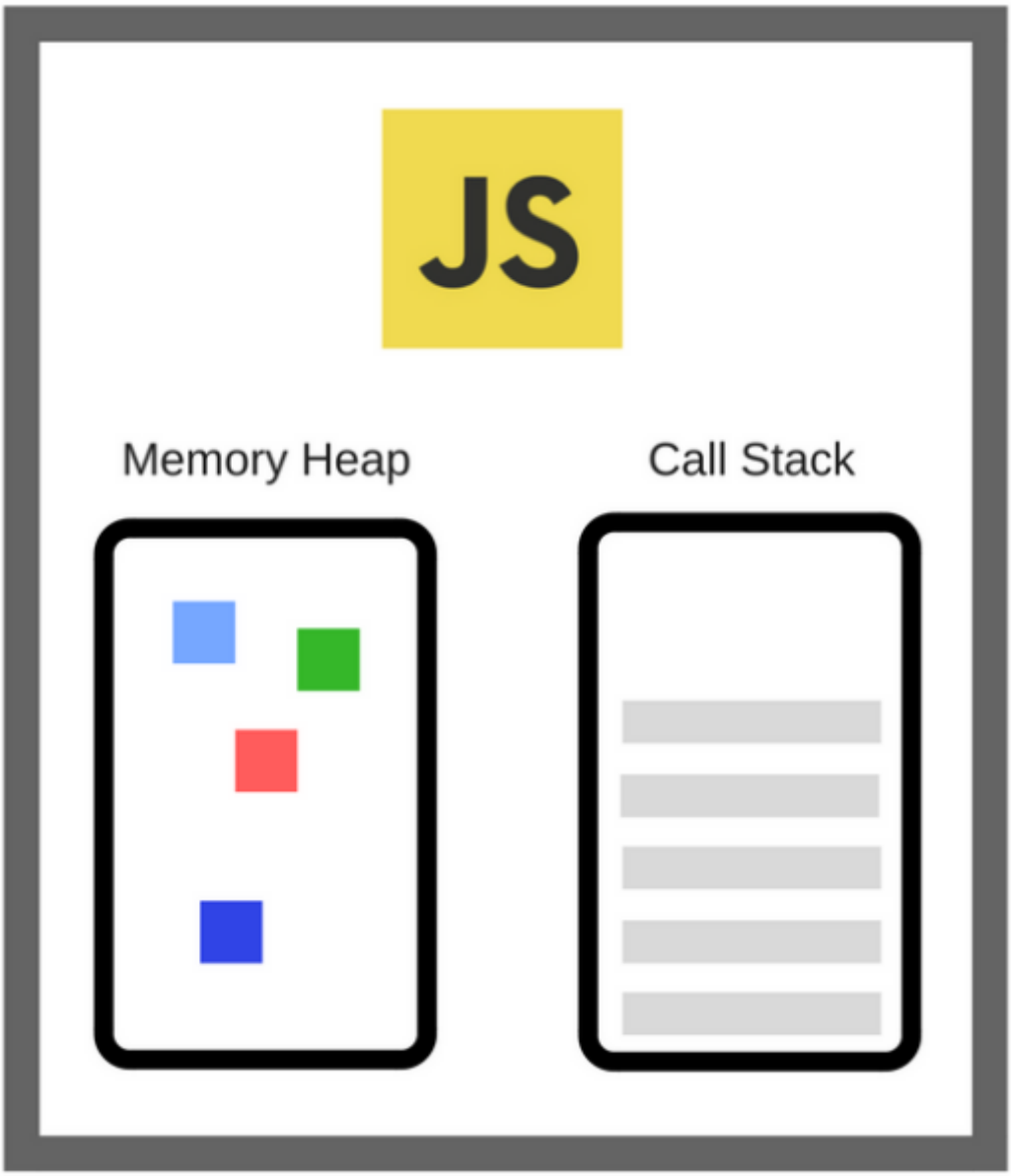
230524_실습

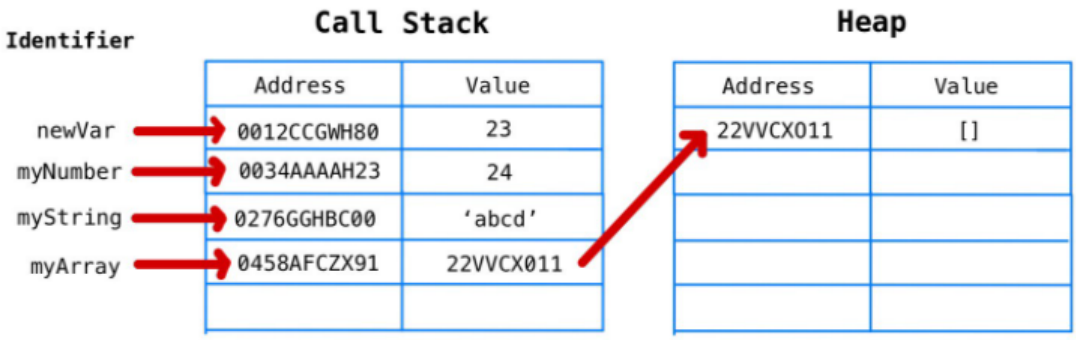
JavaScript

Memory

JavaScript의 메모리 구조

- JavaScript 엔진 메모리 구조





Call Stack

- 원시 타입 `Primitive Type` 값, 실행 컨텍스트(Execution Context) 저장

Memory Heap

- 참조 타입 `Reference Type` (객체, 배열, 함수 등) 저장
- 동적 할당 가능한 영역

let vs const

- let
  - 값의 변경이 아닌 메모리 주소의 재할당
- const
  - 메모리 주소 변경 불가
  - Reference Type의 경우 const로 선언 하는것이 좋음

Garbage Collection

- 참조되지 않는 값을 메모리에서 해제

연산자

할당 연산자

1. 연산자

이름	단축 연산자	뜻
할당	`x = y`	`x = y`
더하기 할당	`x += y`	`x = x + y`

이름	단축 연산자	뜻
빼기 할당	`x -= y`	`x = x - y`
곱하기 할당	`x *= y`	`x = x * y`
나누기 할당	`x /= y`	`x = x / y`
나머지 할당	`x %= y`	`x = x % y`
거듭제곱 할당	`x **= y`	`x = x ** y`
왼쪽 시프트 할당	`x <<= y`	`x = x << y`
오른쪽 시프트 할당 (en-US)	`x >>= y`	`x = x >> y`
부호 없는 오른쪽 시프트 할당	`x >>>= y`	`x = x >>> y`
비트 AND 할당	`x &= y`	`x = x & y`
비트 XOR 할당 (en-US)	`x ^= y`	`x = x ^ y`
비트 OR 할당 (en-US)	`x |= y`	`x = x | y`
논리 AND 할당 (en-US)	`x &&= y`	`x && (x = y)`
논리 OR 할당 (en-US)	`x ||= y`	`x || (x = y)`
널 병합 할당 (en-US)	`x ??= y`	`x ?? (x = y)`

2. 구조 분해

```
var foo = ['one', 'two', 'three'];

// 구조 분해 없음
var one   = foo[0];
var two   = foo[1];
var three = foo[2];

// 구조 분해 사용
var [one, two, three] = foo;
```

비교 연산자

1. 연산자

연산자	설명	`true`를 반환하는 예제
동등 (`==`)	피연산자가 서로 같으면 `true`를 반환합니다.	`3 == var1` `"3" == var1` `3 == '3'`
부등 (`!=`)	피연산자가 서로 다르면 `true`를 반환합니다.	`var1 != 4` `var2 != "3"`
일치 (en-	두 피연산자의 값과 타입이 모두 같은 경우 `true`를 반	`3 === var1`

US) (`===`)	환합니다. `Object.is`와 JavaScript에서의 같음을 참고하세요.	
불일치 (en-US) (`!==`)	피연산자의 값 또는 타입이 서로 다를 경우 `true`를 반환합니다.	`var1 !== "3" 3 !== '3'`
큼 (en-US) (`>`)	왼쪽 피연산자가 오른쪽 피연산자보다 크면 `true`를 반환합니다.	`var2 > var1 "12" > 2`
크거나 같음 (en-US) (`>=`)	왼쪽 피연산자가 오른쪽 피연산자와 같거나 크면 `true`를 반환합니다.	`var2 >= var1 var1 >= 3`
작음 (en-US) (`<`)	왼쪽 피연산자가 오른쪽 피연산자보다 작으면 `true`를 반환합니다.	`var1 < var2 "2" < 12`
작거나 같음 (en-US) (`<=`)	왼쪽 피연산자가 오른쪽 피연산자와 같거나 작으면 `true`를 반환합니다.	`var1 <= var2 var2 <= 5`

산술 연산자

1. 연산자

연산자	설명	예제
나머지 (`%`)	이항 연산자입니다. 두 피연산자를 나눴을 때의 나머지를 반환합니다.	12 % 5 는 2를 반환합니다.
증가 (`++`)	단항 연산자입니다. 피연산자에 1을 더합니다. 전위 연산자(`++x`)로 사용하면 피연산자에 1을 더한 값을 반환합니다. 후위 연산자(`x++`)로 사용한 경우 피연산자에 1을 더하기 전의 값을 반환합니다.	`x`가 3일 때, `++x`는 `x`에 4를 할당한 후 4를 반환합니다. 반면 `x++`는 3을 먼저 반환한 후 `x`에 4를 할당합니다.
감소 (`--`)	단항 연산자입니다. 피연산자에서 1을 뺍니다. 반환 값은 증가 연산자처럼 동작합니다.	`x`가 3일 때, `--x`는 `x`에 2를 할당한 후 2를 반환합니다. 반면 `x--`는 3을 먼저 반환한 후 `x`에 2를 할당합니다.
단항 부정 (`-`)	단항 연산자입니다. 피연산자의 부호를 반대로 바꾼 값을 반환합니다.	`x`가 3일 때, `-x`는 -3을 반환합니다.
단항 플러스 (`+`)	단항 연산자입니다. 피연산자가 숫자 타입이 아니면 숫자로 변환을 시도합니다.	`+"3"`은 `3`을 반환합니다. `+true`는 `1`을 반환합니다.
거듭제곱 (`**`)	`base^exponent`, 즉 `base`를 `exponent`로 거듭제곱한 결과를 반환합니다.	`2 ** 3`은 `8`을 반환합니다. `10 ** -1`은 `0.1`을 반환합니다.

비트 연산자

1. 기본 비트 연산자

연산자	사용법	설명

비트 AND	`a & b`	두 피연산자의 각 자리 비트의 값이 모두 1인 위치에 1을 반환합니다.
비트 OR (en-US)	`a | b`	두 피연산자의 각 자리 비트의 값이 모두 0인 위치에 0을 반환합니다.
비트 XOR (en-US)	`a ^ b`	두 피연산자의 각 자리 비트의 값이 서로 같은 위치에 0을 반환합니다. [두 피연산자의 각 자리 비트의 값이 서로 다른 위치에 1을 반환합니다.]
비트 NOT	`~ a`	피연산자의 각 자리의 비트를 뒤집습니다.
왼쪽 시프트	`a << b`	`a`의 이진 표현을 `b`만큼 왼쪽으로 이동하고, 오른쪽은 0으로 채웁니다.
오른쪽 시프트 (en-US)	`a >> b`	`a`의 이진 표현을 `b`만큼 오른쪽으로 이동하고, 1 미만으로 이동한 비트는 버립니다.
부호 없는 오른쪽 시프트 (en-US)	`a >>> b`	`a`의 이진 표현을 `b`만큼 오른쪽으로 이동하고, 1 미만으로 이동한 비트는 버립니다. 왼쪽은 0으로 채웁니다.

2. 비트 논리 연산자

표현식	결과	이진법 설명
`15 & 9`	`9`	`1111 & 1001 = 1001`
`15 | 9`	`15`	`1111 | 1001 = 1111`
`15 ^ 9`	`6`	`1111 ^ 1001 = 0110`
`~15`	`-16`	`~ 0000 0000 ... 0000 1111 = 1111 1111 ... 1111 0000`
`~9`	`-10`	`~ 0000 0000 ... 0000 1001 = 1111 1111 ... 1111 0110`

3. 비트 시프트 연산자

연산자	설명	예제
왼쪽 시프트 (`<<`)	왼쪽 피연산자를 오른쪽 피연산자만큼 왼쪽으로 시프트합니다. 왼쪽으로 넘치는 비트는 버리고, 오른쪽을 0으로 채웁니다.	`9<<2`는, 1001을 왼쪽으로 2번 시프트하면 100100이므로 36입니다.
오른쪽 시프트 (`>>`) (en-US)	왼쪽 피연산자를 오른쪽 피연산자만큼 오른쪽으로 시프트합니다. 오른쪽으로 넘치는 비트는 버리고, 왼쪽은 제일 큰 비트의 값으로 채웁니다.	`9>>2`는, 1001을 오른쪽으로 2번 시프트하면 10이므로 2입니다. 마찬가지로 `-9>>2`는, 부호를 유지하므로 -3을 반환합니다.
부호 없는 오른쪽 시프트	왼쪽 피연산자를 오른쪽 피연산자만큼 오른쪽으로 시프트합니다.	`19>>>2`는, 10011을 오른쪽으로 2번 시프트하면 100이므로 4입니다. 양의 수

(`>>>`) (en-US)	오른쪽으로 넘치는 비트는 버리고, 왼쪽은 0으로 채웁니다.	에 대해서는 오른쪽 시프트와 부호 없는 오른쪽 시프트 둘 다 같은 결과를 반환합니다.

논리 연산자

1. 연산자

연산자	사용법	설명
논리 AND (`&&`)	`expr1 && expr2`	`expr1`을 `false`로 변환할 수 있으면 `expr1`을 반환합니다. 그 외의 경우에는 `expr2`를 반환합니다. 따라서 불리언 값과 함께 사용한 경우, 둘 다 참일 때 `true`를, 그 외에는 `false`를 반환합니다.
논리 OR (en-US) (`||`)	`expr1 || expr2`	`expr1`을 `true`로 변환할 수 있으면 `expr1`을 반환합니다. 그 외의 경우에는 `expr2`를 반환합니다. 따라서 불리언 값과 함께 사용한 경우, 둘 중 하나가 참일 때 `true`를, 그 외에는 `false`를 반환합니다.
논리 NOT (en-US) (`!`)	`!expr`	단일 피연산자를 `true`로 변환할 수 있으면 `false`를 반환합니다. 그 외에는 `true`를 반환합니다.

2. 예제 코드

- &&

```
var a1 = true && true;      // t && t는 true 반환
var a2 = true && false;     // t && f는 false 반환
var a3 = false && true;     // f && t는 false 반환
var a4 = false && (3 == 4); // f && f는 false 반환
var a5 = 'Cat' && 'Dog';    // t && t는 Dog 반환
var a6 = false && 'Cat';    // f && t는 false 반환
var a7 = 'Cat' && false;    // t && f는 false 반환
```

- ||

```
var o1 = true || true;      // t || t는 true 반환
var o2 = false || true;     // f || t는 true 반환
var o3 = true || false;     // t || f는 true 반환
var o4 = false || (3 == 4); // f || f는 false 반환
var o5 = 'Cat' || 'Dog';    // t || t는 Cat 반환
var o6 = false || 'Cat';    // f || t는 Cat 반환
var o7 = 'Cat' || false;    // t || f는 Cat 반환
```

- !

```
var n1 = !true;  // !t는 false 반환
var n2 = !false; // !f는 true 반환
var n3 = !'Cat'; // !t는 false 반환
```

문자열 연산자

1. 예제 코드

```
console.log('나만의 ' + '문자열'); // 콘솔에 "나만의 문자열"을 기록

var mystring = '한';
mystring += '글'; // "한글"로 평가되며, mystring에 "한글"을 할당함
```

조건 (삼항) 연산자

1. 예제 코드

```
condition ? val1 : val2;

var status = age >= 18 ? '성인' : '미성년자';
```

- 만약 `condition`이 참이라면, 조건 연산자는 `val1`을 반환하고, 그 외에는 `val2`를 반환
- 다른 연산자를 사용할 수 있는 곳이라면 조건 연산자도 사용 가능

쉼표 연산자

1. 예제 코드

```
var x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
var a = [x, x, x, x, x];

for (var i = 0, j = 9; i <= j; i++, j--);
//                                  ^
  console.log('a[' + i + '][' + j + ']= ' + a[i][j]);
```

- 두 피연산자를 모두 평가한 후 오른쪽 피연산자의 값을 반환
- 연산자는 주로 `for` 반복문 안에서 사용하여 한 번의 반복으로 여러 변수를 변경할 때 사용
- 꼭 필요하지 않다면, 그 외의 상황에 사용하는 것은 좋지 않은 코드 스타일

- 쉼표 연산자보다는 두 개의 분리된 명령문 사용 권장

단항 연산자

1. delete

```
delete object.property;
delete object[propertyKey];
delete objectName[index];
```

- 객체의 속성 삭제

2. typeof

```
typeof operand
typeof (operand)
```

```
var myFun = new Function('5 + 2');
var shape = 'round';
var size = 1;
var foo = ['Apple', 'Mango', 'Orange'];
var today = new Date();

typeof myFun;     // "function" 반환
typeof shape;     // "string" 반환
typeof size;      // "number" 반환
typeof foo;       // "object" 반환
typeof today;     // "object" 반환
typeof dontExist; // "undefined" 반환

// 키워드 true와 null
typeof true; // "boolean" 반환
typeof null; // "object" 반환

// 숫자와 문자열
typeof 62;            // "number" 반환
typeof 'Hello world'; // "string" 반환

// 객체의 속성
typeof document.lastModified; // "string" 반환
typeof window.length;         // "number" 반환
typeof Math.LN2;              // "number" 반환

// 메서드와 함수
typeof blur;        // "function" 반환
typeof eval;        // "function" 반환
typeof parseInt;    // "function" 반환
typeof shape.split; // "function" 반환

// 사전 정의된 객체
```

```
typeof Date;     // "function" 반환
typeof Function; // "function" 반환
typeof Math;     // "object" 반환
typeof Option;   // "function" 반환
typeof String;   // "function" 반환
```

3. void

```
void (expression)
void expression
```

관계 연산자

1. in

```
propNameOrNumber in objectName // true or false
```

```
// 배열
var trees = ['redwood', 'bay', 'cedar', 'oak', 'maple'];
0 in trees;        // true 반환
3 in trees;        // true 반환
6 in trees;        // false 반환
'bay' in trees;    // false 반환 (인덱스에 위치한 값이 아니라
                   // 인덱스 자체를 지정해야 함)
'length' in trees; // true 반환 (length는 Array의 속성임)

// 내장 객체
'PI' in Math;         // true 반환
var myString = new String('coral');
'length' in myString; // true 반환

// 사용자 정의 객체
var mycar = { make: 'Honda', model: 'Accord', year: 1998 };
'make' in mycar;  // true 반환
'model' in mycar; // true 반환
```

2. instanceof

```
objectName instanceof objectType
```

- 지정한 객체가 지정한 객체 타입에 속하면 `true`를 반환

```
var theDay = new Date(1995, 12, 17);
if (theDay instanceof Date) {
```

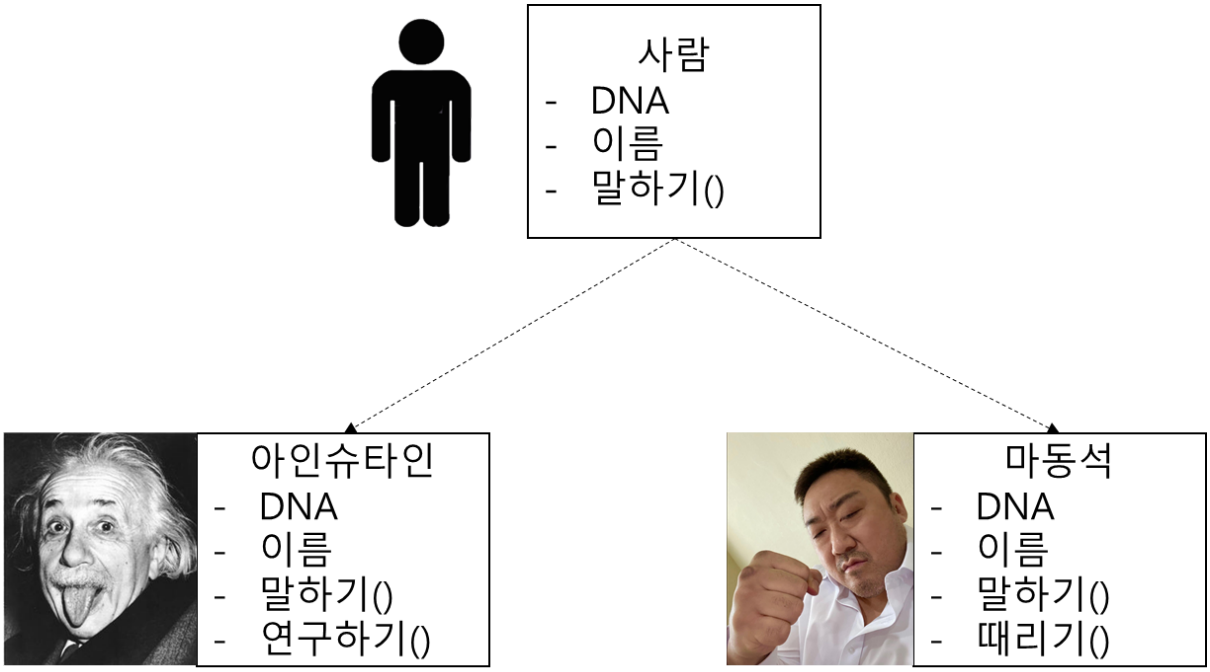
```
  // 실행할 명령문
}
```

연산자 우선순위

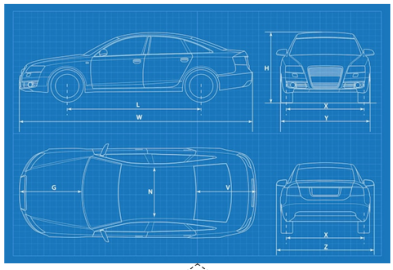
연산자 유형	개별 연산자
맴버 접근	`.` `[]`
인스턴스 호출/생성	`()` `new`
증감	`!` `~` `-` `+` `++` `--` `typeof` `void` `delete`
거듭제곱	`**`
곱하기/나누기	`*` `/` `%`
더하기/빼기	`+` `-`
비트 시프트	`<<` `>>` `>>>`
관계	`<` `<=` `>` `>=` `in` `instanceof`
동등/일치	`==` `!=` `===` `!==`
비트 AND	`&`
비트 XOR	`^`
비트 OR	`|`
논리 AND	`&&`
논리 OR	`||`
조건	`?:`
할당	`=` `+=` `-=` `**=` `*=` `/=` `%=` `<<=` `>>=` `>>>=` `&=` `^=` `|=` `&&=` `||=` `??=`
쉼표	`,`

Class

class



사람 class와 object



자동차
- 모델 예) 소나타, K9, ...
- 종류 예) 승용, SUV, ...
- 시동() 예) (키), (앱 인증), ...
- 기어() 예) 1, 2, 3, P, D ...
- 라이트() 예) 상향, 주행, ...

디 올 뉴 그랜저
- 모델 예) 소나타, K9, ...
- 종류 예) 승용, SUV, ...
- 시동() 예) (키), (앱 인증), ...
- 기어() 예) 1, 2, 3, P, D ...
- 라이트() 예) 상향, 주행, ...
- 자동주차()
- 안마()

펠리세이드
- 모델 예) 소나타, K9, ...
- 종류 예) 승용, SUV, ...
- 시동() 예) (키), (앱 인증), ...
- 기어() 예) 1, 2, 3, P, D ...
- 라이트() 예) 상향, 주행, ...
- 캠핑()
- 주행모드()

자동차 class와 object

- object를 생성하기 위한 템플릿
- class의 이름의 첫 글자는 대문자 사용

Object

- 구현할 대상
- class로 생성된 실체
- class 타입으로 선언

Instance

- class로 생성하여 메모리에 할당된 값으로 존재
- object > instance

Property

- class 내부에 존재하는 변수

Method

- class 내부에 존재하는 함수

constructor()

- class 생성시 자동으로 호출되는 메서드

sample code1

```
class Car {
  var name = "My car";
  gear = "P";
  #key;

  constructor(model, type, key) {
    this.model = model;
    this.type = type;
    this.#key = key;
  }

  set_gear(gear) {
    this.gear = gear;
  }
}

var my_car = new Car("Audi-R8", "Sports", "OKEIF27B#kgp2847HJ");
console.log(my_car.name) // My car
console.log(my_car.model) // Audi-R8
console.log(my_car.type) // Sports
console.log(my_car.type) // ERROR!
```

extends

- class의 상속에 사용되는 키워드

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

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

```
class Dog extends Animal {
  constructor(name) {
    super(name); // super class 생성자를 호출하여 name 매개변수 전달
  }

  speak() {
    console.log(`${this.name} barks.`);
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

배열

배열 생성

1. 배열 리터럴 대괄호([ ])를 사용하여 만드는 방법

```
// 배열 생성 (빈 배열)
var arr = [];

arr[0] = 'zero';
arr[1] = 'one';
arr[2] = 'tow';

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

```
// 배열 생성 (초기 값 할당)
var arr = ['zero', 'one', 'tow'];

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

```
// 배열 생성 (배열 크기 지정)
// 쉼표 개수만큼 크기가 지정됨
var arr = [,,,];

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}

// 값이 할당되지 않아서 undefined 3번 출력
```

2. Array() 생성자 함수로 배열을 생성하는 방법

```
// 배열 생성 (빈 배열)
var arr = new Array();

arr[0] = 'zero';
arr[1] = 'one';
arr[2] = 'tow';

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

```
// 배열 생성 (초기 값 할당)
var arr = new Array('zero', 'one', 'tow');

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

```
// 배열 생성 (배열 크기 지정)
// 원소가 1개이고 숫자인 경우 배열 크기로 사용됨
var arr = new Array(3);

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}

// 값이 할당되지 않아서 undefined 3번 출력
```

JavaScript 배열의 특징

- 배열 내부의 데이터 타입이 서로 다를 수 있다

```
// 서로 다른 데이터 타입을 담을 수 있다
var arr = [1234, 'test', true];
```

- 배열의 크기는 동적으로 변경될 수 있다

```
var arr = [1234, 'test', true];

// 배열의 크기를 임의로 변경( 3 -> 5 )
// arr[3], arr[4]는 값이 할당 되지 않았기 때문에 undefined
arr.length = 5;
```

```
// 새로운 배열을 추가하면 크기는 자동으로 변경 ( 5 -> 6 )
arr[5] = 'apple';

// 새로운 배열 추가로 크기 변경 ( 6 -> 7 )
arr.push('banana');

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}

/*
// 출력 결과
1234
test
true
undefined
undefined
apple
banana
*/
```

반복문

for 문

- for 반복문은 어떤 특정한 조건이 거짓으로 판별될 때까지 반복합니다. 자바스크립트의 반복문은 C의 반복문과 비슷합니다. for 반복문은 다음과 같습니다.

```
for ([초기문]; [조건문]; [증감문]) {
      문장
}
```

- for문이 실행될 때, 다음과 같이 실행됩니다.
    1. 초기화 구문인 초기문이 존재한다면 초기문이 실행됩니다. 이 표현은 보통 1이나 반복문 카운터로 초기 설정이 됩니다. 그러나 복잡한 구문으로 표현 될 때도 있습니다. 또한 변수로 선언 되기도 합니다
    2. 조건문은 조건을 검사합니다. 만약 조건문이 참이라면, 그 반복문은 실행됩니다. 만약 조건문이 거짓이라면, 그 for문은 종결됩니다. 만약 그 조건문이 생략된다면, 그 조건문은 참으로 추정됩니다.
    3. 문장이 실행됩니다. 많은 문장을 실행할 경우엔, { } 를 써서 문장들을 묶어 줍니다.
    4. 갱신 구문인 증감문이 존재한다면 실행되고 2번째 단계로 돌아갑니다.
- 예시

```
<form name="selectForm">
  <p>
    <label for="musicTypes">Choose some music types, then click the button below:
</label>
    <select id="musicTypes" name="musicTypes" multiple="multiple">
      <option selected="selected">R&B</option>
      <option>Jazz</option>
      <option>Blues</option>
      <option>New Age</option>
      <option>Classical</option>
      <option>Opera</option>
    </select>
  </p>
  <p><input id="btn" type="button" value="How many are selected?" /></p>
</form>

<script>
function howMany(selectObject) {
  var numberSelected = 0;
  for (var i = 0; i < selectObject.options.length; i++) {
    if (selectObject.options[i].selected) {
      numberSelected++;
    }
  }
  return numberSelected;
}

var btn = document.getElementById("btn");
btn.addEventListener("click", function(){
  alert('Number of options selected: ' + howMany(document.selectForm.musicTypes))
});
</script>
```

```
for (var i = 0; i < selectObject.options.length; i++) {
  if (selectObject.options[i].selected) {
    numberSelected++;
  }
}
```

do… while 문

- do...while 문은 특정한 조건이 거짓으로 판별될 때까지 반복합니다. do...while 문은 다음과 같습니다.

```
do {
      문장
} while (조건문);
```

- **조건문을 확인하기 전에 문장은 한번 실행됩니다.** 많은 문장을 실행하기 위해선 { }를 써서 문장들을 묶어줍니다. 만약 조건이 참이라면, 그 문장은 다시 실행됩니다. 매 실행 마지막마다 조건문이 확인됩니다. 만약 조건문이 거짓일 경우, 실행을 멈추고 do...while 문 바로 아래에 있는 문장으로 넘어가게 합니다.
- 예시

```
do {
  i += 1;
  console.log(i);
} while (i < 5);
```

while 문

- while 문은 어떤 조건문이 참이기만 하면 문장을 계속해서 수행합니다. while 문은 다음과 같습니다.

```
while (조건문) {
      문장
}
```

- 만약 조건문이 거짓이 된다면, 그 반복문 안의 문장은 실행을 멈추고 반복문 바로 다음의 문장으로 넘어갑니다.
- 조건문은 반복문 안의 문장이 실행되기 전에 확인 됩니다. 만약 조건문이 참으로 리턴된다면, 문장은 실행되고 그 조건문은 다시 판별됩니다. 만약 조건문이 거짓으로 리턴된다면, 실행을 멈추고 while문 바로 다음의 문장으로 넘어가게 됩니다.
- 많은 문장들을 실행하기 위해선, { }를 써서 문장들을 묶어줍니다.
- 예시 1

```
n = 0;
x = 0;
while (n < 3) {
  n++;
  x += n;
}
```

  - 매 반복과 함께, n이 증가하고 x에 더해집니다. 그러므로, x와 n은 다음과 같은 값을 갖습니다.
    - 첫번째 경과 후: `n` = 1 and `x` = 1

        - 두번째 경과 후: `n` = 2 and `x` = 3
        - 세번째 경과 후: `n` = 3 and `x` = 6
    - 세번째 경과 후에, n < 3 은 더이상 참이 아니므로, 반복문은 종결됩니다.
- 예시 2

```
// 다음과 같은 코드는 피하세요.
while (true) {
  console.log("Hello, world");
}
```

Label 문

- 여러분이 프로그램에서 다른 곳으로 참조할 수 있도록 식별자로 문을 제공합니다. 예를 들어, 여러분은 루프를 식별하기 위해 레이블을 사용하고, 프로그램이 루프를 방해하거나 실행을 계속할지 여부를 나타내기 위해 break나 continue 문을 사용할 수 있습니다.

```
label :
       statement
```

- 레이블 값은 예약어가 아닌 임의의 JavaScript 식별자일 수 있습니다. 여러분이 레이블을 가지고 식별하는 문은 어떠한 문이 될 수 있습니다.
- 예시

```
markLoop:
while (theMark == true) {
   doSomething();
}
//레이블 markLoop는 while 루프를 식별합니다.
```

break 문

- break문은 반복문, switch문, 레이블 문과 결합한 문장을 **빠져나올 때 사용**합니다.
    - 레이블 없이 break문을 쓸 때: 가장 가까운 `while`, `do-while`, `for`, 또는 `switch`문을 종료하고 다음 명령어로 넘어갑니다.
    - 레이블 문을 쓸 때: 특정 레이블 문에서 끝납니다.
- break문의 문법은 다음과 같습니다.

1. `break;`

2. `break [레이블];`

- break문의 첫번째 형식은 가장 안쪽의 반복문이나 switch문을 빠져나옵니다. 두번째 형식는 특정한 레이블 문을 빠져나옵니다.
- 예시1

```
for (i = 0; i < a.length; i++) {
  if (a[i] == theValue) {
    break;
  }
}
```

- 예시 2 : Breaking to a label

```
var x = 0;
var z = 0
labelCancelLoops: while (true) {
  console.log("Outer loops: " + x);
  x += 1;
  z = 1;
  while (true) {
    console.log("Inner loops: " + z);
    z += 1;
    if (z === 10 && x === 10) {
      break labelCancelLoops;
    } else if (z === 10) {
      break;
    }
  }
}
```

Continue 문

- `continue`문은 while, do-while, for, 레이블 문을 다시 시작하기 위해 사용될 수 있습니다.
  - 레이블없이 continue를 사용하는 경우, 그것은 가장 안쪽의 while, do-while, for 문을 둘러싼 현재 반복을 종료하고, 다음 반복으로 루프의 실행을 계속합니다. break 문과 달리, continue 문은 전체 루프의 실행을 종료하지 않습니다. while 루프에서 그것은 다시 조건으로 이동합니다. for 루프에서 그것은 증가 표현으로 이동합니다.
  - 레이블과 함께 continue를 사용하는 경우, continue는 그 레이블로 식별되는 루프문에 적용됩니다.
- continue 문의 구문은 다음과 같습니다:

1. `continue;`

2. `continue label;`

- 예시 1

```
i = 0;
n = 0;
while (i < 5) {
  i++;
  if (i == 3) {
    continue;
  }
  n += i;
}
//i 값이 3일 때 실행하는 continue 문과 함께 while 루프를 보여줍니다. 따라서, n은 값 1, 3, 7, 12를 취합니다.
```

- 예시 2

```
checkiandj:
  while (i < 4) {
    console.log(i);
    i += 1;
    checkj:
      while (j > 4) {
        console.log(j);
        j -= 1;
        if ((j % 2) == 0) {
          continue checkj;
        }
        console.log(j + " is odd.");
      }
      console.log("i = " + i);
      console.log("j = " + j);
  }
```

  - checkiandj 레이블 문은 checkj 레이블 문을 포함합니다. continue가 발생하는 경우, 프로그램은 checkj의 현재 반복을 종료하고, 다음 반복을 시작합니다. 그 조건이 false를 반환 할 때까지 continue가 발생할 때마다, checkj는 반복합니다. false가 반환될 때, checkiandj 문의 나머지 부분은 완료되고, 그 조건이 false를 반환 할 때까지 checkiandj는 반복합니다. false가 반환될 때, 이 프로그램은 다음 checkiandj 문에서 계속됩니다.
  - continue가 checkiandj의 레이블을 가지고 있다면, 프로그램은 checkiandj 문 상단에서 계속될 것입니다.

for… in 문

- for… in 문은 객체의 열거 속성을 통해 지정된 변수를 반복합니다. 각각의 고유한 속성에 대해, JavaScript는 지정된 문을 실행합니다. for...in 문은 다음과 같습니다:

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

- 예시

```
//다음 함수는 객체와 객체의 이름을 함수의 인수로 취합니다. 그런 다음 모든 객체의 속성을 반복하고 속성
 이름과 값을 나열하는 문자열을 반환합니다.
function dump_props(obj, obj_name) {
  var result = "";
  for (var i in obj) {
    result += obj_name + "." + i + " = " + obj[i] + "<br>";
  }
  result += "<hr>";
  return result;
}
```

  - 속성 make와 model을 가진 객체 car의 경우, 결과는 다음과 같습니다:

```
car.make = Ford
car.model = Mustang
```

- 배열

  - `배열`요소를 반복하는 방법으로 이를 사용하도록 유도될 수 있지만, **for...in** 문은 숫자 인덱스에 추가하여 사용자 정의 속성의 이름을 반환합니다. 따라서 만약 여러분이 사용자 정의 속성 또는 메서드를 추가하는 등 Array 객체를 수정한다면, 배열 요소 이외에도 사용자 정의 속성을 통해 **for...in** 문을 반복하기 때문에, 배열을 통해 반복할 때 숫자 인덱스와 전통적인 `for` 루프를 사용하는 것이 좋습니다.

for… of 문

- `for...of` 문은 각각의 고유한 특성의 값을 실행할 명령과 함께 사용자 지정 반복 후크를 호출하여, 반복 가능한 객체(`배열`, `Map` (en-US), `Set`, 인수 객체 등을 포함)를 통해 반복하는 루프를 만듭니다.

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

- 다음 예는 for...of 루프와 `for...in` 루프의 차이를 보여줍니다. 속성 이름을 통해 for...in 이 반복하는 동안, for...of은 속성 값을 통해 반복합니다:

```
let arr = [3, 5, 7];
arr.foo = "hello";

for (let i in arr) {
   console.log(i); // logs "0", "1", "2", "foo"
}

for (let i of arr) {
   console.log(i); // logs "3", "5", "7"
}
```

조건문

if … else 문

- 기본 if … else 문법

```
if (조건) {
  만약 조건(condition)이 참일 경우 실행할 코드
} else {
  대신 실행할 다른 코드
}
```

```
if (조건) {
  만약 조건(condition)이 참일 경우 실행할 코드
}

실행할 다른 코드
```

- 예시

```
let shoppingDone = false;
let childsAllowance;

if (shoppingDone === true) {
  childsAllowance = 10;
} else {
  childsAllowance = 5;
}
```

else if

- 두 가지보다 더 많은 것을 원할때 사용
- 예시

```
let year = prompt('ECMAScript-2015 명세는 몇 년도에 출판되었을까요?', '');

if (year < 2015) {
  alert( '숫자를 좀 더 올려보세요.' );
} else if (year > 2015) {
  alert( '숫자를 좀 더 내려보세요.' );
} else {
  alert( '정답입니다!' );
}
```

조건부 연산자 '?'

- 문법

```
let result = condition ? value1 : value2;
```

- 예시

```
let accessAllowed;
let age = prompt('나이를 입력해 주세요.', '');

if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}

alert(accessAllowed);
```

  - '물음표(question mark) 연산자'라고도 불리는 '조건부(conditional) 연산자'를 사용하면 위 예시를 더 짧고 간결하게 변형할 수 있습니다.
  - 조건부 연산자는 물음표`?`로 표시합니다. 피연산자가 세 개이기 때문에 조건부 연산자를 '삼항(ternary) 연산자'라고 부르는 사람도 있습니다. 참고로, 자바스크립트에서 피연산자를 3개나 받는 연산자는 조건부 연산자가 유일합니다.

```
let accessAllowed = (age > 18) ? true : false;
```

- `age > 18` 주위의 괄호는 생략 가능합니다. 물음표 연산자는 우선순위가 낮으므로 비교 연산자 `>`가 실행되고 난 뒤에 실행됩니다.
- 아래 예시는 위 예시와 동일하게 동작합니다.

```
// 연산자 우선순위 규칙에 따라, 비교 연산 'age > 18'이 먼저 실행됩니다.
// (조건문을 괄호로 감쌀 필요가 없습니다.)
let accessAllowed = age > 18 ? true : false;
```

- 괄호가 있으나 없으나 차이는 없지만, 코드의 가독성 향상을 위해 괄호를 사용할 것을 권유합니다.

주의:

비교 연산자 자체가 `true`나 `false`를 반환하기 때문에 위 예시에서 물음표 연산자를 사용하지 않아도 됩니다.

```
// 동일하게 동작함
let accessAllowed = age > 18;
```

method

map

- 배열의 각 요소에 대하여 주어진 함수를 수행한 결과를 모아 새로운 배열을 반환하는 메서드
- 예) 단위변환(섭씨→화씨), 숫자→문자 등
- 요소들에게 일괄적으로 함수를 적용하고 싶을 때 사용하기 적합
- 문법

```
const output = arr.map(function);
```

- 예제

```
const numbers = [1, 2, 3, 4, 5];
const numbersMap = numbers.map(val =&gt; val * 2);
```

```
console.log(numbersMap);

// [ 2, 4, 6, 8, 10 ]
```

```
const arr = [5, 1, 3, 2, 6];

// double
// [10, 2, 6, 4, 12]

// triple
// [15, 3, 9, 6, 18]

// binary
// ["101, "1", "11", "10", "110"]

function double(x) {
  return x * 2;
}

function triple(x) {
  return x * 3;
}

function binary(x) {
  return x.toString("2");
}

const output = arr.map(double);

const output2 = arr.map(triple);

const output3 = arr.map(binary);

console.log(output);
console.log(output2);
console.log(output3);
```

filter

- 배열의 각 요소에 대하여 주어진 함수의 결괏값이 true인 요소를 모아 새로운 배열을 반환하는 메서드
- filter함수는 React 코드에서 DB의 자료를 가지고 왔을 때 특정 조건에 해당되는 것만 추려낼 때 쓰는 아주 유용한 함수
- 문법

```
const arr = [5, 1, 3, 2, 6];
```

- 예제

```
const arr = [5, 1, 3, 2, 6];

// odd number
function isOdd(x) {
  if (x % 2) {
    return true;
  } else return false;
}

const output = arr.filter(isOdd);

console.log(output);
```

```
const arr = [5, 1, 3, 2, 6];

const output = arr.filter((x) => x % 2);

console.log(output);
```

  - arr 배열에서 홀수만 뽑아내고 싶을 때
  - x % 2 로직은 짝수면 0을 리턴하기 때문에 0은 자바스크립트에서는 false가 됩니다.

```
const arr = [5, 1, 3, 2, 6];

const output = arr.filter((x) => x % 2 === 0);

console.log(output);
```

  - 짝수만 뽑아내는 filter() 함수

```
let employees = [
  { id: 20, name: "Kim", salary: 30000, dept: "it" },
  { id: 24, name: "Park", salary: 35000, dept: "hr" },
  { id: 56, name: "Son", salary: 32000, dept: "it" },
  { id: 88, name: "Lee", salary: 38000, dept: "hr" },
];

let departmentList = employees.filter(function (record) {
  return record.dept == "it";
});
console.log(departmentList);
```

- 부서가 it 쪽인 사람만 필터링

reduce

- 배열 각 요소에 대하여 reducer 함수를 실행하고, 배열이 아닌 하나의 결괏값을 반환
- 문법

```
const output = arr.reduce(function (acc, curr) {
  acc = acc + curr;
  return acc;
}, 0);

console.log(output);
```

- 예제

```
const arr = [5, 1, 3, 2, 6];

// sum
function findSum(arr) {
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
    sum = sum + arr[i];
  }
  return sum;
}

console.log(findSum(arr));
```

```
const arr = [5, 1, 3, 2, 6];

// sum
function findMax(arr) {
  let max = 0;
  for (let i = 0; i < arr.length; i++) {
      if (arr[i] > max) {
          max = arr[i];
      }
  }
  return max;
}

console.log(findMax(arr));
```

Rest Operator

Rest Parameter

- Rest 파라미터는 Spread 연산자(...)를 사용하여 함수의 파라미터를 작성한 형태를 말한다. 즉, Rest 파라미터를 사용하면 함수의 파라미터로 오는 값들을 "배열"로 전달받을 수 있다.
- 문법

```
function f(a, b, ...theArgs) {
  // ...
}
```

- 예제

```
function sum(...theArgs) {
  let total = 0;
  for (const arg of theArgs) {
    total += arg;
  }
  return total;
}

console.log(sum(1, 2, 3));
// expected output: 6

console.log(sum(1, 2, 3, 4));
// expected output: 10
```

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}

myFun("one", "two", "three", "four", "five", "six");

// 콘솔 출력:
// a, one
// b, two
// manyMoreArgs, [three, four, five, six]

myFun("one", "two", "three")

// a, "one"
// b, "two"
// manyMoreArgs, ["three"] <-- 요소가 하나지만 여전히 배열임

myFun("one", "two")

// a, "one"
```

```
// b, "two"
// manyMoreArgs, [] <-- 여전히 배열
```

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(element => {
    return multiplier * element
  })
}

let arr = multiply(2, 15, 25, 42)
console.log(arr)  // [30, 50, 84]
```

```
function sortRestArgs(...theArgs) {
  let sortedArgs = theArgs.sort()
  return sortedArgs
}

console.log(sortRestArgs(5, 3, 7, 1)) // 1, 3, 5, 7

function sortArguments() {
  let sortedArgs = arguments.sort()
  return sortedArgs
}

console.log(sortArguments(5, 3, 7, 1))
// TypeError 발생 (arguments.sort is not a function)
```

- 나머지 매개변수와 `arguments` 객체의 차이
  - `arguments` 객체는 **실제 배열이 아닙니다**. 그러나 나머지 매개변수는 `Array` 인스턴스이므로 `sort`, `map`, `forEach`, `pop` 등의 메서드를 직접 적용할 수 있습니다.
  - `arguments` 객체는 `callee` 속성과 같은 추가 기능을 포함합니다.
  - `...restParam`은 후속 매개변수만 배열에 포함하므로 `...restParam` **이전**에 직접 정의한 매개변수는 포함하지 않습니다. 그러나 `arguments` 객체는, `...restParam`의 각 항목까지 더해 모든 매개변수를 포함합니다.