[**배열의 합, 최대값,최소값 알고리즘**](http://webclub.tistory.com/60)

10개월 전 by 하기실음 관두등가 재야재희

배열 Study

배열의 합과 평균 구하기

**var** students = [80,90,100];  
**var** sum = 0;  
**var** avg = 0.0;  
  
**for** (**var** i=0; i < students.length; i++ ) {  
 sum += students[i];  
}  
  
console.log(sum);  
avg = sum / students.length;  
console.log(avg);

배열의 최대값 구하기

**var** arr = [20,77,50,40,55];  
**var** max = 0; // 최대값이 저장될 변수  
  
// max process  
**for**(**var** i = 0; i < arr.length; i++) {  
 // max값이 현재값보다 작다면  
 **if**(max < arr[i]) {  
 max = arr[i]; // breaking point test  
 }  
}  
console.log(max);

배열의 최소값 구하기

**var** arr2 = [20,77,50,40,10,55];  
**var** min = 100; // 최소값이 저장될 변수(단, 변수에 저장 될 값은 데이터의 가장 큰 값으로 정해야 한다.)  
  
// min process  
**for**(**var** i = 0; i < arr2.length; i++) {  
 // min값이 현재값보다 크다면  
 **if**(min > arr2[i]) {  
 min = arr2[i];  
 }  
}  
  
console.log(min);

출처: <http://webclub.tistory.com/60> [Web Club]

[**정렬 알고리즘(선택정렬) - 오름차순,내림차순**](http://webclub.tistory.com/61)

10개월 전 by 하기실음 관두등가 재야재희

전형적인 선택정렬 알고리즘

**var** sort = [33,22,11,55,44];  
**var** temp = 0,  
 i = 0,  
 j;  
  
**for**(; i < sort.length-1; i++) {  
 **for**(j = i + 1; j < sort.length; j++) {  
 **if**(sort[i] > sort[j]) {  
 temp = sort[i]; // 큰 값을 잠시 임시변수에 저장  
 sort[i] = sort[j]; // 작은 값을 앞으로 옮김  
 sort[j] = temp; // 임시변수에 넣어둔 것을 뒤로 옮김  
 }  
 }  
}  
  
// 0번째와 그 이후로 비교후 정렬  
// 1번째와 그 이후로 비교후 정렬...  
console.log(sort);

*※ 내림 차순은 부등호만 바꾸면... 참 쉽죠~ ! ㅋ*

출처: <http://webclub.tistory.com/61> [Web Club]

[**팩토리얼 함수**](http://webclub.tistory.com/117)

10개월 전 by 하기실음 관두등가 재야재희

팩토리얼(factorial) 함수

**1. 명령형 프로그래밍 방식의 팩토리얼**

**function** factorial\_step1(num) {  
 **var** val = 1,  
 i = 2;  
  
 **for**(; i <= num; i++) {  
 val = val \* i;  
 }  
 **return** val;  
}  
  
console.log(factorial\_step1(4))

**2. 재귀호출을 이용한 팩토리얼**

**function** factorial\_step2(num) {  
 **if**(num == 0) **return** 1;  
 **else return** num \* factorial\_step2(num -1);  
}  
console.log(factorial\_step2(5));

**3. 성능향상을 고려한 팩토리얼(캐시를 이용)**

**var** factorial = **function**() {  
 **var** cache = {'0' : 1};  
 **var** func = **function**(n) {  
 **var** result = 0;  
  
 **if**(**typeof**(cache[n]) === 'number') {  
 result = cache[n];  
 } **else** {  
 result = cache[n] = n \* func(n-1);  
 }  
 **return** result;  
 }  
 **return** func;  
}();  
  
console.log(factorial(5));  
console.log(factorial(4));

출처: <http://webclub.tistory.com/117> [Web Club]

[**재귀 함수(Recursive Function)**](http://webclub.tistory.com/72)

10개월 전 by 하기실음 관두등가 재야재희

재귀 함수

함수는 함수 안에서 자기 자신을 다시 호출하는 것을 재귀함수라고 한다.

재귀 함수의 필요성과 특징에 대해 알아보도록 하자.

재귀함수는 대개 factorial(계승) 함수로 설명들을 하고 있다.

다음 코드를 보자.

JavaScript

function fakeFactorial(n) {

var result = 1;

for(var i = 1; i <= n; i++) {

result \*= i;

}

return result;

}

console.log(fakeFactorial(3)); // 3 \* 2 \* 1 = 6

console.log(fakeFactorial(4)); // 4 \* 3 \* 2 \* 1 = 24

// 재귀를 이용한 factorial(계승) 함수

function factorial(n) {

if(n == 0 || n == 1) {

return 1;

}

return n \* factorial(n -1); // 자기 자신을 다시 호출

// 파라미터가 3이라면 3 \* factorial(2) \* factorial(1)

}

console.log(factorial(3);

// 삼항 연산자를 이용한 계승 재귀함수

function fact(n) {

return (n > 1) ? n \* fact(n -1) : 1;

}

console.log(fact(5)); // 120

위의 두번째 함수와 같이 자기 자신을 다시 호출하여 사용하는 것을 재귀 함수라고 한다.

또 다른 예제를 통해 사용법에 대해 좀 더 알아보자.

JavaScript

var sports = {

soccer : {member: 11, time: 90},

basketball : {member: 5, time: 48}

};

function showValues(sports) {

var type, obj, name;

for(type in sports) {

obj = sports[type];

for (name in obj) {

console.log(name + " : " + obj[name]);

}

}

}

showValues(sports);

/\*

\* 실행 결과 \*

member : 11

time : 90

member : 5

time: 48

\*/

위의 예제는 객체의 key : value를 출력하는 예제이다. sports란 객체에는 또 다른 객체를 포함하고 있는 계층적인 객체구조를 이루고 있다.

하지만 여기서 soccer 객체의 soccer.meber의 값이 11이 아닌 {step : {value: 11}}의 형태인 계층적 객체를 또 포함하고 있다면 이 객체의 프로퍼티를 열거하기 위해 또 한번의 for-in문을 작성해야 한다.

그런데 이런 계층적인 객체에 맞추어 코드를 유동적으로 작성할 수는 없다. 이럴 경우에 재귀함수를 사용하여 해결할 수가 있다.

재귀함수를 사용하여 계층적인 객체 프로퍼티를 열거

JavaScript

var sports = {

soccer : {step: {value : 11}, time: 90},

basketball : {member: 5, time: 48}

};

function showValues(sports) {

var type, obj;

for(type in sports) {

obj = sports[type];

typeof obj === "object" ? showValues(obj) : console.log(type + " : " + obj);

}

}

showValues(sports);

/\*

\* 실행 결과 \*

value : 11

time : 90

member : 5

time: 48

\*/

이와 같은 처리로 객체의 프로퍼티를 전부 열거할때까지 반복합니다.

따라서 객체의 계층에 제약을 받지 않고 처리할 수가 있습니다

또 다른 예제

JavaScript

var countDown = function countDown(num) {

console.log(num);

num--;

if(num <= 0 ) {

return false;

}

countDown(num);

};

countDown(5);

출처: <http://webclub.tistory.com/72> [Web Club]

## [자바스크립트 메서드 오버라이드와 오버로딩](http://webclub.tistory.com/404)

10개월 전 by 하기실음 관두등가 재야재희

JavaScript Method Override & Overloading

생성자 함수(클래스)를 이용하여 상속을 하다보면 종종 물려받은 부모의 기능을 그대로 사용하지 않고 재정의해야 할 경우가 발생할 수 있습니다.

이 경우를 메서드 오버라이드라고 하는데 이 글에서는 자바스크립트 오버라이드하는 방법에 대해 알아봅니다.

그리고 마지막에서는 오버로딩(overloading)에 대해서도 간단히 소개합니다.



메서드 오버라이드란?

메서드 오버라이드(override)는 자식 클래스에서 부모 클래스의 기능(method)을 재정의할 때 사용하믄 기능입니다.

오버라이드는 다음과 같은 경우에 주로 사용합니다.

* 부모 클래스의 기능을 사용하지 않고 자식 클래스에서 구현한 기능을 사용하고 싶은 경우
* 부모 클래스의 기능을 자식 클래스에서 확장하고 싶은 경우

그럼 이 두가지 경우에 대해 좀더 알아보도록 합니다.

부모 클래스의 기능을 자식 클래스에서 재정의

MyParent.prototype.부모메서드 = function() {}

MyChild.prototype.부모메서드 = function() {}

부모 클래스의 기능을 자식 클래스에서 재정의(오버라이드)하는 것은 간단합니다.

위의 문법처럼 부모 클래스의 기능 중 재정의(오버라이드)하고자 하는 기능을 자식 클래스에서 똑같은 이름으로 만들어 주면 됩니다.

다음의 예를 살펴봅니다.

javascript

// 부모 클래스

function MyParent() {

this.property1 = 'data1';

console.log('MyParent');

}

MyParent.prototype.method1 = function () {

console.log('property1 = ' + this.property1);

};

// 자식 클래스

function MyChild() {

console.log('MyChild');

}

// 부모 클래스 상속하기

MyChild.prototype = new MyParent();

// 생성자 설정

MyChild.prototype.constructor = MyChild;

/\*\*

\* ------------------

\* 메서드 오버라이드

\* ------------------

\*\*/

MyChild.prototype.method1 = function () {

console.log('프로퍼티 1은 = ' + this.property1 + ' 입니다.');

};

// 자식 인스턴스 생성

var child = new MyChild();

// 메서드 호출

child.method1();

// 콘솔창 결과

// MyParent

// MyChild

// 프로퍼티 1은 = data1입니다.

위 코드를 실행해 보면 더이상 부모 클래스의 method1() 메서드가 동작하지 않고 자식 클래스에서 작성한 method1() 메서드가 동작하는 것을 확인할 수 있을 것입니다.

이처럼 부모의 기능을 직접 수정하지 않고 부모의 기능을 재정의할 때 메서드 오버라이드를 사용합니다.

부모 클래스의 기능을 자식 클래스에서 확장

실무를 진행하다 보면 부모의 기능을 완전히 새롭게 재정의하는 것이 아니라 부모의 기능을 그대로 사용하면서 동시에 기능을 약간 추가하고 싶은 경우가 있습니다.

이런 경우에도 메서드 오버라이드를 활용할 수 있습니다.

Syntax

MyChild.prototype.부모메서드 = function ([param1, param2, ...]) {

부모클래스.prototype.부모메서드.call(this[,param1, param2, ...])

// 추가 확장 구문

}

기본은 오버라이드하고자 하는 부모 클래스의 기능(메서드)를 자식 클래스에서 동일한 이름으로 만들어 주는 것입니다.

이후 메서드 내부에는 함수객체(Fucntion)에서 제공하는 call() 메서드를 활용해 부모의 기능을 호출해 줍니다.

다음으로 추가하고자 하는 구문을 작성해 주면 됩니다.

이해를 돕기 위해 오버라이드를 활용한 기능 확장 예제를 살펴보도록 합니다.

javascript

// 부모 클래스

function MyParent() {

this.property1 = '문자열 데이터';

console.log('MyParent()');

}

MyParent.prototype.info = function () {

console.log('property1 = ' + this.property1);

};

// 자식 클래스

function MyChild() {

console.log('MyChild()');

this.property2 = '데이터이지롱';

}

// 부모 클래스 상속

MyChild.prototype = new MyParent();

// 생성자 설정

MyChild.prototype.constructor = MyChild;

/\*\*

\* -----------------------------------

\* 메서드 오버라이드를 활용한 기능 확장

\* -----------------------------------

\*\*/

MyChild.prototype.info = function () { // 확장할 기능인 info() 메서드를 자식 클래스에서 오버라이드 한다.

MyParent.prototype.info.call(this); // call() 을 이용해 부모 클래스의 info()메서드를 호출한다.

console.log('프로퍼티 두번째는 ' + this.property2); // 확장하려는 기능을 추가한다.

};

// 자식 인스턴스 생성

var child1 = new MyChild();

// 자식 정보 출력

child1.info(); // '프로퍼티 두번째는 데이터이지롱' 을 출력

위와 같이 메서드 오버라이드를 활용하면 부모의 기능을 확장할 수 있습니다.

자바스크립트에서 메서드 오버로딩

메서드 오버로딩(overloading)은 동일한 이름을 가진 여러 개의 메서드를 만든 후 매개변수 타입과 개수에 맞는 메서드가 자동으로 호출되는 기능을 말합니다.

하지만 아쉽게도 자바스크립트는 문법적을 오버로딩을 제공하지 않습니다.

대신 생성자 함수를 이용해 일반 객체지향 언어의 클래스를 흉내 내듯이 오버로딩 또한 흉내낼 수 있습니다.

**Syntax**

javascript

function 함수 또는 메서드이름() {

arguments 객체 활용

}

자바스크립트에서는 매개변수 정보를 담고 있는 arguments를 이용해 오버로딩을 흉내낼 수 있습니다.

오버로딩은 함수와 메서드에서 동일하게 사용할 수 있습니다.

다음의 예제를 이용해 좀더 알아봅니다.

이 예제는 여러 개의 숫자 값을 더하는 함수입니다.

javascript

function sum() {

var result = 0;

for (var i = 0; i < arguments.length; i++) {

result += arguments[i];

}

return result;

}

console.log(sum(10, 20)); // 30

console.log(sum(10, 20, 30)); // 60

console.log(sum(10, 20, 30, 40)); // 100

console.log(sum(10, 20, 30, 40, 50)); // 150

arguments 객체에 담긴 매개변수 값을 매개변수 개수만큼 반복해서 더해주고 있습니다.

오버로딩은 특히 여러분만의 라이브러리를 만들 때 유용하게 사용할 수 있을 것입니다.

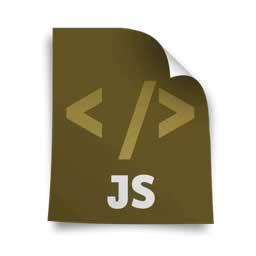
출처: <http://webclub.tistory.com/404> [Web Club]

## [함수 선언문과 함수 표현식에서의 세미콜론(;)](http://webclub.tistory.com/16)

10개월 전 by 하기실음 관두등가 재야재희

Semi Colon

일반적으로 코드를 작성할 때 함수 표현식에서는 ;(세미콜론)을 붙히고 함수 선언문으로 작성할 시에는 ;(세미콜론)을 붙이지 않는다.



이것은 하나의 관습이고 코드 작성시 권장되는 방식이다.

그리고 자바스크립트에서 세미콜론 사용을 강제하지는 않는다.

그 이유는 자바스크립트 인터프리터가 자동으로 세미콜론을 삽입시켜 주기 때문이다.

하지만 세미콜론에 신경을 쓰지 않는다면 소스 압축 배포를 하거나 디버깅을 할 시에 심각한 상황에 직면하게 될 것이다.

다음의 코드 예제를 살펴보자

// 'hello world!!!' 를 출력하려는 코드  
  
// 함수 표현식에 세미콜론을 사용하지 않음  
**var** test = **function**() {  
 **return** 300;  
} // 세미콜론 없음  
  
(**function**() {  
 console.log('즉시 실행함수(자가실행함수)로 바로 실행함')  
})();

이 코드를 실행하면 콘솔창의 내용을 출력하려는 의도와는 다르게 'number is not a function' 이라는 에러가 발생할 것이다.

그 이유는 자바스크립트 파서가  test 함수 표현식에서 세미콜론을 사용하지 않아서 6번째 줄의 중괄호인 }를 함수가 끝났다고 판단하지 않기 때문이다.

그리고 이후에 정의된 즉시 실행함수를 파서하면서 함수 표현식인 test를 호출하지도 않았음에도 괄호에 둘러싸인 익명함수(즉시실행함수로 사용하려고 한)때문에 test()를 호출해 버리기 된다.

즉, test()를 호출하면 return문의 300을 반환되고, 즉시 실행함수를 실행하려고 남겨둔 마지막 괄호()로 인해  300(); 형태로 또다시 함수를 호출하려고 시도한다.

그러나 300은 숫자일 뿐 함수가 아니므로 'number is not a function'라는 에러가 발생하는 것이다.

그렇기 때문에 함수 표현식에서는 특히 세미콜론 사용을 적극 권장하고 있다.

[출처 : 인사이드 자바스크립트]

출처: <http://webclub.tistory.com/16> [Web Club]

## [[자바스크립트] 이벤트 모델의 전파 과정](http://webclub.tistory.com/186)

10개월 전 by 하기실음 관두등가 재야재희

이벤트 전달을 막는 4가지 방법

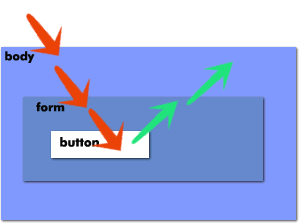
이벤트 모델은 DOM의 구조를 두 번 순회합니다.

이벤트가 호출되면 최상위 루트에서 자식 엘리멘트들을 타고 이벤트 타깃까지 내려가는 캡쳐 단계가 일어나고 다시 반대로 이벤트의 타킷으로부터 최상위 루트까지 올라가는 버블 단계가 일어납니다.

이러한 이벤트 전파 과정을 통해서 예상치 못한 결과가 발생하는 경우가 있는데 예를 들어 부모 요소와 자식 요소 두 곳에 click 이벤트를 생성한 후 자식 요소를 클릭하여 이벤트를 호출할 때 클릭하지 않은 부모 요소에서 클릭이 발생하게 됩니다.

이런 의도치 않은 반응이 일어나 문제가 발생되는 것은 이벤트 모델의 전파과정 중 버블링 단계에서 이벤트를 발생하기 때문입니다.

하지만 event 객체에 stopPropagation() 메소드를 호출하면 이 단계가 진행되지 않도록 사전에 차단할 수 있습니다.



이벤트를 중단할 때 사용되는 방법에 대한 정리는 아래와 같습니다.

event.preventDefault();  
// 현재 이벤트의 기본 동작을 중단합니다.  
  
  
event.stopPropagation();  
// 현재 이벤트가 상위로 전파되지 않도록 중단합니다.  
  
  
event.stopImmediatePropagation();  
// 현재 이벤트가 상위뿐 아니라 현재 레벨에 걸린 다른 이벤트도 동작하지 않도록 합니다.  
  
  
**return false**;  
// jQuery를 사용한 코드내에서는 위의 preventDefault, stopPropagation 두개 모두를 수행한 것과 같고,  
  
// jQuery를 사용하지 않고 return false 를 작성할 경우에는  
// event.preventDefault() 와 같습니다.

이벤트 전파란 무엇인가?!

위의 메소드를 알아보기 전에 이벤트가 전파,전달되는 것이 무엇인지에 대해 다음의 예제를 통해 확인해 보도록 하겠습니다.

위의 결과 창에서 각 영역을 클릭하면 결과 콘솔에 나타나는 결과가 나타납니다.

가장 최상위의 부모 DIV 를 클릭했을 때는 DIV 만을 클릭한 결과만 나타나지만, 가장 아래에 있는 자식 span 을 클릭할 경우에는 span 뿐만 아니라 p 와 div 의 클릭 이벤트까지 모두 포함하여 동작하는 것을 확인할 수 있습니다.

이것이 이벤트가 전파,전달되는 것입니다.

이러한 이벤트 전달을 차단하기 위해 사용되는 코드가 앞서 설명한 세 가지 코드들입니다.

event.stopPropagation() - 이벤트 전파를 차단

stopPropagation() 은 이벤트가 상위 DOM 으로의 전달,전파를 하지 않도록 하는 코드입니다.

다음의 예제에서는 span 클랙 이벤트에 event.stopPropagation() 메소드를 추가해서 상위 DOM 으로 이벤트가 전달되지 않도록 했습니다.

결과는 span 을 클릭했을 경우에는 span 이벤트만 동작하지만, p 를 클릭했을 때에는 p, div 가 모두 호출되게 됩니다

event.preventDefault() - 이벤트의 기본 동작을 차단하는 메소드

다음의 예제는 a 태그에 걸린 이벤트 동작을 수행하면서도 a 태그의 본래 목적을 지닌 기능인 URL 이동하는 기능까지 모두 수행하고 있는 것을 확인 하실 수 있습니다.

이 말인 즉슨, 이벤트 동작과 더불어 a 태그가 지닌 기본동작 이벤트도 함께 수행하고 있다는 것입니다.

event.stopPropagation() 메소드로는 이벤트의 전달을 차단만 할 뿐 기본동작인 URL 이동 기능을 막지 못합니다.

다음의 예제에서는 stopPropagation() 메소드 대신 preventDefault() 메소드로 변경하여 실행해 보도록 하겠습니다.

동작을 확인해 보면 a 태그의 기본 기능 동작이 실행되지 않는 것을 확인해 보실 수 있습니다.

프로젝트를 수행하다 보면 일반적으로 DOM 에서 본인이 원하는 이벤트 동작만을 수행하고 싶을 때에는 stopPropagation() 과 preventDefault() 두 코드를 사용해야 합니다.

하지만 이벤트 동작을 수행함에 있어 jquery 코드내에서 사용한다면 return false 를 사용하여 두 가지의 이 메소드를 수행한 것과 같은 결과를 나타내게 할 수 있습니다.

stopImmediatePropagation() - 같은 DOM 에 걸린 다른 이벤트까지도 전파를 차단

앞서 살펴 본 stopPropagation() 메소드는 이벤트가 상위 DOM 으로의 전달을 차단하는 코드입니다

그렇다면 해당 DOM 에 여러 개의 이벤트가 함께 걸려있었다면 어떻게 될까?!

다음의 예제에서는 span 클릭 이벤트에 stopPropagation() 메소드를 추가하여 상위 전파를 차단하도록 했습니다.

다른 점이 있다면 span 에 클릭 이벤트를 두 개를 주었다는 점입니다.

그렇다면 이렇게 한 개의 DOM 영역에 여러 개의 이벤트를 걸게되면 stopPopagation 은 어떻게 동작하게 될까요?

결과는 span 을 클릭했을 때, p, div 까지 이벤트가 전달되지는 않지만 span 에 걸린 두 개의 이벤트가 모두 동작하는 것을 볼 수 있습니다.

다음의 코드는 span 에 걸린 두 개의 이벤트 중 첫번째 이벤트에 stopPropagation() 대신 stopImmediatePropagation() 을 사용하도록 하겠습니다.

결과를 확인해 보면 이전 코드에서는 span 에 걸려 있던 두 개의 이벤트가 모두 호출되어 동작되었지만, 지금의 코드는 즉시 전파가 중단되어 첫번째 이벤트만 호출되어 동작되는 것을 확인하실 수 있습니다.

이벤트 캡처링(event capture)

이벤트 버블링(event bubble)

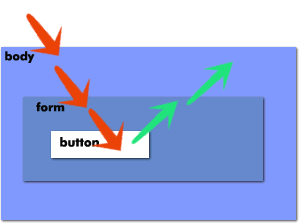
출처: <http://webclub.tistory.com/186> [Web Club]

## [Event flow & Event Handling](http://webclub.tistory.com/116)

10개월 전 by 하기실음 관두등가 재야재희

이벤트 흐름과 이벤트 핸들링

이 글에서는 이벤트 흐름과 핸들링에 대해 알아봅니다.



이벤트 흐름 단계

|  |  |
| --- | --- |
| 단 계 | 설   명 |
| 캡처링 단계(capture phase) | 이벤트가 문서의 루트객체인 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 |

Dom Level 2 - 이벤트 핸들링

addEventListener(), removeEventListener()

elm.addEventListener( string type, Function handler, useCapture)

addEventlistener 메소드의 매개변수

|  |  |
| --- | --- |
| 매개변수 | 설명 |
| type | 핸들링하고 싶은 이벤트 'load', 'click', 'mouseleave' 등... |
| handler | 지정한 타입의 이벤트가 현재 요소로 전달되면 호출된 이벤트 핸들링 함수.   핸들러가 호출되면 Event 객체가 핸들러로 전달된다. |
| userCapture | true 설정시 이벤트 핸들러는 캡처링 단계에서만 호출,   false 라면 타켓 단계와 버블링 단계에서 호출된다.  보통  false를 사용한다 |

Event 객체의 멤버

|  |  |
| --- | --- |
| 속성 및 메소드 | 설         명 |
| type | 발생한 이벤트의 타입. ex) 'click', 'keydown' |
| target | 원래 이벤트가 발생한 요소이며 currentTarget과는 다를 수 있다. |
| currentTarget | 이벤트가 처리되고 있는 요소로서, 현재 실행되고 있는 핸들러가 등록된 요소를 반환한다. 만약 캡처링 또는 버블링하는 동안 이벤트가 처리된다면 이 속성의 값은 원래 이벤트가 발생한 target 속성과는 다를 수가 있다.  또한 이 값은 this와도 다를 수 있다. div.onclick = handler 인 경우는 핸들러에서 사용하는 this가 가리키는 객체가 명확하지만 div.addEventListener(type, handler, userCapture) 로 등록된 핸들러에서의 this는 대상객체가 명확하지 않다. 대부분의 브라우저에서는 addEventListener 방식으로 등록되더라도 이벤트 핸들러가 콜백될때 this의 값을 현재 이벤트 핸들러가 등록되는 요소를 가리키게 하지만 이는 표준이 아니다.  이러한 경우 currentTarget은 정확히 현재 이벤트 핸들러가 등록된 요소를 가리킨다. |
| bubbles | 버블링 이벤트인지를 나타낸다. 버블링 이벤트라면 true, 아니면 false |
| cancelable | 기본 액션이 존재하는지에 대해 알려준다.   즉, preventDefault() 메소드로 취소될 수 있는 이벤트인지를 알려주는 것이다. |
| eventPhase | 이벤트 전파가 어떤 단계인지를 알려주는 상수값, 다음 상수 중 하나를 값으로 취한다   Event.CAPTURING\_PHASE   Event.AT\_TARGET   Event.BUBBLING\_PHASE |
| preventDefault | 브라우저가 이벤트와 관련된 기본 액션을 수행하지 않게 한다. |
| stopPropagation | 캡처링, 타켓팅, 버블링 경로상에서 현재 핸들링되고 있는 객체 이상으로 이벤트가 전달되지 않게 한다 |

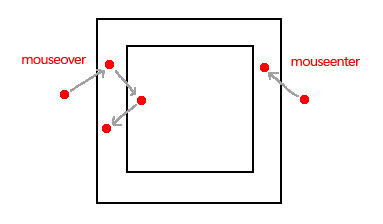
출처: <http://webclub.tistory.com/116> [Web Club]

## [mouseover와 mouseenter 이벤트 차이](http://webclub.tistory.com/456)

10개월 전 by 하기실음 관두등가 재야재희

mouseover vs mouseenter

이 포스팅에서는 마우스오버 이벤트와 마우스엔터 이벤트의 차이점에 대해 알아봅니다.



mouseover 와 mouseenter의 차이점은 이벤트 버블링에 있습니다.

다음의 소스뷰를 살펴보도록 합니다.

위의 영역의 오렌지와 핑크에 마우스를 올렸다 내렸다하면 마우스오버와 마우스엔터 이벤트를 발생시킬 것입니다.

mouseover 이벤트는 이벤트 버블링이 적용되는 이벤트이기 때문에 내부의 div 태그 안에 들어가도 이벤트를 또 다시 발생시킵니다.

반면에, mouseenter 이벤트는 문서 객체의 안에 있는지 외부에 있는지만 따지기 때문에 이벤트가 한 번만 발생하게 됩니다.

버블링에 대해 좀더 자세히 살펴보려면 아래의 참고 링크를 살펴보시기 바랍니다.

### Related Info

1. [이벤트 모델의 전파과정](http://webclub.tistory.com/186)
2. [Event flow & Event Handling](http://webclub.tistory.com/116)

출처: <http://webclub.tistory.com/456> [Web Club]

[**keyup 이벤트 알아보기**](http://webclub.tistory.com/457)

10개월 전 by 하기실음 관두등가 재야재희

이벤트 객체 - keyup

이 글에서는 keyup event 에 대해 알아보도록 하겠습니다.



keyup event

다음의 예제는 글쓰기 입력창과 같은 곳에서 글자 입력수에 따라서 입력 가능한 글자수를 동적으로 글자 수를 세어주는 예제입니다.

아래의 textarea 에 한글이나 영문을 입력해 보시기 바랍니다.

위의 입력받은 키 이벤트는 keyup 을 사용했습니다.

위와 같은 것을 구현하고자 할 경우에는 keypress 이벤트는 한글로 사용할 수 없습니다.

왜 keypress 가 아닌 keyup 이벤트를 사용하는 것일까요?

영문을 입력하면 keypress 이벤트가 사용하기 좋지만 한글은 keypress 이벤트를 지원하지 않습니다.

따라서 keypress 이벤트는 배제해야 합니다.

그리고 keydown 이벤트가 아니가 keyup 이벤트를 사용해야 합니다.

이는 키보드 이벤트가 어떠한 순서로 발생하는지를 안다면 이해하기 수월할 것입니다.

키보드 이벤트는 다음과 같은 순서로 진행됩니다.

1. 사용자가 키보드를 누른다.
2. keydown 이벤트가 발생한다.
3. 글자가 입력된다.
4. keypress 이벤트가 발생한다.
5. 사용자가 키보드에서 손을 뗀다.
6. keyup 이벤트가 발생한다.

따라서 keydown 이벤트가 발생한 순간에는 글자가 입력돼 있지 않습니다.

여기서는 입력한 글자 수를 표시해야 하므로 keyup 이벤트를 사용했습니다.

출처: <http://webclub.tistory.com/457> [Web Club]

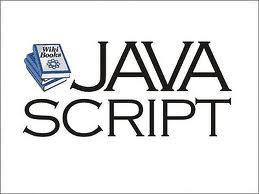
## [(function (window, document, undefined) {})(window, document); 의 진짜 의미](http://webclub.tistory.com/350)

10개월 전 by 하기실음 관두등가 재야재희

즉시실행 함수 패턴

이 포스팅은 원문을 번역한 글을 차용하였습니다.

[What (function (window, document, undefined) {})(window, document); really means]



IIFE(immediately-invoked function expression)

요즘 저는 아래와 같은 IIFE(immediately-invoked function expression, 즉시실행함수 표현식) 형식에 대해 많은 질문을 받았습니다.

JavaScript

(function (window, document, undefined) {

//

})(window, document);

그래서 이와 관련된 포스팅을 하겠습니다.

저는 이 포스팅에서 다양한 내용을 함께 다뤄보려 합니다. 차례대로 살펴보도록 하겠습니다.

Scope

JavaScript는 function scope를 가지기 때문에 이를통해 private scope(지역 범위)를 가질 수 있습니다.

예를 들면:

JavaScript

(function (window, document, undefined) {

var name = ‘Todd’;

})(window, document);

console.log(name); // name is not defined, it’s in a different scope

간단합니다. (역자주: 위 코드에서 name은 함수 안에서만 유효한 scope을 가집니다.)

How it works(동작 방법)

평범한 함수 표현식은 아래처럼 생겼습니다.

(역자주: 함수 표현식은 해당 코드블럭이 JavaScript 엔진의 parser에 의해 실행코드로서 해석되며 실행에 따른 결과값을 가지거나 특정 변수에 할당된 값으로 존재)

JavaScript

var logMyName = function (name) {

console.log(name);

};

logMyName('Todd');

위 함수 표현식을 통해서 우리는 우리가 원하는 시점에 함수를 호출할 수 있습니다. 또한 어디에서든 우리가 원하는 scope을 제공할 수 있습니다.

“IIFE”가 ()로 감싸진 이유는 즉시실행함수 표현식이기 때문입니다.

이런 함수는 런타임 시에 즉시 호출되며 한번 실행 후에는 우리가 다시 호출할 수 없습니다.

JavaScript

var logMyName = (function (name) {

console.log(name); // Todd

})('Todd');

위 예제에서 logMyName 변수에 할당했었던 코드모양을 살펴보시기 바랍니다.

JavaScript

(function () {

})();

만약 아래 코드처럼 괄호쌍 하나가 빠진다면 동작하지 않습니다.

(역자주: 왜냐하면 JavaScript 엔진의 parser는 function 키워드가 처음으로 나오면 함수 선언문으로 인식하기 때문입니다.)

JavaScript

function () {

}();

하지만 JavaScript에서 이런 모습의 코드를 강제로 동작시키기 위한 몇 가지 트릭이 존재합니다.

이 방법은 JavaScript 엔진의 parser가 코드 앞의 `!`문자를 보고 함수 선언이 아닌 표현으로 인식하게 하는 방법입니다.

JavaScript

!function () {

}();

마찬가지로 아래처럼 비슷하게 변형된 방법들도 있습니다.

JavaScript

function () {

}();

-function () {

}();

~function () {

}();

하지만 저는 이런 방법을 사용하지 않습니다.

관련된 내용은 @mariusschulz의 [Disassembling JavaScript’s IIFE Syntax](https://blog.mariusschulz.com/2016/01/13/disassembling-javascripts-iife-syntax)를 확인해보시면 IIFE 문법과 변형된 모습들에 대해서 자세하게 살펴볼 수 있습니다.

Arguments

이제 우리는 IIFE가 동작하는 방법을 알았습니다.

물론 IIFE에도 arguments를 전달할 수 있습니다.

JavaScript

(function (window) {

})(window);

이 코드가 어떻게 동작할까요?

마지막 부분의 (window);를 통해서 함수가 실행되며 이 시점에 우리는 window 객체를 넘겨주게 됩니다.

넘겨 받은 객체는 함수 안에서도 window라는 이름으로 정의되어 있습니다.

여러분은 window라는 파라미터 이름은 의미가 없으며 다른 이름으로 정의해도 되지 않냐고 따질 수 있습니다. 하지만 이 window라는 이름을 잘 사용해 보도록 합니다.

그래서 무엇을 해볼까요? document 객체를 넘겨봅시다.

JavaScript

(function (window, document) {

// we refer to window and document normally

})(window, document);

지역 변수들은 글로벌 변수보다 빠르게 해석될 수 있습니다.

하지만 엄청난 스케일의 코드가 아닌 이상 우리는 눈에띄는 속도 향상은 느낄 수는 없습니다.

그래도 글로벌 변수들을 많이 참조한다면 충분히 고려해볼만 합니다.

What about undefined?

ECMAScript 3에 의하면 undefied는 mutable 합니다.

이는 undefiend = true; 처럼 값을 재할당 할수 있다는 것을 의미합니다.

놀랍죠? 감사하게도 ECMAScript 5의 strict 모드 `user strict;`를 활용하면 parser가 error를 뱉어줍니다.

하지만 그 이전에 우리 스스로 IIFE를 아래 코드처럼 보호해야 합니다.

JavaScript

(function (window, document, undefined) {

})(window, document);

이렇게 하면 누군가 아래처럼 실수를 하더라도 문제가 없습니다.

IIFE argument로 세번째 인자를 넘기지 않았기 때문에 함수 scope 안의 undefined는 실제 undefined 의미로서 동작하게 되는 것입니다.

JavaScript

undefined = true;

(function (window, document, undefined) {

// undefined is a local undefined variable

})(window, document);

Minifying

만약 함수 안으로 지역 변수들이 pass in 되면 변수명 자체는 중요하지 않고 우리가 원하는 새로운 이름으로 호출 할 수 있습니다.

아래의 코드는 minifying을 거쳐

JavaScript

(function (window, document, undefined) {

console.log(window); // Object window

})(window, document);

아래 코드로 바뀝니다.

JavaScript

(function (a, b, c) {

console.log(a); // Object window

})(window, document);

여러분이 참조하는 라이브러리들, window, document 객체들이 멋지게 minified 된 모습을 상상해 보시기 바랍니다.

이뿐만 아니라 jQuery 객체 역시 $ 형태로 전달 할 수 있으며 무엇이든지 lexical scope 안에서 전달 가능합니다.

JavaScript

(function ($, window, document, undefined) {

// use $ to refer to jQuery

// $(document).addClass('test');

})(jQuery, window, document);

(function (a, b, c, d) {

// becomes

// a(c).addClass('test');

})(jQuery, window, document);

이는 우리가 jQuery.noConflict(); 처럼 호출할 필요가 없고 지역 모듈로서 $에 jQuery를 할당해서 사용할 수 있다는 의미입니다.

JavaScript의 scope와 global/local 변수들에 대해서 공부한다면 보다 도움이 될 것입니다.

성능이 좋은 minifier라면 당신의 코드에 있는 undefined 단어를 샅샅이 뒤져서 `c`와 같은 이름으로 치환할 것입니다.

여기서 undefined라는 이름 자체는 치환에 있어서 큰 관계가 없습니다.

우리가 알아야 하는 것은 참조하는 객체(Object)가 undefined라는 것이며 특별한 의미를 가지는 것은 아닙니다.

undefined는 선언은 되었지만 값이 할당된 적이 없다는 의미의 JavaScript에서 제공하는 데이터 형입니다. (역자주: 이 부분은 minifier가 단순히 undefined 이름을 보고 치환하는 것이 아니라 레퍼런스 관계를 고려해서 치환한다는 의미 같습니다.)

Non-browser global environments

Node.js 같은 도구들 때문에 브라우저는 항상 전역 객체가 아닙니다.

여러 환경을 고려해서 작업해야 한다면 IIFE를 생성할 때 브라우저는 항상 글로벌 객체가 아니기 때문에 신경쓰일 수 있습니다.

때문에 저는 IIFE 코드 형태를 아래 모습을 기본으로 생성하는 습관이 있습니다.

JavaScript

(function (root) {

})(this);

브라우저에서는 전역 환경(전역객체)은 window 객체에 레퍼런스 되어 있기 때문에 window를 꼭 넘길 필요는 없고 this를 통해 간결하게 넘길 수 있습니다.

저는 root이라는 네이밍을 선호하는데 이는 브라우저는 물론 브라우저 환경이 아닌 곳에서도 root로 참조가 가능하기 때문입니다.

만약 여러분이 universal solution(오픈소스 프로젝트를 생성할 때 저는 이 방식을 항상 사용합니다.)에 관심이 있다면 아래와 같은 UMD wrapper방식이 있습니다.

JavaScript

(function (root, factory) {

if (typeof define === 'function' && define.amd) {

define(factory);

} else if (typeof exports === 'object') {

module.exports = factory;

} else {

root.MYMODULE = factory();

}

})(this, function () {

//

});

이 방법은 엄청 세련된 방법입니다.

argument로 넘어온 함수가 factory 함수로 호출되고 있습니다.

이런 방식을 통해서 우리는 환경에 따라 적절하게 외부로 할당할 수 있습니다.

브라우저에서는 root.MYMODULE = factory(); 방식으로 우리의 IIFE 모듈을 할당할 수 있고 Node.js의 경우에는 module.exports 방식으로, requireJS 방식(typeof define === ‘function’ && define.amdresolves가 true인 경우)으로 할당할 수 있습니다.

이와 관련된 자세한 이야기는 조금 다른 내용이지만, [UMD repo](https://github.com/umdjs/umd)도 확인해볼 것을 추천합니다.

### 참고 문헌

1. [원문 IIFE](https://toddmotto.com/what-function-window-document-undefined-iife-really-means/)
2. [번역문 IIFE](https://medium.com/@jungseobshin/%EB%B2%88%EC%97%AD-what-function-window-document-undefined-window-document-really-means-b92b0b40304a#.hjnfcjoqx)

출처: <http://webclub.tistory.com/350> [Web Club]

## [if, else 문 처리를 더 간결하게 표현하기](http://webclub.tistory.com/187)

10개월 전 by 하기실음 관두등가 재야재희

코드의 간결성

일반적으로 if 문으로 처리하는 방식은 다음과 같습니다.

JavaScript

if( aaa ) {

return bbb;

} else {

return ccc;

}



다음은 삼항 연산자를 이용한 패턴입니다.

JavaScript

return ( aaa ) ? bbb : ccc;

다음은 && 연산자를 이용한 패턴입니다.

JavaScript

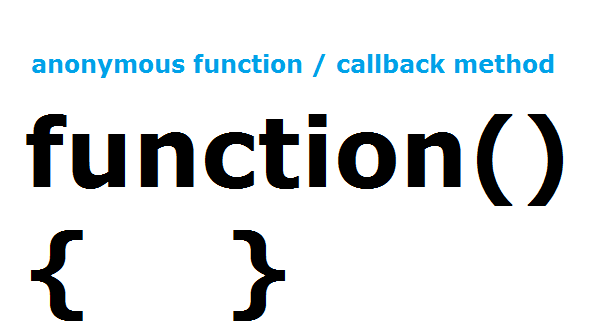
return aaa && bbb || ccc;

Jaehee's WebClub

출처: <http://webclub.tistory.com/187> [Web Club]

## [자바스크립트 함수 패턴](http://webclub.tistory.com/141)

10개월 전 by 하기실음 관두등가 재야재희



즉시 실행 함수( Self-Excuting Anonymous Functions)

*/\*\*  
 \* 작성 방법  
 \* (function(){})() 또는 (function(){}())  
 \*/*(**function** (global, undefined) {  
 // 독립된 공간(Scope)에 코드 작성  
  
})(window !== undefined ? window : **this**);

함수는 별도의 독립된 공간(Scope)을 형성한다

*/\*\*  
 \* 오늘날 자바스크립트는 전역(Window) 공간을 오염시키지 않는 것이 중요하다.  
 \* 필요하다면 명시적으로 `global`을 붙여서 외부 모듈과 공유하도록 한다.  
 \* 가급적 `var`를 붙여서 지역 변수 생성  
 \*/*(**function** (global, undefined) {  
  
 // 지역 내에서만 사용 가능한 로컬 함수  
 **var** \_localFn = **function**() {  
 // ....  
 };  
  
 // 전역 변수 gFn에 로컬 함수 \_localFn 참조 (전역 공유)  
 global.gFn = \_localFn;  
  
})(window !== undfined ? window : **this**);

함수 내부에서 this 가 참조하는 대상

// 일반함수  
**function** normalFn() {  
 console.log(**this**); // 함수가 실행되는 컨텍스트(문맥)  
};  
  
// 일반함수 실행  
normalFn(); // this === window  
document.onclick = normalFn; // this === document  
document.querySelector('a.ext').onmouseenter = normalFn; // this === a.ext  
  
  
// 생성자 함수  
**function** ConstructorFn() {  
 console.log(**this**); // 생성자 함수 내 this는 생성된 객체를 가리킴.  
}  
  
// new 연산자를 붙여 생성자 함수 실행 - 객체 생성  
**var** instance = **new** ConstructorFn; // this === instance 객체

생성자 함수(클래스) 사용 예

Navigation 객체 생성자와 프로토타입

// 객체 생성하는 함수 : 생성자 함수 (Constructor Function)  
**function** Navigation(el) {  
 **this**.el = document.querySelector(el);  
 **this**.children = **this**.el.querySelectorAll('a');  
 **this**.childrenLength = **this**.children.length;  
};  
  
// 생성자 함수의 프로토타입 : 생성되는 객체의 원형(Prototype)  
Navigation.prototype = {  
 nextLink : **function**() {  
 console.log(**this**.el, '다음 링크 활성화');  
 },  
 prevLink : **function**() {  
 console.log(**this**.el, '이전 링크 활성화');  
 },  
 goToLink : **function**(num) {  
 console.log(num + '링크 활성화');  
 },  
 playRollingLinks : **function**() {  
 console.log('링크 롤링 시작');  
 },  
 stopRollingLinks : **function**() {  
 console.log('링크 롤링 중지');  
 }  
};  
  
// Navigation 인스턴스 객체 생성  
**var** hNav = **new** Navigation('header nav');  
**var** aNav = **new** Navigation('aside nav');  
**var** mNav = **new** Navigation('main nav');  
**var** fNav = **new** Navigation('footer nav');  
  
// Navigation 인스턴스 객체 메소드 활용  
hNav.nextLink();  
mNav.prevLink();

출처: <http://webclub.tistory.com/141> [Web Club]

## [eval() 사용과 문제점 : #eval() is evil](http://webclub.tistory.com/512)

10개월 전 by 하기실음 관두등가 재야재희

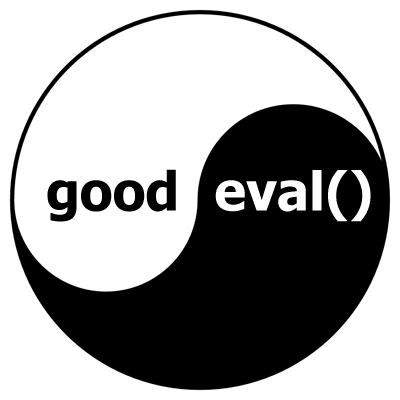
eval() is evil

코드에서 eval() 을 발견하면 'eval() 은 사악하다(eval() is evil)'라는 주문을 기억하라!

이 함수는 임의의 문자열을 받아 자바스크립트 코드로 실행합니다.

만약 문제의 코드를 사전에 알 수 있다면(즉 런타임에 결정되는 게 아니라면) eval()을 쓸 필요가 없습니다.

코드가 런타임에 동적으로 생성된다면 대개 eval() 없이 목표를 달성할 수 있는 더 나은 방법이 존재합니다.



eval() 피하기

예컨대 동적인 프로퍼티에 접근하려고 할때는 대괄호 표기법이 더 간단하고 좋은 방법이 될 것입니다.

JavaScript

// 안티 패턴

var property = 'name';

console.log(eval('obj.' + property));

// 권장 패턴

var property = 'name';

console.log(obj[property]);

eval() 사용은 보안 문제와도 관련됩니다. 누군가 함부로 손댄(예를 들어 네트워크에서 가져온) 코드를 실행시키게 될 수도 있기 때문입니다.

Ajax 요청으로 받아온 JSON 응답을 다룰 때 이런 안티 패턴을 흔히 볼 수 있습니다.

보안과 유효성을 보장하기 위해서는 브라우저의 내장 메서드를 사용하여 JSON 응답을 파상하는 것이 좋습니다.

JSON.parse() 를 내장 지원하지 않느 브라우저에서는 JSON.org 의 라이브러리를 사용할 수 있습니다.

또 하나, setInterval() 과 setTimeout() 그리고 Function() 생성자에 문자열을 넘기는 것도 eval() 을 사용하는 것과 상당히 비슷하기 때문에, 역시 사용을 자제해야 합니다.

자바스크립트가 전달받은 문자열을 프로그래밍 코드로 평가하여 실행하는 것은 마찬가지입니다.

JavaScript

// 안티 패턴

setTimeout('myFunc()', 3000);

setTimeout('myFunc(1, 2, 3)', 1000);

// 권장 패턴

setTimeout(myFunc, 3000);

setTimeout(function () {

myFunc(1, 2, 3);

}, 1000)

new Function() 생성자를 사용하는 것도 eval() 과 비슷하기 때문에 신중하게 접근해야 합니다.

강력한 도구가 될 수도 있지만 대부분 제대로 사용되지 않습니다.

반드시 eval() 을 사용해야만 한다면 그 대신에 new Function() 의 사용을 고려해볼 수 있습니다.

new Function() 안에서 평가되는 코드는 지역 함수의 유효범위 안에서 실행되기 때문에 코드 내에서 var 로 선언된 변수들이 자동으로 전역 변수가 되지 않는 약간의 장점이 있습니다.

자동으로 전역 변수가 되지 못하도록 막기 위해 eval() 호출을 즉시실행 함수로 감싸는 방법도 있습니다.

다음의 예제를 살펴보도록 합니다. 여기서 전역 변수로 남아 네임스페이스를 어지립히는 것은 un 뿐입니다.

JavaScript

console.log(typeof un); // 'undefined' 가 기록된다.

console.log(typeof deux); // 'undefined' 가 기록된다.

console.log(typeof trois); // 'undefined' 가 기록된다.

var jsstring = 'var un = 1; console.log(un);';

eval(jsstring); // 1 이 기록된다.

jsstring = 'var deux = 2; console.log(deux);';

new Function(jsstring)(); // 2 가 기록된다

jsstring = 'var trois = 3; console.log(trois);';

(function () {

eval(jsstring);

})(); // 3 이 기록된다.

// 전역 스코프에서 변수들을 체크해 본다.

console.log(typeof un); // number 가 기록된다.

console.log(typeof un); // undefined 가 기록된다.

console.log(typeof un); // undefined 가 기록된다.

eval() 과 Function 생성자 간의 또 다른 차이는 eval() 은 유효범위 체인에 간섭을 일으킬 수 있지만 Function 은 좀더 봉인되어 있다는 점이 다릅니다.\

Function 은 어디서 실행시키든 상관없이 자신의 유효범위를 가지고 있기 때문에 지역 변수를 덜 오염시킵니다.

다음 예제는 eval() 은 그 자신의 바깥쪽 유효범위에 접근하고 수정을 가할 수 있는 반면, Function 은 그럴 수 없습니다.(Function 을 사용하는 것과 new Function 은 동일하다)

JavaScript

(function () {

var local = 1;

eval('local = 3; console.log(local);'); // 3 이 기록된다.

console.log(local); // 유효범위를 오염시켜 3 이 기록된다.

})();

(function () {

var local = 1;

Function('console.log(typeof local);')(); // undefined 가 기록된다.

})();

출처: <http://webclub.tistory.com/512> [Web Club]

## [자기 자신을 정의하는 함수(lazy function)](http://webclub.tistory.com/514)

10개월 전 by 하기실음 관두등가 재야재희

자기 자신을 재정의하는 함수

함수는 동적으로 정의할 수 있고 변수에 할당할 수 있습니다.

새로운 함수를 만들어 이미 다른 함수를 가지고 있는 변수에 할당하면, 새로운 함수가 이전 함수를 덮어 쓰게 됩니다.

어떤 면에서는 이전의 함수 포인터가 새로운 함수를 가리키도록 재사용하는 것이다.

이러한 일을 이전 함수의 본문 내에서 할 수도 있습니다.

이 경우 함수는 자기 자신을 새로운 구현으로 덮어쓰고 재정의할 수 있습니다.

아마도 실제보다 더 복잡하게 들릴 텐데 간단한 아래의 예제를 통해 알아보도록 합니다.



lazy function definition(게으른 함수 선언)

다음의 예제를 살펴보도록 합니다.

javascript

var scareMe = function () {

console.log('Boo!!!');

scareMe = function () {

console.log('Boooo~~~~~~~~~');

};

};

// 자기 자신을 정의하는 함수를 사용

scareMe(); // Boo!!! 가 기록

scareMe(); // Boooo~~~~~~~~~ 가 기록

이 패턴은 함수가 어떤 초기화 준비 작업을 단 한번만 수행할 경우에 유용합니다.

불필요한 작업을 반복할 이유가 없기 때문에 함수의 일부는 더 이상 쓸모가 없을 것이고, 이런 경우에 함수가 자기 자신을 재정의하여 구현 내용을 갱신할 수 있습니다.

간단히 말해서 재정의된 함수의 작업량이 적기 이 패턴은 애플리케이션의 성능에 확실히 도움이 됩니다.

이 패턴은 '게으른 함수 선언(lazy function definition)' 이라고도 불리는데 그 이유는 최초 사용 시점 전까지 함수를 완전히 정의하지 않고 있다가 호출된 이후에는 더 게을러져서 더 적게 일하기 때문이다.

이 패턴의 단점은 자기 자신을 재정의한 이후에는 이전에 원본 함수에 추가했던 프로퍼티들을 모두 찾을 수 없게 된다는 점입니다.

또한 함수가 다른 이름으로 사용된다면, 예를 들어 다른 변수에 할당하거나, 객체의 메서드로써 사용되면 재정의된 부분이 아니라 원본 함수의 본문이 실행됩니다.

scareMe() 함수를 다음과 같이 일급 객체로 사용하는 예를 살펴보도록 합니다.

javascript

var scareMe = function () {

console.log('Boo!!!');

scareMe = function () {

console.log('Boooo~~~~~~~~~');

};

};

// 1. 새로운 프로퍼티를 추가한다.

scareMe.property = 'properly';

// 자기 자신을 재정의한 함수를 사용한다.

console.log(scareMe.property);

scareMe();

scareMe();

// 2. 다른 이름으로 할당

var prank = scareMe;

// 3. 메서드로 사용하기 위해 정의한다.

var spooky = {

boo: scareMe

};

// 다른 이름으로 할당한 이름으로 호출한다.

prank(); // Boo!!! 가 기록

prank(); // Boo!!! 가 기록

console.log(prank.property); // properly 가 기록

spooky.boo(); // Boo!!! 가 기록

spooky.boo(); // Boo!!! 가 기록

console.log(spooky.boo.property); // properly 가 기록

// 위에서 다른 이름으로 할당, 메서드로 할당한 이후에 자기자신을 재정의한 함수를 사용해 본다.

scareMe(); // Boooo~~~~~~~~~ 가 기록

scareMe(); // Boooo~~~~~~~~~ 가 기록

console.log(scareMe.property); // undefined 가 기록

예제에서 보는 것처럼, 함수가 새로운 변수에 할당되면 예상과 달리 자기 자신을 정의하지 않습니다.

prank() 가 호출될 때마다 콘솔에 "Boo!!" 가 출력됩니다.

또한 전역 scareMe() 함수를 덮어썼는데도 prank() 자신은 여전히 property 프로퍼티를 포함한 이전의 정의를 참조합니다.

spooky 객체의 boo() 메서드로 함수가 사용될 때에도 똑같은 일이 일어납니다.

이 모든 호출들은 계속해서 전역 scareMe() 포인터를 덮어 씁니다.

따라서 마지막에 전역 scareMe() 가 호출되었을 때 비로소 "Boooo~~~~~~~~~" 를 출력하도록 갱신된 본문이 처음르로 제대로 실행됩니다.

또한 scareMe.property 도 더이상 참조할 수 없게 됩니다.

출처: <http://webclub.tistory.com/514> [Web Club]

## [메모제이션(Memoization) 패턴](http://webclub.tistory.com/515)

10개월 전 by 하기실음 관두등가 재야재희

함수 프로퍼티 - Memoization Pattern(메모제이션 패턴)

함수는 객체이기 때문에 프로퍼티를 가질 수 있습니다.

사실 함수는 처음부터(생성될 때부터) 프로퍼티와 메서드를 가지고 있습니다.

예를 들어, 각 함수는 어떤 문법으로 생성하는 자동으로 length 프로퍼티를 가지고 있습니다.

이 프로퍼티는 함수가 받는 인자의 개수를 값으로 가지게 됩니다.

javascript

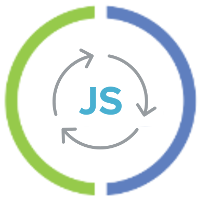
function func(a, b, c) {}

console.log(func.length); // 3 이 기록된다.

언제든지 함수에 사용자 정의 프로퍼티를 추가할 수 있습니다.

함수에 프로퍼티를 추가하여 결과(반환 값)을 캐시하면 다음 호출 시점에 복잡한 연산을 반복하지 않을 수 있습니다.

이런 활용 방법을 메모제이션(Memoization) 패턴이라고 합니다.



다음 예제에서는 myFunc 함수에 cache 프로퍼티를 생성하고, 이 프로퍼티는 일반적인 프로퍼티처엄 myFunc.cache 와 같은 형태로 접근할 수 있습니다.

cache 프로퍼티는 함수로 전달된 param 매개변수를 키로 사용하고 계산의 결과를 값으로 가지는 객체(해시)입니다.

결과 값은 필요에 따라 복잡한 데이터 구조로 저장할 수도 있습니다.

javascript

var myFunc = function (param) {

if (!myFunc.cache[param]) {

var result = {};

// 비용이 많이 드는 로직 수행..

myFunc.cache[param] = result;

}

return myFunc.cache[param];

};

// 캐시 저장 공간

myFunc.cache = {};

위 코드는 myFunc 함수가 param 이라는 단 하나의 매개변수를 받는다고 가정합니다.

이 매개 변수는 문자열과 같은 윈시 데이터 타입이라고 가정합니다.

만약 더 많은 매개변수와 더 복잡한 타입을 갖는다면 일반적으로 직렬화하여 해결할 수 있습니다.

예를 들어, 객체 인자를 JSON 문자열로 직렬화하고 이 문자열을 cache 객체에 키로 사용할 수 있습니다.

javascript

var myFunc = function () {

var cachekey = JSON.stringify(Array.prototype.slice.call(arguments)),

result;

if (!myFunc.cache[cachekey]) {

result = {};

// 비용이 많이 드는 로직 수행..

myFunc.cache[cachekey] = result;

}

return myFunc.cache[cachekey];

};

// 캐시 저장 공간

myFunc.cache = {};

직렬화하면 객체를 식별할 수 없게 되는 것을 주의해야 합니다.

만약 같은 프로퍼티를 가지는 두개의 다른 객체를 직렬화하면, 이 두 객체는 같은 캐시 항목을 공유하게 될 것입니다.

출처: <http://webclub.tistory.com/515> [Web Club]

## [커링 함수 (Curring function)](http://webclub.tistory.com/6)

10개월 전 by 하기실음 관두등가 재야재희

커링(Curring)

커링은 수학자 하스켈 커리로 부터 유래된 이름이고 함수를 변형하는 과정을 말한다.

함수의 전달인자 몇 개를 미리 채움으로써 더 간단한 함수를 만드는 방법입니다.

커링은 인도 음식인 커리와 아무런 관계가 없습니다.

커링은 수학자 하스켈 커리(Haskell Curry)로 부터 유래되었습니다.(하스켈 프로그래밍 언어가 그의 이름에서 따온 것이다.)

커링은 이 변형 방법의 원래의 발명가인 모세 쉔핀켈 이름을 따서 쉔필켄이라고도 부릅니다.

커링을 사용하는 경우

*어떤 함수를 호출할 때 대부분의 매개 변수가 항상 비슷하다면 커링을 사용할 만한 후보라고 할 수 있다.*

*매개변수 일부를 적용하여 새로운 함수를 동적으로 생성하면 이 동적 생성된 함수는 반복적으로 사용되는 매개변수를 내부적으로 저장하여, 매번 인자를 전달하지 않아도 원본함수가 기대하는 기능을 채워 놓게 될 것이다.*



간단한 커링함수 소스코드의 예제

javascript

// curring function

function curring(num) {

// 클로저로 생성된 공간

return function(add) {

return num + add;

};

}

var add = curring(5);

var result = add(5);

console.log(result);

다른 함수형 언어에세는 커링 기능이 언어 자체에 내장되어 있어 모든 함수가 기본적으로 커링됩니다.

자바스크립트에서는 아래의 add() 함수 수정하여 부분 적용을 처리하는 커링 함수로 만들 수 있습니다.

다음의 예제를 살펴봅니다.

javascript

// 커링된 add()

// 부분적인 인자의 목록을 받는다.

function add(x, y) {

var old\_x = x, old\_y = y;

if (typeof old\_y === 'undefined') { // 부분적인 적용

return function (new\_y) {

return old\_x + new\_y;

}

}

// 전체 인자를 적용

return x + y;

}

console.log(add(3, 5));

console.log(add(10)(10));

var add10 = add(10);

console.log(add10(20));

위 코드에서 처음 add() 를 호출할 때, add 가 반환하는 내부 함수에 클로저를 만듭니다.

클로저는 원래의 x, y 값을 비공개 변수인 old\_x, old\_y 에 저장하도록 합니다.

첫 번째 변수인 old\_x 는 내부 함수가 실행될 때 사용됩니다.

부분적인 적용이 없고 x, y 둘다 전달되었다면, 이 함수는 단순히 이 둘을 더하게 됩니다.

add()는 설명을 위해 필요 이상으로 장황하게 구현되어 있습니다.

더 간단한 버전은 다음 예제에서 살펴볼 수 있습니다.

다음 예제는 old\_x, old\_y 가 없습니다.

원래의 x 는 암묵적으로 클로저에 저장되어 있고 이전 예제에서 new\_y 라는 새로운 변수를 만들었던 것과는 달리 지역 변수 y를 재사용합니다.

javascript

// 커링된 add()

// 부분적인 인자의 목록을 받는다.

function add(x, y) {

if (typeof y === 'undefined') { // 부분적인 적용

return function (y) {

return x + y;

}

}

// 전체 인자를 적용

return x + y;

}

console.log(add(3, 5));

console.log(add(10)(10));

var add10 = add(10);

console.log(add10(20));

출처: <http://webclub.tistory.com/6> [Web Club]

## [Global NameSpace Function(범용 네임스페이스 함수)](http://webclub.tistory.com/516)

10개월 전 by 하기실음 관두등가 재야재희

네임스페이스 패턴 생성시 유용한 범용적인 네임스페이스 함수

프로그램의 복잡도가 증가하고 코드의 각 부분들이 별개의 파일로 분리되어 선택적으로 문서에 포함하게 되면, 어떤 코드가 특정 네임스페이스나 그 내부의 프로퍼티를 처음으로 정의한다고 가정하기가 위험합니다.

네임스페이스에 추가하려는 프로퍼티가 이미 존재할 수도 있고 따라서 내용을 덮어쓰게 될 지도 모릅니다.

그러므로 네임스페이스를 생성하거나 프로퍼티를 추가하기 전에 먼저 이미 존재하는지 여부를 확인하는 것이 최선입니다.

다음의 예제를 살펴보도록 합니다.

javascript

// 위험하다.

var MYAPP = {};

// 개선안

if (typeof MYAPP === 'undefined') {

MYAPP = {};

}

// 또는 더 짧게 작성할 수 있다.

var MYAPP = MYAPP || {};

이렇게 추가되는 확인 작업 때문에 상당량의 중복 코드가 생겨날 수 있습니다.

예를 들어 MYAPP.modules.module2 를 정의하려면, 각 단계의 객체와 프로퍼티를 정의할 때마다 확인 작업을 거쳐야 하므로 코드가 세 번 중복됩니다.

따라서 네임스페이스 생성의 실제 작업을 맡아 줄 재사용 가능한 함수를 만들어두면 편리합니다.

다음은 네임스페이스 함수를 구현한 예제입니다.

다음과 같은 방식은 해당 네임스페이스가 존재하면 덮어쓰지 않기 때문에 기존 코드를 망가뜨리지 않습니다.

var MYAPP = MYAPP || {};

MYAPP.namespace = function (ns\_string) {

var parts = ns\_string.split('.'),

parent = MYAPP,

i;

// 처음에 중복되는 전역 객체명은 제거한다.

if (parts[0] === 'MYAPP') {

parts = parts.slice(1);

}

for (i = 0; i < parts.length; i += 1) {

if (typeof parent[parts[i]] === 'undefined') {

parent[parts[i]] = {};

}

parent = parent[parts[i]];

}

return parent;

};

이 코드는 다음 모든 예에서 사용할 수 있습니다.

javascript

var module2 = MYAPP.namespace('MYAPP.modules.module2');

console.log(module2);

console.log(module2 === MYAPP.modules.module2); // true 가 기록

// 첫 부분의 'MYAPP' 을 생략하고도 사용할 수 있다.

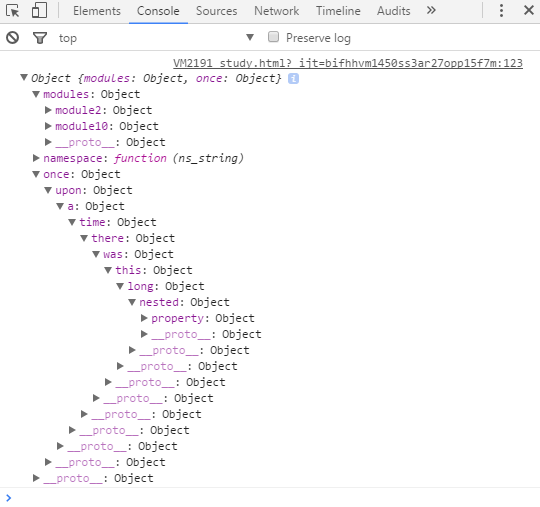
MYAPP.namespace('modules.module10');

// 매우 긴 네임스페이스를 만들어 본다.

MYAPP.namespace('once.upon.a.time.there.was.this.long.nested.property');

console.log(MYAPP);

다음의 그림은 위 예제를 요소 검사기로 MYAPP 을 실제로 검사했을 때의 모습입니다.



출처: <http://webclub.tistory.com/516> [Web Club]

[**자바스크립트의 객체 상수**](http://webclub.tistory.com/527)

9개월 전 by 하기실음 관두등가 재야재희

객체 상수

자바스크립트에는 상수가 없지만 대다수 최신 브라우저 환경에서는 const 문을 통해 상수를 생성할 수 있습니다.

흔히 사용되는 우회적인 방법으로, 명명 규칙을 사용하여 값이 변경되지 말아야 하는 변수명을 모두 대문자로 쓰기도 합니다.

이 규칙은 실제로 자바스크립트 내장 객체에서도 사용됩니다.

javascript

Math.PI; // 3.1415922653589793

Math.SQRT2; // 1.4142135623720951

Number.MAX\_VALUE; // 1.7976931348623157e+308

사용자 정의 상수에도 동일한 명명 규칙을 적용하여, 생성자 함수에 스태틱 프로퍼티로 추가할 수 있습니다.

javascript

// 생성자

var Widget = function () {

// 생성자 구현내용...

};

// 상수

Widget.MAX\_HEIGHT = 320;

Widget.MAX\_WIDTH = 480;

객체 리터럴로 생성한 객체에도 동일한 명명 규칙을 적용할 수 있습니다.

즉, 대문자로 쓴 일반적인 프로퍼티를 상수로 간주하는 것입니다.

실제로도 값이 변경되지 않게 하고 싶다면, 비공개 프로퍼티를 만든 후, 값을 설정하는 메서드(setter) 없이 값을 반환하는 메서드(getter)만 제공하는 방법도 고려해 볼 만 합니다.

그러나 대부분의 경우에는 단순히 명명 규칙만으로도 충분하기 때문에 이 방법은 좀 과도할 수 있습니다.



이어질 예제는 다음과 같은 메서드를 제공하는 범용 constant 객체를 구현한 것입니다.

* set(name, value) : 새로운 상수를 정의한다.
* isDefined(name) : 특정 이름의 상수가 있는지 확인한다.
* get(name) : 상수의 값을 가져온다.

이 예제에서는 상수 값으로 원시 데이터 타입만 허용됩니다.

또한 선언하려는 상수의 이름이 toString 이나 hasOwnProperty 등 내장 프로퍼티의 이름과 겹치지 않도록 보장하기 위해 hasOwnProperty() 를 사용한 별도의 확인 작업을 거칩니다.

마지막으로 모든 상수의 이름 앞에 임으로 생성된 접두어를 붙입니다.

javascript

var constant = function () {

var constants = {},

ownProp = Object.prototype.hasOwnProperty,

allowed = {

string : 1,

number : 1,

boolean: 1

},

prefix = (Math.random() + "\_").slice(2);

return {

set : function (name, value) {

if (this.isDefined(name)) {

return false;

}

if (!ownProp.call(allowed, typeof value)) {

return false;

}

constants[prefix + name] = value;

return true;

},

isDefined : function (name) {

return ownProp.call(constants, prefix + name);

},

get : function (name) {

if (this.isDefined(name)) {

return constants[prefix + name];

}

return null;

}

}

}();

아래에서 테스트 해보도록 합니다.

javascript

// 이미 정의되었는지 확인한다.

console.log(constant.isDefined('maxwidth')); // false 가 기록

// 정의한다.

console.log(constant.set('maxwidth', 480)); // true 가 기록

// 다시 정의를 시도해 본다.

console.log(constant.isDefined('maxwidth')); // true 가 기록

// 값은 그대로인가?

console.log(constant.get('maxwidth')); // 480 이 기록

출처: <http://webclub.tistory.com/527> [Web Club]

## [splice 와 indexOf 를 활용 예제](http://webclub.tistory.com/564)

7개월 전 by 하기실음 관두등가 재야재희

splice 와 indexOf 이용한 배열 원소를 검색하여 없애거나 바꾸기



javascript

var animals = ['dog', 'cat', 'seal', 'walrus','lion','cat'];

// 배열에서 원소를 제거

var deleteElement = animals.splice(animals.indexOf('walrus'), 1);

console.log(deleteElement); // 반환값은 제거한 원소

console.log(animals); // 원래 배열은 제거한 원소를 제외한 원소를 반환

// 새로운 원소를 삽압하기

animals.splice(animals.lastIndexOf('cat'), 1, 'monkey','puma');

console.log(animals);

var animals2 = ['cat', 'walrus', 'lion', 'cat'];

// 새로운 원소 삽입

animals2.splice(-2, 1, 'monkey'); // 끝에서 부터 2번째 인덱스, 1개의 원소를 제거하고 'monkey' 을 삽입

console.log(animals2);

// 잘라내는 원소의 개수가 주어지지 않는다면 시작 위치부터 배열 끝까지 모든 원소가 제거된다.

var animals3 = ['cat', 'walrus', 'lion', 'cat'];

animals3.splice(2); // 2 인덱스부터 배열 끝가지 모든 원소를 제거

console.log(animals3);

var animals4 = ['cat', 'dog', 'lion', 'cat'];

// 두 번째 원소를 다른 두 개의 값으로 치환

animals4.splice(3,1,'zebra','elephant'); // 3 번째 인덱스인 cat, 원소 1개를 제거하고 두 개의 원소를 삽입한다.

console.log(animals4);

// 반복문과 splice 를 사용한 원소 제거와 치환

var charSets = ['ab', 'bb', 'cd', 'ab', 'cc', 'ab', 'dd', 'ab'];

// 원소 치환하기

while (charSets.indexOf('ab') != -1) {

charSets.splice(charSets.indexOf('ab'), 1, '\*\*');

}

console.log(charSets);

// 원소 제거하기

while (charSets.indexOf('\*\*') != -1) {

charSets.splice(charSets.indexOf('\*\*'),1);

}

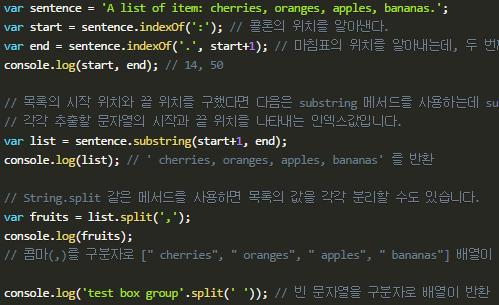
console.log(charSets);

출처: <http://webclub.tistory.com/564> [Web Club]

## [Search String(문자열 검색하기)](http://webclub.tistory.com/568)

7개월 전 by 하기실음 관두등가 재야재희

indexOf, lastIndexOf 를 이용한 문자열 검색



문자열 부분 문자열 검색하기

대상 문자열 안에서 특정한 부분 문자열이 있는지를 찾고자 하는 경우에는 String 객체의 내장 메서드 indexOf 를 사용하면 부분 문자열의 위치를 알 수 있습니다.

javascript

var testValue = 'This is the Cookbook';

var subValue = 'Cookbook';

var iValue = testValue.indexOf(subValue);

console.log(iValue);

// String 객체의 indexOf 메서드는 인덱스 혹은 부분 문자열의 첫 번째 글자가 있는 위치를 나타내는 숫자를 반환하고, 문자열에서 첫 번째

// 문자의 인덱스는 0 입니다.

// 부분 문자열이 대상 문자열 안에 있는지 없는지 확인하기 위해서는 반환되는 값이 -1 인지 살펴보면 됩니다

if (iValue != -1) {

console.log('찾고자 하는 부분 문자열이 있습니다.');

}

// indexOf 메서드는 두 개의 인자를 갖습니다.

// 첫 번째 인자는 찾고자 하는 부분 문자열이고, 두 번째 인자는 검색을 시작할 위치의 인덱스 값입니다.

// 두 번째 인자는 생략 가능합니다.

var str = 'Have you had a dinner?';

var subStr = 'dinner';

var iVal = str.indexOf(subStr, 8);

console.log(iVal); // 15 => 부분 문자열의 인덱스 15를 반환

// indexOf 메서드는 왼쪽에서 오른쪽으로 탐색합니다. 그러나 때로는 문자열을 오른쪽에서 왼쪽으로 탐색하고 싶을 때가 있습니다.

// 이럴 경우에는 String 객체의 또 다른 메서드인 lastIndexOf 를 사용하면 됩니다.

// lastIndexOf 는 가장 마지막에 나타난 부분 문자열의 인덱스 위치를 반환합니다.

var testStr = 'I have been there';

var findStr = 'been';

var findIndex = testStr.lastIndexOf(findStr);

console.log(findIndex); // 7

// lastIndexOf 의 두 번째 인수는 indexOf 에서와 마찬가지로 생략할 수 있으며, 검색 시작 위치를 전달 받습니다.

// 검색을 시작할 위치는 오른쪽 부터 셈한 값입니다.

console.log(testStr.lastIndexOf('have', 5)); // 2

console.log(testStr.lastIndexOf('there', 7)); // -1 반환, 찾지 못함

// 7인 인덱스 위치에서 오른쪽 부터 검색하므로 찾지 못함

// 즉,인덱스 7 의 위치에서 'I have b' 의 b 위치부터 오른쪽에서 왼쪽으로 검색하기 때문에 찾고자 하는 문자열이 없어서 -1 을 반환

문자열에서 부분 문자열을 추출하는 방법

여러 개의 문장으로 이루어진 문자열이 있고, 그 중 하나의 문장에 어떤 목록이 포함되어 있다고 가정합니다.

그리고 목록은 콜론(:)으로 시작하고 마침표(.)로 끝납니다.

이때 이러한 문자열을 추출하고 하는 경우에는 indexOf 메서드를 사용하여 콜론의 위치를 알아낸 후,

다시 indexOf 메서드를 사용하여 콜론 다음에 오는 마침표의 위치를 찾습니다.

그리고 나서 두 개의 위치를 substring 메서드와 함께 사용하면 문자열을 추출할 수 있습니다.

javascript

var sentence = 'A list of item: cherries, oranges, apples, bananas.';

var start = sentence.indexOf(':'); // 콜론의 위치를 알아낸다.

var end = sentence.indexOf('.', start+1); // 마침표의 위치를 알아내는데, 두 번째 인자로 콜론이 위치한 인덱스 위치 다음부터 찾을 인덱스를 인자로 정의한다.

console.log(start, end); // 14, 50

// 목록의 시작 위치와 끝 위치를 구했다면 다음은 substring 메서드를 사용하는데 substring 메서드의 첫 번째 인자와 두 번째 인자는

// 각각 추출할 문자열의 시작과 끝 위치를 나타내는 인덱스값입니다.

var list = sentence.substring(start+1, end);

console.log(list); // ' cherries, oranges, apples, bananas' 를 반환

// String.split 같은 메서드를 사용하면 목록의 값을 각각 분리할 수도 있습니다.

var fruits = list.split(',');

console.log(fruits);

// 콤마(,)를 구분자로 [" cherries", " oranges", " apples", " bananas"] 배열이 반환됩니다.

console.log('test box group'.split(' ')); // 빈 문자열을 구분자로 배열이 반환

문자열이 존재하는지 또는 빈 문자열인지 확인하기

변수가 정의되어 있으며 문자열이 들어 있는지 또는 비어 있는지를 확인하고자 하는 경우에는 다음과 같이 작성할 수 있습니다.

javascript

// 변수가 존재하고 문자열이 0 보다 큰 길이를 가지고 있으면 참

var unknownVal = '';

if ( (typeof unknownVal != 'undefined') && (typeof unknownVal.valueOf() == 'string') && (unknownVal.length > 0) ) {

console.log('문자열이 존재합니다.');

}

typeof 연산자, valueOf 메서드, String 객체의 length 속성을 사용하면 변수가 정의되어 있으며 문자열이 들어 있는지 혹은 비어 있는지를 확인하는 조건문을 작성할 수 있습니다.

(valueOf 메서드는 모든 자바스크립트 객체가 공유하는 메서드입니다.)

length 를 사용하면 문자열의 길이를 알 수 있고, 문자열 변수가 빈 문자열인지 아닌지를 확인할 수 있습니다(빈 문자열이면 length 가 0)

그러나 문자열을 다룰 때 변수가 정의되어 있지 않다면 undefined 란 오류가 발생합니다.

그래서 변수가 정의되어 있는지 확실하지 않다면 문자열 길이를 테스트할 수도 없습니다.

따라서 문자열의 길이를 테스트하기에 앞서 변수가 존재하는지에 대한 테스트 과정을 수반할 필요가 있습니다.

이런 특성 덕분에 정의되지 않은 변수의 속성에 접근하는 오류를 예방할 수 있습니다.

그리고 또 하나 고려해야할 점은 정의된 변수가 문자열이 아닐 경우를 대비해야 합니다.

예를 들어 변수가 숫자라면 이런 경우 숫자에는 length 속성이 없기 때문에 조건문은 거짓이 됩니다.

그러나 변수의 값이 String 객체라면 어떨까요?

변수의 타입을 정확하게 알 수 없다면 문자열 길이를 테스트하기에 앞서 명시적으로 변수의 데이터 타입이 'string' 인지 테스트해야 합니다.

테스트가 성공하면 예상했던 대로 길이가 0 보다 큰 문자열을 얻을 수 있습니다.

하지만 변수가 문자열 리터럴이 아닌 String 객체라면 typeof 연산자는 'string' 대신 'object'를 반환할 것입니다.

이것이 바로 또 다른 자바스크립트 메서드인 valueOf 를 같이 사용해야 하는 이유입니다.

valueOf 메서드는 모든 자바스크립트 객체에서 사용가능하며, 객체가 무엇이든 원시 자료형을 반환합니다.

Number, String, Boolean 에 대해서는 각각의 원시 자료형을 반환하고 Function 에 대해서는 함수의 내용을 반환합니다.

변수가 이미 문자열 리터럴일 경우에는 valueOf 메서드를 사용하면 임시로 String 래퍼객체로 감싼 후 valueOf 메서드를 실행하므로

이 경우에도 반환되는 값은 문자열 리터럴입니다.

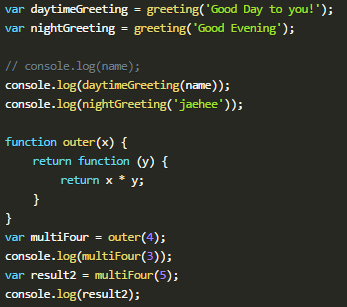
그래서 위의 조건문은 변수가 정의되었는지 확인하고, 정의되어있다면 valueOf 메서드를 사용하여 String 객체 혹은 문자열 리터럴인지 확인한 후 문자열의 길이가 0 보다 큰자 확인하는 것입니다.

일반적으로 빈 문자열인지 아닌지 확인하기 위해 변수가 정의되었는지 여부와 문자열 길이만 테스트합니다.

출처: <http://webclub.tistory.com/568> [Web Club]

## [배열 문자열을 역순으로 반환하는 함수 및 상태를 기억하는 함수 작성하기](http://webclub.tistory.com/569)

7개월 전 by 하기실음 관두등가 재야재희



재귀 알고리즘 구현하기

배열을 재귀적으로 탐색하고 배열 문자열을 역순으로 반환하는 함수를 구현하고자 하는 경우에는 재귀적으로 함수 리터럴을 사용할 수 있습니다.

javascript

var reverseArray = function (x, index, str) {

return index == 0 ? str : reverseArray(x, --index, (str += x[index] + " "));

};

var arr = ['apple', 'orange', 'peach', 'lime'];

var str = reverseArray(arr, arr.length, "");

console.log(str);

상태를 기억하는 함수 작성하기

전역 변수를 사용하지 않고, 함수를 호출할 때마다 같은 데이터를 재전송하지 않으며, 정적 데이터를 기억(저장)할 수 있는 함수를 작성하고자 싶은 경우가 있을 수 있습니다.

이런 경우에는 한 개 이상의 인수를 받는 외부 함수를 만들고 그 외부 함수 내부(안)에 한 개 이상의 인수를 받는 함수를 작성하되 내부 함수의 인자와 부모(외부) 함수의 인자를 모두 사용하면 됩니다.

외부 함수는 내부 함수를 반환하고 이 값을 변수에 할당하도록 합니다.

function greeting(greet) {

function addName(name) {

return greet + ' ' + name;

}

return addName;

}

var daytimeGreeting = greeting('Good Day to you!');

var nightGreeting = greeting('Good Evening');

// console.log(name);

console.log(daytimeGreeting(name));

console.log(nightGreeting('jaehee'));

function outer(x) {

return function (y) {

return x \* y;

}

}

var multiFour = outer(4);

console.log(multiFour(3));

var result2 = multiFour(5);

console.log(result2);

Jaehee'

출처: <http://webclub.tistory.com/569> [Web Club]

## [trim 메서드 : 문자열 양끝 공백 제거하기](http://webclub.tistory.com/572)

7개월 전 by 하기실음 관두등가 재야재희

trim 메서드

출처: <http://webclub.tistory.com/572> [Web Club]