

Javascript

1. 기본 문법

1. 변수 - `var`, `let`, `const`

- Javascript에서는 파이썬과 달리 기본적으로 변수를 선언하여야 한다.
- ES6이전에는 `var` 를 이용하였으나 우리는 `const` 를 중심으로 사용하고, `let` 이 필요한 경우만 사용할 것이다.
- `var` vs `let`, `const`
 - `var` 는 function 스코프이며, `let`, `const` 는 block 스코프이다.

```
// 블록 스코프 - let, const
for(var j=0; j < 1; j++) {
  console.log(j)
}
console.log(j) // 1

for(let i=0; i<1; i++) {
  console.log(i)
}
console.log(i) // Uncaught ReferenceError: i is not defined

// 함수 스코프 - var
const myFunction = function () {
  for(var k=0; k < 1; k++) {
    console.log(k)
  }
  console.log(k) // 1
}
myFunction()
console.log(k) // Uncaught ReferenceError: k is not defined
```

- `var` 는 재선언이 가능하며, `let`, `const` 는 재선언이 불가능하다.

```
var a = 1
var a = 2

let b = 1
let b = 3

const c = 1
const c = 4
```

- `let` 은 재할당이 가능하지만, `const` 는 불가능하다.

```
let d = 3
d = 5

const e = 5
e = 3

// 따라서, const는 선언시 반드시 할당이 필요하다.
const f

let g
g = 3
```

- 위의 변수 선언 키워드를 사용하지 않으면 함수/블록안에서 선언되더라도 무조건 전역변수로 취급된다.

```
function myFunction1 () {
  for( p=0; p < 1; p++) {
    console.log(p)
  }
  console.log(p)
}
myFunction1()
console.log(p) // 1
```

따라서, 무조건 변수 선언 키워드를 작성하고 `const` 를 기본적으로 쓰고, `let` 을 사용해야 하는 상황에 한하여 변경하자.

2. 자료형

자료형은 구분만 하였습니다. 연산, string 조작 등은 수업 내용을 바탕으로 따로 정리하세요.

- 기본 자료형 (primitive)
 - Boolean (true/false)
 - null
 - undefined
 - number (`Infinity`, 양수, 음수 등)
 - string

- Object
예) 배열, object

3. 조건/반복

if / else if / else

while / for / for of

위의 문법을 직접 정리하세요.

- == VS ===
 - 요약



4. 배열 - Object

```
const numbers = [1, 2, 3, 4];
```

```

numbers[0]; // 1
numbers[-1] // undefined => 정확한 양의 정수 index 만 가능
numbers.length; // 4

numbers.reverse(); // [4,3,2,1]
numbers // [4,3,2,1]

numbers.push('a') // 5 (new length)
numbers; // [1,2,3,4,'a']

numbers.pop() // 'a'
numbers; // [1,2,3,4]

numbers.unshift('a'); // 6 (new length)
numbers; // ['a',1,2,3,4]

numbers.shift(); // 'a'
nubers; // [1,2,3,4]

/* 복사본 or 다른 값 return */
numbers.includes(1) // true
numbers.includes(0) // false

numbers.push('a') // 5
numbers.push('a') // 6
numbers // [1,2,3,4,'a','a']
numbers.indexOf('a') // 4 => 처음 찾은 요소의 index!
numbers.indexOf('b') // -1 => 없으면 -1

numbers.join(); // '1,2,3,4,a,a'
numbers.join(''); // 1234aa
numbers.join('-'); // '1-2-3-4-a-a'

numbers; // [1,2,3,4,'a','a']

typeof numbers // object

```

5. Object

Key - Value로 이루어진 데이터 구조이다.

```
const endGame = {
  title: '어벤져스: 엔드게임',
  'my-lovers': [
    {name: '아이언맨', actor: '로다주'},
    {name: '헐크', actor: '마크 러팔로'}
  ]
}

endGame.title
endGame['title']
endGame['my-lovers'][0].name
```

- Object Literal (ES6+)

```
const comics = {
  'DC': ['Aquaman', 'SHAZAM'],
  'Marvel': ['Captain Marvel', 'Avengers']
};

const magazines = null;

const bookShop = {
  comics,
  magazines,
};
```

- method

파이썬과 달리 별도의 문법이 있는 것이 아니라 오브젝트의 value에 함수를 할당한다.

```
const me = {
  name: 'Kim',
  greeting: function(message) {
    return `${this.name} : ${message}`
  }
}

me.greeting('hi') // Kim : hi
me.name = 'John'
me.greeting('hello') // John : hello
```

앞선 변수/object와 같이 함수도 ES6+ 부터 아래와 같이 선언 가능하다.

```
const greeting = function(message) {
  return `${this.name} : ${message}`
}

const you = {
  name: 'Yu',
  greeting
}
you.greeting('hi') // Yu : hi
you.name = 'Jane'
you.greeting('hello') // Jane : hello
```

- 메서드 정의시, `arrow function`을 사용하지 않는다.

6. JSON

key - value 형태의 자료구조를 JS Object 와 유사한 모습으로 표현하는 표기법. 모습만 비슷할 뿐이고 실제로 Object 처럼 사용하려면 다른 언어들처럼 JS 에서도 Parsing(구문 분석)작업이 필요하다.

[MDN 문서 참조](#)

7. 함수

- 함수 선언식

```
function myFunc1 (name) {
  console.log('happy hacking')
  console.log(`welcome, ${name}`)
}
```

- 함수 표현식

```
const myFunc2 = function (name) {
  console.log('happy hacking')
  console.log(`welcome, ${name}`)
}

typeof myFunc1 // function
typeof myFunc2 // function
```

- 화살표 함수 (ES6+)

주의, 화살표 함수의 경우 `function` 키워드로 정의한 위의 함수와 100% 동일한 것이 아님.

```
const myFunc3 = (name) => {  
  console.log('happy hacking')  
  console.log(`welcome, ${name}`)  
}
```

syntactic sugar 와 관련된 내용은 직접 정리하세요.

return문이 한 줄일 때, 인자가 없을 때, 인자가 하나일 때 등

8. Array Helper Methods

아래의 Array Helper Methods를 정리하세요.

- forEach
- map
- filter
- find
- reduce : 도전

2. Event Listener

- **click** - 마우스버튼을 클릭하고 버튼에서 손가락을 떼면 발생한다.
- **mouseover** - 마우스를 HTML요소 위에 올리면 발생한다.
- **mouseout** - 마우스가 HTML요소 밖으로 벗어날 때 발생한다.
- **mousemove** - 마우스가 움직일때마다 발생한다. 마우스커서의 현재 위치를 계속 기록하는 것에 사용할 수 있다.
- **keypress** - 키를 누르는 순간에 발생하고 키를 누르고 있는 동안 계속해서 발생한다.
- **keydown** - 키를 누를 때 발생한다.
- **keyup** - 키를 눌렀다가 떼는 순간에 발생한다.
- **load** - 웹페이지에서 사용할 모든 파일의 다운로드가 완료되었을때 발생한다.
- **scroll** - 스크롤바를 드래그하거나 키보드(up, down)를 사용하거나 마우스 휠을 사용해서 웹페이지를 스크롤 할 때 발생한다. 페이지에 스크롤바가 없다면 이벤트는 발생하지 않다.
- **change** - 폼 필드의 상태가 변경되었을 때 발생한다. 라디오 버튼을 클릭하거나 셀렉트 박스에서 값을 선택하는 경우를 예로 들수 있다.
- **input** - input 또는 textarea 요소의 값이 변경되었을 때
- **submit** - form을 submit 할 때

1. DOM selector

- `querySelector()`
- `querySelectorAll()`

2. 이벤트 리스너 등록

- 특정한 DOM element (무엇을) 를 어떠한 행동을 했을 때 (언제) , 어떻게 한다.

```
// 1. 무엇을
const button = document.querySelector('#some-button')

// 2. 언제 => 버튼을 '클릭' 하면
button.addEventListener('click', function (event) {
  const area = document.querySelector('#my')
  // 3. 어떻게 => 뽕
  area.innerHTML = '<h1>뽕</h1>'
})
```

- 이벤트 리스너에서의 콜백함수에는 arrow function 을 사용하지 않는다.

3. Axios

axios는 XHR(XMLHttpRequest)를 보내주고 그 결과를 promise 객체로 반환해주는 라이브러리이다.

1. axios

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
</body>
</html>
```

2. 기본 활용법

```

axios.get('/posts/')
  .then(function(response) {
    console.log(response)
  })

const data = {title: '제목', content: '내용'}
axios.post('/posts/', data)
  .then(function(response) {
    console.log(response)
  })

```

3. 개념 확인하기

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
  <script>
    const getDog = () => {
      axios.get('https://dog.ceo/api/breeds/image/random')
        .then(response => {
          console.log(response.data)
          console.log('데이터 도착했다!')
        })
    };
    console.log('함수 호출 시작한다!')
    getDog()
    console.log('함수 호출 끝났다!')
  </script>
</body>
</html>

```

해당 파일의 콘솔을 확인하면, 아래와 같이 출력된다.

```

함수 호출 시작한다!
함수 호출 끝났다!
데이터 도착했다!

```

우리가 기존에 작성했던 파이썬으로 따져보면, 아래와 같이 프린트 되었을 것이다.

함수 호출 시작한다!
데이터 도착했다!
함수 호출 끝났다!

이는 파이썬(`blocking`) 과 자바스크립트(`non-blocking`)의 가장 큰 차이점이다.

`axios` 는 `promise` 객체를 반환하여 `.then` 을 통해 해당하는 작업이(`axios` 요청작업) 완료된(`resolve`) 경우 실행 될 로직을 구현할 수 있다. (`.catch` 에서는 `reject`된 결과를 받아서 처리 할 수 있다.) - 콜백지옥을 해결하기 위한 약속

더 자세한, 자바스크립트가 동작하는 방식을 보고 싶다면 다음의 [링크](#) 를 확인하기 바란다.

브라우저는 싱글쓰레드에서 이벤트 기반(event driven) 방식으로 실행된다.

call stack : 함수가 호출되면 순차적으로 call stack에 쌓이고 순차적으로 실행된다. task가 종료하기 전까지는 다른 task를 수행할 수 없다.

callback queue : 비동기처리 함수의 콜백, 타이머(`setTimeout`), 이벤트핸들러 등이 기록되는 곳으로 이벤트 루프에 의해 특정시점에 콜 스택으로 이동되어 실행 됨.

event loop : 콜 스택과 콜백 큐에 작업이 (실행될 함수) 있는지 확인하며 작업을 실행한다.

