11장 원시 값과 객체의 비교

개요

`js` 에서는 7가지 데이터 타입을 제공한다. 이들은 크게 2가지로 타입으로 구분할 수 있다.

1. **원시 타입(`primitive type`)**
   - `Number` , `string` , `boolean` , `null` , `undefined` , `symbol`
   - 변경 불가능한 값
   - 변수에 실제 값이 저장됨.
   - 값에 의한 전달 방식 (`pass by value`)
2. **객체 타입(`object/reference type`)**
   - `Object`
   - 변경 가능한 값
   - 변수에 참조값이 저장됨.
   - 참조에 의한 전달 (`pass by reference`)

변수와 상수

변수

메모리 공간을 확보하고 값을 저장할 수 있다.

언제든지 재할당을 통해 값을 변경(교체)할 수 있다.

`var` , `let`

상수

변수와 동일하게 메모리 공간을 확보하고 값을 저장할 수 있지만 재할당을 통해 값을 변경할 수 없는 변수

> 💡 단, **상수에 객체의 참조값을 저장하면 객체를 직접 바꿀 수는 없어도 객체의 내부 속성값들을 변경할 수는 있다.**

원시 값 (`primitive value`)

11.1.1 변경 불가능한 값

- 원시 값이 변경이 불가능하다는 것이지 변수에 할당된 원시 값이 변경 불가능하다는 것이 아니다.

```
// 원시 값 에
let score = 80;

// 변수에 할당 된 원시값을 교체할 수 있다.
score = 90;
```

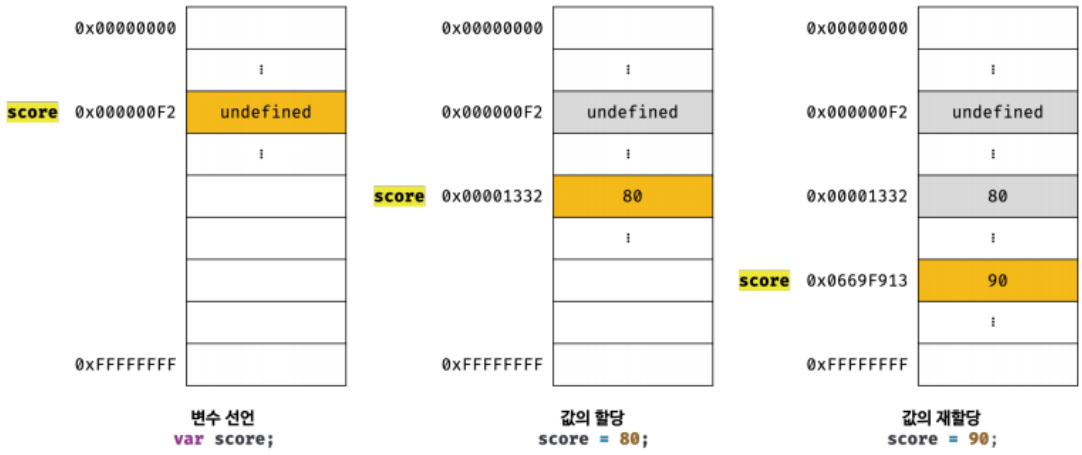

그림 11-1 원시 값은 변경 불가능한 값이다

- 위 코드의 원리를 그림으로 표현하면 그림 11-1과 같다.

1. `js` 엔진이 코드를 실행하기 전, 평가 단계에서 `score` 변수를 식별자로 등록하고 `undefined` 로 초기화한다.
2. 이후 런타임에 `let score = 80` 의 코드문을 만나면 원시값 `80` 을 새로운 메모리 공간에 할당하고 `score` 가 해당 주소를 가리키게 한다.
3. `score = 90` 문을 실행하면 기존의 `80` 이 있던 메모리 주소의 값을 `90` 으로 변경하는 것이 아니라 **새로운 메모리 공간에 `90` 의 값을 할당하고 `score` 가 해당 주소를 가리키게 한다**.

- 이렇게 하는 이유는 `80` 과 `90` 이 바꿀 수 없는 원시 값이기 때문이다.
  - 이러한 원시 값의 특징을 **불변성(immutablilty)**이라 한다.
  - 원시 값은 불변성을 통해 데이터의 신뢰성을 보장한다.

11.1.2 문자열과 불변성

원시 값을 저장하기 위해서는 먼저 메모리에 확보해야 하는 공간의 크기를 결정하는 과정을 거쳐야 한다.

`js` 에서 **숫자**는 `8 bytes` 의 부동소수점 방식을 지원하고 **문자열**은 문자 하나 당 `2 bytes` 의 크기를 필요로 한다.

> 💡 참고로 `js` 에서 숫자는 부동소수점 방식을 사용하기 때문에 정수끼리 나눗셈을 할 경우 소수점이 포함된 결과가 출력된다.
> ex. console.log(8 / 3); // 2.666666…

또한 `js` 에서는 **문자열은 변경 불가능한 원시 값이기 때문에 한 번 문자열이 생성되면 해당 값을 일부만 변경하는 것이 불가능**하다.

```
// 문자열 변경 예
let str = "Test_01";

str[0] = "t";
str[6] = "3";

console.log(str);
// 예상 값 : "test_03"
// 실제 결과 : "Test_01"
```

위의 예제를 보면 알 수 있듯이, 다른 언어처럼 한 번 할당된 문자열의 일부만 접근해서 값을 변경하는 것이 불가능하다.

따라서 `js` 에서 문자열은 원시값이기 때문에 새로운 문자열(원시값)을 재할당하는 과정을 통해 값을 변경해야 한다.

```
let str = "Test_01";
str = "edited_01";

console.log(str);      // "edited_01"
```

11.1.3 값에 의한 전달

원시값이 할당된 변수를 복사할 때, 값이 직접 전달되는 것이 아닌 값에 의한 전달이 일어난다. 예제를 통해 확인해보면

```
let origin = 80;
let copy = origin;
```

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

origin = 100;

console.log(origin, copy);   // 100, 80;
```

- 위의 예제를 보면 알 수 있듯이 `copy` 는 `origin` 의 초기값 `100` 을 복사했다.
- 하지만 `origin` 의 값을 `-1` 로 변경시켜도 `copy` 의 값은 `100` 을 유지하고 있다.
- 여기서 알 수 있는 것은 2가지 이다.
  1. **원시값은 값을 복사해서 전달**한다.
  2. **복사한 값이 있는 메모리 주소는 서로 독립적으로 분리**되어 있다.
     a. 즉, `origin` 과 `copy` 가 가리키는 주소가 다르다는 것

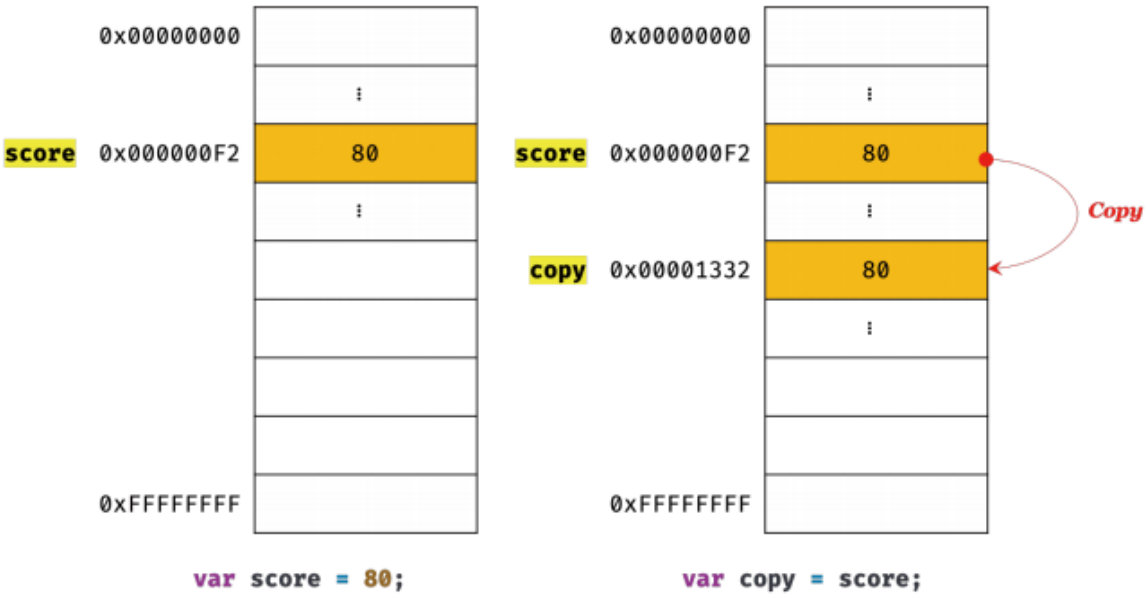

그림 11-3 값에 의한 전달

그림으로 보면 `score === origin` 이고 `copy` 는 동일한 이름을 가졌을 때,

```
// origin === score
let copy = origin;
```

위 코드는 그림 11-3 처럼 값에 의한 전달 과정을 거친다는 것을 알 수 있다.

- **값을 복사해서 독립적인 메모리 공간을 가지는 것**을 알 수 있다.

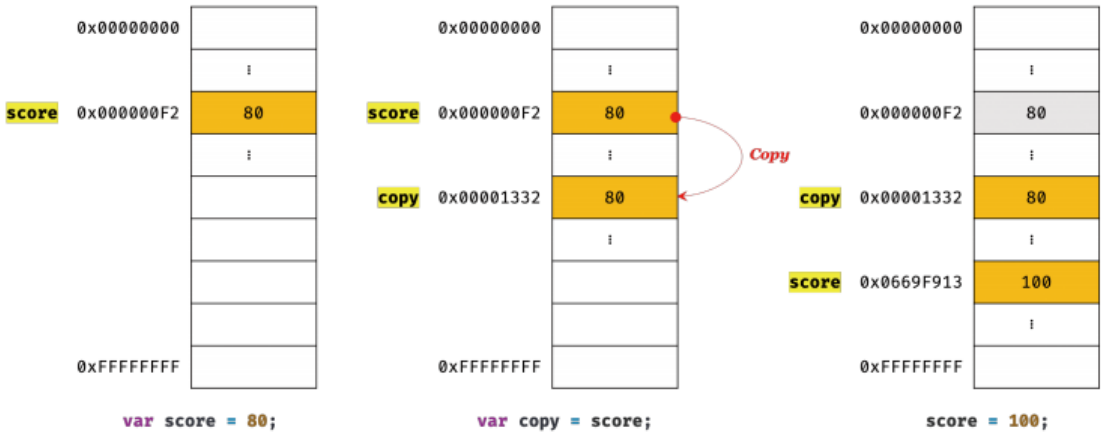

그림 11-4 값에 의해 전달된 값은 다른 메모리 공간에 저장된 별개의 값이다.

```
// origin === score
origin = 100;
```

위 그림 11-4는 위에서 설명한 코드의 과정을 보여준다.

- `80` 이 원시값이기 때문에 `100` 을 위한 새로운 메모리 공간이 할당되고 `origin` 과 `copy` 가 서로 다른 메모리 공간을 가리키기에 `origin` 의 값만 `100` 으로 변경되었다.

값에 의한 전달 방식의 결론

값에 의한 전달 방식은 엄격히 말하자면 값에 의한 전달보다는 **메모리 주소를 전달**한다고 해야 한다.

- 위 그림에서 `score(===origin)` 의 값을 `copy` 에 복사할 때, 실제 값 `80` 을 전달하는 것이 아닌 `score` 의 주소 `0x00000F2` 를 전달하여 해당 주소에 저장된 `80` 의 값을 참조하는 것이기 때문이다.

객체

개요

프로퍼티의 수가 정해져 있지 않고 동적으로 프로퍼티를 추가/삭제 할 수 있다. 그래서 원시 값처럼 메모리에 미리 확보해야 할 공간의 크기를 결정할 수 없다.

- 객체는 복합적인 자료구조이며 원시 값보다 접근 및 다루는데 더 많은 비용이 소모된다.

11.2.1 변경 가능한 값

객체는 참조 타입의 값으로 **변경이 가능한 값**이다.

또한 원시값과 달리 객체의 식별자가 가리키는 메모리 주소에는 해당 객체의 참조 값이 저장되어 있고 이 값을 통해 객체에 접근할 수 있다.

```
const person = {
  name: 'Lee',
}

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

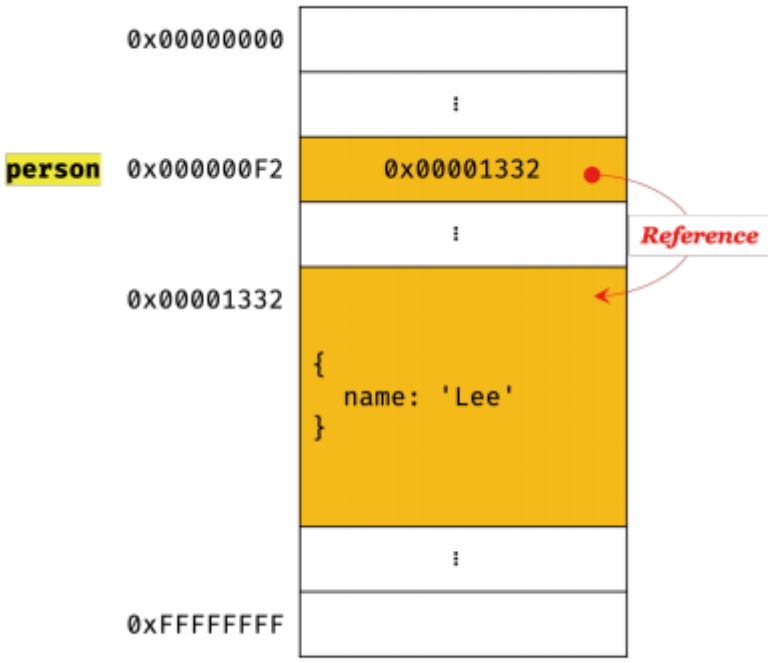

그림 11-7 객체의 할당

위 코드에 대한 구현 과정이 그림 11-7이다.

- `person` 식별자가 가리키는 주소에는 객체가 아닌 객체의 주소(`0x00001332`)가 존재하고 이 주소를 참조하면 `person` 객체의 값에 접근할 수 있다.

객체는 변경 가능한 값이라고 불리는 이유는 재할당 없이 객체의 프로퍼티를 동적으로 추가/삭제가 가능하기 때문이다.

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

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

앞서 사용한 `person` 객체에 위 코드들을 실행하면, `person` 객체에 `name` 속성의 값이 변했고 또한 `address` 속성이 추가된 것을 확인할 수 있다.

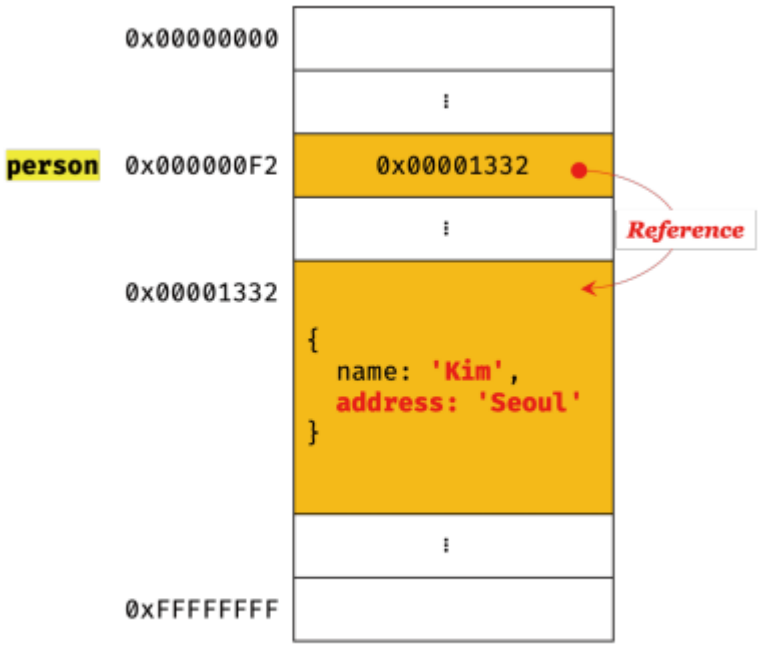

그림 11-8 객체는 변경 가능한 값이다.

이때, 객체를 할당한 변수(식별자)의 참조 값은 변하지 않는다는 것을 기억해야 한다.
실제 객체 내부의 값이 변경된 것이지 객체를 가리키는 참조 값(주소)는 변하지 않는다.

객체의 동작 원리

객체는 원시값과 달리 변경 가능한 값이기에 신뢰성이 떨어지고 관리하는데 비용이 많이 들지만 메모리를 효율적으로 사용하기 위해 변경가능한 값으로 설정한 구조적인 단점을 감안한 설계이다.

이러한 원리 때문에 **2가지 단점**이 존재한다.

1. **여러 개의 식별자가 하나의 객체를 공유**할 수 있다.
2. **얕은 복사가 발생**한다.
    - **얕은 복사 (`Shallow Copy`)**는 복사 단계가 한 단계 까지만 복사 되는 것을 의미.
    - **깊은 복사(`Deep Copy`)**는 객체에 중첩되어 있는 모든 것을 복사하는 것.

```
const origin = {
  shallow_value : 'origin';

  inner: {
    value : 'origin',
  }
}

// 얕은 복사 발생.
const copy = origin;

console.log(origin === copy);    // false , 둘은 다른 객체이다.
console.log(origin.inner === copy.inner);  // true, 내부 객체는 동일한다.

copy.inner.value = 'copy';    // copy 객체의 inner 값을 바꾸면

console.log(origin.inner.value);   // 'copy' , 값이 바뀐 것을 확인할 수 있다.
```

- 위에서 보는 것처럼 **얕은 복사는 겉 껍질인 객체는 서로 다르게 존재하지만 내부 객체들은 공유**하는 것을 볼 수 있다.
    - 이러면 결국 첫 번째 문제였던 **여러 개의 식별자가 하나의 객체를 공유하는 문제가 발생하여 원치 않은 값의 변경이 발생**할 수 있다.

11.2.2 참조에 의한 전달

여러 개의 식별자가 하나의 객체를 공유할 수 있다는 것은 아래 그림 11-9와 같은 상황을 의미한다.

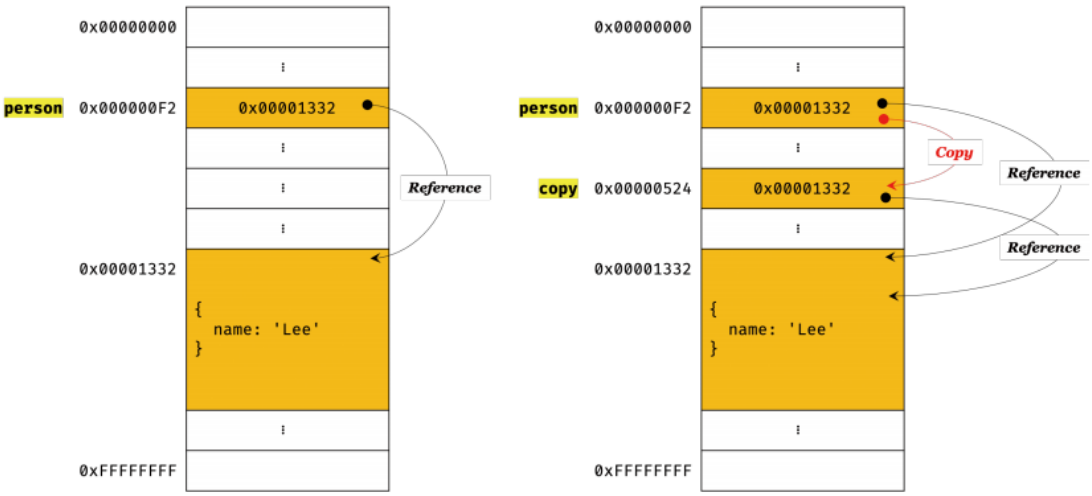

그림 11-9 참조에 의한 전달

```
const person = { name: 'Lee' }
```

```
// person과 copy는 동일한 참조 값을 갖는다.
const copy = person;
```

- 이 과정에서 `person` 식별자는 실제 `person` 객체의 주소인 `0x00001332` 를 참조값으로 갖는다.
- `copy` 에 `person` 을 할당할 때, 원시 값과 달리 `person` 객체의 주소인 `0x00001332` 의 값을 복사해서 저장한 것을 볼 수 있다.
- 즉, **두 식별자가 동일한 참조 값을 갖고 하나의 객체를 공유한 것**이다.
    - 이 방식은 결국 하나의 식별자에서 값을 변경하는 다른 식별자에서도 그 결과가 반영된 객체를 사용해야 한다는 것을 의미한다.

```
copy.name = "change_value";

console.log(copy.name);     // "change_value"
console.log(person.name);   // "change_value"
```

- `person` 식별자로 접근해도 값이 바뀐 문제가 생긴 것을 확인할 수 있다.

결론

값에 의한 전달과 참조에 의한 전달은 모두 **메모리 공간에 저장되어 있는 값을 복사해서 전달하는 면에서 동일**하다.

- **값에 의한 전달** 방식에서는 메모리 주소에 저장된 **원시값을 복사**해서 새로운 메모리 주소에 할당.
- **참조에 의한 전달**도 메모리 주소에 저장된 **참조값을 복사**해서 새로운 메모리 주소에 할당.

하지만 이 값이 원시값이냐 참조값이냐의 차이만 존재할 뿐이다.

> 💡 **결국 `js` 에서는 값(원시값 or 참조값)에 의한 전달만 존재한다고 할 수 있다.**