6th-week
객체 지향 프로그래밍(OOP, Object Oriented Programming)

> 모든 사물 혹은 사건을 속성과 기능을 가진 객체로 생각하여 객체들 간의 상호작용을 중심으로 프로그래밍하는 방식

객체 지향 프로그래밍은 객체를 속성(데이터)과 기능을 가진 것으로 생각하는 것이다.

객체란?

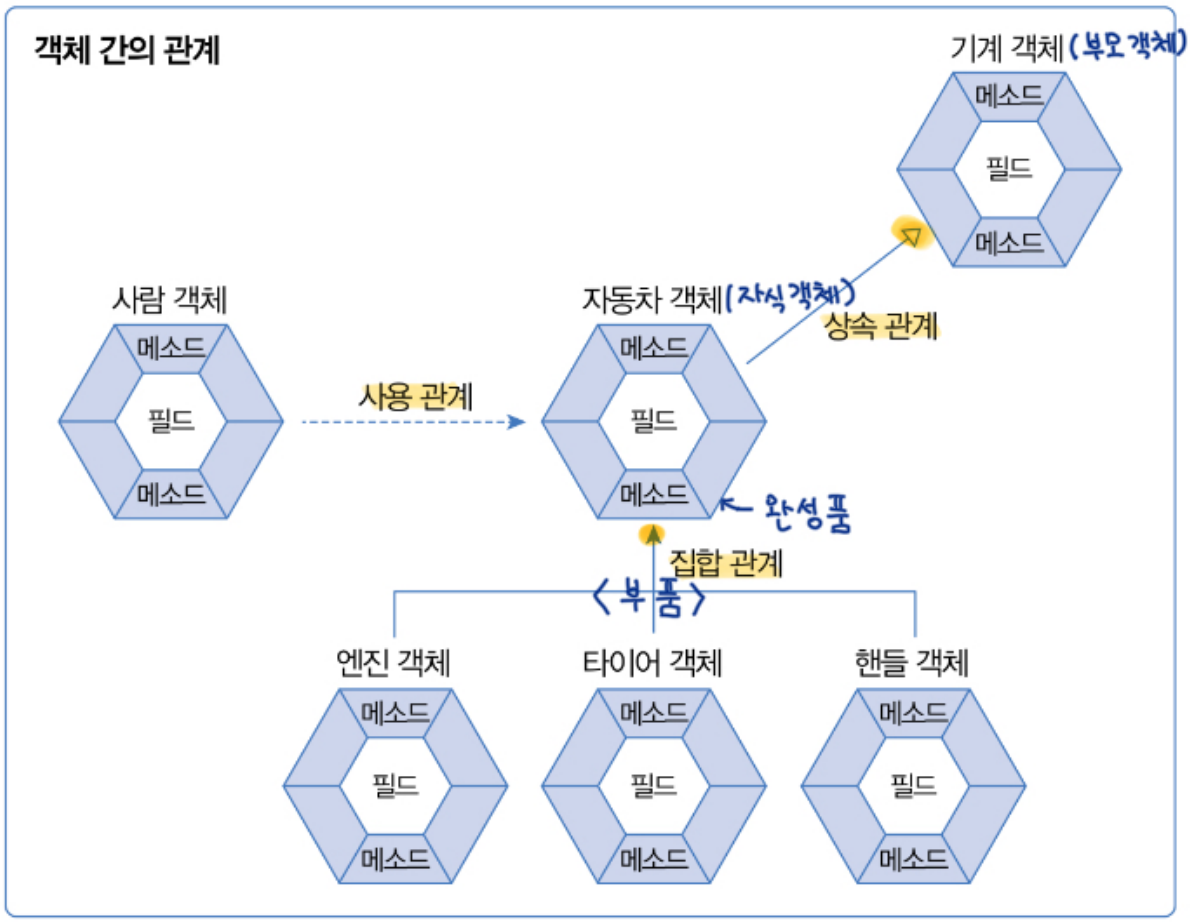
> 물리적으로 존재하거나 개념적인 것 중에서 다른 것과 식별 가능한 것.

객체 모델링(object modeling)

현실 세계의 객체를 소프트웨어 객체로 설계하는 것을 객체 모델링이라고 한다.
현실 세계의 객체를 속성과 기능으로 나누어 객체의 필드와 메소드로 정의하는 과정이다.

객체 간의 관계

> 객체는 다른 객체와 관계를 맺고 있는데 객체 간의 관계의 종류에는 `집합 관계`, `사용 관계`, `상속 관계`가 있다.



`집합 관계`

- 완성품과 부품의 관계
- 자동차는 엔진, 타이어, 핸들 등으로 구성되므로 자동차와 부품은 `집합 관계`라고 볼 수 있다.

`사용 관계`

- 다른 객체의 필드를 읽고 변경하거나 메소드를 호출하는 관계
- 사람은 자동차에게 달린다, 멈춘다 등의 메소드(동작)을 호출하면 사람과 자동차는 `사용 관계`라고 볼 수 있다.

`상속 관계`

- 부모와 자식 관계
- 자동차가 기계의 특징(필드, 메소드)을 사용한다면 기계(부모)와 자동차(자식)는 `상속 관계`에 있다고 볼 수 있다.

객체 지향 프로그램이의 특징

> 객체 지향 프로그램의 특징은 `캡슐화`, `상속`, `다형성`이다.

캡슐화(Encapsulation)

> 속성과 기능을 하나로 묶어서 필요한 기능을 메서드를 통해 외부에 제공하는 것

자바는 캡슐화된 멤버를 노출시킬 것인지 숨길 것인지를 결정하기 위해 접근 제한자(Access Modifier)를 사용한다.
외부 객체는 객체 내부의 구조를 알지 못하며 객체가 노출해서 제공하는 필드와 메소드만 이용할 수 있다.

상속(Inheritance)

> 부모 객체가 갖고 있는 필드와 메소드를 자식 객체가 사용할 수 있도록 하는 것

상속을 하는 이유

- 코드의 재사용성을 높여준다.
  - 하나의 부모에 여러 자식객체를 생성할 수 있어서 중복 코딩하지 않아도 된다.
- 유지 보수 시간을 최소화시켜 준다.
  - 부모 객체의 필드와 메소드를 수정하면 모든 자식 객체들은 수정된 필드와 메소드를 사용할 수 있으므로 추가 수정이 필요없다.

다형성(Polymorphism)

> 사용 방법은 동일하지만 실행 결과가 다양하게 나오는 성질

자동차의 부품을 교환하면 성은이 다르게 나오듯이 프로그램을 구성하는 객체(부품)을 바꾸면 프로그램의 실행 성능이 다르게 나올 수 있다. 이를 구현하기 위해서는 자동 타입 변환과 재정의 기술이 필요하다.

객체와 클래스

> 객체를 생성하려면 설계도에 해당하는 클래스(class)가 필요하다

- `인스턴스(instance)` : 클래스로부터 생성된 객체.
- `인스턴스화` : 클래스로부터 객체를 만드는 과정.

객체 지향 프로그래밍 vs 절차 지향 프로그래밍

- 절차 지향 프로그래밍
  - 절차 지향 프로그래밍은 말 그대로 절차를 지향. 실행 순서를 중요하게 생각하는 방식이다.
  - 프로그램의 흐름을 순차적으로 따르며 처리하는 방식. 즉, `어떻게` 를 중심으로 프로그래밍 한다.
- 객체 지향 프로그래밍
  - 객체 지향 프로그래밍은 객체를 지향한다. 객체를 중요하게 생각하는 방식이다.
  - 객체지향 프로그래밍은 실제 세계의 사물이나 사건을 객체로 보고, 이렇나 객체들 간의 상호작용을 중심으로 프로그래밍하는 방식이다. 즉, `무엇을` 중심으로 프로그래밍 한다.
- 둘의 차이점
  - 서로 대치되는 개념이라기 보다는 어디에 더 초점을 맞추는가에 차이가 있다.
  - 절차 지향은 데이터와 해당 데이터에 대한 처리 방식이 분리되어 있고, 프로그램이 어떻게 작동하는지 그 순서에 초점을 맞춘다.
  - 반면, 객체 지향에서는 데이터와 그 데이터에 대한 행동(메서드)이 하나의 '객체' 안에 함께 포함되어있다. 객체의 설계와 관계를 중시한다.
  - 단순히 객체만 사용한다고 해서 객체 지향 프로그래밍이라 할 수 없다.

절차지향 예시

> 데이터와 기능이 분리됨.

음악 플레이어 예시

1. 음악 플레이어 on/off 기능이 있어야 한다.
2. 볼륨 증가/감소 기능이 있어야한다.
3. 현재 음악 플레이어의 상태를 확인할 수 있어야 한다.

```
// 데이터 묶음
public class MusicPlayerData {
        int volume = 0;
        boolean isOn = false;
}

// 각각의 기능을 메서드로 만들어서 모듈화
public class MusicPlayerMain{

        pulbic static void main(String[] args){
                MusicPlayerData data = new MusicPlayerData();
                //음악 플레이어 on
                on(data);
                //볼륨 증가
                volumeUp(data);
                //볼륨 증가
                volumeUp(data);
                //볼륨 감소
                volumeDown(data);
                //음악 플레이어 상태
                showStatus(data);
                //음악 플레이어 끄기
                off(data);
        }

        static void on(MusicPlayerData data){
                data.isOn = true;
                System.out.println("음악 플레이어를 시작합니다.");
        }

        static void off(MusicPlayerData data){
                data.isOn = false;
                System.out.println("음악 플레이어를 종료합니다.");
        }

        static void volumeUp(MusicPlayerData data){
                data.volume++;
                System.out.println("음악 플레이어 볼륨 증가");
        }

        static void volumeDown(MusicPlayerData data){
                data.volume--;
                System.out.println("음악 플레이어 볼륨 감소");
        }

        static void showStatus(MusicPlayerData data){
                System.out.println("음악 플레이어 상태 확인");
                if(data.isOn){
                        System.out.println("음악 플레이어 ON, 볼륨:" + data.volume);
```

```
                } else {
                        System.out.println("음악 플레이어 OFF");
                }
        }
}
```

- 프로그램의 실행 순서에 중점을 두어 관련된 데이터를 하나로 묶고, 메서드를 사용해 각각 기능을 모듈화했다.
- 하지만, 데이터와 기능이 분리되어 있어 관련 데이터가 변경되면 메서드들도 함께 변경해야 한다. 이렇게 데이터와 기능이 분리되어 있으면 유지보수 관점에서도 관리 포인트가 2곳으로 늘어난다.

객체 지향 예시

> 데이터와 기능을 하나로 묶음.

```
//음악 플레이어 클래스
public class MusicPlayer {
    int volume = 0;
    boolean isOn = false;

     void on(){
        isOn = true;
        System.out.println("음악 플레이어를 시작합니다.");
    }
     void off(){
        isOn = false;
        System.out.println("음악 플레이어를 시작합니다.");
    }

     void volumeUp(){
        volume++;
        System.out.println("음악 플레이어 볼륨 = " + volume);
    }
     void volumeDown(){
        volume--;
        System.out.println("음악 플레이어 볼륨 = " + volume);
    }

     void showStatus(){
        if (isOn) {
            System.out.println("음악 플레이어 ON, 볼륨: " + volume);
        }else{
            System.out.println("음악 플레이어 OFF");
        }
    }
}
//객체 지향
public class MusicPlayerMain{
        public static void main(String[] args){
                MusicPlayer player = new MusicPlayer();
```

```
                //음악 플레이어 켜기
                player.on();
                //볼륨 증가
                player.volumUp();
                //볼륨 증가
                player.volumUp();
                //볼륨 감소
                player.volumDown();
                //음악 플레이어 상태
                player.showStatus();
                //음악 플레이어 끄기
                player.off();
        }
}
```

- MusicPlayer처럼 속성과 기능을 하나로 묶어서 필요한 기능을 메서드를 통해 외부에 제공하는 것을 `캡슐화` 라고 한다.
- MusicPlayer를 사용하는 입장에서는 MusicPlayer 내부에 어떤 속성(데이터)이 있는 전혀 몰라도 사용할 수 있다. MusicPlayer가 제공하는 기능 중 필요한 기능만 호출해서 사용하면 된다.
- 필드명이 변한다고해도 MusicPlayer 내부만 변경하면 되고 기능에 대한 변경도 MusicPlayer 내부만 변경하면 된다.

클래스 선언

> 객체 생성을 위한 설계도를 작성하는 작업. 어떻게 객체를 생성(생성자)하고, 객체가 가져야 할 데이터 (필드)가 무성ㅅ이고, 객체의 동작(메서드)은 무엇인지를 정의하는 내용이 포함된다.

클래스 선언은 소스 파일명과 동일하게 작성한다.

[클래스명.java]

```
//클래스 선언
public class 클래스명{

}
```

- `public class` : 공개 클래스를 선언한다는 뜻
- 클래스명 : 첫 문자를 대문자로 하고 캐멀 스타일로 작성한다. 숫자를 포함해도 되지만 첫 문자는 숫자가 될 수 업속, 특수 문자 중 `$`, `_`를 포함할 수 있다.

하나의 소스 파일은 복수 개의 클래스 선언을 포함할 수 있다. (하나의 소스 파일에 하나의 클래스를 권장)

- 하나의 소스 파일에 복수 개의 클래스를 선언할 때 주의할 점은 소스 파일명과 동일한 클래스만 공개 클래스(public class)로 선언할 수 있다는 것이다. (공개 클래스는 하나의 클래스만 가능하다.)

[SportsCar.java]

```
//하나의 소스 파일에 복수 개의 클래스 선언
public class SportsCar{
}
```

```
class Tire{
}
```

→ 복수 개의 클래스 선언이 포함된 소스 파일을 컴파일하면 바이트코드 파일(.class)은 클래스 선언 수 만큼 생성된다. bin 디렉토리에 `SportsCar.class`와 `Tire.class`가 생성되어 있다.

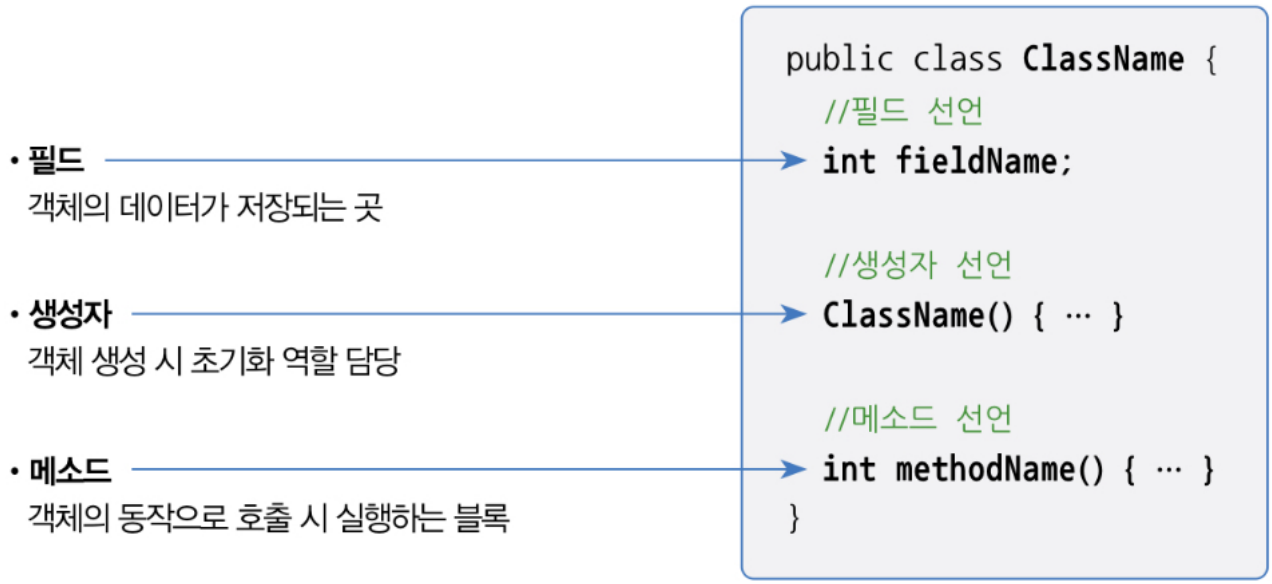
객체 생성과 클래스 변수

클래스로부터 객체를 생성하려면 객체 생성 연산자인 `new`가 필요하다.
`new 연산자`는 객체를 생성시킨 후 객체의 주소를 리턴하여 클래스 변수에 대입이 가능하다.

```
클래스 변수 = new 클래스();
```

클래스의 구성 멤버

> 클래스 선언에는 객체 초기화 역할을 담당하는 생성자와 객체에 포함될 필드, 메서드를 선언하는 코드가 포함된다. 생성자, 필드, 메서드를 클래스의 구성 멤버라고 한다.



- `필드(Field)` : 데이터, 속성. 객체의 데이터를 저장하는 역할.
- `생성자(Constructor)` : 객체를 생성할 때 객체의 초기화 역할을 담당. 선언 형태는 메소드와 비슷하지만, 리턴 타입이 없고 이름은 클래스 이름과 동일.
- `메소드(Method)` : 객체가 수행할 동작. 객체와 객체간의 상호작용을 위해 호출한다.

필드 선언과 사용

> 필드(Field)는 객체의 데이터를 저장하는 역할을 한다.

필드 선언

필드를 선언하는 방법은 변수를 선언하는 방법과 동일하지만, 변수 선언이라 하지는 않는다. 반드시, 클래스 블록에서 선언되어야만 필드 선언이다.

필드 선언시 초기값을 선언하는 부분은 생략 가능하며 생략시 자동으로 해당 타입의 기본값으로 초기화된다.

```
타입 필드명 [ = 초기값];
```

필드명은 첫 문자를 소문자로 하되, 캐멀 스타일로 작성하는 것이 관례이다.

필드 사용

> 필드를 사용한다는 것은 피드값을 읽고 변경하는 것을 말한다.

- 필드는 객체의 데이터이므로 객체가 존재하지 않으면 필드도 존재하지 않는다.
- 외부 객체에서 필드를 읽고 변경할 경우 `참조 변수`와 `도트(.) 연산자`를 이용해야한다.
- `도트(.) 연산자`는 객체 접근 연산자로 **"객체가 가지고 있는"** 필드나 메소드에 접근하고자 할 때 참조 변수 뒤에 붙인다.

생성자 선언과 호출

> `new 연산자`는 객체를 생성한 후 생성자를 호출해서 객체를 초기화하는 역할을 한다.

- `객체 초기화` : 필드 초기화를 하거나 메소드를 호출해서 **"객체를 사용할 준비를 하는 것"**.

기본 생성자

> 모든 클래스는 생성자가 존재하며, 하나 이상을 가질 수 있다.

- 클래스에 생성자 선언이 없으면 컴파일러는 다음과 같은 기본 생성자(Default Constructor)를 바이트코드 파일에 자동으로 추가시킨다.
- 클래스가 public class로 선언되면 기본 생성자도 public이 붙지만, 클래스가 public 없이 class로만 선언되면 기본 생성자에도 public이 붙지 않는다.

생성자 선언

```
//생성자 블록
클래스(매개변수, ...){
        // 객체 초기화 코드
}
```

필드 초기화

> 객체마다 동일한 값을 갖고 있다면 필드 선언 시 초기값을 대입하는 것을 권장, 객체마다 다른 값을 가져야 한다면 생성자에서 필드를 초기화하는 것을 권장

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

- 필드 초기화 시 매개변수명과 필드명이 동일하여 구분을 위해 this 키워드를 붙여준다.
- `this`는 현재 객체를 말하며 `this.name`은 현재 객체의 데이터(필드)로서의 name을 의미한다.

생성자 오버로딩

> 생성자 오버로딩(Overloading)이란 매개변수를 달리하는 생성자를 여러 개 선언하는 것이다.

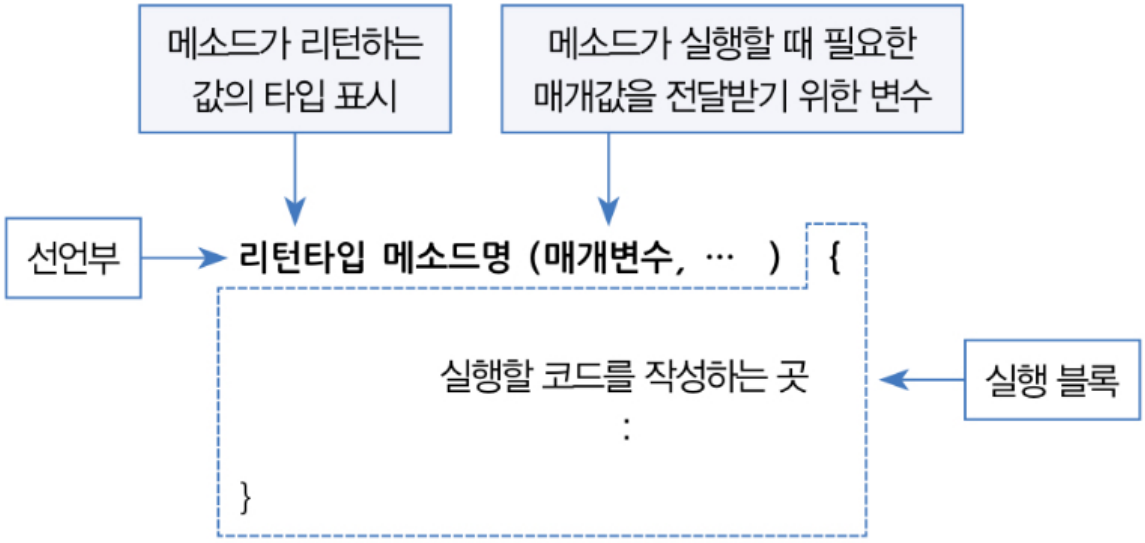
매개변수의 타입과 개수 그리고 선언된 순서가 똑같을 경우 매개변수 이름만 바꾸는 것은 생성자 오버로딩이 아니다.

```
// 순서만 달라져도 가능
Car(int speed, String name){...};
Car(String name, int speed){...};
// 변수명만 달라지는 건 에러
// Car(String name, String color){...};
// Car(String color, String name){...};
```

메소드 선언과 호출

> 메소드는 객체 내부에서도 호출되지만 다른 객체에서도 호출될 수 있기 때문에 객체간의 상호작용하는 방법을 정의하는 것이라 볼 수 있다.

메소드 선언



리턴 타입

> 리턴 타입은 메소드가 실행한 후 호출한 곳으로 전달하는 결과 값의 타입을 말한다. 리턴값이 없는 메소드는 `void`로 작성해야한다.

리턴 타입이 있는 메소드는 실행 블록 안에서 return 문으로 리턴값을 반드시 지정해야 한다.

메소드명

> 메소드명은 첫 문자를 소문자로 시작하고, 캐멀 스타일로 작성한다.

매개 변수

> 실행 블록을 실행하기 위한 값

실행 블록

> 메소드 호출 시 실행되는 부분이다.

메소드 호출

> 메소드를 호출한다는 것은 메소드 블록을 실행한다는 것이다.

- 클래스에서 메소드를 선언했다고 바로 호출할 수 있는 것은 아니다. 메소드는 객체의 동작이므로 객체가 존재하지 않으면 메소드를 호출할 수 없다.
- 클래스로부터 객체가 생성된 후에 메소드는 생성자와 다른 메소드 내부에서 호출될 수 있고, 객체 외부에서도 호출될 수 있다.
- 객체 내부에서 호출 시에는 단순히 메소드 명으로 호출하면된다.
- 외부 객체에서는 참조 변수와 도트(.)연산자를 이용해서 호출한다.
- 메소드가 매개 변수를 가지고 있는 경우 매개변수의 타입과 수에 맞게 매개값을 제공해야 한다.
- 리턴값이 있을 경우에는 대입 연산자를 사용해서 리턴값을 변수에 저장할 수 있다.

가변길이 매개변수

> 메소드를 호출할 때에는 매개변수의 개수에 맞게 매개값을 제공해야 한다. 만약 메소드가 가변길이 매개변수를 가지고 있다면 매개변수의 개수와 상관없이 매개값을 줄 수 있다.

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

- 가변길이 매개변수는 메소드 호출 시 매개값을 쉼표로 구분해서 개수와 상관없이 제공할 수 있다.(단, 같은 타입이여야 한다.)

```
int result = sum(1,2,3);
int result = sum(4,5,6,7,8);
```

- 매개값들은 자동으로 배열 항목으로 변환되어 메소드에 사용된다. 따라서 메소드 호출 시 직접 배열을 매개값으로 제공해도 된다.

```
int[] values = {1,2,3,4};
int result = sum(values);
int result = sum(new int[]{4,5,6});
```

- 매개변수를 직접 배열형태로 메소드 선언시에는 가변길이와 달리 배열만 매개값으로 제공해야한다.

```
// 배열형태의 매개값으로 선언한 경우
int sum(int[] values){
}
// 컴파일 에러
// int result = sum(1,2,3);
// 배열형태만 매개값으로 사용 가능
int result = sum(new int[] {4,5,6,7});
```

return 문

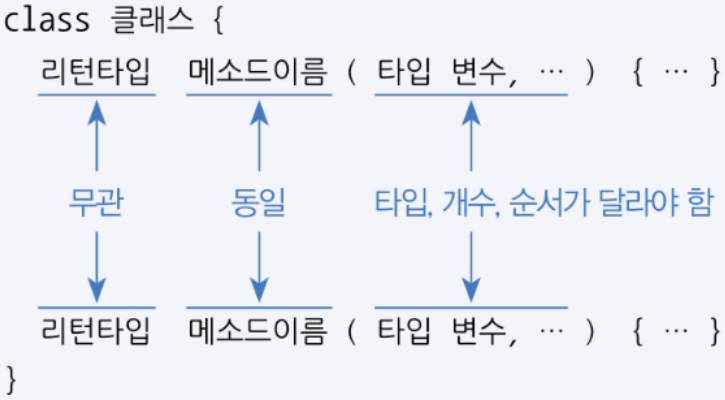
> return 문은 메소드의 실행을 강제 종료하고 호출한 곳으로 돌아간다라는 의미이다. 메소드 선언에 리턴 타입이 있을 경우에는 return 문 뒤에 리턴갑승ㄹ 추가로 지정해야 한다.

```
return [리턴값];
```

- return 문 이후의 실행문을 작성하면 'Unreachable code'라는 컴파일 에러가 발생한다. → return 문 이후의 실행문은 결코 실행되지 않기 때문이다.

메소드 오버로딩

> 메소드 오버로딩(overloading)은 메소드 이름은 같되 매개변수의 타입, 개수, 순서가 다른 메소드를 여러개 선언하는 것을 말한다.



- 메소드 오버로딩의 목적은 다양한 매개값을 처리하기 위해서이다.