JavaScript

유성민

http://makestory.net

2019

정리자료

개요

JavaScript

핵심개념

데이터 타입

객체

함수

생성자 함수

this

실행 컨텍스트

기타 (argument, 스코프체인, event, jQuery, …)

[개념적인 부분 중심 설명]

목표

JavaScript

자바스크립트 언어의 **주요 특성** 이해

문헌 (서적, 문서)

ECMAScript 5, 2009년 기준 발표자료

JavaScript

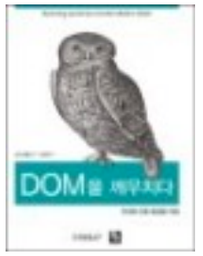
내용의 출처/근거 없는 서적, 자료 주의

근거자료, 표현명시(번역본 용어, 저자 개인생각 구분)

브라우저 표현방식 [[Prototype]], __proto__ 등

ECMA 표준 문서의 버전별 수정, 추가된 내용

빠르게 변화하고 있는 진행형 언어

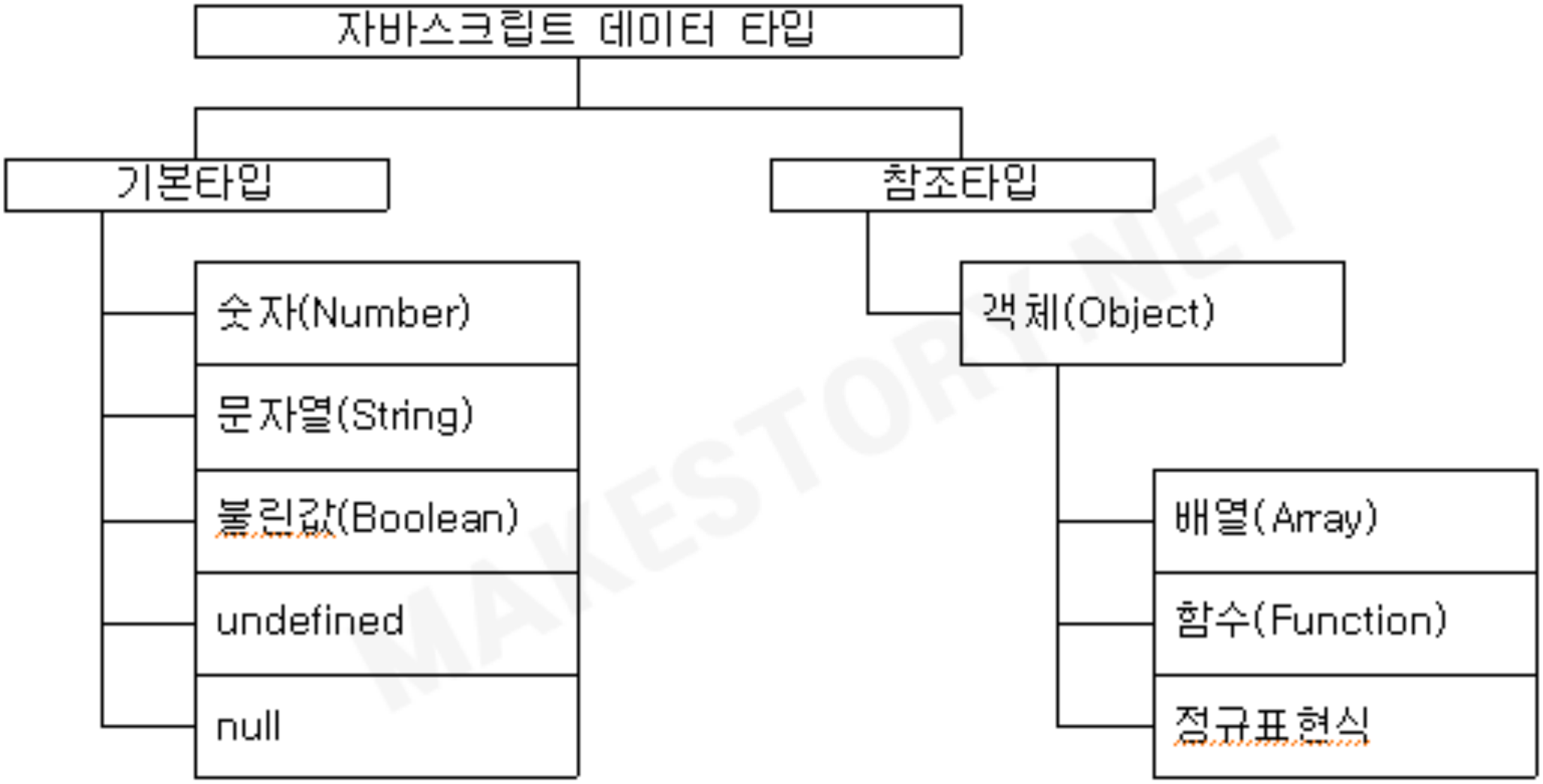


내용의 근거(출처), 기준(버전), 용어(번역) 중점을 두고 발표자료 선택

[참고] MDN 웹기술 공식문서화 관련내용 http://www.yonhapnews.co.kr/bulletin/2018/07/26/0200000000AKR20180726083100030.HTML

데이터 타입

JavaScript



[참고] ES6 기본타입으로 심볼 추가됨

데이터 타입

JavaScript

- **기본타입**(primitive, 원형, 원시값, 단순값 등의 용어로 표현됨)은 **그 자체가 하나의 값**을 나타냄

- **기본타입을 제외한 모든 값은 객체**(복합객체, 합성객체 등의 용어로 표현됨)

- **기본타입은 하나의 값만을 가지는 데 비해, 참조 타입인 객체는 여러 개의 프로퍼티들을 포함**할 수 있으며, 이러한 **객체의 프로퍼티는 기본 타입의 값을 포함하거나, 다른 객체를 가리킬 수 있다.**

기본타입의 데이터는 그 데이터 원본을 내가 가지고 있는 것이고,
참조타입의 데이터는 그 데이터가 있는 주소(위치)정보를 내가 가지고 있는 것.

값/참조에 의한 함수 호출 방식

JavaScript

- **기본타입의 경우**에는 **값에 의한 호출(Call By Value)방식**으로 동작한다.
즉, 함수를 호출할 때 인자로 기본 타입의 값을 넘길 경우, 호출된 함수의 **매개변수로 복사된 값이 전달**

- **참조타입의 경우** 함수를 호출할 때 **참조에 의한 호출(Call By Reference)방식**으로 동작
함수를 호출할 때 인자로 참조 타입인 객체를 전달할 경우,
객체의 프로퍼티값이 함수의 **매개변수로 복사되지 않고,**
인자로 넘긴 객체의 참조값이 그대로 함수 내부로 전달

예를 들어, 기본타입은 데이터를 전달하는 방법은 복사하여 전달,
참조타입은 그 데이터의 주소(위치)를 알려주는 방식

프로토타입

JavaScript

- 자바스크립트의 **모든 객체는 자신의 부모 역할을 하는 객체와 연결되어 있음**
이러한 부모 객체를 **프로토타입 객체** (짧게는 **프로토타입**)

- ECMAScript 명세서에는 자바스크립트의 모든 객체는 자신의 프로토타입을 가리키는 **[[Prototype]] 라는 숨겨진 프로퍼티**를 가진다고 설명

- **크롬 브라우저에서는 __proto__가 바로 이 숨겨진 [[Prototype]] 프로퍼티를 의미**

객체에는__proto__라는 것으로 무언가(상위객체, 부모객체)와 연결되어 있다.

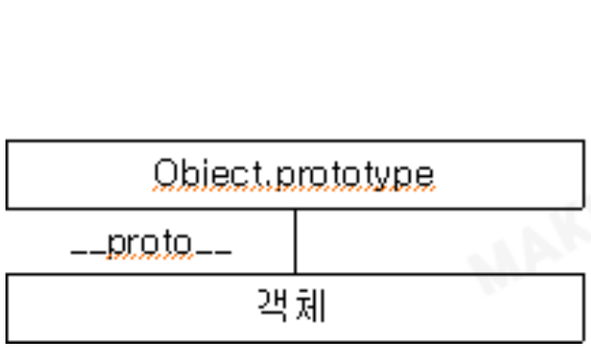
[참고] ECMAScript 명세서 8.6.2절 Object Internal Properties and Methods, Table8
[참고] ECMAScript 명세서 15.2.4절 Properties of the Object Prototype Object

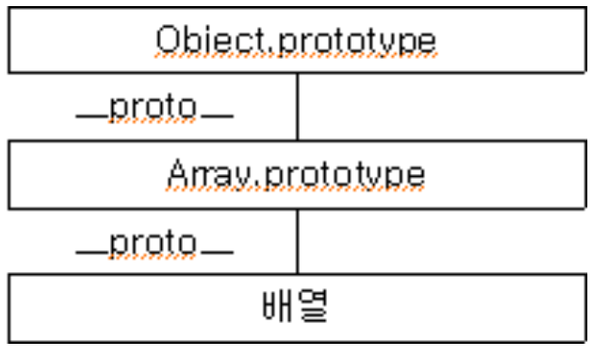
프로토타입 체이닝

JavaScript

자바스크립트에서 특정 **객체의 프로퍼티나 메서드에 접근하려고 할 때**,
해당 객체에 접근하려는 **프로퍼티 또는 메서드가 없다면**
[[Prototype]] 링크를 따라 자신의 부모 역할을 하는 **프로토타입 객체의 프로퍼티를 차례대로 검색**하는 것을
프로토타입 체이닝이라고 말한다.

즉, 객체의 특정 프로퍼티를 읽으려고 할 때, 프로퍼티가 해당 객체에 없는 경우 프로토타입 체이닝이 발생한다.





객체는 필요한 것이 나에게 없으면,
보이지 않게 __proto__ 연결된 통로로 도움을 청한다.

[참고] 자바스크립트에서 Object.prototype 객체는 프로토타입 체이닝의 종점

프로토타입 체이닝

JavaScript

```
var ysm = {name: '유성민'};
ysm.valueOf();
▶ {name: "유성민"}
```

ysm 객체생성
.valueOf() 메서드 호출
메소드가 정의되어 있지 않지만 정상적으로 결과 출력

```
console.dir(ysm);
▼ Object
    name: "유성민"
  ▼ __proto__:
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
```

ysm객체의 __proto__ 프로퍼티가 가리키는 객체는
Object.prototype 객체 (부모 객체에 포함된 다양한 메서드를 자신의 것처럼 사용)

객체 리터럴 방식으로 ysm 객체를 생성하고, 이 객체의 valueOf() 메서드를 호출

ysm 객체에는 valueOf() 메서드가 없으므로(정의하지 않았음) 에러가 발생해야 하지만, 정상적으로 결과가 출력

ysm 객체의 **프로토타입에 valueOf() 메서드가 이미 정의**되어 있고,

ysm 객체가 **상속처럼 valueOf() 메서드를 호출**

모든 객체의 프로토타입(부모 프로토타입 객체)은 자바스크립트 룰에 따라 객체를 생성할 때 결정된다.

함수

JavaScript

- **함수 선언문(함수 리터럴)** : 반드시 함수명이 정의되어 있어야 한다.

```
function add(x, y) {
   return x + y;
}
```

- **함수 표현식** : 함수도 하나의 값으로 취급된다.
이름이 없는 함수 형태를 자바스크립트에서는 익명 함수(anonymous function)라고 부른다.

```
var add = function() { // add 는 함수 이름이 아닌 변수명
   return x + y;
};
```

- 기명 함수 표현식 : 함수 이름이 포함된 함수 표현식을 기명 함수 표현식이라 한다.
함수 표현식에서 사용된 함수 이름은 외부 코드에서 접근 불가능

```
var add = function sum(x, y) {
   return x + y;
};
```

- Function 생성자 함수

```
var add = new Function('x', 'y', 'return x + y');
```

[참고] ES6 화살표함수

```
let test = () => {};
(() => { console.log('test'); })();
```

prototype 프로퍼티

JavaScript

- 일반 객체와는 다르게 추가로 **함수 객체만의 표준 프로퍼티가 정의**되어 있다.
ECMA5 스크립트 명세서에는 모든 함수가 **length와 prototype 프로퍼티**를 가져야 한다고 기술하고 있다.

- 모든 함수는 객체로서 prototype 프로퍼티를 가지고 있다. (함수 객체만의 표준프로퍼티 prototype)
prototype 프로퍼티는 함수가 생성될 때 만들어지며, 단지 constructor 프로퍼티 하나만 있는 객체를 가리킨다.
그리고 **prototype 프로퍼티가 가리키는 프로토타입 객체의 유일 constructor 프로퍼티는**
자신과 연결된 함수를 가리킨다.

즉, 자바스크립트에서는 함수를 생성할 때,
함수 자신과 연결된(constructor 프로퍼티) 프로토타입 객체를 동시에 생성
(new 키워드를 사용할 때 consturctor 이 부분을 참조, 생성자함수)

length와 prototype, 함수 객체만의 표준 프로퍼티

[참고] ECMA 15.3.3 Properties of the Function Constructor
[참고] length 프로퍼티는 인자의 개수를 나타낸다.

생성자 함수가 동작하는 방식

JavaScript

- new 연산자로 자바스크립트 함수를 생성자로 호출하면, 다음과 같은 순서로 동작한다.

1) **빈 객체 생성 및 this 바인딩**

생성자 함수 코드가 실행되기 전 빈 객체가 생성된다. 바로 이 객체가 생성자 함수가 새로 생성하는 객체이며,
이 객체는 this로 바인딩된다. 따라서 이후 생성자 함수의 코드 내부에서 사용된 this는 이 빈 객체를 가리킨다.
하지만 여기서 생성된 객체는 엄밀히 말하면 빈 객체는 아니다. 생성자 함수가 생성한 객체는
자신을 생성한 **생성자 함수의 prototype 프로퍼티**가 가리키는 객체를 **자신의 프로토타입 객체로 설정**한다.

2) **this를 통한 프로퍼티 생성**

이후에는 함수 코드 내부에서 this를 사용해서,
앞에서 생성된 빈 객체에 동적으로 프로퍼티나 메서드를 생성할 수 있다.

3) **생성된 객체 리턴**

리턴문이 동작하는 방식은 경우에 따라 다르므로 주의해야한다.
우선 가장 **일반적인 경우로 특별하게 리턴문이 없을 경우, this로 바인딩된 새로 생성한 객체가 리턴**된다.
이것은 명시적으로 this를 리턴해도 결과는 같다.
하지만 리턴 값이 새로 생성한 객체(this)가 아닌 다른 객체를 반환하는 경우는
생성자 함수를 호출했다고 하더라도 this가 아닌 해당 객체가 리턴된다.

생성자 함수가 동작하는 방식

JavaScript

```
function YSM() {
    this.name = '유성민';
}
YSM.prototype = {
    getName: function() {
        return this.name;
    }
};
var ysm = new YSM();
ysm.getName();
console.dir(ysm);
```

```
▼ YSM
    name: "유성민"
  ▼ __proto__:
    ▶ getName: f ()
    ▼ __proto__:
      ▶ constructor: f Object()
      ▶ hasOwnProperty: f hasOwnProperty()
      ▶ isPrototypeOf: f isPrototypeOf()
      ▶ propertyIsEnumerable: f propertyIsEnumerable()
      ▶ toLocaleString: f toLocaleString()
      ▶ toString: f toString()
      ▶ valueOf: f valueOf()
```

함수는 new 라는 키워드를 만났을 때 생성자 함수로 동작
this = {} **// 1) 빈객체 생성**
this.__proto__ = YSM.prototype **// 2) 자신의 prototype연결**
this.name = '유성민' **// 3) 동적 프로퍼티, 메소드 생성**
return this; **// 4) 생성된 객체 반환**

---

ysm객체는 생성자 함수, 즉 부모 객체로 연결된 __proto__ 링크를 가지고 있습니다.
YSM.prototype 객체가 ysm.__ptoto__ 링크로 연결되어 있음
ysm.__proto__.getName(); 형태로 접근하여 메소드(함수) 실행

new 키워드로 여러 객체를 생성해도,
__ptoto__ 객체를 통해 부모 생성자함수 prototype 객체와 연결되어 있다는 것 확인

var ysm1 = {}; // 객체리터럴 생성방식 ysm1 부모 객체는 Object
var ysm2 = new YSM(); // 생성자함수 객체 생성방식 ysm2 부모객체는 YSM

YSM.prototype = {} 역시 자바스크립트 객체이므로
Object.prototype 를 프로토타입 객체로 가진다. (부모역할 객체)
따라서 프로토로타입 체이닝은 Object.prototype 객체로 이어진다.

자바스크립트의 모든 객체는 / 자신의 부모인 프로토타입 객체를 가리키는 / 참조 링크 형태의 숨겨진 프로퍼티가 있다.

this 바인딩

JavaScript

- 객체의 프로퍼티가 함수일 경우, 이 함수를 메서드라고 부른다.

이러한 **메서드를 호출할 때, 메서드 내부 코드에 사용된** this는 해당 메서드를 호출한 객체로 바인딩 된다.

즉, this는 자신을 호출한 객체에 바인딩 (호출 시점에 그 값을 확인)

- 브라우저에서 자바스크립트를 실행하는 경우 **전역객체는 window 객체**가 된다.

자바스크립트의 **모든 전역 변수는 실제로는 전역 객체의 프로퍼티들**이다.

this 는 함수(메소드) 호출 시점에 그 값을 알 수 있다.

this 바인딩

JavaScript

```
deep.func();
deep.deep1.func();
deep.deep1.deep2.func();
```

```
var deep = {
    value: 0,
    func: function() {
        console.log('deep');
        console.log(this.value);
        console.log(this.func);
    },
    deep1: {
        value: 1,
        func: function() {
            console.log('deep1');
            console.log(this.value);
            console.log(this.func);
        },
        deep2: {
            value: 2,
            func: function() {
                console.log('deep2');
                console.log(this.value);
                console.log(this.func);
            }
        }
    }
};
```

this 바인딩

JavaScript

```
var value = 100;  // window.value
var func = function() {   // window.func
    console.log('전역위치 func 함수');
    console.log(this.value);
};
```

```
func();
window.func();
```

명시적인 this 바인딩

JavaScript

- 자바스크립트는 **this를 특정 객체에 명시적으로 바인딩 시키는 방법도 제공**

- call과 apply 메서드를 이용해 명시적인 this 바인딩 (bind 메소드 포함)
apply() 나 call() 메서드는 this를 원하는 값으로 명시적으로 매핑해서
특정 함수나 메서드를 호출할 수 있다는 장점

this 값은 호출시점 메소드(함수) 기준
앞에 위치한 암묵적인 값 또는 뒤에 명시적으로 호출되는 값

[참고] ES6 화살표함수의 this 값은 언제나 상위 스코프 this값을 가리킨다. 화살표함수도 apply, call 메소드 사용가능

명시적인 this 바인딩

JavaScript

```
deep.deep1.deep2.func.call(deep);
```

```
var deep = {
    value: 0,
    func: function() {
        console.log('deep');
        console.log(this.value);
        console.log(this.func);
    },
    deep1: {
        value: 1,
        func: function() {
            console.log('deep1');
            console.log(this.value);
            console.log(this.func);
        },
        deep2: {
            value: 2,
            func: function() {
                console.log('deep2');
                console.log(this.value);
                console.log(this.func);
            }
        }
    }
};
```

내부함수

JavaScript

- **함수 내부에 정의된 함수를 내부함수(inner function)**라고 부른다.

- **자바스크립트에서는 내부 함수 호출 패턴을 정의해 놓지 않았다.**

내부함수의 this는 전역 개체(window)에 바인딩 된다.

- **내부함수가 this를 참조하는 자바스크립트의 한계를 극복하려면**

부모함수의 this를 내부 함수가 접근 가능한 다른 변수에 저장하는 하는 방법이 사용된다.

(내부함수에서 스코프체이닝으로 접근)

내부함수에서의 this는 전역객체

[참고] ES6 화살표함수 제외

내부함수

JavaScript

```
var innerTest = {
    'deep': {
        func: function() {
            var that = this;

            // 내부함수
            var setInnerThis = function() {
                console.log('내부함수 this');
                console.log(this); // window
                console.log(that); // deep
                // 여기에 내부함수를 선언해도 그 함수 내부 this는 글로벌 ...
            };
            setInnerThis();
        }
    }
};

innerTest.deep.func();
```

실행 컨텍스트

JavaScript

- **자바스크립트가 실행될 때 생성되는 하나의 실행단위를 실행컨텍스트**라고 한다.

- ECMAScript 에서는 **실행 컨텍스트가 형성되는 경우를 세가지로 규정**하고 있다.
① 전역코드 ② eval() 함수로 실행되는 코드 ③ **함수 안의 코드를 실행할 경우**

- ECMAScript 에서는 실행컨텍스트를 “실행 가능한 코드를 형상화하고 구분하는 추상적인 개념”으로 기술한다.

- ECMAScript 에서는 실행 컨텍스트의 생성을 다음처럼 설명한다
“현재 실행되는 컨텍스트에서
이 컨텍스트와 관련 없는 실행 코드가 실행되면,
새로운 컨텍스트가 생성되어 스택에 들어가고,
제어권이 그 컨텍스트로 이동한다.”

하나의 실행단위가 실행되기 전에 하는 준비과정, 실행 컨텍스트

실행 컨텍스트 생성과정

JavaScript

1) **활성 객체 생성 (=변수객체)**

실행 컨텍스트가 생성되면 자바스크립트 엔진은

해당 컨텍스트에서 **실행에 필요한 여러가지 정보를 담을 객체를 생성**하는데, 이를 활성객체라고 한다.

이 객체에 앞으로 매개변수나 사용자가 정의한 변수 및 객체를 저장하고,

새로 만들어진 컨텍스트로 접근 가능하게 되어 있다.

이는 엔진 내부에서 접근할 수 있는 것이지 사용자가 접근할 수 있는 것은 아니다.

2) **arguments 객체 생성**

앞서 만들어진 활성 객체는 argument 프로퍼티로 이 arguments 객체를 참조한다.

(argument 객체는 **함수를 호출할 때 넘긴 인자들이 배열 형태로 저장된 객체**를 의미, 유사배열객체)

3) **스코프 정보 생성**

현재 컨텍스트의 유효 범위를 나타내는 스코프 정보를 생성한다.

(현재 실행 중인 실행 컨텍스트 안에서 **연결리스트와 유사한 형식**으로 만들어진다)

현재 컨텍스트에서 특정 변수에 접근해야 할 경우, 이 리스트를 활용한다.

이 리스트로 현재 컨텍스트의 변수 뿐 아니라, 상위 실행 컨텍스트의 변수도 접근이 가능하다.

이 리스트에서 찾지 못한 변수는 결국 정의되지 않은 변수에 접근하는 것으로 판단하여 에러를 검출한다.

이 리스트를 스코프체인 이라고 하는데, [[scope]] 프로퍼티로 참조된다.

실행 컨텍스트 생성과정

JavaScript

4) **변수생성 (=활성객체)**

현재 실행 컨텍스트 내부에서 사용되는 **지역변수**의 생성이 이루어 진다.

변수 객체 안에서 호출된 함수인자는 각각의 프로퍼티가 만들어지고, 그 값이 할당된다. (변수명: 값 형태)

만약 값이 넘겨지지 않았다면 undefined 가 할당된다.

여기서 **주의해야 할 점은, 이 과정에서는 변수나 내부 함수를 단지 메모리에 생성(instantiation)하고,**

초기화(initialization)는 각 변수나 함수에 해당하는 표현식이 실행되기 전까지는 이루어 지지 않는 다는 점이다.

표현식의 실행은 변수 객체 생성이 다 이루어진 후 시작된다.

(초기화 전이라 **각 변수에는 undefined가 할당**됨, 함수 선언문과 함수 표현식의 차이가 왜 발생하는지 이해)

5) **this 바인딩**

마지막 단계에서는 **this 키워드를 사용하는 값이 할당**된다.

여기서 **this가 참조하는 객체가 없으면 전역 객체를 참조**한다.

6) **코드실행**

이렇게 **하나의 실행 컨텍스트가 생성되고, 변수 객체가 만들어진 후**에,

코드에 있는 여러 가지 표현식 실행이 이루어진다.

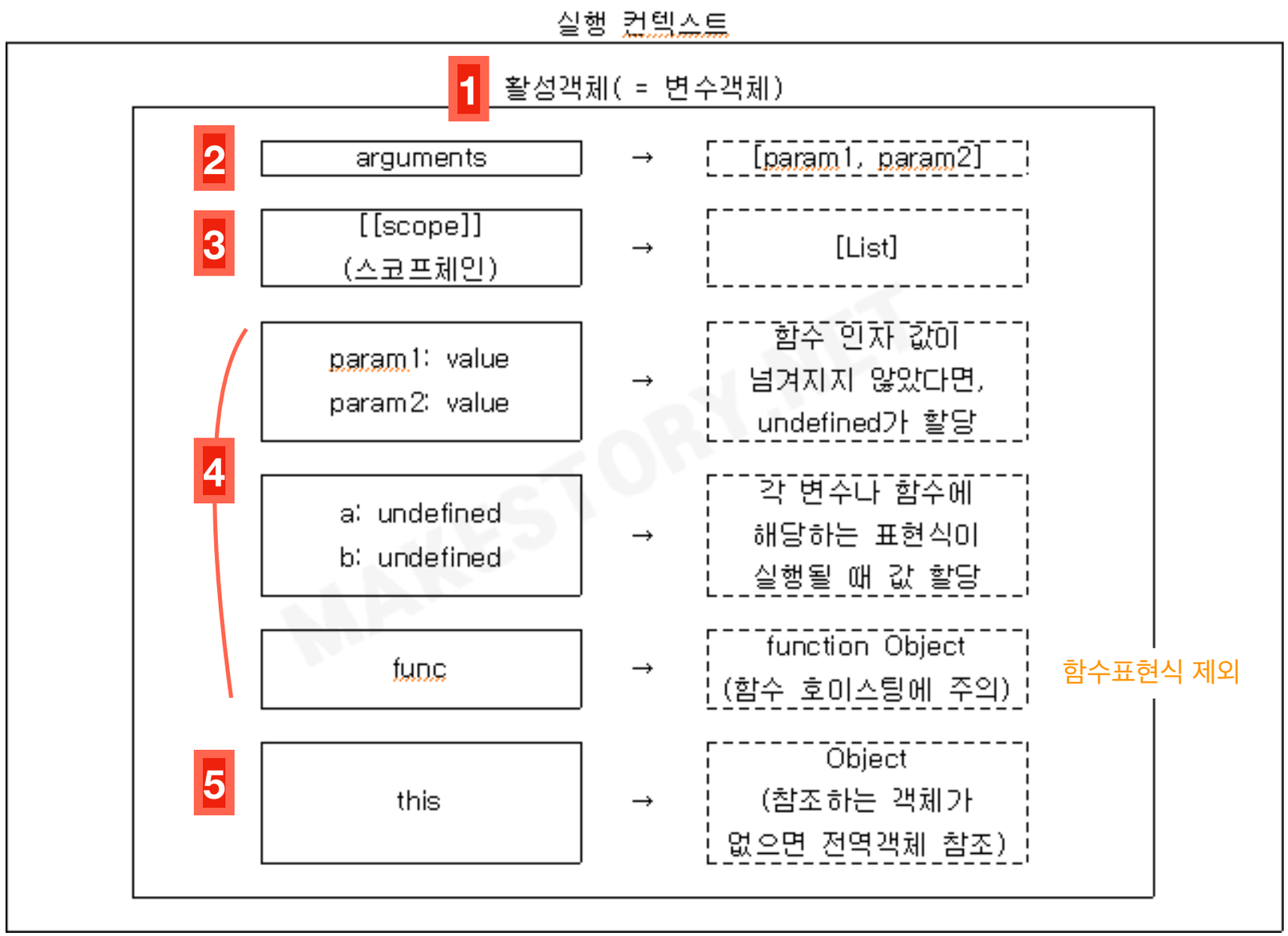
이렇게 실행되면서 **변수의 초기화 및 연산, 또 다른 함수 실행** 등이 이루어 진다.

(undefined 가 할당된 변수에 실제 값이 할당됨)

실행 컨텍스트 생성과정

JavaScript

```
function execute(param1, param2) {
    var a = 1, b = 2;
    function func() {
        return a + b;
    }
    return param1 + param2 + func();
}
excute(3, 4);
```



argument 객체 (유사배열객체)

JavaScript

자바스크립트에서는 함수를 호출할 때 함수 형식에 맞춰 인자를 넘기지 않더라도 에러가 발생하지 않는다.

자바스크립트의 이러한 특성 때문에 함수 코드를 작성할 때,
호출된 인자의 개수를 확인하고 이에 따라 동작을 다르게 해줘야 할 경우가 있다.
이를 가능하도록 하는 것은 **argument객체**다.

자바스크립트에서는 함수를 호출할 때 인수들과 함께
암묵적으로 arguments 객체가 함수 내부로 전달되기 때문이다.
arguments 객체는 함수를 호출할 때 넘긴 인자들이 배열 형태로 저장된 객체를 의미한다.
특이한 점은 **이 객체는 실제 배열이 아닌 유사 배열 객체이라는 것**

length 프로퍼티를 가진 객체(일반객체에 length라는 프로퍼티 존재)를 **유사배열 객체(array-like object)**
유사배열 객체의 경우, **객체지만 표준 배열 메서드를 활용하는 것이 가능**

{0: '', 1: '', ..., **length**: x}

[참고] 화살표 함수는 arguments 객체를 바인드 하지 않습니다.펼침연산자 활용
[참고] 파라미터는 내부프로퍼티 [[FormalParameters]]에 배열 형태로 저장

스코프 체인

JavaScript

자바스크립트도 다른 언어와 마찬가지로 **스코프, 즉 유효 범위**가 있다.

이 유효 범위 안에서 변수와 함수가 존재한다.

자바스크립트는 오직 **함수만이 유효 범위의 한 단위**가 된다.

이 **유효 범위를 나타내는 스코프가 [[scope]] 프로퍼티로 각 함수 객체 내에서 연결 리스트 형식으로 관리되는데, 이를 스코프 체인**이라고 한다.

스코프 체인은 다음과 같이 **각 실행 컨텍스트의 변수 객체가 구성 요소인 리스트와 같다**

3 | ..

2 | 변수 객체(=활성객체) 2

1 | 변수 객체(=활성객체) 1

0 | 변수 객체(=활성객체) 0

각각의 함수는 [[scope]] 프로퍼티로 자신이 생성된 실행 컨텍스트의 스코프 체인을 참조한다.

함수가 실행되는 순간 실행 컨텍스트가 만들어지고,

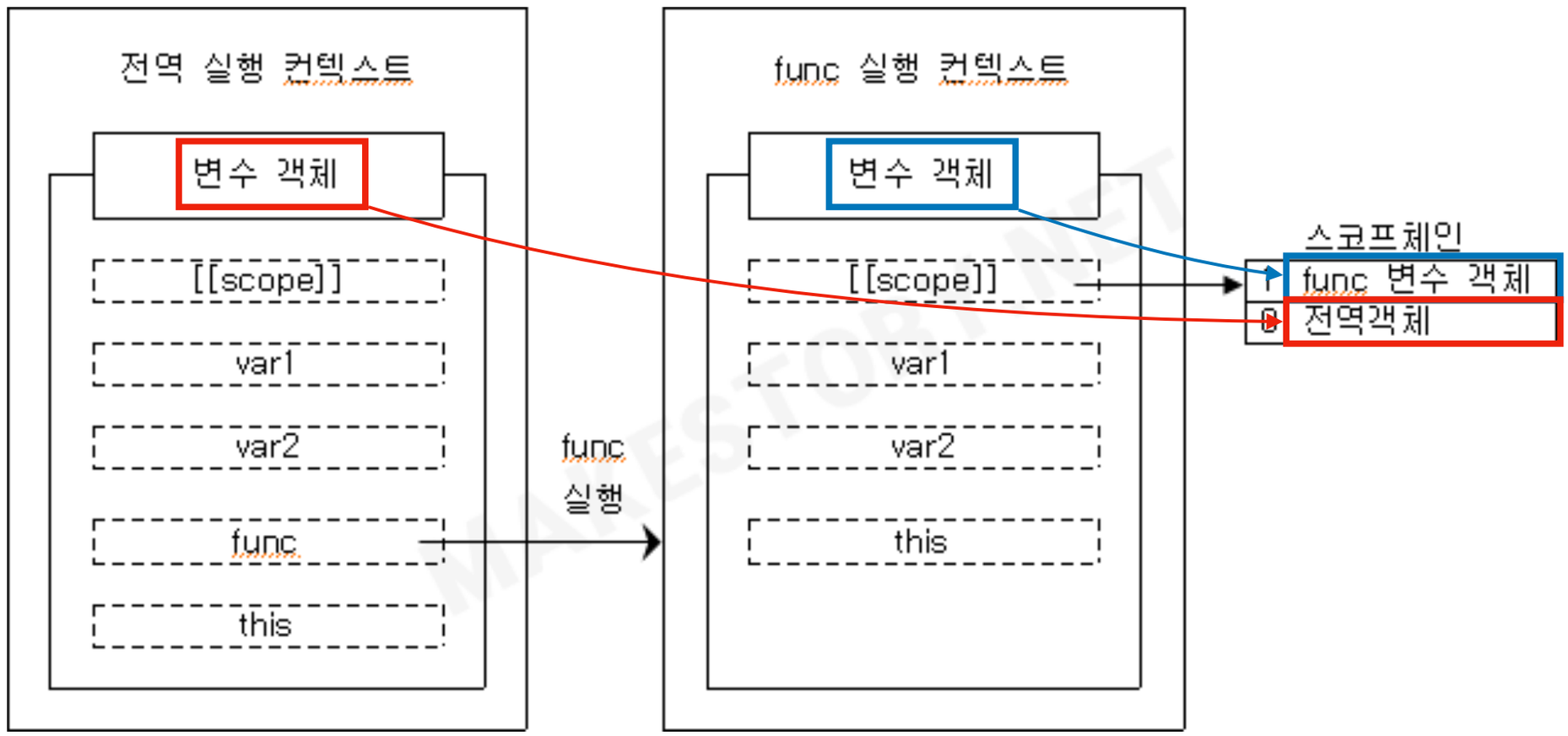
이 실행 컨텍스트는 **실행된 함수의 [[scope]] 프로퍼티를 기반으로 새로운 스코프 체인을 만든다.**

스코프 체인 = 현재 실행 컨텍스트의 변수 객체(=활성객체) + 상위 컨텍스트의 스코프체인

스코프 체인

JavaScript

```
var var1 = 1;
var var2 = 2;
function func() {
    var var1 = 10;
    var var2 = 20;
}
func();
```



스코프 정보는 **부모(상위) 컨텍스트의 스코프 체인 정보를 기반**으로
현재 실행 컨텍스트의 정보를 연결리스트 최상단에 추가시키는 방식

클로저

JavaScript

- 이미 **생명 주기가 끝난 외부 함수의 변수를 참조하는 함수를 클로저**라고 한다.
- 클로저로 참조되는 외부 변수를 자유변수(Free variable)

```
function outerFunc() {
  var x = 1; // 자유변수(free variable)
  return function() { // 클로저
    // x와 arguments를 활용한 로직
  };
}
var new_func = outerFunc();
// outerFunc 실행 컨텍스트가 끝났다
new_func();
```

함수는 호출될 때, 상위 스코프 정보(활성객체)를 참조하여 스코프 정보를 생성한다.
클로저는 이를 활용하는 전형적인 패턴

함수 호이스팅

JavaScript

실행 컨텍스트 생성과정에서 변수생성(=활성객체) 단계와 코드 실행 단계가 있다는 것을 확인했다.
함수 호이스팅은 자바스크립트의 변수생성(Instantiation)과 초기화(Initialization)의 작업이 분리되어 진행되기 때문에 발생하는 것이다.

변수 생성 단계에서 변수에는 undefined 할당되고, 함수선언문은 Function Object로 생성
실제 값이 할당되는 실행단계에서 함수 표현식은 Function Object로 생성되어 변수에 할당

즉, **실행 단계에서 함수선언문은 Function Object로 이미 생성된 후라 함수 호이스팅이 가능**하며,
함수 표현식은 아직 Function Object 생성 전이라 함수 호이스팅이 불가능한 것이다.

호이스팅(hoisting)은 ECMAScript 2015 언어 명세 및 그 이전 표준 명세에서 사용된 적이 없는 용어입니다.
호이스팅은 JavaScript에서 실행 콘텍스트(특히 생성 및 실행 단계)가 어떻게 동작하는가
설명을 위한 마케팅용어 입니다.
일부 호이스팅을 변수 및 함수 선언이 물리적으로 작성한 코드의 상단으로 옮겨지는 것으로 가르치지만,
실제로는 그렇지 않습니다.

[참고] 호이스팅 용어에 대한 내용 https://developer.mozilla.org/ko/docs/Glossary/Hoisting

즉시 실행 함수

JavaScript

- **함수를 정의함과 동시에 바로 실행하는 함수를 즉시 실행 함수(immediate functions)**
즉시 실행 함수의 경우, 같은 함수를 다시 호출할 수 없다.

따라서 즉시 실행 함수의 이러한 특징을 이용한다면,
최초 한 번의 실행만을 필요로 하는 초기화 코드 부분(크로스브라우저 구현에 활용) 등에 사용할 수 있다.
(또는 private, public 패턴과 모듈화 활용)

* 발표자가 즉시 실행함수를 활용했던 경험 (함수 실행컨텍스트 생성 특성 활용)
동일한 변수를 사용하는 라이브러리, window/globl(node.js)등 분기처리, 라이브러리 패턴(샌드박스 패턴)
공개/비공개(캡슐화, 정보은닉), 조건문에서 실행,
초기화 시점의 분기(크로스브라우징 대응에 자주 활용), 재귀호출

정리

JavaScript

객체

__proto__ 링크로 객체는 자신의 것 외 추가적으로 사용할 수 있는 상위(부모) 객체와 연결되어 있다. (추가/수정/제거 가능)

함수

함수 생성 방법(함수 표현식 추천)에 따라 호출가능한 위치가 다를 수 있다. 함수객체만의 prototype, length 프로퍼티

생성자 함수

new 키워드를 통해 생성된 객체는 생성자 함수의 프로토타입(prototype)객체와 연결된다는 것 (__proto__ 암묵적 링크)

this

this 값은 함수(메소드)가 실행되는 암묵적인 객체(앞) 또는 명시적인 값(뒤)

실행 컨텍스트

하나의 실행단위, 실행 가능한 자바스크립트 코드 블록이 실행되는 환경 (전역코드, eval(), 함수실행), 실행 전 준비과정
실행에 필요한 여러가지 정보(생성되는 변수저장 변수객체, 스코프체인, this 값 등)를 담고 있다.
실행컨텍스트 이해하면 스코프체이닝, 클로저, 호이스팅, 초기화(undefined)를 알 수 있음

[참고] 실무에서의 빠른 이해와 적용을 위해, 대략적으로 표현한 것입니다.

event

JavaScript

W3C에서는 웹 브라우저 애플리케이션에서 **이벤트 처리를 표준화**하기 위해
"DOM Level2 이벤트 모델"이라는 표준을 제시

DOM Level 2 이벤트 모델에는 이벤트를 등록하거나 제거하는 표준적인 방법을 비롯해
이벤트 전파 메커니즘에 대한 표준, 이벤트 핸들링 메서드로 전달되는 이벤트 객체에 대한 표준 등이 포함되어 있다.
(IE 는 버전 9부터 표준을 지키고 있다.)

이벤트가 브라우저에서 발생하면 DOM의 최상위 객체인 Document 객체로 이벤트가 전달된다.
그런 다음 이벤트를 발생시킨 요소에 해당하는 DOM 객체로 이벤트가 전달되고
해당 객체에서 이벤트 핸들러가 호출된다.
이처럼 이벤트가 흘러가는 과정을 **이벤트 흐름(event flow)**이라고 한다.

[참고] https://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107/events.html

event

JavaScript

브라우저 id=“deep3” 클릭이 발생 했을 때,

```
<body>
    <div id="deep1">
        <div id="deep2">
            <div id="deep3">
             <!— 여기서 클릭이 발생 //—>
            </div>
        </div>
    </div>
</body>
```

캡처링 단계 (capture phase)

이벤트가 문서의 루트 객체인 Document를 거쳐
이벤트가 발생한 타겟 객체의 부모 객체까지 전달되는 단계

타겟 단계(target phase)

이벤트가 발생한 객체로 전달되는 단계

버블링 단계(bubbling phase)

타겟 객체의 부모에서 Document 객체까지 전달되는 단계

이벤트 흐름(event flow)

document.body -> #deep1 -> #deep2 -> #deep3 -> #deep2 -> #deep1 -> document.body

브라우저 이벤트 흐름에서 개발자가 컵쳐링/버블링 단계 중 선택하여 이벤트를 등록/제거
캡쳐링 단계 : document.body -> div.deep1 -> div.deep2 -> div.deep3
버블링 단계 : div.deep3 -> div.deep2 -> div.deep1 -> document.body

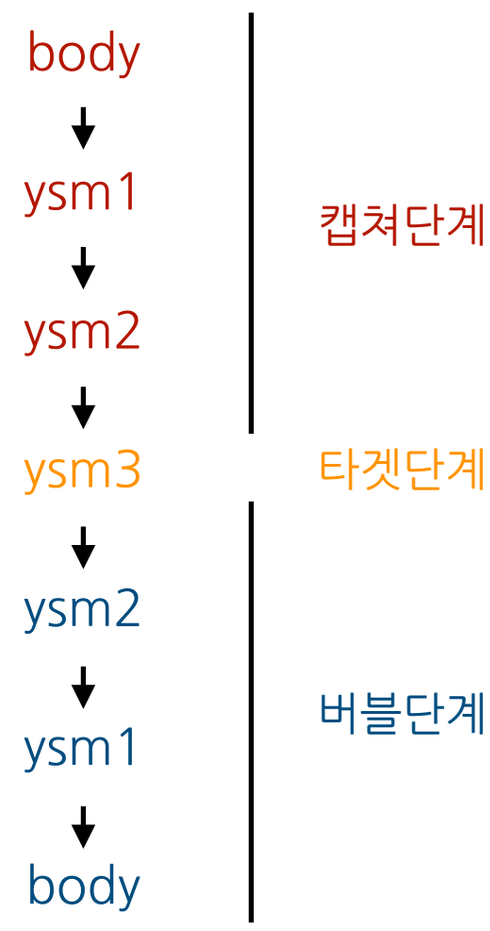
[참고] 구버전 IE는 캡쳐링를 지원하지 않기 때문에 공통적으로 버블링으로 설정하여 대부분 사용 (jQuery 포함)

event

JavaScript

```
<body>                          .addEventListener('click', handler, true);
    <div id="ysm1">             .addEventListener('click', handler, true);
        <div id="ysm2">         .addEventListener('click', handler, true);
            <div id="ysm3">
                <!— 여기서 클릭 발생했을 경우 이벤트 흐름 //—>
            </div>
        </div>                  .addEventListener('click', handler, false);
    </div>                      .addEventListener('click', handler, false);
</body>                         .addEventListener('click', handler, false);
```

이벤트흐름



2000년 11월에 W3C가 발표한 **DOM Level 2 이벤트 모델**에서 제시한 **이벤트 등록/제거 방법**

element.addEventListener(string type, Function handler, boolean useCapture);

element.removeEventListener(string type, Function handler, boolean useCapture);

[참고] ysm1~3 은 element로 document.getElementById('ysm1~3') 반환값 생략

event

JavaScript

이벤트 정지

```
// 이벤트 전파 정지
event.stopPropagation();

// 대상 요소의 디폴트 이벤트 방지
// (document, body 이벤트 등록시 passive 값 확인필요)
event.preventDefault();

// 같은 이벤트의 다른 리스터 호출 정지
event.stopImmediatePropagation();
```

[참고] https://developer.mozilla.org/ko/docs/Web/API/Event

jQuery

JavaScript

$('#ysm') 과 $('#ysm') 은 같지 않다.

```
jQuery = function( selector, context ) {
    return new jQuery.fn.init( selector, context, rootjQuery );
};

jQuery.fn = jQuery.prototype = {
    constructor: jQuery,
    init: function( selector, context, rootjQuery ) {}
};

jQuery.fn.init.prototype = jQuery.fn;

window.jQuery = window.$ = jQuery;
```

[jQuery 1.x 기준 코드참고] https://code.jquery.com/

jQuery

JavaScript

$('#ysm').show() 호출했을 때, 요소의 기본 display 값을 찾는다.

```
function css_defaultDisplay( nodeName ) {
    var doc = document,
        display = elemdisplay[ nodeName ];
    if ( !display ) {
        display = actualDisplay( nodeName, doc );
        if ( display === "none" || !display ) {
            iframe = ( iframe ||
                jQuery("<iframe frameborder='0' width='0' height='0'/>")
                .css( "cssText", "display:block !important" )
            ).appendTo( doc.documentElement );
            doc = ( iframe[0].contentWindow || iframe[0].contentDocument ).document;
            doc.write("<!doctype html><html><body>");
            doc.close();
            display = actualDisplay( nodeName, doc );
            iframe.detach();
        }
        elemdisplay[ nodeName ] = display;
    }
    return display;
}
```

```
function actualDisplay( name, doc ) {
    var elem = jQuery(doc.createElement(name)).appendTo(doc.body),
        display = jQuery.css(elem[0], "display");
    elem.remove();
    return display;
}
```

[jQuery 1.x 기준 코드참고] https://code.jquery.com/

jQuery

JavaScript

$('#ysm').append, prepend, before, after 호출했을 때, 코드 내부 script 존재하는지 검사한다.

```
domManip: function( args, callback, allowIntersection ) {
    // 내부 일부코드 생략…

    if ( hasScripts ) {
        doc = scripts[ scripts.length - 1 ].ownerDocument;
        jQuery.map( scripts, restoreScript );
        for ( i = 0; i < hasScripts; i++ ) {
            node = scripts[ i ];
            if ( rscriptType.test( node.type || "" ) &&
                !jQuery._data( node, "globalEval" ) && jQuery.contains( doc, node ) ) {
                if ( node.src ) {
                    // Hope ajax is available...
                    jQuery._evalUrl( node.src );
                } else {
                    jQuery.globalEval( ( node.text || node.textContent || node.innerHTML || "" ).replace( rcleanScript, "" ) );
                }
            }
        }
    }

    // 내부 일부코드 생략...
}
```

[jQuery 1.x 기준 코드참고] https://code.jquery.com/

jQuery

JavaScript

CSS Style 적용을 위한 Class 값, JavaScript 기능 적용(또는 사용)을 위한 Class 값 등 역할에 따른 분리

```
// CSS Style 적용 목적의 class 값과 JavaScript 에서 사용하는 class 역할분리 사용 추천

// CSS 유지보수/확장 경우, JavaScript 영향도까지 검사해야 한다.

$('.style_class'); // 예를 들어, CSS Style 적용을 위한 class 값을 JavaScript 에서 사용하는 기존 형태

$('.js_style_class'); // 'js_' 시작하는 형태로 JavaScript 에서 활용하는 값 별도 주입 (CSS 담당자와 사전 공유형태에서 사용)
```

jQuery

JavaScript

이벤트 Off 그리고 On, 중복 이벤트 등록/실행 방지

```
// 이벤트 Off 후 On 등록
function setEvent() {
    // #ysm 에 등록된 click 이벤트 전체 해제 후 설정
    $('#ysm').off('click').on('click', function() {});
    // 또는 #ysm 에 등록된 click 이벤트 중 EVENT_CLICK 이벤트키를 가진 이벤트 해제 후 설정
    $('#ysm').off('click.EVENT_CLICK').on('click.EVENT_CLICK', function() {});
}
setEvent();
setEvent(); // 다른 부분에서 이벤트 중복 등록(호출)이 발생된 경우 (코드 복잡도/유지보수 증가에 따른 발생 가능성)

// 또는 이벤트 델리게이션
$(document).off('click.EVENT_DOCUMENT_CLICK').on('click.EVENT_DOCUMENT_CLICK', '.ysm', function(event) {
    // 관련 로직 실행
    // ...
});
```

감사합니다.

http://makestory.net