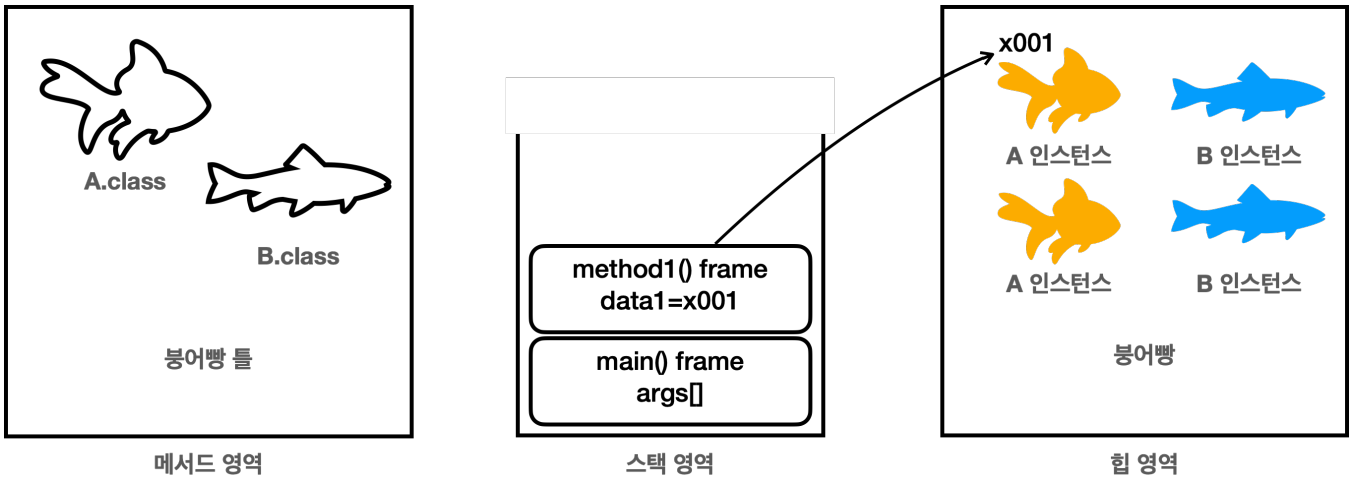
7. 자바 메모리 구조와 static

#0.강의/1.자바로드맵/2.자바-기본

자바 메모리 구조

자바 메모리 구조 - 비유

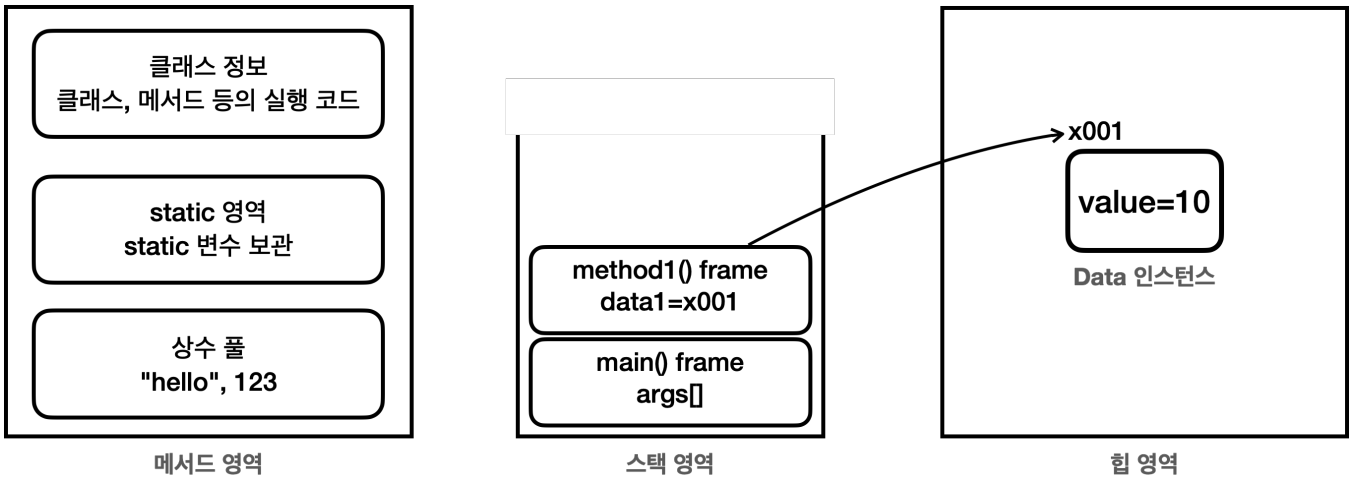


자바의 메모리 구조는 크게 메서드 영역, 스택 영역, 힙 영역 3개로 나눌 수 있다.

- **메서드 영역**: 클래스 정보를 보관한다. 이 클래스 정보가 붕어빵 틀이다.
- **스택 영역**: 실제 프로그램이 실행되는 영역이다. 메서드를 실행할 때 마다 하나씩 쌓인다.
- **힙 영역**: 객체(인스턴스)가 생성되는 영역이다. `new` 명령어를 사용하면 이 영역을 사용한다. 쉽게 이야기해서 붕어빵 틀로부터 생성된 붕어빵이 존재하는 공간이다. 참고로 배열도 이 영역에 생성된다.

방금 설명한 내용은 쉽게 비유로 한 것이고 실제는 다음과 같다.

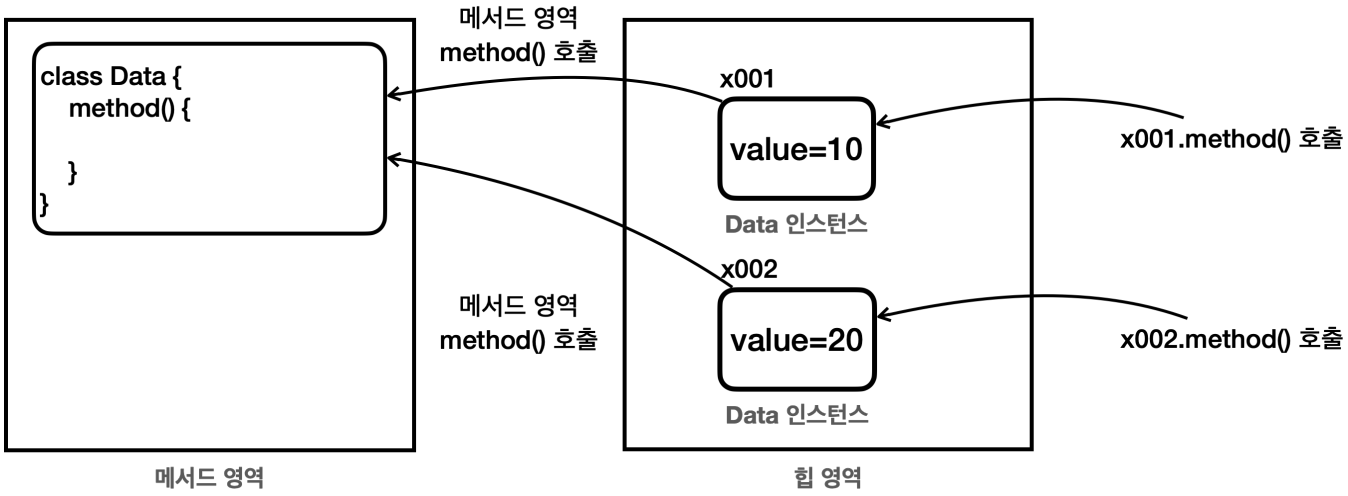
자바 메모리 구조 - 실제



- **메서드 영역(Method Area)**: 메서드 영역은 프로그램을 실행하는데 필요한 공통 데이터를 관리한다. 이 영역은 프로그램의 모든 영역에서 공유한다.
  - 클래스 정보: 클래스의 실행 코드(바이트 코드), 필드, 메서드와 생성자 코드등 모든 실행 코드가 존재한다.
  - static 영역: `static` 변수들을 보관한다. 뒤에서 자세히 설명한다.
  - 런타임 상수 풀: 프로그램을 실행하는데 필요한 공통 리터럴 상수를 보관한다. 예를 들어서 프로그램에 `"hello"` 라는 리터럴 문자가 있으면 이런 문자를 공통으로 묶어서 관리한다. 이 외에도 프로그램을 효율적으로 관리하기 위한 상수들을 관리한다. (참고로 문자열을 다루는 문자열 풀은 자바 7부터 힙 영역으로 이동했다.)
- **스택 영역(Stack Area)**: 자바 실행 시, 하나의 실행 스택이 생성된다. 각 스택 프레임은 지역 변수, 중간 연산 결과, 메서드 호출 정보 등을 포함한다.
  - 스택 프레임: 스택 영역에 쌓이는 네모 박스가 하나의 스택 프레임이다. 메서드를 호출할 때 마다 하나의 스택 프레임이 쌓이고, 메서드가 종료되면 해당 스택 프레임이 제거된다.
- **힙 영역(Heap Area)**: 객체(인스턴스)와 배열이 생성되는 영역이다. 가비지 컬렉션(GC)이 이루어지는 주요 영역이며, 더 이상 참조되지 않는 객체는 GC에 의해 제거된다.

> **참고**: 스택 영역은 더 정확히는 각 쓰레드별로 하나의 실행 스택이 생성된다. 따라서 쓰레드 수 만큼 스택 영역이 생성된다. 지금은 쓰레드를 1개만 사용하므로 스택 영역도 하나이다. 쓰레드에 대한 부분은 멀티 쓰레드를 학습해야 이해할 수 있다.

메서드 코드는 메서드 영역에



자바에서 특정 클래스로 100개의 인스턴스를 생성하면, 힙 메모리에 100개의 인스턴스가 생긴다. 각각의 인스턴스는 내부에 변수와 메서드를 가진다. 같은 클래스로 부터 생성된 객체라도, 인스턴스 내부의 변수 값은 서로 다를 수 있지만, 메서드는 공통된 코드를 공유한다. 따라서 객체가 생성될 때, 인스턴스 변수에는 메모리가 할당되지만, 메서드에 대한 새로운 메모리 할당은 없다. 메서드는 메서드 영역에서 공통으로 관리되고 실행된다.
정리하면 인스턴스의 메서드를 호출하면 실제로는 메서드 영역에 있는 코드를 불러서 수행한다.

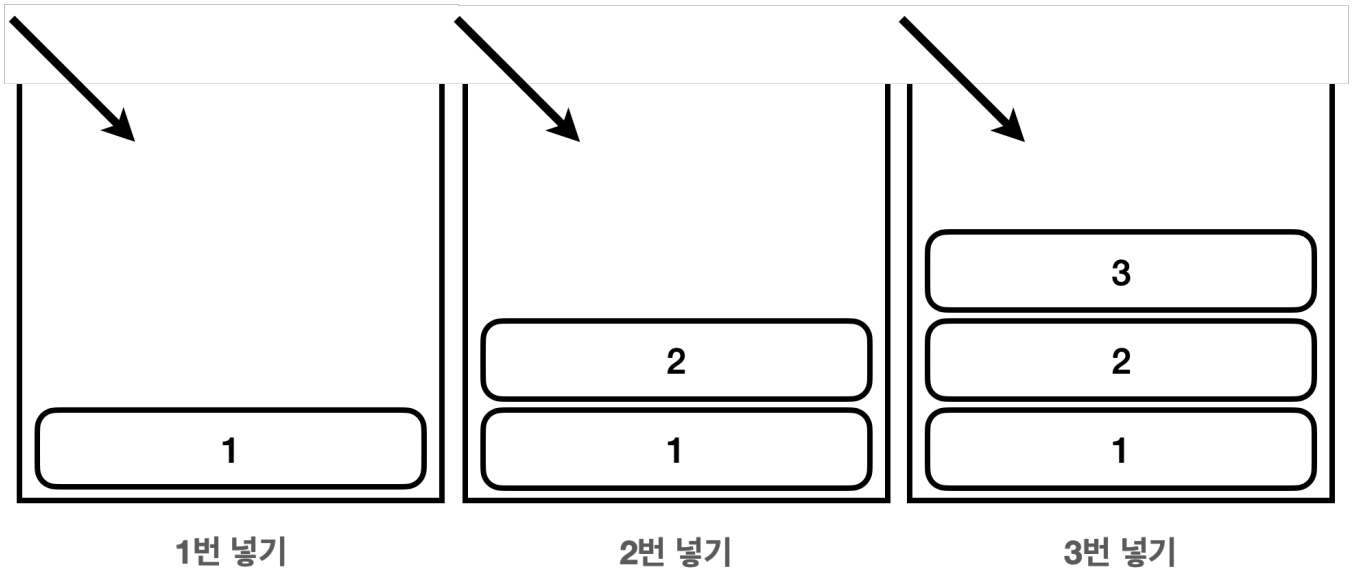
스택과 큐 자료 구조

자바 메모리 구조 중 스택 영역에 대해 알아보기 전에 먼저 스택(Stack)이라는 자료 구조에 대해서 알아보자.

스택 구조

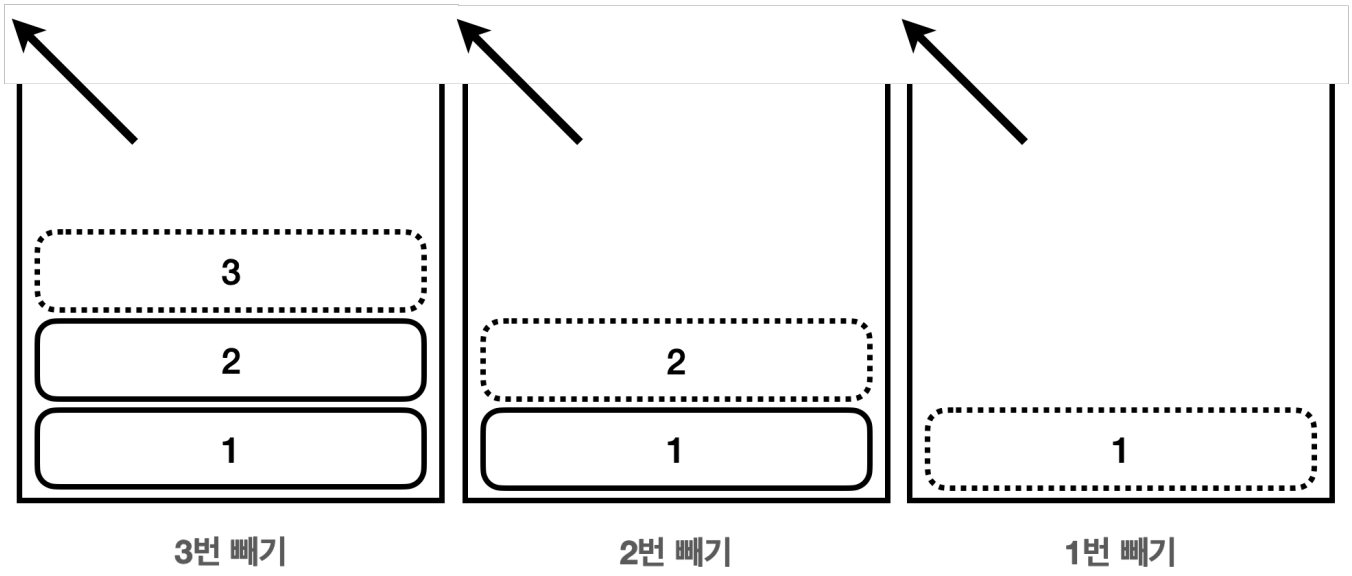
다음과 같은 1, 2, 3 이름표가 붙은 블럭이 있다고 가정하자.

이 블럭을 다음과 같이 생긴 통에 넣는다고 생각해보자. 위쪽만 열려있기 때문에 위쪽으로 블럭을 넣고, 위쪽으로 블럭을 빼야 한다. 쉽게 이야기해서 넣는 곳과 빼는 곳이 같다.



블럭은 1➔ 2➔ 3 순서대로 넣을 수 있다.

이번에는 넣은 블럭을 빼자.



블럭을 빼려면 위에서 부터 순서대로 빼야한다.

블럭은 3➔ 2➔ 1 순서로 뺄 수 있다.

정리하면 다음과 같다.

1(넣기)➔ 2(넣기)➔ 3(넣기)➔ 3(빼기)➔ 2(빼기)➔ 1(빼기)

후입 선출(LIFO, Last In First Out)

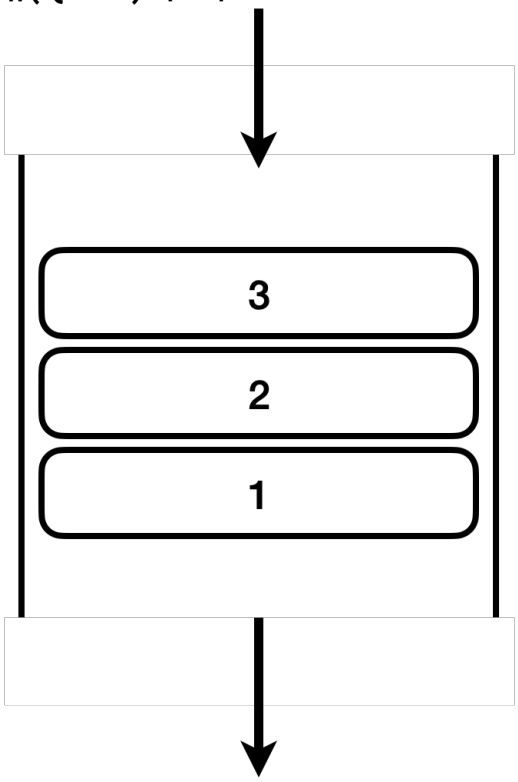
여기서 가장 마지막에 넣은 3번이 가장 먼저 나온다. 이렇게 나중에 넣은 것이 가장 먼저 나오는 것을 후입 선출이라 하고, 이런 자료 구조를 스택이라 한다.

선입 선출(FIFO, First In First Out)

후입 선출과 반대로 가장 먼저 넣은 것이 가장 먼저 나오는 것을 선입 선출이라 한다. 이런 자료 구조를 큐(Queue)라

한다.

큐(Queue) 자료 구조



정리하면 다음과 같다.
1(넣기) ➜ 2(넣기) ➜ 3(넣기) ➜ 1(빼기) ➜ 2(빼기) ➜ 3(빼기)

이런 자료 구조는 각자 필요한 영역이 있다. 예를 들어서 선착순 이벤트를 하는데 고객이 대기해야 한다면 큐 자료 구조를 사용해야 한다.

이번시간에 중요한 것은 스택이다. 프로그램 실행과 메서드 호출에는 스택 구조가 적합하다. 스택 구조를 학습했으니, 자바에서 스택 영역이 어떤 방식으로 작동하는지 알아보자.

스택 영역

다음 코드를 실행하면 스택 영역에서 어떤 변화가 있는지 확인해보자.

JavaMemoryMain1

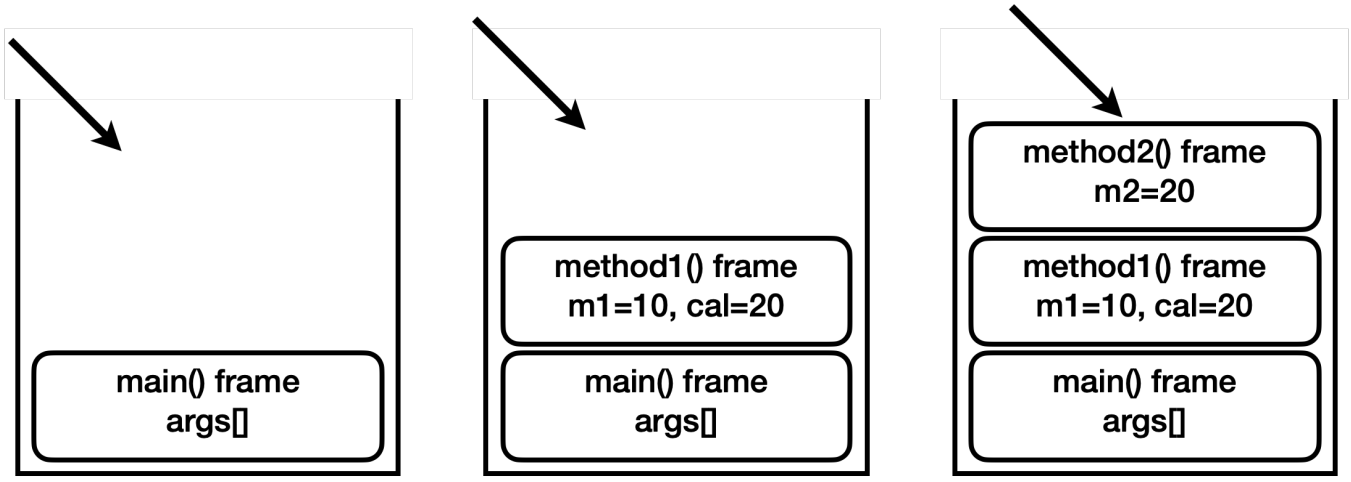
```
package memory;
```

```java
public class JavaMemoryMain1 {

    public static void main(String[] args) {
        System.out.println("main start");
        method1(10);
        System.out.println("main end");
    }

    static void method1(int m1) {
        System.out.println("method1 start");
        int cal = m1 * 2;
        method2(cal);
        System.out.println("method1 end");
    }

    static void method2(int m2) {
        System.out.println("method2 start");
        System.out.println("method2 end");
    }

}
```

실행 결과

```
main start
method1 start
method2 start
method2 end
method1 end
main end
```

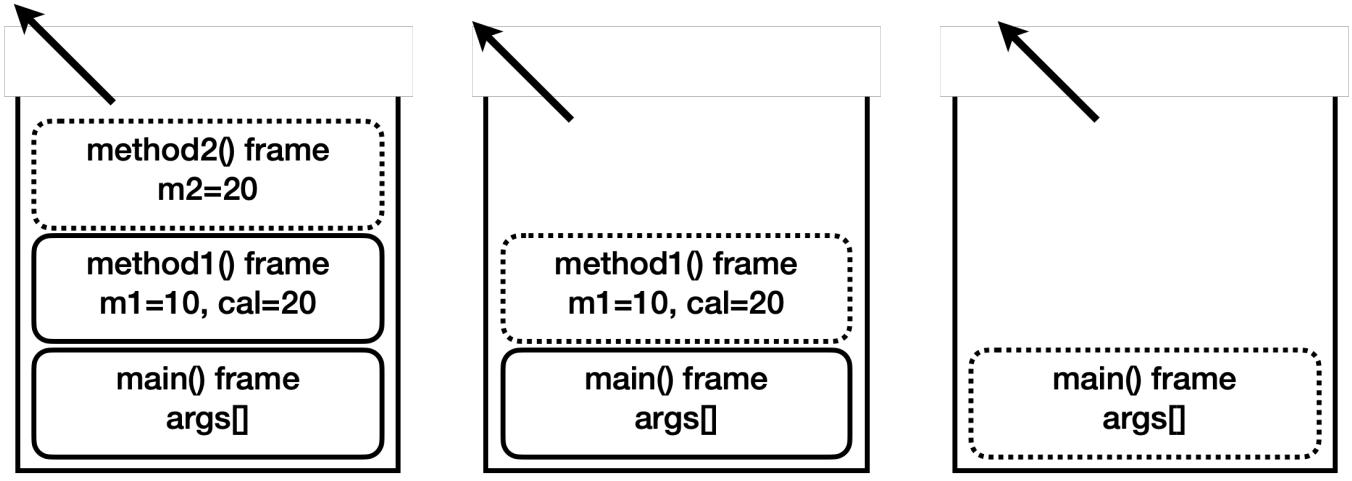
호출 그림



main() 실행 method1() 실행 method2() 실행

- 처음 자바 프로그램을 실행하면 `main()`을 실행한다. 이때 `main()`을 위한 스택 프레임이 하나 생성된다.
  - `main()` 스택 프레임은 내부에 `args`라는 매개변수를 가진다. `args`는 뒤에서 다룬다.
- `main()`은 `method1()`을 호출한다. `method1()` 스택 프레임이 생성된다.
  - `method1()`는 `m1`, `cal` 지역 변수(매개변수 포함)를 가지므로 해당 지역 변수들이 스택 프레임에 포함된다.
- `method1()`은 `method2()`를 호출한다. `method2()` 스택 프레임이 생성된다. `method2()`는 `m2` 지역 변수(매개변수 포함)를 가지므로 해당 지역 변수가 스택 프레임에 포함된다.

종료 그림



method2() 종료 method1() 종료 main() 종료

- `method2()`가 종료된다. 이때 `method2()` 스택 프레임이 제거되고, 매개변수 `m2`도 제거된다. `method2()` 스택 프레임이 제거 되었으므로 프로그램은 `method1()`로 돌아간다. 물론 `method1()`을 처음부터 시작하는 것이 아니라 `method1()`에서 `method2()`를 호출한 지점으로 돌아간다.
- `method1()`이 종료된다. 이때 `method1()` 스택 프레임이 제거되고, 지역 변수(매개변수 포함) `m1`, `cal`도 제거된다. 프로그램은 `main()`으로 돌아간다.
- `main()`이 종료된다. 더 이상 호출할 메서드가 없고, 스택 프레임도 완전히 비워졌다. 자바는 프로그램을 정리하고 종료한다.

정리

- 자바는 스택 영역을 사용해서 메서드 호출과 지역 변수(매개변수 포함)를 관리한다.
- 메서드를 계속 호출하면 스택 프레임이 계속 쌓인다.
- 지역 변수(매개변수 포함)는 스택 영역에서 관리한다.
- 스택 프레임이 종료되면 지역 변수도 함께 제거된다.
- 스택 프레임이 모두 제거되면 프로그램도 종료된다.

스택 영역과 힙 영역

이번에는 스택 영역과 힙 영역이 함께 사용되는 경우를 알아보자.

Data

```
package memory;

public class Data {
    private int value;

    public Data(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

JavaMemoryMain2

```
package memory;

public class JavaMemoryMain2 {

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

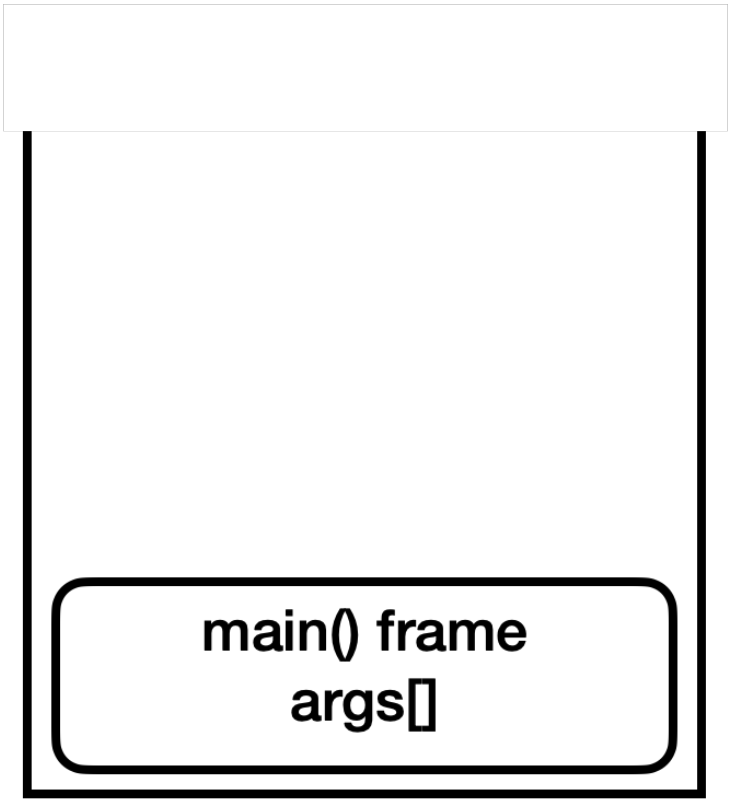
```
        System.out.println("main start");
        method1();
        System.out.println("main end");
    }

    static void method1() {
        System.out.println("method1 start");
        Data data1 = new Data(10);
        method2(data1);
        System.out.println("method1 end");
    }

    static void method2(Data data2) {
        System.out.println("method2 start");
        System.out.println("data.value=" + data2.getValue());
        System.out.println("method2 end");
    }

}
```

- `main()` ➜ `method1()` ➜ `method2()` 순서로 호출하는 단순한 코드이다.
- `method1()`에서 `Data` 클래스의 인스턴스를 생성한다.
- `method1()`에서 `method2()`를 호출할 때 매개변수에 `Data` 인스턴스의 참조값을 전달한다.

실행 결과

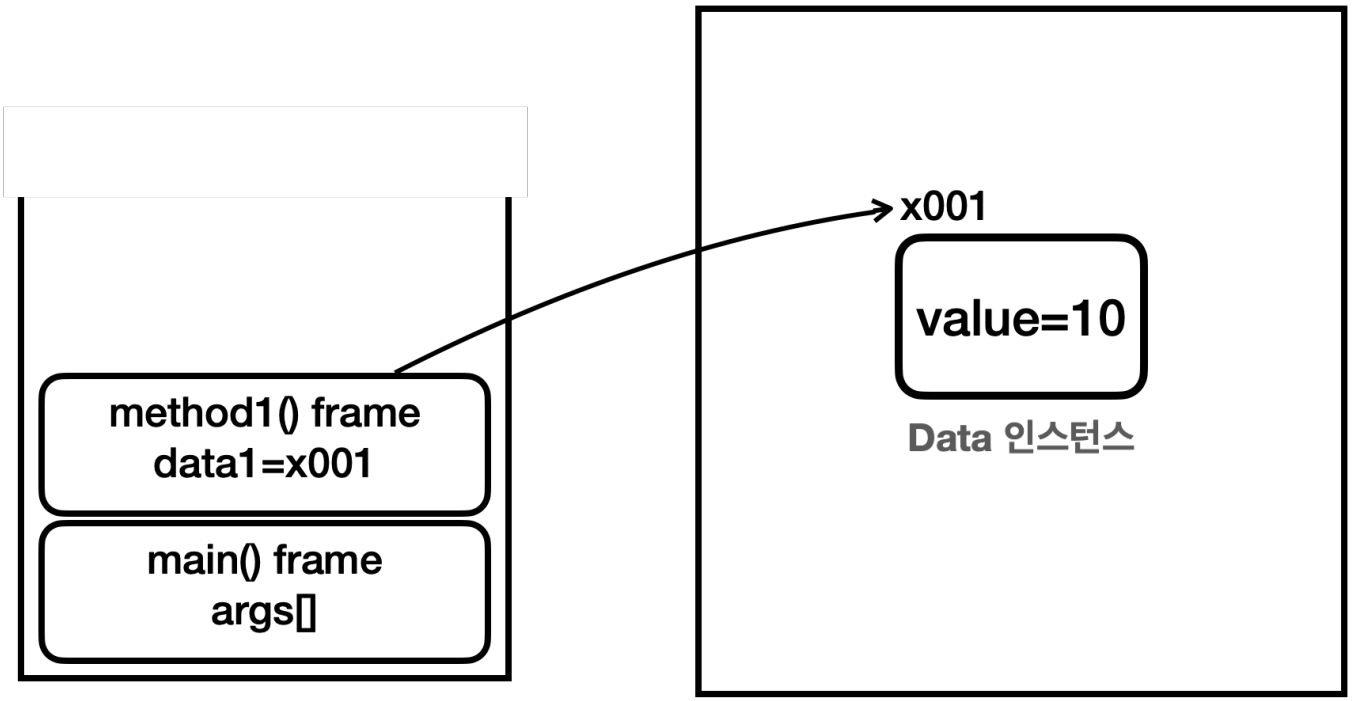
```
main start
method1 start
method2 start
data.value=10
method2 end
method1 end
main end
```

그림을 통해 순서대로 알아보자.



main() 실행

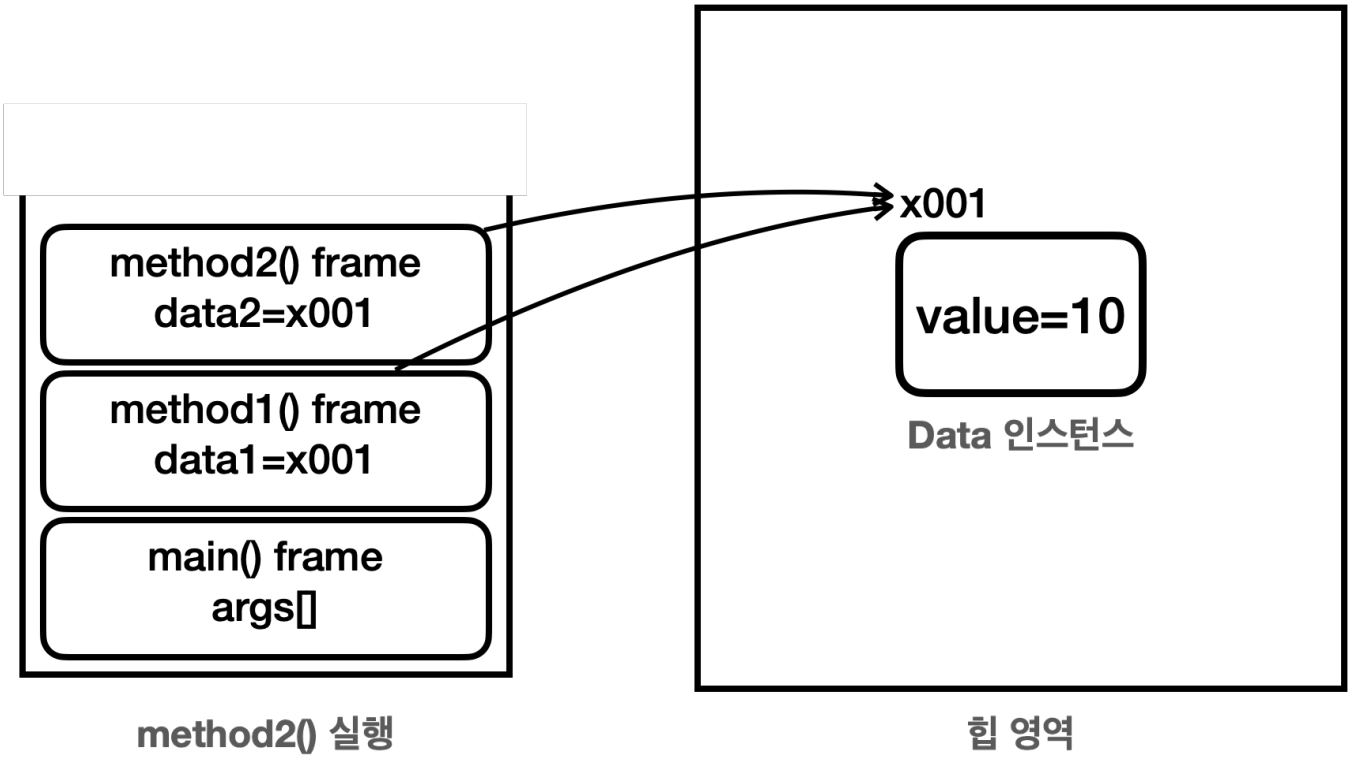
- 처음 `main()` 메서드를 실행한다. `main()` 스택 프레임이 생성된다.



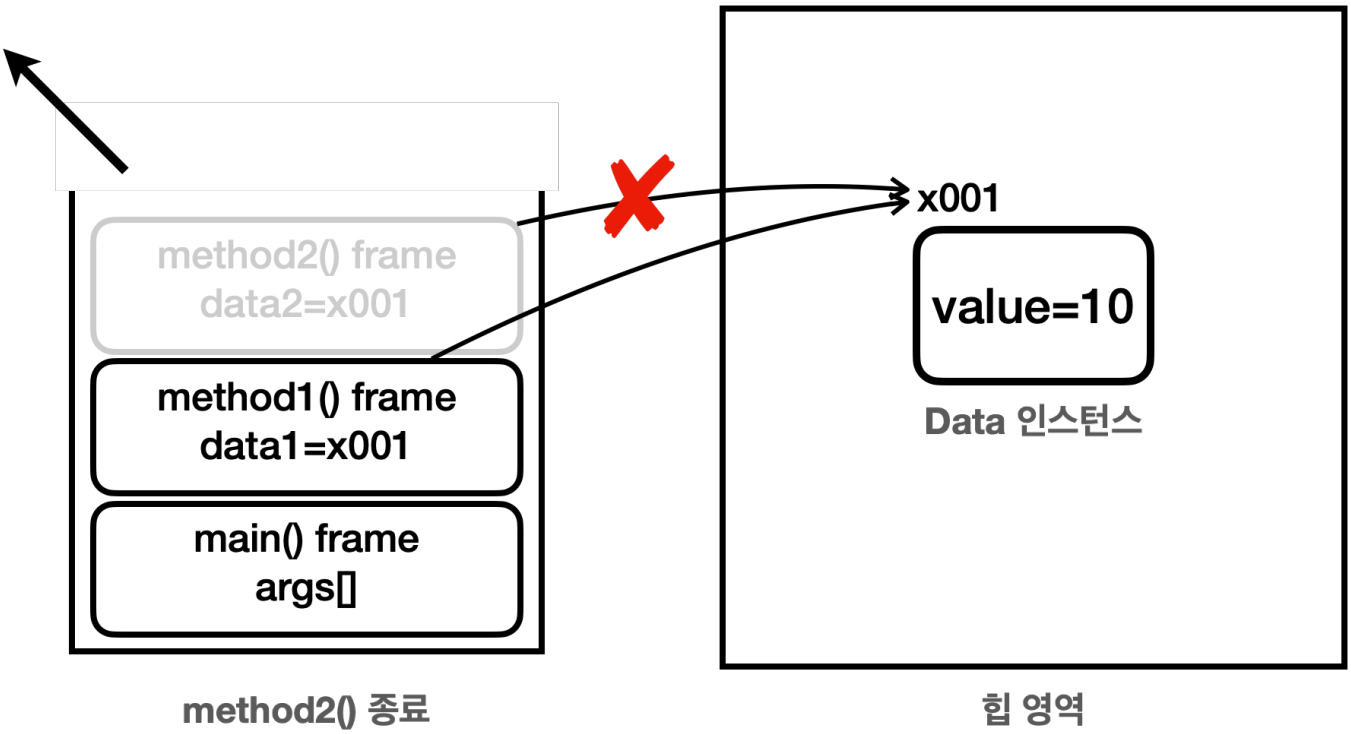
method1() 실행 힙 영역

- `main()` 에서 `method1()` 을 실행한다. `method1()` 스택 프레임이 생성된다.
- `method1()` 은 지역 변수로 `Data data1` 을 가지고 있다. 이 지역 변수도 스택 프레임에 포함된다.
- `method1()` 은 `new Data(10)` 를 사용해서 힙 영역에 `Data` 인스턴스를 생성한다. 그리고 참조값을 `data1`

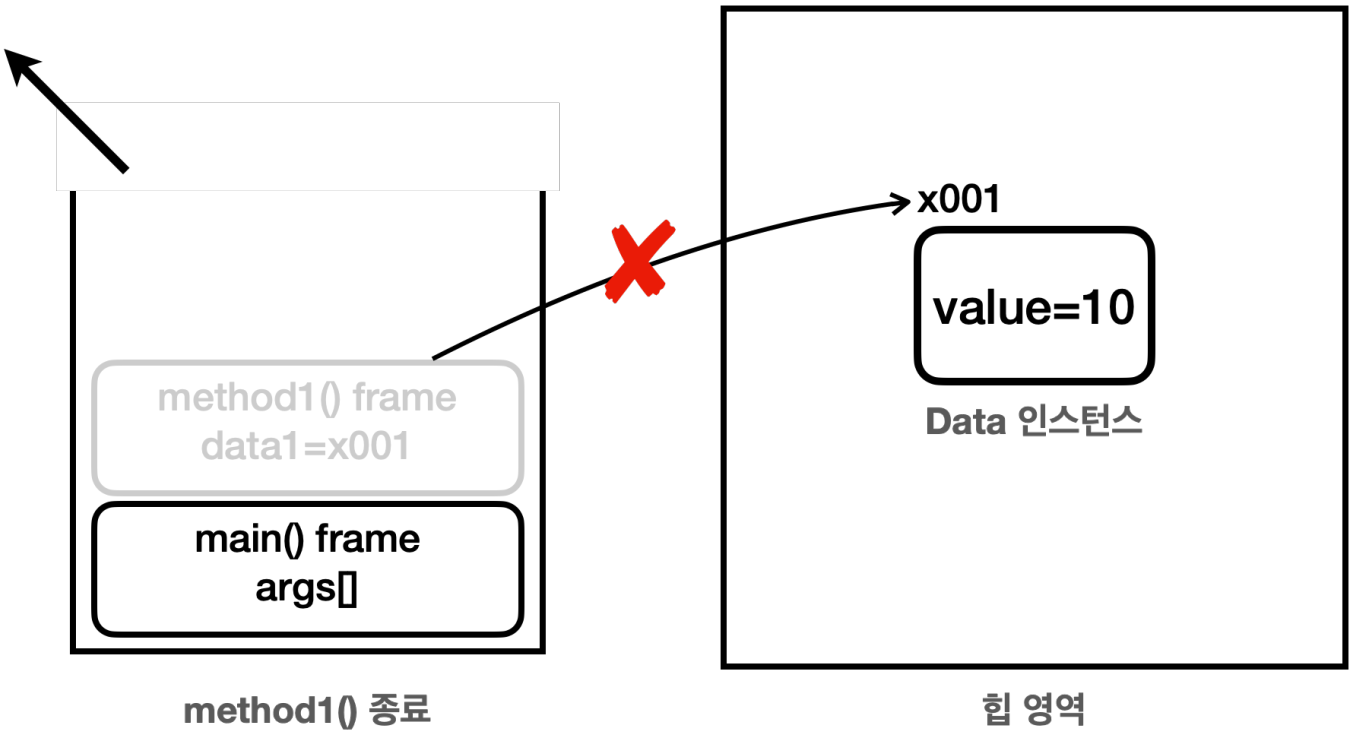
에 보관한다.



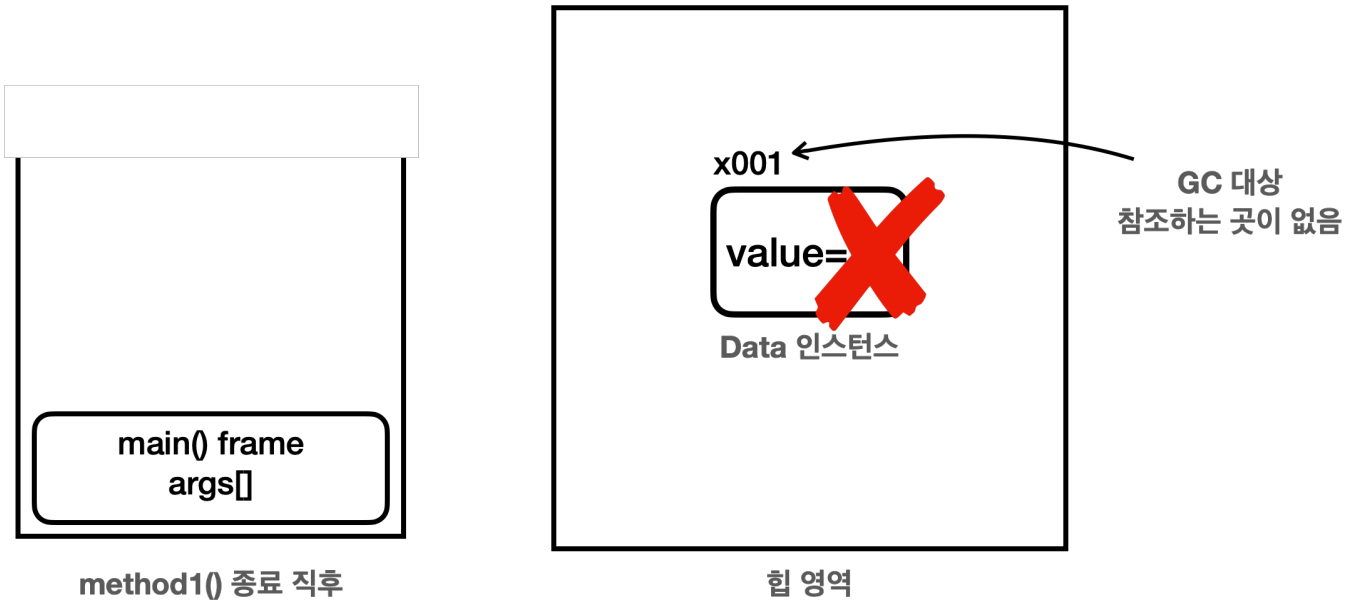
- `method1()`은 `method2()`를 호출하면서 `Data data2` 매개변수에 `x001` 참조값을 넘긴다.
- 이제 `method1()`에 있는 `data1`과 `method2()`에 있는 `data2` 지역 변수(매개변수 포함)는 둘다 같은 `x001` 인스턴스를 참조한다.



- `method2()`가 종료된다. `method2()`의 스택 프레임이 제거되면서 매개변수 `data2`도 함께 제거된다.



- `method1()`이 종료된다. `method1()`의 스택 프레임이 제거되면서 지역 변수 `data1`도 함께 제거된다.



- `method1()`이 종료된 직후의 상태를 보자. `method1()`의 스택 프레임이 제거되고 지역 변수 `data1`도 함께 제거되었다.
- 이제 `x001` 참조값을 가진 `Data` 인스턴스를 참조하는 곳이 더는 없다.
- 참조하는 곳이 없으므로 사용되는 곳도 없다. 결과적으로 프로그램에서 더는 사용하지 않는 객체인 것이다. 이런 객체는 메모리만 차지하게 된다.
- GC(가비지 컬렉션)은 이렇게 참조가 모두 사라진 인스턴스를 찾아서 메모리에서 제거한다.

참고: 힙 영역 외부가 아닌, 힙 영역 안에서만 인스턴스끼리 서로 참조하는 경우에도 GC의 대상이 된다.

정리

지역 변수는 스택 영역에, 객체(인스턴스)는 힙 영역에 관리되는 것을 확인했다. 이제 나머지 하나가 남았다. 바로 메서드 영역이다. 메서드 영역이 관리하는 변수도 있다. 이것을 이해하기 위해서는 먼저 `static` 키워드를 알아야 한다. `static` 키워드는 메서드 영역과 밀접한 연관이 있다.

static 변수1

이번에는 새로운 키워드인 `static` 키워드에 대해 학습해보자.
`static` 키워드는 주로 멤버 변수와 메서드에 사용된다.
먼저 멤버 변수에 `static` 키워드가 왜 필요한지 이해하기 위해 간단한 예제를 만들어보자.

특정 클래스를 통해서 생성된 객체의 수를 세는 단순한 프로그램이다.

인스턴스 내부 변수에 카운트 저장

먼저 생성할 인스턴스 내부에 카운트를 저장하겠다.

Data1

```
package static1;

public class Data1 {
    public String name;
    public int count;

    public Data1(String name) {
        this.name = name;
        count++;
    }

}
```

생성된 객체의 수를 세어야 한다. 따라서 객체가 생성될 때 마다 생성자를 통해 인스턴스의 멤버 변수인 `count` 값을 증가시킨다.
참고로 예제를 단순하게 만들기 위해 필드에 `public`을 사용했다.

DataCountMain1

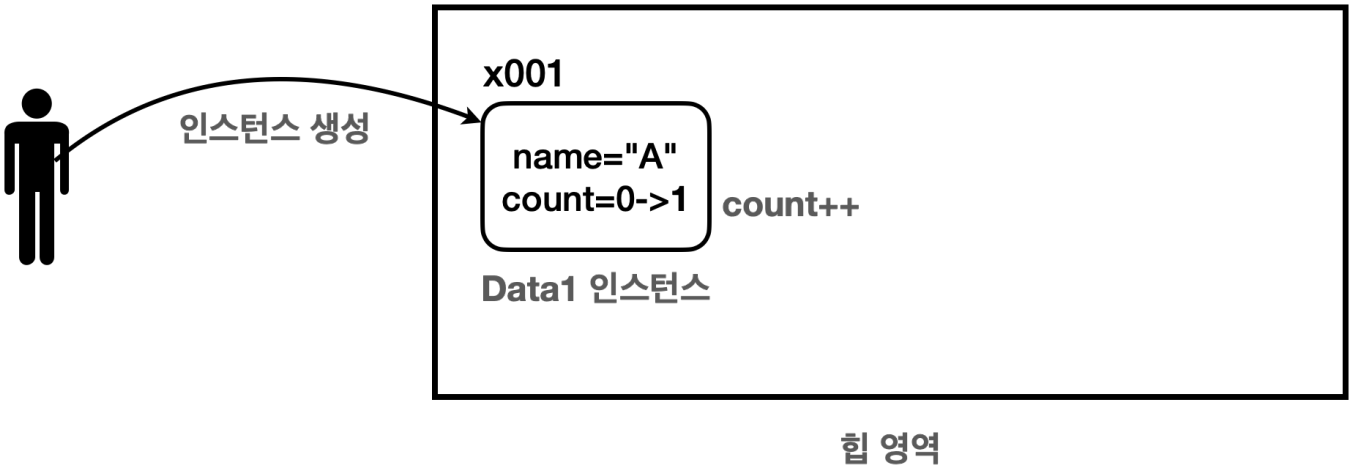
```
package static1;

public class DataCountMain1 {

    public static void main(String[] args) {
        Data1 data1 = new Data1("A");
        System.out.println("A count=" + data1.count);

        Data1 data2 = new Data1("B");
        System.out.println("B count=" + data2.count);

        Data1 data3 = new Data1("C");
        System.out.println("C count=" + data3.count);
    }
}
```

객체를 생성하고 카운트 값을 출력한다.

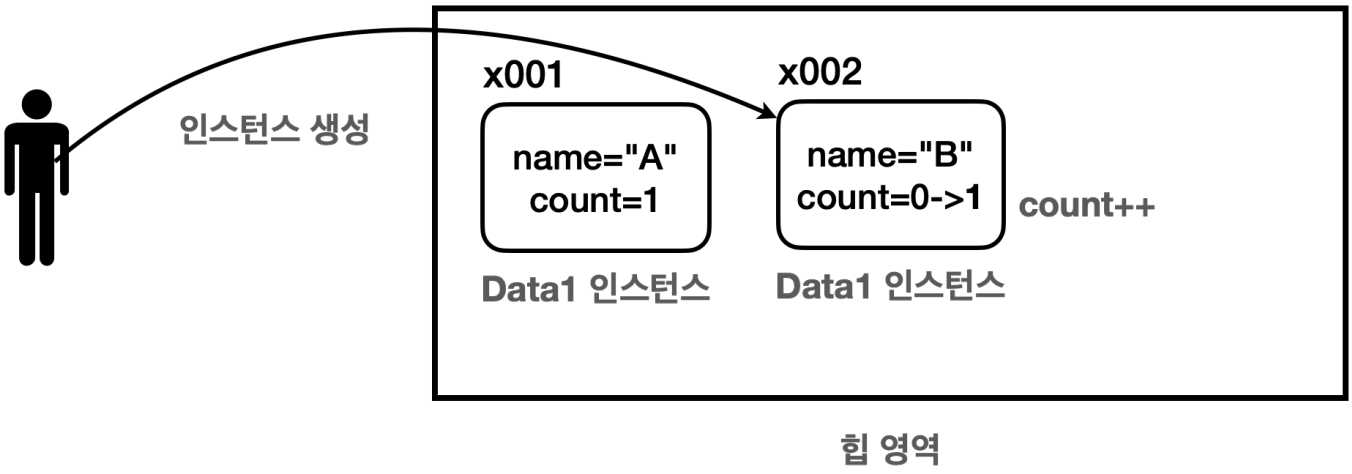
실행 결과

```
A count=1
B count=1
C count=1
```

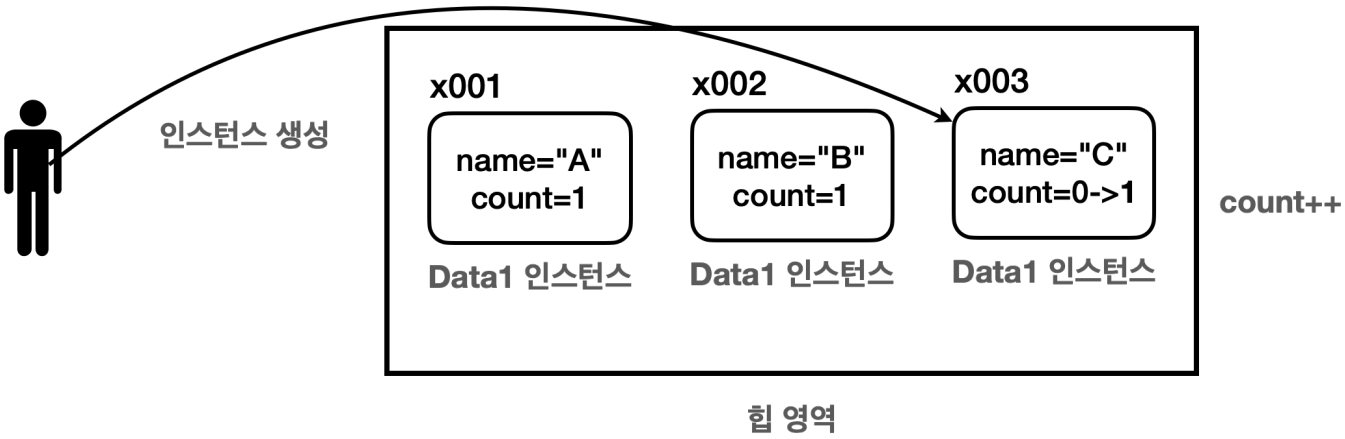
이 프로그램은 당연히 기대한 대로 작동하지 않는다. 객체를 생성할 때 마다 `Data1` 인스턴스는 새로 만들어진다. 그리고 인스턴스에 포함된 `count` 변수도 새로 만들어지기 때문이다.



처음 `Data1("A")` 인스턴스를 생성하면 `count` 값은 `0`으로 초기화 된다. 생성자에서 `count++`을 호출했으므로 `count`의 값은 `1`이 된다.



다음으로 `Data1("B")` 인스턴스를 생성하면 완전 새로운 인스턴스를 생성한다. 이 새로운 인스턴스의 `count` 값은 `0`으로 초기화 된다. 생성자에서 `count++`을 호출했으므로 `count`의 값은 `1`이 된다.



다음으로 `Data1("C")` 인스턴스를 생성하면 이전 인스턴스는 관계없는 새로운 인스턴스를 생성한다. 이 새로운 인스턴스의 `count` 값은 `0`으로 초기화 된다. 생성자에서 `count++`을 호출했으므로 `count`의 값은 `1`이 된다.

인스턴스에 사용되는 멤버 변수 `count`값은 인스턴스끼리 서로 공유되지 않는다. 따라서 원하는 답을 구할 수 없다. 이 문제를 해결하려면 변수를 서로 공유해야 한다.

외부 인스턴스에 카운트 저장

이번에는 카운트 값을 저장하는 별도의 객체를 만들어보자.

Counter

```
package static1;

public class Counter {
    public int count;
}
```

- 이 객체를 공유해서 필요할 때 마다 카운트 값을 증가할 것이다.

Data2

```
package static1;

public class Data2 {
    public String name;

    public Data2(String name, Counter counter) {
        this.name = name;
        counter.count++;
    }

}
```

- 기존 코드를 유지하기 위해 새로운 `Data2` 클래스를 만들었다. 여기에는 `count` 멤버 변수가 없다. 대신에 생성자에서 `Counter` 인스턴스를 추가로 전달 받는다.
- 생성자가 호출되면 `counter` 인스턴스에 있는 `count` 변수의 값을 하나 증가시킨다.

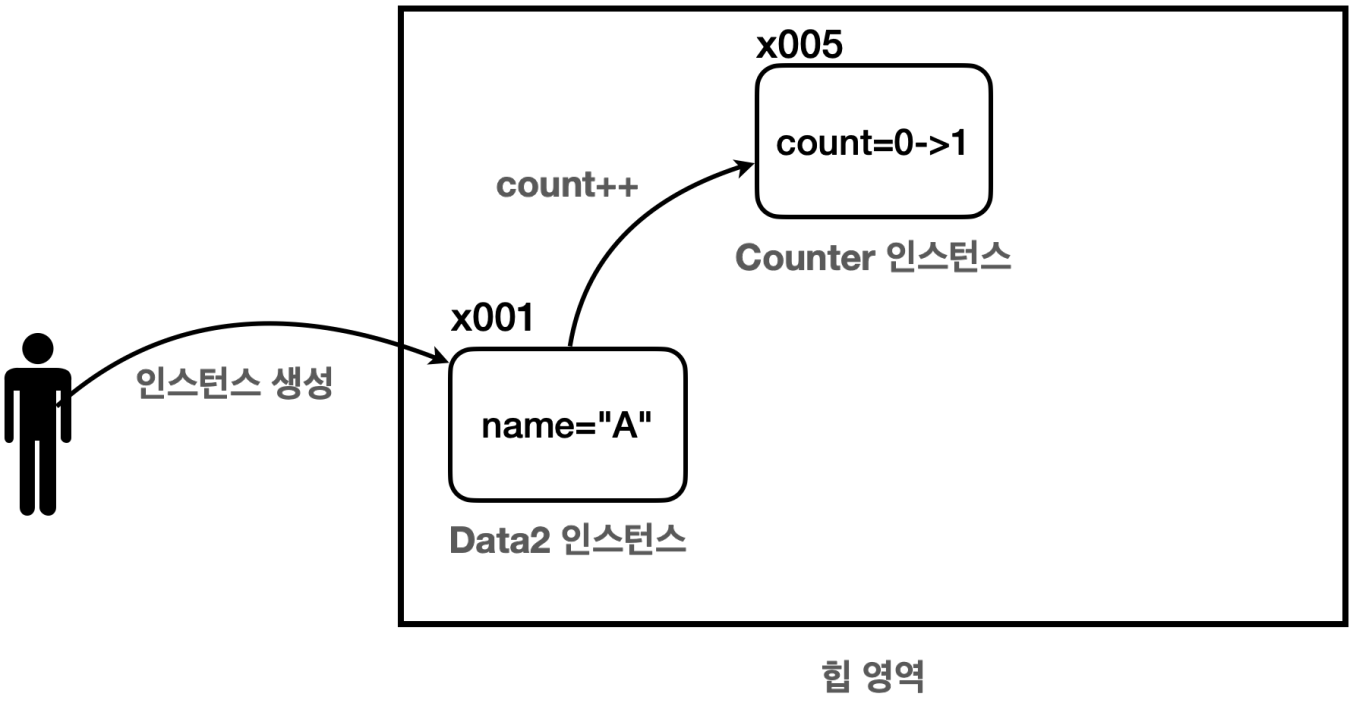
DataCountMain2

```
package static1;

public class DataCountMain2 {

    public static void main(String[] args) {
        Counter counter = new Counter();
        Data2 data1 = new Data2("A", counter);
        System.out.println("A count=" + counter.count);

        Data2 data2 = new Data2("B", counter);
        System.out.println("B count=" + counter.count);

        Data2 data3 = new Data2("C", counter);
        System.out.println("C count=" + counter.count);
    }
}
```

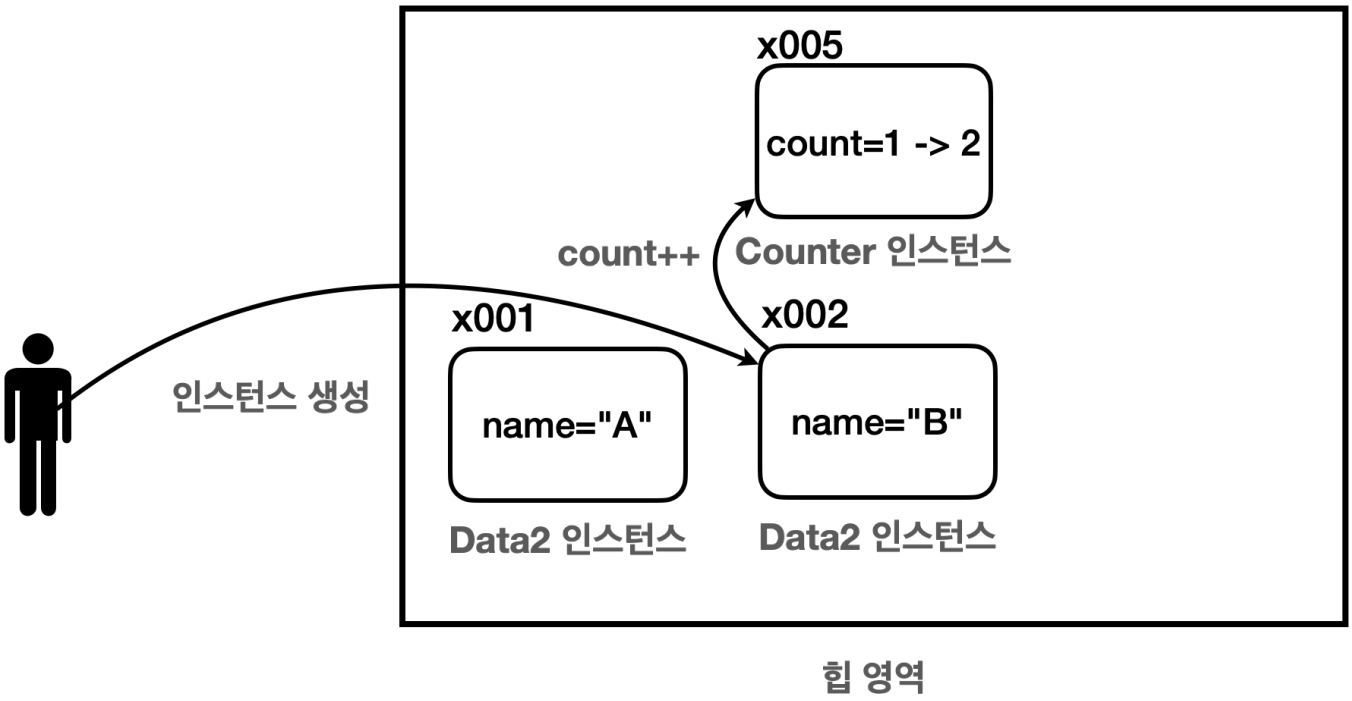
실행 결과

```
A count=1
B count=2
C count=3
```

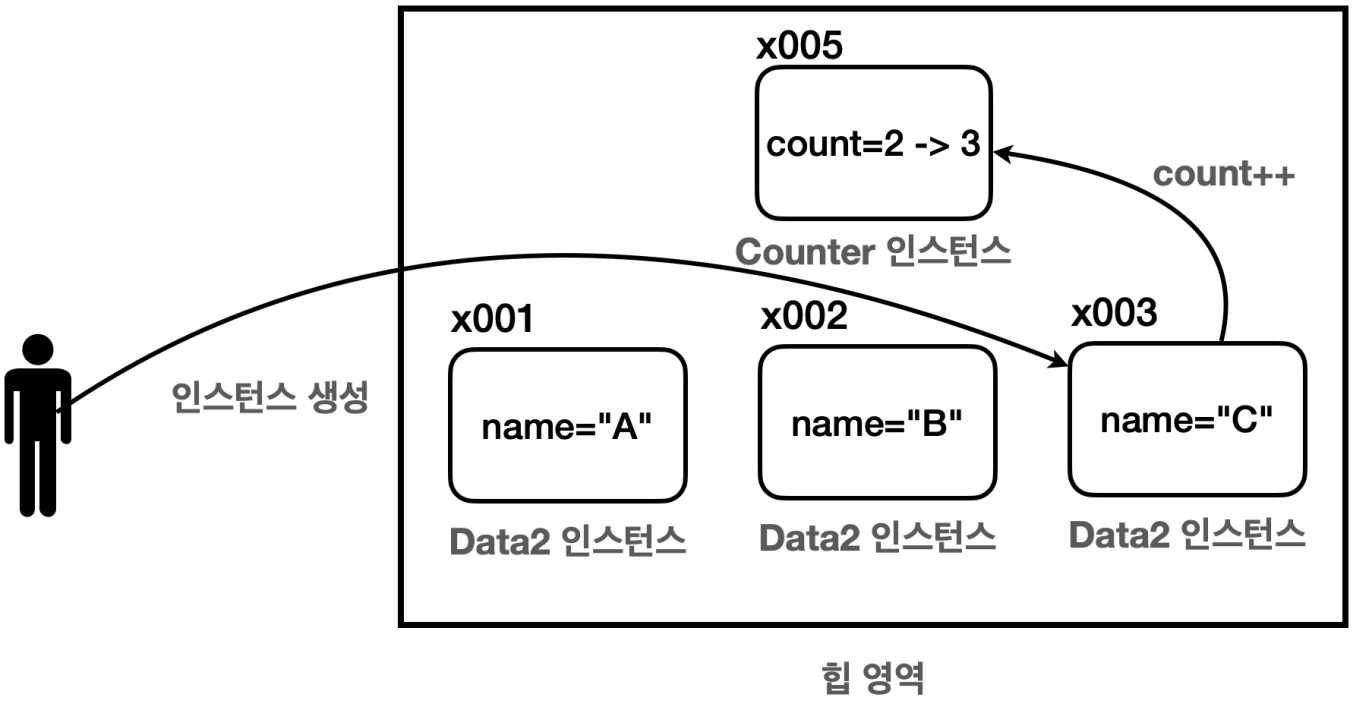
`Counter` 인스턴스를 공용으로 사용한 덕분에 객체를 생성할 때 마다 값을 정확하게 증가시킬 수 있다.



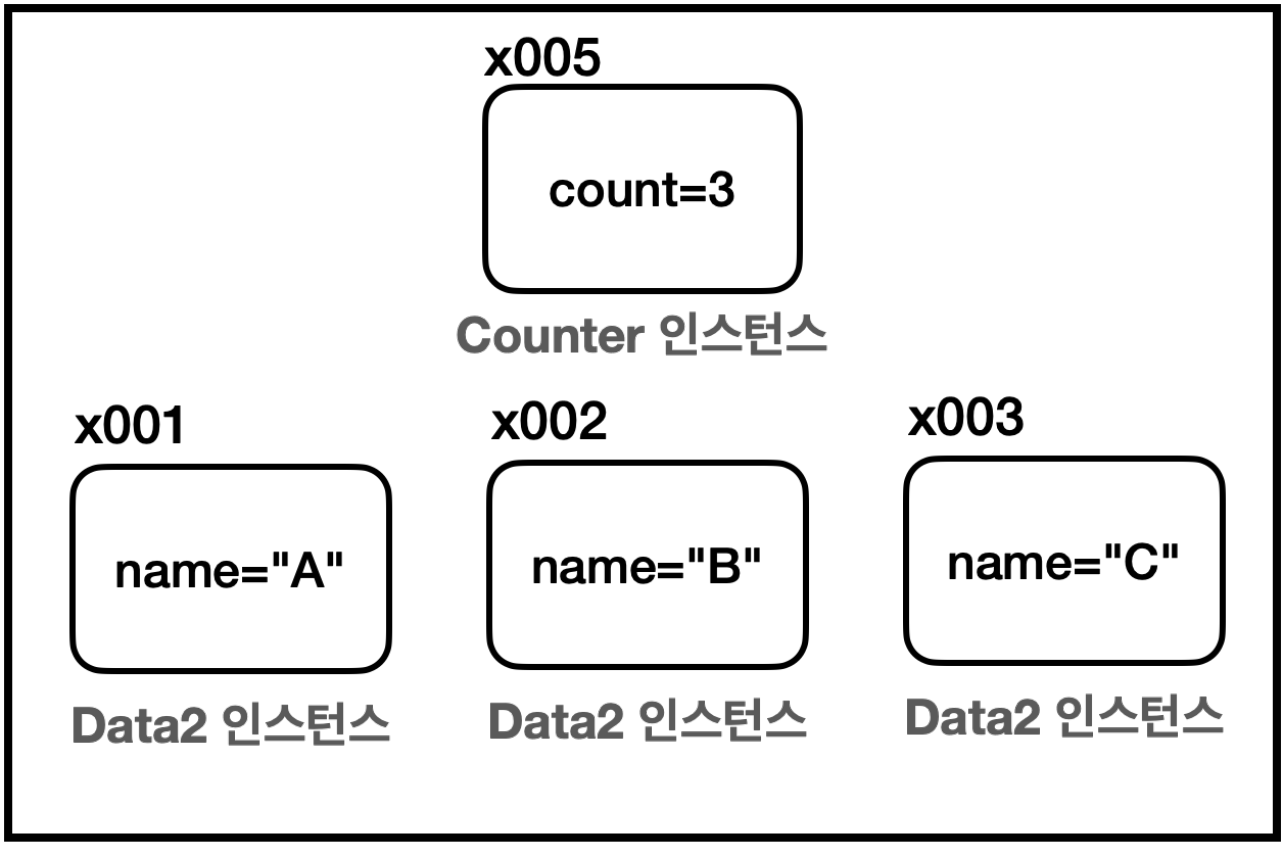
`Data2("A")` 인스턴스를 생성하면 생성자를 통해 `Counter` 인스턴스에 있는 `count` 값을 하나 증가시킨다.
`count` 값은 1이 된다.



`Data2("B")` 인스턴스를 생성하면 생성자를 통해 `Counter` 인스턴스에 있는 `count` 값을 하나 증가시킨다.
`count` 값은 2가 된다.



`Data2("C")` 인스턴스를 생성하면 생성자를 통해 `Counter` 인스턴스에 있는 `count` 값을 하나 증가시킨다.
`count` 값은 3이 된다.



힙 영역

결과적으로 `Data2`의 인스턴스가 3개 생성되고, `count` 값도 인스턴스 숫자와 같은 3으로 정확하게 측정된다.

그런데 여기에는 약간 불편한 점들이 있다.

- `Data2` 클래스와 관련된 일인데, `Counter`라는 별도의 클래스를 추가로 사용해야 한다.
- 생성자의 매개변수도 추가되고, 생성자가 복잡해진다. 생성자를 호출하는 부분도 복잡해진다.

static 변수2

static 변수 사용

특정 클래스에서 공용으로 함께 사용할 수 있는 변수를 만들 수 있다면 편리할 것이다.

`static` 키워드를 사용하면 공용으로 함께 사용하는 변수를 만들 수 있다.

Data3

```
package static1;
```

```
public class Data3 {
    public String name;
    public static int count; //static

    public Data3(String name) {
        this.name = name;
        count++;
    }

}
```

- 기존 코드를 유지하기 위해 새로운 클래스 `Data3`을 만들었다.
- `static int count`부분을 보자. 변수 타입(`int`) 앞에 `static` 키워드가 붙어있다.
- 이렇게 멤버 변수에 `static`을 붙이게 되면 `static` 변수, 정적 변수 또는 클래스 변수라 한다.
- 객체가 생성되면 생성자에서 정적 변수 `count`의 값을 하나 증가시킨다.

DataCountMain3

```
package static1;

public class DataCountMain3 {

    public static void main(String[] args) {
        Data3 data1 = new Data3("A");
        System.out.println("A count=" + Data3.count);

        Data3 data2 = new Data3("B");
        System.out.println("B count=" + Data3.count);

        Data3 data3 = new Data3("C");
        System.out.println("C count=" + Data3.count);
    }
}
```

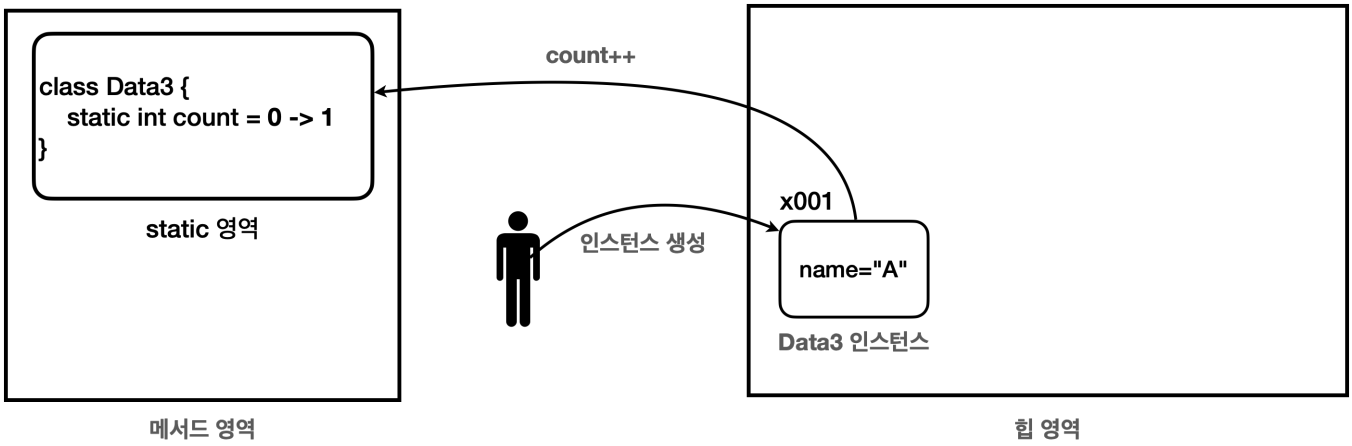
코드를 보면 `count` 정적 변수에 접근하는 방법이 조금 특이한데 `Data3.count`와 같이 클래스명에 `.`(dot)을 사용한다. 마치 클래스에 직접 접근하는 것 처럼 느껴진다.

실행 결과

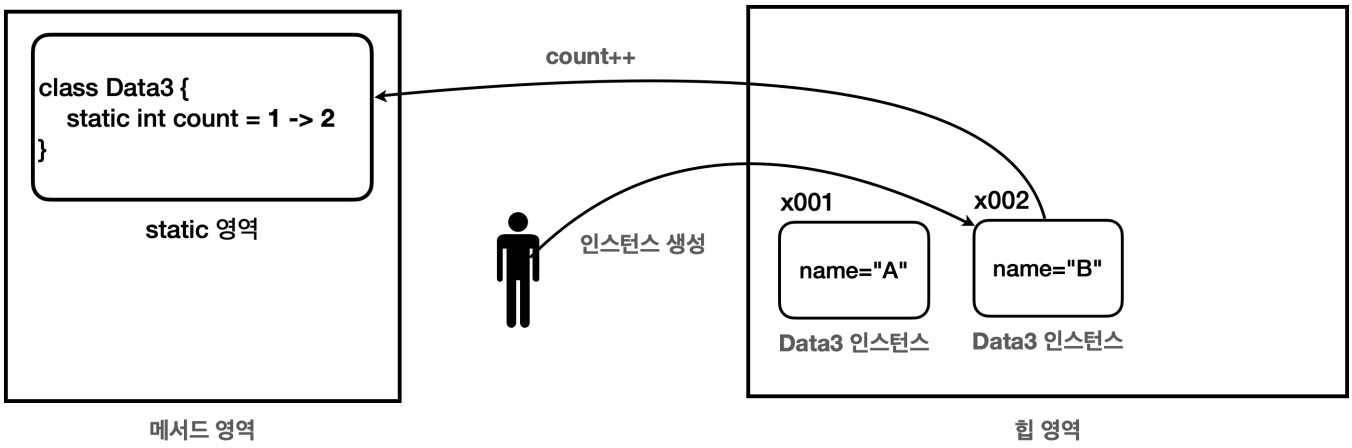
```
A count=1
B count=2
```

```
C count=3
```

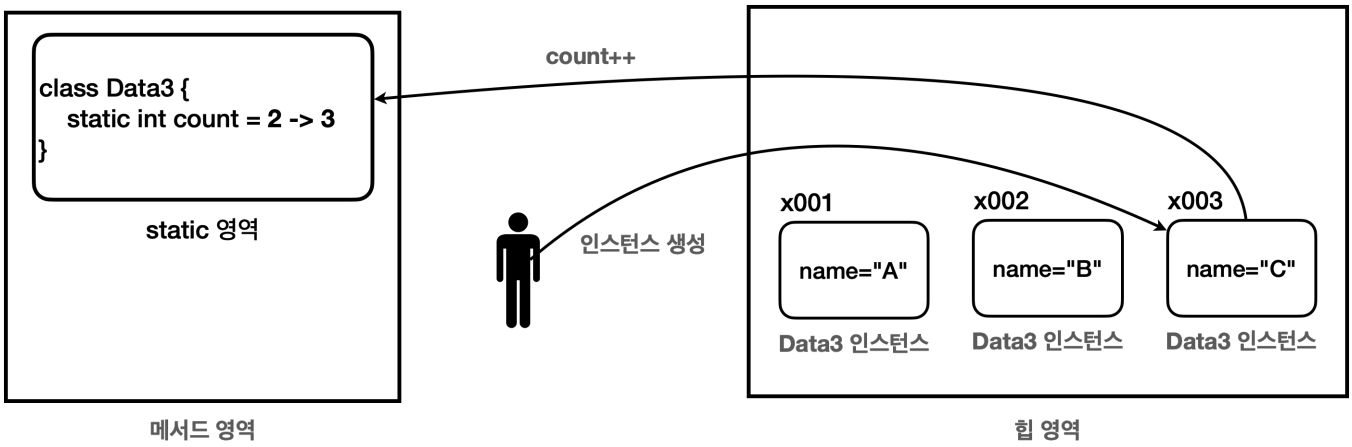
그림으로 알아보자.



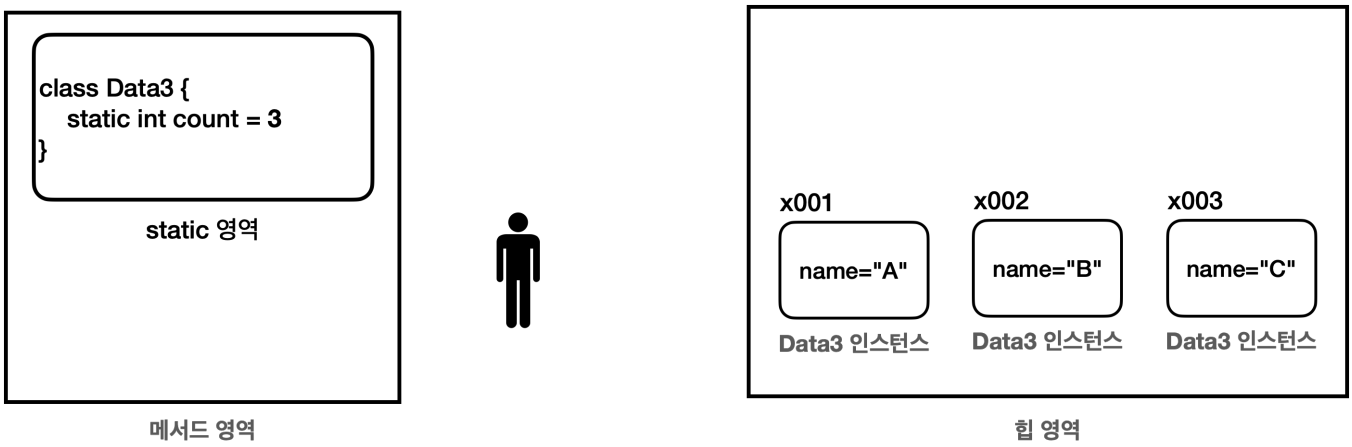
- `static`이 붙은 멤버 변수는 메서드 영역에서 관리한다.
  - `static`이 붙은 멤버 변수 `count` 는 인스턴스 영역에 생성되지 않는다. 대신에 메서드 영역에서 이 변수를 관리한다.
- `Data3("A")` 인스턴스를 생성하면 생성자가 호출된다
- 생성자에는 `count++` 코드가 있다. `count` 는 `static`이 붙은 정적 변수다. 정적 변수는 인스턴스 영역이 아니라 메서드 영역에서 관리한다. 따라서 이 경우 메서드 영역에 있는 `count`의 값이 하나 증가된다.



- `Data3("B")` 인스턴스를 생성하면 생성자가 호출된다
- `count++` 코드가 있다. `count` 는 `static`이 붙은 정적 변수다. 메서드 영역에 있는 `count` 변수의 값이 하나 증가된다.



- `Data3("C")` 인스턴스를 생성하면 생성자가 호출된다
- `count++` 코드가 있다. `count` 는 `static` 이 붙은 정적 변수다. 메서드 영역에 있는 `count` 변수의 값이 하나 증가된다.



최종적으로 메서드 영역에 있는 `count` 변수의 값은 3이 된다.

`static` 이 붙은 정적 변수에 접근하려면 `Data3.count` 와 같이 클래스명 + `.` (dot) + 변수명으로 접근하면 된다.

참고로 `Data3` 의 생성자와 같이 자신의 클래스에 있는 정적 변수라면 클래스명을 생략할 수 있다.

`static` 변수를 사용한 덕분에 공용 변수를 사용해서 편리하게 문제를 해결할 수 있었다.

정리

`static` 변수는 쉽게 이야기해서 클래스인 붕어빵 틀이 특별히 관리하는 변수이다. 붕어빵 틀은 1개이므로 클래스 변수도 하나만 존재한다. 반면에 인스턴스 변수는 붕어빵인 인스턴스의 수 만큼 존재한다.

static 변수3

이번에는 `static` 변수를 정리해보자.

용어 정리

```
public class Data3 {
    public String name;
    public static int count; //static
}
```

예제 코드에서 `name`, `count`는 둘다 멤버 변수이다.
멤버 변수(필드)는 `static`이 붙은 것과 아닌 것에 따라 다음과 같이 분류할 수 있다.

멤버 변수(필드)의 종류

- **인스턴스 변수**: `static`이 붙지 않은 멤버 변수, 예) `name`
  - `static`이 붙지 않은 멤버 변수는 인스턴스를 생성해야 사용할 수 있고, 인스턴스에 소속되어 있다. 따라서 인스턴스 변수라 한다.
  - 인스턴스 변수는 인스턴스를 만들 때 마다 새로 만들어진다.
- **클래스 변수**: `static`이 붙은 멤버 변수, 예) `count`
  - 클래스 변수, 정적 변수, `static` 변수등으로 부른다. **용어를 모두 사용하니 주의하자**
  - `static`이 붙은 멤버 변수는 인스턴스와 무관하게 클래스에 바로 접근해서 사용할 수 있고, 클래스 자체에 소속되어 있다. 따라서 클래스 변수라 한다.
  - 클래스 변수는 자바 프로그램을 시작할 때 딱 1개가 만들어진다. 인스턴스와는 다르게 보통 여러곳에서 공유하는 목적으로 사용된다.

변수와 생명주기

- **지역 변수(매개변수 포함)**: 지역 변수는 스택 영역에 있는 스택 프레임 안에 보관된다. 메서드가 종료되면 스택 프레임도 제거 되는데 이때 해당 스택 프레임에 포함된 지역 변수도 함께 제거된다. 따라서 지역 변수는 생존 주기가 짧다.
- **인스턴스 변수**: 인스턴스에 있는 멤버 변수를 인스턴스 변수라 한다. 인스턴스 변수는 힙 영역을 사용한다. 힙 영역은 GC(가비지 컬렉션)가 발생하기 전까지는 생존하기 때문에 보통 지역 변수보다 생존 주기가 길다.
- **클래스 변수**: 클래스 변수는 메서드 영역의 static 영역에 보관되는 변수이다. 메서드 영역은 프로그램 전체에서 사용하는 공용 공간이다. 클래스 변수는 해당 클래스가 JVM에 로딩 되는 순간 생성된다. 그리고 JVM이 종료될 때 까지 생명주기가 이어진다. 따라서 가장 긴 생명주기를 가진다.

`static`이 정적이라는 이유는 바로 여기에 있다. 힙 영역에 생성되는 인스턴스 변수는 동적으로 생성되고, 제거된다. 반면에 `static`인 정적 변수는 거의 프로그램 실행 시점에 딱 만들어지고, 프로그램 종료 시점에 제거된다. 정적 변수는 이름 그대로 정적이다.

정적 변수 접근 법

`static` 변수는 클래스를 통해 바로 접근할 수도 있고, 인스턴스를 통해서도 접근할 수 있다.
`DataCountMain3` 마지막 코드에 다음 부분을 추가하고 실행해보자.

DataCountMain3 - 추가

```
//추가
//인스턴스를 통한 접근
Data3 data4 = new Data3("D");
System.out.println(data4.count);

//클래스를 통합 접근
System.out.println(Data3.count);
```

실행 결과 - 추가된 부분

```
4
4
```

둘의 차이는 없다. 둘다 결과적으로 정적 변수에 접근한다.

인스턴스를 통한 접근 `data4.count`
정적 변수의 경우 인스턴스를 통한 접근은 추천하지 않는다. 왜냐하면 코드를 읽을 때 마치 인스턴스 변수에 접근하는 것 처럼 오해할 수 있기 때문이다.

클래스를 통한 접근 `Data3.count`
정적 변수는 클래스에서 공용으로 관리하기 때문에 클래스를 통해서 접근하는 것이 더 명확하다. 따라서 정적 변수에 접근할 때는 클래스를 통해서 접근하자.

static 메서드1

이번에는 `static`이 붙은 메서드에 대해 알아보자.

이해를 돕기 위해 간단한 예제를 만들어보자.

특정 문자열을 꾸며주는 간단한 기능을 만들어보자.
예를 들어서 `"hello"` 라는 문자열 앞 뒤에 `*` 을 붙여서 `"*hello*"` 와 같이 꾸며주는 기능이다.

인스턴스 메서드

먼저 지금까지 학습한 방식을 통해 해당 기능을 개발해보자.

```
package static2;

public class DecoUtil1 {

    public String deco(String str) {
        String result = "*" + str + "*";
        return result;
    }
}
```

`deco()` 는 문자열을 꾸미는 기능을 제공한다. 문자열이 들어오면 앞 뒤에 `*` 을 붙여서 반환한다.

```
package static2;

public class DecoMain1 {

    public static void main(String[] args) {
        String s = "hello java";
        DecoUtil1 utils = new DecoUtil1();
        String deco = utils.deco(s);

        System.out.println("before: " + s);
        System.out.println("after: " + deco);
    }
}
```

실행 결과

```
before: hello java
```

```
after: *hello java*
```

앞서 개발한 `deco()` 메서드를 호출하기 위해서는 `DecoUtil1`의 인스턴스를 먼저 생성해야 한다. 그런데 `deco()`라는 기능은 멤버 변수도 없고, 단순히 기능만 제공할 뿐이다. 인스턴스가 필요한 이유는 멤버 변수(인스턴스 변수)등을 사용하는 목적이 큰데, 이 메서드는 사용하는 인스턴스 변수도 없고 단순히 기능만 제공한다.

static 메서드

먼저 예제를 만들어서 실행해보자.

```
package static2;

public class DecoUtil2 {

    public static String deco(String str) {
        String result = "*" + str + "*";
        return result;
    }

}
```

`DecoUtil2`는 앞선 예제와 비슷한데, 메서드 앞에 `static`이 붙어있다. 이 부분에 주의하자. 이렇게 하면 정적 메서드를 만들 수 있다. 그리고 이 정적 메서드는 정적 변수처럼 인스턴스 생성 없이 클래스 명을 통해서 바로 호출할 수 있다.

```
package static2;

public class DecoMain2 {

    public static void main(String[] args) {
        String s = "hello java";
        String deco = DecoUtil2.deco(s);

        System.out.println("before: " + s);
        System.out.println("after: " + deco);
    }
}
```

실행 결과

```
before: hello java
after: *hello java*
```

`DecoUtil2.deco(s)` 코드를 보자.
`static`이 붙은 정적 메서드는 객체 생성 없이 클래스명 + `.`(dot) + 메서드 명으로 바로 호출할 수 있다.
정적 메서드 덕분에 불필요한 객체 생성 없이 편리하게 메서드를 사용했다.

클래스 메서드

메서드 앞에도 `static`을 붙일 수 있다. 이것을 **정적 메서드** 또는 **클래스 메서드**라 한다. 정적 메서드라는 용어는 `static`이 정적이라는 뜻이기 때문이고, 클래스 메서드라는 용어는 인스턴스 생성 없이 마치 클래스에 있는 메서드를 바로 호출하는 것 처럼 느껴지기 때문이다.

인스턴스 메서드

`static`이 붙지 않은 메서드는 인스턴스를 생성해야 호출할 수 있다. 이것을 **인스턴스 메서드**라 한다.

static 메서드2

정적 메서드는 객체 생성없이 클래스에 있는 메서드를 바로 호출할 수 있다는 장점이 있다.
하지만 정적 메서드는 언제나 사용할 수 있는 것이 아니다.

정적 메서드 사용법

- `static` 메서드는 `static`만 사용할 수 있다.
  - 클래스 내부의 기능을 사용할 때, 정적 메서드는 `static`이 붙은 **정적 메서드나 정적 변수만 사용할 수 있다.**
  - 클래스 내부의 기능을 사용할 때, 정적 메서드는 인스턴스 변수나, 인스턴스 메서드를 사용할 수 없다.
- 반대로 모든 곳에서 `static`을 호출할 수 있다.
  - 정적 메서드는 공용 기능이다. 따라서 접근 제어자만 허락한다면 클래스를 통해 모든 곳에서 `static`을 호출할 수 있다.

예제를 통해 정적 메서드의 사용법을 확인해보자.

DecoData

```
package static2;

public class DecoData {

    private int instanceValue;
    private static int staticValue;

    public static void staticCall() {
        //instanceValue++; //인스턴스 변수 접근, compile error
        //instanceMethod(); //인스턴스 메서드 접근, compile error

        staticValue++; //정적 변수 접근
        staticMethod(); //정적 메서드 접근
    }

    public void instanceCall() {
        instanceValue++; //인스턴스 변수 접근
        instanceMethod(); //인스턴스 메서드 접근

        staticValue++; //정적 변수 접근
        staticMethod(); //정적 메서드 접근
    }

    private void instanceMethod() {
        System.out.println("instanceValue=" + instanceValue);
    }
    private static void staticMethod() {
        System.out.println("staticValue=" + staticValue);
    }
}
```

이번 예제에서는 접근 제어자를 적극 활용해서 필드를 포함한 외부에서 직접 필요하지 않은 기능은 모두 막아두었다.

- `instanceValue`는 인스턴스 변수이다.
- `staticValue`는 정적 변수(클래스 변수)이다.
- `instanceMethod()`는 인스턴스 메서드이다.
- `staticMethod()`는 정적 메서드(클래스 메서드)이다.

`staticCall()` 메서드를 보자.
이 메서드는 정적 메서드이다. 따라서 `static`만 사용할 수 있다. 정적 변수, 정적 메서드에는 접근할 수 있지만, `static`이 없는 인스턴스 변수나 인스턴스 메서드에 접근하면 **컴파일 오류가 발생**한다.
코드를 보면 `staticCall()` ➜ `staticMethod()`로 `static`에서 `static`을 호출하는 것을 확인할 수 있다.

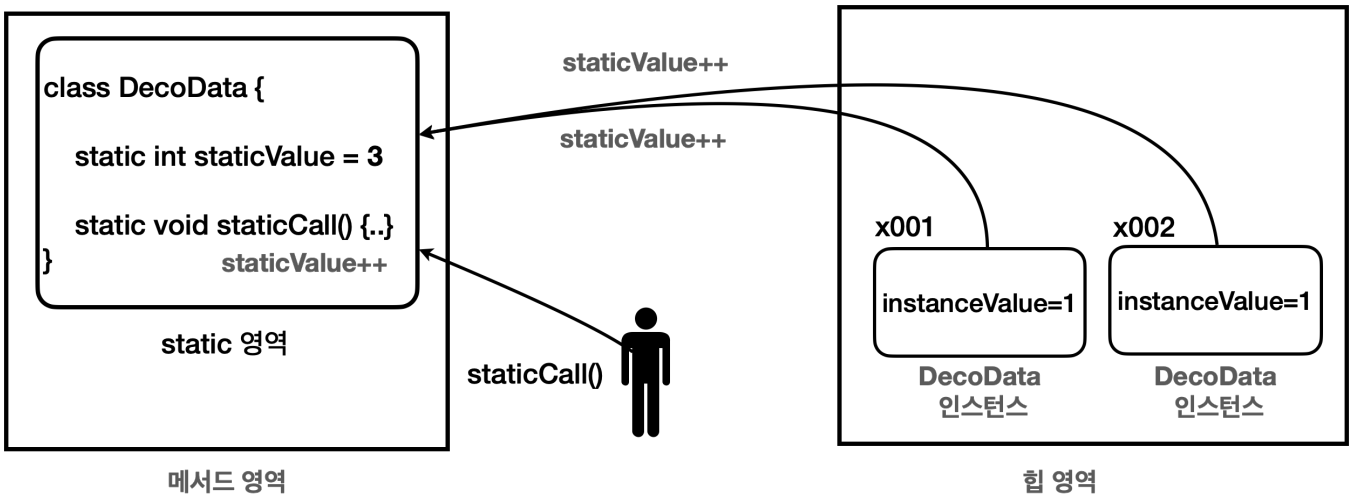
`instanceCall()` 메서드를 보자.
이 메서드는 인스턴스 메서드이다. 모든 곳에서 공용인 `static`을 호출할 수 있다. 따라서 정적 변수, 정적 메서드에 접근할 수 있다. 물론 인스턴스 변수, 인스턴스 메서드에도 접근할 수 있다.

DecoDataMain

```
package static2;

public class DecoDataMain {

    public static void main(String[] args) {
        System.out.println("1.정적 호출");
        DecoData.staticCall();

        System.out.println("2.인스턴스 호출1");
        DecoData data1 = new DecoData();
        data1.instanceCall();

        System.out.println("3.인스턴스 호출2");
        DecoData data2 = new DecoData();
        data2.instanceCall();
    }
}
```

실행 결과

```
1.정적 호출
staticValue=1
2.인스턴스 호출1
instanceValue=1
staticValue=2
3.인스턴스 호출2
instanceValue=1
staticValue=3
```



정적 메서드가 인스턴스의 기능을 사용할 수 없는 이유

정적 메서드는 클래스의 이름을 통해 바로 호출할 수 있다. 그래서 인스턴스처럼 참조값의 개념이 없다.

특정 인스턴스의 기능을 사용하려면 참조값을 알아야 하는데, 정적 메서드는 참조값 없이 호출한다. 따라서 정적 메서드 내부에서 인스턴스 변수나 인스턴스 메서드를 사용할 수 없다.

물론 당연한 이야기지만 다음과 같이 객체의 참조값을 직접 매개변수로 전달하면 정적 메서드도 인스턴스의 변수나 메서드를 호출할 수 있다.

```
public static void staticCall(DecoData data) {
    data.instanceValue++;
    data.instanceMethod();
}
```

static 메서드3

용어 정리

멤버 메서드의 종류

- **인스턴스 메서드**: `static`이 붙지 않은 멤버 메서드
- **클래스 메서드**: `static`이 붙은 멤버 메서드
  - 클래스 메서드, 정적 메서드, `static` 메서드등으로 부른다.

`static`이 붙지 않은 멤버 메서드는 인스턴스를 생성해야 사용할 수 있고, 인스턴스에 소속되어 있다. 따라서 인스턴스 메서드라 한다. `static`이 붙은 멤버 메서드는 인스턴스와 무관하게 클래스에 바로 접근해서 사용할 수 있고, 클래스 자체에 소속되어 있다. 따라서 클래스 메서드라 한다.

참고로 방금 설명한 내용은 멤버 변수에도 똑같이 적용된다.

정적 메서드 활용
정적 메서드는 객체 생성이 필요 없이 메서드의 호출만으로 필요한 기능을 수행할 때 주로 사용한다.
예를 들어 간단한 메서드 하나로 끝나는 유틸리티성 메서드에 자주 사용한다. 수학의 여러가지 기능을 담은 클래스를 만들 수 있는데, 이 경우 인스턴스 변수 없이 입력한 값을 계산하고 반환하는 것이 대부분이다. 이럴 때 정적 메서드를 사용해서 유틸리티성 메서드를 만들면 좋다.

정적 메서드 접근 법

`static` 메서드는 `static` 변수와 마찬가지로 클래스를 통해 바로 접근할 수 있고, 인스턴스를 통해서도 접근할 수 있다.

DecoDataMain - 추가

```
//추가
//인스턴스를 통한 접근
DecoData data3 = new DecoData();
data3.staticCall();

//클래스를 통한 접근
DecoData.staticCall();
```

실행 결과 - 추가된 부분

```
staticValue=4
staticValue=5
```

둘의 차이는 없다. 둘다 결과적으로 정적 메서드에 접근한다.

인스턴스를 통한 접근 `data3.staticCall()`
정적 메서드의 경우 인스턴스를 통한 접근은 추천하지 않는다. 왜냐하면 코드를 읽을 때 마치 인스턴스 메서드에 접근하는 것 처럼 오해할 수 있기 때문이다.

클래스를 통한 접근 `DecoData.staticCall()`

정적 메서드는 클래스에서 공용으로 관리하기 때문에 클래스를 통해서 접근하는 것이 더 명확하다. 따라서 정적 메서드에 접근할 때는 클래스를 통해서 접근하자.

static import

정적 메서드를 사용할 때 해당 메서드를 다음과 같이 자주 호출해야 한다면 `static import` 기능을 고려하자.

```
DecoData.staticCall();
DecoData.staticCall();
DecoData.staticCall();
```

이 기능을 사용하면 다음과 같이 클래스 명을 생략하고 메서드를 호출할 수 있다.

```
staticCall();
staticCall();
staticCall();
```

DecoDataMain - static import 적용

```
package static2;

//import static static2.DecoData.staticCall;
import static static2.DecoData.*;

public class DecoDataMain {

    public static void main(String[] args) {
        System.out.println("1.정적 호출");
        staticCall(); //클래스 명 생략 가능
    ...
}
```

특정 클래스의 정적 메서드 하나만 적용하려면 다음과 같이 생략할 메서드 명을 적어주면 된다.

`import static static2.DecoData.staticCall;`

특정 클래스의 모든 정적 메서드에 적용하려면 다음과 같이 `*`을 사용하면 된다.
`import static static2.DecoData.*;`

참고로 `import static`은 정적 메서드 뿐만 아니라 정적 변수에도 사용할 수 있다.

main() 메서드는 정적 메서드

인스턴스 생성 없이 실행하는 가장 대표적인 메서드가 바로 `main()` 메서드이다.
`main()` 메서드는 프로그램을 시작하는 시작점이 되는데, 생각해보면 객체를 생성하지 않아도 `main()` 메서드가 작동했다. 이것은 `main()` 메서드가 `static`이기 때문이다.

정적 메서드는 정적 메서드만 호출할 수 있다. 따라서 정적 메서드인 `main()`이 호출하는 메서드에는 정적 메서드를 사용했다.
물론 더 정확히 말하자면 정적 메서드는 같은 클래스 내부에서 정적 메서드만 호출할 수 있다. 따라서 정적 메서드인 `main()` 메서드가 같은 클래스에서 호출하는 메서드도 정적 메서드로 선언해서 사용했다.

main() 메서드와 static 메서드 호출 예

```
public class ValueDataMain {

    public static void main(String[] args) {
        ValueData valueData = new ValueData();
        add(valueData);
    }

    static void add(ValueData valueData) {
        valueData.value++;
        System.out.println("숫자 증가 value=" + valueData.value);
    }
}
```

문제와 풀이

문제1: 구매한 자동차 수

다음 코드를 참고해서 생성한 차량 수를 출력하는 프로그램을 작성하자.

`Car` 클래스를 작성하자.

```
package static2.ex;

public class CarMain {

    public static void main(String[] args) {
        Car car1 = new Car("K3");
        Car car2 = new Car("G80");
        Car car3 = new Car("Model Y");

        Car.showTotalCars(); //구매한 차량 수를 출력하는 static 메서드
    }
}
```

실행 결과

```
차량 구입, 이름: K3
차량 구입, 이름: G80
차량 구입, 이름: Model Y
구매한 차량 수: 3
```

정답

```
package static2.ex;

public class Car {

    private static int totalCars;
    private String name;

    public Car(String name) {
        System.out.println("차량 구입, 이름: " + name);
        this.name = name;
        totalCars++;
```

```
    }

    public static void showTotalCars() {
        System.out.println("구매한 차량 수: " + totalCars);
    }

}
```

문제2: 수학 유틸리티 클래스

다음 기능을 제공하는 배열용 수학 유틸리티 클래스(`MathArrayUtils`)를 만드세요.

- `sum(int[] array)` : 배열의 모든 요소를 더하여 합계를 반환합니다.
- `average(int[] array)` : 배열의 모든 요소의 평균값을 계산합니다.
- `min(int[] array)` : 배열에서 최소값을 찾습니다.
- `max(int[] array)` : 배열에서 최대값을 찾습니다.

요구사항

- `MathArrayUtils`은 객체를 생성하지 않고 사용해야 합니다. 누군가 실수로 `MathArrayUtils`의 인스턴스를 생성하지 못하게 막으세요.
- 실행 코드에 `static import`를 사용해도 됩니다.

실행 코드와 실행 결과를 참고하세요.

실행 코드

```
package static2.ex;

public class MathArrayUtilsMain {

    public static void main(String[] args) {
        int[] values = {1, 2, 3, 4, 5};
        System.out.println("sum=" + MathArrayUtils.sum(values));
        System.out.println("average=" + MathArrayUtils.average(values));
        System.out.println("min=" + MathArrayUtils.min(values));
        System.out.println("max=" + MathArrayUtils.max(values));
    }
}
```

실행 결과

```
sum=15
average=3.0
min=1
max=5
```

정답

```
package static2.ex;

public class MathArrayUtils {

    private MathArrayUtils() {
        //private 인스턴스 생성을 막는다.
    }

    public static int sum(int[] values) {
        int total = 0;
        for (int value : values) {
            total += value;
        }
        return total;
    }

    public static double average(int[] values) {
        return (double) sum(values) / values.length;
    }

    public static int min(int[] values) {
        int minValue = values[0];
        for (int value : values) {
            if (value < minValue) {
                minValue = value;
            }
        }
        return minValue;
    }
```

```
    public static int max(int[] values) {
        int maxValue = values[0];
        for (int value : values) {
            if (value > maxValue) {
                maxValue = value;
            }
        }
        return maxValue;
    }
}
```

정리