

Inheritance

Java Programming

Byeongjoon Noh

powernoh@sch.ac.kr



Contents

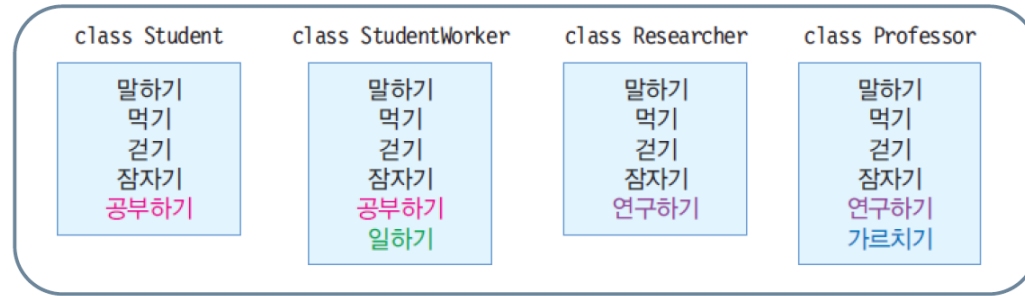
1. Concept of class inheritance
2. Casting
3. Method overriding
4. Abstraction
5. Interface

1. Concept of class inheritance

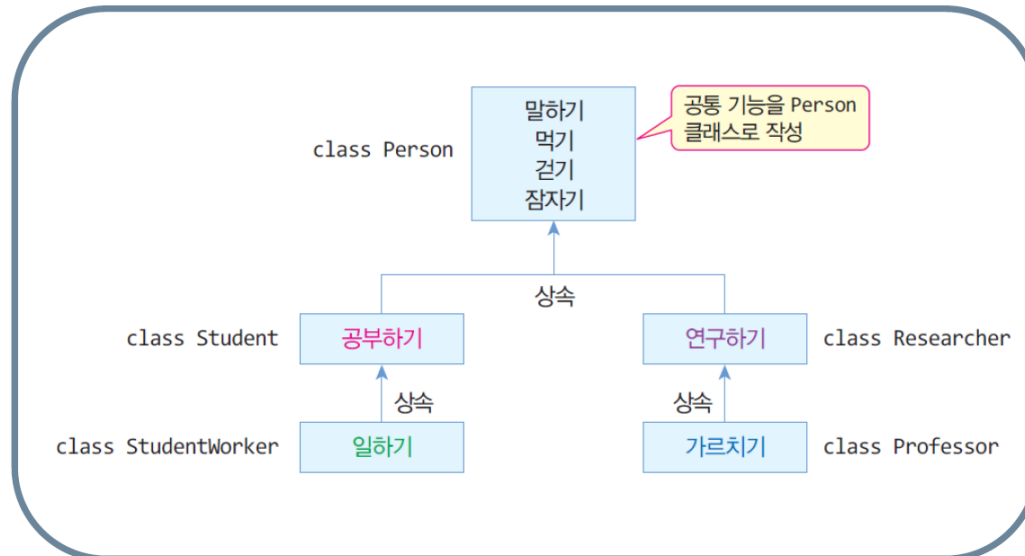
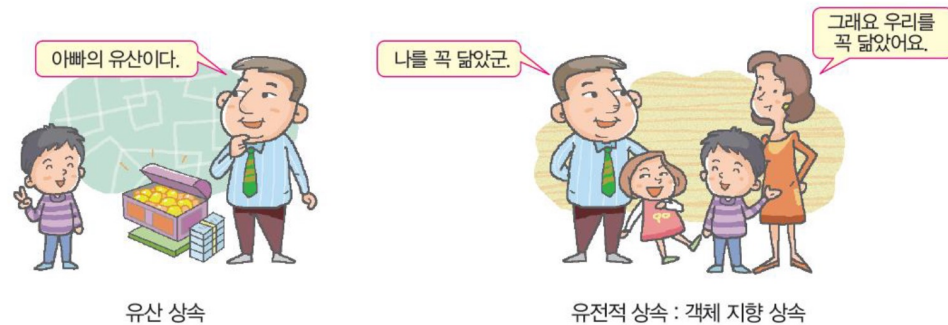
Inheritance

- Definition

- a fundamental concept in OOP that allows a class to inherit properties (fields and methods) from another class



상속이 없는 경우
중복된 멤버를 가진
4 개의 클래스



상속을 이용한
경우 중복이 제거되고
간결해진 클래스 구조

Inheritance

- Characteristics
 - reuse of code
 - allows reuse of existing code without having to rewrite the code in the new class
 - polymorphism
 - enables polymorphic behavior where a subclass can be treated as an instance of its superclass
 - hierarchy
 - establishes a hierarchical classification of classes from more general to more specific

Class inheritance

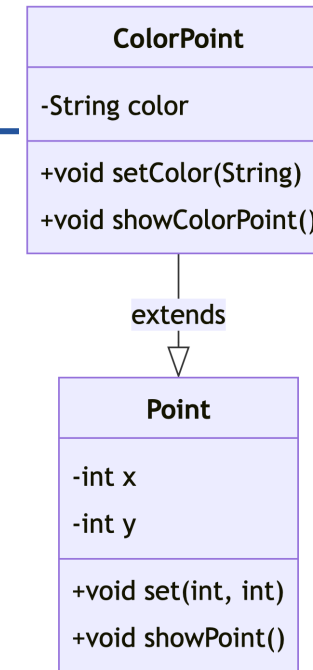
- Inheritance declaration
 - use 'extends' keyword
 - 부모 클래스를 물려 받아 확장한다는 의미
 - 부모 클래스 → 슈퍼 클래스 (super class)
 - 자식 클래스 → 서브 클래스 (subclass)
- Point class: super class
- ColorPoint class: subclass

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

서브 클래스 슈퍼 클래스

Class inheritance

- Example of class inheritance
 - three .java files: Point, ColorPoint, ColorPointEx
 - Point: (x, y)의 한 점을 표현하는 클래스
 - ColorPoint: Point를 상속받아 점에 색을 추가한 클래스



```
public class Point {
    private int x, y;

    void set(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void showPoint() {
        System.out.println("(" + x + ", " + y + ")");
    }
}
```

```
public class ColorPoint extends Point {
    private String color;

    void setColor(String color) {
        this.color = color;
    }

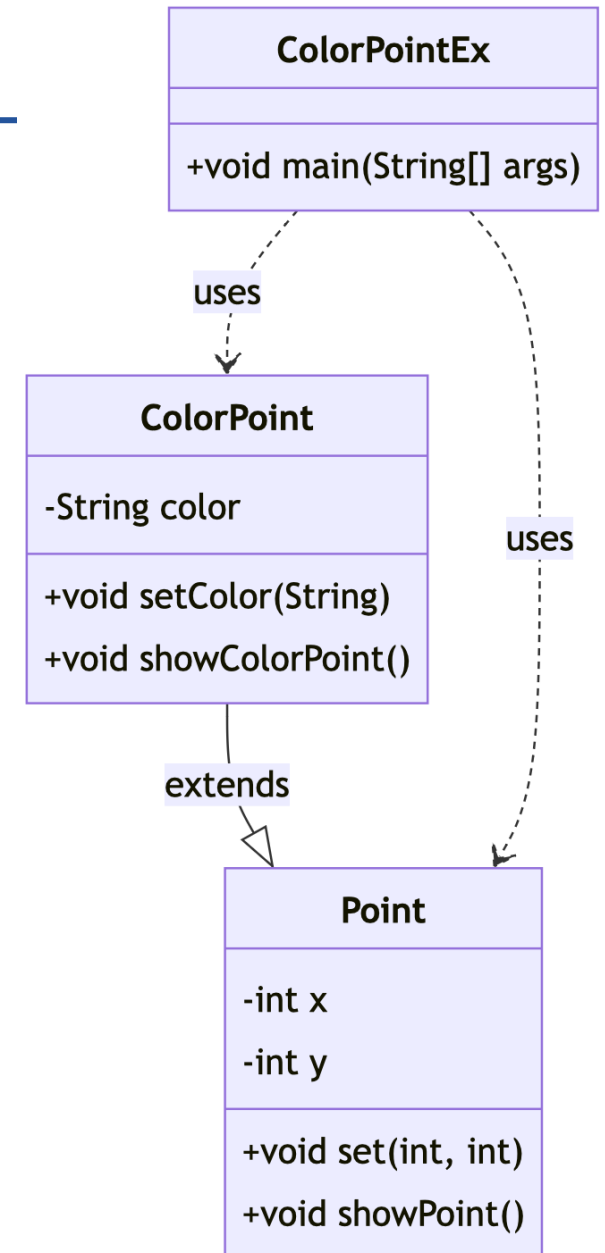
    void showColorPoint() {
        System.out.println(color);
        showPoint();
    }
}
```

Class inheritance

- Example of class inheritance
 - three .java files: Point, ColorPoint, ColorPointEx
 - ColorPointEx: Point, ColorPoint 클래스를 테스트하는 클래스

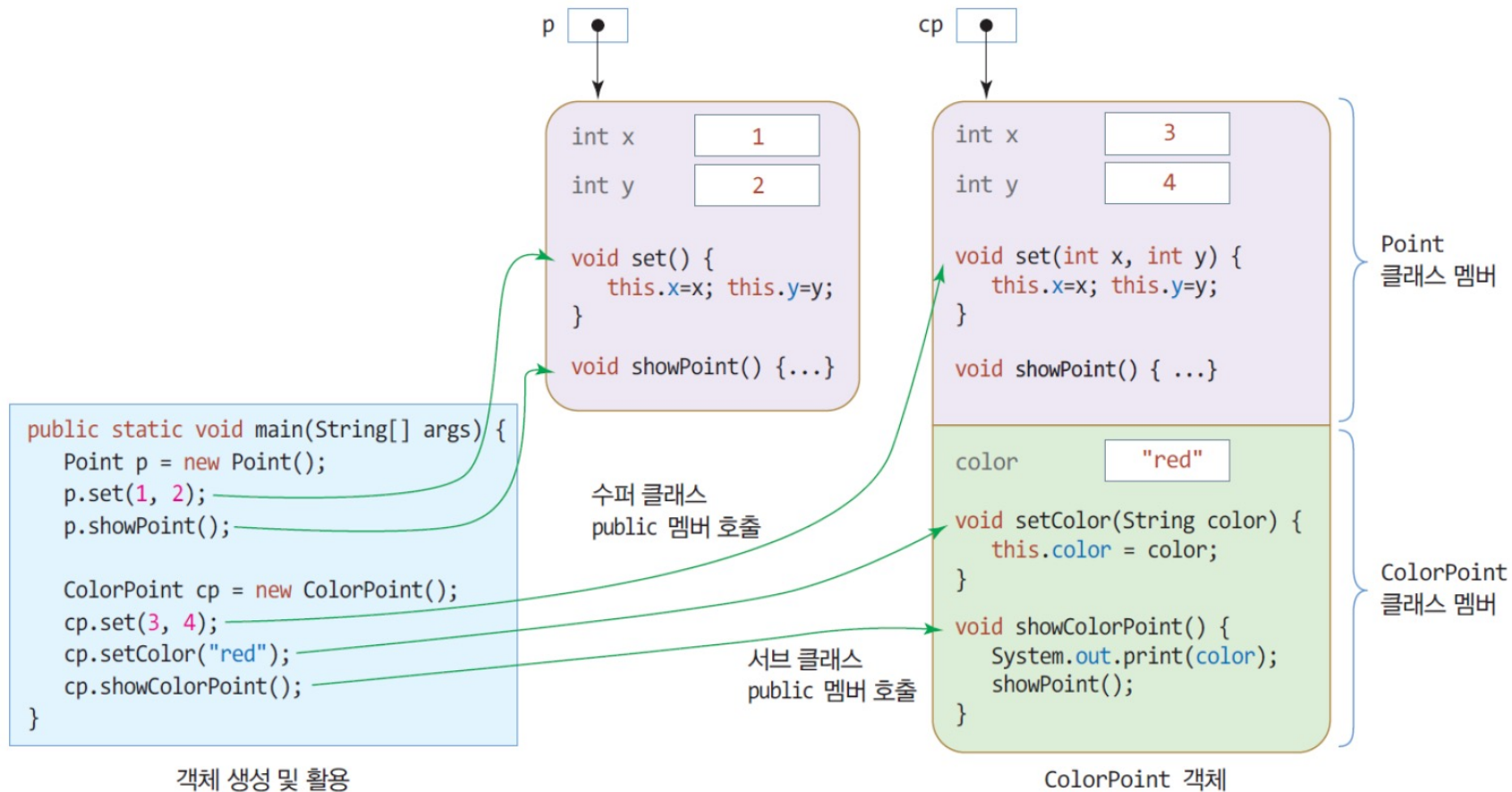
```
public class ColorPointEx {  
  
    public static void main(String[] args) {  
        Point p = new Point();  
        p.set(1, 2);  
        p.showPoint();  
  
        ColorPoint cp = new ColorPoint();  
        cp.set(3, 4);  
        cp.setColor("red");  
        cp.showColorPoint();  
    }  
}
```

```
(1, 2)  
red  
(3, 4)
```



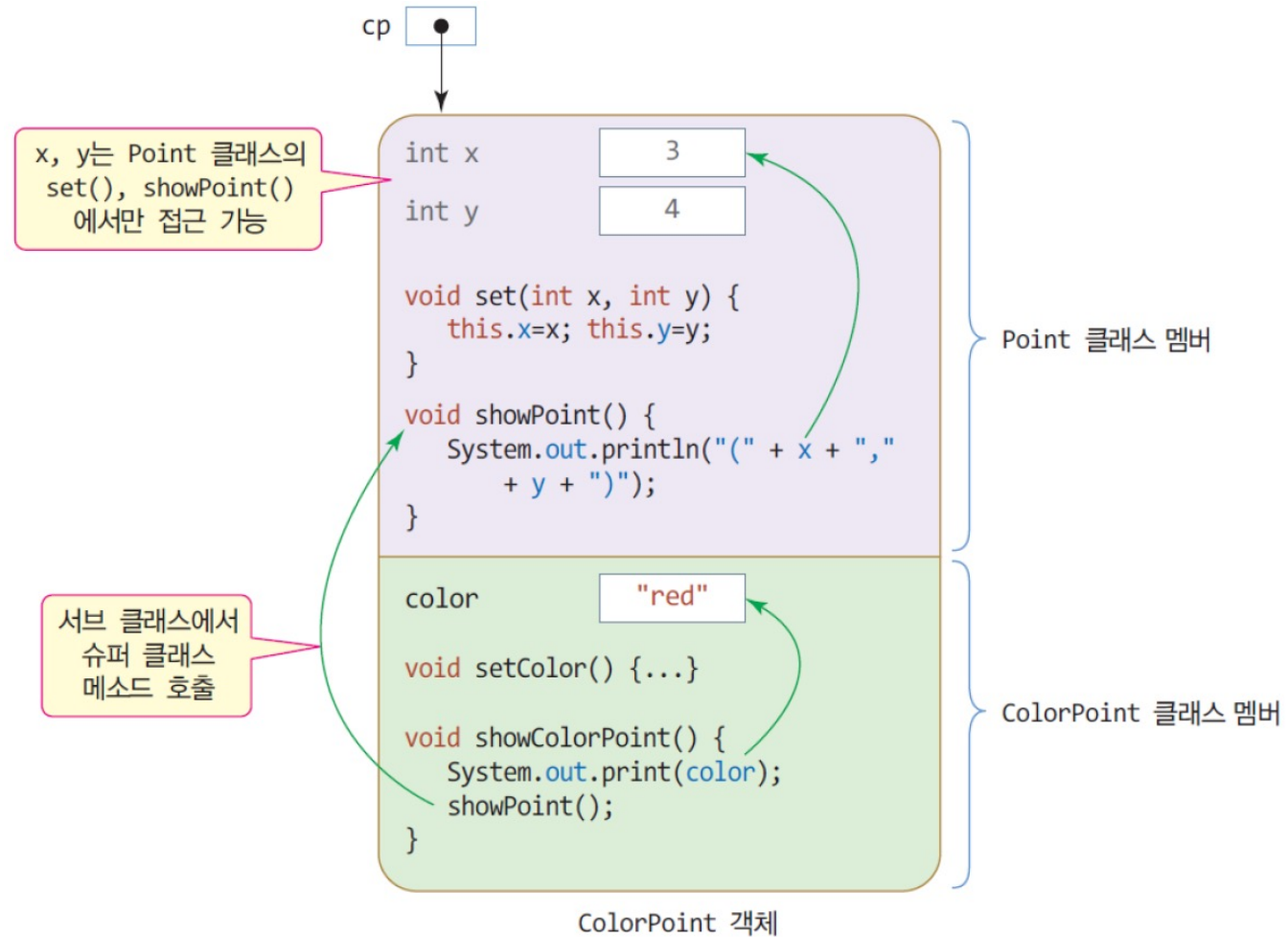
Class inheritance

- Example of class inheritance
 - 슈퍼 클래스 객체와 서브 클래스 객체는 별개의 개체임
 - 서브 클래스 객체는 슈퍼 클래스 멤버를 포함함



Class inheritance

- 서브 클래스에서 슈퍼 클래스 멤버 접근

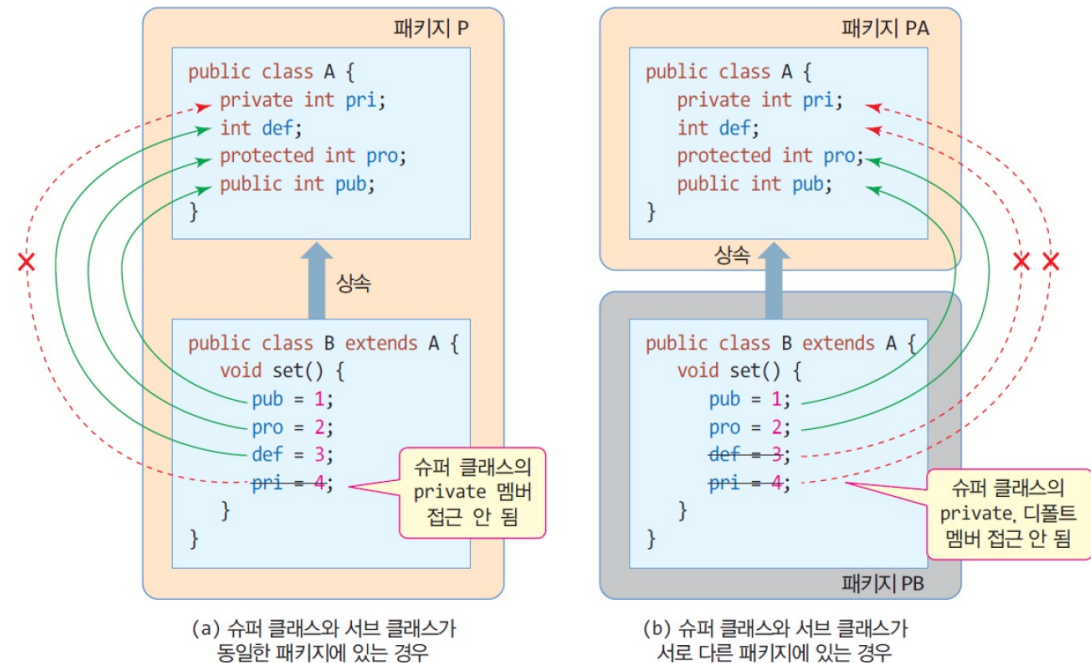


Access modifier in class inheritance

- Superclass의 멤버 접근
 - Superclass의 private member
 - subclass에서 접근 불가
 - Superclass의 default member
 - subclass가 동일한 패키지에 있을 때 접근 가능
 - Superclass의 public member
 - subclass는 항상 접근 가능
 - Superclass의 protected member
 - 같은 패키지 내의 모든 클래스에서 접근 가능
 - 동일 패키지 여부와 상관 없이 subclass는 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)



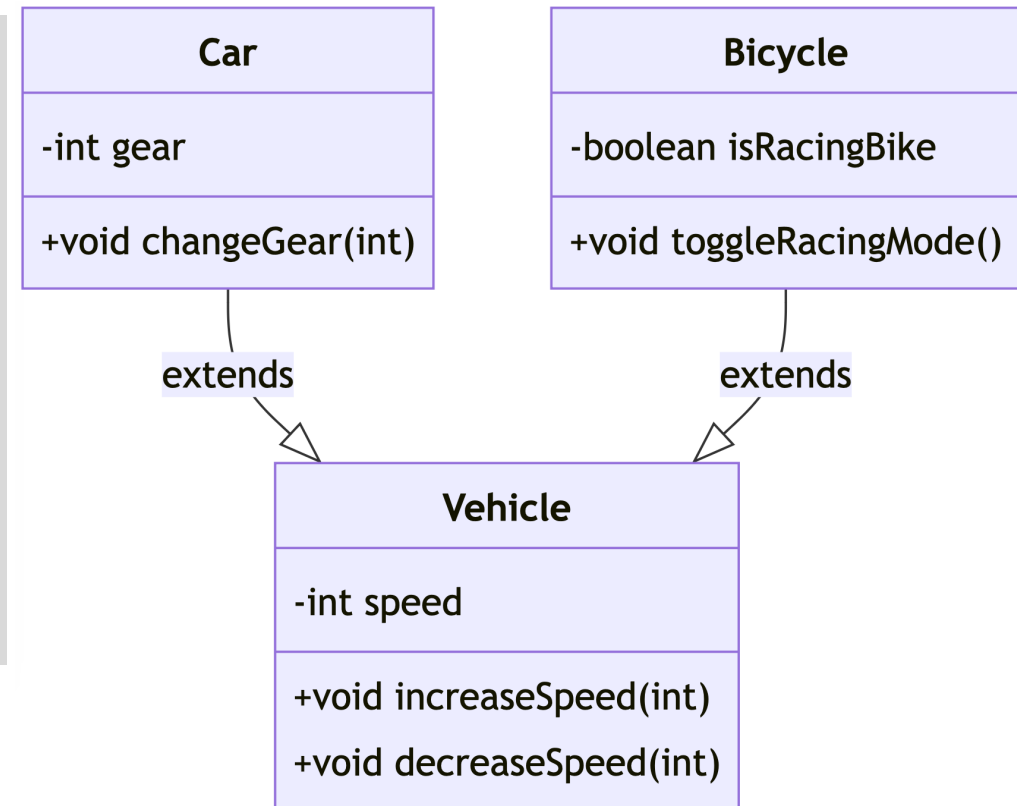
Class inheritance

- One more example of class inheritance: Vehicle, Car, Bicycle, VehicleEx

```
public class Vehicle {
    protected int speed;

    public void increaseSpeed(int increment) {
        speed += increment;
        System.out.println("Speed increased to " + speed);
    }

    public void decreaseSpeed(int decrement) {
        speed -= decrement;
        System.out.println("Speed decreased to " + speed);
    }
}
```

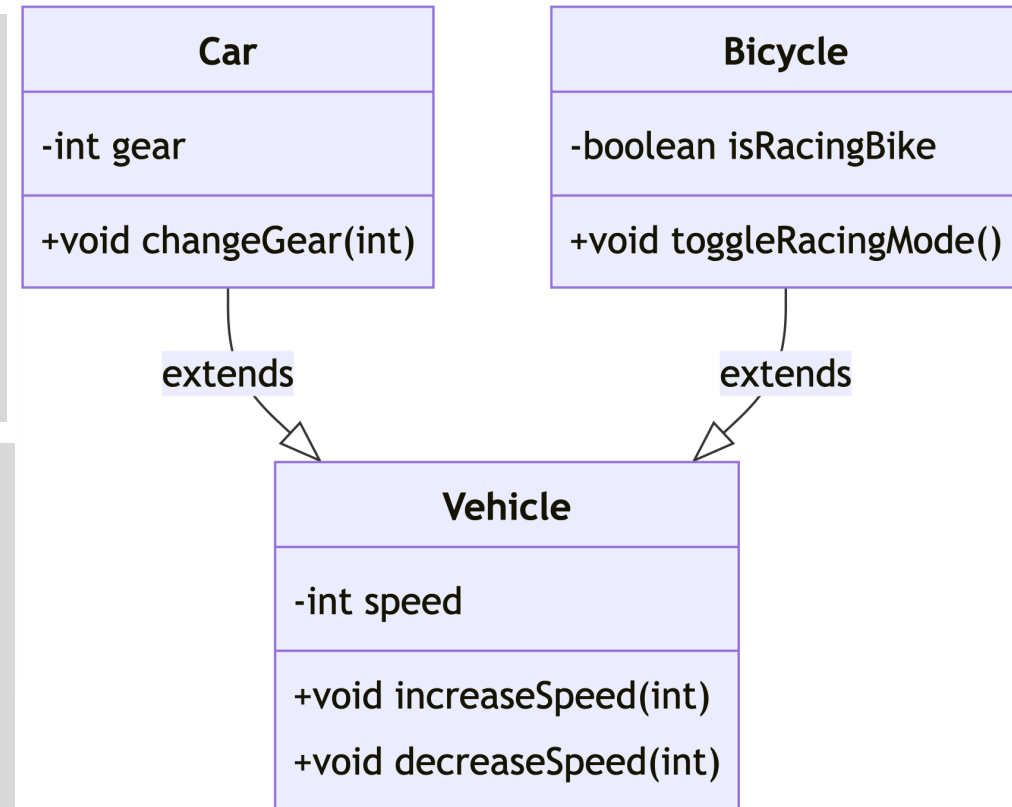


Class inheritance

- One more example of class inheritance: Vehicle, Car, Bicycle, VehicleEx

```
public class Car extends Vehicle {  
    private int gear;  
  
    public void changeGear(int newGear) {  
        gear = newGear;  
        System.out.println("Gear changed to " + gear);  
    }  
}
```

```
public class Bicycle extends Vehicle {  
    private boolean isRacingBike;  
  
    public Bicycle() {  
        this.isRacingBike = false;  
    }  
  
    public void toggleRacingMode() {  
        isRacingBike = !isRacingBike;  
        System.out.println("Racing mode: " + (isRacingBike ? "On" : "Off"));  
    }  
}
```

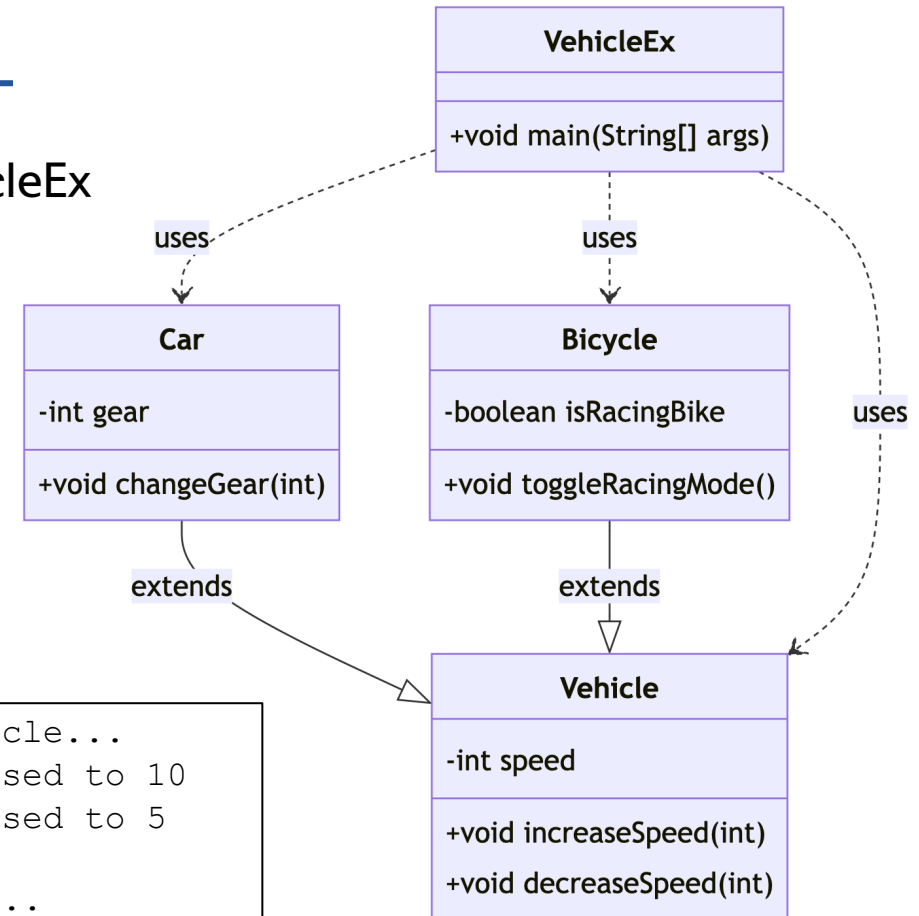


Class inheritance

- One more example of class inheritance: Vehicle, Car, Bicycle, VehicleEx

```
public class VehicleEx {  
    public static void main(String[] args) {  
        Vehicle vehicle = new Vehicle();  
        System.out.println("Testing Vehicle...");  
        vehicle.increaseSpeed(10);  
        vehicle.decreaseSpeed(5);  
  
        Car car = new Car();  
        System.out.println("\nTesting Car...");  
        car.increaseSpeed(20);  
        car.changeGear(2);  
        car.decreaseSpeed(10);  
  
        Bicycle bicycle = new Bicycle();  
        System.out.println("\nTesting Bicycle...");  
        bicycle.increaseSpeed(15);  
        bicycle.toggleRacingMode();  
        bicycle.decreaseSpeed(5);  
    }  
}
```

```
Testing Vehicle...  
Speed increased to 10  
Speed decreased to 5  
  
Testing Car...  
Speed increased to 20  
Gear changed to 2  
Speed decreased to 10  
  
Testing Bicycle...  
Speed increased to 15  
Racing mode: On  
Speed decreased to 10
```

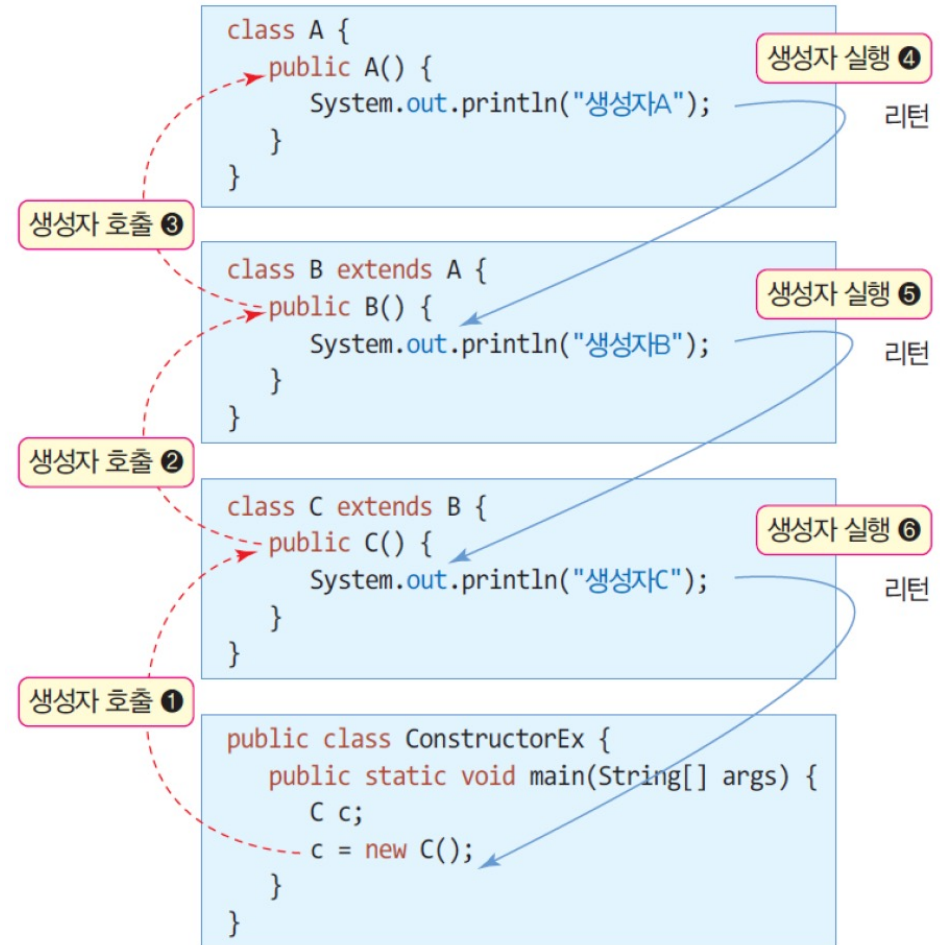


Constructors in class inheritance

- 상속관계에 있는 여러 클래스가 있을 때, 어떤 생성자가 선택되고, 어떤 순서로 호출되는가?
 - method overriding으로 각 클래스마다 여러 개의 생성자가 있을 수 있음
- All constructors for superclass and subclasses will be called
 - case 1) a superclass constructor runs:
 - when an object of the superclass is created
 - when an object of the subclass is created
 - → the constructors of both superclass and subclass run, performing member initialization
 - case 2) a subclass constructor runs:
 - when an object of the subclass is created

Constructors in class inheritance

- Constructor calling **order** in class inheritance follows a **top-down approach**
- Constructor priority when creating a subclass
 - the superclass constructor runs first
 - (calling order) creating a subclass object
 - **subclass** constructor is called
 - **superclass** constructor is called
 - **superclass** constructor runs
 - **subclass** constructor runs



Constructors in class inheritance

- Characteristics of superclass and subclass constructor
 - both **superclass** and **subclass** can have multiple constructors
 - when **subclass** object is created
 - one **superclass** constructor and one **subclass** constructor are executed
- decision method
 - 1) default constructor by the compiler
 - superclass가 명시되어 있지 않을 때, 컴파일러가 자동으로 superclass의 기본 생성자 선택
 - 2) explicit choice by the programmer (개발자의 명시적 선택)
 - 개발자가 superclass의 어떤 생성자를 호출할 지 명시적으로 선택 (using 'super' keyword)

Constructors in class inheritance

- Characteristics of superclass and subclass constructor
 - 1) default constructor by the compiler
 - 컴파일러가 자동으로 superclass의 기본 생성자 선택

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

Constructors in class inheritance

- Characteristics of superclass and subclass constructor
 - 1) default constructor by the compiler
 - 컴파일러가 자동으로 superclass의 기본 생성자 선택

```
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

Constructors in class inheritance

- Characteristics of superclass and subclass constructor

- 1) default constructor by the compiler

- 컴파일러가 자동으로 superclass의 기본 생성자 선택
- → superclass에 기본 생성자가 없으면 오류

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();  
    }  
}
```

Constructors in class inheritance

- Characteristics of superclass and subclass constructor
 - 2) explicit choice by the programmer (개발자의 명시적 선택)
 - 개발자가 superclass의 어떤 생성자를 호출할 지 명시적으로 선택 (using 'super' keyword)
 - How to use?
 - `super(parameter_list)`
 - `parameter_list`와 정확하게 일치하는 superclass의 생성자를 호출
 - **MUST BE the FIRST STATEMENT in the constructor body in subclass** where it is used

Constructors in class inheritance

- Characteristics of superclass and subclass constructor
 - 2) explicit choice by the programmer
 - using 'super' keyword

```
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

Constructors in class inheritance

- Example of super() keyword

```
public class Point {
    private int x, y;

    Point() { this.x = this.y = 0; }
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void showPoint() {
        System.out.println("(" + x + ", " + y + ")");
    }
}
```

```
public class ColorPoint extends Point {
    private String color;

    ColorPoint(int x, int y, String color) {
        super(x, y);
        this.color = color;
    }

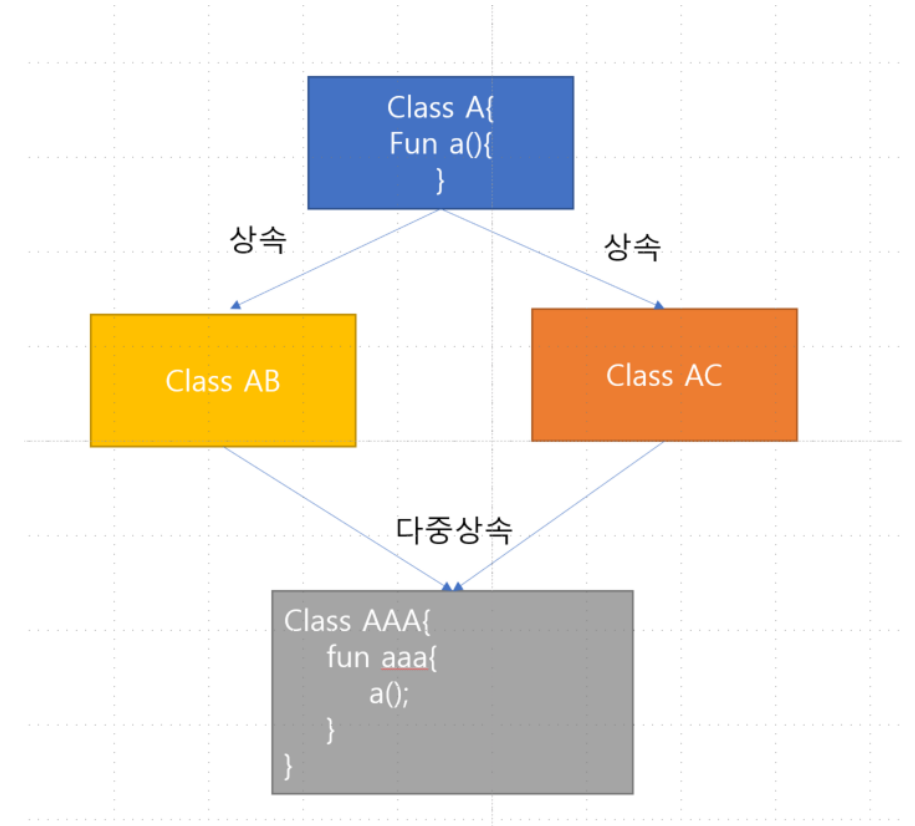
    void showColorPoint() {
        System.out.println(color);
        showPoint();
    }
}
```

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

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

Class inheritance in Java

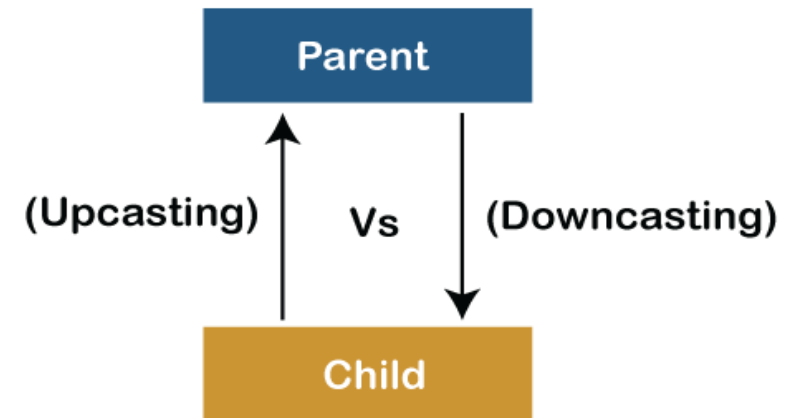
- Java class의 상속 특징
 - 1) 클래스 다중 상속 (multiple inheritance) 불가능
 - 멤버 중복 생성 방지
 - C++/Python → 다중 상속 가능
 - 2) interface의 다중 상속은 가능
 - 3) 모든 Java 클래스는 묵시적으로 'Object' class를 상속 받음
 - java.lang.Object 클래스는 모든 클래스의 superclass



2. Casting

Concept of casting

- Casting의 개념
 - 1) 데이터 간의 타입 변환 (primitive 타입, int → double 등)
 - 2) 상속 관계에 있는 부모와 자식 클래스 간의 형 변환 (reference 타입)
 - 형제 클래스 간 casting 불가능 (타입이 다름)
- Types of casting
 - upcasting (상향 형변환)
 - 하위 클래스의 객체를 상위 클래스로 변환하는 것
 - downcasting (하향 형변환)
 - 상위 클래스의 객체를 하위 클래스로 변환하는 것



Upcasting

- 서브 클래스의 레퍼런스를 슈퍼 클래스 레퍼런스에 대입

➔ 슈퍼 클래스 레퍼런스로 서브 클래스 객체를 가리키게 됨

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

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

class Student extends Person {
    String grade;
    String department;

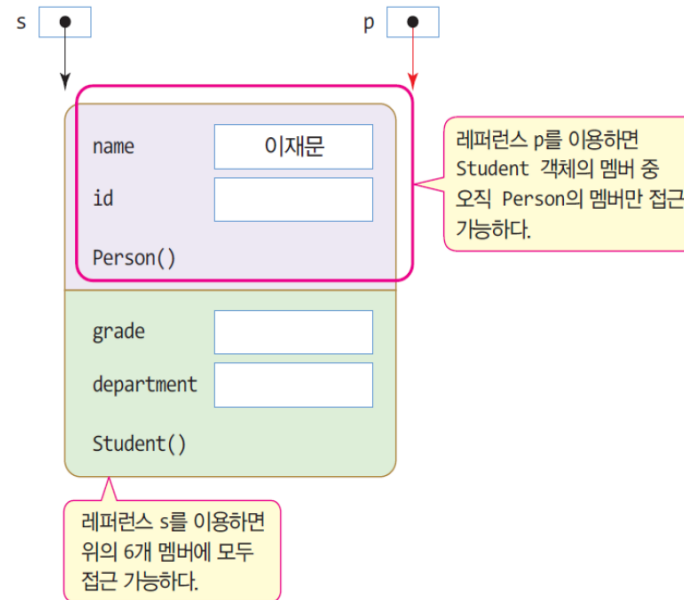
    public Student(String name) {
        super(name);
    }
}

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
    }
}
```

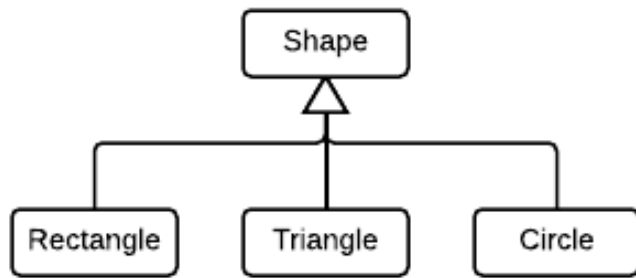
오류



이재문

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();
```

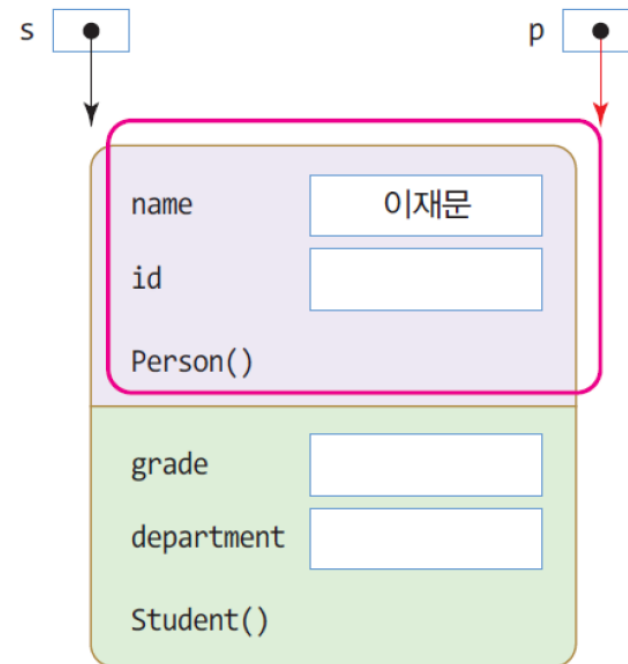
Downcasting

- 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
```

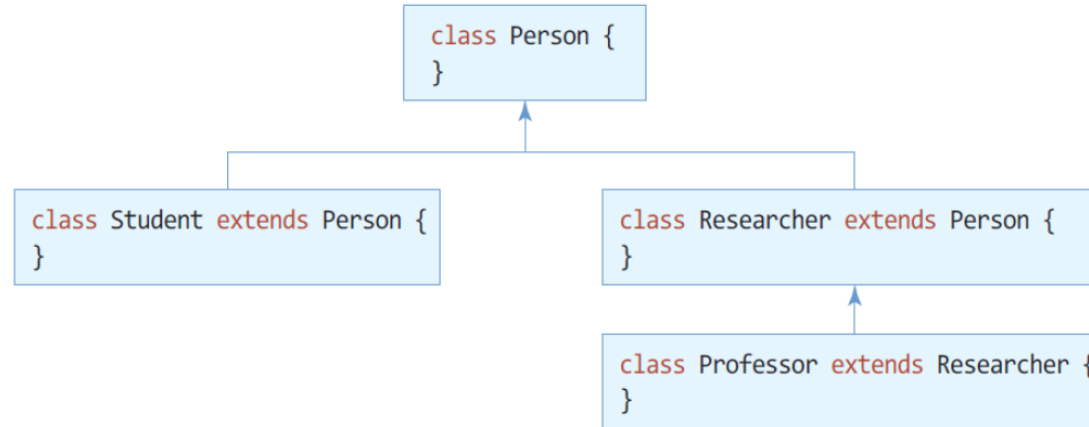
```
public class DowncastingEx {  
    public static void main(String[] args) {  
        Person p = new Student("이재문"); // 업캐스팅  
        Student s;  
  
        s = (Student)p; // 다운캐스팅  
  
        System.out.println(s.name); // 오류 없음  
        s.grade = "A"; // 오류 없음  
    }  
}
```

이재문



instanceof

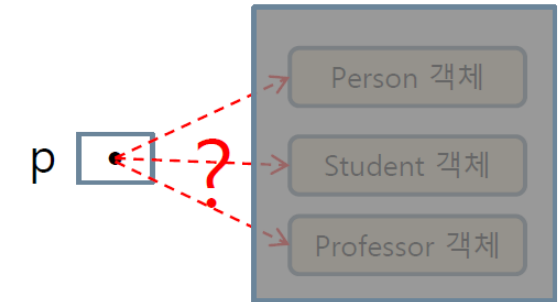
- upcasting된 레퍼런스(객체)의 실제 타입을 구분하기 위하여 활용
 - true 또는 false 반환



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

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

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

```
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
```

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

instanceof

- instanceof 연산자 사용 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

Concept of method overriding

- Allowing a subclass to provide a specific implementation for method that is already defined in its superclass
 - a powerful feature of OOP in aspect of polymorphism
 - enabling actions to be performed differently depending on the actual object's class type at runtime
- 서브 클래스에서 슈퍼 클래스의 메소드를 중복으로 작성하는 기법
- 슈퍼 클래스의 메소드 무력화 → 항상 서브 클래스에 오버라이딩한 메소드의 실행 보장
- “메소드 무시하기”

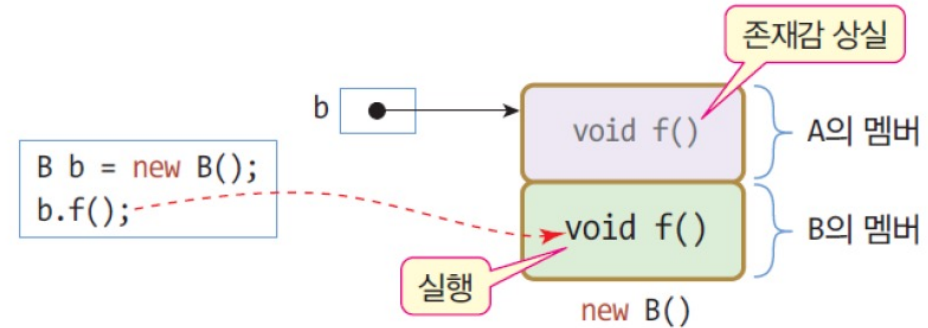
Rules of method overriding

- Method signature
 - the overriding method **MUST HAVE** the same name, return type, and parameter list as the method in the superclass
- Access level
 - the access level **CANNOT** be more restrictive than the method being overridden
 - ex) protected superclass → private subclass (error)
- @Override annotation
 - it is good practice to annotate overridden methods with @Override
 - this annotation tells the compiler that you intend to override a method in the superclass, and the compiler will generate an error if no such method exists

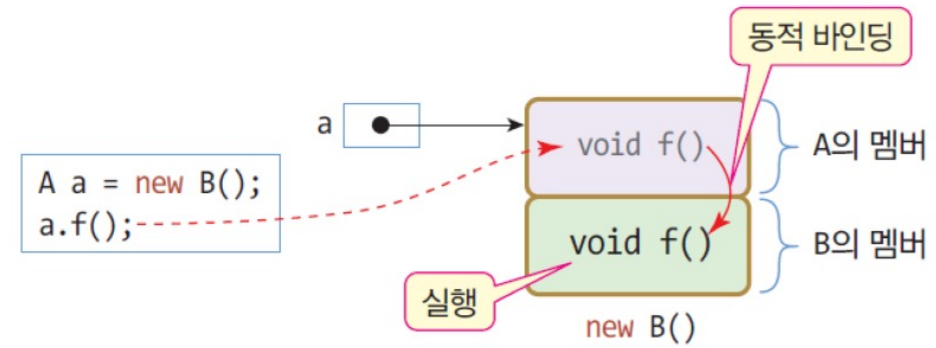
Examples of method overriding

```
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

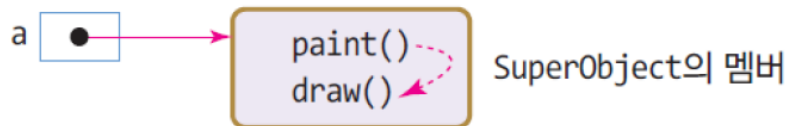


Examples of method overriding

* 오버라이딩 메소드가 항상 호출된다.

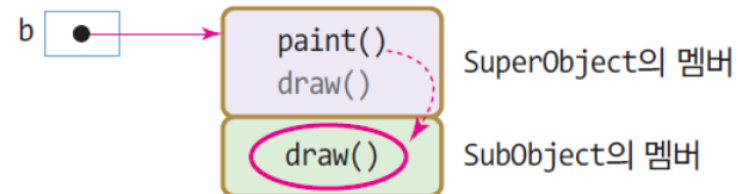
```
public class SuperObject {  
    protected String name;  
    public void paint() {  
        draw();  
    }  
    public void draw() {  
        System.out.println("Super Object");  
    }  
    public static void main(String [] args) {  
        SuperObject a = new SuperObject();  
        a.paint();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

Super Object



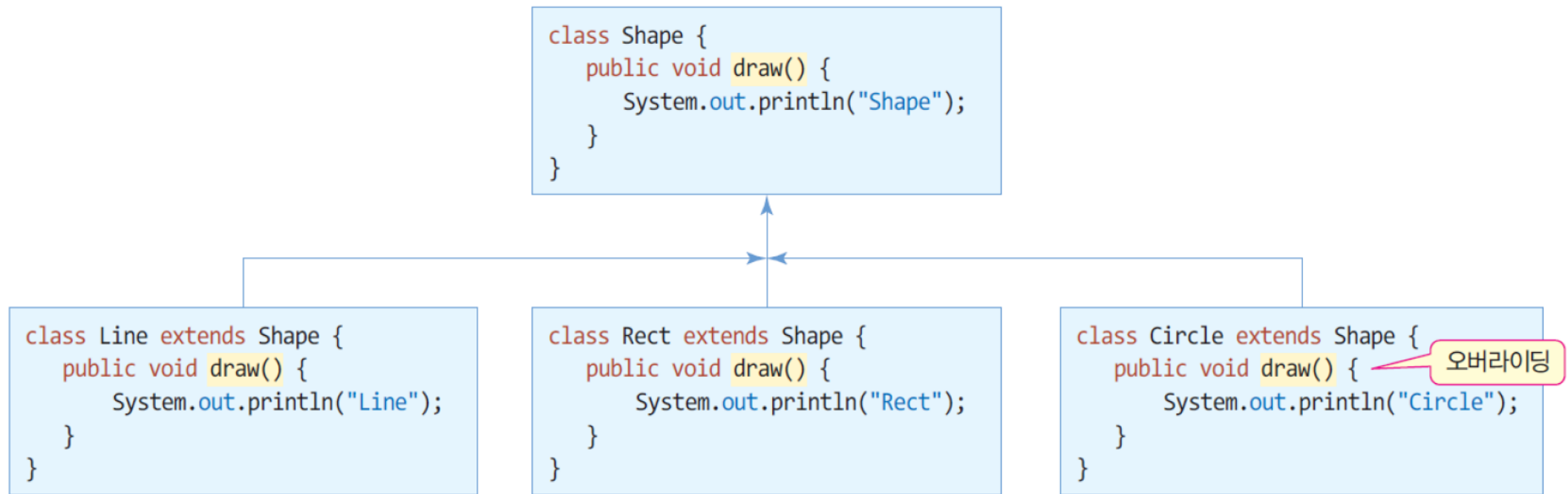
```
class SuperObject {  
    protected String name;  
    public void paint() {  
        draw();  
    }  
    public void draw() {  
        System.out.println("Super Object");  
    }  
}  
public class SubObject extends SuperObject {  
    public void draw() {  
        System.out.println("Sub Object");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

Sub Object



Objectives of method overriding

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



Examples of method overriding

- 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");
    }
}
```

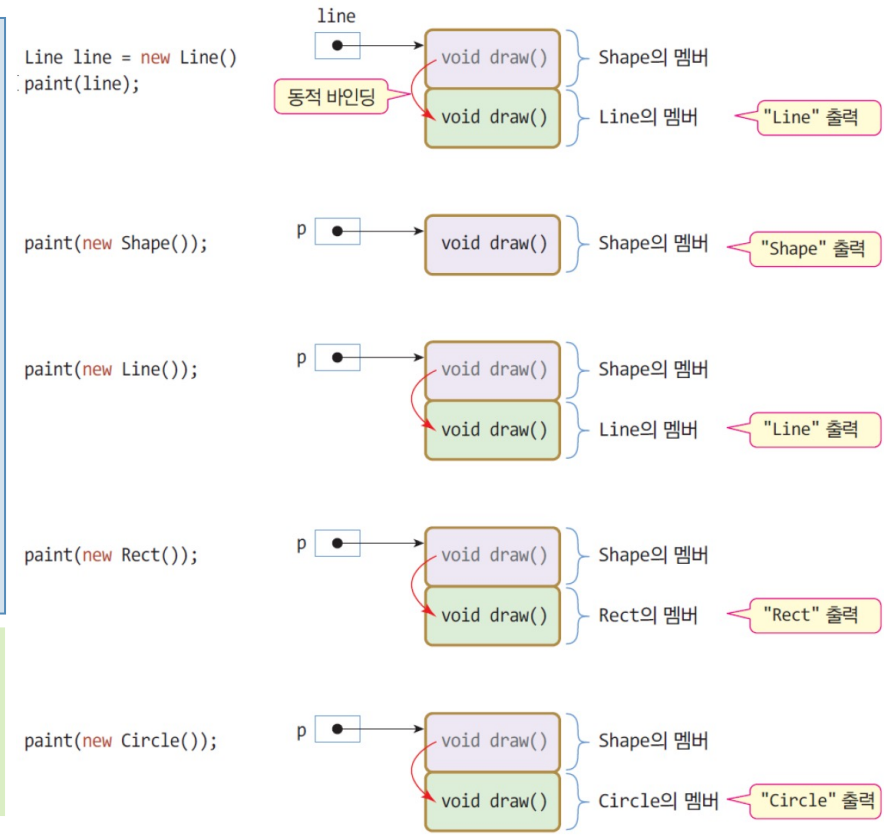
동적바인딩

```
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



Static binding and dynamic binding

- 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

- 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

Static binding and dynamic binding

- Examples of static and dynamic bindings

```
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"); }  
}
```

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

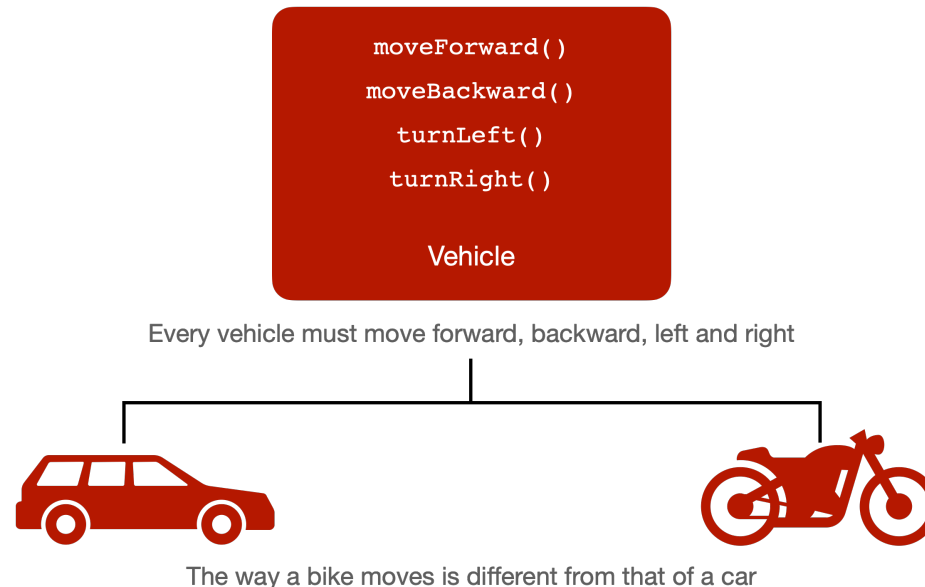

Method overloading vs overriding

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

What is abstract?

- Abstract: 구체적인 존재가 없는 이론적인 것
 - for real life example; 자동차를 디자인할 때
 - 브레이크 페달을 밟으면 → 감속/정지
 - 가속 페달을 밟으면 → 가속/움직임
 - 브레이크 페달을 밟으면 얼만큼 속도가 감속되고 가속 페달을 밟으면 얼만큼 속도가 증가하는지 알지 못함
 - → 속도의 증/감 유무만 알 뿐 구체적인 지식은 없는 상태 → 지식의 추상화



Abstract method

- abstraction in Java
 - abstract method, abstract class, interface
- 추상 메소드
 - a method that is declared **WITHOUT** an implementation; declared using the 'abstract' keyword
 - method body is **provided by the subclass** that extends the abstract class

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



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

Abstract class

- 추상 클래스
 - MUST be inherited by other classes; declared using 'abstract' keyword
 - cannot be instantiated on its own and
 - can contain both **abstract methods** (which have no implementation) and **concrete methods** (which do have an implementation)
 - 추상 메소드를 포함하는 클래스는 반드시 추상 클래스여야 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;
    }
}
```

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

Abstract class

- 추상 클래스의 특징
 - 인스턴스화 (객체 생성) 불가능
 - 추상 → 온전한 형태가 아님

```
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
```



컴파일 오류 메시지

Unresolved compilation problem: Cannot instantiate the type Shape

Abstract class

- 추상 클래스의 상속과 구현
 - 추상 클래스를 상속받으면 추상 클래스가 됨
 - subclass도 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"); }
}
```



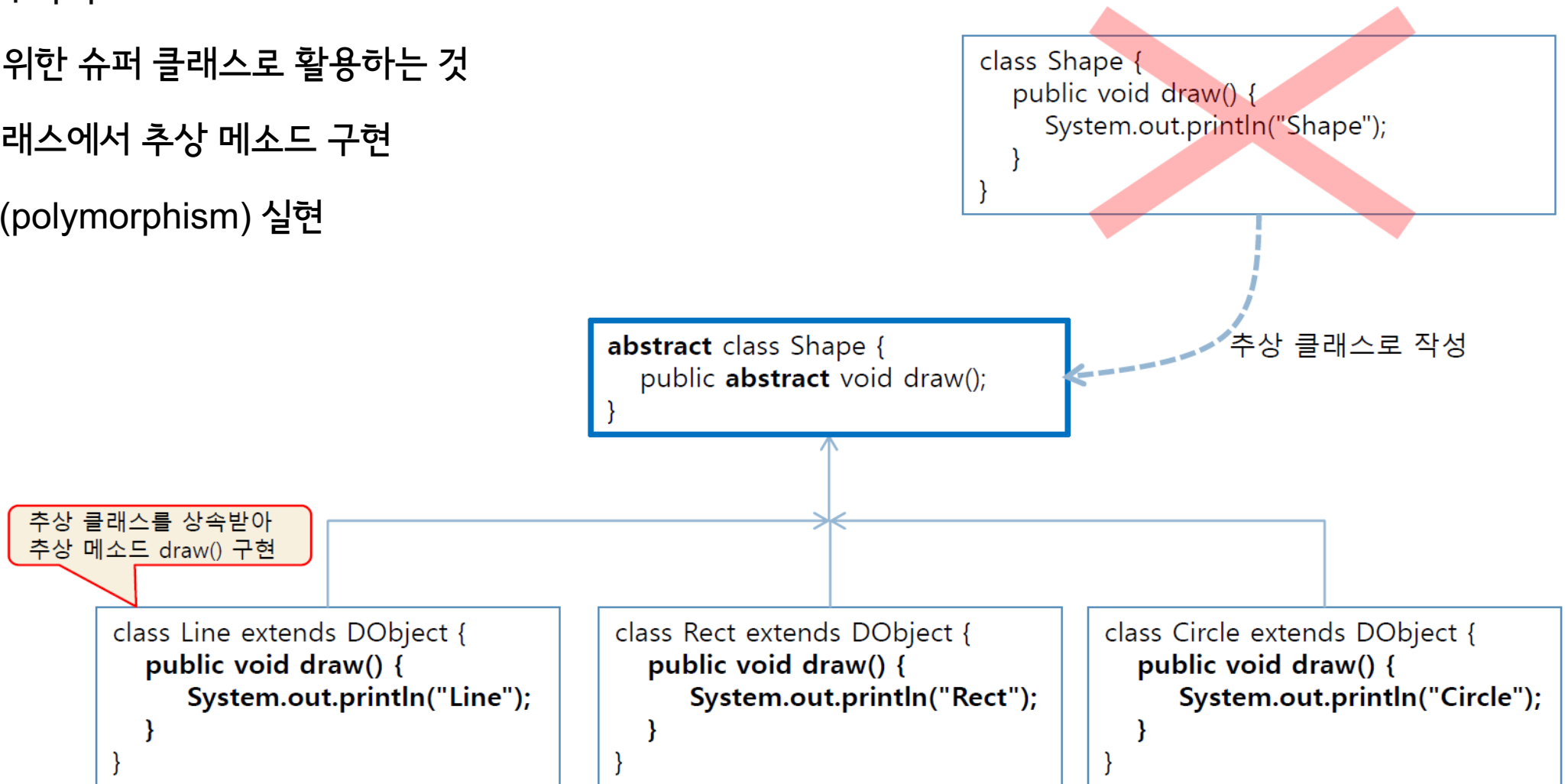
```
A a = new A(); // 컴파일 오류. 추상 클래스의 인스턴스 생성 불가
B b = new B(); // 컴파일 오류. 추상 클래스의 인스턴스 생성 불가
```

- subclass에서 super class (추상 클래스)의 추상 메소드를 구현 시 더 이상 추상 메소드가 아님
 - 메소드 overriding 활용

```
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
```

Abstract class

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



Example of abstraction

- 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
-1
3.0

5. Interface

Java interface

- 인터페이스 (interface)
 - 상수 (변수 X) 와 추상 메소드로만 구성
 - 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.

Interface inheritance

- 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

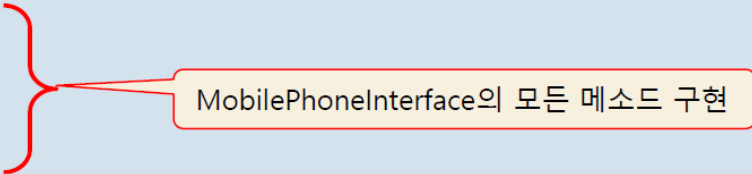
```
interface MusicPhoneInterface extends PhoneInterface, MP3Interface {  
    .....  
}
```

Interface implementation

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

```
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

Example of interface

- 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

abstract class 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
