Call by value와 Call by reference

상당히 기본적인 질문이지만, 헷갈리기 쉬운 주제다.

call by value

> 값에 의한 호출

함수가 호출될 때, 메모리 공간 안에서는 함수를 위한 별도의 임시공간이 생성됨 (종료 시 해당 공간 사라짐)

call by value 호출 방식은 함수 호출 시 전달되는 변수 값을 복사해서 함수 인자로 전달함

이때 복사된 인자는 함수 안에서 지역적으로 사용되기 때문에 local value 속성을 가짐

```
따라서, 함수 안에서 인자 값이 변경되더라도, 외부 변수 값은 변경안됨
```

예시

```
void func(int n) {
    n = 20;
}

void main() {
    int n = 10;
    func(n);
    printf("%d", n);
}
```

> printf로 출력되는 값은 그대로 10이 출력된다.

call by reference

> 참조에 의한 호출

call by reference 호출 방식은 함수 호출 시 인자로 전달되는 변수의 레퍼런스를 전달함

따라서 함수 안에서 인자 값이 변경되면, 아규먼트로 전달된 객체의 값도 변경됨

```
void func(int *n) {
    *n = 20;
}

void main() {
    int n = 10;
    func(&n);
    printf("%d", n);
}
```

> printf로 출력되는 값은 20이 된다.

Java 함수 호출 방식

~~자바의 경우, 함수에 전달되는 인자의 데이터 타입에 따라 함수 호출 방식이 달라짐~~

- ~~primitive type(원시 자료형) : call by value~~
  > ~~int, short, long, float, double, char, boolean~~
- ~~reference type(참조 자료형) : call by reference~~
  > ~~array, Class instance~~

자바의 경우, 항상 **call by value**로 값을 넘긴다.

C/C++와 같이 변수의 주소값 자체를 가져올 방법이 없으며, 이를 넘길 수 있는 방법 또한 있지 않다.

reference type(참조 자료형)을 넘길 시에는 해당 객체의 주소값을 복사하여 이를 가지고 사용한다.

따라서 **원본 객체의 프로퍼티까지는 접근이 가능하나, 원본 객체 자체를 변경할 수는 없다.**

아래의 예제 코드를 봐보자.

```
User a = new User("gyoogle");   // 1

foo(a);

public void foo(User b){        // 2
    b = new User("jongnan");    // 3
}

/*
==========================================

// 1 : a에 User 객체 생성 및 할당(새로 생성된 객체의 주소값을 가지고 있음)
```

```
 a   -----> User Object [name = "gyoogle"]

==========================================

// 2 : b라는 파라미터에 a가 가진 주소값을 복사하여 가짐

 a   -----> User Object [name = "gyoogle"]
              ↑
 b   -----------

==========================================

// 3 : 새로운 객체를 생성하고 새로 생성된 주소값을 b가 가지며 a는 그대로 원본 객체를 가
리킴

 a   -----> User Object [name = "gyoogle"]

 b   -----> User Object [name = "jongnan"]

*/
```

파라미터에 객체/값의 주소값을 복사하여 넘겨주는 방식을 사용하고 있는 Java는 주소값을 넘겨 주소값에 저장되어 있는 값을 사용하는 **call by reference**라고 오해할 수 있다.

이는 C/C++와 Java에서 변수를 할당하는 방식을 보면 알 수 있다.

```
// c/c++

 int a = 10;
 int b = a;

 cout << &a << ", " << &b << endl; // out: 0x7ffeefbff49c, 0x7ffeefbff498

 a = 11;

 cout << &a << endl; // out: 0x7ffeefbff49c

//java

 int a = 10;
 int b = a;

 System.out.println(System.identityHashCode(a));    // out: 1627674070
 System.out.println(System.identityHashCode(b));    // out: 1627674070

 a = 11;

 System.out.println(System.identityHashCode(a));    // out: 1360875712
```

C/C++에서는 생성한 변수마다 새로운 메모리 공간을 할당하고 이에 값을 덮어씌우는 형식으로 값을 할당한다. (* 포인터를 사용한다면, 같은 주소값을 가리킬 수 있도록 할 수 있다.)

Java에서 또한 생성한 변수마다 새로운 메모리 공간을 갖는 것은 마찬가지지만, 그 메모리 공간에 값 자체를 저장하는 것이 아니라 값을 다른 메모리 공간에 할당하고 이 주소값을 저장하는 것이다.

이를 다음과 같이 나타낼 수 있다.

```
  C/C++         |        Java
                |
a -> [ 10 ]     |   a -> [ XXXX ]     [ 10 ] -> XXXX(위치)
b -> [ 10 ]     |   b -> [ XXXX ]
                |
              값 변경
a -> [ 11 ]     |   a -> [ YYYY ]     [ 10 ] -> XXXX(위치)
b -> [ 10 ]     |   b -> [ XXXX ]     [ 11 ] -> YYYY(위치)
```

`b = a;`일 때 a의 값을 b의 값으로 덮어 씌우는 것은 같지만, 실제 값을 저장하는 것과 값의 주소값을 저장하는 것의 차이가 존재한다.

즉, Java에서의 변수는 [할당된 값의 위치]를 [값]으로 가지고 있는 것이다.

C/C++에서는 주소값 자체를 인자로 넘겼을 때 값을 변경하면 새로운 값으로 덮어 쓰여 기존 값이 변경되고, Java에서는 주소값이 덮어 쓰여지므로 원본 값은 전혀 영향이 가지 않는 것이다. (객체의 속성값에 접근하여 변경하는 것은 직접 접근하여 변경하는 것이므로 이를 가리키는 변수들에서 변경이 일어난다.)

```
객체 접근하여 속성값 변경

a : [ XXXX ]  [ Object [prop : ~ ] ] -> XXXX(위치)
b : [ XXXX ]

prop : ~ (이 또한 변수이므로 어딘가에 ~가 저장되어있고 prop는 이의 주소값을 가지고 있는
셈)
prop : [ YYYY ]    [ ~ ] -> YYYY(위치)

a.prop = * (a를 통해 prop를 변경)

prop : [ ZZZZ ]    [ ~ ] -> YYYY(위치)
                   [ * ] -> ZZZZ

b -> Object에 접근 -> prop 접근 -> ZZZZ
```

위와 같은 이유로 Java에서 인자로 넘길 때는 주소값이란 값을 복사하여 넘기는 것이므로 call by value라고 할 수 있다.

출처 : Is Java “pass-by-reference” or “pass-by-value”? - Stack Overflow

정리

Call by value의 경우, 데이터 값을 복사해서 함수로 전달하기 때문에 원본의 데이터가 변경될 가능성이 없다. 하지만 인자를 넘겨줄 때마다 메모리 공간을 할당해야해서 메모리 공간을 더 잡아먹는다.

Call by reference의 경우 메모리 공간 할당 문제는 해결했지만, 원본 값이 변경될 수 있다는 위험이 존재한다.