자바스크립트
객체 지향 프로그래밍

작성자 : 박우영
프론트엔드개발팀 / UIT개발실

대외비

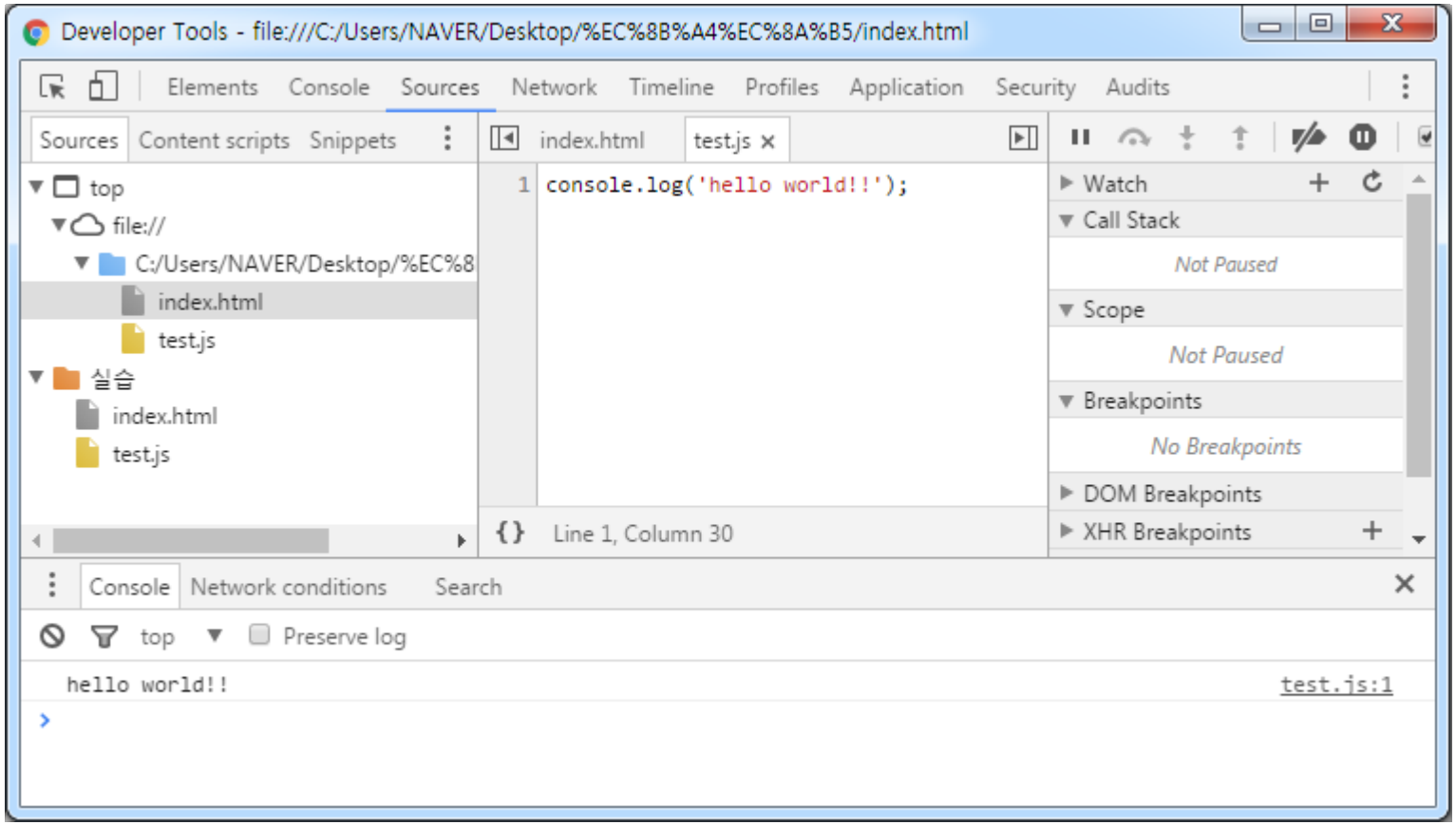
NHN
TECHNOLOGY SERVICES

목차

실습 환경 만들기

실습 환경 만들기

Workspace 추가하기



1.
생성자 함수

1. 생성자 함수

함수란

- 논리적으로 연관 있는 코드의 집합이다.
- 입력(argumets)과 출력(return)을 가진다.
- 자바스크립트에서 유효한 코드 블록을 가지는 구문 중 하나이다.
- 자바스크립트에서는 함수도 일급 객체이기 때문에 값으로 사용할 수 있다.
- 함수의 파라미터는 명시적으로 선언하지만 강제하지는 않는다.

함수 예제

```
function printLog() {
    console.log( 'sample!!' );
}

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

function sum() {
    var n = arguments.length;
    var result = 0;

    while(n--) {
        result += arguments[n];
    }

    return result;
}
```

```
printLog();

var addResult = add(1, 2);

var sumResult = sum(1, 2, 3, 4);
```

1. 생성자 함수

생성자 함수란

- 객체를 생성하기 위한 용도로 사용하는 함수를 의미한다.
- 생성자 함수를 정의하는 특별한 문법은 없다.
- 일반 함수와 구분을 위해 함수명을 대문자로 시작한다.
- 모든 사용자 정의 함수는 생성자 함수로 사용할 수 있다.
- Class 처럼 new 키워드를 사용하여 호출한다.

객체 리터럴 방식과의 차이점

- 동일한 속성과 메서드를 가진 객체를 반복적으로 만들 수 있다.

Class와의 차이점

- 프로토타입을 이용하여 다른 객체를 상속한다.

생성자 함수 호출

```
function Sample() {
}

var instance = new Sample();

console.log( instance );
```

일반 함수 호출과 생성자 함수 호출의 차이

```
function Sample() {
    console.log(this);
}

Sample();
new Sample();

console.log( Sample() );
console.log( new Sample() );
```

참고 - this의 참조

- 함수 내부의 this는 **함수를 참조하는 객체의 참조**이다.
- 생성자 함수로 호출 한 경우 this는 **생성자 함수 호출로 생성된 객체**이다.
- 호출하는 함수를 참조하는 객체가 없으면 this는 **window 객체**를 참조한다.
- this는 함수를 선언하는 시점이 아닌 **호출하는 시점**에 결정된다.
- Function.call, Function.apply, Function.bind 로 this의 참조를 지정할 수 있다.

참고 - this의 참조

```
var test = function(n) {
    console.log( '#' + n );
    console.dir( this );
};
var obj = {
    testMethod : test,

    callParam : function(fnc, n) {
        fnc(n);
    },

    callFnc : function(n) {
        var fnc = this.testMethod;

        fnc(n);
        this.testMethod(n + 1);
    }
};
```

```
test(1); //#1

new test(2); //#2

obj.testMethod(3); //#3

new obj.testMethod(4); //#4

obj.callParam(test, 5); //#5

obj.callFnc(6); //#6, #7
```

참고 - this의 참조 변경

```
function test() {
    console.log( arguments );
    console.dir( this );
}

var obj = {};

test.apply(obj, [1, 2, 3, 4]);
test.call(obj, 1, 2, 3, 4);

var testFnc = test.bind(obj);

testFnc(1, 2, 3, 4);
```

생성자 함수 예제

```
function Sample() {

    this.name = 'Sample';

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

}

var instance = new Sample();

console.log( instance.name );
console.log( instance.getName() );
```

생성자 함수 예제

```
function Sample(value) {

    this.name = value;

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

}

var instance1 = new Sample('ins1');
var instance2 = new Sample('ins2');

console.log( instance2.getName() );
console.log( instance2.getName() );
```

1. 생성자 함수

생성자 함수와 객체 리터럴 비교

```
function Sample() {

    this.name = 'Sample';

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

}

var sample = new Sample();

console.log(sample.name);
console.log(sample.getName());
```

```
var sample = {

    name: 'Sample',

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

};

console.log(sample.name);
console.log(sample.getName());
```

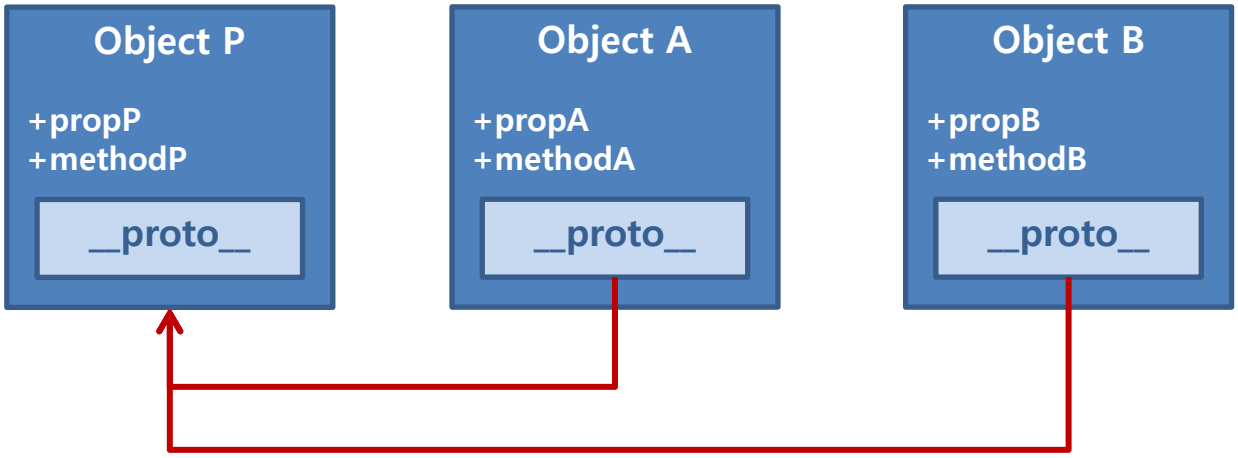
생성자 함수의 일반 호출시 예외 처리

```
function Sample1() {

    if (this === window) {
        throw "This is a Constructor";
    }

}


function Sample2() {

    if (this === window) {
        return new Sample2();
    }

}
```

2.
프로토타입

프로토타입이란

- 어떤 객체의 공통적인 속성과 메서드를 가지고 있는 객체이다.
- 일반적으로 객체들은 프로토타입으로 사용할 다른 객체의 참조를 가지고 있다.
- 프로토타입 객체도 프로토타입을 가지고 있다. (Object.prototype 객체는 예외)
- 프로토타입의 속성이나 메서드를 변경할 수도 있다.



객체의 프로토타입 참조

```
var obj = {};

var protoA = obj.__proto__;
var protoB = Object.getPrototypeOf(obj);

console.log( protoA );
console.log( protoA === protoB );
```

프로토타입 예제

```
var objA = {};
var objB = {};

objA.valueO = 'O';
objA.__proto__.valueP = 'P';

console.log( objA.valueO );
console.log( objB.valueO );

console.log( objA.valueP );
console.log( objB.valueP );

console.log( objA.__proto__ === objB.__proto__ );
```

Object.create를 이용하여 프로토타입 객체 설정

```
var obj = {
    name : "obj",
    getName : function() {
        return this.name;
    }
};

var objA = Object.create(obj);
var objB = Object.create(obj);

console.dir( objA );
console.dir( objB );

console.log( objA.__proto__ === objB.__proto__ );
```

Object.create를 이용하여 생성하는 객체의 속성 설정

```
var obj = {
    name : "obj",
    getName : function() {
        return this.name;
    }
};

var objA = Object.create(obj);

objA.myName = "objA";


var objB = Object.create(obj, {
    myName: {
        value: "objB"
    }
});
```

Object.create 대체 구현

```
function createObject(obj) {

    var F = function() {};

    F.prototype = obj;

    return new F();

}
```

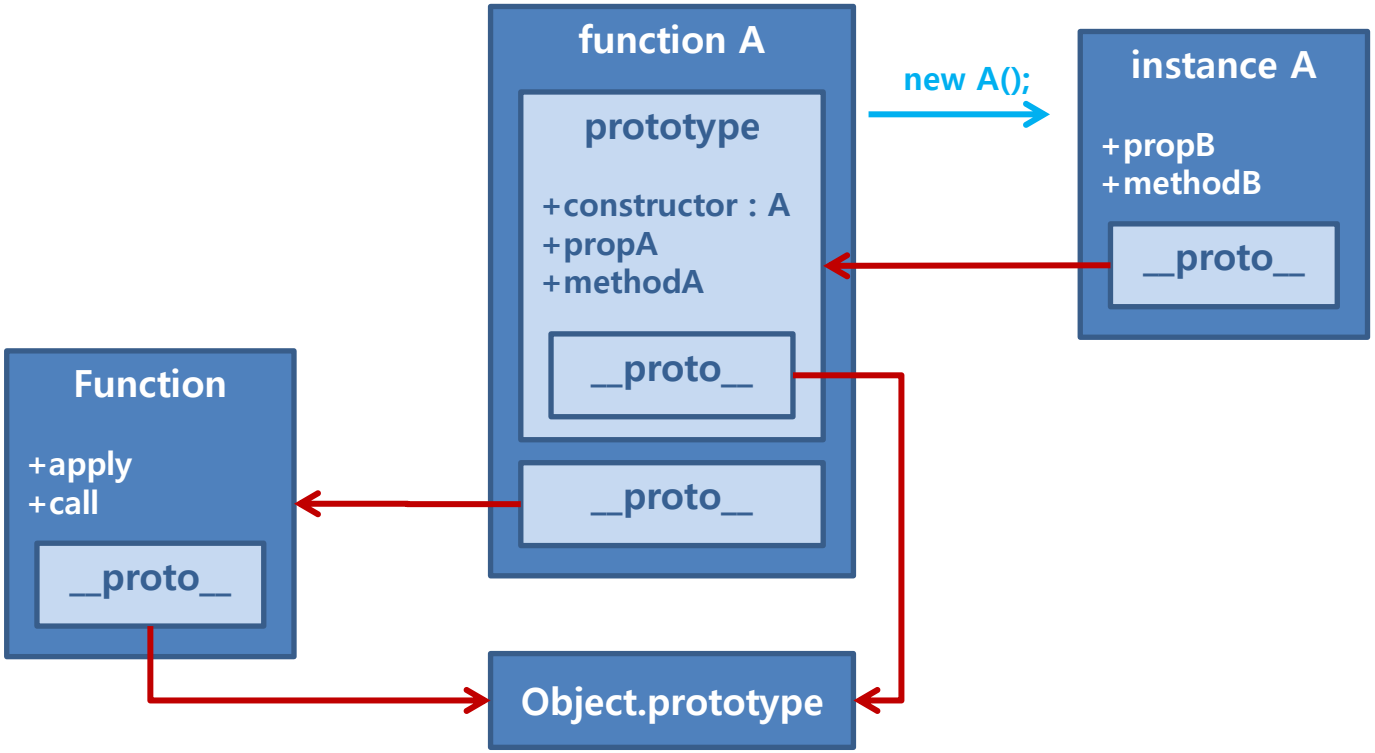
함수의 프로토타입

- 함수도 하나의 객체이므로 프로토타입(__proto__)을 가진다.
- 함수는 자신의 인스턴스가 프로토타입으로 참조할 객체를 prototype 속성으로 가지고 있다.
- prototype은 함수 자신을 constructor로 참조한다.

```
function Test() {
}

console.log( Test.__proto__ );
console.log( Test.prototype );

var ins = new Test();

console.log( ins.__proto__ );
console.log( ins.prototype );

console.log( Test.__proto__ === ins.__proto__ );
console.log( Test.prototype === ins.__proto__ );
```

2. 프로토타입

함수의 프로토타입



함수의 프로토타입 객체의 속성/메서드 설정

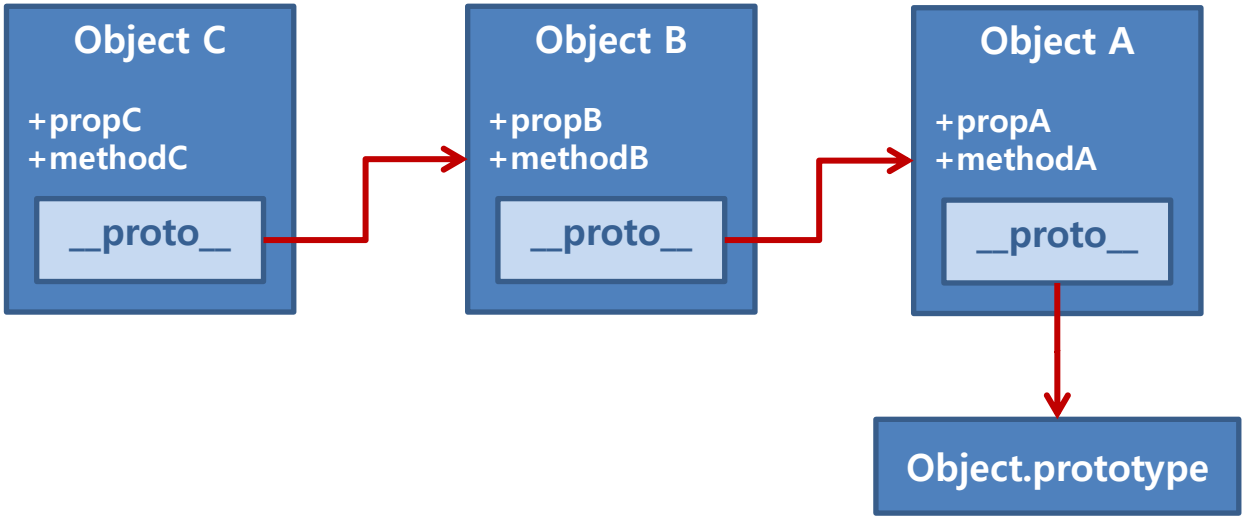
```
function Sample() {
}

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

var sample = new Sample();

console.log( sample.name );
console.log( sample.getName() );
```

함수의 프로토타입 객체 설정

```
function Sample() {
}

Sample.prototype = {

    constructor : Sample,

    name : "Sample",

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

};

var sample = new Sample();
```

프로토타입 체인이란

- 객체의 속성이나 메서드를 참조하는 프로토타입 사슬이다.
- 프로토타입 체인의 끝(최상위)은 Object.prototype이다.
- Object.prototype 객체의 프로토타입은 없다.



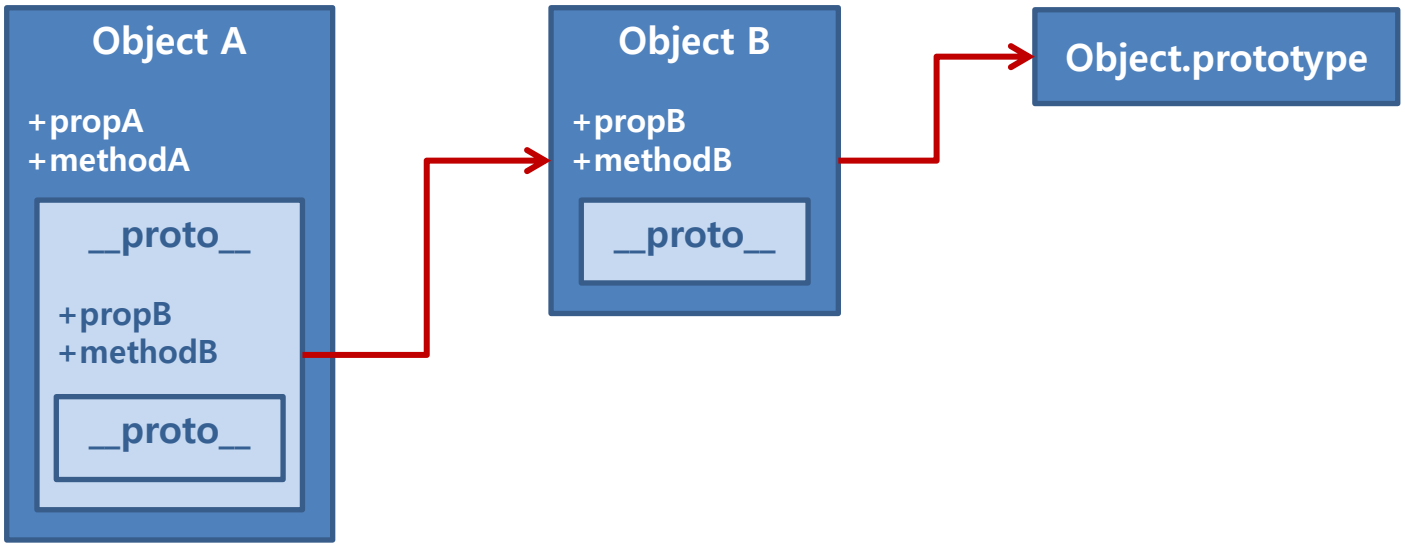
프로토타입 체인 예제

```
var objA = {
    propA : 0
};

var objB = Object.create(objA);
objB.propB = 1;

var objC = Object.create(objB);
objC.propC = 2;

console.dir( obj2 );
```

3.
상속

자바스크립트의 상속

- 어떤 객체가 다른 객체를 프로토타입으로 참조하는 것이다.
- 객체 자신이 속성이나 메서드를 가지고 있지 않으면 프로토타입 체인에서 탐색한다.



Object.create를 이용한 상속

```
var objA = {
    propA : 0
};

var objB = Object.create(objA);
objB.propB = 1;

var objC = Object.create(objB);
objC.propC = 2;

console.dir( obj2 );
```

객체 상속 함수 만들기

```
function extendObj(parent, child) {
    var obj = Object.create(parent);
    var prop;

    for (prop in child) {
        obj[prop] = child[prop];
    }

    return obj;
}

var proto = {
    name : "obj"
};

var myObj = extendObj(proto, {
    myName : "myObj"
});
```

생성자 함수를 이용한 상속

```
function Parent() {
    this.name = "Parent";
}

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


function Child() {
}

Child.prototype = Parent.prototype;

var child = new Child();
```

생성자 함수를 이용한 상속

```
function Parent() {
    this.name = "Parent";
}

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


function Child() {
}

Child.prototype = new Parent();

var child = new Child();
```

생성자 함수를 이용한 상속

```
function Parent() {
    this.name = "Parent";
}

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


function Child() {
    Parent.call(this);
}

Child.prototype = new Parent();

var child = new Child();
```

생성자 함수를 이용한 상속

```
function Parent() {
    this.name = "Parent";
}

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


function Child() {
    Parent.call(this);
}

Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;

var child = new Child();
```

프로토타입 속성에 있는 참조 타입 값을 그대로 사용할 때 발생하는 문제점

```
function Sample() {
}

Sample.prototype.arr = [];

var sample1 = new Sample();
var sample2 = new Sample();

sample1.arr.push(1);
console.log( sample2.arr );
```

```
function Sample() {
    this.arr = [];
}

Sample.prototype.arr = null;

var sample1 = new Sample();
var sample2 = new Sample();

sample1.arr.push(1);
console.log( sample2.arr );
```

생성자 함수에서 함수 표현식과 함수 선언문의 차이

```
var Sample1 = function() {
};

function Sample2() {
}

var sample1 = new Sample1();
var sample2 = new Sample2();

console.dir( sample1 );
console.dir( sample2 );
```

4.
모듈

4. 모듈

모듈이란

독립된 실행 영역에서 개별 단위 기능을 수행하는 프로그램의 일부분을 말한다.

모듈의 장점

- 코드를 일관성 있게 작성하고 관리할 수 있다.
- 개별 단위 기능으로 나뉘어 있기 때문에 코드의 관리와 재사용이 쉽다.
- 코드 전체를 분석하지 않아도 외부에 공개된 속성이나 모듈과 모듈의 관계만 파악하면 코드의 흐름이나 의존성을 파악할 수 있다.

모듈 패턴

함수를 이용하여 독립된 실행 영역을 가지도록 캡슐화 하는 패턴이다.

함수의 클로저를 이용하여 private 영역을 구현하고,

반환하는 객체를 이용하여 public 요소들을 제공한다.

모듈 패턴 예제

```
function createModule() {

    var name = '';

    return {
        setName: function(str) {
            name = str;
        },

        getName: function() {
            return name;
        }
    };

}

var module = createModule();
```

```
module.setName('test module');

console.log(module.name);
console.log(module.getName());
```

모듈 패턴 예제

```
var module = (function() {

    var name = '';

    return {
        setName: function(str) {
            name = str;
        },

        getName: function() {
            return name;
        }
    };

})();
```

```
module.setName('test module');

console.log(module.name);
console.log(module.getName());
```

생성자 함수와 모듈 패턴 비교

```
function Sample() {

    this.name = 'Sample';

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

}

var sample = new Sample();

console.log(sample.name);
console.log(sample.getName());
```

```
function createSample() {

    return {

        name: 'Sample',

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

    };

}

var sample = createSample();

console.log(sample.name);
console.log(sample.getName());
```

5.
의존성 관리

5. 의존성 관리

자바스크립트 개발 패러다임의 변화

예전에는 common.js, main.js 등 코드가 실행되는 영역으로 파일을 나누었으나,

큰 규모의 프로젝트는 개별 기능을 구현한 모듈 단위로 파일을 나눠서 관리한다.

```
<script src="jquery.js"></script>
<script src="common.js"></script>
<script src="main.js"></script>
```

```
<script src="jquery.js"></script>
<script src="nts.module.js"></script>
<script src="nts.class.js"></script>
<script src="avatar.config.js"></script>
<script src="avatar.utils.js"></script>
<script src="avatar.events.EventDispatcher.js"></script>
<script src="avatar.model.CategoryItems.js"></script>
<script src="avatar.controls.Adapter.js"></script>
<script src="avatar.model.BaseViewModel.js"></script>
<script src="avatar.model.AvatarViewModel.js"></script>
<script src="avatar.model.CategoryViewModel.js"></script>
<script src="avatar.model.PageViewModel.js"></script>
<script src="avatar.view.BaseView.js"></script>
<script src="avatar.view.HeaderView.js"></script>
<script src="avatar.view.CategoryAdapter.js"></script>
<script src="avatar.view.AvatarView.js"></script>
<script src="avatar.view.CategoryView.js"></script>
<script src="avatar.ui.ElementPool.js"></script>
<script src="avatar.ui.TemplateManager.js"></script>
<script src="avatar.view.ListView.js"></script>
<script src="avatar.view.CompletionView.js"></script>
<script src="avatar.view.PageView.js"></script>
<script src="naver.profile.avatar.js"></script>
```

자바스크립트의 의존성 관리

자바스크립트는 파일마다 별도의 실행 영역을 가지고 있지 않고 전역에서 실행된다.
이 때문에 특별한 규칙 없이 자유롭게 코드를 작성하더라도 다른 파일의 코드에 접근할 수 있다.
따라서 하나의 파일에서 외부 파일에 있는 코드를 사용할 경우,
외부 파일을 먼저 로드하여 해당 코드를 사용할 수 있는 상태로 만들어야 한다.

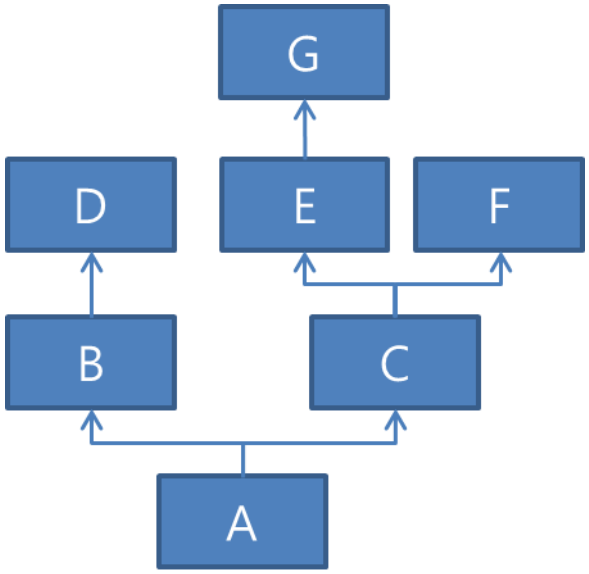
자바스크립트 의존성 관리의 문제점

ES2015에서 의존성을 관리할 수 있는 명세가 추가되긴 했지만, 아직은 언어 차원에서 의존성을 관리하는 장치가 없다. 그래서 많은 파일의 의존성을 관리하는 것이 쉽지 않다.

모듈을 이용한 의존성 관리의 자동화

- 하나의 모듈은 하나의 파일로 관리한다.
- 모듈을 정의하는 방식을 통일하여 개별 파일이 일관된 구조를 가지도록 한다.
- 의존 모듈을 명시적으로 선언하도록 한다.

```
var qs = require('querystring');

module.exports = {
    jsonToQuery: function(json) {
        return qs.stringify(JSON.parse(json));
    },

    queryToJson: function(query) {
        return JSON.stringify(qs.parse(query));
    }
};
```



5. 의존성 관리

모듈 관리 표준화의 필요성

모듈 관리를 자동화 하는 방법이 개발자나 프로젝트마다 달라서 외부의 모듈을 사용하려면 자동화 관리 방식에 맞게 수정이 필요하다.

대표적인 모듈 관리 방식

CommonJS와 AMD 두 가지 방식이 주류를 이루고 있다.

CommonJS

동기 방식의 모듈 관리

기준 파일에서 사용된 모듈을 탐색해서 미리 의존성에 맞게 모듈을 정의한 후에 코드를 실행하는 방식

NodeJS, Browserify, Webpack 등이 이 방식을 따른다.

```
// sample_module.js
var depModule = require('dependency_module');

/* private code... */

module.exports = {
    /* public code... */
};

// app.js
var moduleName = require('./sample_module');

/* works... */
```

5. 의존성 관리

AMD

비동기 방식의 모듈 관리

모듈을 참조하는 시점에 의존하는 모듈을 탐색해서 정의하고 코드를 실행하는 방식

RequireJS, crul.js 등이 이 방식을 따른다.

```
// sample_module.js
define([ 'dependency_module' ], function( depModule ) {
    /* private code... */

    return {
        /* public code... */
    };
});

// app.js
require([ './sample_module' ], function( sample ) {
    /* works... */
});
```

5. 의존성 관리

jQuery

```
function( global, factory ) {

    if ( typeof module === "object" && typeof module.exports === "object" ) {
        // For CommonJS and CommonJS-like environments where a proper `window`
        // is present, execute the factory and get jQuery.
        // For environments that do not have a `window` with a `document`
        // (such as Node.js), expose a factory as module.exports.
        // This accentuates the need for the creation of a real `window`.
        // e.g. var jQuery = require("jquery")(window);
        // See ticket #14549 for more info.
        module.exports = global.document ?
            factory( global, true ) :
            function( w ) {
                if ( !w.document ) {
                    throw new Error( "jQuery requires a window with a document" );
                }
                return factory( w );
            };
    } else {
        factory( global );
    }

// Pass this if window is not defined yet
}
```

jQuery-ui

```
function( factory ) {
    if ( typeof define === "function" && define.amd ) {

        // AMD. Register as an anonymous module.
        define([ "jquery" ], factory );
    } else {

        // Browser globals
        factory( jQuery );
    }
}
```

ECMA 표준 모듈 관리 방식

ES2015에서 모듈 관리를 위해 export와 import가 추가되었다.

자바스크립트 언어 차원에서 모듈과 모듈의 의존성을 관리할 수 있게 되었지만,
아직은 지원하는 브라우저가 없음(2016년 5월 기준)

다만 ES2015 문법을 기존 문법으로 변환해주는 Babel 같은 transpiler를 이용하거나,
React, AngularJS, Webpack 같은 프레임워크들이 ES2015 문법을 지원하고 있어 사용은 가능하다.

export / import

JS 파일에서 외부에 공개할 부분을 export를 이용해 지정하고 import를 통해 사용한다.

```js
// calculator.js

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

```js
// sampleA.js

import sum from './calculator';

export function addOne(a) {
    return sum(a + 1);
}

export function addTwo(a) {
    return sum(a + 2);
}
```

```js
// app.js

import * as sampleA from './sampleA';

console.log( sampleA.addOne(1) ); // 2
console.log( sampleA.addTwo(1) ); // 3
```

2일차 과제

2일차 과제 - 1

1일차 과제를 수정하여 아래 코드를 완성하세요.

```
var array = [ "14", 1, "a", 4, 9, "나", 0, true, 6, "d", null, "", 8 ];
var array2 = [ 6, "11", [], false, "오", "T", {}, 2, null, window ];
var numberArr, stringArr;
var objectArr;

// code...

console.log( numberArr ); // [ 1, 4, 9, 0, 6, 8 ]
console.log( stringArr ); // [ "14", "a", "나", "d", "" ]
console.log( objectArr ); // [ [], {}, null, window ]
```

작성 방법 : 기존 제출한 HTML 파일을 수정하여 제출

파일명 : JS_2_이름_사번.html

2일차 과제 - 2

다음과 같은 형태로 사용할 수 있는 calculator 객체를 구현하세요.

```
calculator.init(10);
console.log( calculator.add(1, 2, 1) ); // 14
console.log( calculator.subtract(3, 2) ); // 9
console.log( calculator.add(2) ); // 11
```

작성 방법 : HTML 파일에 자바스크립트 코드 작성

파일명 : JS_3_이름_사번.html

메일 제목 : [자바스크립트 인턴교육] 2일차 과제

메일 주소 : ascript@nhn.com, sungyu.lee@nhn.com

8월 10일 오전 10시 전까지 위의 메일 주소로 두 개의 html 파일을 첨부해서 보내주세요.

-

End of Document

-

Thank You.

-