JavaScript Objects

개체에는 많은 값이 포함될 수 있습니다.
JavaScript 개체는 속성 및 메서드라고 하는 명명된 값의 컨테이너입니다.

```
const car = {type:"Fiat", model:"500", color:"white"};
```

```
<p id="demo"></p>
<script>
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>
```

개체는 메소드를 가질 수도 있습니다.
메서드는 개체에 대해 수행할 수 있는 작업입니다.
메서드는 함수 정의로 속성에 저장됩니다.

```
const person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

```
<p id="demo"></p>
<script>
const person = {};
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>
```

```
<p id="demo"></p>
<script>
const person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

```
document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>
```

JavaScript Arrays

배열은 둘 이상의 값을 보유할 수 있는 특수 변수입니다.
배열은 단일 이름으로 많은 값을 보유할 수 있으며 색인 번호를 참조하여 값에 액세스할 수 있습니다.
const 키워드로 배열을 선언하는 것이 일반적입니다.
배열 리터럴을 사용하는 것이 JavaScript 배열을 만드는 가장 쉬운 방법입니다.
Javascript에서 배열 리터럴은 표현식 목록으로, 각 표현식은 한 쌍의 대괄호 '[ ] '로 묶인 배열 요소를 나타냅니다.

Syntax:

```
const array_name = [item1, item2, ...];
const points = new Array();
const points = [];

const cars = ["Saab", "Volvo", "BMW"];
let car1 = "Saab";
let car2 = "Volvo";
let car3 = "BMW";

const cars = [
  "Saab",
  "Volvo",
  "BMW"
];

const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";

const cars = new Array("Saab", "Volvo", "BMW");
```

자바스크립트에는 내장 배열 생성자 new Array()가 있습니다.
단순성, 가독성 및 실행 속도를 위해 배열 리터럴 방법을 사용하십시오.

색인 번호를 참조하여 배열 요소에 액세스합니다.
배열 인덱스는 0부터 시작합니다.

[0]은 첫 번째 요소입니다. [1]은 두 번째 요소입니다.

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
```

배열 이름을 참조하여 전체 배열에 액세스할 수 있습니다.

```
<p id="demo"></p>

<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>
```

자바스크립트 typeof

1. 자바 스크립트에는 값을 포함 할 수있는 5 가지 데이터 유형이 있습니다.
   - string
   - number
   - boolean
   - object
   - function
2. 객체에는 6 가지 유형이 있습니다.
   - Object
   - Date
   - Array
   - String
   - Number
   - Boolean
3. 값을 포함 할 수없는 2 가지 데이터 유형 :
   - null
   - undefined

```
typeof "John"                 // Returns "string"
typeof 3.14                   // Returns "number"
typeof NaN                    // Returns "number"
typeof false                  // Returns "boolean"
typeof [1,2,3,4]              // Returns "object"
typeof {name:'John', age:34}  // Returns "object"
typeof new Date()             // Returns "object"
typeof function () {}         // Returns "function"
typeof myCar                  // Returns "undefined" *
typeof null                   // Returns "object"
```

- NaN의 데이터 유형은 숫자입니다.
- 배열의 데이터 유형은 object입니다.
- 날짜의 데이터 유형은 객체입니다.
- null의 데이터 유형은 객체입니다.
- 정의되지 않은 변수의 데이터 형식 undefined. *
- 값이 할당되지 않은 변수의 데이터 형식 undefined

typeof 연산자는 다음 기본 형식 중 하나를 반환할 수 있습니다.
- string
- number
- boolean
- undefined
- function
- object

typeof 연산자는 개체, 배열 및 null에 대해 "object"를 반환합니다.

```
const fruits = ["Banana", "Orange", "Apple"];
let type = typeof fruits;
```

instanceof 연산자는 개체가 지정된 개체의 인스턴스인 경우 true를 반환합니다.

```
const cars = ["Saab", "Volvo", "BMW"];

document.write(
(cars instanceof Array) + "<br>" +
```

```
(cars instanceof Object) + "<br>" +
(cars instanceof String) + "<br>" +
(cars instanceof Number));
```

void연산자

```
<a href="javascript:void(0);">
  Useless link
</a>
```

배열을 문자열로 변환

자바스크립트 toString()메서드는 배열을 쉼표로 구분된 배열 값의 문자열로 변환합니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.toString());
```

join()메서드는 모든 배열 요소를 문자열로 조인합니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits.join(" * "));
```

pop()메서드는 배열에서 마지막 요소를 제거합니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop();
document.write(fruits);
```

push()메서드는 배열에 새 요소를 추가합니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");
document.write(fruits);
```

shift() 메서드는 배열의 첫 번째 요소를 제거하고 다른 요소를 왼쪽으로 "이동"합니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits.shift();
document.write(fruits);
```

unshift() 메서드는 배열의 시작 부분에 새 요소를 추가합니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");
document.write(fruits);
```

length 속성은 배열의 길이(배열 요소의 수)를 반환합니다.

길이 속성은 항상 가장 높은 배열 인덱스보다 하나 더 큽니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.length;
document.write(fruits);
```

concat()메서드는 병합 (연결)하여 새 배열을 만듭니다.

```
const myGirls = ["Cecilie", "Lone"];
const myBoys = ["Emil", "Tobias", "Linus"];
const myChildren = myGirls.concat(myBoys);
document.write(myChildren);
```

```
const array1 = ["Cecilie", "Lone"];
const array2 = ["Emil", "Tobias", "Linus"];
const array3 = ["Robin", "Morgan"];
const myChildren = array1.concat(array2, array3);
document.write(myChildren);
```

splice() 메서드는 배열에 새 항목을 추가하는데 사용할 수 있습니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.write(fruits);

fruits.splice(2, 0, "Lemon", "Kiwi");
document.write(fruits);
```

첫 번째 매개 변수 (2)는 새 요소를 추가 (접합)해야하는 위치를 정의합니다.
두 번째 매개 변수(0)는제거 해야 하는 요소 수를 정의합니다.
나머지 매개 변수 ("Lemon"", "Kiwi"")는추가 할 새 요소를 정의합니다.
splice() 메서드는 삭제된 항목이 있는 배열을 반환합니다.

```
fruits.splice(2, 2, "Lemon", "Kiwi");
```

첫 번째 매개 변수 (2)는 새 요소를 추가 (접합)해야하는 위치를 정의합니다.
두 번째 매개 변수(2)는 제거해야 하는 요소 수를 정의합니다.

slice() 메서드는 배열의 일부를 새 배열로 분할합니다.

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1, 3);
```

sort() 메서드는 배열을 알파벳순으로 정렬합니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
```

reverse() 메서드는 배열의 요소를 반대로 바꿉니다.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.reverse();
```

forEach() 메서드는 각 배열 요소에 대해 한 번씩 함수(콜백 함수)를 호출합니다.
이 함수는 3 개의 인수를 사용합니다.

- 항목 값(`value`)
- 항목 인덱스(`index`)
- 배열 자체(`array`)

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
numbers.forEach(myFunction);
document.write(txt);
```

```
function myFunction(value, index, array) {
  txt += value+ "<br>";
}
```

indexOf() 메서드는 배열에서 요소 값을 검색하고 해당 위치를 반환합니다.
첫 번째 항목의 위치 0, 두 번째 항목의 위치 1 등입니다.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.indexOf("Apple");
```

array.indexOf(item, start)

item	검색할 항목입니다. 필수
start	검색을 시작할 위치. 음수 값은 주어진 위치에서 시작하여 끝에서 세고 끝까지 검색합니다.

Array.indexOf()항목을 찾을 수 없는 경우 -1을 반환합니다.
항목이 두 번 이상 있으면 첫 번째 위치를 반환합니다.

Array.lastIndexOf() 지정된 요소의 마지막 발생 위치를 반환합니다.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.lastIndexOf("Apple")
```

JavaScript에서 배열은 번호가 매겨진 인덱스를 사용합니다.
JavaScript에서 개체는 명명된 인덱스를 사용합니다.

Object.keys() 메서드는 객체의 키가 있는 Array Iterator(배열 반복자) 객체를 반환합니다.

Syntax
Object.keys(object)

```
Use Object.keys() on an array:
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const keys = Object.keys(fruits);

Use Object.keys() on a string:
const fruits = "Banana";
const keys = Object.keys(fruits);

Use Object.keys() on an object:
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
const keys = Object.keys(person);
```

instanceof 연산자는 객체가 주어진 생성자에 의해 생성된 경우 true를 반환합니다.

```
<p id="demo"></p>

<script>
var fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = fruits instanceof Array;
</script>
```

자바스크립트 this 키워드

this는 변수가 아닙니다. 키워드입니다. this의 값은 변경할 수 없습니다.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
document.write(person.fullName());
```

자바스크립트 조건문

JavaScript if, else, and else if

조건문은 다양한 조건에 따라 다양한 작업을 수행하는 데 사용됩니다.
if를 사용하여 지정된 조건이 참인 경우 실행할 코드 블록을 지정합니다.
동일한 조건이 거짓인 경우 실행할 코드 블록을 지정하려면 else를 사용하십시오.
첫 번째 조건이 거짓인 경우 else if를 사용하여 테스트할 새 조건을 지정합니다.
실행할 코드의 많은 대체 블록을 지정하려면 스위치를 사용하십시오.
if 문을 사용하여 조건이 참인 경우 실행할 JavaScript 코드 블록을 지정합니다.
else 문을 사용하여 조건이 거짓인 경우 실행할 코드 블록을 지정합니다.

```
if (hour < 18) {
  greeting = "Good day";
}
if (조건 {
  // 조건이 참이면 실행할 코드 블록
} else {
  // 조건이 거짓일 때 실행할 코드 블록
}
```

첫 번째 조건이 거짓이면 else if 문을 사용하여 새 조건을 지정합니다.

```
if (조건1) {
  // 조건1이 참일 경우 실행할 코드 블록
} else if (조건2) {
  // 조건1이 false이고 조건2가 true인 경우 실행될 코드 블록
} else {
  // 조건1이 false이고 조건2가 false인 경우 실행될 코드 블록
}
```

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

JavaScript Switch Statement

switch 문은 다른 조건에 따라 다른 작업을 수행하는 데 사용됩니다.
실행할 많은 코드 블록 중 하나를 선택하려면 switch 문을 사용하십시오.

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

switch 식은 한 번 평가됩니다.
식의 값은 각 경우의 값과 비교됩니다.

일치하는 항목이 있으면 연결된 코드 블록이 실행됩니다.
일치하는 항목이 없으면 기본 코드 블록이 실행됩니다.

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
    day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

JavaScript For Loop

Loop는 코드 블록을 여러 번 실행할 수 있습니다.
매번 다른 값으로 동일한 코드를 반복해서 실행하려는 경우 편리합니다.

```
text += cars[0] + "<br>";
text += cars[1] + "<br>";
text += cars[2] + "<br>";
text += cars[3] + "<br>";
text += cars[4] + "<br>";
text += cars[5] + "<br>";
```

```
for (let i = 0; i <cars.length; i++) {
  text += cars[i] + "<br>";
}
```

```
for (표현식1; 표현식2; 표현식3) {
  // code block to be executed
}
```

for (let i = 0; i < arr.length; i++) {	X
let n = arr.length; for (let i = 0; i < n; i++) {	O

잘못된 코드는 루프가 있을 때마다 배열의 length 속성에 반복 액세스합니다.
더 나은 코드는 루프 외부의 length 속성에 액세스하여 루프 실행 속도가 빨라집니다.

표현식 1은 코드 블록 실행 전에 실행됩니다(한 번).
루프가 시작되기 전에 변수를 설정합니다(let i = 0).
표현식 2는 코드 블록을 실행하기 위한 조건을 정의합니다.
루프 실행 조건을 정의합니다(i는 5보다 작아야 함).
표현식 3은 코드 블록이 실행된 후 (매번) 실행됩니다.
루프의 코드 블록이 실행될 때마다 값(i++)을 증가시킵니다.

```
for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

```
var i = 5;

for (var i = 0; i < 10; i++) {
  실행문
}
```

var를 사용하여 루프에서 선언된 변수가 루프 외부에서 변수를 다시 선언합니다.

```
// Here i is 10
```

```
let i = 5;

for (let i = 0; i < 10; i++) {
  // some code
}

// Here i is 5
```

루프에서 i 변수를 선언하는 데 let을 사용하면 i 변수는 루프 내에서만 볼 수 있습니다.

JavaScript For In

JavaScript for in 문은 개체의 속성을 반복합니다.

```
for (key in object) {
  // 실행할 코드 블록
}
```

```
<p id="demo"></p>

<script>
const person = {fname:"John", lname:"Doe", age:25};

let txt = "";
for (let x in person) {
  txt += person[x] + " ";
}

document.getElementById("demo").innerHTML = txt;
```

For In Over Arrays

JavaScript for in 문은 배열의 속성을 반복할 수도 있습니다.

```
for (variable in array) {
  code
}
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
   txt += numbers[x] + "<br>";
}

document.getElementById("demo").innerHTML = txt;
```

JavaScript While Loop

루프는 지정된 조건이 참인 한 코드 블록을 실행할 수 있습니다.

```
while (조건) {
  // 실행할 코드 블록
}
<script>
let text = "";
let i = 0;
while (i < 10) {
   text += "<br>The number is " + i;
   i++;
}
document.wrie(text);
</script>
```

The Do While Loop

do while 루프는 while 루프의 변형입니다. 이 루프는 조건이 참인지 확인하기 전에 코드 블록을 한 번 실행한 다음 조건이 참인 동안 루프를 반복합니다.
do while 루프는 while 루프의 변형입니다. 이 루프는 조건이 참인지 확인하기 전에 코드 블록을 한 번 실행한 다음 조건이 참인 동안 루프를 반복합니다.

```
<script>
let text = ""
let i = 0;

do {
   text += "<br>The number is " + i;
   i++;
}
while (i < 10);
document.write(text);
```

JavaScript Break and Continue

break 문을 사용하여 루프에서 빠져나올 수도 있습니다.

```
<script>
let text = "";
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += "The number is " + i + "<br>";
}
document.text);
</script>
```

break 문은 루프 카운터(i)가 3일 때 루프를 종료합니다(루프를 "중단").

The Continue Statement

continue 문은 지정된 조건이 발생하는 경우 (루프에서) 하나의 반복을 중단하고 루프의 다음 반복을 계속합니다.

```
<script>
let text = "";
for (let i = 0; i < 10; i++) {
  if (i === 3) { continue; }
  text += "The number is " + i + "<br>";
}
document.write(text);
</script>
```

JavaScript Functions

JavaScript 함수는 특정 작업을 수행하도록 설계된 코드 블록입니다.
JavaScript 함수는 "무언가"가 함수를 호출(호출)할 때 실행됩니다.

JavaScript Function Syntax

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

세미콜론은 실행 가능한 JavaScript 문을 구분하는 데 사용됩니다.
함수선언은 실행 가능한 문이 아니기 때문에 세미콜론으로 끝내는 것은 일반적이지 않습니다.
JavaScript 함수는 function 키워드, 이름, 괄호()로 정의됩니다.
함수 이름에는 문자, 숫자, 밑줄 및 달러 기호(변수와 동일한 규칙)가 포함될 수 있습니다.

괄호는 쉼표로 구분된 매개변수 이름을 포함할 수 있습니다.
(매개변수1, 매개변수2, ...)
함수 호출에서 매개변수는 함수의 인수입니다.
JavaScript 인수는 값 으로 전달됩니다 .
함수는 인수의 위치가 아닌 값만 알게 됩니다.
함수가 인수의 값을 변경하더라도 매개변수의 원래 값은 변경되지 않습니다.
인수에 대한 변경 사항은 함수 외부에서 표시(반영)되지 않습니다.

함수에 의해 실행될 코드는 중괄호{} 안에 배치됩니다
함수 인수는 호출될 때 함수가 받는 값입니다.
함수 내에서 인수(매개 변수)는 지역 변수로 작동합니다.

함수 호출
1. 이벤트 발생 시(사용자가 버튼을 클릭할 때)
2. JavaScript 코드에서 호출(호출)될 때
3. 자동으로(자체 호출)

Function Return - 함수 반환

JavaScript가 return 문에 도달하면 함수 실행이 중지됩니다.

함수가 문에서 호출된 경우 JavaScript는 호출 문 다음에 코드를 실행하기 위해 "반환"합니다.

함수는 종종 반환 값을 계산합니다. 반환 값은 "호출자"에게 다시 "반환"됩니다.
함수 안에서 return을 만나게 되면 해당 함수를 호출한 곳으로 결과 데이터를 반환해 주고 함수는 종료가 된다.

함수 선언
선언된 함수는 즉시 실행되지 않습니다. 그것들은 "나중에 사용하기 위해 저장"되며 나중에 호출될 때 실행됩니다.

```
function myFunction(a, b) {
  return a * b;
}
```

함수 표현식
표현식 을 사용하여 JavaScript 함수를 정의할 수도 있습니다 .
함수 표현식은 변수에 저장할 수 있습니다.

```
const x = function (a, b) {return a * b};
```

함수 표현식이 변수에 저장된 후 변수를 함수로 사용할 수 있습니다.

```
const x = function (a, b) {return a * b};
let z = x(4, 3);
```

변수에 저장된 함수에는 함수 이름이 필요하지 않습니다. 항상 변수 이름을 사용하여 호출(호출)됩니다.

자체 호출 함수
함수 표현식은 "자기 호출"로 만들 수 있습니다.
자체 호출 식은 호출되지 않고 자동으로 호출(시작)됩니다.
표현식 뒤에 ()가 있으면 함수 표현식이 자동으로 실행됩니다.
함수 선언을 자체 호출할 수 없습니다.
함수식임을 나타내려면 함수 주위에 괄호를 추가해야 합니다.

```
<p id="demo"></p>

<script>
(function () {
  document.getElementById("demo").innerHTML = "Hello! I called myself";
})();
</script>
```

실제로 익명의 자체 호출 함수 (이름 없는 함수)입니다.

JavaScript 함수는 값으로 사용할 수 있습니다.

```
<p id="demo"></p>

<script>
function myFunction(a, b) {
  return a * b;
}
let x = myFunction(4, 3);
document.getElementById("demo").innerHTML = x;
</script>
```

함수는 객체입니다
arguments.length함수가 호출될 때 받은 인수 수를 반환합니다.

```
<p id="demo"></p>

<script>
function myFunction(a, b) {
  return arguments.length;
}
document.getElementById("demo").innerHTML = myFunction(4, 3);
</script>
```

함수를 사용하면 코드를 재사용할 수 있고, 코드를 한 번 정의하면 여러 번 사용할 수 있습니다. 다른 인수로 동일한 코드를 여러 번 사용하여 다른 결과를 생성할 수 있습니다.

```
<p id="demo"></p>
```

```
<script>
function myfnc(f) {
  return (5/9) * (f-32);
}
document.getElementById("demo").innerHTML = myfnc(77);
document.getElementById("demo").innerHTML = myfnc;
</script>
```

() 연산자가 함수를 호출합니다.
위의 예에서 myfnc는 함수 객체를 참조하고 myfnc()는 함수 결과를 참조합니다.

() 없이 함수에 액세스하면 함수 결과 대신 함수 개체가 반환됩니다.

전역변수와 지역변수

변수는 함수 블록{}을 기준으로 변수의 선언 위치에 따라 '전역 변수'와 '지역변수'로 나뉜다.
JavaScript 함수 내에서 선언된 변수는 함수에 대해 LOCAL이 됩니다.

지역 변수는 함수 내에서만 액세스할 수 있습니다.
전역 변수는 함수 블록{} 밖이나 안에서 자유롭게 사용 가능하지만 지역변수는 함수 블록{} 내에서만 사용할 수 있다.

전역변수	지역변수

let 변수;
function 함수() {
}
	function 함수() {
	let 변수;
	}