Java의 정석

Chapter 05~6.2

배열

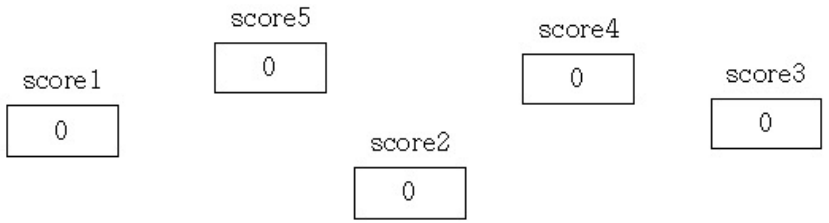
1. 배열(array)

배열

배열이란?

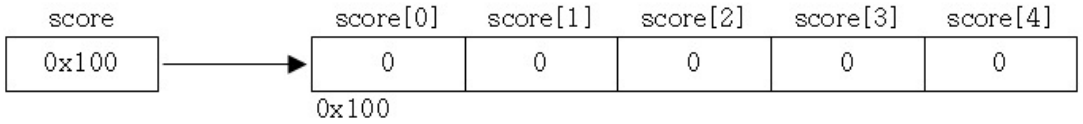
- 같은 타입의 여러 변수를 하나의 묶음으로 다루는 것
- 많은 양의 값(데이터)을 다룰 때 유용하다.
- 배열의 각 요소는 서로 연속적이다.

```
int score1=0, score2=0, score3=0, score4=0, score5=0 ;
```



배열

```
int[] score = new int[5]; // 5개의 int 값을 저장할 수 있는 배열을 생성한다.
```



배열

배열의 선언과 생성

- 타입 또는 변수이름 뒤에 대괄호[]를 붙여서 배열을 선언한다.

선언방법	선언 예
타입[] 변수이름;	**int[] score;** **String[] name;**
타입 변수이름[];	int score[]; String name[];

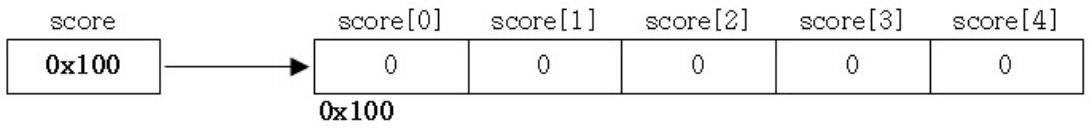
배열

배열의 선언과 생성

- 배열을 생성하기 위해서는 연산자 new 와 함께 배열의 타입과 길이를 지정해 주어야 한다.

```
int[] score;         // 배열을 선언한다.(생성된 배열을 다루는데 사용될 참조변수 선언)
score = new int[5]; // 배열을 생성한다.(5개의 int값을 저장할 수 있는 공간생성)
```

[참고] 위의 두 문장은int[] score = new int[5];와 같이 한 문장으로 줄여 쓸 수 있다.



배열

배열에 값 저장과 읽어오는 법

```
score[3] = 100;       // 배열 score의 4번째 요소에 100을 저장한다.
int value = score[3]; // 배열 score의 4번째 요소에 저장된 값을 읽어서 value에 저장.
```

- 배열의 개수 대신에 배열이름.length 사용 가능

```
int[] score = { 100, 90, 80, 70, 60, 50 };

for(int i=0; i < 6; i++) {
    System.out.println(socre[i]);
}
```

```
for(int i=0; i < score.length; i++) {
    System.out.println(socre[i]);
}
```

배열

배열의 초기화

- 생성된 배열에 처음으로 값을 저장하는 것

```
int[] score = new int[5]; // 크기가 5인 int형 배열을 생성한다.
score[0] = 100;           // 각 요소에 직접 값을 저장한다.
score[1] = 90;
score[2] = 80;
score[3] = 70;
score[4] = 60;
```

생략가능

```
int[] score = { 100, 90, 80, 70, 60};           // 1번
int[] score = new int[]{ 100, 90, 80, 70, 60}; // 2번
```

```
int[] score;
score = { 100, 90, 80, 70, 60}; // 에러 발생!!!

int[] score;
score = new int[]{ 100, 90, 80, 70, 60}; // OK
```

배열

배열의 출력

```
int[] A = new int[]{1, 2, 3, 4, 5};

for(int i=0; i<A.length;i++) {
    System.out.println(A[i]);
}
```

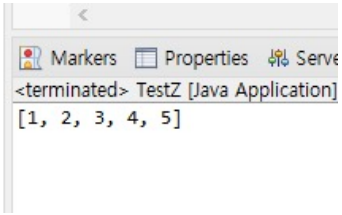
```
<terminated> TestZ [Java App
1
2
3
4
5
```

배열

배열의 출력 (toString 이용)

```
int[] iArr = new int[]{1, 2, 3, 4, 5};

System.out.println(Arrays.toString(iArr));
```



배열

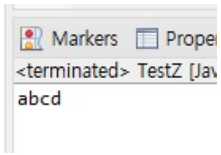
배열의 출력 (toString 이용)

- char배열은 참조변수를 출력하면 요소가 출력되지만, int나 그 외의 배열은 '타입@주소'로 출력됨

```
        char[] chArr = new char[]{'a','b','c','d'};

        System.out.println(chArr);

    }
```



char배열을 제외하고 바로 배열이름을 대입하면 ex) I@14123bb같은 문자열이 출력

배열

배열의 복사

```
int[] arr = new int[5];
int[] newArr = new int[arr.length*2]; //1) 처음보다 두배의 길이의 배열을 생성 10

for(int i=0;i<arr.length;i++) { //2) 처음의 배열을 for문을 이용하여 하나씩 저장
    newArr[i] = arr[i];
}
arr = newArr; // 3) ex)1111100000


System.out.println(Arrays.toString(arr));
```

배열

배열의 복사 (System.arraycopy() 이용)

- 배열의 복사는 for문보다 `System.arraycopy()` 이용이 효율적

```
int[] arr = new int[5];
int[] newArr = new int[arr.length*2]; //1) 처음보다 두배의 길이의 배열을 생성

// 2) arr[0]에서 newArr[0]으로 arr.length개의 데이터를 복사
System.arraycopy(arr, 0, newArr, 0, arr.length);
```

배열

String배열 생성과 초기화

- 초기화 안 할 경우 각 값은 널로 초기화된다.

```
String[] name = new String[3];  // 3개의 문자열을 담을 수 있는 배열

name[0] = "kim";
name[1] = "cho";
name[2] = "lim";
```

배열

String배열 생성과 초기화

- 선언과 동시에 초기화 가능

```
//(방법1)
String[] name = new String[]{"kim", "cho", "lim"};

//(방법2)
String[] name1 = {"kim", "cho", "lim"}; // new String[] 생략
```

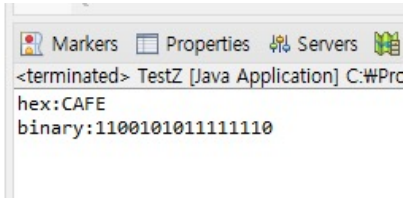
배열

String배열

- String은 "+"연산자를 이용하여 문자열을 붙일 수 있다.
- char배열의 요소를 합쳐서 String으로 생성 가능하다.

```
char[] hex = {'C', 'A', 'F', 'E'};
String[] binary = { "0000", "0001", "0010", "0011"
                   ,"0100", "0101", "0110", "0111"
                   ,"1000", "1001", "1010", "1011"
                   ,"1100", "1101", "1110", "1111" };
String result="";

for(int i=0;i<hex.length;i++) {
    if('0'<=hex[i] && hex[i]<='9') {
        result += binary[hex[i]-'0'];   // point 1) String은 +연산자를 통해서 이어 붙일 수 있다.
    } else {
        result += binary[hex[i]-'A'+10];
    }
}
System.out.println("hex:" + new String(hex));  // point 2) char배열의 요소를 한번에 합쳐서 String배열로 바꾸어 출력
System.out.println("binary:" + result);
```



배열

String 클래스 주요 메서드

- char charAt(int index) : 문자열에서 해당 index에 있는 문자 반환 (index의 값은 0부터 시작)

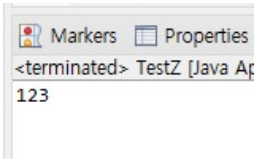
```
String str = "ABCDE";
char ch = str.charAt(3); // 네번째 문자 'D'를 반환
```

배열

String 클래스 주요 메서드

- String substring(int from, int to) : 문자열에서 from~to에 있는 문자열을 반환. (from~(to-1)까지)

```
String str = "012345";
 String tmp = str.substring(1,4); // str에서 index범위 1~3(4는 포함X)반환
 System.out.println(tmp);
```

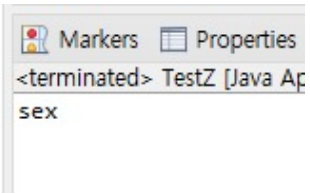


배열

String 클래스 주요 메서드

- boolean equals(String str) : 문자열의 내용이 같은지 확인(대소문자 구분)

```
String str = "abc";
if(str.equals("abc")) { // str과 "abc"의 내용이 같은지 확인
            System.out.println("sex");
}
```

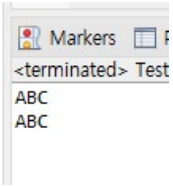


배열

char 배열과 String클래스의 변환

```
char[] chArr = {'A', 'B', 'C'};
String str = new String(chArr); // 1) char배열 -> String
char[] tmp = str.toCharArray(); // 2) String -> char배열

System.out.println(str);
System.out.println(tmp);
```



2. 객체지향 프로그래밍

객체지향 프로그래밍

객체지향언어의 특징

▶ 기존의 프로그래밍언어와 크게 다르지 않다.

- 기존의 프로그래밍 언어에 몇가지 규칙을 추가한 것일 뿐이다.

▶ 코드의 재사용성이 높다.

- 새로운 코드를 작성할 때 기존의 코드를 이용해서 쉽게 작성할 수 있다.

▶ 코드의 관리가 쉬워졌다.

- 코드간의 관계를 맺어줌으로써 보다 적은 노력으로 코드변경이 가능하다.

▶ 신뢰성이 높은 프로그램의 개발을 가능하게 한다.

- 제어자와 메서드를 이용해서 데이터를 보호하고, 코드의 중복을 제거하여 코드의 불일치로 인한 오류를 방지할 수 있다.

객체지향 프로그래밍

클래스와 객체의 정의와 용도

- 클래스의 정의 – 클래스란 객체를 정의해 놓은 것이다.
- 클래스의 용도 – 클래스는 객체를 생성하는데 사용된다.

- 객체의 정의 – 실제로 존재하는 것. 사물 또는 개념.
- 객체의 용도 – 객체의 속성과 기능에 따라 다름.

클래스	객체
제품 설계도	제품
TV설계도	TV
붕어빵기계	붕어빵

객체지향 프로그래밍

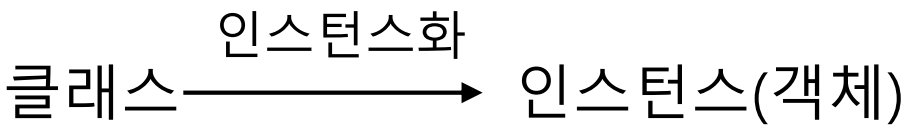
객체와 인스턴스

▶ 객체 ≒ 인스턴스

- 객체(object)는 인스턴스(instance)를 포함하는 일반적인 의미

책상은 인스턴스다. 책상은 객체다.	책상은 책상 클래스의 객체다. 책상은 책상 클래스의 인스턴스다.

▶ 인스턴스화- 클래스로부터 인스턴스를 생성하는 것.



객체지향 프로그래밍

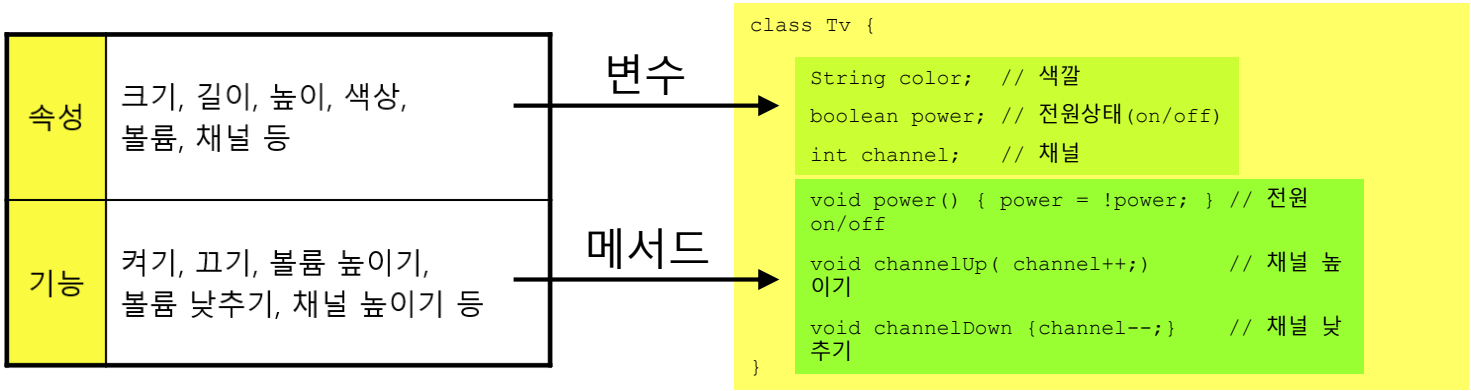
객체의 구성요소

▶ 객체는 속성과 기능으로 이루어져 있다.

- 객체는 속성과 기능의 집합이며, 속성과 기능을 객체의 멤버(member, 구성요소)라고 한다.

▶ 속성은 변수로, 기능은 메서드로 정의한다.

- 클래스를 정의할 때 객체의 속성은 변수로, 기능은 메서드로 정의한다.



객체지향 프로그래밍

객체와 인스턴스 생성과 사용

- 인스턴스의 생성방법

  클래스명 참조변수명; // 객체를 다루기 위한 참조변수 선언

  참조변수명 = new 클래스명(); // 객체생성 후, 생성된 객체의 주소를 참조변수에 저장

```
Tv t;
t = new Tv();
```

```
Tv t = new Tv();
```

객체지향 프로그래밍

클래스

클래스 – 사용자 정의 타입(User-defined type) (객체의 집합)

- 서로 관련된 변수들을 정의, 이들에 대한 작업을 수행하는 함수들을 함께 정의한 것.

- C언어에서는 문자열을 문자의 배열로 다루지만, 자바에서는 String이 클래스이다.

- 문자열을 단순히 문자의 배열로 정의하지 않고 클래스로 정의한 이유는 문자열과 문자열을 다루는데 필요한 함수들을 함께 묶기 위함.

객체지향 프로그래밍

클래스

클래스 – 사용자 정의 타입(User-defined type)

- 프로그래머가 직접 새로운 타입을 정의할 수 있다.
- 서로 관련된 값을 묶어서 하나의 타입으로 정의한다.

```
class Time {
        int hour;
        int minute;
        int second;
}
```

```
int hour;
int minute;
int second;
```

→

```
Time t = new Time();
```

```
int hour1, hour2, hour3 ;
int minute1, minute2. minute3;
int second1, secound2, second3;
```

→

```
Time t1 = new Time();
Time t2 = new Time();
Time t3 = new Time();
```

```
int[] hour = new int[3];
int[] minute = new int[3];
int[] second = new int[3];
```

→

```
Time[] t = new Time[3];
t[0] = new Time();
t[1] = new Time();
t[2] = new Time();
```

SOLID

3. SOLID란

https://www.nextree.co.kr/p6960/ 참조

SOLID

SOLID 원칙을 지키는 이유.

SOLID 원칙들은 결국 자기 자신 클래스 안에 응집도는 내부적으로 높이고, 타 클래스들 간 결합도는 낮추는 High Cohesion - Loose Coupling 원칙을 객체 지향의 관점에서 도입한 것이다.
왜 그랬을까? 간단하다. 좋은 소프트웨어는 응집도가 높고 결합도가 낮기 때문이다.
결국 모듈 또는 클래스 당 하나의 책임을 주어 더욱더 독립된 모듈(클래스)을 만들기 위함이다.
이렇게 설계된 **소프트웨어는 재 사용이 많아지고, 수정이 최소화 되기 때문에 결국 유지 보수가 용이해진다.**

SOLID

SOLID 다섯 원칙

- SRP(Single Responsibility Principle) 단일 책임 원칙
- OCP(Open Closed Principle) 개방 폐쇄 원칙
- LSP(Liskov Substitution Principle) 리스코프 치환 원칙
- ISP(Interface Segregation Principle) 인터페이스 분리 원칙
- DIP(Dependency Inversion Principle) 의존 역전 원칙

SOLID

SRP(단일 책임 원칙)

클래스의 역할과 책임을 너무 많이 주지 마라

즉, 클래스를 설계할 때 어플리케이션의 경계를 정하고, 추상화를 통해 어플리케이션 경계 안에서 필요한 속성과 메서드를 선택 해야 한다.

SOLID

SRP(단일 책임 원칙)

이렇게 사람이라는 **클래스에 모든 사람에 관련된 모든 기능을 다 때려 박기보다 목적과 취지에 맞는 속성과 메서드로 구성 해야 한다.** **즉 관련된 책임만 주라는 것이다.** 이 말은 결국 SRP(단일 책임 원칙)은 추상화와 깊은 관련이 있다는 소리다.

SOLID

OCP(개방 폐쇄 원칙)

자신의 확장에는 열려 있고, 주변의 변화에 대해서는 닫혀 있어야 한다.

따라서 **OCP(개방 패쇄 원칙) 의거하여 수정해보자. 아래와 같이 상위클래스 또는 인터페이스를 중간에 두어 직접적인 연동은 피하게 설계한다.** 드론연동프로그램은 따로 코드 수정이 없으면서 도 다른 제품과의 연동엔 확장적이게 된다. 상위클래스나 인터페이스는 일종의 완충 장치인 것 이다.

SOLID

LSP(리스코프 치환 원칙)

하위 클래스의 인스턴스는 상위형 객체 참조 변수에 대입해 상위 클래스의 인스턴스 역할을 하는데 문제가 없어야 한다.

LSP는 인터페이스와 클래스 관계, 상위 클래스와 하위 클래스 관계를 얼마나 잘 논리적으로 설계했느냐가 관건이다.

LSP는 하위 클래스가 상위클래스의 역할을 대신 할 때 논리적으로 맞아 떨어져야 한다.

아들이 태어나 홍길동이라는 이름을 짓고 아버지의 행위를 한다??? 뭔가 어색하다..
고래 한마디라 태어나 도커라는 이름을 짓고 포유류의 행위를 한다. 깔끔하지 않은가?

객체 지향은 인간이 실세계를 보면서 느끼고 논리적으로 이해한 것과 똑같이 프로그래밍하는 게 목적이기 때문에 논리적으로 맞아 떨어져한다.

하위클래스가 상위클래스를
구현할 때 논리적으로
맞아 떨어져야 한다.

특정 메소드가 상위 타입을
인자로 사용한다고 할 때, 그
타입의 하위 타입도 문제 없이
정상적으로 작동을 해야 한다

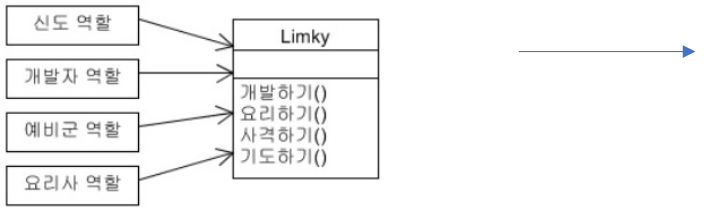
Ex) 정사각형 직사각형 예제

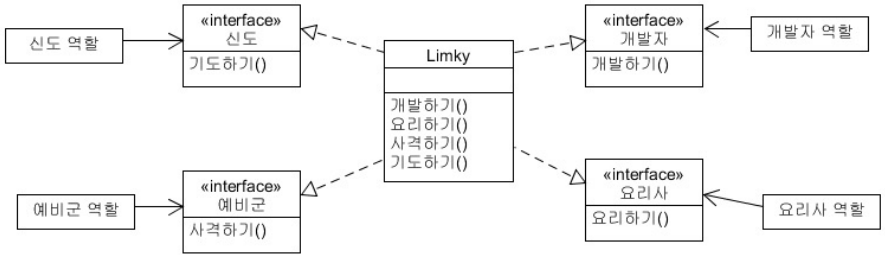
SOLID

ISP(인터페이스 분리 원칙)

상황과 관련 있는 메서드만 제공하라.

ISP(인터페이스 분리 원칙)은 SRP(단일 책임 원칙)과 같은 원인에 대한 다른 해결책을 제시하는 것이다. 너무 많은 책임을 주어 상황에 관련 되지 않은 메서드까지 구현했다면, SRP(단일 책임 원칙)은 그 클래스를 여러개의 클래스로 쪼개버린다. **하지만 ISP(인터페이스 분리 원칙)은 해당 클래스를 그냥 냅 두는 상태에서 인터페이스 최소주의 원칙에 따라 각 상황에 맞는 기능만 제공하도록 필터링 한다고 생각하면 쉽다.** 자 글만 보면 이해가 안되기 때문에 그림을 보자.





만약 신도 역할만 해야 하는 상황에서 개발하기(), 요리하기(), 사격하기() 같이 관련 없는 메서드는 신도 역할에서 필요 없다.
다른 역할도 마찬가지다.
이럴 경우 ISP(인터페이스 분할 원칙)을 적용하여, 각 역할에 맞는 메서드만 제공하도록 수정해보자.

SRP가 클래스의 단일책임을 강조한다면 ISP는 인터페이스의 단일책임을 강조합니다.

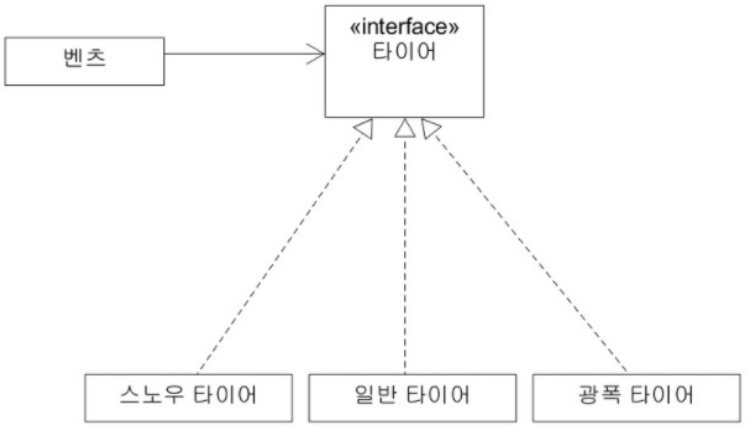
SOLID

DIP(의존 역전 원칙)

자신보다 변하기 쉬운 것에 의존하지 마라

구체적으로 **추상클래스 또는 상위클래스는 구체적인 구현클래스 또는 하위클래스에게 의존적이면 안된다.**

DIP(의존 역전 원칙)을 지키지 않는 설계



DIP(의존 역전 원칙)을 지키는 설계

참고

1. 자바의 정석 ppt 참고