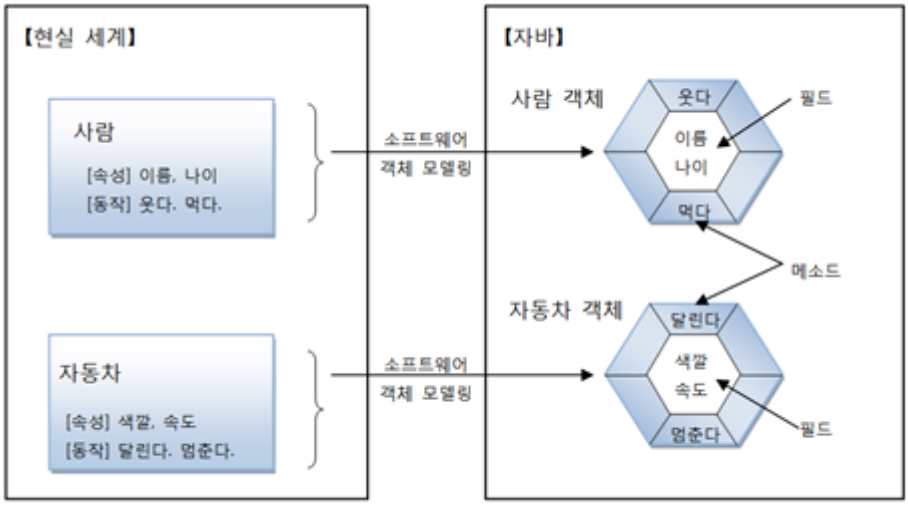
Chapter 06. 클래스

6.1. 객체 지향 프로그래밍

: 부품에 해당하는 객체들을 먼저 만들고, 이것들을 하나씩 조립해서 완성된 프로그램을 만드는 기법

6.1.1) 객체란?

: 자신의 속성을 가지고 있고 다른 것과 식별 가능한 것. 속성과 동작으로 구성되어 있다.



- **필드(field)** : 객체의 속성 , **ex)** 이름, 나이
- **메소드(method)** : 객체의 동작 , **ex)** 먹다, 달린다
- **객체 모델링** : 필드와 메소드를 정의하는 과정

6.1.2) 객체의 상호작용

: 객체들은 각각 독립적으로 존재하고, 다른 객체와 서로 상호작용 하면서 동작한다.

- **상호작용 수단** : 메소드
- **메소드 호출** : 객체가 다른 객체의 기능을 이용하는 것
  - **메소드 사용법**

    ```
    리턴값 = 객체.메소드(매개값1, ...);
    ```

    > 객체에 도트(.) 연산자를 붙인 뒤 메소드 이름을 기술

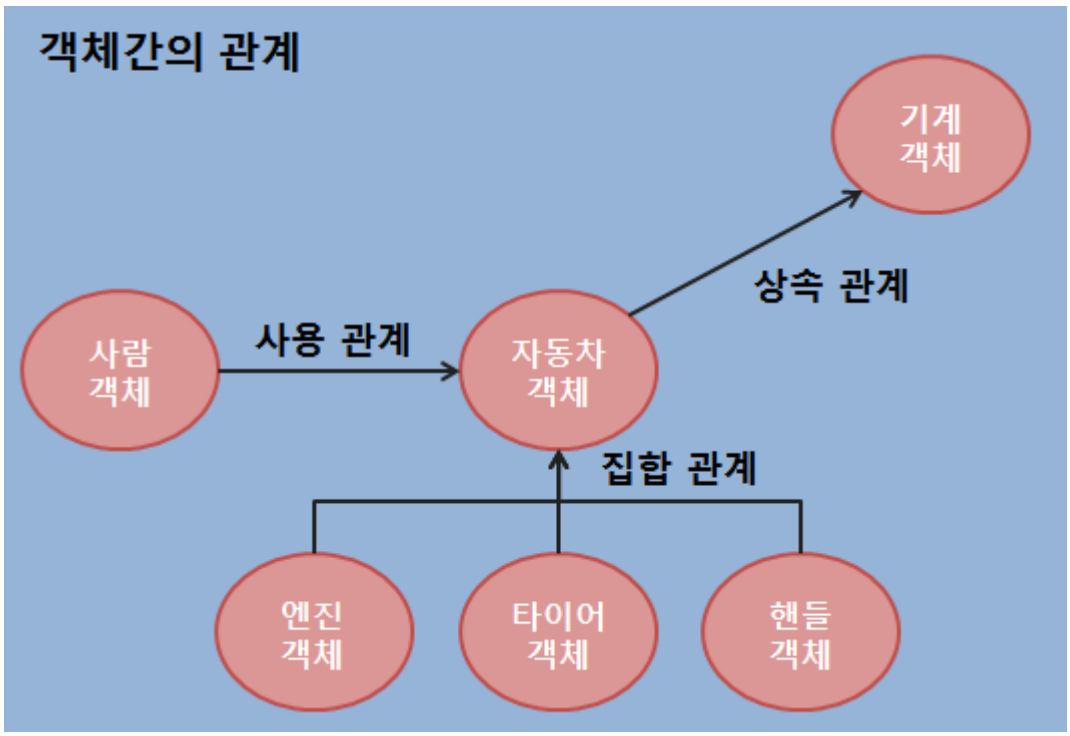
  - **예시**

```
int result = Calcurator.add(10, 20);
```

6.1.3) 객체 간의 관계

: 객체는 대부분 다른 객체와 관계를 맺고 있다.

관계 종류 : 집합 관계, 사용 관계, 상속 관계



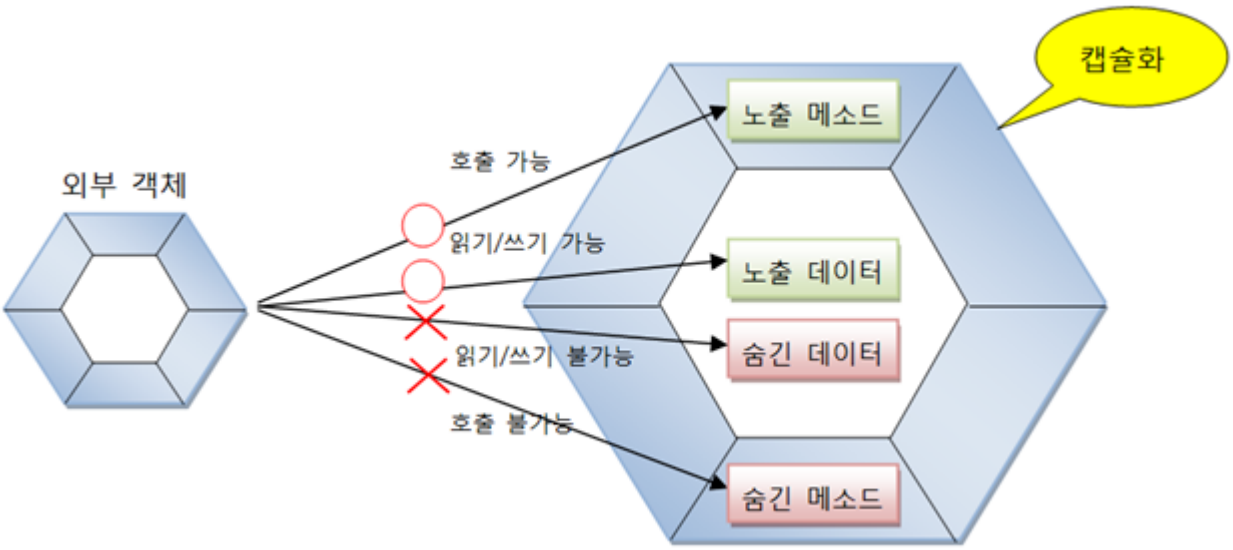
- **집합 관계** : 부품에 해당
- **상속 관계** : 상위(부모) 객체를 기반으로 하위(자식) 객체를 생성하는 관계 **ex)** 기계(상위), 자동차(하위)
- **사용 관계** : 객체 간의 상호작용 **ex)** 사람이 자동차릉 사용한다.

6.1.4) 객체 지향 프로그래밍의 특징

: 캡슐화, 상속, 다형성

캡슐화(Encapsulation)

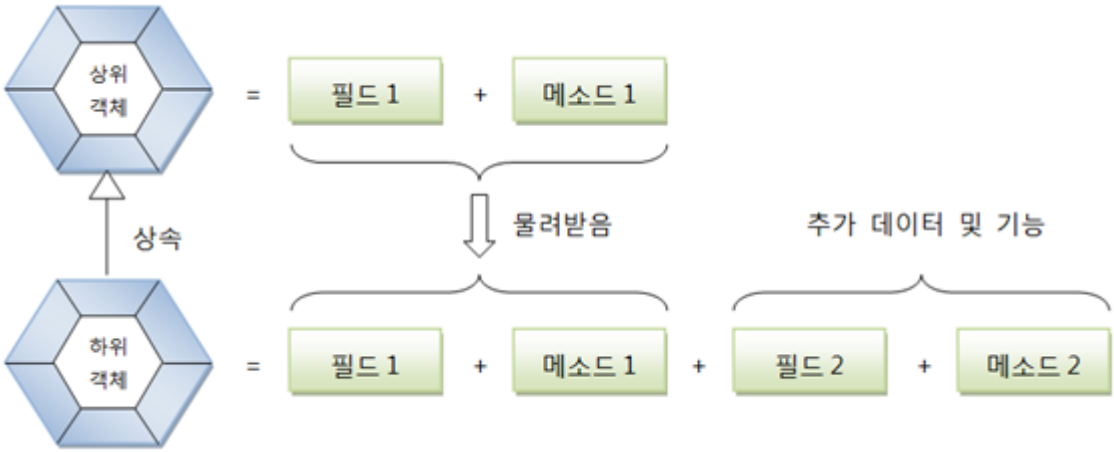
: 객체의 필드, 메소드를 하나로 묶고 실제 구현 내용을 감추는 것.



- **캡슐화하는 이유** : 필드와 메소드를 캡슐화하여 보호하기 위해(외부의 잘못된 사용으로 객체가 손상되지 않도록 함)
- **접근 제한자(Access Modifier)** : 객체의 필드와 메소드의 사용 범위를 제한함으로써 외부로 부터 보호.

상속(Inheritance)

: 부모 역할의 상위 객체가 자식 역할의 하위 객체에게 필드와 메소드를 제공하는 것



- **장점**
  - 상위 객체를 재사용해서 하위 객체를 쉽고 빨리 설계 가능
  - 상위 객체의 수정으로 모든 하위 객체들의 수정 효과로 유지 보수 시간 최소화

다형성(Polymorphism)

: 같은 타입이지만 실행 결과가 다양한 객체를 이용할 수 있는 성질

- **자바의 다형성**
  - 부모 클래스 또는 인터페이스의 타입 변환이 허용된다.

- 하나의 타입에 여러 객체를 대입할 수 있다.

6.2 객체와 클래스

- **클래스** : 객체를 생성하기 위한 필드와 메소드가 정의되어 있다.
  - **인스턴스** : 클래스로부터 만들어진 객체

6.3 클래스 선언

- **작성규칙**
  - 하나 이상의 문자로 이루어져야한다.
  - 첫 번째 글자는 숫자가 올 수 없다.
  - '$', '_' 외의 특수 문자는 사용할 수 없다.
  - 자바 키워드는 사용할 수 없다.
  - 관례적으로 클래스 이름이 단일 단어라면 첫 자를 대문자로 하고 나머지는 소문자로 작성. 서로 다른 단어가 혼합된 이름을 사용한다면 각 단어의 첫 머리 글자는 대문자로 작성.

- **선언 방법**

```
public class 클래스이름{
    ...
}
```

- **선언 예시**

```
public class Car{
    ...
}
```

- **주의할 점**

```
public class Car{
    ...
}
class Tire{
    ...
}
```

> 파일 이름과 동일한 이름의 클래스 선언에만 **public 접근 제한자**를 붙일 수 있다.
>
> 그래서 파일 하나당 동일한 이름의 클래스 하나를 선언하는 것이 좋다.

6.4 객체 생성과 클래스 변수

- **선언 방법**

```
new 클래스();
```

- **new 연산자** : 클래스로부터 객체를 생성시키는 연산자
  - new 연산자 뒤에는 클래스( ) 형태를 가지고 있는 생성자가 온다.
  - new 연산자는 힙 영역에 객체를 생성시킨 후, 객체의 주소를 리턴한다.
- **new 선언 예시**

```
클래스 변수;
변수 = new 클래스();      // 객체의 주소를 변수에 저장

클래스 변수 = new 클래스(); // 변수 선언과 객체 생성 동시에 가능
```

- **클래스 선언 예제**

  Student.java

```
public class Student {
}
```

  StudentExample.java

```java
public class Student {
    public static void main(String[] args) {
        Student s1 = new Student();
        System.out.println("s1 변수가 Student 객체를 참조한다.");

        Student s2 = new Student();
        System.out.println("s1 변수가 또 다른 Student 객체를 참조합니다.");
    }
}
```

실행 결과

```
s1 변수가 Student 객체를 참조한다.
s1 변수가 또 다른 Student 객체를 참조합니다.
```

> - Student 클래스는 하나지만 new 연산자를 사용한 만큼 객체가 메모리에 생성된다. 이러한 객체들은 Student 클래스의 **인스턴스**들이다.
> - s1 과 s2가 참조하는 Student 객체는 완전히 독립된 서로 다른 객체이다.
> - **Student 클래스** : 라이브러리용 클래스
> - **StudentExample 클래스** : 실행 클래스

6.5 클래스의 구성 멤버

구성 멤버 : 필드(Field), 생성자(Constructor), 메소드(Method)

- **클래스**

```java
public class ClassName{

    // 필드
    int fieldName;

    // 생성자
    ClassName(){
        ...
    }

    // 메소드
    void methodName(){
        ...
    }
}
```

6.5.1) 필드

: 객체의 고유 데이터, 상태 정보를 저장하는 곳. 변수(variable)와 비슷하지만 개념이 다르다.

- **변수** : 생성자와 메소드 내에서만 사용되고 생성자와 메소드가 실행 종료되면 자동 소멸
- **필드** : 생성자와 메소드 전체에서 사용되며 객체가 소멸되지 않는 한 객체와 함께 존재

6.5.2) 생성자

: new 연산자로 호출된다. 생성자의 역할은 객체 생성 시 초기화를 담당. 생성자는 메소드와 비슷하게 생겼지만, 클래스 이름으로 되어 있고 리턴 타입이 없다.

6.5.3) 메소드

: 객체의 동작. 객체 간의 데이터 전달의 수단.

6.6 필드

: 객체의 고유 데이터, 상태 데이터.

6.6.1) 필드 선언

: 클래스 블록 어디서든 존재할 수 있다. 하지만 생성자와 메소드 중괄호 블록 내부에는 선언될 수 없다. 생성자와 메소드 중괄호 블록 내부에 선언된 것은 모두 로컬 변수가 된다.

- **선언 방법**

```
타입 빌드 = 초기값;
```

- **선언 예시**

```
String company = "현대자동차";
String model = "그랜저";
int maxSpeed = 300;
int productionYear;
int currentSpeed;
boolean engineStart;
```

- **기본 초기값**
  - **정수 타입, 실수 타입, 논리 타입** : 대부분이 0
  - **참조 타입** : 모두 null

6.6.2) 필드 사용

: 필드값을 읽고, 변경하는 작업

- **사용** : 클래스 내부에서 사용할 경우에는 단순히 필드 이름으로 읽고 변경. 클래스 외부에서 사용할 경우 우선적으로 클래스로부터 객체를 생성한 뒤 필드를 사용해야 한다.

- **필드 사용 예제**

  Car.java

```java
public class Car {
    // 필드
    String company = "현대자동차";
    String mode1 = "그랜져";
    String color = "검정";
    int maxSpeed = 350;
    int speed;
}
```

  CarExample.java

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

        // 필드값 읽기
        System.out.println("제작회사 : " + myCar.company);
        System.out.println("모델명 : " + myCar.mode1);
        System.out.println("색깔 : " + myCar.company);
        System.out.println("최고속도 : " + myCar.mode1);
        System.out.println("현재속도 : " + myCar.speed);

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

  실행 결과

```
제작회사 : 현대자동차
모델명 : 그랜져
색깔 : 현대자동차
최고속도 : 그랜져
현재속도 : 0
수정된 속도 : 60
```

- **필드 자동 초기화 예제**

  FieldInitValue.java

```java
public class FieldInitValue {
    // 필드
    byte byteField;
    short shortField;
    int intField;
    long longField;

    boolean booleanField;
    char charField;

    float floatField;
    double doubleField;

    int[] arrField;
    String referenceField;
}
```

FieldInitValueExample.java

```java
public class FieldInitValueExample {
    public static void main(String[] args) {
        FieldInitValue fiv = new FieldInitValue();

        System.out.println("byteField: " + fiv.byteField);
        System.out.println("shortField: " + fiv.shortField);
        System.out.println("intField: " + fiv.intField);
        System.out.println("longField: " + fiv.longField);
        System.out.println("booleanField: " + fiv.booleanField);
        System.out.println("charField: " + fiv.charField);
        System.out.println("floatField: " + fiv .floatField);
        System.out.println("doubleField: " + fiv.doubleField);
        System.out.println("arrField: " + fiv.arrField);
        System.out.println("referenceField: " + fiv.referenceField);
    }
}
```

실행 결과

```
byteField: 0
shortField: 0
intField: 0
longField: 0
booleanField: false
charField:            // 빈 공백
floatField: 0.0
doubleField: 0.0
arrField: null
referenceField: null
```

6.7 생성자

: new 연산자와 같이 사용되어 클래스로부터 객체를 생성할 때 호출되어 객체의 초기화를 담당한다.

6.7.1) 기본 생성자

: 모든 클래스는 생성자가 반드시 존재한다.

- **생성자 선언**

```
[public] 클래스() { }
```

> **public class**로 선언되면 기본 생성자에서도 public이 붙지만, 클래스가 public 없이 **class로만** 선언되면 기본 생성자에도 public이 붙지 않는다.

6.7.2 생성자 선언

- **명시적으로 생성자 선언**

```
클래스( 매개변수선언, ... ){     // 생성자 블록
    // 객체의 초기화 코드
}
```

> 리턴 타입이 없고 클래스 이름과 동일하다.
>
> **객체 초기화 코드** : 필드에 초기값을 저장하거나 메소드를 호출하여 객체 사용 전에 필요한 준비를 한다.
>
> **매개 변수 생성자** : new 연산자로 생성자를 호출할 때 외부의 값을 생성자 블록 내부로 전달하는 역할

- **매개 변수 생성자 예시**

```
public class Car{
    // 생성자
    Car(String model1, String color, int maxSpeed){
        ...
    }
}
```

- **생성자 선언 예제**

  Car.java

```
public class Car2 {
    String Color;
    int CC;
    // 생성자
    Car2(String color, int cc){
        Color = color;
        CC = cc;
    }
}
```

CarExample.java

```
public class Car2Example {
    public static void main(String[] args) {
        Car2 myCar = new Car2("검점", 3000);
        // Car myCar = new Car(); ( X , 기본 생성자 호출 불가 )

        System.out.println(myCar.Color);
        System.out.println(myCar.CC);
    }
}
```

실행 결과

```
검정
3000
```

6.7.3) 필드 초기화

- **다른 값으로 객체 초기화하는 방법**
  - 필드를 선언할 때 초기값을 주는 방법
  - 생성자에게 초기값을 주는 방법

- **생성자에서 필드 초기화 예제**

  Korean.java

```
public class Korean {
    // 필드
    String nation = "대한민국";
    String name;
    String ssn;

    // 생성자
    public Korean(String n, String s){
        name = n;
        ssn = s;
    }
}
```

KoreanExample.java

```
public class KoreanExample {
    public static void main(String[] args) {
        Korean k1 = new Korean("박자바", "111111-1111111");
        System.out.println("k1.name: " + k1.name);
        System.out.println("k1.ssn: " + k1.ssn);
        System.out.println("k1.nation: " + k1.nation);

        Korean k2 = new Korean("김자바", "222222-2222222");
        System.out.println("k2.name : " + k2.name);
        System.out.println("k2.ssn: " + k2.ssn);
        System.out.println("k2.nation: " + k2.nation);
    }
}
```

실행 결과

```
k1.name: 박자바
k1.ssn: 111111-1111111
k1.nation: 대한민국
k2.name : 김자바
k2.ssn: 222222-2222222
k2.nation: 대한민국
```

> 매개 변수의 이름이 너무 짧으면 가독성이 좋지 않기 때문에 가능하면 초기화시킬 필드 이름과 비슷하게 사용할 것

- **this** : 객체 자신

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

6.7.4) 생성자 오버로딩(Overloading)

: 매개 변수를 달리하는 생성자를 여러 개를 선언하는 것을 말한다.

- **선언 예시**

```
public class Car{
    Car(){ ... }
    Car(String model){ ... }
    Car(String model, String color){ ... }
    Car(String model, String color, int maxSpeed){ ... }
}
```

> **주의할 점** : 선언된 순서가 같을 경우 매개 변수 이름만 바꾸는 것은 생성자 오버로딩이 아니다.

> ex)
>
> ```
> Car(String model, String color){ ... }
> Car(String color, String model){ ... }  // 오버로딩 X
> ```

- **생성자의 오버로딩 예제**

  Car.java

```
public class Car3 {
    // 필드
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    Car3(){

    }

    Car3(String model){
        this.model = model;
    }
    Car3(String model, String color){
        this.model = model;
        this.color = color;
    }
    Car3(String model, String color, int maxSpeed){
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}
```

  CarExample.java

```
public class Car3Example {
    public static void main(String[] args) {
        Car3 car1 = new Car3();
        System.out.println("car1.company: " + car1.company);
        System.out.println();

        Car3 car2 = new Car3("자가용");
        System.out.println("car2.company: " + car2.company);
        System.out.println("car2.model: " + car2.model);
        System.out.println();

        Car3 car3 = new Car3("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();

        Car3 car4 = new Car3("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);
        System.out.println();
    }
}
```

실행결과

```
car1.company: 현대자동차

car2.company: 현대자동차
car2.model: 자가용

car3.company: 현대자동차
car3.model: 자가용
car3.color: 빨강

car4.company: 현대자동차
car4.model: 택시
car4.color: 검정
car4.maxSpeed: 200
```

6.7.5) 다른 생성자 호출(this())

: 생성자에서 다른 생성자를 호출할 때에 사용

```
클래스( [매개 변수 선언, ... ] ){
    this( 매개변수, ... , 값 , ...); // 클래스의 다른 생성자 호출
    실행문;
}
```

- **다른 생성자를 호출해서 중복 코드 줄이기 예제**

  Car.java

```
public class Car4 {
    // 필드
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    // 생성자
    Car4() {
```

```
    }
    Car4(String model){
        this(model,"은색", 250);
    }
    Car4(String model, String color){
        this(model, color, 250);
    }
    Car4(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }

}
```

CarExample.java

```
public class Car4Example {
    public static void main(String[] args) {
        Car4 car1 = new Car4();
        System.out.println("car1.company: " + car1.company);
        System.out.println();

        Car4 car2 = new Car4("자가용");
        System.out.println("car2.company: " + car2.company);
        System.out.println("car2.model: " + car2.model);
        System.out.println();

        Car4 car3 = new Car4("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();

        Car4 car4 = new Car4("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);
        System.out.println();
    }
}
```

실행결과

```
car1.company: 현대자동차

car2.company: 현대자동차
car2.model: 자가용

car3.company: 현대자동차
car3.model: 자가용
car3.color: 빨강

car4.company: 현대자동차
car4.model: 택시
car4.color: 검정
car4.maxSpeed: 200
```

6.8 메소드

: 객체의 동작에 해당한다. 객체 간의 데이터 전달의 수단. 외부로부터 매개값을 받을 수도 있고, 실행 후 어떤 값을 리턴 할 수도 있다.

6.8.1) 메소드 선언

: 선언부(**메소드 시그니처(signature)** : 리턴타입, 메소드이름, 매개변수선언) 와 실행 블록으로 구성

리턴 타입

: 메소드가 실행 후 리턴하는 값의 타입

- **리턴 타입 예시**

```
void powerOn(){ ... }                   // 리턴 타입 없음
double divide(int x, int y){ ... }      // 리턴 타입 있음

powerOn();                          // 리턴 타입이 없으므로 변수에 저장할 내용 X
double result = divide(10, 20);     // 리턴 타입이 없으므로 변수에 저장할 내용 O
int result = divide(10, 20);        // 컴파일 에러 O
divide(10, 20);                     // 컴파일 에러 X
```

메소드 이름

- **메소드 이름 규칙**
  - 숫자로 시작하면 안 되고, $와 _를 제외한 특수 문자 사용 X
  - 관례적으로 메소드명은 소문자로 작성
  - 다른 단어가 혼합된 이름이라면 뒤이어 오는 단어의 첫머리 글자는 대문자로 작성

- **메소드 선언 예시**

```
void run(){ ... }
void startEngine(){ ... }
String getName(){ ... }
int[] getScores(){ ... }
```

- **매개 변수 선언**

  : 필요한 데이터를 외부로부터 받기 위함

- **메소드 선언 예제**

  Calculator.java

```
public class Calculator {

    // 메소드
    void powerOn(){
        System.out.println("전원을 킵니다");
    }

    int plus(int x, int y){
        int result = x + y;
        return result;
    }
    double divide(int x, int y){
        double result = (double)x / (double)y;
        return result;
    }
    void powerOff(){
        System.out.println("전원을 끕니다.");
    }
}
```

  CarculatorExample.java

```
public class CarculatorExample {
    public static void main(String[] args) {
        Calculator myCalc = new Calculator();
        myCalc.powerOn();

        int result1 = myCalc.plus(5, 6);
        System.out.println("result1 : " + result1);

        byte x = 10;
        byte y = 4;

        double result2 = myCalc.divide(x, y);
        System.out.println("result2 : " + result2);
```

```
        myCalc.powerOff();
    }
}
```

실행결과

```
전원을 킵니다
result1 : 11
result2 : 2.5
전원을 끕니다.
```

매개 변수의 수를 모를 경우

: 몇 개의 매개 변수가 입력될지 알 수 없기 때문에 매개 변수의 개수를 결정할 수 없을 것이다. 그러므로 매개 변수를 배열 타입을 선언하여 문제를 해결한다.

- **예시**

```
int sum1(int[] values){ }
```

> sum1( ) 메소드를 호출할 때 배열을 넘겨줌, 배열의 항목 수는 호출할 때 결정.

```
int[] values = { 1, 2, 3 };
int result = sum1(values);
int result = sum1(new int[] { 1, 2, 3, 4, 5 });
```

- **"..." 매개변수**

  : 메소드 호출 시 넘겨준 값의 수에 따라 자동으로 배열이 생성되고 매개값으로 사용

  예시

```
int sum2(int ... values){ }
```

```
int result = sum2(1, 2, 3);
int result = sum2(1, 2, 3, 4, 5);
```

- **매개 변수의 수를 모를 경우 예제**

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

        // 배열을 매개변수로 전달
        int[] values1 = {1, 2, 3};
        int result1 = myCom.sum1(values1);
        System.out.println("result1: " + result1);
```

```
        // 배열 5개를 생성하며 매개변수로 전달
        int result2 = myCom.sum1(new int[]{1, 2, 3, 4, 5});
        System.out.println("result2: " + result2);

        // ... 매개변수가 1, 2, 3 을 배열로 만든다.
        int result3 = myCom.sum2(1, 2, 3);
        System.out.println("result3: " + result3);

        // ... 매개변수가 1, 2, 3, 4, 5 을 배열로 만든다.
        int result4 = myCom.sum2(1, 2, 3, 4, 5);
        System.out.println("result4: " +result4);
    }
}
```

실행결과

```
result1: 6
result2: 15
result3: 6
result4: 15
```

6.8.2) 리턴(return)문

리턴값이 있는 메소드

: 메소드 선언에 리턴 타입이 있는 메소드는 반드시 리턴문을 사용해서 리턴값을 지정해야 한다.

- **자동 타입 변환 리턴 예시**

```
int plus(int x, int y){
    int result = x + y;
    return result;
}

int plus(int x, int y){
    byte result = (byte)(x + y);
    return result;  // 리턴 타입이 int 타입이기 때문에 byte 타입에서 int 타입으로 자동 타입 변환 발생
}
```

- **주의할 점**

  : return문 이후에 실행문은 결코 실행되지 않는 점.

  잘못된 코딩 예시

```
int plus(int x, int y){
    int result = x + y;
    return result;
    System.out.println(result);     // Unreachable code 컴파일 오류 발생
}
```

예외의 경우

```
boolean isLeftGas(){
    if(gas==0){
        System.out.println("gas가 없습니다.");
        return false;
    }
    System.out.println("gas가 있습니다.");
    return true;
}
```

> return false; 는 if의 조건문이 성립해야 실행되기 때문에 return문 이후에 실행문이 있어도 에러를 발생시키지 않는다.

리턴값이 없는 메소드(void)

: void 메소드에서 return문을 사용하면 메소드 실행을 강제 종료시킬 수 있다.

- **예시**

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

- **return 문 예제**

  Car.java

```
public class Car5 {
    // 필드
    int gas;

    // 생성자
```

```
    // 메소드
    void setGas(int gas){
        this.gas = gas;
    }

    boolean isLeftGas(){
        if(gas==0){
            System.out.println("gas가 없습니다.");
            return false;
        }
        System.out.println("gas가 있습니다.");
        return true;
    }

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

CarExample.java

```
public class Car5Example {
    public static void main(String[] args) {
        Car5 myCar = new Car5();

        myCar.setGas(5);

        boolean gasState = myCar.isLeftGas();
        if(gasState){
            System.out.println("출발합니다");
            myCar.run();
        }

        if(myCar.isLeftGas()){
            System.out.println("gas를 주입할 필요가 없습니다.");
        } else {
            System.out.println("gas를 주입하세요.");
        }
    }
}
```

실행결과

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

6.8.3) 메소드 호출

: 메소드는 클래스 내, 외부의 호출에 의해 실행된다. 클래스 외부에서 호출할 경우에는 우선 클래스로부터 객체를 생성한 뒤, 참조 변수를 이용해서 메소드를 호출해야 한다.

객체 내부에서 호출

```
메소드( 매개값, ... );    // 매개 변수의 타입과 수에 맞게 매개값 제공
```

- **예시**

```
public class ClassName{
    void method1( String p1, int p2 ){

    }
    void method2(){
        method1("홍길동", 100);    // method1 을 호출
    }
}
```

- **메소드를 호출하고 리턴값을 받고 싶을 경우**

```
타입 변수 = 메소드(매개값, ...);
```

- **주의할 점**

```
public class ClassName{
    int method1(int x, int y){
        int result = x + y;
        return result;
    }
    void method2(){
        int result1 = method(10, 20);           // result1 = 30
        double result2 = method1(10, 20);       // result2 = 30.0
    }
}
```

> 변수 타입이 메소드 리턴 타입과 동일하거나, 타입 변환이 될 수 있어야 한다.

- **클래스 내부에서 메소드 호출 예제**

  Calculator.java

```
public class Calculator2 {
    int plus(int x, int y){
        int result = x + y;
        return result;
    }

    double avg(int x, int y){
        double sum = plus(x, y);    // plus 호출 후 double 타입 변환
        double result = sum / 2;
        return result;
    }

    void execute(){
        double result = avg(7, 10);     // avg 호출
        println("실행결과: " + result);  // println 호출
    }

    void println(String message){
        System.out.println(message);
    }
}
```

  CalculatorExample.java

```
public class Calculator2Example {
    public static void main(String[] args) {
        Calculator2 myCalc = new Calculator2();
        myCalc.execute();
    }
}
```

  실행결과

```
실행결과: 8.5
```

객체 외부에서의 호출

: 메소드는 객체에 소속된 멤버이므로 객체가 존재하지 않으면 메소드도 존재하지 않기 때문에 우선 클래스로부터 객체를 생성해야 한다.

```
클래스 참조변수 = new 클래스(매개값, ...);
```

- **클래스 외부에서 메소드 호출 예제**

  Car.java

```
public class Car6 {
    // 필드
    int speed;

    // 생성자

    // 메소드
    int getSpeed(){
        return speed;
    }

    void keyTurnOn(){
        System.out.println("키를 돌립니다.");
    }

    void run(){
        for(int i=10; i<=50; i+= 10){
            speed = i;
            System.out.println("달립니다.(시속:"+
                    speed + "km/h");
        }
    }
}
```

  CarExample.java

```
public class Car6Example {
    public static void main(String[] args) {
        Car6 myCar = new Car6();    // 객체 생성
        myCar.keyTurnOn();          // 객체의 메소드 호출
        myCar.run();                // 객체의 메소드 호출
        int speed = myCar.getSpeed();   // 리턴 값이 있는 메소드 호출
        System.out.println("현재 속도: " + speed + "km/h");
    }
}
```

실행결과

```
키를 돌립니다.
달립니다.(시속:10km/h
달립니다.(시속:20km/h
달립니다.(시속:30km/h
달립니다.(시속:40km/h
달립니다.(시속:50km/h
현재 속도: 50km/h
```

6.8.4) 메소드 오버로딩

: 클래스 내에 같은 이름의 메소드를 여러 개 선언하는 것. 메소드 오버로딩의 조건은 매개 변수의 타입, 개수, 순서 중 하나가 달라야 한다.

- **메소드 오버로딩 예제**

  Calculator.java

```java
public class Calculator3 {
    // 정사각형 넓이
    double areaRectangle(double width){
        return width * width;
    }

    // 직사각형 넓이 : 매개 변수의 갯수를 다르게 하여 메소드를 오버로딩 하였다.
    double areaRectangle(double width, double height){
        return width * height;
    }
}
```

  CalculatorExample.java

```java
public class Calculator3Example {
    public static void main(String[] args) {
        Calculator3 myCalcu = new Calculator3();

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

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

        // 결과 출력
        System.out.println("정사각형의 넓이 = " + result1);
        System.out.println("직사각형의 넓이 = " + result2);
    }
}
```

6.9 인스턴스 멤버와 this

- **인스턴스(instance)멤버** : 객체(인스턴스)를 생성한 후 사용할 수 있는 필드와 메소드이다. 인스턴스 필드와 메소드는 객체에 소속된 멤버이기 때문에 객체없이는 사용할 수 없다.

- **인스턴스 멤버와 this 예제 코드**

  Car.java

```
public class Car7 {
    // 필드
    String model;
    int speed;

    // 생성자
    Car7(String model){
        this.model = model;
    }

    // 메소드
    void setSpeed(int speed){
        this.speed = speed;
    }

    void run(){
        for(int i=10; i<=50; i+=10){
            this.setSpeed(i);
            System.out.println(this.model  + " 가 달립니다.(시속: "+
                    this.speed + "km/h");
        }
    }
}
```

  CarExample.java

```
public class Car7Example {
    public static void main(String[] args) {
        Car7 myCar = new Car7("포르쉐");
        Car7 yourCar = new Car7("벤츠");

        myCar.run();
        yourCar.run();
    }
}
```

  실행결과

```
포르쉐 가 달립니다.(시속: 10km/h
포르쉐 가 달립니다.(시속: 20km/h
포르쉐 가 달립니다.(시속: 30km/h
포르쉐 가 달립니다.(시속: 40km/h
포르쉐 가 달립니다.(시속: 50km/h
벤츠 가 달립니다.(시속: 10km/h
벤츠 가 달립니다.(시속: 20km/h
벤츠 가 달립니다.(시속: 30km/h
벤츠 가 달립니다.(시속: 40km/h
벤츠 가 달립니다.(시속: 50km/h
```

6.10 정적 멤버와 static

- **정적 멤버** : 클래스에 고정된 멤버로서 객체를 생성하지 않고 사용할 수 있는 필드와 메소드(객체에 소속된 멤버가 아니라 클래스에 소속된 멤버, 클래스 멤버)

6.10.1) 정적 멤버 선언

- **정적 멤버 선언 방법** : static 키워드를 추가적으로 붙이면 된다.

예시

```
public class 클래스{
    // 정적 필드
    static 타입 필드 { 초기값 };

    // 정적 메소드
    static 리턴 타입 메소드( 매개변수선언, ... ) { ... }
}
```

- **선언할 때**
  - **인스턴스 필드** : 객체마다 가지고 있어야 할 데이터라면 인스턴스 필드로 선언
  - **정적 필드** : 객체마다 가지고 있을 필요성이 없는 공용적인 데이터라면 정적 필드로 선언
  - **예시**

```
public class Calculator{
    String color;                // 계산기별로 색깔이 다를 수 있다.
    static double pi = 3.14159; // 계산기에서 사용하는 파이는 동일하다.

    void setColor(String color){ this.color = color;}   // 인스턴스 메소드
    static int plus(int x, int y){ return x + y; }      // 정적 메소드
    static int minus(int x, int y){ return x - y; }     // 정적 메소드
}
```

1. 객체별로 색깔이 다르면 색깔은 **인스턴스 필드**로 선언한다.
2. 변하지 않는 공용적인 데이터인 pi는 **정적 필드**로 선언한다.
3. 덧셈과 뺄셈은 인스턴스 필드를 이용하기보다는 외부에서 주어진 매개값들을 가지고 덧셈과 뺄셈을 수행하므로 **정적 메소드**로 선언한다.
4. 그러나 인스턴스 필드인 색깔을 변경하는 메소드는 **인스턴스 메소드**로 선언한다.

6.10.2) 정적 멤버 사용

- **사용 예시**

```
클래스.필드;
클래스.메소드( 매개값, ... );    // 도트(.) 연산자로 접근
```

```
public calss Calculator{
    static double pi = 3.14159;
    static int plus(int x, int y){ ... }
    static int minus(int x, int y){ ... }
}

double result1 = 10 * 10 * Calculator.pi;
int result2 = Calculator.plus(10, 5);
int result3 = Calculator.minus(10, 5);

// 정적 필드와 정적 메소드는 원칙적으로는 클래스 이름으로 접근해야 하지만 객체 참조변수로도 접근 가능

Calculator myCalcu = new Calculator();
double result1 = 10 * 10 * myCalcu.pi;
int result2 = myCalcu.plus(10, 5);
int result3 = myCalcu.minus(10, 5);
```

- **정적 멤버 사용 예제**

  Calculator.java

```java
public class Calculator4 {
    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;
    }
}
```

CalculatorExample.java

```java
public class Calculator4Example {
    public static void main(String[] args) {
        double result1 = 10 * 10 * Calculator4.pi;
        int result2 = Calculator4.plus(10, 5);
        int result3 = Calculator4.minus(10, 5);

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

실행결과

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

6.10.3) 정적 초기화 블록

: 정적 필드는 필드 선언과 동시에 초기값을 주는 것이 보통이다.

예시

```java
static double pi = 3.14159;
```

- **정적 블록 초기화**

  : 인스턴스 필드는 생성자에서 초기화하지만, 정적 필드는 객체 생성 없이도 사용해야 하므로 생성자에서 초기화 작업을 할 수 없다. 그래서 정적 블록(static block)을 사용한다.

예시

```
static{
    ...
}
```

> 정적 블록은 메모리가 로딩될 때 자동적으로 실행.

- **정적 초기화 블록 예제**

  Television.java

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

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

  TelevisionExample.java

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

> info 필드는 정적 블록에서 company와 model 필드값을 서로 연결해서 초기값으로 설정한다.

6.10.4) 정적 메소드와 블록 선언 시 주의할 점

: 정적 멤버는 객체가 없이도 실행되기 때문에, 이들 내부에 인스턴스 필드나 인스턴스 메소드를 사용할 수 없다. 또한 this 키워드도 사용할 수 없다.

- **예시**

```
public class ClassName{
    // 인스턴스 필드와 메소드
    int field1;
    void method1(){ ... }

    // 정적 필드와 메소드
    static int field2;
    static void method2(){ ... }

    // 정적 블록
    static{
        field1 = 10;        // x, 컴파일 에러, 인스턴스 멤버이기 때문
```

```
        metho1();           // x, 컴파일 에러, 인스턴스 메소드이기 때문
        field2 = 10;        // o
        method2();          // o
    }

    // 정적 메소드
    static void Method3{
        this.field1 = 10;   // 컴파일 에러, this 사용불가
        this.method1();     // 컴파일 에러
        field2 = 10;        // o
        method2();          // o
    }

    // 정적 메소드에서 인스턴스 멤버를 사용하는 방법
    static void Method3(){
        ClassName obj = new ClassName();
        obj.field1 = 10;
        obj.method1();
    }
}
```

main( ) 메소드 : 메인 메소드도 정적 메소드에 해당한다.

```
public class Car{
    int speed;

    void run(){ ... }

    public static void main(String[] args){
        speed = 60;       // 컴파일 에러
        run();            // 컴파일 에러
    }

    // 수정한 main 메소드
    public static void main(String[] args){
        Car myCar = new Car();
        myCar.speed = 60;     // 컴파일 에러
        myCar.run();          // 컴파일 에러
    }
}
```

- **정적 메소드와 블록 선언 시 주의할 점 예제**

```
public class Car8 {
    int speed;

    void run(){
        System.out.println(speed + "으로 달립니다.");
    }

    public static void main(String[] args) {
        Car8 myCar = new Car8();    // 객체 선언 후 인스턴스 멤버 사용
        myCar.speed = 60;
        myCar.run();
    }
}
```

6.10.5) 싱글톤(Singleton)

: 가끔 전체 프로그램에서 단 하나의 객체만 만들도록 보장해야 하는 경우가 있다. 이 처럼 단 하나의 객체만 생성된다고 해서 이러한 객체를 **싱글톤(Singleton)** 이라고 한다.

싱글톤을 만들려면 new 연산자로 생성자를 호출할 수 없도록 해야한다. 그러기 위해서는 생성자 앞에 **private** 접근 제한자를 붙여주면 된다.

그리고 자신의 타입인 정적 필드를 하나 선언하고 자신의 객체를 생성해 초기화한다. 참고로 클래스 내부에서는 **new 연산자로 생성자 호출이 가능**하다.

외부에서 호출할 수 있는 정적 메소드인 **getInstance( )**를 선언하고 정적 필드에서 참조하고 있는 자신의 객체를 리턴해준다.

- **싱글톤 예시**

```
public class 클래스{
    // 정적 필드
    private static 클래스 singleton = new 클래스();

    // 생성자
    private 클래스();

    // 정적 메소드
    static 클래스 getInstance(){
        return singleton;
    }
}
```

> **getInstance( ) 메소드는 단 하나의 객체만 리턴한다.**

```
클래스 변수1 = 클래스.getInstance();
클래스 변수2 = 클래스.getInstance();
변수1 == 변수2  // 동일한 객체 참조
```

- **싱글톤 예제**

  Singleton.java

```java
public class Singleton {
    private static Singleton singleton = new Singleton();

    private Singleton(){};

    static Singleton getInstance(){
        return singleton;
    }
}
```

  SingletonExample.java

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

        Singleton obj1 = Singleton.getInstance();   // 자신의 객체를 리턴
        Singleton obj2 = Singleton.getInstance();

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

> getInstance( ) 메소드로 자신의 객체를 리턴시켜 obj1 과 obj2가 같은 객체가 된다.

6.11 final 필드와 상수

6.11.1) final 필드

: 초기값이 저장되면 이것이 최종적인 값이 되어서 프로그램 실행 도중에 수정할 수 없다.

```
final 타입 필드 [= 초기값];
```

- **final 필드 초기화**
  1. **필드 선언 시에 주는 방법**
     : 단순 값을 초기화 할 때
  2. **생성자에서 주는 방법**
     : 복잡한 초기화 코드나 객체 생성 시에 외부 데이터로 초기화해야 할 때
  3. **주의할 점**
     : 만약 초기화되지 않은 final 필드를 그대로 남겨두면 컴파일 에러가 발생한다.

- **final 필드 선언과 초기화 예제**

  Person.java

```
public class Person {
    final String nation = "Korea";
    final String ssn;
    String name;

    public Person(String ssn, String name){
        this.ssn = ssn;
        this.name = name;
    }
}
```

  PersonExample.java

```
public class PersonExample {
    public static void main(String[] args) {
        Person p1 = new Person("123456-1234567", "계백");

        System.out.println(p1.nation);
        System.out.println(p1.ssn);
        System.out.println(p1.name);

        // p1.nation = "usa";       컴파일 오류
        // p1.ssn = "64321-121345"; 컴파일 오류
        // final 필드는 값 수정 불가
        p1.name = "을지문덕";
    }
}
```

6.11.2) 상수(static final)

: 불변의 값. final과 다른 점은 **final 필드는 객체마다 저장**되고, 생성자의 매개값을 통해서 여러 가지 값을 가질 수 있기 때문에 상수와 다르다. **상수**는 객체마다 저장되지 않고, **클래스에만 포함**된다. 그래서 **상수는 static이면서 final**이어야 한다.

- **상수 선언**

```
static final 타입 상수 [= 초기값];      // 초기값이 단순할때

static final 타입 상수;                 // 복잡한 초기화일때
static {
    상수 = 초기값;
}
```

> 상수 이름은 모두 대문자로 작성하는 것이 관례이다. 만약 서로 다른 단어가 혼합된 이름이라면 언더바(_)로 단어들을 연결해준다.

- **상수 선언 예제**

  Earth.java

```
public class Earth {
    static final double EARTH_RADIUS = 6400;
    static final double EARTH_SURFACE_AREA;

    static {
        EARTH_SURFACE_AREA = 4 * Math.PI * EARTH_RADIUS *
                EARTH_RADIUS;
    }
}
```

  EarthExample.java

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

  실행결과

```
지구의 반지름: 6400.0 km
지구의 표면력: 5.147185403641517E8 km^2
```

6.12 패키지

: 자바에서는 클래스를 체계적으로 관리하기 위해 패키지(pakage)를 사용한다. 예를 들어 폴더를 만들어 파일을 저장 관리하듯이 패키지를 만들어 클래스를 저장 관리하는 것이다.

패키지는 클래스의 일부분이다. 패키지는 클래스를 유일하게 만들어주는 식별자 역할을 한다. 이름은 같으나 패키지가 다르면 다른 클래스로 인식한다.

- **패키지 선언**

```
상위패키지.하위패키지.클래스      // 도트(.)사용
```

> 패키지가 중요한 이유는 클래스만 따로 복사해서 다른 곳으로 이동하는 것 만으로는 클래스를 사용할 수 없기 때문이다. 그러므로 클래스를 이동할 경우에는 패키지 전체를 이동시켜야 한다.

6.12.1) 패키지 선언

: 패키지는 클래스를 컴파일하는 과정에서 자동적으로 생성되는 폴더이다.

- **선언 예시**

```
package 상위패키지.하위패키지;

public class ClassName { ... }
```

> **선언 규칙**
>
> 1. 숫자로 시작해서는 안되고, _ , $, 를 제외한 특수 문자 사용 불가
> 2. java로 시작하는 패키지는 API에서만 사용하므로 사용해서는 안된다.
> 3. 모두 소문자로 작성하는 것이 관례.
> 4. 선언 마지막에는 프로젝트 이름을 붙여주는 것이 관례
>
>   **ex)** com.samsung.projectname

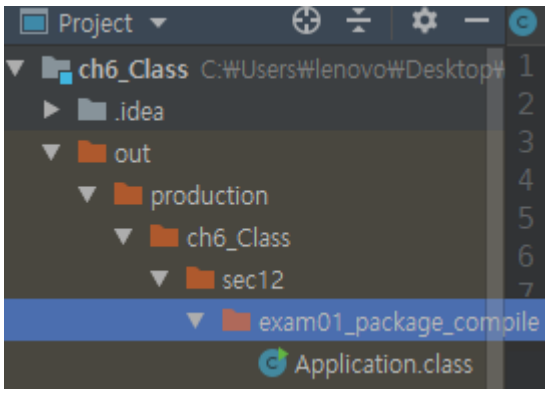
6.12.2) 패키지 선언이 포함된 클래스 컴파일

- **선언 예제**

```
package sec12.exam01_package_compile;

public class Application {
    public static void main(String[] args) {
        System.out.println("애플리케이션을 실행합니다.");
    }
}
```

> 실행과 동시에 패키지가 생성된 것을 볼 수 있다.



6.12.3) 이클립스에서 패키지 생성과 클래스 생성 (생략)

6.12.4) import 문

- **다른 패키지에 속하는 클래스 사용법**
  - 패키지와 클래슬를 모두 기술하는 것

    예시

    ```
    package com.mycompany;

    public class Car{
        com.hankook.Tire tire = new com.hankook.Tire();
     //          타입      필드명           객체 생성
    }
    ```

    > 패키지 이름이 길때는 불편

  - 사용하고자 하는 패키지를 import문으로 선언

    예시

    ```
    package com.mycompany;

    import com.hankook.Tire;    // 또는 import com.hankook.*;

    public class Car{
        Tire tire = new Tire();
    }
    ```

    > import문이 작성되는 위치는 **패키지 선언과 클래스 선언 사이다.**
    >
    > * 는 패키지에 속하는 모든 클래스들을 의미한다.
    >
    > **주의할 점** : import문으로 지정된 패키지의 하위 패키지는 import 대상이 아니므로 하위 패키지에 있는 클래스들도 사용하고 싶다면 import문을 하나 더 작성해야 한다.
    >
    > - **예시**

```
import com.mycompany.*;
import com.mycompany.project.*;
```

서로 다른 패키지에 동일한 클래스 이름이 존재하고, 두 패키지가 모두 import되어 있을 경우 패키지 이름 전체를 기술해야만 한다.

- **예제**

```
package sec12.exam03_import.mycompany;

import sec12.exam03_import.hankook.*;
import sec12.exam03_import.hyndal.Engine;
import sec12.exam03_import.kumho.*;

public class Car{
    // 필드
    Engine engine = new Engine();
    SnowTire tire1 = new SnowTire();
    BigWidthTire tire2 = new BigWidthTire();

    // 두 패키지가 모두 동일한 클래스가 존재할 경우 패키지의 풀네임을 써줘야한다.
    sec12.exam03_import.hankook.Tire tire3 = new sec12.exam03_import.hankook.Tire();
    sec12.exam03_import.kumho.Tire tire3 = new sec12.exam03_import.hankook.Tire();
}
```

6.13 접근 제한자

main( ) 메소드를 가지지 않는 대부분의 클래스는 **외부 클래스에서 이용할 목적**으로 설계된 라이브러리 클래스이다.

- **접근 제한자** : 생성자를 호출하지 못하게 하거나 객체의 특정 데이터를 보호하는 것.

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**

6.13.1) 클래스의 접근 제한

- **클래스를 선언할 때 고려해야 할 사항**
  1. 같은 패키지 내에서만 사용할 것인지
  2. 다른 패키지에서도 사용할 수 있도록 할 것인지

> 클래스에 적용할 수 있는 접근 제한은 **public과 default**
>
> - **예시**
>
> ```
> // default 접근 제한
> class 클래스 { ... }
>
> // public 접근 제한
> public class 클래스 { ... }
> ```

default 접근 제한

: public을 생략하면 클래스는 default 접근 제한을 가진다. 같은 패키지에서는 아무런 제한 없이 사용할 수 있지만 다른 패키지에서는 사용할 수 없도록 제한된다.

public 접근 제한

: 클래스를 선언할 때 public 접근 제한자를 붙이면 된다. public 클래스는 같은 패키지뿐만 아니라 다른 패키지에서도 아무런 제한 없이 사용할 수 있다.

- **접근 제한 예시**

  A.java

```
package sec13.exam01.class.access.package1;

class A {}      // default 접근 제한
```

  B.java

```
package sec13.exam01.class.access.package1;

public class B{
    A a;    // O,   // A 클래스 접근 가능
}
```

  C.java

```
package sec13.exam01_class.access.package2;

import sec13.exam01_class_acccess.package1.*;

public class C{
    A a;    // X, A 클래스 접근 불가( 컴파일 에러 )
    B b;    // O
}
```

6.13.2) 생성자의 접근 제한

: 생성자가 어떤 접근 제한을 갖느냐에 따라 호출 가능 여부가 결정된다.

- **예시**

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

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

    // default 접근 제한
    private ClassName(...) { ... }
}
```

> 자동으로 생성되는 기본 생성자의 접근 제한은 클래스의 접근 제한과 동일하다.

- **생성자의 접근 제한 예제**

  A.java

```
package sec13.exam02_constructor_access.package1;

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

    // 생성자
    public A(boolean b) {}      // public 접근 제한
    A(int b) {}                 // default 접근 제한
    private A(String s) {}      // private 접근 제한
}
```

> public 은 클래스 내부의 모든 생성자를 호출할 수 있다.

B.java

```
package sec13.exam02_constructor_access.package1;

public class B{
    // 필드
    A a1 = new A(true);           // public 생성자 실행, 에러 X
    A a2 = new A(1);          // default 생성자 실행, 에러 X
    A a3 = new A("문자열");    // private 생성자 실행, 에러!!
}
```

> private 는 패키자 동일해도 다른 클래스 이므로 컴파일 에러가 발생한다.

C.java

```
package sec13.exam02_constructor_access.package2; // 패키지가 이전과 다름

import sec13.exam02_constructor_access.package1.A;

public class C {
    // 필드
    A a1 = new A(true);           // public 생성자 실행, 에러 X
    A a2 = new A(1);          // default 생성자 실행, 에러 O
    A a3 = new A("문자열");        // private 생성자 실행, 에러 O
}
```

> default 는 패키지가 다른 클래스의 멤버를 호출할 수 없다.
>
> > private 는 패키지가 다르거나 같아도 다른 클래스의 멤버를 호출할 수 없다.
> >
> > 단 하나의 객체만 만들도록 보장해야 하는 경우에는 **private 접근 제한으로 선언**하고 **getInstance( ) 정적 메소드를** 선언한다. (**싱글톤 패턴**)

6.13.3) 필드와 메소드의 접근 제한

- **필드와 메소드를 선언할 때 고려할 점**
  1. 클래스 내부에서만 사용?
  2. 패키지 내에서만 사용?
  3. 다른 패키지에서도 사용?

  > 이것은 필드와 메소드가 어떤 접근 제한을 갖느냐에 따라 결정.

- **필드와 메소드의 접근 제한 예제**

  A.java

  ```
  package sec13.exam03_filed_method_access.package1;

  public class A{
      // 필드
  ```

```
    public int field1;      // public 접근 제한
    int field2;             // default 접근 제한
    private int field3;     // private 접근 제한

    // 생성자
    public A() {
        // 클래스 내부일 경우 접근 제한자의 영향을 받지 않는다.
        field1 = 1;     // O
        field2 = 1;     // O
        field3 = 1;     // O

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

    public void method1() {}        // public 접근 제한
    void method2() {}               // default 접근 제한
    private void method3() {}       // private 접근 제한
}
```

B.java

```
package sec13.exam03_field_method_access.package1;  // 패키지 동일

public class B {
    public B(){
        A a = new A();
        a.field1 = 1;       // O
        a.field2 = 1;       // O
        a.field3 = 1;       // X    pivate 필드 접근 불가, 컴파일 에러

        a.method1();        // O
        a.method2();        // O
        a.method3();        // X    private 메소드 접근 불가, 컴파일 에러
    }
}
```

C.java

```
package sec13.exam03_field_method_access.package2;  // 패키지가 이전과 다름

import sec13.exam03_field_methode_access.package1.A;

public class{
    public C {
        A a = new A();
        a.field1 = 1;       // O
        a.field2 = 1;       // X, 컴파일 에러
        a.field3 = 1;       // X, 컴파일 에러

        a.method1();        // O
```

```
        a.method2();        // x, 컴파일 에러
        a.method3();        // x, 컴파일 에러
    }
}
```

6.14 Getter와 Setter 메소드

- **Getter 메소드** : 이 메소드를 통해 데이터에 접근하는 것
- **Setter 메소드** : 이 메소드는 매개값을 검증해서 유효한 값만 데이터로 저장할 수 있게 한다.
  - **예제**

```
void setSpeed(double speed){
    if(speed < 0) {     // 매개값이 음수일 경우의 처리
        this.speed = 0;
        return;
    } else {
        this.speed = speed;
    }
}

double getSpeed() {
    // 마일을 km 단위로 환산 후 외부로 리턴
    double km = speed * 1.6;
    return km;
}
```

  - **메소드 선언 예시** : 필드를 private로 보호하고, Setter와 Getter 메소드로 필드값에 접근한다.

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

// Getter
public 리턴 타입 getFieldName(){
    return fieldName;
}

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

- **Getter와 Setter 메소드 선언**

  Car.java

```
public class Car9 {
```

```
    // 필드
    private int speed;
    private boolean stop;

    // 생성자

    // 메소드
    public int getSpeed(){
        return speed;
    }

    public void setSpeed(int speed){
        if(speed < 0){
            this.speed = 0;
            return;
        } else {
            this.speed = speed;
        }
    }

    public boolean isStop() {
        return stop;
    }

    public void setStop(boolean stop){
        this.stop = stop;
        this.speed = 0;
    }
}
```

CarExample.java

```
public class Car9Example {
    public static void main(String[] args) {
        Car9 myCar = new Car9();

        // myCar.speed = 50;
        // speed가 private 접근 제한이여서 컴파일 오류

        // 잘못된 속도 변경
        myCar.setSpeed(-50);

        System.out.println("현재 속도: " + myCar.getSpeed());

        // 올바른 속도 변경
        myCar.setSpeed(60);

        // 멈춤
        if(!myCar.isStop()){
            myCar.setStop(true);
        }

        System.out.println("현재 속도: " + myCar.getSpeed());
```

```
    }
}
```

실행결과

```
현재 속도: 0
현재 속도: 60
현재 속도: 0
```

6.15 어노테이션

- **어노테이션(= 메타데이터)** : 컴파일 과정과 실행 과정에서 **코드를 어떻게 컴파일하고 처리할 것**인지를 알려주는 정보이다.

  어노테이션 작성법

```
@AnnotationName
```

- **어노테이션 용도**
  1. 컴파일러에게 **코드 문법 에러를 체크**하도록 정보를 제공
  2. 소프트웨어 **개발 툴**이 빌드나 배치 시 코드를 자동으로 생성할 수 있도록 정보를 제공
  3. 실행 시(런타임 시) **특정 기능을 실행**하도록 정보를 제공

- **@Override 어노테이션**
  - 컴파일러에게 **코드 문법 에러를 체크**하도록 정보를 제공하는 대표적인 예
  - 메소드 선언시 사용됨
  - **역할** : 메소드가 오버라이드(재정의)된 것임을 컴파일러에게 알려주어 컴파일러가 오버라이드 검사를 하도록 해준다.
  - **빌드 시** : 자동으로 XML 설정 파일을 생성하거나
  - **배포** : JAR 압축 파일을 생성하는데에도 사용

6.15.1) 어노테이션 타입 정의와 적용

- **어노테이션 정의 방법**

```
public @interface AnnotationName{
    ...
}

@AnnotaionName  // 어노테이션 사용 코드
```

> @interface를 사용해서 어노테이션을 정의하고, 그 뒤에 사용할 어노테이션 이름이 온다.

- **엘리먼트 (element)**

```
public @interface AnnotationName{
    타입 elementName() [default 값];   // 엘리먼트 선언
    ...
}
```

> 각 엘리먼트는 타입과 이름으로 구성되며, 디폴트 값을 가질 수 있다.

예시

```
public @interface AnnotationName{
    String elementName1();              // String 타입의 엘리먼트
    int elementName2() default 5;       // int 타입의 엘리먼트
}

// 어노테이션 코드에 적용 예시
@AnnotationName(elementName1 = "값", elementName2 = 3);
@AnnotationName(elementName1 = "값");
```

6.15.2) 어노테이션 적용 대상

- **어노테이션을 적용할 수 있는 대상(열거 상수)**

ElementType 열거 상수	적용 대상
TYPE	클래스, 인터페이스, 열거 타입
ANNOTATION_TYPE	어노테이션
FIELD	필드
CONSTRUCTOR	생성자
METHOD	메소드
LOCAL_VARIABLE	로컬 변수
PACKAGE	패키지

- **@Target 어노테이션** : 어노테이션이 적용될 대상을 지정할 때 사용, 이 어노테이션의기본 엘리먼트인 value는 ElementType 배열을 값으로 가진다.

  적용될 대상을 복수 개로 지정 예시

```
@Target({ElementType.Type, ElementType.FIELD, ElementType.METHOD})
public @interface AnnotationName{
    ...
}
```

```
@AnnotationName
public class ClassName{
    @AnnotationName
    private String fieldName;

    // @AnnotationName     (X), @Target에 CONSTRUCT가 없어 생성자는 적용 못함
    public ClassName(){ }

    @AnnotationName
    public void methodName() { }
}
```

6.15.3) 어노테이션 유지 정책

: 어노테이션 정의 시 @AnnotationName을 어느 범위까지 유지할 것인지 지정해야 한다. 소스상에만 유지할 건지, 컴파일된 클래스까지 유지할 건지, 런타임 시에도 유지할 건지를 지정해야 한다.

- **어노테이션 유지 정책 열거 상수**

RetentionPolicy 열거 상수	설명
SOURCE	소스상에서만 어노테이션 정보를 유지한다.
CLASS	바이트 코드 파일까지 어노테이션 정보를 유지한다.
RUNTIME	런타임시에 어노테이션 정보를 얻을 수 있다.

  - **리플렉션(Reflection)** : 런타임 시에 클래스의 메타 정보를 얻는 기능. 예를 들어 클래스가 가지고 있는 필드, 생성자, 메소드, 적용된 어노테이션이 무엇인지 알아내는 것

- **지정 예시**

```
@Target({ElementType.Type, ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface AnnotationName{
    ...
}
```

6.15.4) 런타임 시 어노테이션 정보 사용하기

: 리플렉션을 이용해서 어노테이션 적용 여부와 엘리먼트 값을 읽고 적절히 처리할 수 있다.

- **필드, 생성자, 메소드에 적용된 어노테이션 정보**

리턴 타입	메소드명(매개 변수)	설명
Field[ ]	getFields( )	필드 정보를 Field 배열로 리턴
Constructor[ ]	getConstructors( )	생성자 정보를 Constructor 배열로 리턴
Method[ ]	getDeclaredMethods( )	메소드 정보를 Method 배열로 리턴

> Class의 위와 같은 메소드를 통해서 java.lang.reflect 패키지의 필드, 생성자, 메소드 타입의 배열을 얻어야한다.

- **적용된 어노테이션 정보 얻는 메소드**

리턴 타입	메소드(매개 변수)
boolean	isAnnotationPresent(Class<? extends Annotation> annotationClass)
	지정한 어노테이션이 적용되었는지 여부, Class에서 호출했을 때 상위 클래스에 적용된 경우도 true를 리턴.
Annotation	getAnnotation(Class annotationClass)
	지정한 어노테이션이 적용되어 있으면 어노테이션을 리턴하고 그렇지 않다면 null을 리턴.
Annotation[]	getAnnotations( )
	적용된 모든 어노테이션을 리턴.
Annotation[]	getDeclaredAnnotations( )
	직접 적용된 모든 어노테이션을 리턴한다.

- **어노테이션 정의 예제**

  PrintAnnotation.java

```
package Annotation;

// @Target과 ElementType을 정의하기 위해 import
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

// @Retention과 RetentionPolicy을 정의하기 위해 import
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.RetentionPolicy;


@Target({ElementType.METHOD})           // 메소드에만 적용
@Retention(RetentionPolicy.RUNTIME)     // 런타임 시까지 어노테이션 정보 유지

public @interface PrintAnnotation{
    String value() default "-";
    int number() default 15;
}
```

Service.java

```
package Annotation;

public class Service {
    @PrintAnnotation        // 엘리먼트의 기본값으로 적용
    public void method1(){
        System.out.println("실행 내용1");
    }

    @PrintAnnotation("*")   // 기본 엘리먼트인 value 값을 "*"로 설정
    public void method2(){
        System.out.println("실행 내용2");
    }

    @PrintAnnotation(value="#", number=20)  // value 값을 "#"으로, number 값을 20으로
설정
    public void method3(){
        System.out.println("실행 내용3");
    }
}
```

PrintAnnotationExample.java

```
package Annotation;

import java.lang.reflect.Method;        // 리플렉션을 사용하기 위함

public class PrintAnnotationExample {
    public static void main(String[] args) {
        // Service 클래스에 선언된 메소드 얻기(리플렉션)
        // 메소드 배열을 생성 후 Service 클래스에 선언된 메소드를 가져와 저장한다.
        Method[] declaredMethods = Service.class.getDeclaredMethods();

        // Method 객체를 하나씩 처리
        for(Method method : declaredMethods){   // 메소드 객체를 하나씩 불러와 method에
저장하여 처리한다.

            // PrintAnnotation이 적용되었는지 확인
            // 지정한 클래스에 method 어노테이션이 적용되었는지 확인
            if(method.isAnnotationPresent(PrintAnnotation.class)){
```

```
                // PrintAnnotation 객체 얻기
                // PrintAnnotation 객체에 method 어노테이션을 리턴
                PrintAnnotation printAnnotation =
method.getAnnotation(PrintAnnotation.class);

                // 메소드 이름 출력
                System.out.println("[" + method.getName() + "] ");

                // 구분선 출력
                for(int i=0; i<printAnnotation.number(); i++){
                    System.out.print(printAnnotation.value());
                }
                System.out.println();
            }

            try{
                // 메소드 호출
                method.invoke(new Service());
            } catch (Exception e){ }
            System.out.println();
        }
    }
}
```

확인문제

1. 객체와 클래스에 대한 설명으로 틀린 것은 무엇입니까?
    1. 클래스는 객체를 생성하기 위한 설계도(청사진)와 같은 것이다.
    2. new 연산자로 클래스의 생성자를 호출함으로써 객체가 생성된다.
    3. **하나의 클래스로 하나의 객체만 생성할 수 있다. (X , 여러 개의 객체를 생성할 수 있다.)**
    4. 객체는 클래스의 인스턴스이다.

2. 클래스의 구성 멤버가 아닌 것은 무엇입니까?
    1. 필드(field)
    2. 생성자(constructor)
    3. 메소드(method)
    4. **로컬 변수(local variable) (X)**

3. 필드, 생성자, 메소드에 대한 설명으로 틀린 것은 무엇입니까?
    1. 필드는 객체의 데이터를 저장한다.
    2. 생성자는 객체의 초기화를 담당한다.
    3. 메소드는 객체의 동작 부분으로, 실행 코드를 가지고 있는 블록이다.
    4. **클래스는 반드시 필드와 메소드를 가져야 한다. (X , 반드시는 아니다.)**

4. 필드에 대한 설명으로 틀린 것은 무엇입니까?

    1. 필드는 메소드에서 사용할 수 있다.
    2. 인스턴스 필드 초기화는 생성자에서 할 수 있다.
    3. **필드는 반드시 생성자 선언 전에 선언되어야 한다. (X , 상관없다.)**
    4. 필드는 초기값을 주지 않더라도 기본값으로 자동 초기화된다.

5. 생성자에 대한 설명으로 틀린 것은 무엇입니까?

    1. **객체를 생성하려면 생성자 호출이 반드시 필요한 것은 아니다. (X , 반드시 필요)**
    2. 생성자는 다른 생성자를 호출하기 위해 this( )를 사용할 수 있다.
    3. 생성자가 선언되지 않으면 컴파일러가 기본 생성자를 추가한다.
    4. 외부에서 객체를 생성할 수 없도록 생성자에 private 접근 제한자를 붙일 수 있다.

6. 메소드에 대한 설명으로 틀린 것은 무엇입니까?

    1. 리턴값이 없는 메소드는 리턴 타입을 void로 해야한다.
    2. 리턴 타입이 있는 메소드는 리턴값을 지정하기 위해 반드시 return문이 있어야 한다.
    3. 매개값의 수를 모를 경우 "..."를 이용해서 매개 변수를 선언할 수 있다.
    4. **메소드의 이름은 중복해서 선언할 수 없다. (X , 중복해서 선언할 수 있다.)**

7. 메소드 오버로딩에 대한 설명으로 틀린 것은 무엇입니까?

    1. 동일한 이름의 메소드를 여러 개 선언하는 것을 말한다.
    2. **반드시 리턴 타입이 달라야 한다. (X , 다른 것이 달라도 된다.)**
    3. 매개 변수의 타입, 수, 순서를 다르게 선언해야 한다.
    4. 매개값의 타입 및 수에 따라 호출될 메소드가 선택된다.

8. 인스턴스 멤버와 정적 멤버에 대한 설명으로 틀린 것은 무엇입니까?

    1. 정적 멤버는 static으로 선언된 필드와 메소드를 말한다.
    2. **인스턴스 필드는 생성자 및 정적 블록에서 초기화될 수 있다. (X , 서로 접근할 수 없다.)**
    3. 정적 필드와 정적 메소드는 객체 생성 없이 클래스를 통해 접근할 수 있다.
    4. 인스턴스 필드와 메소드는 객체를 생성하고 사용해야 한다.

9. final 필드와 상수(static final)에 대한 설명으로 틀린 것은 무엇입니까?

    1. final 필드와 상수는 초기값이 저장되면 값을 변경할 수 없다.
    2. **final 필드와 상수는 생성자에서 초기화될 수 있다. (X , 상수는 생성자에서 초기화할 수 없다.)**
    3. 상수의 이름은 대문자로 작성하는 것이 관례이다.
    4. 상수는 객체 생성 없이 클래스를 통해 사용할 수 있다.

1. 패키지에 대한 설명으로 틀린 것은 무엇입니까?

    1. 패키지는 클래스들을 그룹화시키는 기능을 한다.
    2. 클래스가 패키지에 소속되려면 패키지 선언을 반드시 해야 한다.
    3. import문은 다른 패키지의 클래스를 사용할 때 필요하다.

4. **mycompany 패키지에 소속된 클래스는 yourcompany에 옮겨 놓아도 동작한다. (X , 동작하지 않는다.)**

1. 접근 제한에 대한 설명으로 틀린 것은 무엇입니까?
    1. 접근 제한자는 클래스, 필드, 생성자, 메소드의 사용을 제한한다.
    2. public 접근 제한은 아무런 제한 없이 해당 요소를 사용할 수 있게 한다.
    3. **default 접근 제한은 해당 클래스 내부에서만 사용을 허가한다. (X , 같은 패키지면 사용할 수 있으나 다른 패키지에서는 사용할 수 없다.)**
    4. 외부에서 접근하지 못하도록 하려면 private 접근 제한을 해야 한다.

1. 다음 클래스에서 해당 멤버가 필드, 생성자, 메소드 중 어떤 것인지 빈칸을 채우세요.

```
public class Member{
    private String name;            // 필드

    public Member(String name){     // 생성자
        ...
    }

    public void setName(String name){   // 메소드
        ...
    }
}
```

1. 현실 세계의 회원을 Member 클래스로 모델링하려고 합니다. 회원의 데이터로는 이름, 아이디, 패스워드, 나이가 있습니다. 이 데이터들을 가지는 Member 클래스를 선언해보세요.

데이터 이름	필드 이름	타입
이름	name	문자열
아이디	id	문자열
패스워드	password	문자열
나이	age	정수

Member.java

```
package CheckProblem;

public class Member {
    String name;
    String id;
    String password;
    int age;
}
```

2. 위에서 작성한 Member 클래스에 생성자를 추가하려고 합니다. 다음과 같이 Member 객체를 생성할 때 name 필드와 id 필드를 외부에서 받은 값으로 초기화하려면 생성자를 어떻게 선언해야 합니까?

```
Member user1 = new Member("홍길동", "hong");
Member user2 = new Member("강자바", "java");
```

Member.java

```
package CheckProblem;

public class Member {
    String name;
    String id;
    String password;
    int age;

    Member(){ }

    Member(String name, String id){
        this.name = name;
        this.id = id;
    }
}
```

3. MemberService 클래스에 login( ) 메소드와 logout( ) 메소드를 선언하려고 합니다. login( ) 메소드를 호출할 때에는 매개값으로 id와 password를 제공하고, logout( ) 메소드는 id만 매개값으로 제공합니다. MemberService 클래스와 login( ), logout( ) 메소드를 선언해보세요.
    1. login( ) 메소드는 매개값 id가 "hong", 매개값 password가 "12345"일 경우에만 true로 리턴하고 그 이외의 값일 경우에는 false를 리턴하도록 하세요.
    2. logout( ) 메소드의 내용은 "로그아웃 되었습니다."가 출력되도록 하세요.

리턴 타입	메소드 이름	매개 변수(타입)
boolean	login	id(String), password(String)
void	logout	id(String)

MemberService.java

```
package CheckProblem;
```

```
public class MemberService {

    boolean login(String id, String password){
        if(id == "hong" && password == "12345"){
            return true;
        }
        else{
            return false;
        }
    }

    void logout(String id){
        System.out.println("로그아웃 되었습니다.");
    }

}
```

MemberServiceExample.java

```
package CheckProblem;

public class MemberServiceExample {
    public static void main(String[] args) {
        MemberService memberService = new MemberService();
        boolean result = memberService.login("hong", "12345");

        if(result){
            System.out.println("로그인 되었습니다.");
            memberService.logout("hong");
        } else {
            System.out.println("id 또는 password가 올바르지 않습니다.");
        }
    }
}
```

실행결과

```
로그인 되었습니다.
로그아웃 되었습니다.
```

4. PrinterExample 클래스에서 Printer 객체를 생성하고 println( ) 메소드를 호출해서 매개값을 콘솔에서 출력하려고 합니다. println( ) 메소드의 매개값으로는 int, boolean, double, String 값을 줄 수 있습니다. Printer 클래스에서 println( ) 메소드를 선언해보세요.

Printer.java

```
package CheckProblem;

public class Printer {

    void println(int value){
```

```
        System.out.println(value);
    }

    void println(double value){
        System.out.println(value);
    }

    void println(String value){
        System.out.println(value);
    }

    void println(boolean value){
        System.out.println(value);
    }
}
```

PrinterExample.java

```
package CheckProblem;

public class PrinterExample {
    public static void main(String[] args) {
        Printer printer = new Printer();

        printer.println(10);
        printer.println(true);
        printer.println(5.7);
        printer.println("홍길동");
    }
}
```

실행결과

```
10
true
5.7
홍길동
```

5. 16번 문제에서는 Printer 객체를 생성하고 println( ) 메소드를 생성했습니다. Printer 객체를 생성하지 않고 PrinterExample 클래스에서 다음과 같이 호출하려면 Printer 클래스를 어떻게 수정하면 될까요?

   Printer.java

```
package CheckProblem;

public class Printer {
    // 전체 메소드에 static을 붙인다
    static void println(int value){
        System.out.println(value);
    }

    static void println(double value){
```

```
        System.out.println(value);
    }

    static void println(String value){
        System.out.println(value);
    }

    static void println(boolean value){
        System.out.println(value);
    }
}
```

PrinterExample.java

```
package CheckProblem;

public class PrinterExample {
    public static void main(String[] args) {
        Printer.println(10);
        Printer.println(true);
        Printer.println(5.7);
        Printer.println("홍길동");
    }
}
```

실행결과

```
10
true
5.7
홍길동
```

6. ShopService 객체를 싱글톤으로 만들고 싶습니다. ShopServiceExample 클래스에서 ShopService의 getInstance( ) 메소드로 싱글톤을 얻을 수 있도록 ShopService 클래스를 작성해보세요.

ShopService.java

```
package CheckProblem;

public class ShopService {
    private static ShopService shopService = new ShopService();

    private ShopService(){ };

    static ShopService getInstance(){
        return shopService;
    }
}
```

ShopServiceExample.java

```
package CheckProblem;

public class ShopServiceExample {
    public static void main(String[] args) {
        ShopService obj1 = ShopService.getInstance();
        ShopService obj2 = ShopService.getInstance();

        if(obj1 == obj2){
            System.out.println("같은 ShopService 객체 입니다.");
        } else {
            System.out.println("다른 ShopService 객체 입니다.");
        }
    }
}
```

실행결과

```
같은 ShopService 객체 입니다.
```

7. 은행 계좌 객체인 Account 객체는 잔고(balance) 필드를 가지고 있습니다. balance 필드는 음수값이 될 수 없고, 최대 백만 원까지만 저장할 수 있습니다. 외부에서 balance 필드를 마음대로 변경하지 못하도록 하고, 0 <= balance <= 1,000,000 범위의 값만 가질 수 있도록 Account 클래스를 작성해보세요.
    1. Setter와 Getter를 이용하세요.
    2. 0 과 1,000,000은 MIN_BALANCE 와 MAX_BALANCE 상수를 선언해서 이용하세요.
    3. Setter의 매개값이 음수이거나 백만 원을 초과하면 현재 balance 값을 유지하세요.

Account.java

```
package CheckProblem;

public class Account {
    static final int MIN_BALANCE = 0;
    static final int MAX_BALANCE = 1000000;

    private int balance;

    void setBalance(int money){
        if(money > MAX_BALANCE || money < MIN_BALANCE){
            return;
        } else {
            this.balance = money;
        }
    }

    int getBalance(){
        return balance;
    }
}
```

AccountExample.java

```java
package CheckProblem;

public class AccountExample {
    public static void main(String[] args) {
        Account account = new Account();

        account.setBalance(10000);
        System.out.println("현재 잔고: " + account.getBalance());

        account.setBalance(-100);
        System.out.println("현재 잔고: " + account.getBalance());

        account.setBalance(2000000);
        System.out.println("현재 잔고: " + account.getBalance());

        account.setBalance(300000);
        System.out.println("현재 잔고: " + account.getBalance());
    }
}
```

8. 다음은 키보드로부터 계좌 정보를 입력받아서, 계좌를 관리하는 프로그램입니다. 실행 결과를 보고, 알맞게 BankApplication 클래스의 메소드를 작성해보세요.

Account.java

```java
package CheckProblem;

public class Account1 {
    private String ano;
    private String owner;
    private int balance;

    public Account1(String ano, String owner, int balance){
        this.ano = ano;
        this.owner = owner;
        this.balance = balance;
    }

    public String getAno(){ return ano; }
    public void setAno(String ano) { this.ano = ano; }
    public String getOwner() { return owner; }
    public void setOwner(String owner) { this.owner = owner; }
    public int getBalance() { return balance; }
    public void setBalance(int balance) { this.balance = balance; }
}
```

BankApplication.java

```java
package CheckProblem;

import java.util.Scanner;

public class BankApplication {
```

```
    private static Account1[] accountArray = new Account1[100];
    private static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        boolean run = true;
        while (run) {
            System.out.println("-----------------------------------");
            System.out.println("1.계좌생성 | 2.계좌목록 | 3.예금 | 4. 출금 | 5.종료
|");
            System.out.println("-----------------------------------");
            System.out.print("선택> ");

            int selectNo = scanner.nextInt();

            if(selectNo == 1){
                createAccount();
            } else if (selectNo == 2){
                accountList();
            } else if (selectNo == 3){
                deposit();
            } else if (selectNo == 4) {
                withdraw();
            } else if (selectNo == 5) {
                run = false;
            }


        }
        System.out.println("프로그램 종료");
    }

    // 계좌생성하기
    private static void createAccount(){
        System.out.println("---------------");
        System.out.println("계좌생성");
        System.out.println("---------------");
        System.out.print("계좌번호: ");
        String ano = scanner.next();
        System.out.print("계좌주: ");
        String owner = scanner.next();
        System.out.print("초기입금액: ");
        int balance = scanner.nextInt();

        for(int i=0; i<100; i++){
            if(accountArray[i] == null){
                // 한 객체에 대해 다시 객체를 선언해줘야한다.
                accountArray[i] = new Account1(ano, owner, balance);
                break;
            }
        }
    }

    // 계좌목록보기
```

```
    private static void accountList(){
        System.out.println("----------------");
        System.out.println("계좌목록");
        System.out.println("----------------");
        for(int i=0; i<100; i++){
            if(accountArray[i] == null){
                break;
            }
            System.out.println(accountArray[i].getAno() + '\t' +
                    accountArray[i].getOwner() + '\t' +
                    accountArray[i].getBalance());
        }
    }

    // 예금하기
    private static void deposit(){
        System.out.print("계좌번호: ");
        String ano = scanner.next();
        Account1 acc = findAccount(ano);
        if(acc == null){
            return;
        } else {
            System.out.print("예금액: ");
            int balance = scanner.nextInt();
            acc.setBalance( acc.getBalance() + balance);
        }
    }

    // 출금하기
    private static void withdraw(){
        System.out.print("계좌번호: ");
        String ano = scanner.next();
        Account1 acc = findAccount(ano);
        if(acc == null){
            return;
        } else {
            System.out.print("출금액: ");
            int balance = scanner.nextInt();
            acc.setBalance( acc.getBalance() - balance);
        }
    }

    // Account 배열에서 ano와 동일한 Account 객체 찾기
    private static Account1 findAccount(String ano){
        for(int i=0; i<100; i++){
            if(accountArray[i] != null && accountArray[i].getAno().equals(ano)){
                return accountArray[i];
            }
        }
        System.out.println("다시 입력해주세요.");
        return null;
    }
}
```

실행결과

```
---------------------------------
1.계좌생성 | 2.계좌목록 | 3.예금 | 4. 출금 | 5.종료 |
---------------------------------
선택> 1
---------------
계좌생성
---------------
계좌번호: 111-111
계좌주: 홍길동
초기입금액: 10000
---------------------------------
1.계좌생성 | 2.계좌목록 | 3.예금 | 4. 출금 | 5.종료 |
---------------------------------
선택> 2
---------------
계좌목록
---------------
111-111 홍길동    10000
---------------------------------
1.계좌생성 | 2.계좌목록 | 3.예금 | 4. 출금 | 5.종료 |
---------------------------------
선택> 1
---------------
계좌생성
---------------
계좌번호: 111-222
계좌주: 강자바
초기입금액: 20000
---------------------------------
1.계좌생성 | 2.계좌목록 | 3.예금 | 4. 출금 | 5.종료 |
---------------------------------
선택> 3
계좌번호: 111-111
예금액: 5000
---------------------------------
1.계좌생성 | 2.계좌목록 | 3.예금 | 4. 출금 | 5.종료 |
---------------------------------
선택> 4
계좌번호: 111-222
출금액: 3000
---------------------------------
1.계좌생성 | 2.계좌목록 | 3.예금 | 4. 출금 | 5.종료 |
---------------------------------
선택> 2
---------------
계좌목록
---------------
111-111 홍길동    15000
111-222 강자바    17000
---------------------------------
1.계좌생성 | 2.계좌목록 | 3.예금 | 4. 출금 | 5.종료 |
```

```
---------------------------------
선택> 5
프로그램 종료
```

- 어려웠던 부분

```
for(int i=0; i<100; i++){
            if(accountArray[i] == null){
                // 한 객체에 대해 다시 객체를 선언해줘야한다.
                accountArray[i] = new Account1(ano, owner, balance);
                break;
            }
        }
```

```
// Account 배열에서 ano와 동일한 Account 객체 찾기
    private static Account1 findAccount(String ano){
        for(int i=0; i<100; i++){
            // 객체는 " == " 으로 비교할 수 없고 equals를 사용하여 문자열을 비교한다.
            if(accountArray[i] != null &&
accountArray[i].getAno().equals(ano)){
                return accountArray[i];
            }
        }
        System.out.println("다시 입력해주세요.");
        return null;
    }
```