모던 자바스크립트
Deep Dive

변수 호이스팅

변수 호이스팅

중요한 것은 값의 할당과 선언은 분리된다는 점
자바스크립트 엔진은 소스 코드를 두가지 과정으로 나누어서 실행

(1) 평가 (2) 실행(**런타임 과정**)

```
var result = 10 + 20;
```

평가(선언)

실행(할당)

변수 호이스팅

console.log(score); // 에러가 아닌 undefined가 나오는 현상을 **변수 호이스팅** 이라고 함
var score = 100;

그렇다면 Hoist는 무슨 뜻일까?
소형의 감아올리는 장치. 공장 내의 재료 운반·조립에 사용하며, 들보 위에서 이동시킴.

Device의 개념에서 Hoist는 3가지로 구분됨.
(1) Wire rope or Chain hoist(원치, 크레인용)
(2) Construction Hoists(건축용)
(3) Mine hoists(광산 채굴용)

변수 호이스팅

Hoist의 정의
변수 선언문이 코드의 선두로 끌어 올려진 것처럼 동작하는 자바스크립트 고유의 특징

변수 선언뿐만 아니라 **var, let, const, function, function*, class** 키워드를 사용해서 선언하는 모든 식별자 해당

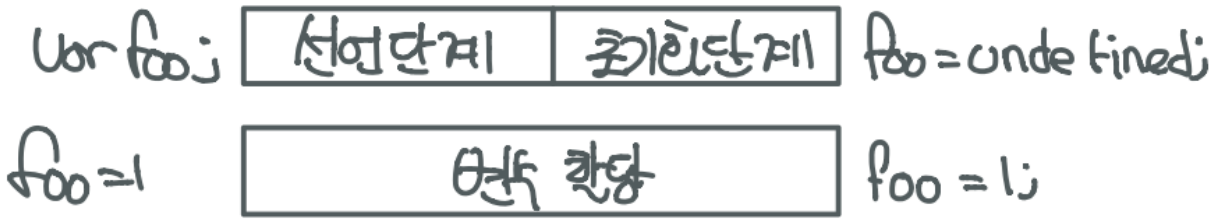
변수 선언은 두 단계로 나뉨
(1) 선언 단계 변수 이름을 실행 컨텍스트의 전역 환경 레코드(혹은 함수 환경 레코드)에 등록(**위 키워드 모두 해당**)
(2) 초기화 단계 메모리 공간 확보 및 연결, 암묵적인 undefined 할당

※ 변수는 **선언(declaration)**하고, 함수는 **정의(definition)**한다.
함수 선언문이 평가되면, 식별자가 암묵적으로 생성되고 함수 객체가 할당됨

var 키워드와 let 키워드의 호이스팅 차이

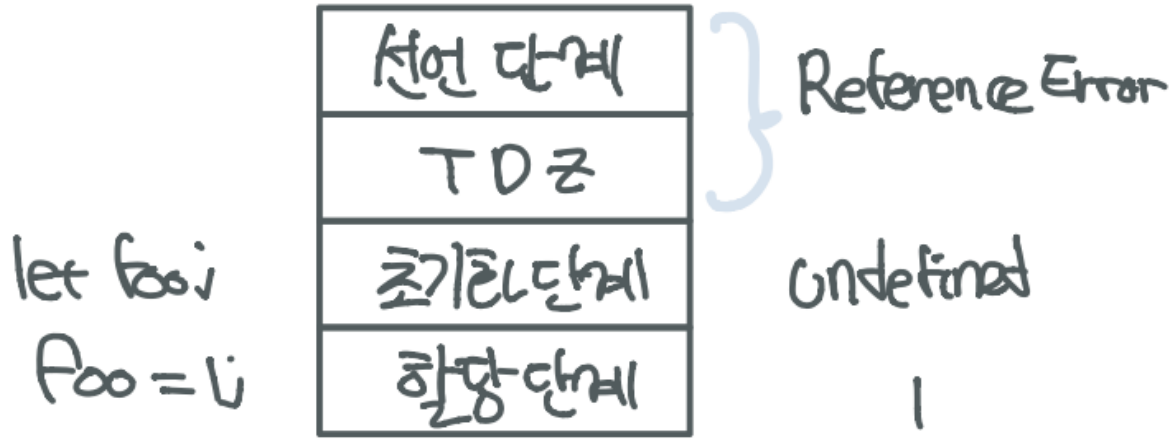
var 키워드의 변수 선언 단계
(1) 선언 단계
(2) 초기화 단계
선언 단계 초기화 단계 동시에 발생

```
console.log(foo); // undefined
var foo;
console.log(foo); // undefined
foo = 1;
console.log(foo); // 1
```



let 키워드의 변수 선언 단계
(1) 선언 단계
(2) TDZ(Temporal Dead Zone)
(3) 초기화 단계
선언 단계 초기화 단계 나뉘어서 발생

```
console.log(foo); // ReferenceError : foo is not defined
let foo;
console.log(foo); // undefined
foo = 1;
console.log(foo); // 1
```



var 키워드와 let 키워드의 호이스팅 차이

그럼 let 키워드는 호이스팅이 안 일어나는 것일까? 아니다!

호이스팅이 안 일어났다면 아래 코드에서 console.log값은 1을 가리키고 있을 것

```
let foo = 1;
{
        console.log(foo); // ReferenceError: Cannot access 'foo' before initializeation;
        let foo = 2;
}
```

10장_객체 리터럴

객체

객체
- 자바스크립트는 객체 기반의 프로그래밍 언어
- 자바스크립트를 구성하는 거의 모든 것이 객체
- 원시 값은 변경 불가능 하지만, 객체 값은 변경이 가능함
- 상태(프로퍼티)와 동작(메서드)을 하나의 단위로 구성함
- **프로퍼티**: 객체의 상태를 나타내는 값
- **메서드**: 프로퍼티를 참고하고 조작할 수 있는 동작.

프로퍼티 값이 함수일 경우, 일반 함수와 구분하기 위해 메서드라고 명명

```
var counter = {
    num : 0,
    increase : function () {
        this.num++;
    }
};
```

프로퍼티 키 : 프로퍼티 값 = 프로퍼티

메서드

객체

※ 메서드의 정의
ES6 사양 이전에는 메서드에 대한 명확한 정의 없음
ES6 사양에서 메서드는 메서드 축약 표현으로 정의된 함수만을 의미
또한 인스턴스를 생성할 수 없는 **non-constructor 함수**임

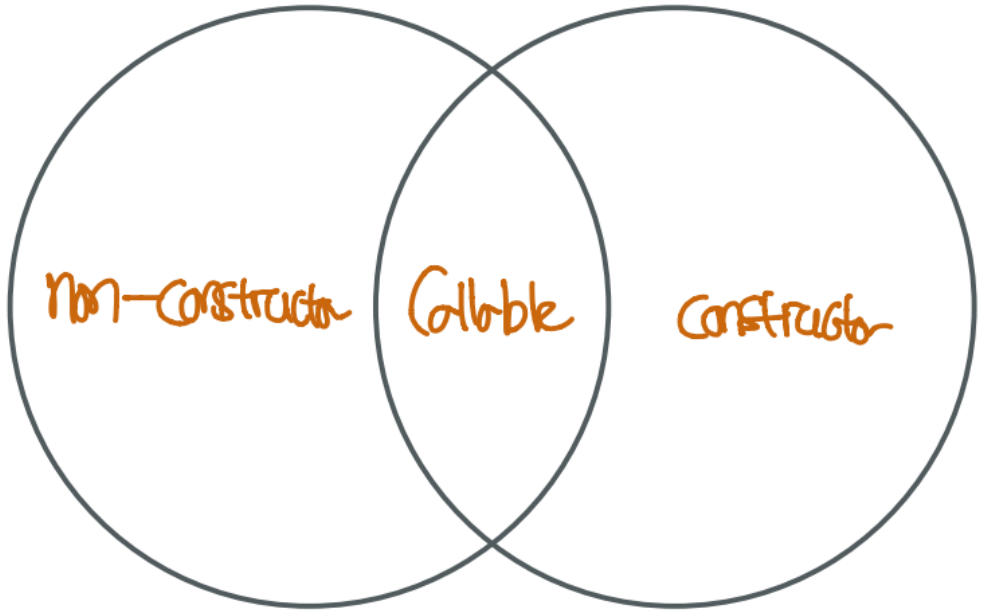
```
const obj = {
        x : 1,
        foo() { return this.x; }; // 메서드
        bar : function() { return this.x; } // 메서드X
}

new obj.foo(); // TypeError : obj.foo is not a constructor
new obj.bar(); // bar{};
```

객체

callable은 함수는 기본적으로 호출 가능

constructor은 일반 함수로만 호출할 수 있는 객체
함수 선언문, 함수 표현식, 클래스

non-constructor은 일반 함수 또는 생성자 함수로서 호출할 수 있는 객체
메서드(ES6 메서드 축약 표현), 화살표 함수



객체

자바스크립트는 다양한 객체 생성 방법을 지원

(1) 객체 리터럴

(2) Object 생성자 함수

(3) 생성자 함수

(4) Object.create 메서드

(5) 클래스(ES6)

객체

프로퍼티
프로퍼티 키는 빈 문자열을 포함하는 모든 문자열 또는 심벌 값
프로퍼티 값은 자바스크립트에서 사용할 수 있는 모든 값

프로퍼티 키는 식별자 역할이며, 일반적으로 문자열 사용
식별자 네이밍 규칙을 준수하지 않는 경우, 반드시 "문자열" 형태로 사용
식별자 네이밍 규칙을 준수하는 경우, 따옴표 생략 가능

```
var person = {
        firstName : 'Ung-mo', // 식별자 네이밍 준수한 경우
        'last-name' : 'Lee' // 식별자 네이밍 준수하지 않은 경우
}

var person = {
        firstName : 'Ung-mo', // 식별자 네이밍 준수한 경우
        last-name : 'Lee' // SyntaxError: Unexpected Token '–' , - 연산자가 있는 표현식으로 해석
}
```

가급적 식별자 네이밍 규칙을 준수하자

객체

프로퍼티를 동적으로 생성하는 경우, 키를 대괄호로 묶는다

```
var obj = {};
var key = 'hello';

obj[key] = 'world';
console.log(obj); // {hello : 'world',}
```

빈 문자열 사용 가능하지만 의미가 없으므로 권장하지 않음

```
var foo = {
        '' : ''
}
```

문자열이나 심벌 값 이외 값을 사용하면 암묵적 타입 변환을 통해 문자열 변환

```
var foo = {
        0 : 1, // '0' : 1
        1 : 2, // '1' : 2
        2 : 3 // '2' : 3
};
```

객체

예약어를 사용해도 에러가 발생하지 않지만, 예상치 못한 상황이 있을 수 있으므로 권장하지 않음

```
var foo = {
        var : '',
        function : ''
};
console.log(foo); // {var:'', function: ''}
```

프로퍼티 키를 중복 선언하면 프로퍼티를 덮어쓴다. 에러가 발상해지 않는다

```
var foo = {
        name : 'Lee',
        name : 'Kim'
};
console.log(foo); // {name: 'Kim'}
```

객체

프로퍼티에 접근하는 방법 두가지

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

// 마침표 표기법에 의한 접근
console.log(person.name);

// 대괄호 표기법에 의한 접근
console.log(person['name']); // 식별자로 인식되기 위해서는 따옴표가 반드시 들어가야 함
```

객체에 존재하지 않는 프로퍼티에 접근하면 undefined 반환

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

console.log(person.age); // undefined
```

객체

식별자 네이밍 규칙을 준수하지 않은 프로퍼티 키를 사용하는 경우, 반드시 대괄호 표기법을 사용

```
var person = {
        'last-name' : 'Lee',
        1:10
}
person.'last-name'; // -> Syntax Error : Unexpected String
person.last-name; // -> 브라우저 환경 : NaN
                  // -> Node.js 환경 : ReferenceError : name is not defiend
person[last-name]; // ReferenceError : last is not defined
person['last-name']; // Lee
```

브라우저 환경에서,
person.last는 평가되어 undefined를 반환, name 식별자는 평가되어 브라우저 전역 변수 (name = '') 반환
결론적으로 undefined – ''는 NaN 출력

브라우저 전역 변수 name은 창의 크기를 나타냄

Node.js 환경에서,
person.last는 평가되어 undefined를 반환, name 식별자는 선언이 없으므로 ReferenceError 반환

객체

프로퍼티 값 갱신

```
var person = {
        name : 'Lee',
}
person.name = 'Kim';
console.log(person.name); // 'Kim'
```

프로퍼티 동적 생성

```
var person = {
        name : 'Lee',
}
person.age = 20;
console.log(person); // {name: "Lee", age: 20}
```

프로퍼티 삭제

```
var person = {
        name : 'Lee',
}
delete person.name;
delete person.address; // 존재하지 않는 프로퍼티를 삭제해도 에러는 발생하지 않음
console.log(person); // {}
```

객체

ES6에서 추가된 객체 리터럴의 확장 기능

(1) 프로퍼티 축약 표현

ES5

```
var x = 1, y = 2;
var obj = {
            x : x,
            y : y
};
```

ES6

```
var x = 1, y = 2;
var obj = {x, y}
```

객체

(2) 계산된 프로퍼티 이름
문자열 또는 문자열로 타입 변환할 수 있는 값으로 평가되는 표현식을 사용해 프로퍼티 키를 동적으로 생성
이때 대괄호로 묶어주어야 함

ES5

```
var prefix = 'prop';
var i = 0;
var obj = {};

obj[prefix + '-' + ++i] = i;
obj[prefix + '-' + ++i] = i;
obj[prefix + '-' + ++i] = i;
console.log(ojb); // {prop-1: 1, prop-2: 2, prop-3: 3}
```

ES6

```
var prefix = 'prop';
var i = 0;
var obj = {;
        [`${prefix}-${++i}`] = i;
        [`${prefix}-${++i}`] = i;
        [`${prefix}-${++i}`] = i;
}
console.log(ojb); // {prop-1: 1, prop-2: 2, prop-3: 3}
```

객체

(3) 메서드 축약 표현

ES5

```
var obj = {
        name: 'Lee',
        sayHi: function() {
                console.log('Hi! ' + this.name);
        }
};
obj.sayHi();
```

ES6

```
var obj = {
        name: 'Lee',
        sayHi() {
                console.log('Hi! ' + this.name);
        }
};
obj.sayHi();
```

11장_원시값과 객체의 비교

원시 타입과 객체 타입

원시 타입
변경 불가능한 값
변수에 실제 값이 저장
값에 의한 전달(call by value)

※ 값에 의한 전달은 ECMAScript 사양에 등장하지 않는 용어
또한 엄격하게 표현하면 변수에는 값이 전달되는 것이 아니라 메모리 주소가 전달됨

객체 타입
변경 가능한 값
변수에 참조 값이 저장
참조에 의한 전달(call by reference)

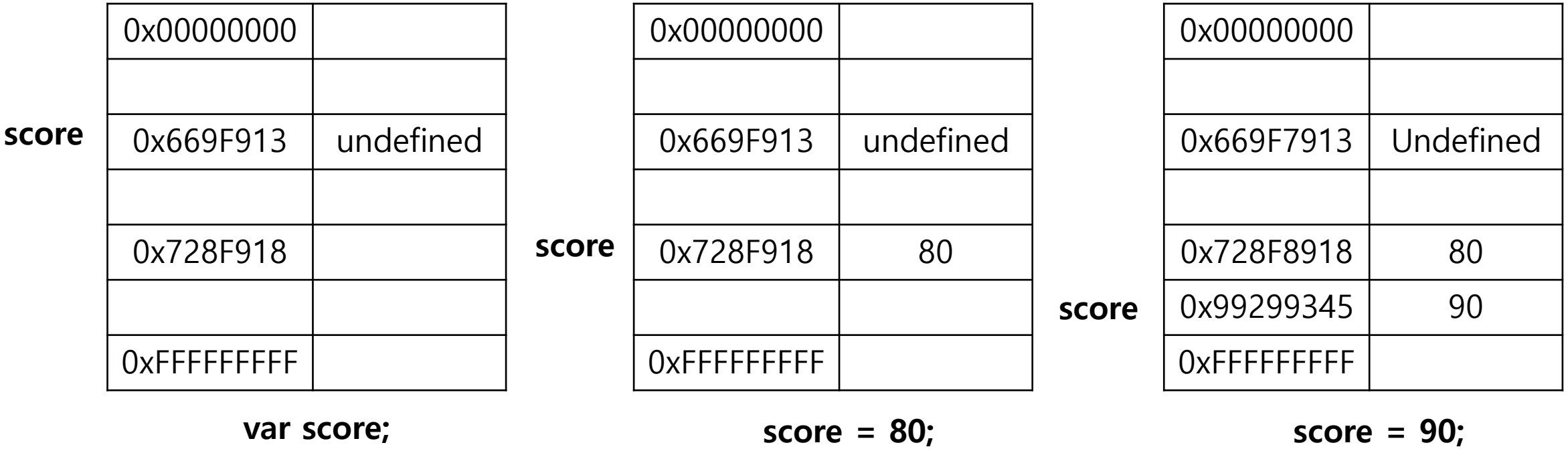
※ 값에 의한 전달, 참조에 의한 전달

기존 변수를 다른 변수에 할당하면,
원시 타입은 값이 복사되어 전달(값에 의한 전달)
객체 타입은 참조 값이 복사되어 전달(참조에 의한 전달)

변경 불가능한 값

변경 불가능한 값(불변성)
읽기 전용 값으로, 어떤 일이 있어도 불변. 이는 데이터의 신뢰성 보장
변수가 아니라 값에 대한 진술
변수는 재할당이 가능

```
var score;
score = 80;
score = 90;
```



문자열

문자열
- 1개의 문자는 2바이트의 메모리 공간에 저장
- 숫자 값은 1도, 1000000도 동일한 8바이트가 필요
- 문자열 길이가 10인 경우, 20바이트 필요
- 주소 방향은 임의로 정한 것

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

str

0x00000000	
0x00000167	e
0x00000168	H
0x00000169	
0xFFFFFFFF	

str

0x00000000	
0x00000167	e
0x00000168	H
0x00000169	
0x12345678	w
0x12345679	
0xFFFFFFFF	

문자열

문자열
- 유사 배열 객체이면서 이터러블
- 읽기 전용 값으로, 어떤 일이 있어도 불변. 이는 데이터의 신뢰성 보장

※ 유사 배열 객체
배열처럼 인덱스로 프로퍼티에 접근 가능, length 프로퍼티를 갖는 객체

```
var str = 'string';
console.log(str[0]);
console.log(str.length);
```

원시 값인 문자열이 객체일수도 있다.
=> 원시 값을 객체처럼 사용하면, 원시 값을 감싸는 래퍼 객체 반환

문자열

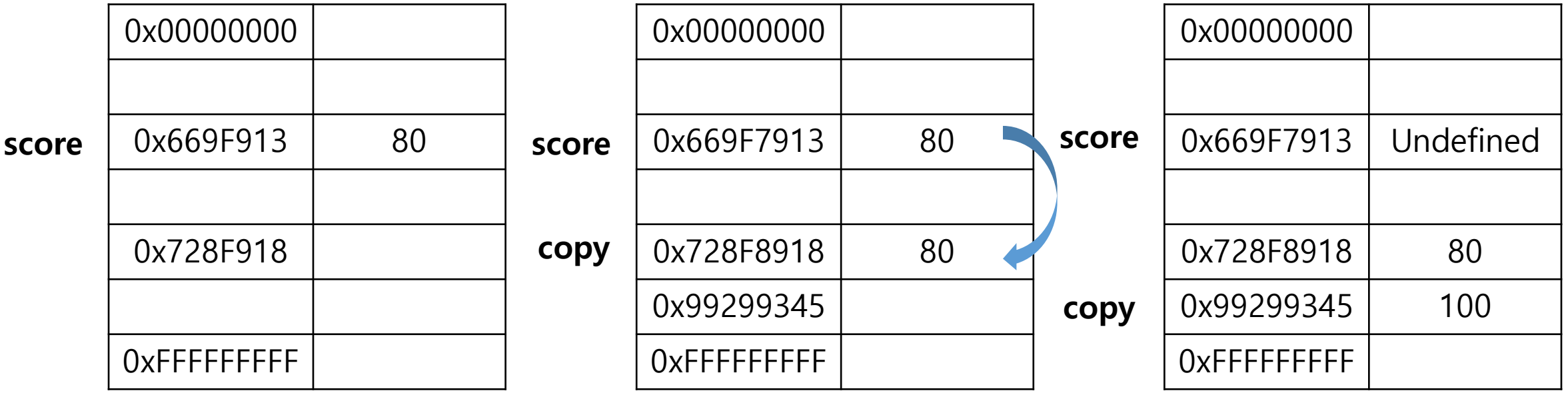
문자열
- 읽기 전용 값으로, 어떤 일이 있어도 불변. 이는 **데이터의 신뢰성 보장**

```
var str = 'string';
str[0] = 'S';
console.log(str); // 'string' 데이터가 안바뀜
```

값에 의한 전달

변수에 원시 값이 저장된 변수를 할당했을 때 무엇이 어떻게 전달되나? **값에 의한 전달**

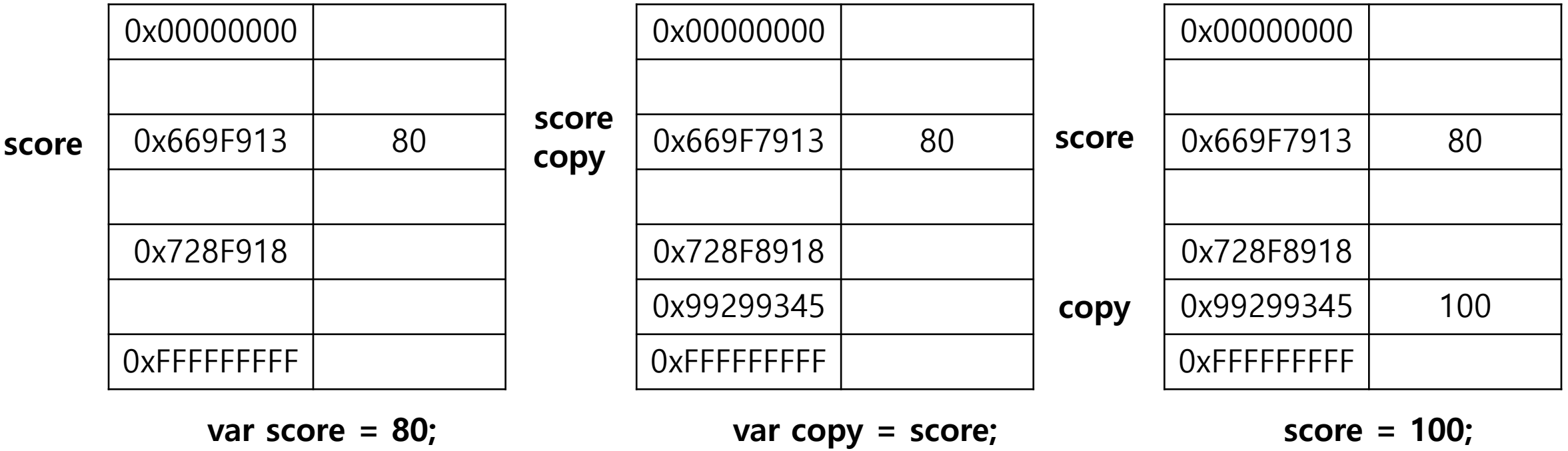
```
var score = 80;
var copy = score;
copy = 100;
console.log(score, copy); // 80 100
```



값에 의한 전달

아래 그림은 JS 엔진의 내부 동작과 정확히 일치하지 않을 수 있음
ECMAScript 사양에는 변수를 통해 메모리를 어떻게 관리해야 하는지 명확하게 정의되어 있지 않음
실제로는 자바스크립트 엔진을 구현하는 제조사에 따라 **내부 동작 방식에 미묘한 차이가 있을 수 있음**

var score = 80;
var copy = score;
copy = 100;
아래는 기존과 다른 방식의 동작



값에 의한 전달

var score = 80;
var copy = score;
var copy = score는 두 가지 평가 방식이 존재
(1) 새로운 80을 생성해서 메모리 주소를 전달. 할당 시점에 두 변수가 기억하는 메모리 주소가 다름
(2) score 메모리 주소를 그대로 전달. 할당 시점에 두 변수가 기억하는 메모리 주소가 같음

중요한 것은 **재할당**
어떤 경우든 재할당을 하게되면 두 변수의 원시 값은 **서로 다른 메모리 공간에 저장된 별개의 값**이 된다는 점