**컴퓨터프로그래밍**

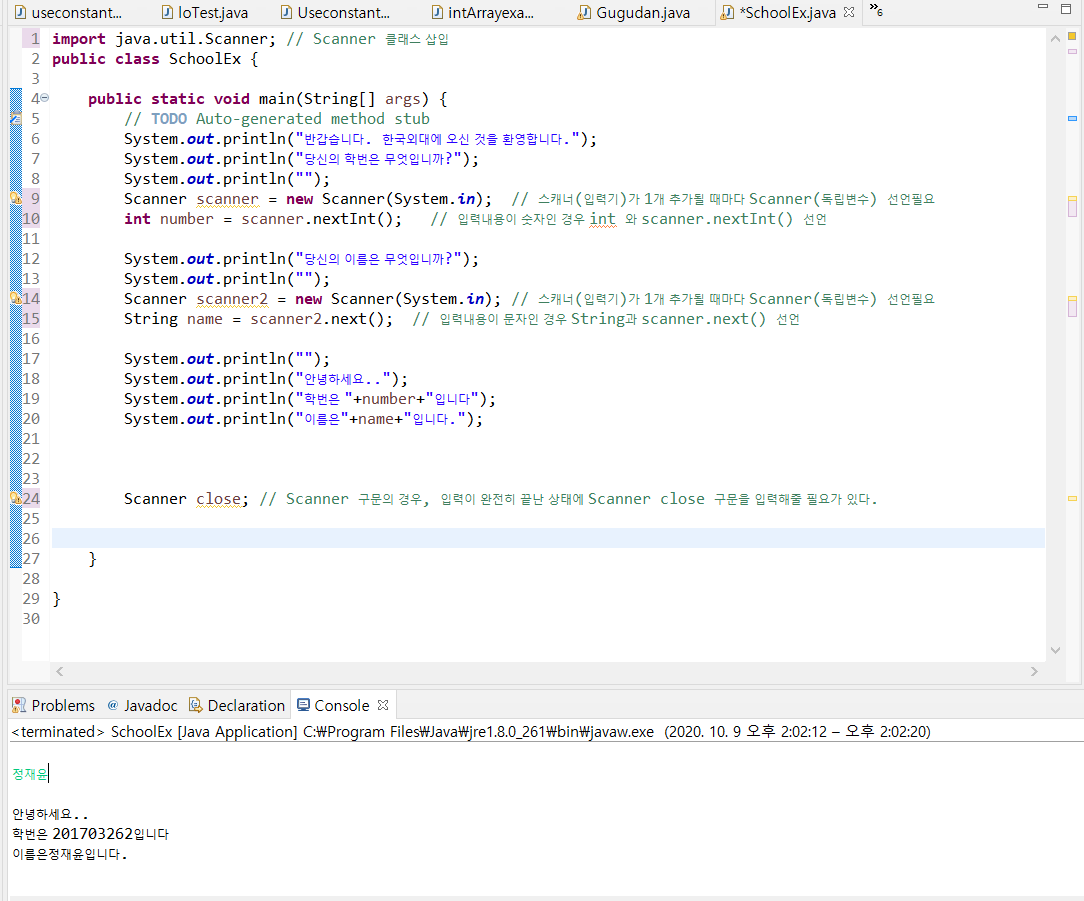
**담당교수님: 이범서 교수**

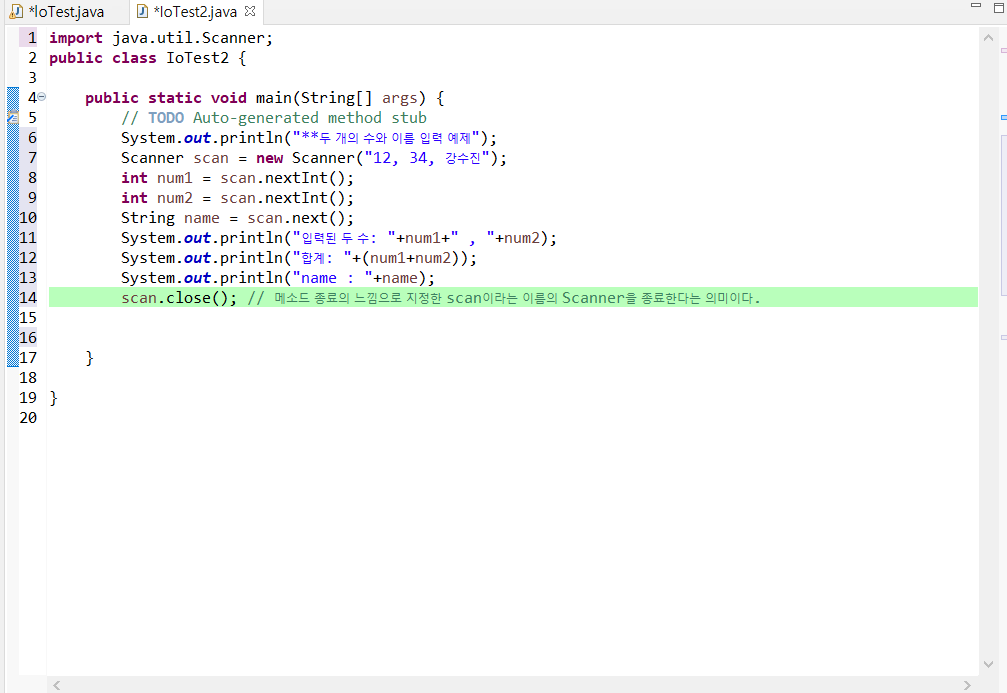
[**ypbums@naver.com**](mailto:ypbums@naver.com)

**http://www.dohr.kr/photo/13**

**System.out.println() 구문**

**Scanner 활용 구문**

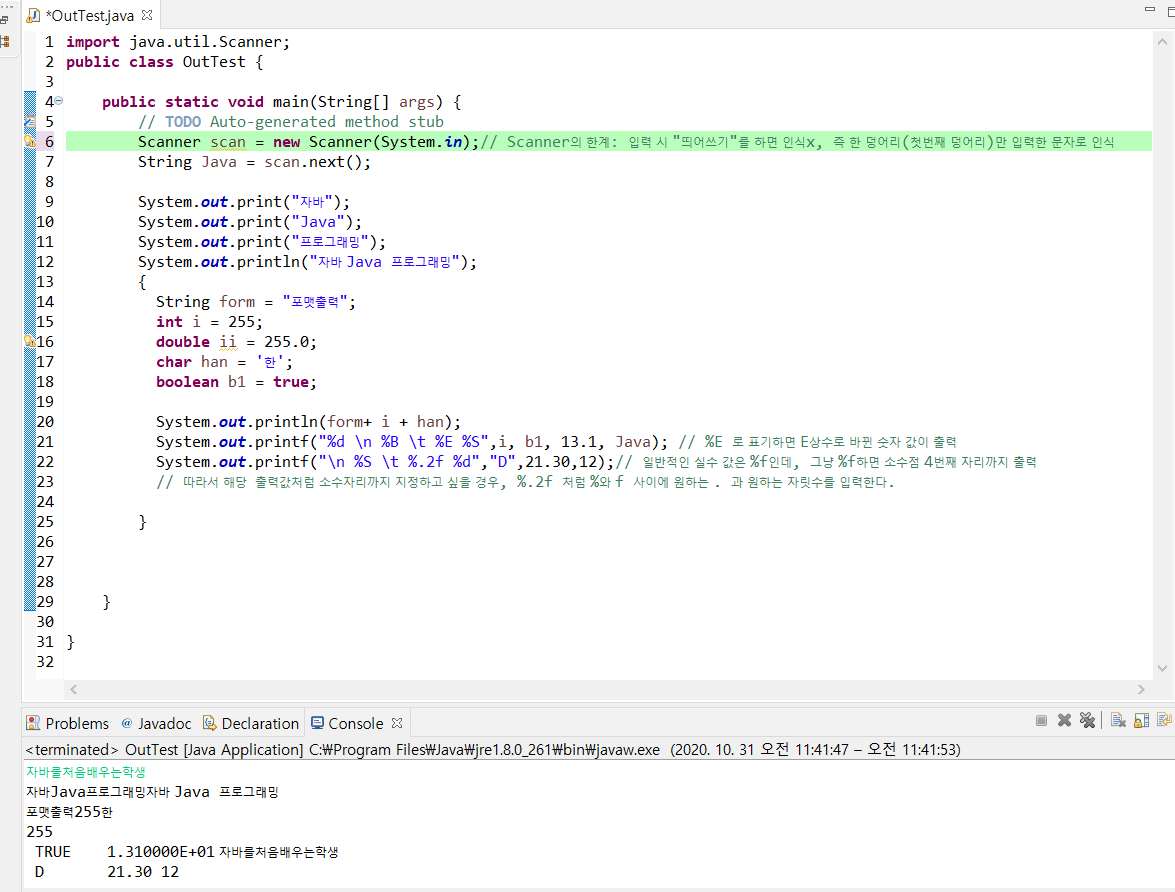


****

**이때 Scanner 종료 시, 단순히 Scanner.close가 아니라 [메소드]를 종료한다는 의미라고 생각하는 것이 바람직하다.**

**다시 말해, 메인 클래스에서 Scanner을 선언할 때  지정한 Scanner의 고유명칭(위 사진 상으로는 “scan”) + .close 로 써야한다,**

**Scanner를 클래스 안에서 한번만 사용하면 굳이 close라고 구문입력할 필요가 없지만 여러 개의 Scanner를 이용하게 되면 해당 Scanner의 기능적 이용이 종료되면 반드시 .close해야 한다.**

****

**Scanner의 한계 :**

**입력 시 "띄어쓰기"를 하면 인식x, 즉 한 덩어리(첫번째 덩어리)만 입력한 문자로 인식한다.**

**다시 말해 띄어쓰기를 한 완벽한 문장을 인식하지 못한다.( “ “로 묶어도 인식x)**

**기본구문**

*Import java.util.Scanner;*

*Scanner 맴버변수명 = new scan(system.in);*

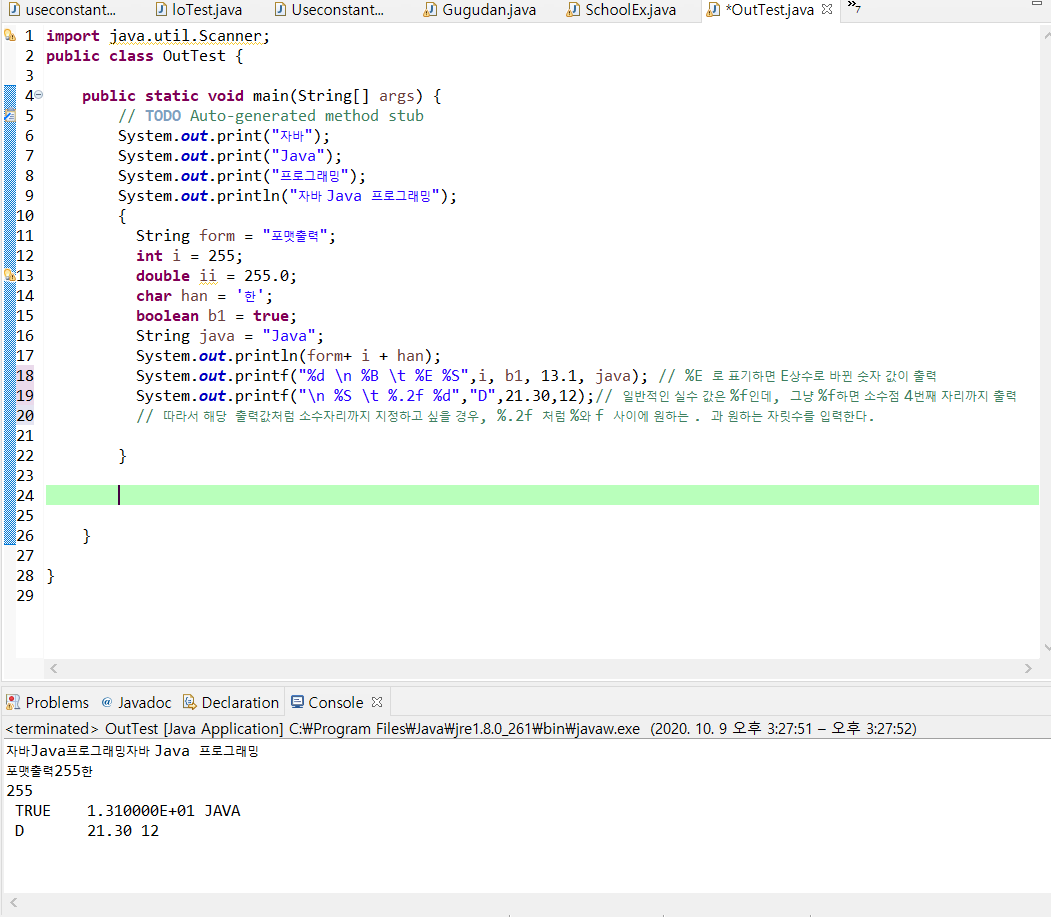
*데이터형태 변수선언 = 맴버변수명.next();* String타입인 경우

*= 맴버변수명.next데이터형태();* 그외 타입인 경우,

첫글자만 대문자(데이터형태)

*맴버변수명.close;*

**printf 구문 활용 방법**



**printf 활용 구문 기본 유형 정리:**

정수= d% %3d 라고 입력하면 정수의 양 옆에 한 칸씩 띄고 값을 입력할 수 있다.

문자= s% S%

단일문자= %c %C

실수= %f %e %E

논리형= %b %B

한 줄 띄어서= \n

한 칸 띄어서= \t

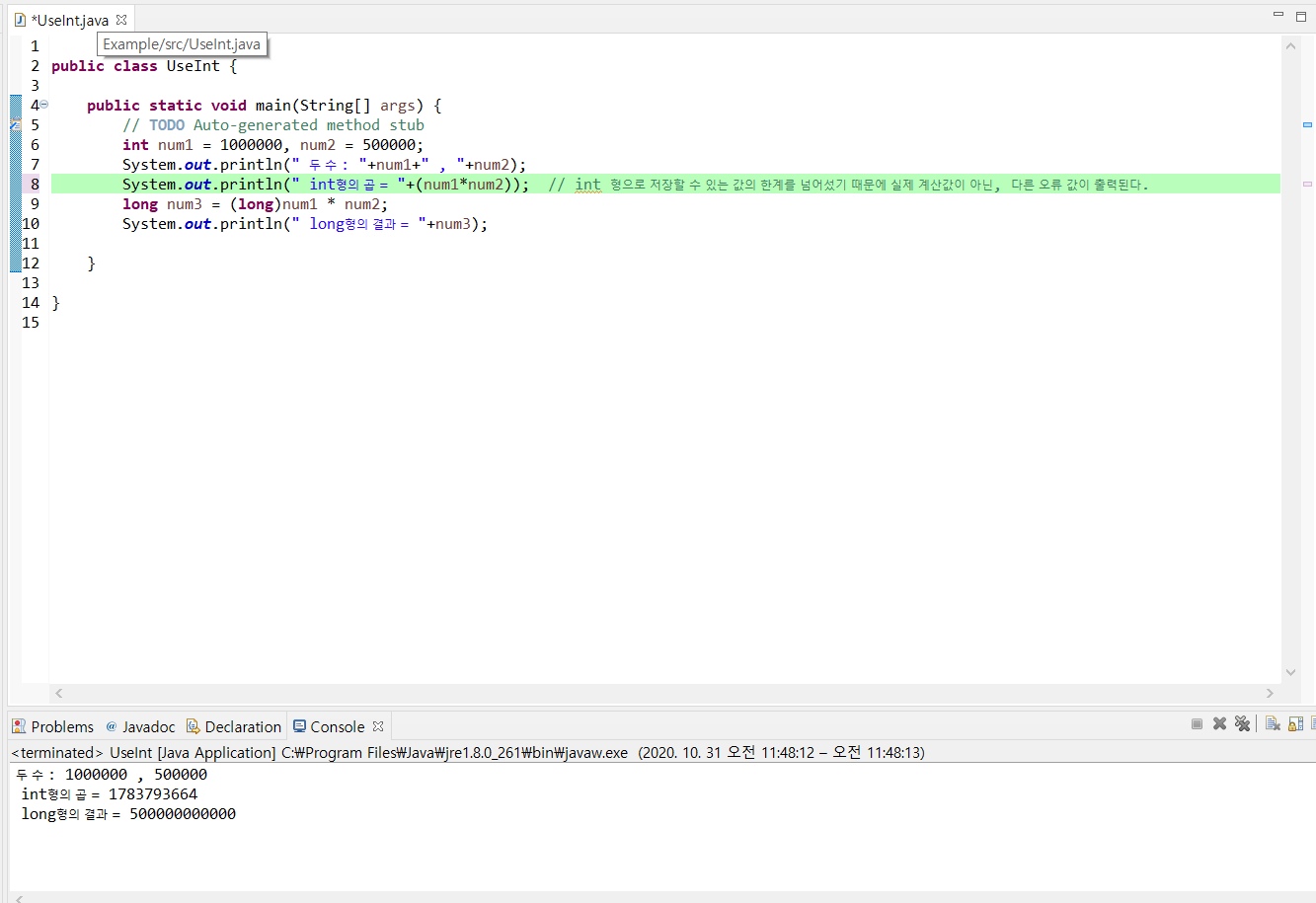
**정수형 각 값 별 크기 지정:**

byte 8비트 -128~127

short 16비트 -32768~32767

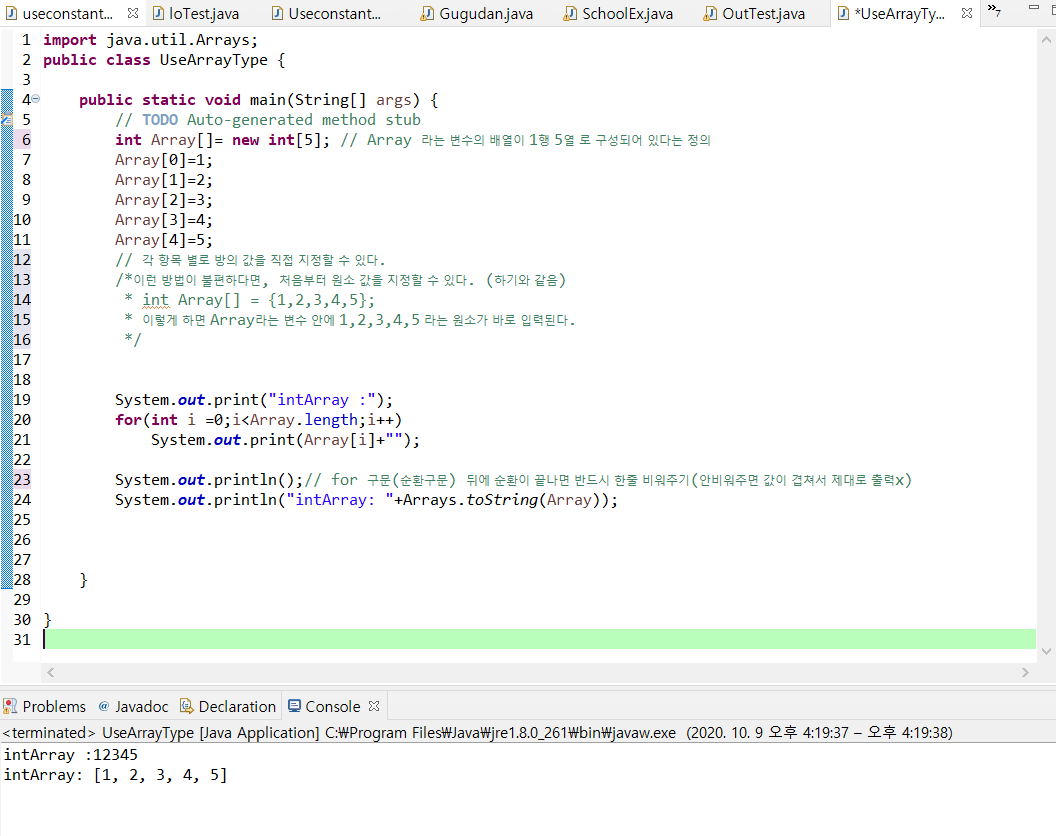
int 32비트 -2,147,483,648 ~ 2,147,483,647

long 64비트 -9223372036854775808~ 9223372036854775807



위의 콘솔창처럼 int 형식의 데이터에 답을 수 있는 수의 범위를 초과한 값이 나올 경우, 정확한 실제의 값이 출력되지 못한다. 따라서 적합한 값을 담을 수 있는 long형식의 데이터 값을 지정해주어야 한다.

Array 배열 함수



**배열 함수 불러오기:**

Import java.util.Arrays;

* 사실 이거 왜 해야 하는 지 모르겠다.. 배열함수 불러오기 설정 안하고,

바로 데이터값유형 [] = new 데이터값유형 [] 해도 배열 생성가능...

Ex) int [] = new int [5] 등...

**배열(단순 배열 자체) 함수 선언 방법 2가지 유형:**

**1. 직접 원소 입력**

변수종류 이름[] = { }; ex) String Name = {정재윤, 배민우, 고대곤, 이경재};

**2. 변수의 행/열 설정 후 차후의 원소입력:**

변수종류 이름[] = new 변수종류[열의 수]; ex) String Name[] = new String[3];

그 다음에 각 열/행 별 원소 값 지정

Ex)

Name[0]=”정재윤”;

Name[1]=”배민우”;

Name[2]=”이경재”;

배열 변수 내의 모든 원소를 [ ] 안에 출력하기:

*Arrays.toString(배열 변수 명);*

활용:

*System.out.println(“집합 내 사람들 이름: “+Arrays.toString(Name));*

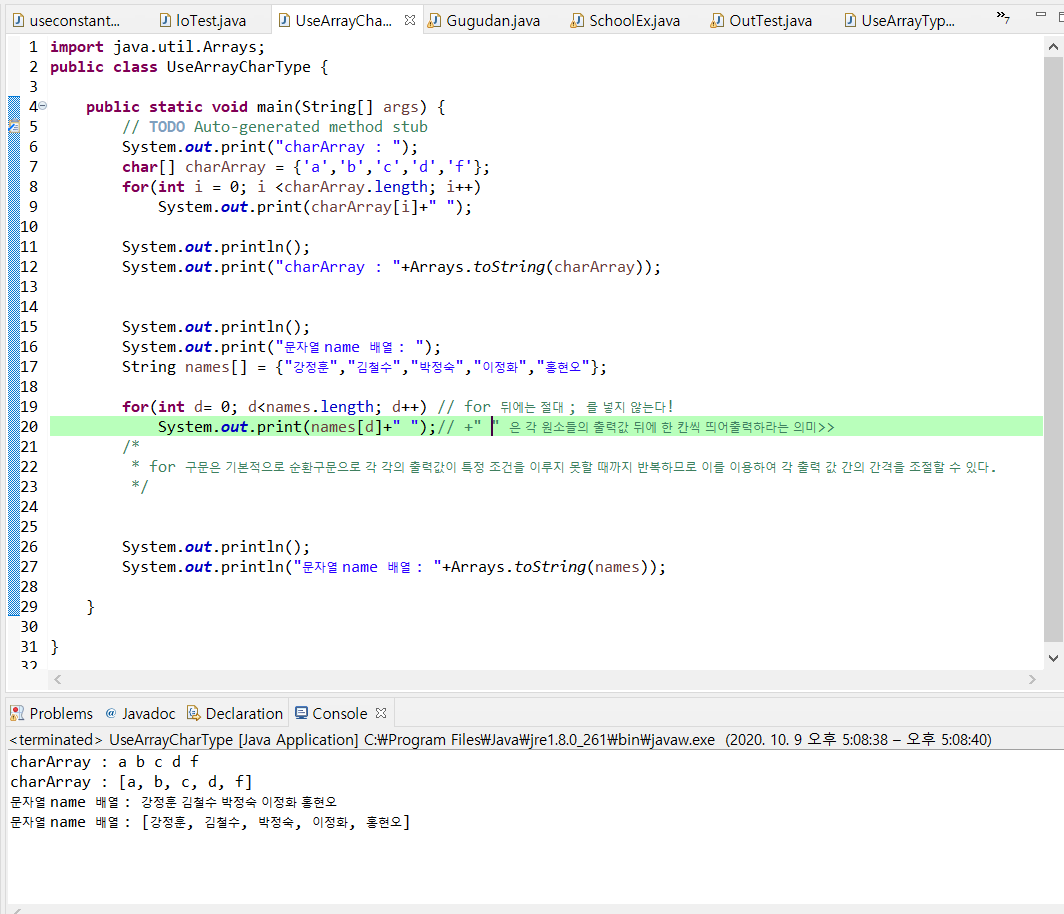
배열 활용2 length 필드 이용:

배열이름.length >> 배열의 내부 원소들의 수를 출력

For 구문: 순환참조

형식 :

for(초기식 ; 조건식; 증감식);



for함수 사용 시 주의 사항: 포함관계 설정에 주의하기



내부에 새로운 순환 정의를 내릴 경우, 반드시 { } 내부 포함 관계 지정을 해야 한다.

만약 포함되는 구문에 대해 { } 처리 안하면 오류 발생

**<2차원 배열>**





행 값을 먼저 지정하고 열 값이 차후에 지정되는 형식을 가진 툴로

위의 식처럼 프로그래밍 된 경우:

i=0이면, a[0]= 0+3 =3이므로 for구문에서 j<3.length가 된다. 따라서 i=0일때 j=0이므로 a[i][j] = 0,1,2의 값을 갖는다.

i=1이면, a[1]= 1+3 =4이므로 for구문에서 j<4.length가 된다. 따라서 i=1일때 j=1이므로 a[i][j] = 1,2,3,4의 값을 갖는다.



배열, for 구문처럼 체계와 순서가 중요한 구문이 있을 때는 반드시 { } 포함관계에 주의!

**객체와 클래스:**

1. public 만약 public을 명시x 이면 동일한 패키지 내에서만 이용가능

여러 클래스 중 반드시 하나에는 public명시 필요

*main(String[] args){ } =* 자바파일(소스 파일)명 인 클래스에 반드시 public 필요

1. final

서브 클래스 생성

1. abstract 추상 클래스

실행할 명령문 x

일반 변수선언과 동일하게 객체 선언

배열과 동일하게 객체와 메소드를 선언

**포함 관계:**

패키지 > 자바파일 > 클래스 > 객체 > 메소드 > (매개)변수

**static 자격자:**

한 클래스에 하나만 만들어지는 정적 멤버 변수 (각 객체마다 새롭게 멤버 변수 생성x)

다시 말해 내부의 변수가 변화하지 않는다.(처음 변수의 유형 값 그대로)

**static + final =** 변수 값을 아예 수정 불가능

**void :** return 값이 없는 경우, void로 선언

만약 메소드 정의할 때 String, int, double, char 등 특정한 데이터 형태를 선언했다면 반드시 return을 입력해주어야 한다.

**this :** 메소드에서 멤버변수를 선언할 때, 사용하는 키워드

다시 말해,

*class SayMyName{* SayMyName이라는 클래스 선언

*String Me;* String Me라는 매개변수 선언

*String Me() {* String Me라는 매개변수를 호출하면 불러낼 값을 설정

*return Me; }* return은 각 값(메소드, 변수)의 마지막 단계, 종료한다는 의미

*void Me(String Me) {*  Me라는 메소드에 String Me 값을 대입한다는 의미

*this.Me=Me;*

*}*

* 이럴 경우 SayMyName이라는 클래스를 호출하면 내부에 포함된 Me 메소드와 Me 매개변수를 이용가능

따라서, 다른 메인 클래스에서

SayMyName WhoAreYou = new SayMyName();

WhoAreYou.Me(Me에 넣을 String 데이터의 값);

WhoAreYou로 정의한 SayMyName클래스의 Me메소드의 값을 정의한다는 의미

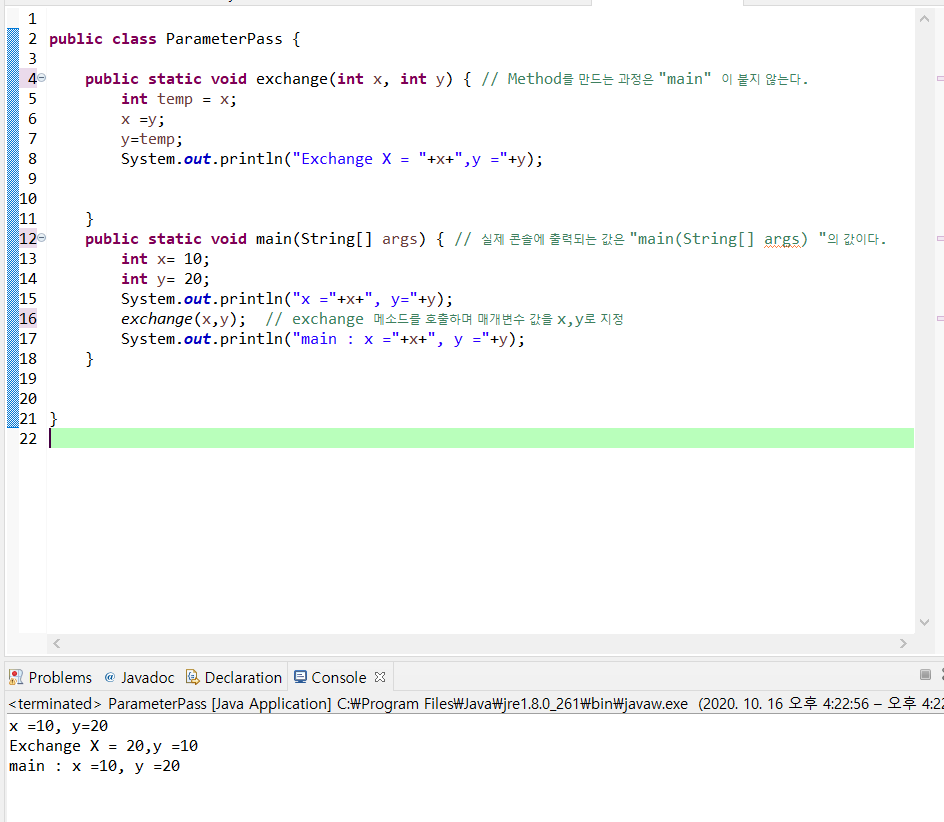
**Method 활용**

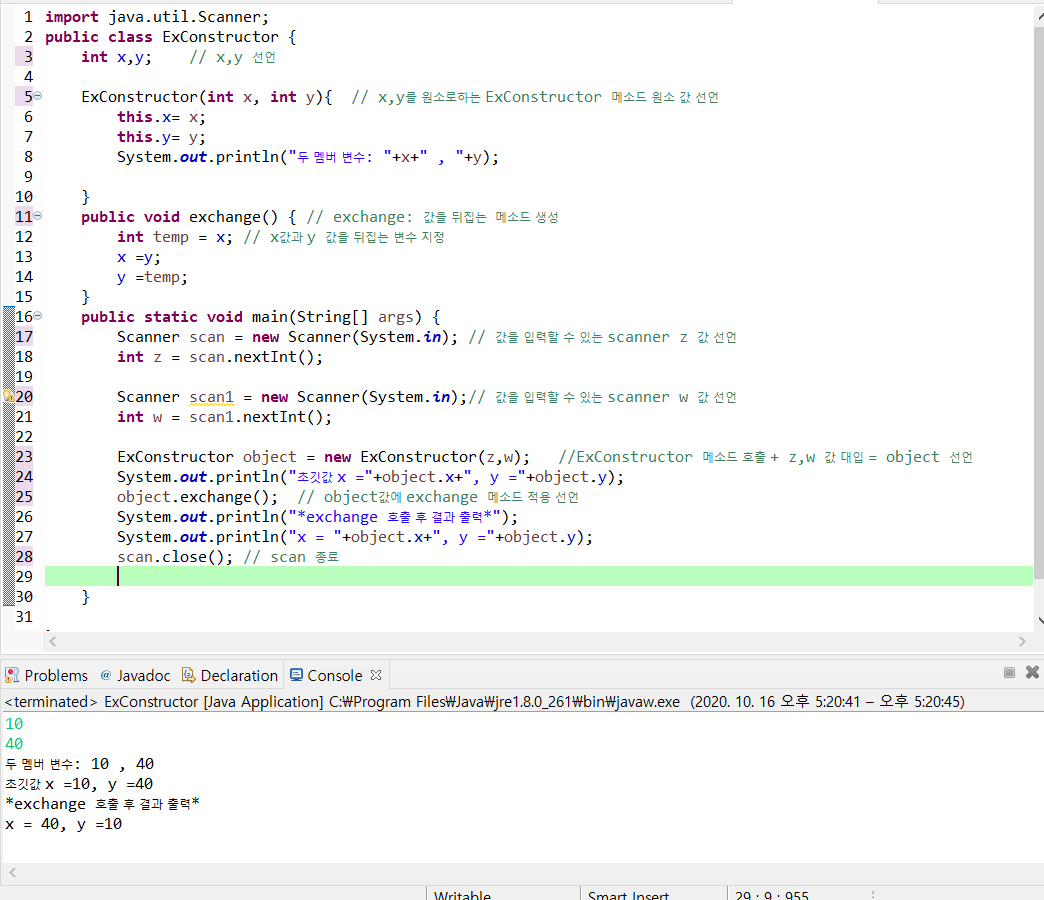
“메소드 생성과 삽입”의 차이와 실제 활용성

메소드 자체 생성은 main(String[] args); 없이

public static void 처럼 앞에 공개여부/ 변수특성/ 값의 종류 만 입력한 뒤 생성한다.

이후 실제 결과값(콘솔창)에 출력하고 싶은 메소드의 경우 main(String[] args) { }를 추가 한다.





미지정 변수의 값을 동일하게 입력할 경우

this 생성자를 이용한다.

ExConstuctor의 값의 유형을 입력 할 때 int x, int y라고 입력하였고,

각각의 int x 와 int y 의 실제 값도 x, y로 지정해야 하기 때문에 각각의 값이 중복되어 혼란이 발생하지 않도록 this x =x ; this y= y; 로 매개변수의 앞에는 this를 붙인다.

실제 값을 콘솔창에 출력하는 메소드(main 메소드)에 scanner을 실행

(이전에 메소드 실행할 때 해도 무방)

다만 메소드 생성 시에 scanner를 활용하게 되면 메소드 자체에 scanner 기능이 내장되어 있는 형태로 이용 가능

Object라는 메소드 함수의 메소드 내 객체 선언 후,

해당 object 내부의 원소를 지칭할 때는 [object.원소]의 형태로 입력하면 object라는 객체 내에 “원소”값에 해당하는 값을 출력하게 된다.

또한 기존에 생성한 메소드를 활용할 때, [적용할 대상.메소드명] object.ExConstructor 처럼 입력하게 되면 main 메소드 안의 object 객체에 ExConstructor가 적용된다.

**메소드 정의와 호출의 차이**

**메소드 정의:**

*public class MethoDemo{*

*public static void numbering(){*

*int j[]=new int[5];*

*for(int i=0;i<j.length;i++)*

*System.out.println(j[i]);*

*}*

*}*

**메소드 호출:**

*Public static void main(String[] args){*

*Numbering();*

만약 메소드 내에 String x , String y 값처럼 부정값을 정의하면 메소드 호출 시

*Numbering(한글 , 영어);*

처럼 해당 데이터 값에 맞는 값을 넣을 수 있다.

**값 서로 바꾸기**

x=y 라고 바로 해버리면 오류 발생

따라서 중간의 환원(거쳐갈 값)을 생성한다.

*int AA = x;*

*x = y;*

*y = AA;* 이럴 경우 AA를 호출하면 x와 y값이 뒤집힌다.

**값 바꾸기의 활용**

(동일한 메소드 내의 변수 값바꾸기)

*public void exchange(NewClass r1){*

*int temp = r1.x;*

*r1.x=r1.y;*

*r1.y=temp;*

**배열을 활용한 메소드 정의 및 활용**

*public class ArrayPassEx{*

*void replace(int b[]) {*

*for(int i=0; i<b.length; i++)*

*b[i] = b[i]+10;*

*}*

b라는 숫자 값을 가진 1차 배열을 생성 [] 로 비워뒀기 때문에 클래스, 메소드 호출을 통해 불러와서 변수 값 입력 가능

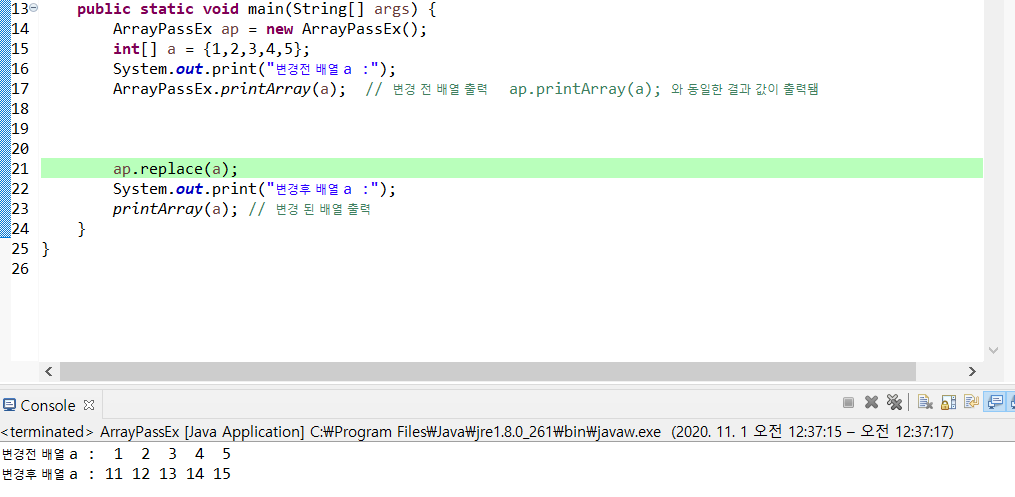
*static void printArray(int b[]) {*

*for(int i =0; i<b.length; i++)*

*System.out.printf(“%3d”,b[i]);*

*}*

B배열의 값을 각 원소별 한 칸씩 띄어서 출력 (각 원소별 값을 출력하는 메소드)



ap.replace를 통해 값을 각 원소 값을 +10씩 한다.

**생성자 이용하기(Constructor)**



**생성자**: 클래스와 동일한 이름을 가진 메소드의 형태로 해당 클래스의 각 항목의 값 형태를 지정해주는 역할을 수행한다.

다른 클래스에서 메소드를 불러와서 사용할 때,

1. 생성자 호출
2. 메소드 호출

의 순서가 가장 기본적인 순서이다.

**배열(메소드)를 호출하여 값을 지정할 때 주의 사항:**

배열 (메소드) + 배열 명= new + 해당배열이 가지고 있을 값 지정 한뒤,

각 배열(메소드)의 내부 변수를 선언할 때,

배열의 명이 아니라, 각 **내부 변수의 데이터 타입을 선언**한다.

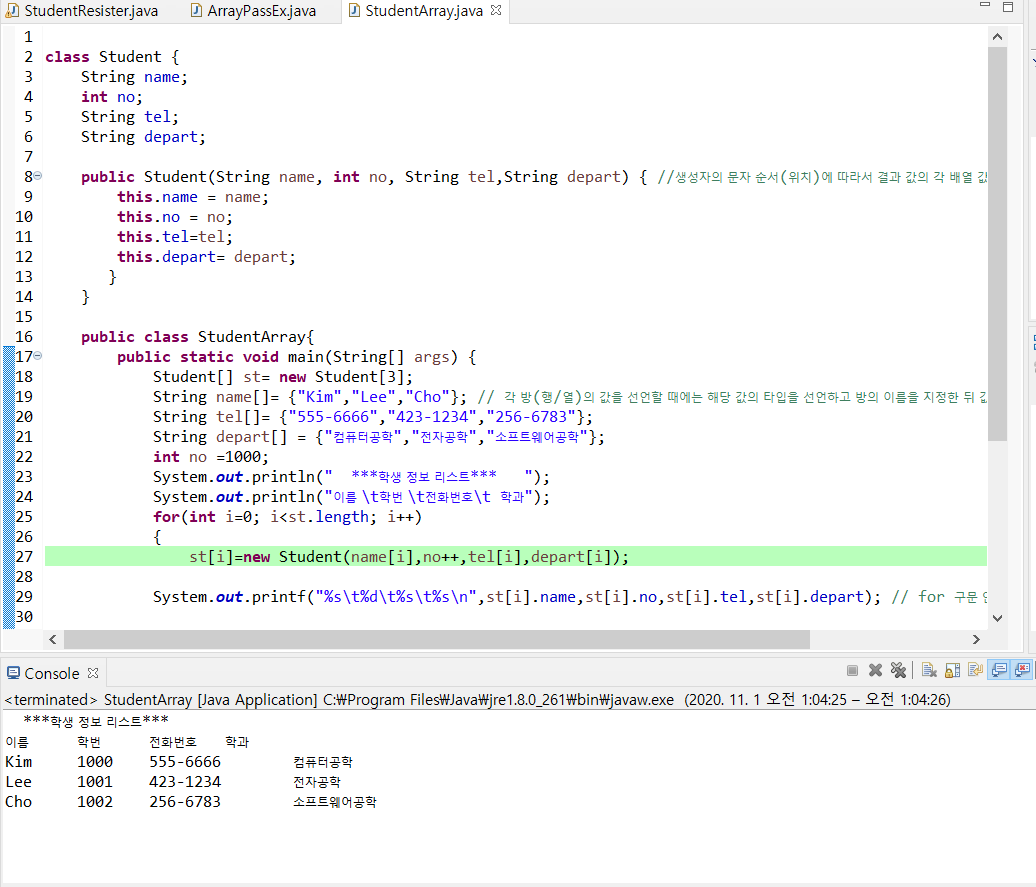
**중첩 클래스**

동일한 클래스를 하나의 파일 하에 여러 개 호출 가능하지만,

1. 매개변수의 갯수
2. 매개변수의 데이터 형태

이 두가지 형태가 달라야한다.

**“1씩 증가”를 뜻하는 ++의 위치에 따른 차이 발생:**

****

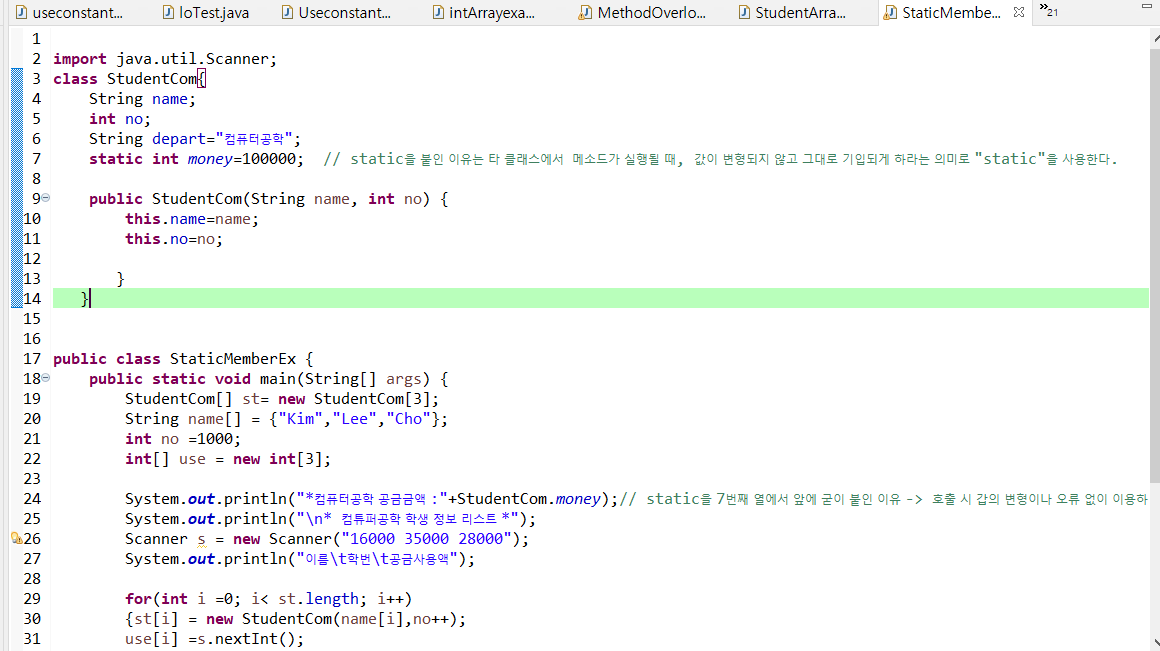
**++이 변수 앞에 위치 : 문장 실행 전 1씩 추가**

**++이 변수 뒤에 위치 : 문장 실행 후 1씩 추가**

**에서 은 한칸 탭씩 칸을 띄운다는 의미**

**한 프로젝트 안에 두개의 클래스 이용하기**

(내부 클래스와 상위 클래스)

**첫번째 클래스 정의 시 “static”을 넣는 이유:**

**Static = class 소속의 메소드로서 인스턴스 내에서도 동일한 값을 갖는다는 의미**

**(class 에 직접 접근하여 실행)**

**만약 static 없이 바로 메소드를 선언한다면 인스턴스를 반드시 선언한 뒤에 메소드를 이용할 수 있다.**

**Ex:**

**Class First{**

**Static void Sum(int b){**

**}**

**}**

**을 새로운 class 에 호출하여 이용할 경우**

**→ First.Sum( b); 를 하여 바로 호출 가능**

**Class second{**

**void Sum(int b){**

**}**

**}**

**을 새로운 class 에 호출하여 이용할 경우**

**→ Second sec = new Second();**

**Sec.Sum(b);**

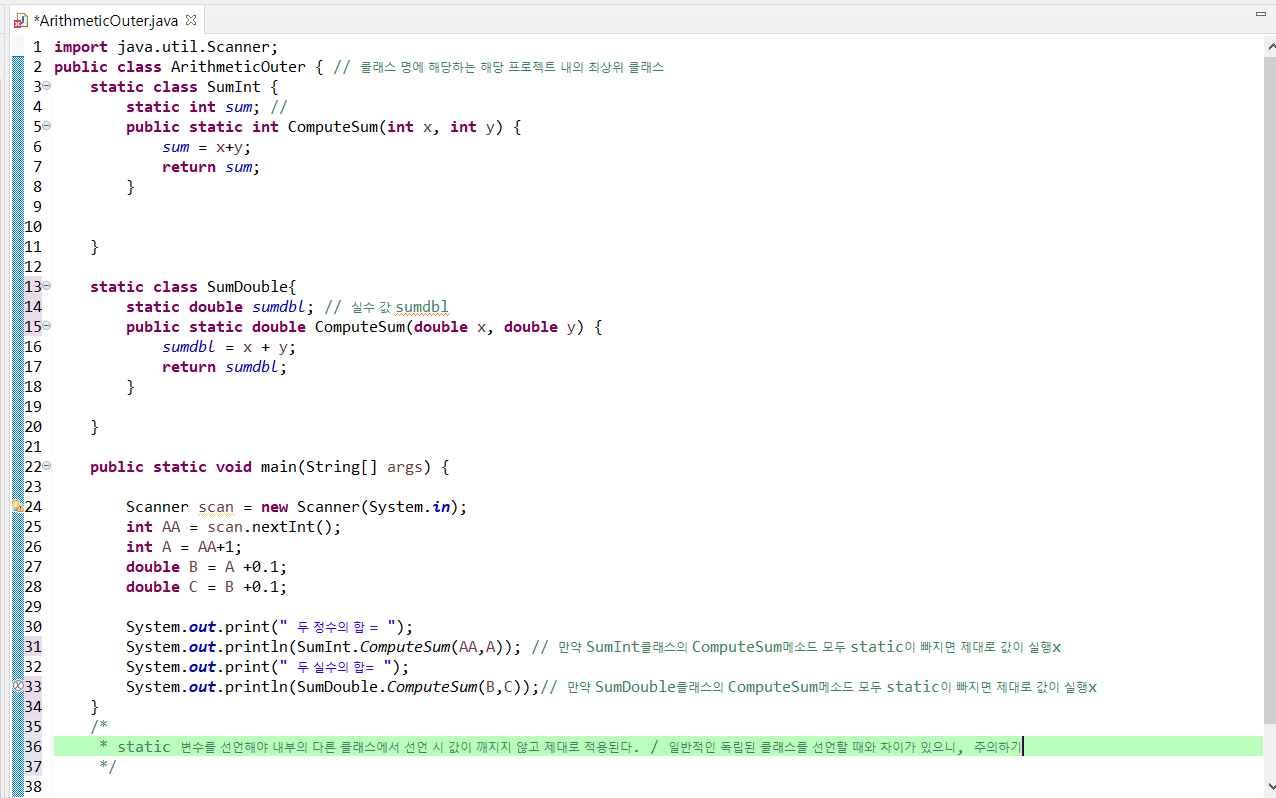
**처럼 앞 구간에 반드시 instance를 선언해주어야 한다.**

다른 클래스에서 해당 클래스의 메소드를 불러올 때, 값이 변형되지 않도록 해당 변수의 값을 고정하는 역할



값 출력 설정 시, 그 외 이용 시 해당 변수가 “배열” 내에 포함되는 지 아니면 독립된 별개의 변수인지 정확히 구분 이 필요하다.

**내부 클래스 정의**



내부 클래스 선언 시, 일반적인 클래스, 메소드 선언과 달리 반드시 static(정적변수) 선언을 우선적으로 해주어야 내부 클래스에서 외부클래스 호출할 때 값이 깨지지 않고 올바르게 적용된다.

(static이 단순히 값을 고정한다는 의미x

이유: 값을 단순히 고정하는 목적이었다면, 변수인 int x, int y, double x, double y 라는 변수로 각클래스 내의 메소드를 정의할 필요가 없지 않겠나? 대신 정형화된 숫자(ex 20, 10, 11.1, 20.3 등)을 썼을 것이다.

따라서 **static이란 값의 “ class 의 소속이며 이하의 instance에 형태 변형을 하지 않는다!”** 라는 의미가 맞다.

**정리:**

**인스턴스 변수 : non-static field**

**클래스 변수 : static field**

**유효범위(scope)**

{ } 를 통해 나뉘는 경우

{ } 밖 즉 클래스, 인스턴스, 메소드 처럼 다양하게 계층이 분류되는 객체지향 프로그램 특성상 { } 밖에 영향을 미칠 수 없다. 따라서 class 내의 모든 메소드에 영향을 미치는 전역변수(global member) 와 지역변수(loccal member)로 구분할 수 있으며, loval member의 경우 해당 local ( { } ) 내에서만 영향을 미친다.

**this의 의미:** 전역변수(global member)를 호출하는 변수

ex)

class second{

int left;

void third(){

int left;

}

}

처럼 두가지 left라는 동일한 이름의 메소드가 존재할 경우,

class second{

int left;

void third(){

int left;

this.left → class second의 left 변수를 의미한다.

}

}

**Condtructor(생성자):**

생성자의 원리 = class 이름 == method 이름

클래스와 메소드의 이름을 같게 하여 클래스를 호출할 경우 자동으로 메소드 내의 값을 지정하는 의미를 갖는다.

자동으로 메소드에 필요한 인자 값을 찾게끔 강제하는 역할

유형1: 인자 지정x

인자가 지정되어 있지 않은 생성자

Class second{

Void second(){

}

}

의 형태를 가지고 있으며 second 를 호출 할 경우 java에서 자동으로 임의의 생성자의 인자를 지정해준다. ( 임의의 값 = 아무 값이 없다)

Ex) 클래스 호출 시

second ss = new second();

유형2: 인자가 지정되어 있는 경우

Class second{

Void second(int a, int b){

}

}

인경우에는 second를 호출 할 때 반드시 해당 class에 상응하는 인자값을 함께 입력해야 한다.

Ex) 클래스 호출 시

second ss = new second(a , b);

상속 : 재활용성을 극대화하는 객체지향의 특성

기존의 객체 특성을 그대로 유지하면서 추가적인 기능을 새로운 객체에 넣어서 구현하는 방법

형식:

Sub class 이름 extends super class 이름(){

}

이때 하나의 super class 내에 여러 sub class가 파생될 수 있으며 파생된 sub class 를 또다시 상속 받는 sub class 를 만들 수 있다. 이때 sub class1 을 상속 받으면 subclass 의 상위 클래스인 super class의 변수와 메소드도 함께 상속 받는다.

상속 & 생성자

만약 인자가 지정된 생성자를 가진 class 를 상속 받았는데 해당 상위 클래스(super class)의 변수를 가져다가 하위 클래스(subclass)에 이용하고 싶을 경우에는 하위 클래스 내부에 “super(변수, 변수 .. etc ) “ 를 입력하여 super 클래스 내의 변수를 호출한다.

→ 상위 클래스의 변수를 호출한다는 의미를 갖는다.

**Overriding(재정의 : 덮어씌우는 기능) → 가장 최근에 정의한 메소드를 인식한다.**

상위 class의 기존 변수/ 메소드를 변형(재정의)하여 이용하는 방법

상위 크래스에 ‘sum” 메소드가 있는데, 하위클래스가 상위클래스를 상속할 경우 ‘sum’ 이라는 메소드를 하위 클래스에서 재정의하면 하위클래스를 상속받은 클래스에서 sum을 사용할 때, 하위 클래스의 sum 이 사용된다.

**Overriding의 조건:**

1. 메소드의 형식이 동일해야 한다.

2. 메소드의 이름이 같아야 한다.

3. 매개변수가 동일해야 한다.(매개변수의 데이터 형식/ 매개변수의 개수)

동일한 부분의 part(부분)은 “super.메소드명” 으로 선언하여 상위클래스의 메소드를 직접 호출할 수도 있다.

Overloading(동일한 이름의 메소드이지만 다른 형태의 값을 갖는 메소드)

**package** 실험실;

**class** SubOverloading{

**int** a, b, c;

**void** SetOprand(**int** a, **int** b) {

**int** sum = a + b;

System.***out***.println(sum);

}

**void** SetOprand(**int** a, **int** b, **int** c) {

**int** sum = a+b+ c;

System.***out***.println(sum);

}

}

**public** **class** overloading {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

SubOverloading SO = **new** SubOverloading();

SO.SetOprand(10,20);

SO.SetOprand(10,20,30);

}

}

위와 같은 코드 형식을 가진다.

**Overloading의 규칙:**

입력한 매개변수의 데이터 형식 / 매개변수의 갯수에 부합하는 메소드를 불러낸다.

**메소드의 이름 (same), data return type (same) , 매개변수(diffrent)**

단 이때 주의 사항:

**package** 실험실;

**class** SubOverloading{

**int** a, b, c;

**void** SetOprand(**int** a, **int** b) {

**int** sum = a + b;

System.***out***.println(sum);

}

**int** SetOprand(**int** a, **int** b) {

**int** sum = a+b+ c;

System.***out***.println(sum);

}

}

**public** **class** overloading {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

SubOverloading SO = **new** SubOverloading();

SO.SetOprand(10,20);

SO.SetOprand(10,20,30);

}

}

이처럼 앞의 데이터 형식만 바뀌게 되면 오류 발생:

이유 → 반환 값이 어떻게 될지 int 와 void라는 데이터 형식만으로는 확인이 불가능하여 에러 발생

즉 return 값의 형식이 같아야 한다.

Overriding vs Overloading

Overriding → 덮어씌우는 기능

Overloading → 동일한 이름의 메소드 but 다른 형태의 메소드

Ex)

**void** A() {

}

**void** A(**int** a) {

}

**void** A(String a) {

}

**void** A(**int** a, **int** b) {

}

이와 같은 형태는 문제가 없다.(overloading 의 예시)

**접근제어자:**

**Private : 보안성의 문제로 인해 파생된 접근제어자**

동일한 클래스 내에서만 사용가능하며, 외부에서 사용할 경우, 간접적으로 호출가능하다.

(직접 호출 불가능)

Ex)

**package** 실험실;

**class** A{

**private** String z () {

**return** "public void z()";

}

**public** String x() {

**return** z();

}

}

**public** **class** Private {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

A AA = **new** A();

AA.x();

}

}

위의 예제처럼 외부의 클래스에서 호출하여 사용할 경우, 다른 접근제어자를 가진 메소드를 통해서 만 호출가능하다.

**package** 실험실;

**class** A{

**private** **int** a,b;

**private** **int** z (**int** a, **int** b) {

**this**.a =a;

**this**.b=b;

**int** sum = a+b;

**return** sum;

}

**public** **int** x(**int** a, **int** b) {

**return** z(a,b);

}

}

**public** **class** Private {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

A AA = **new** A();

**int** sum = AA.x(10, 20);

System.***out***.println(sum);

}

}

위 예제처럼 private 접근제어자를 int 매개변수에 지정해주어도, 간접호출을 통해 호출하여 사용이 가능하다.

**접근제어자 종류와 그 특성:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **같은 클래스** | **같은 패키지** | **다른패키지 서브클래스** | **무관(어디든 가능)** |
| **public** | o | o | o | o |
| **protected** | o | o | o | x |
| **default** | o | o | x | x |
| **private** | o | x | x | x |

부가설명:

1) 같은 패키지 : instance , inherite 모두 가능

2) 다른패키지, 서브클래스 : 타 패키지라해도 상속(inherite)받는 경우 public / protected 사용가능

**접근제어자 와 (클래스) 메소드의 차이**

|  |  |
| --- | --- |
| **접근제어자** | **(클래스) 메소드** |
| private | static |
| protected | final |
| default | abstract |
| public |  |

접근제어자와 (클래스)메소드 둘다 매개변수(field) , 메소드(method), 인스턴스(instance), 클래스(class)에 모두 사용할 수 있다.

**Abstract (추상)클래스/ 메소드:**

직접적으로 사용(호출) 불가능하다. (private을 외부 클래스에서 사용할 때 간접적으로 사용한 것과 유사)

상속하여 사용하도록 강제된 형태

특징:

1) abstract가 사용된 메소드에는 본문이 없다.

따라서 { } 도 없이 바로 ( ) ; 로 구문이 끝나게된다.

2) 클래스 내부에 1개의 메소드라도 abstract가 선언되어 있는 경우에 class 앞에도 반드시 abstract를 선언해야 한다.

**package** 실험실;

**abstract** **class** Example{

**public** **abstract** **void** Set(); // 이처럼 메소드 내부에 body가 없다.

}

**class** Test **extends** Example{

**int** a,b;

**public** **void** Set() { // overriding 시에 반드시 abstract method의 접근제어자와 동일한 제어자 설정 필요

**int** b =1;

System.***out***.println(b);

}

}

**public** **class** AbstractClassEX {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Test TT = **new** Test();

TT.Set();

}

}

**Final : 상속/ 변경을 금지하는 규제(수정 불가능)**

**Interface : 인터페이스(규제 中 일부로서 제작자와 사용자의 원활한 소통을 위한 구조)**

Interface를 참조한 객체는 interface가 선언한 메소드만을 가져야 한다.(일종의 지침서/설명서)

특징:

1) 반드시 인터페이스에 선언되어 있는 메소드의 몸체(body)를 overriding 해야 한다.

2) interface에 선언되어 있는 모든 메소드는 반드시 “public”이어야 한다.

Interface 내의 메소드는 본문이 없다. { } 없이 일반 메소드를 실행하는 것처럼 ( ) ; 로 끝

3) 1개의 class가 여러개의 interface를 구현할 수 있다.

4) 구현할 때는 implements 를 사용한다.

5) 인터페이스 끼리도 상속(구현)이 가능하다.

6) 1개의 인터페이스 파일에는 1개의 public inerface만 위치할 수 있다.

클래스처럼 여러 개의 public 인터페이스 선언 불가능

Publlic이 아닌, default interface는 여러개 선언 가능

Ex)

**1. 인터페이스 파일**

**package** 실험실;

**public** **interface** I3 {

**public** **void** x(); → { } 가 없다. 일반 메소드처럼 실행

}

**interface** I4 **extends** I3{

**public** **void** y(); → { } 가 없다. 일반 메소드처럼 실행

}

**2. 클래스 파일**

**package** 실험실;

**public** **class** InterfaceEx **implements** I4 {

**public** **void** x() {

}

**public** **void** y() {

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

}

}

위의 예제 같은 경우 I4가 I3를 구현하고 있으므로, I4를 class InterfaceEx에서 구현 할 때 I4와 I3 에 사용된 인너페이스의 “메소드”들을 모두 필수적으로 선언해주어야 한다.

**Abstract vs Interface**

**Abstract :** 일반 class와 동일한 특성을 가지며 그저 하위 class에 상속되는 기능을 가지고 있다.

또한 1개의 class 내에 여러 메소드, 매개변수가 있을 때 1개만 abtract으로 설정할 수 있기때문에 모든 멤버가 abtract처럼 본문이 없을 필요가 없다.(본문이 있는 메소드도 존재)

**Interface** : 구체적인 내용을 가진 메소드를 포함할 수 없다. 모든 메소드는 public 으로 구성되어 있으며, 정말 구현(implements)될 class의 설명서 역할을 수행한다.

**Polymorphism(다형성) : 클래스, 인터페이스가 서로 상속되는 관계에 따라 다양한 방법으로 동작**

Overriding의 원리를 활용한 기능으로서 같은 이름/ 다른 동작이 핵심 기능이다.

**사용하는 특징:**

**1) 상속 관계**

**package** 실험실;

**class** AL {

**void** x(String x) {

System.***out***.println(x);

}

}

**class** B **extends** AL {

**void** x(String x) {

System.***out***.println("당신의 국적 : "+x);

}

String y(String y) {

**return** y;

}

}

**public** **class** PolymorphismEx {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

AL obj = **new** B();

obj.x("한국");

}

}

**2) 구현 관계**

Interface:

**package** 실험실;

**public** **interface** I3 {

**public** **void** x();

}

**interface** I4 **extends** I3{

**public** **void** y();

}

Class:

**package** 실험실;

**class** Test2 **implements** I4{

**public** **void** x() {

**int** a =1;

System.***out***.println(a);

}

**public** **void** y() {

**int** a =2;

System.***out***.println(a);

}

}

**public** **class** PolymorphismEx2 {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

I4 obj = **new** Test2();

obj.y();

obj.x();

I3 obj2 = **new** Test2();

obj2.x();

}

}

이처럼, (1) 상속 / (2) 구현 에서 다형성을 가지게 된 근본적인 이유:

상위 클래스/ 인터페이스에서 지정한 내용들이 추가적으로 작업을 거치면서 변형된 내용을 적용하되, 처음(상위 클래스/ 인터페이스)에서 지정한 내용들만을 가지고 작업을 해야할 경우를 위해 이런 “다형성”이 자바에 존재한다.

Exception(예외) : 일반적이지 않은 상황에서 발생하는 특수한 무제

Try : 예외를 발생사킬 위험이 있는 코드를 { } 안에 넣기

Catch(예외클래스 인스턴스) : 예외(오류)의 발생 시 오류 발생 순간부터 try를 대신하여 실행

Finally : 예외여부와 관계없이 실행되는 로직

사용형태

**package** 실험실;

**class** Sum{

**void** Run(**int** a, **int** b) {

**int** run = a +b ;

System.***out***.println(run);

}

}

**public** **class** ExceptionEx {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Sum AA = **new** Sum();

AA.Run(10, 20);

**int** a = 10;

**int** b = 0;

**int** BB = a/b;

**try** {

System.***out***.println(BB);

}**catch**(Exception e){ → Catch(예외클래스 인스턴스) :

System.***out***.println(e.getMessage());

}**finally** {

System.***out***.println(" 끝 ");

}

}

}

예외처리 메소드:

1) getMessage() : 축약정보 (오류)

2) toString() : 자세한 정보 (오류, 원인)

3) printStackTrace() : 모든 에러사항을 출력(오류, 원인, 어떤 위치의 로직)

Try-catch 를 실행하면 그 뒤의 프로그램이 오류가 발생해도 이어서 실행된다.(이후의 로직을 실행)