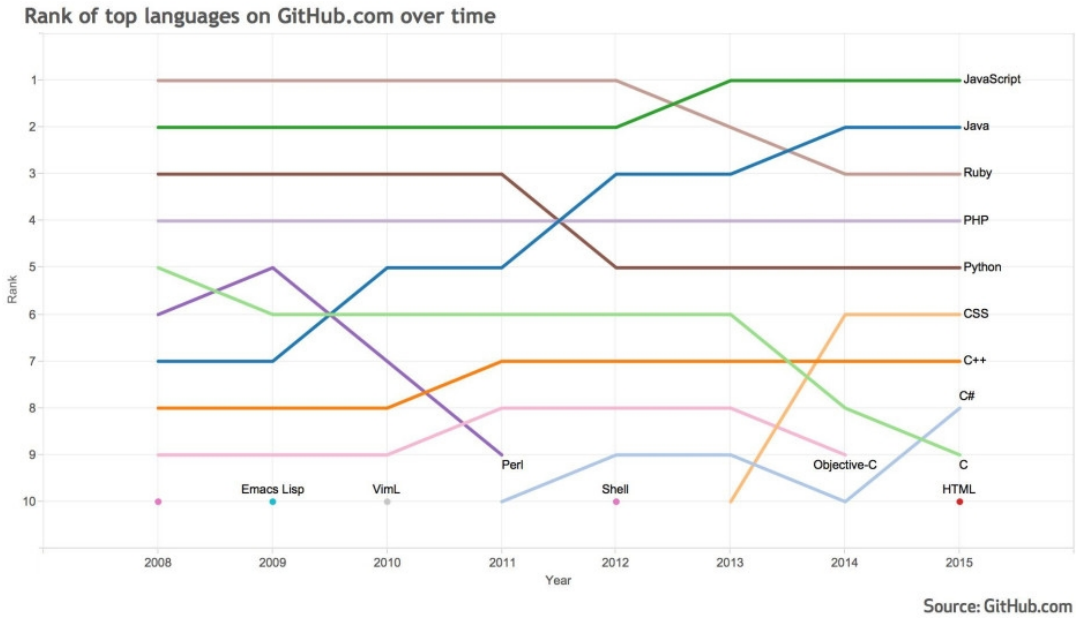
softcontext@gmail.com



1. Why JavaScript First?

노드는 자바스크립트에게 브라우저의 굴레에서 벗어나서 독립적으로 프로그래밍할 수 있는 환경을 제공한다. 자바스크립트를 평소 자주 사용하지 않던 개발자라면 자바스크립트의 핵심을 점검하고 노드나 클라이언트 사이드 프레임워크들을 접하는 것이 도움이 될 것이다.

자바스크립트는 세상에서 가장 오해를 많이 받아온 언어중에 하나지만 살아남아 웹의 세계를 평정했다. 자세히 들어다 보면 가장 우아한 언어중 하나라고 할 수 있는 자바스크립트를 살펴보자.

자바스크립트는 1995 년 브렌든 아이크가 넷스케이프 커뮤니케이션사에 근무하던 시절에 만들었다. 1996 년에 공식 ECMA 명세가 제출되었고 마이크로소프트와 넷스케이프는 주력 브라우저에 자바스크립트를 구현해 넣었다. 20 년이 넘는 기간 동안 계속해서 ECMAScript 명세가 갱신되었다.

클라이언트 측에서는 jQuery 와 MVC 프레임워크인 Angular 자바스크립트 라이브러리가 등장하기에 이르렀다.

자바언어의 인기와 더불어 ECMAScript 라는 원래 이름 보다는 JavaScript 라는 이름으로 더 많이 알려지게 되었으며 기술의 발전과 함께 자바스크립트 역시 진화를 거듭했다.

웹을 위한 주요 돌파구는 광범위한 브라우저 지원과 고속 인터넷 보급과 더불어 찾아왔다. 이는 전체 페이지를 읽는 대신 필요한 부분만 클라이언트에게 전송하고 배경 작업은 Ajax 로 처리하는 기술의 문을 열었다.

Ajax 를 사용하는 앱에서는 네트워크 성능이 기본적으로 필요하다. 연결이 느리면 페이지가 반응하지 않는 것처럼 보여질 수 있기 때문이다.

앱은 점진적으로 페이지 로딩을 줄이고 Ajax 요청을 늘리는 쪽으로 이동하고 있다. 궁극적으로 단일 페이지 앱(Single Page Application)이 탄생했다. 가장 엄격한 의미에서 SPA 는 딱 한 페이지만 읽고 나머지 모든 데이터는 Ajax 호출로 처리한다.

Angular 는 SPA 작성용으로 가장 인기 있는 프레임워크다. 앵귤러의 주목할 만한 특징으로는 MVC 를 적용한 것과 뷰와 모델 사이에서 양방향 데이터 바인딩을 제공한다는 것이다.

더불어서 자바스크립트로 서버 측 프로그래밍을 하는 시대를 노드가 열었다. 노드는 자바스크립트에 익숙한 개발자를 단숨에 초급 풀스택 개발자로 바꿔놓는 능력을 보여주고 있다.

2. Online Test Site : JSFiddle

자바스크립트를 간단하게 테스트하기 위해서 온라인 사이트를 이용할 수 있다.
브라우저는 크롬브라우저를 기본으로 사용하도록 한다.

온라인 자바스크립트 테스트 사이트 주소
https://jsfiddle.net/

JAVASCRIPT 창 영역에 코드를 작성해 넣고 Run 버튼을 클릭하면 된다.
콘솔에 찍는 로그정보는 크롬브라우저인 경우 F12 키를 누르면 뜨는 개발자 모드에서 확인할 수 있다.

3. Types

자바스크립트의 자료형을 점검해 보자.

Type	Primitive	Description
Number	v	the double-precision 64-bit binary format, 정수, 부동형 숫자, +Infinity, -Infinity, and NaN
String	v	문자, 문자열, 0 부터 시작하는 인덱스를 사용
Boolean	v	true, false
Undefined	v	변수를 만들기만 하고 값을 할당하지 않은 상태
Null	v	변수에 값이 없는 상태
Symbol	v	ECMAScript Edition 6 버전에서 새로 도입했다. unique and immutable primitive value, 드디어 상수를 제대로 사용할 수 있다. 객체 속성(object property)들의 식별자로 사용한다. 참조 : http://blog.keithcirkel.co.uk/metaprogramming-in-es6-symbols/
Object		객체

"자바스크립트는 모두 객체다." 라고 얘기하면 사실 반만 맞고 반은 틀리다.
자바스크립트에도 분명히 원시타입이 존재하며 이는 표현될 때 적용된다. 원시타입을 객체로 취급하기 위한 래퍼가 존재한다. 자바스크립트는 "처리 시 모두 객체로 취급하여 처리한다"고 이해하는 것이 옳다.

예제 1

다음 예제의 비교 결과는 모두 true 다. 결과가 예상과 달라 의아하다면 다른 언어에 해당하는 관점을 적용했기 때문이다. 잠시 기존 관점을 벗어두자. 자바스크립트는 자바가 아니다. 자바에 친숙한 개발자들이 오해하기 쉬운 것은 호칭이 비슷해서 일 것이다. 자바와 자바스크립트는 아무 상관이 없는 다른 언어다.

typeof 연산자는 처리 결과인 자료형을 문자열로 돌려준다.

/type/test1.js

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

log(typeof 7 === 'number');
log(typeof 3.14 === 'number');
log(typeof Infinity === 'number');
log(typeof -Infinity === 'number');
log(typeof NaN === 'number');
```

```
log(typeof '' === 'string');
log(typeof 'bla' === 'string');

log(typeof true === 'boolean');
log(typeof false === 'boolean');

log(typeof undefined === 'undefined');
log(typeof blabla === 'undefined');

var a = null;
log(typeof a !== 'null');
log(typeof a === 'object'); // object 를 리턴할 때 null 로 바꾸자는 일부 개발자의 의견은 거절되었다!
log(a === null); // null 체크시 typeof 연산자를 사용하지 말자.

log(typeof {a:1} === 'object');
log(typeof [1, 2, 4] === 'object'); // 배열도 객체다!
log(typeof new Date() === 'object');
log(typeof null === 'object');

log(Array.isArray([1, 2, 4])); // 배열 체크 방법

var sym = Symbol("foo");
log(typeof sym === 'symbol');

log(typeof function(){} === 'function');
log(typeof Math.random === 'function');

log(Number.parseInt === parseInt); // 짧게 parseInt 함수를 사용할 수 있는 이유, 슈가코드!
log(typeof parseInt('3.14', 10) === 'number'); // 리턴 결과의 타입을 비교

log(function(){} instanceof Function);
log(function(){} instanceof Object);
```

typeof 연산자로 배열을 확인하면 'object'를 리턴한다. 이는 배열도 결국 객체이기 때문이다.
대상객체가 배열인지 체크 시 Array.isArray 함수를 사용하자.

instanceof 연산자로 상속관계를 확인할 수 있다. 함수를 만들면 컴파일 될 때 자동적으로 Function 객체의 자식이 된다. Function 객체는 Object 객체의 자식이므로 비교 결과는 true 가 되는 것이다.

예제 2

먼저 결과를 예상해보고 답을 확인해 보자. true 또는 false 중에 하나를 고르자.

/type/test2.js

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

log(123 == '123');
log(123 === '123');

log('------------------------------');

var x;
log(typeof x === 'null');
log(x == null);
log(x === null);
log(x === undefined);

log('------------------------------');

log(isNaN(0 / 0) === true);
log(3 / 0 === Infinity);
log(-3 / 0 === -Infinity);

log(Infinity === +Infinity);
log(Infinity === -Infinity);
log(+Infinity === -Infinity);
```

== 연산자는 값만 비교한다.
=== 연산자는 값과 자료형을 비교한다.

그러므로 항상, === 연산자를 사용하도록 한다.

123 == '123' 비교 시 한 쪽이 자동으로 캐스팅되어 비교되므로 결과는 true 가 되어 원치 않는 버그를 발생시킬 수 있다.

예제 3

```
var log = console.log;

var a = 1;
var b = a; // a 가 갖고 있는 값을 복사해 b 에 담는다.
a = 2;
log(b); // 원시타입인 경우 변수는 값을 담는 그릇이다.

var c = {num:1};
var d = c; // c 가 가리키는 객체를 d 도 가리키게 한다.
c = {num:2}; // 새로 만든 객체를  c 가 가리킨다.
log(d);

var e = {num:1};
var f = e;
e.count = 2;
log(f); // e 와 f 가 같은 객체를 가리키고 있다.
```

4. Regular Expression

정규표현식은 문자들의 패턴을 묘사하는 **객체**다.

정규표현식은 사용자가 입력한 정보가 작성규칙에 맞게 만들어 졌는지 확인하는데 적합하다. 예를 들어 사용자가 작성한 이메일, 전화번호, 아이디나 패스워드 등이 작성규칙에 맞게 만들어 졌는지 체크할 때 주로 사용한다. 잘 작성된 정규표현식은 코드의 양을 획기적으로 줄여주며 관리성을 증대시킨다.

예제 1

/regex/test1.js

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

log('hello world'.replace('world', 'javascript'));

log('Blue has a blue house and a blue car'.replace('blue', 'red'));
log('Blue has a blue house and a blue car'.replace(/blue/g, 'red'));
log('Blue has a blue house and a blue car'.replace(/blue/gi, 'red'));
log('Blue has a blue house and a blue car'
        .replace(/blue|house|car/gi, function myFunction(x){return x.toUpperCase();}));

log('hello'.replace(/h/, 'b'));
log('hello'.replace(/h../, 'bbb'));
```

/정규표현식/ 연산자는 정규표현식 객체 표기법이다. 이렇게 선언한 자체로 객체다.

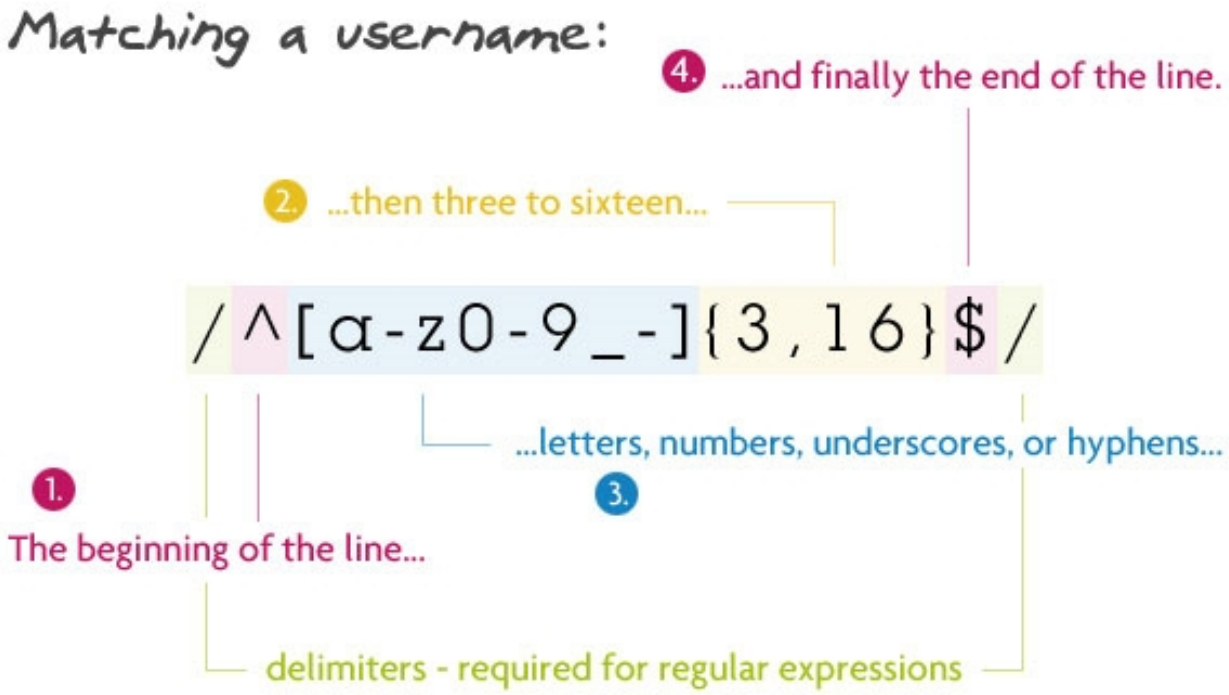
정규표현식 전역 옵션

Option	Meaning	Description
g	global	문장 전체에 작업을 적용한다.
i	case-insensitive	대소문자를 구분하지 않고 모두를 대상으로 작업을 적용한다.
m	multiline	여러 행에 걸쳐 작업을 적용한다.

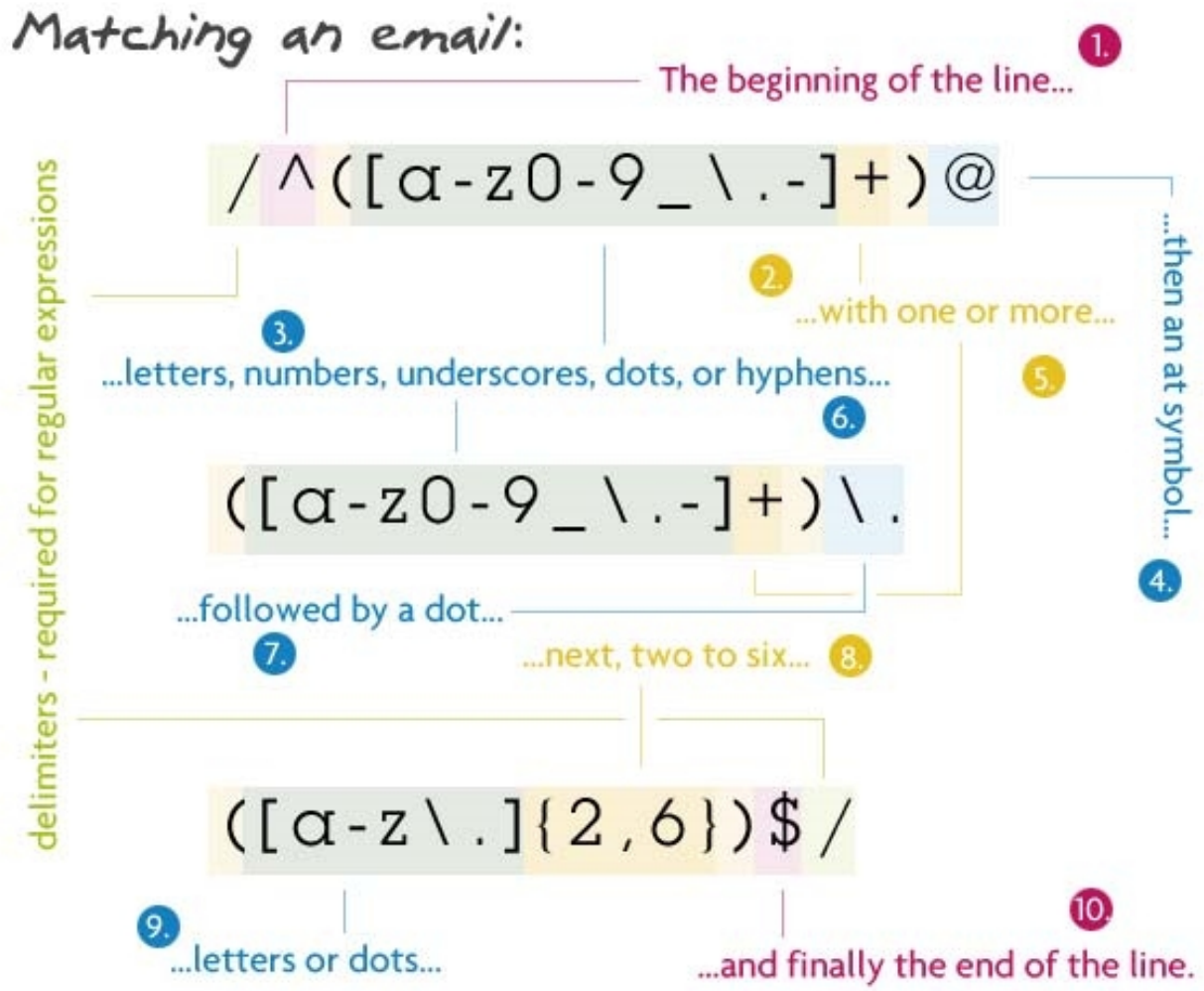
문자열을 | 연산자로 연결하여 사용하면 여러 문자열을 적용대상으로 지정할 수 있다.

replace 함수 두 번째 파라미터로 함수를 사용하고 있다. 함수를 사용하면 부가적인 로직처리를 위한 공간을 확보할 수 있다. 정규표현식으로 찾은 대상이 콜백함수의 파라미터로 전달되고 콜백함수의 리턴 값이 결국 치환되는 값으로 사용 된다.

정규표현식: 사용자이름



정규표현식: 이메일



패턴

/^([a-z0-9_₩.-]+)@([₩da-z₩.-]+)₩.([a-z₩.]{2,6})$/

1. ^([a-z0-9_₩.-]+)

이메일 아이디에 해당하는 부분이다.

알파벳, 숫자, 언더바, 점, 대쉬

1~n 개

점을 ₩ 기호없이 사용하면 임의의 캐릭터 한개를 의미하므로 ₩를 붙여야 한다.

2. @

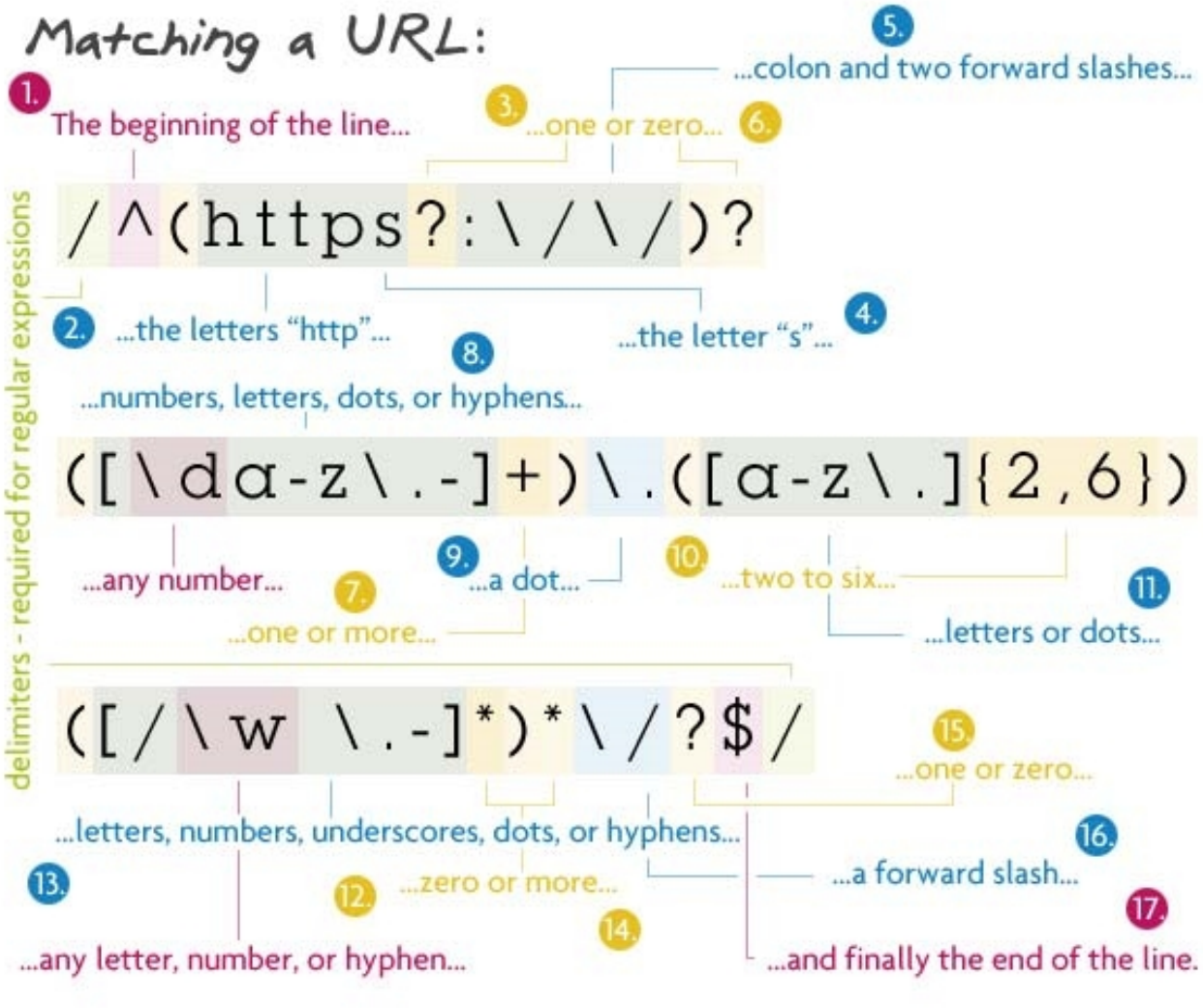
이메일에서 사용하는 골뱅이 기호가 있어야 한다.

3. ([₩da-z₩.-]+)
도메인 이름에 해당하는 부분이다.
숫자, 알파벳, 점, 대쉬가 가능하다.
1~n 개

4. ₩.
도메인 이름 다음 점이 위치해야 한다.

5. ([a-z₩.]{2,6})$
알파벳, 점(id@domain.co.kr)도 포함한다.
2~6 개

정규표현식: URL



패턴

/^(https?:₩/₩/)?([₩da-z₩.-]+)₩.([a-z₩.]{2,6})([₩/₩w ₩.-]*)*₩/?$/

1. ^(https?:₩/₩/)?

"http://", "https://", 생략 셋 모두 가능하다.

1.1 s?

s{0,1} 과 같다.

1.2 ₩/₩/

슬래쉬 // 2 개가 필요하다.

2. (([₩da-z₩.-]+)₩.([a-z₩.]{2,6})

1~n 개 숫자, 문자, 점, 대쉬가 가능하다.
다음에 점 하나가 있어야 한다.
다음에 문자+점이 2~6 개 있을 수 있다.

3. ([₩/₩w ₩.-]*)*₩/?$

슬래쉬 다음에 단어, 공백, 점, 대쉬가 0~n 개 가능하다.
또한 이 조합이 0~n 개 가능하다.
맨 뒤에 슬래쉬가 0~1 개 가능하다.

예제 2

/regex/test2.js

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

var pattern = /h/gi;

log(typeof pattern); //object
log(pattern.__proto__ === RegExp.prototype);
log(RegExp.prototype.__proto__ === Object.prototype);
log('--------------------------');

log(pattern.test("Happy")); //true
pattern = /h/g;
log(pattern.test("HAPPY")); //false
log('--------------------------');

var idExp = /^[a-z][a-z0-9]{3,4}[a-z]$/;
// ^[a-z] : 영문으로 시작
// [a-z0-9]{3,4} 영문과 숫자조합, 3 자 이상  4 자 이하
// [a-z]$ : 영문으로 끝

log(idExp.test("1234"));
log(idExp.test("i1234"));
log(idExp.test("i1234n"));
log(idExp.test("i12345n"));
```

```
log(idExp.test("a123m"));
log('-------------------------');

var passwordExp = /^[0-9]{2,3}$/;
// ^[0-9] : 숫자로 시작
// [0-9]{2,3}$ : 숫자로 끝, 2 자이상 3 자 이하

log(passwordExp.test("1"));
log(passwordExp.test("12"));
log(passwordExp.test("123"));
log(passwordExp.test("1234"));
log(passwordExp.test("12m"));
```

문자 일치 패턴: Brackets

MongoDB 에서 데이터를 구하기 위한 질의를 할 때 사용할 수 있다.

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character **NOT** between the brackets
[0-9]	Find any digit between the brackets
[^0-9]	Find any digit **NOT** between the brackets
(x|y)	Find any of the alternatives specified

문자 일치 패턴: Metacharacters

백슬래시 다음 문자는 문자 그대로 해석하면 안된다고 나타내는 구분 기호이다.

Express 웹 프레임워크에서 URL Pattern 을 매핑할 때 사용할 수 있다.

Expression	Description
.	Find a **single** character, except newline or line terminator
₩w	Find a **word** character
₩W	Find a **non-word** character
₩d	Find a **digit**
₩D	Find a **non-digit** character
₩s	Find a **whitespace** character
₩S	Find a **non-whitespace** character
₩b	Find a **match at the beginning/end of a word**
₩B	Find a **match not at the beginning/end of a word**
₩0	Find a NUL character

₩n	Find a new line character
₩f	Find a form feed character
₩r	Find a carriage return character
₩t	Find a tab character
₩v	Find a vertical tab character
₩xxx	Find the character specified by an octal number xxx
₩xdd	Find the character specified by a hexadecimal number dd
₩uxxxx	Find the Unicode character specified by a hexadecimal number xxxx

문자 수 제한: Quantifiers

*	{0,} 와 동일, **0~n** 개
+	{1,} 와 동일, **1~n** 개
?	{0,1}와 동일, **0~1** 개
n+	at least one, n 이 1~n 개
n*	zero or more, n 이 0~n 개
n?	zero or one, n 이 0~1 개
n{X}	연속적으로 n 이 X 개
n{X,Y}	연속적으로 n 이 X~Y 개
n{X,}	at least X, n 이 X~n 개
n$	n 으로 문자열이 종료
^n	n 으로 문자열이 시작
?=n	Matches any string that is followed by a specific string n, n 다음에 배치되는 모든 문자열
?!n	Matches any string that is not followed by a specific string n, n 다음에 배치되지 않는 모든 문자열

예제 3

/regex/test3.js

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

var re = /quick₩s(brown).+?(jumps)/ig;
```

```
var result = re.exec('The Quick Brown Fox Jumps Over The Lazy Dog');
log(result);
//[
//      'Quick Brown Fox Jumps', // 찾은 문자열
//      'Brown', // 캡쳐 1
//      'Jumps', // 캡쳐 2
//      index : 4, // 찾은 문자열 시작위치
//      input : 'The Quick Brown Fox Jumps Over The Lazy Dog' // 검색 대상 문자열
//]

var str = 'For more information, see Chapter 3.4.5.1';
var re = /(chapter ₩d+(₩.₩d)*)/i;
var found = str.match(re);
log(found);
//[
//      'Chapter 3.4.5.1',
//      'Chapter 3.4.5.1',
//      '.1', //The last value remembered from (₩.₩d)
//      index: 26,
//      input: 'For more information, see Chapter 3.4.5.1'
//]

var str = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz';
var regexp = /[A-E]/gi;
var matches_array = str.match(regexp);
log(matches_array);
// ['A', 'B', 'C', 'D', 'E', 'a', 'b', 'c', 'd', 'e']

var str = 'That was the night before Xmas...';
var newstr = str.replace(/xmas/i, 'Christmas');
log(newstr);
// That was the night before Christmas...

var re = /apples/gi;
var str = 'Apples are round, and apples are juicy.';
var newstr = str.replace(re, 'oranges');
log(newstr);
// oranges are round, and oranges are juicy.
```

```
var re = /(₩w+)₩s(₩w+)/;
var str = 'John Smith';
var newstr = str.replace(re, '$2, $1');
log(newstr);
// Smith, John

var myString = 'Hello World. How are you doing?';
var splits = myString.split(' ', 3);
log(splits);
// [ 'Hello', 'World.', 'How' ]

var myString = 'Hello 1 word. Sentence number 2.';
var splits = myString.split(/(₩d)/);
log(splits);
// [ 'Hello ', '1', ' word. Sentence number ', '2', '.' ]

var str = 'abcdefghi';
var strReverse = str.split('').reverse().join('');
// split() returns an array on which reverse() and join() can be applied
log(strReverse);
// ihgfedcba
```

정규표현식 온라인 매뉴얼
https://developer.mozilla.org/ko/docs/Web/JavaScript/Guide/%EC%A0%95%EA%B7%9C%EC%8B%9D

정규표현식 온라인 연습사이트
http://regexr.com/

5. Conditional Expression

if 문의 조건을 검증하는 괄호블록에서 테스트 대상만을 그대로 사용하면 다음과 같이 처리된다.

대상	처리
Object	true 로 평가된다.
Undefined	false 로 평가된다.
Null	false 로 평가된다.
Boolean	boolean 형의 값으로 평가된다.
Number	true 로 평가된다. 하지만, 0 or NaN 의 경우는 false 이다.
String	true 로 평가된다. 하지만, 빈문자 '' 의 경우는 false 이다.

클라이언트가 작성 폼에 정보를 입력했는지 확인하는 로직으로 간단히 다음과 같이 사용할 수 있다. name 변수 값이 undefined, null, '' 인 경우 false 로 처리되는 점을 이용하는 것이다.

DOM 엘리먼트 name 에 입력된 값이 있는지 체크하는 로직

```
var name = document.getElementById('name');
if(!name) {
	alert('작성된 name 정보가 존재하지 않습니다.');
	return false; // 이벤트의 전파를 막고 로직을 종료한다.
}
```

예제 1

/condition/test1.js

```
var log = console.log;

if ([]) {
	log(true);
} else {
	log(false);
}

log({} ? true : false);

var name = '';
if (name) {
	log(true);
```

```
} else {
        log(false);
}

log(name.length ? true : false);

if (!!-1 == true) {
        console.log('work though');
}

if (-1) {
        console.log('work better');
}
```

6. Arguments

객체지향 언어인 자바는 메소드 구분을 **시그니처**로 한다.
시그니처는 메소드명과 파라미터의 구성으로 이루어진다. 메소드명이 같아도 파라미터의 구성이 다르다면 다른 메소드가 된다. 이를 **메소드의 오버로딩**이라고 부른다.

반면에, 자바스크립트는 **함수의 유일성을 함수명만으로 구분**한다. 그렇다면 자바스크립트에서는 메소드오버로딩 개념이 없다는 얘기일까?

자바스크립트는 함수 호출 시 파라미터는 아무런 영향을 미치지 않는다.
오직 함수명만으로 함수를 구분하고 호출 시 연동된다. 이러한 방식이 가능하도록 하기 위해서 자동으로 모든 파라미터를 **arguments** 라는 객체에 담아 놓는다. 이 arguments 를 이용하면 자바의 메소드 오버로딩과 비슷한 기능을 수행할 수 있다.

그래서 자바스크립트에서는 파라미터의 개수와 형을 명시하고 사용하고자 하는 의도가 분명히 존재하는 것이 아니라면 파라미터 정의를 비워두는 것이 좋다. 자바스크립트는 하나의 함수로 다양한 자료형의 복수(0..n) 개의 파라미터를 받아 처리하는 것이 가능하다.

예제 1

/argument/test1.js

```
function add(){
	var sum = 0;
	for(var i=0; i < arguments.length; i++){
		sum += arguments[i];
	}
	return sum;
}

function add2(a, b){
	return a + b;
}

function add3(a, b, c){
	return a + b + c;
}

function log(data){
```

```
        console.log(data);
}

log('add: ' + add(1, 2)); //3
log('add: ' + add(1, 2, 3)); //6

log('add2: ' + add2(1, 2)); //3
log('add3: ' + add3(1, 2, 3)); //6
log('add3: ' + add3(1, 2, 3, 4)); //6
```

arguments 는 모든 함수에 자동으로 존재한다. 넘겨 받는 파라미터를 자동으로 담고 있는 객체다.
arguments 를 for 문을 사용하여 받은 파라미터 개수만큼 처리할 수 있다.

add(1, 2)로 함수를 호출하면 add 함수 내부에 존재하는 arguments 의 크기는 2 가 된다.
add(1, 2, 3)로 함수를 호출하면 add 함수 내부에 존재하는 arguments 의 크기는 3 이 된다.

add2(a, b) 함수 정의부분에 파라미터 2 개를 명시하고 있다. 이는 개발자가 이 함수는 2 개의 파라미터를 받고자 한다는 의도를 명시하고자 하는 경우라 하겠다. 적절한 대응 로직이 있다면 정의된 파라미터 개수와 상관없이 파라미터를 안 주거나 더 많이 주어도 상관이 없다.

결국, arguments 가 있으므로써 자바스크립트에서 파라미터 정의는 생략이 가능하고 메소드 오버로딩 개념은 필요없게 된다.

arguments 는 배열처럼 보이지만 배열이 아니다. 사용상에 주의가 필요하다. 배열과 비슷한 부분은 length 라는 속성을 쓸 수 있다는 정도 뿐이다. 배열이 아니므로 배열의 pop 이나 push 함수 등은 존재하지 않는다.

예제 2

arguments 를 배열로 변경해서 사용하고자 한다면 다음 코드를 참조하자.

/argument/test2.js

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

function foo() {
        return arguments;
}
```

```
log(foo(1, 2, 3));
// { '0': 1, '1': 2, '2': 3 }

function changeToArray() {
        var args = Array.prototype.slice.call(arguments);
        return args;
}

log(changeToArray(1, 2, 3));
// [ 1, 2, 3 ]

function myConcat(seperator) {
        var args = Array.prototype.slice.call(arguments, 1); // 인덱스 1 부터 slice 한다.
        return args.join(seperator);
}

log(myConcat(", ", "red", "orange", "blue")); // 첫 파라미터는 seperator 다.
//red, orange, blue

log(myConcat("; ", "elephant", "giraffe", "lion", "cheetah"));
//elephant; giraffe; lion; cheetah
```

myConcat 함수에서 변수 args 는 배열로 변경된 값을 갖고 있다. join 함수를 사용하면 지정한 구분자로 구분하는 문자열 데이터를 얻을 수 있다.

예제 3

다음은 실용적인 예로 HTML 리스트를 만드는 함수를 살펴보자.

/argument/test3.js

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

function list(type) {
        var result = "<" + type + "l><li>";

        var args = Array.prototype.slice.call(arguments, 1);
```

```
        result += args.join("</li><li>");

        result += "</li></" + type + "l>"; // end list

        return result;
}

var listHTML = list("u", "One", "Two", "Three");

log(listHTML);
//<ul><li>One</li><li>Two</li><li>Three</li></ul>
```

7. Closure

클로져는 함수와 함수가 사용하는 환경을 메모리에 유지하여 사용할 수 있는 객체다.
자바스크립트에는 클래스가 없다. 그런데 클로져는 자바의 클래스 인스턴스와 가장 닮아 있는 객체다.

자바의 경우 콜백메소드에서 사용하는 외부 변수는 콜백 메소드를 명시한 메소드의 종결과 상관이 없어야 한다. 메소드가 종결연산자를 만나면 스택에 저장된 지역변수들은 자동적으로 모두 파괴되기 때문이다. 이를 극복하기 위해서 자바는 콜백메소드에서 사용하는 외부 변수가 메소드 종결과 상관없는 클래스의 필드변수이거나 지역변수인 경우 final 키워드를 붙여서 콜백메소드가 캐시하도록 한다.

결국, 자바의 final 키워드는 콜백메소드가 외부 변수를 캐시하고 있으므로 값을 변경할 수 없다는 것을 나타내는 키워드다.

자바스크립트는 클로져를 사용하여 콜백함수가 외부 스코프의 자원을 계속해서 사용할 수 있게 만든다.

예제 1

우선 클로져를 왜 사용해야 하는지 필요성을 알아보자.

/closure/counter1.js

```
var counter = 0;

function add() {
        counter += 1;
}

add();
add();
add();

console.log('counter: '+counter);
```

add 함수를 호출하여 카운터를 1 씩 증가 시키는 로직이다.
여기서 문제점은 add 함수를 호출하지 않고도 counter 를 직접 접근할 수 있다는 점이다.

예제 2

사용의 오용을 막기위해 다음과 같이 코드를 변경해보자.

/closure/counter2.js

```
function add() {
        var counter = 0;
        counter += 1;
        return counter;
}

console.log('counter: '+add());
console.log('counter: '+add());
console.log('counter: '+add());
```

counter 변수를 함수안으로 이동하여 add 함수를 호출하지 않고 직접 변수로 접근하는 길을 차단하였다. 이제 counter 는 함수가 갖고 있는 지역변수다.

하지만, 결과가 기대한 것과 다르게 언제나 1 만을 돌려준다. 이는 counter 가 지역변수로써 함수를 호출할 때 마다 초기화가 이루어져 0 이 되기 때문이다.

예제 3

코드를 조금 더 수정 해 보자.

/closure/counter3.js

```
function add() {
        var counter = 0;
        return function () {
                counter += 1;
                return counter;
        };
}

var incrementer = add(); // 함수가 리턴하는 익명함수를 가리킨다.

console.log('counter: ' + incrementer());
console.log('counter: ' + incrementer());
console.log('counter: ' + incrementer());
```

우선, add 함수를 호출하면 counter 를 증가하는 기능의 함수를 리턴하도록 변경한다.
함수 안에 내장된 익명함수는 함수안에서 외부 스코프에 위치한 변수 counter 를 사용한다.

이 경우 add 함수가 리턴하는 함수는 **클로저**가 된다. 클로져가 사용하는 변수는 사용하는 함수와 더불어 별도로 메모리에 존재하게 된다.

이제 사용의 오용을 없애기 위한 변수의 직접 접근도 막았고 의도한 대로 잘 작동하는 코드를 얻었다.

예제 4

이해를 돕기위해 다른 예제를 살펴보자.

/closure/test1.js

```
function init() {
        var name = "chris";
        function displayName() {
                console.log(name);
        }
        displayName();
}

init();
```

init 함수는 name 이라는 지역변수를 만들고 displayName 이라는 지역(내부)함수를 정의하고 있다.
내부 함수는 그 함수를 정의한 내부에서만 사용할 수 있다. 주목해야 할 부분은 내부함수 displayName 이 함수범위 밖에 정의된 name 변수를 사용하고 있는 부분이다.

일반적으로 함수안에서 사용된 자원(변수, 함수)은 그 함수에 스코프가 끝나면 파괴되고 그 후에 사용할 수 없다. 자바스크립트에서 함수가 중첩되는 경우 내부 함수가 외부에서 정의된 자원을 사용할 수 있도록 클로져를 만든다.

예제 5

/closure/test2.js

```
function makeFunc() {
        var name = "chris";
        function displayName() {
                console.log(name);
        }
        return displayName;
```

```
}

var myFunc = makeFunc();
myFunc();
```

이전 예제와 다른 점은 외부함수 makeFunc 이 내부함수 displayName 을 리턴한다는 것이다.
내부함수에서 외부 속성을 사용할 수 있도록 지원하기 위해서 외부함수가 클로져(closure)를 갖는다.

클로져는 함수와 그 함수가 만들어진 환경으로 이루어진 특별한 객체다.

그 함수가 만들어진 환경은 함수가 만들어질 때 사용할 수 있었던 자원(변수, 함수)이다.

예제 6

예제를 하나 더 살펴보자.

/closure/test3-1.js

```
function makeAdder(x) {
        return function (y) {
                return x + y;
        };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2)); // 7
console.log(add10(2)); // 12
```

이 예제에서 makeAdder(x) 라고 하는 하나의 인자를 받는 함수를 만들었다.
이 함수는 x 라는 인자를 받아서 새로운 함수를 리턴한다. 파라미터 변수 x 가 지역변수임을 명심하자. 리턴된 함수는 y 라는 인자를 받아서 x+y 를 돌려주는 함수다.

makeAdder 는 함수 공장(function factory)이다.
특정한 수(x)를 파라미터로 미리 받아서 설정하고 나중에 들어오는 인자(y)에 더해서 돌려주는 기능의 함수들을 '찍어낸다'.

하나의 함수가 처리해야 할 로직을 여러 함수로 분할하는 방법을 커링(currying)이라고 부른다.

변수 add5 와 add10 이 가리키는 함수는 둘다 클로져다.
두 함수는 하나의 함수정의 makeAdder 를 사용하지만 다른 환경(지역변수 x)을 갖고 있다.
add5 의 환경에서 x 는 5 이지만 add10 의 환경에서 x 는 10 이다.

예제 7

앞에서 본 로직은 다음 로직과 같은 의미를 갖는다.

/closure/test3-2.js

```
var add5 = function (y) {
        var x = 5;
        return x + y;
};

var add10 = function (y) {
        var x = 10;
        return x + y;
};

console.log(add5(2)); // 7
console.log(add10(2)); // 12
```

8. Callback

클라이언트 측 자바스크립트에서 사용자와 대화하는 방법으로 이벤트 드리븐 방식을 사용한다.
이벤트 발생 시 해당 함수를 기동시킨다. 이렇게 사용되는 함수를 콜백(callback)함수라고 부른다.

예제 1

예를 들어 페이지의 글자 크기를 조정하는 몇 개의 버튼을 만든다고 생각해보자.
body 엘리먼트에 px 단위로 font-size 를 설정하고 다른 엘리먼트에서는 상대적인 em 단위로 font-size 를 설정한다.

css 코드

```
body {
	font-family: Helvetica, Arial, sans-serif;
	font-size: 12px;
}
h1 {
	font-size: 1.5em;
}
h2 {
	font-size: 1.2em;
}
```

javascript 코드

```
function makeSizer(size) {
	return function() {
		document.body.style.fontSize = size + 'px';
	};
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);

document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
```

size12, size14, size16 은 body 엘리먼트의 글자 크기를 각각 12, 14, 16 픽셀로 바꾸는 함수이다.

함수공장 패턴을 사용하여 중복성이 없는 멋진 로직을 사용하고 있다.
이제 이 함수를 버튼과 연결시키자.

html 코드

```
<p>Some paragraph text</p>
<h1>some heading 1 text</h1>
<h2>some heading 2 text</h2>

<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-16">16</a>
```

온라인에서 자바스크립트를 테스트할 수 있는 jsfiddle 사이트에서 확인 해 보자.
http://jsfiddle.net/vnkuZ/

9. Private in JavaScript

몇몇 언어(예를 들어 자바)는 같은 클래스 내부에서만 호출할 수 있는 private 키워드를 지원한다.

자바스크립트는 이를 지원하지 않지만 클로져를 이용해서 흉내낼 수 있다.
private 함수는 코드에 제한적인 접근만을 허용한다는 점 뿐만 아니라 전역 네임스페이스를 깔끔하게 유지할 수 있다는 장점이 있다.

예제 1

아래에 모듈 패턴이라고 알려진 클로져를 통해 몇 개의 public 함수가 private 함수와 변수에 접근하는 코드가 있다.

/private/test1.js

```
var Counter = (function () {
        var privateCounter = 0;
        function changeBy(val) {
                privateCounter += val;
        }

        return {
                increment : function () {
                        changeBy(1);
                },
                decrement : function () {
                        changeBy(-1);
                },
                value : function () {
                        return privateCounter;
                }
        }
})(); // 즉시실행

console.log(Counter.value()); //0

Counter.increment();
Counter.increment();
console.log(Counter.value()); //2
```

```
Counter.decrement();
console.log(Counter.value()); //1
```

이전 예제에서는 각 클로져가 자신만의 환경을 가졌지만 이 예제에서는 **하나의 환경을 리턴되는 객체의 멤버인 세 함수가 공유**한다.

이 환경에는 두 개의 private 아이템이 존재한다. 하나는 privateCounter 라는 변수고 나머지 하나는 changeBy 라는 함수다. 이 두 아이템 모두 외부에선 접근할 수 없지만 익명함수 안에 정의된 세개의 함수에서는 사용할 수 있다.

예제 2

이번에는 함수를 다른 변수에 저장하고 이 변수를 이용해 여러개의 카운터를 만들어서 사용해 보자.

/private/test2.js

```
var makeCounter = function () {
        var privateCounter = 0;
        function changeBy(val) {
                privateCounter += val;
        }

        return {
                increment : function () {
                        changeBy(1);
                },
                decrement : function () {
                        changeBy(-1);
                },
                value : function () {
                        return privateCounter;
                }
        }
};

var Counter1 = makeCounter();
var Counter2 = makeCounter();

console.log(Counter1.value()); //0
```

```
Counter1.increment();
Counter1.increment();
console.log(Counter1.value()); //2

console.log(Counter2.value()); //0
```

makeCounter 함수를 호출하면서 생긴 환경은 호출할 때마다 다르다.
그러므로, 클로져 변수 privateCounter 는 다른 값을 가진다.

클로져는 정적으로 처리되는 것이 아니라 함수를 사용할 때 동적으로 함수안에 클로져가 만들어진다는 것을 알 수 있다. 클로져는 실행 시점에 만들어진다는 것을 기억하자.

객체지향 프로그래밍을 사용할 때 얻는 이점인 정보 은닉과 캡슐화를 자바스크립트에서는 클로져를 사용함으로써 얻을 수 있다는 것을 살펴보았다.

10. Closure in For

ECMAScript 6 에서 **let** 키워드가 도입되기 이전에는 반복문 안에서 클로져를 생성해서 문제가 되는 경우가 빈번했다.

예제 1

다음 예제 코드를 실행해보면 예상한대로 동작하지 않음을 알 수 있다.
onfocus 에 할당된 함수의 실행 시점을 곰곰히 생각해 보자.

html 코드

```
<p id="help">Helpful notes will appear here</p>
<p>E-mail: <input type="text" id="email" name="email"></p>
<p>Name: <input type="text" id="name" name="name"></p>
<p>Age: <input type="text" id="age" name="age"></p>
```

javascript 코드

```
function showHelp(help) {
        document.getElementById('help').innerHTML = help;
}

function setupHelp() {
        var helpText = [
                {'id': 'email', 'help': 'Your e-mail address'},
                {'id': 'name', 'help': 'Your full name'},
                {'id': 'age', 'help': 'Your age (you must be over 16)'}
        ];

        for (var i = 0; i < helpText.length; i++) {
                var item = helpText[i];
                document.getElementById(item.id).onfocus = function() {
                        showHelp(item.help);
                }
        }
}

setupHelp();
```

확인: http://jsfiddle.net/v7gjv/

helpText 배열은 세개의 도움말을 정의한다.
각 도움말은 입력 필드의 id 와 연관된다. 이 세개의 정의를 반복하며 입력필드에 onfocus 이벤트가 발생했을 때 도움말을 id="help"인 p 태그에 표시한다.

사실 이 코드는 제대로 동작하지 않는다.
어떤 필드에 포커스를 주더라도 나이에 관한 도움말이 표시된다.

이유는 onfocus 이벤트에 지정한 함수들이 클로저인데 모두 하나의 환경을 공유하기 때문이다.
클로저가 개별적으로 환경을 갖도록 하기 위해서는 함수를 할당할 때 실행이 필요하다.

반복문이 끝나고 onfocus **콜백**이 실행될 때 콜백의 환경에서 item 변수는 helpText 리스트의 마지막 요소를 가리키고 있기 때문에 어느 input 박스에 포커스를 주어도 마지막 도움말만 표시된다.

클로져마다 따로 환경을 만들면 이 문제를 해결할 수 있다.
위에서 언급한 함수 공장을 사용해보자. 내부함수가 사용하는 외부변수를 개별적인 환경에서 처리하기위해 범위연산자로 감싸는 형태를 취하고 바로 실행해서 그 결과를 할당한다.

예제 2

```
function showHelp(help) {
	document.getElementById('help').innerHTML = help;
}

function makeHelpCallback(help) {
	return function() {
		showHelp(help);
	};
}

function setupHelp() {
	var helpText = [
		{'id': 'email', 'help': 'Your e-mail address'},
		{'id': 'name', 'help': 'Your full name'},
		{'id': 'age', 'help': 'Your age (you must be over 16)'}
	];

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

```
            var item = helpText[i];
            document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
        }
}

setupHelp();
```

item.help 가 갖고 있는 값을 함수의 파라미터로 넘긴다. 넘어간 값은 makeHelpCallback 함수의 파라미터로써 지역변수가 된다. 지역변수 help 는 리턴되는 함수에서 사용하는 클로져의 환경이 된다.

이로써 루프가 돌면서 사용되던 item.help 는 더 이상 클로져의 환경이 아니다.

확인: http://jsfiddle.net/v7gjv/1/

결과를 확인 해 보면 예상한대로 잘 작동한다.
하나의 환경을 공유하지 않고 makeHelpCallback 함수가 리턴하는 매번 새로운 환경을 가지게 된다.

예제 3

makeHelpCallback 함수를 사용하지 않고 즉시 실행 함수를 사용하는 방법은 다음과 같다.

```
function showHelp(help) {
        document.getElementById('help').innerHTML = help;
}

function setupHelp() {
        var helpText = [
            {'id': 'email', 'help': 'Your e-mail address'},
            {'id': 'name', 'help': 'Your full name'},
            {'id': 'age', 'help': 'Your age (you must be over 16)'}
        ];

        for (var i = 0; i < helpText.length; i++) {
            var item = helpText[i];
            document.getElementById(item.id).onfocus = (function(help) {
                return function() {
                    showHelp(help);
                }
```

```
                })(item.help);
        }
}

setupHelp();
```

11. Closure Performance

클로져가 필요하지 않은 경우인데도 함수안에 함수를 만드는 것은 스크립트 처리 속도와 메모리 사용량 모두에서 현명한 선택이 아니다.

예를들어 새로운 오브젝트를 만들 때 오브젝트 생성자에 함수를 정의하는 것 보다 오브젝트의 프로토타입에 정의하는것이 좋다.

오브젝트 생성자에 정의하게 되면 생성자가 불릴 때마다 함수가 새로 할당되기 때문이다.

예제 1

```
(function () {
	if (typeof Object.prototype.uniqueId === "undefined") {
		var id = 0;
		Object.prototype.uniqueId = function () {
			if (typeof this.__uniqueid === "undefined") {
				this.__uniqueid = ++id;
			}
			return this.__uniqueid;
		};
	}
})();

function MyObject(name, message) {
	this.name = name.toString();
	this.message = message.toString();

	this.getName = function () {
		return this.name;
	};

	this.getMessage = function () {
		return this.message;
	};
}
console.log(MyObject.uniqueId());
```

```
var obj1 = new MyObject('chris', 'hi');
console.log(obj1.getName.uniqueId());
console.log(obj1.getMessage.uniqueId());

console.log(MyObject.uniqueId());

var obj2 = new MyObject('aaron', 'bye');
console.log(obj2.getName.uniqueId());
console.log(obj2.getMessage.uniqueId());
```

위의 코드는 new 키워드로 생성자가 호출될 때마다 매번 생성자가 갖고 있는 함수를 만든다는 단점이 있다.

```
(function () {
	if (typeof Object.prototype.uniqueId === "undefined") {
		var id = 0;
		Object.prototype.uniqueId = function () {
			if (typeof this.__uniqueid === "undefined") {
				this.__uniqueid = ++id;
			}
			return this.__uniqueid;
		};
	}
})();

function MyObject(name, message) {
	this.name = name.toString();
	this.message = message.toString();
}

MyObject.prototype = {
	getName: function() {
		return this.name;
	},
	getMessage: function() {
		return this.message;
	}
};
console.log(MyObject.uniqueId());
```

```
var obj1 = new MyObject('kris', 'hi');
console.log(obj1.getName.uniqueId());
console.log(obj1.getMessage.uniqueId());

console.log(MyObject.uniqueId());

var obj2 = new MyObject('aaron', 'bye');
console.log(obj2.getName.uniqueId());
console.log(obj2.getMessage.uniqueId());

console.log(MyObject.prototype);
console.log(obj1.prototype);
console.log(obj2.__proto__);
```

위 예제에서는 상속된 속성은 모든 오브젝트에서 사용될 수 있고 함수 생성이 오브젝트가 생성될 때마다 일어나지 않는다.

결론적으로 말하자면 중복되는 기능은 prototype 객체내에 배치시키기를 권장한다.

12. Currying

커링(Currying) 은 인자를 여러개 받는 함수를 분리하여, 인자를 하나 받는 함수의 체인(Chain) 으로 만드는 방법이다.

농담이지만 인도의 커리와 상관이 없다. 수학자 해스켈 커리에 의해 정리된 개념이다.

예제 1

/currying/test1.js

```
var sum = function (a, b) {
	return a + b;
}

Function.prototype.curry = function () {
	var slice = Array.prototype.slice;
	var args = slice.apply(arguments); //[5]

	var that = this; //that --> sum
	return function () {
		return that.apply(null, args.concat(slice.apply(arguments)));
		//sum.apply(null, args.concat([7]);
	};
}

var sum5 = sum.curry(5);
console.log(sum5(7));
console.log(sum.call(null, 5, 7));
```

예제 2

일부 값이나 로직이 반복되는 경우가 커링의 적용대상이다. 반복되는 부분을 분리하여 미리 설정하고 변경되는 부분과 결합하여 사용하는 방식이다.

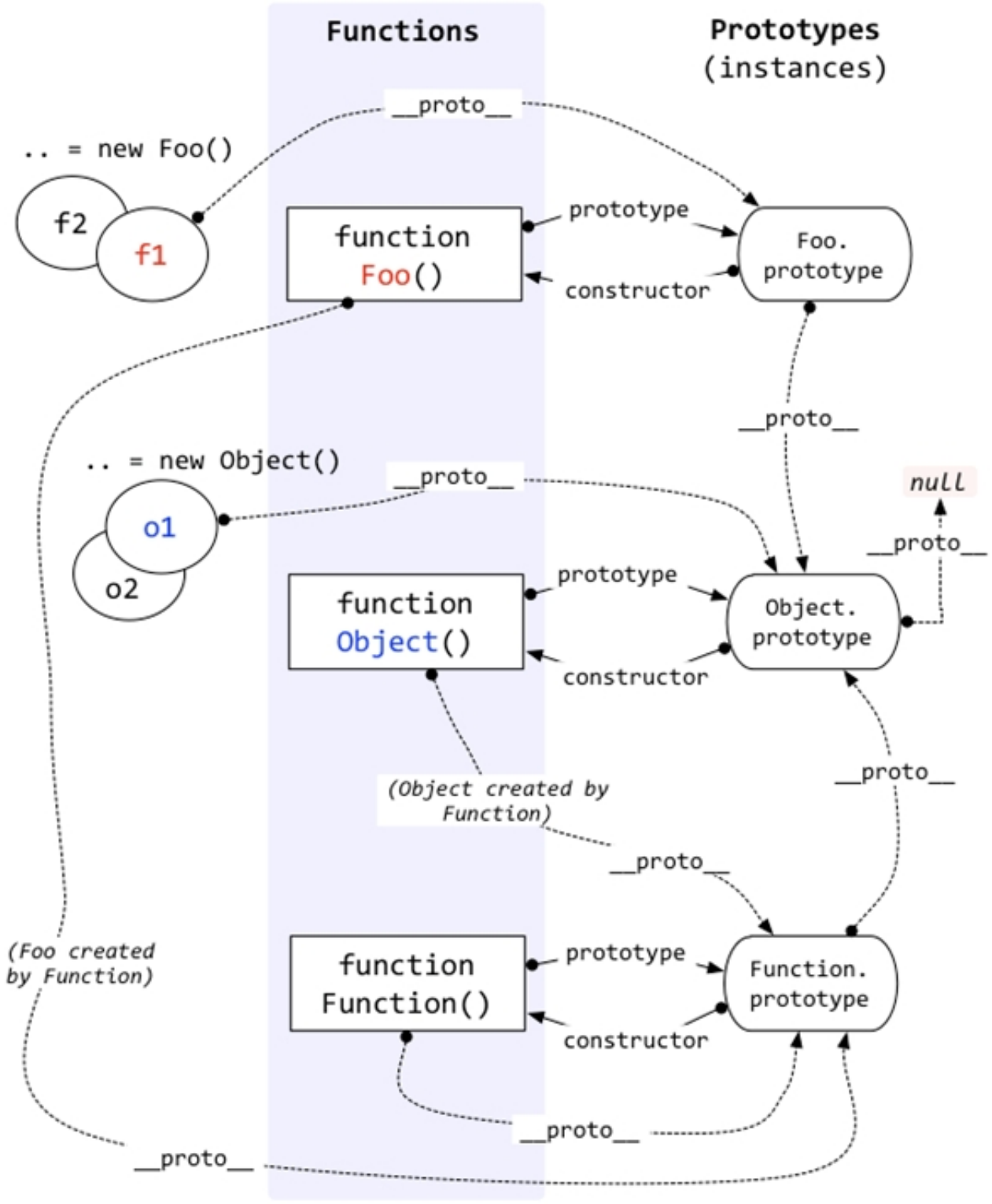
다음으로 실용적인 예제를 살펴보자.

/currying/test2.js

```
Function.prototype.curry = function () {
```

```
        var slice = Array.prototype.slice;
        var args = slice.apply(arguments);

        var that = this;
        console.log('that: '+that);

        return function () {
                return that.apply(null, args.concat(slice.apply(arguments)));
        };
}

function converter(toUnit, factor, offset, input) {
        console.log("t: " + toUnit + ", f: " + factor + ", o: " + offset + ", i: " + input);
        offset = offset || 0;
        return [((offset + input) * factor).toFixed(2), toUnit].join(" ");
}

var milesToKm = converter.curry('km', 1.60936, undefined);
var poundsToKg = converter.curry('kg',0.45460,undefined);
var farenheitToCelsius = converter.curry('degrees C',0.5556, -32);

console.log('------------------------');
console.log(milesToKm(10));
console.log(poundsToKg(2.5));
console.log(farenheitToCelsius(98));
```

13. Inheritance Diagram



실습: 위 다이어그램이 정확함을 하나씩 코드로 테스트 해 본다.

14. Class-based Inheritance

자바 같은 클래스기반(Class-based) 프로그램에서는 객체의 설계도가 되는 파일을 만들고 이를 인스턴스해서 객체를 만든다.

```
public class Shape {
	private int height;
	private int width;

	public Shape(int h, int w) {
		height = h;
		width = w;
	}

	public int area() {
		return height * width;
	}
}
```

```
Shape shape = new Shape(10, 2);
System.out.println(shape.area());
// 20
```

15. Prototype-based Inheritance

자바스크립트에서는 오직 객체와 원시타입 변수만이 존재한다.
변수도 표현할 때만 각 자료형에 적합한 방식을 취하는 것으로 취급할 때는 모두 객체로 취급한다.

객체를 만드는 방법

var obj = new 생성자함수();
var obj = {};
var obj = 함수(return 객체;);

예제 1

프로토타입기반(Prototype-based) 언어에서는 이미 존재하는 객체를 사용하여 객체를 생성한다.
객체를 만들기 위해서 먼저 함수를 만들어 보자.

/inheritance/test1.js

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

function Shape() {
	this.height = 10;
	this.width = 10;
	this.area = function () {
		return this.height * this.width;
	};
};

var shape = new Shape();
log(shape.area()); // 100
log(shape.hasOwnProperty('area')); // true

log(shape.toString()); // "[object Object]";
log(shape.hasOwnProperty('toString')); // false

log(shape.__proto__ === Shape.prototype); // true
log(Shape.prototype.hasOwnProperty('toString')); // false
```

```
log(Shape.prototype.__proto__ === Object.prototype); // true
log(Object.prototype.hasOwnProperty('toString')); // true
```

shape 가 가리키는 객체에는 toString 함수가 없는데 어떻게 사용할 수 있는걸까?

자바스크립트는 상속을 위해 prototype 객체를 사용한다.
new 키워드 사용 시 만들어지는 객체의 히든 내장속성 __proto__에 prototype chaining 이 설정되어 부모의 prototype 객체와 연결된다.

toString() 함수 스캔 순서

1. shape 객체 내 toString 속성키로 함수를 갖고 있는가?
   없다.
2. shape.__proto__ 내장속성이 가리키는 Shape.prototype 객체 내 toString 속성키로 함수를 갖고 있는가?
   없다.
3. Shape.prototype.__proto__ 내장속성이 가리키는 Object.prototype 객체 내 toString 속성키로 함수를 갖고 있는가?
   있다. → 함수를 실행한다.

예제 2

상속관계 사용법을 보다 자세히 알아보기 위해서 자식 함수를 추가로 만들어보자.

/inheritance/test2.js

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

function Shape() {
	this.height = 10;
	this.width = 10;
	this.area = function () {
		return this.height * this.width;
	};
};

function Triangle() {
```

```
        this.area = function () {
                return (this.height * this.width) / 2;
        }
}

Triangle.prototype = new Shape();

log(Triangle.prototype);
// Shape { height: 10, width: 10, area: [Function] }

log(Triangle.prototype.area.toString());
/*
        function () {
                return this.height * this.width;
        }
*/
```

new 생성자() 사용 시 처리내용

1. new 키워드 작업결과를 대입할 대상이 존재하지 않으면 객체를 새로 만든다.
   이미 존재하는 객체이므로 Triangle.prototype 에 결과를 할당할 것이다.

2. Shape 생성자 함수에 선언된 객체속성(this 키워드로 정의된 환경)을 Triangle.prototype 에 복사한다.

3. Triangle.prototype 객체의 내장속성 __proto__가 사용된 생성자 Shape 가 갖고 있는 prototype 객체를 가리키도록 설정한다. 이 작업이 상속을 위한 프로토타입체이닝 설정이다.

다음 코드를 하단에 추가하고 확인해 보자.

```
var triangle = new Triangle();

log(triangle.__proto__ === Triangle.prototype); // true
log(triangle.height + ", " + triangle.width); // 10, 10
log(triangle.area()); // 50

log(triangle.area.toString());
/*
```

```
        function () {
                return (this.height * this.width) / 2;
        }
*/
```

triangle 객체는 height, width 가 없으므로 프로토타입체이닝을 따라 올라가면서 속성을 찾는다.
triangle.__proto__가 가리키는 객체 Triangle.prototype 이 해당 속성을 갖고 있어 사용할 수 있다.

area 속성은 triangle 이 갖고 있으므로 자신의 것을 사용한다.
이것이 자바스크립트의 재정의 개념이다. 자신의 속성이 부모의 속성보다 우선한다.

예제 3

/inheritance/test3.js

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

function Shape() {
        this.height = 10;
        this.width = 10;
        this.area = function () {
                return this.height * this.width;
        };
};

Shape.perimeter = function () {
        return this.height * 2 + this.width * 2;
};

log(Shape.prototype.constructor.toString());
/*
function Shape() {
          this.height = 10;
          this.width = 10;
          this.area = function () {
                    return this.height * this.width;
```

```
        };
}
*/

var shape = new Shape();

log(shape.hasOwnProperty('perimeter')); // false
log(shape.__proto__.hasOwnProperty('perimeter')); // false
```

함수 선언이 끝난 후 나중에 추가한 속성은 new 키워드로 객체 생성 시 사용되지 않는다.
그래서 shape 객체는 perimeter 속성을 갖고 있지 않는다.
new 키워드로 사용하는 부분은 Shape.prototype.constructor 가 가리키는 함수다.

다음 코드를 하단에 추가하고 확인해 보자.

```
Shape.prototype.perimeter = function () {
        return this.height * 2 + this.width * 2;
}

log(shape.hasOwnProperty('perimeter'));
log(shape.__proto__.hasOwnProperty('perimeter')); // false

log(shape.perimeter()); // 40
```

Shape.prototype 객체에 perimeter 속성을 추가한다.
shape 객체에서 perimeter 속성을 쓰고자 하는 경우 프로토타입체이닝 스캔으로 찾아서 사용할 수 있다.

16. Function

자바스크립트의 함수는 객체다. 하지만 객체가 곧 함수는 아니다.
기능적으로 보자면 함수가 객체보다 좀 더 크다.

객체 vs 함수

객체	함수
프로퍼티(속성)만을 갖는다.	함수는 곧 객체이므로 프로퍼티를 갖을 수 있다. 지역 환경(변수, 함수)을 갖을 수 있다. 상속을 위한 prototype 객체를 별도로 갖는다.

객체가 프로퍼티로 갖고 있는 함수를 메소드라 부른다. 메소드 내부의 함수는 메소드가 아니다.
메소드에서 this 는 객체 자신을 가리키지만 메소드 내부에 포함된 함수의 this 는 전역객체를 가리킨다.

자바스크립트 코드처리 2 단계

1. 변수 및 함수를 최 상단으로 끌어올려서 처리한다.
   이것을 호이스팅이라고 한다. 호이스팅 대상은 변수, 함수다.
2. 코드를 위에서부터 절차적으로 한 라인 씩 처리한다.
   한 라인에서는 오른쪽에서 왼쪽으로 코드를 처리한다.

예제 1 : 호이스팅

함수와 객체의 차이를 정확히 이해하기 위해서 함수 사용법을 살펴보자.

/function/test1.js

```
Start(); // 대문자로 시작하는 함수, 결과: 3

function start() {
        console.log(1);
}

start(); // 소문자로 시작하는 함수, 결과: 2
```

```
function start() { // 함수의 재 정의
	console.log(2);
}

start(); // 소문자로 시작하는 함수, 결과: 2

function Start() {
	var number = 1;
	var Number = 3;
	console.log(Number);
}
```

자바스크립트는 대소문자를 구분한다.

함수가 호이스팅이 되기 때문에 아래에 정의한 함수 Start 를 윗 부분에서 호출할 수 있다.
만약 같은 함수를 반복 정의하면 마지막에 정의된 함수가 사용된다.

예제 2 : 생성자 문법

new 연산자로 만들어지는 것은 객체다. new 연산자 뒤에 지정하는 함수를 생성자라 부른다.

```
var c = new Member;
```

위 코드로 다음 처리가 이루어진다.
1. 새 객체를 만든다.
2. Member 함수 내부에 this 로 정의된 속성을 새 객체에 복사해 넣는다.
	"만들어지는 객체에게 기본 틀을 준다."
3. 새 객체의 __proto__ 프로퍼티가 Member.prototype 객체를 가리키게 설정한다.
	"만들어지는 객체에게 참조로 사용할 수 있는 자원을 준다."

/function/test2.js

```
function Member(name, age, gender) {
	this.name = name || undefined;
	this.age = age || 0;
	this.gender = gender || 'female';

	if (!name) {
		return 'unknown';
```

```
        } else {
                return name;
        }
}

var a = Member; // 변수 a 가 함수 Member 를 가리킨다.
console.log(a); // [Function: Member]

var b = Member(); // 함수를 실행해서 결과를 리턴한다. 리턴결과를 변수 b 가 가리킨다.
console.log(b); // unknown

var c = new Member; // 생성자로 사용하면 괄호를 자동으로 붙여서 처리한다.
console.log(c); // { name: undefined, age: 0, gender: 'female' }

var d = new Member();
console.log(d); // { name: undefined, age: 0, gender: 'female' }

var e = new Member('chris', 33, 'male'); // 정상적인 사용법이다.
console.log(e); // { name: 'chris', age: 33, gender: 'male' }

console.log(c === d); // false, 새로 만들어지는 객체는 언제나 메모리 주소가 다르다. singleton
console.log(JSON.stringify(c) === JSON.stringify(d)); // true, 객체가 갖고 있는 값만을 비교한다.

function isEquivalent(a, b) {
        var aProps = Object.getOwnPropertyNames(a);
        var bProps = Object.getOwnPropertyNames(b);

        if (aProps.length != bProps.length) {
                return false;
        }

        for (var i = 0; i < aProps.length; i++) {
                var propName = aProps[i];
                if (a[propName] !== b[propName]) {
                        return false;
                }
        }

        return true;
```

```
}

console.log(isEquivalent(c, d)); // true, 보다 정확한 값 비교 함수를 사용한다.
```

자바스크립트는 객체의 메모리 주소를 알 수 없다.
다른 메모리 주소에 존재하는지 알기 위해서는 간단히 값+형 비교연산자 === 를 사용한다.

객체가 갖고 있는 값을 비교하기 위해서 JSON.stringify 함수를 사용할 수 있다.

추가로, isEquivalent 함수를 살펴보면 어떻게 두 객체가 같은 값을 갖고 있는지 비교하는 로직의 힌트를 얻을 수 있다.

예제 3 : 1 단계 상속

함수 관점에서 자바스크립트의 상속을 살펴보자.

/function/test3.js

```
var log = console.log;

function Korean() {
	this.food = 'kimchi';
	this.getFood = function () {
		return this.food;
	}
}

var korean = new Korean();

log(typeof Korean === 'function'); // true, 함수
log(typeof Korean === 'object'); // false
// 결과가 false 이긴 하지만 함수는 객체다.
// function 구분작업의 편의를 위한 조치라고 생각하자.
log(Korean instanceof Object); // true
log(Korean.__proto__.__proto__ === Object.prototype); // true
// Object 의 자식이 맞다.

log('~~~~~~~~~~~~~~~~~~~~~~~~~');
```

```
log(korean); // { food: 'kimchi', getFood: [Function] }
log(korean.food); // kimchi
log(korean.getFood()); // kimchi
// this.food 에서 this 를 빼서 클로져를 구성하여 food 를 private 로 쓰거나
// this.getFood 의 로직이 변하지 않으므로 생성자안에 설정하지 말고
// Korean.prototype 객체에 두는 것이 합리적이다.
```

예제 4 : 2 단계 상속

함수는 여러 단계에 걸쳐 상속체인을 설정할 수 있다. 이해를 돕기위해 예제를 살펴보자.

/function/test4.js

```
var log = console.log;

function Korean() {
        this.food = 'kimchi';
}

function Hong() {

}

Hong.prototype = new Korean();

log(Hong.prototype); // { food: 'kimchi' }
log(Hong.prototype.__proto__ === Korean.prototype); // true

log('~~~~~~~~~~~~~~~~~~~~~~~~~~');

var hong = new Hong;

log(hong); // {}
// Hong 생성자안에 this 로 정의된 프로퍼티가 없으므르
// hong 은 프로퍼티가 없는 빈 객체다.

log(hong.food); // kimchi
// hong 이 가리키는 객체에는 food 프로퍼티가 없다.
```

```
log(hong.__proto__); // { food: 'kimchi' }
// 부모가 갖고 있으므로 사용할 수 있다.
```

자바 상속 vs 자바스크립트 상속

자바에서는 Child 클래스 정의 부분에 extends 키워드로 Parent 클래스를 선언하여 상속관계를 설정한다.

- 자바의 상속은 정적이다.
- 자바의 상속은 클래스를 설계도로 사용하여 객체를 만들 때 설정된다.
- 자바의 상속은 객체 생성 후 변경할 수 없다.

자바스크립트에서는 함수(이미 객체)를 만든 후 자식함수와 부모함수의 관계설정을 한다.

- 자바스크립트의 상속은 동적이다.
- 자바스크립트는 객체를 대상으로 상속을 연결하는 설정을 한다.
- 자바스크립트 상속은 나중에 변경할 수 있다.

예제 5 : 자식 객체가 참조하는 프로퍼티를 동적으로 변경

/function/test5.js

```
var log = console.log;

function Korean(){}

Korean.prototype.food = 'kimchi';

function Hong(){}

Hong.prototype = new Korean();

log(Hong.prototype); // {}
log(Hong.prototype.__proto__ === Korean.prototype); // true

log('~~~~~~~~~~~~~~~~~~~~~~~~~~');
```

```
var hong = new Hong();

log(hong.__proto__); // {}
log(hong.__proto__ === Hong.prototype); // true
log(Hong.prototype); // {}
log(hong.__proto__.__proto__ === Korean.prototype); // true
log(Korean.prototype); // { food: 'kimchi' }

log('~~~~~~~~~~~~~~~~~~~~~~~~~~~');

log(hong.food); // kimchi

Korean.prototype.food = 'cheese';
// 자식 객체가 참조하는 프로퍼티를 동적으로 변경

log(hong.food); // cheese
```

함수의 prototype 객체에 설정한 속성은 언제든 동적으로 변경하면 바로 모든 자식 객체에 반영된다. 복사하는 것이 아니라 상속체인을 타고 올라가 사용하는 참조하는 방식이기 때문이다.

예제 6 : 재정의

/function/test6.js

```
var log = console.log;

function Korean(){
	this.num = '1';
}

function Hong(){
	this.num = '2'; // 재정의
}

Hong.prototype = new Korean();

var hong = new Hong();
```

```
log(hong); // { num: '2' }
log(hong.num); // 2

log(hong.__proto__ === Hong.prototype); // true
log(Hong.prototype); // { num: '1' }
```

부모 함수 Korean 에서 정의한 속성의 이름과 똑 같은 이름을 자식 함수에서 사용하면 재 정의한 것이다. 제일 먼저 자신이 갖고 있는 프로퍼티를 사용한다.

예제 7 : 확인

이해를 점검하는 차원에서 비슷한 예제를 한번 더 살펴보자.

/function/test7.js

```
var log = console.log;

function Korean() {
	this.num = '1';
}

Korean.prototype.num = '2';

function Hong() {
	this.num = '3';
}

Hong.prototype.num = '4';

Hong.prototype = new Korean();

log(Hong.prototype);
log(Hong.prototype.__proto__);

log('~~~~~~~~~~~~~~~~~~~~~~~~~~~~');

var hong = new Hong();

log(hong);
```

```
log(hong.__proto__);
log(hong.__proto__.__proto__);

log('~~~~~~~~~~~~~~~~~~~~~~~~~~~~');

log(hong.num);
```

예제 8 : Object.create

/function/test8.js

```
var log = console.log;

function Korean(){
	this.num = 1;
	this.a = 1;
}
Korean.prototype.num = 2;
Korean.prototype.b = 2;

function Hong(){
	this.num = 3;
	this.c = 3;
}
Hong.prototype.num = 4;
Hong.prototype.d = 4;

Hong.prototype = Object.create(Korean.prototype);
//	Hong.prototype = { __proto__: Korean.prototype }; 코드와 같다.
//	1. 새 객체를 만든다.
//	2. create 함수 파라미터로 전달된 객체를 가리키는 히든 속성 __proto__를 추가한다.
//	3. Hong.prototype 이 새로 만들어진 객체를 가리키게 한다.

log(Hong.prototype); // {}
log(Hong.prototype.__proto__ === Korean.prototype); // true

log('~~~~~~~~~~~~~~~~~~~~~~~~~~~~');
```

```
var hong = new Hong();

log(hong);
// { num: 3, c: 3 }
log(hong.__proto__); // === Hong.prototype
// {}
// Hong.prototype = new Korean(); 코드를 사용했다면 결과는
// Korean { num: 1, a: 1 } 가 될 것이다.
log(hong.__proto__.__proto__); // === Korean.prototype
// { num: 2, b: 2 }
```

예제 9 : call

/function/test9.js

```
var log = console.log;

function Korean(){
	this.hobby = 'dance';
}
Korean.prototype.food = 'kimchi';

function Hong(){
	Korean.call(this);
	// 생성자 호출 시 this 는 hong 이다.
	// hong 을 Korean 함수를 call(호출)하면서 넘겨주면
	// Korean 함수에 정의된 this 속성을 hong 이 가리키는
	// 새 객체에 복사해 넣는다.
}

var hong = new Hong();

log(hong);
```

call vs apply

```
function print(name, profession) {
	console.log('name: ' + name + ', profession: ' + profession);
}

print('John', 'fireman');
// name: John, profession: fireman

print.apply(undefined, [ 'Susan', 'school teacher' ]);
// name: Susan, profession: school teacher

print.call(undefined, 'Claude', 'mathematician');
// name: Claude, profession: mathematician

/*
 * apply 함수를 사용해야 할 때
 * if you don't know the number of arguments
 * or if they are already in an array.
 */
```

why use call() or apply() ?

```
var obj = {};

obj.value = 10;
obj.addValues = function(additionalValues) { // method
	for (var i = 0; i < arguments.length; i++) {
		this.value += arguments[i];
	}
	return this.value;
};

console.log(obj.addValues(20));
console.log('obj.value: '+obj.value);

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

global.value = 0;
```

```
var f = obj.addValues; // function

f(20);

console.log('global.value: '+global.value);
console.log('obj.value: '+obj.value);

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

f.call(obj, 30);

console.log('global.value: '+global.value);
console.log('obj.value: '+obj.value);
```

17. Class vs Prototype

Java 같은 클래스 기반의 언어들은 클래스와 인스턴스를 구분한다.
클래스로 필드변수와 메서드 정보를 정의한다. 인스턴스는 클래스를 기반으로 실체화 된 것이다.

클래스 기반의 언어는 클래스가 한번 정의된 후에 클래스를 다시 컴파일 하지 않는다면 속성의 갯수나 형식을 변경할 수 없다.

반면에 자바스크립트같은 프로토타입기반의 언어들은 클래스와 인스턴스의 차이가 없다.

자바스크립트 객체는 생성될 때 뿐 아니라 실행 시에도 자기 자신의 속성을 추가 및 변경할 수 있다. 프로토타입기반의 언어는 prototype 객체로 상속 관계를 처리한다.

기존 객체는 다른 객체를 생성했을 때 기존 객체의 속성을 가질 수 있도록 지원하는 형판(template)으로 사용될 수 있다.

클래스 기반 자바와 프로토타입 기반 자바스크립트 간의 비교

클래스 기반	프로토타입 기반
클래스와 인스턴스는 별개다.	클래스와 인스턴스를 구분하지 않는다. 차이가 없다.
클래스 정의를 가지고 클래스를 생성하고 생성자 메서드로 인스턴스를 생성한다.	생성자 함수를 가지고 객체군을 정의 및 생성한다.
new 연산자로 객체를 생성한다.	new 연산자로 객체를 생성한다.
이미 존재하는 클래스에 대한 하위 클래스를 정의함으로써 객체의 계층구조를 생성한다.	하나의 객체를 생성자 함수와 결합된 프로토타입에 할당함으로써 객체의 계층구조를 생성한다.
클래스의 상속 구조에 따라 속성을 상속 받는다.	프로토타입 체인에 따라 속성을 상속 받는다.
실행시에 동적으로 속성을 추가할 수 없다.	동적으로 속성을 추가 삭제할 수 있다.

18. Multi Inheritance Prohibited

몇몇 객체 지향언어들은 다중 상속을 허용한다. 그것은, 관련이 없는 부모 객체들로 부터 속성들과 값들을 상속 받을 수 있는 것을 말한다. 자바스크립트는 다중 상속을 지원하지 않는다.

속성 값의 상속은 속성에 대한 값을 찾기 위한 프로토타입 체인을 검색에 의해 실행 시점에 이루어 진다. 하나의 객체는 오로지 하나의 결합된 prototype 만을 가지기 때문에, 자바스크립트는 동적으로 하나 이상의 프로토타입 체인으로 부터 상속을 할 수 없다.

자바스크립트에서, 하나 이상의 다른 생성자 함수를 호출하는 생성자를 사용할 수 있다.
이것은 다중 상속처럼 보여질 수 있지만 다중상속이 아니다.

/scope/test3.js

```
function Employee(name, dept) {
        this.name = name || '';
        this.dept = dept || 'general';
}

function Manager() {
        Employee.call(this);
        // 생성자를 통해 새로 만드는 객체에 Employee 생성자가
        // this 로 정의한 프로퍼티를 복사해 넣는다.
        this.reports = []; // 속성 추가
}
Manager.prototype = Object.create(Employee.prototype);
// Manager.prototype = { __proto__ : Employee.prototype }; 코드와 같다.
// 프로토타입 체이닝을 설정해 Manager.prototype 객체에 부모로
// Employee.prototype 객체를 연결한다.

function WorkerBee(name, dept, projs) {
        this.base = Employee;
        this.base(name, dept);
        // Employee 생성자에 name, dept 를 파라미터로 전달한다.
        // this 로 Employee 생성자를 호출 했으므로
        // 생성자를 통해 새로 만드는 객체에 Employee 생성자가 내부에서
        // this 로 정의한 프로퍼티를 복사해 넣는다.

        // 위 두줄의 코드는 Employee.call(this, name, dept); 코드와 비슷하다.
        // 차이는 새로 만들어는 this 가 가리키는 객체에 base 프로퍼티가 없다는 것 뿐이다.
```

```
        this.projects = projs || []; // 속성 추가
        // projs 파라미터는 자신에 속성으로 삼는다.
}
WorkerBee.prototype = new Employee;
// 1. 새 객체를 만든다.
// 2. Employee 생성자에 this 로 정의된 name, dept 를 새 객체에 복사해 넣는다.
// 3. 새 객체에 히든속성 __proto__가 Employee.prototype 객체를 가리키게 한다.
// 4. WorkerBee.prototype 객체가 새로 만든 객체를 가리키게 한다.

// 위 한줄에 코드는 WorkerBee.prototype = Object.create(Employee.prototype);
// 코드와 비슷하다. 차이는 Employee 가 this 로 정의한 속성은 복사하지 않고 WorkerBee.prototype 의
// 부모로 Employee.prototype 객체를 연결만 한다는 것이다.

function SalesPerson() {
        WorkerBee.call(this);
        // 생성자를 통해 새로 만드는 객체에 WorkerBee 생성자가
        // this 로 정의한 프로퍼티를 복사해 넣는다.
        this.dept = 'sales'; // 속성 값 재정의
        // WorkerBee.prototype 객체에 이미 dept 프로퍼티가 존재한다.
        this.quota = 100; // 속성 추가
}
SalesPerson.prototype = Object.create(WorkerBee.prototype);
// SalesPerson.prototype = { __proto__ : WorkerBee.prototype };
// 프로토타입 체이닝을 설정해 SalesPerson.prototype 객체에 부모로
// WorkerBee.prototype 객체를 연결한다.

function Engineer(name, projs, mach) {
        this.base = WorkerBee;
        this.base(name, 'engineering', projs);
        // WorkerBee 생성자에 name, 'engineering', projs 를 파라미터로 전달한다.
        this.machine = mach || ''; // 속성 추가
}
Engineer.prototype = new WorkerBee;
// 1. 새 객체를 만든다.
// 2. WorkerBee 생성자에 this 로 정의된 base, projects 를 새 객체에 복사해 넣는다.
//      새 객체에 프로퍼티 base 가 가리키는 생성자 함수를 바로 실행하고
//      name, 'engineering', projs 를 파라미터로 전달한다.
//      위 같이 처리하면 새로 만들어지는 객체에 WorkerBee 가 this 로 정의한 속성이 추가된다.
```

```
// 3. 새 객체에 히든속성 __proto__가 WorkerBee.prototype 객체를 가리키게 한다.
// 4. Engineer.prototype 객체가 새로 만든 객체를 가리키게 한다.



var chris = new Engineer("chris", ["node"], "robot");
console.log(chris);
//{
//        base: [Function: Employee],
//        name: 'chris',
//        dept: 'engineering',
//        projects: [ 'node' ],
//        machine: 'robot' }
```

앞에 코드가 이해가 되었다면 다음 코드를 추가하고 결과를 다시 확인한다.

```
/*
 * 함수의 재정의
 */
function Engineer(name, projs, mach, hobby) {
        this.base1 = WorkerBee;
        this.base1(name, "engineering", projs);

        this.base2 = HobbyList;
        this.base2(hobby);

        this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
//{
//        base1: [Function: WorkerBee],
//        base: [Function: Employee],
//        name: 'chris',
//        dept: 'engineering',
//        projects: [ 'node' ],
//        base2: [Function: HobbyList],
//        hobby: 'scuba',
//        machine: 'robot' }

HobbyList.prototype.equipment = ["mask", "fins", "regulator", "bcd"];
```

```
console.log(chris.equipment);
// HobbyList.prototype 객체와의 프로토타입 체이닝은 연결되어 있지 않다.
console.log(chris.base2.prototype.equipment);
// 하지만 HobbyList 함수를 갖고 있으므로 접근 할 수 있다.
```

HobbyList 생성자의 프로토타입에 속성을 추가 했다. chris 객체는 새로 추가된 equipment 속성을 상속받지 않는다.

19. This Keyword in JavaScript

자바와 C++등 여타언어에서의 this 는 instance 된 객체 자신을 가리키는 용도로 사용한다.
하지만 자바스크립트의 this 는 기존 언어에서 사용하던 this 와는 다른다.
자바스크립트의 this 는 함수를 호출할 때 사용하는 방식에 따라 참조하는 객체가 다를 수 있다.

this 의 객체 참조 규칙
1. 기본적으로 this 는 전역 객체를 참조한다.
2. 메소드 내부의 this 는 해당 메소드를 호출한 객체를 참조한다.
3. 생성자 함수 코드 내부의 this 는 새로 생성된 객체를 참조한다.
4. call(), apply() 메소드로 함수를 호출할 때 함수의 this 는 첫 번째 인자로 넘겨받은 객체를 참조한다.
5. 프로토타입 객체 내부의 메소드에서도 this 는 해당 메소드를 호출한 객체를 참조한다.
6. 자바스크립트의 this 키워드는 접근제어자 public 의 의미가 있다.

1. 기본적으로 this 는 전역 객체를 참조한다.

자바스크립트의 모든 전역 변수는 전역 객체가 갖고 있는 속성이다.

- 브라우저에서는 window 객체
- 노드에서는 global 객체

2. 메소드 내부의 this 는 해당 메소드를 호출한 객체를 참조한다.

객체의 속성이 함수일 경우, 이 함수를 메소드라고 부른다.
메소드 내부의 this 는 해당 메소드를 호출한 객체를 참조한다.

예제 1

/this/test1.js

```
var counter = {
        count : 0,
        increment : function () {
                this.count += 1;
        }
};

console.log(counter); // { count: 0, increment: [Function] }

counter.increment();
counter.increment();
```

```
console.log(counter.count); // 2
```

예제 2

전역 객체를 참조하는 this 예제를 살펴보자.

/this/test2.js

```
count = 100;

var counter = {
        count : 0,
        increment : function () {
                this.count += 1;
        }
};

var inc = counter.increment;
//변수 inc 는 increment 가 가리키는 함수만을 지칭한다.
inc();

console.log(counter.count); //0
console.log(count); //101
```

counter.increment(); 코드는 메소드를 호출하는 것이다.
inc(); 코드는 함수를 호출하는 것이다.

inc 함수를 호출하게 되면 전역 객체에 포함된 일반 함수가 호출되는 것이다.

```
function () {
        this.count += 1;
}
```

그래서 this 는 전역 객체를 참조하게 되어 전역변수 count 의 값을 1 증가시켜 결과가 101 로 출력된다.

예제 3

이번엔 내부 함수의 this 의 예제다.

/this/test3.js

```
count = 100;
```

```
var counter = {
        count : 1,

        func1 : function () { //method
                this.count += 1;
                console.log('func1() this.count: ' + this.count); // func1() this.count: 2

                func2 = function () { //function
                        this.count += 1;
                        console.log('func2() this.count: ' + this.count); // func2() this.count: 101

                        func3 = function () { //function
                                this.count += 1;
                                console.log('func3() this.count: ' + this.count); // func3() this.count: 102
                        }
                        func3();
                }
                func2();
        }
};

counter.func1();
```

예상했던 결과인 2, 3, 4 가 아닌 2, 101, 102 로 결과가 출력된다.
이러한 결과 때문에 내부 함수에서 this 를 사용할 때 주의해야 한다.

내부 함수도 결국 함수이므로 내부 함수의 this 는 전역 객체를 참조한다.
따라서 func1 메소드에서는 객체 내부의 count 값이 출력되고 func2, func3 내부 함수에서는 전역 변수 count 값이 출력된다.

예제 4

다음은 내부 함수의 this 의 문제를 해결한 예제다.

/this/test4.js

```
//count = 100;

var counter = {
        count : 1,
```

```
        func1 : function () {
                this.count += 1;
                console.log('func1() this.count: ' + this.count); // func1() this.count: 2

                var that = this;

                func2 = function () {
                        that.count += 1;
                        console.log('func2() this.count: ' + that.count); // func2() this.count: 3

                        func3 = function () {
                                that.count += 1;
                                console.log('func3() this.count: ' + that.count); // func3() this.count: 4
                        }
                        func3();
                }
                func2();
        }
};

counter.func1();
```

3. 생성자 함수를 호출할 때 생성자 함수 내부의 this 는 새로 생성된 객체를 참조한다.

/this/test5.js

```
function F(v) {
        this.val = v;
}

var f = new F("constructor");
console.log(f.val); //constructor
```

생성자 함수 코드가 실행되면 빈 객체가 생성된다. 이 객체에 val 프로퍼티가 복사된다.
이 객체는 f 가 가리키게 될 것이다. this 는 f 이므로 f 가 가리키는 객체의 프로퍼티 val 에 "constructor" 값이 대입된다.

4. call(), apply() 메소드로 함수를 호출할 때 this 는 첫 번째 인자로 넘겨받은 객체를 참조한다.

자바스크립트에는 this 를 특정 객체에 명시적으로 바인딩시키는 call(), apply() 메소드를 제공한다.
두 메소드의 기능은 같다. 두 메소드로 함수를 호출할 때, 함수는 첫 번째 인자로 넘겨받은 객체를 this 로 바인딩하고, 두 번째 인자로 넘긴 값은 함수의 인자로 전달된다.

call(대상객체, 파라미터 열거형)
apply(대상객체, [파라미터 배열])

예제 1

/this/test6.js

```
var add = function (x, y) {
        this.val = x + y;
};

obj = {
        val : 0
};

add.call(obj, 2, 8);
console.log(obj.val); //10

add.apply(obj, [2, 8]);
console.log(obj.val); //10
```

위 add 함수는 범용성을 위해 로직을 보강하는 것이 좋다.

```
var add = function (x, y) {
        if (this === global) {
                return x + y;
        }
        this.val = x + y;
};
```

5. 프로토타입 객체 메소드 내부의 this 는 메소드를 호출한 객체를 참조한다.

/this/test7.js

```
var log = console.log;

function Person(name) {
	this.name = name;
}

Person.prototype.getName = function () {
	return this.name;
}

var foo = new Person('foo');

log(foo.getName()); // foo
// getName 메소드의 호출자는 foo 다.
// foo 가 가리키는 객체에는 프로퍼티 name 이 있다.

log(foo.__proto__ === Person.prototype); // true
log(Person.prototype.getName()); // undefined
// 메소드의 this 는 호출자이므로 getName 메소드 안에서 this 는
// Person.prototype 이 된다.
// Person.prototype 객체 내 name 은 정의되어 있지 않다.

log('~~~~~~~~~~~~~~~~~~~~~~~~~~~');

log(Person.prototype.getName.call(foo)); // foo
log(Person.prototype.getName.apply(foo)); // foo
// 호출자를 명시적으로 지정하는 메소드 call 또는 apply 를 쓰면
// 때에 따라 편리하다.
// getName 메소드를 호출한 호출자로 foo 객체를 주고 있다.

log('~~~~~~~~~~~~~~~~~~~~~~~~~~~');

Person.prototype.name = 'boo';

log(Person.prototype.getName()); // boo
// Person.prototype 객체에 프로퍼티가 있다면 사용된다.
```

foo 객체에서 getName() 메소드를 호출하면, getName() 메소드는 foo 객체에서 찾을 수 없으므로 프로토타입 체이닝을 따라 올라가면서 찾는다. foo 객체의 프로토타입 객체인 Person.prototytpe 에 getName() 메소드가 있으므로 이 메소드가 호출된다.

이때, getName() 메소드를 호출한 객체는 foo 이므로 this 는 foo 객체를 참조한다.

따라서 foo.getName()의 결과로 foo 가 출력된다. Person.prototype.getName() 메소드를 바로 호출하면 getName() 메소드를 호출한 객체가 Person.prototype 이므로 this 는 여기에 바인딩 되고 이 객체 안에는 name 속성이 없으므로 undefined 가 리턴된다.

6. 자바스크립트의 this 키워드는 접근제어자 public 의 의미가 있다.

자바스크립트에서는 public, private 키워드 자체를 지원하지 않지만 public 역할을 하는 this 키워드와 private 역할을 하는 var 키워드가 있다.

자바스크립트에서는 이 두 키워드를 사용하여 캡슐화를 구현할 수 있다.

/this/test8.js

```
function Person(name, age) {
	var name = name;
	this.age = age;
	this.getName = function () {
		return name;
	}
}
var foo = new Person('foo', 26);
console.log(foo.age); // 26
console.log(foo.name); // undefined
console.log(foo.getName()); // foo
```

이렇게 감추고 싶은 멤버는 var 키워드로 감추고 this 키워드로 외부로 노출시켜 캡슐화를 구현할 수 있다.

20. Array

예제 1

```
var log = console.log;

var fruits = [ '1', '2', '3', '4' ];

log(fruits);
log(fruits.toString());

log('~~~~~~~~~~~~~~~~~~~~~~~~join');

log(fruits.join(','));
log(fruits.join(' '));

log('~~~~~~~~~~~~~~~~~~~~~~~~pop');

log(fruits.pop());
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~push');

fruits.push('5');
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~shift');

fruits.shift();
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~unshift');

fruits.unshift('x');
log(fruits);
```

예제 2

```
var log = console.log;

var fruits = [ '1', '2', '3', '4' ];
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~index overwrite');

fruits[0] = "Kiwi";
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~index push');

fruits[fruits.length] = "Kiwi";
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~delete');

delete fruits[0];
// 항목을 제거하는 것이 아니라 undefined 로 바꾼다.
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~');

fruits = [ '1', '2', '3', '4' ];
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~splice');

fruits.splice(fruits.length-1, 1);
// splice(시작위치, 제거개수)
log(fruits);

fruits.splice(2, 0, "Lemon", "Kiwi");
// splice(시작위치, 제거개수, 추가항목...)
log(fruits);

fruits.splice(0, 1);
log(fruits);
```

예제 3

```
var log = console.log;

var fruits = [ '2', '4', '1', '3' ];
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~sort asc');

fruits.sort();
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~');

fruits = [ '2', '4', '1', '3' ];
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~sort desc');

fruits.sort();
fruits.reverse();
log(fruits);

log('~~~~~~~~~~~~~~~~~~~~~~~~reverse');

fruits.reverse();
log(fruits);
```

예제 4

```
var log = console.log;

var points = [40, 100, 1, 5, 25, 10];
log(points);

log('~~~~~~~~~~~~~~~~~~~~~~~~sort as string');
```

```
points.sort();
// 항목을 문자열로 변경해서 정렬한다.
log(points);

log('~~~~~~~~~~~~~~~~~~~~~~~~');

points = [40, 100, 1, 5, 25, 10];
log(points);

log('~~~~~~~~~~~~~~~~~~~~~~~~sort as number order asc');

points.sort(function(a, b){return a>b});
// 비교함수를 사용한다.
log(points);

log('~~~~~~~~~~~~~~~~~~~~~~~~');

points = [40, 100, 1, 5, 25, 10];
log(points);

log('~~~~~~~~~~~~~~~~~~~~~~~~sort as number order desc');

points.sort(function(a, b){return a<b});
log(points);

log('~~~~~~~~~~~~~~~~~~~~~~~~');

points = [40, 100, 1, 5, 25, 10];
log(points);

log('~~~~~~~~~~~~~~~~~~~~~~~~find highest');

points.sort(function(a, b){return a<b});
log(points[0]);

log('~~~~~~~~~~~~~~~~~~~~~~~~');

points = [40, 100, 1, 5, 25, 10];
log(points);
```

```
log('~~~~~~~~~~~~~~~~~~~~~~~~find lowest');

points.sort(function(a, b){return a>b});
log(points[0]);
```

예제 5

```
var log = console.log;

var nums = [ "1", "2" ];
var strs = [ "a", "b", "c" ];

var result = nums.concat(strs);
log(result);

log('~~~~~~~~~~~~~~~~~~~~~~~~slice');

var fruits = [ "Banana", "Orange", "Lemon", "Apple", "Mango" ];
var citrus = fruits.slice(1);
// 인덱스 1 번째 자리부터 잘라서 새로운 배열을 만든다.

log(fruits);
log(citrus);

log('~~~~~~~~~~~~~~~~~~~~~~~~indexOf');

fruits = [ "Banana", "Orange", "Apple", "Mango" ];
log(fruits.indexOf("Apple"));

fruits = [ "Banana", "Orange", "Apple", "Mango", "Banana", "Orange", "Apple" ];
log(fruits.indexOf("Apple", 4));
// 인덱스 4 번째 자리부터 서치한다.
```

reduce

```
/**
 * array1.reduce(callbackfn[, initialValue])
 * 배열의 모든 요소에 대해 지정된 콜백을 호출한다.
 * 콜백 함수의 반환 값은 결과에 누적되며 다음에 콜백 함수를 호출할 때 인수로 제공된다.
 * 최종 반환 값은 콜백 함수에 대한 마지막 호출로부터 누적된 결과다.
 * callbackfn
 * 필수 요소. 최대 4 개까지 인수를 허용하는 함수다.
 * initialValue
 * 선택 사항. initialValue 가 지정된 경우 누적을 시작하는 초기 값으로 사용된다.
 * callbackfn 함수에 대한 첫 번째 호출은 이 값을 배열 값 대신 인수로 제공한다.
 */

function appendCurrent (previousValue, currentValue) {
        return previousValue + ':' + currentValue;
}

var elements = ['abc', 'def', 123, 456];
var result = elements.reduce(appendCurrent, 'initialValue');

console.log(result);
console.log('~~~~~~~~~~~~~~~~~~~~~~');
function addDigitValue(previousValue, currentDigit, currentIndex, array) {
        console.log(previousValue+','+currentDigit+','+currentIndex+',['+array+']');

        var exponent = (array.length - 1) - currentIndex;
        console.log('exponent: '+exponent);
        var digitValue = currentDigit * Math.pow(10, exponent);
        console.log('Math.pow(10, exponent): '+Math.pow(10, exponent));

        console.log('return: '+(previousValue + digitValue));
        return previousValue + digitValue;
}

var digits = [4, 1, 2, 5];
var result = digits.reduce(addDigitValue, 0);

console.log(result);
```

21. Review

프로토타입 체인이 다음과 같이 생겼다고 가정하자.
var o = { a:1, **b:2**, __proto__:O.prototype } → O.prototype { **b:3**, c:4 }

console.log(o.a); // 1
console.log(o.b); // 2

o 의 프로토타입 역시 'b'라는 속성을 가지지만 이 값은 쓰이지 않는다.
이것을 속성의 가려짐(property shadowing) 이라고 부른다.

console.log(o.c); // 4

o 는 'c'라는 속성을 가지는가? 아니다. o 의 프로토타입은 'c'라는 속성을 가지고 있다.

console.log(o.d); // undefined

속성이 발견되지 않았기 때문에 undefined 를 반환한다.

함수가 오브젝트의 속성으로 지정되었을 때 하나 다른점은 함수가 실행 될 때 this 라는 변수가 가리키는 값이다.

예제 1

/prototype/review1.js

```
var o = {
        a : 2,
        m : function (b) {
                return this.a + 1;
        }
};

console.log(o.__proto__ === Object.prototype); //true
console.log(o.m()); // 3
// o.m 을 호출하면 'this' 는 o 를 가리킨다.

var p = Object.create(o);
console.log(p); //object
```

```
console.log(p.__proto__); //{ a: 2, m: [Function] }
console.log(p.__proto__ === o); //true

p.a = 12; // p 에 'a'라는 새로운 속성을 만들었다.

console.log(p.m()); // 13
// p.m 이 호출 될 때 'this' 는 'p'를 가리킨다.
```

예제 2

/prototype/review2.js

```
var o = {
        a : 1
};

console.log(o.__proto__ === Object.prototype);
console.log(o.hasOwnProperty('a'));
// o 오브젝트는 프로토타입으로 Object.prototype 을 가진다.
// 이로 인해 o.hasOwnProperty('a') 같은 코드를 사용할 수 있다.
// hasOwnProperty 메소드는 Object.prototype 객체가 갖고 있다.

console.log('-----------------------');

var a = ["yo", "whatsup", "?"];

console.log(a.__proto__ === Array.prototype);
console.log(Array.prototype.__proto__ === Object.prototype);
// Array.prototype 을 상속받은 배열도 마찬가지다.
// 이 프로토타입은 indexOf, forEach 등의 메소드를 가진다.
// 프로토타입 체인은 다음과 같다.
// a ---> Array.prototype ---> Object.prototype ---> null

console.log('-----------------------');

function f() {
        return 2;
}
```

```
console.log(f.__proto__ === Function.prototype);
console.log(Function.prototype.__proto__ === Object.prototype);
// 함수는 Function.prototype 을 상속받는다.
// 이 프로토타입은 call, bind 같은 메소드를 가진다.
// f ---> Function.prototype ---> Object.prototype ---> null

console.log('-----------------------');

// 자바스크립트에서 생성자는 new 연산자를  사용할 때 호출되는 함수이다.
function Graph() {
	this.vertexes = [];
	this.edges = [];
}

Graph.prototype = {
	addVertex : function (v) {
		this.vertexes.push(v);
	}
};

var g = new Graph();

console.log(g);
// g 는 'vertexes'와 'edges'를 속성으로 가지는 오브젝트다.

console.log("g.vertexes.length: " + g.vertexes.length);
g.addVertex(1);
g.addVertex(2);
console.log("g.vertexes.length: " + g.vertexes.length);

console.log('-----------------------');

// ECMAScript 5 에서 새로운 방법을 도입했다.
// Object.create 라는 메쏘드를 호출하여 새로운 오브젝트를 만들 수 있다.
// 생성된 오브젝트의 __proto__ 속성은 create 메소드의 첫 번째 파라미터인 객체를 가리킨다.
var a = {
	a : 1
};
// a ---> Object.prototype ---> null
```

```
var b = Object.create(a);
// b ---> a ---> Object.prototype ---> null

console.log(b); // 비여 있다.
console.log(b.a); // 1, 프로토타입체이닝을 통해 상속받는다.

var d = Object.create(null);
// d ---> null

console.log(d.hasOwnProperty); // undefined
// undefined, d 는 Object.prototype 을 상속받지 않는다.

console.log('-----------------------');

//객체의 프로퍼티를 순회할 때는 프로토타입 체인을 따라 상속되는 프로퍼티들을 걸러내기 위해
//hasOwnProperty() 메서드를 사용해야 한다.
var man = {
        hands : 2,
        legs : 2,
        heads : 1
};

// 어디선가 부모 프로토타입 객체에 메서드 하나를 추가한다.
// 프로토타입 체인의 변경사항은 실시간으로 반영되기 때문에, 자동적으로 모든 객체가
// 이 새로운 메서드를 사용할 수 있다.
if (typeof Object.prototype.clone === 'undefined') {
        Object.prototype.clone = function () {};
}

// man 을 열거할 때 clone() 메서드가 나오지 않게 하려면
// 부모 프로토타입 객체의 프로퍼티를 걸러내야 한다.
for (var i in man) {
        if (man.hasOwnProperty(i)) {
                console.log(i, ":", man[i]);
        }
}
```

예제 3

call 메소드를 언제 쓰면 좋을까? 예를 들어 다른 객체에 존재하는 메소드를 써야 하는데 굳이 상속관계를 맺기는 싫고 잠시 빌려쓰려 할 때가 아닌가 한다.

```
var Robert = {
        name : "Robert",
        age : 21,
        height : "178",
        sex : "male",
        describe : function() {
                return "This is me " + this.name + " " + this.age + " " + this.height
                                + " " + this.sex;
        }
};

var Richard = {
        name : "Richard",
        age : 25,
        height : "185",
        sex : "male",
}

console.log(Robert.describe.call(Richard));
// This is me Richard 25 185 male
```

22. Miscellaneousness

유용한 함수들을 살펴보자.

예제 1 : 랜덤 문자열 구하기

/util/random-key-generator.js

```
function RandomString(length) {
	var str = '';
	for (; str.length < length; str += Math.random().toString(36).substr(2));
	return str.substr(0, length);
}

console.log(RandomString(2));
console.log(RandomString(2));
console.log(RandomString(2));

function makeid() {
	var text = "";
	var possible = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

	for (var i = 0; i < 5; i++)
		text += possible.charAt(Math.floor(Math.random() * possible.length));

	return text;
}

console.log(makeid());
console.log(makeid());
console.log(makeid());
```

예제 2 : 로또 번호 구하기

/util/lotto.js

```
MMM = {};
MMM.oneGameNumberCount = 6;
MMM.maxNumber = 45;
MMM.minNumber = 1;

function getNumber() {
	return Math.floor(Math.random() * MMM.maxNumber) + MMM.minNumber;
}

function getOneGameLottoNumbers(){
	'use strict';
	var lotto = [];
	var tmpNum = 0;
	var isDuplicated = false;

	for(var i=0; i < MMM.oneGameNumberCount; ){
		tmpNum = getNumber();
		for (var j = 0; j < lotto.length; j++) {
			if (tmpNum === lotto[j]) {
				isDuplicated = true;
				break;
			}
		}

		if (isDuplicated) {
			isDuplicated = false;
		} else {
			lotto[i] = tmpNum;
			i++;
		}
	}

	lotto.sort(function(a, b){ return a > b; });
	return lotto;
}
```

```
function startLottoGame(games){
	games = games || 1;
	for (var i = 0; i < games; i++) {
		var oneGameLottoNumbers = getOneGameLottoNumbers();
		var printNumbers = [];
		for (var j = 0; j < oneGameLottoNumbers.length; j++) {
			printNumbers.push(pad(oneGameLottoNumbers[j], 2));
		}
		console.log(printNumbers);
	}
}

function pad(num, width, x) {
	x = x || '0';
	num = num + '';
	return num.length >= width ? num : new Array(width - num.length + 1).join(x) + num;
}

startLottoGame(5);
```

예제 3 : 자바의 StringBuffer 클래스 흉내 내기

/util/string-buffer.js

```
var StringBuffer = function () {
	this.buffer = new Array();
};

StringBuffer.prototype.append = function (str) {
	this.buffer[this.buffer.length] = str;
};

StringBuffer.prototype.toString = function () {
	return '['+this.buffer.join(",")+']';
};

var sb = new StringBuffer();

console.log(sb);
```

```
sb.append('apple');
sb.append('kiwi');
sb.append('banana');

console.log(sb.toString());
```

예제 4 : DOM 모델의 Form 에서 배열처리를 위한 편의성 함수 serializeObject

/util/serializeObjectDemo.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>serializeObject</title>
<script src="//code.jquery.com/jquery-2.1.1.min.js"></script>
<script type="text/javascript">

	$.fn.serializeObject = function() {
		var o = {};
		var target = [];
		target.push('input[type="hidden"]');
		target.push('input[type="text"]');
		target.push('input[type="password"]');
		target.push('input[type="checkbox"]:checked');
		target.push('input[type="radio"]:checked');
		target.push('select');

		$(this)
		.find(target.join(','))
		.each(function() {
			//if checkbox is checked do not take the hidden field
			if ($(this).attr('type') == 'hidden') {
				var $parent = $(this).parent();
				var $chb = $parent
						.find('input[type="checkbox"][name="'
								+ this.name.replace(/₩[/g, '₩[').replace(/₩]/g, '₩]')
								+ '"]');
```

```
                        if ($chb != null) {
                                if ($chb.prop('checked'))
                                        return;
                        }
                }
                if (this.name === null || this.name === undefined
                                || this.name === '')
                        return;
                var elemValue = null;
                if ($(this).is('select'))
                        elemValue = $(this).find('option:selected')
                                        .val();
                else
                        elemValue = this.value;
                if (o[this.name] !== undefined) {
                        if (!o[this.name].push) {
                                o[this.name] = [ o[this.name] ];
                        }
                        o[this.name].push(elemValue || '');
                } else {
                        o[this.name] = elemValue || '';
                }
        });

        return o;
}

$(function() {
        $("button").click(function() {
                $("#div1").text($("form").serialize());
                $("#div2").text(JSON.stringify($("form").serialize()));

                $("#div3").text($("form").serializeArray());
                $("#div4").text(JSON.stringify($("form").serializeArray()));

                //한계 : 배열처리 안 됨
                $("#div5").text(getFormData($("form")));

                //추천방법
```

```
                $("#div6").text(JSON.stringify($("form").serializeObject()));

        });
    });

    function getFormData(form) {
        var config = {};
        form.serializeArray().map(function(item) {
            config[item.name] = item.value;
        });
        return JSON.stringify(config);
    }
</script>
</head>
<body>
    <form name="myform" action="">
        first name: <input type="text" name="FirstName" value="Mickey"><br>
        last name: <input type="text" name="LastName" value="Mouse"><br>
        hobby: <input type="checkbox" name="hobby" value="sleep">sleep 
        <input type="checkbox" name="hobby" value="drink">drink 
        <input type="checkbox" name="hobby" value="tv">tv 
    </form>
    <hr>

    <button>Serialize form values</button>
    <hr>

    <b>1 : $("form").serialize()</b><br>
    <div id="div1"></div><br>
    <b>2 : JSON.stringify($("form").serialize())</b><br>
    <div id="div2"></div>
    <hr><br>

    <b>3 : $("form").serializeArray()</b><br>
    <div id="div3"></div><br>
    <b>4 : JSON.stringify($("form").serializeArray())</b><br>
    <div id="div4"></div>
    <hr><br>
```

```
        <b>5 : getFormData($("form"))</b><br>
        <div id="div5"></div>
        <hr><br>

        <b style="color:red;">6 : JSON.stringify($("form").serializeObject())</b><br>
        <div id="div6"></div>
        <hr><br>

</body>
</html>
```

추천 사이트

JavaScript is one of the world's most popular programming languages.
JavaScript is the world's most misunderstood programming Language.
Douglas Crockford

http://www.crockford.com/javascript/javascript.html

수고하셨습니다.


WE'RE LOOKING FOR SOMEONE AGE 22-26
WITH 30 YEARS OF EXPERIENCE