2 학기
JAVA Class

'이것이 자바다'

2025.09

Cover Description

made by Lewis

복습 1

1. 출력 결과는 ?

```
3  public class CalcNumberTest {
4
5      public static void main(String[] args) {
6          int iVal1 = 3;
7          int iVal2 = 7;
8          int iVal3 = 9;
9          int iVal4 = 5;
10
11         int iRslt = iVal1++ * ++iVal2 + ++iVal1 / iVal4-- + iVal3 % 3;
12
13         System.out.println("iRslt : " + iRslt);
14     }
15 }
16
```

복습 2

1. 10개의 임의의 정수 (1~10) 를 중복되지 않게 배열에 등록하고 오름차순 소팅하시오.

01 **객체지향 프로그래밍**

객체란?

02 **클래스 생성**

클래스 선언, 생성자 선언, 메소드 선언

컴퓨터가 이해하는 코드는
누구라도 작성할 수 있습니다.
뛰어난 프로그래머는 사람이
이해하는 코드를 작성합니다.

객체지향 프로그래밍이란 ?

1. 개발 방법론 개념
   - 소프트웨어 전 과정에 지속적으로 적용할 수 있는 방법, 절차, 기법
   - 소프트웨어를 하나의 생명체로 간주하고 소프트웨어 개발의 시작부터 시스템을 사용하지 않는 과정까지의 전 과정을 형상화한 방법론이다

2. 개발 방법론 종류
   - 명령형 or 절차적 프로그래밍 (C, Pascal)
   - 객체지향 프로그래밍 (java, Ruby, C++)
   - 선언형 프로그래밍 (SQL)
   - 함수형 프로그래밍
   - 논리형 프로그래밍

절차 지향 프로그래밍

- 절차대로 수행한다.
  => 실행 순서를 중요 시 한다.

- 프로그램의 흐름을 순차적으로 따르며 처리하는 방식
  => “어떻게” 를 중심으로 프로그래밍 한다.

객체 지향 프로그래밍

- 객체를 지향하면 개발한다.
  => 객체를 중요 시 한다.
- 실제 세계의 사물이나 사건을 객체로 보고, 이러한 객체들 간의 상호작용을 중심으로 프로그래밍한다.
  => "무엇을" 를 중심으로 프로그래밍 한다.

➔ 절차 지향은 데이터와 해당 데이터에 대한 처리 방식이 분리되어 있다.
반면 객체 지향에서는 데이터와 그 데이터 에 대한 행동(메서드)이
하나의 '객체' 안에 함께 포함되어 있다.

객체란?

- 객체란?
  - 존재하는 모든 사물 하나하나 모든 것을 표현할 수 있다.
  - 속성과 동작으로 구성된다.
  - 모델링 관점에서는 명확한 의미를 담고 있는 대상 또는 개념
  - 프로그래머 관점에서는 클래스에서 생성된 변수

클래스	객체
자동차	그랜저, K9, 카니발
스마트폰	S25, iPhone17
도형	삼각형, 사각형, 원

객체지향 프로그래밍 장단점

* 장점
  - 코드 재사용 용이 : 상속을 통한 코드의 재사용
  - 생산성 향상 : 독립적인 객체를 사용 함으로서 개발의 생산성 향상
  - 유지보수 편의성 : 추가 및 수정하더라도, 캡슐화를 통해 영향이 적음
  - 대형 프로젝트에 적합 : 클래스 단위 모듈화 하여, 프로젝트 개발할 시 분업화 가능

* 단점
  - 설계 시 많은 시간 소요

객체지향 프로그래밍 특징

캡슐화(encapsulation)
- 데이터와 함수를 하나의 단위로 묶는 것
- 데이터와 코드의 형태를 외부로부터 알 수 없게 하고, 데이터 구조와 역할, 가능을 하나의 캡슐형태로 만드는 방법

추상화 (abstraction)
- 객체의 공통적인 특징(속성과 기능)을 추출하여 정의하는 것 ex) 동물 < 포유류

상속성(inheritance)
- 자식(하위) 클래스가 부모(상위) 클래스의 특성과 기능을 재사용 + 새로운 하위 클래스에 새로운 기능 추가
- 캡슐화를 유지하면서 오버라이딩(overridina : 상속받은 기능만 수정, 재정의)작업을 진행하여 클래스의 재사용이 용이 ➔ 코드 중복 없애기 위함

다형성(polymorphism)
- 하나의 변수, 또는 함수가 상황에 따라 다른 의미로 해석(응답)될 수 있는 것

객체지향 프로그래밍 특징

캡슐화(encapsulation)

- 외부에서 데이터를 직접 변경하면 예측 불가능한 오류 유발

```java
public class Person {
    private int age;

    public void setAge(int age) {
        if (age >= 0 && age <= 150) {
            this.age = age;
        } else {
            System.out.println("나이는 0~100 사이");
        }
    }

    public int getAge() {
        return age;
    }
}
```

```java
public class PersonEach {

    public static void main(String[] args) {
        Person p = new Person();
        p.setAge(-10);
        System.out.println("Age : " + p.getAge());
    }

}
```

객체지향 프로그래밍 특징

추상화 (abstraction), 다형성(polymorphism)

```
abstract class Vehicle {
    abstract void run(); // 추상 메소드 (각 교통수단이 구현)
}

class Car extends Vehicle {
    void run() { System.out.println("도로를 달립니다 🚗"); }
}

class Airplane extends Vehicle {
    void run() { System.out.println("하늘을 납니다 ✈"); }
}

class Ship extends Vehicle {
    void run() { System.out.println("바다를 항해합니다 🚢"); }
}
```

```
public class Main {
    public static void main(String[] args) {
        Vehicle v1 = new Car();
        Vehicle v2 = new Airplane();
        Vehicle v3 = new Ship();

        v1.run();
        v2.run();
        v3.run();
    }
}
```

클래스 생성

클래스 선언

- 객체 생성을 위한 설계도 작성
- 클래스명은 첫 문자를 대문자로 카멜 스타일로 작성한다.
- 첫 문자는 숫자가 될 수 없고 특수문자 중 $, _ 를 포함할 수 있으며 중간에 숫자는 들어갈 수 있다.

클래스 용도

- 라이브러리 클래스
- 실행 클래스 : main 메소드를 가지고 있는 클래스

클래스 구성 요소

```
class Car {                    // 클래스 이름
    private String modelName;  // 필드
    private int modelYear;     // 필드

    Car(String modelName, int modelYear) { // 생성자
        this.modelName = modelName;
        this.modelYear = modelYear;
    }

    public String getModel() { // 메소드
        return this.modelYear + "년식 " + this.modelName + " " + this.color;
    }
}
```

필드
- 클래스에 포함된 변수(variable)
- 선언된 위치에 따라
  1. 클래스 변수(static variable)
  2. 인스턴스 변수(instance variable)
  3. 지역 변수(local variable)

생성자
- 클래스를 가지고 객체를 생성하면, 해당 객체는 메모리에 즉시 생성
- 객체의 생성과 동시에 인스턴스 변수를 원하는 값으로 초기화할 수는 생성자(constructor)라는 메소드를 제공

메소드
- 어떠한 특정 작업을 수행하기 위한 명령문의 집합
- 코드의 반복적인 내용을 하나로 작성 가능
- 모듈화로 가독성이 좋아짐
- 유지보수 시간 단축

클래스 구성 요소 - 필드

```
class Car {
    static int modelOutput; // 클래스 변수
    String modelName;       // 인스턴스 변수

    void method() {
        int something = 10; // 지역 변수
    }
}
```

1. 클래스 변수(static variable)
- static 키워드를 가지는 변수
- 클래스가 메모리에 올라갈 때 생성되고 프로그램 종료 시 소멸

2. 인스턴스 변수(instance variable)
- static 키워드를 가지지 않는 변수
- 인스턴스 생성(new) 시 생성되고 인스턴스 소멸 시 소멸

3. 지역 변수(local variable)
- 메소드나 생성자, 초기화 블록 내에 위치한 변수
- 블록 내에서만 적용 가능

* 초기화하지 않은 필드는 객체 생성 시 자동으로 선언한 타입의 기본 값으로 초기화 된다.

클래스 구성 요소 - 필드

```java
package ch06.sec06.exam02;

public class Car {
    //필드 선언
    String company = "현대자동차";
    String model = "그랜저";
    String color = "검정";
    int maxSpeed = 350;
    int speed;
}
```

int 값 초기화

```java
package ch06.sec06.exam02;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //Car 객체의 필드 값 읽기
        System.out.println("제작회사: " + myCar.company);
        System.out.println("모델명: " + myCar.model);
        System.out.println("색깔: " + myCar.color);
        System.out.println("최고속도: " + myCar.maxSpeed);
        System.out.println("현재속도: " + myCar.speed);

        //Car 객체의 필드 값 변경
        myCar.speed = 60;
        System.out.println("수정된 속도: " + myCar.speed);
    }
}
```

객체생성

Problems Javadoc Declaration Console

<terminated> CarExample [Java Application] C:₩Users₩Lewis₩.p2₩pool₩plugins₩o

```
제작회사: 현대자동차
모델명: 그랜저
색깔: 검정
최고속도: 350
현재속도: 0
수정된 속도: 60
```

클래스 구성 요소 - 생성자

1. **생성자 생성**
   - 반드시 클래스명과 동일한 이름으로 생성
   - 리턴 타입이 없음(메서드와 차이 점)

2. **생성자 선언과 호출**
   - 클래스 변수 = new 클래스(); -> 생성자 호출

3. **모든 클래스는 생성자가 존재**
   - 클래스에 생성자 선언이 없으면 컴파일 시 바이트 코드에 자동 생성됨

```java
package ch06.sec07.exam01;

public class Car {
    //생성자
    Car(String model, String color, int maxSpeed) {
    }
}

```

```java
package ch06.sec07.exam01;

public class CarExample {
    public static void main(String[] args) {
        Car myCar = new Car("그랜저", "검정", 250);
        //Car myCar = new Car();  //기본 생성자 호출 못함
    }
}

```

클래스 구성 요소 – 생성자 오버로딩

*오버로딩 이란?

1. 자바의 한 클래스 내에 이미 사용하려는 이름과 같은 이름을 가진 메소드가 있더라도 **매개변수의 개수 또는 타입이 다르면, 같은 이름을 사용해서 메소드를 정의할 수 있다.**

2. **메소드의 이름이 같고, 매개변수의 개수나 타입이 달라야 한다.**
주의할 점은 '리턴 값만' 다른 것은 오버로딩을 할 수 없다는 것이다.

→ 같은 기능을 하는 메소드를 하나의 이름으로 사용할 수 있다.
메소드의 이름을 절약할 수 있다.

```java
public class Car {
    //생성자
    Car()
    {
        System.out.println("오버로딩1");
    }

    Car(Integer a)
    {
        System.out.println("오버로딩2");
    }

    Car(String a)
    {
        System.out.println("오버로딩3");
    }

    Car(Integer a, Integer b)
    {
        System.out.println("오버로딩4");
    }
}
```

클래스 구성 요소 – 생성자 오버로딩

```java
package ch06.sec07.exam04;

public class Car {
    //필드 선언
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    //생성자 선언
    Car() {}

    Car(String model) {
        this.model = model;
    }

    Car(String model, String color) {
        this.model = model;
        this.color = color;
    }

    Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}
```

```java
package ch06.sec07.exam04;

public class CarExample {
    public static void main(String[] args) {
        Car car1 = new Car();
        System.out.println("car1.company : " + car1.company);
        System.out.println("car1.model : " + car1.model);
        System.out.println("car1.color : " + car1.color);
        System.out.println("car1.maxSpeed : " + car1.maxSpeed);
        System.out.println();

        Car car2 = new Car("자가용");
        System.out.println("car2.company : " + car2.company);
        System.out.println("car2.model : " + car2.model);
        System.out.println();

        Car car3 = new Car("자가용", "빨강");
        System.out.println("car3.company : " + car3.company);
        System.out.println("car3.model : " + car3.model);
        System.out.println("car3.color : " + car3.color);
        System.out.println();

        Car car4 = new Car("택시", "검정", 200);
        System.out.println("car4.company : " + car4.company);
        System.out.println("car4.model : " + car4.model);
        System.out.println("car4.color : " + car4.color);
        System.out.println("car4.maxSpeed : " + car4.maxSpeed);
    }
}
```

클래스 구성 요소 – 생성자 오버로딩

```java
package ch06.sec07.exam05;

public class Car {
    // 필드 선언
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    Car(String model) {
        //20라인 생성자 호출
        this(model, "은색", 250);
    }

    Car(String model, String color) {
        //20라인 생성자 호출
        this(model, color, 250);
    }

    Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}
```

공통 코드를 하나의 생성자에 생성
나머지는 this 로 호출

```java
package ch06.sec07.exam05;

public class CarExample {
    public static void main(String[] args) {
        Car car1 = new Car("자가용");
        System.out.println("car1.company : " + car1.company);
        System.out.println("car1.model : " + car1.model);
        System.out.println();

        Car car2 = new Car("자가용", "빨강");
        System.out.println("car2.company : " + car2.company);
        System.out.println("car2.model : " + car2.model);
        System.out.println("car2.color : " + car2.color);
        System.out.println();

        Car car3 = new Car("택시", "검정", 200);
        System.out.println("car3.company : " + car3.company);
        System.out.println("car3.model : " + car3.model);
        System.out.println("car3.color : " + car3.color);
        System.out.println("car3.maxSpeed : " + car3.maxSpeed);
    }
}
```

클래스 구성 요소 – 메소드

1. 메소드란?
- 어떠한 특정 작업을 수행하기 위한 명령문의 집합

2. 사용 목적
- 중복되는 코드의 반복적인 프로그래밍을 피할 수 있다
- 프로그램에 문제가 발생하거나 기능의 변경이 필요할 때 손쉽게 유지보수를 할 수 있다

3. 메소드 선언

```
리턴타입 메소드명( 매개변수, --- )
{
        실행할 코드를 작성.....
}
```

1.리턴타입 : 메소드가 실행 후 전달하는 결과값 타입
2.메소드명 : 첫문자는 소문자, 캐멀 스타일 권고
3.매개변수 : 메소드를 호출할 때 전달한 매개값
4.실행블록 : 호출 시 실행되는 영역

클래스 구성 요소 – 메소드

```java
package ch06.sec08.exam01;

public class Calculator {
    //리턴값이 없는 메소드 선언
    void powerOn() {
        System.out.println("전원을 켭니다.");
    }

    //리턴값이 없는 메소드 선언
    void powerOff() {
        System.out.println("전원을 끕니다");
    }

    //호출시 두 정수 값을 전달 받고,
    //호출한 곳으로 결과값 int를 리턴하는 메소드 선언
    int plus(int x, int y) {
        int result = x + y;
        return result;   //리턴값 지정;
    }

    //호출시 두 정수 값을 전달 받고,
    //호출한 곳으로 결과값 double을 리턴하는 메소드 선언
    double divide(int x, int y) {
        double result = (double)x / (double)y;
        return result;   //리턴값 지정;
    }
}
```

```java
package ch06.sec08.exam01;

public class CalculatorExample {
    public static void main(String[] args) {
        //Calculator 객체 생성
        Calculator myCalc = new Calculator();

        //리턴값이 없는 powerOn 메소드 호출
        myCalc.powerOn();

        //plus 메소드 호출시 5와 6을 매개값으로 제공하고,
        //덧셈 결과를 리턴받아 result1 변수에 대입
        int result1 = myCalc.plus(5, 6);
        System.out.println("result1: " + result1);

        int x = 10;
        int y = 4;
        //divide 메소드 호출시 변수 x와 y의 값을 매개값으로 제공하고,
        //나눗셈 결과를 리턴받아 result2 변수에 대입
        double result2 = myCalc.divide(x, y);
        System.out.println("result2: " + result2);

        //리턴값이 없는 powerOff 메소드 호출
        myCalc.powerOff();
    }
}
```

클래스 구성 요소 – 메소드 호출(가변길이 매개변수)

```java
package ch06.sec08.exam02;

public class Computer {
    // 가변길이 매개변수를 갖는 메소드 선언
    int sum(int ... values) {
        //sum 변수 선언
        int sum = 0;

        //values는 배열 타입의 변수처럼 사용
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }

        //합산 결과를 리턴
        return sum;
    }
}
```

int 배열

* `int sum(int[] values)`

```java
package ch06.sec08.exam02;

public class ComputerExample {
    public static void main(String[] args) {
        //Computer 객체 생성
        Computer myCom = new Computer();

        //sum 메소드 호출시 매개값 1, 2, 3을 제공하고
        //합산 결과를 리턴받아 result1 변수에 대입
        int result1 = myCom.sum(1, 2, 3);
        System.out.println("result1: " + result1);

        //sum 메소드 호출시 매개값 1, 2, 3, 4, 5를 제공하고
        //합산 결과를 리턴받아 result2 변수에 대입
        int result2 = myCom.sum(1, 2, 3, 4, 5);
        System.out.println("result2: " + result2);

        //sum 메소드 호출시 배열을 제공하고
        //합산 결과를 리턴받아 result3 변수에 대입
        int[] values = { 1, 2, 3, 4, 5 };
        int result3 = myCom.sum(values);
        System.out.println("result3: " + result3);

        //sum 메소드 호출시 배열을 제공하고
        //합산 결과를 리턴받아 result4 변수에 대입
        int result4 = myCom.sum(new int[] { 1, 2, 3, 4, 5 });
        System.out.println("result4: " + result4);
    }
}
```

클래스 구성 요소 – 메소드 호출(return)

```java
package ch06.sec08.exam03;

public class Car {
    //필드 선언
    int gas;

    //리턴값이 없는 메소드로  매개값을 받아서 gas 필드값을 변경
    void setGas(int gas) {
        this.gas = gas;
    }

    //리턴값이 boolean인 메소드로 gas 필드값이 0이면 false를, 0이 아니면 true를 리턴
    boolean isLeftGas() {
        if (gas == 0) {
            System.out.println("gas가 없습니다.");
            return false; // false를 리턴하고 메소드 종료
        }
        System.out.println("gas가 있습니다.");
        return true; // true를 리턴하고 메소드 종료
    }

    //리턴값이 없는 메소드로 gas 필드값이 0이면 return문으로 메소드를 종료
    void run() {
        while (true) {
            if (gas > 0) {
                System.out.println("달립니다.(gas잔량:" + gas + ")");
                gas -= 1;
            } else {
                System.out.println("멈춥니다.(gas잔량:" + gas + ")");
                return; // 메소드 종료
            }
        }
    }
}
```

```java
package ch06.sec08.exam03;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //리턴값이 없는 setGas 메소드 호출
        myCar.setGas(5);

        //isLeftGas 메소드 호출해서 받은 리턴값이 true 일 경우 if 블록 실행
        if(myCar.isLeftGas()) {
            System.out.println("출발합니다.");

            //리턴값이 없는 run 메소드 호출
            myCar.run();
        }

        System.out.println("gas를 주입하세요.");
    }
}
```

```
gas가 있습니다.
출발합니다.
달립니다.(gas잔량:5)
달립니다.(gas잔량:4)
달립니다.(gas잔량:3)
달립니다.(gas잔량:2)
달립니다.(gas잔량:1)
멈춥니다.(gas잔량:0)
gas를 주입하세요.
```

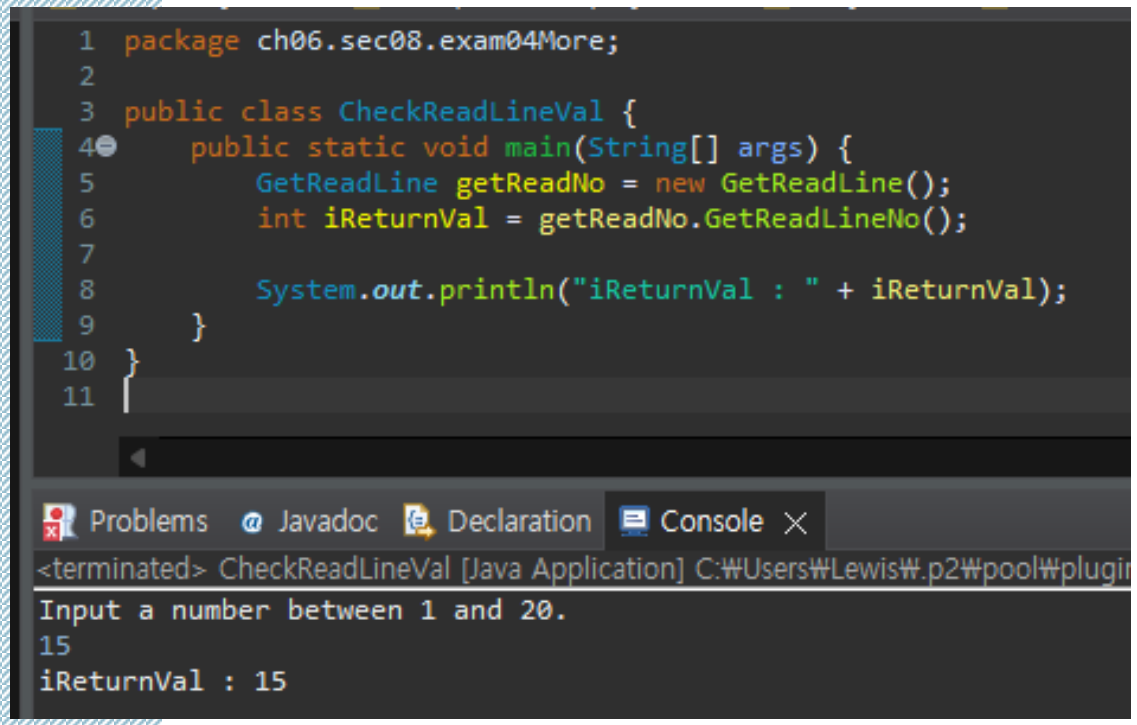
클래스 구성 요소 – 메소드 호출(오버로딩)

```java
package ch06.sec08.exam04;

public class Calculator {
    // 정사각형의 넓이
    double areaRectangle(double width) {
        return width * width;
    }

    // 직사각형의 넓이
    double areaRectangle(double width, double height) {
        return width * height;
    }
}
```

```java
package ch06.sec08.exam04;

public class CalculatorExample {
    public static void main(String[] args) {
        // 객체 생성
        Calculator myCalcu = new Calculator();

        // 정사각형의 넓이 구하기
        double result1 = myCalcu.areaRectangle(10);

        // 직사각형의 넓이 구하기
        double result2 = myCalcu.areaRectangle(10, 20);

        System.out.println("정사각형 넓이=" + result1);
        System.out.println("직사각형 넓이=" + result2);
    }
}
```

클래스 메소드 호출 연습문제 1

GetReadLine 이라는 클래스를 생성하고 1~20 까지의 정수를 입력 받아 입력 받은 수를 리턴하는
GetReadLineNo 메소드를 생성하여 아래와 같이 테스트 하시오

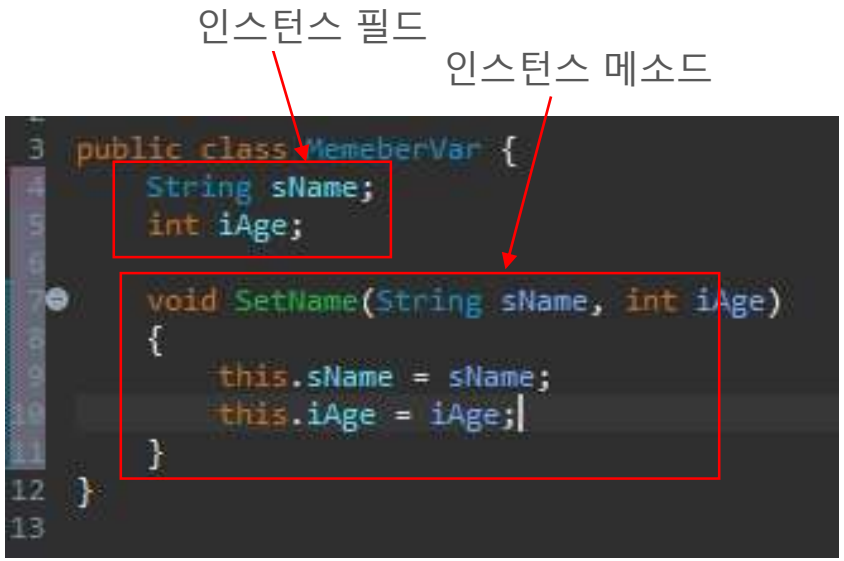


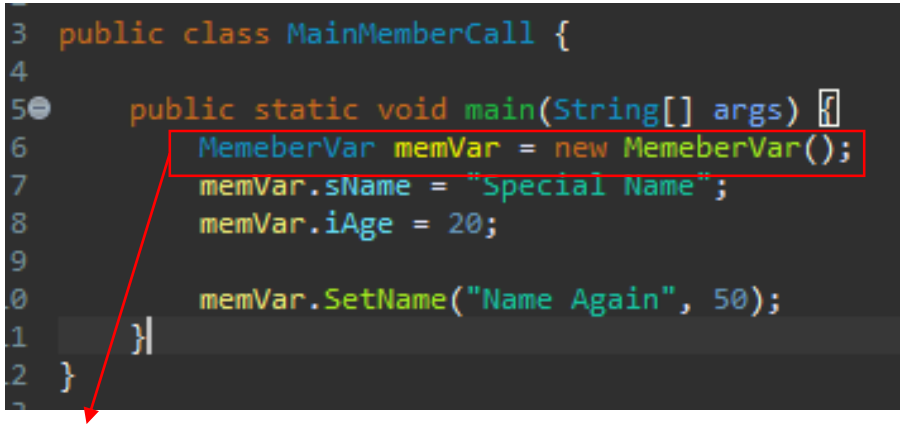
클래스 메소드 호출 연습문제 2

MultipleTable 이라는 클래스를 생성하고 생성자에 정수값이 있으면 해당 구구단을 출력하게 하고
ShowMultipleTable 이라는 메소드를 생성하여 구구단을 출력하게 한다.

인스턴스 멤버

1. **객체에 소속된 멤버**
   - 객체를 생성해야만 사용할 수 있는 멤버
2. **this 키워드**
   - 객체 내부에서 인스턴스 멤버에 접근하기 위한 방법





인스턴스 필드 및 메서드 사용을 위한 객체 생성

정적 멤버

1. 클래스에 고정된 멤버

- 객체를 생성 없이 사용할 수 있는 멤버
- 선언과 동시에 초기값 등록
- 메소드와 필드 모두 정정 멤버 가능
- 객체마다 가지고 있을 필요성이 없는 공용적인 필드를 선언
- 생성자에서 초기화 하지 않음

```java
package ch06.sec10.exam01;

public class Calculator {
    static double pi = 3.14159;

    static int plus(int x, int y) {
        return x + y;
    }

    static int minus(int x, int y) {
        return x - y;
    }
}
```

```java
package ch06.sec10.exam01;

public class CalculatorExample {
    public static void main(String[] args) {
        double result1 = 10 * 10 * Calculator.pi;
        int result2 = Calculator.plus(10, 5);
        int result3 = Calculator.minus(10, 5);

        System.out.println("result1 : " + result1);
        System.out.println("result2 : " + result2);
        System.out.println("result3 : " + result3);
    }
}
```

Problems | Javadoc | Declaration | Console

<terminated> CalculatorExample (1) [Java Application] C:₩Users₩Lewis₩.p2

```
result1 : 314.159
result2 : 15
result3 : 5
```

정적 멤버 - 블록

1. 정정 멤버 초기화 시 사용
2. 메모리 로딩 시 자동 실행

```java
package ch06.sec10.exam02;

public class Television {
    static String company = "MyCompany";
    static String model = "LCD";
    static String info;

    static {
        info = company + "-" + model;
    }
}

```

```java
package ch06.sec10.exam02;

public class TelevisionExample {
    public static void main(String[] args) {
        System.out.println(Television.info);
    }
}

```

Problems Javadoc Declaration Console

<terminated> TelevisionExample [Java Application] C:₩Users₩Lewis

```
MyCompany-LCD
```

정적 멤버 – 인스턴스멤버 사용 불가

1. 객체 생성 없이 사용함으로 내부 인스턴스
   필드/메소드 사용 불가
2. 자신 참조하는 this 사용 불가

```java
package ch06.sec10.exam03;

public class Car {
    // 인스턴스 필드 선언
    int speed;

    // 인스턴스 메소드 선언
    void run() {
        System.out.println(speed + "로 달립니다.");
    }

    static void simulate() {
        this.speed = 10;
        speed = 20;

        // 객체 생성
        Car myCar = new Car();
        // 인스턴스 멤버 사용
        myCar.speed = 200;
        myCar.run();
    }

    public static void main(String[] args) {
        //정적 메소드 호출
        simulate();

        // 객체 생성
        Car myCar = new Car();
        // 인스턴스 멤버 사용
        myCar.speed = 60;
        myCar.run();
    }
}
```

final 필드와 상수

1. final 변수는 한번 선언한 값 변경 불가
2. final 필드 초기화 방법
   - 필드 선언 시 초기값 대입
   - 생성자에서 초기값 대입

```java
package ch06.sec11.exam01;

public class Korean {
    //인스턴스 final 필드 선언
    final String nation = "대한민국";
    final String ssn;

    //인스턴스 필드 선언
    String name;

    //생성자 선언
    public Korean(String ssn, String name) {
        this.ssn = ssn;
        this.name = name;
    }
}
```

```java
package ch06.sec11.exam01;

public class KoreanExample {
    public static void main(String[] args) {
        //객체 생성시 주민번호와 이름 전달
        Korean k1 = new Korean("123456-1234567", "감자바");

        //필드값 읽기
        System.out.println(k1.nation);
        System.out.println(k1.ssn);
        System.out.println(k1.name);

        //Final 필드는 값을 변경할 수 없음
        //k1.nation = "USA";
        //k1.ssn = "123-12-1234";

        //비 final 필드는 값 변경 가능
        k1.name = "감자바";
    }
}
```

Problems Javadoc Declaration Console

<terminated> KoreanExample [Java Application] C:₩Users₩Lewis₩.p2₩pool₩plugi

```
대한민국
123456-1234567
감자바
```

final 필드와 상수

*상수 : 모두가 동일하게 알고 있는 불변의 값
- 객체마다 저장할 필요 없는 값
- 여러 개의 값을 가질 수 없는 것
- static 이면서 final 특성
- 상수명은 모두 대문자 사용

```java
package ch06.sec11.exam02;

public class EarthExample {
    public static void main(String[] args) {
        //상수 읽기
        System.out.println("지구의 반지름: " + Earth.EARTH_RADIUS + " km");
        System.out.println("지구의 표면적: " + Earth.EARTH_SURFACE_AREA + "  km^2");
    }
}
```

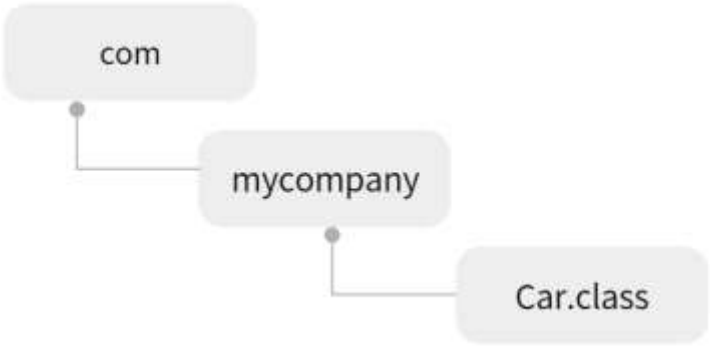
```
Problems  Javadoc  Declaration  Console
<terminated> EarthExample [Java Application] C:₩Users₩Lewis₩.p2₩pool₩plugins₩org.eclipse.justj.openjdk.hots
지구의 반지름: 6400.0 km
지구의 표면적: 5.14718540364151 7E8  km^2
```

```java
package ch06.sec11.exam02;

public class Earth {
    //상수 선언 및 초기화
    static final double EARTH_RADIUS = 6400;

    //상수 선언
    static final double EARTH_SURFACE_AREA = 5.147185403641517E8;

    //정적 블록에서 상수 초기화
//  static {
//      EARTH_SURFACE_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;
//  }
}
```

패키지

- 클래스의 묶음으로 클래스를 용도별이나, 기능별로 그룹화 한 것
- 패키지는 물리적으로 하나의 디렉토리(파일 시스템의 폴더)
- 클래스를 유일하게 만들어주는 식별자 역할
- 같은 이름의 클래스 일지라도 서로 다른 패키지에 존재하는 것이 가능

패키지명 : 상위패키지.하위패키지
패키지 선언 : package 패키지명;



패키지명 : com.mycompany
패키지 선언 : package com.mycompany;

패키지– import 문

- 컴파일러에게 소스파일에 사용된 클래스의 패키지에 대한 정보를 제공하는 것
  ➔ 다른 패키지 안의 클래스를 사용하는 경우 명시
- import문으로 사용하고자 하는 클래스의 패키지를 미리 명시해주면 소스코드에 사용되는 클래스이름에서 패키지명은 생략 가능

*** Import 문 선언**

import 패키지명.클래스명;

import 패키지명.*; -> 지정된 패키지에 속하는 모든 클래스를 패키지명 없이 사용

```java
package ch06.sec12.hankook;

public class SnowTire {
}
```

```java
package ch06.sec12.kumho;

public class AllSeasonTire {
}
```

```java
package ch06.sec12.hankook;

public class Tire {
}
```

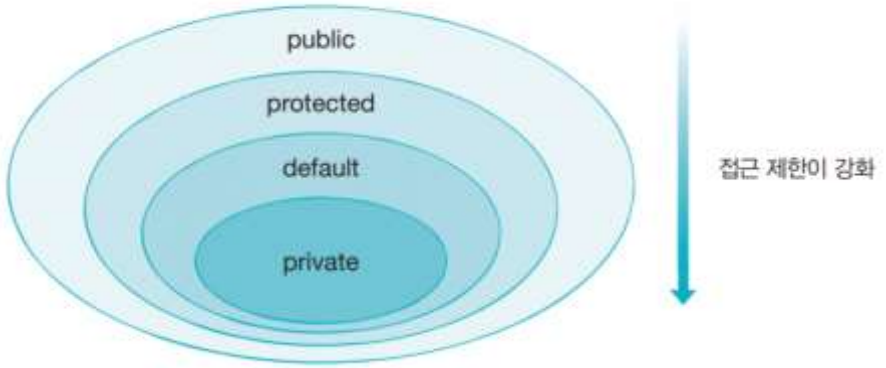
```java
package ch06.sec12.kumho;

public class Tire {
}
```

```java
package ch06.sec12.hyundai;

//import 문으로 다른 패키지 클래스 사용을 명시
import ch06.sec12.hankook.SnowTire;
import ch06.sec12.kumho.AllSeasonTire;

public class Car {
    public static void main(String[] args)
    {
        //부품 필드 선언
        ch06.sec12.hankook.Tire tire1 = new ch06.sec12.hankook.Tire();
        ch06.sec12.kumho.Tire tire2 = new ch06.sec12.kumho.Tire();
        SnowTire tire3 = new SnowTire();
        AllSeasonTire tire4 = new AllSeasonTire();
    }
}
```

접근 제한자

- 클래스와 인터페이스를 다른 패키지에서 사용하지 못하도록 막을 필요가 있을 경우
- 객체 생성을 막기 위해 생성자를 호출하지 못하게 하거나 필드나 메소드를 사용하지 못하도록 막아야 되는 경우

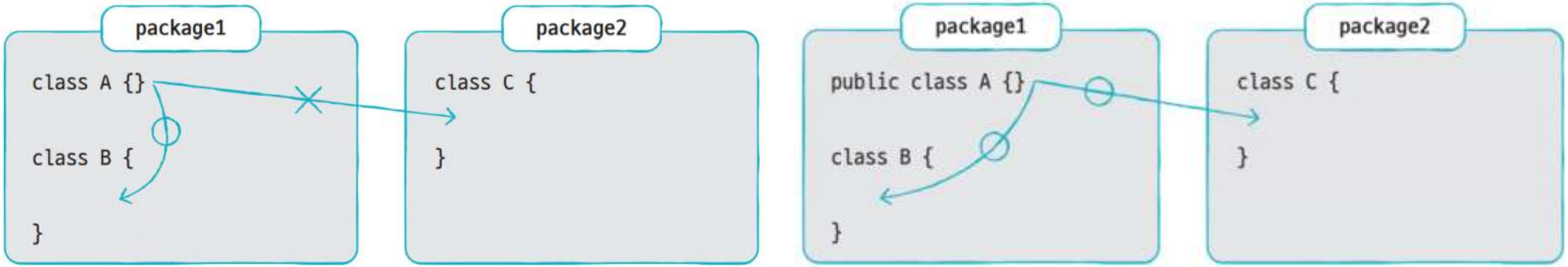
접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스



접근 제한자 - 클래스

```
//default 접근 제한
class 클래스 { … }

//public 접근 제한
public class 클래스 { … }
```

default : 같은 패키지 내에서만 사용할 수 있도록 설정
다른 패키지에서는 사용할 수 없도록 제한
public : 같은 패키지 뿐만 아니라 다른 패키지에서도
아무런 제한 없이 사용할 수 있도록 설정



접근 제한자 - 생성자

```
public class ClassName {
  //public 접근 제한
  public ClassName(…) { … }

  //protected 접근 제한
  protected ClassName(…) { … }

  //default 접근 제한
  ClassName(…) { … }

  //private 접근 제한
  private ClassName(…) { … }
}
```

public : 모든 패키지에서 아무런 제한 없이 생성자를 호출할 수 있음

protected : 같은 패키지에 속하는 클래스에서 생성자를 호출할 수 있음
다른 패키지에 속한 클래스가 해당 클래스의 자식(child) 클래스라면 생성자를 호출할 수 있음

default : 같은 패키지에서는 아무런 제한 없이 생성자를 호출할 수 있음
다른 패키지에서는 생성자를 호출할 수 없음

private : 클래스 내부에서만 생성자를 호출할 수 있고 객체를 만들 수 있음

접근 제한자 - 생성자

```java
package ch06.sec13.exam02.package1;

public class A {
    //필드 선언
    A a1 = new A(true);
    A a2 = new A(1);
    A a3 = new A("문자열");

    //public 접근 제한 생성자 선언
    public A(boolean b) {
    }

    //default 접근 제한 생성자 선언
    A(int b) {
    }

    //private 접근 제한 생성자 선언
    private A(String s) {
    }
}
```

```java
package ch06.sec13.exam02.package1;

public class B {
    // 필드 선언
    A a1 = new A(true);          //o
    A a2 = new A(1);             //o
    //A a3 = new A("문자열");     //x
}
```

```java
package ch06.sec13.exam02.package2;

import ch06.sec13.exam02.package1.*;

public class C {
    // 필드 선언
    A a1 = new A(true);          //o
    //A a2 = new A(1);           //x
    //A a3 = new A("문자열");     //x
}
```

접근 제한자 - 필드, 메소드

public : 모든 패키지에서 아무런 제한 없이 생성자를 호출할 수 있음

protected : 같은 패키지에 속하는 클래스에서 생성자를 호출할 수 있음
다른 패키지에 속한 클래스가 해당 클래스의 자식(child) 클래스라면 생성자를 호출할 수 있음

default : 같은 패키지에서는 아무런 제한 없이 생성자를 호출할 수 있음
다른 패키지에서는 생성자를 호출할 수 없음

private : 클래스 내부에서만 생성자를 호출할 수 있고 객체를 만들 수 있음

접근 제한자 - 필드, 메소드

```java
package ch06.sec13.exam03.package1;

public class A {
    //public 접근 제한을 갖는 필드 선언
    public int field1;
    //default 접근 제한을 갖는 필드 선언
    int field2;
    //private 접근 제한을 갖는 필드 선언
    private int field3;

    // 생성자 선언
    public A() {
        field1 = 1;         //o
        field2 = 1;         //o
        field3 = 1;         //o

        method1();          //o
        method2();          //o
        method3();          //o
    }

    // public 접근 제한을 갖는 메소드 선언
    public void method1() {
    }

    // default 접근 제한을 갖는 메소드 선언
    void method2() {
    }

    // private 접근 제한을 갖는 메소드 선언
    private void method3() {
    }
}
```

```java
package ch06.sec13.exam03.package1;

public class B {
    public void method() {
        //객체 생성
        A a = new A();

        //필드값 변경
        a.field1 = 1;    // o
        a.field2 = 1;    // o
        //a.field3 = 1;      // x

        //메소드 호출
        a.method1();     // o
        a.method2();     // o
        //a.method3();   // x
    }
}
```

```java
package ch06.sec13.exam03.package2;

import ch06.sec13.exam03.package1.*;

public class C {
    public C() {
        //객체 생성
        A a = new A();

        //필드값 변경
        a.field1 = 1;    // (o)
        //a.field2 = 1; // (x)
        //a.field3 = 1; // (x)

        //메소드 호출
        a.method1();     // (o)
        //a.method2();   // (x)
        //a.method3();   // (x)
    }
}
```

Getter, Setter

Getter와 Setter왜 ?

1. 부모 클래스의 멤버 변수는 private로 선언
   → 캡슐화와 자료보호에 대한 목적으로 해당 클래스의 내부에서만 사용할 수 있도록 하기 위함
   ➔ 이러한 부모클래스의 private 멤버 변수에 값을 접근 할때 getter와 setter가 사용
2. 객체의 필드를 외부에서 수정 시 객체 무결성이 깨질 수 있음
   ex) 자동차 속도를 음수로, integer 값에 문자열 등록 등

getter, setter규칙

- private 변수를 다른 클래스에 꺼내는 메서드
  → get + 변수명(첫글자 대문자)
- private 변수에 값을 초기화하는 메서드
  → set + 변수명(첫글자 대문자)

* Eclipse 에서 필드 선택 후 메뉴에서
  [Source] – [Generates Getter and Setters] 선택

```
private 타입 fieldName;    // 필드 접근 제한자 : private

//Getter
public 리턴타입 getFieldName() {
    return fieldName;
}

//Setter
public void setFieldName(타입 fieldName) {
    this.fieldName = fieldName;
}
```

Getter, Setter

```java
package ch06.sec14;

public class Car {
    //필드 선언
    private int speed;
    private boolean stop;

    //speed 필드의 Getter/Setter 선언
    public int getSpeed() {
        return speed;
    }
    public void setSpeed(int speed) {
        if(speed <= 0) {
            this.speed = 0;
            return;
        } else {
            this.speed = speed;
            this.stop = false;
        }
    }

    //stop 필드의 Getter/Setter 선언
    public boolean isStop() {
        return stop;
    }
    public void setStop(boolean stop) {
        this.stop = stop;
        if(stop == true) this.speed = 0;
    }
}
```

```java
package ch06.sec14;

public class CarExample {
    public static void main(String[] args) {
        Car myCar = new Car();

        //잘못된 속도 변경
        myCar.setSpeed(-50);
        System.out.println("현재 속도: " + myCar.getSpeed());

        //올바른 속도 변경
        myCar.setSpeed(60);
        System.out.println("현재 속도: " + myCar.getSpeed());

        //멈춤
        if(!myCar.isStop()) {
            myCar.setStop(true);
        }
        System.out.println("현재 속도: " + myCar.getSpeed());
    }
}
```

Problems @ Javadoc Declaration Console

<terminated> CarExample (4) [Java Application] C:₩Users₩Lewis₩.p2₩pool₩plugins₩org

```
현재 속도: 0
현재 속도: 60
현재 속도: 0
```

싱글톤 패턴

정의

- 객체 지향 프로그래밍에서 특정 클래스가 단 하나의 인스턴스를 생성하여 사용하기위한 패턴

사용 이유

- 커넥션 풀, 스레드 풀, 디바이스 설정 등 인스턴스가 여러 개면 안되는 객체에 사용

싱글톤 패턴 장단점

장점 - 유일한 인스턴스 : 단 하나만 존재 → 객체의 일관된 상태를 유지하고 전역에서 접근 가능
- 메모리 절약 : 인스턴스가 단 하나뿐이므로 메모리를 절약
- 지연 초기화 : 인스턴스가 실제로 사용되는 시점에 생성하여 초기 비용을 줄일 수 있다.

단점 - 결합도 증가 : 전역에서 접근을 허용하기에 해당 인스턴스에 의존하는 경우 결합도가 증가
- 테스트 복잡성 : 단 하나의 인스턴스만을 생성하고 자원을 공유하기에 싱글톤 클래스를 의존하는
클래스는 결합도 증가로 인해 테스트가 어려울 수 있다.
- 상태 관리의 어려움 : 전역에서 사용되어 변경될 수 있어 예상치 못한 동작이 발생할 수 있다.
- 전역에서 접근 가능 : 애플리케이션 내 어디서든 접근이 가능한 경우 복잡성이 증가할 수 있다.

싱글톤 패턴

```java
package ch06.sec15;

public class Singleton {
    //private 접근 권한을 갖는 정적 필드 선언과 초기
    private static Singleton singleton = new Singleton();

    //private 접근 권한을 갖는 생성자 선언
    private Singleton() {
    }

    //public 접근 권한을 갖는 정적 메소드 선언
    static Singleton getInstance() {
        return singleton;
    }
}
```

```java
package ch06.sec15;

public class SingletonExample {
    public static void main(String[] args) {
        /*
        Singleton obj1 = new Singleton();  //컴파일 에러
        Singleton obj2 = new Singleton();  //컴파일 에러
        */

        //정적 메소드를 호출해서 싱글톤 객체 얻
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();

        //동일한 객체를 참조하는지 확
        if(obj1 == obj2) {
            System.out.println("같은 Singleton 객체 입니다.");
        } else {
            System.out.println("다른 Singleton 객체 입니다.");
        }
    }
}
```

Problems Javadoc Declaration Console

<terminated> SingletonExample [Java Application] C:₩Users₩Lewis₩.p2₩pool₩plugins

같은 Singleton 객체 입니다.