

# 12.22 木 \_ 객체지향 언어 3

## class 시작 첫 글자는 대문자!!!!

```
ex) class Betty {  
    }
```

## class • object

| package 는 폴더 개념

자바에서는 모든 프로그램의 작성 단위가 **class단위** 이다

```
package trial; // package는 폴더(디렉토리) 개념 !!!  
                // 자바로 프로그램을 만들면 클래스 종류가 엄청 많아진다.  
                // 그러면 class 이름이 중복이된다 or 어느 class가 무슨 기능을 하는지 모른다  
                // -> 기능별로 패키지(폴더)에 담아서 저장한다  
  
//import java.util.Scanner; => java 폴더안에 util 폴더안에 Scanner를 가져오라!  
  
class Foo{  
  
}  
  
class Bar{  
  
}  
  
class Pos {  
  
}  
  
public class test1  
    public static void main(String[] args) {  
  
        new Foo();  
        // new trial.Foo(); //이렇게 패키지.class 이름으로 사용  
                                //→ 해당 패키지 밖에 있을 때 사용을 한다 [ 폴네임 ]  
  
        new Bar();  
        new Pos();  
  
    }  
}
```

컴파일을 하면 바이트 코드인 중간 코드 **.class 파일**이 나온다 **.class 파일**을 실행하면 **.class 코드**가 **jvm**에 올라가서 실행이 된다.

**public** 이 붙어있는 **class**가 있는데, 이 **class**의 이름은 **.java 파일 이름**과 같은 것이 무조건 하나 있어야 한다.

**.java 파일**을 컴파일 하고 **.java 파일**에서 나온 중간 코드를 실행하면 자바는 **.java 파일**의 이름과 동일한 **class**를 찾는다.

그리고 그 **class**의 **main method**를 사용하도록 미리 약속이 되어있다.

1. 컴파일 하고 실행하면 현재 **.java 파일**을 실행한다.
2. 해당 파일을 컴파일 하는데 각 클래스마다 **.class 파일**이 나온다.
3. **main 메소드**를 찾고 실행 시킨다.

자바 컴파일러가 실행될 때 **class**이름은 2가지가 있다.

#### 숏네임

- 해당 **package(폴더)** 안에서 사용 되는 것
- `new Foo ( ) ;` 이렇게 해당 **class 이름**만 사용

#### 풀네임

- 전체 전역적으로 사용, **package이름** + **class 이름**
- `new trial .Foo ( ) ;` 이렇게 **패키지.class 이름**으로 사용  
→ 해당 패키지 밖에 있을 때 사용을 한다

⇒ 보통은 **숏네임**을 사용하면 된다. **풀네임**은 **class**의 개수가 많아져서 패키지(폴더)를 분류시켜 복잡할 때 사용하면 된다.

실행할 때 실행 단위는 **.java 파일** 단위로 자바에서 실행 시킬 수 있다.  
파일 하나 선택하고 실행 시키면 선택한 파일이 컴파일 되어 실행된다.

그 주위에 다른 class가 있으면 알아서 땡겨온다.

**내가 현재 실행하는 .java파일 이름과 동일한 이름을 가지는 class를 찾아!!**

그 class 안의 **main메소드** 를 실행한다

→ **main메소드** 가 실행된다는 것 = 프로그램이 실행된다는 것

→ **main메소드** 가 종료된다는 것 = 프로그램이 종료된다는 것

※ 실제로 **main메소드** 안은 간단하다 →

컴파일하고 실행한다는 것은 현재 .java파일

.java파일을 열어 놓고 컴파일을 실행 시킨다

컴파일을 하면 각 class 별로 .class 파일들이 나온다

그러면 .class파일을 실행하는데 .java파일을 실행한다

## class 안에 들어갈 수 있는 구성 요소(class가 가지는 구성 요소)

### 1. **생성자** (초기화)

- 객체를 찍을 때 초기화 작업이 필요한데, 초기화 작업 알고리즘을 실행시킨다
- 반환하는 반환 값의 자료형이 없다 ( new 하고 생성된 객체의 주소 값을 반환하기 때문)

**!** 멤버 변수에 바로 초기 값을 넣기 보다는 **생성자** 안에 설정하기 !!!!!

예시 )

```
class StudentInfo{
    // 학생 정보를 "저장"하는 클래스
    static int seqNum; // 입력 순서
    //----- 1번 메소드
    int id; // 학번
    String name; // 학생의 이름
    //----- 2번 메소드
    int scoreKor; // 국어 성적
    int scoreEng; // 영어 성적
}
```

```

int scoreMath;    // 수학 성적
int sum;          // 합계
float avg;        // 평균

StudentInfo(int argId, String argName){ // 학생의 학번과 이름을 저장하겠다
    id = argId;
    name = argName;
}

void setScore(int argKor, int argMath, int argEng) {
    // 점수를 지정
    scoreKor = argKor;
    scoreMath = argMath;
    scoreEng = argEng;
    sum = scoreKor + scoreMath + scoreEng; // 합계를 구함
    avg = sum / (float)GpaManager.NUM_OF_SUBJECT; // 평균을 구함
}
}

```

## 2. 멤버 변수

- 특정 **멤버 메소드** 안에 정의된 알고리즘이 실행되려면 데이터를 가지고 있어야 하는데, 그 데이터들은 **멤버 변수**에 저장되어있다.

## 3. 멤버 메소드

- OOP로 프로그램을 짰다고 하면 복잡한 알고리즘은 **멤버 메소드**에 거의 저장 !!

class 밖에는 어떤 코드도 들어가면 안된다!! → 객체와 함께 다 끌려 와버려서

**생성자**, **멤버 메소드**의 공통점은 그 안에 **알고리즘(코드)**이 들어간다

# 생성자의 역할

```

class Foo {

    int k;

    Foo ( ) {
        k = 10;
    }
}

```

- => 특정 조건이 있는 알고리즘이 들어오면 그때 그때 초기화를 할 수가 없다
- => 생성자안에 모아두면 어떤 초기화 작업을 하는지 한눈에 한꺼번에 다 보이고(전체적인 가독성이 좋음)

==> 알고리즘 적으로 동작하려면 method 안에 있어야 한다.

☆☆ 상수의 초기화 또한 생성자에서 해야 한다!! ☆☆☆

```
class Foo {
    ☆ final int k;    // final 붙이면 상수 / 상수는 선언과 동시에 초기화를 한다

    Foo () {
        ☆ k = 10;    // 상수의 초기화도 생성자에서 해야한다!!!
    }
}
```

OOP 관점에서 **생성자**는 객체가 만들어질 때 **딱 1번 실행**

⇒ 객체 안의 모든 초기화 작업을 **생성자**가 담당한다!! ⇒ **상수** 또한 **생성자**에서 선언한다!

## 메모리 누수 현상

- 객체를 다 사용했는데도 삭제하는 것을 깜빡한다.  
프로그램이 계속 돌면 메모리가 증가한다 → 메모리 누수 현상

```
package trial;

//import java.util.Scanner;

class Scv{
    int myId;

    Scv(){
        // Memory Leak   메모리 누수현상

        // GC (Garbage Collector)  [ 쓰레기 수집자 ] -jvm에 들어가있는 기능
        // => 만들어진 객체를 계속 본다.
        // => 생성된 객체가 참조하는 참조변수가 없으면 해당 객체를 다 없애 버린다
    }
}

public class test1 {

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

```

    // 객체는 찍었는데 이 객체를 가리키는 참조변수가 없다 ..... 그냥 쓰레기다.....;;
    // 메모리도 잡아먹는다.....ㅠ
    new Scv();
    new Scv();
    new Scv();
    new Scv();
    new Scv();
    }
}

```

## 생성자 개념

```

package trial;

//import java.util.Scanner;

class Scv{
    int myId;

    Scv(){          // 생성자
        myId = myId + 1;
    }
}

public class test1 {

    public static void main(String[] args) {

        // . 연산자 = 해당 참조변수가 가리키는 객체 안의 멤버에 접근하기 위해 사용
        System.out.println((new Scv()).myId); // new Scv() 의 반환 값은 주소 값!
        System.out.println((new Scv()).myId); // 여기에 괄호로 넣으면 해당 주소 값 !
        System.out.println((new Scv()).myId); // 예 . 연산자를 붙이면 해당 객체의 멤버에 접근
        System.out.println((new Scv()).myId);
        System.out.println((new Scv()).myId);

        // 객체는 전부다 독립적으로 생성이된다 -> 각각의 객체이기 때문에 다 적용되는것도 각각!

    }
}

```

```
Console × Problems Debug Shell
<terminated> test1 [Java Application] C:\Users\Wjj94W.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe (2022.1
1
1
1
1
1
1
```

## class 멤버 변수의 사용

```
package trial;

//import java.util.Scanner;

class Scv{

    // 멤버 변수
    static int myId;          // 이렇게 하면 각각 myId 에 1씩 증가시킬 수가 있다

    // 멤버 생성
    Scv(){                    // 생성자는 필요 없으면 안써도 된다!!! 필요에 따라 사용 가능
        myId = myId + 1;     // 생성자에 +1 씩 증가시키는 알고리즘이 들어가 있다
    }
}

public class test1 {

    public static void main(String[] args) {

        System.out.println((new Scv()).myId);
        System.out.println((new Scv()).myId);
        System.out.println((new Scv()).myId);
        System.out.println((new Scv()).myId);
        System.out.println((new Scv()).myId);
    }
}
```

```
1 package trial;
2
3 //import java.util.Scanner;
4
5 class Scv{
6     static int myId;
7
8     Scv(){
9         myId = myId + 1;
10    }
11 }
12
13
14 public class test1 {
15
16     public static void main(String[] args) {
17
18         System.out.println(new Scv().myId);
19         System.out.println(new Scv().myId);
20         System.out.println(new Scv().myId);
21         System.out.println(new Scv().myId);
22         System.out.println(new Scv().myId);
23     }
24 }
25
26
```

Console × Problems Debug Shell

```
<terminated> test1 [Java Application] C:\Users\Wji94W\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.4.v20220903-1038\jre\bin\java.exe (2022.12.22 오전 11:29:19 - 오전 11:29:19) [pid: 22408]
1
2
3
4
5
```

## 오버로딩 (Overloading)

한 class 내에 이미 사용하려는 이름과 같은 이름을 가진 메서드가 있더라도 ,

매개변수의 개수 or 타입 이 다르면, 같은 이름을 사용해서 메소드를 정의 가능

★★★★ 한 class 내에 같은 이름의 메서드를 여러 개 정의하는 것 ★★★★★

하나의 class 안에서 동일한 이름의 생성자 또는 method 가 존재하고 매개변수의 자료형 과 개수 에 따라 적합한 생성자 / 메서드를 호출하는 방법

- 생성자 , method 에 적용된다
- 오버로딩 적용 범위는 하나의 클래스 안에서!



- 파라미터 (매개변수) , 매개변수의 개수와 자료형을 보고 지가 동적으로 호출

```
class Ichi {
    void prt() {
        System.out.println("hello : ");
    }

    void prt(int a) {
        System.out.println("hello : " + a);
    }

    void prt(float a) {
        System.out.println("hello : " + a);
    }

    void prt(char a) {
        System.out.println("hello : " + a);
    }

    void prt(int a , float b) {
        System.out.println("hello : " + a + ": " + b);
    }
}

public class T5 {
    public static void main(String[] args) {
        new Ichi().prt();
    }
}
```

- 반환형 , 이름은 동일해야 한다 !!!

```
TEST_2.java T1.java T2.java T3.java T4.java *T5.java X
4 void prt() {
5     System.out.println("hello : ");
6 }
7
8 void prt(int a) {
9     System.out.println("hello : " + a);
10 }
11
12 void prt(float a) {
13     System.out.println("hello : " + a);
14 }
15
16 char prt(char a) {
17     System.out.println("hello : " + a);
18 }
19
20 void prt(int a , float b) {
21     System.out.println("hello : " + a + ": " + b);
22 }
23
24 }
```

- 프로그래머에게 프로그래밍의 편의성을 준다.

자료형에 따라 하는 일이 달라진다

동일한 기능을 수행한다. 들어오는 인자 값에 따라 초기화 하는 방식이 달라진다.



왜 써야 하나?

하는 행동은 똑 같은데 , 출력을 다르게 하고 싶을 때 사용한다 .

오버로딩을 사용 안 하면 출력 시 마다 이름을 계속 바꿔줘야 한다. ex) println

## Overloading of Method

```
package trial;

// Overloading
// -> 구조적언어 부터 시작 된 개념
// -> 적용 대상 : 두 개
// 1) 생성자
// 2) 메서드 ※.

class Scv{

    void prt() {
        System.out.println("hello : ");
    }

    void prt(int a) {
        System.out.println("hello : " + a);
    }

    void prt(float a) {
        System.out.println("hello : " + a);
    }

    void prt(char a) {
        System.out.println("hello : " + a);
    }

    void prt(int a , float b) {
        System.out.println("hello : " + a + ": " + b);
    }

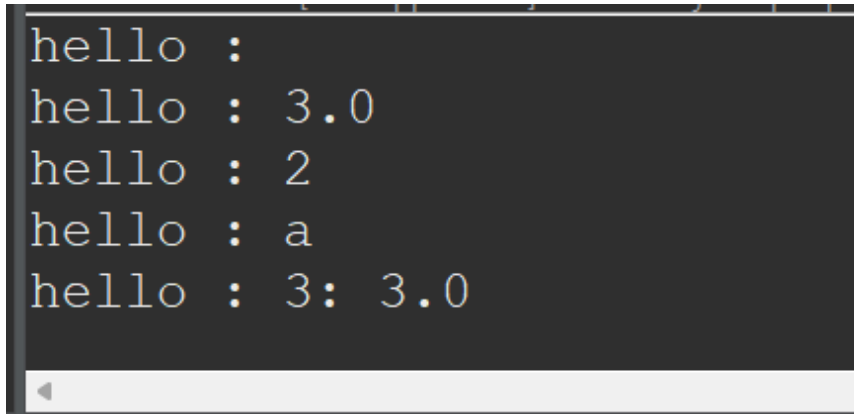
}

public class test1 {

    public static void main(String[] args) {

        (new Scv()).prt(); (new Scv()).prt(3.0f);
    }
}
```

```
(new Scv()).prt(2); (new Scv()).prt('a');  
(new Scv()).prt(3, 3.0f);  
  
System.out.println(1); // println 자체가 Overloading 개념이 적용되어있다  
System.out.println(1.0f);  
System.out.println("betty");  
  
}  
}
```



```
hello :  
hello : 3.0  
hello : 2  
hello : a  
hello : 3: 3.0
```