4/26
4/27
4/29
5/3(1시간 30분)

10. 객체 리터럴

TABLE OF CONTENTS

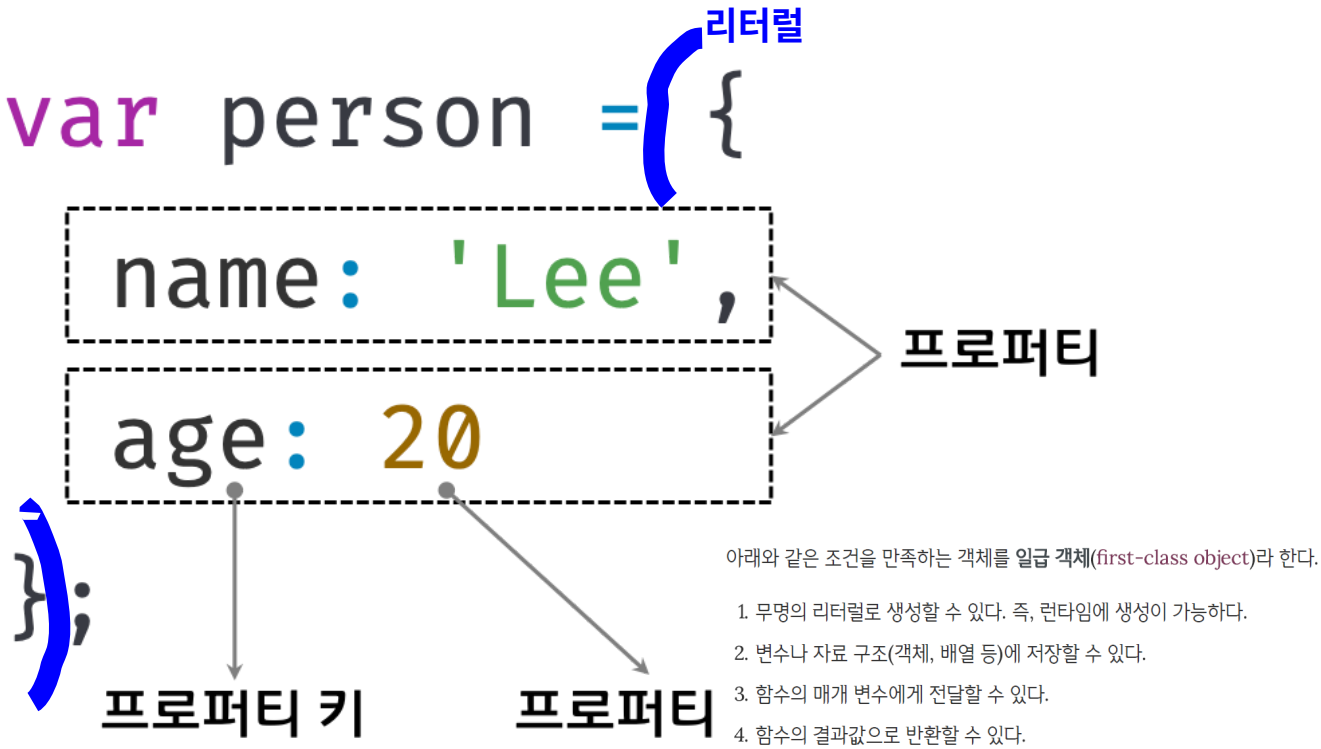
#1. 객체란?

자바스크립트는 객체(object) 기반의 프로그래밍 언어이며 자바스크립트를 이루고 있는 거의 "모든 것"이 객체이다. 원시 값을 제외한 나머지 값들(함수, 배열, 정규표현식 등)은 모두 객체이다.

원시 타입은 단 하나의 값만을 나타내지만 객체 타입(object / reference type)은 다양한 타입의 값(원시 값 또는 다른 객체)들을 하나의 단위로 구성한 복합적인 자료 구조(Data structure)이다. 또한 **원시**

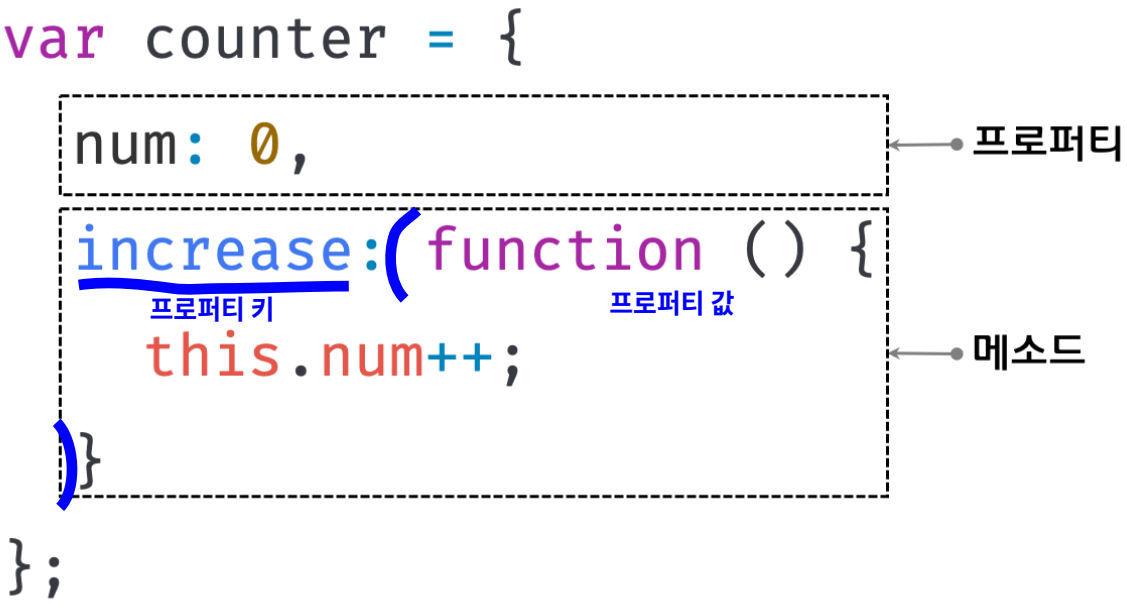
타입의 값, 즉 원시 값은 변경 불가능한 값(immutable value)**이지만 객체 타입의 값, 즉 객체는 변경 가능한 값**(mutable value)**이다.** 이에 대해서는 다음 장 "11. 원시 값과 객체의 비교"에서 자세히 살펴볼 것이다.

객체는 0개 이상의 프로퍼티의 집합이며 프로퍼티는 키(key)와 값(value)으로 구성된다.



객체는 프로퍼티의 집합이다.

자바스크립트에서 사용할 수 있는 모든 값은 프로퍼티 값이 될 수 있다. 자바스크립트의 함수는 일급 객체("18.1. 일급 객체" 참고)이므로 값으로 취급할 수 있다. 따라서 함수도 프로퍼티 값으로 사용할 수 있다. 프로퍼티 값이 함수일 경우, 일반 함수와 구분하기 위해 메소드(method)라 부른다.



객체의 프로퍼티와 메소드

이처럼 객체는 프로퍼티와 메소드로 구성된 집합체이다. 프로퍼티와 메소드의 역할은 아래와 같다.

- 프로퍼티: 객체의 상태를 나타내는 값(data) ->'상태'란 것은 언젠가는 변한다는 것을 가정하고 쓰는 표현임
- 메소드: 프로퍼티(상태 데이터)를 참조하고 조작할 수 있는 동작(behavior)
  ->이름 지을 때 프로퍼티는 명사, 메소드는 동사 혹은 동사+목적어

이와 같이 객체는 객체의 상태를 나타내는 값(프로퍼티)과 프로퍼티를 참조하고 조작할 수 있는 동작(메소드)를 모두 포함할 수 있기 때문에 상태와 동작을 하나의 단위로 구조화할 수 있어 유용하다.

객체와 함수 생성자 함수 메소드

자바스크립트의 객체는 함수와 밀접한 관계를 갖는다. 함수로 객체를 생성하기도 하며 함수 자체가 객체이다. 자바스크립트에서 함수와 객체는 분리해서 생각할 수 없는 개념이다. 즉, 객체를 이해해야 함수를 제대로 이해할 수 있고 반대로 함수를 이해해야 객체를 정확히 이해할 수 있다. 따라서 객체와 함수를 분리하여 설명하는 것은 옳지 않지만 책의 구성 상 객체와 함수를 번갈아 가며 설명하고자 한다.
프로그래밍 언어 자체에는 순서가 없고 서로 물고 물리는 순환 구조를 갖는다. 잘 이해가 되지 않는 개념이 나오면 멈추지 말고 일단은 다음으로 넘어가는 것도 하나의 방법이다. 가급적 상위 개념을 먼저 살펴보고 이를 기반으로 좀 더 복잡한 개념을 알아보도록 하자.

#2. 객체 리터럴에 의한 객체 생성

C++과 Java와 같은 클래스 기반 객체지향 언어는 클래스를 사전에 정의하고 필요한 시점에 new 연산자와 함께 생성자(constructor)를 호출하여 인스턴스를 생성하는 방식으로 객체를 생성한다.

인스턴스
인스턴스(instance)란 클래스에 의해 생성되어 메모리에 저장된 실체를 말한다. 객체 지향 프로그래밍에서 객체는 클래스와 인스턴스를 포함한 개념이다. 클래스는 인스턴스를 생성하기 위한 템플릿의 역할을 한다. 인스턴스는 객체가 메모리에 저장되어 실제로 존재하는 것에 초점을 맞춘 용어이다.

하지만 자바스크립트는 프로토타입 기반 객체지향 언어로서 클래스 기반 객체지향 언어와는 다른 다양한 객체 생성 방법이 존재한다.

- 객체 리터럴
- Object 생성자 함수
- 생성자 함수
- Object.create 메소드
- 클래스 (ES6)

위 객체 생성 방법 중에서 가장 일반적이고 간단한 방법은 객체 리터럴을 사용하는 방법이다. “5.2. 리터럴”에서 살펴보았듯이 리터럴(literal)은 사람이 이해할 수 있는 문자 또는 약속된 기호를 사용하여 값을 생성하는 표기법(notation)을 말한다. 객체 리터럴은 객체를 생성하는 표기법이다.

객체 리터럴은 중괄호({...}) 내에 0개 이상의 프로퍼티를 정의한다. 변수에 할당이 이루어지는 시점에 자바스크립트 엔진은 객체 리터럴을 해석하여 객체를 생성한다.

JAVASCRIPT

```
var person = {
  name: 'Lee',
  sayHello: function () {
    console.log(`Hello! My name is ${this.name}.`);
  }
};

console.log(typeof person); // object
console.log(person); // {name: "Lee", sayHello: ƒ}
```

만약 중괄호 내에 프로퍼티를 정의하지 않으면 빈 객체가 생성된다.

JAVASCRIPT

```
var empty = {}; // 빈 객체
console.log(typeof empty); // object
```

객체 리터럴의 중괄호는 코드 블록을 의미하지 않음에 주의하자. 코드 블록의 닫는 중괄호 뒤에는 세미 콜론을 붙이지 않는다. 하지만 객체 리터럴은 값으로 평가되는 표현식이다. 따라서 객체 리터럴의 닫는 중괄호 뒤에는 세미 콜론을 붙인다.

객체 리터럴은 자바스크립트의 유연함과 강력함을 대표하는 객체 생성 방식이다. 객체를 생성하기 위해 클래스를 먼저 정의하고 new 연산자와 함께 생성자를 호출할 필요가 없다. 숫자 값이나 문자열을 만드는 것과 유사하게 리터럴로 객체를 생성한다. 객체 리터럴에 프로퍼티를 포함시켜 객체의 생성과 동시에 프로퍼티를 만들 수도 있고 객체를 생성한 이후에 프로퍼티를 동적으로 추가할 수도 있다.

객체 리터럴 이외의 객체 생성 방식은 모두 함수를 사용해 객체를 생성한다. 이 방법들에 대해서는 함수를 알아본 이후에 살펴보도록 하자.

#3. 프로퍼티

객체는 프로퍼티(Property)들의 집합이며 프로퍼티는 키(key)와 값(value)으로 구성된다. 프로퍼티를 나열할 때는 쉼표(,)로 구분한다. 일반적으로 마지막 프로퍼티 뒤에는 쉼표를 사용하지 않으나 사용해도 좋다.

JAVASCRIPT

```
var person = {
  // 프로퍼티 키는 name, 프로퍼티 값은 'Lee'
  name: 'Lee',
  // 프로퍼티 키는 age, 프로퍼티 값은 20
  age: 20
};
```

- 식별자는 특수문자를 제외한 문자, 숫자, underscore(_), 달러 기호($)를 포함할 수 있다.
- 단, 식별자는 특수문자를 제외한 문자, underscore(_), 달러 기호($)로 시작해야 한다. 숫자로 시작하는 것은 허용하지 않는다.
- 예약어는 식별자로 사용할 수 없다.

프로퍼티 키와 프로퍼티 값으로 사용할 수 있는 값은 아래와 같다.

- 프로퍼티 키 : 빈 문자열을 포함하는 모든 문자열 또는 symbol 값
- 프로퍼티 값 : 자바스크립트에서 사용할 수 있는 모든 값

식별자역할을 할 뿐 식별자는아님 .
참고로 식별자는 스코프체인에서 찾고,
프로퍼티키는 프로토타입에서 찾음

3.1 프로퍼티 키(4.메소드 전까지 모두 프로퍼티 키에 대한 설명)

프로퍼티 키는 프로퍼티 값에 접근할 수 있는 이름으로 식별자 역할을 한다. 하지만 반드시 식별자 네이밍 규칙(“4.7. 식별자 네이밍 규칙” 참고)을 따라야 하는 것은 아니다. 단, 식별자 네이밍 규칙을 준수하는 프로퍼티 키와 그렇지 않은 프로퍼티 키는 미묘한 차이가 있다.

symbol 값도 프로퍼티 키로 사용할 수 있지만 일반적으로 문자열을 사용한다. 이때 프로퍼티 키는 문자열이므로 따옴표('...' 또는 "...")로 묶어야 한다. 하지만 식별자 네이밍 규칙을 준수하는 이름, 즉 자바스크립트에서 사용 가능한 유효한 이름인 경우, 따옴표를 생략할 수 있다. 반대로 말하면 **식별자 네이밍 규칙을 따르지 않는 이름에는 반드시 따옴표를 사용하여야 한다.** 아래 예제를 살펴보자.

JAVASCRIPT

```
var person = {
  firstName: 'Ung-mo', // 유효한 이름
  'last-name': 'Lee'   // 유효하지 않은 이름
};

console.log(person); // {firstName: "Ung-mo", last-name: "Lee"}
```

프로퍼티 키로 사용한 firstName은 식별자 네이밍 규칙을 준수하고 있다. 따라서 따옴표를 생략할 수 있다. 하지만 last-name은 식별자 네이밍 규칙을 준수하고 있지 않다. 따라서 따옴표를 생략할 수 없다. 자바스크립트 엔진은 따옴표를 생략한 last-name을 - 연산자가 있는 표현식으로 해석한다.

JAVASCRIPT

```
var person = {
  firstName: 'Ung-mo',
  last-name: 'Lee' // SyntaxError: Unexpected token -
};
```

문자열 또는 문자열로 평가할 수 있는 표현식을 사용해 프로퍼티 키를 동적으로 생성할 수도 있다. 이 경우에는 프로퍼티 키로 사용할 표현식을 대괄호([...])로 묶어야 한다. 이를 계산된 프로퍼티 이름(Computed property name, "10.9.2. 프로퍼티 키 동적 생성" 참고)이라고 한다.

JAVASCRIPT

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

// ES5: 프로퍼티 키 동적 생성
obj[key] = 'world';
// ES6: 프로퍼티 키 동적 생성
// var obj = { [key]: 'world' };
                    프로퍼티 키(key) 표현식만 대괄호로 묶어줌
console.log(obj); // {hello: "world"}
```

빈 문자열을 프로퍼티 키로 사용해도 에러가 발생하지는 않는다. 하지만 키로서의 의미를 갖지 못하므로 권장하지 않는다.

JAVASCRIPT

```
var foo = {
  '': ''  // 빈문자열도 프로퍼티 키로 사용할 수 있다.
};

console.log(foo); // {"": ""}
```

프로퍼티 키에 문자열이나 symbol 값 이외의 값을 사용하면 암묵적 타입 변환을 통해 문자열이 된다. 예를 들어, 프로퍼티 키로 숫자 리터럴을 사용하면 따옴표는 붙지 않지만 내부적으로는 문자열로 변환된다.

JAVASCRIPT

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

console.log(foo); // {0: 1, 1: 2, 2: 3}
```

var, function과 같은 예약어(Reserved word)를 프로퍼티 키로 사용해도 에러가 발생하지 않는다. 하지만 예상치 못한 에러가 발생시킬 여지가 있으므로 권장하지 않는다.

JAVASCRIPT

```
var foo = {
  var: '',
  function: ''
};

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

이미 존재하는 프로퍼티 키를 중복 선언하면 나중에 선언한 프로퍼티가 먼저 선언한 프로퍼티를 덮어쓴다. 이때 에러가 발생하지 않는 것에 주의하자.

JAVASCRIPT

```
var foo = {
  name: 'Lee',
  name: 'Kim'
};

console.log(foo); // {name: "Kim"}
```

4. 메소드

자바스크립트에서 사용할 수 있는 모든 값은 프로퍼티 값으로 사용할 수 있다고 했다. 아직 살펴보지 않았지만 자바스크립트의 함수는 객체(일급 객체, First-class object)이다. 따라서 함수는 값으로 취급할 수 있기 때문에 프로퍼티 값으로 사용할 수 있다.

프로퍼티 값이 함수일 경우, 일반 함수와 구분하기 위해 메소드(method)라 부른다. 즉, 메소드는 객체에 제한되어 있는 함수를 의미한다. 함수는 "12. 함수"에서 자세히 살펴볼 것이다.

JAVASCRIPT

```
var circle = {
  radius: 5, // ← 프로퍼티

  // 원의 지름
  getDiameter: function () { // ← 메소드
    return 2 * this.radius; // this는 circle를 가리킨다.
  }
};

console.log(circle.getDiameter());  // 10
```

this 미입력 시 아래와 같은 에러 발생

```
ReferenceError: radius is not defined
```

호출 연산자 미입력시 10이 아닌 아래와 같은 값 반환

```
[Function: getDiameter]
```

메소드 내부에서 사용한 this 키워드는 객체 자신(위 예제에서는 circle 객체)을 가리키는 참조변수이다. 이에 대해서는 "22. this"에서 자세히 살펴볼 것이다.

5. 프로퍼티 접근

프로퍼티 값에 접근하려면 마침표(.)를 사용하는 **마침표 표기법**(Dot notation) 또는 대괄호([...])를 사용하는 **대괄호 표기법**(Bracket notation)을 사용한다.

프로퍼티 키가 식별자 네이밍 규칙을 따르는 이름, 즉 자바스크립트에서 사용 가능한 유효한 이름이면 마침표 표기법과 대괄호 표기법을 모두 사용할 수 있다.

마침표 또는 대괄호의 좌측에는 객체로 평가할 수 있는 표현식을 기술한다. 마침표의 우측 또는 대괄호의 내부에는 프로퍼티 키를 지정한다.

JAVASCRIPT

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

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

// 대괄호 표기법에 의한 프로퍼티 접근
console.log(person['name']); // Lee
```

대괄호 표기법을 사용하는 경우, **대괄호 내부에 지정하는 프로퍼티 키는 반드시 따옴표로 감싼 문자열이어야 한다**. 대괄호 내의 따옴표로 감싸지 않은 이름을 프로퍼티 키로 사용하면 자바스크립트 엔진은 식별자로 해석한다.

JAVASCRIPT

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

console.log(person[name]); // ReferenceError: name is not defined
```

위 예제에서 ReferenceError가 발생한 이유는 대괄호 내의 따옴표로 감싸지 않은 이름, 즉 식별자 name을 평가하기 위해 선언된 name을 찾았지만 찾지 못했기 때문이다.

객체에 존재하지 않는 프로퍼티에 접근하면 undefined를 반환한다. 이때 ReferenceError가 발생하지 않는 것에 주의하자.

JAVASCRIPT

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

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

프로퍼티 키가 식별자 네이밍 규칙을 준수하지 않는 이름, 즉 자바스크립트에서 사용 가능한 유효한 이름이 아니면 반드시 대괄호 표기법을 사용해야 한다. 단, 프로퍼티 키가 숫자로 이루어진 문자열인 경우, 따옴표를 생략 가능하다. 그 외의 경우, 대괄호 내에 들어가는 프로퍼티 키는 반드시 따옴표로 감싼 문자열이어야 함을 잊지 않도록 하자.

*** 대괄호 표기법_식별자 네이밍 규칙을 준수하지 않는 경우 사용(마침표 표기법은 안됨)**
- 문자열 : 반드시 따옴표로 감싸야 함
- 숫자로 이루어진 문자열 : 따옴표 생략 가능
(왜 숫자로 이루어진 문자열? 프로퍼티 키에 숫자 리터럴을 사용해도 암묵적 타입 변환을 통해 문자열로 타입 변환되기 때문에)
-> 식별자가 숫자로 시작하는 경우 식별자 네이밍 규칙에 어긋나므로 마침표 표기법 사용x

JAVASCRIPT

```
var person = {
  'last-name': 'Lee',
  1: 10
};
```

->마침표 표기법 뒤에는 따옴표가 올 수 없음

```
person.'last-name'; // → SyntaxError: Unexpected string
person.last-name;   // -> 브라우저 환경: NaN
                    // → Node.js 환경: ReferenceError: name is not defined
person[last-name];  // → ReferenceError: last is not defined
person['last-name']; // → Lee

// 프로퍼티 키가 숫자로 이루어진 문자열인 경우, 따옴표를 생략 가능하다.
person.1;    // → SyntaxError: Unexpected number
person.'1';  // → SyntaxError: Unexpected string
person[1];   // → 10 : person[1] → person['1']   ->이 표현이 더 컨벤션에 맞음
person['1']; // → 10
```

객체에 존재하지 않는 프로퍼티에 접근하면 undefined를 반환한다. 이때 ReferenceError가 발생하지 않는 것에 주의하자.

여기서 퀴즈를 하나 풀어보자. 위 예제에서 `person.last-name`의 실행 결과는 Node.js 환경에서 ReferenceError: name is not defined이고 브라우저 환경에서는 NaN이다. 이유는 무엇인가?

자바스크립트 엔진은 먼저 person.last를 평가한다. 평가 결과는 undefined이다. person 객체에는 프로퍼티 키가 last인 프로퍼티가 없기 때문에 평가 결과는 undefined이다. 따라서 `person.last-name`는 `undefined - name`과 같다. 다음으로 자바스크립트 엔진은 name이라는 식별자를 찾는다. 이때 name은 프로퍼티 키가 아니라 식별자로 해석되는 것에 주의하자.

last : 프로퍼티 키로 해석->undefined(공통)
name : 식별자로 해석

Node.js 환경에서는 현재 어디에도 name이라는 식별자(변수, 함수 등의 이름) 선언이 없으므로 ReferenceError: name is not defined이라고 에러가 발생한다. 그런데 그런데 브라우저 환경에서는 name이라는 전역 변수가 자바스크립트 엔진에 의해 암묵적으로 존재한다. 전역 변수 name은 창(window)의 이름을 가리키며 기본값은 빈문자열이다. 따라서 `person.last-name`는 `undefined - ''`과 같으므로 NaN이 된다.

6. 프로퍼티 값 갱신

이미 존재하는 프로퍼티에 값을 할당하면 프로퍼티 값이 갱신된다. **재할당이 아닌 갱신이라는 것 !!**

JAVASCRIPT

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

// person 객체에 name 프로퍼티가 존재하므로 name 프로퍼티의 값이 갱신된다.
person.name = 'Kim';

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

7. 프로퍼티 동적 생성

존재하지 않는 프로퍼티에 값을 할당하면 프로퍼티가 동적으로 생성되어 추가되고 프로퍼티 값이 할당된다. **-> 다른 언어에는 없는 자바스크립트의 장점!!!**
cf) 존재하지 않는 프로퍼티를 호출하면? undefined 값 반환

JAVASCRIPT

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

// person 객체에는 age 프로퍼티가 존재하지 않는다.
// 따라서 person 객체에 age 프로퍼티가 동적으로 생성되고 값이 할당된다.
person.age = 20;

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

8. 프로퍼티 삭제

프로퍼티 삭제 안하는 것을 권장
왜? 있었는데 지웠으므로 안정성이ㅍ떨어짐
(난 필요 없어서 삭제했지만 다른사람은 필요할 수도 있음)

delete 연산자는 객체의 프로퍼티를 삭제한다. 이때 delete 연산자의 피연산자는 프로퍼티 값에 접근할 수 있는 표현식이어야 한다. 만약 존재하지 않는 프로퍼티를 삭제하면 아무런 에러없이 무시된다.

JAVASCRIPT

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

// 프로퍼티 동적 생성
person.age = 20;

// person 객체에 age 프로퍼티가 존재한다.
// 따라서 delete 연산자로 age 프로퍼티를 삭제할 수 있다.
delete person.age;

// person 객체에 address 프로퍼티가 존재하지 않는다.
// 따라서 delete 연산자로 address 프로퍼티를 삭제할 수 없다. 이때 에러가 발생하지 않는다.
delete person.address;

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

9. ES6에서 추가된 객체 리터럴의 확장 기능

ES6에서는 더욱 간편하고 표현력 있는 객체 리터럴의 확장 기능을 제공한다.

9.1. 프로퍼티 축약 표현

객체 리터럴의 프로퍼티는 프로퍼티 키와 프로퍼티 값으로 구성된다. 프로퍼티의 값은 변수에 할당된 값, 즉 식별자 표현식일 수도 있다.

JAVASCRIPT

```
// ES5
var x = 1, y = 2;

var obj = {
  x: x,
  y: y
};

console.log(obj); // {x: 1, y: 2}
```

ES6에서는 프로퍼티 값으로 변수를 사용하는 경우, 변수 이름과 프로퍼티 키가 동일한 이름일 때, 프로퍼티 키를 생략(Property shorthand)할 수 있다. 이때 프로퍼티 키는 변수 이름으로 자동 생성된다.

JAVASCRIPT

```
// ES6
let x = 1, y = 2;

// 프로퍼티 축약 표현
const obj = { x, y };

console.log(obj); // {x: 1, y: 2}
```

9.2. 프로퍼티 키 동적 생성 -> 이건 그냥 읽어보세요

문자열 또는 문자열로 변환 가능한 값을 반환하는 표현식을 사용해 프로퍼티 키를 동적으로 생성할 수 있다. 단, 프로퍼티 키로 사용할 표현식을 대괄호([...])로 묶어야 한다. 이를 계산된 프로퍼티 이름(Computed property name)이라 한다.

ES5에서 계산된 프로퍼티 이름으로 프로퍼티 키를 동적 생성하려면 객체 리터럴 외부에서 대괄호([...]) 표기법을 사용해야 한다.

JAVASCRIPT

```
// ES5
var prefix = 'prop';
```

```
var i = 0;

var obj = {};

// 프로퍼티 키 동적 생성-> 객체 리터럴 외부
obj[prefix + '-' + ++i] = i;
obj[prefix + '-' + ++i] = i;
obj[prefix + '-' + ++i] = i;

console.log(obj); // {prop-1: 1, prop-2: 2, prop-3: 3}
```

ES6에서는 객체 리터럴 내부에서도 계산된 프로퍼티 이름으로 프로퍼티 키를 동적 생성할 수 있다.

JAVASCRIPT

```
// ES6
const prefix = 'prop';
let i = 0;

// 객체 리터럴 내부에서 프로퍼티 키 동적 생성
const obj = {
  [`${prefix}-${++i}`]: i,
  [`${prefix}-${++i}`]: i,
  [`${prefix}-${++i}`]: i
};

console.log(obj); // {prop-1: 1, prop-2: 2, prop-3: 3}
```

9.3. 메소드 축약 표현

ES5에서 메소드를 정의하려면 프로퍼티 값으로 함수를 할당한다.

JAVASCRIPT

```
// ES5
var obj = {
  name: 'Lee',
```

```
  sayHi: function() {
    console.log('Hi! ' + this.name);
  }
};

obj.sayHi(); // Hi! Lee
```

ES6에서는 메소드를 정의할 때, function 키워드를 생략한 축약 표현을 사용할 수 있다.

JAVASCRIPT

```
// ES6
const obj = {
  name: 'Lee',
  // 메소드 축약 표현
  sayHi() {
    console.log('Hi! ' + this.name);
  }
};

obj.sayHi(); // Hi! Lee
```

ES6의 메소드 축약 표현으로 정의한 메소드는 프로퍼티에 할당한 함수와 다르게 동작한다. 이에 대해서는 “26.2 메소드”에서 자세히 살펴보도록 하자.