공학 트렌드

자바스크립트(Javascript) UI Part 1
- 모듈 패턴

2012. 12. 10. (제36호)

목 차

Ⅰ. 자바스크립트 유효범위의 문제

Ⅱ. 접근 제어를 위한 모듈화

Ⅲ. 모듈의 재활용

Ⅳ. 결론

소프트웨어공학센터
경영지원TF팀

Ⅰ. 자바스크립트 유효범위의 문제

최근 HTML5나 웹앱 등 웹 기술로 만들어진 어플리케이션이 속속 등장하고 있다. 이런 웹 어플리케이션을 구현하는 핵심기술은 자바스크립트이나, 실제로 많은 개발자들은 자바스크립트의 특성을 잘 이해하지 못하여 생기는 문제가 많이 발생하고 있다.

자바스크립트의 특성 중 아쉬운 점은 유효범위가 다소 혼란스러운 점이다. 특히 자바나 C언어를 다루는 개발자가 자바스크립트를 접하게 되면 더욱 그렇게 느껴질 수 있다. 예를 들어 자바스크립트는 블록 유효범위가 없으며 함수 유효범위가 있을 뿐이다. 이런 차이를 잘 이해하지 못해서 생기는 문제는 실제로 매우 많다.

이런 특징은 결국 중요 함수나 변수가 공개된 영역(자바스크립트에서는 이를 scope이라고 표현)으로 노출 된다는 점이다. 자바스크립트는 public 이나 private과 같은 함수나 변수의 노출 범위를 명시적으로 표현 할 수 있는 키워드가 존재하지 않는다. 하지만 자바스크립트의 다양한 표현방법을 활용하여 이를 모듈화하면 어느 정도 보호 할 수가 있다.

본 문서에는 이러한 방법을 설명할 것이며, 여기에 나와 있는 방식은 대부분의 실제 웹서비스나 자바스크립트 라이브러리 등에서 많이 사용되고 있는 방법들이다.

본 문서를 통해서 '어떤 문제가 실제 발생하는 것인지?', '어떻게 해결 할 수 있는지?'를 알아본다.

Ⅱ. 접근 제어를 위한 모듈화

실제 자바스크립트로 어떻게 구현되는지를 알기 위해서, 실제 코드를 살펴볼 것이다. 예제 코드에서는 '현재 사과의 개수를 확인하는 것' 과 '현재 사과의 개수를 하나 증가' 하는 함수를 간단히 구현한다.

```
var nAppleNumber = 1;

function getApple(){
  console.log("nAppleNumber is : " + nAppleNumber);
}

function plusOneAppleNumber() {
  return nAppleNumber++;
}

getApple();
```

예제 1

예제 1 에서는 두 개의 함수를 getApple, plusOneAppleNumber 라는 이름으로 선언했다.

위 코드는 겉으로 보기에는 아무런 문제가 없으며, 실제로 그 동작도 정상이다.

그런데, 만약 동료 개발자가 우연히 똑같이 생긴 함수나 변수를 만든다면 어떻게 될까? 그런 일은 자바언어를 기반으로 개발 하는 경우에도 일어날 수도 있지만 대부분은 class단위로 나눠서 개발하게 되고 그런 경우 중복된 함수로 생기는 문제는 감춰지게 된다. 또는 개발단계에서 Java를 지원하는 훌륭한 IDE들이 대부분 이런 문제를 사전에 경고해주기도 한다. 자바스크립트에서는 이런 문제를 대부분의 개발도구가 설명해주지도 않고, 코드가 동작하는데 아무런 문제가 없어 보인다.

```
var nAppleNumber = 1;

function getApple(){
  console.log("nAppleNumber is : " + nAppleNumber);
}

function plusOneAppleNumber() {
  return nAppleNumber++;
}

nAppleNumber = 99;
getApple(); //'99'
```

예제 2

예제 2 에서 getApple()함수를 호출하게 되면, '1'이 아닌 '99'가 노출될 것이다. 중간에 nAppleNumber까 변경될 의도가 아니었다면 이런 결과는 원치 않은 상황이다.

이런 문제는 약속을 통해서(coding convention) 코드의 안전성을 확보 할 수 있다. 예를 들어 보호하고자 하는 변수의 접두어에 '_'를 붙이자고 서로 약속하는 것이다. (실제로 자바스크립트에서는 private 속성의 변수에는 '_' 붙여서 주로 이름을 짓는다.) 하지만 이런 약속은 보조적인 안전장치가 되어야지, 근본적인 조치 방법은 아니다.

코드 수준에서 이를 보호 할 수 있는 방법을 살펴보자. 기본 적인 원리는 간단하다. '자바스크립트의 scope 는 함수기반으로 되어 있다'라는 특성을 활용하는 것이다. 따라서 함수라는 울타리를 만들고 이를 통해서 접근 제어를 할 수가 있다. 모듈화패턴이라고 하는 것인 이렇게 함수를 통한 모듈을 만드는 패턴이라고 이해하면 된다. 이를 예제 3 코드를 통해 확인해보자.

```
function wrapFunction() {
  var nAppleNumber = 1;

  function getApple(){
    console.log("nAppleNumber is : " + nAppleNumber);
  }


  function plusOneAppleNumber() {
    return nAppleNumber++;
  }
 getApple();
}

wrapFunction();  //"nAppleNumber is : 1"
```

예제 3

종전의 코드를 wrapFunction라는 함수로 랩핑했다. 함수를 어떻게 또 다른 함수로 랩핑 할 수 있을까? 라고 의문을 제기 할 수 있다. 자바스크립는 함수형 언어[1]이며, 이러한 중첩된 함수의 형태를 가질 수 있다. 위 코드에서 getApple이라는 함수를

1) http://en.wikipedia.org/wiki/Functional_programming

마지막에 호출하도록 했다.

따라서 wrapFunction함수를 호출하면 getApple이라는 함수가 불려지고 현재 사과의 개수가 출력될 것이다. 원하는 대로 getApple 함수를 호출 할 수는 있지만, plusOneAppleNumber 함수는 외부에서 불려서 사용할 수가 없다.

즉 plusOneAppleNumber는 private한 함수가 된 것이다.

마찬가지로 변수로 선언된 nAppleNumber 도 접근이 제한되어 외부에서 선언한 변수와 충돌이 발생하지 않는다. 이것은 사실 wrapFunction이라는 함수의 지역변수가 된 것이다. (자바스크립트의 지역변수는 함수 안에 있더라도 'var' 로 선언해야만 지역변수로 인식한다)

지금까지 함수의 유효범위를 통해서 함수와 변수를 보호하는 방법을 코드를 통해 확인했다. 위의 코드는 몇 가지 불필요한 부분이 존재하며 이를 제거하고 조금 더 다듬을 수 있다.

먼저 wrapFunction함수는 실제 랩핑만 하는 기능으로 이름자체를 다음과 같이 제거 할 수가 있다.

```
(function() {
  var nAppleNumber = 1;

  function getApple(){
    console.log("nAppleNumber is : " + nAppleNumber);
  }

 function plusOneAppleNumber() {
    return nAppleNumber++;
  }
 getApple();
})();     //"nAppleNumber is : 1"
```

예제 4

이렇게 *(function() { ..... })()* 라고 표현한 부분은 2)즉시실행함수(immediately invoked function) 또는 Self-executing function이라고도 한다. 즉 함수를 선언하고 바로 실행하는 방법이다.

2) http://www.jspatterns.com/self-executing-functions/

예제 4가 바로 일반적인 **모듈화 패턴의 형태이며**, 보호하고 싶은 코드와 노출하고 싶은 코드를 구분할 수 있는 일반적인 방법이라고 할 수 있다.

이렇게 코드가 모듈화 된 형태가 어떻게 다른 곳에서 쉽게 재사용될 수 있는지 다음 장에서 확인하도록 하자.

Ⅲ. 모듈의 재활용

모듈화 패턴은 재활용될 수가 있다. 예를 들어 위에서 나온 즉시실행함수에서 어떤 결과를 반환할 수도 있을 것이다. 그 반환 값에 공개된 함수를 포함하면 되는 것이다. 코드를 살펴보자.

```
(function() {
  var nAppleNumber = 1;

  function getApple(){
    console.log("nAppleNumber is : " + nAppleNumber);
  }

 function plusOneAppleNumber() {
    return nAppleNumber++;
  }

  return {
    getApple : getApple,
    plusOneAppleNumber : plusOneAppleNumber
  }
})();
```

예제 5

예제 5를 보면 함수를 두 개의 함수를 포함하는 임의의 객체를 반환하고 있다. 따라서 예제 5의 결과 값은 객체(Object type)가 될 것이다.

두 개의 함수는 객체에 담겨진 채 외부로 노출된 것이지만, 실제로 두 개의 함수자체를 직접 변경할 수는 없으며 단지 외부에서 접근하여 이 함수를 호출 할 수는 있게 한 것이다.

만일 외부에서 이 두 개의 함수를 사용할 일이 없거나 외부로 접근을 막기

위해서는 리턴(return) 객체의 속성에 포함하지 않으면 된다.

이런 구조를 통해서 모듈화 된 정보를 재활용하는 단계에서 , 함수 자체가 변경되는 일이 발생할 수 있다. 다음의 코드를 보면 oAppleInfo라는 내부 객체와 이를 복사한 copyAppleInfo라는 객체 두 개를 선언한 상태이며, 이 두 개를 리턴했을 때의 결과가 어떻게 다른지 확인해보자.

```
var oAppleModule = (function() {
    var _nAppleNumber = 1;
    var oAppleInfo = {
                color : "deepblue",
                size :5
   }

   var oCopyAppleInfo = {
                color : oAppleInfo.color,
                 size : oAppleInfo.size
    }

   function getAppleColor(){
     console.log("nAppleColor is : " + oAppleInfo.color);
   }

   function plusOneAppleNumber() {
     return _nAppleNumber++;
   }

   return {
      getAppleColor : getAppleColor,
      plusOneAppleNumber : plusOneAppleNumber,
      oAppleInfo : oAppleInfo            //객체를 직접 반환
    //oAppleInfo : oCopyAppleInfo      //객체의 복사본을 반환
   }
})();

oAppleModule.oAppleInfo.color = "red";
oAppleModule.getAppleColor();  //“red” , 내부 객체가 변경되었음
```

예제 6

이 코드에서 내부 객체가 변경되지 않게 하기 위해서는 ‘getAppleType’ 함수의

반환 값으로 oAppleInfo 객체를 그대로 노출하지 말고, 그 객체의 복사본인 oCopyAppleInfoObj를 반환하도록 하면 된다. 이렇게 함으로써, 객체의 참조가 그대로 넘겨지는 것을 방지하게 되어 내부 객체의 속성이 외부에서 변경되는 것을 막을 수 있다.

위 예제 6의 코드가 모듈화 패턴을 조금 더 다음은 결과물과 같다.

이와 같이 내부/외부 노출할 대상을 반환 값에 적절히 포함해서 이를 객체 지향적으로 활용할 수 있다.

<예제 6의 소스코드는 여기서 확인 할 수 있다>[3)]

추가로 한 가지 비슷한 방법을 더 소개한다.

함수 생성자(constructor)를 활용해서도 private 속성을 만들 수 있는 방법[4)]이 있다.

```
function wrapFunction() {
    var nAppleNumber = 1;
    this.getApple = function(){
      console.log("nAppleNumber is : " + nAppleNumber);
     }

   function plusOneAppleNumber() {
     return nAppleNumber++;
   }
}

var oWrap = new wrapFucntion();
oWrap.getApple();
```

예제 7

기존 코드와 큰 차이는 'new' 연산자를 통해서 함수의 인스턴스를 만들고, 이때 만들어진 oWrap이라는 인스턴스 객체는 this에 연결이 되어서 this로 선언된 함수에 접근을 할 수 있게 된다. 그래서 this.getApple() 로 함수를 호출 할 수 있다. 이런 식으로 표현해두면 plusOneAppleNumber 함수의 경우는 접근이 되지 않음으로 private와 public 성격의 함수를 구분 할 수 있게 되는 것이다.

참고로 this.getApple과 같은 방식의 메소드를 정의하는 것보다 'prototype'이라는

3) http://jsbin.com/acaliz/1/edit
4) 더글라스 크랙포드가 정의한 방법이다. http://www.crockford.com/javascript/private.html)

속성을 활용해서 메소드를 추가하는 것이 더 효율적이다. 'prototype'은 자바스크립트의 중요하면서도 복잡한 특성 중 하나임으로 별도로 이를 이해할 필요는 있다

IV. 결론

지금까지 자바스크립트에서 private 한 정보를 어떻게 보호할 수 있는지, 모듈화를 통한 코드를 어떻게 외부로 공개 할 수 있는지 예제를 통해서 확인했다.

모듈화 패턴은 자바스크립트 기반의 웹어플리케이션에서도 유용하게 사용될 수 있으며, 특히 자주 재사용이 되는 UI 컴포넌트 단위의 기능이나 소규모 라이브러리를 개발 할 때도 유용하게 사용될 수 있다.

특히 다른 코드와 충돌이 걱정되는 단위 기능이나 자바스크립트 라이브러리를 개발해야 하는 상황에서는 이런 모듈화 된 코드는 적절한 안전장치라고 할 수 있겠다.

<참고 자료>

1. JavaScript Patterns. Stoyan stefanov, 2010
2. JavaScript: The Good Parts. Douglas Crockford. 2008
3. Pro JavaScript Techniques. John Resig. 2006
4. http://www.jspatterns.com
5. http://www.codeproject.com/Articles/247241/Javascript-Module-Pattern