

쇼핑몰을 만드는 프론트엔드 개발

자바스크립트 기초

김태곤



자바스크립트

- 1995년 5월, 넷스케이프 사에서 브랜든 아이크가 개발했다.
- 언어적으로는 자바와 아무런 상관이 없다.
- 공식적으로 JavaScript라 표기한다.
- 사실 오라클이 상표권을 가진 등록 상표이다. 표준 명세의 이름은 **ECMAScript(또는 ECMA-262)**이다.
- 2015년 6월, **ECMAScript 2015(또는 ECMAScript 6)**의 명세가 확정되었다.
- 현재 가장 널리 사용되는 것은 ECMAScript 5이다.

기본 문법

- 대소문자를 구분한다.
- 유니코드를 기반으로 하고 있어서 아스키 코드에 없는 언어로 코드를 작성할 수도 있고 문자열 안에서의 길이도 같다.
- 명령어는 문장(statement)라 부른다.
- 한 문장은 세미콜론(;)으로 끝맺는다.
- 스페이스, 탭, 줄바꿈 문자는 공백(white space)이라 부른다.

〈script〉

- 브라우저 환경에서 자바스크립트를 사용할 때 필요한 엘리먼트
- CSS처럼 HTML 문서 안에 여닫는 태그 사이에 코드를 입력할 수도 있고 외부의 자바스크립트 파일을 불러올 수도 있다.

```
<script type="text/javascript">  
console.log('Hello, world');  
</script>
```

```
<script type="text/javascript" src="a.js"></script>
```

- HTML5부터는 type 속성을 생략할 수 있고 생략하면 기본값은 "text/javascript"가 된다.

〈script〉

- 브라우저 환경에서 자바스크립트를 사용할 때 필요한 엘리먼트
- CSS처럼 HTML 문서 안에 여닫는 태그 사이에 코드를 입력할 수도 있고 외부의 자바스크립트 파일을 불러올 수도 있다.

```
<script>  
console.log('Hello, world');  
</script>  
  
<script src="a.js"></script>
```

- HTML5부터는 type 속성을 생략할 수 있고 생략하면 기본값은 "text/javascript"가 된다.

〈script〉

- 〈script〉는 기본 상태에서는 반드시 순차적으로만 실행된다.
- 브라우저는 〈script〉를 만나면 렌더링을 멈추고 다운로드 및 실행을 끝낸 후 다시 렌더링을 시작한다.

<https://stevesouders.com/hpws/move-scripts.php>

HTML 해석기

```
<!doctype html>  
<html>  
<head>  
  <meta charset="utf-8">  
  <title>자바스크립트 기초</title>
```

```
<script src="path/filename.js"></script>
```

 자바스크립트 엔진

HTML 해석기

```
</head>  
<body>  
  <p>반갑습니다.</p>  
</body>  
</html>
```


〈script〉

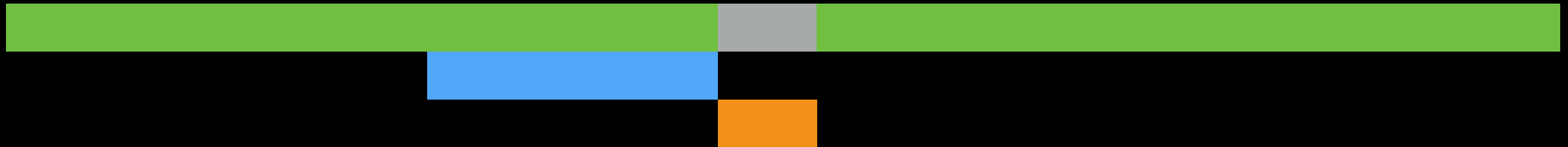
- 〈script〉는 기본 상태에서는 반드시 순차적으로만 실행된다.
- 브라우저는 〈script〉를 만나면 렌더링을 멈추고 다운로드 및 실행을 끝낸 후 다시 렌더링을 시작한다.
<https://stevesouders.com/hpws/move-scripts.php>
- 따라서, 닫는 〈/body〉 바로 위에 두는 것이 가장 좋다.
- defer나 async 속성을 추가하면 병렬로 읽어들이지만, 실행 순서가 보장되지 않으므로 주의해야 한다.

■ HTML 해석 ■ HTML 해석 정지 ■ 스크립트 다운로드 ■ 스크립트 실행

<script>



<script async>



<script defer>

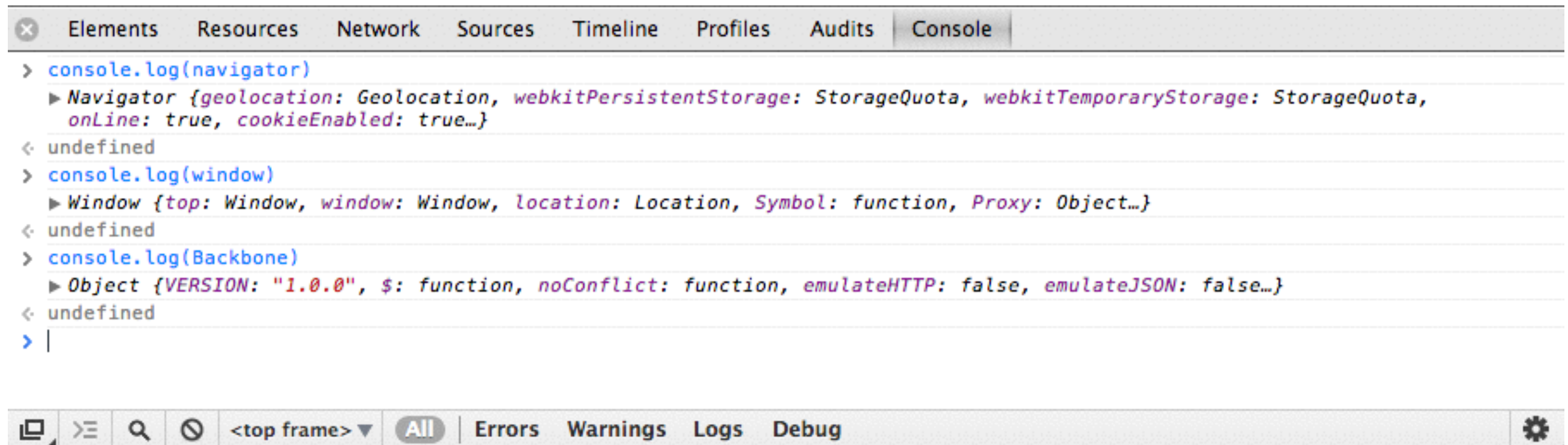


〈script〉

- 〈script〉는 기본 상태에서는 반드시 순차적으로만 실행된다.
- 브라우저는 〈script〉를 만나면 렌더링을 멈추고 다운로드 및 실행을 끝낸 후 다시 렌더링을 시작한다.
<https://stevesouders.com/hpws/move-scripts.php>
- 따라서, 닫는 〈/body〉 바로 위에 두는 것이 가장 좋다.
- defer나 async 속성을 추가하면 병렬로 읽어들이지만, 실행 순서가 보장되지 않으므로 주의해야 한다.
- 가능하다면 async를 사용하자.

Console API

- Chrome, Firefox 등 최신 브라우저에서 지원하는 개발자용 API
<https://developer.chrome.com/devtools/docs/console-api>
- 일단은 `console.log(value)` 사용



주석 (Comment)

- 한 줄 주석 : 슬래시 두 개 (//)로 시작. 줄 끝까지 주석으로 인식.
- 블록 주석 : /* 로 열고 */로 닫음. 여러 줄 주석 가능.

```
// 한 줄 주석입니다.
```

```
/*
```

```
여러 줄이 되는 주석입니다.
```

```
주석을 닫는 기호 뭉치만 아니면 무엇이든 입력할 수 있습니다.
```

```
*/
```

변수(Variables)

- `var` + 공백 + 변수이름 형태로 선언한다.

```
var num;
```

- `변수이름 = 값` 형태로 값을 할당한다.

```
num = 1;
```

- 선언과 할당을 동시에 할 수도 있다.

```
var num = 1;
```

- 쉼표로 구분하면 여러 변수를 선언할 수도 있다.

```
var num = 1, num2 = 2;
```

변수(Variables)

- 변수의 값을 다른 변수에 할당할 수 있다.

```
var num = 1, num2 = 2;  
  
num = num2;  
console.log(num); // 2
```

- 변수 이름은 \$, _를 제외한 공백, 특수 문자, 구두점을 사용할 수 없다. 숫자로 시작할 수 없다. 유니코드 문자도 사용 가능하지만 예약어는 사용할 수 없다.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar

- 선언되어 있지 않은 변수 이름에 var 키워드없이 값을 할당하면 전역 변수가 된다.

자료형 (Data Types)

- 원시 자료형 (Primitive Types):
숫자, 문자열, 불리언, null, undefined
- 객체 자료형 (Object Types):
배열, 오브젝트, 날짜 객체, **함수** 등 원시 자료형을 제외한 전부

In JavaScript, functions are first-class citizens.

자바스크립트에서 함수는 1급 객체이다.

이것만 기억하세요.

= 함수도 다른 자료형처럼 할당, 저장, 복사가 가능하다.

원시 자료형

- 해당하는 **리터럴 (literal)** 표현이 있다.

소스 코드에서 고정된 값을 표현할 때 사용.

ex) 1, true, "이름"

원시 자료형

- 해당하는 리터럴 (literal) 표현이 있다.
- 다른 언어의 원시 자료형과 비슷하지만, 자바스크립트는 모든 값을 객체처럼 다루므로 원시 자료형에도 프로퍼티와 메소드가 존재하는 것처럼 보인다.

```
"Hello, world".length; // 12
```

- 원시 자료형이 저장된 변수를 다른 변수에 할당하면 값 자체가 복사되고 복사된 변수를 변경해도 원래 변수는 변하지 않는다.

```
var color1 = "green";  
var color2 = color1;  
color2 = "blue";
```

```
var color1 = "green";
```

| Variable Object | |
|-----------------|---------|
| color1 | "green" |

```
var color1 = "green";  
var color2 = color1;
```

| Variable Object | |
|-----------------|---------|
| color1 | "green" |
| color2 | "green" |

```
var color1 = "green";  
var color2 = color1;  
color2 = "blue";
```

| Variable Object | |
|-----------------|---------|
| color1 | "green" |
| color2 | "blue" |

원시 자료형

- 해당하는 리터럴 (literal) 표현이 있다.
- 다른 언어의 원시 자료형과 비슷하지만, 자바스크립트는 모든 값을 객체처럼 다루므로 원시 자료형에도 프로퍼티와 메소드가 존재하는 것처럼 보인다(null, undefined 제외).

```
"Hello, world".length; // 12
```

- 원시 자료형이 저장된 변수를 다른 변수에 할당하면 값 자체가 복사되고 복사된 변수를 변경해도 원래 변수는 변하지 않는다.

```
var color1 = "green";  
var color2 = color1;  
color2 = "blue";
```

원시 자료형

- typeof 연산자를 사용해서 타입을 확인하기 쉽다.

```
typeof "Hello, world" // "string"
typeof 100 // "number"
typeof undefined // "undefined"
typeof false // "boolean"
typeof null // "object"
```

- 원시 자료형이지만 각 자료형을 표현하는 생성자가 있다.
String, Number, Boolean

숫자(Number)

- 정수, 실수 등을 표현하는 원시 타입
- 리터럴로 표현하는 숫자는 10진수가 기본이나 8진수 16진수 형태도 표현할 수 있다.

1234 123.4 0.81 .5 0xFF 0x1f 017

- 사칙연산(+, -, *, /), 나머지 연산(%), 증감 연산(++, --), 비트 연산(&, |, ^, ~, >>, <<, >>>), 단항 연산자(+, -)
- 숫자 자료형에는 NaN (Not a Number)도 있다.

문자열 (String)

- 문자열을 표현하는 원시 타입
- 따옴표 또는 작은 따옴표로 묶은 연속된 문자

```
"Hello, world" "문자열" '작은 따옴표'
```

- [] 연산자로 특정 인덱스의 글자만 가져올 수 있다.

```
"Hello, world"[1] // "e"  
"문자열"[0] // "문"
```

- + 연산자로 문자열 두 개를 연결할 수 있다.

```
"Hello, world" + "는 문자열" // "Hello, world는 문자열"
```

- 사실, 문자열 외의 다른 자료형도 연결할 수 있다.

문자열 (String)

- 다른 값을 문자열로 변환할 때는 String 생성자를 사용하거나 빈 문자열("")과 더하면 된다.

```
String(30) // "30"  
"" + 30 // "30"
```

연습해봅시다

"숫자 " + 7

"숫자 " + 7 * 3

"숫자 " + 7 + 3

→ "숫자 7" + 3

→ "숫자 73"

문자열 (String)

- 다른 값을 문자열로 변환할 때는 String 생성자를 사용하거나 빈 문자열("")과 더하면 된다.

```
String(30) // "30"  
"" + 30 // 30
```

- 문자열을 숫자로 바꿀 때는 parseInt 또는 parseFloat 함수를 사용하거나 단항 연산자(+, -)를 사용한다.

연습해봅시다

```
parseInt("30")  
parseInt("30", 16)  
parseInt("30a")  
parseInt("a30")  
parseInt("30.5")  
parseFloat("30.5")  
+"30"  
+"30.5"  
+"30a"  
-"30"
```

문자열 (String)

- 다른 값을 문자열로 변환할 때는 String 생성자를 사용하거나 빈 문자열("")과 더하면 된다.

```
String(30) // "30"  
"" + 30 // 30
```

- 문자열을 숫자로 바꿀 때는 parseInt 또는 parseFloat 함수를 사용하거나 단항 연산자(+, -)를 사용한다.
- 활용도가 높은 메서드: charAt(), charCodeAt(), replace(), indexOf(), lastIndexOf(), search(), slice(), split(), substr(), substring(), trim(), String.fromCharCode()
- 활용도가 높은 이스케이프 문자: \r, \n, \t, \xNN, \uNNNN

문자열 (String)

- ES6 backtick 문자(`)로 묶으면 템플릿 문자열이 된다.

```
var name = "김태곤";  
"안녕하세요, " + name + "님" // "안녕하세요, 김태곤님"  
`안녕하세요, ${name}님` // "안녕하세요, 김태곤님"
```

- ES6 템플릿 문자열 안에서 표현식도 사용할 수 있다.

```
var num = 30;  
"인생은 " + (num * 2) + "부터" // "인생은 60부터"  
`인생은 ${num * 2}부터` // "인생은 60부터"
```

불리언 (Boolean)

- true 또는 false 값을 가지는 자료형

null

- 아무 것도 없이 비어있는 값을 가리킨다.

```
var obj = null; // obj에는 아무 값도 없다.
```

undefined

- 변수를 선언하고 아무런 값도 할당하지 않을 때의 기본값

```
var name; // undefined
```

- 직접 할당할 수도 있다.

```
var name = undefined;
```

- undefined는 예약어가 아니라서 다른 값을 할당해도 문법 에러가 발생하지 않는다.

객체

- 자바스크립트의 모든 값은 객체처럼 다루어진다.



각 타입마다 고유한 속성과 실행 동작이 있다.

| | |
|------------|----------|
| 프로퍼티 | 메소드 |
| (property) | (method) |

```
var 문자열 = "프론트엔드 캠프";  
문자열.length;  
문자열.indexOf('캠프');
```

객체

- 각 타입에는 타입의 성격을 정하는 생성자(constructor)가 있다.
- 예를 들어, 앞서 배웠던 숫자는 Number라는 생성자가 있고, 문자열은 String이라는 생성자가 있다.
- 생성자의 이름은 대문자로 시작하는 관례가 있다.
- 각 프로퍼티와 메소드에는 고유한 이름과 값이 있다.



붕어빵틀 : 붕어빵의 모양을 정한다.
생성자 : 각 타입의 특성을 정한다.



붕어빵 : 붕어빵틀의 모양에 맞춰 만들어진 실제 빵
인스턴스(instance) : 실제로 사용할 수 있는 값

객체 자료형

- 참조 자료형 (Reference Type)이라고도 부른다.
- 객체 자료형의 값은 객체의 인스턴스(instance)이다.
- 객체값은 순서가 없는 프로퍼티(property)로 이루어지며, 프로퍼티는 문자열인 이름과 값으로 구성되어 있다.
- `new` 연산자를 사용해 인스턴스로 만든다.

```
var name = new Object();
```

- `typeof` 연산자를 사용하면 대체로 "object"가 반환된다.
(단, 함수는 "function", 정규표현식은 "regexp" 반환)

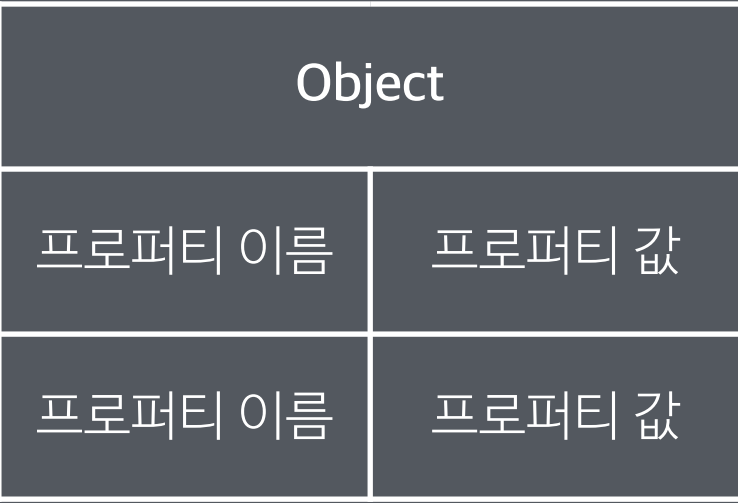
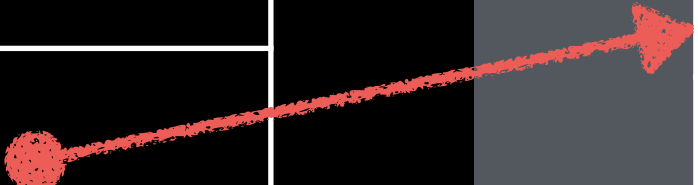
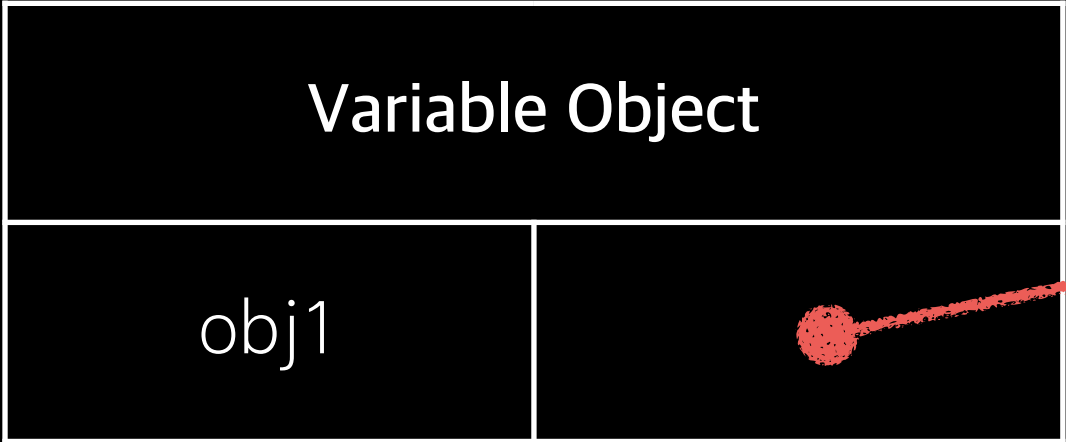
객체 자료형

- 변수에 저장 후 다른 변수로 복사해도 참조 (=연결고리)만 복사되고, 값 자체를 복제하지는 않는다.
따라서, 복사된 값을 변경하면 원래 값도 변경된다.



| Object | |
|---------|--------|
| 프로퍼티 이름 | 프로퍼티 값 |
| 프로퍼티 이름 | 프로퍼티 값 |

메모리

```
var obj1 = new Object();
```





```
var obj1 = new Object();  
var obj2 = obj1;
```

| Variable Object | |
|-----------------|--|
| obj1 |  |
| obj2 |  |

메모리

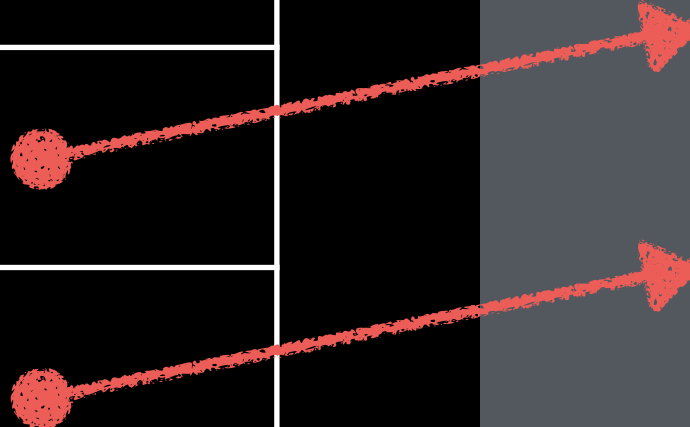
| Object | |
|---------|--------|
| 프로퍼티 이름 | 프로퍼티 값 |
| 프로퍼티 이름 | 프로퍼티 값 |

```
var obj1 = new Object();  
var obj2 = obj1;  
obj2.name = 'value';
```



| Variable Object | |
|-----------------|--|
| obj1 |  |
| obj2 |  |

| Object | |
|---------|--------|
| 프로퍼티 이름 | 프로퍼티 값 |
| 프로퍼티 이름 | 프로퍼티 값 |

메모리



```
var obj1 = new Object();  
var obj2 = obj1;  
obj2.name = 'value';
```

| Variable Object | |
|-----------------|--|
| obj1 |  |
| obj2 |  |

| Object | |
|---------|---------|
| 프로퍼티 이름 | 프로퍼티 값 |
| 프로퍼티 이름 | 프로퍼티 값 |
| name | "value" |

메모리

객체 자료형

- 변수에 저장 후 다른 변수로 복사해도 참조 (=연결고리)만 복사되고, 값 자체를 복제하지는 않는다.
따라서, 복사된 값을 변경하면 원래 값도 변경된다.
- `instanceof` 연산자를 통해 생성자-인스턴스의 관계를 알 수 있다.

```
var arr = new Array();  
arr instanceof Array
```


배열 (Array)

- 여러 값을 하나의 이름으로 묶어두고 사용할 수 있다.

```
var 장볼거1 = '돼지고기';
```

```
var 장볼거2 = '대파';
```

```
var 장볼거3 = '당근';
```

```
var 장볼거4 = '당면';
```

```
var 장볼거 = new Array('돼지고기', '대파', '당근', '당면');
```

- 배열의 각 값을 **원소(element)**라고 부른다.
- 배열의 크기는 `.length` 프로퍼티를 통해 알 수 있다.

```
console.log(장볼거.length);
```

배열 (Array)

- 생성자에 숫자를 한 개만 넣으면 배열의 크기가 되지만, 여러 개를 넣으면 배열의 원소가 된다.

```
var arr1 = new Array(); // 크기가 0인 배열  
var arr2 = new Array(10); // 크기가 10인 배열  
var arr3 = new Array(10, 20, 30); // 원소가 3개인 배열
```

- 리터럴 표현이 있다.

```
var arr4 = [ ]; // 원소가 없는 배열  
var arr5 = [10, 20, 30]; // 원소가 3개인 배열
```

- 각괄호 + 인덱스를 통해 각 원소에 접근할 수 있다.

```
var arr5 = [10, 20, 30]; // 원소가 3개인 배열  
arr5[0]; // 10
```

배열 (Array)

- 각괄호를 통해 특정 위치에 원소를 저장할 수도 있다.

```
var arr = [10, 20, 30];  
arr[1] = 50; // 10, 50, 30
```

- 리터럴 생성과 원소에 접근할 수도 있다.

```
[10, 20, 30][1]; // 20
```

- 활용도가 높은 메소드: concat(), filter(), forEach(), indexOf(), join(), map(), pop(), push(), reverse(), shift(), slice(), sort(), unshift()

오브젝트 (Object)

- 한 그룹으로 묶을 수 있는 값. 각 값에는 고유한 이름이 있다.

```
var 강사 = new Object();  
강사.이름 = "김태곤";  
강사.성별 = "남자";  
강사.이름 // "김태곤"
```

- 사실, 여기서 각 값은 오브젝트 타입 값의 프로퍼티이다.
- 프로퍼티의 이름을 가리켜 **키(key)**라고 부르고,
프로퍼티의 값을 가리켜 **값(value)**이라고 부른다.

오브젝트 (Object)

- 리터럴 표현을 사용하면 객체와 프로퍼티를 동시에 만들 수 있다.

```
var obj = {  
  newProperty: "value",  
  "property": 123  
};
```

- 리터럴 표현에서 키와 값 사이에는 콜론을 두어 구분하고, 키-값 쌍끼리는 쉼표로 구분한다.
- 원칙적으로 모든 키는 문자열이므로 따옴표로 묶어줘야 하지만, 변수 이름 규칙을 따른다면 따옴표를 생략할 수 있다.

오브젝트 (Object)

- 점 문법을 통해 프로퍼티에 접근하고 생성한다.

```
var obj = new Object();  
obj.newProperty = "value";  
obj.newProperty // "value"
```

- 각괄호 문법을 통해서도 프로퍼티에 접근하고 생성할 수 있다.

```
obj["newProperty"] = "value";  
obj["newProperty"] // "value"
```

오브젝트 (Object)

- in 연산자를 통해 특정 프로퍼티 이름이 객체에 존재하는지 확인할 수 있다.

```
"propertyName" in obj // true or false
```

- 활용도가 높은 메소드: ES6 Object.assign(),
Object.create(), Object.defineProperty(),
Object.defineProperties(), Object.keys(),
hasOwnProperty()

```
var obj = { "number": 123 };  
"number" in obj  
obj.hasOwnProperty("number")  
obj.hasOwnProperty("hasOwnProperty")
```

오브젝트 (Object)

- 사실, 자바스크립트의 모든 객체는 Object 타입을 상속하고 있다.

```
❑ instanceof Array  
❑ instanceof Object  
(function(){} ) instanceof Object
```


함수 (Function)

- function + 공백 + 함수 이름 + 괄호로 선언한다.

```
function 인사(){  
  console.log('안녕하세요');  
}
```

- 함수 이름 뒤에 괄호를 붙여서 실행한다.

```
인사();
```

- 다른 모든 타입은 "값"으로만 취급되지만, 함수는 "값"이기도 한 동시에 "행동" 또는 "기능"이기도 하다.

여러 차례 반복해야 하는 코드 뭉치를 편리한 이름으로 묶는다.

라면을 끓여달라고
부탁한다고 생각해봅시다.

치즈라면을 맛있게 드시는 방법

- 물 550ml(큰컵으로 2컵과 3/4컵)에 건더기스프를 넣고 물을 끓인 후 면과 분말스프를 넣고 **4분간** 더 끓여준 후 그릇에 담습니다.



4분간
조리



- **치즈별첨스프**를 넣고 잘 저어준 후 드시면 됩니다.

분말스프, 별첨스프는 식성에 따라 적당량 넣어 주시기 바랍니다.
치즈가 처음에 다 녹지 않더라도, 드시면서 녹을 수 있습니다.

"라면 끓여줘"

함수 선언 키워드


함수 이름

```
function 라면끓이기() {  
    냄비에넣는다(물550ml, 건더기스프);  
    끓인다();  
    냄비에넣는다(면, 분말스프);  
    끓인다(4분);  
    불을 끈다();  
    그릇에담는다();  
}
```

```
라면끓이기();
```

"난 더 꼬들하게"

```
function 라면끓이기() {  
    냄비에넣는다(물55ml, 건더기스프);  
    끓인다();  
    냄비에넣는다(라면, 분말스프);  
    끓인다(4분);  
    불을 끈다();  
    그릇에담는다();  
}
```



```
라면끓이기();
```

인수(argument)

```
function 라면끓이기(물끓일시간) {  
    냄비에넣는다(물550ml, 건더기스프);  
    끓인다();  
    냄비에넣는다(면, 분말스프);  
    끓인다(물끓일시간);  
    불을 끈다();  
    그릇에담는다();  
}
```

```
라면끓이기(4분);
```


함수(Function)

- function + 공백 + 함수 이름 + 괄호로 선언한다.

```
function 인사(){  
  console.log('안녕하세요');  
}
```

- 함수 이름 뒤에 괄호를 붙여서 실행한다.

```
인사();
```

- 다른 모든 타입은 "값"으로만 취급되지만, 함수는 "값"이기도 한 동시에 "행동" 또는 "기능"이기도 하다.

여러 차례 반복해야 하는 코드 뭉치를 편리한 이름으로 묶는다.

함수(Function)

- 함수 안에서 return 키워드를 사용하면 즉시 함수를 종료하고 키워드 뒤에 있는 값을 밖으로 반환한다.

```
function 두배(arg){  
    return arg * 2;  
};
```

```
var num = 두배(3);
```

- 괄호를 붙이지 않으면 "값"으로 취급된다. 복사된 함수도 똑같이 실행할 수 있다.

```
var x2 = 두배;  
var num2 = x2(5);
```

함수(Function)

- 생성자를 사용하는 방법보다는 리터럴 형식이 일반적이다.

```
var func1 = new Function('arg', 'return arg;');  
var func2 = function(arg){  
    return arg;  
};
```

- 이름을 주지 않고 만들 수도 있다.

```
function(arg){  
    return arg;  
}
```

함수 (Function)

- 함수 안에도 함수를 만들 수 있다.

```
function outer(arg) {  
  function inner(a) {  
    return a * 2;  
  }  
  return arg;  
}
```

- 함수를 실행하면 독립된 컨텍스트(context)가 생겨서,
이 때문에 변수와 함수의 스코프(scope)가 정해진다.
- 함수도 반환할 수 있다(closure).

```
function outer(arg) {  
    return arg;  
}
```

```
var name = "kim";
```

```
outer(name);
```

Global Context

outer

name

```
function outer(arg) {  
  return arg;  
}
```

```
var name = "kim";
```

```
outer(name);
```

Global Context

outer

name

Function Context (outer)

arg

```
function outer(arg) {  
  return arg + name;  
}
```

```
var name = "kim";
```

```
outer(name);
```

Global Context

outer

name

Function Context (outer)

arg

```
function outer(arg) {  
  return arg + name;  
}
```

```
var name = "kim";
```

```
outer(name);
```

Global Context

outer

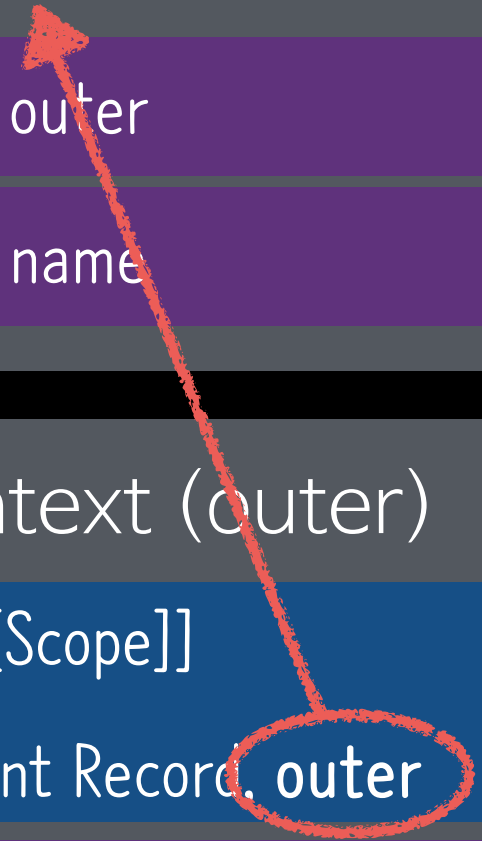
name

Function Context (outer)

[[Scope]]

Environment Record. outer

arg




```
function outer(arg) {  
  function inner( ) {  
    return arg + "name";  
  }  
  return inner;  
}
```

```
var name = "kim";  
var func = outer(name);
```

```
func();
```

Global Context

outer

name

Function Context (outer)

[[Scope]]

Environment Record, outer

inner

arg

```
function outer(arg) {  
  function inner( ) {  
    return arg + "name";  
  }  
  return inner;  
}
```

```
var name = "kim";  
var func = outer(name);
```

```
func();
```

Global Context

outer

name

Function Context (outer)

[[Scope]]

Environment Record, outer

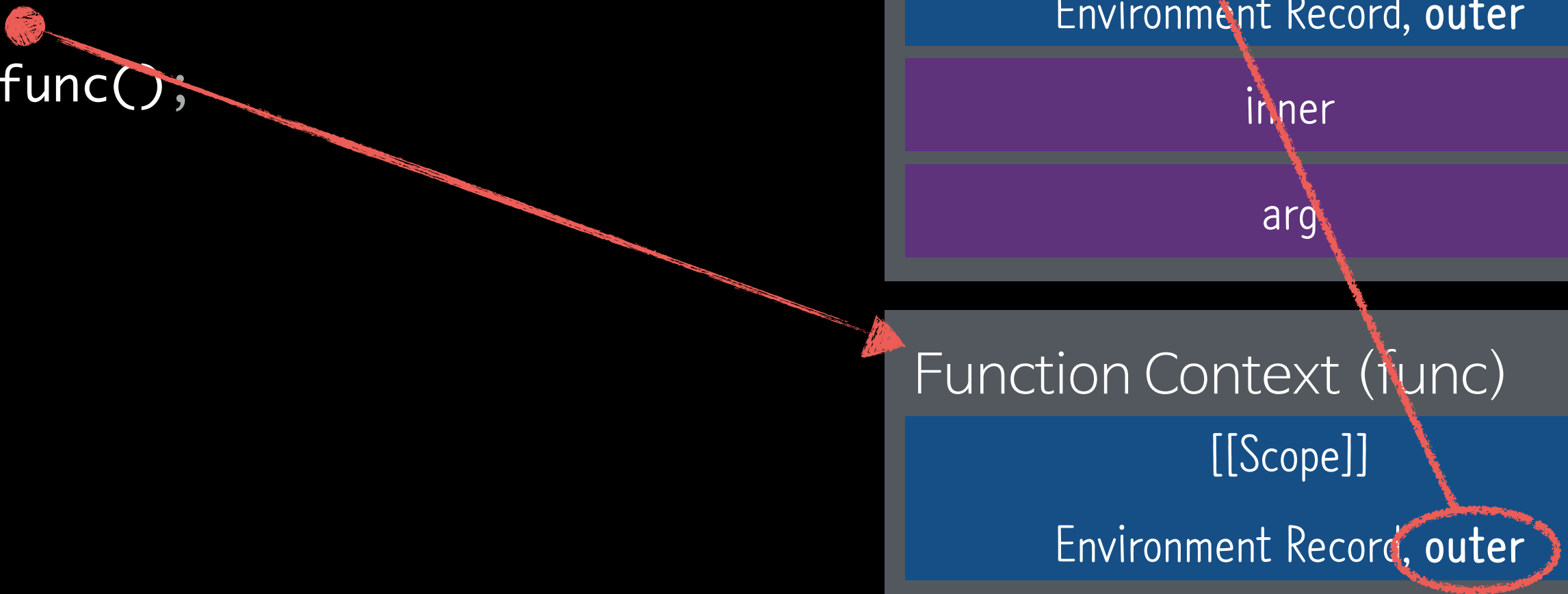
inner

arg

Function Context (func)

[[Scope]]

Environment Record, outer



```
function outer(arg) {  
  function inner( ) {  
    return arg + "name";  
  }  
  return inner;  
}
```

```
var name = "kim";  
var func = outer(name);
```

```
func();
```

Global Context

outer

name

Function Context (outer)

[[Scope]]

Environment Record, outer

inner

arg

Function Context (func)

[[Scope]]

Environment Record, outer

```
function outer(arg) {  
  function inner( ) {  
    return arg + "name";  
  }  
  return inner;  
}
```

```
var name = "kim";  
var func = outer(name);
```

```
func();
```

Global Context

outer

name

Function Context (outer)

[[Scope]]

Environment Record, outer

inner

arg

Function Context (func)

[[Scope]]

Environment Record, outer

arg?

```
function outer(arg) {  
  function inner( ) {  
    return arg + "name";  
  }  
  return inner;  
}
```

```
var name = "kim";  
var func = outer(name);
```

```
func();
```

arg?



Global Context

outer

name

Function Context (outer)

[[Scope]]

Environment Record, outer

inner

arg

Function Context (func)

[[Scope]]

Environment Record, outer

함수 (Function)

- 함수 안에도 함수를 만들 수 있다.

```
function outer(arg) {  
  function inner(a) {  
    return a * 2;  
  }  
  return arg;  
}
```

- 함수를 실행하면 독립된 컨텍스트(context)가 생겨서,
이 때문에 변수와 함수의 스코프(scope)가 정해진다.
- 함수도 반환할 수 있다(closure).
- 변수와 선언된 함수는 가장 먼저 정의된 듯 동작한다(hoisting).

```
function outer(arg) {  
  console.log(nowhere);  
  function inner( ) {  
    return arg + "name";  
  }  
  return inner;  
}
```



Error: nowhere is not defined

```
var func = outer("kim");
```

```
function outer(arg) {  
  console.log(inner);  
  function inner( ) {  
    return arg + "name";  
  }  
  return inner;  
}
```

```
var func = outer("kim");
```



```
function outer(arg) {  
  console.log(value);  
  function inner( ) {  
    return arg + "name";  
  }  
  
  var value = 'hello';  
  
  return inner;  
}  
  
var func = outer("kim");
```

어떻게 될까?

```
function outer(arg) {  
  console.log(value); // undefined  
  function inner( ) {  
    return arg + "name";  
  }  
  
  var value = 'hello';  
  
  return inner;  
}  
  
var func = outer("kim");
```

왜?

```
function outer(arg) {  
  var arg, value;  
  function inner( ) {  
    return arg + "name";  
  }  
  console.log(value);  
  value = 'hello';  
  
  return inner;  
}  
  
var func = outer("kim");
```

자바스크립트 해석기는 함수 내에 있는 변수 선언과 함수 선언을 먼저 처리하는데, 이 때문에 선언이 함수 가장 앞에 있는 듯한 **호이스팅(hoisting)** 현상이 발생한다.

함수 (Function)

- 함수 안에도 함수를 만들 수 있다.

```
function outer(arg) {  
  function inner(a) {  
    return a * 2;  
  }  
  return arg;  
}
```

- 함수를 실행하면 독립된 컨텍스트(context)가 생겨서,
이 때문에 변수와 함수의 스코프(scope)가 정해진다.
- 함수도 반환할 수 있다(closure).
- 변수와 선언된 함수는 가장 먼저 정의된 듯 동작한다(hoisting).

함수 (Function)

- 함수 안에는 arguments라는 변수가 자동으로 만들어진다.

```
function func(arg) {  
  console.log(arguments);  
  return arg;  
}  
func("hello");
```

- 함수도 객체이기 때문에 프로퍼티가 있다.

```
function func(arg) {  
  console.log(arguments);  
  return arg;  
}  
func.length; // 인수의 갯수 = 1  
func.name; // 함수의 이름 (ES6)
```

함수 (Function)

- 사실 자바스크립트의 모든 타입 생성자는 함수이다.

```
typeof Object  
Array
```

- 당연히 직접 만들 수도 있다.

```
function MyClass() {  
    ...  
}  
  
var myVar = new MyClass();
```

연산자

- ==와 === 중 자료형까지 확인하는 ===의 사용을 권장한다.

```
null == undefined  
null === undefined  
"0" == 0  
"0" === 0
```

- 비교 연산자는 Boolean 값을 반환한다.
- 할당 연산자는 할당된 값을 반환한다.

```
name = "김태곤"; // "김태곤"  
(name = "김태곤") + "의 블로그"; // "김태곤의 블로그"
```

연산자

- 논리 AND 연산자는 마지막 truthy 값 또는 첫 falsy 값을 반환하고
논리 OR 연산자는 첫 truthy 값 또는 마지막 falsy 값을 반환한다.
falsy value: false, 숫자 0, NaN, 빈 문자열, null, undefined
truthy value: falsy value를 제외한 나머지
- 부정 연산자(!)를 변수나 값 앞에 붙이면 Boolean 값이 반환된다.

내장 객체 (Built-in Objects)

- Math: 수학 관련 기능을 포함한다.
- JSON: JSON 문자열을 만들거나 해석한다.
- Date: 날짜와 시간을 다룬다.
- RegExp: 정규 표현식을 만들고 실행한다.

Math

- 인스턴스를 만들지 않는다.
- 자주 사용하는 상수: `Math.PI`
- 자주 사용하는 메소드: `Math.sin()`, `Math.cos()`, `Math.tan()`, `Math.ceil()`, `Math.round()`, `Math.floor()`, `Math.max()`, `Math.min()`, `Math.pow()`, `Math.random()`

```
// 연습1: 최댓값과 최솟값을 인수로 전달하면 랜덤한 수를 반환하는 함수
/* 연습2: 최댓값, 최솟값, 소숫점 자리수를 전달하면 랜덤한 수를 반환하
는 함수.
예) fn(5, 10, 2) // 5부터 10까지 무작위 실수. 소숫점 둘째자리
*/
```

DATE

- 날짜와 시간을 다루는 객체
- 기본 단위는 ms (1/1000 초)이다.
- 사용자 시스템의 시간을 기준으로 한다.
- RFC2822 또는 ISO 8601의 시간 포맷 문자열을 다룰 수 있다.

```
var birthday = new Date('1995-12-17T03:24:00');
```
- 아무런 인수없이 인스턴스를 만들면 현재 시각을 표현한다.

```
var nowDate = new Date();
```
- *FullYear(), *Month(), *Date(), *Hours(), *Minutes(), *Seconds(), *Milliseconds() 각 시간 컴포넌트별 메소드 존재

DATE

- *Day() 요일을 구하거나 설정 (0 ~ 6까지의 숫자로 반환)

```
// 연습: 날짜를 인수로 전달하면 요일을 한국어로 반환하는 함수
```

- *UTC*() 세계 표준시 기준 시간 메소드

- *getTimezoneOffset() 세계 표준시와 지역시의 차이(분 단위)

```
// 연습: 한국시를 문자열로 입력하면 뉴욕의 날짜와 시간을 반환하는 함수  
// nyTime("2015-09-25 06:30:00") > "2015-09-24 17:30:00"
```