Java 기초

클래스

객체(object)

객체

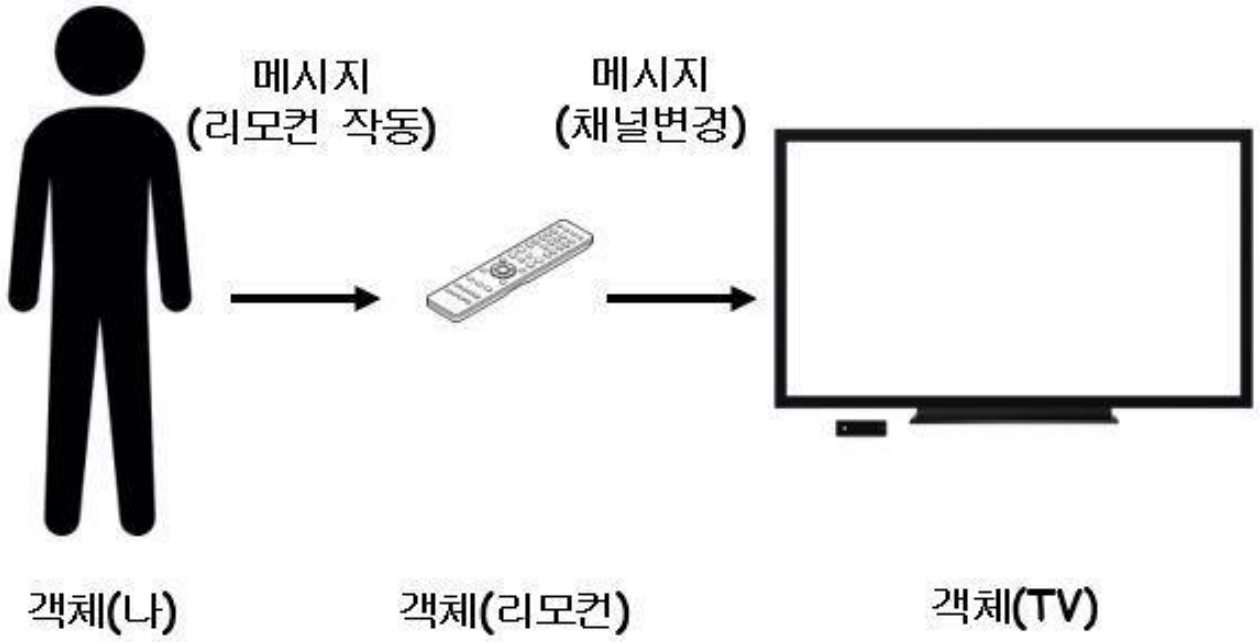
- 객체(Object)
  - 현실에서 명사형으로 이야기 할 수 있는 모든 것
  - 유,무형의 모든 명사형은 객체가 될 수 있다.
    - 유형 : 칼, 자, 도마, 자, 컴퓨터, 키보드, 마우스, 모니터, etc…
    - 무형 : 사랑, 분노, 협상, 싸움, 의지, etc…
  - 객체 안에는 객체 고유의 [속성]과 객체 고유의 [기능]을 가지고 있다.

객체 지향 프로그래밍

코드는 현실을 반영한다

객체 지향 프로그래밍

- 객체 지향 프로그래밍이란?
  - 컴퓨터 프로그래밍 패러다임 중 하나
  - 프로그래밍에서 필요한 데이터를 추상화시켜 상태와 행위를 가진 객체를 만들고 그 객체들 간의 유기적인 상호작용을 통해 로직을 구성하는 프로그래밍 방법



객체 지향 프로그래밍

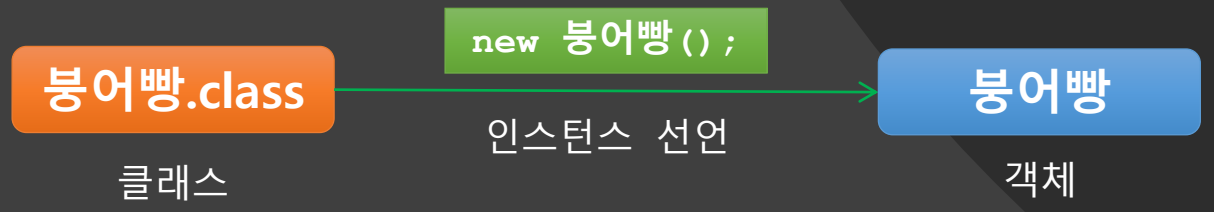
- 객체 지향 프로그래밍 장, 단점
  - 장점
    - 코드 재사용이 용이 : 남이 만든 클래스를 가져와서 이용할 수 있고 상속을 통해 확장해서 사용할 수 있음.
    - 유지보수가 쉬움 : 절차 지향 프로그래밍에서는 코드를 수정해야할 때 일일이 찾아 수정해야하는 반면 객체 지향 프로그래밍에서는 수정해야 할 부분이 클래스 내부에 멤버 변수혹은 메서드로 있기 때문에 해당 부분만 수정하면 됨.
    - 대형 프로젝트에 적합 : 클래스단위로 모듈화 시켜서 개발할 수 있으므로 대형 프로젝트처럼 여러 명, 여러 회사에서 개발이 필요할 시 업무 분담하기 쉽다.
  - 단점
    - 처리속도가 상대적으로 느림
    - 객체가 많으면 용량이 커질 수 있음
    - 설계 시 많은 시간과 노력이 필요

클래스(Class)

- 클래스

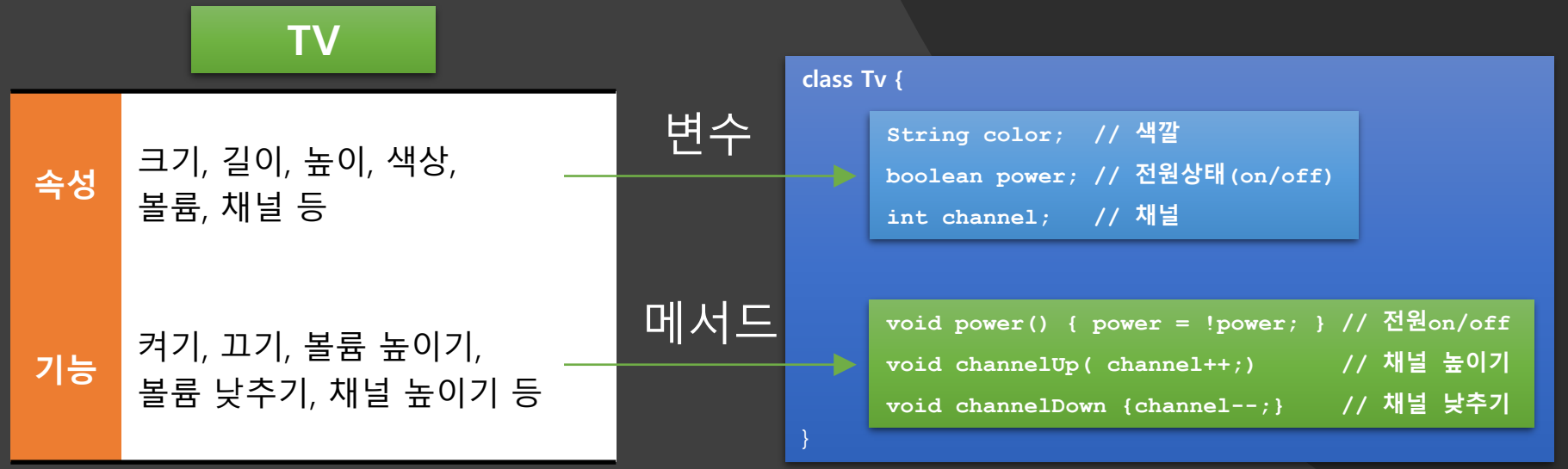
클래스

- 클래스의 정의
  - 클래스는 객체를 생산하기 위한 틀
  - 클래스는 자신을 통해서 생산되는 객체를 정의할 분 클래스 자체가 객체가 될 수는 없다.
  - 자바에서 클래스를 통해 나오는 객체를 인스턴스라고 부른다.
  - 클래스 == 객체(x) / 인스턴스 == 객체(o)



클래스와 객체

- 클래스의 구성요소
  - 모든 객체는 크게 속성과 기능을 가지고 있다고 판단함.
  - 클래스에서 속성을 '맴버변수' 혹은 '필드' 라고 부른다.
  - 클래스에서 기능은 '메서드' 라고 부른다.
  - 모든 클래스는 필드와 메서드로 정의할 수 있다.



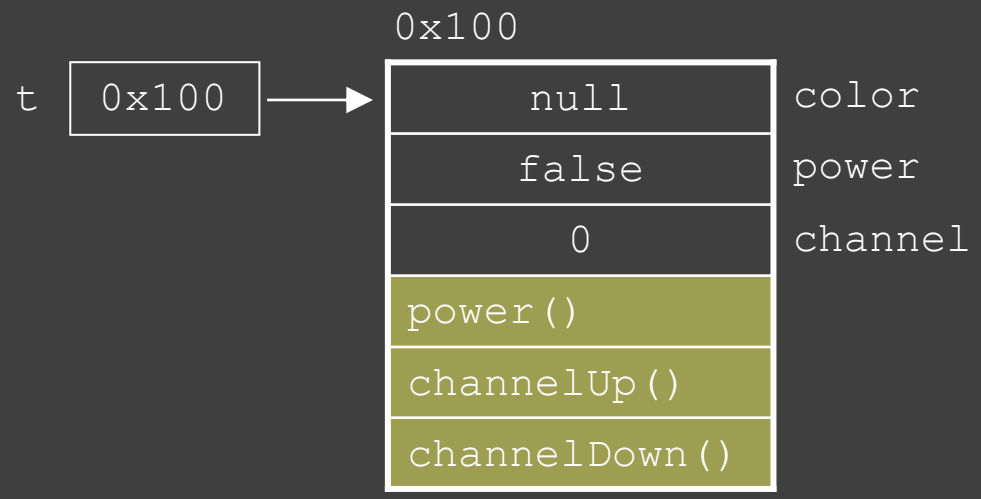
클래스와 객체

- 인스턴스 생성과 사용법

인스턴스 선언1	‘클래스 명’ ‘참조변수 명’; ‘참조변수 명’ = new ‘클래스 명’();
인스턴스 선언2	‘클래스 명’ ‘참조변수 명’ = new ‘클래스 명’();
필드 접근	‘클래스 명’.’필드 명’ = ‘선언할 값’;
메서드 접근	‘클래스 명’.’메서드 명’() ;

클래스와 객체

```
TV t;
t = new TV();
t.channel = 7;
t.channelDown();
System.out.println(t.channel);
```

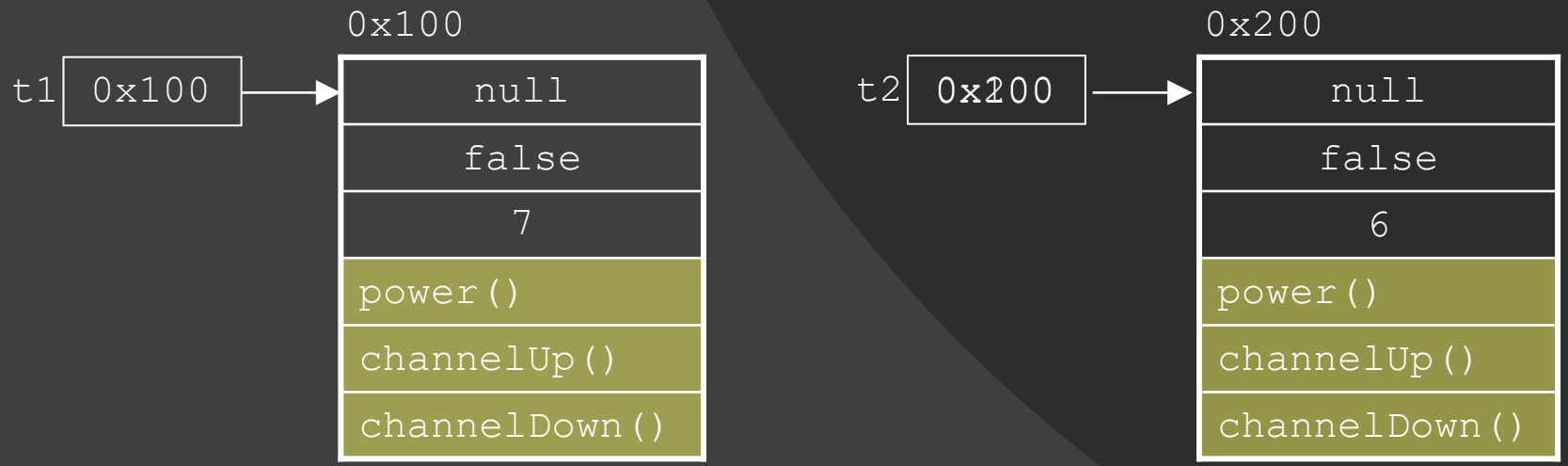



```
public class TV {
    String color; // 색깔
    boolean power; // 전원상태(on/off)
    int channel; // 채널

    void power() {power = !power;} // 전원on/off
    void channelUp() {channel++;} // 채널 높이기
    void channelDown() {channel--;} // 채널 낮추기
}
```

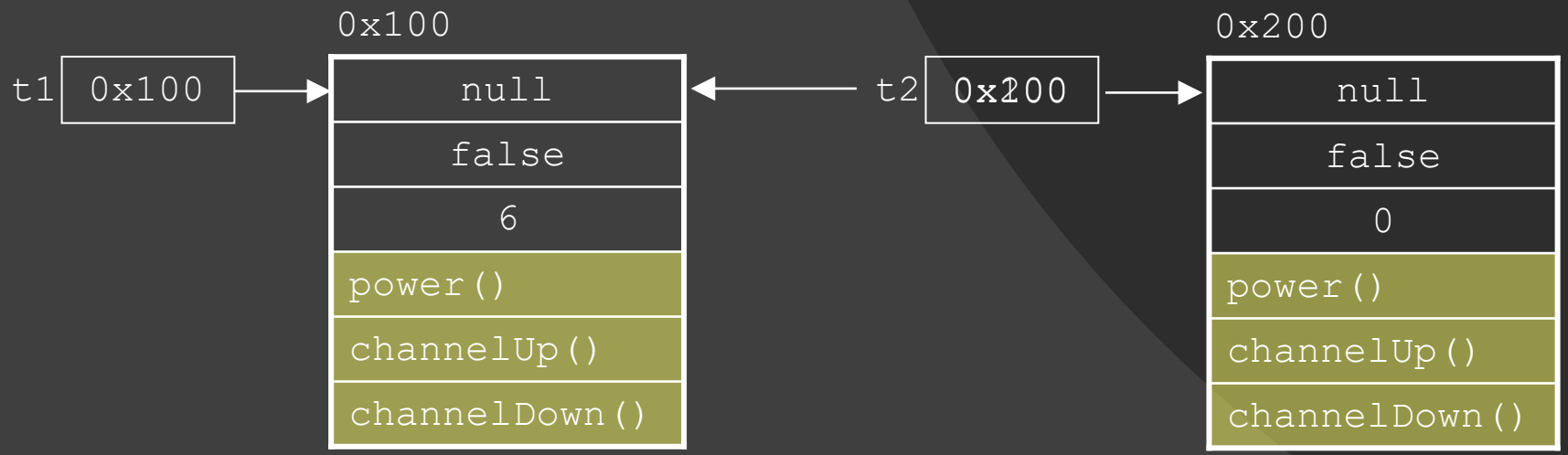
클래스와 객체

```
TV t1 = new TV();
TV t2 = new TV();
t1.channel = 7;
t2.channel = 6;
System.out.println(t1.channel);
System.out.println(t2.channel);
```



클래스와 객체

```
TV t1 = new TV();
TV t2 = new TV();
t2 = t1;
t1.channel = 6;
System.out.println(t1.channel);
System.out.println(t2.channel);
```



클래스와 객체

- 클래스 선언 시 유의점
  - 원 파일 원 클래스(파일 하나에 메인 클래스는 하나의 클래스만 쓰도록 권장)
  - 클래스 이름은 파스칼 표기법으로 선언
    - 첫 단어를 대문자로 시작하는 표기법
    - 이후 이어지는 단어의 첫 글자를 대문자로 넣는다.
    - 예시: BackgroundColor, TypeName, PowerPoint
  - 클래스 명은 기존의 변수명 명명 규칙을 그대로 따라간다.
    - 앞에 숫자가 들어갈 수 없으며
    - 특수문자

클래스 변수

- 클래스 변수
  - 값을 담을 수 있는 장소
  - 변수는 크게 인스턴스 변수 와 맴버 변수(필드)로 나뉜다.
  - 인스턴스 변수
    - 메서드 안에서 선언되며 메서드가 종료될 시 소멸되는 변수
    - {}블록 안에서 선언 될 경우 {}블록이 종료될 시 자동 소멸된다.
  - 맴버 변수(필드)
    - 클래스 영역에 선언되며 인스턴스가 생성될 때 만들어진다.
    - 인스턴스 영역 내에서의 어떤 메서드도 접근이 가능하다.
    - 인스턴스 소멸 시 맴버 변수(필드)도 같이 소멸된다.
    - 다른 표현으로는 인스턴스 변수라고도 한다.

클래스 변수

- 변수의 초기화
  - 지역변수의 초기화는 반드시 초기화가 이루어져야 하며 맴버 변수와 배열은 선택적으로 초기화를 해줘도 상관없다.
  - 맴버변수는 아무것도 선언하지 않았을 때 그 값의 default 로 선언이 된다.

Ex)

```
public class StartVariable {
    int x; // 맴버변수(필드)는 초기화 시 생성자나 메서드나
           // 아니면 맴버변수(필드) 자체에서 초기화가 가능하다.

    public void vSet() {
        int y = 1; // 지역 변수는 반드시 처음 선언 시 초기화
    }
}
```

Default값

자료형	기본값
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형 변수	null

메서드

- 메서드
  - 실제 객체에서 일어나는 행위, 동작을 구현한 단위
  - 작업을 수행하는 명령문의 집합
  - 코드를 단위별로 관리하기가 쉬우며 유지보수가 용이하다.

메서드 사용 시 Tip

하나의 메서드는 한 가지 기능만 수행하도록 작성하는 것이 좋다.

반복적으로 수행되어야 하는 여러 기능을 하나의 메서드로 정의해 놓으면 좋다.

메서드 하나에 너무 많은 기능을 담기보다 여러 개로 분리시켜서 담는 것이 좋다.

메서드

- 메서드 구현 방법

```
접근자 리턴타입 메서드이름(타입 변수명, 타입 변수명,…){
    //메서드 호출 시 수행할 코드
}
```

Ex)

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

메서드

- 메서드 구현 방법

매개변수 선언(X), 리턴(X)

```
public void a(){
    int a = 0;
}
```

매개변수 선언(O), 리턴(X)

```
public void b(int x){
    int a = x;
}
```

매개변수 선언(X), 리턴(O)

```
public int c(){
    int a = 1;
    return a;
}
```

매개변수 선언(O), 리턴(O)

```
public int d(int x){
    int a = x;
    return a;
}
```

필드 & 메서드

- 필드 & 메서드 호출 방법

참조변수를 통해 객체를 선언 할 경우

```
참조변수.메서드 이름();                    // 메서드에 선언된 매개변수가 없는 경우
참조변수.메서드 이름(값1, 값2, ... );      // 메서드에 선언된 매개변수가 있는 경우
참조변수.필드1 = 값;
```

같은 클래스 내의 메서드를 참조할 경우

```
메서드 이름();                    // 메서드에 선언된 매개변수가 없는 경우
메서드 이름(값1, 값2, ... );      // 메서드에 선언된 매개변수가 있는 경우
필드1 = 값;
```

필드 & 메서드

- 필드 & 메서드 호출 방법
  - 참조 변수를 통해 해당 클래스를 선언 후 필드 혹은 메서드를 호출한다.
  - 해당 메서드를 호출 시 인자값이 있을 경우 반드시 대입한다.
  - 해당 메서드에 리턴값이 있을 경우 다른 변수로 받아서 사용하거나 식에 대입하여 사용하는 것이 가능하다.
  - 메서드의 인자
  - 같은 클래스 내에서 메서드나 필드를 참조할 경우 객체에 접근할 참조변수 없이 선언 및 접근이 가능하다.
  - 하지만 같은 클래스 일지라도 static 내에서 메서드를 참조하는 경우 혹은 다른 외부의 클래스에 있는 메서드나 필드를 참조할 경우 반드시 객체를 선언하여 참조변수로 선언하여야 한다.

필드 & 메서드

- 필드 & 메서드 호출 방법

```java
public class ExternalMethod01 {

    String s = "외부에서 호출하는 필드";

    public void method1() {
        System.out.println("외부에서 호출하는 메서드 입니다.");
    }
}
```

```java
public class MethodEx03 {

    String s = "내부에서 호출하는 필드";

    public void method1() {
        System.out.println("내부에서 사용되는 리턴값이 없는 메소드");
    }

    public String method2() {
        return "내부에서 사용되는 리턴값이 있는 메소드";
    }
```

필드 & 메서드

- 필드 & 메서드 호출 방법

```
public void method3() {
    // 내부 메소드에서 내부 메소드의 접근 혹은 필드의 접근 시
    // 참조 변수가 따로 필요 없으며 자연스럽게 접근이 가능하다.
    method1();
    System.out.println(s);
    // 리턴값이 존재하는 메서드 실행 시 호출한 메서드 내에서 리턴한
    // 결과값을 사용이 가능하다.
    String s1 = method2();
    System.out.println(s1);
}

public void method4() {
    // 외부의 객체에 메소드 혹은 필드에 접근하기 위해서 반드시 객체를
    // 선언하고 해당 객체의 참조변수를 선언하여 해당 객체의 메서드와
    // 필드에 접근하여야만 한다.
    ExternalMethod01 ex1 = new ExternalMethod01();
    System.out.println(ex1.s);
    ex1.method1();
}

public static void main(String[] args) {
    // main은 static 메서드 이므로 같은 클래스 내의 일반 메서드나 필드에
    // 접근이 불가능하며 해당 클래스의 참조 변수를 선언하여 접근이 가능하다.
    MethodEx03 ex = new MethodEx03();
    ex.method1();
    System.out.println(ex.s);
    // 리턴값이 존재하는 메서드 실행 시 호출한 메서드 내에서 리턴한
    // 결과값을 사용이 가능하다.
    String s1 = ex.method2();
    System.out.println(s1);
}
```

필드 & 메서드

- 가변 인자 사용
  - 만약 매개변수의 개수가 불투명할 경우 배열타입으로 선언할 수 있다.
  - 하지만 배열타입으로 선언하게 될 경우 메서드를 호출하기 전 배열을 생성해야 하는 불편한 점이 뒤따른다.
  - 메서드의 매개 변수를 [...]를 사용하여 선언하면 메서드 호출 시 넘겨준 값의 수에 따라 자동으로 배열의 생성되고 매개값으로 사용된다.

```
public class ArbitraryArgsEx01 {
    public int sumValues(int ...values) {
        int sum = 0;
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }
        return sum;
    }

    public static void main(String[] args) {
        ArbitraryArgsEx01 aa = new ArbitraryArgsEx01();
        System.out.println(aa.sumValues(1,2,3,4,5,6));
    }
}
```

생성자

- 생성자란
  - 맨 처음 객체를 선언(초기화)할 시에 가장 먼저 접근하는 메서드
  - 해당 메서드는 객체 명이 클래스와 동일하게 선언되어야 한다.
  - 해당 메서드는 리턴값을 가지지 않는다.
  - 매개변수는 메서드와 동일하게 입력이 가능하다. 단 객체 생성 시 new 뒤의 생성자 메서드 호출에 해당 매개변수가 올바르게 입력되어 있지 않는다면 에러가 발생한다.
  - 생성자를 선언하지 않았다면 아무 인자값도 선언하지 않은 생성자가 Default로 정의된다.(기본 생성자)
  - 만약 생성자를 따로 선언할 시에 Default 생성자는 소멸한다.
  - 만약 아무 인자값도 선언되지 않은 생성자를 기존의 생성자와 같이 쓰고 싶다면 따로 선언을 해야한다.

생성자

- 생성자 쓰는 법

```java
public class ConstructorEx01 {
    public ConstructorEx01() {
        System.out.println("ConstructorEx01 의 생성자");
    }

    public static void main(String[] args) {
        ConstructorEx01 ce1 = new ConstructorEx01();
    }
}
```

```java
public class ConstructorEx02 {

    int i;
    int j;

    public ConstructorEx02(int x, int y) {
        i = x;
        j = y;
    }

    public static void main(String[] args) {
        // ConstructorEx02 ex = new ConstructorEx02(); // 에러
        ConstructorEx02 ex = new ConstructorEx02(1,2);
    }
}
```

This

- this
  - 자기 자신을 가리키는 선언자
  - 객체 내부에서 자신의 생성자, 메서드, 멤버변수로 접근할 때 선언할 때 쓴다.
  - 생성자 접근을 제외한 메서드와 맴버변수는 생략이 가능하다.
  - 자신의 생성자 접근시에는 this()라고 쓰며 생성자 내에서만 선언이 가능하다.

This

- this

```java
public class ThisEx01 {
    int a = 1;
    int b = 2;

    public void method1() {
        System.out.println("method1 이다.");
    }

    public void method2() {
        // this는 필드 혹은 메서드를 안에서 참조할 경우
        // 대부분 생략이 가능하지만 외부에서 참조하는 클래스의
        // 필드 혹은 메서드 부분은 this로 접근이 불가능하다.
        this.a = 3;
        System.out.println(a);

        b = 4;
        System.out.println(b);

        this.method1();
        method1();
    }
}
```

```java
public class ThisEx02 {
    int x;
    int y;

    public ThisEx02(int x, int y) {
        // 만약 매개변수와 필드의 이름이 겹칠경우
        // 해당 변수가 필드인지 매개변수인지를 명시하기 위해
        // 명시적으로 this를 써준다.
        this.x = x;
        this.y = y;
    }
}
```

메소드 오버로딩

- 메소드 오버로딩
  - 같은 이름의 메소드를 여러 개 정의하는 것을 말한다.
  - 오버로딩은 아래와 같은 조건을 따른다.

1. 메서드 이름이 같아야 한다.
2. 매개변수의 개수 또는 타입이 달라야 한다.
3. 매개변수는 같고 리턴타입이 다른 경우는 오버로딩이 아니다.
   (리턴타입은 오버로딩을 구현하는데 아무런 영향을 주지 못한다)
4. 매개변수의 이름을 달리 선언한 경우는 오버로딩이 아니다
   (인자값 이름은 오버로딩을 구현하는데 아무런 영향을 주지 못한다)

메소드 오버로딩

- 메소드 오버로딩

```java
public class OverloadEx01 {
    public void m1(int i) {}
    public void m1(int i, int j) {}
    // public void m1(int x) {} // 에러
    // public void m1(int i) { return 1; } // 에러
}
```

```java
public class OverloadEx02 {
    // 오버로드는 일반 메소드 뿐만이 아닌
    // 생성자 메소드에도 적용이 가능하다.
    // 이렇게 되면 생성자에서 생성자로의 호출이 가능하며
    // 호출 시 this(매개변수) 를 넣어 호출할 수 있다.
    // 일반 메서드에서 생성자의 호출은 불가능하다.
    public OverloadEx02() {
        this(1);
    }

    public OverloadEx02(int x) {
        this("hello"+x);
    }

    public OverloadEx02(String x) {
        System.out.println(x);
    }

    public void method1() {
        //this(); // 에러
    }

    public static void main(String[] args) {
        OverloadEx02 ex = new OverloadEx02();
    }
}
```