자바스크립트 문법

2/ 자바스크립트 문법

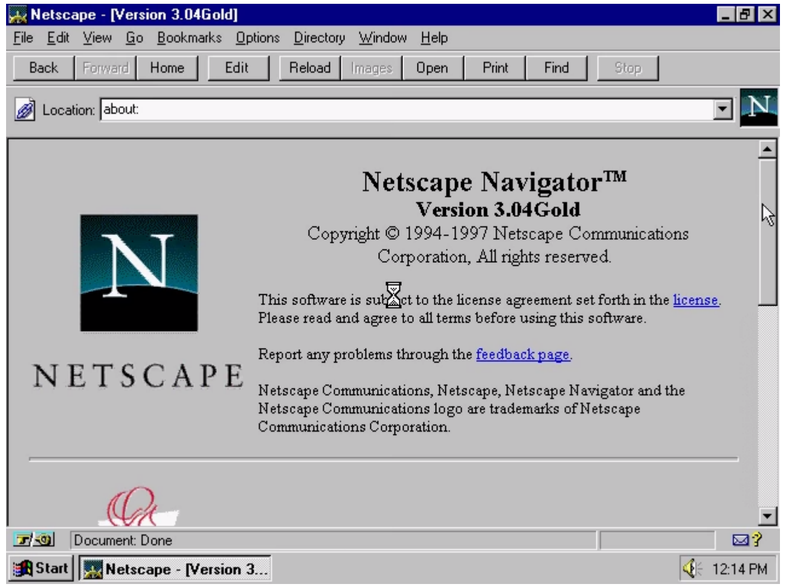
목차

01

자바스크립트 등장 배경, 특징

자바스크립트 등장 배경

- 넷스케이프 커뮤니케이션즈에서 웹페이지의 보조적인 기능을 수행하기 위해 브라우저에서 동작하는 경량 프로그래밍 언어를 도입
  - 자바스크립트가 탄생하게 된 계기
- ECMAScript로 불리기도 함



자바스크립트 특징

프론트엔드	React, Vue, Angular
백엔드	NodeJS
데스크톱앱	Electron
모바일앱	Cordova / React Native

02

변수(식별자)

변수

변수란?

- 하나의 값을 저장하기 위해 확보한 메모리 공간 또는 그 메모리 공간을 식별하기 위해 붙인 이름

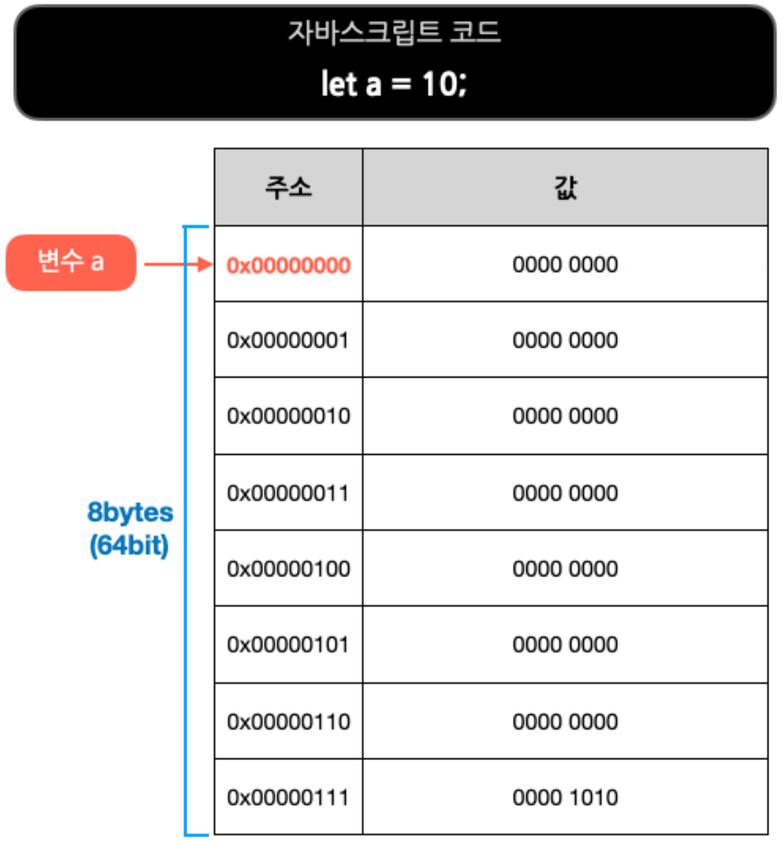
- 메모리 공간에 저장된 값을 식별할 수 있는 고유한 이름을 **변수 이름(변수명)**이라 하고 변수에 저장된 값을 **변수 값** 이라 한다.

- 변수에 값을 저장하는 것을 **할당(대입, 저장)** 이라 하고, 변수에 저장된 값을 읽어 들이는 것을 **참조** 라고 한다.

참고: 모던 자바스크립트 Deep Dive

변수

변수 이름을 식별자 라고도 하며, 식별자는 값이 아니라 메모리 주소를 기억하고 있다.



변수를 사용하려면 선언이 필요하다.
JS에서 변수 선언시 var, let, const 키워드를 사용함

이미지 출처: https://curryyou.tistory.com/275

문

- 문은 프로그램을 구성하는 기본 단위이자 최소 실행 단위이다.
- 문은 선언문, 할당문, 조건문, 반복문 등으로 구분할 수 있다.

```
// 변수 선언문
var x;

// 할당문
x = 5;

// 함수 선언문
function foo() {}

// 조건문
if(x > 1) { console.log(x) }

// 반복문
for (var i = 0; i < 2; i++) { console.log(i); }
```

03

연산자와 제어문

연산자와 제어문

- 연산자는 하나 이상의 표현식을 대상으로 해서
  - 산술, 할당, 비교, 논리, 타입, 지수 연산 등을 수행
- 연산 대상: 피연산자

```
// 산술 연산자
3 * 5 // 15

// 문자열 연결 연산자
'My name is ' + 'Park' // -> 'My name is Park'

// 할당 연산자
color = 'blue' // -> 'blue'

// 비교 연산자
3 > 5 // false

// 논리 연산자
true && false // -> false

// 타입 연산자
typeof 'Hi' // -> string
```

➔ 연산자 피연산자 연산자

연산자와 제어문

산술 연산자

- 피연산자에 대해 수학적인 계산을 하여 숫자 값을 만듬. 산술 연산이 불가능한 경우 NaN 반환
  - 이항 산술 연산자
  - 2개의 피연산자를 산술 연산하여 숫자 값을 만듬
  - + (덧셈), - (뺄셈), * (곱셈), / (나눗셈), % (나머지)

연산자와 제어문

단항 산술 연산자

- 1개의 피연산자를 산술 연산하여 숫자 값을 만든다
- ++ (증가), -- (감소), + (효과 X), - (양수 → 음수, 음수 → 양수로 반전)

```
// ++ 연산자는 값을 증가 시킨다
x++; // x = x + 1;
console.log(x); // 2

// -- 연산자는 값을 감소 시킨다
x--; // x = x - 1;
console.log(x); // 1
```

연산자와 제어문

증가/감소 연산자는 위치에 따라 동작이 달라짐

- 피연산자 앞: 먼저 피연산자의 값을 증가/감소시킨 후, 다른 연산을 수행
- 피연산자 뒤: 먼저 다른 연산을 수행한 후, 피연산자의 값을 증가/감소

```
// 선할당 후증가
result = x++;
console.log(result, x); // 5 6

// 선증가 후할당
result = ++x;
console.log(result, x); // 7 7

// 선할당 후감소
result = x--;
console.log(result, x); // 7 6

// 선감소 후할당
result = --x;
console.log(result, x); // 5 5
```

연산자와 제어문

문자열 연결 연산자

+ 연산자는 피연산자 중 하나 이상이 문자열인 경우 문자열 연결 연산자로 동작한다

```
// 문자열 연결 연산자
'1' + 2; // '12'
1 + '2' // '12'

// true는 1로 타입 변환된다.
1 + true; // 2

// false는 0으로 타입 변환된다.
1 + false; // 1

// null은 0으로 타입 변환된다.
1 + null; // 1

// undefined는 숫자로 타입 변환되지 않는다.
+undefined // NaN
1 + undefined // NaN
```

연산자와 제어문

할당 연산자

할당 연산자	예	동일 표현
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x – 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5

연산자와 제어문

비교 연산자

비교 연산자는 좌항과 우항의 피연산자를 비교한 다음 그 결과를 불리언 값으로 반환

1. 동등/일치 비교 연산자

비교 연산자	의미	사례	설명
==	동등 비교	x == y	x와 y의 값이 같음
===	일치 비교	x === y	x와 y의 값과 타입이 같음
!=	부동등 비교	x != y	x와 y의 값이 다름
!==	불일치 비교	x !== y	x와 y의 값과 타입이 다름

연산자와 제어문

비교 연산자

2. 대소 관계 비교 연산자

대소 관계 비교 연산자는 피연산자의 크기를 비교하여 불리언 값을 반환한다

대소 관계 비교 연산자	예제	설명
>	x > y	x가 y보다 크다
<	x < y	x가 y보다 작다
>=	x >= y	x가 y보다 크거나 같다
<=	x <= y	x가 y보다 작거나 같다

연산자와 제어문

삼항 조건 연산자

- 삼항 조건 연산자는 조건식의 평가 결과에 따라 반환할 값을 결정한다
- 조건식 ? 조건식이 true일 때 반환할 값 : 조건식이 false일 때 반환할 값

```
score >= 60 ? 'pass' : 'fail';
```

- 삼항 조건 연산자 표현식은 값으로 평가할 수 있는 표현식인 문

연산자와 제어문

논리 연산자

- 우항과 좌항의 피연산자를 논리 연산한다.

논리 연산자	의미
||	논리합(OR)
&&	논리곱(AND)
!	부정(NOT)

연산자와 제어문

쉼표 연산자

- 왼쪽 피연산자부터 차례대로 피연산자를 생성하고 마지막 값을 반환

```
var x, y, z;
x = 1, y = 2, z = 3; // 3
```

```
let x = 1;

x = (x++, x);

console.log(x);
// Expected output: 2

x = (2, 3);

console.log(x);
// Expected output: 3
```

그룹 연산자

- 소괄호로 피연산자를 감싸는 그룹 연산자는 자신의 피연산자인 표현식을 가장 먼저 평가한다, 그룹 연산자를 사용하면 연산자의 우선순위를 조절할 수 있다.

연산자와 제어문

그룹 연산자

```
10 * 2 + 3; // 23
// 그룹 연산자를 사용하여 우선순위를 조절
10 * (2 + 3) // 50
```

typeof 연산자

- typeof 연산자는 피연산자의 데이터 타입을 문자열로 반환한다

연산자와 제어문

typeof 연산자

```
typeof ''               // -> "string"
typeof 1                // -> "number"
typeof NaN              // -> "number"
typeof true             // -> "boolean"
typeof undefined        // -> "undefined"
typeof Symbol()         // -> "symbol"
typeof null             // -> "object"
typeof []               // -> "object"
typeof {}               // -> "object"
typeof new Date()       // -> "object"
typeof /test/gi         // -> "object"
typeof function () {}   // -> "function"
```

연산자와 제어문

지수 연산자

- ES7에서 도입된 지수 연산자는 좌항의 피연산자를 밑으로, 우항의 피연산자를 지수로 거듭 제곱하여 숫자 값을 반환한다.

```
2 ** 2;   // -> 4
2 ** 2.5; // -> 5.65685424949238
2 ** 0;   // -> 1
2 ** -2;  // -> 0.25
```

- 지수 연산자가 도입되기 전에는 Math.pow 메서드를 사용했다

```
Math.pow(2, 2);   // -> 4
Math.pow(2, 2.5); // -> 5.65685424949238
Math.pow(2, 0);   // -> 1
Math.pow(2, -2);  // -> 0.25
```

연산자와 제어문

그 외 연산자

연산자	개요
?.	옵셔널 체이닝 연산자
??	null 병합 연산자
delete	프로퍼티 삭제
new	생성자 함수를 호출할 때 사용하여 인스턴스를 생성
instanceof	좌변의 객체가 우변의 생성자 함수와 연결된 인스턴스인지 판별
in	프로퍼티 존재 확인

자바스크립트 문법

우선순위	연산자
1	()
2	new(매개변수 존재), ., [](프로퍼티 접근), ()(함수호출), ?.(옵셔널 체이닝 연산자)
3	new(매개변수 미존재)
4	x++, x--
5	!x, +x, -x, ++x, --x, typeof, delete
6	**
7	*, /, %
8	+, -
9	<, <=, >, >=, in, instanceof
10	==, !=, ===, !==
11	??(null 병합 연산자)
12	&&
13	||
14	? … : …
15	할당 연산자(=, +=, -=, …)
16	,

연산자와 제어문

- 제어문은 조건에 따라 코드 블록을 실행(조건문)하거나 반복 실행(반복문)할 때 사용

- 블록문
  - 0개 이상의 문을 중괄호로 묶은 것, 코드 블록이라고 부르기도 함
  - JS에서 블록은 하나의 실행 단위로 취급됨

```
// 블록문
{
  var foo = 10;
}

// 제어문
var x = 1;
if (x < 10) {
  x++;
}

// 함수 선언문
function sum(a, b) {
  return a + b;
}
```

연산자와 제어문

조건문

- 주어진 조건에 따라 코드 블록의 실행을 결정

```
if (조건식1) {
   // 조건식 1이 참이면 이 코드블록이 실행

} else if(조건식2) {
   // 조건식 2가 참이면 이 코드블록이 실행

} else {
   // 조건식 1과 2가 모두 거짓이면 이 코드블록이 실행

}
```

연산자와 제어문

switch 문

- 주어진 표현식을 평가하여 그 값과 일치하는 표현식을 갖는 case 문을 실행

```
switch (표현식) {

    case 표현식1:
        // 실행될 내용
        …
        break;
    case 표현식2:
        // 실행될 내용
        …
        break;
    default:
        // 실행될 내용
        …
```

연산자와 제어문

- 반복문

1. 조건식의 평가 결과가 참인 경우 코드 블록을 실행함
2. 그 후 조건식을 다시 평가하여 여전히 참인 경우 다시 코드 블록을 실행함
3. 조건식이 거짓이 될때까지 실행

- for문
  - for (변수 선언문 또는 할당문; 조건식; 증감식) {
    조건식이 참일 경우 반복 실행될 문;
    }

```
for (var i = 0; i < 2; i++) {
  console.log(i);
}
```

연산자와 제어문

- 반복문

  - while문

```
var count = 0;

// count가 3보다 작을 때까지 코드 블록을 계속 반복 실행한다.
while (count < 3) {
  console.log(count); // 0 1 2
  count++;
}
```

  - do-while 문

```
var count = 0;

do {
    console.log(count);
    count++;
} while(count < 3);
```

연산자와 제어문

break문

- 코드 블록을 탈출

continue 문

- 반복문의 코드 블록 실행을 현 지점에서 중단하고 반복문의 증감식으로 실행 흐름을 이동 시킴

```
var string = 'Hello World.';
var search = 'l';
var count = 0;

// 문자열은 유사배열이므로 for 문으로 순회할 수 있다.
for (var i = 0; i < string.length; i++) {
  // 'l'이 아니면 현 지점에서 실행을 중단하고 반복문의 증감식으로 이동한다.
  if (string[i] !== search) continue;
  count++; // continue 문이 실행되면 이 문은 실행되지 않는다.
}

console.log(count); // 3
```

04

데이터 타입

데이터 타입

- 자바스크립트의 모든 값은 데이터 타입을 가짐, 총 7개의 데이터 타입 제공

구분	데이터 타입	설명
원시 타입	숫자 타입	숫자. 정수와 실수 구분 없이 하나의 숫자 타입만 존재
	문자열 타입	문자열
	불리언(Boolean) 타입	참(true)과 거짓(false)
	undefined 타입	var 키워드로 선언된 변수에 암묵적으로 할당되는 값
	null 타입	값이 없다는 것을 의도적으로 명시할 때 사용하는 값
	심벌(symbol) 타입	ES6에서 추가된 7번째 타입
객체 타입		객체, 함수, 배열 등

데이터 타입

- 숫자 타입
  - 숫자 타입의 값은 실수(Real number)로 처리함

```
var integer = 10;    // 정수
var double = 10.12;  // 실수
var negative = -20;  // 음의 정수
```

  - 그 외 숫자값
    - Infinity: 양의 무한대
    - -Infinity: 음의 무한대
    - NaN: 산술 연산 불가(Not-a-Number)

데이터 타입

- 문자열 타입
  - 텍스트 데이터를 나타내는데 사용
  - 작은 따옴표(‘’), 큰 따옴표(“”), 백틱(``)으로 텍스트를 감싼다

```
var string;
string = '문자열'; // 작은따옴표
string = "문자열"; // 큰따옴표
string = `문자열`; // 백틱 (ES6)
```

- 템플릿 리터럴
  - 멀티라인 문자열, 표현식 삽입, 태그드 템플릿 등 편리한 기능을 제공함

```
// 에러
var str = "Hello
 world";

// 정상
var str = "Hello\n World"
```

```
// 표현식 삽입
var str1 = "Hello";
var str2 = "World"
var rs = `${str1} ${str2}`
```


https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/String

데이터 타입

- 불리언 타입
  - true, false
- undefined 타입
  - var 키워드로 선언한 변수는 할당이 이뤄지기 전 암묵적으로 undefined로 초기화됨
  - 값 초기화를 할땐 undefined 를 할당하는 대신 null을 할당하자
- null 타입
  - 변수에 값이 없다는 것을 의도적으로 명시할 때 사용
- 객체 타입
  - 함수, 객체, 배열 등

05

객체 리터럴, 함수

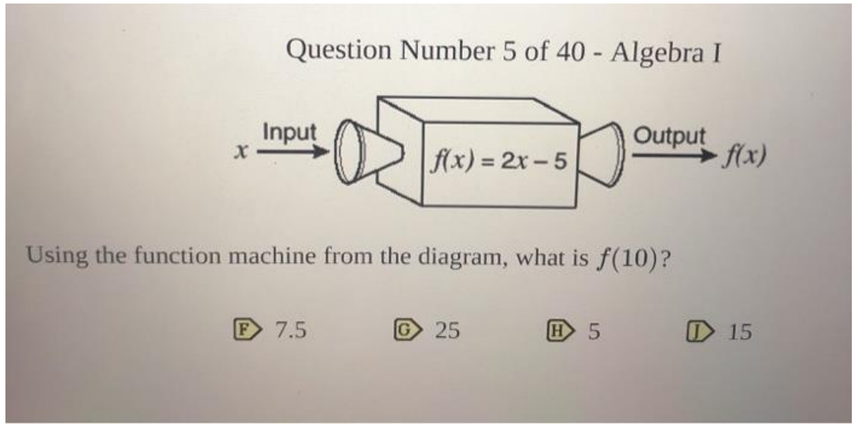
객체 리터럴, 함수

함수란?

- 수학에서 함수: 입력을 받아 출력을 내보내는 과정

함수 구성 요소

- 매개변수
- 인수
- 반환값



함수 이름 매개변수

함수 정의

```
function add(x, y) {
  return x + y;
}

// f(2, 5) = 7
add(2, 5); // 7
```

반환 값

함수 호출

인수

객체 리터럴, 함수

함수의 필요성

```
var x = 0;
var y = 0;
var result = 0;

x = 1;
y = 2;
result = x + y; // 3

x = 3;
y = 4;
result = x + y; // 7

x = 5;
y = 6;
result = x + y; // 11
```


```
function add(x, y) {
    return x + y;
}

var result = 0;

result = add(1, 2); // 3
result = add(3, 4); // 7
result = add(5, 6); // 11
```

객체 리터럴, 함수

함수 리터럴

- 함수는 객체 타입의 값이다, 함수 리터럴로 생성 가능

```
// 변수에 함수 리터럴을 할당
var f = function add(x, y) {
  return x + y;
};
```

함수는 객체다.
하지만 일반 객체와 차이점은
일반 객체는 호출할 수 없지만 함수는 호출할 수 있다.

- 구성 요소
  - function 키워드
  - 함수 이름
  - 매개변수 목록
  - 함수 몸체

객체 리터럴, 함수

함수 정의

함수 정의 방식	예시

함수 선언문	function add(x, y) {
	  return x + y;
	};

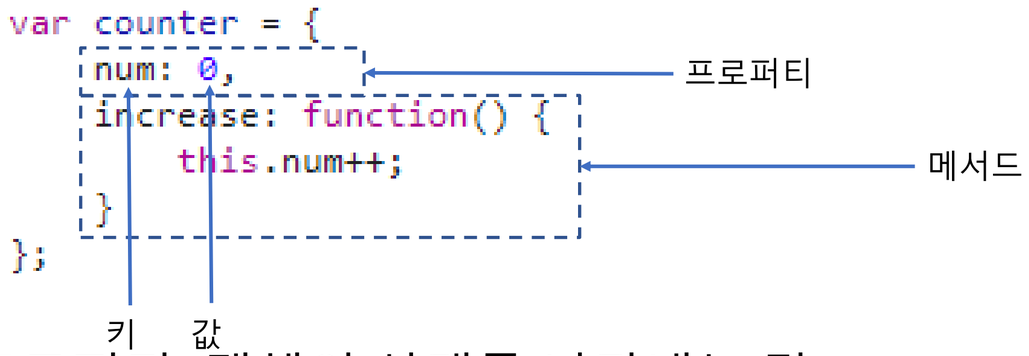
함수 표현식	var add = function add(x, y) {
	  return x + y;
	};

Function 생성자 함수	var add = new Function('x', 'y', 'return x + y');
화살표 함수(ES6)	var add = (x, y) => x + y;

객체 리터럴, 함수

객체란?

- 객체란 0개 이상의 프로퍼티로 구성된 집합이며, 프로퍼티는 키와 값으로 구성
  - 프로퍼티 값이 함수일 경우 메서드라고 부름



- 프로퍼티: 객체의 상태를 나타내는 값(data)
- 메서드: 프로퍼티(상태 데이터)를 참조하고 조작할 수 있는 동작

객체 리터럴, 함수

객체 생성 방법

- 객체 리터럴 (객체를 생성하기 위한 표기법)
- Object 생성자 함수
- 생성자 함수
- Object.create 메서드
- 클래스 (ES6)

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

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

객체 리터럴, 함수

프로퍼티 접근

- 마침표 표기법(.)
- 대괄호 표기법([...]) 로 접근 가능

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

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

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

프로퍼티 동적 생성 및 갱신

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

// person 객체에는 age 프로퍼티가 존재하지 않는다.
// 따라서 person 객체에 age 프로퍼티가 동적으로 생성되고 값이 할당된다.
// person 객체에 name 프로퍼티가 존재하므로 name 프로퍼티의 값이 갱신된다.
person.age = 20;
person.name = "Kim";

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

객체 리터럴, 함수

프로퍼티 삭제
- delete 연산자로 프로퍼티 삭제할 수 있음

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

// 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: "Park"}
```

객체 리터럴, 함수

ES6에서 추가된 객체 리터럴 확장 기능

- 프로퍼티 축약 표현
- 메서드 축약 표현

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

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

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

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

// 프로퍼티 축약 표현
const obj = { x, y };

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

```
// ES5
var obj = {
  name: 'Park',
  sayHi: function() {
    console.log('Hi! ' + this.name);
  }
};

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

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

클래스

- Class는 객체를 생성하기 위한 템플릿
  - class 표현식, class 선언 두 가지 정의 가능

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

```
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

- new 연산자로 호출해야 하며 상속을 지원하는 extends, super 키워드 제공

```
const square = new Rectangle(10, 10);
```

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}
```

```
class Lion extends Cat {
  speak() {
    super.speak();
    console.log(`${this.name} roars.`);
  }
}
```

클래스

- constructor 메서드를 통해 프로퍼티 추가 가능
- 클래스에 특정 메서드 추가 기능 지원

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }

  // 메서드
  calcArea() {
    console.log(this.height * this.width);
  }
}

const square = new Rectangle(10, 10);

square.calcArea(); // 100
```

06

문자열, 배열

문자열

- 문자열 타입은 텍스트 데이터를 나타내는 데 사용한다.
  - 표기방법: 작은 따옴표(‘’), 큰 따옴표(“”), 백틱(``)

```
// 문자열 타입
var string;
string = '문자열'; // 작은따옴표
string = "문자열"; // 큰따옴표
string = `문자열`; // 백틱 (ES6)
```

- 문자열은 원시 타입이며 변경이 불가능, 그리고 유사 배열 객체이다

```
var str = 'string';

// 문자열은 유사 배열이므로 배열과 유사하게 인덱스를 사용해 각 문자에 접근할 수 있다.
console.log(str[0]); // s

// 원시 값인 문자열이 객체처럼 동작한다.
console.log(str.length); // 6
console.log(str.toUpperCase()); // STRING
```

문자열

- 문자열 메서드
  - 알아두면 좋은 메서드 목록
    - indexOf
    - search
    - includes
    - startsWith
    - charAt
    - substring
    - slice
    - toUpperCase
    - toLowerCase
    - trim
    - replace
    - split

참고: https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/String

배열

- 배열은 여러 개의 값을 순차적으로 나열한 자료구조이다.

```
const arr = ['apple', 'banana', 'orange'];
```

- 배열 생성
  - 배열 리터럴

```
const arr = [1, 2, 3];
console.log(arr.length); // 3
```

  - Array 생성자 함수

```
const arr = new Array(10);

console.log(arr); // [empty × 10]
console.log(arr.length); // 10
```

배열

- 배열 요소의 참조
  - 대괄호([]) 표기법 사용

```
const arr = [1, 2];

// 인덱스가 0인 요소를 참조
console.log(arr[0]); // 1
// 인덱스가 1인 요소를 참조
console.log(arr[1]); // 2
```

- 배열 요소의 추가와 갱신

```
const arr = [0];

// 배열 요소의 추가
arr[1] = 1;

console.log(arr); // [0, 1]
console.log(arr.length); // 2
```

```
// 요소값의 갱신
arr[1] = 10;

console.log(arr); // [0, 10, empty × 98, 100]
```

배열

- 배열 요소의 삭제

```
const arr = [1, 2, 3];

// 배열 요소의 삭제
delete arr[1];
console.log(arr); // [1, empty, 3]

// length 프로퍼티에 영향을 주지 않는다. 즉, 희소 배열이 된다.
console.log(arr.length); // 3
```

배열

- 배열 메서드
  - 알아두면 좋은 메서드 목록
    - push, pop
    - indexOf
    - forEach
    - shift, unshift
    - splice
    - slice
    - includes
    - map
    - filter
    - reduce
    - some, every

참고: https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Array

07

let, const, var

let, const, var

var 키워드로 선언한 변수의 문제점

- 변수 중복 선언 허용
- 함수 레벨 스코프: 함수 코드 블록만 지역 스코프로 인정
- 변수 호이스팅

var 키워드의 단점을 보완하기 위해 ES6 에선 let, const 키워드를 도입함

let, const, var

let 키워드 특징

- 변수 중복 선언 금지
- 블록 레벨 스코프 가능
- 변수 호이스팅 발생하지 않도록 함

let, const, var

let 키워드 특징

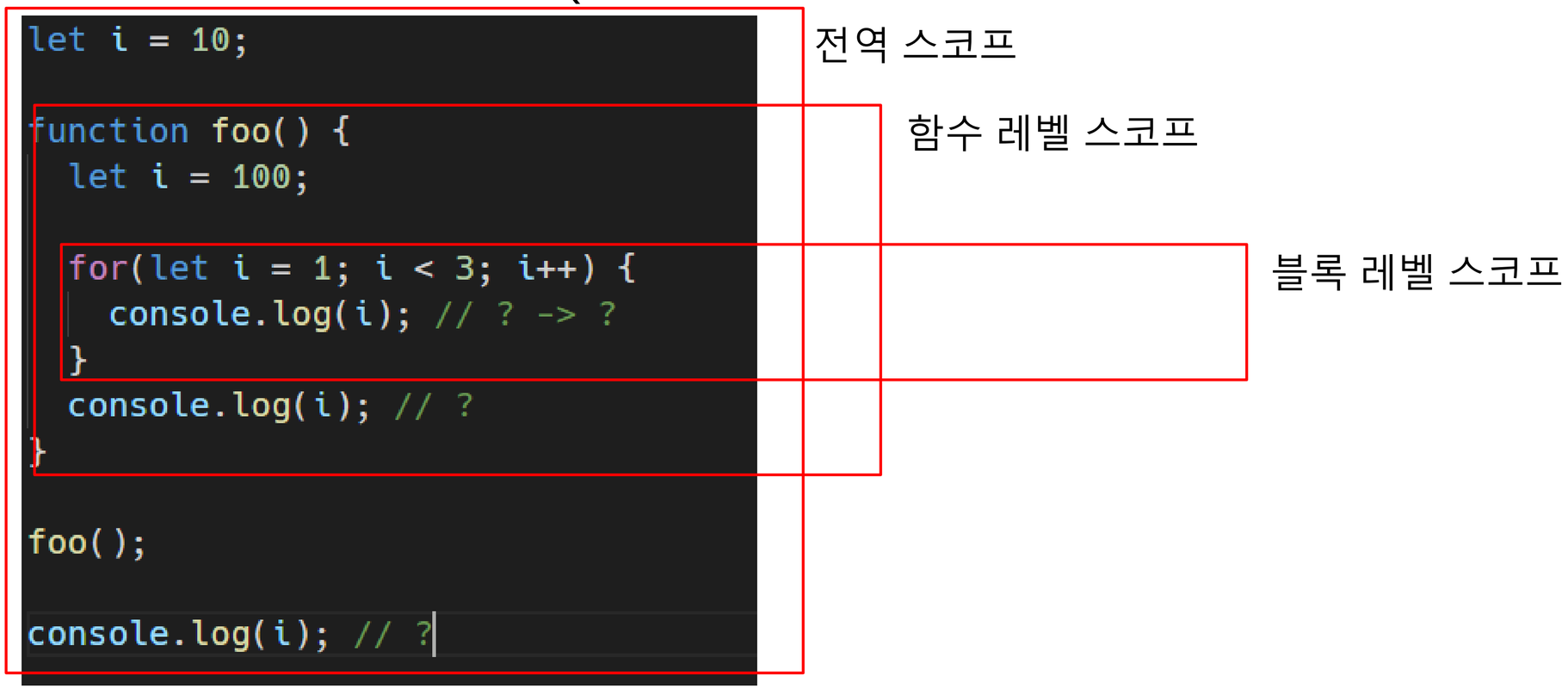
- 블록 레벨 스코프 가능(코드 블록으로 감싸진 부분을 지역 스코프로 인정함)

```
let foo = 1; // 전역 변수

{
  let foo = 2; // 지역 변수
  let bar = 3; // 지역 변수
}

console.log(foo); // 1
console.log(bar); // ReferenceError: bar is not defined
```

let, const, var

let 키워드 특징

- 블록 레벨 스코프 가능(코드 블록으로 감싸진 부분을 지역 스코프로 인정함)



let, const, var

let 키워드 특징

- 변수 호이스팅 발생하지 않도록 함

```
// 런타임 이전에 선언 단계가 실행된다. 아직 변수가 초기화되지 않았다.
// 초기화 이전의 일시적 사각 지대에서는 변수를 참조할 수 없다.
console.log(foo); // ReferenceError: foo is not defined

let foo; // 변수 선언문에서 초기화 단계가 실행된다.
console.log(foo); // undefined

foo = 1; // 할당문에서 할당 단계가 실행된다.
console.log(foo); // 1
```

let, const, var

let 키워드 특징

- 변수 호이스팅 발생하지 않도록 함
- TDZ: 스코프의 시작 지점부터 초기화 시작 지점까지 변수를 참조할 수 없는 구간

var 키워드로 선언한 변수 생명주기

let 키워드로 선언한 변수 생명주기

let, const, var

const 키워드

- 선언과 초기화: 반드시 선언과 동시에 초기화 해야 함
  - 블록 레벨 스코프를 가짐
  - 변수 호이스팅 동작하지 않음
- 재할당 금지
- 상수: 재할당이 금지된 변수를 의미
  - 상태 유지와 가독성, 유지보수의 편의를 위해 적극적으로 사용해야 함
- const 키워드와 객체
  - 원시 값을 할당한 경우 값 변경 X, 하지만 객체를 할당한 경우 값 변경 O

let, const, var

```
// 세전 가격
let preTaxPrice = 100;

// 세후 가격
// 0.1의 의미를 명확히 알기 어렵기 때문에 가독성이 좋지 않다.
let afterTaxPrice = preTaxPrice + (preTaxPrice * 0.1);

console.log(afterTaxPrice); // 110
```

```
// 세율을 의미하는 0.1은 변경할 수 없는 상수로서 사용될 값이다.
// 변수 이름을 대문자로 선언해 상수임을 명확히 나타낸다.
const TAX_RATE = 0.1;

// 세전 가격
let preTaxPrice = 100;

// 세후 가격
let afterTaxPrice = preTaxPrice + (preTaxPrice * TAX_RATE);

console.log(afterTaxPrice); // 110
```

let, const, var

var vs let vs const

- ES6를 사용한다면 var 키워드 사용 X
- 재할당이 필요한 경우에 한정해 let 키워드를 사용한다. 이때 변수의 스코프는 최대한 좁게 만든다.
- 변경이 발생하지 않고 읽기 전용으로 사용하는(재할당이 필요 없는 상수) 원시 값과 객체에는 const 키워드를 사용한다. const 키워드는 재할당을 금지하므로 var, let 키워드보다 안전하다

참고

- 모던 자바스크립트 Deep Dive
- 짐코딩 CODING GYM
- 코딩애플
- https://ko.javascript.info/dom-navigation