Class

▼ 객체 (필드, 메소드)

물리적으로 존재하거나 개념적인 것 중에서 다른 것과 식별 가능한 것

예) 이름, 나이, 색깔, 속도 (필드)

/ 웃다, 먹다, 달린다, 멈춘다 (메소드)

▼ 관계 (집합, 사용, 상속)

집합 관계 : 완성품과 부품 (자동차 ↔ 엔진, 타이어, 핸들)

사용 관계 : 사람 ↔ 자동차(달린다, 멈춘다 등의 메소드)

상속 관계 : 부모 ↔ 자식 / 자동차(자식) ↔기계(부모)

▼ 객체 지향 프로그래밍 특징(캡슐화, 상속,다형성)

- 캡슐화(Encapsulation)

: 필드, 동작을 하나로 묶고 실제 구현 내용을 외부에 숨기는 것 (⇒ 외부의 잘못된 사용으로 인해 데이터가 손상되지 않도록 하기 위해 사용, '싱글톤 패턴')

: 접근제한자

`public : 모든 패키지에서 사용 가능`

`private : 현재 클래스 내에서만 사용 가능  (같은 패키지에 있는 다른 클래스에서도 사용 불가)`

`protected : 같은 패키지(폴더) 내에서만 사용 가능`

`( 같은 패키지에 있는 다른 클래스에서는 사용 가능, 다른 패키지에 있는 클래스에서는 사용 불가능)`

`(default) : 접근제어자를 기입하지 않은 상태, protected와 같음`

- 상속 : 부모 객체가 가지고 있는 필드와 메소드를 자식 객체가 사용할 수 있도록 함

: extends _nterface.클래스명

- 다형성 : 사용 방법은 동일하지만 실행 결과가 다양하게 나오 성질

```
Parent adultJjangah = jjangah;
    System.out.println(adultJjangah);
// 타입이 Parent이지만 본체인 Child가 출력, Child{name=짱아,age =
```

```
// 자식은 부모타입이 될 수 있지만 부모는 자식 타입이 될 수 없음
// -> 왜냐하면 부모에게 없는 객체가 자식에게는 있으므로
```

▼ 클래스 선언

```
package ch06.sec03;

public class SportsCar {}
// 마우스 우클릭 -> [New] -> [Java Class] 로 항상 자동 생성해왔음

class Trie{} // 한 자바 클래스에서 여러 클래스 생성 가능
```

▼ 객체 생성과 클래스 변수

Student.java

```
package ch06.sec04;
public class Student {}
```

StudentExample.java

```
package ch06.sec04; // 다른 패키지여도 import하면 사용가능

public class StudentExample{
public static void main(String[] args) {
Student stu1 = new Student();
// 빈 껍데기인 Student 객체의 stu1 변수가 만들어진다.
Student stu2 = new Student();
// 빈 껍데기인 Student 객체의 stu2 변수가 만들어진다.
// 변수명만 다르면 생성하는데 문제 없음, 서로 다른 힙 메모리를 차지함
```

▼ 클래스 구성 요소(필드, 생성자, 메소드)

- 필드(Field) : 객체의 데이터를 저장하는 역할

  (변수 선언과 비슷하지만 쓰임새는 다르다.)

```java
public class NextStudent {
    // 필드변수(= 객체의 속성, 상태)
    String name;        /* 학생의 이름 */
    int level;          /* 학생의 레벨(기본값 = 1) */
    int exp;            /* 학생의 경험치 */
```

- 생성자 (Constructor) : new 연산자로 객체를 생성할 때 객체의 초기화 역할 (메소드 선언과 비슷하지만, 리턴 타입이 없고 이름은 클래스와 동일)

```java
// 기본 생성자
// 클래스 내에 생성자를 별도로 만들지 않으면
// 기본 생성자는 보이지 않더라도 존재한다.
public NextStudent(){}

// 파라미터를 두개만 입력받는 생성자
public NextStudent(String name, int exp) {
    this.name = name;
    this.exp = exp;
    this.level = 1; // 기본값 설정 두번째 방법
```

- 메소드(Method) : 객체가 수행할 동작, 함수라고 하기도 함

```java
// 하루가 지남.. 에 대한 메소드 만들기
public void afterOneDay() {
    // NextStudent 객체들은 afterOneDay가 실행되면
    // 경험치가 30 오르고, 이때 경험치가 100 이상이 되면 레벨업을 한다.
    exp += (int) (Math.random() * 31) + 10;

    if (exp >= 100) {
        level++;
        exp %= 100;
        System.out.println(name + " 레벨업!!" + (level - 1) + " -> " + level);
    }

    System.out.println(name + "의 경험치: " + exp);
}
```

▼ 기본 생성자, 풀 생성자

기본 생성자 : (constructor → select none → enter)

```
// 기본 생성자
// 클래스 내에 생성자를 별도로 만들지 않으면
// 기본 생성자는 보이지 않더라도 존재한다.
public NextStudent(){}
```

풀 생성자 (constructor → all → enter)

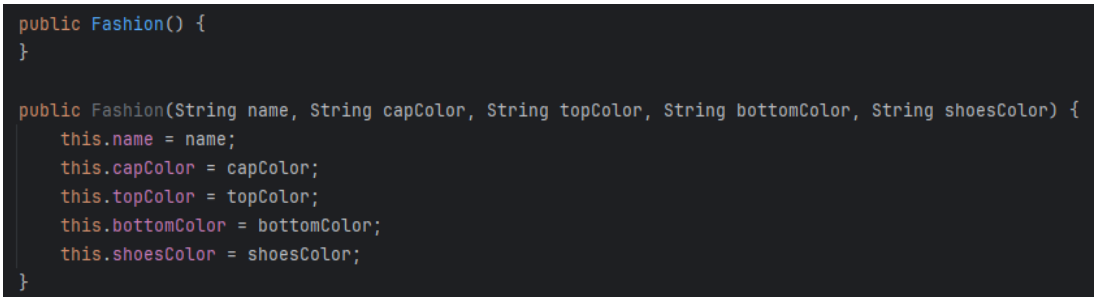
: 클래스 내 모든 필드를 파라미터로 받음

```
public class NextStudent {
    // 필드변수(= 객체의 속성, 상태)
    String name;        /* 학생의 이름 */
    int level;          /* 학생의 레벨(기본값 = 1) */
    int exp;            /* 학생의 경험치 */
```

```
// 파라미터를 입력받는 생성자 만들기
// 생성자를 별도로 만들어 주었기 때문에,
// 보이지 않지만 존재했던 기본 생성자가 사라짐
public NextStudent(String name, int level, int exp) {
    // this는 현재 객체를 의미
    // 현재 객체의 필드변수에 생성자의 파라미터로 들어온 값 부여
    this.name = name;
    this.level = level;
    this.exp = exp;
}
```

▼ 생성자 오버로딩

- 매개변수를 달리하는 생성자를 여러 개 선언하는 것



```java
public Fashion() {
}

public Fashion(String name, String capColor, String topColor, String bottomColor, String shoesColor) {
    this.name = name;
    this.capColor = capColor;
    this.topColor = topColor;
    this.bottomColor = bottomColor;
    this.shoesColor = shoesColor;
}
```

```
// 매개변수를 아무것도 받지 않는 기본생성자
public Fashion(){}

// 매개변수로 문자열을 하나 받는 생성자
public Fashion(String name){
this.name = name;
}

// 매개변수로 name,capColor 문자열을 두개 받는 생성자
public Fashion(String name, String capColor){
this.name = name;
this.capColor = capColor;
}

// ~~~ 선언된 필드의 갯수만큼 생성자 오버로딩 할 수 있다.
```

▼ 다른 생성자 호출 .this()

- 생성자 오버로딩이 많아질 경우 중복 코드 개선하기 위해 사용

```
public Class Car {
String company = "현대자동차";
String model;
String color;
int maxSpeed;

Car(String model){
this(model, "은색", 250);
} //  첫번째 : 필드에 "은색", 250 데이터 들어감

Car(String model, String color){
this(model, color, 250);
```

```
}

Car(String model, String color, int maxSpeed) {
this.model = model;
}// Car의 필드변수에 company = "현대자동차"로 선언되어있는것을 참조

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

▼ 메소드 (메소드 체이닝, 메소드 오버로딩)

- 메소드 체이닝 : 메소드의 리턴 값이 객체 자신이라면 메소드를 줄줄이 이어서 사용하는 것 (자바에서는 잘 보이지 않지만, 자바스크립트에서 자주 보이는 형태)

```
yeonji.setName("연지")
      .setCapColor("빨간색")
      .setShoesColor("하얀색");
```

- 메소드 오버로딩 : 파라미터의 타입이나 파라미터의 개수가 다르다면 중복된 함수명을 사용할 수 있는 것

```
// 첫번째 print 메소드는 문자열 파라미터를 받음
public static void print(String word) {
      System.out.println(word);
  }

// 두번째 print 메소드는 정수형 파라미터를 받음
  public static void print(int num) {
      System.out.println(num);
  }

// 첫번째 두번째 메소드의 파라미터가 서로 달라서
// 중복된 함수명을 사용할 수 있음 (= 메소드 오버로딩)
```

- return 문

: void가 아닌 리턴 타입이 있는 경우 메소드의 실행을 강제 종료하고 호출한 곳으로 리턴 값을 호출한 곳으로 돌려준다는 의미

```
public int plus(int x, int y) {
int result = x + y;
return result;
System.out.println(result)  // Unreachable code
// return 이후에는 실행되지 않음
```

▼ instance member

: 객체에 소속된 멤버

ex) 필드, 메소드

▼ static

정적 멤버 : 메소드의 영역의 클래스에 고정적으로 위치하하는 멤버

(⇒ 객체 생성할 필요 없이 클래스를 통해 바로 사용 가능,

필요한 곳에서 쉽게 호출하여 쉽게 공유하기 위해서 static 붙여서 사용)

▼ final field, final constant

필드 선언 : 한 번 선언이 되면 최종적인 값이 되어 수정 불가

상수 타입의 변수명은 모두 대문자 사용해도 됨

```
// 1. 고정적인 값을 사용할 경우
final String nation = "대한민국";

// 2. 불변의 값인 경우 : static fianl 으로 선언하여
// 여러 개의 값을 가지지 못하도록 함
// -> HTML 선생님이 static이 없는 final은
// 완전한 상수가 아니라고 한 이유
static final double PI = 3.14;
```

▼ 접근제한자 (접근제어자)

- public : `모든 패키지에서 사용 가능`
- private : `현재 클래스 내에서만 사용 가능`

(같은 패키지에 있는 다른 클래스에서도 사용 불가)

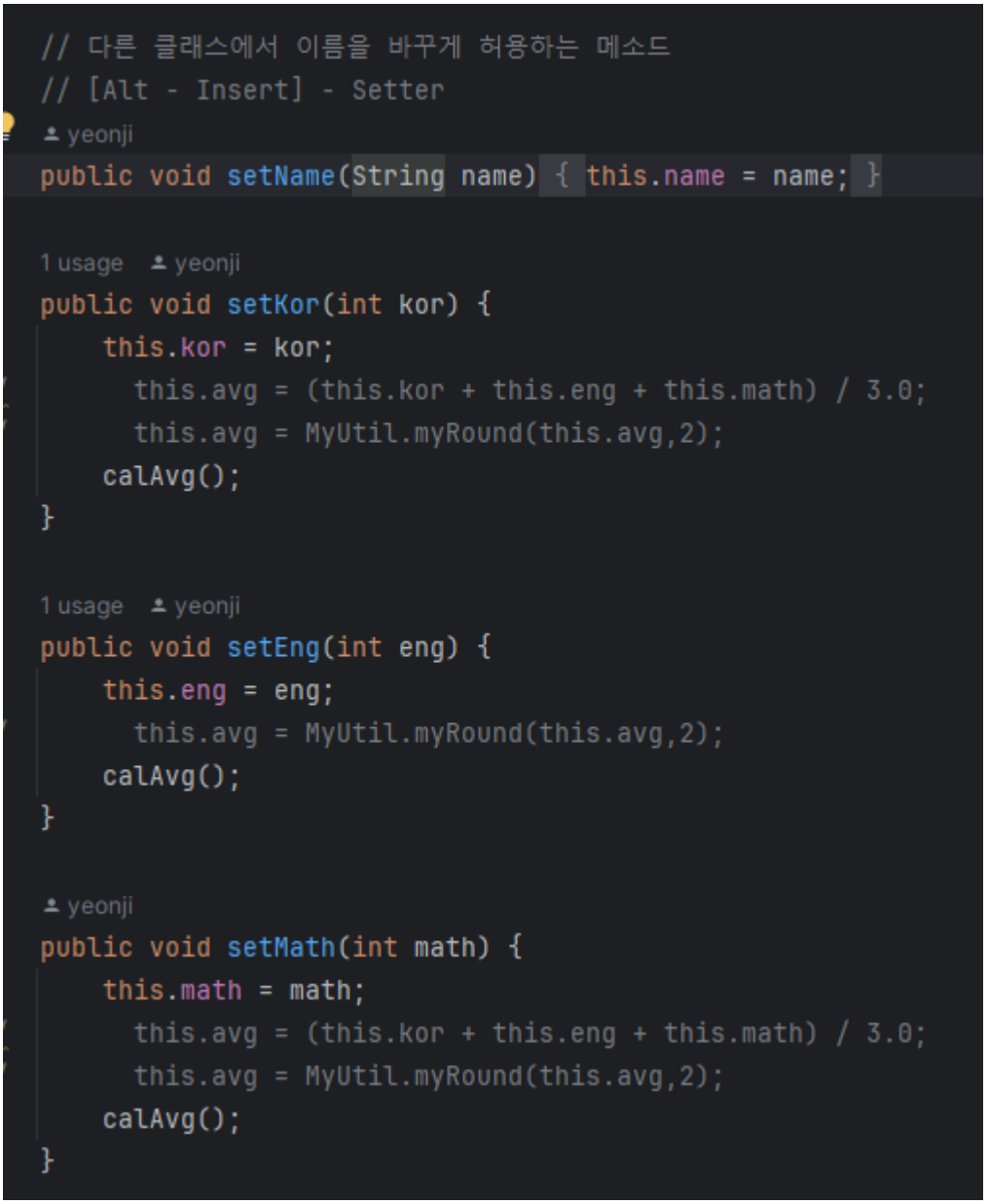
- proteched : 같은 패키지(폴더) 내에서만 사용 가능
  (같은 패키지에 있거나 자식 객체만 사용 가능)
- default : 접근제어자를 기입하지 않은 상태, protected와 같음

▼ getter, setter

- 직접적인 외부에서의 필드 접근을 허용하면 데이터를 변경하다 객체의 무결성을 해칠 수 있기 때문에 사용하는 메소드
- getter

```
// Getter 라고 부름
// name 에 저장된 값을 보여주는 메소드
yeonji
public String getName() { return name; }

// [Alt - Insert] - Getter를 사용해서 자동완성가능
1 usage  yeonji
public int getKor() { return kor; }

1 usage  yeonji
public int getEng() { return eng; }

yeonji
public int getMath() { return math; }

18 usages  yeonji
public double getAvg() { return avg; }
```

- setter



▼ 싱글톤 패턴

- 애플리케이션 전체에서 단 한 개의 객체만 생성해서 사용하고 싶은 경우 사용하는 패턴
- private 으로 생성자의 외부 접근을 제한하고 new 연산자로 생성자를 호출할 수 없도록 막는 것이 핵심

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

// private 접근 권한을 갖는 생성자 선언
```

```
private Singleton {}

// public 접근 권한을 갖는 정적 메소드 선언
// (= 외부에서 읽기전용으로 가져다쓰는 것 같은 형식)
public static Singleton getInstane() {
return singleton;
    }
}
```

▼ 확인문제 (계좌 관리 프로그램 만들기)

👨🏻‍💻 **계좌 관리 프로그램**