Javascript_2

for 문과 if 문, switch case

for

```
var length = 10;

for (var i = 0; i < length; i++) {
  console.log('length : ' + i);
}
```

if

```
var value = 0;
// var value = '';

if (value) {
  console.log('Y', typeof(value), value);
} else {
  console.log('N', typeof(value), value);
}
```

case

```
var caseValue = 2;

switch (caseValue) {
  case 1 :
  console.log('case1 : ' + caseValue);
  break;
  case 2 :
  console.log('case2 : ' + caseValue);
  break;
  case 3 :
  console.log('case3 : ' + caseValue);
  break;
}
```

Array와 Object의 활용

Array

- Array 생성

함수 생성법

```
var myArray = new Array(10);     // 10개의 공간을 가진 비어있는 배열이 생성된다.

var myArray = new Array(1,2,3);  // 1, 2, 3 으로 초기화된 3개의 공간을 가진 배열
```

Array 리터럴 사용

```
var myArray = [];      // 0개의 공간을 가진 빈 배열생성

var myArray = [1,2,3]; //  1, 2, 3 으로 초기화된 3개의 공간을 가진 배열
```

배열을 선언하는 방법은 위의 두가지 방법이 있지만 첫번째인 *함수 생성법*은 기본적으로 권장하지는 않는다.
함수 생성법보다는 Array 리터럴을 사용하여 선언하는 방법이 훨씬 직관적이며 실수를 줄일 수 있다.

Array에 데이터 저장하기

```
var myArray = [3, "hello!", function(){
    return 5;
}, { "color" : "blue" }, true];
```

JavaScript의 배열에는 어떠한 값도 저장할 수 있다.
위 예제는 Array 리터럴을 사용하여 다양한 값을 할당하는것을 보여준다.

- 배열의 index를 직접적으로 접근하여 데이터를 저장할 수 있다.

```
var myArray = [];
myArray[0] = 3;
myArray[1] = "hello!";
myArray[2] = function(){ return 5; };
myArray[3] = {"color" : "blue" };
myArray[4] = true;
```

- JavaScript의 Array는 여러가지 메소드를 제공한다. push 메소드를 사용하면 배열의 마지막에 새로운 원소를 추가하는것이 가능하다.

```
var myArray = [];
myArray.push(3);
myArray.push("hello");
myArray.push(function(){ return 5; });
myArray.push( {"color" : "blue" });
myArray.push(true);
```

- Array의 데이터 사용하기

```
var myArray = [];
myArray[0] = 3;
myArray[1] = "hello!";
myArray[2] = function(){ return 5; };
myArray[3] = {"color" : "blue" };
myArray[4] = true;
```

위와 같은 자료구조를 가진 배열 myArray가 있다고 하면 해당 index를 사용하여 해당 index에 저장되어 있는 데이터를 각자의 형에 맞게 사용하면 된다.

```
console.log(myArray[0]);
console.log(myArray[1]);

var getFive = myArray[2];
console.log(getFive());

console.log(myArray[3].color);
console.log(myArray[4]);
```

메소드

Array객체에서 다루는 메소드들을 소개한다.

메소드	반환값	설명
concat(Array arr)	Array	하나의 Array에 다른 Array를 붙여 하나의 Array로 만든다.
join(String delimeter)	String	모든 Array의 원소를 파라메터로 받은 구분자를 추가하여 하나의 String으로 반환한다.

pop( )	any Object	Array의 가장 마지막 원소를 반환후 제거한다.
push( )	Number index	Array의 가장 마지막 index에 새로운 원소를 추가한다.
reverse( )	Array	현재 Array의 순서를 뒤집는다.
sort(Function)	Array	현재 Array를 입력받은 Function의 결과 값에 의하여 정렬한다.
shift( )	any Object	Array의 가장 처음 원소를 반환후 제거한다.
unshift( )	Number 0	Array의 가장 처음 위치에 새로운 원소를 추가한다.
slice(Number index1, Number index2)	Array	입력받은 index1위치로부터 index2까지의 원소를 새로운 배열로 반환한다.
splice(Number index, Number n,new object)	Array	입력받은 index의 원소로부터 n개의 원소를 제거하고 _new object_를 넣는다
toString( )	String	Array의 원소를 String으로 반환한다.

- Tip : String 값 중 원하는 부분을 제거할 때 데이터 양이 많지 않을 경우 split()와 join()을 활용하여 간단하게 구현할 수있다
  ex) var temp = 'abcd.f';
  '.'을 제거하고 싶은 경우
  temp = temp.split('.').join('');

Object

- object 생성

```
var myObject = new Object();
var myObject = {};
```

- object 탐색

```
for(var p in employee){
   console.log( p + ":" + employee[p]);
};
```

- object 접근

```
var employee = {
    "developer" : [
        { "name" : "K.K.T", "ages" : 29 },
        { "name" : "L.H.Y", "ages" : 30 }
    ],
    "designer" : [
        { "name" : "K.E.S", "ages" : 31 },
        { "name" : "J.Y.H", "ages" : 33 }
    ]
};

console.log(employee.developer[0].ages);
console.log(employee[“designer”][1][“name”]);
```

- object의 활용

```
var commentObj = {
  defaultViewCount: 10,

  makeCommentElement: function(data) {//data 는 object
    var commentArray = [];
    var comments = data.comments;//comments는 data의 key값으로 value로는 각 댓글들이 array로 들어있음
    for (var i = 0, l = comments.length; i < l; i++) {
      var comment = comments[i];
      commentArray.push(jQuery('<li id="' + comment.id + '">' + comment.comment + '</li>'));
    }
    return commentArray;
  }
};
```

Function의 사용과 Scope 이해

Function 의 선언

Function의 프로시져 선언법

```
function hello(who) {
   console.log('Hello '+who);
}

hello('world');
```

Function의 변수적 선언법

```
var hello = function (who) {
  console.log('Hello '+who);
}

hello('world');
```

두 방식의 큰 차이점은 없다.
JavaScript에서는 named된 Function 이외에 변수처럼 접근가능한 방식을 제공한다.

Function 을 Function으로 넘기기

Function을 또다른 Function의 파라메터로 넘기기

```
var say = function(what) {
  what('say function');
}

var hello = function (who) {
  console.log('Hello '+who);
}

say(hello);
```

JavaScript의 Function은 파라메터로 받는값에 제한없이, Array, Object 혹은 다른 변수로도 받을수 있다.
Function 또한 하나의 Object로 취급되기 때문에 같은 방법으로 파라메터로 넘길수 있다.

또 다른 Function을 넘기는 방법들을 소개합니다. Function 자체의 참조를 넘길수도 Function의 실행 결과를 넘길수도 있다.

onclick 이벤트 헨들러로써 Function의 참조를 사용하는 방법

```
function doSomething() {
  console.log('you pushed my button!');
}

document.onclick = doSomething;
```

Function의 실행결과를 Function으로 취하는 방법

```
function pushedMyButton() {
  console.log('you pushed my button!');
}

function doSomething() {
  return pushedMyButton;
}

document.onclick = doSomething();
```

Event 헨들링에 관해서는 나중에 배우게 되겠지만 여기서는 document의 onclick 이벤트 헨들러로써 Function의 참조를 사용하고 있다. 첫번째 예제에서는 doSomething이라는 Function 자체를 넘김으로써 직접적으로 Function을 참조하고 있으며
두번째 에제에서는 doSomething의 return 값으로 pushedMyButton 이라는 Function을 사용함으로써 pushedMyButton Function을 간접적으로 참조하고 있다.

다음예제를 보면 Function이 어떤식으로 참조가 이루어 지는지를 볼수 있다.

```
var test = function () {
  return "This is a String";
}

var testCopy = test;
var testStr = test();
var testing = testCopy();
console.log(testCopy, testStr, testing);
```

Anonymous Function

지금껏 보아온 예제는 변수나 name으로 미리 선언된 Function들을 사용하는 방법이었으나. JavaScript와 같은 함수형 언어에서는 Anonymous(익명) Function을 사용할 수 있다.
Lambda Function 이라고도 부른다.

onmousedown 이벤트헨들러로써 즉석에서 function 표현식을 사용하는 예제

```
document.onmousedown = function(){
   console.log("You pressed the mouse button");
}
```

위 예제는 document에 mousedown 이벤트가 발생하면 "You pressed the mouse button" 이라는 console.log을 보여준다.
그러나 저 onmousedown 에 할당된 function은 다른곳에서는 참조할 수 없고 즉석에서 선언되어진 익명함수이다.

JavaScript의 Anonymous Function은 즉석에서 실행까지도 할 수가 있는데 이런 표현식을 좀 더 Heavy하게 사용하면 좀 더 다이나믹한 프로그래밍을 할수가 있다.

Anonymous Function을 선언 즉시 실행하는 방법

```
var x = (function () {return(5+6)})();
console.log(typeof(x));
console.log(x);
```

Function Scope

자바스크립트는 Block Scope를 지원하지 않는다.
오직 Function Scope만을 지원하며, Function 내부에 nested Function 또한 작성할 수 있다.

nested Function의 작성 예

```
var globalFunction = function(){
   console.log("I'm global");
}
var containerFunction = function(){
   var subFunction = function(){
      console.log("I'm Private");
      globalFunction();
   }

   globalFunction();
   subFunction();
}

containerFunction();
subFunction();
```

위 에제를 보면 globalFunction 과 containerFunction이 있는데 containerFunction의 내부를 보면 subFunction 이라는 nested Function이 선언이 되어있다.
containerFunction 에서는 global 영역에 있는 globalFunction 과 자신의 Function Scope 내에 있는 subFunction을 호출할 수 있다.
containerFunction은 globalFunction과 동일한 global 영역에 있기 때문에 어디서든 호출할 수 있지만,
subFunction은 containerFunction 내부에서만 영역을 갖기 때문에 global 영역에서 사용할 수 없다.
이는 JavaScript에서는 Function Scope 내에서는 지역적으로 다른 영역을 갖게 된다는것을 의미한다.

> subFunction 선언 시 주의점
> var subFunction 선언 시 var를 누락하게 되면 전역변수로 선언되니 유의하여 사용

다음의 예제를 실행해보자.

Block Scope를 지원하지 않는 JavaScript

```
var age = 34;
var someFunction = function(who) {
   if(who == "Koo") {
      var age = 29;
   } else {
      var age = 100;
   }

   for(var i=0; i<age; i++) { //29
       console.log(i); // 0 ~ 28
   }
};
someFunction("Koo");
console.log(age);   // 34
```

재미있는 것은 JavaScript는 이처럼 if, switch, for 문처럼 { } 으로 묶여진 block 영역을 별도의 Scope로 취급하지 않는다는 점이다.
오직 Function Scope만을 지원하기때문에 if문 block에서 사용되었던 age 변수가 for문에서도 그대로 유효하게 된다.
그러나 global 변수 age 는 someFunction 에서 재선언 되었지만 Function Scope에 의해 보호 받기 때문에 global 영역에서는 원래 값을 그대로 유지하고 있다.

Arguments

JavaSciprt의 매개변수는 JAVA나 C와 같은 정적타이핑을 하는 언어와는 달리 매개변수의 type을 별로 신경쓰지 않는다.
또한 매개변수의 갯수또한 함수 선언시에 에러를 발생시키지 않는다.
Function은 해당 Function 내부에서 유효한 특별한 객체인 arguments 를 제공하는데 이 arguments 객체는 유사배열 (정확히 말하면 배열이 아니다)형태로 해당 Function에 제공되어진 매개변수들을 저장하고 있다.

arguments 사용 예

```
var myFunc = function(a, b, c){
   document.writeln(a +","+ b +","+ c);  // 1, undefined, undefined
   document.writeln(myFunc.length);          // 3
   document.writeln(arguments.length);       // 1
};
myFunc(1);
```

myFunc는 a, b, c 총 3개의 매개변수를 받도록 선언되었지만 실제로 호출은 1 로 하나의 값만을 전달하여 호출되었다.
그럼에도 불구하고 JavaScript 인터프리터는 에러를 발생시키지 않는다.
실행결과는 a라는 이름으로 매개변수 1이 확인되었으나 b와 c는 값을 전달하지 않았으므로 undefined 라는 결과를 보여준다.
Function 객체인 myFunc 의 length 는 매개변수를 3개 받도록 선언하였기 때문에 3이 되며,
myFunc 안에서 arguments 의 length 는 실제 전달된 매개변수의 길이인 1이 설정된다.

Function의 유연한 매개변수 구조는 다음처럼 매개변수 길이에 제한을 두지 않는다.

```
myFunc(1,2,3,4,5,6,7,8,9,10);
```

전달된 매개변수를 모두 탐색하려면 유사배열객체인 arguments를 순서대로 검사하면 된다.

arguments의 내용을 모두 탐색하는 예제

```
var myFunc = function(a, b, c){
   console.log(a +","+ b +","+ c);  // 1, 2, 3
   console.log(myFunc.length);          // 3
   console.log(arguments.length);       // 10
   for(var i=0; i<arguments.length; i++){
      console.log(arguments[i] +", "); // 1,2,3,4,5,6,7,8,9,10
   }
};
myFunc(1,2,3,4,5,6,7,8,9,10);
```

클로져(Closure)

클로져는 JavaScript와 같은 함수형 언어에서 매우 유용한 기능이다.
쉽게 설명하자면 현재 Function Scope 내에서 참조할수 있는 범위에 존재하는 객체를 해당 Function Scope가 종료되어도 객체가 계속 참조됨을 의미한다. 아래 예제를 보면 좀 더 이해할 수 있을 것이다.

예제1

```
function sayHello(name) {
  var text = 'Hello ' + name;
  var sayAlert = function() { console.log(text); }
  sayAlert();
}
```

sayHello라는 함수내에는 sayAlert 이라는 sayHello 내부에서만 사용되는 Function이 정의 되었다.
그리고 sayAlert은 sayHello내에서 호출된다.
sayAlert 과 변수 text 는 논리적으로 같은 Scope에 위치하기 때문에 sayAlert 내부에서 변수 *text*를 사용할 수 있다.
여기서 이 sayHello라는 함수를 다음과 같이 바꿔보자.

예제2

```
function sayHello(name) {
  var text = 'Hello ' + name;
  var sayAlert = function() { console.log(text); }
  return sayAlert;
}

var say = sayHello("Daum");
say();
```

이제 sayAlert을 sayHello내부에서 직접사용하지 않고 return 값으로 반환을 하여 say 라는 함수로써 사용하고 있다.
이 경우에도 예제1과 마찬가지로 "Hello Daum" 이라는 값을 출력한다.
sayHello 함수의 호출이 종료되었음에도 sayHello 내부에서 선언된 text의 값이 보존되고 있는것을 볼 수 있다.
그리고 sayHello로부터 반환받은 함수 say의 내용을 검사해보자.

예제1

```
console.log(say);
```

```
function() {
    console.log(text);
}
```

say라는 함수에서는 sayHello에서 선언된것과 같이 console.log(text); 라는 코드만이 들어있을 뿐이다.
여기서 text라는 변수는 global 변수가 아님에도 불구하고 값이 할당되어 있으며 참조할 수 있었다는것을 볼 수 있다.
엄밀히 얘기하자면 text라는 변수는 sayHello 의 Function Scope에서 선언된 값이며 해당 Scope 내에서만 유효하지만
sayAlert에서 text라는 값을 사용하고 이를 return 시킴으로써 closure 가 생성된 것이다.

좀 더 자세한 예제는 다음 (http://www.javascriptkit.com/javatutors/closures2.shtml)을 참조하기 바란다.

Function의 this 그리고 call과 apply

- function에서 this는 해당 scorp 자체를 의미하는것으로 javascript를 OOP 방식으로 구현을 할때 많이 사용하는 변수입니다.
- this는 사용여부와 상황에 따라 변하기때문에 유의해서 사용해야합니다.
- this가 우리의 의도와는 다르게 변할수 있어 call과 apply 메소드를 이용해서 우리가 필요로 하는 scorp을 적용할수 있습니다.

```
var scorpObject = {
  count: 10,

  serviceList: ['blog', 'tistory', 'cafe'],

  objFunction: function() {
    console.log('this is objFunction');
  }
};

function globalScorp(arg1, arg2) {
  console.log(this);
  console.log(arg1);
  console.log(arg2);
}

document.onclick = globalScorp;
globalScorp(1, 2);
globalScorp.call(scorpObject, 3, 4);
globalScorp.apply(scorpObject, [5, 6]);
```

- call 메소드는 첫번째 인자로 scorp을 받고 두번째, 세번째....계속 인자를 전달할수 있고 해당 실행이 되는 함수의 첫번째, 두번째....인자로 전달됩니다.
- apply 메소드는 첫번째 인자로 scorp을 받고 두번째 인자로 배열을 받아서 해당 배열의 순서대로 실행되는 함수의 인자로 차례대로 전달됩니다.
- jQuery를 이용하는 방법

```
jQuery.proxy(globalScorp, scorpObject)();
jQuery.proxy(globalScorp, scorpObject, {"framework": "jQuery"}, 'argument2')();
```