**기초 Java 정리**

### Java 언어의 탄생

Java는 제임스 고슬링과 연구원들이 개발한 객체 지향적 프로그래밍 언어이다.

#### Write Once, Run Anywhere

직역을 하면, '한 번 작성하면 어디에서나 실행된다'는 의미로, 자바로 개발된 프로그램은 자바 실행 환경 JRE가 설치된 모든 환경에서 실행이 가능하다는 것을 의미한다.

**1) 자료형**

Java의 자료형은 크게 Primitive type(기본자료형)과 Reference type(참조자료형)으로 구분된다.

**primitive type의 종류:**

short, int, long, float, double, char, boolean, byte 등이 있다.  
short, int, long은 숫자형으로 각각 나타낼 수 있는 숫자의 크기가 다르며, 이는 각 타입에 할당되는 메모리의 크기가 다르기 때문이다.  
순서대로 각 2바이트, 4바이트, 8바이트가 할당 가능하다.  
또한 아래의 코드로 최소 및 최대값을 찍어보며 실제 값을 볼 수 있다.

System.out.println(Short.MAX\_VALUE);

System.out.println(Short.MIN\_VALUE);

**reference type의 종류:**

참조 자료형은 위의 기본 자료형을 제외한 모든 자료형을 말한다. 쉽게 말해 자바의 인스턴스를 가리킬 수 있는 자료형이다. 클래스, 배열, 열거, 인터페이스 등이 존재한다.  
가장 친근한 String 또한 참조 자료형에 속한다.(클래스)

**+) 배열 (Array)**

1) 배열 변수명 선언 (아래의 코드에선 arr이 배열의 변수 명)

int arr[];

2) 배열 생성 (크기 지정도 이 때 해준다.)

arr = new int[5]

3) 초기화 (생성된 배열에 값을 넣어준다.)

배열은 선언과 동시에 크기를 지정받는다, 따라서 고정된 크기를 갖는다.  
하지만 실제로 고정된 크기의 배열만을 쓰기는 쉽지 않기 때문에, 보통 배열보다는 ArrayList라는 Collection을 사용한다.

**2) 조건문**

사전지식: char는 문자열 하나만을 담으며 작은 따옴표로 표현된다.

switch의 소괄호에는 파라미터를 넣어주며 각각의 케이스 별 기능을 구현하면 된다.  
default는 if문의 else와 비슷한 기능으로 생각하면 될 것 같다. 단, 주의해야할 점은 각 케이스마다 break가 없다면 조건을 충족한 케이스 이후에는 판별하지 않고 모두 실행시킨다.

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

String score = sc.next();

switch(score){

case "A":

System.out.println("A등급입니다.");

break;

case "B":

System.out.println("B등급입니다.");

break;

case "C":

System.out.println("C등급입니다.");

break;

default:

System.out.println("D등급 이하 입니다.");

break;

}

}

}

**삼항 연산자**

변수명 = (논리 조건) ? true일 때 실행되는 기능 : false일 때 실행되는 기능의 구조로 사용된다. (변수 지정을 안해주면 오류 발생!)

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int score = sc.nextInt();

// result의 값은 조건에 따라 좌항 혹은 우항으로 결정된다.

String result = ( score < 10 ) ? "10보다 작습니다." : "10보다 큽니다.";

System.out.println(result);

}

}

**3) 반복문**

for 반복문은 자바스크립트와 형태가 비슷하다.

for( int i = 0 ; i < 100; i++){

//이와 같은 형태로 사용한다.

}

**for each 반복문**

파이썬에서 주로 쓰이는 반복문의 형식이다.  
배열, 리스트 등에서 하나 하나의 원소들을 iterate하며 사용하는 방식이다.

public class Main {

public static void main(String[] args) {

// write your code here

String[] days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};

for (String day : days){

System.out.println(day);

}

}

}

while 반복문 : 특정 조건이 충족될 때까지 반복된다.  
중간에 break, continue를 줄 수 있다.

public class Main {

public static void main(String[] args) {

// write your code here

int i = 0;

int sum = 0;

while (i<10){

sum += (i+1);

i++;

}

System.out.println(sum);

}

}

do - while 반복문은 do블록의 마지막에 while문을 붙이는 구조이다.  
디버그를 돌려보니 do의 코드블록이 먼저 실행된 후 while에서 조건을 체크한 뒤 false면 반복, true면 벗어나는 구조다.

public class Main {

public static void main(String[] args) {

// write your code here

int i = 0;

int sum = 0;

do{

sum += (i+1);

i ++;

}while (i<10);

System.out.println(sum);

}

}

**4) 클래스, 인스턴스, 메소드**

**클래스**

표현하고자 하는 대상의 공통 속성을 한 군데에 정의해 놓은 것  
즉, 객체의 속성을 정의한 것이다.  
객체의 속성에는 필드, 메소드, 생성자 등이 있다.  
필드는 클래스 내부 전역에서 사용 가능한 전역 변수를 지칭한다.  
필드(전역변수)는 멤버 변수라고도 한다.  
반면 메소드와 생성자에서 정의된 변수는 지역변수라고 한다.

**인스턴스**

어떠한 클래스로부터 만들어진 객체를 그 클래스의 인스턴스라고 한다.  
아래의 코드에서 Phone은 클래스이고, 메인 함수의 galaxy와 iphone은 인스턴스들이다.

class Phone {

String model;

String color;

int price;

}

public class Main {

public static void main(String[] args) {

Phone galaxy = new Phone();

galaxy.model = "Galaxy10";

galaxy.color = "Black";

galaxy.price = 100;

Phone iphone =new Phone();

iphone.model = "iPhoneX";

iphone.color = "Black";

iphone.price = 200;

System.out.println("철수는 이번에 " + galaxy.model + galaxy.color + " + 색상을 " + galaxy.price + "만원에 샀다.");

System.out.println("영희는 이번에 " + iphone.model + iphone.color + " + 색상을 " + iphone.price + "만원에 샀다.");

}

}

**+) 클래스와 인스턴스 비유**

따라서 클래스는 붕어빵을 만들어내는 틀이라고 할 수 있으며, 인스턴스는 붕어빵 틀에 의해 만들어진 붕어빵이라고 할 수 있다.

**메소드**

메소드는 어떠한 작업을 수행하는 코드를 하나로 묶어 놓은 것이다.  
함수라고도 한다.  
메소드 내의 변수는 지역변수로, 메소드 내부에서만 사용할 수 있다.  
코드 컨벤션으로는, 시작은 동사로하되 Camel Case를 사용한다.

//반환타입 메소드이름 (타입 변수명,타입 변수명, ...){

// 수행되어야 할 코드

//}

int add(int x, int y) {

int result = x + y;

return result;

}

**5) 생성자**

**생성자**

인스턴스가 생성될 때 사용되는 인스턴스 초기화 메소드  
즉 new 와 같은 키워드로 해당 클래스의 인스턴스가 새로 생성될 때, 자동으로 호출되는 메소드  
대표적으로 인스턴스의 변수를 초기화하는 용도의 생성자가 있다.  
생성자의 이름은 클래스 명과 같아야하며, 리턴값이 없다는 특징을 지닌다.  
클래스를 선언할 때 따로 선언하지 않아도 매개변수와 내용이 없는 디폴트 생성자가 생성된다.  
+)팁 alt + insert를 누르면 자동으로 생성자, getter, setter, toString등의 코드를 작성할 수 있다.

//클래스이름 (타입 변수명, 타입 변수명, ...){

// 인스턴스 생성 될 때에 수행하여할 코드

// 변수의 초기화 코드

//}

class Phone {

String model;

String color;

int price;

//생성자

Phone(String model, String color, int price) {

this.model = model;

this.color = color;

this.price = price;

}

}

class 에 선언된 변수는 instance 가 생성될 때 값이 초기화(initialize)된다.  
따라서 변수의 선언부나 생성자를 통해서 초기화를 해주지 않는다면, 각 자료형의 디폴트 값이 할당된다.  
아래 코드로 각 자료형의 디폴트 값을 확인할 수 있다.  
단 필드의 마지막에 정의된 참조 자료형은 디폴트 값이 아닌 참조할 값이 없다는 뜻인 null이 된다.

class DefaultValueTest {

byte byteDefaultValue;

int intDefaultValue;

short shortDefaultValue;

long longDefaultValue;

float floatDefaultValue;

double doubleDefaultValue;

boolean booleanDefaultValue;

String referenceDefaultValue;

}

public class Main {

public static void main(String[] args) {

DefaultValueTest defaultValueTest = new DefaultValueTest();

System.out.println("byte default: " + defaultValueTest.byteDefaultValue);

System.out.println("short default: " + defaultValueTest.shortDefaultValue);

System.out.println("int default: " + defaultValueTest.intDefaultValue);

System.out.println("long default: " + defaultValueTest.longDefaultValue);

System.out.println("float default: " + defaultValueTest.floatDefaultValue);

System.out.println("double default: " + defaultValueTest.doubleDefaultValue);

System.out.println("boolean default: " + defaultValueTest.booleanDefaultValue);

System.out.println("reference default: " + defaultValueTest.referenceDefaultValue);

}

}

**6) 상속**

상속이란 기존의 클래스를 재사용하는 방식 중의 하나이다.  
한 번 작성한 코드가 재사용이 필요하다면, 변경사항만 코드로 작성하므로 상대적으로 적은 양의 코드를 작성할 수 있다.  
이렇게 코드를 재사용하면, 코드와 클래스가 많아질수록 관리가 용이하다.

상속을 하게되면  
1. 부모 클래스로에서 정의된 필드와 메소드를 물려 받는다.  
2. 새로운 필드와 메소드를 추가할 수 있다.  
3. 부모 클래스스에서 물려받은 메소드를 수정할 수 있다.

상속은 extends로 사용한다. (하나의 클래스만 상속 가능)

class Animal{}

class Dog extends Animal{}

class Cat extends Animal{}

**super, super() 메소드**

부모 클래스로부터 상속받은 필드나 메소드 및 생성자를 자식 클래스에서 참조하여 사용하고 싶을 때 사용하는 키워드이다.

**this, this() 메소드와 차이점**

this() 메소드가 같은 클래스의 다른 생성자를 호출할 때 사용된다면, super() 메소드는 부모 클래스의 생성자를 호출할 때 사용된다.  
따라서 super()의 매개변수 자리에는 부모 클래스 생성자의 매개변수와 동일하게 들어가야 한다.  
예를 들어 부모 클래스에 기본생성자가 없는 경우 자식 클래스에서 super(); 만을 선언하면 컴파일 에러가 발생한다. (매개변수 자리가 빈 기본 생성자가 부모 클래스에 정의되어 있지 않기 때문이다.)

+) 쉽게 생각하면 this는 현재 scope의 클래스를 지칭한다.

**오버로딩과 오버라이딩**

오버로딩  
한 클래스 내에 동일한 이름의 메소드를 여러개 정의하는 것  
메소드 간 이름이 동일해야한다.  
단, 매개변수의 개수 및 타입이 전부 동일하다면 오버로딩이 아니다.

int add(int x, int y, int z) {

int result = x + y + z;

return result;

}

long add(int a, int b, long c) {

long result = a + b + c;

return result;

}

int add(int a, int b) {

int result = a + b;

return result;

}

오버라이딩  
부모 클래스로부터 상속받은 메소드의 내용을 변경하는 것  
상속받은 메소드를 그대로 사용하기도 하지만, 필요에 의해 변경해야할 경우 오버라이딩을 한다.  
부모 클래스의 메소드와 이름, 매개변수, 반환타입이 동일해야한다.

class Animal {

String name;

String color;

public void cry() {

System.out.println(name + " is crying.");

}

}

class Dog extends Animal {

//생성자

Dog(String name) {

this.name = name;

}

// 오버라이딩

@Override

public void cry() {

System.out.println(name + " is barking!");

}

public void swim() {

System.out.println(name + " is swimming.");

}

}

public class Main {

public static void main(String[] args) {

Animal dog = new Dog("코코");

// Animal 클래스인 dog는 자식 클래스 Dog의 swim() 사용 불가

// 쉽게 말해 부모 클래스는 자식 클래스의 메소드를 사용하지 못한다.

// 하지만 할당 시 오버라이딩한 cry()는 적용되어 is barking이 출력된다.

dog.cry();

}

}

자식(Dog) 객체는 자식(Dog) 타입으로 선언된 변수에도 할당할 수 있고, 부모(Animal) 타입으로 선언된 변수에도 할당할 수 있습니다. 단, 부모(Animal) 타입의 변수로 사용할 때는, 실제 객체를 만들(new) 때 사용한 자식(Dog) 타입에 있는 함수 (여기서는 swim())을 호출할 수 없습니다. 컴파일 에러입니다.

**즉 오버로딩은 기존에 없는 새로운 메소드를 정의하는 것이며 오버라이딩은 상속받은 메소드의 내용을 변경하는 것이다.**

**7) 접근 제어자 (access modifier)**

* **접근 제어자는** 멤버 변수/함수 혹은 클래스에 사용되며 외부에서의 접근을 제한하는 역할을 합니다.

→ private : 같은 클래스 내에서만 접근이 가능합니다

→ default(nothing) : 같은 패키지 내에서만 접근이 가능합니다.

→ protected : 같은 패키지 내에서, 그리고 다른 패키지의 자손클래스에서 접근이 가능합니다.

→ public : 접근 제한이 전혀 없습니다.

접근 제어자로 제어하는 것을 캡슐화 라고 한다. (encapsulation)

객체지향 프로그래밍이란 객체들 간의 상호작용을 코드로 표현하는 것입니다.

* 이때 객체들간의 관계에 따라서 접근 할 수 있는 것과 아닌 것, 권한을 구분할 필요가 생깁니다.
* 클래스 내부에 선언된 데이터의 부적절한 사용으로부터 보호하기 위해서!
  + 이런 것을 캡슐화(encapsulation)라고 합니다.
* 접근 제어자는 캡슐화가 가능할 수 있도록 돕는 도구입니다.

**8) 추상 클래스**

* **추상클래스란**?  
  추상클래스는 추상메소드를 선언할 수 있는 클래스를 의미합니다. 또한 추상클래스는 클래스와는 다르게 상속받는 클래스 없이 그 자체로 인스턴스를 생성할 수는 없습니다.
* 먼저 **추상메소드에** 대해서 알아보겠습니다.
* 추상메소드는 설계만 되어있으며 수행되는 코드에 대해서는 작성이 안된 메소드입니다.
* 이처럼, 미완성으로 남겨두는 이유는 상속받는 클래스 마다 반드시 동작이 달라지는 경우에 상속받는 클래스 작성자가 반드시 작성하도록하기 위함입니다. (자식 클래스에서 오버라이딩하여 작성됨)
* 추상 메소드 형식 👉 abstract 리턴타입 메소드이름();

// 추상 클래스 Bird -> 그 자체로 인스턴스 생성 불가함.

abstract class Bird{

private int x,y,z;

void fly(int x, int y, int z){

printLocation();

System.out.println("이동합니다.");

this.x = x;

this.y = y;

if(flyable(z)){

this.z = z;

}else{

System.out.println("그 높이로는 날 수 없습니다.");

}

printLocation();

}

//추상 메서드 flyable() -> 자식 클래스에서 오버라이딩하여 사용

abstract boolean flyable(int z);

public void printLocation(){

System.out.println("현재 위치 {" + x + "," + y + "," + z + ")");

}

}

class Pigeon extends Bird{

// 추상 메서드 오버라이딩

@Override

boolean flyable(int z) {

return z < 10000;

}

}

class Peacock extends Bird{

// 추상 메서드 오버라이딩

@Override

boolean flyable(int z) {

return false;

}

}

public class Main {

public static void main(String[] args) {

Bird pigeon = new Pigeon();

Bird peacock = new Peacock();

System.out.println("---비둘기---");

pigeon.fly(1, 2, 3);

System.out.println("---공작새---");

peacock.fly(1, 2, 3);

System.out.println("---비둘기---");

pigeon.fly(1, 2, 30000);

}

}

**9) 인터페이스**

* **인터페이스는** 객체의 특정 행동의 특징을 정의하는 간단한 문법입니다. 인터페이스는 함수의 특징(method signature)인 접근제어자, 리턴타입, 메소드 이름만을 정의합니다. 함수의 내용은 없습니다. 인터페이스를 구현하는 클래스는 인터페이스에 존재하는 함수의 내용({} 중괄호 안의 내용)을 반드시 구현해야합니다.
* 인터페이스 형식 👉 interface 인터페이스명{ public abstract void 추상메서드명(); } → 인터페이스의 메소드는 추상메소드, static메소드, default 메소드 모두 허용됩니다. (JDK 1.8부터)

interface Flyable{

// 인터페이스는 멤버를 가지지 못하고, 동작(메서드)만 정의할 수 있다.

// 또한 정의한 메서드는 내용을 가질 수 없다

void fly(int x, int y, int z);

}

// 인터페이스는 다중 상속(implements)할 수 있다.

class Pigeon implements Flyable{

private int x, y, z;

@Override

public void fly(int x, int y, int z) {

printLocation();

System.out.println("날아갑니다.");

this.x = x;

this.y = y;

this.z = z;

printLocation();

}

public void printLocation(){

System.out.println("현재 위치 (" + x + ", " + y + ", " + z + ")");

}

}

public class Main {

public static void main(String[] args) {

Flyable pigeon = new Pigeon();

pigeon.fly(1,2,3);

}

}

**+) 추상 클래스와 인터페이스의 차이점**

* 인터페이스

1. 인터페이스의 모든 메서드는 추상메서드이다.
2. 구현하려는 객체의 동작의 명세
3. 다중 상속 가능
4. implements를 이용하여 구현
5. 메소드 시그니처(이름, 파라미터, 리턴 타입)에 대한 선언만 가능
6. 같은 이름의 메서드가 클래스에 따라 다르게 동작하도록 구현되는 것을 말하는 다형성을 지님

* 추상클래스

1. 클래스를 상속받아 이용 및 확장을 위함
2. 다중 상속 불가능 , 단일 상속
3. extends를 이용하여 구현
4. 추상메소드에 대한 구현 가능
5. 추상 클래스의 기능을 재사용, 확장한다는 측면에서 상속성을 지님

참고: <https://myjamong.tistory.com/150>  
추상클래스 사용 시기 : 상속 관계를 쭉 타고 올라갔을때 같은 조상클래스를 상속하는데 기능까지 완변히 똑같은 기능이 필요한 경우

인터페이스 사용 시기 : 상속 관계를 쭉 타고 올라갔을때 다른 조상클래스를 상속하는데 같은 기능이 필요할 경우 인터페이스 사용

**10) 예외 처리**

예외처리란(Exception, Error Handling)

* **예외처리**의 목적
  1. 예외의 발생으로 인한 실행 중인 프로그램의 비정상 종료를 막기 위해서
  2. 개발자에게 알려서 코드를 보완할 수 있도록 하게 위해서
* Throwable 에는 크게 두 종류의 자식 클래스가 있다 (Error & Exception)
* 자바에 미리 정의 되어있는 예외 클래스 들이 있습니다. 기본적으로 이미 있는 것을 사용하시되, 필요한 것으로 표현할 수 없거나 구체적인 목적을 가진 예외를 정의하고 싶다면, Throwable 또는 그 하위에 있는 예외 클래스를 상속받아서 자신만의 예외 클래스를 정의할 수 있습니다
* try-catch(-finally) 형식

public class Main {

public static void main(String[] args) {

int number = 10;

int result;

for(int i = 10; i >= 0; i--){

//예외가 발생할 가능성이 있는 코드

try{

result = number / i;

System.out.println(result);

//예외가 발생했을 시 실행되는 코드 (catch는 여러개 작성 가능)

}catch (Exception e){

System.out.println("Exception 발생: " + e.getMessage());

//예외의 발생여부에 관계없이 항상 수행되어야하는 코드

}finally {

System.out.println("항상 실행되는 finally 구문");

}

}

}

}

* try-with-resource 형식

입출력과 함께 자주 쓰이는 구문입니다! 일반적으로 사용되었던 자원을 끝난 후에 닫아줘야 하는 것들이 존재하는데 여기서 try-catch-finally구문보다 편리한 것이 지금부터 설명드릴 try-with-resource 문입니다,

기존의 try-catch(-finally)문은 자원을 닫을 때 close()를 사용해야 합니다.

try-with-resource문은 try문을 벗어나는 순간 자동적으로 close()가 호출됩니다.

try()안의 입출력 스트림을 생성하는 로직을 작성할 때 해당 객체가 AutoClosable 인터페이스를 구현한 객체여야 합니다.

import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;

import java.nio.charset.StandardCharsets;

public class Main {

public static void main(String[] args) {

//try 소괄호 안에 Closable의 인스턴스만 선언할 수 있다.

//만약 소괄호 안에 쓰지 않는다면 메서드에도 exception 시그니쳐를 써야하며 close()구문도 추가해야한다.

try (FileOutputStream out = new FileOutputStream("test.txt")){

out.write("Hello".getBytes());

out.flush();

}catch (IOException e){

System.out.println("IO Exception발생: " + e.getMessage());

e.printStackTrace();

}

}

}

* 메소드에서의 예외 선언  
  catch문을 이용해서 예외처리를 하지 않은 경우, 메소드에 throws로 예외가 발생할 수 있다는 것을 알려주어야 합니다. throws 키워드가 있는 함수를 호출한다면, caller 쪽에서 catch와 관련된 코드를 작성해주어야 합니다.

void method() throws IndexOutOfBoundsException, IllegalArgumentException {

//메소드의 내용

}