JS 정리

Vanilla JavaScrip: 웹 브라우저에서 바로 실행할 수 있는 JS문법, 즉 순수한 JS 라는 의미

```
Node.js 설치

$ node -v
$ npm -v

JavaScript 파일 실행해 보기
$ node hello.js
```

JS 기초 코드 작성법

들여쓰기 2칸

```
if (조건) {

}

for (one in many) or (one of many) {

}
```

JS에도 Python의 코드 스타일 가이드인 PEP8과 같이 여러 코드 스타일 가이드가 있지만 수업에선 Airbnb Style Guide.

단일 주석 : //

여러 줄 주석 : /* */

변수와 식별자

식별자 정의와 특징

- 카멜 케이스
    - 변수, 객체, 함수에 사용

```
// 변수
let dog
let variableName

// 객체
const userInfo = { name: 'Tom', age: 20}

// 함수
function add() {}
function getName() {}
```

- 파스칼 케이스
  - 클래스, 생성자에 사용

```
// 클래스
class User{
    constructor(options) {
        this.name = options.name
    }
}

// 생성자 함수
function User(options) {
    this.name = options.name
}
```

- 대문자 스네이크 케이스
  - 상수(constants)에 사용
  - 상수: 개발자의 의도와 상관없이 변경될 가능성이 없는 값을 의미

```
// 값이 바꾸지 않을 상수
const API_KEY = 'my-key'
const PI = Math.PI

// 재할당이 일어나지 않는 변수
const NUMBERS = [1,2,3]
```

함수 스코프(function scope)

- 함수의 중괄호 내부를 가리킴
- 함수 스코프를 가지는 변수는 함수 바깥에서 접근 불가능

변수 선언 키워드 정리

- Airbnb 스타일 가이드에서는 기본적으로 const 사용을 권장
  - 재할당해야 하는 경우에만 let을 사용

데이터 타입

- Number
  - 정수 또는 실수형 숫자를 표현하는 자료형
  - NaN
    - Not-A-Number(숫자가 아님)를 나타냄
    - Number.isNaN()의 경우 주어진 값의 유형이 Number이고 NaN이면 true, 아니면 false를 반환
- Sting
  - 문자열을 표현하는 자료형
  - 곱셈, 나눗셈, 뺄셈은 안되지만 덧셈을 통해 문자열을 붙일 수 있다.
  - 개행 문자 \n 을 넣어 다음 줄로 넘어갈 수 있다.
  - 백틱(`)사이에 문자를 입력하고 가변 변수를 넣고 싶을 경우 ${{변수명}} 해주면 파이썬의 f-string과 유사.
- null
  - 변수의 `값이 없음을 의도적으로 표현할 때` 사용하는 데이터 타입
- nudefined
  - 값이 정의되어 있지 않음을 표현하는 값
  - 변수 선언 이후 직접 값을 할당하지 않으면 자동으로 할당됨
- null vs undefined
  - 차이점은 typeof 연산자를 통해 확인을 해보면
    - null: object
    - nudefined: undefined

연산자

- 할당 연산자

```
let c = 0

c += 10
c -= 3
c*= 10
c++
c--
```

- 비교 연산자

```
3 > 2
'A' < 'B'
'Z' < 'a'
```

- 동등 연산자(==)
  - 두 피연산자가 같은 값으로 평가되는지 비교 후 boolean 값을 반환
  - 비교할 때 `암묵적 타입 변환`을 통해 타입을 일치시킨 후 같은 값인지 비교\
- 일치 연산자(===)

    - 두 피연산자의 값과 타입이 모두 같은 경우 true를 반환
    - 엄격한 비교가 이뤄지며 `암묵적 타입 변환이 발생하지 않음`
- 논리 연산자
    - &&
    - ||
    - !
- 삼항 연산자
    - 3개의 피연산자를 사용하여 조건에 따라 값을 반환하는 연산자
    - 가장 앞의 조건식이 참이면 : 앞의 값이 반환되며, 그 반대일 경우: 뒤의 값이 반환되는 연산자.

```
true ? 1:2 //1
false ? 1:2 //2

const result = Math.PI > 4 ? 'Yep' : 'Nope' // Nope
```

조건문

조건문의 종류

- if

```
const name = 'manager'

if (name === 'admin') {
    consol.log('관리자님 환영합니다.')
}
else if (name === 'manager') {
    consol.log('매니저님 환영합니다.')
}
else {
    consol.log(`${name}님 환영합니다.`)
}
```

- switch

```
switch(expression) {
    case 'first value': {
        // do something
        [break]
    }
    case 'second value': {
        // do something
        [break]
    }
    [default: {
     // do something
     }]
}
```

```
// break문이 없는 경우 break문을 만나거나 default문을 실행할 때까지 다음 조건문 실행
```

반복문

반복문의 종류

- while
  - 조건문이 참이기만 하면 문장을 계속해서 수행

```
while (조건문) {
    // do something
}

let i = 0

while (i<6) {
    console.log(i)
    i += 1
}
```

- for
  - 특정한 조건이 거짓으로 판별될 때까지 반복

```
for ([초기문];[조건문];[증감문]) {
    // do something
}

for (let i=0; i<6; i++) {
    console.log(i)
}
```

- for ...in
  - 객체(object)의 속성을 순회할 때 사용
  - 배열도 순회 가능하지만 인덱스 순으로 순회한다는 보장이 없으므로 권장하지 않음

```
for (variable in object) {
    statements
}

const fruits = {a:'apple', b:'banana'}

for (const key in fruits) {
    console.log(key) // a, b
    console.log(fruits[key]) // apple, banna
}
```

- for ...of
  - 반복 가능한 객체를 순회할 때 사용
  - 반복 가능한(iterable)객체의 종류: Array, Set, String 등

```
for (variable of object) {
    statements
}

const numbers = [0, 1, 2, 3]

for (const number of numbers) {
    console.log(number) // 0, 1, 2, 3
}
```

- for ...in 과 for ...of의 차이
  - for ...in은 `속성 이름`을 통해 반복 (객체 순회 적합)
  - for ...of는 `속성 값`을 통해 반복 (iterable 순회 적합)

```
// Array

const numbers = [10, 20, 30]

// for ...in
for (const number in numbers) {
    console.log(number) // 0 1 2
}

// for ...of
for (const number of numbers) {
    console.log(number) // 10 20 30
}
```

```
// Object


const capitals = {
    korea: '서울',
    france: '파리',
    japan: '도쿄'
}

// for ...in
for (const capital in capitals) {
    console.log(capital) // korea france japan
}

// for ...of
for (const capital of capitals) {
    console.log(capital) // TypeError: capitals is not iterable (객체를 접근하는 타입도 아니고 iterable 하지 못해서 에러)
}
```

- const [참고]
  - for문
    - for (let i=0; i<6; i++) {...} 의 경우에는 최초 정의한 i를 재할당 하면서 사용하기 때문에 const를 사용시 에러
  - for ...in, for ...of
    - 재할당이 아닌, 매 반복 시 해당 변수를 새로 정의하여 사용하므로 에러가 발생하지 않음

함수

JS에서 함수를 정의하는 주로 사용하는 2가지 방법

- 함수 선언식 (function declaration)

```
// 일반적인 프로그래밍 언어의 함수 정의 방식
function 함수명() {
    // do something
}

function add(num1,num2) {
    return num1 + num2
}
add(2,7)  // 9
```

- 함수 표현식 (function expression)

```
// 표현식 내에서 함수를 정의하는 방식
// 함수 표현식은 함수의 이름을 생략한 익명 함수로 정의 가능

변수키워드 함수명 = function() {
    // do something
}

const sub = function(num1,num2) {
    return num1 + num2
}

위의 코드와 아래의 코드들은 같다.

const sub = (num1,num2) =>{
    return num1 + num2
}

const sub = (num1,num2) => num1 + num2

sub(2,7)  // 9
```

```
// 표현식에서는 함수 이름을 명시하는 것도 가능하다
// 다만 이 경우 함수 이름은 호출에 사용되지 못하고 디버깅 용도로 사용됨

const mySub = function namedSub(num1,num2) {
    return num1 - num2
}

mySub(1, 2) // -1
namedSub(1,2) // ReferenceError: namedSub is not defined
```

- 기본 인자
  - 인자 작성 시 '=' 문자 뒤 기본 인자 선언 가능하다.
  - ex) const greeting = function(name='Anoymous')

- 함수의 정의
  - 매개변수보다 인자의 개수가 많을 경우

```
const noArgs = function() {
    return 0
}
noArgs(1,2,3) // 0

const twoArgs = function(arg1,arg2) {
    return [arg1,arg2]
}
twoArgs(1,2,3) // [1,2]
```

  - 매개변수보다 인자의 개수가 적을 경우

```
const threeArgs = function (arg1, arg2, arg3) {
    return [arg1,arg2,arg3]
}

threeArgs()    // [undefined,undefined,undefined]
threeArgs(1)   // [1,undefined,undefined]
threeArgs(1,2) // [1,2,undefined]
```

- Spread syntax
  - 전개 구문
  - 전개 구문을 사용하면 배열이나 문자열과 같이 반복 가능한 객체를 배열의 경우는 요소,

    함수의 경우는 인자로 확장할 수 있음

    1. 배열과의 사용
    2. 함수와의 사용 (Rest parameters)

```
let parts = ['shoulders', 'knees']
let lyrics = ['head', ...parts, 'and', 'toes']
// ['hard', 'shoulders', 'knees', 'and', 'toes']
```

선언식과 표현식

```
// 함수 표현식
const add = function (args) {}

// 함수 선언식
function sub(args) {}

console.log(typeof add) // function
console.log(typeof sub) // function
```

- 호이스팅
  - 표현식

```
sub(2,7)
// Uncaught ReferenceError: Cannot access 'sub' before initialization
// 함수 표현식으로 선언한 함수는 함수 정의 전에 호출 시 에러 발생

const sub = function(num1,num2){
    return num1+num2
}
```

  - 선언식

```
add(2, 7) // 9

function add (num1,num2) {
    return num1+num2
}
```

Arrow Function

- 화살표 함수
  - 함수를 비교적 간결하게 정의할 수 있는 문법
  - function 키워드와 중괄호를 이용한 구문을 짧게 사용하기 위해 탄생
    1. function 키워드 생략가능
    2. 함수의 매개변수가 하나 뿐이라면 매개변수의 () 생략 가능
    3. 함수의 내용이 한 줄이라면 {} 와 return 또한 생략 가능
  - 화살표 함수는 항상 익명 함수

        - === 함수 표현식에서만 사용가능

- 즉시 실행 함수(IIFE, Immediately Invoked Function Expression)
    - 선언과 동시에 실행되는 함수
    - 함수의 선언 끝에 () 를 추가하여 선언되자 마자 실행하는 형태
    - ()에 값을 넣어 인자로 넘겨줄 수 있음
    - 즉시 실행 함수는 선언과 동시에 실행되기 때문에 같은 함수를 다시 호출할 수 없음
    - 이러한 특징을 살려 초기화 부분에 많이 사용
    - 일회성 함수이므로 익명하무로 사용하는 것이 일반적

```
(function(num) {return num**3}(2)) // 8

(num => num**3)(2) //8
```

Array와 Object

- JavaScript의 데이터 타입 중 참조 타입에 해당 하는 타입은 Array와 Object이며 객체라고도 말한다.
- 객체는 속성들의 모음(collection)

- **배열(Array)**
    - 키와 속성들을 담고 있는 참조 타입의 객체
    - 순서를 보장하는 특징이 있음
    - 배열의 길이는 array.length 형태로 접근 가능
        - 배열의 마지막 원소는 array.length -1 로 접근
    - 배열 메서드 기초
        - reverse
            - 원본 배열의 요소들의 순서를 반대로 정렬

```
const number = [1, 2, 3, 4, 5]
numbers.reverse()
console.log(numbers) // [5, 4, 3, 2, 1]
```

        - push & pop
            - 배열의 가장 뒤에 요소를 추가(push) 또는 제거(pop)

```
const numbers = [1, 2, 3, 4, 5]

numbers.push(100)
console.log(numbers) // [1, 2, 3, 4, 5, 100]

numbers.pop()
console.log(numbers) // [1, 2, 3, 4, 5]
```

- unshift & shift
  - 배열의 가장 앞에 요소를 추가 또는 제거

- includes
  - 배열에 특정 값이 존재하는지 판별 후 참/거짓 반환

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers.includes(1)) // true
console.log(numbers.includes(100)) // false
```

- index0f
  - 배열에 특정 값이 존재하는지 판별 후 인덱스 반환
  - 요소가 없을 경우 -1 반환

```
const numbers = [1, 2, 3, 4, 5]
let result

result = numbers.indexOf(3) // 2
console.log(result)

result = numbers.indexOf(100) // -1
console.log(result)
```

- join
  - 배열의 모든 요소를 구분자를 이용하여 연결
  - 구분자 생략 시 쉼표 기준

```
const numbers = [1, 2, 3, 4, 5]
let result

result = numbers.join() // 1,2,3,4,5
result = numbers.join('') // 12345
result = numbers.join(' ') // 1 2 3 4 5
result = numbers.join('-') // 1-2-3-4-5
```

Array Helper Methods

- 배열을 순회하며 특정 로지글 수행하는 메서드
- 메서드 호출 시 인자로 callback 함수를 받는 것이 특징
  - callback 함수: 어떤 함수의 내부에서 실행될 목적으로 인자로 넘겨받는 함수

- forEach (반환 값 없음)
  - 배열의 각 요소에 대해 콜백 함수를 한 번씩 실행

```
array.forEach(callback(element[, index[, array]]))

// 인자로 주어지는 함수(콜백 함수)를 배열의 각 요소에 대해 한 번씩 실행
// // 콜백 함수는 3가지 매개변수로 구성
// // // 1. element: 배열의 요소
// // // 2. index: 배열의 요소읜 인덱스
// // // 3. array: 배열 자체
// 반환 값(return)없음

// 1. 일단 사용해보기
const colors = ['red','blue','green']

printFunc = function(color) {
    console.log(color)
}
colors.forEach(printFunc)

// 2. 함수 정의를 인자로 넣어보기
colors.forEach(function(color) {
    consol.log(color)
})

// 3. 화살표 함수 적용하기
colors.forEach((color) => {
    return console.log(color)
})
```

- map
  - 콜백 함수의 반환 값을 요소로 하는 새로운 배열 반환
  - forEach + return 이라고 생가가기

```
// 1. 일단 사용해보기
const numbers = [1,2,3]

// 함수 정의 (표현식)
const doubleFunc = function (number) {
    return number + 2
}

// 함수를 다른 함수의 인자로 넣기(콜백 함수)
const doubleNumbers = numbers.map(doubleFunc)
```

```
console.log(doubleNumbers) //[2,4,6]

// 2. 함수 정의를 인자로 넣어보기
const doubleNumbers = numbers.map(function (number) {
    return number *2
})

// 3. 화살표 함수 적용하기
const doubleNumbers = numbers.map((number) => { return number *2})
```

- filter
  - 콜백 함수의 반환 값이 참인 요소들만 모아서 새로운 배열을 반환
  - 기존 배열의 요소들을 필터링할 때 유용

```
const products = [
    {name: 'cucumber', type: 'vegetable'},
    {name: 'banana', type: 'fruit'},
    {name: 'carrot', type: 'vegetable'},
    {name: 'apple', type: 'fruit'},
]

const fruits = products.filter((product) => {
    return product.type === 'fruit'
})
console.log(fruits)
```

- reduce
  - 콜백 함수의 반환 값들을 하나의 값(acc)에 누적 후 반환
  - acc = 이전 callback 함수의 반환 값이 누적되는 변수
  - initialValue(optional) = 최초 callback 함수 호출 시 acc에 할당되는 값,
    default 값은 배열의 첫 번째 값

```
const tests = [90, 90, 80, 77]

const sm = tests.reduce(function(total,x) {
return total+x}, 0)

const sum = tests.reduce(total,x)=> total+x,0)
```

- find
  - 콜백 함수의 반환 값이 참이면 해당 요소를 반환
  - 콜백 함수의 반환 값이 true면, 조건을 만족하는 첫번째 요소를 반환
  - 찾는 값이 배열에 없으면 undefined 반환

```
const avengers = [
    {name: 'Tony Stark', age:45},
    {name: 'Steve Rogers', age:32},
    {name: 'Thor', age:40},
]

const avenger = avengers.find((avenger) => {
    return avenger.name === 'Tony Stark'
})
```

- some
  - 배열의 요소 중 하나라도 판별 함수를 통과하면 참을 반환
  - 모든 요소가 통과하지 못하면 거짓 반환
  - 빈 배열은 항상 false 반환

```
const arr = [1, 2, 3, 4, 5]
const result = arr.some((elem) => {
    return elem %2 === 0
})
// true
```

- every
  - 배열의 모든 요소가 판별 함수를 통과하면 참을 반환
  - 하나의 요소라도 통과하지 못하면 false 반환
  - 빈 배열은 항상 true 반환

```
const arr = [1,2,3,4,5]

const result = arr.every((elem)=> {
    return elem % 2 === 0
})
// false
```

객체(Object)

- 객체는 속성의 집합이며, 중괄호 내부에 key와 value의 쌍으로 표현
- key
  - 문자열 타입만 가능
  - key 이름에 띄어쓰기 등의 구분자가 있으면 따옴표로 묶어서 표현
- value
  - 모든 타입(함수포함)가능
- 객체 요소 접근
  - 점(.) 또는 대괄호([])로 가능
  - key 이름에 띄어쓰기 같은 구분자가 있으면 대괄호 접근만 가능

```
    const me = {
      name: 'jack',
      pN: '01012345678',
      'samsung products': {
        buds: 'Galaxy Buds pro',
        galaxy: 'Galaxy s99',
      }
    }

    console.log(me.name);
    console.log(me['name']);
    console.log(me['samsung products']);
    console.log(me['samsung products'].buds);
```

JSON

- JavaScript Object Notation
- Key-Value 형태로 이루어진 자료 표기법
- JavaScript의 Object와 유사한 구조를 가지고 있지만 Object는 그 자체로 타입이고, JSON은 형식이 있는 `문자열`
- 즉, JSON을 Object로 사용하기 위해서는 변환 작업이 필요

```
const jsObject = {
    coffee: 'Americano',
    iceCream: 'Cookie and cream'
}

// Object -> JSON
const objToJson = JSON.stringify(jsObject)

console.log(objToJson) // {"coffee":"Americano", "iceCream":"Cookie and cream"}
console.log(ytpeof objToJson) // string

// JSON -> Object
const jsonToObj = JSON.parse(objToJson)
console.log(objToJson) // {"coffee":"Americano", "iceCream":"Cookie and cream"}
console.log(ytpeof objToJson) // object
```