ECMAScript 6

A bright new future is coming...

softcontext@gmail.com

1. ECMAScript

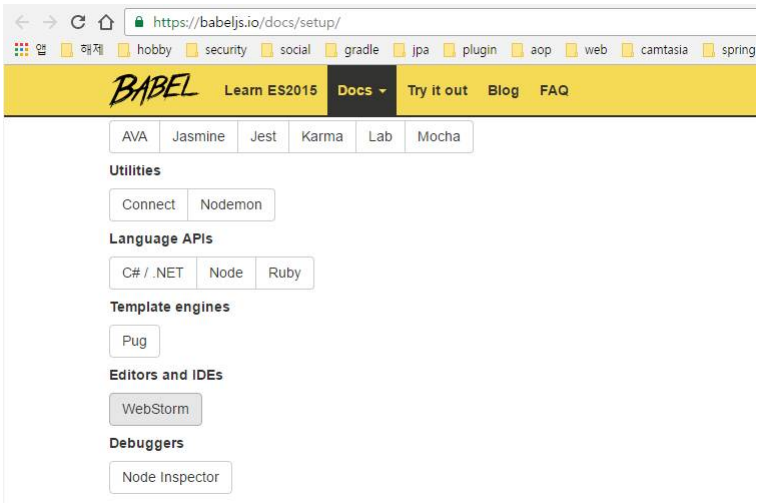
ES6 는 ECMAScript 6 의 줄임말이다. ECMAScript 2015 라고도 부른다. ES7 은 ECMAScript 2016 이라 부른다. 아직까지 대부분의 브라우저는 새로운 문법을 지원하지 않는다. 그럼에도 불구하고 많은 개발자들이 ES6, 7 의 문법을 사용하여 개발할 수 있는 이유는 트랜스파일링 도구 덕분이다. 새 문법으로 작성된 코드를 트랜스파일링 작업을 통해 현 브라우저가 지원하는 ES5 문법으로 변경하여 배포한다.
인기있는 트랜스파일링 도구인 바벨은 ES6 문법을 ES5 호환 JS 파일로 변환해 주는 대표적인 기술이다. 따라서 리액트, 앵귤러 등에서 ES6 문법을 사용하여 개발할 수 있다.

2. ES6 개발환경 설정

■ Node 설치 : JS 의 단독 실행환경 + 기본 라이브러리(fs, http 모듈 등)
■ Git 설치 : Git CMD, Git Bash(리눅스 명령어 사용가능)
■ WebStorm 설치 : IDE Tool

WebStorm 바벨 설정

참고: https://babeljs.io/docs/setup/



"WebStorm" 버튼을 클릭하면 바벨 설정안내 페이지로 이동한다.

Installation

```
npm install --save-dev babel-cli
```

Usage

In **Preferences - Tools - File watchers**, **click + button** and **select Babel file watcher** from the list.

Specify the path to Babel executable and click Ok.

By default all files with a **.js** extension will be automatically compiled with Babel upon change. The generated ES5 files and source maps will be saved next to original files.

Lastly, in **Languages & Frameworks - JavaScript - JavaScript language version**, **choose ECMAScript 6**.

Create .babelrc configuration file

Great! You've configured Babel but you haven't made it actually do anything. Create a .babelrc config in your project root and enable some plugins.

To start, you can use the the latest preset, which enables transforms for ES2015+

npm install babel-preset-latest --save-dev

In order to enable the preset you have to define it in your .babelrc file, like this:

```
{
  "presets": ["latest"]
}
```

Note: Running a Babel 6.x project using npm 2.x can cause performance problems because of the way npm 2.x installs dependencies. This problem can be eliminated by either switching to npm 3.x or running npm 2.x with the dedupe flag. To check what version of npm you have run

npm --version

바벨 트랜스파일링

깃쉘에서 다음 명령을 사용하여 트랜스파일링 직접할 수 있다
대상 ES6.js 를 트랜스파일링한 후 디렉토리 build 밑에 생성하는 명령이다.

babel --presets latest **ES6.js** -d build

3. var

변수 선언 시 사용하는 키워드인 var 연산자의 문제점을 파악해 보자.

var1.js

```
// var 키워드로 선언한 변수는 function scoped variable 이다.
// 유독 JS 만이 블록연산자 스코프 변수라는 개념이 없었다.

var a = 1; // 전역 접근 가능
console.log('a = '+a);

(function some(){
    console.log('a = '+a);

    var b = 2; // 함수안에서 접근 가능
    console.log('b = '+b);

    if(true){
        console.log('b = '+b);

        var c = 3;
        console.log('c = '+c);
    }

    console.log('c = '+c); // 접근 가능!
})();

// console.log('b = '+b); // 에러
// console.log('c = '+c); // 에러

// #출력결과
// a = 1
// a = 1
// b = 2
// b = 2
// c = 3
// c = 3
```

var2.js

```
// if, for 블록연산자는 var 키워드로 선언된 변수의 스코프를 제한하지 못했다.
// 함수의 블록연산자만이 var 키워드로 선언된 변수의 스코프를 제한한다.

for(var i=1; i <= 3; i++){
      console.log('i = '+i);
}

console.log('i = '+i); // 접근 가능!

// #출력결과
// i = 1
// i = 2
// i = 3
// i = 4
```

var3.js

```
var a = 1;
var a = 2; // 심지어 재선언도 가능!

console.log('a = '+a);

let b = 1;
// let b = 2; // 에러, 변수 재선언 불가!
```

var4.js

```
var a = 1;
let b = 1;

(function some() {
    var a = 2; // 다른 변수
    let b = 2; // 다른 변수

    if(true){
        var a = 3; // 변수 재선언!
        let b = 3; // 다른 변수
        console.log('a = '+a); // 3
        console.log('b = '+b); // 3
    }

    console.log('a = '+a); // 3 (재선언 변수 사용)
    console.log('b = '+b); // 2
})();

console.log('a = '+a); // 1
console.log('b = '+b); // 1
```

4. let

변수 선언 시 사용하던 var 키워드는 조건문의 범위연산자를 무시하여 버그를 양산하던 주범이었다.
변수 선언 시 let 키워드를 사용하면 다른 언어들과 마찬가지로 철저하게 범위 연산자를 "{ }"에 따라 변수의 스코프가 결정된다. 따라서 자바의 범위연산자의 스코프와 같게 된다.

let1.js

```
// let 키워드로 변수를 선언하면 범위연산자에 따른 스코프를 갖는다.
// 이제 JS 도 변수의 스코프가 다른 언어와 같아졌다.

let a = 1; // 파일 내 접근 가능
console.log('a = '+a); // 1

(function some(){
    console.log('a = '+a); // 1

    let b = 2; // 함수안에서 접근 가능
    console.log('b = '+b); // 2

    if(true){
        console.log('b = '+b); // 2

        let c = 3;
        console.log('c = '+c); // 3
    }

    // console.log('c = '+c); // 에러, 접근 불가!
})();
```

let2.js

```
// 모든 블록연산자는 let 키워드로 선언된 변수의 스코프를 제한한다.

for(let i=1; i < 3; i++){
	console.log('i = '+i);
}

// console.log('i = '+i); // 에러, 접근 불가!
```

5. const

드디어 자바스크립트에서도 상수를 사용할 수 있게 되었다. 최초 할당 이후에는 값의 변경이 허용되지 않는다.

const.js

```
var PI = 3.141592;
console.log('PI = '+PI);

PI = 3.14;
console.log('PI = '+PI);

// 변수는 대소문자를 구분한다.
const pi = 3.141592;
console.log('pi = '+pi);

// pi = 3.14; // error!

// --------------------------
// # 상수 변수에 객체를 할당

const a = {name:'Chris'};
console.log('a = '+JSON.stringify(a));

a.name = 'Aaron';
console.log('a = '+JSON.stringify(a));

// a 는 객체의 주소(참조값)를 가리킨다.
// a 는 상수로 참조값은 변하지 않지만 객체의 프로퍼티는 변경할 수 있다.
```

6. parameter

자바스크립트에서는 함수 호출은 함수명만으로 연동된다. 함수가 받는 파라미터는 옵션으로써 명시해도 되고 선언하지 않아도 된다. 이에 따라 많은 개발자들이 파라미터를 생략하곤 했었다. 심지어 라이브러리 개발자들도 여기에 포함된다.

문제는 라이브러리 함수를 이용하는 컨슈머 개발자들의 입장에서는 파라미터의 생략이 불편함을 초래한다는 점이다. 대부분의 개발자들이 IDE 툴을 사용하여 개발을 한다. 그런데 그 동안 IDE 툴들이 자동완성 기능을 제공하지 못했었다. 함수 선언 시 파라미터가 선언되어 있지 않으면 아무리 좋은 IDE 툴이라 하더라도 도움말 및 자동완성 기능을 제공할 수가 없기 때문이다.

따라서 다른 개발자들이 이용하는 함수를 만드는 개발자는 반드시 파라미터를 명시하는 것이 좋다라는 결론에 다다르게 된다. 이에 따라 파라미터를 명시하는 타입스크립트가 등장했다.

다음 예제에서 만약 파라미터를 주지 않고 함수를 호출하는 경우 편리하게 초기값을 설정해서 사용할 수 있는 새로운 문법을 살펴본다.

parameter.js

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

// 함수가 파라미터를 못 받는 경우 대신 기본값을 설정할 필요가 있다
console.log(add()); // NaN


// 연산자 || 를 사용하여 받은 파라미터가 없는 경우 대신 기본값을 할당한다.
function add2(x, y){
    x = x || 0;
    y = y || 0;
    return x + y;
}

console.log(add2()); // 0



// 새로운 문법으로 더 쉽게 파라미터를 받지 못하면 대신 기본값을 사용하도록 설정한다.
function add3(x = 0, y = 0){
    return x + y;
}
```

```
console.log(add3()); // 0


// 심지어 파라미터 자리에서 간단한 연산도 된다!
function add4(x = 0, y = 0, z = x+y){
    return z;
}

console.log(add4()); // 0
console.log(add4(1, 2)); // 3
console.log(add4(1, 2, 999)); // 999
```

7. spread operator

스프레드 연산자 또는 펼침 연산자라고 부른다. 배열의 요소를 하나씩 나누어 처리할 수 있는 편리한 문법이다.

spread.js

```
function add(a, b) {
    return a + b;
}

let data = [1, 2];



// 배열 요소를 직접 하나씩 꺼내서 파라미터로 전달한다. 불편하다!
console.log(add(data[0], data[1])); // 3



// apply 메소드의 두번째 파라미터로 배열을 전달하면서 함수 add 를 호출한다.
// 배열의 아이템들이 많을수록 이전 방법 보다는 편리하다.
console.log(add.apply(null, data)); // 3



// 스프레드 연산자 ...를 사용한다. 배열의 아이템들을 꺼내고 펼쳐서 전달한다.
// 1, 2 로 치환한 다음 add 함수를 호출한다.
console.log(add(...data)); // 3

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

// #다른 사용예

let array1 = [2, 3, 4];
// 배열 array1 을 분해해서 array2 에 넣는다.
let array2 = [1, ...array1, 5];

console.log(array2); // [ 1, 2, 3, 4, 5 ]

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

```
// #다른 사용예

array1 = [1, 2];
array2 = [3, 4];


// 두 배열을 붙인다.
let array3 = array1.concat(array2);
console.log(array1); // [ 1, 2 ]
console.log(array2); // [ 3, 4 ]
console.log(array3); // [ 1, 2, 3, 4 ]


// push 함수를 사용하면 array2 가 가리키는 배열이 통째로 들어간다.
array1.push(array2);
console.log(array1); // [ 1, 2, [ 3, 4 ] ]


// reset
array1 = [1, 2];
array2 = [3, 4];


// array2 배열의 아이템을 꺼내고 펼쳐서 array1 배열에 추가한다.
array1.push(...array2);
console.log(array1); // [ 1, 2, 3, 4 ]

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

// #rest parameter
// 파라미터 자리에 쓰이면 스프레드 연산자란 용어 대신
// 레스트 파라미터라고 부른다.


// 세 번째 파라미터 3 은 사용되지 않는다.
function subtract(a, b) {
    return a - b;
}

// 전달 받은 파라미터 중 맨 앞에 하나를 변수 a 에 할당한다.
// 다른 파라미터는 레스트 파라미터 문법으로 설정하여 args 배열이 갖고 있다.
function calc(a, ...args){
    console.log(Array.isArray(args)); // true
```

```
    console.log(args); // [ 1, 2, 3 ]

    switch(a){
        case '+':
            return add(...args);
        case '-':
            return subtract(...args);
        default:
            return 0;
    }
}

console.log(calc('+', 1, 2, 3)); // 3
console.log(calc('-', 1, 2, 3)); // -1
```

8. Destructuring Assignment

해체할당이라고 번역해서 부른다.

destructuring-assignment.js

```
var log = console.log;

var myObj = {a: 1, b: 2, c: 3};

// 객체.프로퍼티 문법으로 접근해서 하나씩 할당한다. 불편하다!
var aa = myObj.a;
var bb = myObj.b;

log('aa = ' + aa); // 1
log('bb = ' + bb); // 2

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

// 객체는 프로퍼티명을 사용하여 해체할당 문법을 사용할 수 있다.
// 사용된 변수 a, c 는 반드시 객체의 프로퍼티명과 같아야 한다.
var {a, c} = myObj;

log('a = ' + a); // 1
log('c = ' + c); // 3

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

var array = [1, 2, 3];

// 배열은 인덱스기반으로 해체 할당할 수 있다.
var [x1, , x3] = array;

log('x1 = ' + x1); // 1
log('x3 = ' + x3); // 3
```

9. Module System

옛날 옛적에 호랑이 담배피던 시절에는 JS 코드를 담고 있는 파일을 다른 JS 파일에서 사용할 수 없었다. 어처구니 없지만 사실이다! 하는 수 없이 HTML 에 도움을 받아 결합해서 사용해야만 했다.

HTML 파일에서 .js 확장자의 파일들을 임포트하면 JS 파일들은 하나의 파일처럼 취급된다. 이 방법에 문제점은 파일이 많아지거나 여러 개발자들이 나누어서 개발한다면 변수의 충돌이 빈번하다는 점이다.
이에 따라 글로벌 스코프 변수를 사용하는 것음 금기가 되었고 스코프를 완벽하게 분리하는 방법이 필요했다.

불편함을 느낀 많은 개발자들이 앞 다투어 자바스크립트의 치명적인 약점을 개선하고자 했다. 이에 따라 JS 코드를 여러 파일로 분리해서 개발하고 조립하여 사용하는 모듈시스템들이 속속 발표되었다.

자바스크립트에게 자유를 안겨 준 노드는 CommonJS 방식의 모듈시스템을 사용한다.
CommonJS 는 스펙이고 구현체는 require.js 라이브러리이다.

한편 표준 단체인 ECMA 는 ES6 에서 export, import 로 대표되는 모듈시스템을 지원하기 시작했다.
노드를 사용하여 자바스크립트를 실행하는 서버환경이라면 require 문법을 사용하는 것이 좋다.

문제는 클라이언트 사이드인데 앵귤러, 리액트 모두 import 문법을 선호해서 발생한다.
import 문법을 노드는 지원하지 않기 때문이다. 따라서 6to5 라고 불렸던 Babel 을 통해 별도로 트랜스파일링을 진행해야 노드 기반에서 코드를 실행할 수 있다.

대부분의 자바스크립트 IDE 툴은 노드를 JS 실행도구로 사용하므로 불편하더라도 import 문법 사용 시 require 문법으로 바꾸는 작업을 선행해야 한다. 이에 따라 발생하는 불편함은 환경설정을 통해 자동으로 트랜스파일링 작업이 수행되게 IDE 툴을 셋팅하면 최소화 할 수 있다.

1. CommonJS 방식

노드기반 서버사이드에서 주로 사용하는 모듈시스템이다.

hello.js

```
var asia = function () {
    console.log('Hello Asia');
};

exports.korea = asia;

exports.world = function () {
    console.log('Hello World');
};
```

exports 키워드는 노드가 지원(require.js)하는 빌트인 객체다.

main.js

```
var hello = require('./hello');

hello.world();
hello.korea();
```

노드 내장 모듈과 npm install 로 설치해서 node_modules 폴더에 위치한 모듈은 ./ 기호를 사용하지 않고 임포트한다. "./" 기호는 현재 파일과 같은 위치에 있다는 의미이며 개발자가 추가로 만든 hello.js 파일을 임포트할 때 붙여야 한다. 확장자 .js 는 붙이지 않는다.

main.js 파일을 실행하여 정상적으로 hello.js 에서 제공하는(exports) 자원을 사용할 수 있는지 확인한다.

다른 예제를 한번 더 살펴보자.

hello2.js

```
module.exports = {
    message: 'hello javascript!',
    writer: 'Chris'
};
```

외부에 제공할 자원이 하나라면 module.exports 를 사용할 수 있다.

main2.js

```
var {message, writer} = require('./hello2');

console.log('message = '+message);
console.log('writer = '+writer);
```

"require('./hello2')" 코드는 객체를 가리키므로 객체의 프로퍼티명을 그대로 사용하여 해체할당 할 수 있다.

2. ES6(ECMA2015) 방식

JS 표준방식이라는 의미를 갖는다. 기본적으로 노드의 V8 엔진을 통해 JS 파일을 컴파일하기 때문에 웹스톰이나 아톰 같은 IDE 툴에서 노드가 지원하지 않는 ES6 문법을 사용할 수 없다. 트랜스파일링을 통해 생성된 JS 파일을 실행하는 방식으로 코드를 테스트 한다.

Syntax

다음은 사용 문법이다.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

```
export { name1, name2 };
export { name1 as alias1 };
export let name1, name2; // also var, const
export let name1 = ..., name2 = ...; // also var, const

export default expression;
export default function (...) { ... } // also class, function*
export default function name1(...) { ... } // also class, function*
export { name1 as default };

export * from 'anotherModule';
export { name1, name2, ..., nameN } from ...;
export { import1 as name1, import2 as name2, ..., nameN } from ...;
```

There are two different types of export, each type corresponds to one of the above syntax:

Named exports:

```
export { myFunction }; // exports a function declared earlier
export const foo = Math.sqrt(2); // exports a constant
```

Default exports (only one per script):

```
export default function() {} // or 'export default class {}'
// there is no semi-colon here
```

수동으로 Babel 트랜스파일링

hello.js

```
var asia = function () {
    console.log('Hello Asia');
};

export let korea = asia;

export let world = function () {
    console.log('Hello World');
};
```

main.js

```
import {world, korea} from './hello';

world();
korea();
```

Babel CLI 설치

```
npm install -g babel
```

Babel 트랜스파일링

```
babel * -d build --ignore build
```

* : 현재 폴더 위치에 모든 파일을 대상으로 한다.
-d build : 작업결과를 build 폴더 밑에 생성한다.
--ignore build : 반복적인 트랜스파일링 작업 시 build 폴더를 무시하기 위한 옵션을 사용한다.

build/hello.js

```
'use strict';

Object.defineProperty(exports, "__esModule", {
    value: true
});
var asia = function asia() {
    console.log('Hello Asia');
};
```

```
var korea = exports.korea = asia;

var world = exports.world = function world() {
    console.log('Hello World');
};
```

build/main.js

```
'use strict';

var _hello = require('./hello');

(0, _hello.world)();
(0, _hello.korea)();
```

위 JS 파일을 실행하는 것으로 코드의 정상작동을 확인한다.

트랜스파일링 작업에 익숙해지기 위해서 한번 더 연습해 보자.

my-module.js

```
export default function cube(x) {
    return x * x * x;
}
```

my-main.js

```
import cube from './my-module';

console.log(cube(3)); // 27
```

Babel 트랜스파일링

```
babel my-* -d build --ignore build
```

build/my-module.js

```
"use strict";

Object.defineProperty(exports, "__esModule", {
    value: true
});
exports.default = cube;
function cube(x) {
```

```
    return x * x * x;
}
```

build/my-main.js

```
'use strict';

var _myModule = require('./my-module');

var _myModule2 = _interopRequireDefault(_myModule);

function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : { default: obj }; }

console.log((0, _myModule2.default)(3)); // 27
```

위 JS 파일을 실행하는 것으로 코드의 정상작동을 확인한다.

자동으로 Babel 트랜스파일링 : ATOM 사용

1. Babel 의 필요성 확인

hello.js

```
var asia = function () {
  console.log('Hello Asia');
};

export let korea = asia;

export let world = function () {
  console.log('Hello World');
};
```

hello-importer.js

```
import {world, korea} from './hello';

world();
korea();
```

테스트 실행

위 파일에서 실행(alt+r)하면 에러가 발생한다.

```
D:₩sooup₩atom₩workspace₩hello-es6₩es-import₩hello.js:7
export let korea = asia;
^^^^^^
SyntaxError: Unexpected token export
... 생략
```

노드가 지원하지 않는 export/import 문법을 사용할 수 없음을 파악했다. 바벨을 사용해서 6to5 작업을 통해 실행할 수 있는 소스코드(노드가 지원하는 문법의 코드)를 얻도록 설정해 보자.

참고로 언급하자면 노드 버전 4.0 부터 ES6 문법인 class 키워드를 지원한다. 따라서 ATOM 에서 .js 확장자로 파일을 만들고 그 안에서 class 문법을 사용하고 바로 실행하는 것이 가능하다.

2. language-babel 패키지

2.1. 설치

File > Settings > Install >
type 'language-babel' in search box and enter > Install

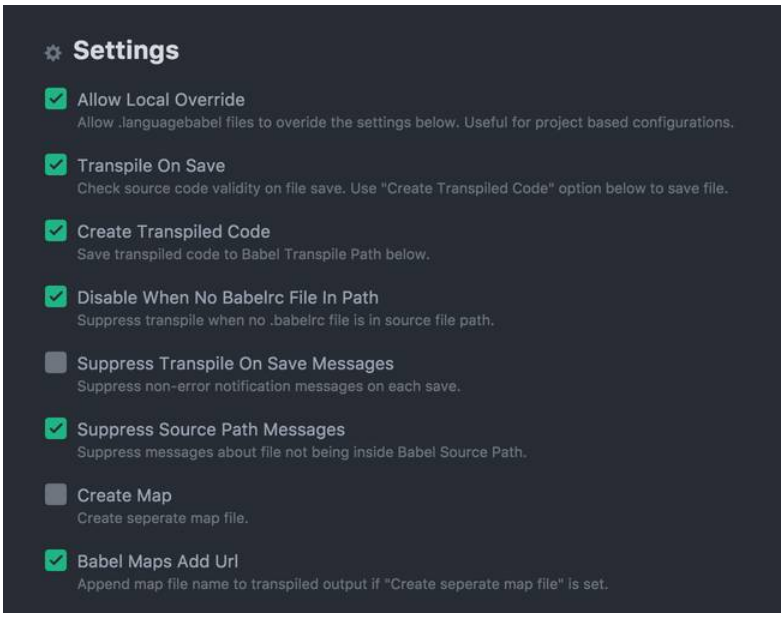
2.2. 패키지 기능 설명

https://atom.io/packages/language-babel

2.3. 패키지 설정

File > Settings > Packages >
type 'language-babel' in Installed Packages Search Box >
Community Packages > click 'Settings' button

다음화면을 참고하여 설정을 변경한다.



3 프로젝트 환경 설정

3.1. package.json 파일생성

npm init --yes

3.2. 바벨 디펜던시 설치

npm install --save-dev babel-core babel-preset-es2015

```
npm install --save-dev babel-preset-react babel-plugin-syntax-flow
```

3.3. 바벨 설정파일 작성

.babelrc

```
{
  "only": [
    "*.babel",
    "*.jsx",
    "*.es6"
  ],
  "presets": [
    "es2015",
    "react"
  ]
}
```

.languagebabel

```
{
"babelMapsPath": "",
"babelMapsAddUrl": false,
"babelSourcePath": "",
"babelTranspilePath": "",
"createMap": false,
"createTargetDirectories": false,
"createTranspiledCode": true,
"disableWhenNoBabelrcFileInPath": true,
"suppressSourcePathMessages": false,
"suppressTranspileOnSaveMessages": false,
"transpileOnSave": true
}
```

For most projects, it is better to configure language-babel via project-based **.languagebabel** file properties which will override the package settings.

Transpile On Save

On any file save of a language-babel grammar enabled file the Babel transpiler is called. No actual transpiled file is saved but any Babel errors and/or successful indicators are notified by an ATOM pop-up. Not all files are candidates for transpilation as other settings can affect this. For example see Disable When No Babelrc File In Path and Babel Source Path below.

{"transpileOnSave": true} or {"transpileOnSave": false}

.languagebabel 파일의 기타 설정은 사이트를 참고한다.
https://atom.io/packages/language-babel

3.4. 트랜스파일링 작업대상으로 설정하기

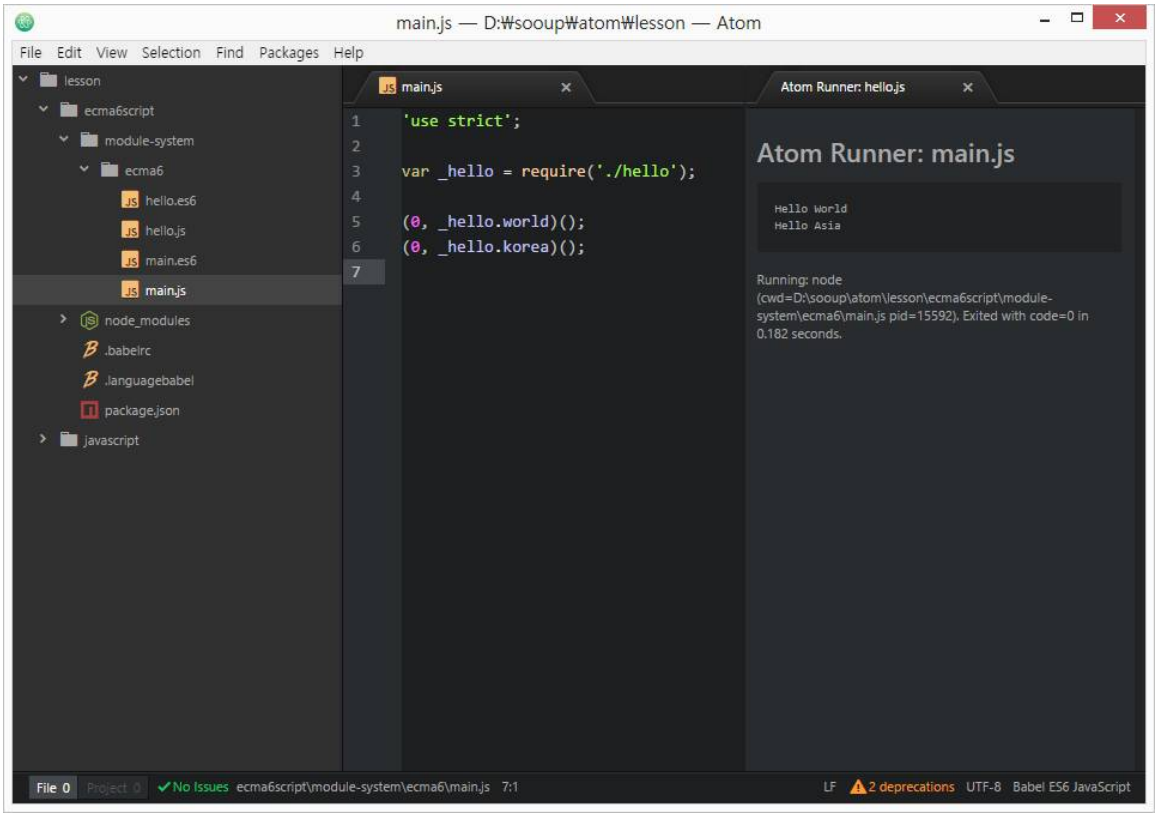
hello.js/ hello-importer.js 파일의 확장자를 .es6 로 모두 바꾼다.

파일을 저장하는 단축키(ctrl-s)를 누르면 바벨이 트랜스파일링을 진행하여 파일명은 같고 확장자가 .js 인 파일을 같은 폴더위치에 생성한다.

3.5. 테스트

확장자가 .js 인 파일을 실행(alt+r)한다.

4. 작업결과 화면



3. IIFE 방식

Immediately Invoked Function Expression 은 즉시 실행 함수 표현식이라고 부른다.
완벽히 분리된 스코프를 갖고 있으므로 복사해서 JS 파일 어느 위치에 끼어 넣어 사용하거나 HTML 을 통한 JS 파일들 결합시 스코프 충돌에 대한 걱정없이 사용할 수 있다.

module-iife.js

```
/*
 자바스크립트 모듈
 객체, 함수, 기타 컴포넌트의 묶음으로써 분리된 스코프를 갖는다.
 */

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

    var sub = function (x, y) {
        return x - y;
    };

    var calc = {
        add: function (a, b) {
            return add(a, b);
        },
        sub: function (a, b) {
            return sub(a, b);
        }
    };

    global.calc = calc;
})(global);

console.log(calc.add(1, 2)); // 3
console.log(calc.sub(1, 2)); // -1
```

4. AMD

Asynchronous Module Definition 의 약자다. AMD 는 브라우저에서 모듈을 구현하기 위해 만든 명세로 웹 페이지 로딩을 차단하지 않고도 비동기적으로 모듈을 임포트할 수 있다. 브라우저에 내장된 기능이 아니므로 AMD 라이브러리가 필요하고 require.js 는 가장 잘 알려진 AMD 라이브러리이며 모듈은 모두 별도 파일로 구현해야 한다.

동적 로딩

참고: http://d2.naver.com/helloworld/591319

<script> 태그는 페이지 렌더링을 방해한다. <script> 태그의 HTTP 요청과 다운로드, 파싱(Parsing), 실행이 일어나는 동안 브라우저는 다른 동작을 하지 않는다. 브라우저 입장에서는 당연하고 안전한 동작 방식이지만 사용자 입장에서는 빨리 화면이 보이고 버튼이 동작하기를 바랄 뿐이다. 그래서 최적화 기법 중의 하나로 <script> 태그를 가능한 한 <body> 태그의 마지막에 배치하는 방법이 있다.

하지만 사용자의 첫 인터랙션이 가능할 때까지 걸리는 시간이 줄어들지는 않는다. 페이지를 더 빨리 렌더링할 수는 있어도 첫 렌더링과 첫 인터랙션에 필요하지 않은, 페이지에 필요한 모든 JavaScript 를 로딩하기 때문이다. 화면이 복잡하고 AJAX 로 점철된 웹앱 수준의 규모에서는 이 시간이 큰 폭으로 커진다. 웹앱은 AJAX 로 전환되는 여러 뷰(View)를 가지고 있는 경우가 흔하다. 더 최적화를 하자면 첫 렌더링과 인터랙션에 필요한 JavaScript 만 먼저 로딩하고 후에 사용자의 반응에 따라 나머지를 로딩하는 점진적인 방식이 필요하다.

동적 로딩(Dynamic Loading, Lazy Loading 이라고도 부른다)은 페이지 렌더링을 방해하지 않으면서 필요한 파일만 로딩할 수 있다. 이를 구현하는 방법 중 하나로 <script> 태그의 동적 삽입이 있다. 이는 JavaScript 로 <script> 태그를 생성하여 추가하는 방법이다. 이 외에도 XMLHttpRequest, document.write(), defer 같은 방법이 있지만 범용적으로 사용하기에는 치명적인 단점이 하나씩은 있어서 <script> 태그의 동적 삽입이 제일 안전하고 합리적이다.

math.js

```
define("calc", [], function () {
  var sum = function (x, y) {
    return x + y;
  };

  var sub = function (x, y) {
    return x - y;
  };

  return {
```

```
    getSum : function (a, b) {
      return sum(a, b);
    },
    getSub : function (a, b) {
      return sub(a, b);
    }
  };
});
```

index.js

```
// 디펜던시 calc 가 먼저 로드된 뒤에 콜백이 수행된다.
requirejs(['calc'], function (calc) {
  console.log(calc.getSum(1, 2)); // 3
  console.log(calc.getSub(1, 2)); // -1
});
```

index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>AMD</title>
</head>
<body>
  <h1>AMD Running Test</h1>
  <h3>check out console</h3>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/require.js/2.3.2/require.min.js"></script>
  <script src="math.js"></script>
  <script src="index.js"></script>
</body>
</html>
```

10. 함수 축약 표현식

function-sugar-code.js

```
var obj = {
    render() {
        console.log('hello world');
    }
}

obj.render();
```

바벨 트랜스파일링

babel --presets latest function-sugar-code.js -d build

결과 확인

```
'use strict';

var obj = {
    render: function render() {
        console.log('hello world');
    }
};

obj.render();
```

"render()" 은 "render: function render()" 코드를 줄여 쓰는 문법이다.

11. Arrow function

애로우함수는 선언하는 문법만 다른 것이 아니라 많은 부분에서 일반적인 함수와 다르므로 명확한 이해가 필요하다.

■ 애로우 함수는 선언된 위치를 기준으로 가장 가까운 스코프의 소유주(Context)에 자동으로 바인딩 된다. 바인딩은 고정이며 바뀌지 않는다.

■ 애로운 함수는 이름을 가질 수 없으면 언제나 익명함수로 선언해서 사용한다.

■ 애로운 함수는 생성자로 사용될 수 없다. 따라서 new 키워드와 같이 사용할 수 없다.

■ 애로운 함수는 prototype 프로퍼티가 없다. 따라서 상속관계를 맺을 수 없다.

■ 애로운 함수는 arguments 객체를 갖고 있지 않다. 따라서 파라미터를 명시적으로 선언해야 전달되는 인수를 받아서 사용할 수 있다.

fat-arrow-function.js

```
var foo1 = () => {return 1;}          // without any parameters

var foo2 = a => {return a*2;}         // with single parameter

var foo3 = (a,b) => {return a+b;} // with muliplte parameter

console.log(foo1()); // 1

console.log(foo2(2)); // 4

console.log(foo3(3, 4)); // 7
```

Return in arrow function

애로우 함수는 바디부분에서 범위연산자 "{ }"를 명시적으로 사용하지 않으면 return 구문이 선언된 것처럼 행동한다. 사실 자바스크립트는 함수 끝에서 언제나 return 하는 행동방식을 보인다. 다만 일반 함수에서 return 구문을 생략하면 "return;" 와 같이 행동하지만 애로우 함수에서는 "return 처리결과" 와 같이 해동한다는 차이가 있을 뿐이다.

명시적으로 범위연산자 "{ }"를 사용하면 컴파일러는 개입하지 않는다.

fat-arrow-function2.js

```
// because there is no block statment, hence JS implicit `return`
var fn = a => a*2;
console.log(fn(1)); // 2

// because there is a block statment,and JS depends on explicit `return;`
var fn = a => {a*2};
console.log(fn(1)); // undefined

// because there is a block statment,and we are explicitly returning
var fn = a => {return a*2};
console.log(fn(1)); // 2

// because there is no curly braces,
// but parantheses, which is considered as expression,
// hence it implicitly `return` the object
var fn = a => ({id:a});
console.log(fn(1)); // {id:1}
```

How this is different for arrow functions

애로우 함수는 일반함수처럼 상황에 따라 별도의 컨텍스트 바인딩을 가질 수 없다. 애로우 함수에서 this 는 애로우 함수 위치를 기준으로 애로우 함수를 소유하고 있는 소유주를 찾아서 소유주의 this 를 자신의 this 로 삼는다.

다음은 일반 함수안에서 this 의 행동방식을 살펴보는 예제이다.

```
global.a = 'global';

function foo() {
    console.log(this.a);
}

foo.a = 'func';

foo(); // global

foo.call(foo); // func
```

다음 예제는 잘 못 이해하기 쉽다.

```
// The callback function is called from the setTimeout function,
// and the setTimeout function is defined in the global scope,
// which means, that the callback function is called from the global scope.
function foo() {
    setTimeout(function(){
        console.log(this.a); // undefined
    },100);
}

foo.a = 'func';

foo.call(foo);
```

setTimeout 비동기 함수가 사용하는 콜백함수는 함수코드만이 큐에 저장되었다가 0.1 초 후에 기동한다. 이 때에 콜백함수는 호출자가 없이 함수만을 실행하는 것이 되므로 this 가 가리키는 대상은 글로벌객체가 된다. 글로벌 객체는 a 라는 프로퍼티가 없으므로 결과적으로 "undefined"가 출력된다.

bind 메소드를 사용하여 함수가 참조할 this 를 미리 고정시킬 수 있다.

```
function foo() {
    setTimeout((function(){
        console.log(this.a); // func
    }).bind(this),100);
}

foo.a = 'func';
foo.call(foo);
```

일반함수를 bind 메소드로 호출자 this 를 고정시키는 방법은 앞서서 살펴보았다. 애로우 함수를 사용하면 보다 간단히 같은 결과를 얻을 수 있다. 왜냐하면 애로우 함수의 위치에서 가장 까운 스코프의 소유주는 foo 함수이기 때문에 애로우 함수에서의 this 는 foo 함수가 호출될 때 참조하는 호출자 this 로 자동적으로 바인딩되기 때문이다.

```
function foo() {
    setTimeout(()=>{
        console.log(this.a); // func
    },100);
}

foo.a = 'func';
foo.call(foo);
```

"foo.call(foo);" 라는 코드를 살펴보면 함수의 호출자는 foo 객체다. 따라서 애로운 함수의 this 는 자동적으로 호출자 foo 객체를 가리킨다.

이번에는 반대로 애로우 함수로부터 출발해서 설명하겠다. 애로우 함수의 this 를 처리하기 위해서 컴파일러는 애로우 함수 스코프를 살핀다. 애로우 함수는 단독으로 자신만의 바인딩을 별도로 가질 수 없으므로 찾을 수 없다. 다음으로 애로우 함수를 감싸고 있는 외부 스코프를 뒤진다. foo 함수는 일반함수로써 this 를 가지고 있다. 이제 this 를 찾았으므로 애로우 함수의 this 를 foo 함수의 this 를 가리키는 것으로 바인딩한다.

이해를 돕기 위해 위 예제를 살짝 변경한 아래 코드와 비교하면서 살펴보자.

```
function foo() {
    var that = this;
    setTimeout(()=>{
        console.log(that.a); // func
    },100);
```

```
}
foo.a = 'func'; foo.call(foo);
```

이번에는 애로우 함수를 멤버로 갖고 있는 객체에 대한 예제를 살펴보자.

arrow-as-object-member.js

```
global.a = 'global';

var obj = {
   a: 'object',
   foo: () => {
      console.log(this.a)
   },
   print: function(){
      console.log(this.a);
   }
};

obj.foo(); // undefined
obj.print(); // object

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

var foo = () => {
   console.log(this.a)
};

var print = function(){
   console.log(this.a);
}

foo(); // undefined
print(); // global
```

테스트 결과를 살펴보면 애로우 함수의 this 는 글로벌 스코프와는 연관되지 않는다는 것을 알 수 있다. 애로우 함수의 this 는 애로우 함수를 갖고 있는 큰 함수의 this 와만 바인딩된다는 것을 알 수 있다.

따라서 위 예제에서 애로운 함수가 제대로 작동하게 하려면 함수의 스코프가 필요하므로 다음과 같이 변경하면 제대로 작동하는 코드를 얻을 수 있다.

```
var obj = {
    a: 'object',
    foo: function () {
        return (() => {
            console.log(this.a)
        })();
    }
};

obj.foo(); // object
```

애로우 함수는 arguments 객체를 지원하지 않는다.

```
function foo(a, b, c) {
    return (x, y) => {
        console.log(arguments); // { '0': 1, '1': 2, '2': 3 }
    }
}

foo(1, 2, 3)(4, 5);
```

애로우 함수에서는 파라미터 자리에 변수를 명시해야만 함수 호출 시 전달하는 파라미터를 이용할 수 있다. 따라서 arguments 객체가 없는 애로우 함수에서는 나머지 연산자를 사용하는 것이 편리하다.

```
function foo(a, b, c, ...restOfFoos) {
    console.log(restOfFoos); // [0]

    return (x, y, ...restOfArrows) => {
        console.log(restOfArrows); // [6,7]
    }
}

foo(1, 2, 3, 0)(4, 5, 6, 7);
```

use case

애로운 함수는 콜백함수가 사용되는 모든 부분에서 맹활약을 펼칠 수 있다.

```
var smartPhones = [
    {name: 'iphone', price: 649},
    {name: 'Galaxy S6', price: 576},
    {name: 'Galaxy Note 5', price: 489}
];

// ES5
console.log(smartPhones.map(
    function (smartPhone) {
        return smartPhone.price;
    }
)); // [649, 576, 489]

// ES6
console.log(smartPhones.map(
    smartPhone => smartPhone.price
)); // [649, 576, 489]
```

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

console.log(numbers.filter(function (number) {
    return number % 3 === 0;
}));
// [ 3, 6, 9 ]

console.log(numbers.filter(number => number % 3 === 0));
// [ 3, 6, 9 ]
```

It Doesn’t Work The Way You Think It Does

it-does-not-work-the-way-u-think.js

```
this.test = "attached to the module";

var foo = {
    test: "attached to an object"
};

// method 프로퍼티는 메소드이름, 콜백함수를 받아서
// this 가 가리키는 객체에 추가한다.
foo.method = function (name, cb) {
    this[name] = cb;
};

foo.method("bar", () => {
    console.log(this.test);
});

// 여러분이 기대한 대로 출력되는가?
foo.bar(); // attached to the module
```

"attached to an object"가 출력되도록 코드를 바꾸어 보자.

```
this.test = 'attached to the module';

var foo = {
    test: 'attached to an object'
};

foo.method = function (name, cb) {
    this[name] = cb;
};

foo.method('bar', function () {
    console.log(this.test);
});

foo.bar(); // attached to an object
```

애로우 함수가 모든 곳에 적합한 것은 아니라는 사실을 알 수 있다.

12. $ expression

$-expression.js

```
var keyWord = 'react';
// use back-tick `
var url = `https://www.google.com/#newwindow=1&q=${keyWord}`;

console.log('url = '+url);
// url = https://www.google.com/#newwindow=1&q=react

// babel --presets latest $-expression.js -d build
```

```
'use strict';

var keyWord = 'react';

var url = 'https://www.google.com/#newwindow=1&q=' + keyWord;

console.log('url = ' + url);
```

13. class

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

클래스 주요 특징

1. class declarations are **not hoisted**.

2. brackets {}.
This is where you define class members, such as methods or constructors.
The bodies of class declarations and class expressions are **executed in strict mode**.

3. Constructor
The constructor method is a special method for creating and initializing an object created with a class.
A constructor can **use the super keyword** to call the constructor of a parent class.

클래스 기본 문법

1. constructor 함수안에서 this 키워드로 선언된 변수는 new 키워드로 만들어지는 객체의 프로퍼티가 된다.
2. 함수들은 생성자.prototype 객체에 메소드로 추가된다.
ES5 이전에는 생성자 함수를 선언한 후 생성자.prototype 객체에 메소드를 추가했었다.
추가로 getter, setter 접근자 메소드를 정의해서 사용할 수 있다.
3. 함수앞에 static 키워드를 붙이면 함수객체 자신의 프로퍼티로 추가된다.

class1.js

```
class Polygon {
    constructor(height, width) {
        let writer = 'Chris';
        this.height = height;
        this.width = width;
    }

    // 게터, 세터 메소드는 함수처럼 작성하지만 변수처럼 사용한다.
    get area() {
        return this.calcArea();
    }

    calcArea() {
        return this.height * this.width;
    }
```

```
    static help() {
        console.log('new Polygon(height, width)');
    }
}

const p = new Polygon(10, 10);

/*
 constructor 함수안에서 this로 정의한 자원은
 new 키워드로 새로 만들어지는 객체에 프로퍼티로 추가된다.
 */
console.log(p); // {"height":10,"width":10}
console.log(p.area); // 100
// 게터, 세터 메소드는 ()를 사용하지 않고 변수처럼 사용한다.
console.log(p.calcArea()); // 100

/*
 클래스안에 선언된 함수들은 클래스.prototype 객체에 메소드로 추가된다
 */
console.log(Object.getOwnPropertyNames(Polygon.prototype));
// [ 'constructor', 'area', 'calcArea' ]
console.log(typeof Polygon.prototype.area); // number
console.log(typeof Polygon.prototype.calcArea); // function

/*
 함수앞에 static 키워드를 붙이면 함수객체 자신에 프로퍼티로 추가된다
 */
console.log(Object.getOwnPropertyNames(Polygon));
// [ 'length', 'name', 'prototype', 'help' ]
Polygon.help();
// new Polygon(height, width)

/*
 babel --presets latest class-usecase.js -d build
 */
```

클래스 문법 코드를 ES5 코드로 트랜스파일링한 결과 확인

ES5 로 트랜스파일링을 수행하기 위해서 콘솔에서 다음 명령을 실행한다.

```
babel --presets latest class-usecase.js -d build
```

build/class1.js

```
/*
 클래스는 strict 모드에서 수행된다.
 */
'use strict';

var _typeof = typeof Symbol === "function" && typeof Symbol.iterator === "symbol" ?
    function (obj) {
        return typeof obj;
    } : function (obj) {
        return obj && typeof Symbol === "function" && obj.constructor === Symbol && obj !==
Symbol.prototype ?
            "symbol" : typeof obj;
    };

var _createClass = function () {
    function defineProperties(target, props) {
        for (var i = 0; i < props.length; i++) {
            var descriptor = props[i];
            descriptor.enumerable = descriptor.enumerable || false;
            descriptor.configurable = true;
            if ("value" in descriptor) descriptor.writable = true;
            Object.defineProperty(target, descriptor.key, descriptor);
        }
    }

    return function (Constructor, protoProps, staticProps) {
        if (protoProps) defineProperties(Constructor.prototype, protoProps);
        if (staticProps) defineProperties(Constructor, staticProps);
        return Constructor;
    };
}();

function _classCallCheck(instance, Constructor) {
    if (!(instance instanceof Constructor)) {
        throw new TypeError("Cannot call a class as a function");
    }
}

var Polygon = function () {
    function Polygon(height, width) {
```

```
        _classCallCheck(this, Polygon);

        /*
         constructor 함수선언 부분과의 비교
         ---------------------------------
         constructor(height, width) {
              let writer = 'Chris';
              this.height = height;
              this.width = width;
         }
         */
        var writer = 'Chris';
        this.height = height;
        this.width = width;
    }

    // 게터, 세터 메소드는 함수처럼 작성하지만 변수처럼 사용한다.

    /*
        _createClass 함수 호출 : class 문법 처리
        ---------------------------------
        파라미터 1: 클래스명
        파라미터 2: 함수 배열(게터/세터 포함)
        파라미터 3: 정적 메소드 배열
     */
    _createClass(Polygon, [{
        key: 'calcArea',
        value: function calcArea() {
            return this.height * this.width;
        }
    }, {
        key: 'area',
        get: function get() {
            return this.calcArea();
        }
    }], [{
        key: 'help',
        value: function help() {
            console.log('new Polygon(height, width)');
        }
    }]);

    return Polygon;
}();

var p = new Polygon(10, 10);

/*
```

```
 constructor 함수안에서 this로 정의한 자원은
 new 키워드로 새로 만들어지는 객체에 프로퍼티로 추가된다.
 */
console.log(p); // {"height":10,"width":10}
console.log(p.area); // 100
// 게터, 세터 메소드는 ()를 사용하지 않고 변수처럼 사용한다.
console.log(p.calcArea()); // 100

/*
 클래스안에 선언된 함수들은 클래스.prototype 객체에 메소드로 추가된다
 */
console.log(Object.getOwnPropertyNames(Polygon.prototype));
// [ 'constructor', 'area', 'calcArea' ]
console.log(_typeof(Polygon.prototype.area)); // number
console.log(_typeof(Polygon.prototype.calcArea)); // function

/*
 함수앞에 static 키워드를 붙이면 함수객체 자신에 프로퍼티로 추가된다
 */
console.log(Object.getOwnPropertyNames(Polygon));
// [ 'length', 'name', 'prototype', 'help' ]
Polygon.help();
// new Polygon(height, width)
```

extends 키워드

class-extends.js

```
class Animal {
	constructor(name) {
		this.name = name;
	}

	speak() {
		console.log(this.name + ' makes a noise.');
	}
}

class Dog extends Animal {
	speak() {
		console.log(this.name + ' barks.');
	}
}

var d = new Dog('Jindo');
d.speak();
// Jindo barks.

console.log(d);
// {"name":"Jindo"}

console.log(d.__proto__ === Dog.prototype);
// true
console.log(Object.getOwnPropertyNames(Dog.prototype));
// [ 'constructor', 'speak' ]

console.log(Dog.prototype.__proto__ === Animal.prototype);
// true
console.log(Object.getOwnPropertyNames(Animal.prototype));
// [ 'constructor', 'speak' ]

/*
  babel --presets latest class-extends.js -d build
  */
```

ES5, ES6 문법 혼용

함수는 다른 함수 또는 객체를 상속받을 수 있다. class 문법과 기존의 함수 상속 문법을 섞어서 사용할 수 있다.

class-extends-es5.js

```
function Animal(name) {
    this.name = name;
}

Animal.prototype.speak = function () {
    console.log(this.name + ' makes a noise.');
};

var animal = new Animal('Animal');
animal.speak(); // Animal makes a noise.
animal.__proto__.speak.call(animal); // Animal makes a noise.

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

class Dog extends Animal {
    speak() {
        console.log(this.name + ' barks.');
    }
}

var dog = new Dog('Dog');
dog.speak(); // Dog barks.
dog.__proto__.speak.call(dog); // Dog barks.
dog.__proto__.__proto__.speak.call(dog); // Dog makes a noise.

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

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

Pet.prototype.speak = function () {
    console.log(this.name + ' barks bowwow.');
};

Pet.prototype.__proto__ = Dog.prototype;
```

```
var pet = new Pet('Pet');

pet.speak(); // Pet barks bowwow.
pet.__proto__.speak.call(pet); // Pet barks bowwow..
pet.__proto__.__proto__.speak.call(pet); // Pet barks.
pet.__proto__.__proto__.__proto__.speak.call(pet); // Pet makes a noise.
```

클래스가 객체를 대상으로 상속

class-inherit-regular-object.js

```
var log = console.log;

var Animal = {
    speak() {
        console.log(this.name + ' makes a noise.');
    }
};

class Dog {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(this.name + ' barks.');
    }
}

/*
 클래스는 비 컨스트럭터블(non-constructible) 객체를 상속할 수 없다.
 객체를 상속하기 위해서는 Object.setPrototypeOf() 메소드를 사용해야 한다.

 Object.setPrototypeOf(자식 객체, 부모 객체)
 */
Object.setPrototypeOf(Dog.prototype, Animal);
log(Dog.prototype.__proto__ === Animal); // true

var d = new Dog('Jindo');
d.speak();

log(Object.getOwnPropertyNames(d));
// [ 'name' ]
log(Object.getOwnPropertyNames(d.__proto__));
// [ 'constructor', 'speak' ]
log(Object.getOwnPropertyNames(d.__proto__.__proto__));
// [ 'speak' ]
```

super 키워드

class-super.js

```
var log = console.log;

class Cat {
	constructor(name) {
		this.name = name;
	}

	speak() {
		console.log(this.name + ' makes a noise.');
	}
}

/*
 super 키워드로 부모가 갖고 있는 함수를 호출할 수 있다.
 */
class Lion extends Cat {
	speak() {
		super.speak();
		console.log(this.name + ' roars.');
	}
}

var lion = new Lion('leo');

log(lion.name);
// leo
lion.speak();
// leo makes a noise.
// leo roars.
```

class-super2.js

```
class Cat {
    constructor(state = {}, props = {}) {
        console.log('Cat > constructor() called.');
        this.state = state;
        this.props = props;
    }
}

class Lion extends Cat {
    /*
     클래스 내 constructor 함수를 명시하지 않으면
     super()를 호출하는 constructor 함수가 있는 것과 같다.
     */
    constructor(state, props, data = []) {
        /*
         super()의 기능은 Cat.call(this)와 같다.
         */
        super();
        /*
         문법적으로 super() 호출 없이 this 를 사용할 수 없다.
         에러메시지 : 'this' is not allowed before super()
         */
        this.data = data;
        console.log('Lion > constructor() called.');
    }
}

var lion = new Lion({a: 1}, {b: 2}, [1, 2, 3]);
// Cat > constructor() called.
// Lion > constructor() called.

console.log(lion);
// { state: {}, props: {}, data: [ 1, 2, 3 ] }
```

클래스 표현식 vs 선언식

class-super-super.js

```
let log = console.log;

/*
 1. 클래스 표현식
 class 키워드 다음 클래스명을 지정하지 않는다.
 */
let Korean = class {
    constructor(a) {
        // new 키워드로 새로 만들어지는 객체에 추가된다.
        this.a = a;
    }

    // 메소드는 prototype 객체에 추가된다.
    printA() {
        console.log('a = ' + this.a);
    }

    // 정적메소드는 함수객체에 추가된다.
    static say(caller) {
        console.log('Hello ' + caller.a);
    }
};

/*
 2. 클래스 선언식
 */
class HongFamily extends Korean {
    constructor(a, b) {
        super(a);
        this.b = b;
    }

    printB() {
        console.log('b = ' + this.b);
    }

    fetchB() {
        return this.b;
    }
```

```
    // getter, setter 메소드는 일반적으로 메소드명을 중복해서 선언한다.
    get bValue() {
        return this.b;
    }

    set bValue(b) {
        this.b = b;
    }
}

class Gildong extends HongFamily {
    constructor(a, b, c) {
        super(a, b);
        this.c = c;
    }

    printC() {
        console.log('c = ' + this.c);
    }
}

let gildong = new Gildong(1, 2, 3);

log(gildong);
// { a: 1, b: 2, c: 3 }

log(Object.getOwnPropertyNames(Gildong.prototype));
// [ 'constructor', 'printC' ]

gildong.printC();
// c = 3

log(Object.getOwnPropertyNames(HongFamily.prototype));
// [ 'constructor', 'printB', 'fetchB', 'bValue' ]

gildong.printB();
// b = 2

log(Object.getOwnPropertyNames(Korean.prototype));
// [ 'constructor', 'printA' ]

gildong.printA();
```

```
// a = 1

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

Korean.say(gildong); // Hello 1

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

log(gildong.fetchB()); // 2

log(gildong.b); // 2

log(gildong.bValue); // 2
```

ES5 상속과 class 를 사용한 ES6 상속의 차이점

https://www.sitepoint.com/object-oriented-javascript-deep-dive-es6-classes/

JavaScript's class syntax is often said to be **syntactic sugar**, and in a lot of ways it is,
but there are also real **differences** things **we can do with ES6 classes** that **we couldn't with ES5**.

Static Properties Are Inherited

```
// ES5
function B() {}
B.f = function () {};

function D() {}
D.prototype = Object.create(B.prototype);

D.f(); // error
```

```
// ES6
class B {
    static f() {}
}

class D extends B {}

D.f(); // ok
```

Built-in Constructors Can Be Subclassed

```
// ES5
function D() {
      Array.apply(this, arguments);
}

D.prototype = Object.create(Array.prototype);

var d = new D();
d[0] = 42;

console.log(d instanceof Array); // true
console.log(d.length); // 0 - bad
```

```
// ES6
class D extends Array {}

let d = new D();
d[0] = 1;

console.log(d.length); // 1 - good
```

Multiple Inheritance with Proxies

```
let transmitter = {
    transmit() {
        console.log('transmit() called');
    }
};

let receiver = {
    receive() {
        console.log('receive() called');
    }
};

// Create a proxy object that intercepts property accesses
// and forwards to each parent, returning the first defined value it finds
let inheritsFromMultiple = new Proxy([transmitter, receiver], {
    get: function (proxyTarget, propertyKey) {
        const foundParent = proxyTarget.find(parent => parent[propertyKey] !== undefined);
        return foundParent && foundParent[propertyKey];
    }
});

inheritsFromMultiple.transmit(); // transmit() called
inheritsFromMultiple.receive(); // receive() called

console.log(Array.isArray(inheritsFromMultiple)); // true
console.log(inheritsFromMultiple instanceof Array); // true
```

Symbol.species

class-symbol.species.js

```
/*
 static 접근자 [Symbol.species]() 함수는
 부모 생성자 함수가 새 인스턴스를 반환하면
 어떤 생성자를 써야 할 지 알려줘야 할 때
 자식 생성자에 선택적으로 추가한다.

 @@species 함수를 명시적으로 정의하지 않으면
 부모 생성자 함수는 기본 생성자를 사용한다.
 */

/*
 기본 생성자 대신 다른 생성자를 사용해야 할 필요가 있는 예
 */

class MyArray1 extends Array {

}

var arr1 = new MyArray1(1, 2, 3);
console.log(arr1 instanceof MyArray1); // true
console.log(arr1 instanceof Array); // true

arr1 = arr1.map(function (item) {
    return item * 10;
});

/*
 map 메소드는 새 배열을 만들고 루프 순회 시 리턴한 값들을
 모아서 새 배열에 담아서 최종적으로 새로 만든 배열을 리턴한다.

 그런데 결과 배열이 MyArray1 의 클래스의 자식일 이유가 없는데
 arr1 instanceof MyArray1 조회 결과를 보면 상속관계가 맺어져 있다.
 */
console.log(arr1 instanceof MyArray1); // true
console.log(arr1 instanceof Array); // true

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

class MyArray2 extends Array {
```

```
    /*
     newArr.map 메소드가 새 배열을 만들 때 newArr 객체를 만들 때 사용한
     MyArray2 함수의 기본 생성자를 사용하는 대신
     static get [Symbol.species]() 함수가 리턴하는 함수의 생성자를 사용한다.
     */
    static get [Symbol.species]() {
        return Array;
    }
}

var arr2 = new MyArray2(1, 2, 3);
console.log(arr2 instanceof MyArray2); // true
console.log(arr2 instanceof Array); // true

arr2 = arr2.map(function (item) {
    return item * 10;
});

/*
 map 메소드의 처리결과로 리턴되는 배열은 Array 하고만 상속관계가 되었다.
 */
console.log(arr2 instanceof MyArray2); // false
console.log(arr2 instanceof Array); // true
```

class-symbol.species2.js

```
class MyArray1 {
	constructor(){
		console.log('MyArray1 > constructor() called');
	}

	mapping() {
		return new this.constructor[Symbol.species]();
	}
}

class MyArray2 extends MyArray1{
	constructor(){
		super();
		console.log('MyArray2 > constructor() called');
	}

	static get [Symbol.species]() {
		console.log('MyArray2 > static get [Symbol.species]() called');
		return MyArray1;
	}
}

var arr = new MyArray2();
// MyArray1 > constructor() called
// MyArray2 > constructor() called

console.log(arr instanceof MyArray2);
// true

var newArr = arr.mapping();
// MyArray2 > static get [Symbol.species]() called
// MyArray1 > constructor() called

console.log(newArr instanceof MyArray2);
// false
console.log(newArr instanceof MyArray1);
// true

console.log(arr === newArr);
// false
```

new.target

class-new.target.js

```
/*
 new.target
 ES6 에서 모든 함수에 추가된 히든 파라미터.
 새 객체 생성 시 사용된 생성자 함수가 무엇인지 알수 있다.

 1. 생성자를 new 키워드로 호출하면 new.target 은 생성자를 가리킨다.
 2. 생성자를 super 키워드로 호출하면 new.target 은 super 가 사용된 생성자의 new.target 을 가리킨다.
 3. 화살표 함수에서는 자신을 둘러싸고 있는 비 화살표 함수의 new.target 을 가리킨다.
 */

function Parent() {
    console.log('Parent > ' + new.target.name);
}

class Child extends Parent {
    constructor() {
        super();
        console.log('Child > ' + new.target.name);
    }
}

var child = new Child();
// Parent > Child
// Child > Child

var parent = new Parent();
// Parent > Parent
```

리액트에서의 클래스 사용 예

react-class-usecase.js

```
import React from 'react';

var TextAreaCounter = React.createClass({
    // #1
    propTypes: {
        text: React.PropTypes.string,
    },
    // #2
    getDefaultProps: function () {
        return {
            text: '',
        };
    },
    // #3
    getInitialState: function () {
        return {
            text: this.props.text,
        };
    },
    // #4
    _textChange: function (ev) {
        this.setState({
            text: ev.target.value,
        });
    },
    // #5
    render: function () {
        return React.DOM.div(null,
            React.DOM.textarea({
                value: this.state.text,
                onChange: this._textChange,
            }),
            React.DOM.h3(null, this.state.text.length)
        );
    }
});


// ES6 의 클래스로 바꿔서 코딩하면 다음과 같다.
```

```
class TextAreaCounter2 extends Component {
    constructor(props) {
        super(props);

        // #3
        this.state = {
            text: this.props.text,
        };

        // #4-1
        // 추가로 직접 바인딩을 설정해야 한다.
        // 또는 arrow 연산자로 선언해서 bind 처리할 수도 있다.
        this._textChange = this._textChange.bind(this);
    }

    _textChange(ev) {
        this.setState({
            text: ev.target.value,
        });
    }

    // #4-2
    // 화살표 연산자를 사용하면 자동으로 bind 처리가 된다.
    _textChange2 = (ev) => {
        this.setState({
            text: ev.target.value,
        });
    }

    // #5
    // function 키워드는 필요없다.
    render() {
        return React.DOM.div(null,
            React.DOM.textarea({
                value: this.state.text,
                onChange: this._textChange,
            }),
            React.DOM.h3(null, this.state.text.length)
        );
    }

    // ES7 문법인 static 속성을 사용하여 처리할 수도 있다.
    // static propTypes = {
```

```
    //      text: React.PropTypes.string,
    // }
}

// #1
TextAreaCounter2.propTypes = {
    text: React.PropTypes.string,
}

// #2
TextAreaCounter2.defaultProps = {
    text: '',
};
```

14. Mix-ins

Subclassing in JavaScript is used for two reasons:

■ Interface inheritance:

Every object that is an instance of a subclass (as tested by instanceof) is also an instance of the superclass. The expectation is that subclass instances behave like superclass instances, but may do more.

■ Implementation inheritance:

Superclasses pass on functionality to their subclasses.
The usefulness of classes for implementation inheritance is limited,
because they **only support single inheritance (a class can have at most one superclass).**
Therefore, it is impossible to inherit tool methods from multiple sources – they must all come from the superclass.

So how can we solve this problem? Let's explore a solution via an example.
Consider a management system for an enterprise where **Employee is a subclass of Person**.

```
class Person { ··· }
class Employee extends Person { ··· }
```

Additionally, **there are tool classes for storage and for data validation**:

```
class Storage {
    save(database) { ··· }
}
class Validation {
    validate(schema) { ··· }
}
```

It would be nice if we could include the tool classes like this:

```
// Invented ES6 syntax:
class Employee extends Storage, Validation, Person { ··· }
```

That is, we want Employee to be a **subclass of Storage**
which should be a **subclass of Validation**
which should be a **subclass of Person**.

Employee and Person will only be used in one such chain of classes.
But Storage and Validation will be used multiple times.

Such templates(Validation, Person) are called mixins.

One way of implementing a mixin in ES6 is to view it as a function
whose **input is a superclass and whose output is a subclass** extending that superclass:

```
const Storage = Sup => class extends Sup {
      save(database) { ··· }
};

const Validation = Sup => class extends Sup {
      validate(schema) { ··· }
};
```

Here, we profit from the operand of the extends clause
not being a fixed identifier, but an arbitrary expression.

With these mixins, Employee is created like this:

```
class Employee extends Storage(Validation(Person)) { ··· }
```

Abstract subclasses or mix-ins are templates for classes.
An ECMAScript class can **only have a single superclass**,
so multiple inheritance from tooling classes, for example, is not possible.
The functionality must be provided by the superclass.

A function with a superclass as input and a subclass extending that superclass as output
can be used to implement mix-ins in ECMAScript:

class-mixin.js

```
class Person {
	constructor() {
		this.name = 'person';
	}

	printName() {
		console.log(this.name);
	}
}

var randomizerMixin = Base => class extends Base {
	randomize() {
		console.log('randomize() called.');
		return this;
	}
};

var calculatorMixin = Base => class extends Base {
	calc() {
		console.log('calc() called.');
		return this;
	}
};

console.log(typeof randomizerMixin); // function
console.log(typeof calculatorMixin); // function

/*
에로우 함수는 가장 가까운 컨텐스트의 스코프에 자동으로 바인딩이 된다.
 */
class Employee extends calculatorMixin(randomizerMixin(Person)) {
	constructor() {
		super();
		this.age = '20';
	}
}

var emp = new Employee();

console.log(emp.__proto__ === Employee.prototype); // true
console.log(emp instanceof Employee); // true
console.log(emp instanceof Person); // true
```

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

/*
 Employee 생성자로부터 age 를 받고 Person 생성자로부터 name 을 받아서 emp 객체를 만들었다.
 */
console.log(Object.getOwnPropertyNames(emp)); // [ 'name', 'age' ]

/*
emp 객체는 프로토타입 체이닝을 따라
Employee.prototype 객체로부터
부모 객체에 존재하는 calc 메소드와
한 단계 위 부모 객체에 존재하는 randomize 메소드를 이용할 수 있으면
맨 위 부모 객체인 Person.prototype 객체가 갖고 있는 printName 메소드를 이용할 수 있다.
 */
emp.calc(); // calc() called.
emp.randomize(); // randomize() called.
emp.printName(); // person

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

/*
함수의 prototype 객체와 연결되는 것이 아니다.
 */
console.log(Employee.prototype.__proto__ === calculatorMixin.prototype); // false
console.log(Employee.prototype.__proto__ === randomizerMixin.prototype); // false
console.log(Employee.prototype.__proto__ === Person.prototype); // false

/*
emp (new 연산자로 만든 객체)
--> Employee.prototype (emp 객체의 부모 객체)
--> Employee.prototype.__proto__ (calculatorMixin 이 가리키는 함수를 사용하여 만들어진 객체)
--> Employee.prototype.__proto__.__proto__ (randomizerMixin 이 가리키는 함수를 사용하여 만들어진 객체)
--> Employee.prototype.__proto__.__proto__.__proto__ (Person.prototype 과 같다)
 */
console.log(Employee.prototype.__proto__.__proto__.__proto__ === Person.prototype); // true

console.log('--------------------');
/*
calculatorMixin, randomizerMixin 가 가리키는 함수는 애로우 함수로써
prototype 객체를 갖고 있지 않다.
 */
```

```
console.log(calculatorMixin.prototype); // undefined
console.log(randomizerMixin.prototype); // undefined

/*
calculatorMixin, randomizerMixin 를 사용하여 만든 익명 객체 2 개는 Person 생성자로 만든 것이다.
 */
console.log(Employee.prototype.__proto__); // Person {}
console.log(Employee.prototype.__proto__.__proto__); // Person {}

/*
기능 함수 calc, randomize, printName 의 소유주를 확인하자.
 */
console.log(Object.getOwnPropertyNames(Employee.prototype.__proto__));
// [ 'constructor', 'calc' ]
console.log(Object.getOwnPropertyNames(Employee.prototype.__proto__.__proto__));
// [ 'constructor', 'randomize' ]
console.log(Object.getOwnPropertyNames(Employee.prototype.__proto__.__proto__.__proto__));
// [ 'constructor', 'printName' ]


/*
 babel --presets latest class-mixin.js -d build
 */
```

15. generator

https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Statements/function*

제너레이터 함수의 처리순서

example1.js

```
/*
제너레이터 함수의 처리 순서
 */

// * 기호를 붙이면 제너레이터 함수가 된다.
// the first call of next executes from the start of the function
// until the first yield statement
function* idMaker() {
    var index = 0;
    while (index < 3) {
        // 3 번 양보한다.
        yield index++;
    }
    // 제너레이터 함수가 종료되면 자동으로
    // { value: undefined, done: true } 객체를 리턴한다.
}

// 이터레이터를 받는다.
var gen = idMaker();

// next 함수로 idMaker 함수를 yield 한 곳에서부터 다시 실행한다.
console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next()); // { value: undefined, done: true }

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

/*
제너레이터 함수의 중복 호출
 */
function* anotherGenerator(i) {
    yield i + 1;
```

```
    yield i + 2;
    yield i + 3;
}

function* generator(i) {
    yield i;
    // 제너레이터 함수 로직에서 다른 제너레이터 함수를 호출할 수 있다.
    yield* anotherGenerator(i);
    yield i + 10;
}

var gen = generator(10);

console.log(gen.next().value); // 10
console.log(gen.next().value); // 11
console.log(gen.next().value); // 12
console.log(gen.next().value); // 13
console.log(gen.next().value); // 20

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

/*
제너레이터에게 값 전달
 */
function* logGenerator() {
    console.log(yield); // pretzel
    console.log(yield); // california
    console.log(yield); // mayonnaise
}

var gen = logGenerator();

gen.next();
gen.next('pretzel');
gen.next('california');
gen.next('mayonnaise');
```

제너레이터가 리턴하는 객체의 구조

example2.js

```
// http://hacks.mozilla.or.kr/2015/08/es6-in-depth-generators/

/*
제너레이터가 리턴하는 객체의 구조
 */
function* quips(name) {
    yield "hello " + name + "!";
    yield "i hope you are enjoying the blog posts";
    if (name.startsWith("X")) {
        yield "it's cool how your name starts with X, " + name;
    }
    yield "see you later!";
}

var iter = quips("korea");
console.log(iter.next()); // { value: 'hello korea!', done: false }
console.log(iter.next()); // { value: 'i hope you are enjoying the blog posts', done: false }
console.log(iter.next()); // { value: 'see you later!', done: false }
console.log(iter.next()); // { value: undefined, done: true }
```

커스텀 이터레이터

example3.js

```
/*
    커스텀 이터레이터
 */

// [Symbol.iterator] 메소드와 .next 메소드를 구현하는 것만으로
// 커스텀 이터레이터를 만들 수 있다.
class RangeIterator {
    constructor(start, stop) {
        this.value = start;
        this.stop = stop;
    }

    [Symbol.iterator]() {
        return this;
    }

    next() {
        var value = this.value;
        if (value < this.stop) {
            this.value++;
            return {done: false, value: value};
        } else {
            return {done: true, value: undefined};
        }
    }
}

// 이 함수는 start 에서 stop 전 까지 더해나가는 새로운 이터레이터를 리턴한다.
function range(start, stop) {
    return new RangeIterator(start, stop);
}

/*
    for ~ of 이터레이터 구문
 */
for (var value of range(0, 3)) {
    console.log("Ding! at floor #" + value);
}
```

```
// Ding! at floor #0
// Ding! at floor #1
// Ding! at floor #2

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

// 4 줄짜리 제너레이터가 이전에 RangeIterator class 를 이용해 만든
// range 함수 구현로직과 같다.

function* range2(start, stop) {
      for (var i = start; i < stop; i++) {
            yield i;
      }
}

for (var value of range2(0, 3)) {
      console.log("Ding! at floor #" + value);
}

// Ding! at floor #0
// Ding! at floor #1
// Ding! at floor #2
```

제너레이터 유스케이스

example4.js

```
/*
 제너레이터 유스케이스
 */

// 파라미터로 받은 배열을 size 만큼 잘라서 새 배열에 담는다.
// 작업결과를 모은 하나의 배열을 최종적으로 리턴한다.
function split(array, size) {
	var newArray = [];
	for (var i = 0; i < array.length; i += size) {
		newArray.push(array.slice(i, i + size));
	}
	return newArray;
}

// 루프가 돌 때마다(yield 할 때마다) 결과를 리턴한다.
function* split2(array, size) {
	for (var i = 0; i < array.length; i += size) {
		yield array.slice(i, i + size);
	}
}

let array = [1, 2, 3, 4, 5, 6, 7, 8];

/*
 split 함수의 로직(루프)이 다 처리된 후에야 결과를 받는다.
 */
console.log(split(array, 3)); // [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8 ] ]

/*
 split2 함수에서 yield 할 때마다 결과를 받는다.
 받은 결과를 사용하고 다음 결과를 받기 위해서 next 메소드를 호출한다.
 */
let iter = split2(array, 3);
console.log(iter.next()); // { value: [ 1, 2, 3 ], done: false }
console.log(iter.next()); // { value: [ 4, 5, 6 ], done: false }
console.log(iter.next()); // { value: [ 7, 8 ], done: false }
console.log(iter.next()); // { value: undefined, done: true }

/*
```

```
 for-of 구문의 사용이유
 for-of 문을 사용하면 필요한 횟수 만큼 자동적으로 처리되어서 편리하다.
 */
for (var value of split2(array, 3)) {
    console.log(value);
}
// [ 1, 2, 3 ]
// [ 4, 5, 6 ]
// [ 7, 8 ]
```

배열 이터레이터

example5.js

```
var arr = [3, 5, 7];
// 배열도 객체이므로 프로퍼티를 추가할 수 있다.
arr.foo = "hello";

console.log(arr); // [ 3, 5, 7, foo: 'hello' ]
console.log(arr.length); // 3, 배열의 길이(기본 작동 방식)

/*
 배열로써 갖고 있는 값만을 확인한다.
 */
console.log(JSON.stringify(arr)); // [3,5,7]
/*
 배열을 객체관점에서 프로퍼티를 확인한다.
 */
console.log(Object.getOwnPropertyNames(arr)); // [ '0', '1', '2', 'length', 'foo' ]

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

/*
 for-in 구문
 객체에서 열거할 수 있는 프로퍼티를 대상으로 루프 처리를 한다.
 An enumerable property can be included in and visited during for..in loops
 */
for (var i in arr) {
      console.log(i);
}
// 0
// 1
// 2
// foo

/*
 배열에서 length 프로퍼티는 enumerable: false 로써 열거 대상이 아니다.
 */
console.log(Object.getOwnPropertyDescriptor(arr, 'length'));
// {     value: 3,
//       writable: true,
//       enumerable: false,
//       configurable: false }
```

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

/*
 for-of 구문
 이터레이터를 사용하여 제너레이터 함수를 호출하고 결과를 받아서 루프 처리를 한다.
 배열은 이터러블 규약(@@iterator 메소드)과 이터레이터(next 메소드) 규약이 모두 구현되어 있다.
 */
for (var i of arr) {
      console.log(i);
}
// 3
// 5
// 7

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

/*
 forEach 메소드
 forEach is an method that is available only in Array objects.
 It allows you to iterate through elements of an array.
 The callback can access both index and value of the array elements.
 */
arr.forEach((value, index) => console.log(index + ':' + value));
// 0:3
// 1:5
// 2:7
```

example6.js

```
function count(n) {
	var res = [];
	for (var x = 0; x < n; x++) {
		res.push(x)
	}
	return res;
}

for (var x of count(3)) {
	console.log(x)
}
// 0
// 1
// 2

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

console.log(count(3)); // [ 0, 1, 2 ]

for (var x of [0, 1, 2]) {
	console.log(x)
}
// 0
// 1
// 2

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

function* demo() {
	var res = yield 10;
	console.log(res === 32 ? 'equal' : 'not equal'); // equal
	return 42;
}

var d = demo();
console.log(d.next()); // {value: 10, done: false}
console.log(d.next(32)); // {value: 42, done: true}
```

제너레이터 함수에게 예외를 전파

example7.js

```
var sentinel = new Error('foo');

function* demo() {
    try {
        yield 10;
    } catch (ex) {
        if (ex === sentinel) {
            console.log(ex.message); // foo
        } else {
            console.log('error');
        }
    }
}

var d = demo();

console.log(d.next()); // {value: 10, done: false}

d.throw(sentinel);
```

비동기함수와의 연동방식을 동기방식으로 코딩

example8.js

```
//--------------------------------------------------------------------------
// Generators enable us to write asynchronous code in a synchronous manner.
// 제너레이터를 사용하면 비동기 연산을 동기방식으로 작성할 수 있다.
//--------------------------------------------------------------------------

function getFirstName() {
	console.log('f-1');
	setTimeout(function () {
		console.log('f-3');
		gen.next('alex');
	}, 2000);
	console.log('f-2');
}

function getSecondName() {
	console.log('s-1');
	setTimeout(function () {
		console.log('s-3');
		gen.next('perry');
	}, 500);
	console.log('s-2');
}

function *sayHello() {
	console.log('--1');
	var a = yield getFirstName();
	console.log('--2');
	var b = yield getSecondName();
	console.log('--3');
	console.log(a, b); //alex perry
}

var gen = sayHello();

gen.next();
```

다음 예를 살펴보면 앞에 사용방식이 얼마나 편리한지 알게 될 것이다.

generator8-1.js

```
function getFirstName() {
	console.log('f-1');
	setTimeout(function () {
		console.log('f-3');
		return 'alex';
	}, 2000);
	console.log('f-2');
}

function getSecondName() {
	console.log('s-1');
	setTimeout(function () {
		console.log('s-3');
		return 'perry';
	}, 500);
	console.log('s-2');
}

function sayHello() {
	console.log('--1');
	var a = getFirstName();
	console.log('--2');
	var b = getSecondName();
	console.log('--3');
	console.log(a, b);
}

sayHello();
```

16. Async / Await

You may have heard of the as-yet not officially standard async/await proposal for JavaScript.
It did not make it into ES6. It will not make it into ES2016.

It could become standard **in ES2017**,
and then we'll need to wait for all the JS engine implementations to land before we can use it.

Note: it works in Babel now, but that's no guarantee.
Tail call optimization worked in Babel for several months but got subsequently removed.

async-await1.js

```
const fetchSomething = () => new Promise((resolve) => {
    setTimeout(() => resolve('future value'), 500);
});

const promiseFunc = () => new Promise((resolve) => {
    fetchSomething().then(result => {
        resolve(result + ' 2');
    });
});

promiseFunc().then(result => console.log(result));
// future value 2
```

async-await2.ts

```
const fetchSomething = () => new Promise((resolve) => {
    setTimeout(() => resolve('future value'), 500);
});

async function asyncFunction() {
    const result = await fetchSomething(); // returns promise

    // waits for promise and uses promise result
    return result + ' 2';
}

asyncFunction().then(result => console.log(result));
```

```
/*
    트랜스파일링을 한 후 .js 파일을 실행한다.
 */
```

The async/await version looks like regular, synchronous code, but it's not.
It yields the promise and exits the function, **freeing the JS engine to do other things**,
and when the promise from `fetchSomething()` resolves, the function **resumes**,
and the resolved promise value is assigned to `result`.

It's asynchronous code that looks and feels synchronous.
For JavaScript programmers who do a ton of asynchronous programming every day,
this is basically the holy grail.

All of the performance benefits of asynchronous code with none of the cognitive overhead.

```
function* foo() {
    yield 'a';
    yield 'b';
    yield 'c';
}

for (const val of foo()) {
    console.log(val);
}
// a
// b
// c

const [...v] = foo();
console.log(v); // ['a','b','c']
console.log(Array.isArray(v)); // true
```

17. 비동기 처리

일반적으로 동기식으로 수행되는 언어는 멀티 스레드를 지원한다. 하지만 자바스크립트는 단일 스레드 언어다. 따라서 일부 로직이 무한루프에 빠진다면 전체 앱이 작동을 멈춘다.

비동기 연산의 결과를 노드앱에 전달하는 인기있는 방식은 다음과 같다.

■**Call-back Function**
■**Event Emitter**
■**Promise**

1. Call-back Function

다음은 fs 모듈을 사용하여 비동기식 방식으로 파일 내용을 읽는다.

readme.txt

```
i love u
```

file-read.js

```
var fs = require('fs');

fs.readFile('readme.txt', 'utf8', function (error, data) {
    if (error) {
        return console.error(error);
    }
    console.log(data);
});
```

콜백함수는 비동기식 함수의 인자로 전달되며 비동기식 함수의 연산이 끝나면 연산 결과를 콜백함수의 파라미터로 받으면서 호출된다. 관례적으로 함수 파라미터 정의에서 콜백함수는 마지막 인자로 지정한다.
관례적으로 콜백함수의 파라미터 정의에서 error 정보를 담고 있는 인자는 첫 번째로 사용한다.
이는 에러처리를 실수로 누락하는 일이 없도록 하기 위함이다.

이번에는 동기식 방식으로 파일 내용을 읽는 코드를 살펴보자.

file-read-sync.js

```
var fs = require('fs');

try {
    var data = fs.readFileSync('readme.txt', 'utf8');
    console.log(data);
} catch (error) {
    console.error(error);
}
```

동기식 코드에서는 try 문을 사용하여 에러를 처리한다. 하지만, **비동기식 코드에서는 try 문으로 에러를 처리할 수 없다**. 이미 코드는 다음으로 진행되었고 결과는 나중에 리턴되기 때문이다.
그러므로 비동기식 코드에서 try 문은 존재 의미가 없다. 대안으로 예외처리는 콜백함수에 포함시켜서 전달하고 비동기함수가 처리를 마치면 전달받은 콜백함수를 사용하는 방식으로 처리한다.

콜백함수 사용 시 코드처리의 흐름

guess-console-log.js

```
var fs = require("fs");
var sourceFile = './why-use-var-count.js';
var targetFile = './new-file-copyed.js';

fs.chmod(sourceFile, 777, function (err) {
    if (err) {
        console.log(err.message);
        sourceFile = './new-file-copyed.js';
        targetFile = './why-use-var-count.js';
    }

    console.log('1');
    fs.rename(sourceFile, targetFile, function (err) {
        console.log('2');
        fs.lstat(targetFile, function (err, stats) {
            console.log('3');
            console.log(stats.size);
        });
        console.log('4');
    });
    console.log('5');
});
```

반복문에 의한 다수의 콜백함수들의 결과를 모아서 사용하기

why-use-var-count.js

```
var fs = require('fs');

function calculateBytes() {
    fs.readdir('.', function (err, filenames) {
        var count = filenames.length;
        var totalBytes = 0;
        var i;
        for (i = 0; i < filenames.length; i++) {
            fs.stat('./' + filenames[i], function (err, stats) {
                totalBytes += stats.size;
                count--;
                if (count === 0) {
                    console.log(totalBytes);
                }
            });
        }
    });
}

calculateBytes();
```

2. Event Emitter

비동기식 코드를 구현하는 두 번째 방법은 이벤트 전송자를 사용하는 것이다.
복수의 콜백함수를 사용해야 할 필요가 있을 때 적합한 방법이다.

emitter 로 불리는 어떤 종류의 객체를 이벤트 이름으로 정의된 특정 이벤트에 정기적으로 전달해 listener 로 불리는 함수 객체를 실행한다. 이벤트를 내보내는 모든 객체는 EventEmitter 클래스의 인스턴스다. 이 객체는 하나 이상의 함수를 이벤트로 사용할 수 있도록 이름을 넣어 추가하는 eventEmiter.on() 함수를 사용할 수 있다.

eventEmitter.on() 메소드로 등록된 리스너는 이벤트 이름이 호출되는 매 횟수만큼 실행된다.
eventEmitter.once()메소드로 등록한 리스너는 호출한 직후 제거되어 다시 호출해도 실행되지 않는다.

/event-emitter/counter.js

```
var EventEmitter = require('events').EventEmitter;

function Counter(limit) {
    var limit = limit || 1;

    EventEmitter.call(this);

    var self = this;
    var count = 0;

    this.start = function () {
        this.emit('start');

        var intervalId = setInterval(function () {
            ++count;
            self.emit('count', count);
            if (count >= limit) {
                clearInterval(intervalId);
            }
        }, 1000);
    };
}

Counter.prototype = Object.create(EventEmitter.prototype);

exports.Counter = Counter;
```

/event-emitter/use-counter.js

```
var Counter = require('./counter').Counter;

var stopWatch = new Counter(10);

stopWatch.on('start', function () {
    console.log('started');
});

stopWatch.on('count', function (count) {
    console.log('count: ' + count);
});

stopWatch.start();
```

새 객체는 프로토타입체이닝 연결에 따라 EventEmitter 의 프로토타입 객체에 있는 함수들을 사용할 수 있다. 위 경우는 on(), start() 함수가 이 경우에 해당한다.

/event-emitter/counter2.js

```
var util = require('util');
var EventEmitter = require('events').EventEmitter;

function Counter() {
    EventEmitter.call(this);

    var self = this;
    var count = 0;
    var intervalId = null;

    this.start = function (limit) {
        var limit = limit || 0;
        this.emit('start');

        intervalId = setInterval(function () {
            if (!limit) {
                self.emit('error', new Error('limit parameter missing.'));
                end();
                return;
            }

            ++count;
            self.emit('count', count);

            if (count >= limit) {
                end();
            }
        }, 1000);
    };

    this.stop = function () {
        end();
    };

    function end(){
        self.emit('stop');
        clearInterval(intervalId);
    }
}

util.inherits(Counter, EventEmitter);
```

```
exports.Counter = Counter;
```

/event-emitter/use-counter2.js

```
var Counter = require('./counter2').Counter;

var stopWatch = new Counter();

stopWatch.on('start', function () {
    console.log('started');
});

stopWatch.on('count', function (count) {
    console.log('count: ' + count);
});

stopWatch.on('stop', function () {
    console.log('stoped');
});

stopWatch.on('error', function (error) {
    console.log(error.message);
});

stopWatch.start(4);
```

3. Promise

프라미스는 아직 통보받지 못한 값을 표현하는 객체다.
프라미스는 비동기식 함수를 나중에 호출하겠다는 약속을 담은 객체다.
우선 프라미스 객체를 반환하여 처리의 흐름이 연속되게 만들고 미래의 콜백함수를 호출할 준비가 되었을 때 프라미스 객체에 설정된 콜백함수를 호출할 것임을 약속한다.

프라미스를 처음 생성하면 미결(pending) 또는 미이행(unfulfilled) 상태가 되며 관련된 비동기식 코드가 수행을 완료할 때까지 이 상태로 남는다.
비동기식 코드가 성공적으로 완료되면 프라미스는 이행(fulfilled) 상태로 변하며 비동기식 호출이 패하면 거절(rejected) 상태로 변한다.

/promise/test1.js

```
var fs = require('fs');

var promise = new Promise(function(resolve, reject){
  fs.readFile('data.txt', 'utf-8', function(error, data){
        if (error) {
              return reject(error);
        }

        resolve(data);
  });
});

promise.then(function(result){
  console.log(result);
}, function(error){
  console.log(error.message);
});
```

/promise/test2.js

```
var fs = require('fs');

var promise = new Promise(function(resolve, reject){
  fs.readFile('none.txt', 'utf-8', function(error, data){
        if (error) {
              return reject(error);
        }
```

```
            resolve(data);
    });
});

promise.then(function(result){
    console.log(result);
}).catch(function(error){
    console.log(error.message);
}).then(function(){
    console.log('The End');
});
```

/promise/test3.js

```
new Promise(function (resolve, reject) {
        setTimeout(function () {
                resolve(10);
        }, 3000);
})
.then(function (num) {
        console.log('first then: ', num);
        return num * 2;
})
.then(function (num) {
        console.log('second then: ', num);
        return num * 2;
})
.then(function (num) {
        console.log('last then: ', num);
});
```

프라미스를 보다 쉽게 사용할 수 있는 모듈을 살펴보자. 다음 모듈을 먼저 설치하고 다음을 진행한다.

```
npm install bluebird
```

/promise/test4.js

```
var Promise = require('bluebird');
var fs = Promise.promisifyAll(require('fs'));

fs.readFileAsync('data.txt')
  .then(function(fileData){
        return fs.writeFileAsync('message.txt', fileData);
  })
  .catch(function(error){
        console.log(error);
  })
  .finally(function(){
        console.log('done');
  });
```

there are times when you trigger multiple async interactions but only want to respond when all of them are completed.that's where Promise.all comes in.

The Promise.all method takes an array of promises and **fires one callback once they are all resolved**.

/promise/test5.js

```
var req1 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve('First!');
    }, 4000);
});

var req2 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        reject('Second!');
    }, 3000);
});

Promise.all([req1, req2]).then(function (results) {
    console.log('Then: ', results);
}).catch(function (err) {
    console.log('Catch: ', err);
});
```

instead of waiting for all promises to be resolved or rejected,
Promise.race triggers as soon as any promise in the array is resolved or rejected.

/promise/test6.js

```
var req1 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve('First!');
    }, 8000);
});

var req2 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve('Second!');
    }, 3000);
});

Promise.race([req1, req2]).then(function (one) {
    console.log('Then: ', one);
}).catch(function (one, two) {
    console.log('Catch: ', one);
});
```

18. TypeScript 소개

Learn TypeScript in 30 Minutes

http://tutorialzine.com/2016/07/learn-typescript-in-30-minutes/

Today we're going to take a look at TypeScript, a compile-to-JavaScript language designed for developers who build large and complex apps. It **inherits many programming concepts from languages such as C# and Java** that add more discipline and order to the otherwise very relaxed and free-typed JavaScript.

This tutorial is aimed at people who are fairly proficient in JavaScript but are still beginners when it comes to TypeScript. We've covered most of the basics and key features while including lots of examples with commented code to help you see the language in action. Let's begin!

The Benefits of Using TypeScript

JavaScript is pretty good as it is and you may wonder **Do I really need to learn TypeScript?**
Technically, you do not need to learn TypeScript to be a good developer, most people do just fine without it. However, working with TypeScript definitely has its benefits:

■ Due to the **static typing**, code written in TypeScript is **more predictable**, and is generally **easier to debug**.
■ Makes it easier to **organize the code** base for very large and complicated apps **thanks to modules, namespaces and strong OOP support**.
■ TypeScript has a **compilation step to JavaScript** that **catches all kinds of errors before** they reach **runtime** and break something.
■ The upcoming **Angular 2 framework is written in TypeScript** and it's recommended that developers use the language in their projects as well.

The last point is actually the most important to many people and is the main reason to get them into TypeScript. Angular 2 is one of the hottest frameworks right now and although developers can use regular JavaScript with it, a majority of the tutorials and examples are written in TS. As Angular 2 expands its community, it's natural that more and more people will be picking up TypeScript.

19. Interface

Interfaces are used to type-check whether an object fits a certain structure.

https://medium.com/@radekqwerty/typescript-how-to-check-that-argument-implements-interface-in-javascript-version-559e1bd2d83b#.7q2w3cd17

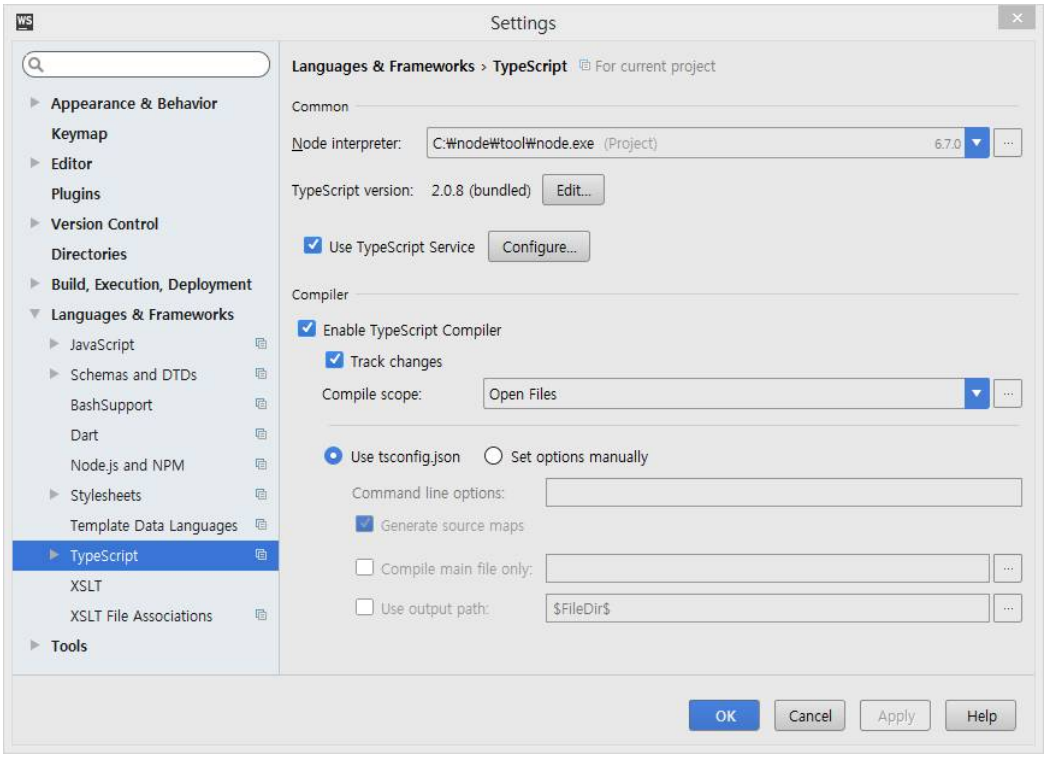
basic-concept.js

```
/**
 * @interface
 */
function Device() {}

Device.prototype.getName = function (){
    throw new Error('Not implemented!');
}

/**
 * @implements Device
 */
function DeviceImpl() {
    Device.call(this);
}

DeviceImpl.prototype = Object.create(Device.prototype);

var device = new DeviceImpl();

// Somewhere in code
if (!(device instanceof Device)) {
    throw new TypeError('Bad interface!');
} else {
    console.log('device instanceof Device');
}

/*
 Note, that in code above @interface and @implements from JSDoc are used.
 It could really help in everyday work — with it IDE can,
 for example, highlight class when not implement every methods from "interface"
 or when signature are mismatched.
 */
```

TypeScript Config in WebStorm

Transpiling TypeScript to JavaScript

https://www.jetbrains.com/help/webstorm/2016.3/transpiling-typescript-to-javascript.html



Interfaces in TypeScript

example.ts

```
interface IGreeter {
   greet(): string;
}

class Greeter implements IGreeter {
   greeting: string;

   constructor(message: string) {
      this.greeting = message;
   }

   greet(): string {
      return 'Hello, ' + this.greeting;
   }
}

var greeter = new Greeter('Everybody');
var result = greeter.greet();
console.log(result);
```

example.js

```
var Greeter = (function () {
      function Greeter(message) {
            this.greeting = message;
      }
      Greeter.prototype.greet = function () {
            return 'Hello, ' + this.greeting;
      };
      return Greeter;
}());
var greeter = new Greeter('Everybody');
var result = greeter.greet();
console.log(result);
//# sourceMappingURL=2.js.map
```

20. Abstract class

sample.ts

```
interface IGreeter {
  greet(): any;
  print(greeting: string): void;
}

// for compile-time type checking, we can use abstract class
abstract class BaseGreeter implements IGreeter {
  greet(): any {
    throw new Error('not implemented');
  }

  print(greeting: string) {
    console.log('Hello, ' + greeting);
  }
}

class MyGreeter extends BaseGreeter {
  private greeting: string;

  constructor(message: string) {
    super();
    this.greeting = message;
  }

  greet() {
    return 'Hello, ' + this.greeting;
  }
}

var myGreeter = new MyGreeter('Everybody');

var result = myGreeter.greet();
console.log(result);

myGreeter.print('Gentlemen');

(function sayHello(greeter: IGreeter) {
  if (!(greeter instanceof BaseGreeter)) {
```

```
        console.log('Something went wrong');
    }
    else {
        myGreeter.print('Gentlemen');
    }
})(myGreeter);
```

sample.js

```
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};
// for compile-time type checking, we can use abstract class
var BaseGreeter = (function () {
    function BaseGreeter() {
    }
    BaseGreeter.prototype.greet = function () {
        throw new Error('not implemented');
    };
    BaseGreeter.prototype.print = function (greeting) {
        console.log('Hello, ' + greeting);
    };
    return BaseGreeter;
}());
var MyGreeter = (function (_super) {
    __extends(MyGreeter, _super);
    function MyGreeter(message) {
        _super.call(this);
        this.greeting = message;
    }
    MyGreeter.prototype.greet = function () {
        return 'Hello, ' + this.greeting;
    };
    return MyGreeter;
}(BaseGreeter));
var myGreeter = new MyGreeter('Everybody');
var result = myGreeter.greet();
console.log(result);
myGreeter.print('Gentlemen');
(function sayHello(greeter) {
    if (!(greeter instanceof BaseGreeter)) {
```

```
            console.log('Something went wrong');
        }
        else {
            myGreeter.print('Gentlemen');
        }
})(myGreeter);
//# sourceMappingURL=3.js.map
```

21. Symbol

객체가 어떤 상태인지 추적할 수 있는 방법을 갖고 싶다고 가정해 봅시다. 예를 들어 객체의 부가적인 정보를 취급할 프로퍼티를 추가해서 객체의 상태정보를 관리하면 될 듯 합니다.

하지만, 이런 목적으로 프로퍼티를 추가한다면 for-in 이나 Object.keys()를 사용하는 기존 코드가 새로 추가한 프로퍼티 때문에 오작동을 할 수도 있습니다. 다른 라이브러리 작성자가 똑같은 기법을 더 먼저 생각한 관계로, 우리의 라이브러리가 기존 라이브러리와 충돌할 수도 있습니다.

이 문제를 회피하기 위해 어느 누구도 이름으로 사용하지 않을 만한 유니크한 프로퍼티 문자열이 필요하고 일반적인 로직에서는 무시되는 행동을 보인다면 좋겠습니다.

이러한 기능을 심볼을 사용하면 얻을 수 있습니다. 심볼은 이름 충돌의 위험 없이 프로퍼티 키로

사용할 수 있는 자바스크립트의 7 번째 자료형입니다.

심볼값은 유니크하면 변경할 수 없다. 심볼인지 판단하려면 typeof 연산자를 사용한다.
ES6 부터 객체의 프로퍼티 키로 문자열과 더불어 심볼을 사용할 수 있다.

객체의 프로퍼티들이 행동하는 방식을 조작하기 위한 용도로써 별도의 프로퍼티를 만들고자 한다.
별도의 프로퍼티는 객체를 대상으로 하는 메타데이터라고 할 수 있다. 객체는 상태와 기능을 가지므로 하나의 프로그램 단위로 볼 수 있으며 프로그램을 메타데이터로 조작하고자 하는 기술이므로 메타프로그래밍이라 부른다.

메타프로그래밍을 위한 별도의 프로퍼티가 예기치 않게 개발자가 추가하는 프로퍼티 키와 충돌하는 것을 원천적으로 방지하기 위하여 유니크한 값이 필요하였고 이에 따라서 심볼을 프로퍼티 키로 사용한다.

Object.getOwnPropertyNames() 메소드로 심볼 프로퍼티를 조회할 수 없다. for-in 구문에서도 심볼은 제외된다. 이는 객체의 프로그램이 메타데이터 프로퍼티에 의해 오작동하는 것을 막기 위해서 의도한 것이다.

Object.getOwnPropertySymbols() 메소드로 심볼 프로퍼티만을 조회할 수 있다.

Symbol.for() 메소드로 전역 심볼을 만들 수 있다. 이미 존재하는 서술로 심볼을 요청하면 기존 심볼을 반환하므로 이 경우 심볼의 유니크성은 깨진다. 하지만 심볼을 만든 위치로부터 외부 스코프에서 심볼을 키 값으로 객체의 값을 구하고자 할 때는 전역 심볼을 사용할 필요가 생긴다.

```
let obj = {};

(function () {
  let s1 = Symbol('name');
  let s2 = Symbol.for('age');
  obj[s1] = 'Chris';
  obj[s2] = 18;
})();

// console.log(obj[s1]); // s1 is not defined
console.log(obj[Symbol('name')]); // undefined
console.log(obj[Symbol.for('age')]); // 18
```

심볼 생성 시 new 키워드를 사용하지 않는다. Symbol('주석') 안에 있는 '주석' 문자열은 설명이다.

이 문자열은 디버깅할 때 유용하다. 심볼 값을 console.log()로 찍거나, .toString()을 써서 심볼 값을 문자열로 바꾸거나, 에러 메시지에서 참조할 경우 이 문자열을 출력하여 구분하는 용도로만 사용한다.

배열의 요소처럼, 심볼을 키로 갖는 속성(symbol-keyed property)은 obj.name 같이 점(dot)을 사용해서 접근할 수 없다. 심볼을 키로 갖는 속성은 반드시 대괄호 "[ ]" 기호를 써서 접근해야 한다.

"if (mySymbol in obj)" 처럼 속성을 참조하거나, "delete obj[mySymbol]"처럼 속성을 제거하는 것도 가능하다. 새로운 API Reflect.ownKeys(obj)는 문자열 키(key)와 심볼 키(key)를 모두 리턴한다.

심볼에 관련된 주의사항이 있다. 랭귀지의 다른 요소들과 달리, 심볼은 문자열로 자동 변환되지 않는다. 심볼을 다른 문자열에 이어 붙이려고 하면 TypeError 가 발생할 것이다.

```
 var sym = Symbol("mySymbol");
 var str = "your symbol is " + sym;
// TypeError: can't convert symbol to string
```

```
 Var str = `your symbol is ${sym}`;
// TypeError: can't convert symbol to string
```

String(sym) 또는 sym.toString()을 써서 심볼을 명시적으로 문자열로 변환하면 TypeError 를 피할 수 있다.

심볼의 종류

심볼을 얻는 3 가지 방법이 있습니다.

1. Symbol()

Symbol()을 호출합니다. 이미 논의한 바와 같이, 이 메소드는 호출될 때마다 새롭고 고유한 심볼을 리턴합니다.

2. Symbol.for(string)

Symbol.for(string)을 호출합니다. 이 메소드는 심볼 레지스트리(symbol registry)라고 불리는 심볼들의 목록을 참조합니다. Symbol() 메소드가 만드는 고유한 심볼들과 달리, 심볼 레지스트리에 있는 심볼들은 공유됩니다. 만약 당신이 Symbol.for("cat")을 30 번 호출한다면, 이 메소드는 매번 같은 심볼을 리턴할 것입니다.

3. Well-known 심볼

Symbol.iterator 처럼 표준에 의해 미리 정의된 심볼을 사용합니다. 표준이 자체적으로 정의하는 심볼이 몇 개 있습니다. 표준에 의해 정의된 심볼들은 나름의 특별한 용도가 있습니다.

ES6 가 스스로 정의한 심볼은 일부 브라우저에서 구현되지 않았습니다. 이 이용 사례들 중 어떤 것도 여러분이 매일 쓰는 코드에 큰 영향을 주고 있지 않습니다. 표준이 정의한 심볼들은 장기적인 관점에서 유용합니다. 표준이 정의한 심볼들은 PHP 와 Python 에 있는 __doubleUnderscores 을 JavaScript 방식으로 개선시킨 버전입니다. 장차 표준은 당신의 기존 코드에 아무런 위협을 주지 않고 랭귀지에 새로운 hook 을 추가하기 위해 미리 정의된 심볼들을 이용합니다.

이터레이션 규약

루프, 생성자가 어떤 객체의 값들을 순회하기 위한 인터페이스 구현 규칙이다.
이터레이터 규약, 이터러블 규약으로 나뉜다.

이터레이터 규약

다음 요소를 반환하는 next() 메소드를 구현해야 한다.

example-iterator.js

```
let obj = {
    array: [1, 2, 3],
    nextIndex: 0,
    next: function(){
        return this.nextIndex < this.array.length ?
        {value: this.array[this.nextIndex++], done: false} :
        {value: undefined, done: true};
    }
};

console.log(obj.next());
console.log(obj.next());
console.log(obj.next());
console.log(obj.next());
// { value: 1, done: false }
// { value: 2, done: false }
// { value: 3, done: false }
// { value: undefined, done: true }
```

이터러블 규약

[Symbol.iterator] 메소드를 구현해야 한다.

example-iterable.js

```
let array = [4, 5, 6];
let iter = array[Symbol.iterator]();

console.log(iter.next());
console.log(iter.next());
```

```
console.log(iter.next());
console.log(iter.next());
// { value: 4, done: false }
// { value: 5, done: false }
// { value: 6, done: false }
// { value: undefined, done: true }

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

let objX = {
	array: [7, 8, 9],
	nextIndex: 0,
	[Symbol.iterator]: function(){
		return {
			array: this.array,
			nextIndex: this.nextIndex,
			next: function(){
				return this.nextIndex < this.array.length ?
				{value: this.array[this.nextIndex++], done: false} :
				{value: undefined, done: true};
			}
		}
	}
};

let iterable = objX[Symbol.iterator]();

console.log(iterable.next());
console.log(iterable.next());
console.log(iterable.next());
console.log(iterable.next());
```

제너레이터

제너레이터는 한번에 하나씩 여러 값을 반환하는 함수다. 제너레이터 함수는 function* 으로 표기한다.
제너레이터 함수를 호출하면 즉시 바디를 실행하지 않고 제너레이터 객체의 새 인스턴스를 반환한다.

새 인스턴스는 이터레이션 규약을 구현한 객체다.
따라서 next() 메소드와 [Symbol.iterator] 메소드를 갖고 있다.

제너레이터 함수의 실행순서는 다음과 같다.

1. next() 메소드를 실행하면 제러레이터 함수 바디를 실행하다가 yield 키워드를 만나면 중지하고 yield 우측 표현식에 해당하는 값을 리턴한다.
next() 메소드 호출 시 파라미터를 전달할 수 있다.

2. 그리고 다시 next() 메소드를 실행하면 멈춘 지점부터 실행이 재개되고 그 다음 yield 키워드를 만날 때까지 코드를 실행한다.

3. 제너레이터 함수에 더 이상 yield 할 값이 없을 때 done 프로퍼티가 true 값을 갖고 있는 객체를 최종적으로 리턴한다.
제너레이터의 모든 yield 가 수행되기전 제너레이터의 return 메소드를 사용하여 언제라도 도중 하찰할 수 있으며 throw 메소드를 사용하여 예외를 제너레이터 함수에 전달할 수 있다.
제너레이터 함수 안에서 다른 이터러블 객체를 순회하려면 yield* 키워드를 사용하면 된다.

for-of

이터러블 객체를 next()로 한 단계씩 진행하는 불편함을 해소하기 위해 도입된 구문이다.
of 키워드 다음에 이터레이션규약 구현 객체를 배치하면 제너레이터 함수가 끝날 때까지 수행된다.

15 장. generator 를 참조한다.

22. Metaprogramming in ES6

with Symbols and why they're awesome

You've heard of ES6 right? It's the new version of JavaScript that is awesome in so many ways. I frequently wax lyrical about all of the amazing new features I keep discovering with ES6, much to the chagrin of my colleagues (consuming someone's lunch break talking about ES6 Modules seems to be not to everyone's liking).

A set of great new features for ES6 comes in the form of a slew of new metaprogramming tools, which provide low level hooks into code mechanics. Not much has been written on them, so I thought I'd do a teensy weensy 3 part post on them (sidebar; because I'm so lazy and this post has been sat in my drafts folder - 90% done - for three months, a bit more has been written about them since I said that):

Part 1: Symbols (this post)
https://www.keithcirkel.co.uk/metaprogramming-in-es6-part-2-reflect/

Part 2: Reflect
https://www.keithcirkel.co.uk/metaprogramming-in-es6-part-2-reflect/

Part 3: Proxies
https://www.keithcirkel.co.uk/metaprogramming-in-es6-part-3-proxies/

Metaprogramming

First, let's take a quick detour and discover the wonderful world of Metaprogramming.
Metaprogramming is (loosely) all about the underlying mechanics of the language,
rather than "high level" data modelling or business logic.

If programming can be described as "making programs", metaprogramming could be described as "**making programs making programs**" - or something. You probably use metaprogramming every day perhaps without even noticing it.

Metaprogramming has a few "subgenres" - one is **Code Generation**, aka eval & friends - which JavaScript has had since its inception (JS had eval in ES1, even before it got try/catch or switch statements). Pretty much every other language you'd reasonably use today has code generation features.

Another facet of metaprogramming is **Reflection** - finding out about and adjusting the structure and semantics of your application. JavaScript has quite a few tools for Reflection. Functions have Function#name and Function#length, as well as Function#bind, Function#call, and Function#apply.

All of the available methods on Object are Reflection, e.g. Object.getOwnProperties (As an aside, Reflection tools that don't alter code, but instead gather information about it are often called Introspection). We also have Reflection/Introspection operators, like typeof, instanceof, and delete.

Reflection is a really cool part of metaprogramming, because it allows you to alter the internals of how an application works. Take for example Ruby, in Ruby you can specify operators as methods which lets you override how those operators work when used against the class (sometimes called "**operator overloading**"):

```
class BoringClass
end

class CoolClass
  def ==(other_object)
    other_object.is_a? CoolClass
  end
end

BoringClass.new == BoringClass.new #=> false
CoolClass.new == CoolClass.new #=> true!
```

Compared to other languages like Ruby or Python, JavaScript's metaprogramming features are not yet as advanced - especially when it comes to nifty tools like Operator Overloading, but ES6 is starting to level the playing field.

Metaprogramming within ES6

The new APIs in ES6 come in three flavours: **Symbol, Reflect, and Proxy**.
Upon first glance this might be a little confusing - three separate APIs all for metaprogramming?
But it actually makes a lot of sense when you see how each one is split:

Symbols are all about Reflection within implementation - you sprinkle them on your existing classes and objects **to change the behaviour**.

Reflect is all about Reflection through introspection - used to discover very low level information about your code.

Proxy is all about Reflection through intercession - wrapping objects and intercepting their behaviours through traps.

So how does each one work? How are they useful? This post will cover Symbols, while the next two posts will cover Reflect and Proxy respectively.

Symbols - Reflection within Implementation

Symbols are a new primitive. Just like the Number, String, and Boolean primitives, Symbols have a Symbol function which can be used to create them. Unlike the other primitives, Symbols do not have a literal syntax (e.g how Strings have '') - the only way to make them is with the Symbol constructor-not-constructor-thingy:

```
Symbol(); // symbol
console.log(Symbol()); // prints "Symbol()" to the console
assert(typeof Symbol() === 'symbol')
new Symbol(); // TypeError: Symbol is not a constructor
```

Symbols have debuggability built in

Symbols can be given a description, which is really just used for debugging to make life a little easier when logging them to a console:

```
console.log(Symbol('foo')); // prints "Symbol(foo)" to the console.
assert(Symbol('foo').toString() === 'Symbol(foo)');
```

Symbols can be used as Object keys

This is where Symbols get really interesting. They are heavily intertwined with Objects. Symbols can be assigned as keys to Objects (kind of like String keys), meaning you can assign an unlimited number of unique Symbols to an object and be guaranteed that these will never conflict with String keys, or other unique Symbols:

```
var myObj = {};
var fooSym = Symbol('foo');
var otherSym = Symbol('bar');

myObj['foo'] = 'bar';
myObj[fooSym] = 'baz';
myObj[otherSym] = 'bing';

assert(myObj.foo === 'bar');
assert(myObj[fooSym] === 'baz');
assert(myObj[otherSym] === 'bing');
```

In addition to that, Symbols **do not show up** on an Object using **for in, for of or Object.getOwnPropertyNames** - the only way to get the Symbols within an Object is **Object.getOwnPropertySymbols**:

```
var fooSym = Symbol('foo');
var myObj = {};

myObj['foo'] = 'bar';
myObj[fooSym] = 'baz';

Object.keys(myObj); // -> [ 'foo' ]
Object.getOwnPropertyNames(myObj); // -> [ 'foo' ]
Object.getOwnPropertySymbols(myObj); // -> [ Symbol(foo) ]

assert(Object.getOwnPropertySymbols(myObj)[0] === fooSym);
```

This means **Symbols give a whole new sense of purpose to Objects** - they **provide a kind of hidden under layer to Objects** - not iterable over, not fetched using the already existing Reflection tools and **guaranteed not to conflict with other properties** in the object!

Symbols are completely unique...

By default, each new Symbol has a completely unique value. If you create a symbol (var mysym = Symbol()) it creates a completely new value inside the JavaScript engine. If you don't have the reference for the Symbol, you just can't use it. This also means two symbols will never equal the same value, even if they have the same description.

```
assert.notEqual(Symbol(), Symbol());
assert.notEqual(Symbol('foo'), Symbol('foo'));
assert.notEqual(Symbol('foo'), Symbol('bar'));

var foo1 = Symbol('foo');
var foo2 = Symbol('foo');
var object = {
      [foo1]: 1,
      [foo2]: 2,
};
assert(object[foo1] === 1);
assert(object[foo2] === 2);
```

...except when they're not.

Well, there's a small **caveat**(경고) to that - as there is also another way to make Symbols that can be easily fetched and re-used: **Symbol.for()**. This method creates a Symbol in a "global Symbol registry". Small aside: this registry is also cross-realm, meaning a Symbol from an iframe or service worker will be the same as one generated from your existing frame:

assert.notEqual(Symbol('foo'), Symbol('foo'));
assert.equal(Symbol.for('foo'), Symbol.for('foo'));

```
// Not unique:
var myObj = {};

var fooSym = Symbol.for('foo');
var otherSym = Symbol.for('foo');
```

```
myObj[fooSym] = 'baz';
myObj[otherSym] = 'bing';

assert(fooSym === otherSym);
assert(myObj[fooSym] === 'bing');
assert(myObj[otherSym] === 'bing');

// Cross-Realm
iframe = document.createElement('iframe');
iframe.src = String(window.location);
document.body.appendChild(iframe);

assert.notEqual(iframe.contentWindow.Symbol, Symbol);
assert(iframe.contentWindow.Symbol.for('foo') === Symbol.for('foo')); // true!
```

Having global Symbols does make things more complicated, but for good reason, which we'll get to.
Right now some of you are probably saying
"Argh!? How will I know which Symbols are unique Symbols and which Symbols aren't?",
to that I say
"it's okay, I got you, nothing bad is going to happen, we have **Symbol.keyFor()**":

```
var localFooSymbol = Symbol('foo');
var globalFooSymbol = Symbol.for('foo');

assert(Symbol.keyFor(localFooSymbol) === undefined);
assert(Symbol.keyFor(globalFooSymbol) === 'foo');
assert(Symbol.for(Symbol.keyFor(globalFooSymbol)) === Symbol.for('foo'));
```

What Symbols are, what Symbols aren't.

So we've got a good overview for what Symbols are, and how they work - but it's just as important to know **what Symbols are good for**, and what they're not good for, as they could easily be assumed to be something they're not:

Symbols will never conflict with Object string keys.
This makes them great **for extending objects** you've been given (e.g. as a function param) without affecting the Object in a noticeable way.
Symbols cannot be read using existing reflection tools.
You need the new Object.getOwnPopertySymbols() to access an Object's symbols, this makes Symbols great **for storing bits of information** you don't want people getting at through normal operation. Using Object.getOwnPropertySymbols() is a pretty special use-case.
Symbols are not private. The other edge to that sword - all of the Symbols of an object can be gotten by using Object.getOwnSymbols() - not very useful for a truly private value. Don't try to store information you want to be really private in an Object using a symbol - it can be gotten!
Enumerable Symbols can be copied to other objects using new methods like Object.assign. If you try calling Object.assign(newObject, objectWithSymbols) all of the (enumerable) Symbols in the second param (objectWithSymbols) will be copied to the first (newObject). If you don't want this to happen, make them non-enumerable with Object.defineProperty.
Symbols are not coercible(강압할수있는) into primitives. If you try to coerce a Symbol to a primitive (+Symbol(), ''+Symbol(), Symbol() + 'foo') it will throw an Error. This prevents you accidentally stringifying them when setting them as property names.
Symbols are not always unique. As mentioned above, **Symbol.for() returns you a non-unique Symbol**. Don't always assume the Symbol you have is unique, unless you made it yourself.
Symbols are nothing like Ruby Symbols. They share some similarities - such as having a central Symbol registry, but that's about it. They should not be used the same as Ruby symbols.

Okay, but what are Symbols really good for?

In reality, Symbols are just a slightly different way to attach properties to an Object - you could easily provide the well-known symbols as standard methods, just like Object.prototype.hasOwnProperty which appears in everything that inherits from Object (which is basically everything). In fact, other languages such as Python do just that - Python's equivalent of Symbol.iterator is __iter__, Symbol.hasInstance is __instancecheck__, and I guess Symbol.toPrimitive draws similarities with __cmp__. Python's way is, arguably, a worse approach though, as JavaScript Symbols don't need any weird syntax, and in no way can a user accidentally conflict with one of these special methods.

Symbols, in my opinion, can be used 2 fold:

1. As a unique value where you'd probably normally use a String or Integer:

Let's assume you have a logging library, which includes multiple log levels such as logger.levels.DEBUG, logger.levels.INFO, logger.levels.WARN and so on. In ES5 code you'd like make these Strings (so logger.levels.DEBUG === 'debug'), or numbers (logger.levels.DEBUG === 10). Both of these aren't ideal as those values aren't unique values, but Symbols are! So logger.levels simply becomes:

```
log.levels = {
    DEBUG: Symbol('debug'),
    INFO: Symbol('info'),
    WARN: Symbol('warn'),
};
log(log.levels.DEBUG, 'debug message');
log(log.levels.INFO, 'info message');
```

2. A place to put metadata values in an Object

You could also use them to store custom metadata properties that are secondary to the actual Object. Think of this as an extra layer of non-enumerability (after all, non-enumerable keys still come up in Object.getOwnProperties). Lets take our trusty Collection class and add a size reference, which is hidden behind the scenes as a Symbol (just remember that Symbols are not private - and you can - and should - only use them in for stuff you don't mind being altered by the rest of the app):

```
var assert = require('assert');

var size = Symbol('size');

class Collection {
  constructor() {
    this[size] = 0;
  }

  add(item) {
    console.log('this[size] = '+this[size]);
    this[this[size]] = item;
    this[size]++;
  }

  static sizeOf(instance) {
    return instance[size];
  }

}

var x = new Collection();
assert(Collection.sizeOf(x) === 0);

x.add('foo');
assert(Collection.sizeOf(x) === 1);

assert.deepEqual(Object.keys(x), ['0']);
assert.deepEqual(Object.getOwnPropertyNames(x), ['0']);
assert.deepEqual(Object.getOwnPropertySymbols(x), [size]);
```

3. Giving developers ability to add hooks to their objects, through your API

Ok, this sounds a little weird but bear with me. Let's pretend that we have a console.log style utility function - this function can take any Object, and log it to the console. It has its own routines for how it displays the given Object in the console - but you, as a developer who consumes this API, can **override** those **by providing a method, under a hook**: an inspect Symbol:

```
// Retreive the magic inspect Symbol from the API's Symbol constants
var inspect = console.Symbols.INSPECT;

var myVeryOwnObject = {};
console.log(myVeryOwnObject); // logs out `{}`

myVeryOwnObject[inspect] = function () { return 'DUUUDE'; };
console.log(myVeryOwnObject); // logs out `DUUUDE`
```

An implementation of this theoretical inspect hook could look a little something like this:

```
console.log = function (...items) {
    var output = '';
    for(const item of items) {
        if (typeof item[console.Symbols.INSPECT] === 'function') {
            output += item[console.Symbols.INSPECT](item);
        } else {
            output += console.inspect[typeof item](item);
        }
        output += '  ';
    }
    process.stdout.write(output + '₩n');
}
```

To clarify, this does not mean you should write code that modifies objects given to it. That would most definitely be a no-no (for this, have a look at WeakMaps which can provide ancillary objects for you to gather your own metadata on Objects).

Node.js already has similar behaviour with its implementation of console.log. Sort of. It uses String ('inspect') not a Symbol, meaning you can set x.inspect = function(){} - but this is clunky because it could clash with your classes methods, and occur by accident. Using Symbols is a very purposeful way for this kind of behaviour to happen.

This way of using Symbols is so profound, that it is actually part of the language, and with that we segue into the realm of well known Symbols…

Well Known Symbols

A key part of what makes Symbols useful, is a set of Symbol constants, known as "well known symbols". These are effectively a bunch of static properties on the Symbol class which are implemented within other native objects, such as Arrays, Strings, and within the internals of the JavaScript engine. This is where the real "Reflection within Implementation" part happens, as these well known Symbols alter the behaviour of (what used to be) JavaScript internals. Below I've detailed what each one does and why they're just so darn awesome!

Symbol.hasInstance: instanceof

Symbol.hasInstance is a Symbol which drives the behaviour of **instanceof**. When an ES6 compliant engine sees the instanceof operator in an expression it calls upon Symbol.hasInstance. For example, lho instanceof rho would call rho[Symbol.hasInstance](lho) (where rho is the right hand operand and lho is the **left hand operand**). It's then up to the method to determine if it inherits from that particular instance, you could implement this like so:

```
class MyClass {
    static [Symbol.hasInstance](lho) {
        return Array.isArray(lho);
    }
}
assert([] instanceof MyClass);
```

Symbol.iterator

If you've heard anything about Symbols, you've probably heard about Symbol.iterator. With ES6 comes a new pattern - the **for of** loop, which calls Symbol.iterator on **right hand operand** to get values to iterate over. In other words these two are equivalent:

```
var myArray = [1,2,3];

// with `for of`
for(var value of myArray) {
	console.log(value);
}

// without `for of`
var _myArray = myArray[Symbol.iterator]();
while(var _iteration = _myArray.next()) {
	if (_iteration.done) {
		break;
	}
	var value = _iteration.value;
	console.log(value);
}
```

Symbol.iterator will allow you to **override the of operator** - meaning if you make a library that uses it, developers will love you:

```
class Collection {
  *[Symbol.iterator]() {
    var i = 0;
    while(this[i] !== undefined) {
      yield this[i];
      ++i;
    }
  }
}

var myCollection = new Collection();
myCollection[0] = 1;
myCollection[1] = 2;

for(var value of myCollection) {
	console.log(value); // 1, then 2
```

```
}
```

Symbol.isConcatSpreadable

Symbol.isConcatSpreadable is a pretty specific Symbol - driving the behaviour of **Array#concat**. You see, Array#concat can take multiple arguments, which - if arrays - will themselves be flattened (or spread) as part of the concat operation. Consider the following code:

```
x = [1, 2].concat([3, 4], [5, 6], 7, 8);
assert.deepEqual(x, [1, 2, 3, 4, 5, 6, 7, 8]);
```

As of ES6 the way Array#concat will determine if any of its arguments are spreadable will be with Symbol.isConcatSpreadable. This is more used to say that the class you have made that extends Array won't be particularly good for Array#concat, rather than the other way around:

```
class ArrayIsh extends Array {
	get [Symbol.isConcatSpreadable]() {
		return true;
	}
}
class Collection extends Array {
	get [Symbol.isConcatSpreadable]() {
		return false;
	}
}
arrayIshInstance = new ArrayIsh();
arrayIshInstance[0] = 3;
arrayIshInstance[1] = 4;
collectionInstance = new Collection();
collectionInstance[0] = 5;
collectionInstance[1] = 6;
spreadableTest = [1,2].concat(arrayInstance).concat(collectionInstance);
assert.deepEqual(spreadableTest, [1, 2, 3, 4, <Collection>]);
```

Symbol.unscopables

This Symbol has a bit of interesting history. Essentially, while developing ES6, the TC found some old code in a popular JS libraries that did this kind of thing:

```
var keys = [];
with(Array.prototype) {
	keys.push('foo');
}
```

This works well in old ES5 code and below, but ES6 now has Array#keys - meaning when you do with(Array.prototype), **keys is now the method Array#keys** - not the variable you set. So there were three solutions:

1. Try to get all websites using this code to change it/update the libraries (impossible).
2. Remove Array#keys and hope another bug like this doesn't crop up (not really solving the problem)
3. Write a hack around all of this which prevents some properties being scoped into with statements.

Well, the TC went with option 3, and so Symbol.unscopables was born, which defines a set of "unscopable" values in an Object which should not be set **when used inside the with statement**. You'll probably never need to use this - nor will you encounter it in day to day JavaScripting, but it demonstrates some of the utility of Symbols, and also is here for completeness:

```
Object.keys(Array.prototype[Symbol.unscopables]);
// -> ['copyWithin', 'entries', 'fill', 'find', 'findIndex', 'keys']

// Without unscopables:
class MyClass {
	foo() { return 1; }
}
var foo = function () { return 2; };
with (MyClass.prototype) {
	foo(); // 1!!
}

// Using unscopables:
class MyClass {
	foo() { return 1; }
	get [Symbol.unscopables]() {
		return { foo: true };
	}
```

```
}
var foo = function () { return 2; };
with (MyClass.prototype) {
	foo(); // 2!!
}
```

Symbol.match

This is another Symbol specific to a function. String#match function will now use this to determine if the given value can be used to match against it. So, you can **provide your own matching implementation** to use, **rather than using Regular Expressions**:

```
class MyMatcher {
	constructor(value) {
		this.value = value;
	}
	[Symbol.match](string) {
		var index = string.indexOf(this.value);
		if (index === -1) {
			return null;
		}
		return [this.value];
	}
}
var fooMatcher = 'foobar'.match(new MyMatcher('foo'));
var barMatcher = 'foobar'.match(new MyMatcher('bar'));
assert.deepEqual(fooMatcher, ['foo']);
assert.deepEqual(barMatcher, ['bar']);
```

Symbol.replace

Just like Symbol.match, Symbol.replace has been added to allow custom classes, where you'd normally use Regular Expressions, for String#replace:

```
class MyReplacer {
	constructor(value) {
		this.value = value;
	}
	[Symbol.replace](string, replacer) {
		var index = string.indexOf(this.value);
		if (index === -1) {
			return string;
		}
		if (typeof replacer === 'function') {
			replacer = replacer.call(undefined, this.value, string);
		}
		return `${string.slice(0, index)}${replacer}${string.slice(index + this.value.length)}`;
	}
}
var fooReplaced = 'foobar'.replace(new MyReplacer('foo'), 'baz');
var barMatcher = 'foobar'.replace(new MyReplacer('bar'), function () { return 'baz' });
assert.equal(fooReplaced, 'bazbar');
assert.equal(barReplaced, 'foobaz');
```

Symbol.search

Yup, just like Symbol.match and Symbol.replace, Symbol.search exists to prop up String#search - allowing for custom classes instead of Regular Expressions:

```
class MySearch {
	constructor(value) {
		this.value = value;
	}
	[Symbol.search](string) {
		return string.indexOf(this.value);
	}
}
var fooSearch = 'foobar'.search(new MySearch('foo'));
var barSearch = 'foobar'.search(new MySearch('bar'));
var bazSearch = 'foobar'.search(new MySearch('baz'));
assert.equal(fooSearch, 0);
assert.equal(barSearch, 3);
assert.equal(bazSearch, -1);
```

Symbol.split

Ok, last of the String symbols - Symbol.split is for String#split. Use like so:

```
class MySplitter {
	constructor(value) {
		this.value = value;
	}
	[Symbol.split](string) {
		var index = string.indexOf(this.value);
		if (index === -1) {
			return string;
		}
		return [string.substr(0, index), string.substr(index + this.value.length)];
	}
}
var fooSplitter = 'foobar'.split(new MySplitter('foo'));
var barSplitter = 'foobar'.split(new MySplitter('bar'));
assert.deepEqual(fooSplitter, ['', 'bar']);
assert.deepEqual(barSplitter, ['foo', '']);
```

Symbol.species

Symbol.species is a pretty clever Symbol, it points to the constructor value of a class, which allows classes to create new versions of themselves within methods. Take for example Array#map, which creates a new Array resulting from each return value of the callback - in ES5 Array#map's code might look something like this:

```
Array.prototype.map = function (callback) {
    var returnValue = new Array(this.length);
    this.forEach(function (item, index, array) {
        returnValue[index] = callback(item, index, array);
    });
    return returnValue;
}
```

In ES6 Array#map, along with all of the other non-mutating Array methods have been upgraded to create Objects using the Symbol.species property, and so the ES6 Array#map code now looks more like this:

```
Array.prototype.map = function (callback) {
    var Species = this.constructor[Symbol.species];
    var returnValue = new Species(this.length);
    this.forEach(function (item, index, array) {
        returnValue[index] = callback(item, index, array);
    });
    return returnValue;
}
```

Now, if you were to make a class Foo extends Array - every time you called Foo#map while before it would return an Array (no fun) and you'd have to write your own Map implementation just to create Foos instead of Arrays, now Foo#map return a Foo, thanks to Symbol.species:

```
class Foo extends Array {
    static get [Symbol.species]() {
        return this;
    }
}

class Bar extends Array {
    static get [Symbol.species]() {
```

```
        return Array;
    }
}

assert(new Foo().map(function(){}) instanceof Foo);
assert(new Bar().map(function(){}) instanceof Bar);
assert(new Bar().map(function(){}) instanceof Array);
```

You may be asking "why not just use this.constructor instead of this.constructor[Symbol.species]?". Well, Symbol.species provides a customisable entry-point for what type to create - you might not always want to subclass and have methods create your subclass - take for example the following:

```
class TimeoutPromise extends Promise {
    static get [Symbol.species]() {
        return Promise;
    }
}
```

This timeout promise could be created to perform an operation that times out - but of course you don't want one Promise that times out to subsequently effect the whole Promise chain, and so Symbol.species can be used to tell TimeoutPromise to return Promise from it's prototype methods. Pretty handy.

Symbol.toPrimitive

This Symbol is the closest thing we have to overloading the Abstract Equality Operator (== for short). Basically, Symbol.toPrimitive is used when the JavaScript engine needs to convert your Object into a primitive value - for example if you do +object then JS will call object[Symbol.toPrimitive]('number');, if you do ''+object' then JS will call object[Symbol.toPrimitive]('string'), and if you do something like if(object) then it will call object[Symbol.toPrimitive]('default'). Before this, we had valueOf and toString to juggle with - both of which were kind of gnarly and you could never get the behaviour you wanted from them. Symbol.toPrimitive gets implemented like so:

```
class AnswerToLifeAndUniverseAndEverything {
    [Symbol.toPrimitive](hint) {
        if (hint === 'string') {
            return 'Like, 42, man';
        } else if (hint === 'number') {
            return 42;
        } else {
            // when pushed, most classes (except Date)
            // default to returning a number primitive
            return 42;
        }
    }
}

var answer = new AnswerToLifeAndUniverseAndEverything();
+answer === 42;
Number(answer) === 42;
''+answer === 'Like, 42, man';
String(answer) === 'Like, 42, man';
```

Symbol.toStringTag

Ok, this is the last of the well known Symbols. Come on, you've got this far, you can do this! Symbol.toStringTag is actually a pretty cool one - if you've ever tried to implement your own replacement for the typeof operator, you've probably come across Object#toString() - and how it returns this weird '[object Object]' or '[object Array]' String. Before ES6, this behaviour was defined in the crevices of the spec, however today, in fancy ES6 land we have a Symbol for it! Any Object passed to Object#toString() will be checked to see if it has a property of [Symbol.toStringTag] which should be a String, and if it is there then it will be used in the generated String - for example:

```
class Collection {

  get [Symbol.toStringTag]() {
    return 'Collection';
  }

}
var x = new Collection();
Object.prototype.toString.call(x) === '[object Collection]'
```

As an aside for this - if you use Chai for testing, it now uses Symbols under the hood for type detection, so you can write expect(x).to.be.a('Collection') in your tests (provided x has the Symbol.toStringTag property like above, oh and that you're running the code in a browser with Symbol.toStringTag).

The missing well-known Symbol: Symbol.isAbstractEqual

You've probably figured it out by now - but I really like the idea of **Symbols for Reflection**. To me, there is one piece missing that would make them something I'd be really excited about: Symbol.isAbstractEqual. Having a Symbol.isAbstractEqual well known Symbol could bring the abstract equality operator (==) back into popular usage. Being able to use it in your own way, for your own classes just like you can in Ruby, Python, and co. When you see code like **lho == rho** it could be converted into rho[Symbol.isAbstractEqual](lho), allowing classes to override what == means to them. This could be done in a backwards compatible way - by defining defaults for all current primitive prototypes (e.g. Number.prototype) and would tidy up a chunk of the spec, while giving developers a reason to bring == back from the bench.

Conclusion

What do you think about Symbols? Still confused? Just want to rant to someone? I'm @keithamus over on the Twitterverse - so feel free to hit me up there, who knows, one day I might be taking up your whole lunchtimes telling you about sweet new ES6 features I like way too much.

Now you're done reading all about Symbols, you should totally read Part 2 - Reflect.

Also lastly I'd like to thank the excellent developers @focusaurus, @mttshw, @colby_russell, @mdmazzola, and @WebReflection for proof reading this, and making much needed improvements.

23. String 에 추가된 메소드

newly-added-method.js

```
/*
	String.repeat
 */
console.log('A'.repeat(5)); // AAAAA

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

var str = 'Hello World';

console.log(str.indexOf('World')); // 6
console.log(str.indexOf('world')); // -1

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

/*
	String.includes
 */
console.log(str.includes('World')); // true
console.log(str.includes('world')); // false

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

function isExist(str, keyWord){
	var regex = new RegExp(keyWord, 'i');
	return regex.test(str);
}

console.log(isExist(str, 'World')); // true
console.log(isExist(str, 'world')); // true

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

/*
	String.startsWith
 */
console.log(str.startsWith('Hello')); // true

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

```
/*
	String.endsWith
 */
console.log(str.endsWith('World')); // true
```

24. Proxy

프록시는 타겟 객체에 부가적인 로직을 적용하기 위해서 사용하는 타겟 객체의 래퍼 객체다.
타겟 객체의 동작을 가로채는 함수를 트랩이라 부르며 트랩을 포함한 객체를 핸들러라 부른다.
트랩 함수 내부에서 this 는 항상 핸들러를 가리킨다.

Symbol

심볼은 심볼을 객체의 프로퍼티 키로 사용하여 객체의 행동을 조작하는 메소드를 추가합니다.

Reflect

리플렉트는 객체의 상태를 조사하는 새롭게 추가된 라이브러리입니다.

Proxy

프록시는 타겟 객체를 감싸서 객체의 행동을 트랩을 사용하여 조작하고자 할 때 사용합니다.

프록시 만들기

기본 문법은 다음과 같습니다.

```
new Proxy(target, {/*handler hooks*/});
```

target : 원래 로직을 제공하는 객체
handler hooks : 타겟 객체의 행동을 가로채서 추가적인 로직을 수행하는 함수들을 정의한 객체

타겟 객체를 조작하면 프록시 객체도 영향을 받고 프록시 객체를 조작하면 타겟 객체도 영향을 받는다.

make-proxy.js

```
var target = {};
var proxy = new Proxy(target, {/*handler hooks*/});

console.log(target !== proxy); // true

target.foo = true;
console.log(proxy.foo === true); // true

proxy.bar = true;
console.log(target.bar === true); // true
```

핸들러 객체는 훅 함수들을 갖고 있다.

프록시와 리플렉트는 연계되어 잇다.

Reflect methods/Proxy handler hooks 리스트

apply (call a function with a this argument and a set of arguments)

construct (call a class/constructor function with a set of arguments and optional constructor for prototype)

defineProperty (define a property on an Object, including metadata about its enumerability and whatnot)

getOwnPropertyDescriptor (get a properties "property descriptor": the metadata about an object property such as its enumerability)

deleteProperty (delete a property from an object)

getPrototypeOf (get an instances prototype)

setPrototypeOf (set an instances prototype)

isExtensible (determine if an object is "extensible" (can have properties added to it))

preventExtensions (prevent the object from being extensible)

get (target, property, receiver)

. 또는 [] 기호를 사용해서 프로퍼티 값을 조회할 때 기동한다.

수신자는 프로퍼티 조회 시 이용된 객체다. obj.name 으로 조회했다면 obj 이고 proxy.name 으로 조회했다면 proxy 가 된다.

set (target, property, value, receiver)

has (target, property)

in 연산자로 프로퍼티가 존재하는지 확인할 때 기동한다.

ownKeys (retreive all of the own keys of a Object: the keys it has, but not the keys its prototype has)

enumerate(target)

Reflect.enumerate, for-in 루프로 프로퍼티 키를 순회할 때 기동한다.

Proxy's default behaviour essentially implements Reflect calls for every handler hook (the internal mechanics of JavaScript engines might be slightly different, but we can just pretend that unspecified hooks will just default to their Reflect counterparts). This also means that any hook you don't specify will behave just like it normally would, as if it was never proxied:

```
// Here is a Proxy where we're defining the same behaviour as the default:
var proxy = new Proxy({}, {
    apply: Reflect.apply,
    construct: Reflect.construct,
    defineProperty: Reflect.defineProperty,
    getOwnPropertyDescriptor: Reflect.getOwnPropertyDescriptor,
    deleteProperty: Reflect.deleteProperty,
    getPrototypeOf: Reflect.getPrototypeOf,
    setPrototypeOf: Reflect.setPrototypeOf,
    isExtensible: Reflect.isExtensible,
    preventExtensions: Reflect.preventExtensions,
    get: Reflect.get,
    set: Reflect.set,
    has: Reflect.has,
    ownKeys: Reflect.ownKeys
});
```

Using Proxy to...

```
function urlBuilder(domain) {
    var parts = [];

    var proxy = new Proxy(function() {
        var returnValue = domain + '/' + parts.join('/');
        parts = [];
        return returnValue;
    }, {
        has: function() {
            return true;
        },
        get: function(object, prop) {
            parts.push(prop);
            return proxy;
        }
    });

    return proxy;
}

var google = urlBuilder('http://google.com');
console.log(typeof google);
// function

console.log(google());
// http://google.com/

console.log(google.search.products.bacon.and.eggs());
// http://google.com/search/products/bacon/and/eggs
```

You could also use this same pattern to make a tree traversal fluent API, something like you might see as part of jQuery or perhaps a React selector tool:

Proxy.revocable(target, handler)
도중에 취소가 가능한 프록시다. revoke 메소드로 취소한다.

Bonus Round: Revocable Proxies

Proxies have one last trick up their sleeve: some Proxies can be revoked. To create a revocable Proxy, you need to use Proxy.revocable(target, handler) (instead of new Proxy(target, handler)), and instead of returning the Proxy directly, it'll return an Object that looks like { proxy, revoke(){} }. An example:

25. Reflect

Reflect is a new global Object (like JSON or Math) that provides a bunch of useful introspection methods. Introspection tools already exist in JavaScript; Object.keys, Object.getOwnPropertyNames, etc. So why the need for a new API when these could just be added to Object?

ES6 는 객체상태를 조사하고 조작하는 새로운 글로벌 객체 Reflect 를 도입했다. 기본의 Object 객체가 지원하는 메소드는 체계가 없고 쓰기 불편했다. Reflect 객체는 실패하는 경우 예외가 아닌 불리언 값을 리턴한다.

Reflect 는 함수 객체가 아니므로 호출할 수 없고 new 연산자와 같이 사용될 수 없다.

“Internal Methods”

All JavaScript specs, and therefore engines, come with a series of “internal methods”. Effectively these let the JavaScript engine perform essential operations on your Objects as it hops around your code. If you read through the spec, you’ll find these everywhere, things like [[Get]], [[Set]], [[HasOwnProperty]] and so on (if you’re having trouble sleeping, the full list of internal methods in ES5 Section 8.12/ES6 Section 9.1).

Some of these **“internal methods” were hidden** from JavaScript code, others were applied in part by some methods, and even if there were available, they were tucked away(감춰지다) inside various crevices(틈들), for example **Object.prototype.hasOwnProperty is an implementation of [[HasOwnProperty]]**, except not every object inherits from Object, and so you have to perform convoluted(구불구불한) incantations(주문) just to use it - for example:

```
// Happens more often than you might think (especially with new ES6 classes)
var myObject = Object.create(null);

console.log(myObject.hasOwnProperty === undefined); // true
```

```
// If you want to use hasOwnProperty on `myObject`:
console.log(Object.prototype.hasOwnProperty.call(myObject, 'foo')); // false
```

As another example, the **[[OwnPropertyKeys]] internal method gets all of an Objects String keys and Symbol keys as one Array**. The only way to get these (outside of Reflect) is to combined the results of Object.getOwnPropertyNames and Object.getOwnPropertySymbols:

```
var s = Symbol('foo');
var k = 'bar';

var o = {
    [s]: 1,
    [k]: 1
};

var keys = Object.getOwnPropertyNames(o).concat(Object.getOwnPropertySymbols(o));

console.log(keys); // [ 'bar', Symbol(foo) ]
```

Reflect methods

Reflect is effectively a collection of all of those "internal methods" that were available exclusively through the JavaScript engine internals, now exposed in one single, handy object. You might be thinking "yeah, but why not just attach these to Object like Object.keys, Object.getOwnPropertyNames etc are?". Here is why:

Reflect has methods that are not meant just for Objects, for example Reflect.apply - which targets a Function. Calling Object.apply(myFunction) would just look weird.

Having a single object to house these methods is a great way to keep the rest of JavaScript clean, rather than dotting Reflection methods throughout constructors and prototypes - or worse - globals.

typeof, instanceof, and delete already exist as Reflection operators - adding new keywords like this is not only cumbersome(다루기 어려운) for developers, but also a nightmare for backwards compatibility and explodes the number of reserved words.

Reflect.apply(타겟함수, 타겟함수 내 this [, 파라미터 배열])

타겟함수의 반환값을 그대로 반환한다.

Reflect.apply is pretty much just Function#apply - it takes a function, and calls it with a context, and an array of arguments. From this point on you could consider the Function#call/Function#apply versions deprecated. This isn't mind blowing, but it makes good sense. Here's how you use it:

```
var ages = [ 11, 33, 12, 54, 18, 96 ];

/*
      Function.prototype style:
 */
var youngest = Math.min.apply(Math, ages);
var oldest = Math.max.apply(Math, ages);
console.log(youngest); // 11
console.log(oldest); // 96

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

/*
      Reflect style:
 */
var youngest = Reflect.apply(Math.min, Math, ages);
var oldest = Reflect.apply(Math.max, Math, ages);
console.log(youngest); // 11
console.log(oldest); // 96
```

The real benefit of Reflect.apply over Function.prototype.apply is defensibility: any code could trivially(사소하게) change the functions call or apply method, leaving you stuck with(꼼짝 못하는) broken code or horrible workarounds(제 2 의 해결책). This doesn't really end up being a huge deal in the real world, but code like the following could certainly exist:

```
function totalNumbers() {
```

```
    return Array.prototype.reduce.call(arguments, function(total, next) {
        return total + next;
    }, 0);
}

totalNumbers.apply = function() {
    throw new Error('Aha got you!');
}

// totalNumbers.apply(null, [1, 2, 3, 4]); // throws Error('Aha got you!');

// The only way to defensively do this in ES5 code is horrible:
console.log(Function.prototype.apply.call(totalNumbers, null, [1, 2, 3, 4]) === 10); // true

//You could also do this, which is still not much cleaner:
console.log(Function.apply.call(totalNumbers, null, [1, 2, 3, 4]) === 10); // true

// Reflect.apply to the rescue!
console.log(Reflect.apply(totalNumbers, null, [1, 2, 3, 4]) === 10); // true
```

Reflect.construct (생성자, 파라미터 배열 [, prototype 으로 사용할 생성자])

새로 만드는 객체를 생성자의 prototype 객체와 연결하지 않고 다른 생성자에 prototype 객체와 연결하고자 할 때 사용한다.

Similar to Reflect.apply - this lets you call a Constructor with a set of arguments. This will work with Classes, and sets up the correct object so that Constructors have the right this object with the **matching prototype**. In ES5 land, you'd use the **Object.create(Constructor.prototype)** pattern, and **pass that to Constructor.call or Constructor.apply**. The difference with Reflect.construct is that rather than passing an object, you just pass the constructor - and Reflect.construct will handle all that jazz (alternatively, just **omit it and it'll default to the target argument**). The old style of doing this was quite cumbersome, the new style can be much more succinct(간결한), as little as a one liner(한 줄 코드):

```
class Greeting {

    constructor(name) {
        this.name = name;
    }

    greet() {
```

```
        return `Hello ${this.name}`;
    }

}

// ES5 style factory:
function greetingFactory1(name) {
    var instance = Object.create(Greeting.prototype);
    Greeting.call(instance, name);
    return instance;
}

// ES6 style factory
function greetingFactory2(name) {
    return Reflect.construct(Greeting, [name], Greeting);
}

// Or, omit the third argument, and it will default to the first argument.
function greetingFactory2(name) {
    return Reflect.construct(Greeting, [name]);
}

// Super slick ES6 one liner factory function!
const greetingFactory3 = (name) => Reflect.construct(Greeting, [name]);

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

let obj = greetingFactory3('Chris');

console.log(obj);
console.log(obj.greet());
```

Reflect.defineProperty (타겟객체, 프로퍼티 키, 프로퍼티 서술 객체)

객체에 새 프로퍼티를 정의하거나 기존 프로퍼티를 수정하는 메소드다.

Object.defineProperty 메소드는 수정된 객체를 반환하지만 Reflect.defineProperty 메소드는 성공여부의 불리언 값을 리턴한다.

프로퍼티 서술 객체 설정은 데이터 프로퍼티와 접근자 프로퍼티에 따라 예약된 프로퍼티 키 값들이 다르다. 데이터와 접근자 프로퍼티는 상호 변환이 가능하다.

데이터 프로퍼티

value : 프로퍼티에 할당된 값, 기본 값은 undefined
writable : 값의 변경 가능여부를 결정, 기본 값은 false
configurable : 프로퍼티 서술의 변경 가능여부를 결정, 기본 값은 false
enumerable : for-in, Object.keys 메소드로 프로퍼티의 열거 가능여부를 결정, 기본 값은 false

접근자 프로퍼티

get : 프로퍼티의 값을 조회하는 함수, 기본 값은 undefined
set : 프로퍼티의 값을 설정하는 함수, 기본 값은 undefined
configurable : 프로퍼티 서술의 변경 가능여부를 결정, 기본 값은 false
enumerable : for-in, Object.keys 메소드로 프로퍼티의 열거 가능여부를 결정, 기본 값은 false

Reflect.defineProperty, Object.defineProperty, Object.defineProperties, Object.create 메소드를 쓰지 않고 추가한 프로퍼티는 writable, configurable, enumerable 값들이 모두 true 인 데이터 프로퍼티로 설정된다.

Reflect.defineProperty pretty much takes over from Object.defineProperty - it lets you **define metadata about a property**. It fits much better here because Object.* implies that it acts on object literals (it is, after all, the Object literal constructor), while Reflect.defineProperty just implies(암시하다) that what you're doing is Reflection, which is more semantic(의미적인).

An important note is that Reflect.defineProperty - just like Object.defineProperty - will throw a TypeError for invalid targets, such Number or String primitives (Reflect.defineProperty(1, 'foo')). This is a good thing, because throwing errors for wrong argument types notifies you of problems much better than silently failing.

Once again, you could consider Object.defineProperty pretty much deprecated from here on out. Use Reflect.defineProperty instead.

```
function MyDate() {

}
```

```
/*
      Old Style, weird because we're using Object.defineProperty to define
      a property on Function (why isn't there a Function.defineProperty?)
 */
Object.defineProperty(MyDate, 'now', {
      value: () => currentms
});

console.log(Object.getOwnPropertyNames(MyDate));
// [ 'length', 'name', 'arguments', 'caller', 'prototype', 'now' ]

/*
      New Style, not weird because Reflect does Reflection.
 */
Reflect.defineProperty(MyDate, 'now', {
      value: () => currentms
});

console.log(Object.getOwnPropertyNames(MyDate));
// [ 'length', 'name', 'arguments', 'caller', 'prototype', 'now' ]
```

Reflect.getOwnPropertyDescriptor (타겟 객체, 프로퍼티 키)

객체 프로퍼티의 서술자를 조회한다.

This, once again, pretty much **replaces Object.getOwnPropertyDescriptor**, getting the descriptor metadata of a property. The key difference is that while Object.getOwnPropertyDescriptor(1, 'foo') silently fails, **returning undefined**, Reflect.getOwnPropertyDescriptor(1, 'foo') will **throw a TypeError** - it throws for invalid arguments, just like Reflect.defineProperty does. You're probably getting the idea by now - but Reflect.getOwnPropertyDescriptor pretty much deprecates Object.getOwnPropertyDescriptor.

```
var myObject = {};

Object.defineProperty(myObject, 'hidden', {
      value: true,
      enumerable: false
});

var theDescriptor = Reflect.getOwnPropertyDescriptor(myObject, 'hidden');
```

```
console.log(theDescriptor, {
      value: true,
      enumerable: true
});
// { value: true,
//    writable: false,
//    enumerable: false,
//    configurable: false } { value: true, enumerable: true }

// Old style
var theDescriptor = Object.getOwnPropertyDescriptor(myObject, 'hidden');

console.log(theDescriptor, {
      value: true,
      enumerable: true
});
// { value: true,
//    writable: false,
//    enumerable: false,
//    configurable: false } { value: true, enumerable: true }

console.log(Object.getOwnPropertyDescriptor(1, 'foo') === undefined); // true

// Reflect.getOwnPropertyDescriptor(1, 'foo'); // throws TypeError
```

Reflect.deleteProperty (타겟 객체, 프로퍼티 키)

객체 프로퍼티를 삭제하는 메소드로 delete 연산자와 기능이 같다.

Reflect.deleteProperty will, surprise surprise, delete a property off of the target object. Pre ES6, you'd typically write delete obj.foo, now you can write Reflect.deleteProperty(obj, 'foo'). This is slightly more verbose, and the semantics are slightly different to the delete keyword, but it has the same basic effect for objects. Both of them call the internal target[[Delete]](propertyKey) method - but the delete operator also "works" for non-object references (i.e. variables), and so it does more checking on the operand passed to it, and has more potential to throw:

```
var myObj = { foo: 'bar' };
delete myObj.foo;
```

```
assert(myObj.hasOwnProperty('foo') === false);

myObj = { foo: 'bar' };
Reflect.deleteProperty(myObj, 'foo');
assert(myObj.hasOwnProperty('foo') === false);
```

Once again, you could consider this to be the "new way" to delete properties - if you wanted to. It's certainly more explicit to its intention.

Reflect.getPrototypeOf (타겟 객체)

타겟 객체의 __proto__(또는 [[prototype]]) 프로퍼티가 가리키는 prototype 객체를 조회한다.

The theme of replacing/deprecating Object methods continues - this time Object.getPrototypeOf. Just like its siblings, the new Reflect.getPrototypeOf method will throw a TypeError if you give it an invalid target such as a Number or String literal, null or undefined, where Object.getPrototypeOf coerces the target to be an object - so 'a' becomes Object('a'). Syntax otherwise, is exactly the same.

```
var myObj = new FancyThing();
assert(Reflect.getPrototypeOf(myObj) === FancyThing.prototype);

// Old style
assert(Object.getPrototypeOf(myObj) === FancyThing.prototype);

Object.getPrototypeOf(1); // undefined
Reflect.getPrototypeOf(1); // TypeError
```

Reflect.setPrototypeOf (타겟 객체, 프로토타입 객체)

타겟 객체의 __proto__(또는 [[prototype]]) 프로퍼티가 가리키는 객체를 두 번째 파라미터로 받은 객체로 설정한다.

Of course, you couldn't have getPrototypeOf without setPrototypeOf. Now, Object.setPrototypeOf will throw for non-objects, but it tries to coerce the given argument into an Object, and also if the [[SetPrototype]] internal operation fails, it'll throw a TypeError, if it succeeds it'll return the target argument. Reflect.setPrototypeOf is much more basic - if it receives a non-object it'll throw a TypeError, but other than that, it'll just return the result of [[SetPrototypeOf]] - which is a Boolean indicating if the

operation was successful. This is useful because then you can manage the outcome without resorting to using a try/catch which will also catch any other TypeErrors from passing in incorrect arguments.

```
var myObj = new FancyThing();
assert(Reflect.setPrototypeOf(myObj, OtherThing.prototype) === true);
assert(Reflect.getPrototypeOf(myObj) === OtherThing.prototype);

// Old style
assert(Object.setPrototypeOf(myObj, OtherThing.prototype) === myObj);
assert(Object.getPrototypeOf(myObj) === FancyThing.prototype);

Object.setPrototypeOf(1); // TypeError
Reflect.setPrototypeOf(1); // TypeError

var myFrozenObj = new FancyThing();
Object.freeze(myFrozenObj);

Object.setPrototypeOf(myFrozenObj); // TypeError
assert(Reflect.setPrototypeOf(myFrozenObj) === false);
```

Reflect.isExtensible (타겟 객체)

타겟 객체에 새 프로퍼티를 추가할 수 있는지 체크한다.

Object.preventExtensions, Object.freeze, Object.seal 메소드로 확장할 수 없게 설정할 수 있다.

Ok, once again this one is just a replacement of Object.isExtensible - but its a bit more complicated than that. Prior to ES6 (so… ES5) Object.isExtensible threw a TypeError if you fed it a non-object (typeof target !== 'object'). ES6 semantics have changed this (Gasp! A change to the existing API!) so that passing in a non-object to Object.isExtensible will now return false - because non-objects are all not extensible. So code like Object.isExtensible(1) === false would throw, whereas ES6 runs the statement like you'd expect (evaluating to true).

The point of the brief history lesson is that Reflect.isExtensible uses the old behavior, of throwing on non-objects. I'm not really sure why it does, but it does. So technically Reflect.isExtensible changes the semantics against Object.isExtensible, but Object.isExtensible changed anyway. Here's some code to illustrate:

```
var myObject = {};
var myNonExtensibleObject = Object.preventExtensions({});

assert(Reflect.isExtensible(myObject) === true);
assert(Reflect.isExtensible(myNonExtensibleObject) === false);
Reflect.isExtensible(1); // throws TypeError
Reflect.isExtensible(false);   // throws TypeError

// Using Object.isExtensible
assert(Object.isExtensible(myObject) === true);
assert(Object.isExtensible(myNonExtensibleObject) === false);

// ES5 Object.isExtensible semantics
Object.isExtensible(1); // throws TypeError on older browsers
Object.isExtensible(false);   // throws TypeError on older browsers

// ES6 Object.isExtensible semantics
assert(Object.isExtensible(1) === false); // only on newer browsers
assert(Object.isExtensible(false) === false); // only on newer browsers
```

Reflect.preventExtensions (타겟 객체)

타겟 객체를 확장할 수 없게 설정한다.

This is the last method in the Reflection object that borrows from Object. This follows the same story as Reflect.isExtensible; ES5's Object.preventExtensions used to throw on non-objects, but now in ES6 it returns the value back, while Reflect.preventExtensions follows the old ES5 behaviour - throwing on non-objects. Also, while Object.preventExtensions has the potential to throw, Reflect.preventExtensions will simply return true or false, depending on the success of the operation, allowing you to gracefully handle the failure scenario.

```
var myObject = {};
var myObjectWhichCantPreventExtensions = magicalVoodooProxyCode({});

assert(Reflect.preventExtensions(myObject) === true);
assert(Reflect.preventExtensions(myObjectWhichCantPreventExtensions) === false);
Reflect.preventExtensions(1); // throws TypeError
Reflect.preventExtensions(false);   // throws TypeError
```

```
// Using Object.isExtensible
assert(Object.isExtensible(myObject) === true);
Object.isExtensible(myObjectWhichCantPreventExtensions); // throws TypeError

// ES5 Object.isExtensible semantics
Object.isExtensible(1); // throws TypeError
Object.isExtensible(false);   // throws TypeError

// ES6 Object.isExtensible semantics
assert(Object.isExtensible(1) === false);
assert(Object.isExtensible(false) === false);
```

Reflect.enumerate (타겟 객체)

타겟 객체의 열거 가능한 프로퍼티와 객체가 상속 받은 열거 가능 프로퍼티를 이터레이터 객체로 반환한다.

Update: This was removed in ES2016 (aka ES7). myObject[Symbol.iterator]() is the only way to enumerate an Object's keys or values now.

Finally a completely new Reflect method! Reflect.enumerate uses the same semantics as the new Symbol.iterator function (discussed in the previous article), both use the hidden [[Enumerate]] method that JavaScript engines are aware of. In other words, the only alternative to Reflect.enumerate is myObject[Symbol.iterator](), except of course the Symbol.iterator can be overridden, while Reflect.enumerate can never be overridden. Used like so:

```
var myArray = [1, 2, 3];
myArray[Symbol.enumerate] = function () {
   throw new Error('Nope!');
}
for (let item of myArray) { // error thrown: Nope!
}
for (let item of Reflect.enumerate(myArray)) {
   // 1 then 2 then 3
}
```

Reflect.get (타겟 객체, 프로퍼티 키 [, this])

타겟 객체의 프로퍼티 값을 조회한다. 프로퍼티가 접근자 프로퍼티일 경우 세 번째 파라미터로 get 메소드 내부의 this 값을 지정할 수 있다.

Reflect.get is also a completely new method. It's quite a simple method; it effectively calls target[propertyKey]. If target is a non-object, the function call will throw - which is good because currently if you were to do something like 1['foo'] it just silently returns undefined, while Reflect.get(1, 'foo') will throw a TypeError! One interesting part of Reflect.get is the receiver argument, which essentially acts as the this argument if target[propertyKey] is a getter function, for example:

```
var myObject = {
  foo: 1,
  bar: 2,
  get baz() {
    return this.foo + this.bar;
  },
}

assert(Reflect.get(myObject, 'foo') === 1);
assert(Reflect.get(myObject, 'bar') === 2);
assert(Reflect.get(myObject, 'baz') === 3);
assert(Reflect.get(myObject, 'baz', myObject) === 3);

var myReceiverObject = {
  foo: 4,
  bar: 4,
};
assert(Reflect.get(myObject, 'baz', myReceiverObject) === 8);

// Non-objects throw:
Reflect.get(1, 'foo'); // throws TypeError
Reflect.get(false, 'foo'); // throws TypeError

// These old styles don't throw:
assert(1['foo'] === undefined);
assert(false['foo'] === undefined);
```

Reflect.set (타겟 객체, 프로퍼티 키, 프로퍼티 값 [, this])

타겟 객체의 프로퍼티 값을 설정한다. 접근자 프로퍼티일 경우에는 네 번째 인자로 set 함수 내부의 this 값을 지정할 수 있다.

You can probably guess what this method does. It's the sibling to Reflect.get, and it takes one extra argument - the value to set. Just like Reflect.get, Reflect.set will throw on non-objects, and has a special receiver argument which acts as the this value if target[propertyKey] is a setter function. Obligatory code example:

```
var myObject = {
  foo: 1,
  set bar(value) {
    return this.foo = value;
  },
}

assert(myObject.foo === 1);
assert(Reflect.set(myObject, 'foo', 2));
assert(myObject.foo === 2);
assert(Reflect.set(myObject, 'bar', 3));
assert(myObject.foo === 3);
assert(Reflect.set(myObject, 'bar', myObject) === 4);
assert(myObject.foo === 4);

var myReceiverObject = {
  foo: 0,
};
assert(Reflect.set(myObject, 'bar', 1, myReceiverObject));
assert(myObject.foo === 4);
assert(myReceiverObject.foo === 1);

// Non-objects throw:
Reflect.set(1, 'foo', {}); // throws TypeError
Reflect.set(false, 'foo', {}); // throws TypeError

// These old styles don't throw:
1['foo'] = {};
false['foo'] = {};
assert(1['foo'] === undefined);
assert(false['foo'] === undefined);
```

Reflect.has (타겟 객체, 프로퍼티 키)

타겟 객체에 프로퍼티가 존재하는지 조회한다. 타겟 객체가 상속받은 프로퍼티도 체크 대상이다.

in 연산자와 기능 상 같다.

Reflect.has is an interesting one, because it is essentially the same functionality as the in operator (outside of a loop). Both use the [[HasProperty]] internal method, and both throw if the target isn't an object. Because of this there's little point in using Reflect.has over in unless you prefer the function-call style, but it has important use in other parts of the language, which will become clear in the next post. Anyway, here's how you use it:

```
myObject = {
  foo: 1,
};
Object.setPrototypeOf(myObject, {
  get bar() {
    return 2;
  },
  baz: 3,
});

// Without Reflect.has
assert(('foo' in myObject) === true);
assert(('bar' in myObject) === true);
assert(('baz' in myObject) === true);
assert(('bing' in myObject) === false);

// With Reflect.has:
assert(Reflect.has(myObject, 'foo') === true);
assert(Reflect.has(myObject, 'bar') === true);
assert(Reflect.has(myObject, 'baz') === true);
assert(Reflect.has(myObject, 'bing') === false);
```

Reflect.ownKeys (타겟 객체)

타겟 객체의 프로퍼티 키 값들을 담은 배열을 리턴한다. 상속한 프로퍼티는 제외한다.

This has already been discussed a tiny bit in this article, you see Reflect.ownKeys implements [[OwnPropertyKeys]] which if you recall above is a combination of Object.getOwnPropertyNames and Object.getOwnPropertySymbols. This makes Reflect.ownKeys uniquely useful. Lets see shall we:

```
var myObject = {
  foo: 1,
  bar: 2,
  [Symbol.for('baz')]: 3,
  [Symbol.for('bing')]: 4,
};

assert.deepEqual(Object.getOwnPropertyNames(myObject), ['foo', 'bar']);
assert.deepEqual(Object.getOwnPropertySymbols(myObject), [Symbol.for('baz'), Symbol.for('bing')]);

// Without Reflect.ownKeys:
var keys = Object.getOwnPropertyNames(myObject).concat(Object.getOwnPropertySymbols(myObject));
assert.deepEqual(keys, ['foo', 'bar', Symbol.for('baz'), Symbol.for('bing')]);

// With Reflect.ownKeys:
assert.deepEqual(Reflect.ownKeys(myObject), ['foo', 'bar', Symbol.for('baz'), Symbol.for('bing')]);
```

Conclusion

We've pretty exhaustively gone over every Reflect method. We've seen some are newer versions of common existing methods, sometimes with a few tweaks, and some are entirely new methods - allowing new levels of Reflection within JavaScript. If you want to - you could totally ditch Object.*/Function.* methods and use the new Reflect ones instead, if you don't want to - don't sweat it, nothing bad will happen.

Now, I don't want you to go away empty handed. If you want to use Reflect, then I've got your back - as part of the work behind this post, I submitted a pull request to eslint and as of v1.0.0, ESlint has a prefer-reflect rule which you can use to get ESLint to tell you off when you use the older version of Reflect methods. You could also take a look at my eslint-config-strict config, which has the prefer-reflect turned

on (plus a bunch of others). Of course, if you decide you want to use Reflect, you'll probably need to polyfill it; luckily there's some good polyfills out there, such as core-js and harmony-reflect.

What do you think about the new Reflect API? Plan on using it in your project? Let me know, in the comments below or on Twitter, where I'm @keithamus.

Oh - also don't forget, the third and final part of this series - Part 3 Proxies - will be out soon, and I'll try not to take 2 months to release it again!

Lastly, thanks to @mttshw and @WebReflection for scrutinising my work and making this post much better than it would have been.