

능동적 사고 방식의

# java

강사 박주병

Park Ju Byeong

Park Ju Byeong



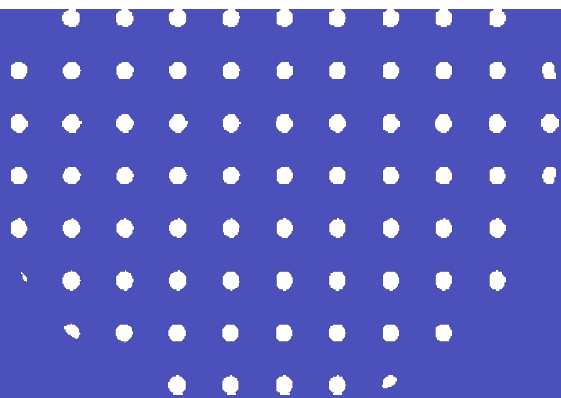
## Part10 객체지향2. —

01 패키지와 `import`

02 제어자

03 접근제어자

04 실습 문제

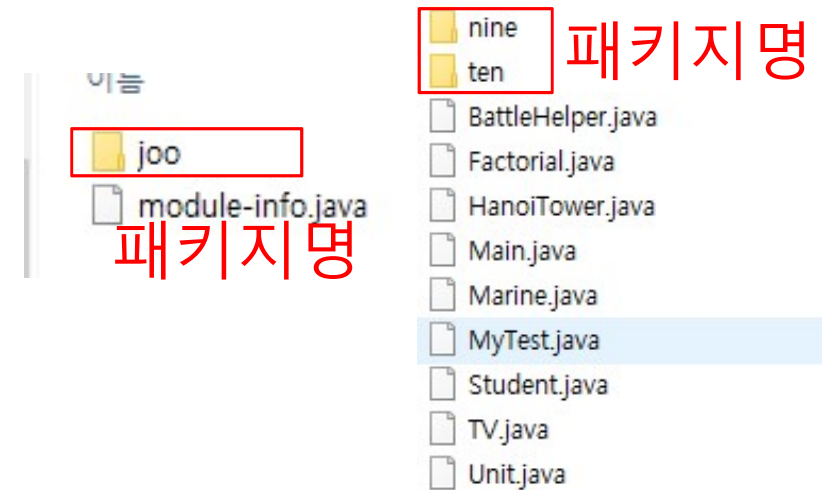
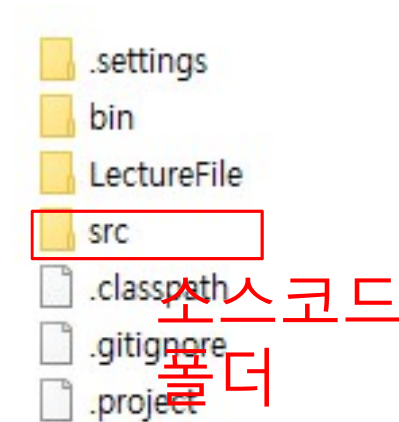
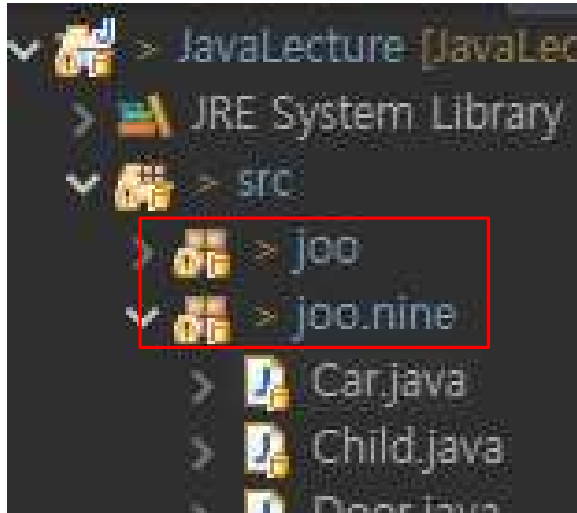


01 —

패키지와 **import**

# 패키지

클래스를 분류하는 폴더

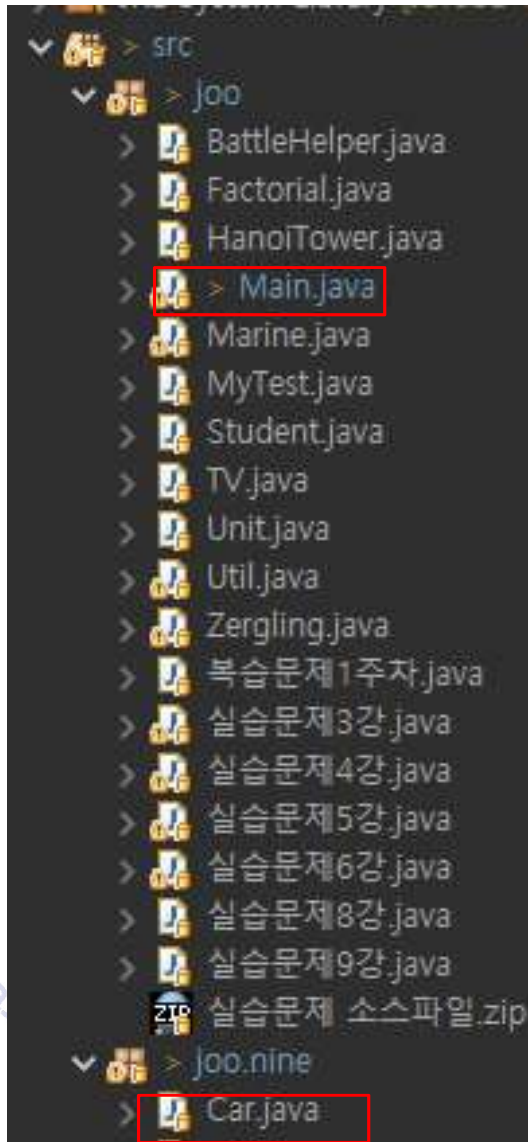


**패키지는 실제로 폴더로 생성되어 관리된다.**

# 엔트리포인트

프로그램의 시작 진입점

```
public class Main {  
    public static void main(String[] args) {  
        프로그램의 시작 지점이 되는 특별한 메서드  
    }  
}
```

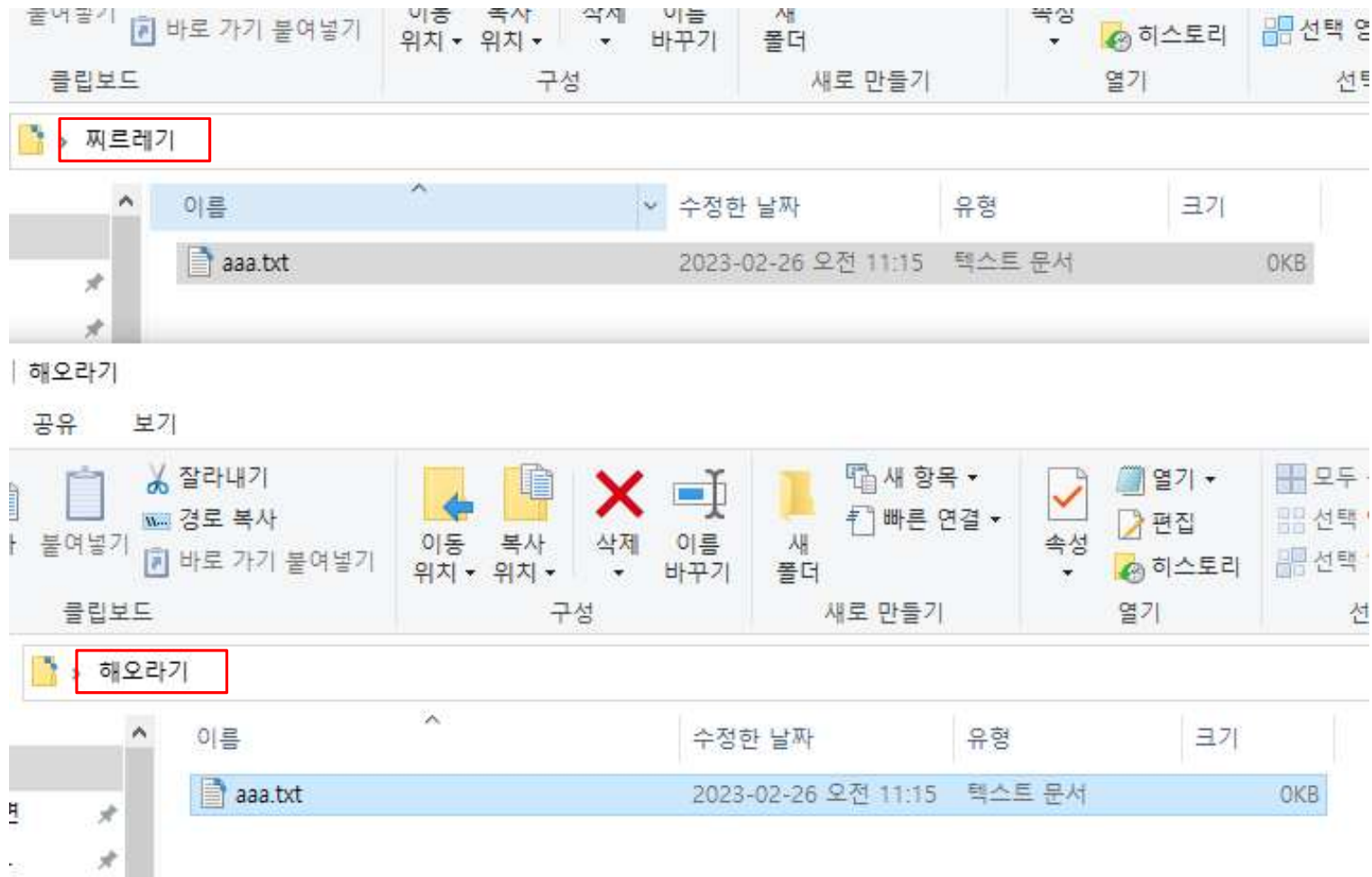


```
1 package joo;  
2  
3 import joo.nine.Car;  
4  
5  
6  
7 public class Main {  
8  
9  
10  
11     public static void main(String[] args) {  
12         |  
13         Car test = new Car();  
14     }
```

현재 파일의 패키지명  
(하나의 패키지명만 가질수  
있다.)

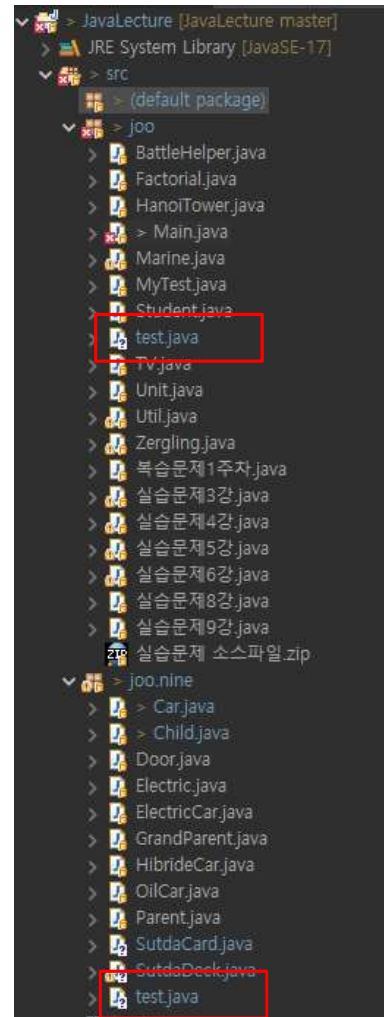
다른 패키지의 클래스를 사용하기 위해서  
**import**를 해야 한다.

왜 쓰는걸까?



**폴더가 다르면 파일명이 같아도 된다.**



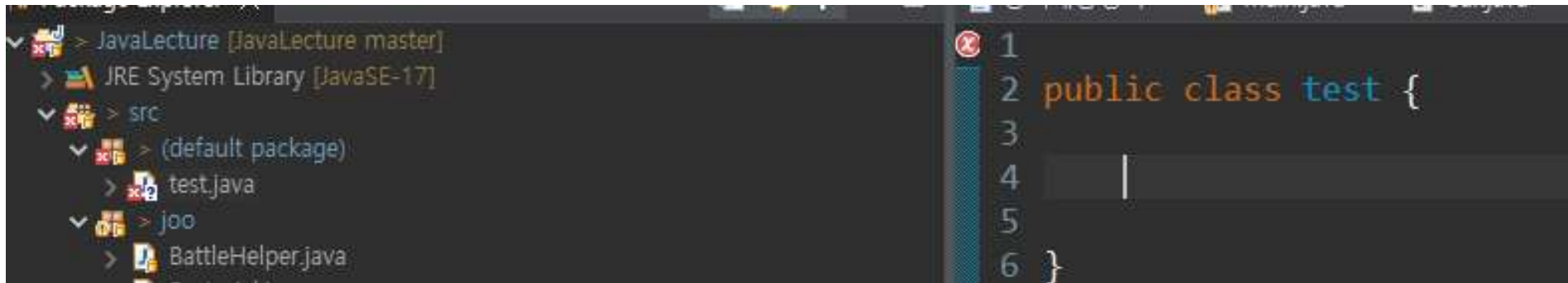


패키지가 다르면 클래스명이 같아도 된다.

## 패키지 규칙

1. 모든 클래스는 반드시 하나의 패키지 안에 속해야 한다.  
(자바 버전에 따라 다를 수 있음)
2. 대소문자 모두 사용 가능하지만 클래스명과의 구분을 위해 소문자만 사용한다.
3. 문법적 제약은 아니지만 일반적으로 도메인 형식으로 만든다.

지금까지 패키지를 만들지 않아도 켜는데??



패키지를 만들지 않으면 자동으로 디폴트 패키지를 만든다.

\* java 버전에 따라 패키지가 없어도 되나  
자바9에서부터 모듈이 추가됨에 따라 무조건 패키지를 만들어야 된다.

# import

다른 패키지의 클래스를 사용할때 사용한다.

```
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();
14     }
```

# import

```
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();
14 }
```

**import** 하지 않아도 패키지명을 다 적어주어 사용가능하다.

import는 성능에 영향을 주지 않는다.



```
4  
5 import joo.nine.Car;  
6 import joo.nine.Child;  
7  
8  
9 public class Main {  
10  
11  
12  
13     public static void main(String[] args) {  
14  
15         Car test = new Car();  
16         Child child = new Child(null, 0);  
17     }  
18 }
```

클래스명

```
4  
5 import joo.nine.*;  
6  
7  
8 public class Main {  
9  
10  
11  
12     public static void main(String[] args) {  
13  
14         Car test = new Car();  
15         Child child = new Child(null, 0);  
16  
17         System.out.print("");  
18     }  
19 }
```

모든 클래스

```
1 package joo;  
2 |  
3 public class Main {  
4  
5  
6  
7 public static void main(String[] args) {  
8  
9     String name = "이름";  
10    System.out.print("");  
11    /*  
12
```

```
5  
6 package java.lang;  
7
```

String, System은 import 하지 않아도 써지는데??



java.lang 패키지는 매우 빈번하게 쓰이므로 자바에서  
자동으로 import 해준다.

## static import

```
System.out.print("");
```



클래스

멤버변수(객체)

PrintStream의 메서드

```
public static final PrintStream out = null;
```

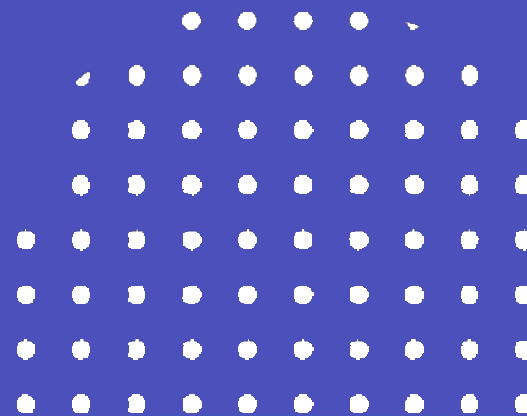
## static import

```
• import static java.lang.Math.random;  
import static java.lang.System.out;  
  
public class Main {  
  
• public static void main(String[] args) {  
    String name = "이름";  
    out.print("");  
    random();|
```

**오히려 가독성을 떨어트리므로 성능의 이점도 없어 잘 사용하지 않는다.**

# — 02

## 제어자



## 제어자

```
graph TD; A([제어자]) -.- B[접근제어자]; A -.- C[그외 키워드]; B --- D[public<br/>protected<br/>private<br/>default]; C --- E[static<br/>final<br/>abstract<br/>...];
```

접근제어자

`public`  
`protected`  
`private`  
`default`

그외 키워드

`static`  
`final`  
`abstract`  
...

```

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("");

```

```

final static public String name="이름";

static public void main(String[] args) {

```

제어자들 끼리 순서는 상관 없으나 일반적으로  
접근제어자를 가장 먼저 작성한다.

**final** : 변하지 않는

변수 -> 상수(변하지않는값)으로 지정한다

```
static final String name="이름";  
  
public static void main(String[] args) {  
  
    name = "파이널 변경";  
}
```

```
final class Card{  
    final int number;  
    Card(int number)  
    {  
        this.number = number;  
    }  
}
```

선언과 동시에 혹은 생성자에서  
딱 한번 초기화 가능



**상수인데 생성자에서 초기화가 가능하게 풀어준건 왜일까요?**

```
Card card1 = new Card();  
Card card2 = new Card();  
Card card3 = new Card();  
  
card1.number = 5;
```

객체 마다 다른값을 가져야 하는 상수일경우...  
불가능해진다.  
**ex)** 주민번호

# final

메서드 -> 오버라이딩 할 수 없다.

```
7  class a{
8
9  public final void test()
10 {
11
12 }
13 }
14
15 class b extends a
16 {
17 public final void test()
18 {
19 |
20 }
21 }
```

# final

클래스 -> 상속될수 없다.

```
final class a{  
    public final void test()  
    {  
    }  
}  
  
class b extends a  
{  
|  
}
```

```
6 */  
7  
8 public final class Math {  
9  
10 /**
```

**abstract** : 추상의, 미완성의  
객체 생성을 못하게 막을때 사용한다.

```
4
5 abstract class Person
6 {
7
8 }
9
10 public class Main {
11
12     public static void main(String[] args) {
13
14         Person p = new Person();
15
16     }
17
18 }
```

추상클래스

```
1  
2  
3  
4  
5 abstract class Person  
6 {  
7     abstract void go();  
8  
9 }  
0
```

추상메서드

메서드 내부를 구현 안 할때에도 쓰인다.

객체를 만들지도 못하는 추상 클래스  
내용이 비어 있는 추상 메서드

왜 필요 할까?

사람, 책, 노트북 등등은 실제로 존재하는것인가?





```
Car car = new Car();
```

현실 세계와는 맞지 않는 코드이다.  
상속의 용도로만 제한을 뒀야 한다.

추상메서드는 왜 필요할까?

```
abstract class Animal
{
    abstract void go();
}

class Tiger extends Animal
{
    추상메서드를 반드시  
오버라이딩 해야 한다.
}
```

```
abstract class Animal
{
    abstract void go();
}

class Tiger extends Animal
{
    void go()
    {
        System.out.println("기어간다");
    }
}
```

```
class Tiger extends Animal
{
    void go()
    {
        System.out.println("기어간다");
    }
}

class Bird extends Animal
{
    void go() {
        System.out.println("날아간다");
    }
}
```

일관된 메서드 시그니처를  
강요할수 있다.  
(더 자세한건 다형성에서 다룬다)

```
class Card
{
    final int number =3;

    Card()
    {

    }

    abstract void shuffle();
}
```



```
abstract class Card
{
    final int number =3;

    Card()
    {

    }

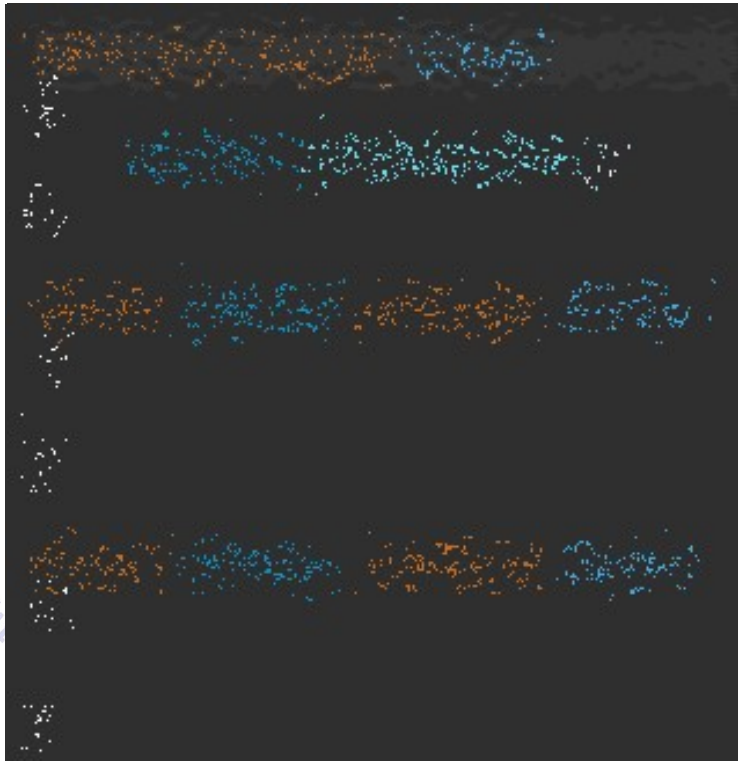
    abstract void shuffle();
}
```

추상 메서드가 있다면 해당 클래스는 인스턴스화 할 수 없다.

**추상메서드를 가진 클래스는 추상클래스가 될 수밖에 없다.**

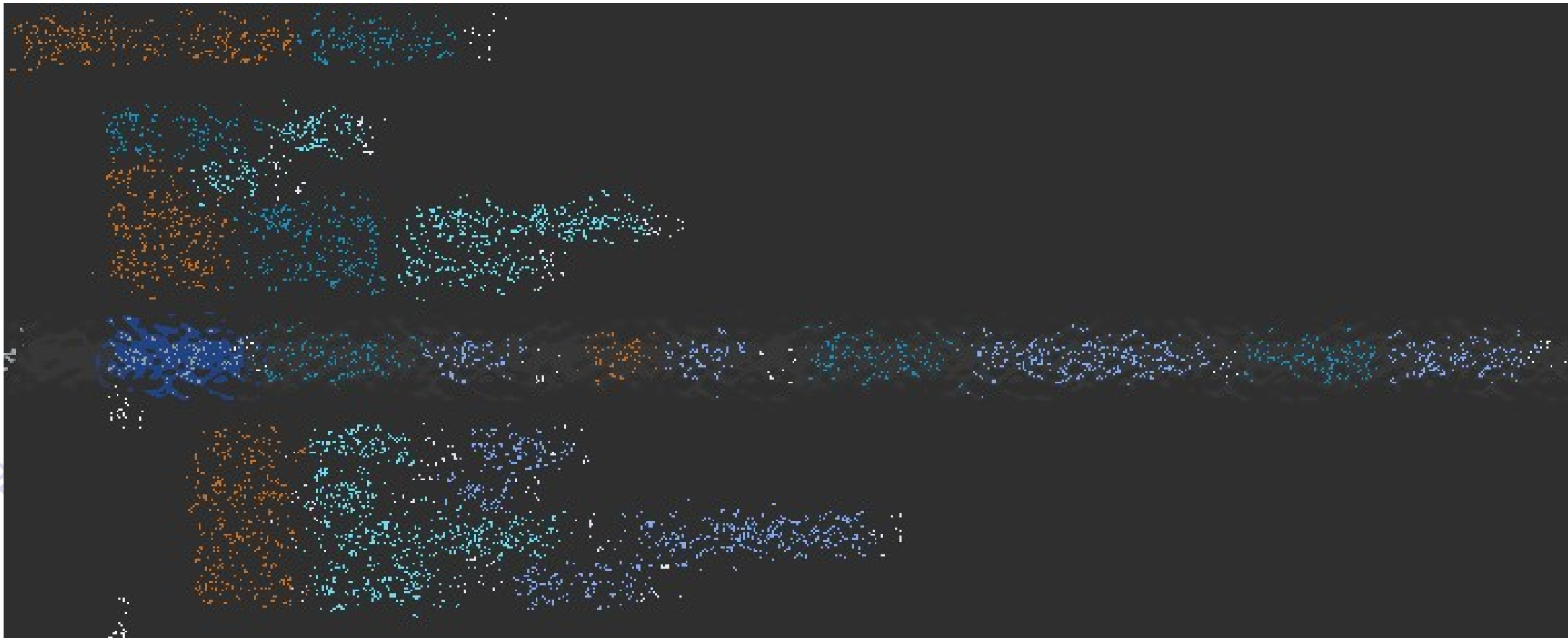
## 실습문제1

1. Phone, Galaxy, iPhone 클래스를 만들자.
  - Galaxy, iPhone 클래스는 Phone 클래스를 상속 받는다.
  - Phone 클래스는 객체화 될 필요가 있는가?
  - 모든 클래스는 **String phoneNumber** 멤버변수를 가진다.



## 2. People 클래스를 만들고 멤버변수마다 적절한 제어자를 사용하자

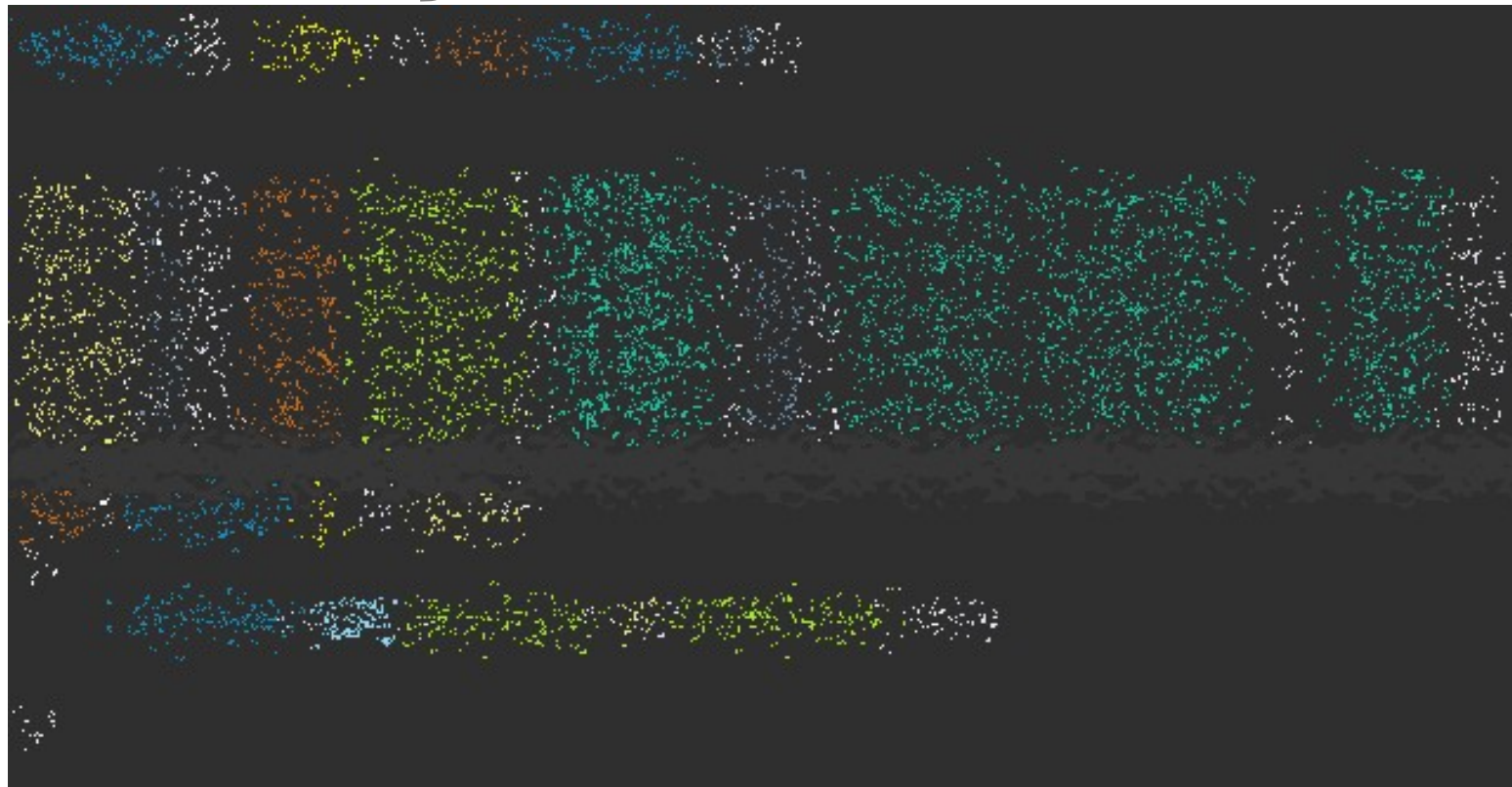
- String name, int age  
  , String juminNumber  
  , String gender



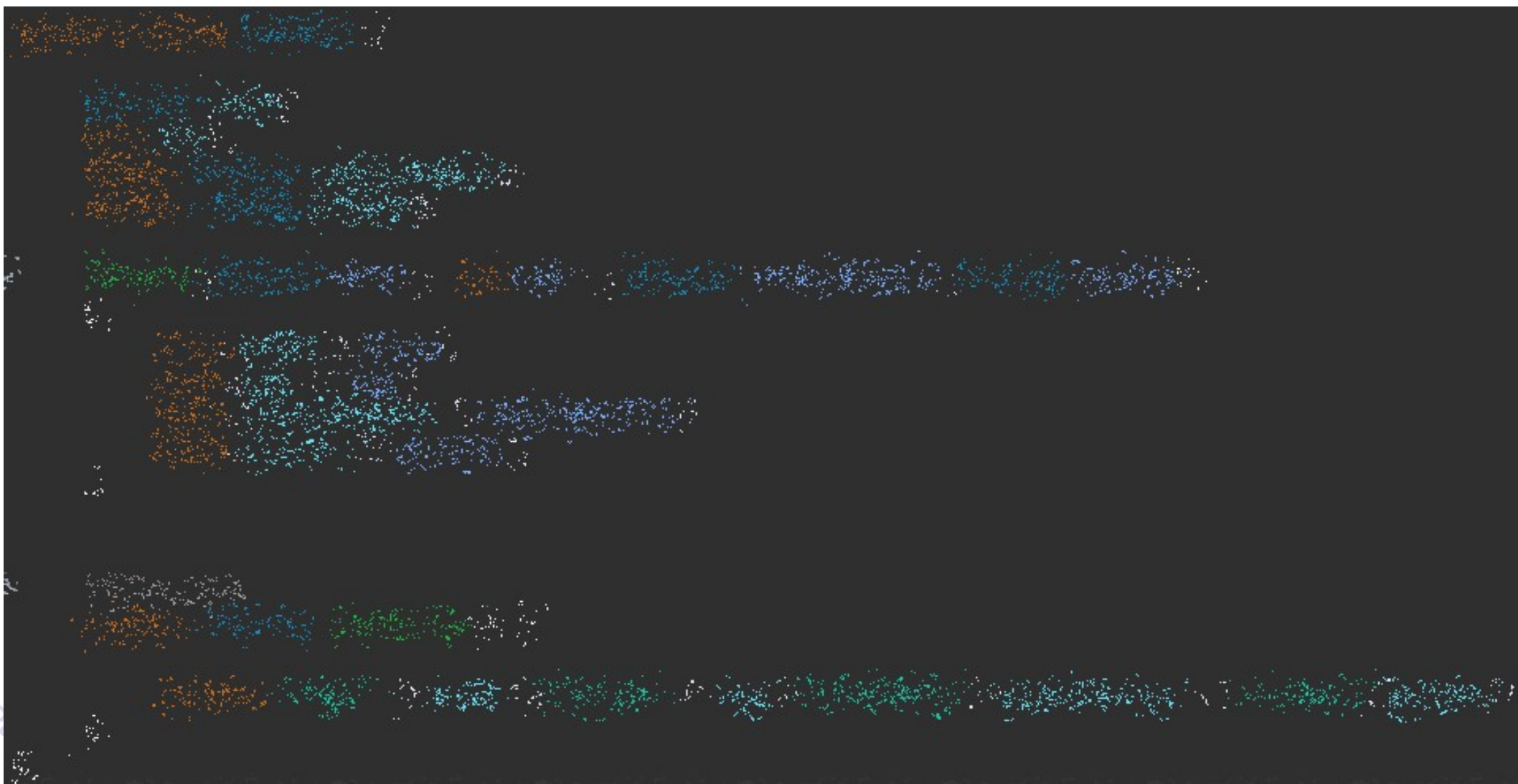


### 3. 앞서 만든 **People** 타입의 객체배열을 만들어 아래와 같이 출력해보자

- **toString()** 을 오버라이딩 해서 사용해보자.



```
<terminated> Main (1) [Java Application] C:\Users\zest1\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win
이름:홍길동1      나이:25      주민번호:000825-3456789      성별:남자
이름:홍길동2      나이:30      주민번호:950825-1456789      성별:남자
이름:홍길동3      나이:25      주민번호:000825-4456789      성별:여자
이름:홍길동4      나이:25      주민번호:000825-2456789      성별:여자
이름:홍길동5      나이:25      주민번호:000825-2456789      성별:여자
```



Park Ju Byeong

Park Ju Byeong



03

접근 제어자

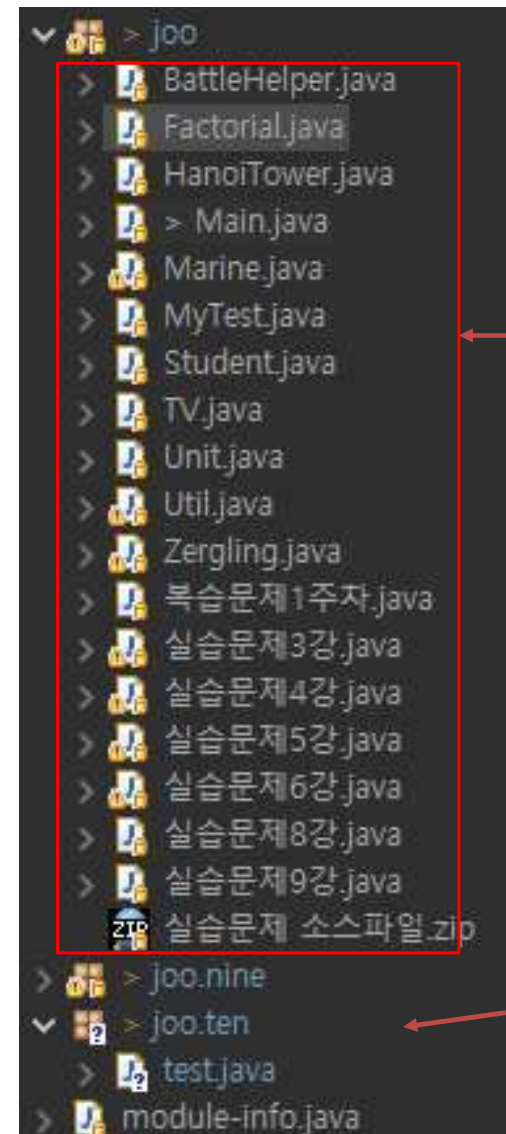


# java의 영역

```
6
7 class a
8 {
9
10
11 void test()
12 {
13
14 }
15
16 }
17
```

← 클래스 내부

← 메서드 내부



← 같은  
패키지 영역

← 외부 패키지

## 접근제어자

**public** : 제한없이 어디서든 접근가능

```
public int a;
```

**protected** : 같은 패키지내에서, 혹은  
다른패키지더라도 자손클래스에서 접근가능

```
protected int a;
```

**default** : 같은 패키지내에서 접근 가능

```
int a;
```

**private** : 같은 클래스 내에서만 접근 가능

```
private int a;
```

| 대상   | 접근제어자           |
|------|-----------------|
| 클래스  | public, default |
| 메서드  | 전부 사용           |
| 멤버변수 | 전부 사용           |

## 멤버변수의 접근제어(private)

```
class test
{
    private String name;

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

public class Main {
    |
    public static void main(String[] args) {

        test t= new test();

        t.name = "외부에서 사용불가능";
    }
}
```

내부에서는 사용가능

**private** 이라서 클래스 외부에서는 사용할수 없다.

## 멤버변수의 접근제어(default)

```
class test
{
    String name;

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

public 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";
    }
}
```

## 멤버변수의 접근제어(protected)

```
2  
3 public class test {  
4  
5  
6     protected String name;  
7  
8  
9 }  
10
```

```
public static void main(String[] args) {  
  
    test t= new test();  
  
    t.name = "디폴트는 다른패키지에서 사용불가";  
}
```

**protected**는 같은 패키지  
혹은 상속받은 자식에서 접근가능(패키지가  
달라도 상속 받은 자손이면 가능)

## 멤버변수의 접근제어(public)

```
4 import joo.ten.test;
5
6
7 public class Main {
8
9     public static void main(String[] args) {
10
11         test t= new test();
12
13         t.name = "public은 어디서든 사용가능";
14     }
```

```
class test
{
    private void add(int a, int b)
    {

    }
}
```

메서드는 멤버변수와 동일하게 적용된다.

**클래스에는 왜 private 과 protected 가 없을까?**

```
private class test  
{  
  
}
```

아무도 해당 클래스를 사용 할 수 없다.

```
protected class
```

→ 같은 패키지내에서 사용가능한거면 **class**에도 적용 되어야 하는거 아닌가?

```
{  
    void test()  
    {  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        test t= new test();  
        t.name = "public은 어디서든 사용가능";  
    }  
}
```

그 역할은 이미 **default**가 하고 있다.

**protected**는 사실상 상속 관계에서 유의미한 접근제어이다.

따라서 클래스 내부의 변수나 메서드들에만 사용해도 그 역할을 다하는것이다.

## 퀴즈

```
class a
{
    private a()
    {
    }
}

class b extends a
{
    b()
    {
    }
}
```

생성자를 **private** 으로 한다면?

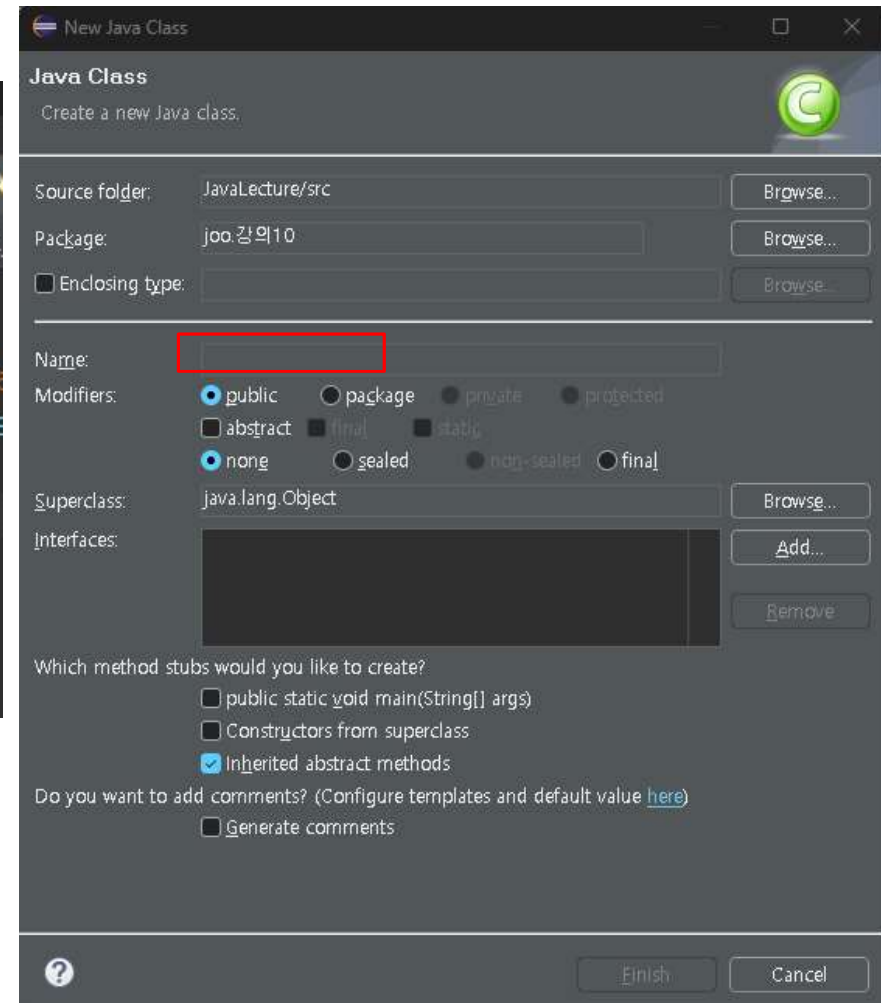
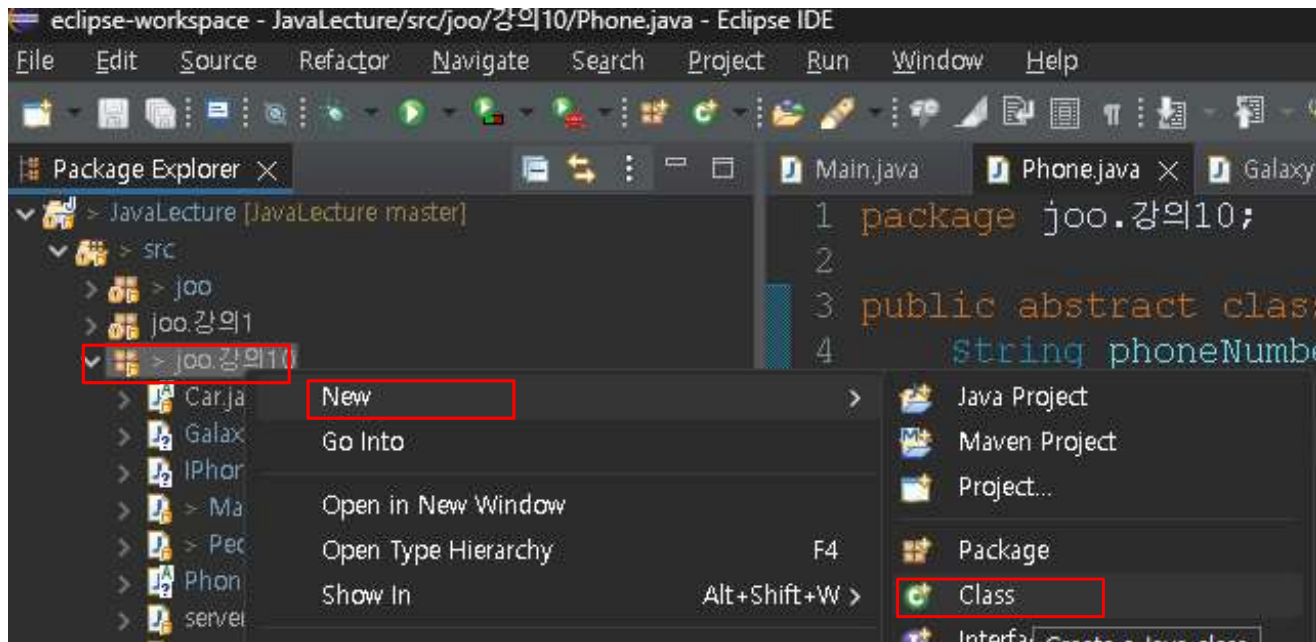
**b**의 생성자가 없으니 **b**의 디폴트 생성자가 만들어진다.

**super()**가 자동으로 삽입되어야 하는데 부모의 생성자가 **private** 으로 접근할수 없다. **ERROR!**

생성자가 **private** 이라면 클래스에 **final**을 적어줌으로써 상속이 불가능하다고 알려주는것이 좋다.



# 클래스 파일 만들기



## 실습문제2

1. 오늘 만든 클래스들을 외부 파일로 빼도록 하자.

- 파일 하나당 여러 개의 클래스를 만들수 있으나 일반적으로 1개의 클래스만 넣는다.
- 파일 하나에는 최소한 1개 이상의 **public** 클래스가 있어야 된다.

```
abstract class Phone
{
    String phoneNumber;
}

class Galaxy extends Phone
{
}

class IPhone extends Phone
{
}
```



## 2. Time 클래스를 만들어 보자.

**hour**는 0~23까지만 가질수 있다. 따라서 멤버변수에 직접 접근을 막고 메서드를 통해서 값을 필터링 해야 한다.

```
Time t= new Time();

t.setHour(30);|
System.out.println(t.toString());

t.setHour(20);
System.out.println(t.toString());

t.setMinute(-50);
System.out.println(t.toString());

t.setMinute(50);
System.out.println(t.toString());

}
```

```
<terminated> main [Java Application] C:\Users\zest1\p2\pool
hour: 0 minute: 0 second: 0
hour: 20 minute: 0 second: 0
hour: 20 minute: 0 second: 0
hour: 20 minute: 50 second: 0
```

| 클래스명 | Time  |                             |
|------|---|-----------------------------|
| 멤버변수 | int hour;                                       | 시                           |
|      | int minute;                                     | 분                           |
|      | int second;                                     | 초                           |
| 메서드  | void setHour(int hour)<br>int getHour()<br>.... | Hour 멤버변수의 값을 셋팅하고 가져오는 메서드 |
|      | String toString()                               | 멤버변수의 값을 문자열로               |



### 3. 싱글톤 패턴 만들기

**serverConnection** 객체를 만들자. 해당 객체는 프로그램이 서버와 통신하기 위한 클래스 이다.

일반적으로 이런 클래스는 프로그램당 한 개의 객체만을 이용하여 서버의 자원을 절약한다.

이러한 구조를 체계화 해놓은 것이 디자인패턴중 싱글톤패턴이다.

**getInstance()** 메서드를 만들어 객체는 항상 1개만 유지되도록 하자.

**Static**과 **private**를 활용하자.

- 생성자를 직접 사용못하게 막아야 한다.
- **getInstance()** 메서드를 통해서만 객체를 가져갈수 있도록 해야 한다.
- **Static**을 활용해보자

```
serverConnection con = serverConnection.getInstance();

serverConnection con1 = serverConnection.getInstance();
serverConnection con2= serverConnection.getInstance();
serverConnection con3 = serverConnection.getInstance();

System.out.println(con);
System.out.println(con1);
System.out.println(con2);
System.out.println(con3);
```

```
<terminated> main [Java Application] C:\Users\Wzest1\p2\pool\plugins
joo.ten.serverConnection@27d415d9
joo.ten.serverConnection@27d415d9
joo.ten.serverConnection@27d415d9
joo.ten.serverConnection@27d415d9
```







# THANK YOU



강사 박주병