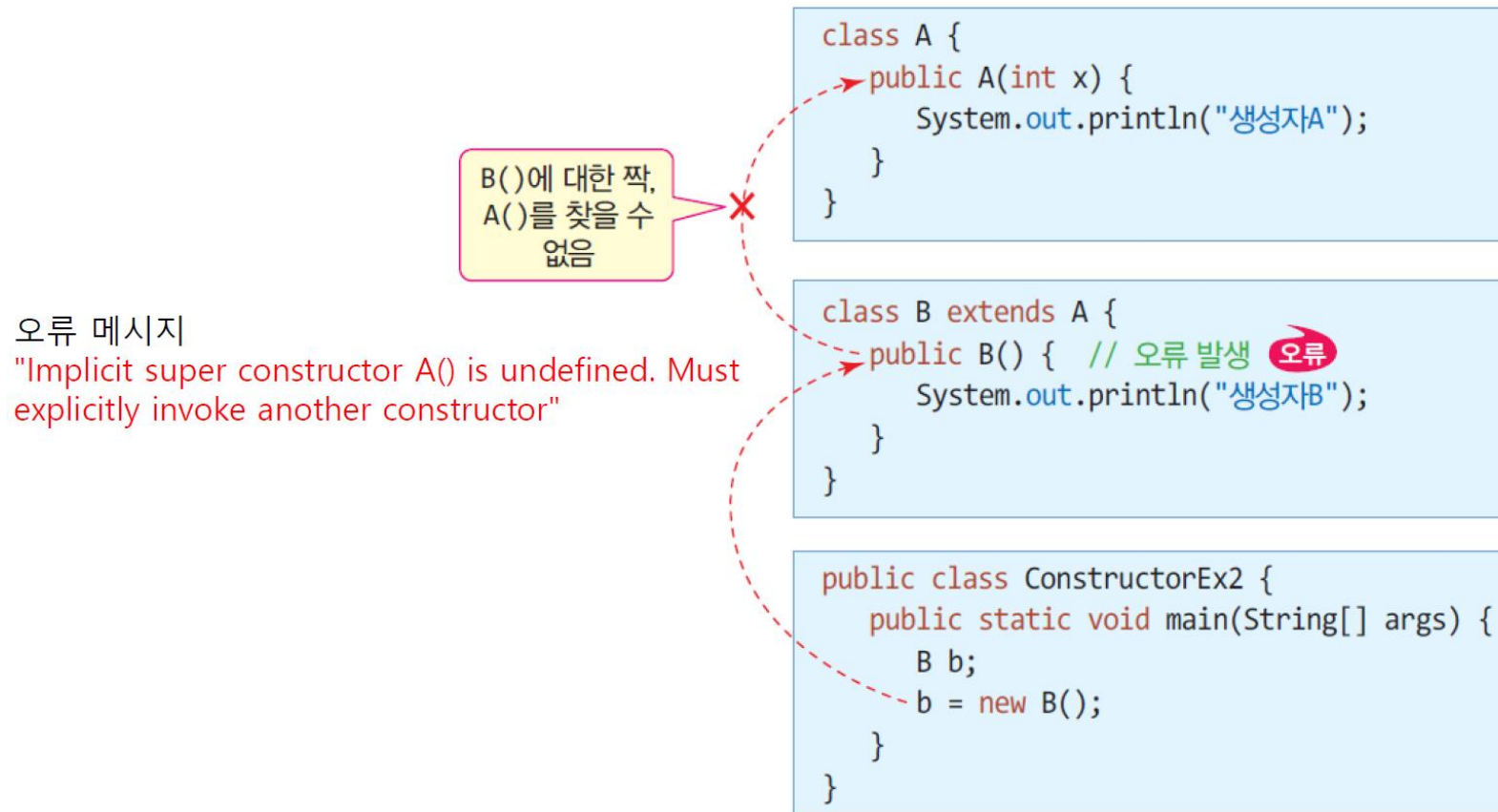
```
public class ConstructorEx3 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(5);  
    }  
}
```

생성자A
매개변수생성자B

슈퍼/서브 클래스 생성자

Example) 컴파일러에 의한 슈퍼 클래스 기본 생성자 호출 시 오류 예제

- 슈퍼 클래스에 기본 생성자가 없음



super()

서브 클래스에서 명시적으로 슈퍼 클래스의 생성자 선택 호출

- 사용 방법
 - `super(parameters);`
 - 인자를 이용하여 슈퍼 클래스의 적당한 생성자 호출
 - 반드시 서브 클래스 생성자 코드의 제일 첫 줄에 위치 해야함!

super()

Example) super()로 슈퍼 클래스 생성자의 명시적 선택 예제

```
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); // 첫 줄에 와야 함  
        System.out.println("매개변수생성자B" + x);  
    }  
}
```

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

매개변수생성자A5
매개변수생성자B5

super()

Example) ColorPoint 클래스의 생성자에서 super()를 활용한 예제

```
class Point {  
    private int x, y; // 한 점을 구성하는 x, y 좌표  
    Point() {  
        this.x = this.y = 0;  
    }  
    Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    void showPoint() { // 점의 좌표 출력  
        System.out.println("(" + x + "," + y + ")");  
    }  
}  
  
class ColorPoint extends Point {  
    private String color; // 점의 색  
    ColorPoint(int x, int y, String color) {  
        super(x, y); // Point의 생성자 Point(x, y) 호출  
        this.color = color;  
    }  
    void showColorPoint() { // 컬러 점의 좌표 출력  
        System.out.print(color);  
        showPoint(); // Point 클래스의 showPoint() 호출  
    }  
}
```

x=5,
y=6

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

blue(5,6)

x=5, y=6,
color = "blue" 전달

2. Casting

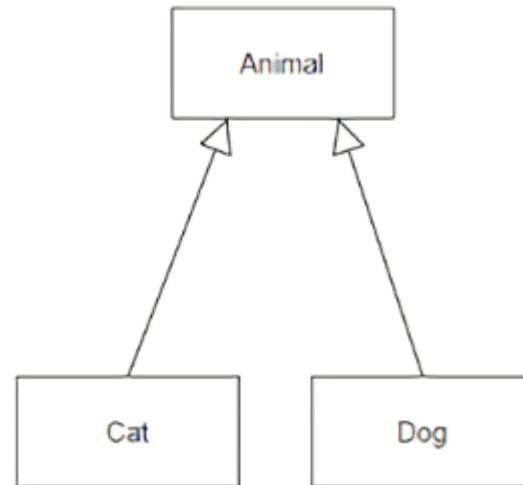
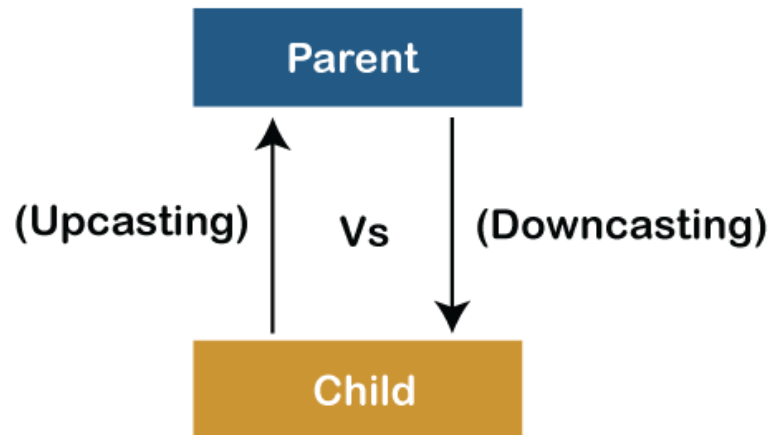
Upcasting

Casting의 개념

- 데이터 간의 타입 변환 (primitive 타입, int ➔ double 등)
- 상속 관계에 있는 부모와 자식 클래스 간의 형 변환 (reference 타입)
 - 형제 클래스간에는 casting 불가능 (타입이 다름)

Upcasting

- 서브(자식) 클래스에 있는 객체가 슈퍼(부모) 클래스 타입으로 형 변환
- 서브 클래스의 레퍼런스를 슈퍼 클래스 레퍼런스에 대입
 - ➔ 슈퍼 클래스 레퍼런스로 서브 클래스 객체를 가리키게 됨



Upcasting

Example) 업캐스팅 예제

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

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

class Student extends Person {
    String grade;
    String department;

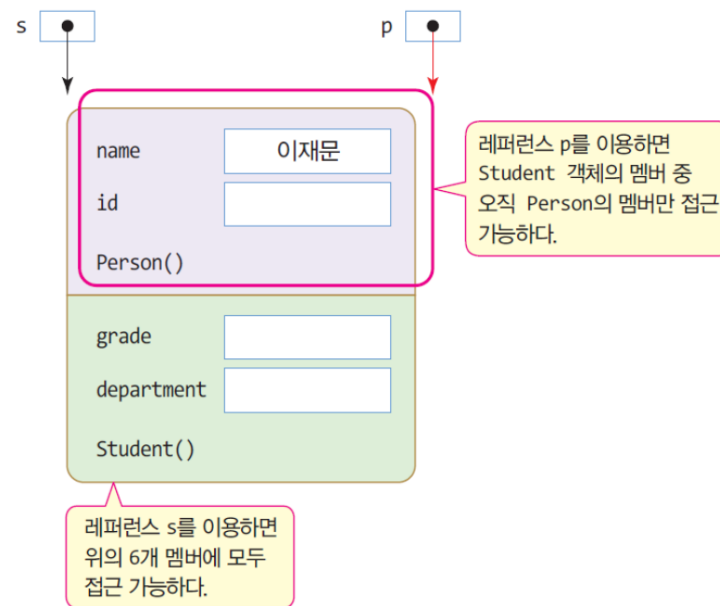
    public Student(String name) {
        super(name);
    }
}

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
    }
}
```

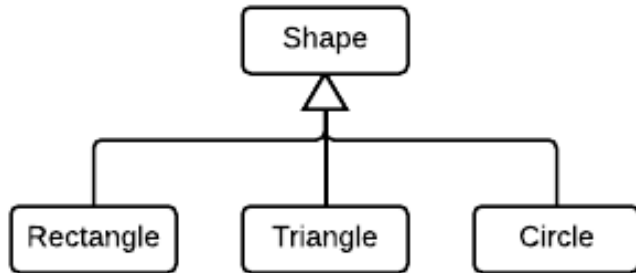
오류



Upcasting

Upcasting의 목적

- 공통적으로 할 수 있는 부분을 만들어 간단하게 다루기 위해서 ➔ 하나의 인스턴스로 묶어 관리
- 추후 downcasting으로 서브 클래스의 고유 메소드를 활용



```
Rectangle[] r = new Rectangle[];  
r[0] = new Rectangle();  
r[1] = new Rectangle();
```

```
Triangle[] t = new Triangle[];  
t[0] = new Triangle();  
t[1] = new Triangle();
```

```
Circle[] c = new Circle[];  
c[0] = new Circle();  
c[1] = new Circle();
```



```
Shape[] s = new Shape[];  
s[0] = new Rectangle();  
s[1] = new Rectangle();  
s[2] = new Triangle();  
s[3] = new Triangle();  
s[4] = new Circle();  
s[5] = new Circle();
```

Downcasting

Downcasting

- 슈퍼 클래스 레퍼런스를 서브 클래스 레퍼런스에 대입
- 업캐스팅된 것을 다시 원래대로 되돌리는 것
- 반드시 명시적 타입 변환 지정

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

```
Person p = new Student("이재문"); // 업캐스팅
```

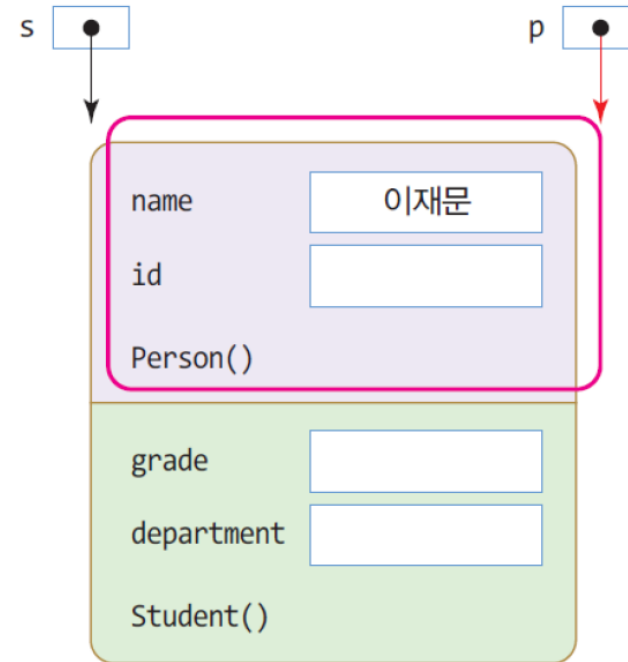
```
Student s = (Student)p; // 다운캐스팅, 강제타입변환
```

Downcasting

Example) Downcasting 예제

```
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  
    }  
}
```

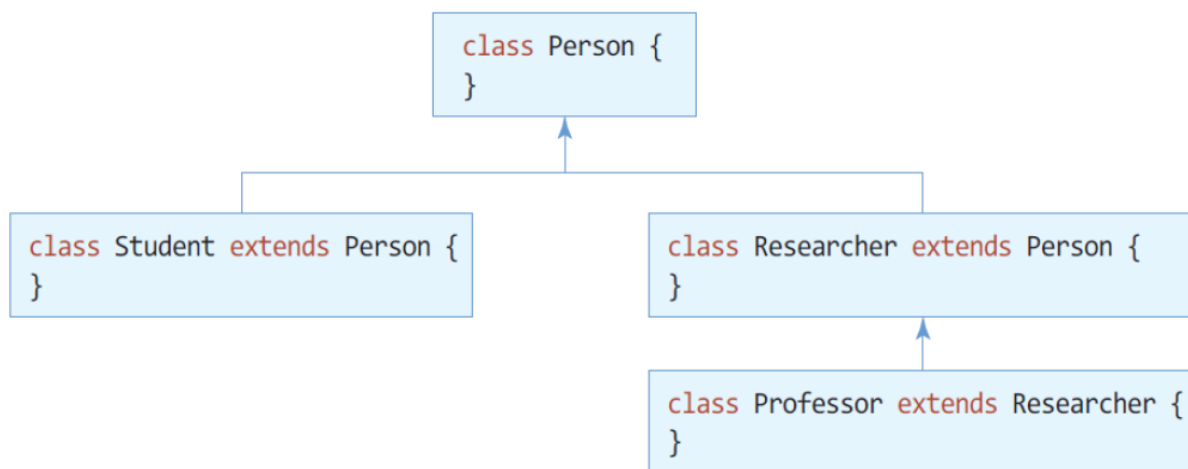
이재문



instanceof

업캐스팅된 레퍼런스는 객체의 실제 타입을 구분하기 어려움

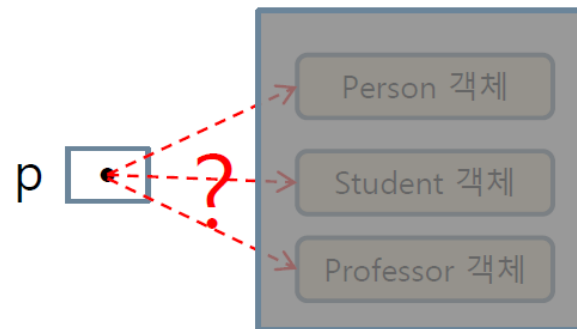
- ➔ instanceof 키워드 활용



샘플 클래스 계층 구조

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

Person 타입의 레퍼런스 p로 업캐스팅



p가 가리키는 객체가
Person 객체인지, Student 객체인지,
Professor 객체인지 구분하기 어려움

instanceof

instanceof 연산자

- 레퍼런스가 가리키는 객체의 타입 식별
 - true 또는 false 반환

객체레퍼런스 instanceof 클래스타입

- Example) instanceof 연산자 사용 예제

```
Person p = new Professor();
```

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

```
if(p instanceof Person)           // true
if(p instanceof Student)          // false. Student를 상속받지 않기 때문
if(p instanceof Researcher)       // true
if(p instanceof Professor)        // true
```

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



if(3 instanceof **int**) // 문법 오류. instanceof는 객체에 대한 레퍼런스에만 사용

instanceof

Example) instanceof 연산자 활용 예제

```
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() -> ");    print(new Student());
        System.out.print("new Researcher() -> "); print(new Researcher());
        System.out.print("new Professor() -> ");  print(new Professor());
    }
}
```

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

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

3. Method Overriding

Method overriding

개념

- 서브 클래스에서 슈퍼 클래스의 메소드 중복 작성
- 슈퍼 클래스의 메소드 무력화 ➔ 항상 서브 클래스에 오버라이딩한 메소드의 실행 보장
- “메소드 무시하기”

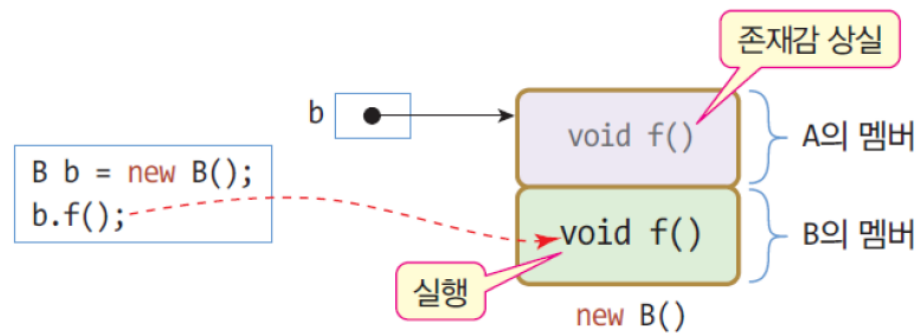
조건

- 슈퍼 클래스 멤버 메소드의 원형 (메소드 이름, 인자 타입 및 개수, 리턴 타입) 동일하게 작성

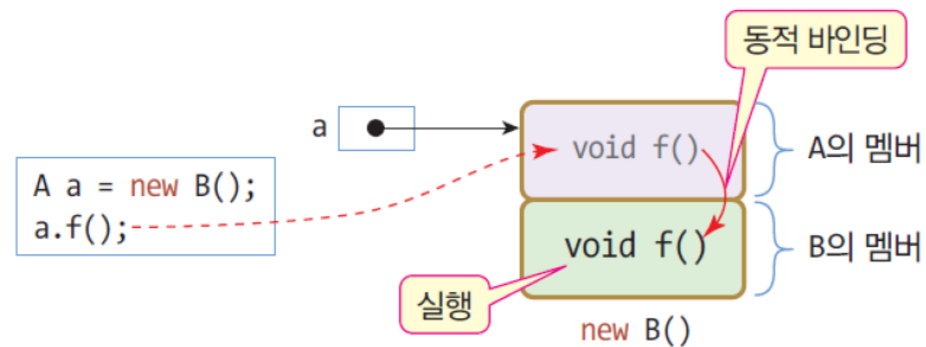
Method overriding

```
class A {  
    void f() {  
        System.out.println("A의 f() 호출");  
    }  
}  
class B extends A {  
    void f() { // 클래스 A의 f()를 오버라이딩  
        System.out.println("B의 f() 호출");  
    }  
}
```

(a) 오버라이딩된 메소드, **B의 f()** 직접 호출



(b) **A의 f()**를 호출해도, 오버라이딩된 메소드, **B의 f()**가 실행됨

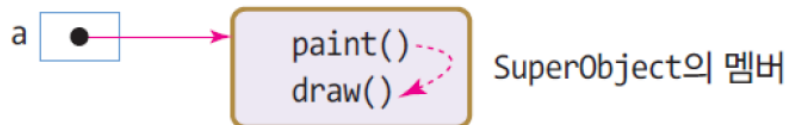


Method overriding

* 오버라이딩 메소드가 항상 호출된다.

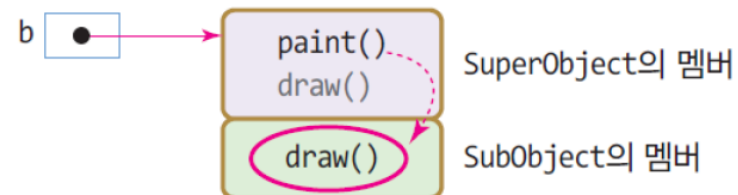
```
public class SuperObject {  
    protected String name;  
    public void paint() {  
        draw();  
    }  
    public void draw() {  
        System.out.println("Super Object");  
    }  
    public static void main(String [] args) {  
        SuperObject a = new SuperObject();  
        a.paint();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

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();  
    }  
}
```

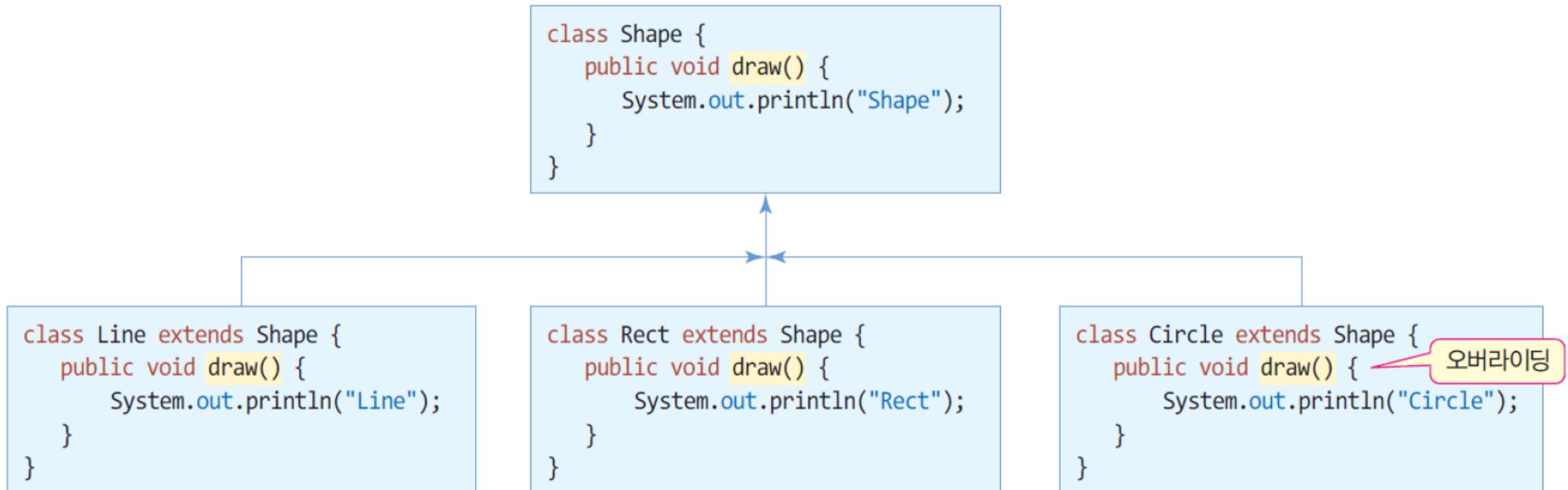
Sub Object



Method overriding

목적

- 다형성 실현
- 하나의 인터페이스(같은 이름)에 서로 다른 구현
- 슈퍼 클래스의 메소드를 서브 클래스에서 각각 목적에 맞게 다르게 구현



Method overriding

Example) Shape 클래스의 draw()메소드를 Line, Circle, Rect 클래스에서 오버라이딩하는 예제

```
class Shape { // 도형의 슈퍼 클래스
    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");
    }
}
```

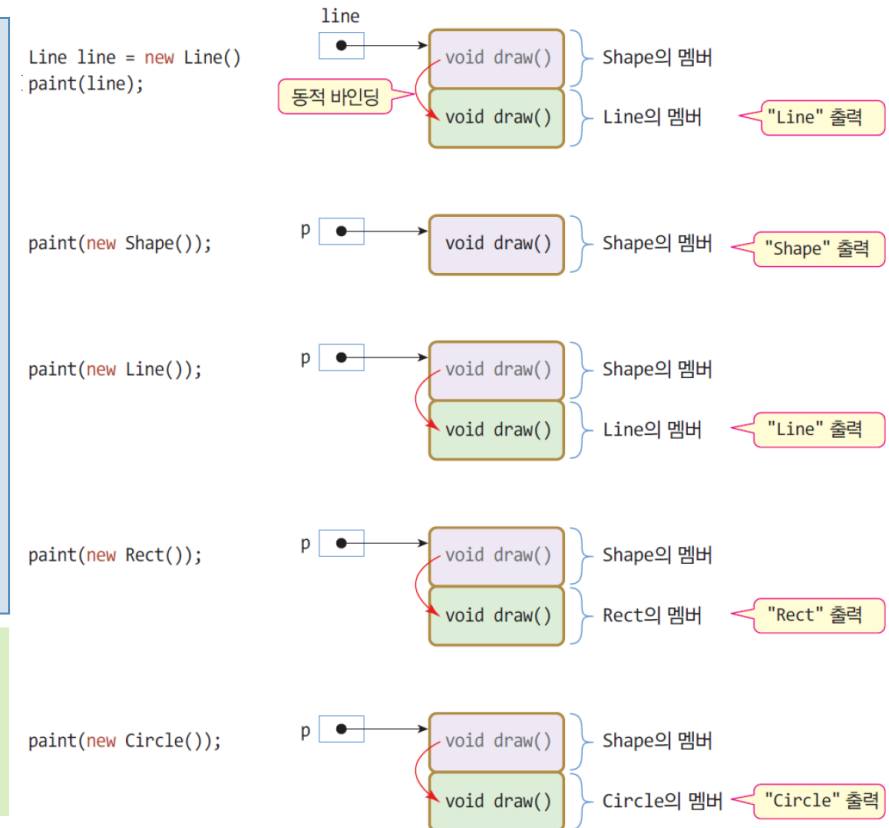
동적바인딩

```
public class MethodOverridingEx {
    static void paint(Shape p) { // Shape을 상속받은 객체들이
        // 매개 변수로 넘어올 수 있음
        p.draw(); // p가 가리키는 객체에 오버라이딩된 draw() 호출.
        // 동적바인딩
    }

    public static void main(String[] args) {
        Line line = new Line();
        paint(line); // Line의 draw() 실행. "Line" 출력

        paint(new Shape()); // Shape의 draw() 실행. "Shape" 출력
        paint(new Line()); // 오버라이딩된 메소드 Line의 draw() 실행
        paint(new Rect()); // 오버라이딩된 메소드 Rect의 draw() 실행
        paint(new Circle()); // 오버라이딩된 메소드 Circle의 draw() 실행
    }
}
```

Line
Shape
Line
Rect
Circle



Method overriding

동적 바인딩 (Dynamic Binding)

- 다형성을 사용하여 메소드를 호출 할 때 발생하는 현상
 - 실행시간 (Runtime)에 속성이 결정됨
 - 실제 참조하는 객체 == 서브 클래스 → 서브 클래스의 메소드 호출
 - * Binding: 프로그램에 사용된 구성 요소의 실제 값/속성을 결정짓는 행위
-
- 정적 바인딩 (Static Binding)
 - 컴파일 (Compile) 시간에 속성이 결정됨
 - 상속관계에서 오버라이딩되지 않은 메소드를 호출할 때
 - super 키워드를 통해 메소드 호출할 때
 - Static으로 명시된 메소드를 호출 할 때
 - Static 특징: Chapter 7. Class 강의자료 32page 참조

Method overriding

Example) Dynamic / Static binding 예제

```
public static void main(String[] args) {  
    SuperClass superClass = new SuperClass();  
    superClass.methodA();  
    superClass.methodB();  
  
    SuperClass subClass = new SubClass();  
    subClass.methodA();  
    subClass.methodB();  
}
```

```
public class SuperClass {  
    public SuperClass() { System.out.println("SuperClass Constructor with params"); }  
    void methodA() { System.out.println("SuperClass A "); }  
    static void methodB() { System.out.println("SuperClass B"); }  
}
```

```
public class SubClass extends SuperClass {  
    public SubClass() { System.out.println("SubClass Constructor"); }  
    @Override  
    void methodA() { System.out.println("SubClass A"); }  
    static void methodB() { System.out.println("SubClass B"); }  
}
```

Method overriding

Method overloading vs. overriding

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 선언하여 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

4. Abstraction

Abstract class

추상 메소드 (Abstract method)

- abstract로 선언된 메소드
 - 메소드의 코드는 없고 원형만 선언 ➔ 서브 클래스에서 활용하도록...

```
public abstract String getName(); // 추상 메소드
```



```
public abstract String fail() { return "Good Bye"; } // 추상 메소드 아님. 컴파일 오류
```

추상 클래스 (Abstract class)

- 추상 메소드를 가지며, abstract로 선언된 클래스
- 추상 메소드 없이 abstract로 선언한 클래스
- 추상 메소드를 가지면
반드시 추상 클래스로 선언되어야 함

```
// 추상 메소드를 가진 추상 클래스
```

```
abstract class Shape {  
    Shape() { ... }  
    void edit() { ... }  
  
    abstract public void draw(); // 추상 메소드  
}
```

```
// 추상 메소드 없는 추상 클래스
```

```
abstract class JComponent {  
    String name;  
    void load(String name ) {  
        this.name= name;  
    }  
}
```




```
class fault { // 오류. 추상 메소드를 가지고 있으므로 abstract로 선언되어야 함  
    abstract void f(); // 추상 메소드  
}
```

Abstract class

추상 클래스 (Abstract class) 특징

- 추상 클래스 → 온전한 클래스가 아님 → 인스턴스화 불가능 (객체 생성 불가)

 JComponent p; p = new JComponent(); Shape obj = new Shape() ;	// 오류 없음. 추상 클래스의 레퍼런스 선언 // 컴파일 오류. 추상 클래스의 인스턴스 생성 불가 // 컴파일 오류. 추상 클래스의 인스턴스 생성 불가
--	---

컴파일 오류 메시지

Unresolved compilation problem: Cannot instantiate the type Shape

추상 클래스의 상속과 구현

추상 클래스의 상속

- 추상 클래스를 상속받으면 추상 클래스가 됨
- 서브 클래스도 `abstract`로 선언해야 함

```
abstract class A { // 추상 클래스
    abstract int add(int x, int y); // 추상 메소드
}
abstract class B extends A { // 추상 클래스
    void show() { System.out.println("B"); }
}
```



```
A a = new A(); // 컴파일 오류. 추상 클래스의 인스턴스 생성 불가
B b = new B(); // 컴파일 오류. 추상 클래스의 인스턴스 생성 불가
```

추상 클래스의 상속과 구현

추상 클래스 구현

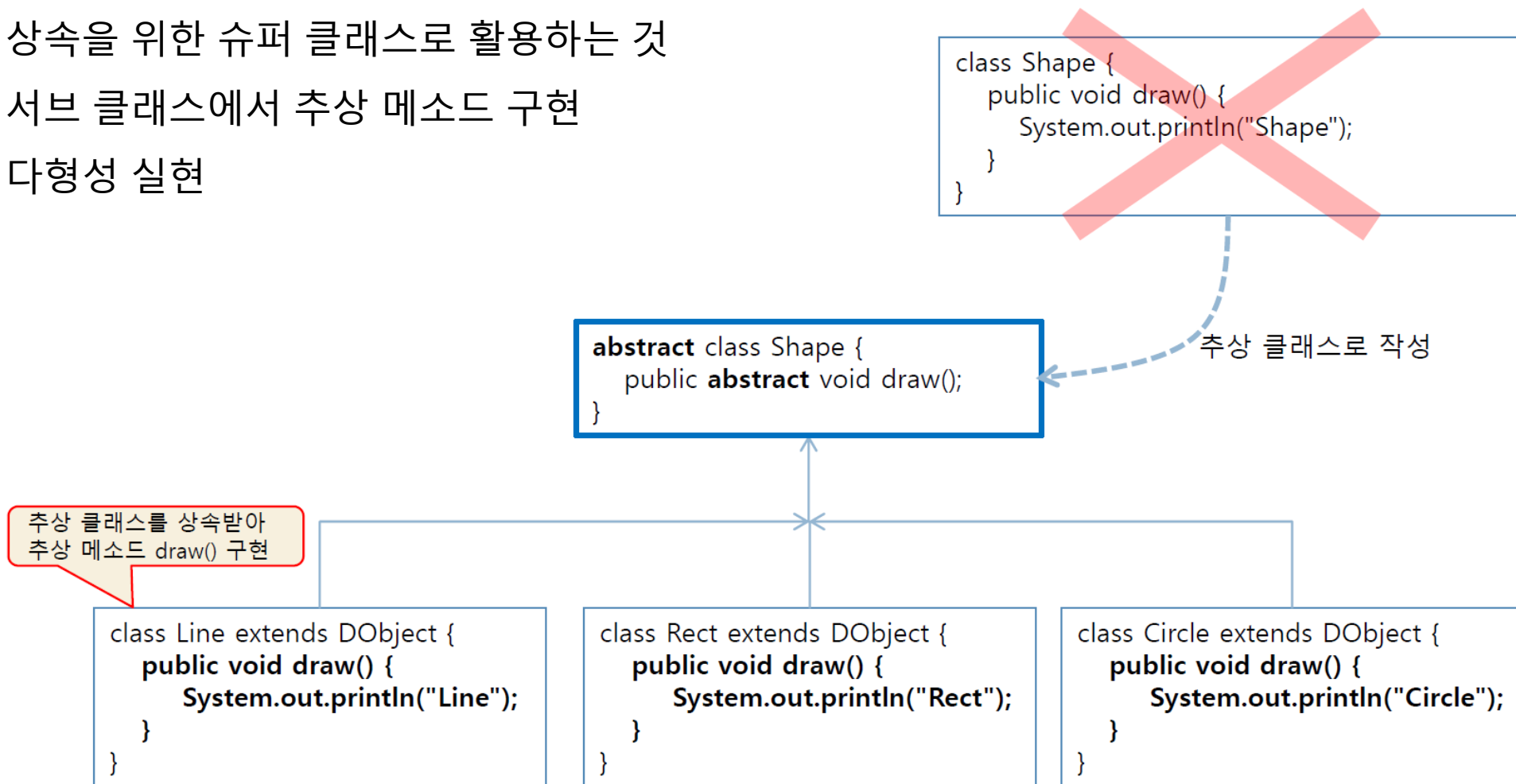
- 서브 클래스에서 슈퍼 클래스의 추상 메소드 구현 (오버라이딩)
- 추상 클래스를 구현한 서브 클래스는 더 이상 추상 클래스가 아님

```
class C extends A { // 추상 클래스 구현. C는 정상 클래스
    int add(int x, int y) { return x+y; } // 추상 메소드 구현. 오버라이딩
    void show() { System.out.println("C"); }
}
...
C c = new C(); // 정상
```

추상 클래스의 상속과 구현

추상 클래스의 목적

- 상속을 위한 슈퍼 클래스로 활용하는 것
- 서브 클래스에서 추상 메소드 구현
- 다형성 실현



추상 클래스의 상속과 구현

Example) 추상클래스 Calculator를 상속받는 GoodCalc 클래스를 구현하는 예제

```
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);  
}
```

```
public class GoodCalc extends Calculator {  
    public int add(int a, int b) { // 추상 메소드 구현  
        return a + b;  
    }  
    public int subtract(int a, int b) { // 추상 메소드 구현  
        return a - b;  
    }  
    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. Interface

Java Interface

인터페이스 (interface)

- 상수와 추상 메소드로만 구성됨 (변수 X)
- 인터페이스 선언
 - interface 키워드를 활용
 - 인터페이스 내의 모든 변수는 상수이고, 메소드는 추상 메소드임
 - 인터페이스의 객체 생성 불가

```
interface PhoneInterface {  
    int BUTTONS = 20; // 상수 필드 선언  
    void sendCall(); // 추상 메소드  
    void receiveCall(); // 추상 메소드  
}
```

public interface로서 public 생략 가능

public static final로서 public static final 생략 가능

abstract public 으로서 abstract public 생략 가능



`new PhoneInterface();` // 오류. 인터페이스의 객체를 생성할 수 없다.

Java Interface

인터페이스 상속

- 인터페이스 간 상속 가능 (extends 키워드 활용)
- 인터페이스를 상속하여 확장된 인터페이스 작성 가능

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

- 다중 상속 허용

```
interface MusicPhoneInterface extends PhoneInterface, MP3Interface {  
    .....  
}
```

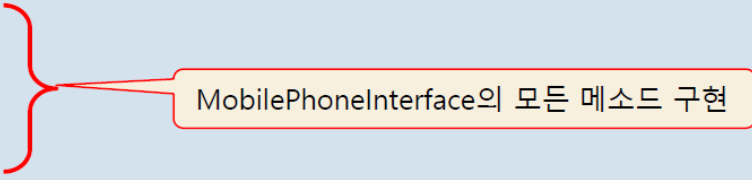
인터페이스 활용

인터페이스 구현

- 인터페이스를 상속받아, 모든 추상 메소드를 구현한 클래스 선언
- implements 키워드를 활용하여 인터페이스를 구현함

```
class FeaturePhone implements MobilePhoneInterface { // 인터페이스 구현
    public void sendCall() { ... }
    public void receiveCall() { ... }
    public void sendSMS() { ... }
    public void receiveSMS() { ... }

    // 다른 메소드 추가 가능
    public int getButtons() { ... }
}
```



MobilePhoneInterface의 모든 메소드 구현

- 여러 개의 인터페이스 동시 구현도 가능
- 클래스 상속과 인터페이스 동시 구현 가능

인터페이스 활용

Example) 인터페이스 구현과 동시에 슈퍼 클래스 상속

```
interface PhoneInterface {
    int BUTTONS = 20;
    void sendCall();
    void receiveCall();
}

interface MobilePhoneInterface
    extends PhoneInterface {
    void sendSMS();
    void receiveSMS();
}

interface MP3Interface {
    public void play();
    public void stop();
}

class PDA {
    public int calculate(int x, int y) {
        return x + y;
    }
}
```

```
// SmartPhone 클래스는 PDA를 상속받고,
// MobilePhoneInterface와 MP3Interface 인터페이스에 선언된
// 메소드를 모두 구현
```

```
class SmartPhone extends PDA implements
    MobilePhoneInterface, MP3Interface {
    public void sendCall() { System.out.println("전화 걸기"); }
    public void receiveCall() { System.out.println("전화 받기"); }
    public void sendSMS() { System.out.println("SMS 보내기"); }
    public void receiveSMS() { System.out.println("SMS 받기"); }

    public void play() { System.out.println("음악 재생"); }
    public void stop() { System.out.println("재생 중지"); }

    public void schedule() { System.out.println("일정 관리"); }
}

public class InterfaceEx {
    public static void main(String [] args) {
        SmartPhone p = new SmartPhone();
        p.sendCall();
        p.play();
        System.out.println(p.calculate(3,5));
        p.schedule();
    }
}
```

MobilePhoneInterface
모든 메소드 구현

MP3Interface의
모든 메소드 구현

새로운
메소드 추가

전화 걸기
음악 재생
8
일정 관리

Summary

추상 클래스 vs Interface

- 둘 다 추상화 개념을 구현하는데 사용됨 (둘 다 메소드를 구현해야 함)

비교 항목	추상 클래스	Interface
예약어	abstract class	interface
상속	단일 상속	다중 상속 가능
메소드	추상 메소드, 일반 메소드 모두 가질 수 있음	추상 메소드, default 메소드, static 메소드,
변수	인스턴스 변수, 클래스 변수 등	상수만 가질 수 있음
구현	일부 메소드는 구현, 나머지는 하위 클래스에서 구현	메소드의 정의만 가능. 모든 메소드는 하위 클래스에서 구현
사용	일반적인 구현 + 확장	일반적인 행동을 정의
상속 시 예약어	extends	implements

End of slide