**1장. 자바스크립트 입문**

**\* JavaScript는 뭔가요?**

JavaScript는 웹 브라우저에서 사용하기 위하여 만들어진 프로그래밍 언어입니다. 이 언어는 90년대부터 주로 웹 브라우저 상에서 UI를 동적으로 보여주기 위하여 사용을 해왔습니다. 기존에는 브라우저에서만 사용해왔던 언어인데, 이제는 단순히 웹페이지에서만 국한되지 않고 Node.js 런타임을 통하여 서버 쪽에서도 사용을 할 수 있게 되었습니다.

추가적으로, NW.js, Electron 등의 도구를 사용하여 데스크탑 앱을 만들 수도 있고, react-native, Native Script 등의 도구를 사용하여 모바일 앱도 만들 수 있습니다.

그리고 하드웨어에서도 Node.js 를 통하여 JavaScript를 사용할 수 있기 때문에 IoT 진영에서도 사용될 수도 있죠.

JavaScript는 나날이 발전해 가고 있고, 매년마다 새로운 문법들도 나오고 있습니다. 이 강의에서는 모든 문법들을 다루지는 않습니다. 하지만, 알아 두면 개발하는데 충분히 쓸모 있는 만큼 학습하게 됩니다.

**<<01. Hello JavaScript>>**

\* Code Sandbox: 코드를 작성하고 바로 결과물을 확인할 수 있는 서비스

**<<02. 변수와 상수>>**

\* let value = 1;

\* 변수: 바뀔 수 있는 값. 한 번 값을 선언하고 나서 바꿀 수 있음. 하지만 한 번 선언했으면 똑 같은 이름으로 선언 못함.

let value = 1;

let value = 2;

const a = 1;

\* 상수: 한 번 선언하고 값이 바뀌지 않는 값. 고정적인 값.

\* var: 모던 자바스크립트에서는 사용하지 않음.

\* var이 let과 다른 차이점: 똑 같은 이름으로 여러 번 선언할 수도 있음. 추가적으로 var과 let은 사용할 수 있는 범위가 다름.

\* IE9, IE10과 같은 구형 브라우저에서는 let과 const를 사용할 수 없음. 하지만 보통 개발을 하게 될 때는 Babel과 같은 도구를 사용하여 코드가 구형 브라우저에서도 돌아갈 수 있게끔 변환작업을 함.

\* Null: 이 값이 없다

\* Undefined: 아직 값이 설정되지 않음

**<<03. 연산자>>**

\* 연산 순서: NOT -> AND -> OR

const value = !((true && false) || (true && false) || !false);

!true;

결과값: false

\* = 문자 3개와 2개의 차이점: //3개가 권장됨

- 2개: 값 비교 O, 타입 검사 X

- 3개: 값 비교 O, 타입 검사 O

**<<04. 조건문>>**

//아는 내용이라 스킵

**<<05. 함수>>**

\* 문자열 조합(+ 대신 더욱 편하게 조합하는 방법): ES6의 템플릿 리터럴 (Template Literal)이라는 문법을 사용하는 것

function hello(name) {

console.log(`Hello, ${name}!`); //템플릿 리터럴 사용

}

hello('velopert');

\* 화살표 함수: 함수를 선언하는 방식 중 또 다른 방법

const add = (a, b) => { //변수값 함수명 = (함수의 파라미터) => {코드 블록}

return a + b;

};

console.log(add(1, 2));

- function 키워드 대신 => 문자를 사용해서 함수를 구현.

- 화살표의 좌측에는 함수의 파라미터, 화살표 우측에는 코드 블록

const add = (a, b) => a + b;

console.log(add(1, 2));

**<<06. 객체>>**

\* 키에 공백이 있어야 하는 상황이라면 이를 따옴표로 감싸서 문자열로 넣어주면 됨

const sample = {

'key with space': true

};

\* 함수에서 객체를 파라미터로 받기

const ironMan = {

name: '토니 스타크',

actor: '로버트 다우니 주니어',

alias: '아이언맨'

};

const captainAmerica = {

name: '스티븐 로저스',

actor: '크리스 에반스',

alias: '캡틴 아메리카'

};

function print(hero) {

const text = `${hero.alias}(${hero.name}) 역할을 맡은 배우는 ${

hero.actor} 입니다.`;

console.log(text);

}

print(ironMan);

print(captainAmerica);

- 결과물

아이언맨(토니 스타크) 역할을 맡은 배우는 로버트 다우니 주니어 입니다.

캡틴 아메리카(스티븐 로저스) 역할을 맡은 배우는 크리스 에반스 입니다.

\* 객체 비구조화 할당

- print 함수를 보시면 파라미터로 받아온 hero 내부의 값을 조회 할 때 마다 hero. 를 입력하고 있는데, 객체 비구조화 할당이라는 문법을 사용하면 코드를 더욱 짧고 보기 좋게 작성 할 수 있습니다. 이런 문법은 “객체 구조 분해”라 불리기도 합니다.

const ironMan = {

name: '토니 스타크',

actor: '로버트 다우니 주니어',

alias: '아이언맨'

};

const captainAmerica = {

name: '스티븐 로저스',

actor: '크리스 에반스',

alias: '캡틴 아메리카'

};

function print(hero) {

const { alias, name, actor } = hero;

const text = `${alias}(${name}) 역할을 맡은 배우는 ${actor} 입니다.`;

console.log(text);

}

print(ironMan);

print(captainAmerica);

const { alias, name, actor } = hero;

- 이 코드가 객체에서 값들을 추출해서 새로운 상수로 선언해 주는 것임

- 더 나아가, 파라미터 단계에서 객체 비구조화 할당을 할 수도 있음

const ironMan = {

name: '토니 스타크',

actor: '로버트 다우니 주니어',

alias: '아이언맨'

};

const captainAmerica = {

name: '스티븐 로저스',

actor: '크리스 에반스',

alias: '캡틴 아메리카'

};

function print({ alias, name, actor }) {

const text = `${alias}(${name}) 역할을 맡은 배우는 ${actor} 입니다.`;

console.log(text);

}

print(ironMan);

print(captainAmerica);

\* 객체 안에 함수 넣기

const dog = {

name: '멍멍이',

sound: '멍멍!',

say: function say() {

console.log(this.sound);

}

};

dog.say(); //멍멍!

\* 객체 안에 함수를 넣을 때, 화살표 함수로 선언하면 제대로 작동하지 않음.

- 그 이유는 function으로 선언한 함수는 this가 제대로 자신이 속한 객체를 가리키게 되는데, 화살표 함수는 그렇지 않기 때문임

\* Getter 함수: 특정 값을 조회할 때 우리가 설정한 함수로 연산된 값을 반환함

const numbers = {

a: 1,

b: 2,

get sum() {

console.log('sum 함수가 실행됩니다!');

return this.a + this.b;

}

};

console.log(numbers.sum); //3

numbers.b = 5;

console.log(numbers.sum); //6

\* Setter 함수:

const numbers = {

\_a: 1,

\_b: 2,

sum: 3,

calculate() {

console.log('calculate');

this.sum = this.\_a + this.\_b;

},

get a() {

return this.\_a;

},

get b() {

return this.\_b;

},

set a(value) {

console.log('a가 바뀝니다.');

this.\_a = value;

this.calculate();

},

set b(value) {

console.log('b가 바뀝니다.');

this.\_b = value;

this.calculate();

}

};

console.log(numbers.sum); //3

numbers.a = 5;

numbers.b = 7;

numbers.a = 9;

console.log(numbers.sum); //16

console.log(numbers.sum); //16

console.log(numbers.sum); //16

**<<07. 배열>>**

\* 배열

const objects = [{ name: '멍멍이' }, { name: '야옹이' }];

console.log(objects); //멍멍이, 야옹이

console.log(objects[0]); //멍멍이

console.log(objects[1]); //야옹이

\* 배열에 새 항목 추가하기

const objects = [{ name: '멍멍이' }, { name: '야옹이' }];

objects.push({

name: '멍뭉이'

});

console.log(objects); //멍멍이, 야옹이, 멍뭉이

\* 배열 크기 알아내기

const objects = [{ name: '멍멍이' }, { name: '야옹이' }];

console.log(objects.length); //2

objects.push({

name: '멍뭉이'

});

console.log(objects.length); //3

**<<08. 반복문>>**

\* for…if 반복문:

let numbers = [10, 20, 30, 40, 50];

for (let number of numbers) {

console.log(number); //10 20 30 40 50

}

\* 객체를 위한 반복문 for…in:

const doggy = {

name: '멍멍이',

sound: '멍멍',

age: 2

};

console.log(Object.entries(doggy)); //객체 전부 보여줌

console.log(Object.keys(doggy)); //name, sound, age

console.log(Object.values(doggy)); //멍멍이, 멍멍, 2

\* 각 함수의 역할:

- Object.entries: [[키, 값], [키, 값]] 형태의 배열로 변환

- Object.keys: [키, 키, 키] 형태의 배열로 변환

- Object.values: [값, 값, 값] 형태의 배열로 변환

\* for…in:

const doggy = {

name: '멍멍이',

sound: '멍멍',

age: 2

};

for (let key in doggy) {

console.log(`${key}: ${doggy[key]}`); //name: 멍멍이 sound: 멍멍 age: 2

}

\* 배열을 파라미터로 받아 총합을 구하는 함수:

function sumOf(numbers) {

let sum = 0;

for (let i = 0; i < numbers.length; i++) {

sum += numbers[i];

}

return sum; //15

}

const result = sumOf([1, 2, 3, 4, 5]);

console.log(result);

\* 퀴즈:

-숫자로 이루어진 배열이 주어졌을 때, 해당 숫자 배열안에 들어있는 숫자 중 3보다 큰 숫자로만 이루어진 배열을 새로 만들어서 반환해보세요.

function biggerThanThree(numbers) {

      let array = [];

        for (let key in numbers) {

          if (3 < numbers[key]) {

            array.push(numbers[key]);

          }

        }

        return array;

}

const numbers = [1, 2, 3, 4, 5, 6, 7];

console.log(biggerThanThree(numbers)); // [4, 5, 6, 7]

**<<09. 배열 내장 함수>>**

**\* forEach**

const superheroes = ['아이언맨', '캡틴 아메리카', '토르', '닥터 스트레인지'];

superheroes.forEach(hero => {

console.log(hero);

});

- forEach 함수의 파라미터로는, 각 원소에 대하여 처리하고 싶은 코드를 함수로 넣어줌. 이 함수의 파라미터 hero는 가 원소를 가리키게 됨

    <script>

      function countBiggerThanTen(numbers) {

        let count = 0;

        numbers.forEach((element) => {

          if (element > 10) {

            count++;

          }

        });

        return count;

      }

      const count = countBiggerThanTen([1, 2, 3, 5, 10, 20, 30, 40, 50, 60]);

      console.log(count); *// 5*

    </script>

**\* 콜백함수:** 함수형태의 파라미터를 전달하는 것

**\* map:** 배열 안의 각 원소를 변환할 때 사용되며, 이 과정에서 새로운 배열이 만들어짐

**\* 모든 숫자를 제곱해서 새로운 배열을 만들고 싶다면**

- forEach 사용

const array = [1, 2, 3, 4, 5, 6, 7, 8];

const squared = [];

array.forEach(n => {

squared.push(n \* n);

});

console.log(squared); //1, 4, 9, 16, 25, 49, 64

- map 사용

const array = [1, 2, 3, 4, 5, 6, 7, 8];

const square = n => n \* n;

const squared = array.map(square);

console.log(squared);

const squared = array.map(n => n \* n);

console.log(squared);

    <script>

      function countBiggerThanTen(numbers) {

        let count = 0;

        numbers.map((num) => {

          if (num > 10) {

            count++;

          }

        });

        return count;

      }

      const count = countBiggerThanTen([1, 2, 3, 5, 10, 20, 30, 40, 50, 60]);

      console.log(count); *// 5*

    </script>

**\* 변화함수**: map 함수의 파라미터로 변화를 주는 함수를 전달해줌

**\* indexOf:** 원하는 항목이 몇번째 원소인지 찾아주는 함수

const superheroes = ['아이언맨', '캡틴 아메리카', '토르', '닥터 스트레인지'];

const index = superheroes.indexOf('토르'); //2

console.log(index);

**\* findIndex:**

- 배열 안에 있는 값이 숫자, 문자열, 또는 불리언이라면 찾고자하는 항목이 몇번째 원소인지 알아내려면 indexOf를 사용하면 됨. 하지만 배열 안의 값이 객체거나 배열이라면 indexOf로 찾을 수 없음

- id가 3인 객체가 몇번째인지 찾으려면, findIndex 함수에 검사하고자 하는 조건을 반환하는 함수를 넣어서 사용할 수 있음

const todos = [

{

id: 1,

text: '자바스크립트 입문',

done: true

},

{

id: 2,

text: '함수 배우기',

done: true

},

{

id: 3,

text: '객체와 배열 배우기',

done: true

},

{

id: 4,

text: '배열 내장함수 배우기',

done: false

}

];

const index = todos.findIndex(todo => todo.id === 3);

console.log(index); //2

**\* find:** findIndex와 비슷한데, 찾아낸 값이 몇번째인지가 아니라 찾아낸 값 자체를 반환함

const todos = [

{

id: 1,

text: '자바스크립트 입문',

done: true

},

{

id: 2,

text: '함수 배우기',

done: true

},

{

id: 3,

text: '객체와 배열 배우기',

done: true

},

{

id: 4,

text: '배열 내장함수 배우기',

done: false

}

];

const todo = todos.find(todo => todo.id === 3);

console.log(todo); //{id: 3, text: “객체와 배열 배우기”, done: true}

**\* filter:** 배열에서 특정 조건을 만족하는 값들만 따로 추출하여 새로운 배열을 만듬.

const todos = [

{

id: 1,

text: '자바스크립트 입문',

done: true

},

{

id: 2,

text: '함수 배우기',

done: true

},

{

id: 3,

text: '객체와 배열 배우기',

done: true

},

{

id: 4,

text: '배열 내장함수 배우기',

done: false

}

];

const tasksNotDone = todos.filter(todo => todo.done === false);

// const tasksNotDone = todos.filter(todo => !todo.done);

console.log(tasksNotDone);

- 결과값:

[

{

id: 4,

text: '배열 내장 함수 배우기',

done: false

}

];

- filter 함수에 넣는 파라미터는 조건을 검사하는 함수를 넣어주며, 이 함수의 파라미터로 각 원소의 값을 받아오게 됨.

    <script>

      function countBiggerThanTen(numbers) {

        let count = 0;

        numbers.filter((num) => {

          if (num > 10) {

            count++;

          }

        });

        return count;

      }

      const count = countBiggerThanTen([1, 2, 3, 5, 10, 20, 30, 40, 50, 60]);

      console.log(count); *// 5*

    </script>

**\* splice:** 배열에 특정 항목을 제거할 때 사용

- 첫번째 파라미터는 어떤 인덱스부터 지울지를 의미하고

- 두번째 파라미터는 그 인덱스부터 몇 개를 지울지를 의미함

const numbers = [10, 20, 30, 40];

const index = numbers.indexOf(30);

numbers.splice(index, 1);

console.log(numbers); //[10, 20, 40]

**\* slice:** 배열을 잘라낼 때 사용하는데, 기존의 배열은 건들지 않음

- 첫번째 파라미터는 어디서부터 자를지

- 두번째 파라미터는 어디까지 자를지를 의미함

const numbers = [10, 20, 30, 40];

const sliced = numbers.slice(0, 2); // 0부터 시작해서 2전까지

console.log(sliced); // [10, 20]

console.log(numbers); // [10, 20, 30, 40]

**\* shift와 pop**

- shift: 첫번째 원소를 배열에서 추출함. 추출하는 과정에서 배열에서 해당 원소는 사라짐.

const numbers = [10, 20, 30, 40];

const value = numbers.shift();

console.log(value); //10

console.log(numbers); //[20, 30, 40]

- pop: 맨 마지막 항목을 추출

const numbers = [10, 20, 30, 40];

const value = numbers.pop();

console.log(value); //40

console.log(numbers); //[10, 20, 30

**\* unshift:** 배열의 맨 앞에 새 원소를 추가함

const numbers = [10, 20, 30, 40];

numbers.unshift(5);

console.log(numbers); //[5, 10, 20, 30, 40]

**\* concat:** 여러 개의 배열을 하나의 배열로 합쳐줌

- 기존 함수에 변화를 주지 않음

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

const concated = arr1.concat(arr2);

console.log(concated); //[1, 2, 3, 4, 5, 6]

**\* join:** 배열 안의 값들을 문자열 형태로 합쳐줌

const array = [1, 2, 3, 4, 5];

console.log(array.join()); // 1,2,3,4,5

console.log(array.join(' ')); // 1 2 3 4 5

console.log(array.join(', ')); // 1, 2, 3, 4, 5

**\* reduce:**

- accumulator와 current를 파라미터로 가져와서 결과를 반환하는 콜백함수

- reduce 함수에서 사용할 초기값

- 첫번째 파라미터: 누적값

- 두번재 파라미터: 현재 가리키고 있는 값

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

let sum = array.reduce((accumulator, current) => accumulator + current, 0);

console.log(sum);

**\* reduce를 이용한 평균 계산**

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

let sum = numbers.reduce((accumulator, current, index, array) => {

if (index === array.length - 1) {

return (accumulator + current) / array.length;

}

return accumulator + current;

}, 0);

console.log(sum); //3

    <script>

      function countBiggerThanTen(numbers) {

        let count = 0;

        numbers.reduce((acc, current) => {

          if (current > 10) {

            count++;

          }

        }, 0);

        return count;

      }

      const count = countBiggerThanTen([1, 2, 3, 5, 10, 20, 30, 40, 50, 60]);

      console.log(count); *// 5*

    </script>

**\* 퀴즈:**

숫자 배열이 주어졌을 때 10보다 큰 숫자의 갯수를 반환하는 함수를 만드세요.

function countBiggerThanTen(numbers) {

/\* 구현해보세요 \*/

}

const count = countBiggerThanTen([1, 2, 3, 5, 10, 20, 30, 40, 50, 60]);

console.log(count); // 5

**<<10. 프로토타입과 클래스>>**

**\* 프로토타입:** 객체 생성자 함수 아래 ‘.prototype.[원하는 키] = 코드’를 입력하여 설정할 수 있음

function Animal(type, name, sound) {

this.type = type;

this.name = name;

this.sound = sound;

}

Animal.prototype.say = function() {

console.log(this.sound);

};

Animal.prototype.sharedValue = 1;

const dog = new Animal('개', '멍멍이', '멍멍');

const cat = new Animal('고양이', '야옹이', '야옹');

dog.say(); //멍멍

cat.say(); //야용

console.log(dog.sharedValue); //1

console.log(cat.sharedValue); //1

**\* 객체 생성자 상속 받기**

- 예를 들어 Cat과 Dog라는 새로운 객체 생성자를 만들다고 가정하자. 그리고 해당 객체 생성자들에서 Animal의 기능을 재사용한다고 가정하자.

function Animal(type, name, sound) {

this.type = type;

this.name = name;

this.sound = sound;

}

Animal.prototype.say = function() {

console.log(this.sound);

};

Animal.prototype.sharedValue = 1;

function Dog(name, sound) {

Animal.call(this, '개', name, sound);

}

Dog.prototype = Animal.prototype;

function Cat(name, sound) {

Animal.call(this, '고양이', name, sound);

}

Cat.prototype = Animal.prototype;

const dog = new Dog('멍멍이', '멍멍');

const cat = new Cat('야옹이', '야옹');

dog.say();

cat.say();

**\* 클래스:** ES6부터 class라는 문법이 추가되었음. 객체 생성자로 구현했던 코드를 조금 더 명확하고 깔끔하게 구현할 수 있게 해줌. 추가적으로 상속도 쉽게 해줄 수 있음

- 클래스 내부의 메서드를 이용하여 자동 prototype 등록

class Animal {

constructor(type, name, sound) {

this.type = type;

this.name = name;

this.sound = sound;

}

say() {

console.log(this.sound);

}

}

const dog = new Animal('개', '멍멍이', '멍멍');

const cat = new Animal('고양이', '야옹이', '야옹');

dog.say(); //멍멍

cat.say(); //야옹

- class 이용하여 다른 클래스를 쉽게 상속

class Animal {

constructor(type, name, sound) {

this.type = type;

this.name = name;

this.sound = sound;

}

say() {

console.log(this.sound);

}

}

class Dog extends Animal {

constructor(name, sound) {

super('개', name, sound);

}

}

class Cat extends Animal {

constructor(name, sound) {

super('고양이', name, sound);

}

}

const dog = new Dog('멍멍이', '멍멍');

const cat = new Cat('야옹이', '야옹');

dog.say(); //멍멍

cat.say(); //야옹

**\* extends, super():**

- 상속을 할 때는 extends 키워드를 이용

- constructor에서 사용하는 super() 함수가 상속 받은 클래스의 생성자를 가리킴

class Animal {

constructor(type, name, sound) {

this.type = type;

this.name = name;

this.sound = sound;

}

}

class Dog extends Animal {

constructor(name, sound) {

super('개', name, sound);

}

}

class Cat extends Animal {

constructor(name, sound) {

super('고양이', name, sound);

}

}

const dog = new Dog('멍멍이', '멍멍');

const dog2 = new Dog('왈왈이', '왈왈');

const cat = new Cat('야옹이', '야옹');

const cat2 = new Cat('냐옹이', '냐옹');

dog.say(); //멍멍

dog2.say(); //왈왈

cat.say(); //야옹

cat2.say(); //냐옹