# Class And Instance, Object

## Class

- **객체를 만들기 위한 설계도**

## Instance

- **설계도에 따라 메모리 상에 실체화된 구체적인 객체**

## Java에서의 객체

**서로 연관되어 있는 Variable과 Method를 Grouping한 문법적인 기능**

- 상태(State) : Variable
- 행위(Behave) : Method

---

## ? 클래스와 인스턴스가 생겨난 이유와 선대 개발자들이 직면했던 문제점들

i. **메소드화 이전의 프로그래밍**

```
public class CalculatorDemo1 {
     public static void main(String[] args) {
          // 아래의 로직이 1000줄 짜리의 복잡한 로직이라고 가정하자
          System.out.println(10 + 20);
          System.out.println(20 + 40);
     }
}
```

- 중복의 제거가 필요하다. (Refactoring)

---

i. **메소드화 이후의 프로그래밍**

```
public class CalculatorDemo2 {
     public static void main(String[] args) {
          sum(10, 20);
          sum(20, 40);
     }

     public static void sum(int left, int right) {
          System.out.println(left + right);
     }
}
```

- 가독성과 유지보수성, 재활용성이 좋아졌다.

---

### i. 객체화 이전의 프로그래밍

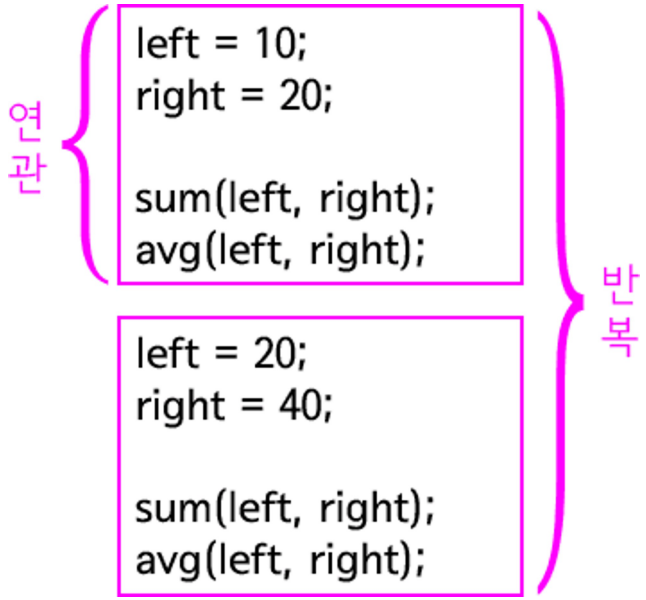
```
public class CalculatorDemo3 {
     public static void main(String[] args) {
          int left, right;

          // Group A
          left = 10;
          right = 20;
                                  ← 이곳에 다른 로직이 추가되면?
          sum(left, right);
          avg(left, right);

          // Group B
          left = 20;
          right = 40;

          sum(left, right);
          avg(left, right);
     }

     public static void sum(int left, int right) {
          System.out.println(left + right);
     }

     public static void avg(int left, int right) {
          System.out.println((left + right) / 2);
     }
}
```

- 예상치 못한 로직의 추가와 변수의 사용, 동일한 이름의 메소드의 정의같은 Case들이 문제를 발생시킬 수 있다.
- 하나의 프로그램 안에서 여러 취지의 코드들이 섞이게 된다. → 막장에 도달

---

### 객체지향 프로그래밍 - 객체화

어떻게 하면 Variable로 상징되는 데이터와 기능으로 상징되는 Method를 서로 연관되어 있는 것 끼리 **그룹핑**할 것인가




변수 left, right의 값은 유지하면서 어떤 때에는 sum 메소드를, 어떤 때에는 avg 메소드를 선택적으로 호출하고 싶다.

```
// 객체 Calculator 설계도
class Calculator {
    int left, right;

    public void setOprands(int left, int right) {

        // this = 실체화 시킨 Instance 자신를 의미
        this.left = left;
        this.right = right;
    }

    public void sum() {
        System.out.println(this.left + this.right);
    }

    public void avg() {
        System.out.println((this.left + this.right) / 2);
    }
}

public class CalculatorDemo4 {
```

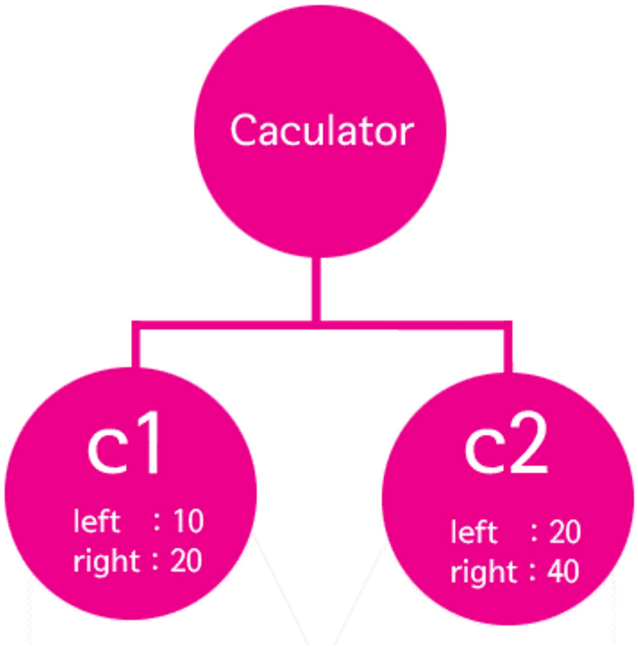
```
    public static void main(String[] args) {

        // 객체는 프로그램 안의 작은 프로그램이다.
        Calculator c1 = new Calculator(); // 객체생성
        c1.setOprands(10, 20);
        c1.sum();
        c1.avg();

        Calculator c2 = new Calculator();
        // 또 다른 객체생성(앞서 선언한 객체와 구조는 같아도 서로 독립적이다!)
        c2.setOprands(20, 40);
        c2.sum();
        c2.avg();
    }
}
```

Instance화

| Calculator | c1 | = | new | Calculator(); |
|---|---|---|---|---|
| ⑤C1은 Calculator형 객체를 담을 수 있는 Data type이여야 한다. ( class를 정의하는 것은 사용자 정의 데이터타입을 만드는 것 ) | ③c1이라는 참조변수에 | ④저장하라 | ②새로 만들고 instance의 메모리주소 참조값을 반환해라 | ①Calculator형 객체를 |

Instance

- 설계도에 따라 메모리 상에 실체화된 구체적인 객체
- C1, C2는 구조는 동일하나, 상태(변수에 저장된 값)가 다르다.
- C1, C2는 각각 독립된 객체이다.




접근제어자 ( . )

c1은 new 연산자를 통해서 만들어진 instance의 메모리주소 참조값을 저장하고 있고, ( . )을 참조변수 뒤에 찍어주면 참조변수의 참조값을 통해 가리키는 Instance에 접근할 수 있다.

동일한 표현 두가지

- `c1.setOprand();`
- `'c1의 참조값'.setOprand();`