JAVA 10강 요약

강사 박주병

1. 패키지

패키지란 자바의 기본 단위가 되는 클래스들을 묶어두는 폴더 역할을 한다. 우리가 바탕화면에 똑같은 이름의 파일을 2개 놔둘 수 없으니 폴더를 만들어 분류해 놓은 것과 같은 것이다. 예를 들어 Util 이라는 클래스명은 흔히 쓰이는 클래스 명이다. 이는 다른 팀 혹은 다른 회사와 협업 또는 다른 라이브러리들을 가져올 때 이런 흔한 클래스명 들이 충돌 할 수 있다. 패키지라는 개념이 없다면 한쪽이 클래스 이름을 다르게 바꿔줘야 하는데 관련된 모든 소스코드를 수정해야 하는 어려움이 있다. 그리고 혼자서 개발을 한다 해도 클래스의 개수가 많아지면 용도별로 분류할 필요가 있다.

그림1 처럼 패키지는 실제 윈도우 상에서 폴더랑 동일하게 관리된다. joo라는 패키지는 실제 joo 라는 폴더를 만들어서 그 안에 .java 파일을 놔두게 된다.

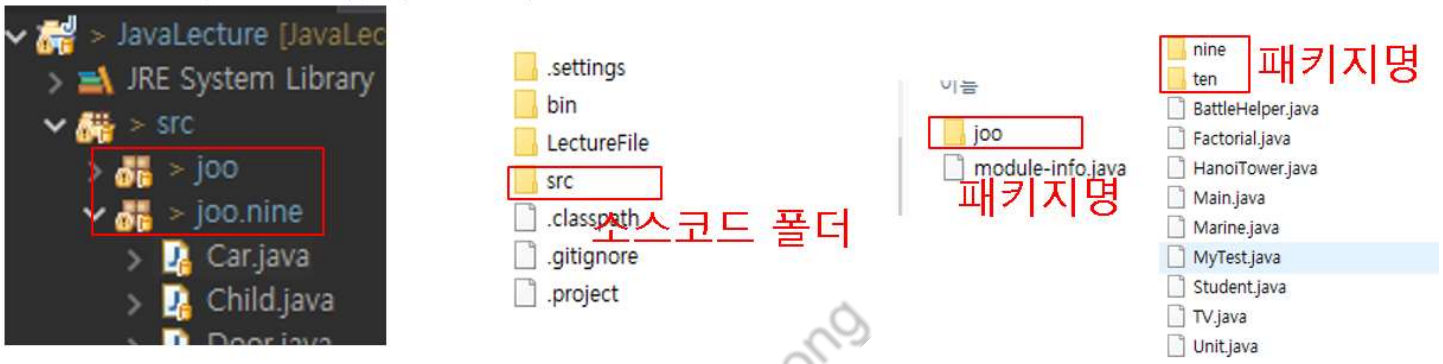


그림 1

1.1 패키지의 규칙

1.1.1 - 모든 클래스는 반드시 하나의 패키지 안에 속해야 한다(이는 자바 버전에 따라 다르지만 패키지를 무조건 생성하는걸 추천한다)

1.1.2 대소문자 모두 사용가능하지만 클래스명과 구분하기 위해 소문자만 사용하는 것이 일반적이다.

1.1.3 문법적 제약은 아니지만 일반적으로 도메인형식을 으로 네이밍을 한다.

2. import

패키지가 다른 클래스를 사용하려면 그냥 사용 할 수 없다 클래스 선언 위쪽에 import 키워드를 사용하여 사용하려는 클래스의 패키지명을 적어주어야 한다.

```
1  package joo;
2
3  import joo.nine.Car;
4
5
6
7  public class Main {
8
9
10
11     public static void main(String[] args) {
12         |
13         Car test = new Car();
```

그림 2

```
1  package joo;
2
3
4
5
6
7  public class Main {
8
9
10
11     public static void main(String[] args) {
12
13         joo.nine.Car test = new joo.nine.Car();
```

그림 3

그림3처럼 import 하지 않고 사용 할 때마다 해당 클래스의 패키지명을 다 적어주어 사용할 수도 있다.

하지만 이는 가독성이 떨어지고 성능에도 전혀 영향을 주지 않는다. 간혹 직접 적어주는게 더 빠르지 않을까 라고 생각 하시는분이 있을텐데 import 보다 컴파일속도가 아주 약간 빠를 뿐이다. 하지만 코드 가독성은 매우 중요한 요소이므로 단순히 컴파일 속도만을 위해 가독성을 떨어 트릴순없다. 그래서 웬만하면 import 하는 것을 추천한다.

그림4처럼 같은 패키지안에 여러개의 클래스를 사용할 때 import를 여러개 해야 한다.
이럴 때는 *을 사용하여 해당 패키지 안의 모든 클래스를 import 할 수 있다.

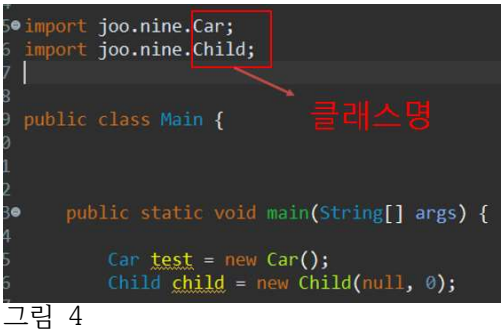


그림 4

그림 5

2.1 자동으로 되는 import

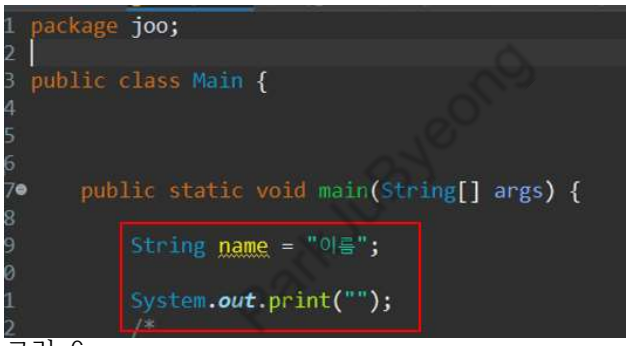


그림 6

String, System 역시도 클래스 인데 이들은 분명 다른 패키지에 있는데도 불구하고 import 없이 사용할수 있는 것을 볼수 있다. 이 클래스들은 java.lang 패키지에 소속되어 있는데 해당 패키지는 워낙 빈번하게 쓰이다 보니 자동으로 import 시켜주기에 그냥 쓸수 있다.

2.2 static import

static 변수나 메서드를 사용할 때 우리는 클래스이름.static멤버 형태로 클래스 이름을 항상 적어줘서 사용했었다. 이러한 것들도 import를 이용하여 생략 할수 있다.

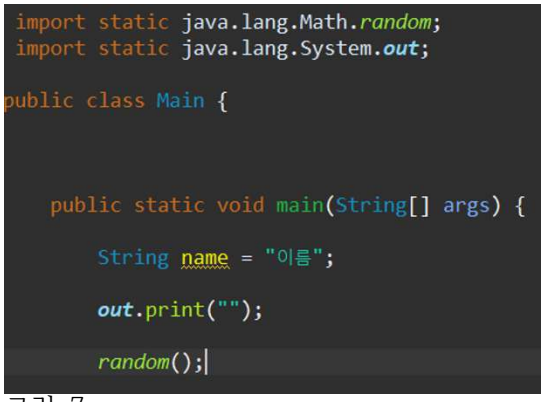
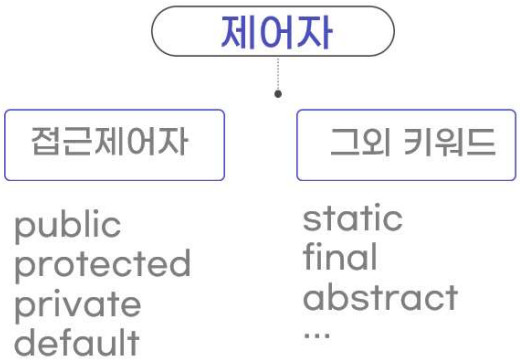


그림 7

그림7의 out 과 random은 static으로 되어 있는 변수와 메서드이다 import static을 사용하여 바로쓰는 것을

볼수 있다.
하지만 이런식으로 클래스 명을 생략하는 것은 개인적으로 좋지 않다고 생각한다. 객체지향 언어는 클래스를 기본 단위로 돌아가는 언어이다. 위의 예시처럼 out 과 random이 어느 클래스에 들어있는지 볼려면 선언부로 가서 보거나 임폴트 부분을 참조해야만 한다. 이는 코드가 길어진다면 상당히 불편한 요소이다.

3. 제어자



제어자는 크게 접근제어자 와 그 외 키워드들로 구성되어 있다. 이러한 제어자들은 스코프(참조할수 있는 범위)를 조절하거나 라이프 사이클 그 외 특수한 제어를 하고 싶을 때 사용 한다. 이러한 제어자들은 클래스, 메서드, 멤버 변수 앞에 작성을 하며 서로간의 순서는 정해져 있지 않으나 일반적으로 접근제어자를 가장 왼쪽에 적어준다.

```
7 public class Main {
8
9
0
1     public static final String name="이름";
2
3●    public static void main(String[] args) {
4
5         String name = "이름";
6
7         out.print("");
```

3.1 final

변하지 않는것들을 표현할 때 쓰인다. 앞서 제어자는 클래스, 메서드, 멤버 변수 에 적용된다고 하였다. 어디에 적용하냐에 따라 기능이 조금씩 다를수 있으나 근본적인 의미는 변하지 않는 것을 의미한다.

3.1.1 변수에 적용될 때

그림10처럼 멤버변수에 적용이 된다면 해당 변수는 상수가 되어 값을 변경할수 없다.

```
static final  String name="이름";

public static void main(String[] args) {

    name = "파이널 변경";
```

그림 10

상수가 된다면 최초 1번만 초기화 할 수 있고 그 이후에는 값을 변경 할 수 없다. JAVA에서는 최초 선언시 or 생성자에서 딱 한번 초기화를 허용 한다. 초기화를 하지 않으면 컴파일 에러가 발생한다. 아래의 그림을 참고하자

```java
final class  Card{

    final int number;

     Card(int number)
    {
         this.number = number;

    }
}
```

3.1.2 메서드에 적용될 때

final이 메서드에 적용된다면 이는 상속은 가능하되 오버라이딩 하여 덮어 쓸 수 없다. 변하지 않는의 의미로 생각한다면 쉽다.

```java
class a{

    public final void test()
    {

    }
}

class b extends a
{
    public final void test()
    {

    }
}
```


3.1.3 클래스에 적용될 때

클래스에 적용된다면 해당 클래스는 상속될수 없다. 이역시 변하지 않는의 의미를 잘 보여주고 있다. 대표적인 final 클래스는 Math가 있다.

```java
final class  a{

    public final void test()
    {

    }
}

class b extends a
{

}
```

```java
 */

public final class Math {

    /**
```

3.2 abstract

추상의 라는 의미이다. 추상적이다라는 것은 의미가 모호하고 명확하지 않다는 것이다. 프로그래밍에서는 완성하지 않는 것을 의미 한다.
abstract는 변수에는 적용될수 없다. 메서드에 적용된다면 이를 추상메서드라고 부르며 메서드의 내부가 구현되어 있지 않다. 클래스에 적용된다면 해당 클래스는 객체 생성을 하지 못한다. 객체라는 것은 실질적으로 존재하는 것을 말한다. 즉 미완성 이라는 말은 실질적으로 존재하지 않는다는 것을 말하며 이를 추상클래스로써 표현한다. 따라서 추상 클래스는 객체를 생성 할 수 없는 제약을 만드는 것이다.

추상메서드를 가지는 클래스는 무조건 추상클래스가 되어야 한다. 이는 생각해보면 당연한 것이다. 추상메서드는 내부가 구현되어 있지 않아 호출되면 어떤 코드를 수행해야할지 알 수 없다. 이러한 불완전한 클래스가 객체 생성되어져 메서드가 호출된다면 동작하지 않을 것 이다

```
abstract class  Card{

    public final int number = 3;

     Card()
    {

    }

    abstract void Shuffle();

     void Pick()
     {

     }

}
```

하지만 착각하면 안된다. 위의 말이 추상메서드가 없으면 추상클래스가 될 수 없다가 아니다.
그림15처럼 추상메서드가 없지만 클래스를 추상화 하여 객체생성을 못하게 막을수 있다.

```
public abstract class Car {


    int speed;

    public Door doors[] = new Door[4];

    public Car()
    {

       //문 4개를 생성한다.
        for(int i =0;i<doors.length;i++)
            doors[i] = new Door();
    }

    void go(int speed)
   {
       this.speed = speed;
    }



    void stop()
```

그림 15

여기서 드는 의문점이 있을 것이다. 사용하려고 만든 클래스와 메서드인데 객체 생성도 못하고 심지어 메서드 내부가 비어 있는 형태이다. 이러한 코드들이 왜 필요할까?
차라는 것이 실제 존재하는 것 인가를 생각해볼 필요가 있다. 객체지향 프로그래밍은 모든 사물을 객체 중심으로 인식하고 이를 프로그램 코드로 그대로 녹여내는 방식이다.

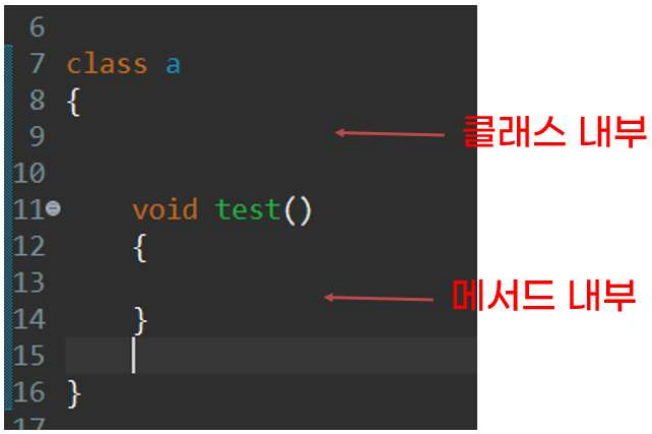
잘 생각해보면 이 세상에 차라는 것은 존재하지 않는다. 차의 특성을 가지는 번호판이 부여된 k3, 아반떼, 소나타 등등이 존재하는 것이다. 즉 Car 라는 것은 실제로 객체화 되는 용도라기보다는 k3, 아반떼, 모닝 등과 같은 객체들의 공통점을 모아 놓은 추상적인 개념인 것이다. 따라서 Car를 객체화 한다는 것은 현실세계를 그대로 반영하는 것과는 거리가 먼 것이다.

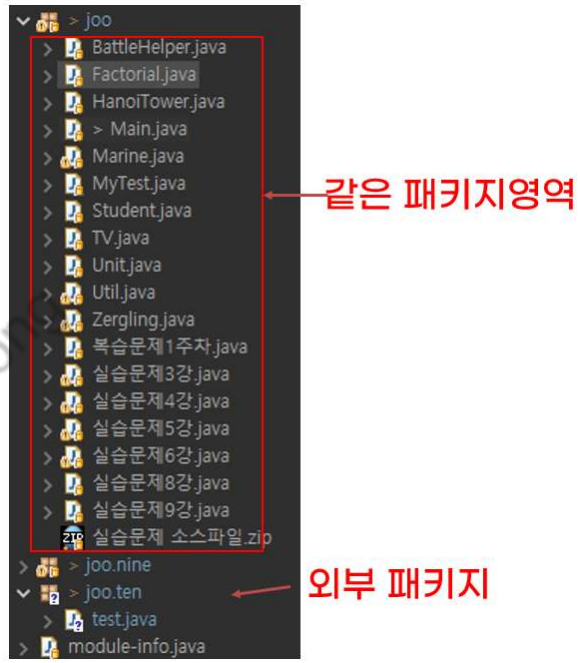
4. 접근제어자

4.1 자바의 영역

접근제어자를 다루기 이전에 자바 언어에서의 영역을 확인 해둘 필요가 있다. 접근제어자는 이런 영역을 기준으로 스코프를 조절하는 기능이기 때문이다. 스코프란 변수나 메서드, 클래스 등을 접근 할 수 있는 범위를 말하는 것이다.

우선 자바는 크게 4가지 영역으로 나눠 볼수 있다.





메서드 내부에서 선언되어 지는 변수를 지역 변수라고 부르며 지역변수는 해당 메서드 영역을 벗어날 수 없다.

접근제어자는 4가지 종류가 있으며 private -> default -> protected -> public 순서로 접근할수 있는 범위가 넓어진다. 아래의 표를 참고 하고 차례대로 보도록 하자.

public	어디서든 접근 할 수 있다.
protected	default + 외부패키지의 자식 클래스
default	같은 패키지에서 사용가능
private	클래스 내부에서만 사용가능

4.2 private

가장 좁은 범위의 접근제어자 이다.

```
class test
{
    private String name;


    void a()
    {
        name = "클래스내부 사용가능";
    }
}
```

내부에서는 사용가능

```
ublic class Main {

    public static void main(String[] args) {

        test t= new test();

        t.name = "외부에서 사용불가능";
```

4.3 default

디폴트는 키워드를 작성하는 것이 아니라 아무것도 적지 않으면 적용되는 접근제어이다.

```
class test
{
     String name;


    void a()
    {
        name = "클래스내부 사용가능";
    }
}


ublic class Main {

    public static void main(String[] args) {

        test t= new test();

        t.name = "외부에서 사용가능";
```


```
import joo.ten.test;


public class Main {

    public static void main(String[] args) {

        test t= new test();

        t.name = "디폴트는 다른패키지에서 사용불가";
```

다른 패키지에서는 접근할수 없다.

4.4 protected

디폴트에서 외부패키지 이지만 해당 클래스를 상속받은 자식이라면 접근할 수 있다.

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

    test t= new test();

    t.name = "디폴트는 다른패키지에서 사용불가";
```

protected는 같은 패키지
혹은 상속받은 자식에서 접근가능(패키지가 달라도
상속 받은 자손이면 가능)

4.5 public

public은 어디서든 다 접근가능하기에 따로 예제를 보여주진 않겠다.

지금까지는 접근제어자가 변수에 적용됐을때를 설명하였으나 메서드, 클래스 역시 이와 모두 동일하기에 따로 설명하진 않겠다.

다만 클래스의 경우 위의 표에서 봤듯이 private과 protected는 사용이 불가능하다. 이는 잘 생각해보면 당연한 것이다. 클래스는 private으로 선언해버리면 아무도 해당 클래스를 사용할수 없어 절대 실행할수 없는 코드가 되는 것이다.

그리고 protected의 경우 디폴트+ 상속받은 자식의 경우 접근가능 이라는 특성을 지녔는데 상속의 관계에서만 디폴트와 유의미한 차이를 나타낸다. 그런데 상속이란 것은 클래스 내부의 변수와 메서드를 사용하는것이지 클래스 자체를 포함관계로 사용하는 것이 아니다. 따라서 클래스에 protected를 적용하는 것은 의미가 없는 것이다. 이는 default과 다를게 없다.