

JavaScript Brief Review

Bok, Jong Soon
javaexpert@nate.com
<https://github.com/swacademy/Vue.js>

JavaScript Type

■ Primitive Type

- Number

- 실수, 부동소수점 64비트(double)

- String

- 문자열

- Boolean

- True, False

- undefined

- 변수에 값이 할당되지 않을 때 Interpreter가 undefined로 할당.

- null

- 개발자가 의도적으로 할당하는 값.
- `typeof` 값이 Object 로 반환.
- 따라서 `===` 로 확인

```
var nullCheck = null;  
console.log(typeof nullCheck === null); // false  
console.log(nullCheck === null); // true
```

JavaScript Type (Cont.)

■ Reference Type

- Object
- Array
- Date
- Math
- RegExp
- Function

NaN (Not a Number)

- 수치 연산을 해서 정상적인 값을 얻지 못할 때 발생하는 Error

```
console.log(1 - 'hello'); // NaN

var foo = {
  name: 'foo',
  major: 'cs'
};
foo['full-name'] = 'ffoo';
console.log(foo['full-name']); // 'ffoo'
console.log(foo.full-name); // NaN, 프로퍼티명이 연산자를 포함할 경우
```

delete Operator

- 객체 Property를 삭제하는 기능
- 객체 삭제는 불가능

```
// 1. 객체 프로퍼티를 삭제
```

```
var foo = {  
  name: 'foo',  
  nickname: 'pangyo'  
};
```

```
delete foo.nickname;  
console.log(foo.nickname);  
console.log(foo); // {name: "foo"}
```

```
// 2. delete 로 객체를 삭제할 경우 (변화 없음)
```

```
var foo = {  
  name: 'foo',  
  nickname: 'pangyo'  
};
```

```
delete foo;  
console.log(foo); // {name: "foo", nickname: "pangyo"}
```

값 비교와 주소 비교

- Primitive Type은 값 비교
- Reference Type은 주소 비교
- `==` operator 사용

```
var a = 10;
var b = 10;

var objA = {
  value: 100
};
var objB = {
  value: 100
};
var objC = objB;

console.log(a == b); // true
console.log(objA == objB); // false
console.log(objB == objC); // true
```

Array와 Object 구분하기

```
var arr = [];
```

```
var obj = {};
```

```
arr.constructor.name; // "Array"
```

```
obj.constructor.name; // "Object"
```

delete Operator & splice() Method in Array

- Array에서 delete를 사용하면 Element의 값만 undefined로 변경하고, 해당 Element Index는 지우지 않는다.

```
var arr = [1, 2, 3];  
delete arr[1];  
console.log(arr); // [1, undefined × 1, 3]
```

- splice() Method는 해당 Element를 삭제한다.

```
var arr = [1, 2, 3];  
arr.splice(1, 1);  
console.log(arr); // [1, 3]
```


typeof Operator

- 각 Data Type에 대한 **typeof** 수행결과는 다음과 같다.

```
var num = 10;
var str = "a";
var boolean = true;
var obj = {};
var undefined;
var nullValue = null;
var arr = [];
function func() {}

console.log(typeof num); // number
console.log(typeof str); // string
console.log(typeof boolean); // boolean
console.log(typeof obj); // object
console.log(typeof undefined); // undefined
console.log(typeof nullValue); // object (null 은 object)
console.log(typeof arr); // object (배열도 object)
console.log(typeof func); // function
```

== Operator와 === Operator

- 가장 큰 차이점은 값 뿐만 아니라 Type까지 비교하는지 여부이다.
- == Operator는 수행할 때 Type이 다를 경우 Type을 일치시킨 후 값을 비교하는 특징이 있다.

```
console.log(1 == '1'); // true  
console.log(1 === '1'); // false
```

함수의 length Property

```
function func1(a) { return a; }  
function func2(a, b) { return a + b; }  
function func3(a, b, c) { return a + b + c; }  
  
console.log('func1 length : ' + func1.length); // func1 length : 1  
console.log('func2 length : ' + func2.length); // func2 length : 2  
console.log('func3 length : ' + func3.length); // func3 length : 3
```

내부 함수

- 함수의 내부에 또 다른 함수를 정의한 함수

```
function parent() {  
  var a = 10;  
  var b = 20;  
  
  function child() {  
    var b = 30;  
    console.log(a);  
    console.log(b);  
  }  
  child();  
}  
parent(); // 10, 30  
child(); // child is not defined
```

Constructor Function

- 일반 객체 선언과 다르게 여러 개의 객체를 생성할 수 있는 함수.
- 함수이름 제일 앞에는 대문자로, 호출시에는 **new** Operator 사용.

```
function Student(name, age, tel){  
    this.name = name;  
    this.age = age;  
    this.tel = tel;  
}  
var chulsu = new Student('김철수', 24, '010-1234-5678');  
var younghee = new Student('이영희', 26, '010-9876-5432');  
console.log(chulsu);  
console.log(younghee);
```

instanceof를 활용한 생성자 함수 구분하기

- JavaScript는 생성자 함수 형식이 별도로 없기 때문에 기존 함수에 **new** Operator를 붙여서 생성자 함수 생성 가능.
- 생성자 함수가 아님에도 **new**를 붙일 수 있기 때문에 이를 방지하기 위해 다음과 같은 기법이 필요.
- 대부분의 Open Source Library에서 사용하는 Pattern.

```
function Func(arg) {  
  // instanceof 로 생성자 함수임을 확인  
  if (!(this instanceof arguments.callee)) // 'this instanceof 함수명' 도 가능  
    return new Func(arg);  
  this.value = arg || 0;  
}  
  
var a = new Func(100);  
var b = Func(200);  
console.log(a.value);  
console.log(b.value);
```

prototype & constructor

```
function func() {  
    return true;  
}  
console.log(func.prototype);  
console.log(func.prototype.constructor);
```

Prototype Chaining

- 해당 함수에 존재하지 않는 Property, Method를 부모 객체(Prototype 객체)를 찾는다.

```
var obj = {  
  name: 'Smith',  
  printName: function () {  
    console.log(this.name);  
  }  
};  
obj.printName(); // 'Smith'  
obj.hasOwnProperty('name'); // true  
obj.hasOwnProperty('city'); // false
```

- **obj**에서 사용한 **printName()** Method는 **obj**에 선언되었기 때문에 사용 가능.
- 하지만 **hasOwnProperty()** Method는 선언되지도 않았는데 사용할 수 있다.
- 이유는 **obj**의 Prototype 객체가 **Object**이고, **Object**에 내장된 Method가 **hasOwnProperty()** 이기 때문에, **obj**에서 Prototype 객체의 **hasOwnProperty()**를 호출한다.

Object, String, Number Prototype 객체 Method 재정의

- JavaScript에서 기본으로 제공하는 Object, String, Number등의 표준 객체에 사용자가 원하는 기능을 재정의하여 사용 가능.

```
String.prototype.printText = function (text) {  
    console.log('Hello ' + text);  
};  
var name = "Smith";  
name.printText('JavaScript World!!!'); // 'Hello JavaScript World!!!'
```

즉시 실행 함수

- 함수를 정의함과 동시에 바로 실행하는 함수.
- 함수를 다시 호출할 수 없다는 특징
- 최초 한 번의 실행만 요구되는 초기화 Code에 적합.
- jQuery와 같은 Open Source Library 에서 사용.

```
(function (name) {  
    console.log('This is the immediate function : ' + name);  
})('foo');
```

```
$(function(){  
    |      $('body').css('background-color', 'yellow');  
});
```

Closure

- 실행이 끝난 함수의 Scope를 참조할 수 있는 함수.

```
function parent() {  
  var a = 'Parent is done';  
  function child() {  
    console.log(a);  
  }  
  return child;  
}  
var closure = parent();  
closure();
```

- 위 Code에서 **parent**의 내부함수인 **child()**는 외부에서 접근이 불가능하다.
- 하지만 **return** 값에 **child**를 넘김으로써 외부에서도 **child**를 호출할 수 있게 된다.
- 따라서, **child()**에서 **parent**의 값을 참고하고 있다면, **child()**를 밖에서 호출함으로써 **parent()**의 변수에 접근 가능

map() 구현

```
// definition
Array.prototype.myMap = function(callback) {
    arr = [];
    for (var i = 0; i < this.length; i++)
        arr.push(callback(this[i], i, this));
    return arr;
};

//tests
var arrs = ['dic tanin', 'boo radley', 'hans gruber'];
var numbers2 = [1, 4, 9];

var goodT = arrs.myMap(function(n) {
    return n;
});

var squareRoot = numbers2.myMap(function(num) {
    return Math.sqrt(num);
});

console.log(goodT); // [ 'dic tanin', 'boo radley', 'hans gruber' ]
console.log(squareRoot); // [ 1, 2, 3 ]
```

JavaScript Variable Initialization

- Variable Declaration
 - 변수를 활성 객체에 할당
- Variable Initialization
 - 변수 값에 **undefined** 할당
- Variable Value Assignment
 - 변수에 실제 값 할당

실행 Context 이해하기

- 비동기 실행 방식인 **setTimeout()**를 이용한 예제.

```
console.log("1");
function exec() {
  setTimeout(function() {
    console.log("2");
  }, 3000);
  setTimeout(function() {
    console.log("3");
  }, 0);
  console.log("4");
  setTimeout(function() {
    console.log(5);
  }, 1000);
}
console.log(exec());
// 위 코드 실행 결과 : 1, 4, 3, 5, 2
```

실행 Context 이해하기 (Cont.)

- **setTimeout()**이 지연시간이 0이더라도 실행 Context가 다르기 때문에 1,4가 먼저 출력된다.
- 다음의 **for**문과 **setTimeout()**을 보자.

```
var i;  
for (i = 0; i < 5; i++) {  
    setTimeout(function() {  
        console.log(i); // 5, 5, 5, 5, 5  
    }, 1000);  
}
```

- 위의 Code를 실행하면, 이 Code가 실행되는 Main Context와 **setTimeout**이 실행되는 Context가 다르기 때문에 일반 프로그래밍 지식 관점에서는 0,1,2,3,4 이라고 추측하지만, 실제로는 **for** 문의 실행이 모두 끝난 후에 **setTimeout**의 함수가 실행되기 때문에 숫자 5가 다섯 번 출력된다.

argument Object

- 함수 호출시에 넘겨진 실제 인자 값을 가진 배열

```
// 아래 함수 정의에 포함된 인자 값은 2개  
function add(a, b) {  
    console.dir(arguments);  
}  
console.log(add(1)); // Arguments(1), 0: 1  
console.log(add(1, 2)); // Arguments(2), 0: 1, 1: 2  
console.log(add(1, 2, 3)); // Arguments(3), 0: 1, 1: 2, 2: 3
```


argument Object (Cont.)

■ arguments의 활용

- Method에 넘겨 받을 인자의 개수를 모를 때 유용하다.

```
function sum() {  
    for (var i = 0, result = 0; i < arguments.length; i++) {  
        result += arguments[i];  
    }  
    return result;  
}  
  
console.log(sum(1,2,3)); // 6  
console.log(sum(1,2,3,4,5,6)); // 21
```

this Binding

- 일반적으로 함수 내부에서 **this**를 사용하면 전역 Scope(**window**)에 접근한다.

```
// 함수 선언식
var text = 'global';
function binding() {
  var text = 'local';
  console.log(this.text); // 'global'
  console.log(this); // Window {stop: f, open: f, alert: f, confirm: f, prompt: f, ...}
}
binding();

// 함수 표현식
var text = 'global';
var binding = function() {
  var text = 'local';
  console.log(this.text); // 'global'
  console.log(this); // Window {stop: f, open: f, alert: f, confirm: f, prompt: f, ...}
}
binding();
```

this Binding (Cont.)

- 객체의 속성에서 함수를 선언하고 **this**를 사용하면 해당 객체에 접근한다.

```
var text = 'global';
var binding = {
  text: 'local',
  printText: function () {
    console.log(this.text); // 'local'
    console.log(this); // {text: "local", printText: f}
  }
};
binding.printText();
```

this Binding (Cont.)

- 함수의 내부함수에서 **this**를 사용하면 전역 객체(**window**)에 접근한다.

```
var text = 'global';
var binding = {
  text: 'local',
  printText: function () {
    console.log(this.text); // local
    var innerFunc = function () {
      console.log(this.text); // global
    };
    innerFunc();
  }
};
binding.printText();
```

Scope Chain을 이해하기 위한 예제

- 전역 scope와 함수 scope를 구분하자.

```
// ex.1
var a = 1;
var b = 2;
function func() {
  var a = 10;
  var b = 20;
  console.log(a); // 10
  console.log(b); // 20
}
func();
console.log(a); // 1
console.log(b); // 2
```

Scope Chain을 이해하기 위한 예제 (Cont.)

- 다음은 내부 함수 **innerfunc**에서 외부 함수인 **func**의 변수에 접근하고 있다.

```
// ex.2
var a = 1;
function func() {
  var a = 2;
  function innerfunc() {
    return a;
  }
  console.log(innerfunc()); // 2
}
func();
```

Scope Chain을 이해하기 위한 예제 (Cont.)

- 다음은 **func1**의 실행 Context가 전역인 것에 주의하자.

```
// ex.3
var a = 1;
function func1() {
    return a;
}
function func2(func1) {
    var a = 2;
    console.log(func1()); // 1
}
func2(func1);
```

Closure 정의 및 Code

- 외부 함수의 실행이 종료되어 Context가 반환되더라도, 내부 함수로 종료된 외부 함수의 Scope에 접근이 가능한 기법 → Scope Chaining
- 이미 생명주기가 끝난 외부 함수의 변수를 참조하는 함수

```
function func() {  
  var a = 1;  
  var cl = function () {  
    console.log(a);  
  };  
  return cl  
}  
var result = func();  
console.dir(result); // [[Scope]] 에서 Closure 함수임을 확인 가능  
result();
```

▼ `[[Scopes]]`: Scopes[2]

- ▶ 0: Closure (func) {type: "closure", object: {...}, name: "func"}
- ▶ 1: Global {type: "global", object: Window, name: ""}

Closure 정의 및 Code (Cont.)

- 일정한 형식을 가진 Template에서 입력된 값에 따라 다른 결과물을 내는 Code 작성 가능.

```
var str = [  
    'hello ',  
    '',  
    ' world'  
];  
  
function completeSentence(name) {  
    str[1] = name;  
    return str.join('');  
}  
  
completeSentence('js');    //hello js world
```

Closure 정의 및 Code (Cont.)

- 앞 Slide의 Code에 Closure를 적용해 보자.

```
function completeSentence(name) {  
    var str = [  
        'hello ',  
        '',  
        ' world'  
    ];  
    return function () {  
        str[1] = name;  
        return str.join('');  
    };  
}  
  
var result = completeSentence('js');  
result();    //hello js world
```

Closure 정의 및 Code (Cont.)

- 앞 Slide의 Code를 좀 더 기능 단위로 분할해 보자.

```
function completeSentence(name) {  
  var str = [  
    'hello ',  
    '',  
    ' world'  
  ];  
  // 입력된 문자열로 문장을 완성하는 기능  
  var complete = function () {  
    str[1] = name;  
    return str.join('');  
  };  
  // 문장 완성 기능을 클로저로 빼는 역할  
  var closure = function () {  
    return complete();  
  };  
  return closure;  
}  
var result = completeSentence('js');  
result();
```

Closure 활용

- Closure를 활용하여 Java나 다른 언어처럼 속성 및 Method의 범위를 정할 수 있다.

```
// Closure로 Java의 Class와 유사하게 모듈화한 예제
var Module = (function () {
    var privateProperty = 'foo';
    function privateMethod(args) {
        console.log('private method');
    }

    return {
        publicProperty: '',
        publicMethod: function (args) {
            console.log("public method");
        },
        privilegedMethod: function (args) {
            return privateMethod(args);
        }
    };
})();

Module.privilegedMethod(); //private method
```

Reference

- 인사이드 자바스크립트, 한빛미디어
- [Understanding Scope and Context in Javascript](#)
- [alex gist - native js implementation](#)
- [Demystifying JavaScript Closures, Callbacks and IIFEs](#)
- [간단히 훑어보는 자바스크립트 기본기 다지기](#)

