

# Java Programming

Object Oriented Programming

# Object Oriented Programming

: 객체지향 프로그래밍이란?

- ▶ 현대 컴퓨터 프로그래밍 패러다임의 하나
- ▶ 컴퓨터 프로그램을 여러 개의 독립된 단위(객체)들의 모임으로 파악
- ▶ 특징
  - ▶ 객체는 데이터와 그 데이터를 처리할 수 있는 메서드를 갖는다
  - ▶ 소프트웨어의 부품화, 재사용을 주요 목표로 함
  - ▶ 코드의 재사용성이 높다
  - ▶ 코드의 관리가 쉬워짐 : 개발과 유지보수에 강점, 직관적 코드 분석을 가능하게 함
  - ▶ 신뢰도 높은 프로그램 개발을 가능하게 함

# Object Oriented Programming

: 객체(Object)와 클래스(Class)

## ▶ 객체의 정의

- ▶ 실세계에 존재하는 것, 사물 또는 개념
- ▶ 객체는 정보를 효율적으로 관리하기 위해 의미를 부여하고 분류하는 논리적 단위
- ▶ 객체는 다른 객체와 서로 상호작용하면서 동작한다

## ▶ 객체의 구성 요소

- ▶ 객체는 속성(정보)과 동작(기능)의 집합 -> 객체의 멤버(member: 구성요소)라 함
- ▶ 속성은 필드(field), 동작은 메서드(method)로 정의

## ▶ 클래스

- ▶ 객체를 정의해 놓은 것
- ▶ 객체를 생성하는데 사용되는 설계도

클래스	객체
제품 설계도	제품
TV설계도	TV
붕어빵기계	붕어빵

# Object Oriented Programming

: 인스턴스 (Instance)

- ▶ 객체 (Object)는 인스턴스 (Instance)의 일반적 의미
- ▶ 객체가 메모리에 할당되어 실제 사용될 때 인스턴스라고 부른다
- ▶ 인스턴스 (instance): 객체가 메모리에 할당되어 실제 사용될 때
- ▶ 인스턴스화 (instantiate): 클래스로부터 인스턴스를 생성하는 것



# Object Oriented Programming

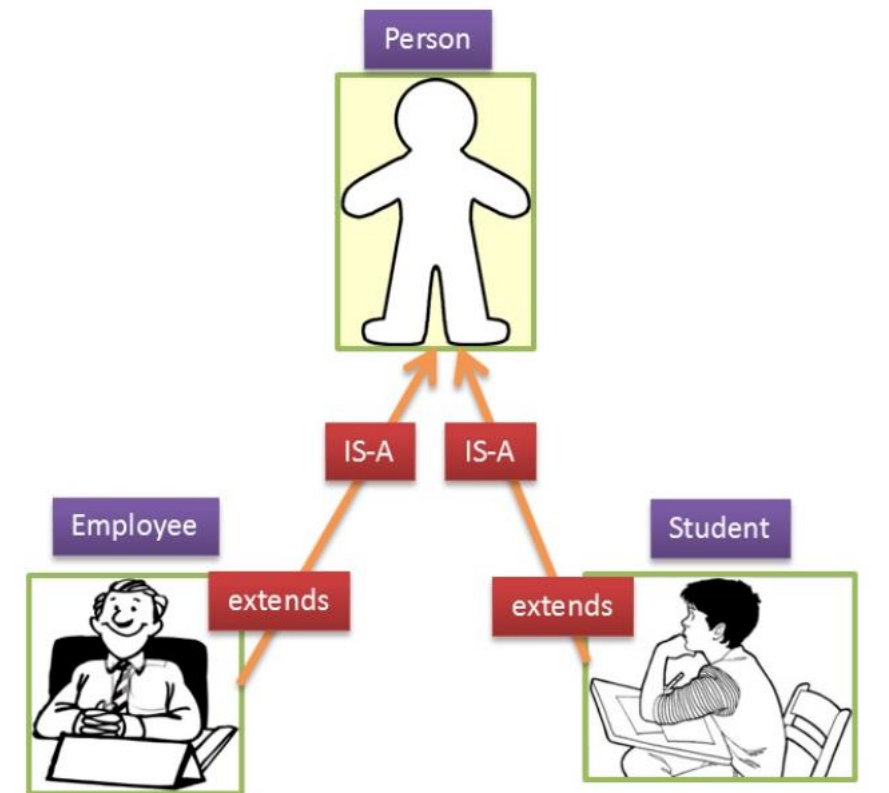
: 객체지향 네 가지 특성

- ▶ 상속성 (Inheritance)
- ▶ 캡슐화 (Encapsulation)
- ▶ 다형성 (Polymorphism)
- ▶ 추상화 (Abstraction)

# Object Oriented Programming

: 상속 (Inheritance)

- ▶ 이미 만든 객체와 비슷하지만 필드와 메서드가 약간 차이가 나는 객체를 생성
  - ▶ 기존의 클래스에서 공통된 필드와 메서드를 상속(재사용)
  - ▶ 더 필요한 필드와 메서드를 추가
- ▶ 코드를 간결하게 하고 코드의 재사용성을 높임



# Object Oriented Programming

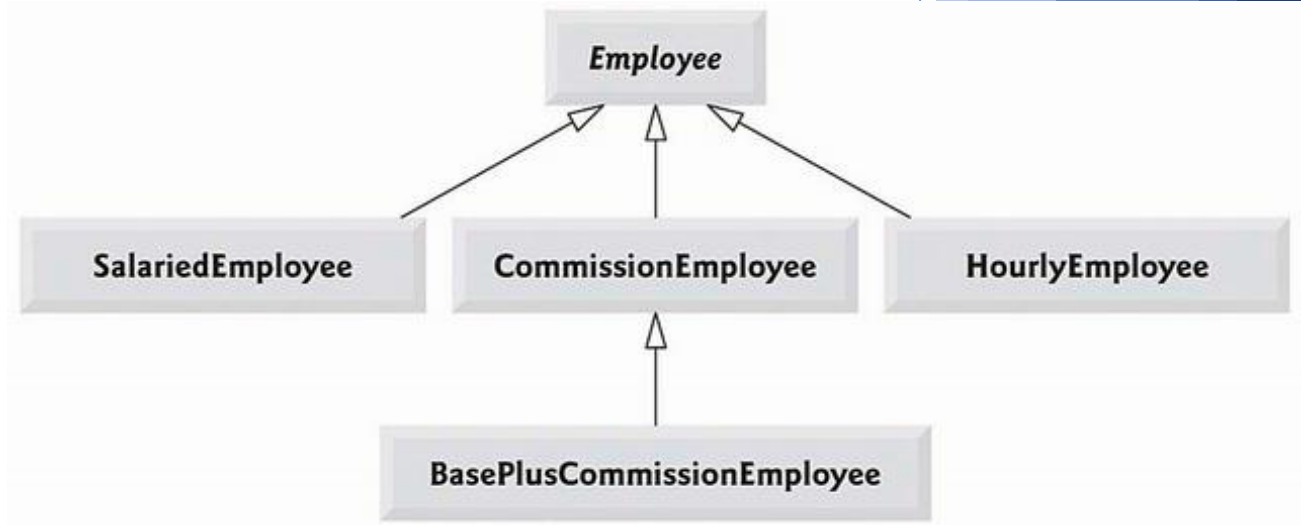
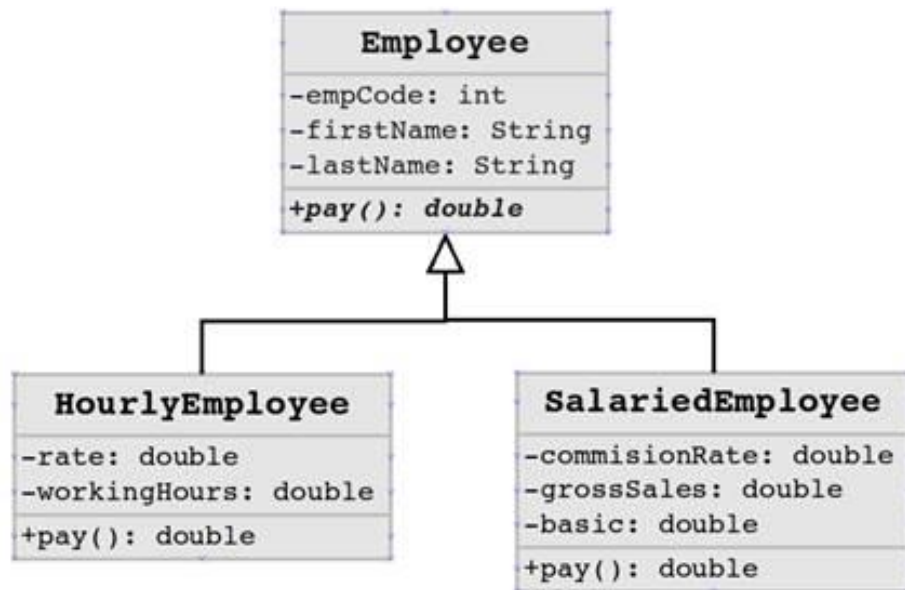
## : 캡슐화 (Encapsulation)

- ▶ 객체의 실제 구현된 내용을 감추고 접근 방법만 노출하는 것
  - ▶ 외부 객체(객체를 사용하는 쪽)에서는 객체의 내부 구조를 알지 못하며
  - ▶ 객체가 노출하여 제공하는 필드와 메서드만 이용할 수 있음
  - ▶ 객체 작성시 개발자는 숨겨야하는 필드와 메서드, 공개하는 필드와 메서드를 구분하여 작성한다
- 
- ▶ 캡슐화하는 이유는 외부의 잘못된 사용으로 객체가 손상되는 것을 피하기 위함
  - ▶ 접근 제한자(**Access Modifier**)를 사용하여 객체의 필드와 메서드의 사용 범위를 제한한다

# Object Oriented Programming

: 다형성 (Ploymorphism)

- ▶ Java 객체지향에서 가장 중요한 개념
- ▶ 하나의 메서드나 클래스를 다양한 구현으로 사용 가능하게 하는 개념
- ▶ 오버로드(Overload)와 오버라이드(Override)를 통해 다형성을 구현



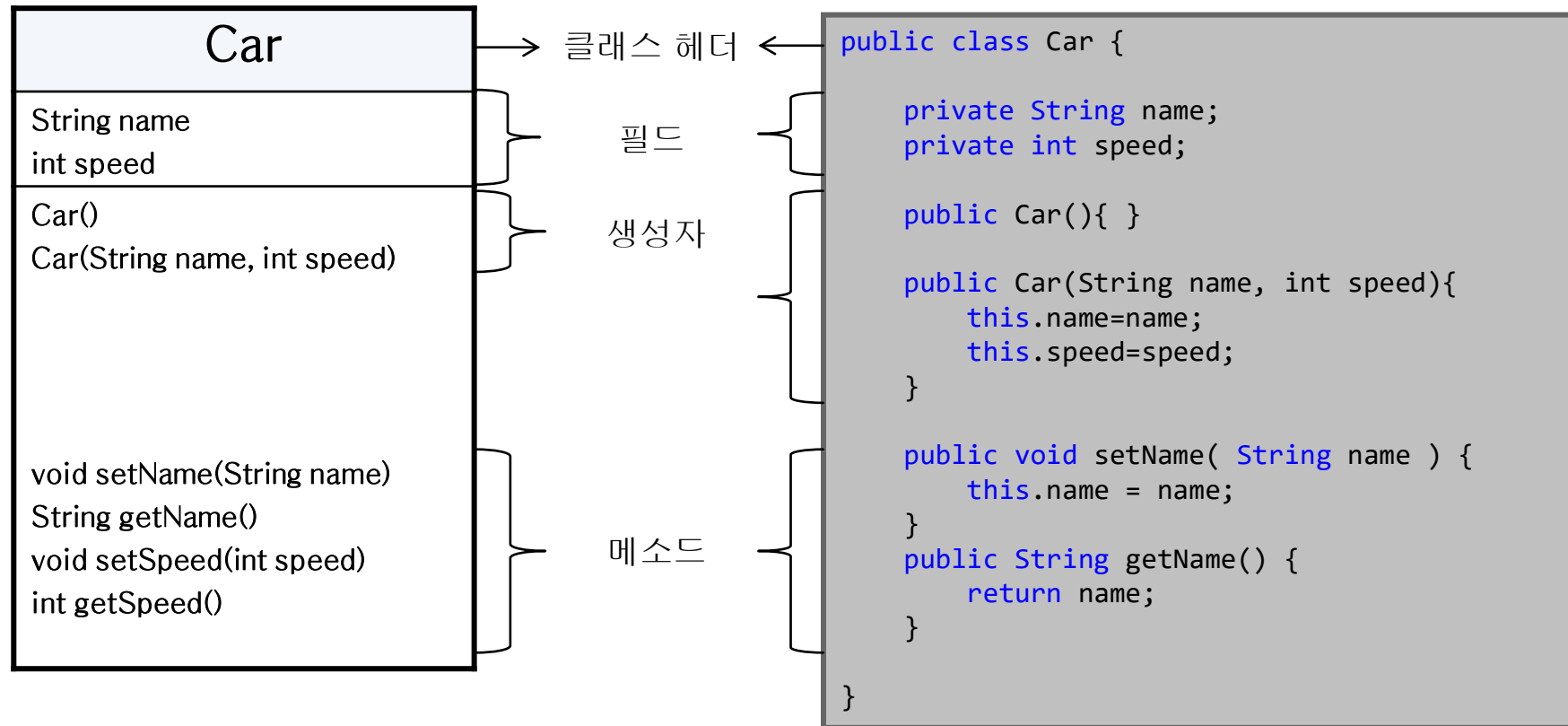


# Java Programming

Java Class

# Java Class

: 클래스의 구조



# Java Class

: 필드 (field)

- ▶ 객체의 데이터, 상태를 저장하는 변수
- ▶ 주로 기본 타입 또는 참조 타입으로 정의
- ▶ 멤버 변수라고도 함

## [문제]

쇼핑몰에서 상품을 관리하기 위해 상품관리 프로그램을 만들려고 합니다. 프로그램을 만들기 전에 업무(비즈니스) 분석을 통해 상품 객체를 분석하고 다음과 같은 Goods클래스를 정의 하였습니다.

Goods 클래스를 정의하고 GoodsApp 클래스에서 Goods 클래스를 테스트 하세요.

- 1) Goods 객체를 하나 생성하고 이 객체에 대한 레퍼런스 변수를 camera 로 합니다.
- 2) 이 객체의 데이터인 각 각의 인스턴스 변수는 다음과 같은 값을 가지도록 합니다.  
**상품명 : "nikon", 가격: 400000**
- 3) 값을 세팅 한 후, 객체의 데이터를 출력해 보세요.

Goods
String name int price

# 연습문제

: 클래스의 정의와 사용

## [문제]

다음의 데이터를 추가한 후, 출력해 보세요

-상품명 : "LG그램", 가격: 900000

-상품명 : "머그컵", 가격: 2000

Goods
String name int price

# Java Class

: 접근자 (Access Modifier)

- ▶ 객체의 필드와 메서드에 접근을 제한하기 위해 사용
- ▶ 정보 은닉을 위한 방법 (캡슐화)
- ▶ 정보 접근 수준에 따라 **public**, **protected**, **default**, **private** 네 가지가 있다

지시자	클래스 내부	동일 패키지	상속받은 클래스	이외의 영역
private	●	×	×	×
default	●	●	×	×
protected	●	●	●	×
public	●	●	●	●

`public > protected > default > private`

# 연습문제

## : 접근자 연습

### [문제]

- Goods 클래스의 필드 접근자를 public으로 변경해 봅니다.
- Goods 클래스의 필드 접근자를 default보다 강한 접근 제어자인 private로 지정하여 어떤 변화가 있는 지 확인해 봅니다.

지시자	클래스 내부	동일 패키지	상속받은 클래스	이외의 영역
private	●	×	×	×
default	●	●	×	×
protected	●	●	●	×
public	●	●	●	●

public > protected > default > private

# Java Class

: 메서드 (method)

- ▶ 객체의 기능 또는 행동을 정의
- ▶ 정의 방법

```
1 public 2 int 3 getSum( 4 int i, int j ) {  
    int result = i + j; 5  
    return result; 6  
}
```

1. 접근 지정자
2. 리턴 타입
3. 메서드 이름
4. 메서드 인자 (파라미터)
5. 구현코드
6. 리턴문

- ▶ 호출 방법

```
1 int 2 sum = util.getSum(3 3, 4 2);  
5
```

1. 자료형
2. 변수명
3. 레퍼런스변수
4. 메서드명
5. 메서드 인자 (파라미터)

# Java Class

## : 메서드 (method)

- ▶ 매개변수(parameter)
  - ▶ 메서드를 선언할 때 괄호 안에 표현된 Input 값을 나타내는 변수 (type1 name1, type2 name2, ...)
  - ▶ 메서드 호출에서 들어가는 구체적인 값은 인자(Argument)라고 함
- ▶ 반환타입(return type)
  - ▶ 메서드는 0개 혹은 1개의 값을 Output으로 반환할 수 있음
  - ▶ 반환 값이 없을 때: void
  - ▶ 반환 값이 있을 때: int, boolean, Goods, ...
  - ▶ 반환 되는 값은 메서드 선언에서 정의된 반환 타입과 일치해야 함
- ▶ 메서드 이름
  - ▶ 자바의 식별자 규칙 대/소문자, 숫자, \_, \$ 조합하여 지을 수 있고 숫자로 시작할 수 없다.
  - ▶ 관례에 따라 소문자로 작성하고 두 단어가 조합될 경우 두 번째 시작문자는 대문자로 짓는다.
  - ▶ 메서드명은 기능을 쉽게 알 수 있도록 작성하는 것이 좋다



# Java Class

: 가변 인수

- ▶ 메소드의 매개 변수의 개수를 알 수 없을 때 사용
- ▶ 가장 간단한 해결방법은 매개변수를 배열로 선언하는 것

```
double sum(double[] values) { }
```

- ▶ 이러한 경우, 메소드 호출시 배열을 넘겨주어 여러 개의 값을 전달할 수 있다

```
double[] numbers= { 1, 2, 3, 4, 5 };  
double result = sum(numbers);  
double result = sum(new double[] { 1, 2, 3, 4, 5 });
```

- ▶ 방법 2: 매개 변수를 ... 를 이용하여 선언
  - ▶ 자동으로 배열이 생성되고 매개값으로 사용된다

```
double sum(double ... values) { }
```

```
double result = sum(1, 2, 3, 4, 5 );
```

# Java Class

## : Getter, Setter

- ▶ 일반적으로 객체의 데이터는 객체 외부에서 직접적으로 접근하는 것을 막는다.
- ▶ 객체의 외부에서 객체 내부의 데이터를 마음대로 읽고 쓸 경우 데이터의 무결성을 보장하기 힘들기 때문이다.
- ▶ 메소드를 통한 접근을 하게 되면 객체의 데이터를 변경할 경우 무결성 체크를 할 수 있다.

✓ 클래스를 정의할 때 필드는 private로 하여 객체 내부의 정보를 보호하고(정보은닉) 필드에 대한 Setter와 Getter를 두어 객체의 값을 변경하고 참조하는 것이 좋다.

- 외부에서 읽기만 가능하게 하기 위해선 Getter만 해당 필드에 대해서만 작성하면 된다.
- 외부에서 쓰기만 가능하게 하기 위해선 Setter만 해당 필드에 대해서만 작성하면 된다.
- Getter와 Setter가 없으면 객체 내부 전용 변수가 된다.
- 보통 Getter는 getXXX로 명명하지만, 필드 타입이 boolean인 경우 isXXX로 명명하는 것이 관례

# 연습문제

: 클래스 정의/확장 연습

## [문제]

아래와 같이 클래스를 정의하여 프로그램을 작성하세요

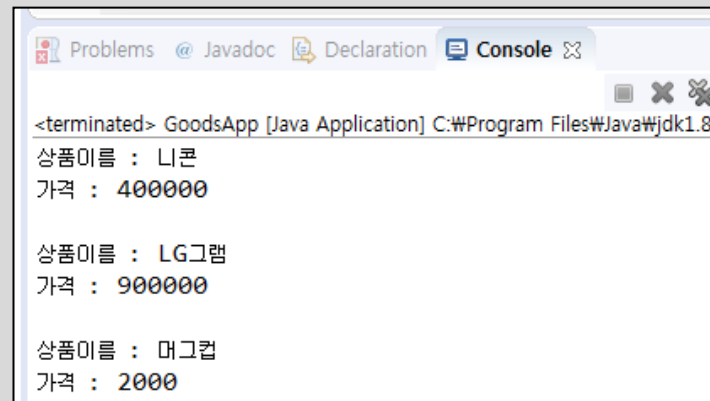
### Goods 클래스를 만드세요

- 1) 필드접근자를 private로 작성해서 외부에서 접근할 수 없게 합니다.
- 2) 필드에 값을 저장할 수 있도록 set메소드를 만드세요.
- 3) 필드에 값을 읽을 수 있도록 get메소드를 만드세요.
- 4) 아래와 같이 상품의 모든 정보를 출력해 주는 showInfo()를 만드세요.

Goods
String name int price
- showInfo()

### GoodsApp 클래스 만드세요

- 1) showInfo()메소드를 이용하여 다음과 같이 출력하세요.



```
<terminated> GoodsApp [Java Application] C:\Program Files\Java\jdk1.8
상품이름 : 니콘
가격 : 400000

상품이름 : LG그램
가격 : 900000

상품이름 : 머그컵
가격 : 2000
```

# 연습문제

: 클래스 정의/확장 연습

Goods 예제를 떠올리며 만들어 봅시다

## [문제]

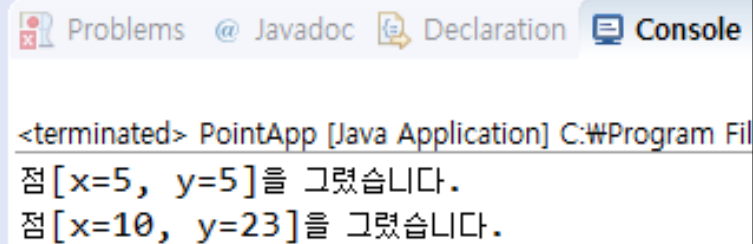
아래와 같이 클래스를 정의하여 프로그램을 작성하세요

### Point 클래스를 만드세요

- x, y 좌표를 나타낼 수 있는 필드 작성
- x, y 좌표에 접근할 수 있는 getter/setter 메소드 작성
- 다음 실행 결과를 참조하여 draw() 메소드 작성

### PointApp 클래스 만드세요

- 1) draw() 메소드를 호출하여 다음과 같이 출력하세요



The screenshot shows an IDE console window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the following text:

```
<terminated> PointApp [Java Application] C:\WProgram Fil  
점 [x=5, y=5]을 그렸습니다.  
점 [x=10, y=23]을 그렸습니다.
```

# 연습문제

: 클래스 정의/확장 연습

Goods 예제를 떠올리며 만들어 봅시다

## [문제]

아래와 같이 클래스를 정의하여 프로그램을 작성하세요

### Song 클래스를 만드세요

**Song 클래스는 다음과 같은 필드를 가지고 있습니다.**

- 노래의 제목을 나타내는 title
- 가수를 나타내는 artist
- 노래가 속한 앨범 제목을 나타내는 album
- 노래의 작곡가를 나타내는 composer
- 노래가 발표된 연도를 나타내는 year
- 노래가 속한 앨범에서 트랙 번호를 나타내는 track

**1) 필드의 접근을 제한하고 getter/setter 메소드를 통해 접근하세요.**

**2) 노래정보를 출력하는 showInfo() 메소드를 작성하세요.**

### SongApp 클래스 만드세요

1) showInfo() 메소드를 호출하여 다음과 같이 출력하세요

Problems @ Javadoc Declaration Console

No consoles to display at this time.

아이유, 좋은날 ( Real, 2010, 3번 track, 이민수 작곡 )

BIGBANG, 거짓말 ( Always, 2007, 2번 track, G-DRAGON 작곡 )

버스커버스커, 벚꽃엔딩 (버스커버스커1집, 2012, 4번 track, 장범준 작곡 )

# Java Class

## : 생성자 (Constructor)

- ▶ **new** 연산자와 같이 사용되어 클래스로부터 객체를 생성할 때 호출되고 객체의 초기화를 담당 한다.
- ▶ 생성자를 실행 시키지 않고 클래스로부터 객체를 만들 수 없다.
- ▶ 생성자가 성공적으로 실행되면 **JVM의 Heap**영역에 객체가 생성되고 객체의 참조 값이 참조변수에 저장 된다.
- ▶ 몇 가지 조건을 제외하면 메소드와 같다.
- ▶ 모든 클래스에는 반드시 하나 이상의 생성자가 있어야 한다.

# Java Class

## : 생성자 (Constructor)

- ▶ 생성자의 정의
  - ▶ 생성자의 이름은 클래스와 같아야 한다
  - ▶ 생성자의 리턴 값은 없다. (하지만 **void**를 쓰지 않는다)

```
접근자 클래스이름 ( 파라미터 ) {  
  
    // 인스턴스 생성시 수행할 코드  
    // 주로 인스턴스 변수의 초기화 코드  
  
}
```

```
public class Goods {  
    public Goods() {  
        // 초기화 코드  
    }  
  
    public Goods( String name, int price ) {  
        // 초기화 코드  
    }  
}
```

# Java Class

## : 생성자 (Constructor)

- ▶ 기본 생성자
  - ▶ 매개 변수가 없는 생성자
  - ▶ 클래스에 생성자가 하나도 없으면 컴파일러가 기본 생성자를 추가한다
  - ▶ 생성자는 필요에 따라 여러 개 작성할 수 있다.

```
public class Goods {  
  
    public Goods() {  
        // 초기화 코드  
    }  
}
```



# 연습문제

## : 클래스 정의/확장 연습

### [문제]

- 1) Goods 클래스에서 생성되는 모든 필드 초기화 하는 생성자를 정의합니다.
- 2) 오류가 발생하면 오류가 발생하는 원인을 생각해 보세요.
- 3) 생성자 오버로딩을 확인합니다.

L

### [문제]

- 1) Point 클래스의 기본 생성자와 모든 필드를 초기화 할 수 있는 생성자를 작성하고 테스트 합니다.

### [문제]

- 1) Song 클래스의 기본 생성자와 모든 필드를 초기화 할 수 있는 생성자를 작성하고 테스트 합니다.

# Java Class

: this

- ▶ **this** 키워드는 메서드 호출을 받는 객체를 의미한다
- ▶ 현재 사용중인 객체 그 자체를 의미한다
- ▶ **this()**는 클래스의 한 생성자에서 다른 생성자를 호출할 때 사용할 수 있다.

## [문제]

- 1) Goods 클래스에서 이름만 입력 받아 초기화하는 생성자를 오버로딩합니다.
- 2) 모든 필드 입력 생성자에서 이 생성자를 호출하도록 합니다.
- 3) 이 생성자에서 다른 생성자를 호출하도록 합니다.

## [문제]

- 1) Song 클래스에서 노래 제목과 가수만 입력 받아 필드를 초기화하는 생성자를 하나 더 오버로딩 합니다.
- 2) 모든 필드 입력 생성자에서 이 생성자를 호출하도록 합니다..

# Java Class

: method overloading

반환값이 다른 것은 오버로딩이 아닙니다

- ▶ 하나의 클래스에 같은 이름의 메서드가 여러 개 존재할 수 있다
- ▶ 각 메서드들은 매개변수 타입, 개수, 그리고 순서가 다른 형태로 구별된다

※ **메소드 시그니처(Signature) : 메소드 인자의 타입, 개수, 순서**

## [문제]

1) Point 클래스에 점을 안보일 수 있는 기능까지 추가된 draw() 메소드를 하나 더 추가하고 아래 실행결과가 나도록 테스트 하세요



```
<terminated> PointApp [Java Application] C:\Program Files\Java
점 [x=5, y=5]를 그렸습니다.
점 [x=10, y=23]를 그렸습니다.
점 [x=5, y=5]를 지원합니다.
점 [x=10, y=23]를 지원합니다.
```

# 연습문제

아래 TV 클래스의 main 메소드를 실행할 수 있도록, 요구 조건을 참조하여 TV 클래스를 정의 하세요.

- 1) 모든 필드는 private으로 접근 제어를 하고 getter만 작성합니다. (channel, volume, power 필드 read-only)
- 2) channel, volume, power의 초기값을 각각 7, 20, false 로 초기화 하는 생성자 작성
- 3) 기본 생성자 오버로딩
- 4) void power( boolean on ) 메소드 구현
- 5) void channel( int channel ) 메소드 구현 (1~255 유지)
- 6) void channel( boolean up ) 메소드 오버로딩 (1~255 유지, 1씩 증감)
- 7) void volume( int volume ) 메소드 구현 ( 0 ~ 100 유지 )
- 8) void volume( boolean up ) 메소드 오버로딩 (0 ~ 100 유지, 1씩 증감)
- 9) void status() 메소드 구현( TV 정보 출력)

## TV

int channel  
int volume  
boolean power

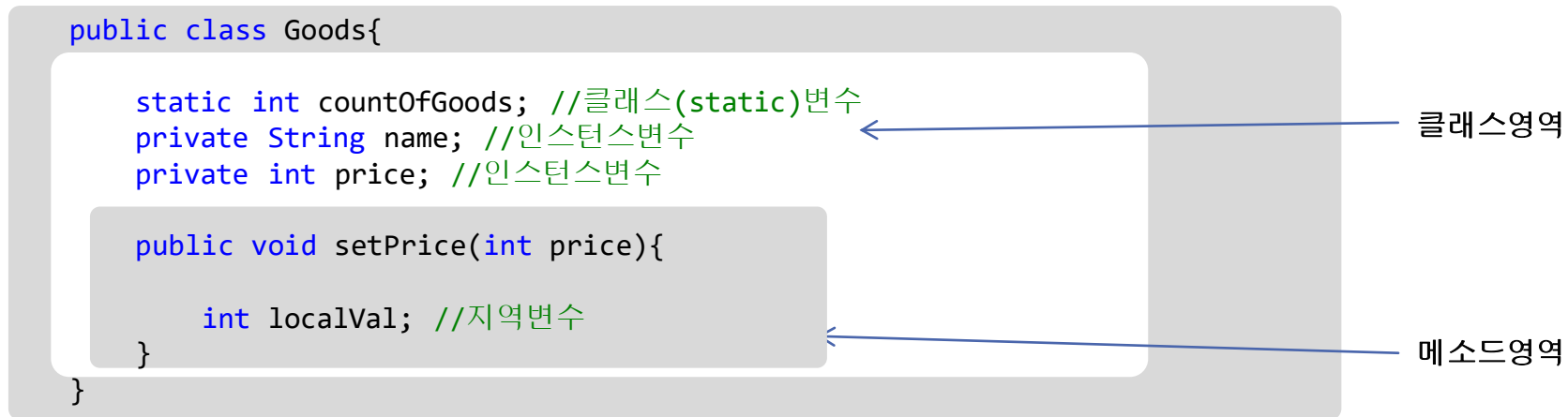
getChannel()  
getVolume()  
power( boolean )  
channel( int )  
channel( boolean )  
volume( int )  
volume( boolean )  
status();

```
public class TVApp {  
    public static void main( String[] args ) {  
        TV tv = new TV( 7, 20, false);  
  
        tv.status();  
  
        tv.power( true );  
        tv.volume( 120 );  
        tv.status();  
  
        tv.volume( false );  
        tv.status();  
  
        tv.channel( 0 );  
        tv.status();  
  
        tv.channel( true );  
        tv.channel( true );  
        tv.channel( true );  
        tv.status();  
  
        tv.power( false );  
        tv.status();  
    }  
}
```

# Java Class

: static 변수, instance 변수, local 변수

## ▶ 선언 위치에 따른 변수의 종류

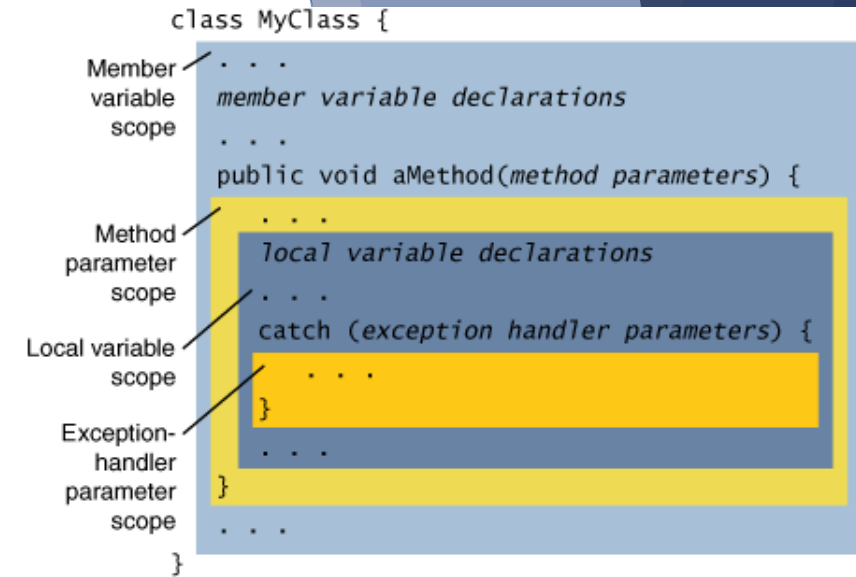


변수의 종류	선언위치	생성시기
클래스(static) 변수	클래스 영역	클래스가 메모리에 올라갈 때
인스턴스 변수		인스턴스 생성 시
지역변수	메서드 영역	변수 선언문 수행 시

# Java Class

## : 변수의 범위 (Scope)

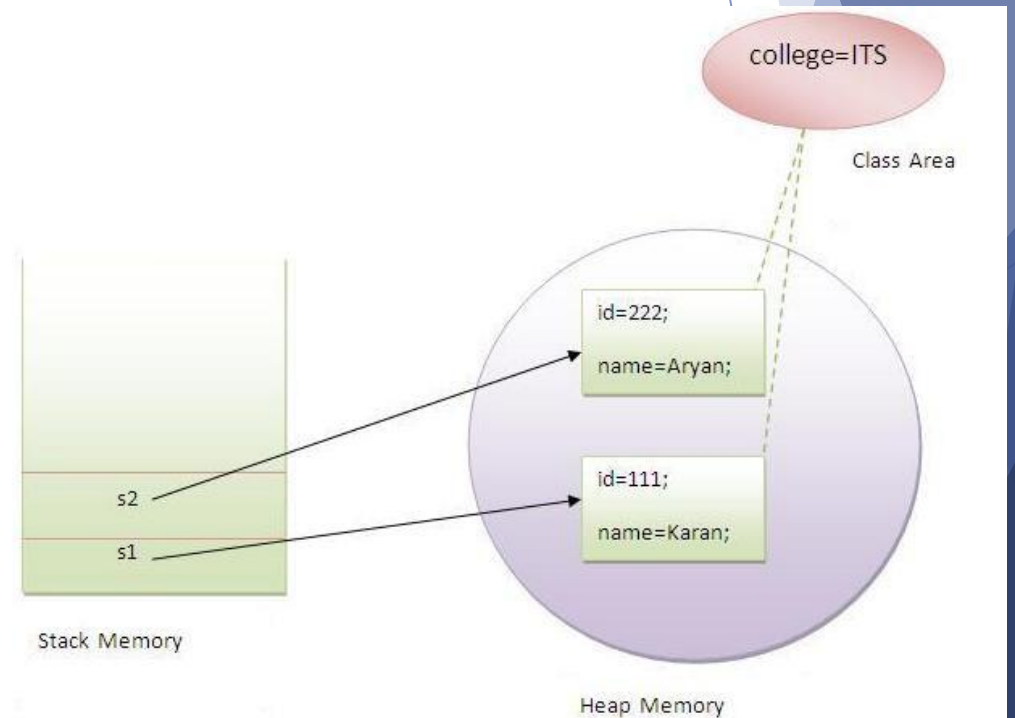
- ▶ **인스턴스 변수(instance variable)**
  - ▶ 각 인스턴스의 개별적인 저장공간. 인스턴스 마다 다른 값 저장가능
  - ▶ 인스턴스 생성 후, '참조변수.인스턴스 변수명' 으로 접근
  - ▶ 인스턴스를 생성할 때 생성되고, 참조변수가 없을 때 가비지 컬렉터에 의해 자동 제거됨
- ▶ **클래스(static) 변수(class variable)**
  - ▶ 같은 클래스의 모든 인스턴스들이 공유하는 변수
  - ▶ 인스턴스 생성없이 '클래스이름.클래스(스태틱)변수명' 으로 접근
  - ▶ 클래스가 로딩될 때 생성되고 프로그램이 종료될 때 소멸
- ▶ **지역 변수(local variable)**
  - ▶ 메소드 내에 선언되며, 메소드의 종료와 함께 소멸
  - ▶ 조건문, 반복문의 블록{} 내에 선언된 지역변수는 블록을 벗어나면 소멸
- ▶ **인스턴스 변수 와 클래스(static) 변수**
  - ▶ 인스턴스 변수는 인스턴스가 생성될 때마다 생성되므로 인스턴스 마다 각기 다른 값을 유지할 수 있지만, 클래스(static)변수는 모든 인스턴스가 하나의 저장공간을 공유하므로 항상 공통된 값을 갖는다.



# Java Class

: 클래스 (static) 멤버

- ▶ 전역 변수와 전역 함수를 만들 때 사용
- ▶ 모든 클래스에서 공유하는 전역 변수나 전역 함수를 만들어 사용할 수 있다
  - ▶ 클래스 멤버라고도 함
- ▶ 객체를 생성하지 않고 접근할 수 있다
- ▶ static 메서드 내에서는 **this** 사용 불가
- ▶ static 메서드 내에서는 **static** 멤버만 접근할 수 있다
- ▶ static 멤버의 초기화는 **static** 블록에서 할 수 있다
  - ▶ static 블록은 클래스가 메모리에 로딩될 때 실행된다
  - ▶ static 블록은 생성자보다 앞서 실행된다



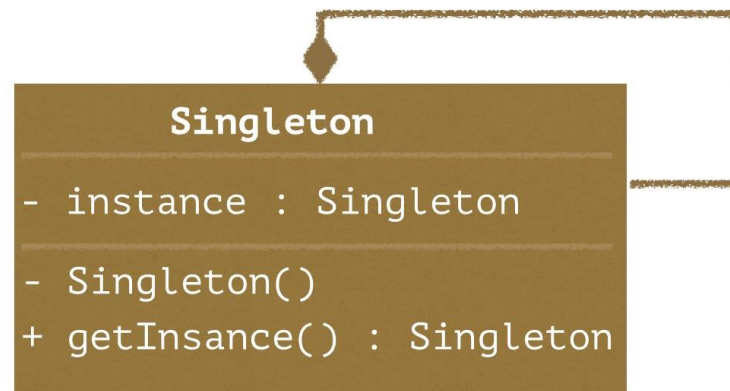
# Java Class

: static의 활용 - Singleton

- ▶ 전체 프로그램 상에서 동일한 인스턴스를 사용해야 할 경우 활용
  - ▶ 어떠한 상황에서도 **단 하나**의 인스턴스만 유지한다
- ▶ 4대 디자인 패턴에 포함될 정도로 유명하고, 유용한 패턴
- ▶ 예) 스마트폰의 주소록

## Singleton Design Pattern

“Ensure that a class has only one instance and provide a global point of access to it.”





# Java Class

## : 패키지(package)

### ▶ 패키지란?

- ▶ 서로 관련 있는 클래스 또는 인터페이스들의 묶음

### ▶ 패키지 이용의 장점

- ▶ 클래스들을 묶음 단위로 제공하여 필요할 때만 사용 가능 (import 문 이용)
- ▶ 클래스 이름의 혼란을 막고 충돌을 방지
- ▶ 패키지 단위의 접근 권한 지정 가능

### ▶ 사용법

- ▶ 1) import 패키지명.클래스명(인터페이스명);     //     지정된 클래스(인터페이스)만 불러옴
  - ▶ 예) `import java.applet.Applet;` //     java.applet 패키지 내의 Applet을 불러옴
- ▶ 2) import 패키지명.\*; // 패키지 내의 모든 클래스(인터페이스)를 불러옴
  - ▶ 예) `import java.io.*;`     //     java.io 패키지 내 모든 클래스(인터페이스)를 불러옴

# Java Class

: 새로운 패키지의 작성

- ▶ 패키지의 이름과 계층 구조를 결정
- ▶ 패키지를 위치시킬 디렉토리에 패키지의 계층 구조와 동일한 디렉토리 구조 생성
- ▶ 패키지에 추가할 클래스들을 생성하고 해당 디렉토리로 이동
- ▶ 새로 생성된 패키지가 위치한 디렉토리를 환경변수에 추가

```
package com.example.myproject;

public class MyClass {
    public void greeting( String name ) {
        System.out.println( "Hello " + name );
    }
}
```

```
package com.example.myproject.test;
import com.example.myproject.MyClass;

public class MyProjectTest {
    public static void main( String[] args ) {
        MyClass myClass = new MyClass();
        myclass.greeting( "Java" );
    }
}
```

# 접근 제한자

- ▶ 외부에서 접근할 수 있는 것과 없는 것을 구분하는 것이 바람직
- ▶ 클래스, 멤버 등에 적용 가능
- ▶ 접근자를 생략하면 **default** 제한자가 설정된다

제한자	적용대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스

# Java Programming

Inheritance and Polymorphism

상속과 다형성

# Inheritance and Polymorphism

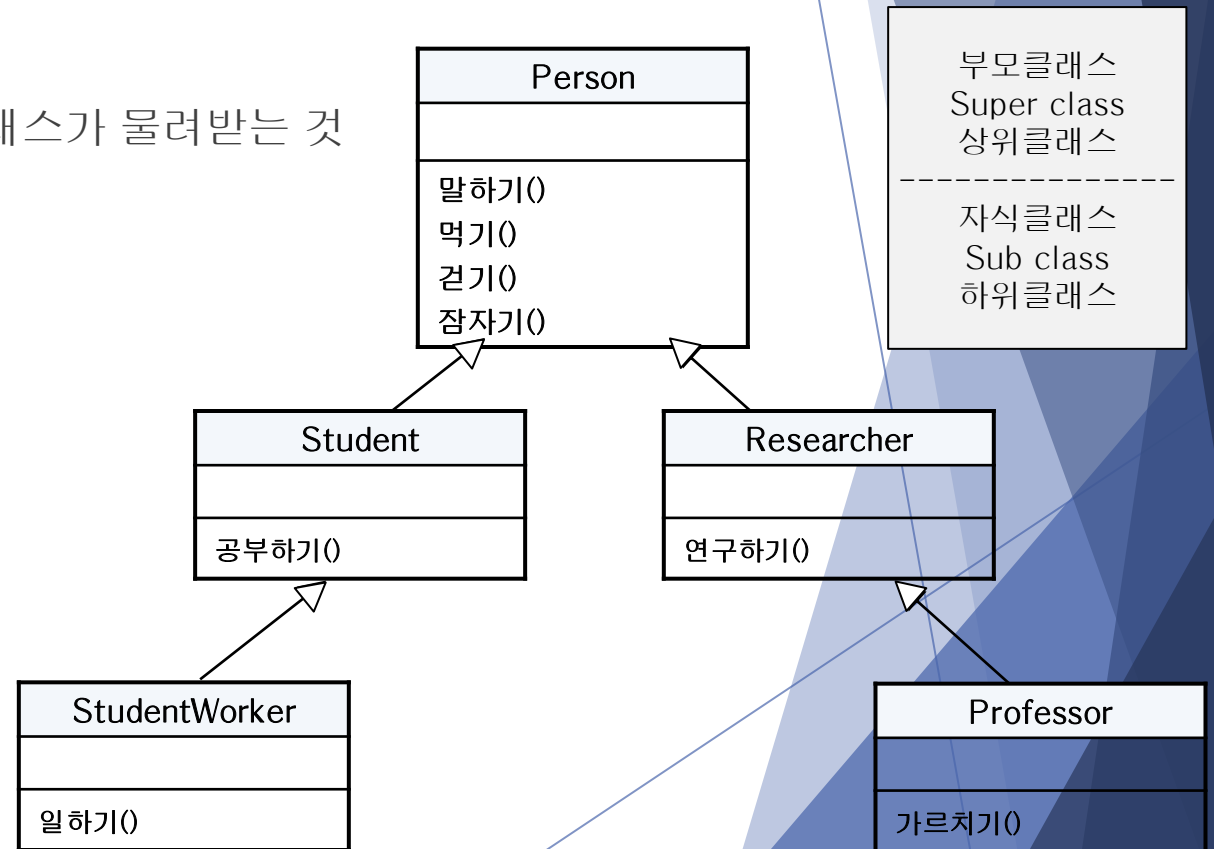
: 상속 (Inheritance)

## ▶ 상속

- ▶ 부모 클래스에 정의된 필드와 메서드를 자식 클래스가 물려받는 것

## ▶ 상속의 필요성

- ▶ 클래스 사이의 멤버 중복 선언 불필요
- ▶ 필드, 메서드 재사용으로 클래스가 간결
- ▶ 클래스간 계층적 분리 및 관리



# Inheritance and Polymorphism

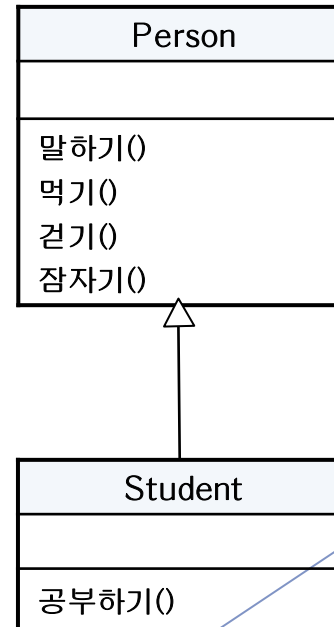
## : 상속 (Inheritance)

- ▶ 자바 언어 상속의 특징
  - ▶ 다중 상속을 지원하지 않음
  - ▶ 상속의 회수에 제한을 두지 않음
  - ▶ 최상위 클래스는 `java.lang.Object`

### ▶ 상속 선언의 예

```
public class Person {  
    public void tell();  
    public void eat();  
    public void walk();  
    public void sleep();  
}
```

```
public class Student extends Person  
{  
    public void study()  
}
```



# Inheritance and Polymorphism

: 상속 (Inheritance)

## ▶ 상속과 생성자

- ▶ 자식 생성자에서 특별한 지시가 없으면 부모 클래스의 기본 생성자가 선택된다
- ▶ 부모 클래스의 특정 생성자를 호출해야 할 경우 `super()` 로 명시적으로 호출
- ▶ 부모의 필드나 메서드에 접근시에는 **super** 키워드를 사용

## ▶ 상속과 접근제한자

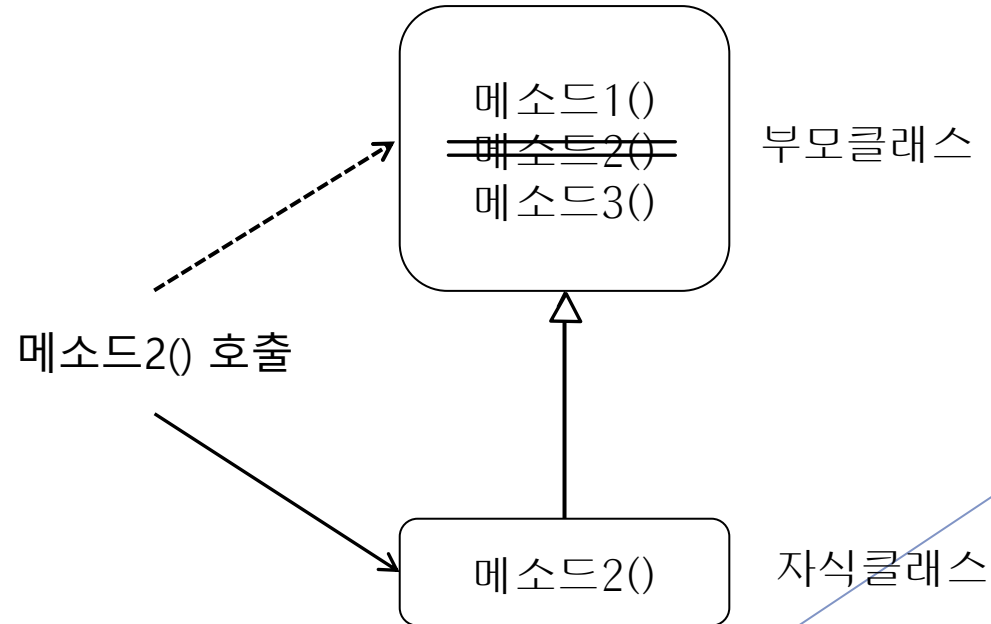
지시자	클래스 내부	동일 패키지	상속받은 클래스	이외의 영역
private	●	×	×	×
default	●	●	×	×
protected	●	●	●	×
public	●	●	●	●

public > protected > default > private

# Inheritance and Polymorphism

: 메서드 오버라이딩 (Method Overriding)

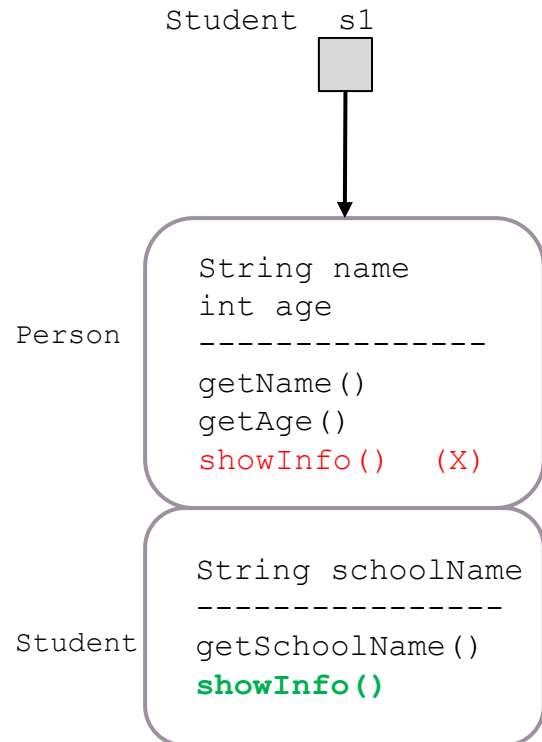
- ▶ 부모 클래스와 자식 클래스의 메서드 사이에서 발생하는 관계
- ▶ 부모 클래스의 메서드를 동일한 이름으로 재 작성
  - ▶ 같은 이름, 같은 리턴타입, 같은 시그너처
- ▶ 부모 클래스 메서드 무시하기
- ▶ @Overriding



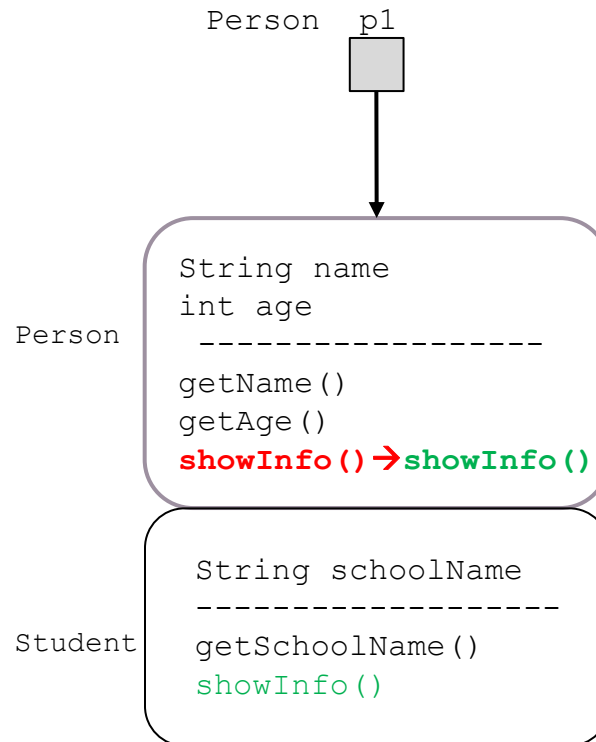


# Inheritance and Polymorphism

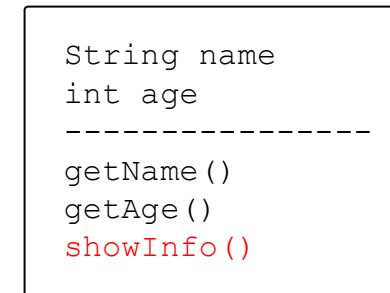
`Student s1 = new Student();`



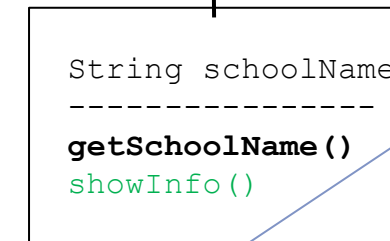
`Person p1 = new Student();`



**Person**



**Student**

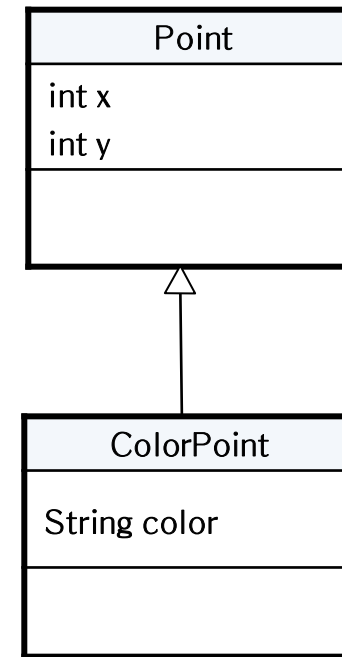


# 연습문제

: 상속 연습

## [문제]

- Point 클래스를 만드세요.
  - ✓생성자, getter/setter, draw()
- Point 클래스를 상속받아 ColorPoint 클래스를 만드세요.
  - ✓생성자, getter/setter, draw()
- PointApp 클래스를 통해서 인스턴스를 생성하고 showInfo()를 통해 확인하세요.
  - ✓Point p = **new Point(4,4);**
  - ✓ColorPoint cp1 = **new ColorPoint("red");**
  - ✓ColorPoint cp2 = **new ColorPoint(10,10,"blue");**
- 자식 클래스와 부모클래스의 생성자 순서를 확인하세요

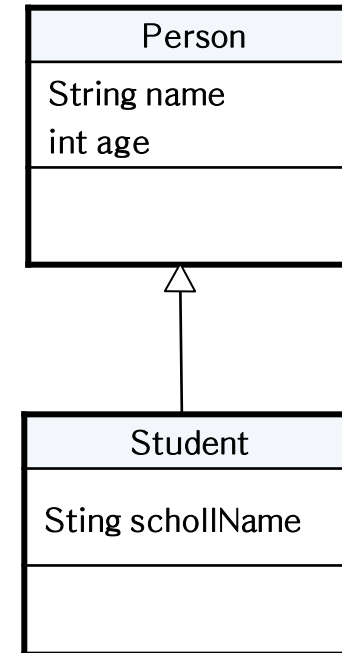


# 연습문제

: 상속 연습

## [문제]

- Person 클래스를 만드세요.  
✓생성자, getter/setter, showInfo()
- Person 클래스를 상속받아 Student 클래스를 만드세요.  
✓생성자, getter/setter, showInfo()
- PersonApp 클래스를 통해서 인스턴스를 생성하고 showInfo()를 통해 확인하세요.  
✓Person p = **new Person("정우성", 45);**  
✓Student s1 = **new Student("서울고등학교");**  
✓Student s2 = **new Student("이정재", 45, "한국고등학교" );**
- 자식 클래스와 부모클래스의 생성자 순서를 확인하세요



# Inheritance and Polymorphism

## : Upcasting and Downcasting

- ▶ 업캐스팅 (Up Casting or Promotion)
  - ▶ 자식 클래스가 부모 클래스 타입으로 변환되는 것
  - ▶ 명시적으로 타입 변환을 하지 않아도 된다
- ▶ 다운캐스팅 (Down Casting)
  - ▶ 업캐스팅된 것을 원래대로 되돌리는 것
  - ▶ 명시적으로 타입 변환을 해야 한다
  - ▶ 다운캐스팅시 어떤 클래스를 객체화한 것인지 알고자 한다면 **instanceof** 를 사용한다
- ▶ 다형성(Polymorphism)
  - ▶ 같은 타입이지만, 실행 결과가 다른 객체를 이용할 수 있는 성질
  - ▶ 자바는 부모 클래스로의 타입 변환을 허용한다

# Java Programming

Abstract Class and Interface

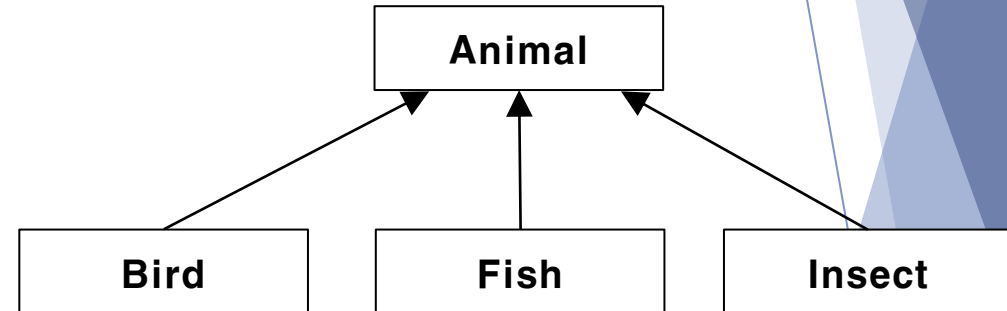
추상 클래스와 인터페이스

# Abstract Class and Interface

## : 추상 클래스

- ▶ 추상화
  - ▶ 객체의 속성과 기능 중 중요한 것들은 남기고 필요 없는 것은 없애는 것
  - ▶ 또는 객체들간의 공통되는 특성을 추출하는 것
- ▶ 추상 클래스
  - ▶ 실체 클래스의 공통적인 특성들을 추출해서 선언한 클래스
  - ▶ 실체 클래스를 만들기 위한 **부모 클래스**로만 사용
    - ▶ 스스로 객체가 될 수는 없다
  - ▶ 확장을 위한 용도로만 사용
  - ▶ 하나 이상의 추상 메서드를 가짐
  - ▶ 속성(필드)과 기능(메서드)을 정의할 수 있다
- ▶ 추상 메서드
  - ▶ 구현이 불가능한 메서드로서 선언만 한다
  - ▶ 추상 클래스를 상속하는 실체 자식 클래스는 추상 메서드를 반드시 구현해야 한다
  - ▶ 추상 메서드는 추상 클래스에만 존재한다

실체 클래스를 위한 **설계 규격**



# Abstract Class and Interface

## : 추상 클래스

### ▶ 추상 클래스의 선언

- ▶ 클래스 선언에 **abstract** 키워드

```
public abstract class 클래스명 {  
    //필드  
    //생성자  
    //메소드  
}
```

### ▶ 추상 클래스의 상속

- ▶ **extends** 키워드 사용
- ▶ 추상 클래스를 상속하는 클래스는 반드시 추상 클래스 내의 추상 메서드를 구현해야 함
  - ▶ 특정 기능의 구현을 강요하는 측면도 있음 (예: 자동차의 브레이크 기능은 꼭 구현되어야 함)

### ▶ 활용

- ▶ 여러 클래스들이 **상당수 공통점**을 가지고 있으나 **부분적으로 그 처리 방식이 다른 경우**  
부모 클래스를 추상 클래스로 정의하여 자식 클래스들이 각각 해당 메서드를 구현

# 연습문제

: 추상 클래스

- 연습문제

- Shape 클래스를 상속받은 Circle, Rectangle 클래스를 정의 하세요.
- Shape 클래스는 추상 클래스로 추상 메소드 double area(), void draw()를 가짐
- Shape 클래스를 상속 받은 각 클래스들은 자신만의 area(), draw() 메소드를 구현

L

- 연습문제

- 다음 두 클래스의 공통된 속성과 기능을 추출하여 부모 클래스 Phone을 정의한 후, Telephone(유선) 과 SmartPhone(무선) 클래스도 정의하고 테스트해 보세요.
- 전원메소드를 trunOn(boolean on) 메소드로 사용하세요

Telephone
String number;
power( boolean on ) call( String number )

Smartphone
String number;
turnOn() turnOff() call( String number ) searchInternet(String url)



# Abstract Class and Interface

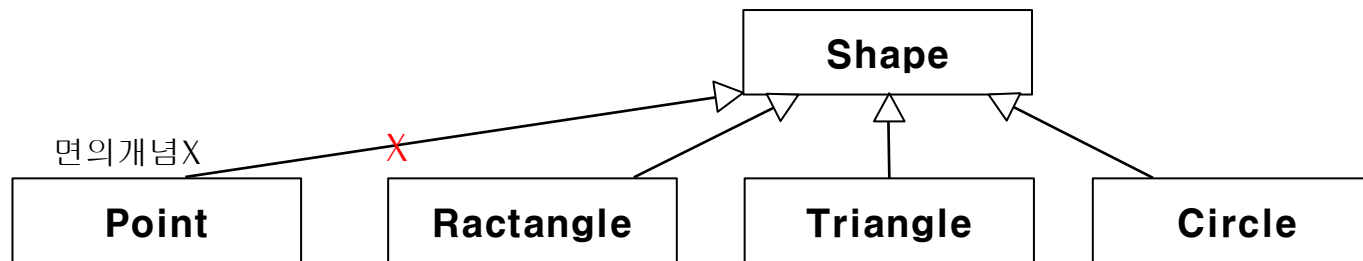
## : 인터페이스 (Interface)

### ▶ 개념

- ▶ 서로 관계가 없는 물체들이 상호작용을 하기 위해 사용하는 장치나 시스템
- ▶ 클래스 구조상 관계와 상관 없이 클래스들에 의해 구현될 수 있는 규약

### ▶ 사용 목적

- ▶ 클래스들 사이의 유사한 특성을 부자연스러운 상속 관계를 설정하지 않고 얻어냄
- ▶ 개발 코드를 수정하지 않고, 가용하는 객체를 변경할 수 있도록 하기 위함



### ▶ 활용

- ▶ 하나 혹은 그 이상의 클래스들에서 똑같이 구현되어질 법한 메서드를 선언하는 경우
- ▶ 클래스 자체를 드러내지 않고 객체의 프로그래밍 인터페이스를 제공하는 경우

# Abstract Class and Interface

## : 인터페이스 (Interface)

### ▶ 인터페이스 선언

```
public interface 인터페이스명 {  
    //추상메소드 (abstract 키워드를 지정하지 않아도 됨)  
}
```

### ▶ 인터페이스 구현

```
public class Point implements 인터페이스명 {  
    //추상메소드 구현 (abstract 키워드를 지정하지 않아도 됨)  
}
```

### ▶ 다중 상속을 지원하지 않는 자바에서 다중 상속의 장점을 활용할 수 있음

#### ▶ 인터페이스는 다중 `implements`를 할 수 있다

```
public class Point implements Drawable, Resizable {  
  
}
```

# 연습문제

: interface 연습

- ▶ Shape 예제에서 **Point** 클래스를 추가하고 **Drawable** 인터페이스를 추가해 봅니다

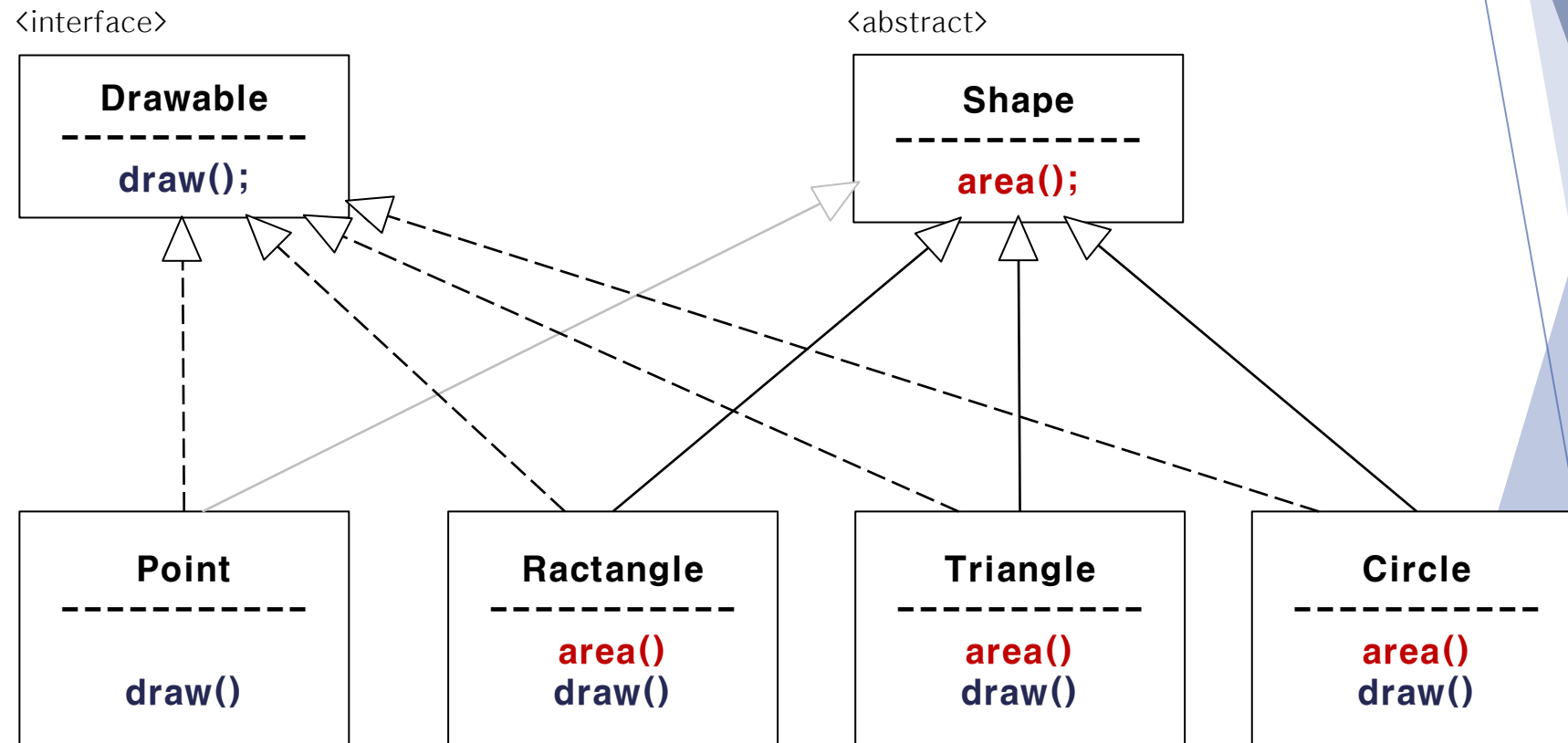
```
public interface Drawable {  
    public void draw();  
}
```

- ▶ instanceof 연산자

```
Shape c = new Circle();  
  
// 객체가 Circle 클래스의 인스턴스 인가?  
System.out.println( c instanceof Circle );  
  
// 객체가 Drawable 인터페이스를 구현하였는가?  
System.out.println( c instanceof Drawable );  
  
// 객체가 Rectangle 클래스의 인스턴스 인가?  
System.out.println( c instanceof Rectangle );  
  
// 객체가 Shape 클래스의 인스턴스 인가?  
System.out.println( c instanceof Shape );
```

# Abstract Class and Interface

: 추상클래스 vs 인터페이스 (Interface)



면의개념이 아님(shape이 될 수 없음)  
그림판에서는 같이 관리 되어야 함

# Abstract Class and Interface

: 인터페이스 (Interface)

## ▶ 일반 클래스 vs 추상 클래스 vs 인터페이스

	일반클래스	추상클래스	인터페이스
메소드 (Method)	모두 실체 메소드 *	실체 메소드 추상 메소드	모두 추상 메소드
필드 (Instance Variable)	가질 수 있음	가질 수 있음	가질 수 없음 상수 필드는 가능
객체화 (Instantiation)	가능	불가	불가

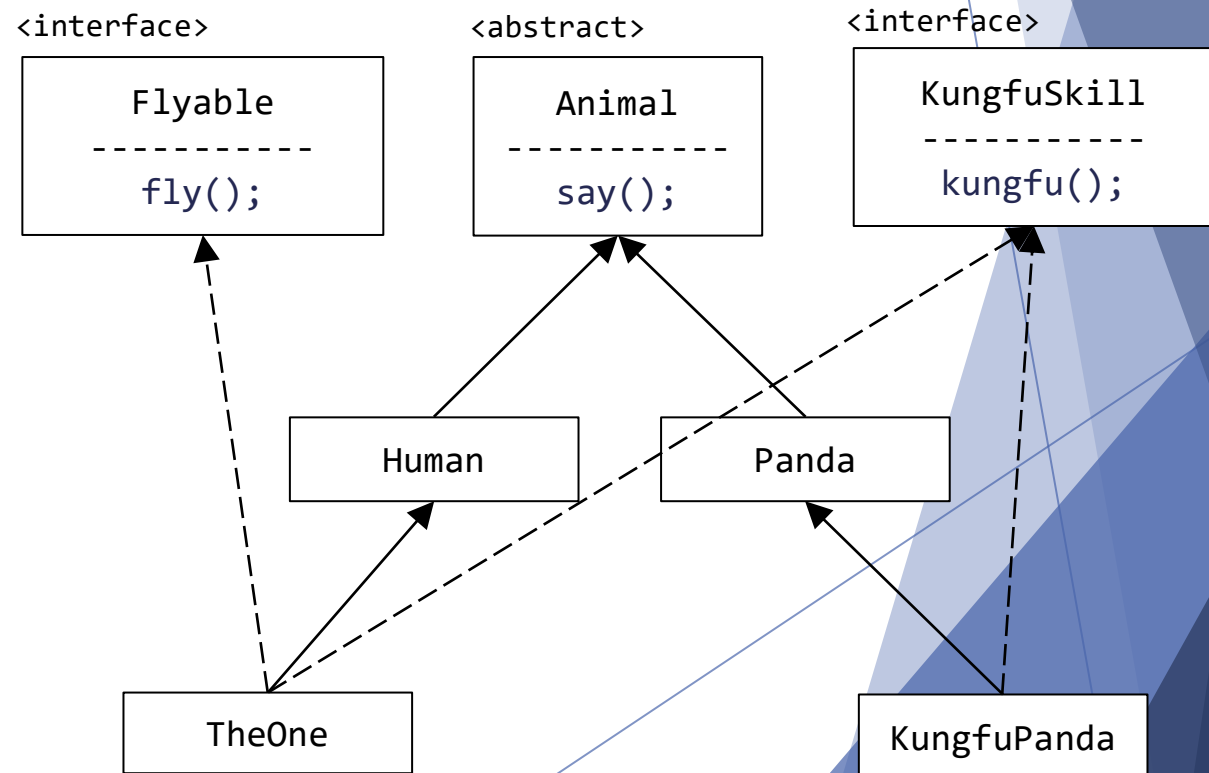
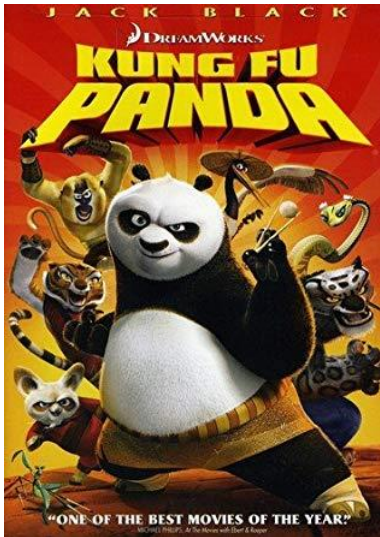
## ▶ 실체 메서드 (Concrete Method)

- ▶ 메서드의 구현을 포함한 일반적인 메서드를 Concrete Method라 함
- ▶ 추상 메서드 (Abstract Method)의 반대 개념

# Abstract Class and Interface

: 정리

- ▶ 추상 클래스와 인터페이스 모두 규약으로서의 의미가 강하지만
  - ▶ 추상 클래스는 객체를 일반화한 공통 필드와 메서드의 정의에 집중 (종적 확장)
  - ▶ 인터페이스는 자연스러운 상속 관계를 해치지 않으면서도 추가 내용을 규정할 수 있음(횡적 확장)



# OOP Advanded

# 어노테이션

- ▶ 프로그램에게 추가적인 정보(Metadata)를 제공
  - ▶ 컴파일러에게 코드 문법 에러를 체크하도록 지정
  - ▶ IDE에게 빌드나 배치 수행 시 코드를 자동으로 생성할 수 있도록 정보 제공
  - ▶ 실행(런타임)시 특정 기능을 실행하도록 정보 제공

- ▶ 어노테이션 타입 정의

```
public @interface AnnotationName {  
}
```

- ▶ 어노테이션의 사용

```
@AnnotationName
```



# 어노테이션

- ▶ 어노테이션은 **Element**를 멤버로 가질 수 있다.
  - ▶ 엘리먼트 타입: 기본형(primitives), String, enum, Class, 이들의 배열 타입
  - ▶ 엘리먼트는 메서드가 아니긴 하지만, 이름 뒤에 ()를 명시해야 한다.
  - ▶ 기본 엘리먼트는 **value()**로 선언한다

타입 엘리먼트 이름() [**default** 값];

- ▶ [예] Annotation의 선언 예제

```
public @interface AnnotationName {  
    String value(); //기본 엘리먼트  
    int element1() default 10; //기본 값이 있는 엘리먼트  
    int element2(); //기본 값이 없는 엘리먼트  
}
```

# 어노테이션

## ▶ 어노테이션의 적용 대상

- ▶ 어노테이션 적용 대상은 `java.lang.annotation.ElementType` 열거 상수로 정의

ElementType 열거 상수	적용 대상
TYPE	클래스, 인터페이스, 열거타입
ANNOTATION_TYPE	어노테이션
FIELD	필드
CONSTRUCTOR	생성자
METHOD	메서드
LOCAL_VARIABLE	로컬 변수
PACKAGE	패키지

# 어노테이션

- ▶ 어노테이션 적용 대상을 지정할 때에는 **@Target** 어노테이션을 사용
  - ▶ **@Target** 어노테이션의 기본 **value**는 **ElementType**의 배열을 값으로 가짐

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
```

- ▶ 위 **@Target**이 부여된 어노테이션은 어디에 적용할 수 있습니까?
- ▶ 어노테이션 유지 정책(**RetentionPolicy**)
  - ▶ 어노테이션을 어느 범위까지 유지할 것인지를 지정해야 한다
  - ▶ 유지 정책은 **java.lang.annotation.RetentionPolicy** 열거 상수로 정의

RetentionPolicy	설명
SOURCE	소스에만 유지, 바이트 코드 파일에는 정보 유지 안됨
CLASS	바이트 코드 파일까지 정보 유지. 리플렉션 불가
RUNTIME	바이트 코드 파일까지 정보 유지. 리플렉션 가능

# 어노테이션 : 리플렉션

- ▶ 리플렉션(Reflection) : 런타임 시 클래스에 메타 정보를 얻는 기능
  - ▶ 런타임 시에 어노테이션 관련 정보를 얻으려면 유지 정책을 **RUNTIME**으로 설정해야
- ▶ 런타임 유지 정책 설정 예

```
@Retention(RetentionPolicy.RUNTIME)
```

- ▶ 런타임시 어노테이션 정보 확인
  - ▶ 리플렉션을 이용하여 어노테이션의 적용 여부와 엘리먼트 값을 읽고 그에 상응하는 적절한 처리를 추가할 수 있다.

메서드명	리턴 타입	설명
getFields()	Field[]	필드 정보를 배열로 반환
getConstructors()	Constructor[]	생성자 정보를 배열로 반환
getDeclaredMethods()	Method[]	메서드 정보를 배열로 반환

# 어노테이션 : 리플렉션

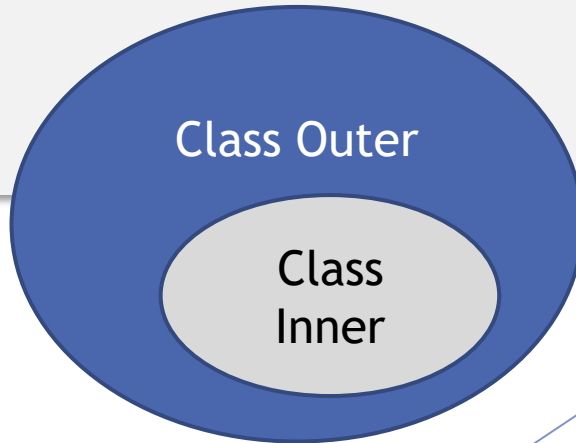
- ▶ Class, Field, Constructor, Method의 어노테이션 관련 주요 메서드
  - ▶ 어노테이션 관련 정보를 획득할 수 있음

메서드명	설명
<code>boolean isAnnotationPresent(Class annotationClass)</code>	지정한 어노테이션이 적용되었는지의 여부 확인
<code>Annotation getAnnotation(Class annotationClass)</code>	지정한 어노테이션이 적용되었으면 해당 어노테이션 반환 그렇지 않으면 <code>null</code>
<code>Annotation[] getAnnotations()</code>	상위 클래스에 선언된 어노테이션 호환 적용된 모든 어노테이션을 반환. 적용된 어노테이션이 없으면 길이가 0인 배열 리턴.
<code>Annotation[] getDeclaredAnnotations()</code>	직접 적용된 모든 어노테이션을 반환. 상위 클래스에 선언된 어노테이션은 반환하지 않음.

# 중첩 클래스

- ▶ 클래스가 여러 클래스와 관계를 맺지 않고, 특정 클래스와만 관계를 맺을 때는 관계 클래스를 클래스 내부에 선언하는 것이 좋다(Nested Class)
  - ▶ 중첩 클래스 사용의 이점
    - ▶ 두 클래스의 멤버들을 서로 쉽게 접근할 수 있다
    - ▶ 외부에는 불필요한 관계 클래스를 감춤으로써 코드의 복잡도를 줄여준다

```
class Outer {  
    // 외부 클래스  
    class Inner {  
        // 내부 클래스  
    }  
}
```



# 중첩 클래스

## ▶ 중첩 클래스 종류와 특징

분류		선언 위치	설명
멤버 클래스	인스턴스 멤버 클래스	<pre>class A {     class B { } }</pre>	<ul style="list-style-type: none"><li>- 외부 클래스의 멤버 변수 선언 위치에 선언</li><li>- 내부에 정적 필드나 메서드는 선언할 수 없다</li><li>- class B는 A 객체를 생성해야만 사용할 수 있다</li></ul>
	정적 멤버 클래스	<pre>class A {     static class B { } }</pre>	<ul style="list-style-type: none"><li>- 외부 클래스의 멤버 변수 선언 위치에 선언한다.</li><li>- 내부에 모든 종류의 필드와 메서드를 선언할 수 있다.</li><li>- class B는 A 클래스로 바로 접근할 수 있다.</li></ul>
로컬 클래스		<pre>class A {     void method() {         class B { }     } }</pre>	<ul style="list-style-type: none"><li>- 외부 클래스의 메서드 또는 초기화 블록 안에 선언한다.</li><li>- 로컬 클래스는 접근제한자 및 <b>static</b>을 붙일 수 없다<ul style="list-style-type: none"><li>- 메서드 내부에서만 사용되므로 접근을 제한할 필요가 없기 때문</li></ul></li><li>- class B는 method() 가 실행될 때만 사용할 수 있다.</li></ul>

## ▶ 중첩 클래스의 선언:

- ▶ 선언 위치에 따라 동일 위치의 변수와 동일한 유효범위와 접근성을 갖는다.

# 중첩 클래스

## ▶ 중첩 클래스 종류와 특징

분류		선언 위치	설명
멤버 클래스	인스턴스 멤버 클래스	<pre>class A {     class B { } }</pre>	<ul style="list-style-type: none"><li>- 외부 클래스의 멤버 변수 선언 위치에 선언</li><li>- 내부에 정적 필드나 메서드는 선언할 수 없다</li><li>- class B는 A 객체를 생성해야만 사용할 수 있다</li></ul>
	정적 멤버 클래스	<pre>class A {     static class B { } }</pre>	<ul style="list-style-type: none"><li>- 외부 클래스의 멤버 변수 선언 위치에 선언한다.</li><li>- 내부에 모든 종류의 필드와 메서드를 선언할 수 있다.</li><li>- class B는 A 클래스로 바로 접근할 수 있다.</li></ul>
로컬 클래스		<pre>class A {     void method() {         class B { }     } }</pre>	<ul style="list-style-type: none"><li>- 외부 클래스의 메서드 또는 초기화 블록 안에 선언한다.</li><li>- 로컬 클래스는 접근제한자 및 <b>static</b>을 붙일 수 없다<ul style="list-style-type: none"><li>- 메서드 내부에서만 사용되므로 접근을 제한할 필요가 없기 때문</li></ul></li><li>- class B는 method() 가 실행될 때만 사용할 수 있다.</li></ul>

## ▶ 중첩 클래스의 선언:

- ▶ 선언 위치에 따라 동일 위치의 변수와 동일한 유효범위와 접근성을 갖는다.



# 중첩 클래스

- ▶ 중첩 클래스에서 바깥 클래스 참조 얻기
  - ▶ 중첩 클래스에서 **this** 키워드는 중첩 클래스의 객체 참조
  - ▶ 중첩 클래스 내부에서 바깥쪽 클래스의 객체 참조를 얻으려면
    - ▶ 바깥클래스.this.필드명
    - ▶ 바깥클래스.this.메서드명()

```
class Outer {  
    // 외부 클래스  
    String field = "Outer Field";  
    class Nested {  
        // 중첩 클래스  
        String field = "Nested Field";  
        System.out.println(this.field);  
        System.out.println(Outer.this.field);  
    }  
}
```

# 익명 클래스

- ▶ 이름 없는(Anonymous) 클래스
  - ▶ 클래스 선언과 동시에 단 한번만 사용 가능
  - ▶ 생성자를 선언할 수 없다.
  - ▶ 단독으로 선언될 수 없고, 한 개의 클래스를 상속 받거나, 하나의 인터페이스만을 구현한다.

```
슈퍼클래스 변수 = new 슈퍼클래스() {  
    // 클래스 선언  
} // 상속에 의한 익명 클래스
```

```
인터페이스 변수 = new 인터페이스() {  
    // 클래스 선언  
} // 구현에 의한 익명 클래스
```

# Java Programming

Exception Handling

예외 처리

# Exception Handling

## ▶ 예외 (Exception)

- ▶ 프로그램이 실행되는 동안 발생할 수 있는 비정상적인 상태
- ▶ 컴파일시의 에러가 아닌 실행시의 에러를 예외라 함

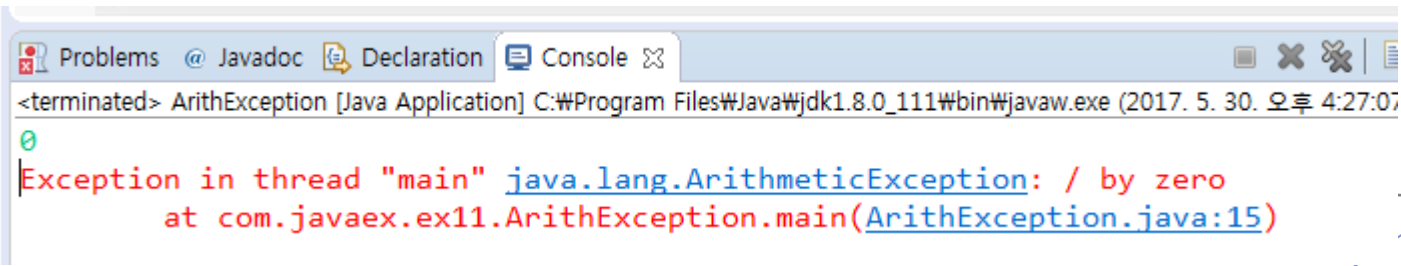
## ▶ 자바의 예외 처리

- ▶ **Exception** 클래스 정의
- ▶ 기본적인 예외는 자바에 미리 정의된 예외를 통해 처리 가능
- ▶ 사용자가 필요한 예외를 직접 정의할 수 있음
- ▶ 예상되는 예외는 미리 처리해주면 비정상적인 프로그램의 종료를 피할 수 있음
- ▶ 예외처리는 프로그램의 신뢰도를 높여줌

# Exception Handling

: Example

```
public class ArithException {  
  
    public static void main(String[] args) {  
  
        double result;  
        int num;  
  
        Scanner sc = new Scanner(System.in);  
        num = sc.nextInt();  
  
        result = 100/num; // java.lang.ArithmeticException 발생  
  
        System.out.println(result); // 예외 발생으로 수행되지 않음  
        sc.close();  
  
    }  
}
```



The screenshot shows an IDE console window with the following text:

```
<terminated> ArithException [Java Application] C:\Program Files\Java\jdk1.8.0_111\bin\javaw.exe (2017. 5. 30. 오후 4:27:07)  
0  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at com.javaex.ex11.ArithException.main(ArithException.java:15)
```

The console window has tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, showing the error message. The error message is in red text, indicating a runtime exception. The exception is a `java.lang.ArithmeticException: / by zero` occurring in the `main` method of `ArithException.java` at line 15.

# Exception Handling

: try ~ catch ~ finally

```
0  
try {  
    // 예외가 발생할 가능성이 있는 실행문 1  
} catch ( 처리할 예외 타입 선언 ) { 2  
    // 예외 처리문  
} finally { 3  
    // 예외 발생 여부와 상관없이 무조건 실행되는 문장 ( 생략가능 )  
} 4
```

- ▶ try 블록에서 예외가 발생했을 경우 : 0 -> 1 -> 2 -> 3 -> 4
- ▶ try 블록에서 예외가 발생하지 않았을 경우 : 0 -> 1 -> 3 -> 4

# Exception Handling

: Exception Class

## ▶ 주요 예외 클래스

예외 클래스	예외 발생 경우
ArithmeticException	어떤 수를 0으로 나눌 때
NullPointerException	null 객체를 참조할 때
ClassCastException	변환할 수 없는 타입으로 객체를 변환 할 때
ArrayIndexOutOfBoundsException	배열을 참조하는 인덱스가 잘못된 경우
NumberFormatException	문자열이 나타내는 숫자와 일치하지 않은 타입의 숫자로 변환한 경우
IOException	입출력 동작 실패, *인터럽트 발생할 경우

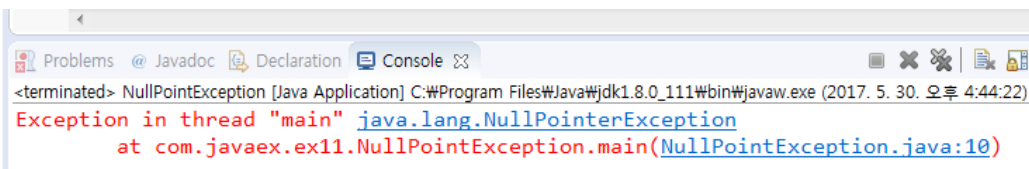
# 연습문제

: 예외 처리

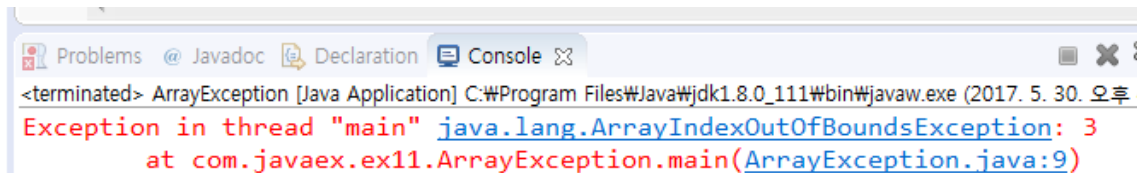
ArithmeticException을 처리했던 방식과 마찬가지로  
ArrayException과 NullPointerException을 처리해보자

```
public class ArrayExceptionEx {  
  
    public static void main(String[] args) {  
  
        int[] intArray = new int[]{3,6,9};  
  
        System.out.println(intArray[3]);  
  
    }  
}
```

```
public class NullPointerExceptionEx {  
  
    public static void main(String[] args) {  
  
        String str = new String("hello");  
        str = null;  
  
        System.out.println(str.toString());  
  
    }  
}
```



```
Problems @ Javadoc Declaration Console  
<terminated> NullPointerException [Java Application] C:\Program Files\Java\jdk1.8.0_111\bin\javaw.exe (2017. 5. 30. 오후 4:44:22)  
Exception in thread "main" java.lang.NullPointerException  
    at com.javaex.ex11.NullPointerException.main(NullPointerException.java:10)
```



```
Problems @ Javadoc Declaration Console  
<terminated> ArrayException [Java Application] C:\Program Files\Java\jdk1.8.0_111\bin\javaw.exe (2017. 5. 30. 오후 4:44:22)  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3  
    at com.javaex.ex11.ArrayException.main(ArrayException.java:9)
```



# Exception Handling

## : Exception Class

- ▶ 예외 (Exception)의 구분
  - ▶ **Checked Exception** : 컴파일할 때 확인되는 예외 -> 반드시 예외 처리가 필요함
  - ▶ **Unchecked Exception** : 실행 시점에서 확인되는 예외 -> 예외 처리 없어도 컴파일 됨
- ▶ 예외 처리가 필요한 시점
  - ▶ 파일을 다루는 경우  
파일이 없거나 다른 프로세스에 의해 사용중인 경우 예외 발생
  - ▶ 입출력을 다루는 경우  
이미 닫힌 입출력 스트림에 대해 작업하려 할 경우 예외 발생
  - ▶ 네트워크를 이용한 데이터 통신  
서버나 클라이언트 한 쪽에서 응답이 없는 경우  
네트워크 상태가 좋지 않아 정해진 시간 동안 데이터를 받지 못하는 경우

# Exception Handling

: 강제 예외 발생

- ▶ 메서드 정의 시 예외 처리
  - ▶ 해당 메서드를 호출하는 메서드에서 예외를 처리하도록 명시한다
  - ▶ **throws** 키워드를 사용하여 예외의 종류를 적어준다

ThrowsExepApp.java

```
public class ThrowsExceptApp{  
    public static void main( String[] args ) {  
  
        ThrowsExcept except= new ThrowsExcept();  
        // IO exception발생  
        except.executeExcept();  
  
    }  
}
```

ThrowsExep.java

```
public class ThrowsExcept {  
    public void executeExcept() throws IOException {  
  
        System.out.println( “강제예외발생” );  
        throw new IOException(); //강제로 예외 발생  
  
    }  
}
```