**Javascript**

**1. 자바스크립트 실습 환경**

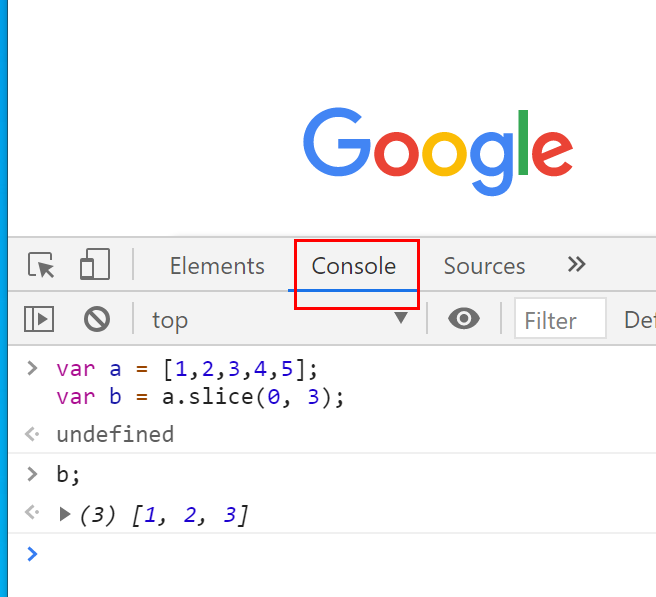
**1.1 웹브라우저 콘솔**

별도 파일을 생성하지 않는 간단한 실습시 사용

웹브라우저도 일종의 자바스크립트 런타임이므로 자바스크립드를 실행할 수 있습니다.

실습의 통일성을 위해 “크롬” 웹브라우저를 사용합니다.

크롬 -> F12 -> Console 탭 -> 스크립트 코드 테스트



**1.2 node.js**

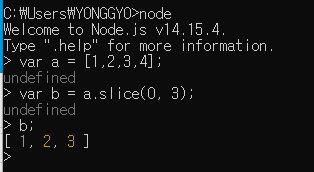
node.js도 일종의 자바스크립트 런타임입니다. node.js는 크롬에 탑재된

Chrome V8 Javascript 엔진을 사용합니다.

자바스크립트 런타임이란 단순히 탑재된 자바크립트 엔진을 사용하여 실행하는 환경을 제공해 주는 프로그램이라고 생각할 수 있습니다.

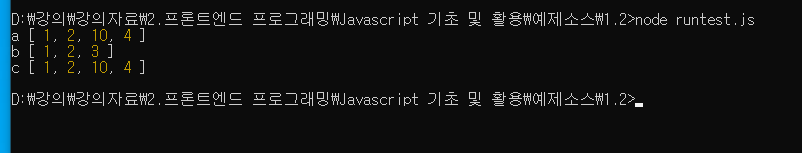
node.js에서도 자바스크립트를 실행할 수 있습니다.

**방법1. 콘솔(REPL(Read Eval Print Loop))에서 실행하는 방법**



**방법2. js 파일로 실행하는 방법**

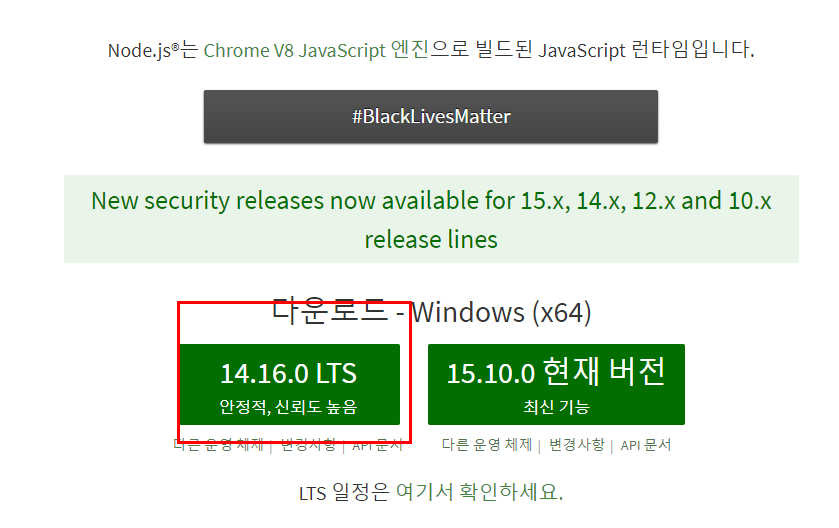
node 파일명.js



**1.3 node.js 설치**

하기 URL에서 LTS버전을 선택하여 설치합니다.

[https://nodejs.org/ko/](file://https:)



**1.4 HTML문서 파일로 작성하여 브라우저에서 확인**

**2. 자바스크립트 기초 문법**

**2.1 자바스크립트 사용하는 방법**

\* <script>~</script> 태그 안쪽에 소스코드를 넣어서 사용하는 방법

<script>

alert(”안녕하세요.”);

</script>

\*<script> 태그에 src 속성에 외부 javascript를 불러와서 사용하는 방법

<script src=”js/script.js”></script>

\* HTML 요소의 이벤트 속성에 정의 하는 방법**(권장 하지 않음)**

<div onclick=”alert('안녕하세요');”>인사</div>

**2.2 자바스크립트 주석 처리**

한줄 주석

//

여러줄 주석

/\* ~ \*/

**2.3 변수**

**2.3.1 정의**

변수는 값을 담기 위해 이름을 붙인 상자

컴퓨터의 메모리에 일정한 크기의 영역으로 생성됩니다.

**2.3.2 변수선언**

자바스크립트는 런타임시에 자료형이 결정되는 동적타입 변수입니다.

따라서 변수에는 자료형을 명시하지 않고 var, let, const 선언자만 사용합니다.

**2.3.3 선언자 종류**

**var 선언자**

일반적인 변수 선언자 입니다.

**let 선언자**

블록 유효 범위를 가지는 지역변수를 선언합니다.

let x = "outer x";

{

let x = "inner x";

let y = "inner y";

console.log(x); // inner x

console.log(y); // inner y

}

console.log(x); // outer x

console.log(y); // ReferenceError: y is not defined

**const 선언자**

블록 유효 범위를 가지면서 한 번만 할당 할수 있는 변수(상수)를 선언합니다.

const로 선언한 상수는 let문으로 선언한 변수처럼 동작합니다.

단, 반드시 초기화해야 한다는 차이점이 있습니다.

const 문으로 선언한 변수에 다시 대입을 시도하면 타입 오류가 발생합니다.

const c = 2;

c = 5; // Uncaught TypeError

const 문으로 선언한 상수 값은 수정할 수 없지만, 상수 값이 객체이거나 배열일 경우에는 프로퍼티 또는 프로퍼티 값을 수정할 수 있습니다.

**2.3.4 선언 방법**

선언자 변수명; (선언만) 예) var sum;

선언자 변수명 = 값; (선언후 초기화) 예) var sum = 10;

쉼표를 사용하면 변수 여러개를 한문장으로 선언할 수 있습니다.

예) var sum, a;

변수를 선언하기만 하면 변수 안에는 정의되지 않았음을 뜻하는 undefined라는 값이

들어갑니다.

var x;

console.log(x); // -> undefined

대입연산자(=)를 사용하면 변수에 값을 대입할 수 있습니다.

예)

var x = 5;

var a = 1, b = 2, c = 3;

**2.3.5 변수 선언 생략**

var 문으로 선언하지 않은 변수 값을 읽으려고 시도하면 참조 오류가 발생

console.log(x); // -> ReferenceError : x is not defined(오류 메시지)

그러나 var 문으로 선언하지 않은 변수에 값을 대입할 때는 오류가 발생하지 않음

x = 2;

console.log(x); // 2

- 변수를 선언하지 않은 상태에서 값을 대입하면 자바스크립트 엔진이 그 변수를 자동으로 전역 변수로 선언하게 됨

- 변수를 선언하지 않고 변수를 사용하는 행위는 오류의 원인이 될 수 있습니다.

**2.3.6 변수 끌어올림과 변수 중복 선언**

프로그램은 작성한 순서에 따라 윗줄 부터 차례대로 실행됩니다. 하지만 변수 선언은 이 원칙을 따르지 않습니다.

console.log(x); // -> undefined

var x;

이 코드에서 1번줄은 아직 변수 x가 선언되지 않았기 때문에 오류가 발생할 것 같지만 실제로는 오류가 발생하지 않고 undefined가 출력됩니다.

프로그램 중간에서 변수를 선언하더라도 변수가 프로그램 첫머리에 선언된 것 처럼 다른 문장 앞에 생성되기 때문입니다.

이를 변수 선언의 끌어올림(호이스팅 hoisting)이라고 합니다.

단 선언과 동시에 대입하는 코드는 끌어올리지 않습니다.

예)

console.log(x); // -> undefined

var x = 5;

console.log(x) // -> 5

**2.3.6 변수의 명명규칙**

\* 사용할 수 있는 문자는 알파벳(a-z, A-Z), 숫자(0--9), 밑줄(\_), 달러 기호($)

\* 첫 글자로는 숫자를 사용할 수 없다. 즉 첫 글자는 알파벳(a-z, A-Z), 밑줄(\_), 달러기호($)중 하나

\* 예약어를 식별자로 사용할 수 없다

사용가능 예)

key, sum1, \_name, $width, sum\_all, sumAll, newVale

사용불가 예)

1st - 첫 글자가 숫자이므로

sum-all (하이픈은 사용할 수 없읍)

new(예약어)

예약어 예시

[https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Lexical\_grammar](file://https:)

(예약어 부분 참조)

**2.3.7 변수에 저장할 수 있는 자료형**

**\* 문자형(String)**

var 변수 = “사용할 문자나 숫자”;

**\* 숫자형(Number)**

var 변수 = 숫자; 또는 Number(”문자와 숫자”);

**\* 논리형(Boolean)**

var 변수 = true 또는 false; Boolean(데이터);

false - false, 0, null, '', undefined

**\* null**

변수의 값이 비어 있다는 것을 표시할 경우

**\* undefined**

변수가 선언되었을때 값이 지정되지 않았을 경우

(변수를 선언하면 기본값은 undefined 입니다.)

\* **자료형 체크**

typeof 변수 또는 데이터

**\* 객체(Object)**

**\* 원시값 -** 객체 이외의 값(숫자, 문자, 논리값, null, undefined, “”(빈값)

**2.4 연산자**

**2.4.1 산술연산자**

더하기(+), 빼기(-), 곱하기(\*), 나누기(/), 나머지(%)

**2.4.2 문자 결합 연산자**

\* 여러개의 문자를 하나의 문자형 데이터로 결합할 때 사용

\*더하기에 피연산자로 문자형 데이터가 한 개라도 포함되어 있으면 다른 피연산자의 데이터는 자동으로 문자형 데이터로 형 변환되고 문자 결합이 이루어져 하나의 문자형 데이터로 반환

“Text1 “ + “Text2” = “Text1 Text2”;

“100” + 200 = “100200;

100 + “200” = 100200;

**2.4.3 대입연산자**

연산된 데이터를 변수에 저장할 때 사용

A = B

A = 1 + 2

**복합대입연산자** - 산술연산자 + 대입연산자

|  |  |
| --- | --- |
| A += B | A = A + B |
| A \*= B | A = A \* B |
| A /= B | A = A / B |
| A %= B | A = A % B |

**문자형 데이터 결합**

var str = "<table border='1'>";

str += "<tr>";

str += "<td>1</td><td>2</td><td>3</td>";

str += "</tr>";

str += "</table>";

document.body.innerHTML = str;

**2.4.4 증감연산자**

\* 증가연산자(++) - 숫자형 데이터를 1씩 증가

변수++; 또는 ++변수;

\* 감소연산자(--) - 숫자형 데이터를 1씩 감소

변수--; 또는 --변수;

var a = ++b; b를 1 증가 시킨 후 a에 대입

var a = b++; a에 대입 후 b를 1 증가

**2.4.5 비교연산자**

두 데이터를 '크다', '작다', '같다'와 같이 비교할때 사용

연산된 결과는 논리형데이터(true - 참, false - 거짓)

|  |  |  |
| --- | --- | --- |
| 종류 | 설명 |  |
| A > B |  |  |
| A < B |  |  |
| A >= B |  |  |
| A <= B |  |  |
| A == B | A와 B는 같다 | 데이터 일치 여부만 체크( 10 == “10” -> true) |
| A != B | A와 B는 다르다 | 데이터 불일치 여부만 체크( 10 != “10” -> false) |
| A === B | A와 B는 같다 | 데이터 일치 + 데이터 종류 일치 여부 체크 (10 === “10” --> false) |
| A !== B | A와 B는 다르다 | 데이터 불일치 + 데이터 종류 불일치 여부 체크 (10 !== “10” --> true) |

**2.4.6 논리연산자**

|  |  |
| --- | --- |
| || | OR 연산자, 비교하는 대상 중 하나라도 true면 true가 됨 |
| && | AND 연산자, 비교하는 대상 모두 true일때 true, 그렇지 않다면 false |
| ! | not 연산자, 피 연산자의 값을 반대로 바꿈, true -> false, false -> true |

**2.4.7 연산자 우선순위**

아래에서 위로 우선순위가 높아짐

1. ()

2. 단한 연산자(--, ++, !)

3. 산술 연산자(\*, /, %, +, -)

4. 비교 연산자(>, >=, <, <=, ==, ===, !=, !==)

5. 논리 연산자(&&, ||)

6. 대입(복합 대입)연산자(=, +=, -=, \*=, /=, %=)

예) ++A \* B <= C

단항 연산자 ++A

\* B

C와 비교(<=)

**2.4.8 삼항 조건 연산자**

조건식? 조건식이 true일때 : 조건식이 false 일때

var a = (b > 2)?”2 보다 크다”:”2보다 작거나 같다”;

**2.5 제어문**

**2.5.1 조건문**

조건식의 값이 참(true), 거짓(false)인지에 따라 제어

if (조건식) {

// 처리할 내용

}

if (조건식) {

} else { // 조건식이 false 일때

// 처리할 내용

}

if (조건식1) {

} else if (조건식2) { // 조건식1이 false일때

} else if (조건식3) { // 조건식1, 조건식2가 false 일때

} else { // 조건식1, 조건식2, 조건식3이 false 일때

}

**\* 조건식에서 false가 되는 기타 데이터**

0, null, “”(빈문자), null, undefined는 false로 인식을 하고 그 이외의 값은 true로 인식

**\* 조건식을 여러개 사용할 경우 논리연산자 사용**

조건식1|| 조건식2 -> 조건식1이 참이거나 조건식2가 참일때

조건식1 && 조건식2 -> 조건식1과 조건식2가 모두 참일때

**\* if문 안에 if 문을 중첩해서 사용할 수 있다**

**2.5.2 선택문**

여러개(case) 중에서 하나를 선택

var 변수 = 초기값;

switch (변수) {

case 값1 : 코드1;

break;

case 값2 : 코드2;

break;

case 값3 : 코드3;

break;

case 값4 : 코드4;

break;

default : 코드5;

}

변수에 할당된 값이 case의 각 값에 매칭되면 매칭된 코드가 실행 됩니다.

최종적으로 매칭되는 값이 없는 경우는 default 부분의 코드5가 실행됩니다.

각 case에서 break가 없다면 다음 case로 넘어값니다.

case 값1: 코드1;

case 값2: 코드2;

break

값1로 매칭이 되면 코드1, 코드2가 실행이 됩니다.

**2.6.3 반복문**

**2.6.3.1 while 문**

while (조건식) {

// 조건식이 true일때 반복 실행되는 부분

}

while구분에서는 반복 구간을 탈출할 수 있는 조건을 반드시 구현해야 합니다.

(없을 경우 무한 loop)

**탈출 키워드 (while, do~while, for)**

**break** -> 반복을 종료

**continue** -> 현재 반복 실행을 종료 하고 다음 반복 실행으로 넘어감

var num = 0;

while (num < 1000) {

if (num >= 100) break; // 반복횟수가 100이상이면 멈춤

num++;

}

**2.6.3.2 do ~ while 문**

do { }로 정의된 반복 실행을 적어도 1번은 실행하고 while 조건식에 따라 반복

var 변수 = 초기값;

do {

// 최소 한번 이상 실행되는 반복 처리

} while(조건식)

var num = 10;

do {

console.log(num); // 10

num++;

} while (num < 10)

**2.6.3.3 for문**

while, do~while과 마찬가지로 조건식을 만족하면 반복을 합니다. 다만 for문은 초기값, 증감식, 조건식을 통한 일정 구간 반복

for(초기값; 조건식; 증감식) {

// 반복 처리

}

var total = 0;

for (var i = 0; i < 100; i++) {  
 total += i;  
}

**2.6.3.4 중첩반복문**

반복문도 조건문(if)와 마찬가지로 중첩해서 반복할 수 있습니다.

예)

for(초기값; 조건식; 증감식) {

for (초기값; 조건식; 증감식) {

// 반복 처리 코드

}

}

**3. 객체**

**3.1 객체의 기초**

**3.1.1 객체 리터럴**

\* 객체는 이름과 값을 한쌍을 묶은 데이터를 여러개 모은 것

\* 즉, 객체는 데이터 여러개를 하나로 모은 복합 데이터로 **연관배열** 또는 **사전(Diction)**

이라고 부릅니다.

|  |  |
| --- | --- |
| 이름 | 값 |
| suit | “하트” |
| rank | “A” |

\* 객체에 포함된 데이터 하나(이름과 값의 쌍)를 가리켜 객체의 **프로퍼티**라고 부른다

\* 프로퍼티 이름 부분을 “**프로퍼티 이름”** 또는 “**키”**라고 부릅니다.

\* 객체를 생성하는 방법 2가지

객체 리터럴 사용, 생성자 함수 사용

**3.1.2 객체 리터럴로 객체 생성하기**

var card = { suit : “하트”, rank : “A” };

\* { ... } 부분이 객체 리터럴 이며 객체 리터럴을 변수 card에 대입하고 있음

\* 프로퍼티 값은 suite : “하트” 처럼 콜론(:)을 사용해서 구분

\* 중괄호({})안에 있는 프로퍼티들은 쉼표(,)로 구분

\* 변수에 대입된 객체 안의 프로퍼티 값을 읽거나 쓸 때에는 **마침표(.) 연산자** 또는

대괄호 ([]) 연산자를 사용

card.suit // -> 하트

card['rank'] // -> A

\* 객체에 없는 프로퍼티를 읽으려고 시도하면 undefined를 반환

card.color // -> undefined

\* 객체 리터럴 안에 어떠한 프로퍼티도 작성하지 않으면 빈 객체가 생성됨

var obj = {};

console.log(obj); // -> Object{}

**3.1.3 프로퍼티 추가와 삭제**

\* 없는 프로퍼티 이름에 값을 대입하면 새로운 프로퍼티가 추가됨

card.value = 14;

console.log(card); // Object {suit : “하트”, rank : “A”, value : 14 }

\* delete 연산자를 사용하여 프로퍼티 삭제

delete card.rank;

console.log(card); // Object { suit : “하트”, value : 15 }

**3.1.4 in 연산자로 프로퍼티가 있는지 확인하기**

프로퍼티가 객체에 포함되어 있을때 true, 없을때 false를 반환

var card = { suite : “하트”, rank : “A” };

console.log(”suit” in card); // -> true

console.log(”color” in card); // false

**3.1.5 객체 리터럴 예제**

**예1) 좌표평면의 점을 표현하는 객체**

var p = { x : 1.0, y : 2.0 };

**예2) 원을 표현하는 객체**

var circle = {

center : { x : 1.0, y : 2.0 }, // 원의 중심을 표현하는 객체

radius : 2.5 // 원의 반지름

};

**예3) 회원 정보를 표현하는 객체**

var person = {

name : “이용교”,

age : 41,

gender : “남”,

married : true

};

**3.1.6 메서드**

프로퍼티에 저장된 값의 타임이 함수인 경우

var person = {

name : “이용교”,

age : 41,

gender : “남”,

married : true,

getLoginInfo : function() {  
  
}

};

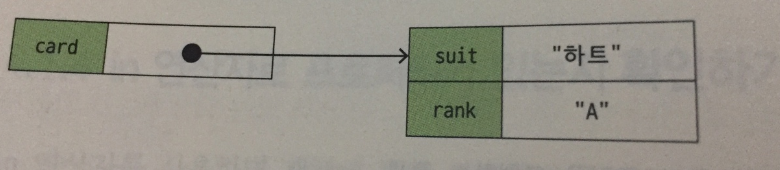
person.getLoginInfo();

**3.1.7 객체는 참조 타입**

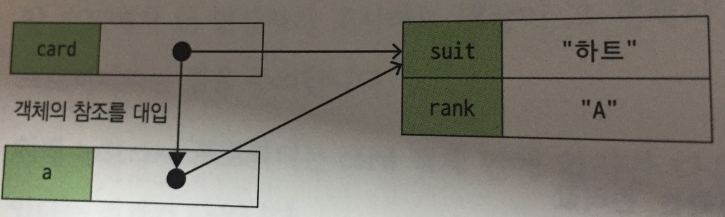
\* 생성된 객체는 메모리의 영역을 차지하는 한 덩어리가 됩니다.

\* 객체 타입의 값을 변수에 대입하면 그 변수에는 객체의 참조(메모리에서의 위치 정보)가

저장됩니다.



\* 변수에 저장된 객체의 참조는 다른 원시값과 마찬가지로 다른 변수에 저장할 수 있음



var a = card;

console.log(a.suit); // -> 하트

a.suit = “스페이드”;

console.log(a.suit); // -> 스페이드

console.log(card.suit); // 스페이드

**3.1.8 함수**

\* 일련의 처리를 하나로 모아 언제든 호출할 수 있도록 만들어 놓은것

\* 수학 함수와 비슷

\* 함수의 입력 값을 **인수**, 함수의 출력 값을 **반환값**

function 함수명 (인수) {  
 처리 로직

return 출력 반환값

}

**3.1.9 함수 선언문으로 함수 정의하기**

함수는 function 키워드를 사용해서 정의

function square(x) {  
 var result = x \* x;

return result;  
 }

\* square - 함수 이름

\* x - 인수

\* var result = x \* x - 처리 로직

\* return result - 처리 후 반환값

참조) 함수명 캐멀 표기법

**3.1.10 함수 호출**

함수를 호출하려면 함수 이름 뒤에 소괄호 인수를 묶어 입력

square(3); // 9

**3.1.11 인수**

\* 함수는 인수를 여러 개 받을 수 있음

\* 인수가 여러 개라면 인수와 인수를 쉼표(,)로 구분

function add(a, b) {  
 var c = a + b;

return c;

}

\* 인수를 받지 않는 함수도 정의할 수 있음

function bark() {  
 console.log(”멍멍”);  
 };

bark(); // 멍멍

console.log(bark()); // undefined - 반환값이 없으므로

**3.1.12 함수의 실행흐름**

\* 호출된 코드에 있는 인수가 함수 정의문에 대입된다.

\* 함수 정의문의 중괄호 안에 작성된 플그램이 순차적으로 실행된다.

\* return 문이 실행되면 호출된 코드로 돌아간다. return 문의 값은 함수의 반환값이 된다.

\* return 문이 실행되지 않은 상태로 마지막 문장이 실행되면, 호출한 코드로 돌아간 후에

undefined가 함수의 반환값이 된다.

**3.1.13 함수 선언문의 끌어올림**

\* 자바스크립트 엔진은 변수 선언문과 마찬가지로 함수 선언문을 프로그램의 첫머리로

끌어올림

\* 따라서 함수 선언문은 프로그램 어떤 위치에서도 작성할 수 있다.

console.log(square(5)); // 25

function square(x) { return x \* x; }

**3.1.14 값으로서의 함수**

\* 자바스크립트에서는 함수가 객체이다(PHP에서 함수는 객체가 아니다)

함수를 선언하면

함수 abc는 function Object()를 상속 하는 객체입니다.



함수를 선언하면 prototype 객체가 생성이 되며

prototype객체에는

constructor 프로퍼티와(생성자)

\_\_proto\_\_ 프로퍼티가 생성이 된다(링커)

constructor는 사용자가 정의한 함수 원형이며

\_\_proto\_\_는 상속 받은 객체를 나타냅니다.

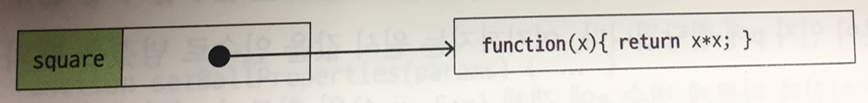
constructor가 있다면 new 연산자를 통해 독립된 메모리 공간에 객체를 생성할 수 있습니다.

function abc() {  
  
}

abc 함수는 abc()로 호출할 수 있고 또는 new abc();로 호출할 수도 있습니다.

\* 함수 선언문으로 함수를 선언하면 내부적으로는 그 함수의 이름을 변수이름으로 한

변수와 함수 객체가 만들어 지고, 그 변수에 함수 객체의 참조가 저장됩니다.



함수명을 다른 변수에 할당하면 그 변수 이름으로 함수를 실행할 수 있습니다.

var sq = square;

console.log(sq(5)); // 25

\* 함수를 다른 함수의 인수로 넘길 수도 있습니다.

function callback(scrollTop) {

console.log(`이벤트 콜백 부분 - 인수${scrollTop}`);

return scrollTop;

}

function goTop(callback) {

var scrollTop = window.pageYOffset;

return callback(scrollTop);

}

**3.1.15 참조에 의한 호출과 값에 의한 호출**

\* 원시값을 함수에 인수로 넘겼을때는 대입되어 복사가 됩니다.

\* 객체 값을 인수로 넘겼을때 객체의 참조가 대입되어 복사 됩니다.

즉 객체의 값을 인수로 넘기게 되면 인자와 인수가 동일하게 변경이 됩니다

(동일한 데이터의 주소를 참조하므로)

원시값은 값 자체가 복사가 되므로 인자와 인수가 각각 유지가 됩니다.

**3.1.16 변수의 유효범위**

**\* 전역 유효 범위와 지역 유효범위**

**전역변수**

- 함수 바깥에서 선언된 변수

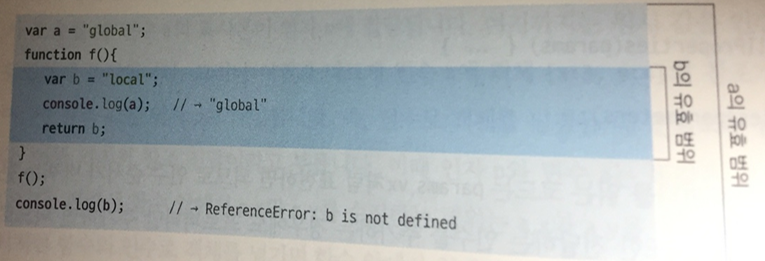
- 유효 범위는 전체 프로그램

**지역변수**

- 함수 안에서 선언된 변수

- 함수 인자

- 유효범위는 변수가 선언된 함수 내부



**3.1.17 변수의 충돌**

\* 변수에 유효 범위가 있는 이유는 프로그램의 다른 부분에서 선언된 이름이 같은 변수와

충돌하지 않도록 하기 위해서입니다.

\* 다른 함수 내부에서 선언된 각각의 지역변수는 모두 선언된 함수 내부에서만 유효하므로

이름이 같아도 충돌하지 않습니다.

\* 함수 안에서 변수 이름을 지을 때는 다른 함수 안에서 사용하는 변수 이름까지 신경쓸

필요 없음

\* 전역변수와 지역변수 이름이 같아지면 전역변수를 숨기고 지역변수를 사용

var a = “global”;

function f() {  
 var a = “local”;

console.log(a); // -> local

return a;  
}

f();

console.log(a); // -> global

**3.1.18 함수 안에서 변수 선언과 변수 끌어올림**

\* 함수 안에서 선언된 지역변수의 유효범위는 함수 전체입니다.

\* 함수 중간 부분에서 변수를 선언하더라도 변수는 함수의 첫머리로 끌어올립니다.

function f() {

console.log(a); // undefined -> 끌어올리지 않는다면 Reference Error 발생

var a = “local”;

console.log(a); // -> local

return a;

}

**3.1.19 함수 안에서 변수 선언 생략**

\* 변수를 선언하지 않은 상태에서 값을 대입하면 전역변수로 선언됨

\* 지역변수로써만 정의하는 경우는 반드시 변수 선언을 해야 함

예) var a = “local”;

function f() {  
 a = “local”;

console.log(a); // -> local

return a;  
 }

f();

console.log(a);// local

**3.1.20 블록 유효 범위 : let과 const**

**\* let 선언자**

let 문은 블록 유효 범위를 갖는 지역변수를 선언합니다. 사용법은 var문과 같다.

let x = “outer x”;

{

let x = “inner x”;

let y = “inner y”;

console.log(x); // -> inner x

console.log(y); // inner y

}

console.log(x); // -> outer x

console.log(y); // -> ReferenceError : y is not defined

**\* const 선언자**

- const 문은 블록 유효 범위를 가지면서 한 번만 할당할 수 있는 변수(상수)를 선언

한다

- const로 선언한 상수는 let 문으로 선언한 변수처럼 동작한다

- 단, 반드시 초기화해야 한다는 차이점이 있다

const c = 2;

- const 문으로 선언한 변수에 다시 대입을 시도하면 타입 오류가 발생

c = 5; // -> uncaught TypeError

- const 문으로 선언한 상수 값은 수정할 수 없지만, 상수값이 객체이거나 배열일 경우

프로퍼티 또는 프로퍼티 값을 수정할 수 있다.

const origin = {x:1, y:2};

origin.x = 3;

console.log(origin); // -> Object {x:3, y:2}

**3.1.21 함수 리터럴로 함수 정의하기**

\* 함수는 리터럴로도 정의할 수 있다

var square = function(x) {

return x \* x;

};

\* function(x) { ... } 부분이 함수 리터럴

\* 함수 리터럴은 이름이 없느 함수 이므로 익명함수 또는 무명함수라고 한다

\* 함수 선언문에서는 끝에 세미콜론(;)을 붙일 필요가 없으나 함수 리터럴을 사용할땐

끝에 세미콜론(;)을 붙여야 한다

\* 함수 리터럴로 정의한 함수는 끌어올리지 않는다

console.log(square(3)); // -> TypeError : square is not a function

var square = function(x) { return x \* x; }

\* 익명함수에도 이름을 붙일 수 있다

var square = function sq(x) { return x \* x; };

sq라는 함수 이름은 함수 안에서만 유효하며 함수 바깥에서는 sq라는 이름으로

함수를 호출할 수 없다.

**3.1.22 객체의 메서드**

\* 객체의 프로퍼티 중에서 함수 객체의 참조를 값으로 담고 있는 프로퍼티를 메서드라 한다

\* 메서드를 정의할 때는 프로퍼티 값으로 함수 리터럴을 대입합니다.

var circle = {

center : { x : 1.0, y: 2.0 }, // 원의 중점을 표현하는 객체

radius : 2.5, // 원의 반지름

area : function() { // 원의 넓이를 구하는 메서드

return Math.PI \* this.radius \* this.radius;

}

};

\* 함수 객체 안에 적힌 this는 그 함수를 메서드로 가지고 있는 객체를 가리킨다

(즉, circle, this.radius -> circle.radius)  
 \* 메서드 호출

circle.area();

\* 메서드 또한 프로퍼티의 일종이므로 나중에 추가 할 수 있다

circle.translate = function(a, b) {

this.center.x = this.center.x + a;

this.center.y = this.center.y + b;

};

circle.translate(1,2);

circle.center; // Object { x = 2, y = 4 }

**3.1.23 즉시 실행 함수**

\* 자바스크립트에는 익명 함수를 정의하고 곧바로 실행하는 '즉시 실행 함수'라는 구분이 있음

(function() { .. })();

\* 즉시 실행 함수에도 인수를 넘길 수 있음

(function (a, b) {

return a + b;

})(1,2);

\* 반환값 역시 받을 수 있음

var x = (function(a,b) {

return a + b;

})(a,b);

**3.1.24 가변길이 인수 목록(Arguments 객체)**

\* 모든 함수에 있는 지역변수로 arguments로 접근

\* 유사 배열 객체 이며 다음과 같이 접근 가능

arguments[0]; // 첫 번째 인수 값

arguments[1]; // 두번째 인수 값

...

arguments[n-1] : // n번째 인수 값

\* arguments.length : 인수 개수

\* arguments.callee : 현재 실행되고 있는 함수의 참조

function f(x, y) {  
 arguments[1] = 3;  
 console.log(”x = “ + x + “, y = “ + y);

}

f(1,2); // -> x = 1, y = 3;

\* 가변 변수로 인수로 지정하는 방법도 있음

가변 변수는 ...변수명

function abc(a, ...b) {  
   
 }

abc(1, 2,3,4,5);로 호출한다면

a는 1, b는 [2,3,4,5]

**3.1.25 함수를 활용하면 얻을 수 있는 장점**

\* 재사용할 수 있다

\* 만든 프로그램을 이해하기 쉽다

\* 프로그램 수정이 간단해진다

**3.2 생성자**

생성자를 사용하면 이름이 같은 메서드와 프로퍼티를 가진 객체를 여러개를 효율적으로 생성할수 있다

**3.2.1 생성자로 객체 생성하기**

자바스크립트에서는 생성자라고 하는 함수로 객체를 생성할 수 있다,

function Card(suit, rank) {

this.suit = suit;

this.rank = rank;

}

var card = new Card(“하트”, “A”);

console.log(card); //-> Card { suit: “하트”, rank : “A” }

**\* 생성자**

- new 연산자로 객체를 생성할 것이라 기대하고 만든 함수를 생성자라고 한다.

- 생성자는 일반 함수와 구분할 수 있도록 관용적으로 첫 글자를 대문자로 쓴다.

(파스칼 표기법)

- 생성자 안에서 this.프로퍼티 이름에 값을 대입하면 그 이름을 가진 프로퍼티에 값이

할당된 객체가 생성된다.   
 - 생성자와 new 연산자로 생성한 객체를 생성자의 인스턴스라고 부른다.

- 앞의 예) var card = new Card(“하트”, “A”); 객체 리터럴로 고쳐 쓸 수 있다.

var card = {};

card.suit = “하트”;

card.rank = “A”;

**\* 생성자의 역할**

- 생성자는 객체를 생성하고 초기화 하는 역할

- 생성자 이름은 같지만 프로퍼티 값이 다른 객체(인스턴스)를 생성할 수 있다.

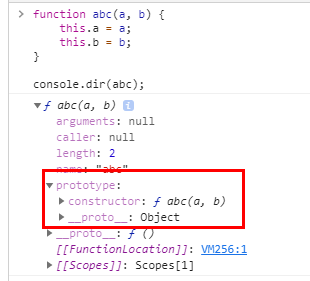
var card1 = new Card(“하트”, “A”);

var card2 = new Card(“클럽”, “K”);

var card3 = new Card(“스페이드”, “2”);

**\* 객체 리터럴과 생성자의 차이**

객체 리터럴은 생성자 함수가 이미 생성된 상태, 즉 객체가 변수에 대입된 상태이다.

 따라서 new 연산자를 사용하여 객체를 여러개 생성할 수 없다.

**생성자 함수**

생성자 함수는 prototype 객체로 이뤄져 있고

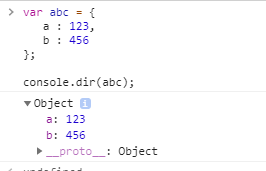
구성요소로 constructor(생성자 – 생성자 함수 원형)와

\_\_proto\_\_(링커 – 상속받은 객체)로 구성되어 있다

생성자 함수는 함수 원형(constructor)가 있어 이를 new 연산자로 여러 변수에 대입하여

객체를 생성할 수 있다.

**객체 리터럴**



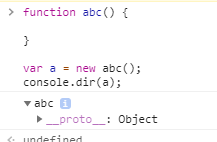
객체 리터럴은 이미 abc라는 변수에 {}라는 객체가 대입된 상태 이므로 내부적으로는 new 연산자로 생성된 상태이다. 따라서 prototype 객체는 존재 하지 않고 \_\_proto\_\_(링커)만 노출이 되며, 상속 관계(Object를 상속 받음)만 나타나게 된다.

즉, 객체 리터럴은 prototype객체가 없고 contructor가 없으므로 new 연산사로

생성 할 수 없다.

생성자 함수도 new 연산자를 이용하여 생성된 인스턴스는 객체 리터럴과 동일한 상태

객체(인스턴스) a 역시 \_\_proto\_\_만 존재하며 객체 상속 관계만 나타냅니다.



**3.2.2 메서드를 가진 객체를 생성하는 생성자**

\* 생성자에서 this.프로퍼티 이름에 함수의 참조를 대입하면 메서드를 정의할 수 있다.

function Circle(center, radius) {  
 this.center = center;

this.radius = radius;

this.area = function() {

return Math.PI \* this.radius \* this.radius;

};  
 }

var p = {x:0, y: 0};

var c = new Circle(p, 2.0);

console.log(“넓이 = ”+ c.area());

\* this는 생성된 객체(인스턴스)를 가리킨다.

예)

function abc(a, b) {  
 this.a = a;

this.b = b;  
 }

var obj1 = new abc(1,2); -> this는 obj1를 가리킴

var obj2 = new abc(3,4); -> this는 obj2를 가리킴

**3.2.3 프로토타입**

**3.2.3.1 생성자 안에서 메서드를 정의하는 방식의 문제점**

생성자 안에서 this 뒤에 메서드를 정의하면 그 생성자로 생성한 모든 인스턴스에 똑같은

메서드가 추가됩니다. 따라서 메서드를 포함한 생성자로 인스턴스를 여러개 생성하면 같은

작업을 하는 인스턴스 개수만큼 생성하게 되며 결과적으로 그만큼의 메모리를 소비하게

된다.

function Circle(center, radius) {

this.center = center;

this.radius = radius;

this.area = function() {

return Math.PI \* this.radius \* this.radius;

};

}

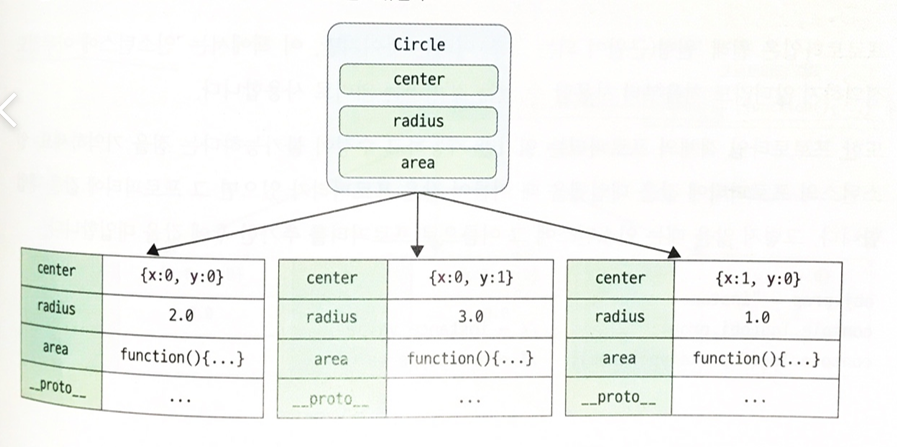
var c1 = new Circle({x:0, y: 0}, 2.0);

var c2 = new Circle({x:0, y: 1}, 3.0);

var c3 = new Circle({x:1, y: 0}, 1.0);

console.log(c1.area === c2.area) // false -> 생성된 생성자 내부에 있는 area메소드 이므로

서로 다르다(즉, 서로 다른 메모리에 있다)



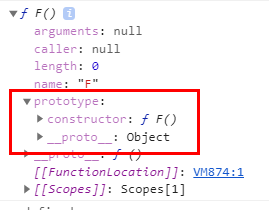
**3.2.3.2 프로토타입 객체**

\* 자바스크립트에서는 함수도 객체이므로 함수 객체가 기본적으로 prototype 프로퍼티를

갖고 있다.

function F() {};

console.log(F.prototype); // -> Object {}



\* 함수의 prototype 프로퍼티가 가리키는 객체를 그 함수의 프로토타입 객체라고 한다

\* 프로토타입 객체의 프로퍼티는 생성자로 생성한 모든 인스턴스에서 그 인스턴스의

프로퍼티처럼 사용할 수 있다.

\* 인스턴스에 아무것도 정의 하지 않더라도(즉, 객체를 생성하지 않더라도) 처음부터

사용할 수 있는 것

\* 1번만 생성자 Prototype 프로퍼티에 정의하면 인스턴스 생성시에 생성자 함수안에 포함

되지않더라도 계속 접근 및 사용이 가능 - 메모리 절약 가능

function Circle(center, radius) {  
 this.center = center;  
 this.radius = radius;

}

// Circle 생성자의 prototype 프로퍼티에 area 메서드를 추가

Circle.prototype.area = function() {  
 return Math.PI \* this.radius \* this.radius;  
 };

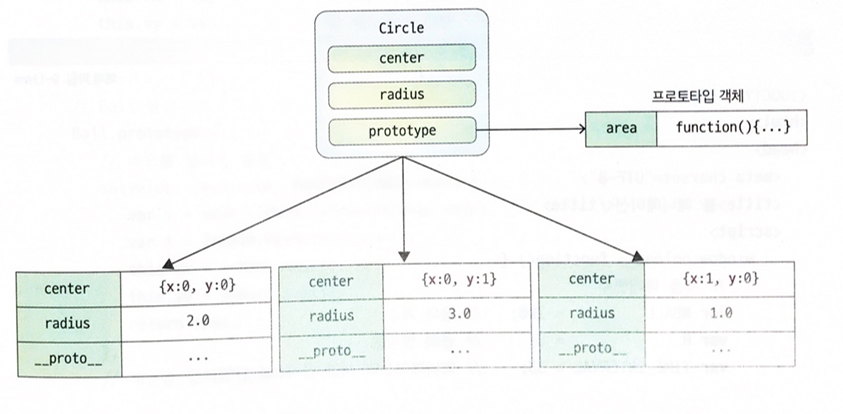
var c1 = new Circle({x:0, y: 0}, 2.0);

var c2 = new Circle({x:0, y: 1}, 3.0);

var c3 = new Circle({x:1, y: 0}, 1.0);

console.log(c1.area === c2.area); // true c1.area와 c2.area는 Circle.prototype.area

를 서로 공유하므로 동일하다



**3.3 내장 객체**

\* 자바스크립트에서는 처음부터 사용할 수 있는 내장 객체(빌트인 오브젝트)가 마련되어

있다.

\* 내장객체는 자바스크립트라는 프로그래밍 언어의 뼈대를 구성합니다.

**3.3.1 내장 생성자**

\* 자바스크립트에서는 사용할 수 있는 생성자에는 사용자가 정의하는 생성자 외에도

자바스크립트에 처음부터 포함된 “내장생성자”가 있습니다.

|  |  |
| --- | --- |
| 생성자 이름 | 생성되는 객체 |
| Object | 일반 객체 |
| String | 문자열 객체 |
| Number | 숫자 객체 |
| Boolean | 논리값 객체 |
| Array | 배열 |
| Date | 날짜와 시간을 다루는 객체 |
| Function | 함수 객체 |
| RegExp | 정규표현식 객체 |
| Math | 수학관련 객체 |
| Map | key/value 맵을 생성 |
| Set | 중복을 허용하지 않는 데이터 집합을 생성 |

**3.3.1.1 String**

문자열을 처리하는 생성자

**\* 문자열 연결**

- + 연산자는 문자열을 문자열로 연결

- 피 연산자 중 하나가 문자열이면 다른 피연산자의 타입을 문자열로 바꾼 다음 연결

“Hello ” + “World!” -> “Hello World!“

“1” + “2” -> “12”

10 + “ little indians” -> “10 little indians”

**\* 문자열을 조작하는 메서드**

문자열을 처리하기 위한 객체로 String 객체가 마련되어 있습니다. 문자열을 String 객체

로 변환하려면 String 생성자를 사용합니다.

var msgObj = new String(“Everything is practice.”);

원시값을 객체로 변환하는 행위 -> 객체로 래핑(wrapping)한다고 함

객체로 변환하면 문자열을 처리하기 위한 다양한 프로퍼티와 메서드를 사용할 수 있다.

원시값에 메소드를 바로 호출하는 경우 자동으로 객체 래핑이 이뤄지게 됩니다.

var msg = “Everything is practice”;

var a = msg.chatAt(0);

위와 같이 msg라는 원시값에 chatAt(0)을 호출하게 되면 내부적으로는 String 객체가

생성이 되고 chatAt으로 호출이 됩니다.

var msg = “Everything is practice” => var msg = new String(“Everything is practice”);

var a = msg.chatAt(0);

\* 자주 사용하는 메서드

**String.charAt(n)**

문자열내 특정 위치에 있는 문자를 반환

const a = "Javascript";

console.log(a.charAt(4)) // s (4번째 문자)

**String.concat([s1, s2, ..])**

s1, s2, ...을 서로 붙여서 새로운 문자열을 반환

const str1 = 'Hello';

const str2 = 'World';

var newStr = str1.concat(' ', str2); // Hello World

console.log(newStr);

**String.endsWith(s)**

문자열에서서 특정 문자열로 끝나는지 확인

반환값 – true, false

var str = "I have an apple.";

console.log(str.endsWith("apple.")) // true

console.log(str.endsWith("have")) // false

**String.includes(s)**

문자열 내에서 찾고자 하는 문자열이 포함되어 있는지 확인

const a = "apple, orange, melon";

console.log(a.includes("orange")); // true

**String.indexOf(s)**

문자열(“s”)이 문자열 내에서 첫 번째로 일치하는 위치 순서(인덱스) 값을 반환,

일치하는 값이 없을 때는 –1을 반환(왼쪽 -> 오른쪽으로 검색)

const a = "Apple, orange, melon, melon";

console.log(a.indexOf("melon")); // 15

console.log(a.indexOf("water melon")); // -1

**String.lastIndexOf(s)**

문자열(“s”)이 문자열 내에서 반대방향부터 첫 번째로 일치하는 위치 순서(인덱스) 값을

반환, 일치하는 값이 없을 때는 –1을 반환(오른쪽 -> 왼쪽으로 검색)

const a = "Apple, orange, melon, melon";

console.log(a.lastIndexOf("melon")); // 22

console.log(a.lastIndexOf("water melon")); // -1

**String.repeat(n)**

문자열을 주어진 횟수(n)만큼 반복해 붙인 새로운 문자열을 반환합니다.

“abc”.repeat(1); // 'abc'

“abc”.repeat(2); // 'abcabc'

**String.replace(s1, s2)**

문자열 중 s1을 s2로 치환합니다. 다만 첫번째 문자열만 치환됩니다.

전부 치환하는 경우는 정규 표현식 중 g 플래그를 사용합니다.

var a = "1 little 2 little indians";

var b = a.replace("little", "big");

console.log(b); // 1 big 2 little indians

var c = a.replace(/little/g, "big");

console.log(c); // 1 big 2 big indians

**String.slice(start,end)**

문자열 일부를 추출하면서 새로운 문자열을 반환한다

(추출 범위 - start ~ (end - 1) - end 번째는 포함하지 않음)

var str1 = "안녕하세요";

var str2 = str1.slice(1,3);

console.log(str2); // 녕하

var str3 = str1.slice(2);

console.log(str3); // 하세요

var str3 = str1.slice(-3, -1);

console.log(str3); // 하세

**String.split(s)**

지정한 구ㅁ분자(s)를 이용하여 여러개의 문자열로 나눕니다.

var str = "apple,banana,pineapple,melon";

var s1 = str.split(',');

console.log(s1); // [ 'apple', 'banana', 'pineapple', 'melon' ]

**String.startsWith(s, [,n])**

어떤 문자열이 특정 문자로 시작하는지 확인하여 결과를 true 또는 false로 반환합니다.

var a = "I'm a boy";

console.log(a.startsWith("I'm")); // true

**String.substring(m, n)**

시작(m) 인덱스에서 종료(n) 인덱스 전(n-1) 까지 문자열의 부분 문자열을 반한

var str = "Apple, Orange";

console.log(str.substring(1, 3)); // pp

console.log(str.substring(2)); // ple, Orange

**String.toLowerCase()**

문자열을 소문자로 변환해 변환

var a = "APPLE";

console.log(a.toLowerCase()); // apple

**String.toUpperCase()**

문자열을 대문자로 변환해 반환

var a = "apple";

console.log(a.toUpperCase()); // APPLE

**String.toString()**  
 호출한 객체를 나타내는 문자열

var a = new String("Hello world");

console.log(a.toString()); // Hello world

**String.trim()**

양 끝에서 공백을 제거한 새로운 문자열

var a = ' apple ';

console.log(a.trim()); // apple

**\* 문자열을 배열로 읽고 쓰기**

문자열을 읽을 때는 charAt 메서드 대신 대괄호 연산자를 사용할 수 있다.

var a = "Apple";

console.log(a[3]); // l

**3.3.1.2 Number**

\* Number 객체는는 숫자값으로 작업할 수 있게 해주는 래퍼(wrapper)객체 입니다.

\* Number 객체는 Number() 생성자를 사용하여 만듭니다.

\* 원시 숫자 자료형은 Number() 함수를 사용해 생성합니다.

\* 생성자로써 사용하지 않으면(new 연산자를 사용하지 않으면) Number를 사용하여 형 변환을

할 수 있다.

var a = new Number('123'); // 객체 생성

console.log(a == 123); // true

console.log(a === 123); // false // 숫자로 동작하지만 Number 객체 이므로 숫자 원시타입과는 다르다

console.log(a); // [Number: 123]

var b = Number('123');

console.log(b === 123); // true -> Number 함수는 원시 숫자 자료형을 만든다

console.log(a instanceof Number); // true -> a는 new Number(123)으로 생성된 Number 객체

console.log(b instanceof Number); // false -> b는 Number 함수로 생성된 원시 숫자 자료형, Number 객체가 아니다

\* 자주 사용하는 속성

**Number.MAX\_VALUE** - 표현가능한 가장 큰 양수

**Number.MIN\_VALUE** - 표현가능한 가장 작은 양수, 즉, 0보다 크지만 0에 가장 가까운 양수

**Number.NaN** - “숫자가 아님”을 나타내는 특별한 값

\* 자주 사용하는 메서드

**Number.isNaN()**

주어진 값이 NaN(숫자가 아님)인지 확인

전역 isNaN() 함수의 더 엄격한 버전

숫자 여부를 체크 할땐 Number.isNaN 보다는 isNaN() 전역 함수를 사용할 것

console.log(isNaN("apple")); //true -> 숫자가 아니므로

console.log(isNaN(1234)); // false -> 숫자이므로

**Number.isFinite()**

주어진 값이 유한수 인지 확인

console.log(Number.isFinite(1 / 0)); // false -> 무한수 이므로

console.log(Number.isFinite(10/5)); // true -> 유한수 이므로

console.log(Number.isFinite(0 / 0)); // true -> 무한수 이므로

**Number.isInteger()**

주어진 값이 정수인지 확인 합니다.

console.log(Number.isInteger(1.256)); // false -> 실수이므로

console.log(Number.isInteger(2)); // true -> 정수이므로

**Number.parseFloat()**

전역 함수 parseFloat()와 동일한 값 입니다.

문자열을 분석해 실수(소수점이 있는 수)로 반환합니다.

var a = "1234.1231";

var b = a + 1;

console.log(b); // 1234.12311 -> 1234.1231를 문자로 인식

var c = parseFloat(a) + 1;

console.log(c); // 1235.1231 // a를 실수로 변경 한 후 1 더하기 연산

**Number.parseInt()**

전역 함수 parseInt()와 동일한 값입니다.

문자열 인자를 구문분석하여 정수를 반환합니다.

var a = "123";

var b = a + 1;

console.log(b); // 1231

var c = parseInt(a) + 1;

console.log(c); // 124

**3.3.1.3 Boolean**

논리값(Boolean)을 감싸고 있는 객체 입니다.

첫 번째 매개변수로 전달한 값은 필요한 경우 논리값(Boolean)으로 변환됩니다.

**0, -0, null, false, NaN, undefined, 빈 문자열(””)**이라면 초기값은 **false**가 됩니다.

그 외의 모든 다른 값은 초기값을 true로 설정합니다.

Boolean 객체의 true와 false 값을 원시 Boolean 값 true, false와 혼동해서는 안됨

즉, 값이 0, -0, null, false, NaN, undefined, 빈 문자열(””)이 아니라면 조건문에서 true로 계산됩니다.

이는 값이 false인 Boolean 객체도 포함됩니다.

var x = new Boolean(false);

if (x) { // x는 true  
   
 }

var x = false;

if (x) { // x는 원시값 false

}

논리값(Boolean)이 아닌 값을 변환할 때 Boolean 객체를 사용해선 안됩니다. 대신 Boolean 함수를

사용하세요.

var x = Boolean(expression); // 추천

var x = new Boolean(expression); // 사용하지 말 것

console.log(Boolean(null)); // false

console.log(Boolean(0)); // false

console.log(Boolean(false)); // false

console.log(Boolean(true)); // true

console.log(Boolean("Apple")); // true

**3.3.1.4 Math**

Math는 수학적인 상수와 함수를 위한 속성과 메서드를 가진 내장 객체 입니다.

\* 자주 사용하는 속성

**Math.PI** - 원주율

\* 자주 사용하는 메서드

**Math.abs(x)**

숫자의 절댓값을 반환합니다.

console.log(Math.abs(-2)); // 2

console.log(Math.abs(2)); // 2

**Math.ceil(x)**

실수 값을 올림한 정수 값을 반환

console.log(Math.ceil(0.95)); // 1

console.log(Math.ceil(6.1)); // 7

console.log(Math.ceil(-4.2)) // -4

**Math.floor(x)**

실수 값을 내림처리한 정수 값을 반환

console.log(Math.floor(5.95)); // 5

console.log(Math.floor(5.05)); // 5

console.log(Math.floor(5)); // 5

console.log(Math.floor(-5.05)); // -6

**Math.round(x)**

실수 값을 반올림 처리한 정수 값을 반환

console.log(Math.round(0.9)); // 1

console.log(Math.round(5.95)); // 6

console.log(Math.round(-5.05)); // -5

console.log(Math.round(-5.95)); // -6

**Math.max(x, y, z, ... )**

0개 이상의 인수에서 제일 큰 수를 반환

var a = Math.max(10, 20, 40, 50);

console.log(a); // 50

**Math.min(x, y, z, ...)**

0개 이상의 인수에서 제일 작은 수를 반환

var a = Math.min(10, 20, 40, 50);

console.log(a); // 10

**Math.pow(x, y)**

x의 y 제곱을 반환

console.log(Math.pow(2, 3)); // 8

**Math.random()**

0과 1사이의 난수를 반환

// Math.random()

console.log(Math.random()); // 0.740439573128189(랜덤으로 변경됨)

// 난수 생성

var uid = Math.round(Math.random() \* 10000000000);

console.log(uid); // 3050097735 (랜덤으로 변경됨)

**Math.sign(x)**

x의 양의 수인지 음의 수인지 나타내는 부호를 반환

양수인 경우 1, 음수인 경우 -1, 영이면 0, 숫자가 아니라면 NaN을 반환

console.log(Math.sign(3)); // 1

console.log(Math.sign(-3)); // -1

console.log(Math.sign(0)); // 0

console.log(Math.sign('Apple')); // NaN

**Math.sqrt(x)**

숫자의 제곱근을 반환

console.log(Math.sqrt(9)); // 3

**3.3.1.5 Date**

\* Date생성자는 날짜와 시간을 표현하는 객체를 생성한다

\* Date 객체는 Date 생성자로 생성

var now = new Date();

console.log(now); // Thu Mar 04 2021 23:33:35 GMT+0900 (대한민국 표준시)

\* Date 생성자의 인수로 날짜와 시간을 전달하면 그 날짜와 시간을 가리키는 Date객체가

생성된다.

var then = new Date(2020, 5, 10);

console.log(then); // Wed Jun 10 2020 00:00:00 GMT+0900 (대한민국 표준시)

월은 0~11로 입력(즉 0이 1월)

\* Date객체는 계산식 안에서 밀리초 단위 정수로 값의 타입이 바뀐다.

var elapsed = now - then;

console.log(elapsed); // 23153926012

**Date 객체가 제공하는 주요 메서드**

|  |  |
| --- | --- |
| 메서드명 | 설명 |
| getFullYear() | 연도를 뜻하는 숫자 값 예) 2016 |
| getMonth() | 월을 뜻하는 숫자 값, 0부터 시작, 예) 4 - 5월 |
| getDate() | 날짜를 뜻하는 숫자 값 |
| getHours() | 시각의 시간을 뜻하는 숫자 값 |
| getMinutes() | 시각의 분을 뜻하는 숫자 값 |
| getSeconds() | 시각의 초를 뜻하는 숫자 값 |
| getDay() | 요일일 뜻하는 숫자 값, 0~6사이 값(일~토) |
| getMilliseconds() | 시각의 밀리초를 뜻하는 숫자 값 |
| toString() | Date객체의 문자 표기값(날짜 시간이 출력됨) |
| toLocaleString() | 지역화된 시간과 날짜 정보 |
| toLocaleDateString() | 지역화된 날짜 정보 |
| toLocaleTimeString() | 지역화된 시간 정보 |
| getUTCHours() | UTC(협정 세계 시) 시각의 시간을 뜻하는 숫자 값 |
| toUTCString() | UTC 시간과 날짜 정보 |

**3.3.1.6 Array**

배열(Array)은 값의 목록으로 값마다 번호가 매겨져 있습니다.

**3.3.1.6.1 배열 리터럴로 생성하기**

배열 리터럴은 쉼표로 구분한 값으 대괄호([])로 묶어서 표현합니다.

var evens = [2, 4, 6, 8];

\* 앞 코드에서 [...] 부분이 배열 리터럴이며, 배열 값 하나를 **배열 요소**라고 부릅니다.

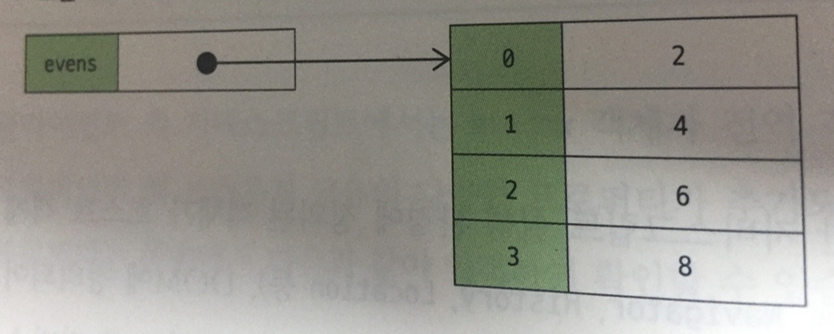
\* 배열요소에서는 왼쪽부터 순서대로 0, 1, 2 ... 라는 번호가 매겨져 있습니다. 요소에 매긴

번호는 **요소번호** 또는 **인덱스**라고 부릅니다.

\* 자바스크립트의 배열은 객체 타입이므로 배열을 변수에 대입하면 배열의 참조가 변수에

저장됩니다.

변수 evens는 배열[...]을 가리킨다



\* 배열 리터럴 안에 어떠한 요소도 작성하지 않으면 빈 배열일 생성됩니다.

var empty = [];

console.log(empty); // -> []

\* 배열 리터럴 요소의 값을 생략하면 그 요소는 생성되지 않습니다.

var a = [2, , 4];

console.log(a); // -> [2, undefined, 4]

앞 코드의 결과에서 인덱스가 1인 요소, 즉 두 번쨰 요소에 undefined가 표시되어 있지만

실제로는 없는 값

\* 변수의 요소에는 변수와 마찬가지로 모든 타입의 값이 올 수 있습니다.

var various = [3.14, 'pi', true, { x:1, y: 2}, [2,4,6,8]];

**3.3.1.6.2 length 프로퍼티**

\* 배열의 length 프로퍼티에는 배열 요소의 최대 인덱스 값 + 1이 담겨 있습니다.

\* length 프로퍼티 값을 가리켜 배열의 길이라고 부릅니다.

var evens = [2,4,6,8];

evens.length; // 4

\* length 프로퍼티데 현재의 배열 요소 개수보다 작고 0보다 큰 정수 값을 대입하면

배열의 길이가 줄어듭니다. 즉, 그 배열의 길이가 넘는 인덱스 번호에 할당된 배열 요소는

삭제 됩니다.

var a = [”A”,”B”,”C”,”D”];

a.length = 2;

console.log(a); // -> [”A”, “B”];

**3.3.1.6.3 Array 생성자로 생성하기**

\* 배열은 Array 생성자로도 생성할 수 있습니다.

var evens = new Array(2,4,6,8);

console.log(evens); // [ 2, 4, 6, 8 ]

var empty = new Array(); // 빈 배열 []을 생성

var a = new Array(2, 4);

console.log(a); // [ 2, 4 ], 배열 리터럴 [ 2, 4 ]와 똑같은 배열을 생성

var various = new Array(3.14, "pi", true, {x:1, y:2}, [2, 4, 6, 8]);

console.log(various); // [ 3.14, 'pi', true, { x: 1, y: 2 }, [ 2, 4, 6, 8 ] ]

\* Array 생성자 인수가 한 개이고 그 값이 양의 정수면 배열 길이를 뜻하며 그 길이 만큼 생성

var x = new Array(3);

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

**3.3.1.6.4 배열 요소의 참조**

\* 특정 인덱스의 요소는 대괄호([]) 연산자를 사용해서 읽거나 쓸 수 있습니다.

evens[2] // -> 6

\* 배열 요소 하나는 변수 하나로 사용할 수 있습니다. 즉 배열 요소에는 모든 타입의

데이터를 할당할 수 있으며 프로그램에서 참조할 수 있습니다.

**3.3.1.6.5 배열은 객체**

C나 Java, PHP 같은 프로그래밍 언어의 배열 요소는 메모리의 연속된 공간에 차례대로 배치되어

있습니다. 따라서 인덱스를 지정하면 인덱스가 가리키는 요소를 매우 빠르게 읽거나 쓸 수 있습니다.

그러나 자바스크립트의 배열은 다릅니다.

자바스크립트의 배열은 Array 객체이며 객체로 배열의 기능을 가상으로 흉내 낸 것 입니다.

Array 객체는 배열의 인덱스를 문자열로 변환해서 그것을 프로퍼티로 이용합니다. 즉, 배열에 대괄호

연산자를 사용하는 것과 마찬가지이며 배열의 요소 번호로 숫자 값 대신 문자열을 사용할 수

있습니다.

var a = [”A”, “B”, “C”, “D”];

console.log(a[”2”]); // C

이때 없는 배열 요소를 읽으려고 시도하면 undefined가 반환됩니다.

a[4] // undefined

이를 배열이 객체라는 사실에 비추어 생각하면 **3.1.2**에서 언급한 **객체에 없는 프로퍼티를 읽으려고**

**시도하면 undefined가 반환된다**와 같은 내용임을 알 수 있습니다.

**3.3.1.6.6 배열 요소의 추가와 삭제**

**\* 배열 요소 추가**

- 없는 배열 요소에 값을 대입

var a = [”A”,”B”,”C”];

a[3] = “D”;

console.log(a); // [ 'A', 'B', 'C', 'D' ]

- push 메서드를 사용

a.push("E");

console.log(a); // [ 'A', 'B', 'C', 'D', 'E' ]

**\* 배열 요소 삭제**

- delete 연산자를 사용하면 특정 배열 요소를 삭제할 수 있습니다.

delete a[1];

console.log(a); // [ 'A', undefined, 'C', 'D', 'E' ]

delete 연산자를 사용하여 배열의 요소를 삭제해도 그 배열의 length 프로퍼티 값은 바뀌지

않습니다.

즉, 삭제한 요소만 사라집니다.

**\* 자주 사용하는 메서드**

**Array.join(separator)**

배열의 모든 요소를 특정 구분자(separator) 연결해 하나의 문자열로 만듭니다.

구분자는 설정하지 않으면 콤마(,)로 연결 됩니다.

var a = ['바람', '비', '불'];

var myVar1 = a.join();

console.log(myVar1); // 바람,비,불

var myVar2 = a.join(',');

console.log(myVar2); // 바람,비,불

var myVar3 = a.join("+");

console.log(myVar3); // 바람+비+불

var myVar4 = a.join('');

console.log(myVar4); // 바람비불

**Array.reverse()**

배열의 순서를 역순으로 변경합니다. 첫 번째 요소는 마지막 요소가 되며 마지막 요소는 첫 번째

요소가 됩니다.

var a = ['one', 'two', 'three'];

console.log(a); // [ 'one', 'two', 'three' ]

var b = a.reverse();

console.log(b); // [ 'three', 'two', 'one' ]

**Array.sort()**

배열의 순서를 오름차순으로 변경합니다.

// 문자 오름 차순

var months = ["March", "Jan", "Feb", "Dec"];

var sorted = months.sort();

console.log(sorted); // [ 'Dec', 'Feb', 'Jan', 'March' ]

// 숫자 오름 차순

var a = [1, 30, 4, 21, 29];

var b = a.sort(function(a, b) {

return a - b;

});

console.log(b); // [ 1, 4, 21, 29, 30 ]

// 숫자 내림 차순

var c = a.sort(function(a, b) {

return b - a;

});

console.log(c); // [ 30, 29, 21, 4, 1 ]

**Array.slice(begin, end)**

배열의 begin부터 end까지(end 미포함)에 대한 복사본을 새로운 배열 객체로 반환합니다.

begin이 음수라면 배열의 끝에서부터 요소를 세어나갑니다( -1이면 요소 끝에서 n번째 요소)

var a = ['a', 'b','c','d','e','f'];

var b = a.slice(2);

console.log(b); // [ 'c', 'd', 'e', 'f' ]

var c = a.slice(2, 4);

console.log(c); // [ 'c', 'd' ]

var d = a.slice(1,5);

console.log(d); // [ 'b', 'c', 'd', 'e' ]

**Array.splice(start, deleteCount, item1, item2, ... )**

배열의 기존 요소를 삭제 또는 교체하거나 새 요소를 추가하여 배열의 내용을 변경합니다.

start - 배열의 변경을 시작할 인덱스, 음수라면 배열의 끝에서부터 요소를 세어나갑니다.

( -1이면 요소 끝에서 n번째 요소)

deleteCount - 배열에서 제거할 요소의 수

item1, item2, ... 배열에 추가할 요소입니다. 아무 요소도 지정하지 않으면 splice()는 요소를

제거하기만 합니다.

반환값 - 제거된 요소를 담은 배열

// 하나도 제거하지 않고 2번 인덱스에 "drum"추가

var a = ['angel', 'clown', 'mandarin', 'sturgeon'];

var removed = a.splice(2, 0, 'drum');

console.log(removed); // []

console.log(a); // [ 'angel', 'clown', 'drum', 'mandarin', 'sturgeon' ]

// 하나도 제거하지 않고, 2번 인덱스에 "drum"과 "guitar" 추가

var a = ['angel', 'clown', 'mandarin', 'sturgeon'];

var removed = a.splice(2, 0, 'drum', 'guitar');

console.log(removed); // []

console.log(a); // [ 'angel', 'clown', 'drum', 'guitar', 'mandarin', 'sturgeon' ]

// 3번 인덱스에서 한개 요소 제거

var a = ['angel', 'clown', 'mandarin', 'sturgeon'];

var removed = a.splice(3, 1);

console.log(removed); // [ 'sturgeon' ]

console.log(a); // [ 'angel', 'clown', 'mandarin' ]

// 2번 인덱스에서 두개 요소 제거하고 "trumpet", "parrot" 추가

var a = ['angel', 'clown', 'mandarin', 'sturgeon'];

var removed = a.splice(2, 2, "trumpet", "parrot");

console.log(removed); // [ 'mandarin', 'sturgeon' ]

console.log(a); // [ 'angel', 'clown', 'trumpet', 'parrot' ]

**Array.concat(value1, value2, ...)**

인자로 주어진 배열이나 값들을 기존 배열에 합쳐서 새 배열을 반환합니다.

var a = ['a','b','c'];

var b = ['d','e','f'];

var c = a.concat(b);

console.log(c); // [ 'a', 'b', 'c', 'd', 'e', 'f' ]

var d = ['g','h'];

var e = a.concat(a,b,d);

console.log(e); // [ 'a', 'b', 'c', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' ]

**Array.pop()**

배열에서 마지막 요소를 제거하그 그 요소를 반환합니다.

원 배열에서 마지막 요소가 제거 되며, 반환값은 제거된 요소

var a = ['a', 'b', 'c', 'd', 'e', 'f'];

var removed = a.pop();

console.log(removed); // f

console.log(a); // [ 'a', 'b', 'c', 'd', 'e' ]

**Array.push()**

배열의 끝에 하나 이상의 요소를 추가하고 배열의 새로운 길이를 반환

var a = ['a', 'b', 'c'];

var cnt = a.push('d');

console.log(cnt); // 4 - 추가 후 현재 배열 요소 갯수

console.log(a); // [ 'a', 'b', 'c', 'd' ]

var cnt = a.push('e', 'f');

console.log(cnt);// 6 - 추가 후 현재 배열 요소 갯수

console.log(a); // [ 'a', 'b', 'c', 'd', 'e', 'f' ]

**Array.shift()**

배열에서 첫 번째 요소를 제거하고 제거된 요소를 반환

원 배열에서 첫번째 요소가 제거 되며, 반환값은 제거된 요소

var a = ['a', 'b', 'c', 'd'];

var removed = a.shift();

console.log(removed); // a

console.log(a); // [ 'b', 'c', 'd' ]

**Array.unshift()**

새로운 요소를 배열의 맨 앞쪽에 추가하고 새로운 길이를 반환

var a = ['a', 'b', 'c'];

var cnt = a.unshift('d', 'e');

console.log(cnt); // 5

console.log(a); // [ 'd', 'e', 'a', 'b', 'c' ]

**Array.map(callback(currentValue[, index[, array]]))**

배열내에 모든 요소 각각에 대하여 주어진 함수를 호출한 결과를 모아 새로운 배열을 반환

callback

새로운 배열 요소를 생성하는 함수, 함수 안에는 다음 세가지 인수를 가집니다.

- currentValue 처리할 현재 요소

- index(선택) - 처리할 현재 요소의 인덱스

- array(선택) - map()을 호출한 배열

var a = [1, 4, 9, 15];

var b = a.map(function(el) {

return el +2;

});

console.log(b); // [ 3, 6, 11, 17 ]

var a = ['a', 'b', 'c', 'd'];

var b = a.map(function(el, index, a) {

a[index] = index + "=" + el;

return index;

});

console.log(a); // [ '0=a', '1=b', '2=c', '3=d' ]

console.log(b); // [ 0, 1, 2, 3 ]

**Array.forEach(callback(currentValue[, index[, array]])**

forEach 메서드는 주어진 함수를 배열 요소 각각에 대해 실행

callback

새로운 배열 요소를 생성하는 함수, 함수 안에는 다음 세가지 인수를 가집니다.

- currentValue 처리할 현재 요소

- index(선택) - 처리할 현재 요소의 인덱스

- array(선택) - forEach 을 호출한 배열

**3.3.1.7 내장 객체를 공부 하는 방법**

자바스크립트의 내장 객체의 프로퍼티와 속성을 모두 알기는 어렵습니다. 다만 큰 틀에서

Object, String, Number, Boolean, Array, Date, Function, RegExp, Math, Map, Set 정도를 공부하시면

됩니다.

물론 이 사양이 큰 틀이며 전부는 아닙니다. 자바스크립트는 매년 버전이 증가하면서 새로운 스펙이

추가가 됩니다. (ECMAScript6 사양 중 Promise 등)

자바스크립트의 내장 객체에 대한 설명 및 프로퍼티는 하기 문서에서 검색을 통해 쉽게 찾아

볼수 있습니다.

[https://developer.mozilla.org/ko/docs/Web/JavaScript/Guide](file://https://developer.mozilla.org/ko/docs/Web/JavaScript/Guide)

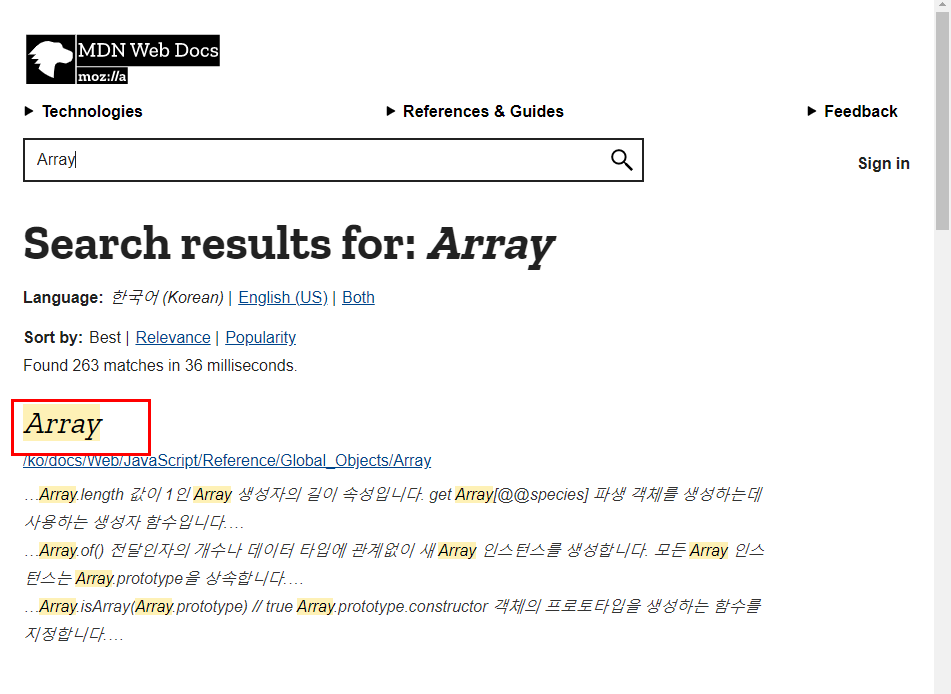
객 내장 객체를 검색하시면 왼쪽에 속성(Properties)와 메서드(Methods)를 확인 하실 수 있습니다.

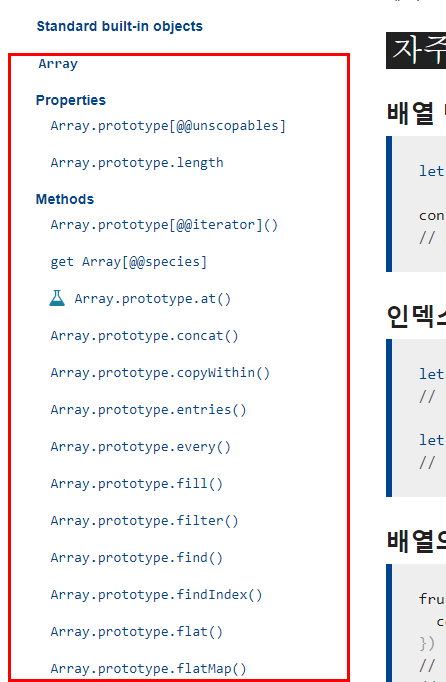
대강 어떠한 속성이 있는지 메서드가 있는지 눈으로라도 한번씩 익혀 놓으면 실제로 사용할때

쉽게 찾아서 적용하실 수 있습니다.

모든 속성, 메서드를 암기할 필요는 없습니다. 다만 자주 쓰는 속성과 메서드 위주로 용법을 익혀

두시고 주변적인 부분은 눈으로, 또는 한번씩만 실습해 주시면 됩니다.





**3.4 브라우저 객체 모델(Browser Object Model)**

브라우저에 내장된 객체를 '브라우저 객체'라고 합니다. window는 브라우저 객체의 최상의 객체이며,

window 객체에는 하위 객체가 포함되어 있습니다. 즉, 계층적인 구조로 이루어져 있습니다.

**브라우저의 객체의 계층적 구조**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| window | | | | |
| document | screen | location | history | navigator |

document는 window.document, screen은 window.screen, location은 window.location으로 접근

할 수 있으나 window.은 생략 가능합니다.

**3.4.1 window 객체**

window는 브라우저 객체의 최상위 객체입니다.

\* 자주 사용하는 메서드

window 객체의 메서드는 window는 생략이 가능합니다.

window.open(URL, “새창 이름”, “새창 옵션”)

URL 페이지를 새창으로 나타냅니다.

새창 옵션

- width 새창의 너비

- height 새창의 높이

- left 새창의 수평(x 축) 위치

- top 새창의 수직(y 축) 위치

- scrollbars 새창의 스크롤바의 숨김/노출을 설정합니다.

window.alert(”메세지”)

경고창을 나타내고 메세지를 보여줍니다.

window.prompt(”질문”, “답변”);

질문과 답변으로 질의응답 창을 나타냅니다.

window.confirm(”질문 내용”);

질문 내용으로 확인이나 취소 창을 나타냅니다. 확인 버튼을 누르면 true, 취소 버튼을

누르면 false를 반환 합니다.

window.moveTo(x, y)

지정한 새 창의 위치를 이동합니다.

window.resizeTo(width, height)

지정한 새 창의 크기를 변경합니다.

window.setInterval(function() { 자바스크립트 코드}, 일정시간 간격)

지속적으로 일정한 시간 간격으로 함수를 호출하여 코드를 실행합니다.

window.setTimeout(function() { 자바스크립트 코드}, 일정시간 간격)

단 한번 일정한 시간 간격으로 함수를 호출하여 코드를 실행합니다.

**3.4.2 screen 객체**

screen 객체는 사용자의 모니터 정보(속성)을 제공하는 객체입니다.

**\* 자주 사용하는 속성**

**screen.width** - 화면의 너비값을 반환

**screen.height** - 화면의 높이값을 반환

**screen.availWidth** - 작업 표시줄을 제외한 화면 너비값을 반환

**screen.availHeight** - 작업 표시줄을 제외한 화면 높이값을 반환

**3.4.3 location 객체**

location 객체는 현재 URL에 대한 정보(속성)와 새로고침(reload) 메서드를 제공합니다.

\* 자주 사용하는 속성

**location.href**

주소영역에 참조 주소를 설정하거나 URL을 반환합니다.

**location.hash**

URL의 해시값(#에 명시된 값)을 반환합니다.

**location.hostname**

URL의 호스트 이름을 설정하거나 반환합니다.

**location.protocol**

URL의 프로토콜을 반환합니다.

**location.search**

URL의 쿼리 요청값을 반환합니다.

\* 자주 사용하는 메서드

**location.reload()**

브라우저 새로고침

**3.4.4 history 객체**

history객체는 사용자가 방문한 사이트의 기록을 남기고 이전 방문사이트로 다시 돌아갈 수 있는

속성과 메서드를 제공합니다.

**\* 자주 사용하는 속성**

**history.length** - 방문 기록에 저장된 목록의 갯수를 반환

**\* 자주 사용하는 메서드**

**history.back()**

이전 방문 사이트로 이동, 숫자를 입력하면 숫자 단계만큼 이전 방문 사이트로이동

**history.forward()**

다음 방문사이트도 이동

**history.go(이동 숫자)**

이동 숫자 만큼 이전의 방문 사이트로 이동(없다면 1)

**3.4.5 navigator 객체**

navigator 객체는 현재 방문자가 사용하는 브라우저 정보와 운영체제 정보를 제공하는

객체입니다.

\* 자주 사용하는 속성

**navigator.language**

현재 브라우저가 사용하고 있는 언어를 반환합니다.

한국어를 사용할 경우에는 “ko”를 반환 합니다.

**navigator.platform**

현재 컴퓨터의 운영체제 정보를 제공합니다.

**navigator.onLine**

온라인 상태 여부에 대한 정보 제공, 인터넷이 정상적으로 연결되어 있는 경우 true

**navigator.userAgent**

브라우저의 운영체제의 종합정보를 제공

**3.4.6 브라우저 객체 속성, 메서드 공부 방법**

브라우저 객체의 속성 및 메서드는 상기 열거된 항목 외에도 더 있습니다. 다만 찾아서 확인 할 수

있는 방법이 있으며 실제 개발시에도 그러한 방식으로 찾아서 적절한 메서드를 찾아 보시면 됩니

다.

브라우저 객체의 최상위 책체는 window이며 그 하위의 객체인 document(문서 객체 모델), screen,

location. history 객체명을 기억하고 다음과 같은 방법으로 실제 속성과 메서드를 확인 하시면

됩니다. 물론 내장 객체도 마찬가지 입니다.

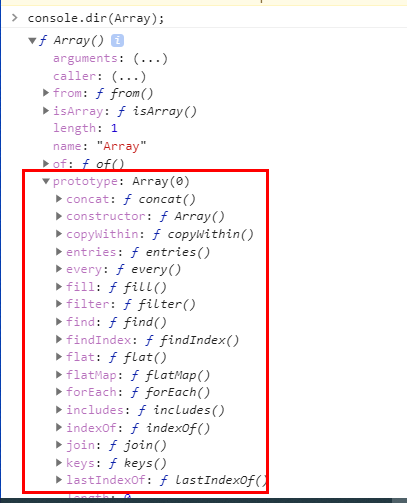
**브라우저 객체**

**console.dir(window.location)**



**내장객체**

**console.dir(Array);**



**3.5 자바스크립트 내장 함수**

내장 함수란 자바스크립트 엔진에 내장된 함수를 말합니다

|  |  |  |
| --- | --- | --- |
| 종류 | 설명 | 사용 예 |
| encodeURI() | 문자를 유니코드 값으로 인코딩 합니다. (영문, 숫자, 일부 기호(;, / ? : @ & = + $)는 제외 | [https://www.naver.com?value=값](file://https://www.naver.com/?value=값) https://www.naver.com?value=%EA%B0%92 |
| encodeURIComponent() | 문자를 유니코드 값으로 인코딩 합니다(영문, 숫자 제외) | [https://www.naver.com?value=값](file://https://www.naver.com/?value=값)  https%3A%2F%2Fwww.naver.com%3Fvalue%3D%EA%B0%92 |
| decodeURI() | 유니코드 값을 디코딩해 다시 문자화 합니다. |  |
| decodeURIComponent() | 유니코드 값을 디코딩해 다시 문자화 합니다. |  |
| parseInt() | 문자열 데이터를 정수형 데이터로 반환 | parseInt(”5.12”) -> 1 parseInt(”15px”) -> 15 |
| parseFloat() | 문자열 데이터를 실수형 데이터로 반환 | parseInt(”5.12”) -> 5.12 parseInt(”65.5%”) -> 65.5 |
| String() | 문자형 데이터로 반환합니다. | String(5) -> “5” (문자 5) |
| Number() | 숫자형 데이터로 반환합니다. | Number(”5”) -> 5 |
| Boolean() | 논리형 데이터로 반환합니다. | Boolean(5) -> true Boolean(null) -> false |
| isNaN() | is Not a Number의 약어이며 숫자가 아닌 문자가 포함되어 있으면 true를 반환 | isNaN(”abc”) -> true isNaN(52) -> false |

**3.6 기타**

**일반 함수**

function abc(x, y) {

return x + y;

}

**리터럴 방식**

var abc = function(x, y) {

return x + y;

}

**화살표 함수**

var abc = (x, y) => x + y;

var abc = (x, y) => {

return x + y;

}