

# HTML (Hyper Text + Markup Language)

## HTML이란?

### -Hyper Text

하이퍼텍스트는 하이퍼링크를 통해 독자가 한 문서에서 다른 문서로 즉시 접근할 수 있는 텍스트이다.

### -Markup Language

마크업 언어는 태그 등을 이용하여 문서나 데이터의 구조를 명시하는 언어의 한 가지이다.

즉 HTML은 하이퍼 텍스트로 이루어진 데이터 체계를 명시하는 언어이다.

HTML(HyperText Markup Language)은 World Wide Web(www)을 위한 마크업 언어이며, 제목이나 문단, 표, 꼬리글 및 기타 요소를 이용하여 콘텐츠의 구조를 정의함으로써 웹 문서를 만드는 기능을 제공한다

## HTML의 역사

팀 베너스 리(Tim Berners Lee)가 1989년 제안

1991년 HTML1

1995년 HTML2

1997년 1월 HTML3

1997년 12월 HTML4

2007년 HTML 5 발행

2014년 HTML 5 권고안 지정

2017 현재 HTML5 미완성

-W3C(1994년) 재단 창설:W3C는 웹표준 및 HTML 표준을 지정하는 기관(강제성 없음)

### 웹표준(HTML5) 등장 배경

HTML4 이후 각 웹브라우저 제조사들은 독자적인 언어와 플러그인을 추가하여 자신들의 브라우저를 발전시킴 동시에 W3C는 XHTML이라는 새로운 언어를 개발하기 시작했다. 이에 웹브라우저 제작사들은 새로운 언어인 XHTML에 따라 다시 브라우저를 개발해야 하고 각 브라우저마다 다른 표현으로 인해 하나의 웹 문서가 각 브라우저마다 다르게 보이는 문제점들이 생겼다. 그로 인해 WHATWG라는 새로운 웹표준 기관을 설립했다. W3C는 2009년 XHTML을 폐기하고 WHATWG의 요구 사항을 받아들여 웹표준 개발을 시작했다

-WHATWG :マイ크로 소프트를 제외한 애플, 모질라, 오페라 등의 브라우저 기업들이 설립한 독자적인 웹표준 기관

W3C의 XHTML의 호환성 문제에 대해 반발하여 업계의 요구를 수용한 새로운 웹표준을 제정하기 위해 설립됨

# CSS (Cascading Style Sheets)

## CSS란?

CSS는 마크업 언어(Markup Language)의 표현 방법을 기술하는 언어로, 미리 정해진 요소를 조합하여 마크업 언어가 전달하고자 하는 내용을 꾸미는 역할을 한다.

## CSS의 역사

**하킴 비움 리가 1994년 제안**

1996년 CSS1

1998년 CSS2

2005년 CSS3 발표 후 개발 중

W3C에서 개발 현황 확인 가능

# JavaScript

## JavaScript란?

JavaScript는 동적으로 컨텐츠를 바꾸고, 멀티미디어를 다루고, 움직이는 이미지등 웹 페이지를 꾸며주도록 하는 객체 기반 프로그래밍 언어이다

## JavaScript의 역사

1995년 넷스케이프 네비게이터(기관)가 브라우저 접두어를 장악하고 있는 환경에서 디자이너와 초보자를 위한 인터랙션 언어 javascript 제작했다. 1996년 표준화 작업 시작 이후에는 ECMAScript 라고도 불렸으며, 새로 나온 JavaScript 표준안인 ECMAScript 6를 줄여서 ES6라고 부르고 있다.

**초기 JavaScript는 웹브라우저 안에서 돌아가며, 주로 웹페이지에 효과를 주거나 기능을 향상시키는 목적으로 사용했으나 2009년 node.js의 등장으로 상황이 바뀐다**

Ryan Lienhart Dahl 이 2009년 처음 만든 node.js 는 서버에서 동작하는 JavaScript 프레임워크로, 이후 DB-서버-클라이언트를 모두 JavaScript로 구성할 수 있게 되면서 JavaScript의 영역을 넓힐 수 있는 계기를 만들었다. LAMP stack(Linux, Apache, MySQL, PHP/Python/Perl) 을 대체할 수 있는 MEAN stack (MongoDB, Express.js, Angular.js, Node.js) 이 부각되었으며 개발언어 전체를 JavaScript로 단일화 할 수 있으면서 DB에서 들고온 JSON을 클라이언트로 넘기는 데에 별다른 처리 없이 자연스럽게 전달할 수 있게 되었다. 이로 인해 현재는 V8엔진등과 같은 기술을 기반으로 서버(nodeJS, ...)와 클라이언트(electron, ...), 어플리케이션(React native, ...)등 여러 분야에서 사용되고 있다.

# 전역객체

## 전역객체란?

전역객체는 최상위에 위치한 객체이다. 웹 브라우저에서 사용되는 자바스크립트의 전역객체는 window라는 이름의 객체이고 node.js에서는 global이라는 이름의 전역객체가 있다.

자바스크립트에서는 코딩할 때 전역객체를 생략할 수 있다.

## 전역객체의 사용법

```
var number = 1;  
var o = {'func':function() {var number = 10};  
  
window.o.func();
```

window은 전역객체, 변수 number와 o는 전역변수, 함수 내 변수 number는 지역변수

함수내부에서 재선언(변수 앞에 var를 붙임)을 하면 함수내부에 종속되서 외부 호출이 불가능하고 재선언 없이 재정의(변수 앞에 var를 붙이지 않음)만 하면 전역변수의 값을 변경할 수 있다.

# 문법

## HTML의 문법

HTML은 태그로 이루어져 있다.

<tag></tag>와 같은 형태를 가지고 있고 꺽쇠 안에 tag가 들어가며 여는 태그와 닫는 태그가 영역을 이루고 있다.

그리고 단일태그라고 하는 태그들이 존재한다. <img>와 같이 태그가 없이 훈자 역할을 수행하는 태그들이며 대표적으로 id와 class같은 속성을 사용할 수 있다.

## CSS의 문법

```
selector{name:value;name:value;name:value}
```

## JavaScript의 문법

### camelCase

소문자로 시작하며 단어 뒤에 다른 단어가 오게 될 경우 다음 단어의 첫 스펠링은 대문자로 표기하는 방법

### 변수 var

```
var name= value;
```

name을 호출하면 value가 출력이 됨

### selector

```
document.querySelector('name');
```

querySelector()는 메소드

name은 css선택자

### method

```
selector.addEventListener();
```

```
document.querySelector('name') = selector
```

```
addEventListener = method
```

### function

```
function name() {}
```

함수 선언은 function으로 시작한다.

그 다음 함수명인 name이 오며 소괄호()는 함수에게 전달할 인자를 지정하는 곳이다.

중괄호 안에는 여러 명령어들을 넣어 하나의 기능으로 만들 수 있다.

### web page default structure

브라우저에 도메인 주소를 입력하면 브라우저에서 고유 주소를 알아낸다. 고유 주소를 알아낸 뒤 해당 주소에서 웹페이지를 띄우기 위해 필요한 정보를 서버에 요청한다. 서버는 요청 받은 항목들을 다시 브라우저에 전송한다.

브라우저는 html css javascript를 해석해서 우리가 보는 화면을 제공한다(=뷰어기능)

# 문법

## HTML과 XHTML의 문법 차이

HTML에서 인호부호는 가독성차원에서 생략하지 않으며 대부분은 속성 생략으로 전환되어 사용함

2017년 12월 14일을 기준으로 HTML5.2가 릴리즈 되었으며 <!doctype>에 HTML4와 XHTML 선언이 불가능

차이	HTML	XHTML
대소문자	구분 안함	구분
닫는태그	</i> (일부태그 가능)	< i></i> (불가능)
단일태그 슬레시	<img> (미사용)	<img /> (사용)
속성	<option selected> (생략)	<option selected="selected">
인호부호	<option selected=selected> (생략)	<option selected="selected">

# CODE 표기법

## 카멜 케이스(Camel Case)

낙타 등이 들쑥날쑥한 것처럼 각 단어의 첫 글자들만 대문자로 표기하는 방법이기 때문에 낙타표기법이라고 한다. 주로 클래스나 모듈의 명칭에 사용되며 각 단어의 첫 문자를 대문자로 표시하고 붙여쓰되, 맨 처음 문자는 소문자로 표기한다.  
ex)camelCase

## 스네이크 케이스(Snake Case)

여러 단어로 이루어진 경우 단어 사이를 구분자로 나누는 방식, 주로 변수명 등에 사용되는 경우가 많다.  
ex)snake\_case

## 케밥 케이스(Kebab Case)

하이픈으로 단어를 연결하는 방식이며 html 태그의 id, class속성으로 흔히 쓰인다.  
ex)kebab-case

# 웹 페이지를 표현하는 방식

## DOM(Document Object Model)과 CSSOM(CSS Object Model)

<https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model?hl=ko>

## Render-Tree Construction, Layout, and Paint

<https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction?hl=ko>

# 웹 페이지를 표현하는 방식

1. 링크가 클릭될 때 요청이 이루어진다.
2. 페이지와 소스가 다운로드된다.
3. 웹 브라우저가 페이지를 구현하기 위해 소스를 사용한다.
4. 페이지가 사용자에게 보여진다.

위의 각 단계마다 많은 구성 요소와 세부 정보가 있지만 이러한 4단계는 웹 페이지를 사용자에게 표현함에 있어서 주요한 것들이다. 기술적으로는 request, response, build, render라고 언급된다.

위의 각 단계들은 웹 페이지가 로드되는 동안 여러번 수행되는 경우가 많다.

## 1. REQUEST

우리가 웹 페이지가 요청받을 때 무슨 일이 일어나는지 알아보기 전에 웹 페이지가 무엇을 요청하는지 알아봐야 한다.

가장 보편적인 방법은 링크가 클릭될 때 웹페이지가 요청받는 것이지만 또 웹 페이지가 새로고침 될 때나 url이 브라우저에 입력될 때 요청받기도 한다.

이러한 순간들을 navigation start라고 부르며 이것은 기본적으로 웹 페이지를 표현하는 전체 과정 중 시작부분이다.

### DOCUMENT

링크를 클릭하면 문서에 대한 요청이 이루어진다.

문서는 웹 페이지 파일이다. 이 파일은 웹 서버에 전송된다. 이러한 파일은 html파일인 경우가 많지만 파일의 유형은 중요하지 않다. 중요한 것은 웹 브라우저로부터 요청받은 텍스트 파일이다.

HTTP라는 시스템을 이용하여 요청받는다. 우리는 웹 브라우저가 웹 서버로부터 파일을 요청한 것만 이해하면 되기 때문에 HTTP에 대한 이해는 필요하지 않다.

## 2. RESPONSE

웹 서버는 요청받은 파일을 웹 브라우저에 제공한다.

요청받은 파일이 단순한 HTML파일인 경우에는 페이지 로드의 요청 및 응답 단계가 완료된다.

하지만 거의 모든 웹 페이지들은 이미지나 CSS JS같은 파일을 가지고 있다.

이러한 것들을 리소스(resource)라고 부르며 이 리소스들을 웹 페이지에 나타내기 위해서는 웹 브라우저가 페이지 리소스들을 얻어야 한다.

### 웹 브라우저는 웹 페이지가 추가적인 리소스들을 필요해하는 것을 어떻게 알 수 있을까?

브라우저가 HTML 문서를 받으면 읽는다.

컴퓨터가 무언가를 찾고 있는 파일을 읽는 것을 'PARSING'이라고 한다.

웹 브라우저가 전체 HTML문서를 보고 페이지에 언급된 리소스들 예를 들어 CSS, JS, IMG 파일을 찾아낸다.

이러한 리소스들이 HTML에서 발견되면 웹 브라우저는 이러한 리소스 파일들을 웹 서버로부터 요청하고 그 리소스들은 브라우저로부터 다운로드 된다.

## 3.BUILD

웹 브라우저가 요청받은 리소스들을 가지게 되면 페이지 작성은 시작할 수 있다.

웹 브라우저가 페이지를 작성하는 방법은 HTML 문서에서 찾아낸 정보와 리소스 안에서 찾아낸 정보들을 결합하는 것이다.

1.BUILD THE DOM

2.BUILD THE CSSOM

3.BUILD THE RENDER TREE

### 1.BUILD THE DOM

DOM은 "Document Object Map"의 약자이다.

이것은 기본적으로 HTML에 따라서 페이지에 어떤 것들(태그)이 표시되는 지에 대한 지도이며 HTML을 기준으로 관계적인 맵핑 방식으로 웹 브라우저에 요소가 들어갈 틀을 만든다.

### 2.BUILDING THE CSSOM

CSSOM은 "CSS Object Map"의 약자이다. 이것은 기본적으로 CSS에 따라 스타일이 페이지의 다른 부분에 적용되어야 하는지를 나타내는 지도이다.

CSSOM은 스타일을 사용하여 상황을 표현하는 방법을 지정한다.

### 3.BUILDING THE RENDER TREE

렌더링 트리는 본질적으로 페이지가 실제로 배치되고 그려지는 방법에 대한 전체적인 지도를 만들기 위해 DOM과 CSSOM을 가져온 다음 결합한다.

## 4.RENDER

위 단계들이 끝나고 나면 마침내 브라우저는 화면에 무언가를 띠울 수 있다.

이러한 시점에서 브라우저는 표시해야 할 항목(DOM)과 표시 방법(CSSOM)과 그 두 가지(RENDERING TREE) 사이의 관계를 알 수 있다.

### Layout / Reflow

브라우저가 모르는 것은 표현된 모든 것들과 그 것들이 화면에서 끝나는 크기이다.

예를 들어 "사이드 바"라는 div가 오른 쪽 화면의 25%라고 가정해보자. 25%는 무엇인가? 이 부분을 layout 또는 reflow라고 하며 기본적으로 브라우저가 화면의 크기를 결정하고 페이지가 표시되는 방식에 영향을 미친다.

### Paint

위 계산이 완료되면 브라우저는 실제로 화면에 무언가를 표시할 수 있다.

이것을 paint라고 하며 이 마지막 단계에서 브라우저는 랜더링 트리의 각 노드를 실제 픽셀로 변환한다.

# HTML 콘텐츠 범주

각각의 HTML 요소는 자신이 가질 수 있는 내용의 종류를 정의하는 규칙을 준수해야 한다. 이 규칙들은 여러 요소들의 공통의 콘텐츠 모델로 그룹화된다. 각각의 HTML 요소는 하나 또는 여러 개의 콘텐츠 모델에 속해 있으며 각각의 HTML 를 설정에 대해서는 반드시 HTML 권고안을 따라야 한다.

## 메인 콘텐츠 범주

여러 요소들이 서로 공유하는 일반적인 콘텐츠 규칙을 설명한다.

### 메타데이터 콘텐츠

메타데이터 콘텐츠 범주에 속한 요소는 문서의 표현이나 동작을 수정하거나 다른 문서에 대한 링크를 설정하거나 다른 대역 외 정보를 전달한다.

메타데이터 콘텐츠 요소: <base>, <command>, <link>, <meta>, <noscript>, <script>, <style>, <title>

### 플로우 콘텐츠

플로우 콘텐츠 범주에 속한 요소들은 전형적으로 텍스트 또는 내장된 콘텐츠를 포함한다.

플로우 콘텐츠 요소: <a>, <abbr>, <address>, <article>, <aside>, <audio>, <b>, <bdo>, <bdi>, <blockquote>, <br>, <button>, <canvas>, <cite>, <code>, <command>, <data>, <datalist>, <del>, <details>, <dfn>, <div>, <dl>, <em>, <embed>, <fieldset>, <figure>, <footer>, <form>, <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, <header>, <hgroup>, <hr>, <i>, <iframe>, <img>, <input>, <ins>, <kbd>, <keygen>, <label>, <main>, <map>, <mark>, <math>, <menu>, <meter>, <nav>, <noscript>, <object>, <ol>, <output>, <p>, <pre>, <progress>, <q>, <ruby>, <s>, <samp>, <script>, <section>, <select>, <small>, <span>, <strong>, <sub>, <sup>, <svg>, <table>, <template>, <textarea>, <time>, <ul>, <var>, <video>, <wbr>, 텍스트

특정 조건이 충족된 경우에만 플로우 콘텐츠에 속하는 요소

<map> 엘리먼트의 자손인 경우: <area>

itemprop 속성을 가지고 있는 경우: <link>

itemprop 속성을 가지고 있는 경우: <meta>

scoped 속성을 가지고 있는 경우: <style>

### 구획 콘텐츠

섹션 콘텐츠 모델에 속한 엘리먼트는 <header> 엘리먼트, <footer> 엘리먼트 및 제목 콘텐츠 내용의 범위를 정의하는 섹션을 만든다.

구획 콘텐츠 요소: <article>, <aside>, <nav>, <section>

### 제목 콘텐츠

제목 콘텐츠는 명시적인 섹션 콘텐츠에 사용되거나 제목 콘텐츠 자체에 의해 암시적으로 정의되는 섹션의 제목을 정의한다

제목 콘텐츠 요소: <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, <hgroup> (<hgroup> 요소는 W3C에서 삭제되었으며 WhatWG에서만 존재함. 이론적인 시멘트 마크업만 가능)

## 구문 콘텐츠

구문 컨텐츠는 텍스트 또는 텍스트를 포함한 마크업을 정의한다

구문 콘텐츠 요소:`<abbr>`, `<audio>`, `<b>`, `<bdo>`, `<br>`, `<button>`, `<canvas>`, `<cite>`, `<code>`, `<command>`, `<datalist>`, `<dfn>`, `<em>`, `<embed>`, `<i>`, `<iframe>`, `<img>`, `<input>`, `<kbd>`, `<keygen>`, `<label>`, `<mark>`, `<math>`, `<meter>`, `<noscript>`, `<object>`, `<output>`, `<progress>`, `<q>`, `<ruby>`, `<samp>`, `<script>`, `<select>`, `<small>`, `<span>`, `<strong>`, `<sub>`, `<sup>`, `<svg>`, `<textarea>`, `<time>`, `<var>`, `<video>`, `<wbr>`,

공백으로 이루어지지 않은 일반 텍스트

특정 조건이 충족된 경우에만 구문 콘텐츠에 속하는 요소

구문 콘텐츠만 포함한 경우:`<a>`

`<map>`의 자손인 경우:`<area>`

구문 콘텐츠만 포함한 경우:`<del>`

구문 콘텐츠만 포함한 경우:`<ins>`

`itemprop`속성을 가지고 있는 경우:`<link>`

구문 콘텐츠만 포함한 경우:`<map>`

`itemprop`속성을 가지고 있는 경우:`<meta>`

## 내장형 콘텐츠

마크업 언어 혹은 문서 공간의 외부 리소스 포함한다

내장형 콘텐츠 요소:`<audio>`, `<canvas>`, `<embed>`, `<iframe>`, `<img>`, `<math>`, `<object>`, `<svg>`, `<video>`

## 대화형 콘텐츠

대화형 콘텐츠는 유저 인터렉션을 위해 특별하게 설계된 요소를 포함한다

대화형 콘텐츠 요소:`<a>`, `<button>`, `<details>`, `<embed>`, `<iframe>`, `<keygen>`, `<label>`, `<select>`, `<textarea>`

특정 조건이 충족된 경우에만 대화형 콘텐츠에 속하는 요소

`controls`속성을 가진 경우:`<audio>`

`usemap`속성을 가진 경우:`<img>`

`type`속성이 `hidden`이 아닌 경우:`<input>`

`type`속성이 `toolbar`에 속한 경우:`<menu>`

`usemap`속성을 가진 경우:`<object>`

`controls`속성을 가진 경우:`<video>`

## 폼 콘텐츠 범주

폼 관련 구성 콘텐츠 요소는 form을 소유자로 삼으며 form 속성에 의해 표현된다. form 소유자는 <form> 요소이거나 id가 form속성에 지정된 요소이다.

### LISTED

form.elements 및 fieldset.elements IDL collections 에 나열된 요소

<button>, <fieldset>, <input>, <keygen>, <object>, <output>, <select>, <textarea>.

### LABELABLE

<label> 요소와 연결할수 있는 요소

<button>, <input>, <keygen>, <meter>, <output>, <progress>, <select>, <textarea>.

### SUBMITTABLE

form을 제출할때 formdata를 구성할수 있는 요소

<button>, <input>, <keygen>, <object>, <select>, <textarea>.

### RESETTABLE

form 을 reset 할때 영향을 받는 엘리먼트

<input>, <keygen>, <output>, <select>, <textarea>.

## 특정 콘텐츠 범주

일부 요소만 공유하는 특정한 범주를 설명하며 특정 상황에서만 가능하다

### 특정 조건이 충족된 경우에만 플로우 콘텐츠에 속하는 요소

<map>엘리먼트의 자손인 경우:<area>

itemprop속성을 가지고 있는 경우:<link>

itemprop속성을 가지고 있는 경우:<meta>

scoped속성을 가지고 있는 경우:<style>

### 특정 조건이 충족된 경우에만 구문 콘텐츠에 속하는 요소

구문 콘텐츠만 포함한 경우:<a>

<map>의 자손인 경우:<area>

구문 콘텐츠만 포함한 경우:<del>

구문 콘텐츠만 포함한 경우:<ins>

itemprop속성을 가지고 있는 경우:<link>

구문 콘텐츠만 포함한 경우:<map>

itemprop속성을 가지고 있는 경우:<meta>

### 특정 조건이 충족된 경우에만 대화형 콘텐츠에 속하는 요소

controls속성을 가진 경우:<audio>

usemap속성을 가진 경우:<img>

type속성이 hidden이 아닌 경우:<input>

type속성이 toolbar에 속한 경우:<menu>

usemap속성을 가진 경우:<object>

controls속성을 가진 경우:<video>

# SECTIONING CONTENT와 HEADING CONTENT

## SECTIONING CONTENT

sectioning content는 제목을 가지고 있는 콘텐츠의 주제 그룹을 말한다. sectioning content는 식별되어야 하며 일반적으로 `<h1>~<h6>` 요소들을 자식으로 가진다.

sectioning content 요소:`<article>`, `<aside>`, `<nav>`, `<section>`

## HEADING CONTENT

sectioning content의 제목을 정의하는 역할을 한다.

heading content 요소:`<h1>~<h6>`

### **heading content 사용 시 주의사항**

heading content는 sectioning content가 선언되어 있지 않더라도 암시적으로 생성된 sectioning content의 제목을 정의한다. 예를 들어 `<header>`에 `<h1>`을 선언하게 되면 sectioning content에 속하지 않는 `<header>`는 sectioning이 이루어 지지 않는다. 대신 `<h1>`은 가장 가까운 sectioning content의 제목을 정의한다.

# SECTIONING ROOT

sectioning root는 자체 outline을 가질 수 있는 html요소이지만 내부의 section과 header는 해당 조상의 outline에 기여하지 않는다. 문서의 논리적 sectiong root인 `<body>` 옆에는 `<blockquote>`, `<details>`, `<fieldset>`, `<figure>` 및 `<td>` 같은 외부 콘텐츠를 페이지에 도입하는 요소가 있다.

ex)

```
<section>
<h1>Forest elephants</h1>
<section>
  <h2>Introduction</h2>
  <p>In this section, we discuss the lesser known forest elephants</p>
</section>
<section>
  <h2>Habitat</h2>
  <p>Forest elephants do not live in trees but among them. Let's
    look what scientists are saying in "<cite>The Forest Elephant in Borneo</cite>":</p>
  <blockquote>
    <h1>Borneo</h1>
    <p>The forest element lives in Borneo...</p>
  </blockquote>
</section>
</section>
```

위 예제의 outline 결과이다

1. Forest elephants
  - 1.1 Introduction
  - 1.2 Habitat

This outline doesn't contain the internal outline of the `<blockquote>` element, which, being an external citation, is a sectioning root and isolates its internal outline

# HTML TAG

## BASIC

### **<body>**

문서의 본문을 정의한다.

<body> html문서의 모든 내용을 포함한다.

### **<h1>~<h6>**

html의 제목을 정의한다.

검색 엔진은 표제를 사용하여 웹 페이지의 구조와 내용을 색인화한다. 사용자는 제목을 기준으로 페이지를 훑어보기 때문에 문서 구조를 표시할 땐 표제를 사용하는 것이 중요하다.

<h1>표제는 주 표제 다음에 <h2>표제 다음 덜 중요한 <h3>등으로 사용되어야 한다.

### **<p>**

paragraph 단락을 정의한다.

<p>요소 앞뒤에 약간의 여백을 자동으로 추가한다.

### **<span>**

<span>은 문서의 인라인 요소를 그룹화하는데 사용된다. <span>은 시각적인 변화를 주지 않는다. 이것은 텍스트의 일부 또는 문서의 일부에 hook를 더하는 방법을 제공한다. css로 스타일을 지정하거나 javascript로 조작할 수 있다.

### **<br>**

단일 줄 바꿈을 삽입한다.

### **<hr>**

콘텐츠의 주제별 변화를 정의한다.

## SECTIONING CONTENT

### **<section>**

<section>은 문서의 일반적인 구획을 나타낸다. 즉 전형적으로 제목을 가지고 있는 콘텐츠의 주제 그룹을 말한다. 각 <section>은 식별되어야 하며, 일반적으로 제목요소들을 자식으로 가진다. <section>을 일반적인 컨테이너로 사용하면 안된다. 그럴때는 구획화를 스타일링 목적으로 하는 <div>요소를 사용해야 한다.

### **<nav>**

네비게이션 링크를 정의한다.

문서의 모든 링크가 <nav>요소 내에 있어야 하는 것은 아니다. <nav>요소는 탐색 링크의 주요 블럭에만 사용된다. 문서는 여러개의 <nav>태그를 소유할 수 있다. 예를 들면 하나는 사이트 탐색에 다른 하나는 내부 페이지 탐색을 위해 사용할 수 있다.

장애를 가진 사용자가 이용하는 스크린 리더와 같은 사용자 에이전트는 처음 콘텐츠가 랜더링될 때 이 요소를 제거할 것인지를 결정할 수 있다.

### **<article>**

<article>은 문서, 페이지, 애플리케이션, 또는 사이트 안에 독립적으로 구분되거나 재사용 가능한 영역을 구성할 수 있다. 포럼의 글, 매거진/신문의 기사, 블로그 글 등이 여기에 포함된다. 이 요소는 일반적으로 제목을 자식으로 포함하여 식별되어야 한다. <article>요소가 중첩되어 있을 때 안쪽에 있는 요소는 바깥쪽에 있는 요소와 관련된 글을 나타낸다. 예를 들어 블로그 포스트의 댓글은 포스트를 나타내는 <article>요소 안에 중첩되는 <article>요소가 될 수 있다.

### **<aside>**

문서의 주요 콘텐츠에 별도로 이어진 콘텐츠가 있는 한 구획을 말한다. 주로 사이드바로 나타낸다. 주요 흐름의 일부로 파악되는 문단으로서 사용하면 안된다.

<address>의 후손이 될 수 없다.

<address>, <footer>, 다른 <footer>요소의 후손이여서는 안된다.

# NON-SECTIONING CONTENT

## 〈header〉

〈header〉는 소개나 탐색을 돋는 것의 그룹을 나타낸다. 이것은 일부 제목 요소들을 포함할 수도 있으나, 로고나 구획의 제목, 탐색 품과 같은 것들이 포함될 수 있다.  
이 요소는 후손이 될 수 없다. 〈address〉, 〈footer〉, 다른 〈footer〉요소의 후손이여서는 안된다.

## 〈footer〉

〈footer〉는 가장 가까운 구획화 콘텐츠나 구획화 루트의 푸터를 나타낸다. 〈footer〉는 일반적으로 구획, 저작권 데이터, 관련된 문서의 링크에 대한 정보를 포함한다.  
이 요소는 후손이 될 수 없다. 〈address〉, 〈header〉, 다른 〈footer〉요소의 후손이여서는 안된다..  
〈footer〉는 구획 콘텐츠가 아니므로 개요에서 새로운 구획을 소개하지 않는다.

# MAIN

## 〈main〉

〈main〉태그는 문서의 주요 내용을 지정한다. 〈main〉요소의 내용은 고유해야 한다. 사이드 바, 탐색 링크, 저작권 정보, 사이트 로고, 입력 양식과 같이 문서 전반에 걸쳐 반복되는 내용이 포함해서는 안된다.  
문서는 하나만의 〈main〉을 가져야하며 〈article〉, 〈aside〉, 〈footer〉, 〈header〉, 〈nav〉의 후손이 될 수 없다.

# FORMAGGING

## 〈addres〉

문서/기사의 저자/쇼유자에 대한 연락처 정보를 정의한다. 연락처 정보와 관련없는 임의의 주소를 나타내려면 〈address〉요소 대신에 〈p〉요소를 사용해야 한다. 전형적으로 이 요소는 〈footer〉내부에 위치된다.  
〈address〉요소에 들어 있는 작성자에 정보도 〈footer〉요소에 포함 될 수 있고 플로우 컨텐츠를 하용하는 모든 요소를 부모로 삼을 수 있다

## 〈blockquote〉

다른 소스에서 인용 된 section을 정의한다.

## 〈em〉

강조 텍스트를 정의한다.

# STYLE AND SEMANTICS

## 〈style〉

문서에 대한 스타일 정보를 정의한다.

## 〈div〉

〈div〉는 본질적으로 아무것도 나타내지 않는 플로우 컨텐츠의 일반적인 컨테이너이다. 꾸미는 목적(class나 id 속성을 쓰는 것과 같이), 또는 lang 속성처럼 속성 값을 공유하는 이유로 쓰는 요소들을 묶는데 사용된다. 이 요소는 다른 적절한 시멘틱 요소가 없을 때만 사용해야 한다.

# LINKS

## 〈a〉

〈a〉태그는 한 페이지에서 다른 페이지로 연결하는 데 사요외는 하이퍼 링크를 정의한다. 〈a〉요소의 가장 중요한 속성은 링크의 목적지를 나타내는 href속성이다. href속성이 없으면 download, herflang, media, rel, target, type속성은 존재할 수 없다. 링크 된 페이지는 다른 대상을 지정하지 않는 한 일반적으로 현재 브라우저 창에 표시된다.

## 〈link〉

문서와 외부 리소스 간의 관계를 정의한다. (대부분 style sheets에 연결하는데 사용된다)

# FORMS AND INPUT

## **<form>**

사용자 입력을 위한 html 양식을 정의한다.

## **<input>**

입력 컨트롤을 정의한다.

## **<textarea>**

여러 줄 입력 컨트롤(텍스트 영역)을 정의한다.

## **<button>**

클릭 가능한 버튼을 정의한다.

## **<select>**

drop down 목록을 정의한다.

## **<option>**

drop down 목록의 옵션을 정의한다.

## **<label>**

<input>요소에 대한 라벨을 정의한다.

## **<fieldset>**

<form>의 관련 요소를 그룹화한다.

# IMAGES

## **<img>**

이미지를 정의한다.

## **<map>**

클라이언트 측 이미지 맵을 정의한다.

## **<area>**

이미지 맵 내부의 영역을 정의한다.

## **<figcaption>**

<figure>요소에 대한 표제를 정의한다

## **<figure>**

자체 포함 된 콘텐츠를 지정한다.

# CSS BOX MODEL

## BOX MODEL

웹 사이트에서는 표시되는 방식이 모두 box모양이기 때문에 box모양을 표현하기 위한 방법이 필요하다. 가장 많이 쓰이는 3가지 속성은 block, inline-block, inline이다.

### BLOCK

block은 기본적으로 부모 사이즈(width)를 모두 상속받지만 자기 자신의 사이즈를 가질 수도 있다.

### INLINE-BLOCK

inline-block은 기본적으로 자손의 사이즈를 갖지만 자기 자신의 사이즈를 가질 수도 있다. inline속성 때문에 텍스트 관련 속성의 영향을 받는다. (text-align:right를 하면 요소 자체가 오른쪽으로 이동한다)

### INLINE

inline은 기본적으로 자손의 사이즈를 가지며 자기 자신의 사이즈를 가질 수 없다. (항상 auto) 텍스트 관련 속성의 영향을 받는다.

## BOX-SIZING

엘리먼트의 사이즈의 범주를 어디까지 잡을 것인지에 대한 속성이다.

### CONTENT-BOX

content-box는 default속성이며 width가 padding과 border를 제외한 content부분까지만 적용된다.

### BORDER-BOX

border-box는 width가 padding과 border까지 포함한 값으로 적용된다.

# RESET CSS와 NORMALIZE CSS

모든 웹 브라우저는 자신의 스타일 형식을 html 요소에 추가하여 읽기 쉬운 문서로 만든다. 하지만 모든 브라우저가 html을 동일한 방식으로 취급하는 것은 아니다. 예를 들어 사파리와 크롬은 같은 html문서를 다르게 나타내고 이러한 차이점은 기본적으로 제공하는 브라우저의 스타일 때문이다. 브라우저간의 스타일 불일치를 피하기 위해서는 내장 된 브라우저 스타일을 제거하여야 한다.

## RESET CSS

reset css는 html 요소에 대한 모든 defalt 스타일을 제거하기 위한 기본 템플릿이다. reset css는 커스텀될 수 있고 선호하는 스타일링 옵션을 추가할 수 있다.

## NORMALIZE CSS

normalize css는 html 요소에 대한 브라우저 간의 불일치를 제거한다. 이 것 또한 커스텀이 가능하며 선호하는 스타일링 옵션을 추가할 수 있다.

## RESET CSS와 NORMALIZE CSS의 차이점

두 css는 각종 브라우저들의 디폴트 값이 다르기 때문에 생겨났지만, reset css는 그 값을 0으로 초기화 시키려는 목적이 있으며 normalize는 초기화가 아닌 브라우저마다 다른 스타일을 동일하게 맞추려는 목적이 있다. 예를 들어 normalize css만을 사용하면 li에 기본적으로 적용된 스타일처럼 기본적으로 적용된 스타일은 초기화되지 않는다. normalize css는 common 스타일 영역에 가까우며 reset css와는 목적이 다르기 때문에 reset css 사용 후 중복되는 요소들을 적절히 제거하고 normalize css를 사용해야 한다.

# CSS방법론

css의 활용도가 높아지고 대규모 프로젝트가 많아짐에 따라 css에 다양한 방법론들이 생기기 시작했다.

SMACSS, BEM, OOCSS가 대표적 예이며

- 1.코드의 재사용성을 높임
- 2.유지보수를 쉽게 함
- 3.확장 가능함
- 4.클래스명 만으로도 무슨 의미인지 예측 가능하게 함  
와 같은 지향점을 가지고 있다.

## SMACSS(Scalable and Modular Architecture for CSS)

역할에 따른 분류-분류 및 큰 레이아웃

### SMACSS의 정의

CSS에 대한 확장형 모듈식 구조이며 CSS의 프레임워크가 아닌 하나의 스타일 가이드이다.

### SMACSS의 사용목적

- 1.class명을 통한 예측
- 2.재사용
- 3.쉬운 유지보수
- 4.확장 가능

### 1.기초(Base)

초기값 설정을 의미한다. 대부분 하나의 엘리먼트로 이루어져 있으며 속성 선택자, 가상 클래스선택자, 후손 선택자, 자식 선택자 등으로 이루어져 있다. 이것은 어떠한 클래스 또는 아이디 선택자를 포함하지 않는다.

예를 들어 heading 사이즈, 기본 링크 스타일, 기본 폰트 스타일과 바디의 배경 선언이 이에 포함된다. 스타일을 정의할 때 !important는 하용하지 않고 css reset파일도 이에 포함된다.

ex)  
body, form{  
 margin:0;  
 padding:0;  
}  
a{  
 color:#fff;  
}  
a:hover{  
 color:#000;  
}

## 2. 레이아웃(Layout)

css에는 두 가지 스타일이 있다. minor와 major 컴포넌트이다. minor 컴포넌트는 로그인 폼과 네비게이션 아이템들을 말한다. major 컴포넌트는 minor 컴포넌트를 포함하고 있는 <header>와 <footer> 등을 말한다. minor 컴포넌트를 만들 때는 다양한 곳에 사용될 수 있도록 아이디 선택자가 아닌 클래스 선택자를 사용해야 한다. 일반적으로 레이아웃 스타일은 단일 선택자를 사용해서 구성한다. 하지만 클라이언트 요청에 따라 다양한 레이아웃이 필요할 경우, 이미 선언한 스타일들을 조합해서 사용한다. 또한 두 개 이상의 클래스 혹은 아이디를 조합해서 기존에 선언했던 css속성을 바꿔줄 수 있다. 네이밍을 할 때도 아이디 선택자에는 특별한 네임 스피이싱 없이 사용하며 클래스 선택자는 접두사를 사용해서 목적을 분명히 하며 모듈화를 할 때도 용이하다.

ex)

### 레이아웃 선언

```
#header, #article, #footer{  
    width:960px;  
    margin:0 auto;  
}  
  
#article{  
    border:solid red;  
    border-width:1px 0 0;  
}
```

### 다른 레이아웃 스타일에 영향을 주는 상위 레벨 레이아웃 스타일 사용

```
#article{  
    float:left;  
}  
  
#sidebar{  
    float:right;  
}  
  
.l-flipped #article{  
    float:right;  
}  
  
.l-flipped #side-bar{  
    float:left;  
}
```

### 두 가지 레이아웃 스타일을 함께 사용하여 고정 레이아웃으로 전환

```
#article{  
    width:80%;  
    float:left;  
}  
  
#sidebar{  
    width:20%;  
    float:right;  
}  
  
.l-fixed #article{  
    width:600px;  
}  
  
.l-fixed #sidebar{  
    width:200px;  
}
```

### 3. 모듈(Module)

모듈은 페이지에서 좀 더 명확한 컴포넌트이다. navigation, slide, widget 등이 이에 포함되며 레이아웃 또는 다른 모듈안에 위치할 수 있다. 그렇기 때문에 독립적으로 존재할 수 있도록 디자인해야 한다. 모듈의 스타일을 선언할 때는 아이디 선택자와 엘리먼트 선택자를 피하고 클래스를 사용해야 한다. 만약 엘리먼트 선택자를 사용해야 한다면 child 선택자를 사용한다.

ex)

```
<div class="fld">
  <span>folder name</span>
</div>
```

```
.fld > span{
  padding-left:20px;
  background:url(icon.png);
}
```

여기서 문제는 프로젝트가 복잡해질수록 구성요소의 기능을 확장해야 할 가능성이 높아지고 규칙 내에서 이러한 일반 요소를 사용하는데 제한이 커진다는 것이다.

```
<div class="fld">
  <span>folder name</span>
  <span>folder icon</span>
</div>
```

그럴 때는 요소에 클래스를 추가하여 스타일을 지정할 때 모호성을 제거한다.

```
<div class="fld">
  <span class="fld-name">folder name</span>
  <span class="fld-icon">folder icon</span>
</div>
```

```
.fld > span.fld-name{
  padding-left:20px;
}
.fld > span.fld-icon{
  background:url(icon.png);
}
```

### 4. 상태(Status)

아코디언 섹션에서 본문이 접히고 확장되는 상태를 말한다. 같은 엘리먼트에 레이아웃 또는 base module 클래스로 적용할 수 있다. 예를 들어 header 엘리먼트에 아이디만 존재하면 현재 div, 또는 아코디언 섹션의 상태를 알 수 없다. 그렇기 때문에 현재 상태를 알려주기 위해 “is-collapsed” 클래스를 함께 적용한다.

ex)

```
<div id="header" class="is-collapsed">
  <form>
    <label for="searchbox" class="is-hidden">search</label>
    <input type="search" id="searchbox">
  </form>
</div>
```

#header 요소에는 아이디만 있다. 따라서 이 요소에 어떤 스타일이 있다면 이 요소에 레이아웃 용도로 사용하고 접힌 상태는 접힌 상태를 나타낼 것이라고 가정할 수 있다. 이 상태 규칙이 없으면 기본값은 펼쳐진 상태라고 가정할 수 있다.

## 5. 테마(Theme)

다른 규칙과 다르게 스타일의 핵심적인 부분을 담고 있지 않아 프로젝트에서 종종 제외되어 왔다. 애플리케이션과 사이트의 색과 이미지를 정의내려 사용자로 하여금 더욱 멋진 결과물을 만들어 낼 수 있다. 또한 해외 사용자를 겨냥한 폰트 스타일도 이에 포함된다.

### SMACSS의 장단점과 해결법

#### 장점

분류 및 큰 레이아웃

#### 단점

모듈화 규칙 부족

#### 해결법

모듈 부분 BEM방식으로 대체

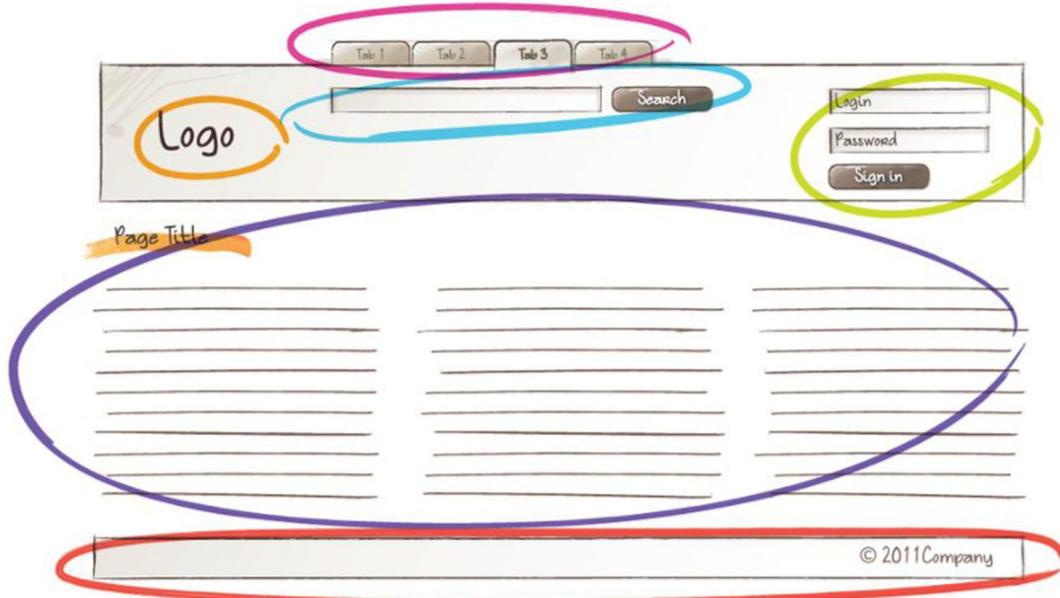
## BEM(Block Element Modifier)

클래스 네이밍 규칙-모듈 구성 최적화

### BEM의 정의

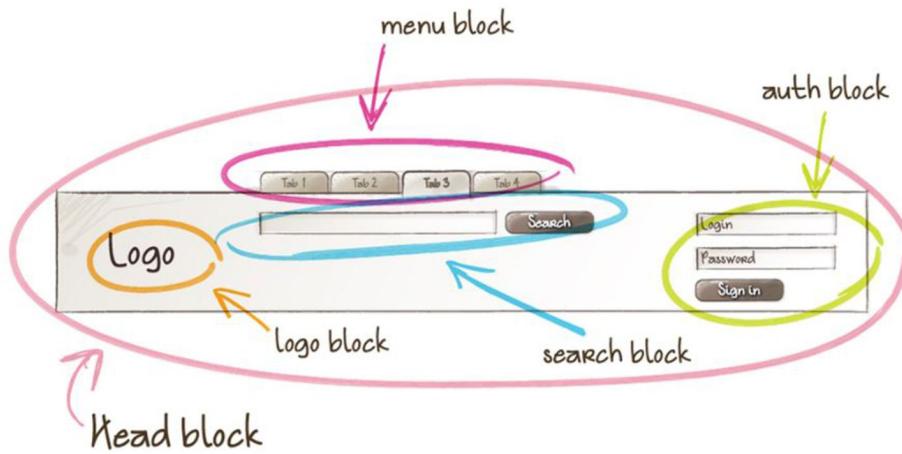
OOP와 유사하며 아이디는 사용할 수 없고 클래스명만 사용할 수 있다.

### 1. 블럭(Block)



기능적으로 독립적인 페이지 컴포넌트이다. JS, CSS, 템플릿 등을 담고 있으며 독립된 블럭 요소는 재사용이 가능하다. 블럭은 블럭 요소안에 위치할 수 있다. 예를 들어 head 안에는 로고, 검색폼, 메뉴가 포함되어 있다. 또한 이러한 컴포넌트들은 블럭 요소 안에서 자유롭게 배치되어야 한다.

## 2. 요소(Element)



블럭의 구성요소들은 블럭 밖에서는 사용을 할 수 없다. 예를 들어 메뉴라는 블럭에서 각각의 탭 메뉴들은 독립적으로 다른 곳에서는 작동을 하지 않는다. 이러한 요소를 엘리먼트라고 부른다.

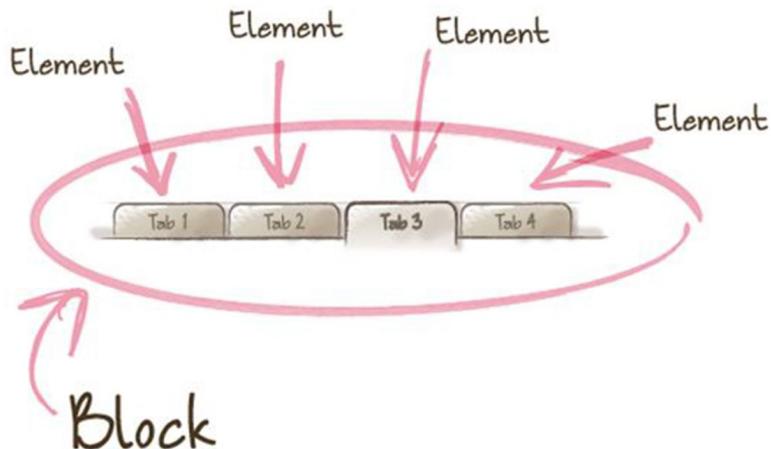
### 엘리먼트 네이밍

각 엘리먼트는 두 개의 밑줄표시로 연결하여 block 다음에 작성한다.

ex)

```
.header__logo{...}  
.header__menu{...}  
.header__search{...}
```

## 3. 수식어(Modifier)



블럭의 엘리먼트의 외관과 기능을 결정하는 BEM 독립체이며 필수보다는 옵션으로 사용한다. BEM 독립체는 blocks, element, modifiers를 부르는 단어이다. 수식어의 예로는 상단에 위치한 탭 메뉴를 하단이나 측면에 위치시킬 때는 똑같은 기능이지만 디자인이 수정된 것을 말한다.

### 수식어 네이밍

클래명은 "--"에 modifier를 추가한다

ex)

```
.header__navigation{...}  
.header__navigation--secondary{...}
```

## BEM의 장단점과 해결법

### 장점

모듈 구축에 특화

### 단점

필요 이상의 모듈화

### 해결법

레이아웃 요소 SMACSS로 제작

## OOCSS(Object Oriented CSS)

재사용 전략-재사용되는 부분요소

### OOCSS의 정의

Nicole Sullivan에 의해 개발된 프레임워크이며 CSS를 모듈 방식으로 코딩하여 중복을 최소화한다.

### 구조와 모양(Skin)의 분리

반복적인 시각적 기능(배경, 테두리 등)을 별도의 “skin”으로 정의하여 다양한 객체와 혼합하여 중복 코드없이 시각적 다양성을 표현할 수 있다.

ex)

```
.button { ... }
.box { ... }
.widget { ... }
.skin {
  background: linear-gradient(#ccc, #222);
  box-shadow: rgba(0, 0, 0, .5) 2px 2px 5px;
}
```

### 컨테이너와 콘텐츠의 분리

스타일을 정의할때 위치에 의존적인 스타일을 사용하지 않는다. 사물의 모양은 어디에 위치 하던지 동일하게 보인다.

ex)

```
<div class="header-inside globalwidth"></div>
<div class="main globalwidth"></div>
<div class="footer-inside globalwidth"></div>
```

```
.globalwidth{
  position: relative;
  padding-left: 20px;
  padding-right: 20px;
  margin: 0 auto;
  width: 980px;
  overflow: hidden;
}
.header-inside{
  padding-top: 20px;
  padding-bottom: 20px;
  height: 260px;
}
```

## OOCSS 네이밍

1. 가능한 짧고 간결하게 작성한다.
2. 동작과 형태가 예상 가능하도록 명확하게 작성한다.
3. 어떻게 생겼는지 보다는 어떤 목적인지 알 수 있도록 의미있게 작성한다.
- .지나치게 구체적 이지 않게 일반적으로 사용가능 하도록 작성한다.

## OOCSS의 장단점과 해결법

### 장점

반복적으로 사용되는 요소 특화

### 단점

모듈화를 위하여 너무 많은 클래스명이 필요함

### 해결법

반복적으로 사용되는 요소만 부분사용

# 객체지향 프로그래밍 (Object-Oriented Programming)

객체란 서로 연관된 변수와 함수를 그룹핑한 container이다. 객체 내 변수를 프로퍼티, 객체 내 함수를 메소드라고 부른다. 객체지향의 핵심은 객체라는 단단한 테두리 안에 서로 연관된 메소드와 변수를 모아서 정리하는 것이다.

## 1. 생성자와 NEW

생성자는 객체를 만드는 역할의 함수다. 자바스크립트에서 함수는 재사용 가능한 로직의 묶음이 아니라 객체를 만드는 창조자라고 할 수 있다. 함수를 호출할 때 `new`를 붙이면 새로운 객체를 만든 후에 이를 리턴한다. 자바스크립트에서 객체를 만드는 주체는 함수이다. 함수에 `new`를 붙이는 것을 통해서 객체를 만들 수 있다는 점은 자바스크립트에서 함수의 위상을 암시하는 단서이면서 자바스크립트가 추구하는 자유로움을 보여주는 사례이다.

## 2. THIS

`this`는 함수 내에서 함수 호출 맥락을 의미한다. 맥락이라는 것은 상황에 따라서 달라진다는 의미인데 즉 함수를 어떻게 호출하느냐에 따라서 `this`가 가리키는 대상이 달라진다는 뜻이다. 함수와 객체의 관계가 느슨한 자바스크립트에서 `this`는 이 둘을 연결시켜주는 실질적인 연결점의 역할을 한다.

## 3. 상속

상속이라는 것은 근본적으로 한 번 만들어 놓은 기능을 다시 만들지 않겠다는 의지의 표현이다. 즉 코드를 다시 재사용 할 수 있게 하는 것이다. 예를 들어 A라는 기능을 만들었는데 거기에 B라는 기능을 추가하고 싶다면 A를 복사해서 B기능을 추가해 넣으면 된다. 하지만 A코드가 중복되면서 코드 자체의 용량이 증가하게 되고 유지보수가 어렵게 된다. 그래서 객체의 로직을 그대로 물려 받는 또 다른 객체를 만들 수 있는 상속이 생겨난 것이다.

## 4. PROTOTYPE

`prototype`이란 말 그대로 객체의 원형이라고 할 수 있다. 함수는 객체다. 그러므로 생성자로 사용될 함수도 객체다. 객체는 프로퍼티를 가질 수 있는데 `prototype`이라는 프로퍼티는 그 용도가 약속되어 있는 특수한 프로퍼ти이다. `prototype`에 저장된 속성들은 생성자를 통해서 객체가 만들어질 때 그 객체에 연결된다.

## 5. 표준 내장 객체의 확장

표준 내장 객체는 자바스크립트가 기본적으로 가지고 있는 객체들을 의미한다. 내장 객체가 중요한 이유는 프로그래밍을 하는데 기본적으로 필요한 도구들이기 때문이다. 결국 프로그래밍이라는 것은 언어와 호스트 환경에 제공하는 기능들을 통해서 새로운 소프트웨어를 만들어내는 것이기 때문에 내장 객체에 대한 이해는 프로그래밍의 기본이라고 할 수 있다.

## 6. OBJECT

`Object` 객체는 객체의 가장 기본적인 형태를 가지고 있는 객체이다. 다시 말해서 아무것도 상속받지 않는 순수한 객체이다. 자바스크립트에서는 값을 저장하는 기본적인 단위로 `Object`를 사용한다. 동시에 자바스크립트의 모든 객체는 `Object` 객체를 상속받는데 그런 이유로 모든 객체는 `Object` 객체의 프로퍼티를 가지고 있다.

## 7. 데이터 타입

데이터 타입이란 데이터의 형태를 의미한다. 데이터 타입은 크게 두 가지로 구분할 수 있다. 객체와 객체가 아닌 것, 객체가 아닌 데이터 타입을 원시 데이터 타입이라고 한다. 그 외에 모든 데이터 타입들은 객체이다.

# 객체지향 프로그래밍 (Object-Oriented Programming)

객체란 서로 연관된 변수와 함수를 그룹핑한 container이다. 객체 내 변수를 프로퍼티, 객체 내 함수를 메소드라고 부른다. 객체지향의 핵심은 객체라는 단단한 테두리 안에 서로 연관된 메소드와 변수를 모아서 정리하는 것이다.

## 1. 생성자와 NEW

생성자는 객체를 만드는 역할의 함수이다. 자바스크립트에서 함수는 재사용 가능 로직의 묶음이 아니라 객체를 만드는 창조자이다. 자바스크립트에서 함수 앞에 new를 붙임으로서 객체를 만든다는 것은 함수의 위상을 알려주는 단서이면서 자바스크립트가 추구하는 자유로움을 보여주는 사례이다.

ex)

```
function Person(){}
var p = new Person();
p.name = 'egoing';
p.introduce = function(){
    return 'My name is'+this.name;
}
```

p는 Person{}, {}는 비어있는 객체라는 뜻이다.

앞에 new가 붙은 Person()을 생성자라고 부른다. Person()은 객체의 생성자이다.  
함수 앞에 new를 붙이면 그 리턴 값은 객체가 된다.

```
function Person(){}
var p1 = new Person();
p1.name = 'egoing';
p1.introduce = function(){
    return 'My name is '+this.name;
}
document.write(p1.introduce()+"<br />");

var p2 = new Person();
p2.name = 'leezche';
p2.introduce = function(){
    return 'My name is '+this.name;
}
document.write(p2.introduce());
```

중복이 일어난 코드를 다음과 같이 바꿔보자

```
function Person(name){
    this.name = name;
    this.introduce = function(){
        return 'My name is '+this.name;
    }
}
var p1 = new Person('egoing');
document.write(p1.introduce)+"<br />";

var p2 = new Person('leezche');
document.write(p2.introduce());
```

Person이라는 함수를 정의한다. 변수 p1에서 Person() 앞에 new를 붙여 Person()을 생성자로 만들고 인자 값으로 egoing을 준다. 현재 객체의 name의 프로퍼티 값은 egoing이 된다. 위 코드가 다 실행된 뒤 변수 p1은 egoing을 name의 프로퍼티 값으로 가지고 introduce 메소드를 가지고 있는 객체를 가지게 된다.  
여기서 중요한 건 각각 다른 name프로퍼티 값을 정의한 p1과 p2가 introduce 메소드를 재사용하고 있다는 것이다. 위와 같은 예제를 통해 코드 재사용의 중요성을 알 수 있다.

생성자 내에서 객체의 프로퍼티를 정의하는 것을 초기화(init)라고 한다.  
생성자 함수는 일반 함수와 구분하기 위해 첫 글자를 대문자로 표시한다.

## 2.THIS

this는 함수 내에서 함수 호출 맥락을 의미한다. 맥락이라는 것은 의미가 고정되어 있지 않고 상황에 따라서 달라진다는 의미인데 즉 함수를 어떻게 호출하느냐에 따라서 this가 가르키는 대상이 달라진다는 뜻이다. 함수와 객체의 관계가 느슨한 자바스크립트에서 this는 이 둘을 연결시켜주는 실질적인 연결점의 역할을 한다.

```
function func(){
  if(window === this){
    console.log("window === this")
  }
}
func();
```

결과는 window === this이다. func()를 호출했을 때 this는 전역객체인 window와 같다. func() 앞에는 window가 생략되어 있다(window.func()). func는 전역객체 window에 소속되어 있는 메소드이다.

### 메소드의 호출

```
var o = {
  func : function(){
    if(o === this){
      console.log("o === this")
    }
  }
}
o.func();
```

결과는 o === this이다. 이를 통해 알 수 있는 것은 어떤 메소드를 호출하면 메소드가 소속되어 있는 객체를 this로 접근할 수 있다는 것이다.

### 생성자와 THIS

```
var funcThis = null;

function Func(){
  funcThis = this;
}

var o1 = Func();           -> 일반 함수의 호출
if(funcThis === window){  함수를 호출하면 함수가 소속된 window가 this가 된다.
  console.log('window')
}

var o2 = new Func();        -> 생성자의 호출
if(funcThis === o2) log(message?: any, ...optionalParams: any[]): void
  console.log('o2')         this의 값은 생성자가 만든 객체를 가진 o2가 this가 된다. 생성자가 실행되기 전까지는
                           객체는 변수에 할당되지 않는다.
```

## APPLY

함수의 메소드인 apply를 이용하면 this의 값을 제어할 수 있다.

```
var o = {}
var p = {}

function func(){
    switch(this){
        case o:
            console.log(o);
            break;
        case p:
            console.log(p);
            break;
        case window:
            console.log('window');
            break;
    }
}

func(); //결과 window
func.apply(o); //결과 o
func.apply(p); //결과 p
```

위 예제에서 window, o, p 그리고 함수가 있다. func()를 호출했을 때 func()는 window의 메소드가 된다. func.apply(o)를 호출했을 때 func()는 o의 메소드가 된다. func.apply(p)를 호출했을 때 func()는 p의 메소드가 된다. func()를 어떻게 호출하느냐에 따라 소속되는 객체가 달라진다.  
객체는 주인이고 함수는 노예이다. 함수 또한 객체이기 때문에 함수와 객체는 대등하다. 하지만 함수를 어떤 맥락에서 호출하느냐에 따라 함수는 노예가 될 수 있다. 즉 함수를 어떻게 호출하느냐에 따라 함수는 맥락적으로 어떠한 객체에 소속될 수 있다는 것이다.

### 3. 상속

객체는 연관된 로직들로 이루어진 작은 프로그램이라고 할 수 있다. 상속은 객체의 로직을 그대로 물려 받는 또 다른 객체를 만들 수 있는 기능을 의미한다. 단순히 물려받는 것이라면 의미가 없을 것이다. 기존의 로직을 수정하고 변경해서 파생된 새로운 객체를 만들 수 있게 해준다.

#### 상속의 사용방법

```
function Person(name){  
    this.name = name;  
}  
Person.prototype.name=null;  
Person.prototype.introduce = function(){  
    return 'My name is '+this.name;  
}  
  
function Programmer(name){  
    this.name = name;  
}  
Programmer.prototype = new Person();  
  
var p1 = new Programmer('egoing');  
document.write(p1.introduce()+"<br />");//결과 My name is egoing
```

Programmer 안에는 introduce() 메소드가 정의되지 않았다. p1이 introduce() 메소드를 사용할 수 있는 이유는 Programmer가 introduce() 메소드를 상속하기 때문이다. 상속이 가능한 이유는 Programmer.prototype 값으로 Person() 생성자를 가지고 있기 때문이다. Person() 생성자를 통해 만든 객체는 name 프로퍼티와 introduce() 메소드를 가지게 된다.

어떠한 객체를 상속받고 싶다면 상속받고 싶은 객체를 다른 객체의 prototype 값에 생성자로 할당시키면 된다.

#### 기능의 추가

```
function Person(name){  
    this.name = name;  
}  
Person.prototype.name=null;  
Person.prototype.introduce = function(){  
    return 'My name is '+this.name;  
}  
  
function Programmer(name){  
    this.name = name;  
}  
Programmer.prototype = new Person();  
Programmer.prototype.coding = function(){  
    return "Hello world";  
} -> 추가된 부분  
  
var p1 = new Programmer('egoing');  
document.write(p1.introduce()+"<br />");//My name is egoing  
document.write(p1.coding()+"<br />");//Hello world
```

Programmer.property에 coding() 메서드를 추가함으로서 Programmer는 Person()의 기능을 가지고 있으면서 Person()이 가지고 있지 않은 기능인 메서드 coding()을 가지게 된다

## 04. PROTOTYPE

한국어로는 원형정도로 번역되는 prototype은 말 그대로 객체의 원형이라고 할 수 있다. 함수는 객체다. 그러므로 생성자로 사용될 함수도 객체다. 객체는 프로퍼티를 가질 수 있는데 prototype이라는 프로퍼티는 그 용도가 약속되어 있는 특수한 프로퍼티다. prototype에 저장된 속성들은 생성자를 통해서 객체가 만들어질 때 그 객체에 연결된다.

### prototype이란?

```
function Ultra(){}
Ultra.prototype.ultraProp = true;

function Super(){}
Super.prototype = new Ultra();

function Sub(){}
Sub.prototype = new Super();

var o = new Sub();
console.log(o.ultraProp); // 결과는 true
```

Ultra()라는 최상위 객체 안에 ultraProp 값이 true로 되어있기 때문에 결과는 true가 나온다.

우리가 얻고자하는 객체의 원형은 prototype이라고 하는 프로퍼티에 저장이 된다. 특수한 프로퍼티인 prototype 안에는 객체가 정의되어 있다. 생성자를 호출할 때 생성자는 생성자 함수의 prototype 프로퍼티에 정의되어 있는 객체를 꺼내서 리턴해준다.

### prototype chain

```
function Ultra(){}
Ultra.prototype.ultraProp = true;

function Super(){}
Super.prototype = new Ultra();

function Sub(){}
Sub.prototype = new Super();

var o = new Sub();
o.ultraProp = 1;
console.log(o.ultraProp); // 결과는 1
```

코드를 실행시키면 결과는 1이 된다.

생성자 Sub를 통해서 만들어진 객체 o가 Ultra의 프로퍼티 ultraProp에 접근 가능한 것은 prototype 체인으로 Sub와 Ultra가 연결되어 있기 때문이다. 내부적으로는 아래와 같은 일이 일어난다.

1. 객체 o에서 ultraProp를 찾는다.
2. 없다면 Sub.prototype.ultraProp를 찾는다.
3. 없다면 Super.prototype.ultraProp를 찾는다.
4. 없다면 Ultra.prototype.ultraProp를 찾는다.

prototype는 객체와 객체를 연결하는 체인의 역할을 하는 것이다. 이러한 관계를 prototype chain이라고 한다.

## 05. 표준 내장 객체

표준 내장 객체(Standard Built-in Object)는 자바스크립트가 기본적으로 가지고 있는 객체들을 의미한다. 내장 객체가 중요한 이유는 프로그래밍을 하는데 기본적으로 필요한 도구들이기 때문이다. 결국 프로그래밍이라는 것은 언어와 호스트 환경에 제공하는 기능들을 통해서 새로운 소프트웨어를 만들어내는 것이기 때문에 내장 객체에 대한 이해는 프로그래밍의 기본이라고 할 수 있다.

자바스크립트의 내장 객체: Object, Function, Array, String, Boolean, Number, Math, Date, RegExp

표준 내장 객체에 우리가 필요로 하는 기능이 없을 수도 있다. 그럴 때는 그 함수를 직접 만들 수도 있지만 객체에 메소드로 확장시키는 방법도 있다. 객체.prototype.메소드이름 = function() {}하게 되면 그 객체의 prototype에 메소드가 추가되면서 객체를 사용할 때도 메소드를 사용할 수 있게 된다.

# 06.OBJECT

Object 객체는 객체의 가장 기본적인 형태를 가지고 있는 객체이다. 다시 말해서 아무것도 상속받지 않는 순수한 객체이다. 자바스크립트에서는 값을 저장하는 기본적인 단위로 Object를 사용한다.

ex) var grades = {'egoing':10, 'k8805':6, 'sorialgi':801}

동시에 자바스크립트의 모든 객체는 Object 객체를 상속받는데 그런 이유로 Object 객체의 프로퍼티를 가지고 있다. 또한 Object 객체를 확장하면 모든 객체가 접근할 수 있는 API를 만들 수 있다. Object라고 하는 내장 객체는 모든 객체들이 가지고 있는 최상위 객체 즉 공통적인 최초의 조상이다. 그렇기 때문에 Object가 가지고 있는 메소드 중 prototype이 중간에 끼어있는 메소드들은 모든 객체들이 상속받고 있는 공통의 기능이다. 바꿔서 말하면 모든 객체들이 공통적으로 가져야 하는 기능이 있다면 그 기능은 Object의 prototype 객체를 수정하는 것을 통해 만들어질 수 있다.

다음 예제를 통해 어느 객체든지 사용할 수 있는 메소드를 만들어보자.

```
Object.prototype.contains = function(needle){
    for(var name in this){
        if(this[name] === needle){
            return true;
        }
    }
    return false;
}

var o = {'name':'egoing','city':'seoul'}//this는 o
console.log(o.contains('age'));//결과는 false
var a = ['egoing','leezche','grapittie'];//this는 a
console.log(a.contains('leezche'));//결과는 true
```

이런 식의 확장은 하지 않는 것이 좋다 그 이유는 모든 객체가 확장의 영향을 받기 때문이다. 아래 예제를 보자

```
Object.prototype.contains = function(needle){
    for(var name in this){
        if(this[name] === needle){
            return true;
        }
    }
    return false;
}

var o = {'name':'egoing','city':'seoul'}//this는 o
console.log(o.contains('age'));//결과는 false
var a = ['egoing','leezche','grapittie'];//this는 a
console.log(a.contains('leezche'));//결과는 true

for(var name in o){
    console.log(name); // 결과는 name city contain
}
```

작성자는 객체를 정의했을 때 for in문을 동작시켰을 때 자신이 정의한 데이터들(name,city)만 열거될 것이라는 것을 기대하고 있지만 결과는 그렇지 않다. Object에 프로퍼티와 메소드를 추가할 때는 신중해야 한다.

이 문제를 회피하기 위해서는 프로퍼티의 해당 객체의 소속인지를 체크해볼 수 있는 hasOwnProperty를 사용하면 된다.

```
Object.prototype.contains = function(needle){
    for(var name in this){
        if(this[name] === needle){
            return true;
        }
    }
    return false;
}

var o = {'name':'egoing','city':'seoul'}//this는 o
var a = ['egoing','leezche','grapittie'];//this는 a

for(var name in o){
    if(o.hasOwnProperty(name)){
        console.log(name); // 결과는 name city
    }
}
```

## 07. 데이터 타입

데이터 타입이란 데이터의 형태를 의미한다. 데이터 타입은 크게 두가지로 구분할 수 있다. 객체와 객체가 아닌 것. 그럼 객체가 아닌 것은 무엇일까?  
객체가 아닌 데이터 타입을 원시 데이터 타입이라고 한다.  
원시 데이터 타입: 문자, 문자열, 불리언, null, undefined  
이 외의 모든 데이터 타입들은 객체다.

### 래퍼 객체

```
var str = 'coding';
console.log(str.length);           // 결과는 6
console.log(str.charAt(0));        // 결과는 "c"
```

'coding'은 문자열이다. 하지만 console.log(str.length)나 console.log(str.charAt(0))의 문자열은 객체처럼 동작하고 있다. '.'의 정식 명칭은 object access operator이다. '.'을 썼다는 것은 앞에 있는 것이 객체라는 것을 의미한다. 따라서 str에 담겨있는 것은 객체이고 그것은 'coding' 즉 문자열이다. 하지만 자바스크립트에서 문자열은 원시 데이터 타입이다.

문자열은 분명히 프로퍼티와 메소드가 있다. 그렇다면 객체다. 그런데 왜 문자열이 객체가 아니라고 할까? 그것은 내부적으로 문자열이 원시 데이터 타입이고 문자열과 관련된 어떤 작업을 하려고 할 때 자바스크립트는 임시로 문자열 객체를 만들고 사용이 끝나면 제거하기 때문이다. 이러한 처리는 내부적으로 일어난다. 그렇기 때문에 몰라도 된다. 하지만 원시 데이터 타입과 객체는 좀 다른 동작 방법을 가지고 있기 때문에 이들을 분별하는 것은 결국엔 필요하다.

```
var str = 'coding';
str.prop = 'everybody';
console.log(str.prop); // 결과는 undefined
```

str.prop를 하는 순간에 자바스크립트 내부적으로 String 객체를 만들고 String 객체는 원시 데이터 타입을 감싸준다. prop 프로퍼티는 String 객체에 저장되고 String 객체는 곧 제거된다. 그렇기 때문에 prop라는 속성이 저장된 객체는 존재하지 않게 되기 때문에 결과는 undefined가 나온다. 이러한 특징은 일반적인 객체의 동작 방법과는 다르다. 자바스크립트는 원시 데이터 타입을 객체처럼 다룰 수 있도록 하기 위한 객체를 제공하는데 그것이 래퍼 객체(wrapper object)다.

래퍼 객체로는 String, Number, Boolean이 있다. null과 undefined는 래퍼 객체가 존재하지 않는다.

# 자바스크립트(ECMAScript) 버전

## JavaScript와 ECMAScript, JScript

우리가 흔히 자바스크립트라고 부르는 언어는 Ecma 인터내셔널이라 불리는 표준화 기구에서 제정한 ECMA-262 스크립트이다. 자바스크립트는 넷스케이프에서 제작하였고 이를 Ecma 표준으로 등록하는 과정에서 이미 등록되어 있는 JAVA와 유사한 명칭으로 인해 ECMA-262라는 이름을 가지게 되었으며 이를 표준화한 것이 ECMAScript이다. 마이크로소프트에서도 자바스크립트와 유사한 JScript를 제작하였고 이를 브라우저에 포함하였다. 자바스크립트와 JScript는 ECMAScript와의 호환을 목표로하면서 ECMA 규격에 포함되지 않는 확장기능들을 제공하고 있다. 우리가 자바스크립트라고 부르는 언어가 가리키는 것은 통칭 ECMAScript의 표준안에 해당하는 부분이기 때문에 각각의 브라우저에서 제공하는 자바스크립트 지원사항은 달라진다.

## ECMA-262 버전

ECMA-262는 지금까지 여섯 개의 버전이 출시되었고, 현재 6 버전(2015년 6월)이 최신버전이다. 4 버전은 중도 포기되었는데 타언어와 비슷한 개념들을 도입하면서 복잡성이 크게 증가하였고 관련된 정치적 문제점이 있었기 때문이다.

버전	출시일	차이점	브라우저지원
1	1997년 6월	초판	넷스케이프 4.5, IE4.0
2	1998년 6월	ISO/IEC 16262 국제 표준과 완전히 동일한 규격을 적용하기 위한 변경.	넷스케이프 4.5, IE4.0
3	1999년 12월	강력한 정규 표현식, 향상된 문자열 처리, 새로운 제어문, try/catch 예외 처리, 엄격한 오류 정의, 수치형 출력의 포매팅 등.	넷스케이프 6.0 이후의 넷스케이프와 모질라의 모든 버전에 포함, IE5.5
4	포기됨	4 버전은 언어에 얹힌 정치적 차이로 인해 버려졌다. 이 판은 작업 가운데 일부는 5 버전을 이루는 기본이 되고 다른 일부는 ECMAScrip트의 기본을 이루고 있다.	
5	2009년 12월	더 철저한 오류 검사를 제공하고 오류 경향이 있는 구조를 피하는 하부집합인 "strict mode"를 추가한다. 3 버전의 규격에 있던 수많은 애매한 부분을 명확히 한다.	이후 부터는 ECMAScript 호환 테이블이 제공되니 이를 참고. <a href="#">ECMAScript 5 호환성</a>
5.1	2011년 6월	ECMAScrip트 표준의 5.1 버전은 ISO/IEC 16262:2011 국제 표준 제3판과 함께 한다.	
6	2015년 6월	6 버전에서는 클래스와 모듈 같은 복잡한 응용 프로그램을 작성하기 위한 새로운 문법이 추가되었다. 하지만 이러한 문법의 의미는 5 버전의 strict mode와 같은 방법으로 정의된다. 이 버전은 "ECMAScript Harmony" 혹은 "ES6 Harmony" 등으로 불리기도 한다.	<a href="#">ECMAScript 6 호환성</a>
7	작업중	6 버전에 이어서 새로운 언어 기능이 추가될 예정이다.	

가장 범용적인 자바스크립트 버전은 1.5 버전이며, 이는 ECMAScript 3 버전에 해당하며, JScript 5.5 버전이 포함된 IE 5.5 및 기타 브라우저에서 모두 지원한다.

## ES5와 ES6

JavaScript는 ECMAScript(ECMA262)라는 사양을 기반으로 구현되어 있다. 현재 Web 브라우저는 ECMAScript 5.1th Edition을 기반으로 한 JavaScript 실행 엔진을 탑재하고 있다. 그리고 다음 버전인 ECMAScript 6th Edition이 현재 개정중으로, 약칭으로 ES6이라는 명칭이 사용되고 있다.

### 변수

ES6에서 새롭게 생겨난 변수, 상수 선언 키워드

`let` : 정의된 블록내에서만 존재하는 변수 선언 (지역 변수)

`const` : 위와 동일, 상수(변하지 않는 값) 선언

```
var x = 'global'; // 전역 변수
let y = 'logical'; // 지역 변수
console.log(this.x); // "global"
console.log(this.y); // undefined
```

### Scoping rules – var와 let 비교

```
//var
function varTest() {
  var x = 1;
  if (true) {
    var x = 2; // 위의 x와 같은 변수
    console.log(x); // 결과 : 2
  }
  console.log(x); // 결과 : 2
}
```

```
//let
function letTest() {
  let x = 1;
  if (true) {
    let x = 2; // 위의 x와 다른 변수
    console.log(x); // 결과 : 2
  }
  console.log(x); // 결과 : 1
}
```

위 예제에서 `let x`는 `if`문 내에서만 사용되는 변수이다. `let`은 내부 함수 코드를 명확하게 해준다.

## 변수와 호이스팅

호이스트란, 변수의 정의가 그 범위에 따라 선언과 할당으로 분리되는 것을 의미한다. 즉, 변수가 함수내에서 정의되었을 경우 선언이 함수의 최상위로, 함수 바깥에서 정의되었을 경우는 전역 컨텍스트의 최상위로 변경된다.

`var` : scope내 최상단으로 호이스팅된다.  
`let`과 `const` : TDZ (temporal dead zone, 임시사각지대) 블락 scope 내에서는 지역변수/상수에 대한 호이스팅이 이뤄지기는 하나, 선언된 위치 이전까지는 해당 변수/상수를 인식하지 못한다.

```
//var
function do_something() {
    console.log(foo); // var foo 까지만 호이스팅되어 undefined
    var foo = 5;
}
{
    var a = 2;
    {
        console.log(a); // 2
        var a = 3;
    }
}
```

```
//let
function do_something() {
    console.log(foo); // var foo 까지만 호이스팅되어 undefined
    let foo = 5;
}
{
    let a = 2;
    {
        console.log(a); // undefined
        let a = 3;
    }
}
```

## 전역변수

```
//var
var a = 1;
window.a; // 1
delete window.a; // false
```

```
//let
let b = 2;
window.b; // undefined
window.c = 3;
c; // 3
delete window.c; // true
c; // undefined
```

`var a = 1;`처럼 `var`로 변수를 선언하면 전역 변수가 되어 `window` 객체의 프로퍼티로 된다. 하지만 `delete window.a`로 삭제를 하려고 해도 삭제를 할 수가 없다.

`let`은 전역변수가 아니므로 `window` 객체에 담기지 않는다.

전역변수로 만들고 싶으면 `window.c = 3;`처럼 직접 `window` 객체의 프로퍼티로 만든다. 이 때에는 `delete window.c;`로 지울 수 있다.

## 클래스

Javascript는 프로토타입 기반(prototype-based) 객체지향형 언어다. 프로토타입 기반 프로그래밍은 클래스가 필요없는(class-free) 객체지향 프로그래밍 스타일로 프로토타입 체인과 클로저 등으로 객체 지향 언어의 상속, 캡슐화(정보 은닉) 등의 개념을 구현할 수 있다. Javascript Object-Oriented Programming ES5에서는 생성자 함수와 프로토타입을 사용하여 객체 지향 프로그래밍을 구현하였다. 하지만 클래스 기반 언어에 익숙한 프로그래머들은 프로토타입 기반 프로그래밍 방식에 혼란스러울 수 있으며 JavaScript를 어렵게 느끼게하는 하나의 장벽처럼 인식되었다.

ES6의 클래스는 기존 프로토타입 기반 객체지향 프로그래밍보다 클래스 기반 언어에 익숙한 프로그래머가 보다 빠르게 학습할 수 있는 단순명료한 새로운 문법을 제시하고 있다. ES6의 클래스가 새로운 객체지향 모델을 제공하는 것이 아니며 사실 클래스도 함수이고 기존 프로토타입 기반 패턴의 Syntactic sugar일 뿐이다.

### 클래스의 선언

```
//ES5
function Box(length, width) {
    this.length = length;
    this.width = width;
}
Box.prototype.calculateArea = function() {
    return this.length * this.width;
}
var box = new Box(2, 2); box.calculateArea(); //4
```

new 키워드로 생성자 함수를 만들고 this를 사용해 생성자 함수로 생성될 객체의 속성을 지정한다. 그 후에 prototype 메소드를 사용해 선언한 클래스(Box)에서 새로운 객체로 확장하는데 이런 방식 때문에 자바스크립트를 프로토타입 기반 프로그래밍 언어라 부른다. 프로토타입 사용의 장점은 메소드의 재사용성이다. 즉 자바스크립트에서 클래스와 같은 의미로 사용을 하는 것이다. 효율적으로 코드를 짜기 위해서는 새로 만드는 모든 메소드(예제 코드에서는 calculateArea)는 프로토타입 안에 있어야 한다.

위의 코드에서 calculateArea() 메소드를 생성 한 뒤에 box에 원형 객체 Box에 인자를 넣은 값을 담고 box.calculateArea()로 프로토타입에 선언한 메소드를 호출하면 정의한 연산을 거쳐 값이 출력된다.

```
//ES6
class Box {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
    calculateArea() {
        return this.length * this.width;
    }
}
let box = new Box(2, 2); box.calculateArea(); // 4
```

ES6에서 새로 생긴 클래스를 정의하는 방법은 다른 언어에서 클래스를 생성할 때와 비슷하다. class로 새로운 클래스를 선언을 해주고 동시에 속성과 메소드까지 정의를 해주고 호출을 할때는 마찬가지로 new키워드로 새로운 객체를 생성해 메소드를 호출해 결과값을 출력한다.

## 모듈

ES6 모듈은 JS 코드를 담고 있는 파일이다. module 같은 모듈을 위한 특별한 키워드는 존재하지 않는다. 모듈은 보기에도 그냥 일반 스크립트처럼 보이지만 다른 점이 있다면 딱 2개이다.

1. ES6 모듈은 모듈 안에서 "use strict"; 라고 적지 않아도 자동적으로 strict 모드로 처리된다.
2. 모듈 안에서 import 와 export 키워드를 사용할 수 있습니다.

모듈 안에 선언한 모든 것들은 기본적으로 해당 모듈 안에서만 참조 가능하다. 만약 모듈 안에 선언한 항목을 외부에 공개하고 싶다면, 그래서 다른 모듈들이 이용할 수 있게 하고 싶다면, 해당 항목을 export 해야 한다. export 하기 위한 몇 가지 방법이 있는데 가장 간단한 방법은 export 키워드를 덧붙이는 것이다.

### 모듈의 정의

모듈을 정의하는 'export' 문법은 아래와 같다.

```
export name1, name2, ..., nameN;  
export *;  
export default name;
```

name: export 할 속성이나 함수, 객체  
\*: 파일 내 정의된 모든 속성, 함수, 객체  
default: 모듈의 기본값. 파일 내에서 한 번만 호출될 수 있다.

ES6에 추가된 객체 리터럴(Enhanced Object Literals)을 함께 사용하면 더 간결하게 변수를 export 할 수 있다.

```
//ES6  
var foo = function () {};  
var bar = function () {};  
export { foo, bar };
```

# 서버 사이드 언어 ASP PHP JSP

웹 사이트의 프론트엔드 구현에는 HTML/CSS가 사용된다. 프론트엔드는 웹의 모양을 만들어주며 서버로 데이터를 전달하고 결과를 출력해주는 등의 역할을 수행하게 된다.

서버 개발언어 ASP PHP JSP는 서버사이드 언어라고 하며 실제 웹이 동작하게 하고 기능들이 구현되게 하는 프로그래밍 언어들 중 가장 많이 사용되는 언어이다.

## 1. ASP

Active Server Pages의 약자이다. MS사에서 서비스하는 프로그래밍 언어이다.

### 장점

비주얼 베이직 스크립트와 함께 사용이 가능하다. MS에서 제공하는 다양한 컴포넌트들을 활용할 수 있다.

### 단점

윈도우에서만 사용할 수 있다.

## 2. PHP

Hypertext Preprocessor의 약자이다. 오픈소스로 제공되는 명령형, 객체지향형 언어로 1/3에 해당하는 오픈소스 소프트웨어가 PHP로 구성되었다. 대표적으로 블로깅 도구 워드프레스, 미디어 위키등을 들 수 있고 국내에서도 그누보드 등이 PHP로 작성되어 있다.

### 장점

유닉스 계열 운영체제 뿐만 아니라 윈도우에서 사용 가능하다

Mysql, oracle, PostgreSQL, sysbase등 다양한 데이터베이스가 지원되기 때문에 사용자 편의성이 제공된다.  
설치도 쉽고 배우기도 쉬운 언어이다.

개발기간이 JSP에 비해 짧게 걸린다.

상대적으로 가볍기 때문에 구동 속도가 빠르다.

### 단점

JSP에 비해 안정적이지 못한 언어이기 때문에 트래피 다량 발생시 서버 속도가 느려진다.

## 3. JSP

JavaServer Pages의 약자이다. 이름에서 알 수 있듯 Oracle사에서 관리중인 Java 기반의 언어로 Java에서 제공하는 기능들을 그대로 사용할 수 있다.

### 장점

우수한 보안성과 다양한 기능으로 전자정부 표준으로 사용되는 언어이다.

Java의 이식성을 그대로 이어받아 리눅스, 윈도우 뿐 아니라 대부분의 다른 플랫폼에서도 운용이 가능하다.

# 반응형 웹(Responsive Web)

반응형 웹이란 디바이스 종류에 따르 웹 페이지의 크기가 자동적으로 재조정되는 것을 말한다. 어떠한 환경에서도 그에 맞게 사이즈가 변화되어 사용자가 보기 편리하게 만드는 웹이다. 오직 하나의 HTML 소스만으로 특정 장치에 최적화된 환경을 사용자에게 제공한다.

반응형 웹이 있어서 빼놓을 수 없는 개념이 모바일 퍼스트이다. 모바일 퍼스트는 웹 디자인을 할 때 PC보다 모바일 기기를 먼저 생각해서 디자인하고 프로그래밍하는 기법이다. 이 개념은 태블릿, 스마트 폰 등 모바일 기기의 이용이 늘어나면서 일반 웹 사용자보다 모바일 웹 사용자가 더 많아지면서 생기게 되었다. 모바일 퍼스트의 핵심은 모바일의 제약을 집중의 기회로 본다는 것에 있다. 모바일의 제약은 크게 세가지로 나눌 수 있다.

1. 모바일 기기의 스크린 크기
2. 네트워크 속도 및 품질
3. 사용하는 모드

위 세가지 제약을 통해 일반 웹은 모바일 웹에 있어서 불필요한 요소(기능/형식/꾸미기/이동)들을 가지고 있다고 판단한다. 불필요한 요소를 최소화시킨 모바일 웹은 사이트가 진짜로 제공해야 할 내용과 기능이 무엇인지 나타낸다. 그리고 사용자들에게 사용하기 편하고 작업을 빠르게 처리할 수 있는 웹을 제공한다.

반응형 웹은 고 사양(고 해상도) 웹이 저 사양(저 해상도) 모바일 기기에서도 불편함이 없이 구현이 되는데 초점이 맞춰져 있다. 주로 레이아웃 스타일 변화에 초점을 맞추어서 진행이 된다. 하지만 레이아웃과 스타일 보다는 웹 콘텐츠를 모바일 퍼스트 관점으로 재구성하는 반응형 웹도 있다. 이는 사용자가 사용하는 기능에 대해 연구하고 사용자가 필요로 하는 기능을 중심으로 우선 순위를 둘서 제공한다. 반응형 웹의 최종형은 모든 기기에서 사용자가 원하는 콘텐츠를 보기 좋게 제공하는 것이다.

## 반응형 웹과 적응형 웹

### 반응형 웹

하나의 페이지가 유동적인 레이아웃으로 pc 모바일 등 다양한 기기의 디스플레이 해상도에 맞추어 구성되는 웹 기술로서 %단위를 사용한 각 디자인의 폭에 유동적으로 반응형 콘텐츠의 크기가 커지거나 줄어드는 것을 말한다.

### 적응형 웹

정해진 해상도에 맞춰서 제작한 내용에 따라 화면이 구성되는 기술로서 브라우저가 미리 정해놓은 범위 사이즈에 속하여 그에 따라 레이아웃이 맞춰져서 보이는 것을 말한다. 적응형 웹은 px 단위를 사용하여 정해진 해상도에 적응한 형태로 화면에 표현되는 것이다.

반응형 웹은 어떤 기기에서든 큰 불편 없이 브라우저를 볼 수 있지만 신상품을 수시로 업데이트 해야 하는 쇼핑몰처럼 방대한 정보를 보여주어야 하는 웹사이트는 오히려 불편할 수 있다. 반면에 적응형 웹은 레이아웃과 디자인에 있어서 비교적 자유롭지만 모든 기기의 화면에 맞추기는 어렵다는 단점이 있다. %단위를 사용하여 디바이스의 해상도에 유동적으로 반응하여 컨테츠가 달라지는 반응형 웹의 단점을 보완하여 나온 것이 적응형 웹이다. 적응형 웹은 미리 정해놓은 해상도가 되면 정해진 레이아웃으로 변경된다. %단위를 사용하여 유동적으로 변경되는 반응형 웹과는 달리 px 단위를 사용하므로 유동적으로 변하지 않고 정해진 해상도에 적응된 형태로 레이아웃이 나타난다. 즉 반응형 웹은 모든 레이아웃이 서로 영향을 주며 유동적으로 바뀌는 반면 적응형 웹은 해상도가 달라지만 각각의 해상도에 적응시켜 놓은 전혀 다른 레이아웃과 디자인을 보여줄 수 있게 된다. 반응형 웹은 브라우저의 크기를 줄일 때마다 반응이 일어나 레이아웃의 변화가 생기지만 적응형 웹의 경우 특정 해상도까지 줄어들기 전에는 반응이 없다가 특정 해상도가 되면 레이아웃의 변화가 생긴다.

# 미디어 쿼리(media query)

css2.1부터 미디어 타입으로 단마리 종류에 따라 각각 다른 스타일을 적용시키는게 가능했다. 하지만 미디어 타입만으로는 해당 기기의 특성을 정확히 판단하기가 어려워 많이 사용되지 않았다. css3는 위 미디어 타입을 개선하여 좀 더 구체적인 조건으로 필요한 스타일을 적용할 수 있도록 확장하였는데 이를 미디어 쿼리라고 한다. 출력 장치의 여러 특징 가운데 일부를 참조하여 css코드를 분기 처리함으로써 하나의 HTML소스가 여러가지 뷰를 갖도록 구현할 수 있다. 일반적으로 뷰포트 해상도에 따라 css코드를 분기한다.

## 1.CSS 코드 내부에서 분기하는 방법

CSS코드 내부에서 사용하는 미디어 쿼리의 기본적인 문법 예는 다음과 같다. 일반적으로 권장하고 널리 쓰이는 방식이다.

```
@media only all and (조건문) {  
    /* 실행문 */  
}
```

### @media

미디어 쿼리가 시작됨을 선언한다. @media, only, all, and, (조건문) 사이에 포함되어 있는 공백은 필수적이다

### only

only 키워드는 미디어 쿼리를 지원하는 에이전트만 미디어 쿼리 구문을 해석하라는 명령이며 생략 가능하다. 생략했을 때 기본 값은 only로 처리된다.

### all

all 키워드는 미디어 쿼리를 해석해야 할 대상 미디어를 선언한 것이다. all이면 모든 미디어가 이 구문을 해석해야 한다. all 대신 screen 또는 print와 같은 특정 미디어를 구체적으로 언급할 수도 있다. all키워드는 생략 가능하고 생략했을 때 기본 값은 all로 처리된다.

### and

and키워드는 논리적으로 'AND'연산을 수행할 앞과 뒤의 조건을 모두 만족해야 한다는 것을 의미한다. 조건이 유일하거나 또는 only, all과 같은 선행 키워드가 생략되면 and 키워드는 사용하지 말아야 한다. and 대신 콤마 기호를 사용하면 'OR'연산을 수행한다. 'OR'연산은 나열된 조건 중 하나만 참이어도 실행문을 해석한다.

### 조건문

브라우저는 조건문이 참일 때 실행문을 처리하고 거짓일 때 무시한다. 조건문은 두 가지 이상 등장할 수 있다. 둘 이상의 조건문은 'and'키워드 또는 콤마 기호로 연결해야 한다.

### 실행문

일반적인 CSS코드를 이 괄호 안에 작성한다. 브라우저는 조건문이 참일 때 실행문 안쪽에 있는 CSS코드를 해석한다.

## 2.CSS 코드 외부에서 분기하는 방법

조건문에 따라 별도의 외부 CSS파일을 참조하여 분기하는 방법은 다음과 같다. 이 방식은 성능 최적화 측면에서 권장하지 않는다.

```
<link rel="stylesheet" type="text/css" media="all and (조건A)" href="A.css">
<link rel="stylesheet" type="text/css" media="all and (조건B)" href="B.css">
```

데스크탑 부라우저 사용자가 언제든 조건을 변경(예를 들면 창 크기를 조절해서 해상도를 바꿈)할 수 있기 때문에 웹 브라우저는 조건에 관계 없이 A.css파일과 B.css파일을 항상 요청한다. HTTP 요청을 불필요하게 두 번 발생시켜 이 페이지를 처음 로딩하는 사용자에게는 성능 저하의 원인이 된다 CSS파일은 하나로 병합하고 CSS 코드 내부에서 조건 분기하는 방식을 권장한다.

## 3.미디어 쿼리 템플릿

아래 코드는 모든 해상도를 커버하기 위한 미디어 쿼리 코드 템플릿이다. All, Mobile, Tablet, Desktop 으로 기기별 대응 코드를 분류 했지만 Liquid 레이아웃 기법을 사용하면 사실상 모든 해상도를 커버할 수 있다.

```
/* All Device */
/* 모든 해상도를 위한 공통 코드를 작성한다. 모든 해상도에서 이 코드가 실행됨. */

/* Mobile Device */
/* 768px 미만 해상도의 모바일 기기를 위한 코드를 작성한다. 모든 해상도에서 이 코드가 실행됨.
미디어 쿼리를 지원하지 않는 모바일 기기를 위해 미디어 쿼리 구문을 사용하지 않는다. */

/* Tablet & Desktop Device */
@media all and (min-width:768px) {
    /* 사용자 해상도가 768px 이상일 때 이 코드가 실행된다. 테블릿과 데스크톱의 공통 코드를 작성한다. */
}

/* Tablet Device */
@media all and (min-width:768px) and (max-width:1024px) {
    /* 사용자 해상도가 768px 이상이고 1024px 이하일 때 이 코드가 실행된다.
    아이패드 또는 비교적 작은 해상도의 랩톱이나 데스크톱에 대응하는 코드를 작성한다. */
}

/* Desktop Device */
@media all and (min-width:1025px) {
    /* 사용자 해상도가 1025px 이상일 때 이 코드가 실행된다. 1025px 이상의 랩톱 또는 데스크톱에 대응하는 코드를 작성한다. */
}
```

# CSS 단위

## 상대 길이 단위

상대적인 크기를 지니는 단위이다.

### %

% 단위로 지정된 속성 값은 다른 속성 값에 대한 상대적 비율로 정해진다. % 단위의 값이 어떻게 계산되는지는 CSS 속성에 따라 다르다. 예를 들어 width의 값을 %로 지정하면, 상위 요소의 width에 대한 상대값으로 해석된다. 상위 요소가 300px이고, 하위 요소가 50%라면 해당 하위 요소의 width는 150px로 표현된다.

### 글꼴에 상대적인 길이

#### em

em은 요소에 지정하는 글자 크기 단위다. em단위를 쓰면 부모 요소에서 지정한 글자 크기를 기준으로 배율을 조정한다. 따라서 2em을 지정하면 부모 요소의 글자 크기의 2배가 된다. 문서 레이아웃을 글자 크기에 따라 유동적으로 만들 때 많이 사용된다.

#### rem

rem은 root em이라는 뜻으로 HTML문서의 root요소인 <html>을 가리키며 이 요소에 지정된 크기를 기준으로 상대적인 값을 가지게 된다는 것이다.

### 브라우저 창에 비례한 길이

#### vh

사용자 창 높이의 1%에 비례한다.

#### vw

사용자 창 너비의 1%에 비례한다.

## 고정 길이 단위

부모나 기타 요소들에 영향을 받지 않고 항상 일정한 크기를 유지하는 단위이다.

#### px

px은 화소를 의미한다. 가장 이해하기 쉽다.

# 자바스크립트 모듈

프로그램은 작고 단순한 것에서 크고 복잡한 것으로 진화한다. 그 과정에서 코드의 재활용성을 높이고 유지보수를 쉽게 할 수 있는 다양한 기법들이 사용된다. 그 중의 하나가 코드를 여러 개의 파일로 분리하는 것이다. 이를 통해서 얻을 수 있는 효과는 다음과 같다.

1. 자주 사용되는 코드를 별도의 파일로 만들어서 필요할 때마다 재활용할 수 있다.
2. 코드를 개선하면 이를 사용하고 있는 모든 애플리케이션의 동작이 개선된다.
3. 코드 수정 시에 필요한 로직을 빠르게 찾을 수 있다.
4. 필요한 로직만을 로드해서 메모리의 낭비를 줄일 수 있다.
5. 한번 다운로드된 모듈은 웹브라우저에 의해서 저장되기 때문에 동일한 로직을 로드 할 때 시간과 네트워크 트래픽을 절약 할 수 있다. (브라우저에서만 해당)

순수한 자바스크립트에서는 모듈(module)이라는 개념이 분명하게 존재하지는 않는다. 하지만 자바스크립트가 구동되는 호스트 환경에 따라서 서로 다른 모듈화 방법이 제공되고 있다.

## 호스트 환경이란?

호스트 환경이란 자바스크립트가 구동되는 환경을 의미한다. 자바스크립트는 브라우저를 위한 언어로 시작했지만, 더 이상 브라우저만을 위한 언어가 아니다. 예를 들어 node.js는 서버 측에서 실행되는 자바스크립트다. 이 언어는 자바스크립트의 문법을 따르지만 이 언어가 구동되는 환경은 브라우저가 아니라 서버 측 환경이다. 우리는 자바스크립트의 문법을 이용해서 PHP와 같은 서버 시스템을 제어(node.js) 할 수 있다. 우리는 언어와 그 언어가 구동되는 환경에 대해서 구분해서 사고 할 수 있어야 하며 이를 위해서는 다양한 언어를 접해봐야 한다.

## 모듈의 효용

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
</head>
<body>
    <script>
        function welcome(){
            return 'Hello world'
        }
        alert(welcome());
    </script>
</body>
</html>
```

welcome 함수가 자주 사용되는 것이라고 가정해보자. 이런 경우 이것이 필요할 때마다 이 함수를 정의해서 사용하는 것은 유지보수도 어렵고 낭비가 될 것이다. 이럴 때 모듈이 필요하다. 함수 welcome을 모듈로 만들어보자.

greeting.js

```
function welcome(){
    return 'Hello world';
}
```

main.js

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <script src="greeting.js"></script>
</head>
<body>
    <script>
        alert(welcome());
    </script>
</body>
</html>
```

# 라이브러리와 프레임워크

## 라이브러리란?

라이브러리는 모듈과 비슷한 개념이다. 모듈이 프로그램을 구성하는 작은 부품으로서의 로직을 의미한다면 라이브러리는 자주 사용되는 로직을 재사용하기 편리하도록 잘 정리한 일련의 코드들의 집합을 의미한다고 할 수 있다. 사용 여부는 코드 작성자 선택 사항이며 새로운 라이브러리 제작 시에도 엄격한 규칙이 존재하지 않는다. 제작 의도에 맞게 작성하면 된다.

## 프레임워크란?

프레임워크는 Application 개발시 코드의 품질, 필수적인 코드, 알고리즘, 암호화, 데이터베이스 연동 같은 기능들을 어느 정도 구성이 되어있는 뼈대(구조)를 제공하도록 만들어진 것이다.

프레임워크는 이미 프로그래밍할 규칙이 정해져 있다. 예를 들어, 설정파일로 사용되는 XML에 어떤 태그를 써야하며, 어떤 함수를 추가적으로 작성해야하고, 소스 파일을 어느 위치에 넣어야하며, DB와 연동하기 위해 무엇을 써넣어야 하는지 정해져 있다. 보통 이런 대부분의 작업은 프레임워크가 하고자 하는 일에 비하면 아주 작은 일이며, 우리는 극히 일부분만 조정함으로써 목적을 달성할 수 있다.

## 라이브러리와 프레임워크의 차이

라이브러리와 프레임워크의 차이는 제어 흐름에 대한 주도성이 ‘누구에게’, ‘어디에’ 있는가이다. 라이브러리는 라이브러리를 가져다가 사용하고 호출하는 측에 전적으로 주도성이 있다. 반면, 프레임워크는 그 틀안에 이미 제어 흐름에 대한 주도성이 내재한다. 프레임워크는 가져다가 사용한다기보다는 거기에 들어가서 사용한다는 느낌이다. 즉, 어플리케이션의 Flow를 누가 주고 있느냐다.

# 함수 선언식과 함수 표현식

## 함수 선언식(Function Declarations)

일반적인 프로그래밍 언어에서의 함수 선언과 비슷한 형식이다.

```
function funcDeclarations() {
  return 'A function declaration';
}
funcDeclarations();
```

## 함수 표현식(Function Expressions)

유연한 자바스크립트 언어의 특징을 활용한 선언 방식이다.

```
var funcExpression = function () {
  return 'A function expression';
}
funcExpression();
```

## 함수 선언식과 함수 표현식의 차이점

함수 선언식은 호이스팅에 영향을 받지만, 함수 표현식은 호이스팅에 영향을 받지 않는다.

함수 선언식은 코드를 구현한 위치와 관계없이 자바스크립트의 특징인 호이스팅에 따라 브라우저가 자바스크립트를 해석할 때 맨 위로 끌어 올려진다.

예를 들어 아래 코드가 실행될 때

```
logMessage();
sumNumbers();

function logMessage() {
  return 'worked';
}

var sumNumbers = function () {
  return 10 + 20;
};
```

호이스팅에 의해 자바스크립트 해석기는 코드를 아래와 같이 인식한다.

```
function logMessage() {
  return 'worked';
}

var sumNumbers;

logMessage(); // 함수 선언식 결과는 'worked'
sumNumbers(); // 함수 표현식 결과는 Uncaught TypeError: sumNumbers is not a function

sumNumbers = function () {
  return 10 + 20;
};
```

함수 표현식 sumNumbers에서 var는 호이스팅이 적용되어 위치가 상단으로 끌어올려지만 실제 sumNumbers에 할당될 function 로직은 호출된 이후에 선언되므로, sumNumbers를 함수로 인식하지 않고 변수로 인식한다.

# Arguments

```
function sum(){
    var i, _sum = 0;
    for(i = 0; i < arguments.length; i++){
        document.write(i+' : '+arguments[i]+'<br />');
        _sum += arguments[i];
    }
    return _sum;
}
document.write('result : ' + sum(1,2,3,4));
```

위 예제의 결과는 10이다. 함수 sum은 인자로 전달된 값을 모두 더해서 리턴하는 함수다. 그런데 1행처럼 함수 sum은 인자에 대한 정의가 없다. 하지만 마지막 라인에서는 4개의 인자를 함수 sum으로 전달하고 있다. 함수의 정의부분에서 인자에 대한 구현이 없음에도 인자를 전달 할 수 있는 것은 왜 그럴까? 그것은 arguments라는 특수한 배열이 있기 때문이다.

arguments는 함수안에서 사용할 수 있도록 그 이름이나 특성이 약속되어 있는 일종의 배열이다. arguments[0]은 함수로 전달된 첫번째 인자를 알아낼 수 있다. 또 arguments.length를 이용해서 함수로 전달된 인자의 개수를 알아낼 수도 있다. 이러한 특성에 반복문을 결합하면 함수로 전달된 인자의 값을 순차적으로 가져올 수 있다. 그 값을 더해서 리턴하면 인자로 전달된 값에 대한 총합을 구하는 함수를 만들 수 있다.

## 매개변수의 수

매개변수와 관련된 두가지 수가 있다. 하나는 함수.length, 다른 하나는 arguments.length이다. arguments.length는 함수로 전달된 실제 인자의 수를 의미하고, 함수.length는 함수에 정의된 인자의 수를 의미한다.

```
function zero(){
    console.log(
        'zero.length', zero.length,
        'arguments', arguments.length
    );
}
function one(arg1){
    console.log(
        'one.length', one.length,
        'arguments', arguments.length
    );
}
function two(arg1, arg2){
    console.log(
        'two.length', two.length,
        'arguments', arguments.length
    );
}
zero(); // zero.length 0 arguments 0
one('val1', 'val2'); // one.length 1 arguments 2
two('val1'); // two.length 2 arguments 1
```

# 값으로서의 함수와 콜백

자바스크립트에서는 함수도 객체다. 다시 말해서 일종의 값이다. 거의 모든 언어가 함수를 가지고 있다. 자바스크립트의 함수가 다른 언어의 함수와 다른 점은 함수가 값이 될 수 있다는 점이다. 함수는 객체의 값으로 포함될 수 있고 객체의 속성 값으로 담겨진 함수를 메소드(method)라고 부른다.

## 값으로서의 함수

함수는 함수의 리턴 값으로도 사용할 수 있다.

```
function cal(mode){  
    var funcs = {  
        'plus' :function(left,right){return left + right},  
        'minus':function (left,right){return left - right}  
    }  
    return funcs [mode];  
}  
alert(cal('plus')(2,1));  
alert(cal('minus')(2,1));
```

함수는 배열의 값으로도 사용할 수 있다. 아래 예제의 결과는 60.5이다.

```
var process = [  
    function(input){return input + 10;},  
    function(input){return input * input;},  
    function(input){return input /2;}  
];  
var input = 1;  
for(var i = 0; i <process.length;i++){  
    input = process [i](input);  
}  
alert (input);
```

## 콜백

값으로 사용될 수 있는 특성을 이용하면 함수의 인자로 함수를 전달할 수 있다. 값으로 전달된 함수는 호출될 수 있기 때문에 이를 이용하면 함수의 동작을 완전히 바꿀 수 있다. 인자로 전달된 함수 sortNumber의 구현에 따라서 sort의 동작방법이 완전히 바뀌게 된다.

array.sort(sortfunc)는 원소들 간에 무엇이 우선인지를 판단하고 정렬된 배열을 리턴한다.

```
var numbers = [20,10,9,8,7,6,5,4,3,2,1];  
console.log(numbers.sort(sortNumber));//array,[1,10,2,20,3,4,5,6,7,8,9]
```

아래 예제의 방법을 사용해서 배열을 오름차순으로 정렬할 수 있다.

```
var numbers = [20,10,9,8,7,6,5,4,3,2,1];  
var sortfunc = function(a,b){  
    console.log(a,b);  
    if(a > b){  
        return 1;  
    } else if(a < b){  
        return -1;  
    } else{  
        return 0;  
    }  
}  
console.log(numbers.sort(sortfunc));//결과는 [1,2,3,4,5,6,7,8,9,10,20]
```

console.log(numbers.sort(sortfunc))의 sortfunc는 콜백 함수다. 콜백 함수를 sort의 인자로 전달함으로서 sort 함수의 기본적인 동작방법을 변경된다. 함수는 값으로서 사용할 수 있기 때문에 오리지널 함수의 동작방법을 바꿀 수 있다.

# 클로저

클로저는 내부함수가 외부함수의 맥락에 접근할 수 있는 것을 가르킨다.

## 내부함수와 외부함수의 관계

자바스크립트는 함수 안에서 또 다른 함수를 선언할 수 있다.

```
function outer(){//외부함수
    var title = 'coding everybody';//외부함수에 정의된 지역변수
    function inner(){//내부함수
        alert(title);
    }
    inner();
}
outer();
```

결과는 coding everybody이다. 위의 예제는 내부함수 inner에서 title을 호출했을 때 외부함수인 outer의 지역변수에 접근할 수 있음을 보여준다. 이러한 것을 클로저라고 한다.

## 클로저란?

클로저는 내부함수와 밀접한 관계를 가지고 있는 주제다. 내부함수는 외부함수의 지역변수에 접근할 수 있는데 외부함수의 실행이 끝나서 외부함수가 소멸된 이후에도 내부함수가 외부함수의 변수에 접근할 수 있다. 이러한 매커니즘을 클로저라고 한다

```
function outer(){
    var title = 'coding everybody';
    function inner(){
        alert(title);
    }
}
var inner = outer();
inner();
```

inner 안에 담겨있는 outer함수를 호출하면 coding everybody가 출력된다.

outer함수는 내부함수를 리턴함으로서 종료된다. 그럼에도 불구하고 내부함수가 외부함수의 변수에 접근할 수 있다는 것은 클로저의 중요한 특징이다. 클로저는 내부함수가 외부함수에 접근할 수 있고 외부함수는 외부함수의 지역변수를 사용하는 내부함수가 소멸될 때까지 소멸되지 않는 특성을 의미한다.

# JavaScript Execution context

## 1. 실행 문맥

실행 문맥은 scope, hoisting, this, function, closure 등의 동작원리를 담고 있는 자바스크립트의 핵심원리이다. 실행문맥을 이해하지 못하면 코드 이해가 어려워지며 디버깅도 매우 곤란해진다. ECMAScript 스펙에 따르면 실행 문맥을 실행 가능한 코드를 형상화하고 구분하는 추상적인 개념이라고 정의한다. 쉽게 말하면 실행 문맥은 실행 가능한 코드가 실행되는 환경이라고 말할 수 있다. 실행 가능한 코드란 다음과 같다.

- 전역 코드: 전역 영역에 존재하는 코드
- Eval 코드: Eval 함수로 실행되는 코드 (eval()은 매개변수에 전달된 문자열 구문 또는 표현식을 평가 또는 실행한다.)
- 함수 코드: 함수 내에 존재하는 코드

일반적으로 실행 가능한 코드는 전역 코드와 함수 내 코드이다.

자바스크립트 엔진은 코드를 실행하기 위하여 실행에 필요한 여러 가지 정보를 알고 있어야 한다. 실행에 필요한 여러 가지 정보란 다음과 같다.

- 변수: 전역변수, 지역변수, 매개변수, 객체의 프로퍼티
- 함수선언
- 함수의 유효범위(scope)
- this

이와 같이 실행에 필요한 정보를 형상화하고 구분하기 위해 자바스크립트 엔진은 실행 문맥을 물리적 객체의 형태로 관리한다.

```
var x = 'xxx';

function foo(){
    var y = 'yyy';
    function bar(){
        var z = 'zzz';
        console.log(x + y + z);
    }
    bar();
}
foo();
```

위 코드가 실행되면 실행 문맥스택(stack)이 생성하고 소멸한다. 현재 실행중인 문맥에서 이 문맥과 관련없는 코드(예를 들어 다른 함수)가 실행되면 새로운 문맥이 생성된다. 이 문맥은 스택에 쌓이게되고 컨트롤은 이동한다.

1. 컨트롤이 실행 가능한 코드로 이동하면 논리적 스택구조를 가지는 새로운 실행문맥 스택이 생성된다. 스택은 LIFO (Last In First Out, 후입선출)의 구조를 가지는 나열 구조이다.
2. 전역 코드로 컨트롤이 진입하면 전역 실행 문맥이 생성되고 실행 문맥 스택에 쌓인다. 전역 실행 문맥은 애플리케이션이 종료될 때(웹 페이지에서 나가거나 브라우저를 닫을 때)까지 유지된다.
3. 함수를 호출하면 해당 함수의 실행 문맥이 생성되며 직전에 실행된 코드블럭의 실행 문맥 위에 쌓인다.
4. 함수 실행이 끝나면 해당 함수의 실행 문맥을 파기하고 직전의 실행 문맥에 컨트롤을 반환한다.

## 2. 실행 문맥의 3가지 객체

실행 문맥은 실행 가능한 코드를 형상화하고 구분하는 추상적인 개념이지만 물리적으로는 객체의 형태를 가지며 아래의 3가지 프로퍼티를 소유한다.

- Variable object: {vars, function declarations, arguments...}
- Scope chain: [Variable object + all parent scopes]
- thisValue: Context object

## 2-1.Variable Object(변수 객체)

실행 문맥이 생성되면 자바스크립트 엔진은 실행에 필요한 여러 정보들을 담을 객체를 생성한다. 이를 Variable Object(변수 객체)라고 한다.

변수 객체는 아래의 정보를 담는 객체이다.

변수

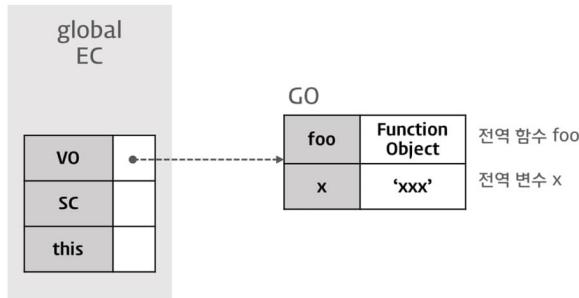
매개변수(parameter)와 인수 정보(arguments)

함수 선언(함수 표현식은 제외)

변수 객체는 실행 문맥의 프로퍼티이기 때문에 값을 갖는데 이 값은 다른 객체를 가리킨다. 그런데 전역 코드 실행시 생성되는 전역 문맥의 경우와 함수를 실행할 때 생성되는 함수 문맥의 경우 가르키는 객체가 다르다. 이는 전역 코드와 함수 코드의 내용이 다르기 때문이다. 예를 들어 전역 코드에는 매개변수가 없지만 함수에는 매개변수가 있다.

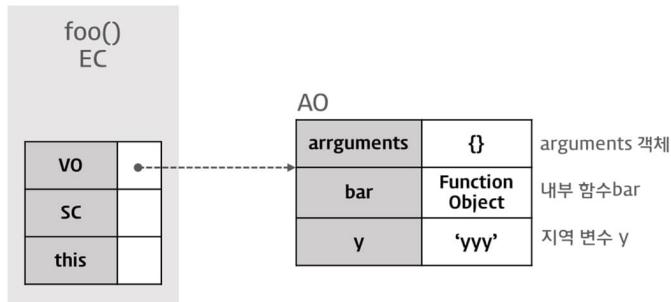
변수 객체가 가리키는 객체는 아래와 같다.

전역 문맥의 경우



변수 객체는 유일하며 최상위에 위치하고 모든 전역 변수, 전역 함수 등을 포함하는 전역 객체를 가리킨다. 전역에 선언된 전역 변수와 전역 함수를 프로퍼티로 소유한다.

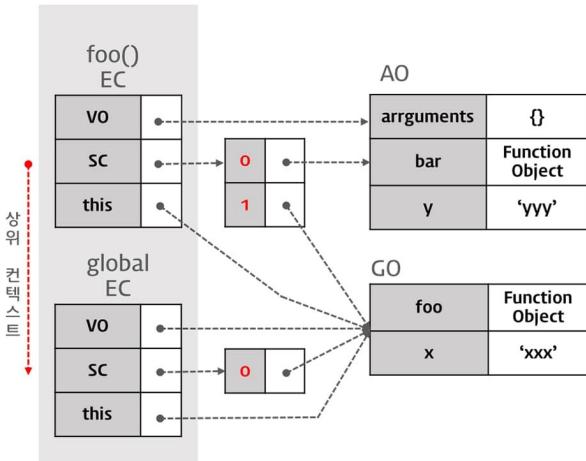
함수 문맥의 경우



변수 객체는 Activation Object(활성 객체)를 가리키며 매개변수와 인수들의 정보를 배열의 형태로 담고 있는 arguments object가 추가된다.

## 2-2.Scope Chain

스코프 체인은 일종의 리스트로서 중첩된 함수의 스코프의 레퍼런스를 차례로 저장하고 있는 개념이다. 즉 스코프 체인은 현재 실행 문맥의 활성 객체를 선두로하여 순차적으로 상위 문맥의 활성 객체를 가리키며 마지막 리스트는 전역 객체를 가리킨다.



엔진은 이를 통해 변수의 스코프를 파악한다. 함수가 중첩 상태일 때 하위 함수 내에서 상위 함수의 유효범위(scope)까지 참조할 수 있는데 이것은 스코프 체인을 검색하였기 때문이다. 함수가 중첩되어 있으면 중첩될 때마다 부모 함수의 스코프가 자식 함수의 스코프 체인에 포함된다. 함수 실행중에 변수를 만나면 그 변수를 우선 현재 스코프 즉 활성 객체에서 검색해보고 만약 검색에 실패하면 스코프 체인에 담겨진 순서대로 그 검색을 이어가게 되는 것이다. 이것이 스코프 체인이라고 불리는 이유다.

예를 들어 함수 내의 코드에서 변수를 참조하면 엔진은 스코프 체인의 첫번째 리스트가 가리키는 활성 객체에 접근하여 변수를 검색하다. 만일 검색에 실패하면 다음 리스트가 가리키는 활성 객체 또는 전역 객체를 검색한다. 이와 같이 순차적으로 스코프 체인에서 변수를 검색하는데 결국 검색에 실패하면 정의되지 않은 변수에 접근하는 것으로 판단하여 reference 에러를 발생시킨다.

## 2-3.this value

`this` 프로퍼티에는 `this` 값이 할당된다. `this`에 할당되는 값은 함수 호출 패턴에 의해 결정된다.

### 3. 실행 문맥의 생성 과정

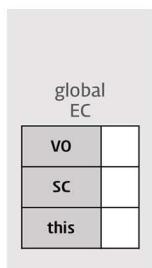
앞에서 살펴본 아래의 코드를 가지고 실제로 어떻게 실행 문맥이 생성되는지 알아보자.

```
var x = 'xxx';

function foo(){
    var y = 'yyy';
    function bar(){
        var z = 'zzz';
        console.log(x + y + z);
    }
    bar();
}
foo();
```

#### 3-1. 전역 코드에의 진입

컨트롤이 실행 문맥에 진입하기 이전에 유일한 전역 객체가 생성된다. 전역 객체는 단일 사본으로 존재하며 이 객체의 프로퍼티는 코드의 어떤 곳에서도 접근할 수 있다. 코드가 종료되면 전역 객체의 라이프 사이클은 종료한다. 초기 상태의 전역 객체에는 빌트인 객체(Math, String, Array 등)와 BOM, DOM이 설정되어 있다. 전역 객체가 생성된 이후 전역 코드로 컨트롤이 진입하면 전역 실행 문맥이 생성되고 실행 문맥 스택에 쌓인다.



실행컨텍스트 스택

그리구 이 실행 문맥을 바탕으로 아래의 처리가 실행된다.

1. 스코프 체인의 생성과 초기화
2. Variable Instantiation(변수 객체화) 실행
3. this value 결정

#### 3-1-1. 스코프 체인의 생성과 초기화

실행 문맥이 생성된 이후 가장 먼저 스코프 체인의 생성과 초기화가 실행된다. 이 때 스코프 체인은 전역 객체의 레퍼런스를 포함하는 리스트가 된다.

#### 3-1-2. Variable Instantiation(변수 객체화) 실행

스코프 체인의 생성과 초기화가 종료하면 변수 객체화가 실행된다.

변수 객체화는 변수 객체에 프로퍼티와 값을 추가하는 것을 의미한다. 이는 매개변수와 arguments, 함수 선언을 변수 객체에 추가하여 객체화하기 때문이다. 전역 코드의 경우 변수 객체화는 전역 객체를 가리킨다.

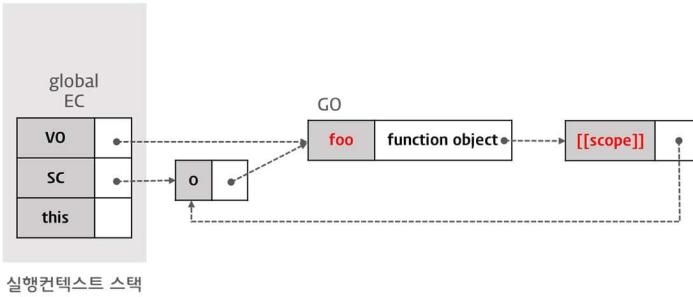
변수 객체화는 아래의 순서로 변수 객체에 프로퍼티와 값을 정의한다.

1. 함수 코드인 경우 매개변수가 변수 객체의 프로퍼티로, arguments가 값으로 설정된다.
2. 대상 코드 내의 함수 선언(함수 표현식 제외)을 대상으로 함수명이 변수 객체의 프로퍼티로, 생성된 함수 객체가 값으로 값으로 설정된다.(함수 호이스팅)
3. 대상 코드 내의 변수 선언을 대상으로 변수명이 변수 객체의 프로퍼티로, undefined가 값으로 설정된다.(변수 호이스팅)

위 예제 코드를 보면 전역 코드의 변수 x와 함께 함수 foo(매개변수 없음)가 선언되었다. 변수 객체화의 실행 순서 상 우선 2. 함수 foo의 선언이 처리되고(함수 코드가 아닌 전역 코드이기 때문에 1의 매개변수 처리는 실행되지 않는다.) 그 후 변수 x의 선언이 처리된다.

### 3-1-2-1. 함수 foo의 처리

함수 선언은 변수 객체화 실행 순서 2와 같이 선언된 함수명 foo가 변수 객체의 프로퍼티로, 생성된 함수 객체가 값으로 설정된다.



생성된 함수 객체는 [[scope]] 프로퍼티를 가지게 된다. [[scope]] 프로퍼티는 현재 실행 문맥의 스코프 체인이 참조하고 있는 객체를 값으로 설정한다.

```
> function foo () {}
<- undefined
> console.dir(foo);
▼ ⓘ function foo()
  arguments: null
  caller: null
  length: 0
  name: "foo"
  ► prototype: Object
  ► __proto__: function ()
    [[FunctionLocation]]: VM627:1
  ▼ [[Scopes]]: Scopes[1]
    ► 0: Global
```

지금까지 살펴본 실행 문맥은 아직 코드가 실행되기 이전이다. 하지만 스코프 체인이 가리키는 변수 객체에 이미 함수가 등록되어 있으므로 이후 코드를 실행할 때 함수 선언식 이전에 함수를 호출할 수 있게 되었다. 이러한 현상을 **함수 호이스팅**이라고 한다.

### 3-2-2-2. 변수 x의 선언 처리

변수 선언은 변수 객체화 실행 순서 3과 같이 선언된 변수명(x)이 변수 객체의 프로퍼티로 undefined가 값으로 설정된다.

**선언 단계:** 변수 객체에 변수를 등록한다. 이 변수 객체는 스코프가 참조할 수 있는 대상이 된다.

**초기화 단계:** 변수 객체에 등록된 변수를 메모리에 할당한다. 이 단계에서 변수는 undefined로 초기화된다.

**할당 단계:** undefined로 초기화된 변수에 실제 값을 할당한다.

var 키워드로 선언된 변수는 선언 단계와 초기화 단계가 한번에 이루어진다. 다시 말해 스코프 체인이 가리키는 변수 객체에 변수가 등록되고 변수는 undefined로 초기화 된다. 따라서 변수 선언문 이전에 변수에 접근하여도 변수 객체에 변수가 존재하기 때문에 에러가 발생하지 않는다. 다만 undefined를 반환한다. 이러한 현상을 **함수 호이스팅**이라 한다.

아직 변수 x는 'xxx'로 초기화되지 않았다. 이후 변수 할당 단계에 도달하면 비로소 값의 할당이 이루어진다.

### 3-1-3. this value 결정

변수 선언 처리가 끝나면 다음은 this value가 결정된다. this value가 결정되기 이전에 this는 전역 객체를 가리키고 있다가 함수 호출 패턴에 의해 this에 할당 되는 값이 결정된다. 전역 코드의 경우, this는 전역 객체를 가리킨다.

## 3-2.전역 코드의 실행

```
var x = 'xxx';

function foo(){
    var y = 'yyy';
    function bar(){
        var z = 'zzz';
        console.log(x + y + z);
    }
    bar();
}
foo();
```

위 예제에서 전역 변수 x에 문자열 'xxx' 할당과 함수 foo의 호출이 실행된다.

### 3-2-1.변수 값의 할당

전역 변수 x에 문자열 'xxx'를 할당할 때 현재 실행 문맥의 스코프 체인이 참조하고 있는 변수 객체를 선두부터 검색하여 변수명에 해당하는 프로퍼티가 발견되면 값('xxx')을 할당한다.

### 3-2-2.함수 foo의 실행

전역 코드의 함수 foo가 실행되기 시작하면 새로운 함수 문맥이 생성된다. 함수 foo의 실행 문맥으로 컨트롤이 이동하면 전역 코드와 마찬가지로 1.스코프 체인의 생성과 초기화 2.변수 객체화 실행 3.this value 결정이 순차적으로 실행된다. 단, 전역 코드와 다른 점은 이번 실행되는 코드는 함수 코드라는 것이다. 따라서 1.스코프 체인의 생성과 초기화 2.변수 객체화 실행 3.this value 결정은 전역 코드의 룰이 아닌 함수 코드의 룰이 적용된다.

#### 3-2-2-1.스코프 코드의 생성과 초기화

그 후 전역 문맥의 [[scope]] 프로퍼티가 참조하고 있는 객체가 스코프 체인에 push된다. 따라서 이 경우 함수 foo를 실행한 직후 실행 문맥의 스코프 체인은 활성 객체(함수 foo의 실행으로 만들어진)과 전역 객체를 순차적으로 참조하게 된다.

#### 3-2-2-2.변수 객체화 실행

함수 코드의 경우 스코프 체인의 생성과 초기화에서 생성된 함수 객체를 변수 객체로서 변수 객체화가 실행된다. 이것을 제외하면 전역 코드의 경우와 같은 처리가 실행된다. 즉, 함수 객체를 변수 객체에 바인딩한다.(프로퍼티는 bar, 값은 새로 생성된 함수 객체. bar함수 객체의 [[scope]] 프로퍼티 값은 활성객체와 전역 객체를 참조하는 리스트) 이 때 변수 y의 변수 객체에 설정한다. y는 프로퍼티 값은 undefined이다.

#### 3-2-2-3.this value 결정

변수 선언 처리가 끝나면 다음은 this value가 결정된다. this에 할당되는 값은 함수 호출 패턴에 의해 결정된다. 내부 함수의 경우, this의 value는 전역 객체이다

### 3-3.foo 함수 코드의 실행

```
var x = 'xxx';

function foo(){
    var y = 'yyy';
    function bar(){
        var z = 'zzz';
        console.log(x + y + z);
    }
    bar();
}
foo();
```

이제 함수 foo의 코드 블럭 내 구문이 실행된다. 위 예제를 보면 변수 y에 문자열 'yyy'의 할당과 함수 bar가 실행된다.

#### 3-3-1.변수 값의 할당

지역 변수 y에 문자열 'yyy'를 할당할 때 현재 실행 컨텍스트의 스코프 체인이 참조하고 있는 변수 객체를 선두부터 검색하여 변수명에 해당하는 프로퍼티가 발견되면 값 'yyy'를 할당한다.

#### 3-3-2.함수 bar의 실행

함수 bar가 실행되기 시작하면 새로운 실행 문맥이 생성된다. 이전 함수 foo의 실행 과정과 동일하게 1.스코프 체인의 생성과 초기화 2.변수 객체와 실행 3.this value 결정이 순차적으로 실행된다. 이 단계에서 console.log(x + y + z); 구문의 실행 결과는 xxxyyyzzz가 된다.

# JavaScript Event

## 1.이벤트

웹 페이지에서는 어떤 종류의 상호작용이 발생할 때 이벤트가 일어난다. 사용자가 무언가를 클릭하거나 특정 요소 위로 마우스를 가져가거나 특정한 키를 누르는 것 등이 이러한 상호작용에 포함된다. 이벤트는 또한 브라우저에 의해서도 일어날 수 있는데 웹 페이지의 로딩이 끝났거나 사용자가 페이지를 스크롤하거나 브라우저 창의 크기를 조절하는 것 등이 포함된다. 자바스크립트의 사용 전체에 걸쳐서 당신은 어떤 특정 이벤트가 일어나는 것을 감지할 수 있으며 그러한 이벤트들에 대응해서 어떤 일어나게 될지 조합시킬 수 있다.

## 2.이벤트가 움직이는 방법

웹 페이지 내의 HTML 요소에서 이벤트가 일어나게 되면 그 요소는 이벤트에 대해 어떤 이벤트 핸들러가 할당되어 있는지 찾아보게 된다. 할당된 이벤트 핸들러가 있다면 적합한 방법으로 그것을 호출하게 되는데 참조값과 일어난 이벤트에 대한 추가적인 정보들을 함께 전송한다. 그러면 이벤트 핸들러가 동작하게 된다. 이벤트 호출에는 두 가지 타입이 있다. 캡춰링(capturing)과 버블링(bubbling)이 그것이다.

### 2-1.캡춰링(capturing)

이벤트 캡춰링은 DOM 트리의 가장 바깥 요소에서부터 시작하여 이벤트가 일어난 요소에 도착할 때 까지 안쪽으로 찾아 들어가고 다시 바깥으로 나온다. 예를 들어 웹 페이지에서 무언가를 클릭하면 처음에는 HTML 요소에서 onclick 이벤트 핸들러를 찾고 다음에는 body 요소에서 찾고 다음에는 다음에는 하는 식으로 이벤트가 일어난 요소에 도착할 때까지 반복한다.

### 2-2.버블링(bubbling)

이벤트 버블링은 정확히 반대의 방법으로 동작한다 이벤트가 일어난 요소에서부터 체크-이 요소에 무슨 이벤트 핸들러가 할당되어 있는지-를 시작해서 그 부모 요소로 그 부모 요소로 하는 식으로 HTML 요소까지 거슬러 올라간다.

## 3.이벤트의 혁명

자바스크립트의 초기에 우리들은 HTML 요소의 내부에서 직접적으로 이벤트 핸들러를 사용했다. 아래 예제를 보자.

```
<a href="http://www.naver.com" onclick="alert('hello')">HELLO</a>
```

이러한 접근 방법의 문제점은 이벤트 핸들러가 HTML 코드 전체에 섞여있는 것이다. 통제는 불가능에 가까우며 외부 자바스크립트 파일을 include해 올 때는 웹 브라우저의 캐시 기능을 전혀 사용할 수 없게 된다.

-캐시란? 컴퓨터의 중앙처리장치 (CPU)는 메모리부터 데이터를 읽거나 쓰는 기능을 한다. 그런데 CPU의 처리 속도가 훨씬 빠르기 때문에 캐시 메모리를 두어 자주 읽거나 쓰는 데이터를 저장해둔다. 이렇게 하면 CPU의 작업 속도가 훨씬 빨라진다. 인터넷 웹 브라우저에서도 캐시를 설정, 웹 사이트에서 읽은 데이터를 하드디스크에 저장해 두었다가 같은 내용을 읽을 때는 하드디스크에서 읽도록 한다. 속도가 다른 두장치 사이에 두어 데이터 전송 속도를 빠르게 하기 위한 것이 캐시다. 캐시란 빠른 작업 속도를 위해 존재하는 것이다.

## 4.이벤트와 접근성

이벤트를 어떻게 호출하고 제어할지 더 깊이 파고들기 전에 접근성을 알아야 한다. 대부분의 사람들에게 접근성은 다소 넓은 주제이지만 여기에서 말하는 접근성이란 자바스크립트가 꺼져 있거나 웹 서버에서 차단당하는 경우 등에도 이벤트는 반드시 동작해야 한다는 의미이다.

브라우저에서 자바스크립트를 꺼두는 사람들도 있긴 하지만 대부분은 프락시 서버나 방화벽 혹은 안티바이러스 프로그램들이 자바스크립트를 프로그래머의 의도대로 동작하지 못하게 막고 있다. 그렇기 때문에 우리는 자바스크립트가 불가능한 상황에서도 사용자가 여전히 사이트를 이용할 수 있게끔 에러 처리 fallback 옵션을 제공할 수 있어야 한다. 일반적으로 특정 이벤트에 대해서 내장된 행동 방식을 갖고 있지 않은 HTML 요소들에 이벤트를 할당하지 말아야 한다. 즉, onclick 이벤트는 a와 같은 요소에만 사용해야 한다. a 요소는 이미 클릭 이벤트에 대해 에러처리 방법(링크에 정의된 곳으로 이동한다거나, 입력야익을 제출한다거나)을 갖고 있기 때문이다.

## 5.이벤트 제어

간결함을 위해서 위에 제시했던 addEvent 함수를 사용해서 매 예제마다 크로스 브라우징의 복잡함 속으로 파고드는 일을 피해야 한다.

예를 들어 onload 이벤트를 사용해보자. onload 이벤트는 window 객체에 종속된다. 일반적으로 브라우저 윈도우에 영향을 미치는 모든 이벤트(ex onload, onresize, onscroll 등)는 window 객체를 통해 제어할 수 있다.

onload 이벤트는 웹 페이지에 포함된 모든 것들이 완전히 로드되었을 때 발생한다. 모든 것이라 하는 것은 HTML 코드 자신과 함께 이미지, CSS파일, 자바스크립트 파일과 같은 외부 리소스들을 말한다. 그런 것들이 전부 로드되었을 때 window.onload 이벤트가 발생하며 이 이벤트에 대응해서 웹 페이지의 특정 기능이 실행되도록 할 수 있다.

### 5-1.특정한 요소에 이벤트 적용하기

페이지에 있는 다른 요소들에 이벤트를 할당하는 법을 알아보자. 링크를 클릭할 때마다 특정한 이벤트가 발생하길 원한다면 다음과 같이 하면 된다.

```
addEvent(window, 'load',function(){
    var links = document.getElementsByTagName("a");
    for (var i = 0; i<links.length; i++){
        addEvent(link[i],"click",function(){
            alert("hello");

            evt.preventDefault();
        })
    }
})
```

첫번째로 일어난 일은 onload이벤트를 사용해서 페이지가 완전히 로딩된 시점을 알아냈다. 다음으로는 document 객체의 getElementByTagName 메소드를 이용해서 페이지 내에 있는 모든 링크를 찾아냈다. 참조 관계가 만드렁지고 모든 링크들에 대해 루프를 돌려서 이벤트를 할당했으므로 링크가 클릭될 때마다 액션이 일어난다. preventDefault()는 return false, 즉 기본 동작(url이동)이 실행되지 않도록 막는다.