Javascript Module

이항희 (javarouka@gmail.com)

시작하며

- 발표자가 왕 초보입니다.
- 자바스크립트에 약간의 기초가 필요할지도 모릅니다.
- 특성 상 코드 레벨에서 설명할 일이 많습니다.
  - 하지만 쉬운 코드이니 조금만 집중하신다면 어려움이 없을 겁니다.

차례

모듈화 - 나누어서 정복하라.

- **"작업의 세분화"**로 정의가능.
- 자신의 목표에는 충실하나, 그 외에는 자유로운 코드 단위라고 할 수 있음.
- 잘 정의된 세분화 작업은 타 작업에도 사용
- 캡슐화가 잘 되어있어야 재사용성도 높아짐.

"캡슐화?"

보여주고 싶은 것, 보여주기 싫은 것

- 적당한 신비주의를 쓰는 객체가 잘 나간다.
  - java에서는 public 과 private 등의 접근제한자로 구현
  - 외부의 잘못된 조작에 방어 능력이 생긴다.
  - 복잡해보이는 것보다, 심플한 것이 사용되기 편하다.

캡슐화가 잘 된 객체일수록 사용이 편하고 안전합니다

함수의 특징 #1,
함수는 팔방미인입니다

- Function이 할 수 있는 일
  - First-Class-Object
  - 인자를 받아 정의된 코드를 실행
  - 객체의 생성
  - 다른 함수의 실행에 파라미터로 사용
  - 함수 실행 결과 반환값으로 사용
  - 연산 가능
    - valueOf - 사칙 연산시
    - toString - 문자열 연산시

그리고, 변수 유효범위 (Variable Scope)를 정의할 수 있습니다.

함수의 특징 #2,
유효범위는 함수로 정의됩니다.

```
var coffee = "아메리카노"; // 전역에 coffee 변수가 있습니다.
function drink() {
      if(coffee == null) { // coffee에 null 체크를 해봅니다.
       var coffee = "카페모카"; // 없을 경우 새로운 변수 할당
      }
      console.log( " 역시 " + coffee + " 맛있어! " );
}
drink(); // ???
```

역시 카페모카 맛있어! 입니다.

자바스크립트는 "**{ }**" 가 아닌 "**함수**" 가 변수 유효범위를 제어하기 때문이죠.

Hoisting - 호이스팅

- 변수 정의는 함수 수준에서 수행됩니다.
- 변수 값 할당은 실행 시에 이뤄집니다.

```
// 방금 전의 코드는 실제 실행시에 아래와 같이 해석됩니다
function drink() {
   var coffee; // 먼저 선언됩니다. 전역 coffee를 가립니다.
   if(coffee == null) { // coffee에 null 체크를 해봅니다.
        coffee = "카페모카"; // 없을 경우 새로운 변수 할당
   }
   console.log( " 역시 " + coffee + " 맛있어! " );
}
```

함수는 중요합니다.

- 타 객체들이 가지지 못한 몇가지 기능을 더 가진 것이 함수입니다.

방금 본 유효범위 정의는 아주 중요한 특성이죠

서론이 길었네요

- 이제 모듈화의 조건 중 하나인 캡슐화부터 파헤쳐 봅시다.

JAVA 언어에서의 캡슐화

```
class Person {
    private String name;
    private int age;

    // 초기화되지 않으면 기본값 출력. 잘못된 접근 우회.
    public String getName() {
        if(name == null) name = "이름없음";
        return name;
    }
    public int getAge() { return age; }
}
Person person = new Person();
String n1 = person.name; // Oops. Exception!!
String n2 = person.getName(); // OK. 이름없음.
```

접근 제한자가 없어!

- Javascript 에는 접근제한자가 없습니다...

어떻게 캡슐화를 구현해야 할까요...?

Javascript 캡슐화 #1

- 객체의 속성은 모두 공개**(public)**되어 있습니다

```
var People = function(name, age) {
    this.name = name;
    this.age = age;
};
var javarouka = new People("이항희", 33);

console.log(javarouka.name); // 이항희가 출력되겠죠.
javarouka.age = 21; // 값도 외부에서 멋대로 조작할 수 있습니다.
```

객체안의 모든 속성은 외부에서 멋대로 조작이 가능합니다...
이래서는 캡슐화가 불가능합니다.

Javascript 캡슐화 #2

- 이 방법은 어떨까요?

```
var People = function(name, age) {
    var name = name; // this 키워드를 var로 바꿨습니다.
    var age = age;
};

var javarouka = new People("이항희", 33);

console.log(javarouka.name); // undefined 결과가 나옵니다
javarouka.age = 21; // 그냥 public인 속성age가 생깁니다
```

이 방법은 객체의 속성 자체에 접근할수가 없습니다...
모두 **private** 속성이 되어버렸습니다.

그럼 어떻게...?

- Javascript 에서는 공개할것만 공개할 수 있는 **잘 캡슐화된 객체**를 구현할 수 없을까요??

클로저 (Closure) 라는 것이 있습니다!

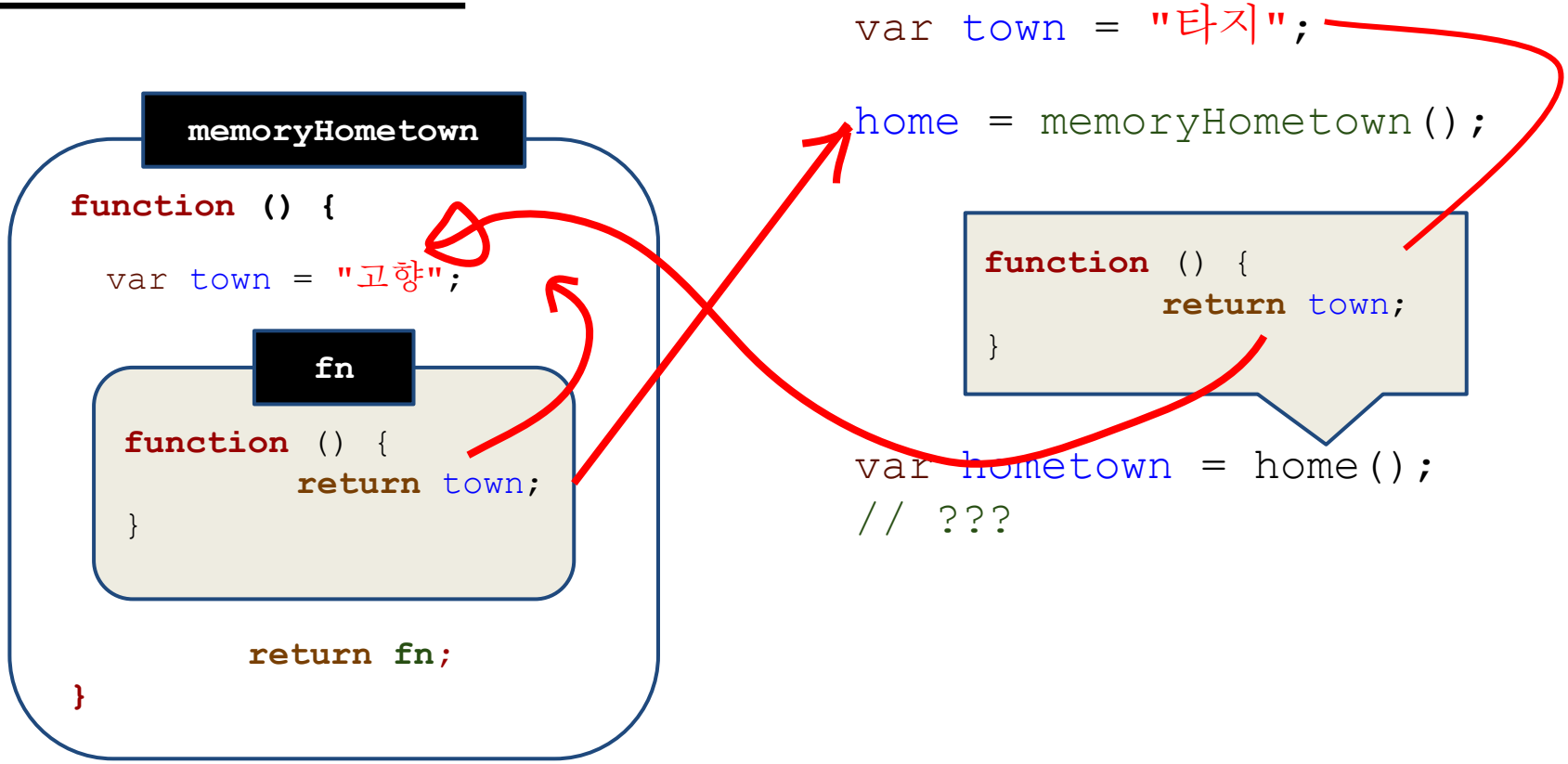
클로저 간단히 살펴보기... #1

- 자신이 태어난 고향을 기억하는 함수 (?)
  - 어머니 == 컨텍스트
  - 태어남 == 함수 선언
  - 고향집 == 변수 영역

정말 간단히만 살펴보지요

클로저 간단히 살펴보기... #2

Execution Context



클로저 간단히 살펴보기... #3

- 함수가 생성되면 그 함수는 자신이 생성된 환경을 변수 유효범위에 포함시킬 수 있습니다.
- 이 함수가 생성 환경을 반환값 등으로 벗어날때가 중요한데, 벗어나도 여전히 자신을 생성해준 환경을 참조가능합니다.
- 약간 어려운 말로, 어휘적 유효 범위(lexical scope) 라고도 합니다.

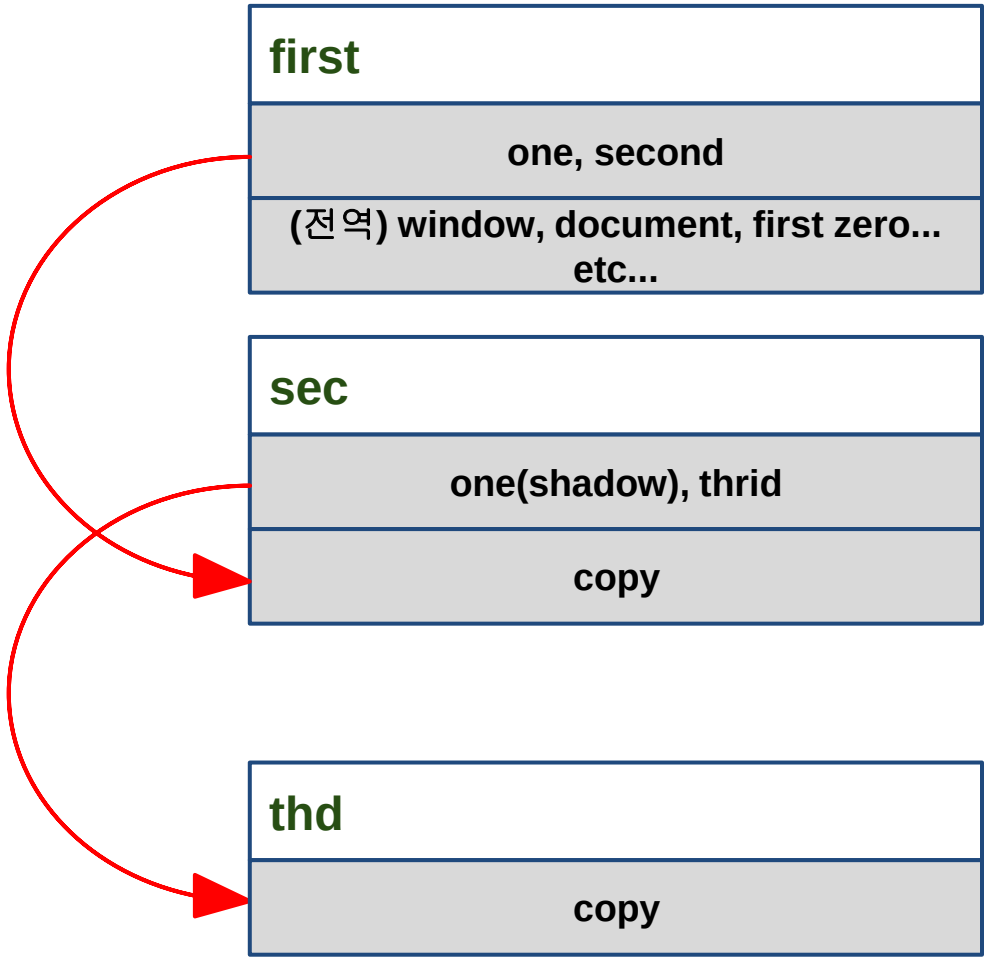
클로저 간단히 살펴보기... #4

```
var zero = 0;
function first() {
    var one = 1;

    function second() {
        var one = 2;

        function third() {
            return one;
        }
    }
    return second;
}

var sec = first();
var thd = sec();
```



이제 People에 적용할까요...

```
var People = function(name, age) {
    var name = name;
    var age = age;
    var publicProps = new Object();
    // 클로저 함수를 객체 속성에 할당
    publicProps.callName = function(){ return name; };
    publicProps.sayAge = function(){ return age+"살"; };

    return publicProps; // 객체를 리턴
};
var javarouka = new People("이항희", 33);
console.log(javarouka.name); // undefined
javarouka.age = 26; // 변경이 안됩니다.
console.log(javarouka.getAge()); // 이건 가능. 33살 출력.
```

정리해보죠...public 과 private 구현.

```
var JavaBeansStyle = function(value) {
   var val = value; // private
   function setValue(v) {
        if(v) val = v;
   };
   function getValue() {
        return val;
   };
   return {
        set: setValue, // 좀더 축약된 별칭으로 반환했습니다.
        get: getValue
   };
};
var module = JavaBeansStyle("모듈이군요");
module.get(); // 공개된 것만 호출할 수 있지요.
```

생짜로 모듈화 구현해보기.

- 이제 모듈화를 다른 도움 없이 순수 자바스크립트로 구현하면서 모듈화를 좀더 자세히 알아봅시다.

다수의 스크립트 사용시...

- 두 개발팀에서 보내온 스크립트 두개가 있고, **통합 작업**을 하려고 합니다.

```
// Coffee.js

function makeCoffee(n) {
    // 커피 만드는 코드
}

function drink(coffee) {
    // 커피 마시는 코드
}
```

```
// Tea.js

function makeTea(n) {
    // 차 만드는 코드
}

function drink(tea) {
    // 차 마시는 코드
}
```

이 두 스크립트를 같은 컨텍스트(페이지)에서 사용한다면 어떨까요?

drink 함수가 이름이 중복되어 덮어 써지겠죠. **(이름 충돌)**

전역공간 오염

- Global Scope Pollution
  - 전역 공간에 무분별한 변수 선언은 매우 나쁘다.
  - 캡슐화에 정 반대되는 코딩 방법
  - 모듈화가 불가능
    - 사실 가능은 하지만, 충돌위험이 매우 높다.
    - 최근같이 사이트의 외부 스크립트가 많아지는 시기에는 더욱 심각한 문제를 가져온다.
  - jQuery와 prototype.js 를 혼용 시 오류...
    - $ 변수 충돌.
    - 불필요한 jQuery.noConflict 함수 등이 나온 이유

해결책

- 익명 함수 (Anonymous Function) 사용.
  - 함수가 자신만의 스코프를 생성한다는 점을 이용
  - 실행 코드를 익명 클로저 함수로 감싸고 코딩

```
(function(bwin) {

    // 뭔가 하는 코드.
    // 이 코드 안에서는 어떤 변수를 선언해도
    // 전역변수에 영향이 없습니다.


})(window); // 익명함수 선언 후 바로 실행
```

돌아가서, 실제 적용해보죠.

- Coffee.js 파일에 적용했습니다.

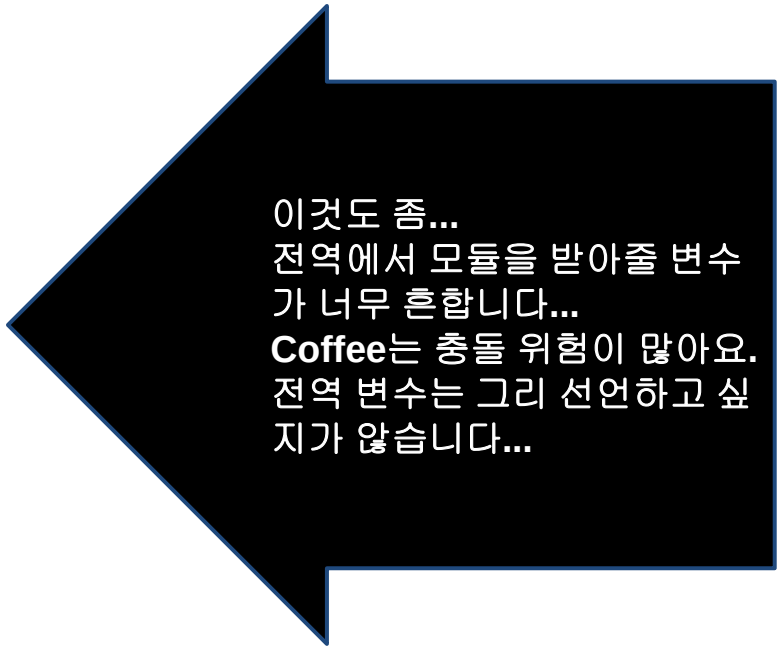
```
// 익명 함수의 사용
(function () {
    function makeCoffee(n) {
        // 커피 만드는 코드
    }
    function drink(coffee) {
        // 커피 마시는 코드
    }
})(); // 선언 후 바로실행
```

아무 의미도 없네요...
익명 함수 안에서 실행되고 실행 후 사라질 겁니다.
전역에 변수를 선언하지 않았지만,
어디서도 접근도 할수 없게 되었군요.

그렇다면 클로저를.

- 클로저를 적용했지만, 뭔가 아쉽습니다.

```
// 전역 변수 Coffee 에 할당
var Coffee = (function () {
    function makeCoffee(n) {
        // ...
    }
    function drink(coffee) {
        // ...
    }
    // 클로저 함수들을 반환
    return {
        make: makeCoffee,
        drink: drink
    }
})();
```



모듈 임포트, 익스포트

- 네임스페이스 사용
  - 특정 식별자를 통한 모듈 접근
  - 이 식별자는 유니크해야 한다.
    - 자바에서의 패키지 이름과 동일한 개념
    - 전통적인 도메인 이름의 역순 사용
    - 실제 파일시스템의 경로 사용
- 임포트, 익스포트
  - 네임스페이스를 모듈 실행시 인자로 전달하여 프로퍼티로 정의하거나 읽음

이제 완성해보죠

```
// index.js
// 네임스페이스 선언이라고 하죠. 보통 자바스크립트에서는 모듈 파일시스템 경로.

var me = new Object();
me.javarouka = new Object(); // 이제 여기에 모듈이 설정되겠네요.
```

```
// Coffee.js
(function(exports) {
    function makeCoffee(n) {
        // 커피 만드는 코드
    }
    function drink(coffee) {
        // 커피 마시는 코드
    }

    exports.Coffee = {
        make: makeCoffee,
        drink: drink
    }
})(me.javarouka);
```

```
// Tea.js
(function(exports) {
    function makeTea(n) {
        // 차 만드는 코드
    }
    function drink(coffee) {
        // 차 마시는 코드
    }

    exports.Tea = {
        make: makeTea,
        drink: drink
    }
})(me.javarouka);
```

사용.

```
// 전역 오염을 방지하기 위해 익명 함수를 썼습니다.
// 익명 함수로 감싸 코딩하는 건 매우 좋은 습관입니다.
(function(javaroukaModules) {

    // 사용모듈 찾기. java의 import 구문이라고 생각하면 됩니다.
    var coffee = javaroukaModules.Coffee;
    var tea = javaroukaModules.Tea;

    // 커피 만들고 마시기
    var aCoffee = coffee.makeCoffee(1);
    coffee.drink(aCoffee);

     // 차 만들고 마시기.
     var aTea = tea.makeCoffee(4);
     tea.drink(aTea);

})(me.javarouka) // 사용할 모듈 네임스페이스를 인자로 줍니다.
```

정리

- 모듈 패턴
  - 전역 공간에 변수를 무분별하게 선언하지 않음
  - 로직의 캡슐화
  - 재사용 용이
  - 임포트나 익스포트가 자유롭다.

자바스크립트의 표준 모듈화

- 대표적인 모듈화 표준
  - CommonJS
  - AMD (Asynchronous Module Definition)
- 공통 목표
  - 브라우저 뿐 아니라 서버사이드, 로컬 애플리케이션에 쓰일 수 있는 자바스크립트
  - 모듈별 자신만의 스코프 영역 정의
  - 모듈 정의(export) 방법 정의
  - 모듈 삽입 방법(import) 정의

부록 #1 RequireJS 간단소개

- AMD 표준을 따름
  - 비동기 및 늦은(Lazy-loading) 스크립트 로딩 지원
- CommonJS와 호환.
  - require와 exports 를 지원하여 CommonJS 스타일로도 작성가능.
- 브라우저에서도 쉽게 사용할 수 있다.
- define, require 으로 모듈 생성 및 의존성 정의
- 모듈별로 독립된 변수 스코프를 지원

부록 #2 예제코드

- 소스
  - https://github.com/javarouka/javarouka.github.com/tree/master/requirejs-sample
- 예제
  - http://javarouka.github.com/requirejs-sample

참고 링크

- 자바스크립트 표준을 위한 움직임
  - http://helloworld.naver.com/helloworld/12864
- Javascript Module Pattern - In depth
  - http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth
- Build a simple client-side MVC app with RequireJS
  - http://verekia.com/requirejs/build-simple-client-side-mvc-app-require-js
- 자바스크립트 모듈 패턴
  - http://blog.javarouka.me/2012/02/javascripts-pattern-2-module-pattern.html

QnA

감사합니다