

복습 : 클래스 생성자

생성자

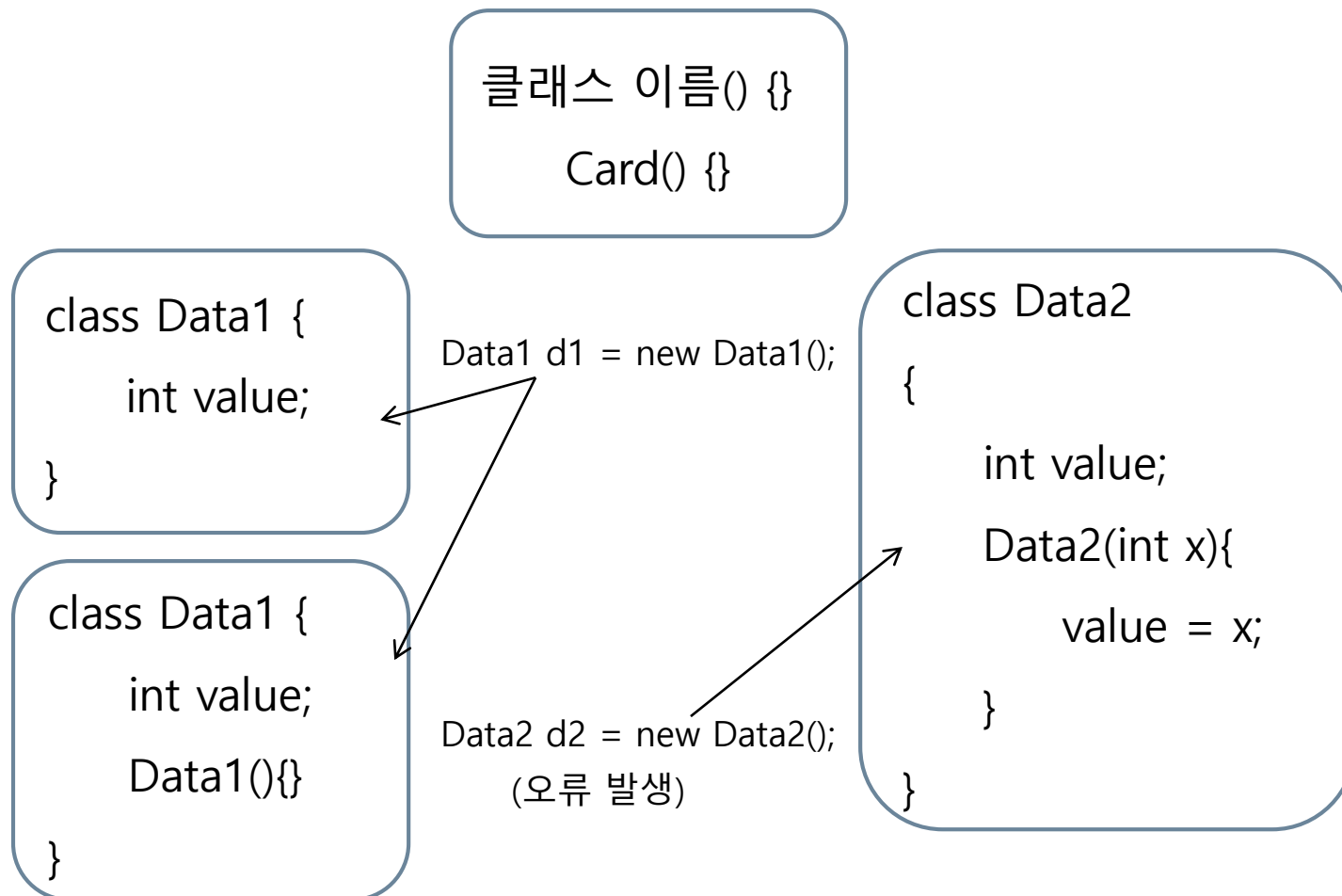
- 생성자(Constructor) 정의
 - ▣ 객체의 생성과 동시에 자동으로 호출되는 메서드
 - ▣ 생성자를 만들지 않으면 디폴트 생성자가 자동으로 만들어진다.
 - ▣ 이 때 만들어지는 디폴트 생성자는 하는 일이 없다.
- 디폴트 생성자(Default Constructor)
 - ▣ 생성자가 없는 클래스를 위해서 컴파일러는 매개변수를 가지지 않는 생성자를 자동으로 만들어 준다.

- * 생성자의 이름은 클래스의 이름과 같아야 한다.
- * 생성자는 리턴 값이 없다.
- * 생성자도 오버로딩 가능하다.

디폴트 생성자

□ 디폴트 생성자

- ▣ 클래스 내에 생성자가 하나도 없을 때 자동으로 추가되는 생성자



매개변수가 있는 생성자

```
class Car
```

```
{
```

```
    String color;
```

```
    String gearType;
```

```
    int door;
```

```
    Car() {}
```

```
    Car(String c, String g, int d){
```

```
        color = c;
```

```
        gearType = g;
```

```
        door = d; }
```

```
}
```

```
Car c = new Car();
```

```
c.color = "white";
```

```
c.gearType = "auto";
```

```
c.door = 4;
```

```
Car c = new Car("white","auto",4);
```

this 생성자

□ this(), this(매개변수)

- 같은 클래스의 다른 생성자를 호출할 때 사용
- 생성자 이름 대신에 this 사용

```
Car() {  
    color = "white";  
    gearType = "auto";  
    door = 4;  
}
```

```
Car() {  
    this("white","auto",4);  
}
```

```
Car(String color, String gearType, int door){  
    this.color = color;  
    this.gearType = gearType;  
    this.door = door;  
}
```

this

- 인스턴스 자기를 가리키는 참조변수
- 인스턴스 주소 저장
- 모든 인스턴스 메서드에 지역변수로 숨겨진채 존재

복사 생성자

□ 복사 생성자

- ▣ 인스턴스의 복사를 위한 생성자

```
Car(Car c){  
    color = c.color;  
    gearType = c.gearType;  
    door = c.door;  
}
```

```
Car c1 = new Car();  
Car c2 = new Car(c1);
```



c1 과 c2는 같은 값을 갖고 있는
서로 다른 인스턴스

초기화 블록

- 클래스 초기화 블록
 - ▣ 클래스 변수의 복잡한 초기화에 사용
- 인스턴스 초기화 블록
 - ▣ 인스턴스 변수의 복잡한 초기화에 사용

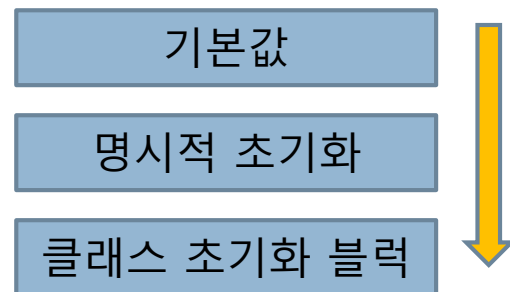
```
class InitBlock {  
  
    static { /* 클래스 초기화 블록 */ }  
  
    { /* 인스턴스 초기화 블록 */ }
```

초기화 블록 내에는 조건문, 반복문,
예외처리 구문등 사용 가능

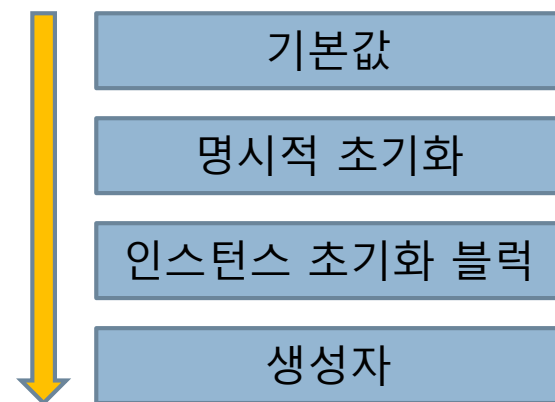
초기화 블록

```
class InitBlock {  
    static int[] arr = new int[10];  
    static {  
        for(int i=0; i<arr.length; i++)  
            arr[i] = (int)(Math.random()*10)+1  
    }  
  
    { System.out.println("인스턴스가 생성되었습니다. ")  
      InitBlock(){}  
    .....  
}
```

클래스 변수 초기화 순서



인스턴스 변수 초기화 순서



05. 상속

학습 목표

1. 객체 지향 상속과 자바 상속 개념 이해
2. 클래스 상속 작성 및 객체 생성
3. protected 접근 지정
4. 상속 시 생성자의 실행 과정
5. 업캐스팅과 instanceof 연산자
6. 메소드 오버라이딩과 동적 바인딩의 이해 및 활용
7. 추상 클래스
8. 인터페이스

상속 (inheritance)

11

□ 객체 지향 상속

- ▣ 자식이 부모 유전자를 물려 받는 것과 유사한 개념



유산 상속



유전적 상속 : 객체 지향 상속

□ 상속이란?

- 상속(Inheritance)이란 미리 만들어 둔 클래스를 다시 이용하는 방법을 말합니다.
- 상속은 클래스에서만 통용되는 개념이다.
- 자바에서는 extends라는 키워드를 사용한다.

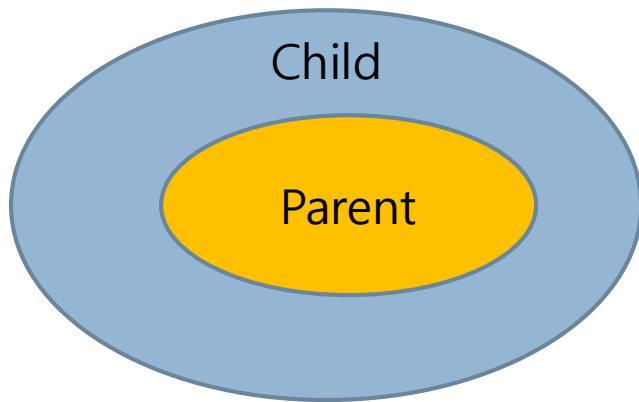
■ 자바의 상속

```
class Child extends Parent{  
    // 멤버 변수와 멤버 메서드 선언  
}
```

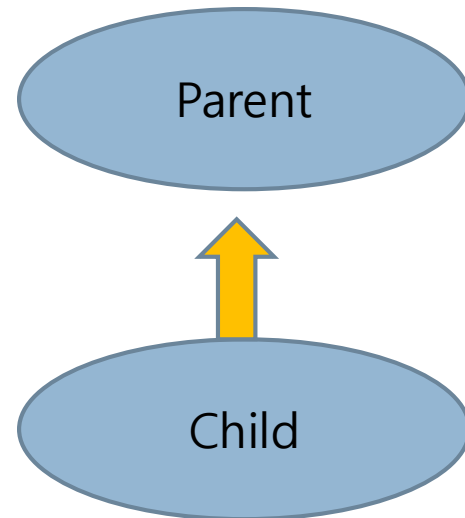
- 조상 클래스 : 부모 클래스, 상위 클래스, 기반 클래스
- 자손 클래스 : 자식 클래스, 하위 클래스, 파생된 클래스

상속

```
class Parent { }  
class Child extends Parent{  
    // 멤버 변수와 멤버 메서드 선언  
}
```



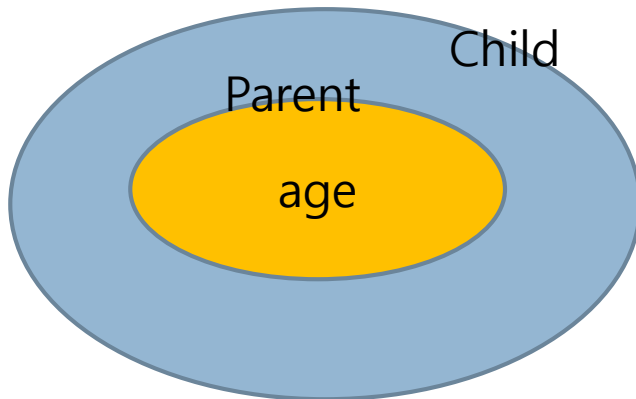
다이어그램



상속
관계도

상속

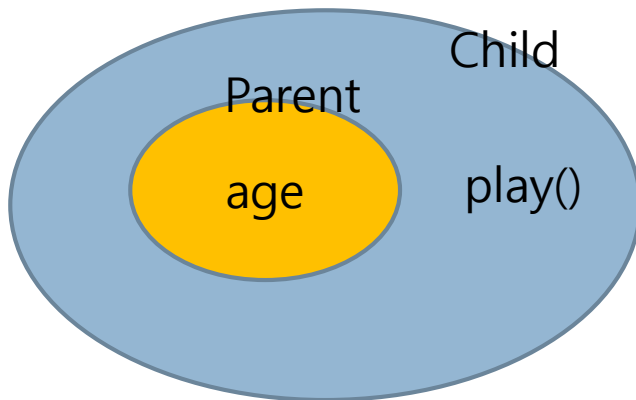
```
class Parent { int age; }  
class Child extends Parent{  
    // 멤버 변수와 멤버 메서드 선언  
}
```



클래스 이름	클래스의 멤버
Parent	age
Child	age

상속

```
class Parent { int age; }  
class Child extends Parent{  
    void play() {}  
}
```



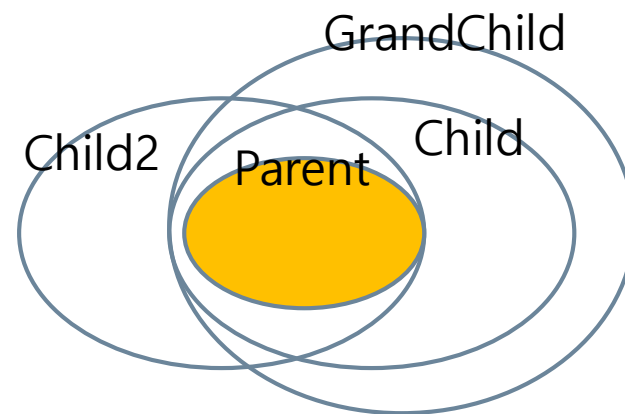
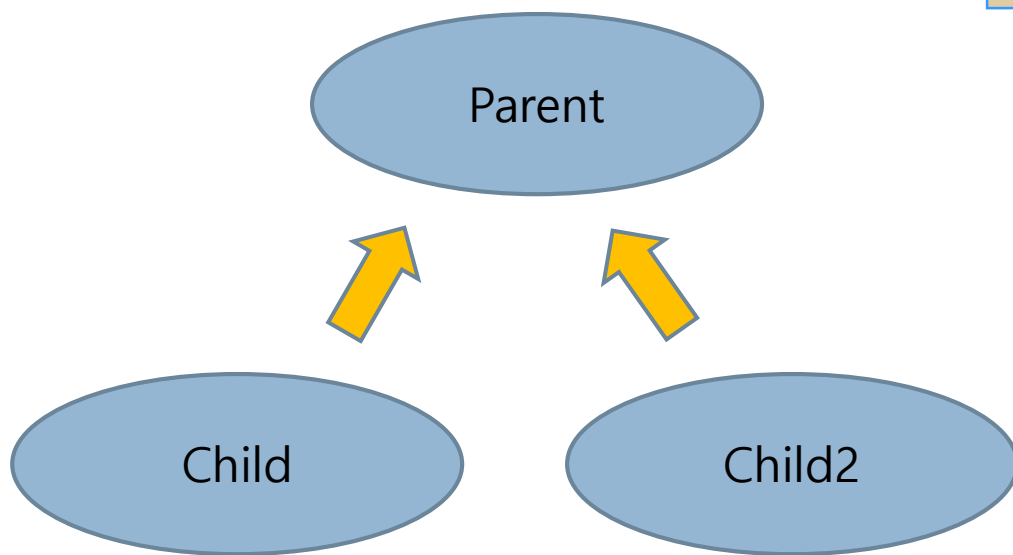
클래스 이름	클래스의 멤버
Parent	age
Child	age, play()

- * 생성자와 초기화 블록은 상속되지 않는다. 멤버만 상속된다.
- * 자손 클래스의 멤버 개수는 조상 클래스보다 항상 같거나 많다.

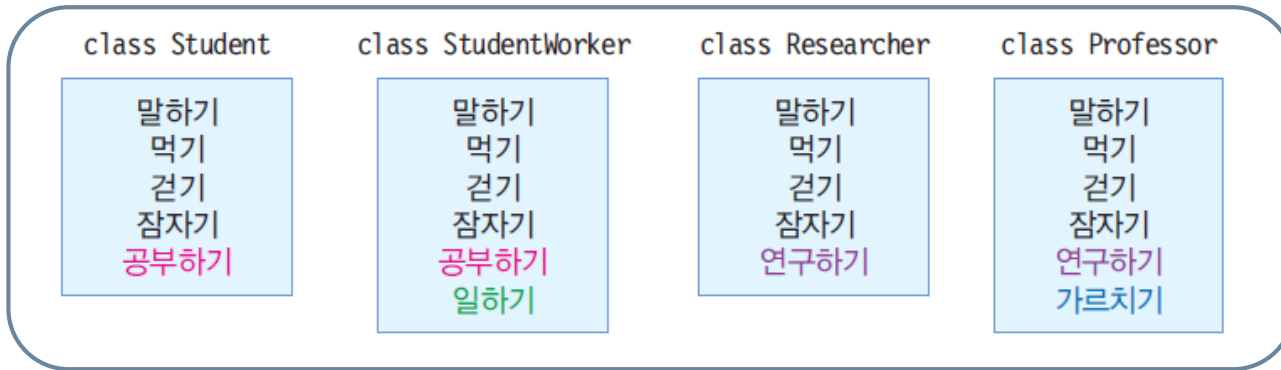
상속

```
class Parent {  
class Child extends Parent{ }  
class Child2 extends Parent{ }
```

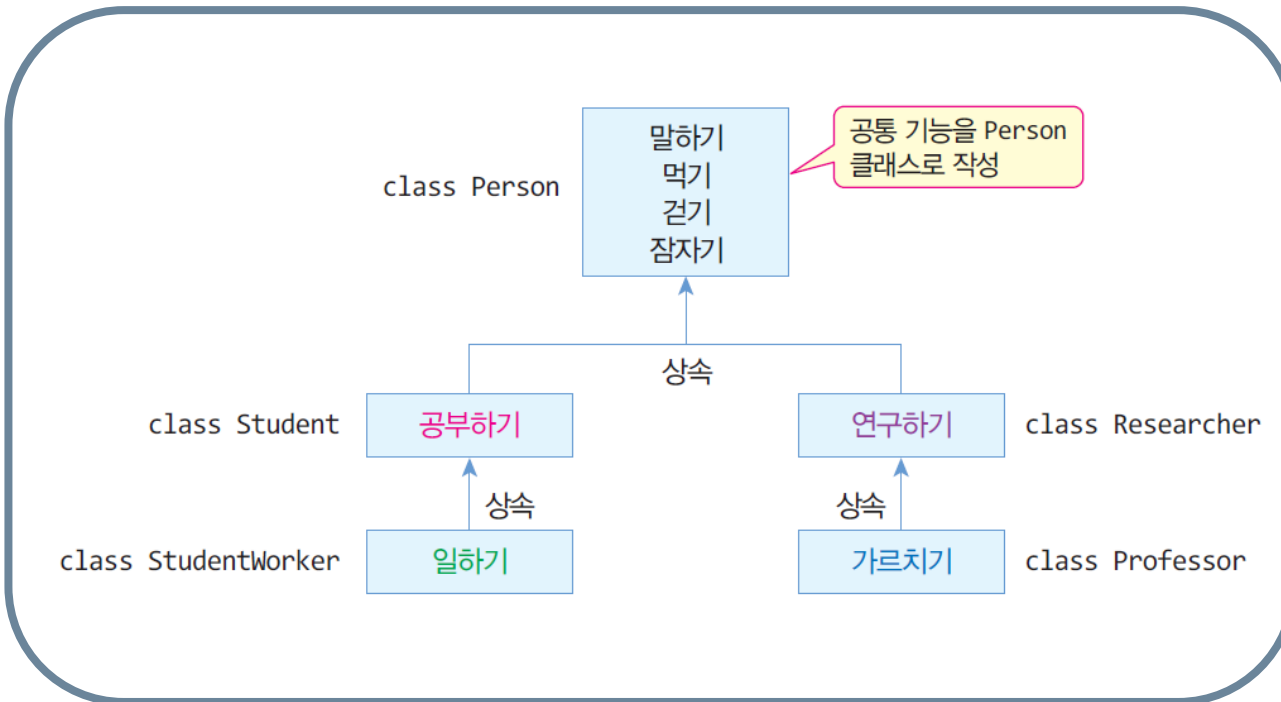
```
class Parent {  
class Child extends Parent{ }  
class Child2 extends Parent{ }  
class GrandChild extends Child{ }
```



상속의 필요성



상속이 없는 경우
중복된 멤버를 가진
4 개의 클래스



상속을 이용한
경우 중복이 제거되고
간결해진 클래스 구조

포함

□ 포함관계

- ▣ 클래스의 멤버변수로 다른 클래스를 선언하는 것

```
class Circle {  
    int x;  
    int y;  
    int r;  
}
```



```
class Circle {  
    Point c = new Point();  
    int r;  
}
```

```
class Car{  
    Engine e = new Engine();  
    Door[] d = new Door[4];  
    .....  
}
```

포함 / 상속 관계

□ 관계

- 상속관계 : ~은 ~이다 (is-a) 관계 성립할 때
- 포함관계 : ~은 ~을 가지고 있다(has-a) 관계 성립할 때

상속

```
class Circle extends Point
{
    int r;
}
```

포함

```
class Circle {
    Point c = new Point();
    int r;
}
```

클래스 상속과 객체

20

□ 상속 선언

▣ extends 키워드로 선언

- 부모 클래스를 물려받아 확장한다는 의미

▣ 부모 클래스 -> 슈퍼 클래스(super class)

▣ 자식 클래스 -> 서브 클래스(sub class)

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

서브 클래스 슈퍼 클래스

- ColorPoint는 Point를 물려 받으므로, **Point에 선언된 필드와 메소드 선언 필요 없음**

예제 5-1 : 클래스 상속

21

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

```
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();

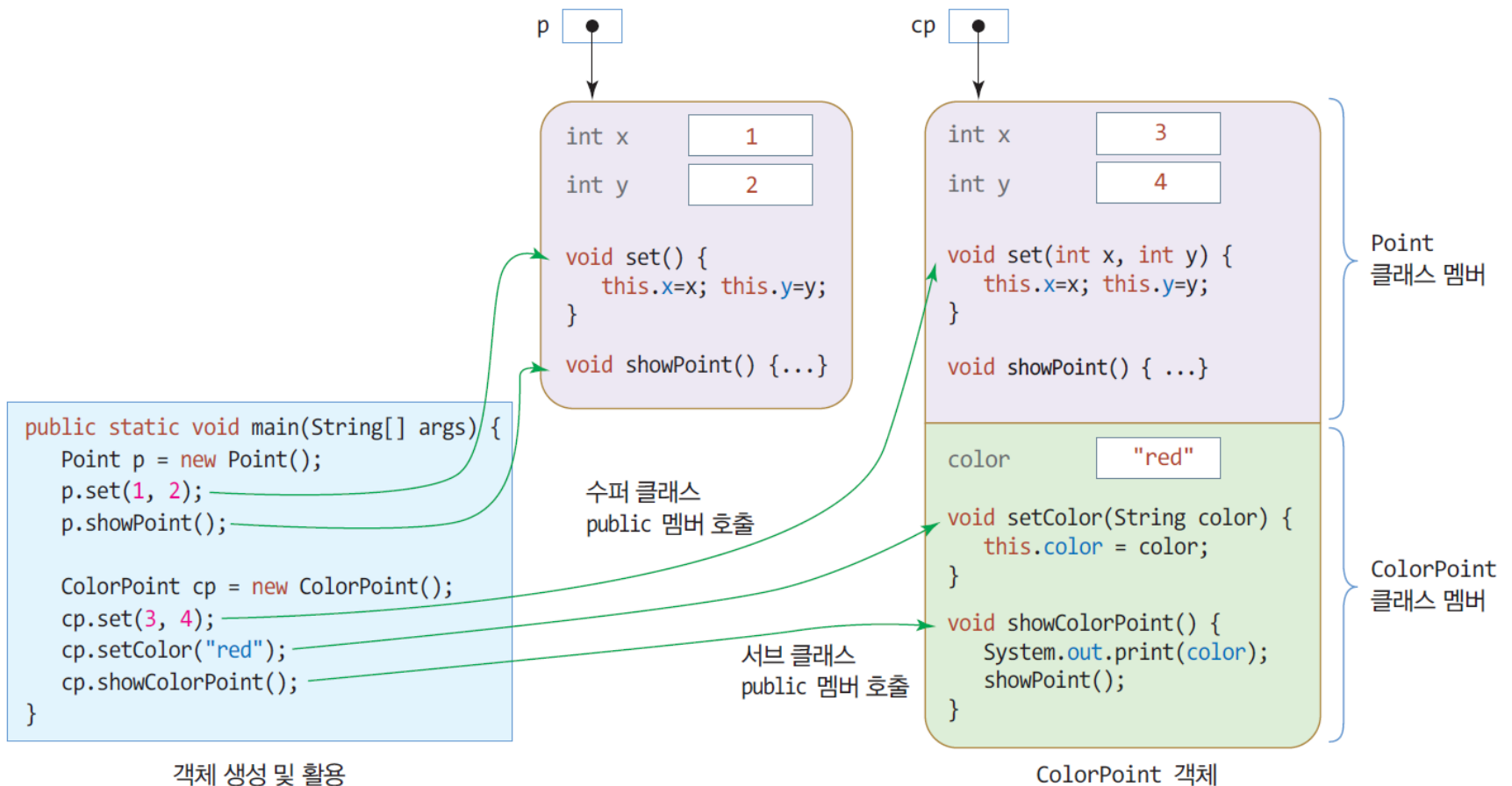
        ColorPoint cp = new ColorPoint();
        cp.set(3, 4);          // Point 클래스의 set() 호출
        cp.setColor("red");    // ColorPoint의 setColor() 호출
        cp.showColorPoint();   // 컬러와 좌표 출력
    }
}
```

(1,2)
red(3,4)

서브 클래스 객체의 모양

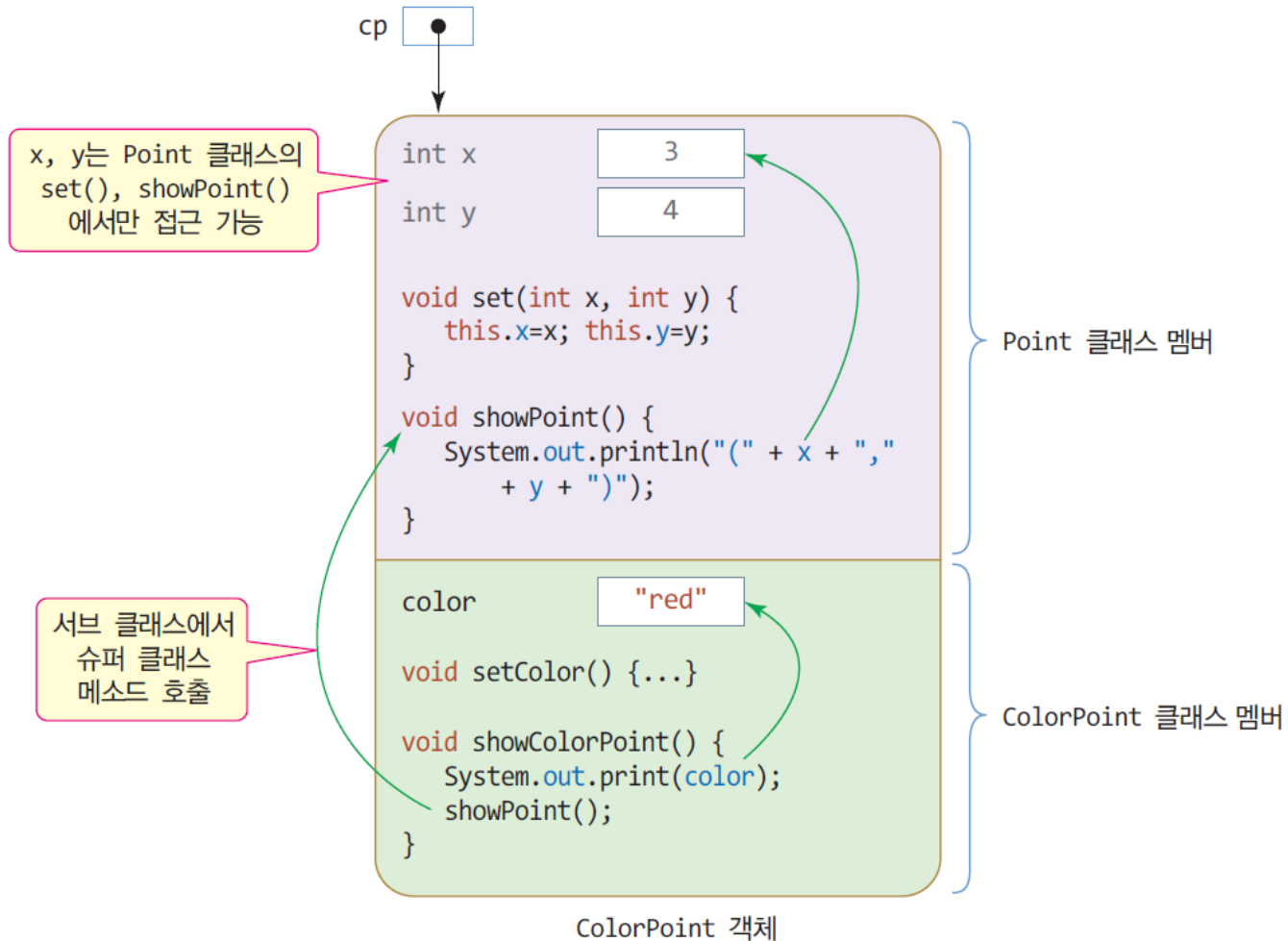
22

- 슈퍼 클래스 객체와 서브 클래스의 객체는 별개
- 서브 클래스 객체는 슈퍼 클래스 멤버 포함



서브 클래스에서 슈퍼 클래스 멤버 접근

23



자바 상속의 특징

24

- 클래스 다중 상속(multiple inheritance) 불허
 - ▣ C++는 다중 상속 가능
 - C++는 다중 상속으로 멤버가 중복 생성되는 문제 있음
 - ▣ 자바는 인터페이스(interface)의 다중 상속 허용
- 모든 자바 클래스는 묵시적으로 Object클래스 상속받음
 - ▣ java.lang.Object는 클래스는 모든 클래스의 슈퍼 클래스

슈퍼 클래스의 멤버에 대한 서브 클래스의 접근

25

- 슈퍼 클래스의 private 멤버
 - ▣ 서브 클래스에서 접근할 수 없음

- 슈퍼 클래스의 디폴트 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 허용
 - ▣ 패키지 여부와 상관없이 서브 클래스는 접근 가능

슈퍼 클래스 멤버의 접근 지정자

26

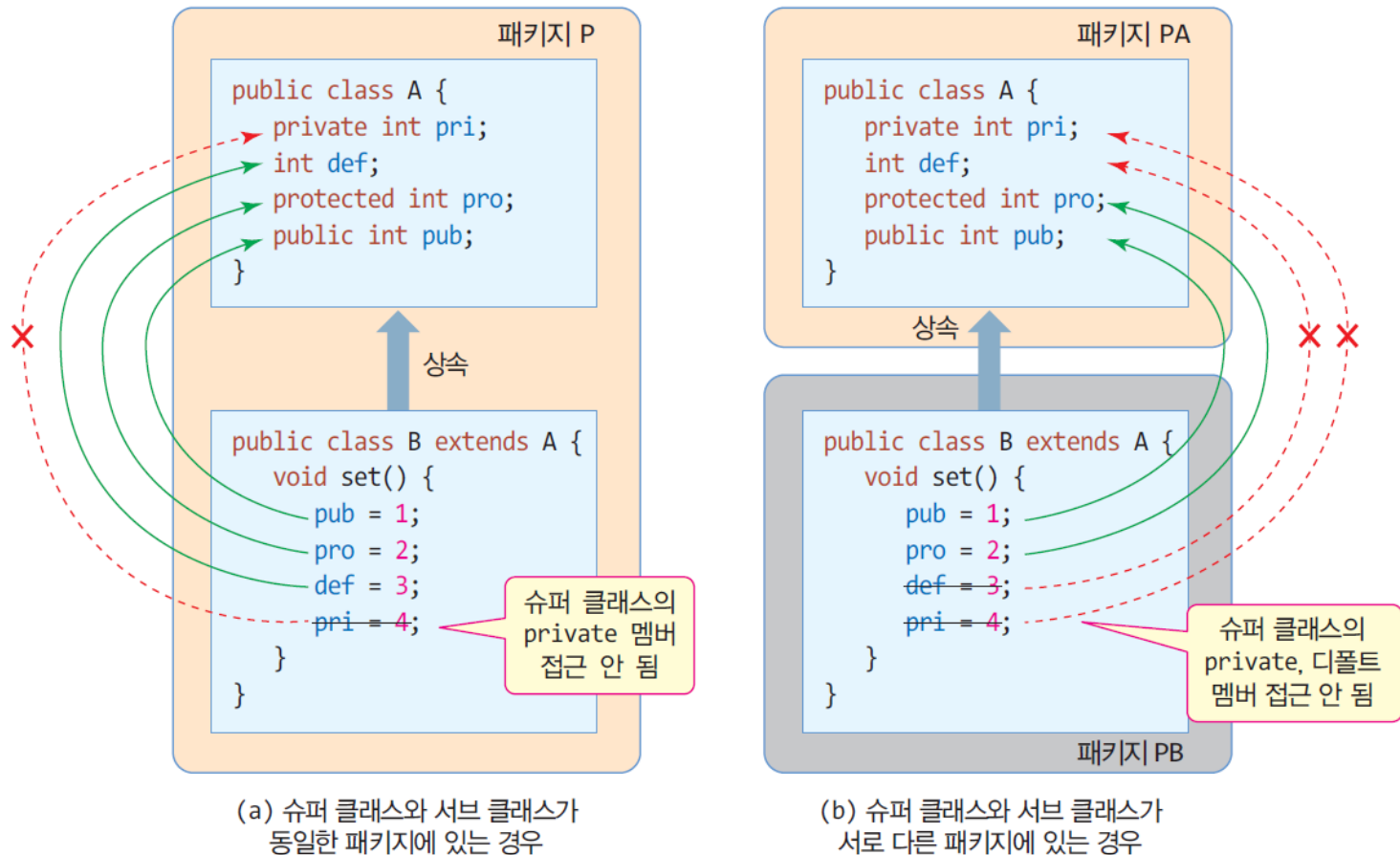
슈퍼 클래스 멤버에 접근하는 클래스 종류	슈퍼 클래스 멤버의 접근 지정자			
	private	디폴트	protected	public
같은 패키지의 클래스	×	○	○	○
다른 패키지의 클래스	×	×	×	○
같은 패키지의 서브 클래스	×	○	○	○
다른 패키지의 서브 클래스	×	×	○	○

(○는 접근 가능함을, ×는 접근 불가능함을 뜻함)

protected 멤버

27

- protected 멤버에 대한 접근
 - 같은 패키지의 모든 클래스에게 허용
 - 상속되는 서브 클래스(같은 패키지든 다른 패키지든 상관 없음)에게 허용



서브 클래스/슈퍼 클래스의 생성자 호출과 실행

28

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

답 둘 다 실행된다. 생성자의 목적은 객체 초기화에 있으므로, 서브 클래스의 생성자는 서브 클래스의 멤버나 필요한 초기화를 수행하고, 슈퍼 클래스의 생성자는 슈퍼 클래스의 멤버나 필요한 초기화를 각각 수행한다.

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

답 슈퍼 클래스의 생성자가 먼저 실행된다.

□ 서브 클래스의 객체가 생성될 때

- 슈퍼클래스 생성자와 서브 클래스 생성자 모두 실행

- 호출 순서

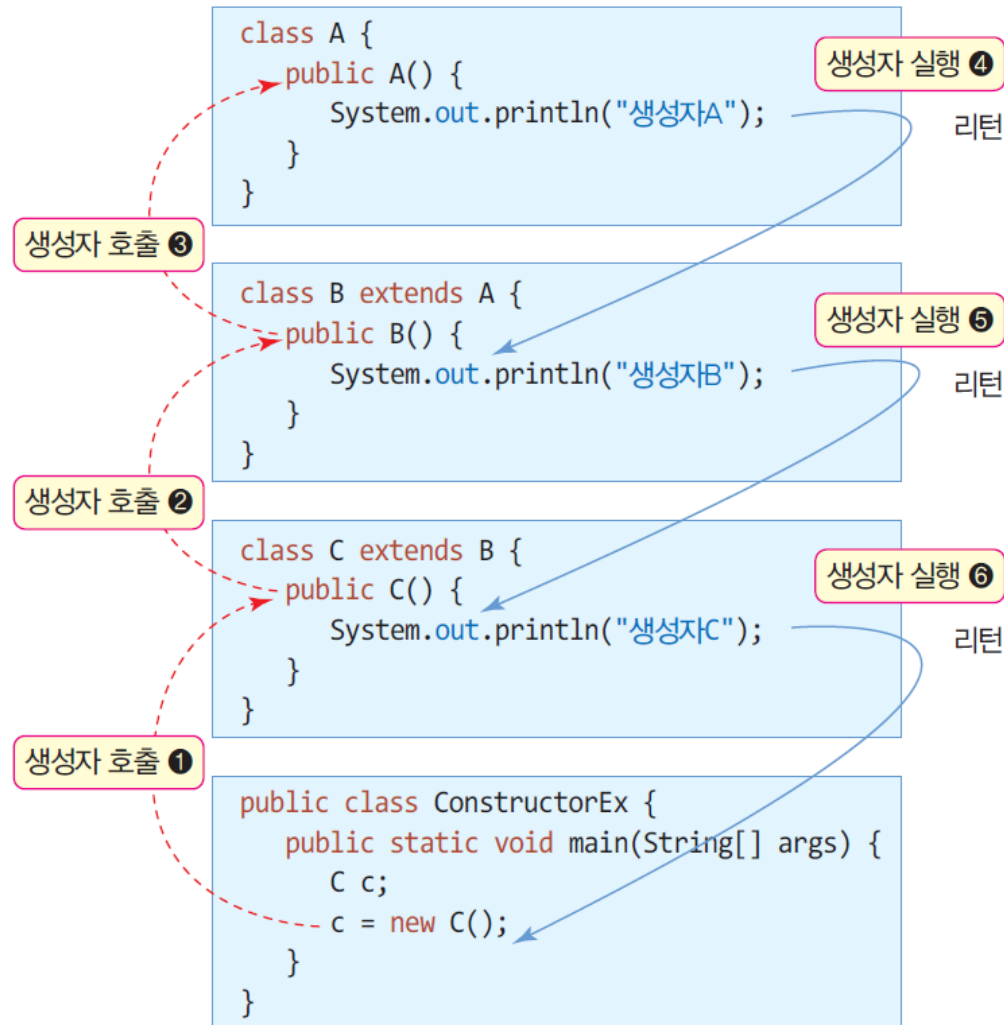
- 서브 클래스의 생성자 먼저 호출,
- 서브 클래스의 생성자는 실행 전 슈퍼 클래스 생성자 호출

- 실행 순서

- 슈퍼 클래스의 생성자가 먼저 실행된 후 서브 클래스의 생성자 실행

슈퍼 클래스와 서브 클래스의 생성자 호출 및 실행 관계

29



생성자A
생성자B
생성자C

서브 클래스와 슈퍼 클래스의 생성자 선택

30

- 슈퍼 클래스와 서브 클래스
 - ▣ 각각 여러 개의 생성자 작성 가능

- 서브 클래스의 객체가 생성될 때
 - ▣ 슈퍼 클래스 생성자 1 개와 서브 클래스 생성자 1개가 실행

- 서브 클래스의 생성자와 슈퍼 클래스의 생성자가 결정되는 방식
 1. 개발자의 명시적 선택
 - 서브 클래스 개발자가 슈퍼 클래스의 생성자 명시적 선택
 - `super()` 키워드를 이용하여 선택
 2. 컴파일러가 기본생성자 선택
 - 서브 클래스 개발자가 슈퍼 클래스의 생성자를 선택하지 않는 경우
 - 컴파일러가 자동으로 슈퍼 클래스의 기본 생성자 선택

컴파일러에 의해 슈퍼 클래스의 기본 생성자가 묵시적 선택(1)

31

개발자가 서브 클래스의 생성자에 대해 슈퍼 클래스의 생성자를 명시적으로 선택하지 않은 경우

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

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

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

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

생성자A
생성자B

슈퍼 클래스에 기본 생성자가 없어 오류 난 경우

32

B()에 대한 짝,
A()를 찾을 수
없음

오류 메시지

"Implicit super constructor A() is undefined. Must explicitly invoke another constructor"

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

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

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


서브 클래스의 매개 변수를 가진 생성자에 대해서도 슈퍼 클래스의 기본 생성자가 자동 선택

33

개발자가 서브 클래스의 생성자에 대해 슈퍼 클래스의 생성자를 명시적으로 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()로 슈퍼 클래스의 생성자 명시적 선택

34

□ super()

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

super()로 슈퍼 클래스의 생성자를 명시적으로 선택한 사례

35

```
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

예제 5-2 : super()를 활용한 ColorPoint 작성

36

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

```
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

업캐스팅 개념

37

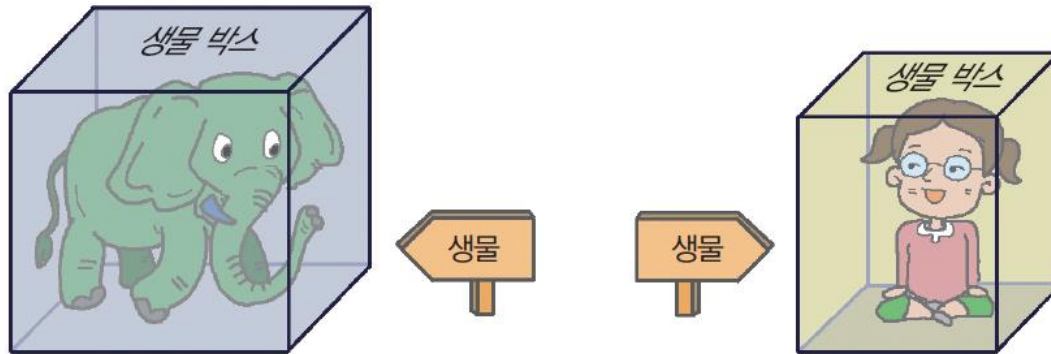
'고양이만 가리킬 수 있는 손가락'으로는 오직 고양이만 가리킬 수 있고, '고래를 가리킬 수 있는 손가락'으로는 고래만 가리킬 수 있다. '사람을 가리킬 수 있는 손가락'으로 고래를 가리킨다면 일종의 오류이다. 그러나 만일 '생물만 가리킬 수 있는 손가락'이 있다고 하자. 이 손가락으로 고양이, 고래, 나무, 사람을 가리키는 것은 자연스럽다. 그 이유는 고양이, 고래, 나무, 사람이 모두 생물을 상속받은 객체이기 때문이며, 이들은 모두 생물적 속성을 가지고 있기 때문이다.



이처럼 업 캐스팅은 기본 클래스의 포인터(생물을 가리키는 손가락)로 파생 클래스의 객체(고양이, 고래, 사람, 나무)를 가리키는 것을 말한다. 그러나 컵은 무생물이므로 '생물을 가리키는 손가락'으로 컵을 가리킬 수는 없다.

업캐스팅

38



생물이 들어가는 박스에
사람이나 코끼리를 넣어도 무방

* 사람이나 코끼리 모두 생물을
상속받았기 때문

□ 업캐스팅(upcasting)

- ▣ 서브 클래스의 레퍼런스를 슈퍼 클래스 레퍼런스에 대입
- ▣ 슈퍼 클래스 레퍼런스로 서브 클래스 객체를 가리키게 되는 현상

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

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

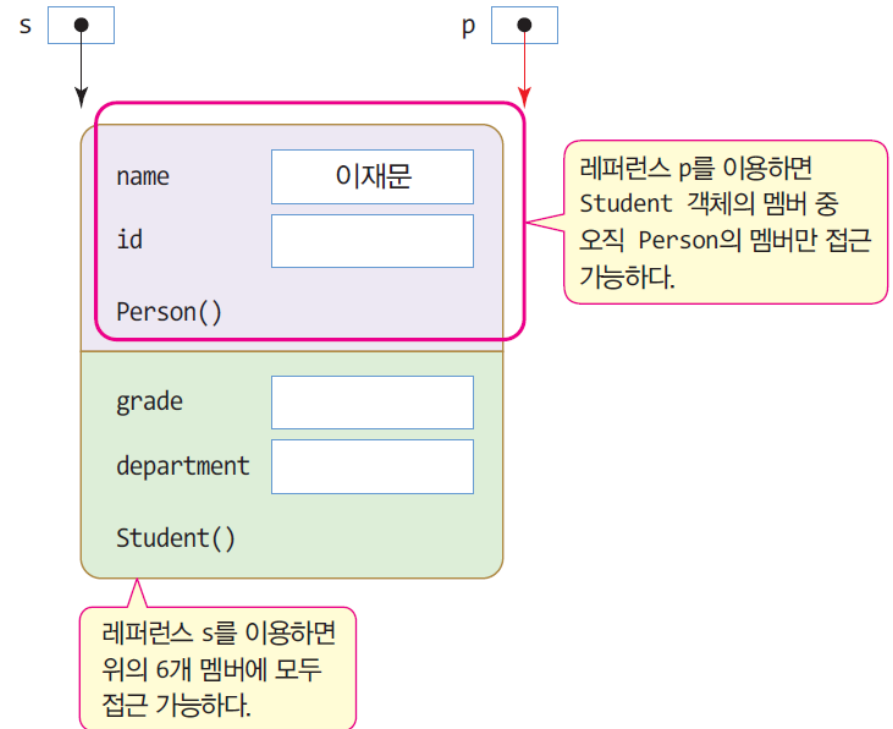
- ▣ 슈퍼클래스 레퍼런스로 객체 내의 슈퍼 클래스의 멤버만 접근 가능

오류

```
p.grade = "A"; // grade는 Person의  
                // 멤버가 아니므로  
                // 컴파일 오류
```

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

다운캐스팅

40

- 다운캐스팅(downcasting)
 - ▣ 슈퍼 클래스 레퍼런스를 서브 클래스 레퍼런스에 대입
 - ▣ 업캐스팅된 것을 다시 원래대로 되돌리는 것
 - ▣ 반드시 명시적 타입 변환 지정

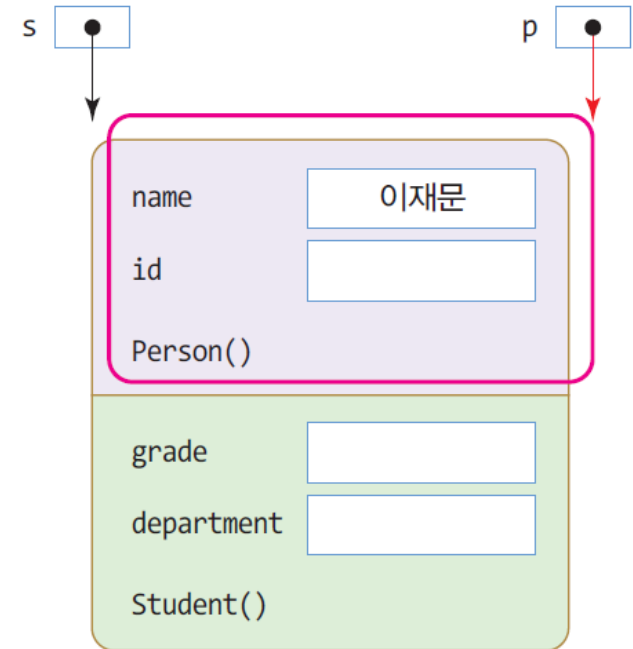
```
class Person { }  
class Student extends Person { }  
  
Person p = new Student("이재문"); // 업캐스팅  
  
Student s = (Student)p; // 다운캐스팅, 강제타입변환
```


다운캐스팅 사례

41

```
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  
    }  
}
```

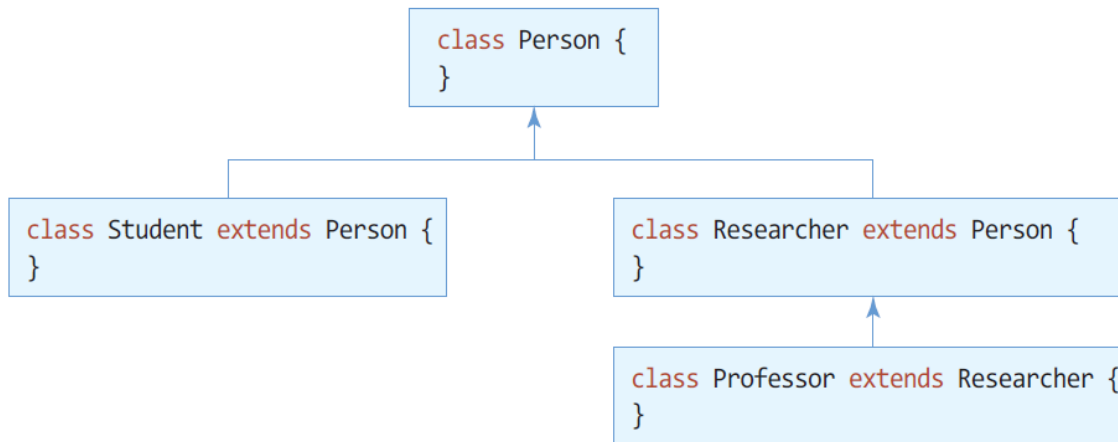
이재문



업캐스팅 레퍼런스로 객체 구별?

42

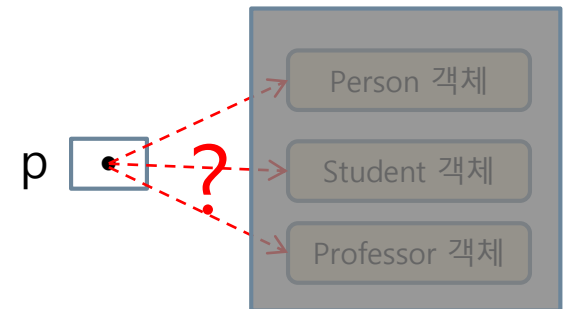
- 업캐스팅된 레퍼런스로는 객체의 실제 타입을 구분하기 어려움
 - ▣ 슈퍼 클래스는 여러 서브 클래스에 상속되기 때문
- ▣ 예) 아래의 클래스 계층 구조에서, p가 가리키는 객체가 Person 객체인지, Student 객체인지, Professor 객체인지 구분하기 어려움



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 연산자 사용

43

□ instanceof 연산자

▣ instanceof 연산자

- 레퍼런스가 가리키는 객체의 타입 식별

객체레퍼런스 **instanceof** 클래스타입

연산의 결과 : true/false의 불린 값

▣ instanceof 연산자 사용 사례

```
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

예제 5-3 : instanceof 연산자 활용

44

instanceof 연산자를 이용하여 [그림 5-15]의 상속 관계에 따라 레퍼런스가 가리키는 객체의 타입을 알아본다. 실행 결과는 무엇인가?

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

public class InstanceOfEx {
    static void print(Person p) {
        if(p instanceof Person)
            System.out.print("Person ");
        if(p instanceof Student)
            System.out.print("Student ");
        if(p instanceof Researcher)
            System.out.print("Researcher ");
        if(p instanceof Professor)
            System.out.print("Professor ");
        System.out.println();
    }

    public static void main(String[] args) {
        System.out.print("new Student() -> ");    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

메소드 오버라이딩의 개념

45

기태네 집에 '원래 기태'와 똑같이 생긴 '기태'가 들어와서, '원래 기태'의 목을 조른 채 '기태'를 부르면 항상 '새로운 기태'가 대답한다. '새로운 기태'가 '원래 기태'를 무력화시키는 관계가 오버라이딩입니다.



- 메소드 오버라이딩(Method Overriding)
 - ▣ 서브 클래스에서 슈퍼 클래스의 메소드 중복 작성
 - ▣ 슈퍼 클래스의 메소드 무력화, 항상 서브 클래스에 오버라이딩한 메소드가 실행되도록 보장됨
 - ▣ "메소드 무시하기"로 번역되기도 함
- 오버라이딩 조건
 - ▣ 슈퍼 클래스 메소드의 원형(메소드 이름, 인자 타입 및 개수, 리턴 타입) 동일하게 작성

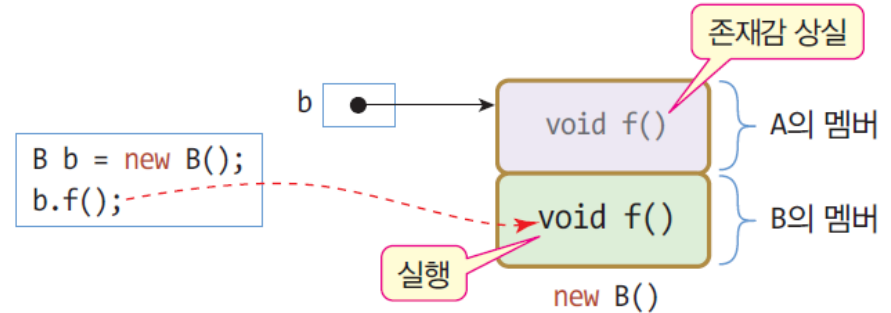
서브 클래스 객체와 오버라이딩된 메소드 호출

- 오버라이딩한 메소드가 실행됨을 보장

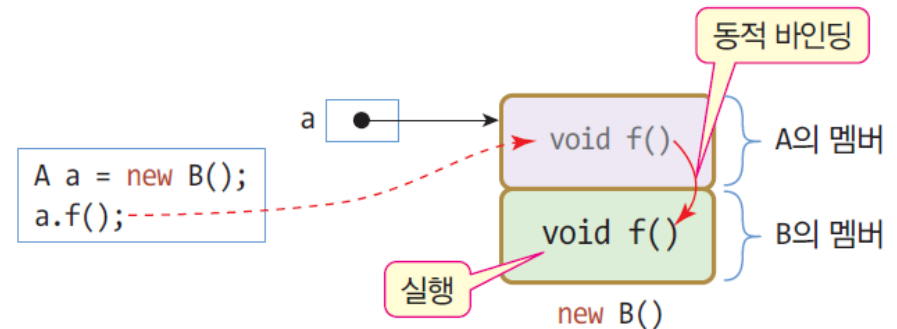
46

```
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



다형성 (Polymorphism)

다형성

- ▣ 여러 가지 형태를 가질 수 있는 능력을 의미
- ▣ 자바에서는 한 타입의 참조변수로 여러 타입의 객체를 참조
 - 조상클래스 타입의 참조변수로 자손클래스의 인스턴스 참조
 - 자손타입의 참조변수는 조상클래스의 인스턴스를 참조할 수 없음

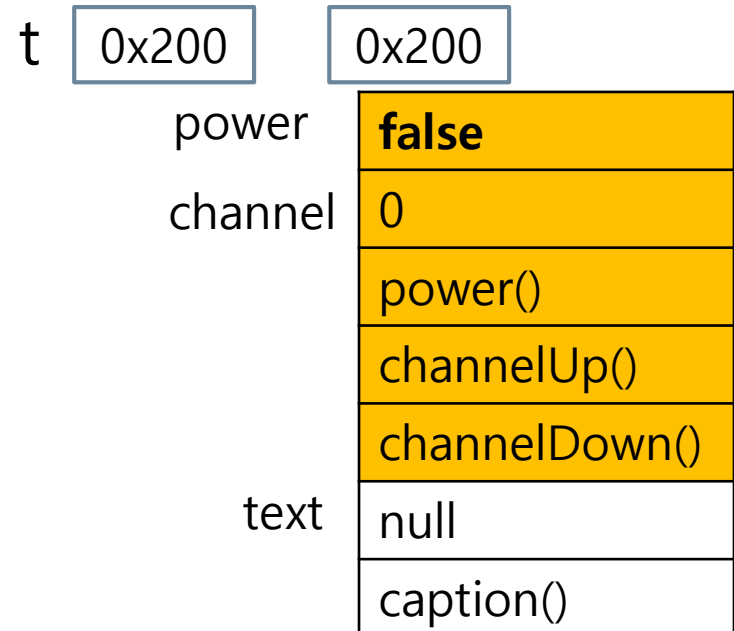
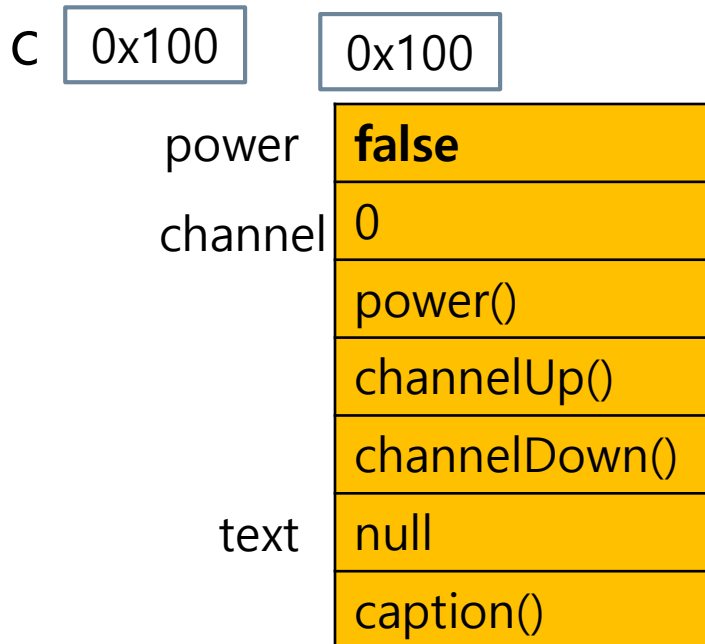
```
class TV{  
    Boolean power;  
    Int channel;  
    Void power(){ power!=power;}  
    Void channelUp(){++channel;}  
    Void channelDown(){--channel;}  
}
```

```
class CaptionTV extends TV{  
    String text;  
    Void caption() { //....  
    }  
}
```

```
TV t = new TV();  
CaptionTV c = new CaptionTV();  
  
TV t = new CaptionTV();  
CaptionTV c = new TV(); //오류
```

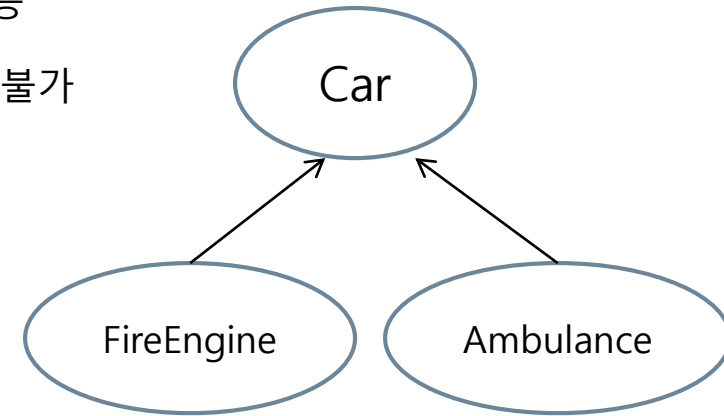

다형성

```
CaptionTV c = new CaptionTV();  
TV t = new CaptionTV()
```



참조변수의 형변환

- 서로 상속관계에 있는 클래스 사이의 형변환
 - ▣ 자손타입 -> 조상타입 (up-casting) : 형변환 생략 가능
 - ▣ 자손타입 <- 조상타입 (down-casting) : 형변환 생략 불가

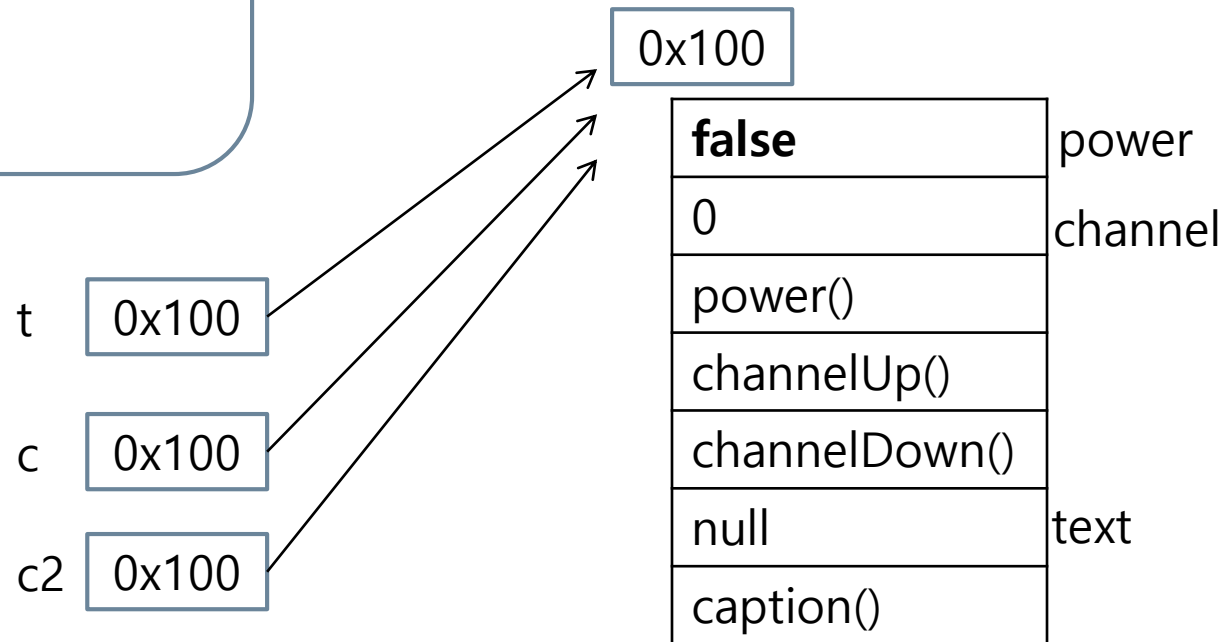


```
FireEngine f;  
Ambulance a;  
a = (Ambulance) f; //컴파일 에러  
f = (FireEngine) a; //컴파일 에러
```

```
Car car = null;  
FireEngine fe = new FireEngine();  
FireEngine fe2 = null;  
  
car = fe; // car = (Car)fe; 형변환 생략  
fe2 = (FireEngine)car;
```

참조변수의 형변환

```
TV t = null;  
CaptionTV c = new CaptionTV();  
CaptionTV c2 = null;  
t = c;  
c2 = (CaptionTV)t;
```



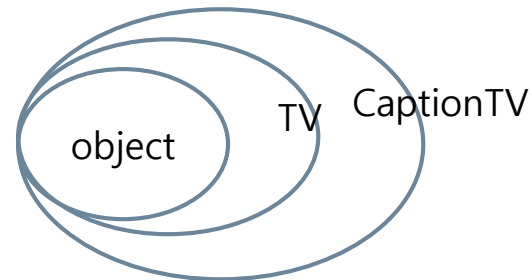
instanceof 연산자

□ instanceof 연산자

- 참조변수가 참조하고 있는 인스턴스의 실제 타입을 알아보기 위한 연산자
- 참조변수 instance of 클래스명 → boolean 값 반환

```
TV t = new TV();  
TV t2 = null;  
CaptionTV c = null;  
c = (CaptionTV) t; // 오류발생  
t2 = c;
```

```
if(t instanceof CaptionTV) {  
    c = (CaptionTV)t;  
}
```



매개변수의 다형성

```
class Product {  
    int price;  
    int bonusPoint;  
}  
  
class TV extends Product{}  
class Computer extends Product{}  
class Audio extends Product{}  
  
class Buyer{  
    int money = 100;  
    int bonusPoint = 0;  
}
```

```
void buy(TV t){  
    money = money - t.price;  
    bonusPoint = bonusPoint + t.bonusPoint;  
}
```

```
void buy(Computer t){  
    money = money - t.price;  
    bonusPoint = bonusPoint + t.bonusPoint;  
}
```

```
void buy(Audio t){  
    money = money - t.price;  
    bonusPoint = bonusPoint + t.bonusPoint;  
}
```

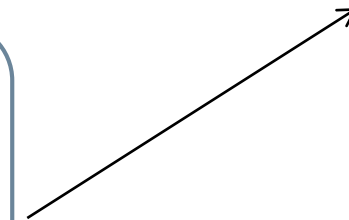
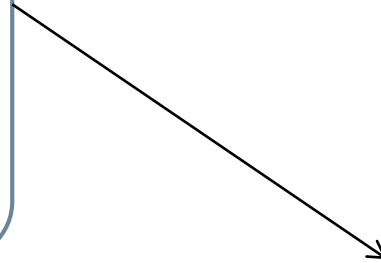
매개변수의 다형성

```
void buy(TV t){  
    money = money - t.price;  
    bonusPoint = bonusPoint + t.bonusPoint;  
}
```

```
void buy(Computer t){  
    money = money - t.price;  
    bonusPoint = bonusPoint + t.bonusPoint;  
}
```

```
void buy(Audio t){  
    money = money - t.price;  
    bonusPoint = bonusPoint + t.bonusPoint;  
}
```

```
void buy(Product t){  
    money = money - t.price;  
    bonusPoint = bonusPoint + t.bonusPoint;  
}
```



오버라이딩의 목적, 다형성 실현

55

- 오버라이딩으로 다형성 실현
 - 하나의 인터페이스(같은 이름)에 서로 다른 구현
 - 슈퍼 클래스의 메소드를 서브 클래스에서 각각 목적에 맞게 다르게 구현
 - 사례
 - Shape의 draw() 메소드를 Line, Rect, Circle에서 오버라이딩하여 다르게 구현

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

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

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

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

오버라이딩

예제 5-4 : 메소드 오버라이딩으로 다형성 실현

56

Shape의 draw() 메소드를 Line, Circle, Rect 클래스에서 목적에 맞게 오버라이딩하는 다형성의 사례를 보여준다.

```
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
    }

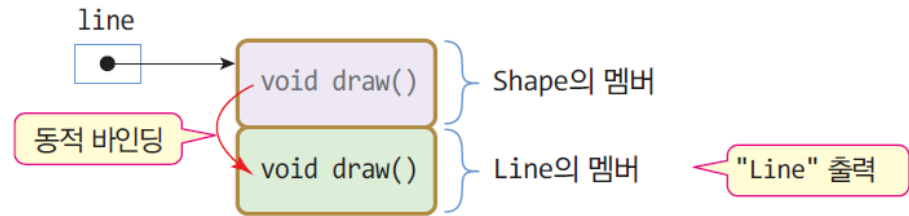
    public static void main(String[] args) {
        Line line = new Line();
        paint(line); // Line의 draw() 실행. "Line" 출력

        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

예제 5-4 실행 과정

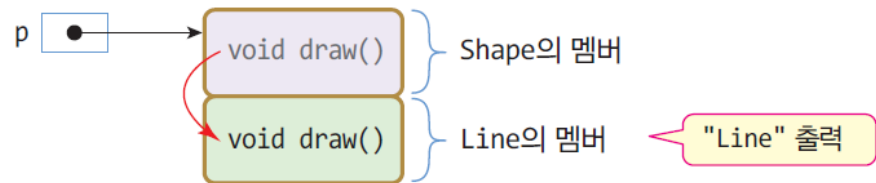
```
Line line = new Line();  
paint(line);
```



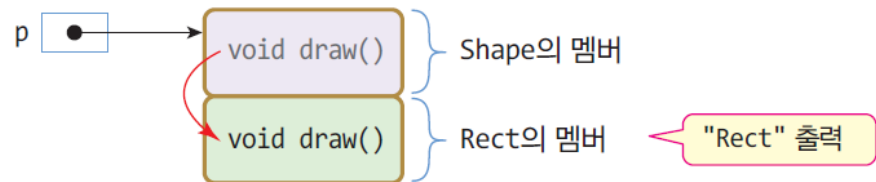
```
paint(new Shape());
```



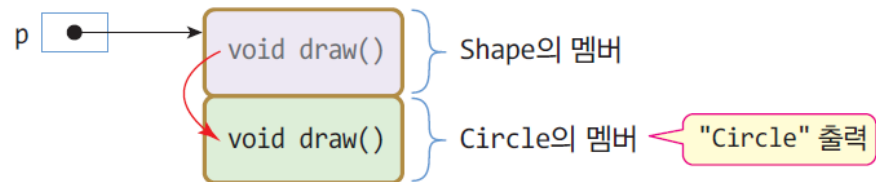
```
paint(new Line());
```



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



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



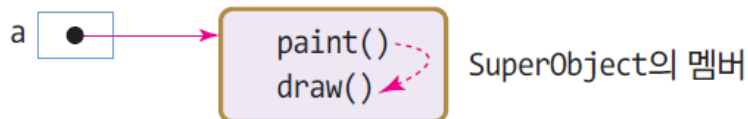
동적 바인딩 - 오버라이딩된 메소드 호출

58

* 오버라이딩 메소드가 항상 호출된다.

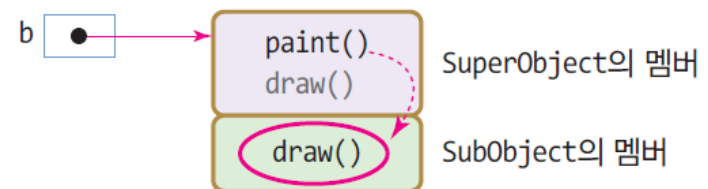
```
public class SuperObject {  
    protected String name;  
    public void paint() {  
        draw();  
    }  
    public void draw() {  
        System.out.println("Super Object");  
    }  
    public static void main(String [] args) {  
        SuperObject a = new SuperObject();  
        a.paint();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

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



super 키워드로 슈퍼 클래스의 멤버 접근

59

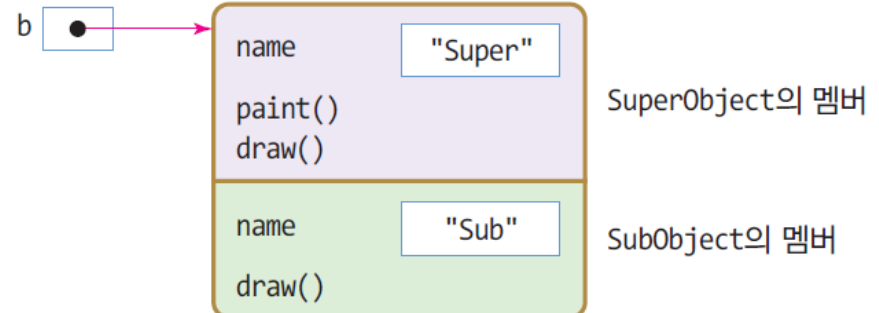
동적 바인딩

```
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();  
    }  
}
```

정적 바인딩

■ super

- 슈퍼 클래스의 멤버를 접근할 때 사용되는 레퍼런스
super.슈퍼클래스의멤버
- 서브 클래스에서만 사용
- 슈퍼 클래스의 필드 접근
- 슈퍼 클래스의 메소드 호출 시
- super로 이루어지는 메소드 호출 : 정적 바인딩



Super
Sub

오버로딩과 오버라이딩

60

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

추상 클래스

61

□ 추상 메소드(abstract method)

- abstract로 선언된 메소드, 메소드의 코드는 없고 원형만 선언

```
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 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  
}
```

추상 클래스의 인스턴스 생성 불가

62

- 추상 클래스는 온전한 클래스가 아니기 때문에 인스턴스를 생성할 수 없음

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

오류

컴파일 오류 메시지

Unresolved compilation problem: Cannot instantiate the type Shape

추상 클래스의 상속과 구현

63

▣ 추상 클래스 상속

- 추상 클래스를 상속받으면 추상 클래스가 됨
- 서브 클래스도 abstract로 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"); }
}
```

오류

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

▣ 추상 클래스 구현

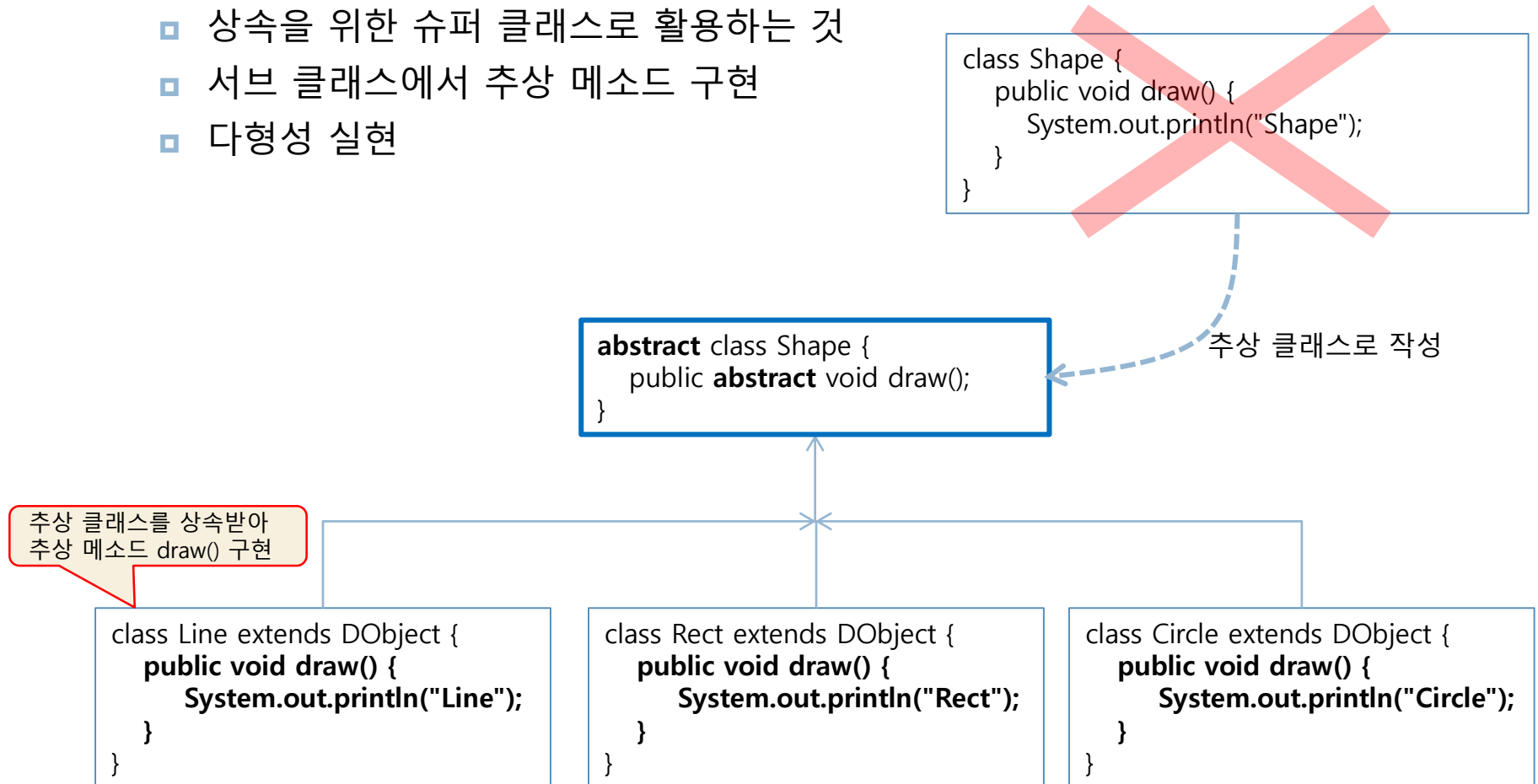
- 서브 클래스에서 슈퍼 클래스의 추상 메소드 구현(오버라이딩)
- 추상 클래스를 구현한 서브 클래스는 추상 클래스 아님

```
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 목적

64

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



예제 5-5 : 추상 클래스의 구현

65

추상 클래스 Calculator를 상속받는 GoodCalc 클래스를 구현하라.

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

예제 5-5 정답

66

```
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
-1
3.0

인터페이스의 필요성

67



A사 제품



B사 제품



C사 제품



D사 제품

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

정해진 규격(인터페이스)에 맞지
않으면 연결 불가

자바 인터페이스

68

□ 자바 인터페이스

- 상수와 추상 메소드로만 구성 : 변수 필드 없음
- 인터페이스 선언
 - interface 키워드로 선언

```
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

□ 자바 인터페이스의 특징

- 상수와 추상 메소드로만 구성
 - 메소드 : public abstract 타입으로 생략 가능
 - 상수 : public static final 타입으로 생략 가능
- 인터페이스의 객체 생성 불가

오류

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

인터페이스 상속

69

- 인터페이스 간에 상속 가능
 - ▣ 인터페이스를 상속하여 확장된 인터페이스 작성 가능
 - ▣ extends 키워드로 상속 선언

■ 예)

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

- 인터페이스 다중 상속 허용

■ 예)

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

인터페이스 구현

70

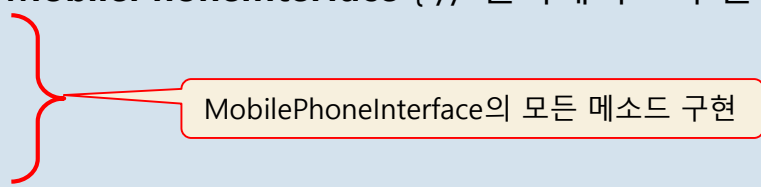
□ 인터페이스 구현

- 인터페이스를 상속받아, 모든 추상 메소드를 구현한 클래스 선언
- implements 키워드로 인터페이스 구현

□ 예)

```
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() { ... }
}
```



- 여러 개의 인터페이스 동시 구현도 가능
- 클래스 상속과 인터페이스 구현을 동시에 할 수 있음
 - 예제 5-6

예제 5-6 : 인터페이스 구현과 동시에 슈퍼 클래스 상속

71

```
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