[Java] 클래스와 객체

1. 클래스와 객체

1-1. 객체 지향 프로그래밍(OOP)이란?

현실 세계는 독립된 객체로 이루어져 있고, 객체 간의 상호작용에 의해 사건이 발생한다는 개념을 프로그램을 만들 때 적용한 프로그래밍 패러다임이다.

1-2. 현실 세계에서 객체(Object)란?

자신만의 **속성(값, data)**과 **기능(동작,행동)**을 가지고 다른 것들 과 구분되어 식별 가능한 것

ex)

속성: 자동차의 색상, 브랜드, 가격 등

기능: 자동차의 운전, 정비, 주유 등

1-3. 클래스(Class)란

사용자가 필요한 형태의 객체를 만들기 위해 속성과 기능을 정의한 것으로

이를 통해 새로운 타입을 만들 수 있기 때문에 **'사용자 정의 자료형'** 이라고도 한다.

💡 현실 세계의 예시
- 클래스 = 붕어빵 틀
- 객체 = 실제 만들어진 붕어빵

1-4. 클래스 구성요소

- **필드(Fields)**: 객체의 상태를 저장하는 변수
- **메서드(Methods)**: 객체의 행동을 정의하는 함수

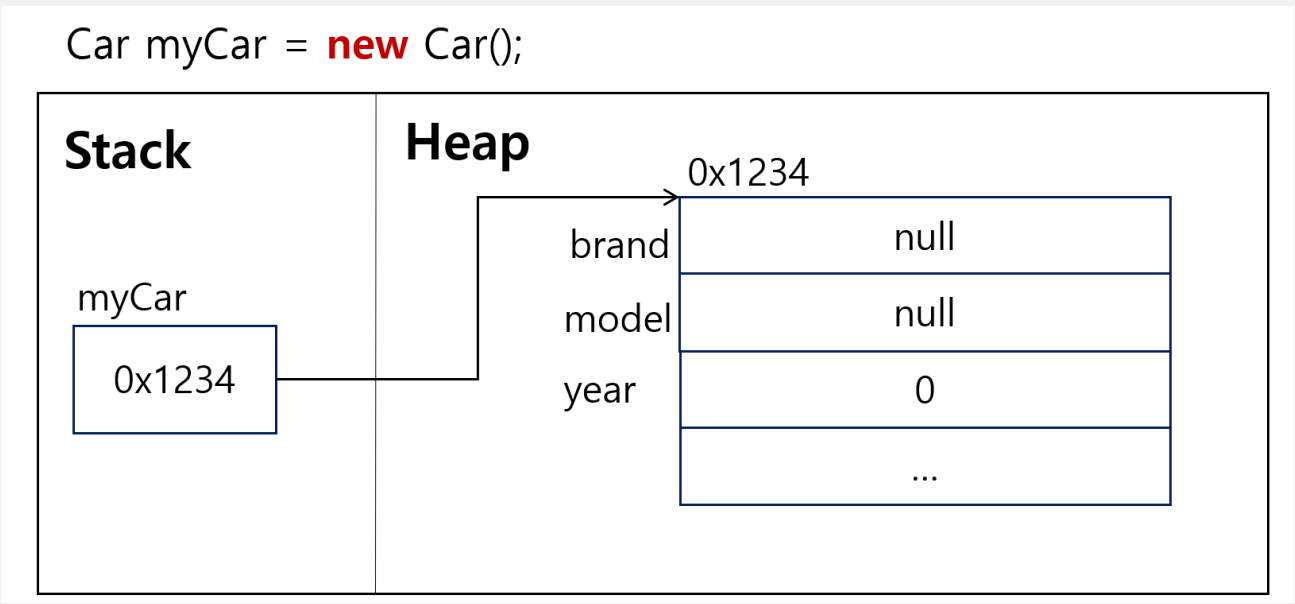
```
public class Car {
    // 필드(Fields) - 객체의 상태/속성
    String brand;
    String model;
    int year;

    // 메서드(Methods) - 객체의 행동
    public void start() {
        System.out.println("시동을 겁니다.");
    }

    public void stop() {
        System.out.println("시동을 끕니다.");
```

```
    }
}
```

1-5. Java에서 객체란?

- 클래스에 정의된 내용대로 JVM Heap 메모리에 영역에 생성된 것
- new 연산자를 이용해 생성

💡 **객체 생성 시 JVM 메모리 구조 예시**



1-6. 객체 생성과 사용방법

```
// 객체 생성
Car myCar = new Car();

// 참조변수명.필드명 , 참조변수명.메서드명으로 필드/메서드에 접근

// 객체 필드에 값 대입
myCar.brand = "현대";
myCar.model = "소나타";
myCar.year = 2025;

// 객체 필드에 저장된 값 얻어오기
System.out.println("myCar brand : " + myCar.brand);
System.out.println("myCar model : " + myCar.model );
System.out.println("myCar year : " + myCar.year);

// 객체의 메서드 호출
myCar.start();
myCar.stop();
```

▼ 실행 결과

```
myCar brand : 현대
myCar model : 소나타
myCar year : 2025
시동을 겁니다.
시동을 끕니다.
```

---

2. 캡슐화(Encapsulation)

2-1. 캡슐화란?

- 객체의 속성(필드)와 기능(메서드)를 하나로 묶고,
  private 접근 제어자를 통해 외부로부터 데이터를 보호하는 클래스 작성 기법이다.
- 특별한 목적이 있는 경우가 아니라면 캡슐화는 기본 원칙으로 사용된다.

2-2. 캡슐화 사용 목적

- **데이터 보호, 검증**

  필드 직접 접근을 막고, 간접 접근 메서드에 데이터 검증 로직을 추가해 잘못된 값이 설정되지 않게함.
- **유지보수성 향상**
  필드의 이름 또는 데이터 타입이 수정된 경우 외부에서 직접 접근하던 모든 코드를 수정하는 상황을 최소화함.

2-3. 접근 제어자

클래스, 메서드, 필드 등의 접근 범위를 제한하는 키워드

구분	같은 클래스	같은 패키지	자식 클래스	전체
public	O	O	O	O
protected	O	O	O	
(default)	O	O		
private	O			

2-4. 캡슐화 미적용 시 문제점과 해결방법

문제점 1. 잘못된 데이터가 필드에 대입됨

```
public class Person{
    String name;   // 이름
```

```
    double height; // 키
}
```

```
Person person1 = new Person();
person1.name = "홍길동";
person1.height = 180.5;

Person person2 = new Person();
person2.name = "김미영";
person2.height = -160.7; // 음수 값이 대입됨(키는 음수가 존재할 수 없음)


System.out.println("person1의 이름: " + person1.name + " / 키: " + person1.height);
System.out.println("person2의 이름: " + person2.name + " / 키: " + person2.height);
```

▼ 실행 결과

```
person1의 이름: 홍길동 / 키: 180.5
person2의 이름: 김미영 / 키: -160.7
```

문제점 2. 필드의 이름 또는 데이터 타입을 수정한 경우

```
public class Person{
    String name;   // 이름
    String personName; // 이름  (필드명 수정)
    double height; // 키
}
```

```
Person person1 = new Person();
person1.name = "홍길동"; // 필드명이 수정되어 컴파일 에러 발생
person1.personName = "홍길동";
person1.height = 180.5;

Person person2 = new Person();
person2.name = "김미영"; // 필드명이 수정되어 컴파일 에러 발생
person2.personName = "김미영";
person2.height = -160.7;

// 필드명이 수정되어 컴파일 에러 발생
System.out.println("person1의 이름: " + person1.name + " / 키: " + person1.height);
System.out.println("person2의 이름: " + person2.name + " / 키: " + person2.height);

System.out.println("person1의 이름: " + person1.personName + " / 키: " + person1.height);
System.out.println("person2의 이름: " + person2.personName + " / 키: " + person2.height);
```

해결 방법

1. 필드에 직접 접근을 제한하고, 간접 접근하여 값을 설정, 검증할 수 있는 메서드를 추가
2. 필드명이 수정 되어도 호출한 곳에서 컴파일 에러 발생 방지 및 수정 코드의 양을 줄이기 위해
   하나의 이름으로 일정한 기능을 수행할 수 있는 메서드 추가

```
public class Person{

    /* private 접근 제어자로 필드 직접 접근 제한 */
    private String personName ;   // 이름 (필드명 수정됨)
  private double height; // 키

    /* public 접근 제어자로 어디서든 간접 접근 허용 */
    public void setPersonName(String personName ){ // 필드명이 수정되어 메서드 수정
        this.personName = personName ;
    }

    /* 메서드 내부에 검증 로직 추가 */
    public void setHeight(double height){
        if(height >= 0){
            this.height = height;
        } else {
            this.height = height * -1;  // 키가 음수인 경우 -1을 곱해 양수로 변환
        }
    }

    /* Person 객체의 정보를 반환 하는 메서드 */
    public String getInfo(){
        // 필드명이 수정되어도 해당 메서드 내부 코드만 수정
        // -> 외부에선 계속 getInfo() 메서드만 호출하면 수정 사항이 적용됨

        return "이름: " + personName + " / 키: " + height;
    }


}
```

```
// 객체 생성 후 간접 접근 메서드를 이용해 필드 값 세팅
Person person1 = new Person();
person1.setPersonName("홍길동"); // setter 메서드명 수정
person1.setHeight(180.5);

Person person2 = new Person();
person2.setPersonName("김미영"); // setter 메서드명 수정
person2.setHeight(-160.7);


System.out.println("person1의 " + person1.getInfo() );
System.out.println("person2의 " + person2.getInfo());
```

▼ 실행 결과

```
person1의 이름: 홍길동 / 키: 180.5
person2의 이름: 김미영 / 키: 160.7
```

3. 추상화(Abstraction)

3-1. 추상화란?

- 복잡한 시스템에서 공통적인 개념과 기능을 추출하고 불필요한 요소를 제거하여 단순화함으로써 유연성을 확보하는 것을 의미한다.
- 추상화 과정을 통해 구현하려는 객체의 속성과 기능이 도출되고, 이를 통해 클래스를 설계할 수 있다.

3-2. 추상화를 이용한 클래스 설계

💡

1. 학생의 성적을 관리하는 프로그램을 만들려고 한다.
   - "학생" 하면 생각할 수 있는 속성
   → 학교, 학년, 반, 번호, 이름, 성별, 국어 점수, 영어 점수, 수학 점수 등...
2. 모든 학생 객체에는 이름, 국어 점수, 영어 점수, 수학 점수가 기록 되어야 한다.
   - 공통적은 가져야 하는 속성을 추출하는 과정 == 추상화

3. 모든 학생 객체는 점수 관련 기능이 있어야 한다.
   - 국어/영어/수학 점수, 평균, 합계 반환 기능 필요
   → 공통 기능을 도출하는 과정 == 추상화

```
public class Student{
    private String name;   // 이름
    private int korScore;  // 국어 점수
    private int engScore;  // 영어 점수
    private int mathScore; // 수학 점수

    /* 필드 간접 접근 메서드(getter/setter) */
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getKorScore() {
        return korScore;
    }
    public void setKorScore(int korScore) {
        this.korScore = korScore;
    }
    public int getEngScore() {
        return engScore;
    }
    public void setEngScore(int engScore) {
        this.engScore = engScore;
    }
    public int getMathScore() {
        return mathScore;
    }
    public void setMathScore(int mathScore) {
        this.mathScore = mathScore;
```

```
    }

    /* 점수 합계 반환 메서드*/
    public int scoreSum() {
        return korScore + engScore + mathScore;
    }

    /* 평균 반환 메서드 */
    public double scoreAverage() {
        return scoreSum() / 3.0;
    }

    public String getInfo() {
        return String.format("%s - 국어 : %d, 영어 : %d, 수학 %d / 합계 : %d, 평균 : %.1f",
                name, korScore, engScore, mathScore, scoreSum(), scoreAverage());
    }

}
```

---

4. 메서드(Method)

4-1. 메서드란?

- 객체의 기능(동작)을 정의하는 것으로, 호출 시 수행되는 명령문의 집합이다.

4-2. 메서드 선언부 구성

```
접근제어자 반환타입 메서드명(매개변수) {
    // 메서드 실행 코드
    return 반환값;
```

4-3. 메서드 종류

- **매개변수와 반환값이 모두 있는 메서드**

```
public int add(int a, int b) {
    return a + b;
}
```

- **매개변수는 있고 반환값이 없는 메서드**

```
public void printSum(int a, int b) {
    System.out.println("합계: " + (a + b));
```

```
}
```

- **매개변수는 없고 반환값이 있는 메서드**

```
public int getCurrentTime() {
    return System.currentTimeMillis();
}
```

- **매개변수와 반환값이 모두 없는 메서드**

```
public void printHello() {
    System.out.println("Hello!");
}
```

4-4. 메서드 호출

```
public class Calculator {
    // 메서드 정의
    public int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // 메서드 호출
        int result = calc.add(10, 20);
        System.out.println("결과: " + result);  // 결과: 30
    }
}
```

```
// Calculator 객체 생성
Calculator calc = new Calculator();

// 메서드 호출
int result = calc.add(10, 20); // 메서드 호출
System.out.println("결과: " + result);  // 결과: 30
```

---

5. 생성자(Constructor)

5-1. 생성자란?

- new 연산자에 의해 객체가 생성될 때 딱 한번 자동으로 호출되는 특수한 메서드로,
  객체 생성 시 수행할 명령어를 작성한다. (객체 생성 시 필드 초기화에 주로 사용)

5-2. 생성자 특징

- 클래스와 이름이 동일해야 한다.
- 반환형이 없음.
- 객체 생성 시 new 연산자와 함께 호출된다.
- 모든 클래스는 최소 하나 이상의 생성자가 있어야 한다.
  → 명시적으로 생성자를 정의하지 않으면 컴파일러가 기본 생성자를 자동으로 추가한다.

5-3. this 키워드

- 현재 객체를 가리키는 참조 변수
- 필드와 매개변수의 이름이 같을 때 구분하기 위해 사용

5-4. 생성자 사용 예시

```
public class Car {
    String brand;
    String model;

    // 기본 생성자
    public Car() {
        this.brand = "Unknown";
        this.model = "Unknown";
    }

    // 매개변수가 있는 생성자
    public Car(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }
}
```

```
// 기본 생성자를 이용한 객체 생성
Car car1 = new Car();
System.out.println("Car1 - Brand: " + car1.brand + ", Model: " + car1.model);

// 매개변수가 있는 생성자를 이용한 객체 생성
Car car2 = new Car("Toyota", "Camry");
System.out.println("Car2 - Brand: " + car2.brand + ", Model: " + car2.model);
```

실행 결과:

```
Car1 - Brand: Unknown, Model: Unknown
Car2 - Brand: Toyota, Model: Camry
```

---

6. 오버로딩(OverLoading)

6-1. 오버로딩이란?

- 동일한 클래스 내에 같은 이름의 생성자/메서드를 작성할 수 있게 하는 기술이다.
- 비슷한 기능을 제공하는 메서드들을 하나의 이름으로 관리할 수 있다.

6-2. 오버로딩의 조건

- 메서드 이름이 동일해야 한다.
- 매개변수의 개수 또는 타입이 달라야 한다.
- 접근제어자, 반환 타입은 오버로딩과 관계없다.

6-3. 오버로딩 예시

```
public class Calculator {
    // 정수 두 개를 더하는 메서드
    public int add(int a, int b) {
        return a + b;
    }

    /* 오버로딩 성립 O */

    // 실수 두 개를 더하는 메서드
    public double add(double a, double b) { // 매개 변수 타입이 다름
        return a + b;
    }

    // 정수 2개, 실수 1개를 더하는 메서드
    public double add(int a, int b, double c) { // 매개 변수 개수가 다름
        return a + b + c;
    }

    // 정수 2개, 실수 1개를 더하는 메서드
    public double add(double a, int b, int c) { // 매개 변수 순서가 다름
        return a + b + c;
    }


    /* 오버로딩 성립 X */

    // 매개 변수명 순서를 변경(오류)
```

```
    public int add(int b, int a) { // 매개 변수의 이름은 메서드 선언에 영향을 주지 않음
        return a + b;
    }

    // 접근 제어자 변경(오류)
    protected int add(int a, int b) { // 접근 제어자는 오버로딩과 관계 없음
        return a + b;
    }

    // 반환형 변경(오류)
    protected long add(int a, int b) { // 반환형은 오버로딩과 관계 없음
        return a + b;
    }

}
```

7. static

7-1. static이란?

- 클래스의 멤버(필드, 메서드)에 적용되는 키워드로, 객체 생성 없이 클래스를 통해 직접 접근할 수 있게 한다.
- 모든 객체가 공유하는 멤버를 선언할 때 사용한다.

7-2. static의 특징

- 프로그램이 시작될 때 **static 메모리 영역에 로드**된다.
- 객체 생성 없이 **클래스명.필드명 / 클래스명.메서드명()** 으로 직접 접근이 가능하다.

7-3. static 예시

```
public class Counter {
    // static 변수 (클래스 변수)
    static int count = 0;

    // 인스턴스 변수
    int instanceCount = 0;

    public Counter() {
        count++;           // 모든 객체가 공유
        instanceCount++;   // 각 객체마다 독립적
    }

    // static 메서드
    public static int getCount() {
        return count;
```

```
    }
}
```

```
// static 변수는 객체 생성 없이 접근 가능
System.out.println("객체 생성 전 Counter.count : " + Counter.count);  // 0

Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();

System.out.println("객체 생성 후 Counter.count : " + Counter.count);  // 3
System.out.println("c1.instanceCount : " + c1.instanceCount);  // 1
System.out.println("c2.instanceCount : " + c2.instanceCount);  // 1
System.out.println("c3.instanceCount : " + c3.instanceCount);  // 1

// static 메서드도 객체 생성 없이 호출 가능
System.out.println("static 메서드 직접 호출 : " + Counter.getCount());  // 3
```

▼ 실행결과

```
객체 생성 전 Counter.count : 0
객체 생성 후 Counter.count : 3
c1.instanceCount : 1
c2.instanceCount : 1
c3.instanceCount : 1
static 메서드 직접 호출 :  3
```

---

8. 변수의 종류

- **지역 변수**: 메서드 내에서 선언되며, 해당 메서드 내에서만 사용 가능
- **인스턴스 변수**: 객체마다 독립적인 값을 가지며, 객체 생성 후 사용 가능
- **클래스 변수**: 모든 객체가 공유하는 값이며, 객체 생성 없이 클래스명으로 접근 가능

구분	선언 위치	생성 시기	소멸 시기	사용 범위
지역 변수	메서드 내부	메서드 실행 시	메서드 종료 시	메서드 내부
인스턴스 변수	클래스 내부	객체 생성 시	객체 소멸 시 (GC 소관)	클래스 전체
클래스 변수	클래스 내부(static 키워드)	클래스 로딩 시	프로그램 종료 시	어디서나

---

9. 초기화 블럭

9-1. 초기화 블럭이란?

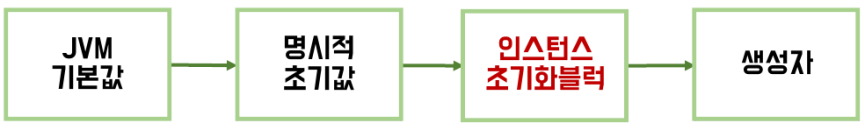
- 클래스나 인스턴스의 필드를 초기화하는 코드 블럭이다.
- 생성자보다 먼저 실행된다.

9-2. 초기화 블럭의 종류

- **static 초기화 블럭**: 클래스가 메모리에 로드될 때 한 번만 실행된다.
- **인스턴스 초기화 블럭**: 객체가 생성될 때마다 실행된다.

9-3. 초기화 블럭 적용 시점

- 인스턴스 변수



- 클래스 변수

9-4. 초기화 블럭 예시

```
public class InitBlock {
    static int staticField = 1;         // 클래스 변수 초기화
    int instanceField = 1;              // 인스턴스 변수 초기화

    // static 초기화 블럭
    static {
        System.out.println("static 초기화 블럭 실행");
        staticField = 2;
    }

    // 인스턴스 초기화 블럭
    {
        System.out.println("인스턴스 초기화 블럭 실행");
        instanceField = 2;
    }

    // 생성자
    public InitBlock() {
        System.out.println("생성자 실행");
        instanceField = 3;
    }
}
```

```
// 객체 생성
System.out.println("첫 번째 객체 생성");
InitBlock obj1 = new InitBlock();
```

```
System.out.println("staticField: " + InitBlock.staticField);
System.out.println("instanceField: " + obj1.instanceField);

System.out.println("\n두 번째 객체 생성");
InitBlock obj2 = new InitBlock();
System.out.println("staticField: " + InitBlock.staticField);
System.out.println("instanceField: " + obj2.instanceField);
```

▼ 실행결과

```
첫 번째 객체 생성
static 초기화 블럭 실행
인스턴스 초기화 블럭 실행
생성자 실행
staticField: 2
instanceField: 3

두 번째 객체 생성
인스턴스 초기화 블럭 실행
생성자 실행
staticField: 2
instanceField: 3
```