

# 11. 원시 값과 객체의 비교

“6. 데이터 타입”에서 살펴보았듯이 자바스크립트가 제공하는 7가지 데이터 타입(숫자, 문자열, 불리언, null, undefined, symbol, 객체 타입)은 크게 원시 타입(primitive type)과 객체(object / reference type) 타입으로 구분할 수 있다. 데이터 타입을 원시 타입과 객체 타입으로 구분하는 이유는 무엇일까? 원시 타입과 객체 타입은 근본적으로 다르다는 의미일 것이다. 원시 타입과 객체 타입은 크게 세가지 측면에서 다르다. -> 가

- 원시 타입의 값, 즉 원시 값은 변경 불가능한 값(immutable value)이다. 이에 비해 객체(참조) 타입의 값, 즉 객체는 변경 가능한 값(mutable value)이다.
- 원시 값을 변수에 할당하면 변수에는 실제 값이 저장된다. 이에 비해 객체를 변수에 할당하면 변수에는 참조 값이 저장된다.
- 원시 값을 갖는 변수를 다른 변수에 할당하면 원본의 원시 값이 복사되어 전달된다. 이를 값에 의한 전달(Pass by value)라 한다. 이에 비해 객체를 가리키는 변수를 다른 변수에 할당하면 원본의 참조 값이 복사되어 전달된다. 이를 참조에 의한 전달(Pass by reference)라 한다.

## # 1. 원시 값

### # 1.1. 변경 불가능한 값

원시 타입(primitive type)의 값, 즉 원시 값은 변경 불가능한 값(immutable value)이다. 다시 말해, 한번 생성된 원시 값은 read only한 값이므로 변경할 수 없다.

값을 변경할 수 없는 것이 구체적으로 무엇을 말하는지 살펴보도록 하자. 먼저 변수와 값은 구분해서 생각해야 한다. 변수는 하나의 값을 저장하기 위해 확보한 메모리 공간 자체 또는 그 메모리 공간을 식별하기

위해 붙인 이름이고, **값**은 변수에 저장된 데이터로서 표현식이 평가되어 생성된 결과를 말한다. **변경 불가능하다는 것은 변수가 아니라 값에 대한 진술이다.**

**값을 변경할 수 없다는 것은 재할당을 할 수 없다는 의미와는 다르다. 변수는 언제든지 재할당을 통해 변수 값을 변경(엄밀히 말하자면 교체)할 수 있다. 그렇기 때문에 변수라고 부른다.**

변수의 상대 개념인 상수는 재할당이 금지된 변수를 말한다. 상수도 값을 저장하기 위한 메모리 공간이 필요하므로 변수라고 할 수 있다. 단, 변수는 언제든지 재할당을 통해 변수 값을 변경(교체)할 수 있지만 상수는 단 한번만 할당이 허용되므로 변수 값을 변경(교체)할 수 없다. 따라서 상수와 변경 불가능한 값을 동일시하는 것은 곤란하다.

#### JAVASCRIPT

```
// const 키워드를 사용해 선언한 상수는 재할당이 금지된다.
```

```
const o = {};
```

```
// 하지만 const 키워드를 사용해 선언한 상수에 할당된 객체는 변경할 수 있다.
```

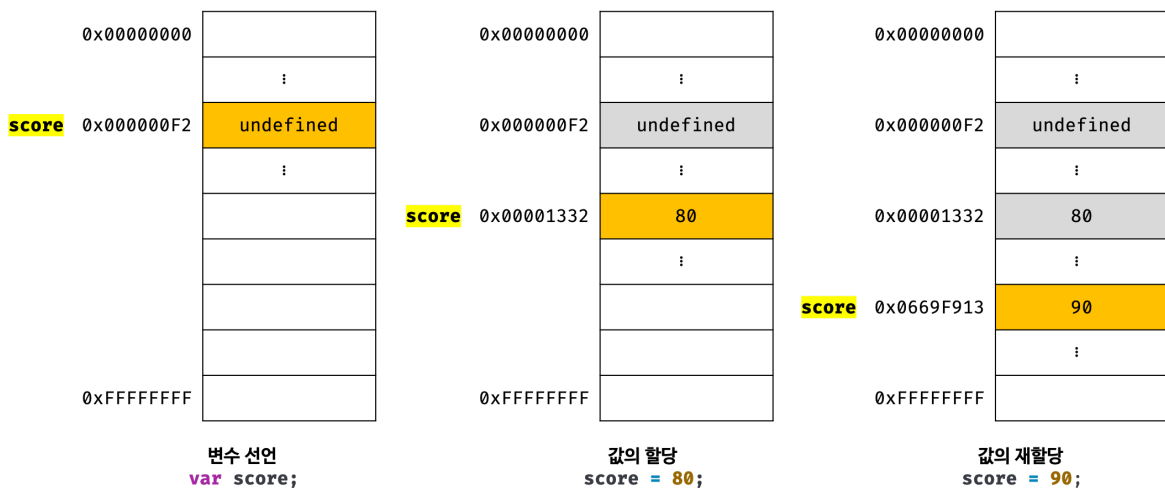
```
// 즉, 상수는 재할당이 금지된 변수일 뿐이다.
```

```
o.a = 1;
```

```
console.log(o); // {a: 1}
```

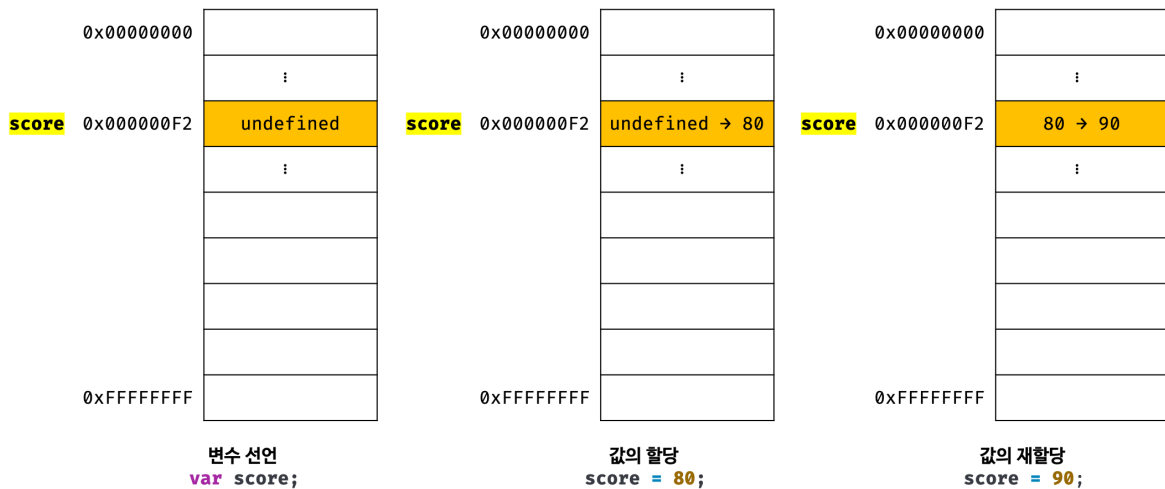
원시 값은 변경 불가능한 값, 즉 read-only한 값이다. 원시 값은 어떤 일이 있어도 불변한다. 이러한 원시 값의 특성은 데이터의 신뢰성을 보장한다.

“4.5. 값의 할당”에서 살펴보았듯이 원시 값을 할당한 변수에 새로운 원시 값을 재할당하면 메모리 공간에 저장되어 있는 재할당 이전의 원시 값을 변경하는 것이 아니라 새로운 메모리 공간을 확보하고 재할당한 원시 값을 저장한 후, 변수는 새롭게 재할당한 원시 값을 가리킨다. 이때 변수가 참조하던 메모리 공간의 주소가 바뀐다.



원시 값은 변경 불가능한 값이다

변수가 참조하던 메모리 공간의 주소가 변경된 이유는 변수에 할당된 원시 값이 변경 불가능한 값이기 때문이다. 만약 원시 값이 변경 가능한 값이라면 변수에 새로운 원시 값을 재할당했을 때 변수가 가리키던 메모리 공간의 주소를 바꿀 필요없이 원시 값 자체를 수정하면 그만이다. 만약 그렇다면 변수가 참조하던 메모리 공간의 주소는 바뀌지 않는다.



원시 값이 변경 가능한 값인 경우

하지만 원시 값은 변경 불가능한 값이기 때문에 값을 직접 변경할 수 없다. 따라서 변수 값을 변경하기 위해 원시 값을 재할당하면 새로운 메모리 공간을 확보하고 재할당한 값을 저장한 후, 변수가 참조하던 메모리 공간의 주소를 변경한다. 원시 값의 이러한 특성을 **불변성(immutability)**이라 한다.

불변성을 갖는 원시 값을 할당한 변수는 재할당 이외에 변수 값을 변경할 수 있는 방법이 없다. 만약 재할당 이외에 원시 값인 변수 값을 변경할 수 있다면 예기치 않게 변수 값이 변경될 수 있다는 것을 의미한다. 이는 값의 변경, 즉 상태 변경을 추적하기 어렵게 만들기 때문에 신뢰성을 떨어뜨린다.

## # 1.2. 문자열과 불변성

“6.9.1. 데이터 타입에 의한 메모리 공간의 확보”에서 살펴보았듯이 원시 값을 저장하려면 먼저 확보해야 하는 메모리 공간의 크기를 결정해야 한다. 이를 위해 원시 타입 별로 메모리 공간의 크기가 미리 정해져 있다고 했다. 단, ECMAScript 사양에 문자열 타입(2byte)과 숫자 타입(8byte) 이외의 원시 타입은 크기를 명확히 규정하고 있지는 않아서 브라우저 제조사의 구현에 따라 원시 타입의 크기는 다를 수 있다.

원시 값인 문자열은 다른 원시 값과 비교할 때 독특한 특징이 있다. 문자열은 0개 이상의 문자(character)들로 이루어진 집합을 말하며 1개의 문자는 2byte의 메모리 공간에 저장된다. 따라서 문자열은 몇개의 문자로 이루어졌는지에 따라 필요한 메모리 공간의 크기가 결정된다. 숫자 값은 1도, 1000000도 동일한 8byte가 필요하지만 문자열의 경우, (실제와는 다르지만 단순하게 계산했을 때) 1개의 문자로 이루어진 문자열은 2byte, 10개의 문자로 이루어진 문자열은 20byte가 필요하다.

## JAVASCRIPT

```
// 문자열은 0개 이상의 문자들로 이루어진 집합이다.
var str1 = '';           // 0개의 문자로 이루어진 문자열(빈 문자열)
var str2 = 'Hello';      // 5개의 문자로 이루어진 문자열
```

이와 같은 이유로 C는 하나의 문자를 위한 데이터 타입(char)만 존재할 뿐 문자열 타입이란 존재하지 않는다. C는 문자열을 문자들의 배열로 처리하고 Java는 문자열을 String 객체로 처리한다.

하지만 자바스크립트는 개발자의 편의를 위해 원시 타입인 문자열 타입을 제공한다. 즉, 자바스크립트의 문자열은 원시 타입이며 변경 불가능하다. 이것은 문자열이 생성된 이후, 변경할 수 없다는 것을 의미한다. 아래의 코드를 살펴보자.

## JAVASCRIPT

```
var str = 'Hello';
str = 'world';
```

첫번째 문이 실행되면 메모리에 문자열 'Hello'가 생성되고 식별자 str은 문자열 'Hello'가 저장된 메모리 공간의 메모리 셀 주소를 가리킨다. 그리고 두번째 문이 실행되면 이전에 생성된 문자열 'Hello'을 수정하는 것이 아니라 새로운 문자열 'world'를 메모리에 생성하고 식별자 str은 이것을 가리킨다. 이때 문자열 'Hello'와 'world'는 모두 메모리에 존재하고 있다. 식별자 str은 문자열 'Hello'를 가리키고 있다가 문자열 'world'를 가리키도록 변경되었을 뿐이다.

문자열의 한 문자를 변경해 보자. 문자열은 유사 배열 객체이므로 배열과 유사하게 각 문자에 접근할 수 있다.

### 유사 배열 객체(Array-like Object)

유사 배열 객체는 마치 배열처럼 인덱스로 프로퍼티 값에 접근할 수 있고 length 프로퍼티를 갖는 객체를 말한다. 문자열은 마치 배열처럼 인덱스를 통해 각 문자에 접근할 수 있으며 length 프로퍼티를 갖기 때문에 유사 배열 객체이고 for 문으로 순회할 수도 있다.

갑자기 원시값인 문자열이 객체일 수도 있다니 혼란스러울 수 있겠다. 아직 살펴보지 않았지만 원시값을 객체처럼 사용하면 원시 값을 감싸는 래퍼 객체로 자동 변환된다. 이에 대해서는 “21.3. 원시값과 래퍼 객체”에서 자세히 살펴볼 것이다.

## JAVASCRIPT


```
var str = 'string';
```

// 문자열은 유사 배열이므로 배열과 유사하게 인덱스를 사용하여 각 문자에 접근할 수 있다.

// 하지만 문자열은 원시 값이므로 변경할 수 없다. 이때 에러가 발생하지 않는다.

```
str[0] = 'S';
```

```
console.log(str); // string
```



`str[0] = 'S'` 처럼 이미 생성된 문자열의 일부 문자를 변경해도 반영되지 않는다. 문자열은 변경 불가능한 값(immutable value)이기 때문이다. 이처럼 한번 생성된 문자열은 read only한 값으로서 변경할 수 없다. 원시 값은 어떤 일이 있어도 불변한다. 따라서 예기치 못한 변경으로부터 자유롭다. 이는 신뢰성을 보장한다.

그러나 변수에 새로운 문자열을 재할당하는 것은 물론 가능하다. 이는 기존 문자열을 변경하는 것이 아니라 새로운 문자열을 새롭게 할당하는 것이기 때문이다.

## # 1.3. 값에 의한 전달

아래의 예제를 살펴보자.

## JAVASCRIPT

```
var score = 80;
```

```
var copy = score;
```

```
console.log(score); // 80
```

```
console.log(copy); // 80
```

```
score = 100;
```

```
console.log(score); // 100
```

```
console.log(copy); // ?
```

변수 `score`에 숫자값 80을 할당했다. 그리고 변수 `copy`에 변수 `score`를 할당했다. 그 후, 변수 `score`에 새로운 숫자값 100을 재할당하면 변수 `copy`의 값은 어떻게 될까?

이 질문의 핵심은 “변수에 변수를 할당했을 때 무엇이 어떻게 전달되는가?”이다. `copy = score` 에서 `score`는 변수값 80으로 평가되므로 변수 `copy`에도 80이 할당될 것이다. 이때 새로운 숫자값 80이 생성되어 변수 `copy`에 할당된다. (이 절의 후반부에 다른 가능성에 대해서도 설명한다.)

이처럼 변수에 원시값을 갖는 변수를 할당하면 할당받는 변수(`copy`)에는 할당되는 변수(`score`)의 원시값이 복사되어 전달된다. 이를 **값에 의한 전달(Pass by value)**라 한다. (이 용어는 오해가 있을 수 있다. 이에 대해서는 이어지는 설명을 참고하기 바란다.) 위 예제의 경우, 변수 `copy`에 원시값을 갖는 변수 `score`를 할당하면 할당받는 변수(`copy`)에는 할당되는 변수(`score`)의 원시값 80이 복사되어 전달된다.

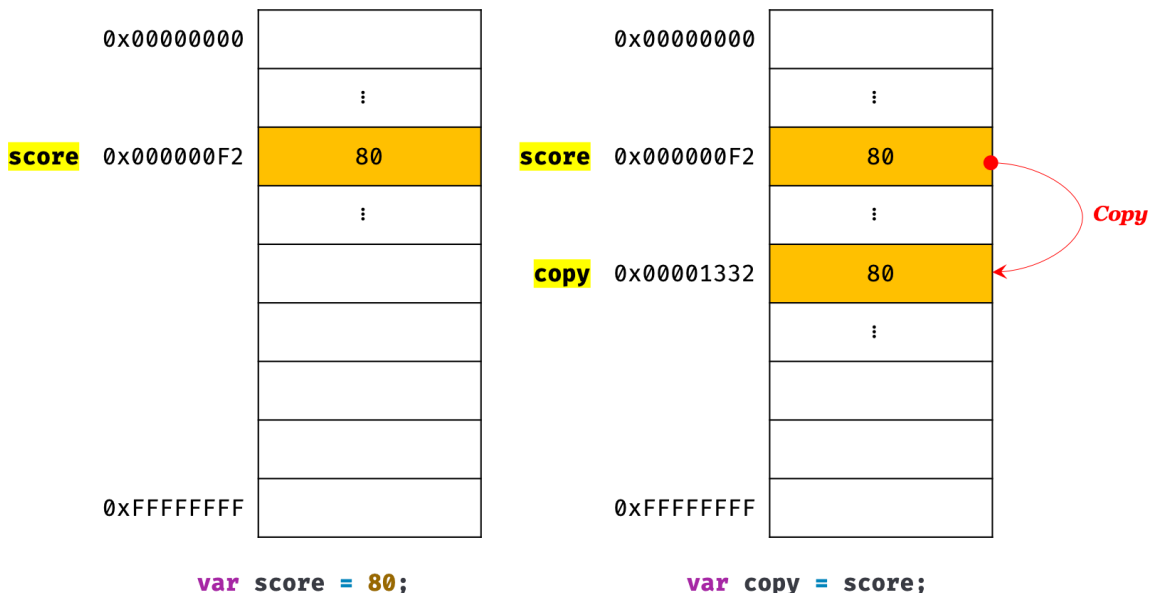
## JAVASCRIPT

```
var score = 80;

// 변수 copy에는 변수 score의 값 80이 복사되어 할당된다.
var copy = score;

console.log(score, copy); // 80 80
console.log(score === copy); // true
```

이때 변수 `score`와 `copy`는 숫자값 80을 갖는다는 점에서는 동일하다. 하지만 변수 `score`와 `copy`의 값 80은 다른 메모리 공간에 저장된 별개의 값이다.



값에 의한 전달(Pass by value)

이제 변수 `score`의 값을 변경해 보자.

## JAVASCRIPT

```

var score = 80;

// 변수 copy에는 변수 score의 값 80이 복사되어 할당된다.
var copy = score;

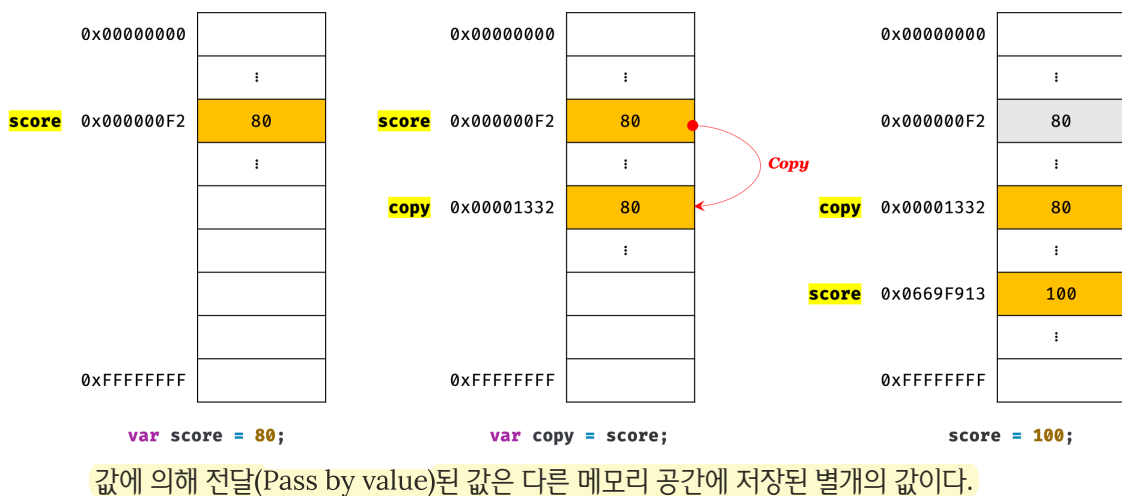
console.log(score, copy);    // 80 80
console.log(score === copy); // true

// 변수 score와 변수 copy의 값은 다른 메모리 공간에 저장된 별개의 값이다.
// 따라서 변수 score의 값을 변경하여도 변수 copy의 값에는 어떠한 영향도 주지 않는다.
score = 100;

console.log(score, copy);    // 100 80
console.log(score === copy); // false

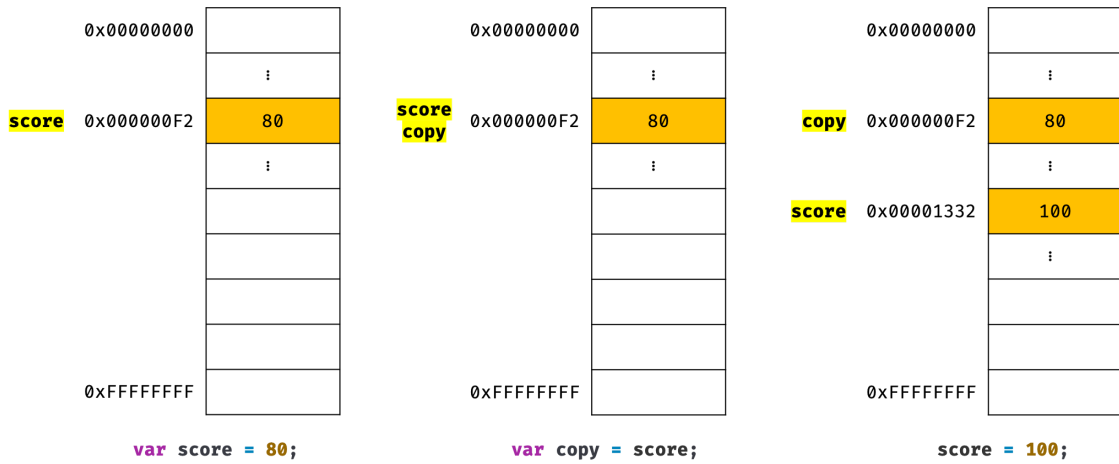
```

변수 score와 copy의 값 80은 다른 메모리 공간에 저장된 별개의 값이라는 것에 주의하기 바란다. 따라서 변수 score의 값을 변경하여도 변수 copy의 값에는 어떠한 영향도 주지 않는다.



사실 위 그림은 실제 자바스크립트 엔진의 내부 동작과 정확히 일치하지 않을 수 있다. ECMAScript 사양에는 변수를 통해 메모리를 어떻게 관리해야 하는지 명확하게 정의되어 있지 않다. 따라서 실제 자바스크립트 엔진을 구현하는 제조사에 따라 실제 내부 동작 방식은 미묘한 차이가 있을 수 있다.

위 그림에서는 변수에 원시값을 갖는 변수를 할당하면 원시값이 복사되는 것으로 표현했다.(MDN의 원시 값에서도 이 방식으로 설명하고 있다.) 하지만 변수에 원시값을 갖는 변수를 할당하는 시점에는 두 변수가 같은 원시값을 참조하다가 어느 한쪽의 변수에 재할당이 이루어졌을 때 비로소 새로운 메모리 공간에 재할당된 값을 저장하도록 동작할 수도 있다. 참고로 파이썬은 이처럼 동작한다.



변수에 원시값을 갖는 변수를 할당하는 시점에는 두 변수가 같은 원시값을 참조하다가 어느 한쪽의 변수에 재할당이 이루어졌을 때 비로소 새로운 메모리 공간에 재할당된 값을 저장하는 경우

또한 “값에 의한 전달”이란 용어도 ECMAScript 사양에는 등장하지 않는다. 이 책에서는 타 언어에서 자주 사용하는 “값에 의한 전달”과 “참조에 의한 전달”이라는 용어를 사용하지만 “공유에 의한 전달(pass by sharing)”이라고 표현하는 경우도 있다.

참고로 “값에 의한 전달”이란 용어는 자바스크립트를 위한 용어가 아니므로 사실 오해가 있을 수도 있다. 엄격하게 표현하면 변수에는 값이 전달되는 것이 아니라 메모리 주소가 전달되기 때문이다. 이는 변수와 같은 식별자는 값이 아니라 메모리 주소를 기억하고 있기 때문이다.

식별자에 대해 다시 한번 생각해보자. “4.2. 식별자”에서 언급한 바와 같이 식별자는 어떤 값을 구별하여 식별해낼 수 있는 고유한 이름이다. 값은 메모리 공간에 저장되어 있다. 따라서 식별자는 메모리 공간에 저장되어 있는 어떤 값을 구별하여 식별해낼 수 있어야 하므로 변수와 같은 식별자는 값이 아니라 메모리 주소를 기억하고 있다.

식별자로 값을 구별하여 식별한다는 것은 식별자가 기억하고 있는 메모리 주소를 통해 메모리 공간에 저장된 값에 접근할 수 있다는 것을 의미한다. 즉, 식별자는 메모리 주소에 붙인 이름이라고 할 수 있다.

#### JAVASCRIPT

```
var x = 10;
```

위 예제의 경우, 할당 연산자는 숫자 리터럴 10에 의해 생성된 숫자값 10이 저장되어 있는 메모리 공간의 주소를 전달한다. 이로서 식별자 x는 메모리 공간에 저장된 숫자값 10을 식별할 수 있다.

#### JAVASCRIPT

```
var copy = score;
```



위 예제의 경우, `score`는 식별자 표현식으로서 숫자값 80으로 평가된다. 이때 2가지 평가 방식이 가능하다.

1. 새로운 80을 생성(복사)하여 메모리 주소를 전달(그림 11-4)하는 방식. 이 방식은 할당 시점에 두 변수가 기억하는 메모리 주소가 다르다.
2. `score`의 변수값 80의 메모리 주소를 그대로 전달(그림 11-5)하는 방식. 이 방식은 할당 시점에 두 변수가 기억하는 메모리 주소가 같다.

이처럼 “값의 의한 전달”도 사실은 값을 전달하는 것이 아니라 메모리 주소를 전달한다. 단, 전달된 메모리 주소를 통해 메모리 공간에 접근하면 값을 참조할 수 있다.

중요한 것은 변수에 원시값을 갖는 변수를 할당하는 경우, 변수 할당 시점이든, 두 변수 중 어느 하나의 변수에 원시값을 재할당하는 시점이든 결국은 두 변수의 원시값은 서로 다른 메모리 공간에 저장된 별개의 값이 되어 어느 한쪽에서 재할당을 통해 값을 변경하더라도 서로 간섭할 수 없다는 것이다.

## # 2. 객체

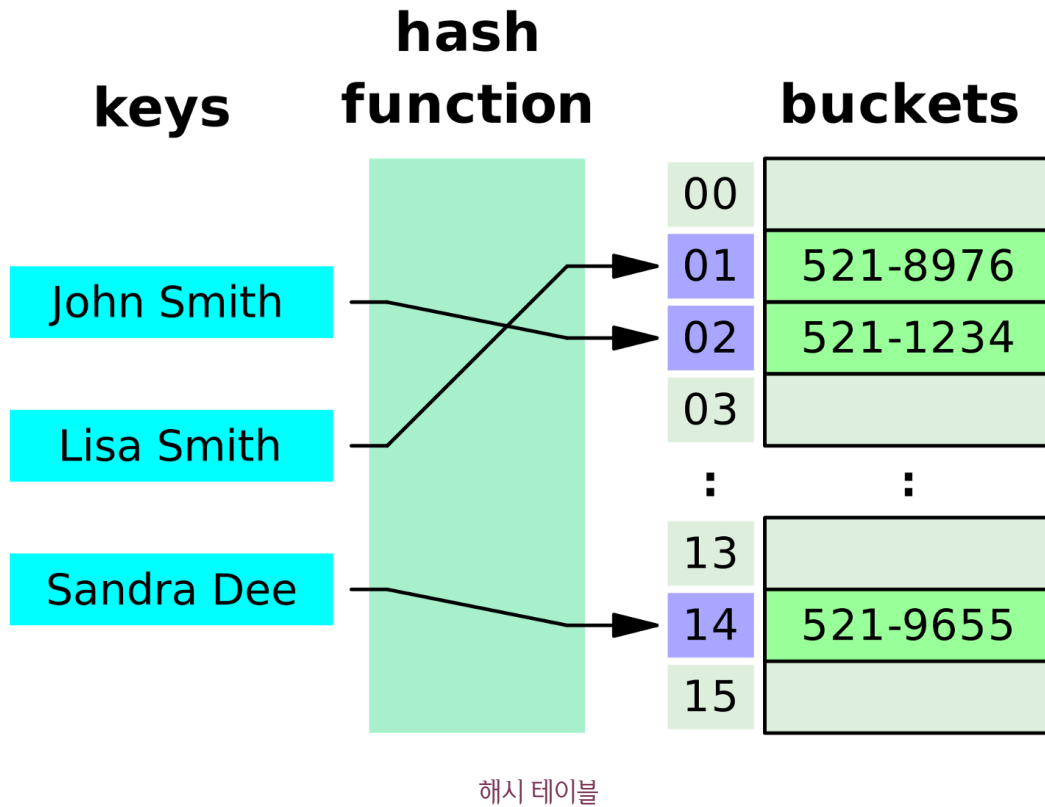
객체는 프로퍼티의 개수가 정해져 있지 않으며 동적으로 추가되고 삭제할 수 있다. 또한 프로퍼티의 값에도 제약이 없다. 따라서 객체는 원시 값과 같이 확보해야 할 메모리 공간의 크기를 사전에 정해 둘 수 없다.

객체는 복합적인 자료 구조이므로 객체를 관리하는 방식이 원시 값과 비교하여 복잡하고 구현 방식도 브라우저 제조사마다 다를 수 있다. 원시 값은 상대적으로 적은 메모리를 소비하지만 객체는 경우에 따라 크기가 매우 클 수도 있다. 객체를 생성하고 프로퍼티에 접근하는 것도 원시값과 비교할 때 비용이 많이 드는 일이다.

따라서 객체는 원시 값과는 다른 방식으로 동작하도록 디자인되어 있다. 원시 값과의 비교를 통해 객체를 이해해 보도록 하자.

### 자바스크립트 객체의 관리 방식

자바스크립트 객체는 프로퍼티 키를 인덱스로 사용하는 **해시 테이블**(hash table, 해시 테이블은 연관 배열, map, dictionary, lookup table이라고 부르기도 한다)이라고 생각할 수 있다. 대부분의 자바스크립트 엔진은 해시 테이블과 유사하지만 보다 높은 성능을 위해 해시 테이블보다 나은 방법으로 객체를 구현한다.



Java, C++과 같은 클래스 기반 객체 지향 프로그래밍 언어는 사전에 정의된 클래스에 기반하여 객체 (인스턴스)를 생성한다. 다시 말해, 객체를 생성하기 이전에 이미 프로퍼티와 메소드가 정해져 있으며 그대로 객체를 생성한다. 객체가 생성된 이후에는 프로퍼티를 삭제하거나 추가할 수 없다. 하지만 자바스크립트는 클래스없이 객체를 생성할 수 있으며 객체가 생성된 이후라도 동적으로 프로퍼티와 메소드를 추가할 수 있다. 이는 이론적으로 클래스 기반 객체 지향 프로그래밍 언어의 객체보다 생성과 프로퍼티 접근에 비용이 더 많이 드는 비효율적인 방식이다.

따라서 V8 자바스크립트 엔진의 경우, 프로퍼티에 접근하기 위해 동적 탐색(dynamic lookup) 대신 **히든 클래스**(Hidden class, 아래 아티클 참고)라는 방식을 사용해 C++ 객체의 프로퍼티에 접근하는 정도의 성능을 보장한다. 히든 클래스는 Java와 같이 고정된 객체 레이아웃(클래스)과 유사하게 동작한다. 크롬 V8 자바스크립트 엔진이 객체를 어떻게 관리하는지에 대해 관심이 있다면 아래의 아티클을 참고하기 바란다.

- Fast properties in V8
- V8의 히든 클래스 이야기
- 자바스크립트 엔진의 최적화 기법 (2) - Hidden class, Inline Caching
- How the V8 engine works?
- How JavaScript works: inside the V8 engine + 5 tips on how to write optimized code
- Breaking the JavaScript Speed Limit with V8

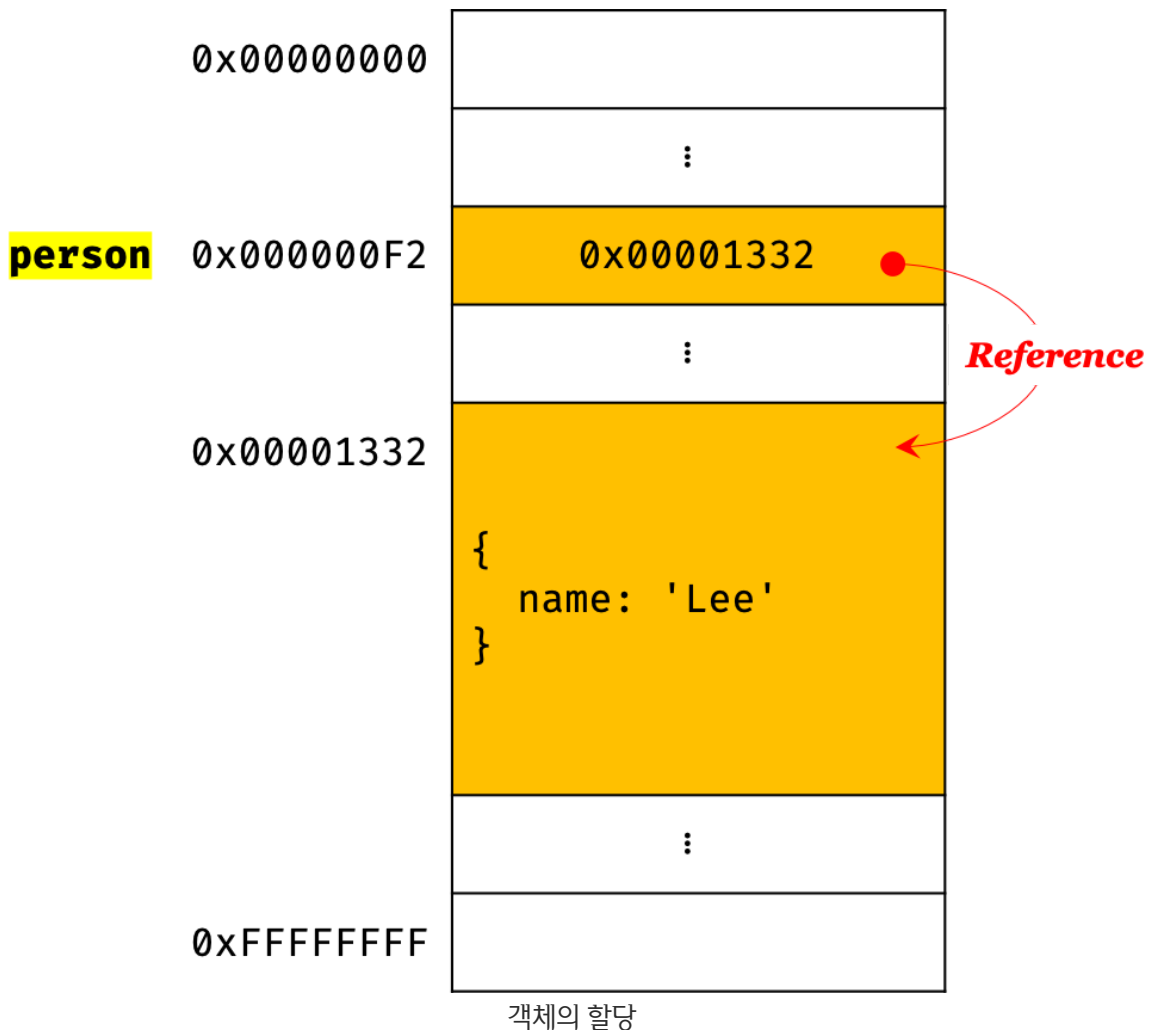
## # 2.1. 변경 가능한 값

객체(참조) 타입의 값, 즉 객체는 변경 가능한 값(`mutable value`)이다. 먼저 변수에 객체를 할당하면 어떤 일이 일어나는 지부터 살펴보자.

JAVASCRIPT

```
var person = {
  name: 'Lee'
};
```

원시 값을 할당한 변수가 기억하는 메모리 주소를 통해 메모리 공간에 접근하면 원시 값에 접근할 수 있다. 즉, 원시 값을 할당한 변수는 원시 값 자체를 값으로 갖는다. 하지만 객체를 할당한 변수가 기억하는 메모리 주소를 통해 메모리 공간에 접근하면 참조 값(`Reference value`)에 접근할 수 있다. 참조 값은 생성된 객체가 저장된 메모리 공간의 주소, 그 자체이다. , =



위 그림을 보면 객체를 할당한 변수에는 생성된 객체가 실제로 저장된 메모리 공간의 주소가 저장되어 있다. 이 값을 **참조 값**이라고 한다. 변수는 이 참조 값을 통해 객체에 접근할 수 있다.

원시 값을 할당한 변수를 참조하면 메모리에 저장되어 있는 원시 값에 접근한다. 하지만 객체를 할당한 변수를 참조하면 메모리에 저장되어 있는 참조 값을 통해 실제 객체에 접근한다.

## JAVASCRIPT

```
// 할당이 이루어지는 시점에 객체 리터럴이 해석되고 그 결과 객체가 생성된다.
```

```
var person = {
  name: 'Lee'
};
```

```
// person 변수에 저장되어 있는 참조값으로 실제 객체에 접근하여 그 객체를 반환한다.
```

```
console.log(person); // {name: "Lee"}
```

일반적으로 원시 값을 할당한 변수의 경우, “변수는 ○값을 갖는다.” 또는 “변수의 값은 ○이다.”라고 표현한다. 하지만 객체를 할당한 변수의 경우, “변수는 객체를 참조하고 있다” 또는 “변수는 객체를 가리키고 (point) 있다”라고 표현한다. 위 예제에서 변수 person은 객체 `{ name: 'Lee' }`를 가리키고(참조하고) 있다.

원시 값은 변경 불가능한 값이므로 원시 값을 갖는 변수의 값을 변경하려면 재할당 이외에는 다른 방법이 없다. 하지만 객체는 변경 가능한 값이다. 따라서 객체를 할당한 변수는 **재할당없이** 객체를 직접 변경할 수 있다. 즉, 재할당없이 프로퍼티를 동적으로 추가할 수도 있고 프로퍼티 값을 갱신할 수도 있으며 프로퍼티 자체를 삭제할 수도 있다.

## JAVASCRIPT

```
var person = {
  name: 'Lee'
};
```

```
// 프로퍼티 값 갱신
```

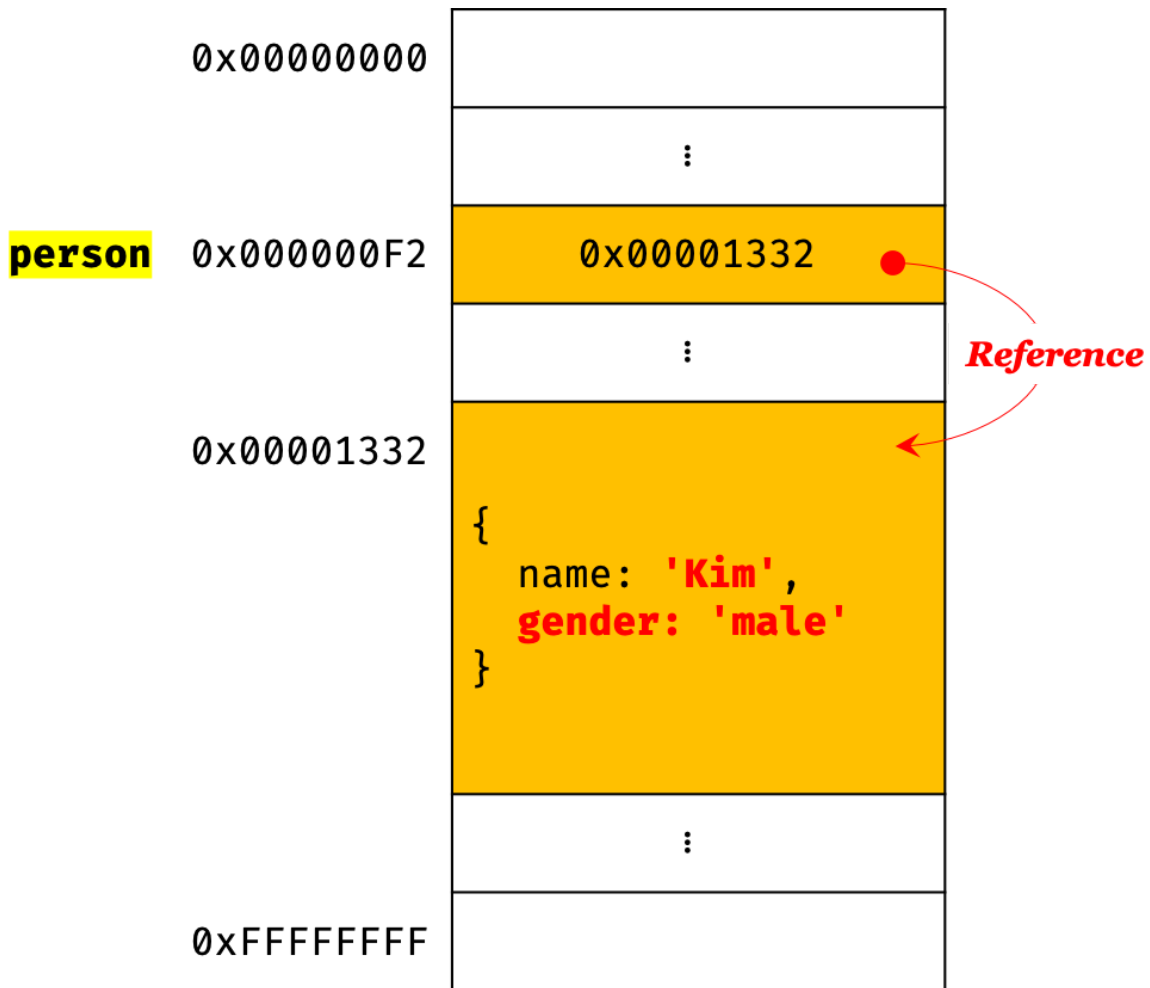
```
person.name = 'Kim';
```

```
// 프로퍼티 동적 생성
```

```
person.address = 'Seoul';
```

```
console.log(person); // {name: "Kim", address: "Seoul"}
```

원시 값은 변경 불가능한 값이므로 원시 값을 갖는 변수의 값을 변경하려면 재할당을 통해 메모리에 원시 값을 새롭게 생성해야 한다. 하지만 객체는 변경 가능한 값이므로 메모리에 저장된 객체를 직접 수정할 수 있다. 이때 객체를 할당한 변수에 재할당을 하지 않았으므로 객체를 할당한 변수의 참조 값은 변경되지 않는다.



객체는 변경 가능한 값이다.

앞에서 언급했듯이 객체를 생성하고 관리하는 방식은 매우 복잡하며 비용이 많이 드는 일이다. 객체를 변경할 때 마다 원시 값처럼 이전 값을 복사하여 새롭게 생성한다면 명확하고 신뢰성이 확보되겠지만 객체는 크기가 매우 클 수도 있고, 원시 값처럼 크기가 일하지도 않으며, 프로퍼티 값이 객체일 수도 있어서 복사(Deep copy)하고 생성하는 비용이 많이 든다. 다시 말해, 메모리의 효율적 소비가 어렵고 퍼포먼스가 나빠진다.

#### 얕은 복사(shallow copy)와 깊은 복사(deep copy)

객체를 프로퍼티 값으로 갖는 객체의 경우, 얕은 복사는 한 단계까지만 복사하는 것을 말하고 깊은 복사는 객체에 중첩되어 있는 객체까지 모두 복사하는 것을 말한다.

얕은 복사와 깊은 복사로 생성된 객체는 원본과는 다른 객체이다. 즉, 원본과 복사본은 참조값이 다른 별개의 객체이다. 하지만 얕은 복사는 객체에 중첩되어 있는 객체의 경우, 참조값을 복사하고 깊은 복사는 객체에 중첩되어 있는 객체까지 모두 복사하여 원시 값처럼 완전한 복사본을 만든다는 차이가 있다.

## JAVASCRIPT

```

const o = {
  a: {
    b: 2
  },
  f() {}
};

// 얕은 복사
let c = { ... o };
console.log(o.a === c.a); // true

// 얕은 복사
c = Object.assign({}, o);
console.log(o.a === c.a); // true

// JSON.parse와 JSON.stringify를 사용한 깊은 복사
c = JSON.parse(JSON.stringify(o));
console.log(o.a === c.a); // false
// 메소드가 사라진다!
console.log(c.f); // undefined

// lodash의 cloneDeep를 사용한 깊은 복사
const _ = require('lodash'); // npm i lodash

c = _.cloneDeep(o);
console.log(o.a === c.a); // false
console.log(c.f); // f

```

따라서 메모리를 효율적으로 사용하기 위해 그리고 객체의 복사하고 생성하는 비용을 절약하여 퍼포먼스를 향상시키기 위해 객체는 변경 가능한 값으로 디자인되어 있다. 메모리 사용의 효율성과 퍼포먼스를 위해 어느 정도의 구조적인 단점을 감당한 디자인이라고 할 수 있다.

객체는 이러한 구조적 단점에 따른 부작용(Side effect)이 있다. 그것은 원시 값과는 다르게 여러 개의 식별자가 하나의 객체를 공유할 수 있다는 것이다.

## # 2.2. 참조에 의한 전달

여러 개의 식별자가 하나의 객체를 공유할 수 있다는 것이 무엇을 의미하는지, 이로 인해 어떤 부작용이 발생하는지 확인해 보자.

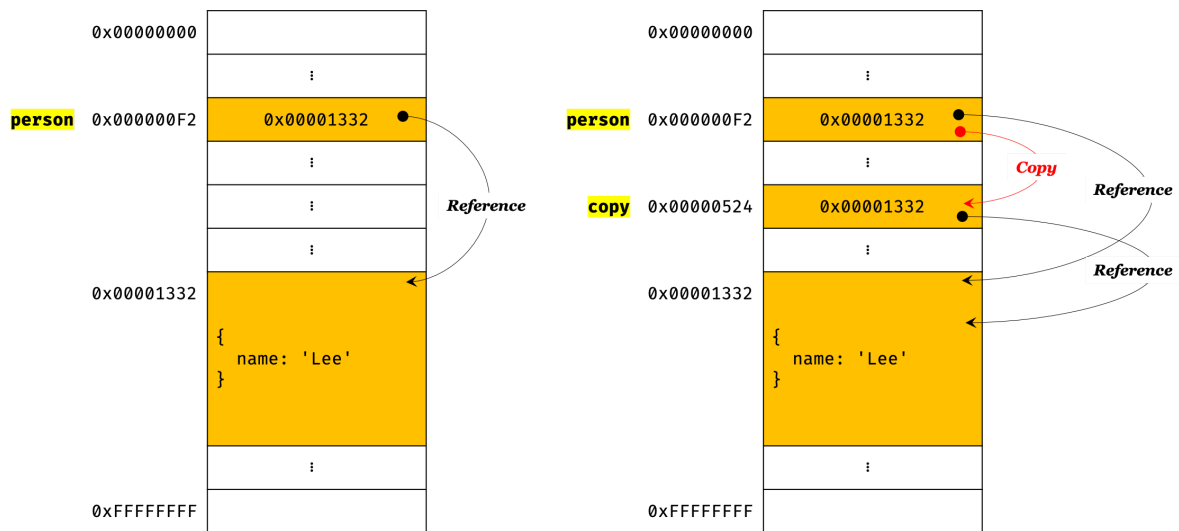
## JAVASCRIPT

```
var person = {
  name: 'Lee'
};

// 참조 값을 복사
var copy = person;
```

가

객체를 가리키는 변수(원본, person)를 다른 변수(사본, copy)에 할당하면 원본의 참조 값이 복사되어 전달된다. 이를 **참조에 의한 전달(Pass by reference)**라 한다. (이 용어는 오해가 있을 수 있다. 이에 대해서는 이어지는 설명을 참고하기 바란다.)



참조에 의한 전달(Pass by reference)

위 그림처럼 원본 person를 사본 copy에 할당하면 원본 person의 참조 값을 복사하여 copy에 저장한다. 이때 원본 person와 사본 copy는 메모리 주소는 다르지만 동일한 참조 값을 갖는다. 다시 말해, 원본 person와 사본 copy 모두 동일한 객체를 가리키고 있다. 이것은 두개의 식별자가 하나의 객체를 공유한다는 것을 의미한다. 따라서 원본 또는 사본 어떤 한쪽에서 객체를 변경(변수에 새로운 객체를 재할당하는 것이 아니라 객체의 프로퍼티 값 변경 또는 추가, 삭제)이 하면 서로 영향을 주고 받는다.

## JAVASCRIPT

```
var person = {
  name: 'Lee'
};
```

```
// 참조 값을 복사. copy와 person은 동일한 참조 값을 갖는다.
var copy = person;

// copy와 person은 동일한 객체를 참조한다.
console.log(copy === person); // true

// copy를 통해 객체를 변경한다.
copy.name = 'Kim';

// person을 통해 객체를 변경한다.
person.address = 'Seoul';

// copy와 person은 동일한 객체를 가리키고 있다.
// 따라서 어느 한쪽에서 객체를 변경하면 서로 영향을 주고 받는다.
console.log(person); // {name: "Kim", address: "Seoul"}
console.log(copy);    // {name: "Kim", address: "Seoul"}
```

결국 “값에 의한 전달”과 “참조에 의한 전달”은 식별자가 기억하는 메모리 공간에 저장되어 있는 값을 복사하여 전달한다는 면에서 동일하다. 다만 식별자가 기억하는 메모리 공간, 즉 변수에 저장되어 있는 값이 원시 값인지 참조 값인지의 차이만 있을 뿐이다. 따라서 자바스크립트에는 “참조에 의한 전달”은 존재하지 않고 “값에 의한 전달”만이 존재한다고 말할 수 있다.

앞에서 언급했듯이 자바스크립트의 이같은 동작 방식을 설명하는 정확한 용어가 존재하지 않는다. 이런 이유로 “값에 의한 전달”이나 “참조에 의한 전달”이라는 용어를 사용하지 않고 “공유에 의한 전달(pass by sharing)”이라고 표현하는 경우도 있다. 하지만 이 용어 또한 ECMAScript 사양에 정의된 자바스크립트의 공식적인 용어는 아니며 자바스크립트의 동작 방식을 정확히 설명하지 못한다. 따라서 이 책에서는 전달되는 값의 종류에 원시값인지 참조값인지를 구별하여 강조하는 의미에서 “값에 의한 전달”과 “참조에 의한 전달”로 구분하여 부르기로 한다. 다만 자바스크립트에는 포인터(pointer)가 존재하지 않기 때문에 포인터가 존재하는 다른 프로그래밍 언어의 “참조에 의한 전달”과 의미가 정확히 일치하지 않다는 것에 주의하기 바란다.

c

```
#include <stdio.h>

void main(void) {
    int x = 1;
    int *a = &x; // 정수 x의 참조를 전달
```



```
*a += 1; // 정수값이 변경된다.  
printf("%d\n", x); // 2  
}
```

마지막으로 퀴즈를 풀어보고 마치도록 하자. 아래 예제를 살펴보고 결과를 예측해 보자.



```
var person1 = {  
  name: 'Lee'  
};
```

```
var person2 = {  
  name: 'Lee'  
};
```

```
console.log(person1 === person2); // ①      ->    ->false
```

```
console.log(person1.name === person2.name); // ② true
```