JavaScript Quiz by @kangax 풀이

@kangax의 JavaScript Quiz - http://perfectionkills.com/javascript-quiz/

Author : **rhio.kim**
Twitter : @rhiokim
Blog : http://rhio.tistory.com
Email : rhio.kim@gmail.com

문제

```
( function( ) {
        return typeof arguments;
) )( );
```

설명

일반적으로 arguments는 코드 블록 내에서 사용할 때에는 함수에 전달된 인자를 배열처럼 접근하게 되지만 arguments는 Array가 아닌 length와 index(0, 1 … n-1, n)를 속성으로 갖는 개체입니다.

```
( function() {
        alert( arguments instanceof Array); //false
} )( );
```

사양

1. ECMA-262 5th
   A. 10.6 Arguments Object
   B. 11.1.6 The Grouping Operator

정답

"object"

문제

```
var f = function g( ) { return 23; };
typeof g( );
```

설명

Function 생성자, 함수선언(FunctionDeclaration), 함수표현식(FunctionExpression), 함수 호출에 대한 정확한 이해가 선행되어야 한다. 아래의 4가지에 대한 구별하기 위한 문제이기도 하다.

1. Function 생성자에 정의된 sum변수에 할당함수
   var sum = new Function("x", "y", "return x + y;");
2. sum으로 명명된 함수의 함수선언
   function sum(x, y) { return x + y; }
3. sum 변수에 할당된 익명함수 함수 표현식
   var sum = function(x, y) { return x + y; }
4. sum변수에 함수 표현식 plus이 할당되었다라고 명명
   var sum = function plus(x, y) { return x + y; }

위의 4가지는 거의 같은 역할을 하지만 아주 조금씩 다른점이 있다.

- 함수선언은 선언과 동시에 함수의 이름과 동일한 이름의 변수를 VariableEnvironment의 Environment Record에 생성한다. 따라서 함수 표현식으로 정의된 것과는 달리 함수 선언에 정의된 함수는 정의된 범위 내에서 이름을 사용할 수 있게 된다.
  (MDC 와 ECMA-262 인용)
  https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Functions#section_14

즉 아래의 코드는 함수의 선언으로 자동으로 g라는 변수가 생성되고 이 변수는 g 함수를 가리키게 된다. (ECMAS-262spec 59p – 10.5 Declaration Binding Instantiation)

```
function g( ) { ... }
```

하지만 문제에는 함수의 선언이 아닌 var 구문을 이용해 f 라는 변수를 선언하고 변수에 '함수표현식' 즉 함수 리터럴을 할당한 것으로 함수 선언이 없다. 그래서 g 함수를 가리키는 변수는 존재하지 않게 된다.

```
var f = function g( ) { return 23; }
```

그런데 g 함수를 호출을 시도하기 때문에 undefined 인 g를 함수로서 호출하려고 하기 때문에 ReferenceError: g is not defined 이 발생하게 된다.

증명

사양

1. MDC
   A. https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Functions
2. ECMA-262 5th
   A. 10.5 Declaration Binding Instantiation
   B. 11.2.5 Function Expressions
   C. 13. Function Defintion

정답

Error

문제

```
( function ( x ) {
        delete x;
        return x;
) ) ( 1 );
```

설명

이것은 delete 연산자(Operator)와 변수 구체화(Variable Instantiation)에 대한 이해를 묻는 문제이다.

JavaScript의 delete 연산자는 특정 속성이 주어질 때 그 속성이 있는 개체에서 속성을 삭제한다. 하지만 이것은 모든 속성을 삭제할 수 있는 것은 아니다. 대상이 되는 객체 자신이 가진 속성이 아닌 것은 삭제할 수 없다.

JavaScript의 모든 객체에는 prototype 체인에 의해서 외부 객체의 속성을 자신의 속성처럼 취하게 되지만 delete 연산 시에 이러한 속성을 발견하게 되더라도 제거되지 않는다.

```
function foo( ) { this.x = 21; }
foo.prototype.x = 12;
var o = new foo( );
alert( o.x );        //21
delete o.x;
alert( o.x );        //12
```

또한 개체의 속성은 ReadOnly와 DontDelete의 속성을 가질 수 있고 아래의 예시처럼 내장개체에 처음부터 존재하는 속성 중 DontDelete 속성들은 삭제되지 않는다. 하지만 내장 개체라 할 지라도 새롭게 지정하는 경우에는 DontDelete 속성이 아니기 때문에 삭제된다.

```
var arr = new Array(1,2,3);
delete arr.length;
alert(arr.length);              //3
arr.newProperty = 21;
alert(arr.newProperty);      //21
delete arr.newProperty;
alert(arr.newProperty;       //undefined
```

이 문제를 이해하기 위해서는 delete 연산자뿐만 아니라 변수(variable)의 특징도 이해하고 있어야 한다. var문을 사용하여 만들어지는 속성은 DontDelete 특성을 가진다. 즉 var 구문을 통해 선언된 변수는 삭제할 수 없고 함수 선언에 의해 생성되는 속성들도 같다. 함수 선언에 의해 생성되는 속성들은 함수가 실행될 때 생성되어지고 자동으로 소멸한다.

```
<script>
        var x = 21;
        delete x;
        alert( x );          // 21

        function foo( ) {
            var z = 12;
            delete z;
            alert( z );
        }
        foo( );              // 12
</script>
```

하지만 한가지 예외가 있는 것이 eval 함수를 사용하여 실행되는 코드에서 var 구문으로 변수를 생성하는 경우에는 DontDelete 특성이 없다. (만약 위의 코드를 firebug나 webinspector 에서 실행할 경우에는 DontDelete 습성이 없는 채로 생성됨)

결론적으로 인자로 넘어온 x 변수는 함수의 런타임에 x 라는 변수가 함수블록 내에 DontDelete 습성을 갖으며 생성되고 1의 값이 할당된다. 그래서 x는 삭제되지 않고 delete x; 구문은 false를 반환한다. (DontDelete 습성인 경우 false를 반환)

참고

1. MDC
   A. https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Operators/Special_Operators/delete_Operator
2. Blog
   A. http://perfectionkills.com/understanding-delete/

사양

1. ECMA-262 5th
   A. 11.4.1 The delete Operator
   B. 11.1.6 The Grouping Operator
2. ECMA-262 3rd
   A. 10.1.3 Variable Instantiation
   B. 10.2.2 Eval Code

정답

1

문제

```
var y = 1, x = y = typeof x;
x;
```

설명

이 것은 Comma(,) 연산자와 Assignment(=) 연산자의 조합에 있어서 연산자 우선순위에 대한 이해를 묻는 문제이다.

MDC의 내용을 보면 다음과 같다.

> Assignment(=) 연산자는 오른쪽에서 왼쪽 (*RightHandSideExpression*)
> Comma(,) 연산자는 왼쪽에서 오른쪽 (*LeftHandSideExpression*)

Comma 연산자에 의해서 y = 1 가 코드가 선행 평가가 이루어지고 나서 x = y = typeof x; 가 평가되어진다. 그 다음 Assignment 연산자에 의해서 y = typeof x; 가 평가되고 x = y 가 순차적(right to left)으로 평가된다. 이때 y = typeof x; 는 x의 값은 정의되지 않았으므로 "undefined" 이다.

결론적으로 (x = (y = "undefined")) 처럼 되며 x, y는 "undefined"가 할당된다.

참고

1. MDC
   A. https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Operators/Operator_Precedence

사양

1. ECMA-262 5th
   A. 11.1.6 Left-Hand-Side Expressions
   B. 11.13.1 Simple Assignment
   C. 11.13 Comma Operator (,)

정답

"undefined"

문제

```
( function f( f ) {
        return typeof f( );
} ) ( function ( ) { return 1; } ) );
```

설명

이 것은 코드를 상당히 난해하게 해놓았지만 의외로 간단한 문제이다.
함수 이름과 지역변수 이름이 같은 경우 어느 것이 우선하는 지에 대한 이해를 묻고 있다. 변수의 우선순위는 기본적으로 지역변수가 우선한다.

```
x = 21;
function foo( x ) {
        alert( x );
}
alert( x );          // 21
foo( 22 );           // 22
```

문제에서 제시한 코드는 외부 f 함수 호출과 함께 1을 반환하는 함수 리터럴이 인자로 넘어가고 이 인자는 외부 f 함수 블록 내에서 인자 f가 우선시 되어 typeof f( )는 함수 리터럴을 호출하고 1을 반환 받는다.

결국 return typeof 1; 이 된다.

사양

1. ECMA-262 5th
   A. 11.1.6 The Grouping Operator

정답

"number"

문제

```
var foo = {
        bar: function( ) { return this.baz; },
        baz: 1
};
( function( ) {
        return typeof arguments[0]();
} ) ( foo.bar );
```

설명

이 것은 처음 JavaScript 접할 때 난해한 것 중에 하나인 this에 대한 이해를 묻는 문제이다. this 키워드는 일반적으로 다음의 세가지 형태일 때 사용되어 진다.

1. 인스턴스화 된 객체 내에서 메소드 호출(인스턴스화 된 객체가 암묵적으로 this 에 바인딩)
2. Function.call(scope, …) 처럼 call 메소드를 통한 명시적 scope 지정
3. Function.apply(scope, …) 처럼 apply 메소드를 통한 명시적 scope 지정

위의 foo 객체의 경우에도 Object 리터럴도 변수에 할당될 때 new Object( ) 동일하게 평가된다. 그렇기 때문에 foo 는 인스턴스화 된 객체로 foo 객체가 this로 바인딩 되어 bar 메소드 호출 시 foo.baz 에 할당된 1값을 반환하게 된다.

익명함수로 foo.bar 의 함수 리터럴을 익명함수의 첫번째 인자로 넘겨 arguments[0] 에는 foo.bar 의 함수 리터럴이 할당되어지고 arguments[0]( ) 는 foo.bar 함수를 실행하게 된다.

단 여기에서 주의해 할 것은 함수 리터럴의 scope(유효범위 즉 this)의 변화이다. foo.bar( )를 호출하는 것과 arguments[0]에 담긴 함수 리터럴 즉 function() { return this.baz; } 를 호출하는 arguments[0]( ) 는 scope가 arguments로 변경됨을 알 수 있다.

결론적으로 arguments[0]( ) 는 arguments.function() { return this.baz; } 로 평가되어지고 arguments 에는 baz 속성은 존재하지 않아 할당된 값이 존재하지 않으므로 “undefined”가 반환된다.

사양

1. ECMA-262 5th
   A. 11.1.1 The this Keyword
   B. 11.1.5 Object Initialiser

정답

"undefined"

문제

```
var foo = {
        bar: function( ) { return this.baz; }
        baz: 1
}
typeof ( f = foo.bar )( );
```

설명

이것은 6번의 문제를 정확히 이해하고 있다면 쉽게 해결할 수 있는 문제이다.

먼저 f = foo.bar 를 보면 f 에 foo.bar 의 함수 리터럴이 할당된다. 그러면 (f)( ); 로 축약된다. 이것은 다시 풀어보면 (function( ) { return this.baz; })( ) 와도 같고 이 함수 리터럴이 수행되는 scope는 즉 this는 window가 된다.
window 객체에도 baz가 존재하지 않기 때문에 6번 문제와 동일하게 "undefined"가 된다.

사양

6문항과 동일

정답

"undefined"

문제

```
var f = ( function f( ) { return "1"; }, function g( ) { return 2; } )( );
typeof f;
```

설명

4번 문항에서도 언급했지만 Comma 연산자에 의해서 왼쪽요소에서 우측요소로 코드 평가가 진행된다.

```
var x = ( 1, 2, 3 );
alert( x );             // 3;
var y = ( alert( 4 ), 5, function z( ) { return 6; } )( );    // 4
alert( y );             // 5
```

위의 예시로 알 수 있듯이 Comma 에 의해서 코드 평가가 좌에서 우측으로 진행되면서 alert(4) 가 수행되는 것을 알 수 있다. 문제에 제시된 코드 역시 g 함수를 취하게 되면서 2를 반환하게 된다.

사양

4번 문항과 동일

정답

"number"

문제

```
var x = 1;
if ( function f( ) { } ) {
        x += typeof f;
}
x;
```

설명

이 것은 2번 문항의 '함수선언(FunctionDeclaration)' 과 '함수표현식(FunctionExpression)'에 대한 문항과 같다. 조건문에 함수표현식이 들어가 있기 때문에 true가 된다. true가 되는 이유는 함수표현식의 경우에도 클로저(closure)를 반환하게 된다. (ECMA-262 5th 98p 참조)

그러므로 if(closure) 는 만족하게 된다. 다만 함수 표현식은 함수의 선언과는 다르게 자동으로 동일한 이름의 변수가 생성되지 않는다. 그래서 x 는 숫자형 1 과 typeof f 의 문자열 "undefined" 의 += 연산자에 의해 x 값은 "1undefined"

사양

2번 문항과 유사

정답

"1undefined"

문제

```
var x = [typeof x, typeof y][1];
typeof typeof x;
```

설명

이 것은 이전 문제에 이어서 혼동을 유발하기 위해서 ( ) 에서 [ ]로 변경하였고 Comma 연산자처럼 느끼도록 했다. (개인적인 느낌일 수도 있음 -.-a) 이것은 Array 리터럴로 x 값에는 배열의 1 위치에 있는 typeof y 가 할당되게 된다. y는 아무런 값이 할당되지 않았기 때문에 결국 x 는 "undefined" 가 된다.

관건은 typeof 인데 typeof 의 경우 오른쪽에서 왼쪽 코드(right to left) 평가를 수행한다. type of value 즉 "값 의 형태" 라고 해석하듯이 생각하면 쉽다.

결론은 typeof typeof x; 는 x는 문자열의 "undefined" 가 할당되어 있음으로 다음과 같이 풀어 쓸 수 있다.

```
typeof typeof "undefined";
//typeof "undefined" == "string"
typeof "string";
//typeof "string" == "string"
```

참고

1. MDC
   https://developer.mozilla.org/ja/Core_JavaScript_1.5_Reference/Operators/Operator_Precedence#.e7.b5.90.e5.90.88.e6.80.a7

사양

1. ECMA-262 5th
   A. 11.1.4 Array Initialiser
   B. 11.4.3 The typeof Operator

정답

"string"

문제

```
( function( foo ) {
        return typeof foo.bar;
} )( { foo : { bar : 1 } } );
```

설명

이 것은 Object 리터럴에 접근을 묻는 문제로 함수 표현식을 호출할 때 넘기는 Object 리터럴 { foo : { bar : 1 } } 는 foo 인자값으로 지정되었기 때문에 foo.foo.bar 로 접근해야 한다.

문제에서는 foo.bar 로 접근하였기 때문에 "undefined"가 반환된다.

사양

1. ECMA-262 5th
   A. 11.2.1 Property Accessors

정답

"undefined"

문제

```
( function f ( ) {
        function f( ) { return 1; }
        return f( );
        function f( ) { return 2; }
} )( );
```

설명

이 것은 JavaScript 함수선언(FunctionDeclaration)에 오류가 아닌 함정이 숨어있다.

> ***TIP 컴파일 타임과 런타임***
>
> 인터프리터[http://ko.wikipedia.org/wiki/인터프리터]에 의해서 해석되는 언어의 특징은 소스코드를 직접 실행하는 것인데 이 과정에서 코드의 해독단계와 실행단계로 나뉜다. 이것은 구문들의 표현식 검증, 소스 코드를 효율적인 중간 코드로 변환하거나 일부 시스템에서 이미 정의된 저장 코드를 수행하기 위해서이다.
>
> 그렇기 때문에 중간에 코드상의 오류가 있는 경우 선행 코드조차 실행되지 않는다. 예를 들어 아래의 코드처럼 function 구문 자체(즉 함수표현식)가 잘못되었을 때 alert('12') 조차도 수행되지 않는다.
>
> ```
> <script>
>         var x = 12;
>         var y = 21;
>         alert( x );
>
>         function ( ) {
>                 alert( y );
> </script>
> ```
>
> 즉 코드를 분석단계를 일컬어 컴파일 타임이라 하고 코드 분석이 완료되고 실제 코드를 실행하는 단계를 런타임 이라고 한다. 이것은 또한 ECMA-262 사양에 정의된 구문 수행을 위한 환경을 미리 만드는 과정이라고 할 수도 있다.
>
> 더불어 이런 특징 때문에 브라우저에서 JIT(Just In Time Compilation)[http://www.terms.co.kr/JITcompiler.htm]을 통해 JavaScript의 속도 향상을 도모하고 있다.

함수 표현식과는 달리 함수 선언에서는 컴파일 타임에 함수명과 같은 이름으로 변수가 생성되기 때문에 다음과 같이 foo 함수가 호출된다.

```
<script>
        foo( );
        function foo( ) { alert( 21 ); }
</script>
```

하지만 다음의 코드는 에러가 발생한다. 왜냐하면 함수 표현식은 호출할 때 비로소 함수가 정의되기 때문이다.

```
<script>
        foo( );
        var foo = function( ) { alert( 21 ); }
</script>
```

여기서 알 수 있는 특징은 함수선언과 함수 표현식의 할당은 함수의 생성 시기가 다르다는 것이다. 즉 위의 코드에서는 컴파일 타임에 f 함수가 호출되고 f 함수 내에서 반환값이 다른 f 충첩함수가 f 함수내의 지역함수로 선언된다. 하지만 동일한 이름(Identifier)이기 때문에 2를 반환하는 나중에 선언된 f 중첩함수가 덮어 씌여지게 되어 return f( )에서 f는 function f( ) { return 2; }가 할당되어져 있다.

참고

1. The Definitive Guid, 5th Edition
    A. 6.14 function
        i. function 구문은 프로그램의 정적 구조를 정의하는 것뿐이다.
        ii. JavaScript 코드가 구문 분석되고 컴파일 때 함수가 정의된다.

사양

1. ECMA-262 5th
    A. 11.1.6 The Grouping Operator
    B. 13. Function Definition

정답

2

문제

```
function f( ) { return f; }
new f( ) instanceof f;
```

설명

이 것은 new 연산자와 instanceof 연산자에 대해 이해를 묻는 문제입니다.

new 연산자는 개체를 생성하고 그것을 초기화하기 위해서 생성자(constructor) 함수를 호출한다.

```
function Identify( ) { ... }
new Identify( arguments );
```

이 코드는 다음과 같은 과정을 통해 비로소 인스턴스가 생성된다.

1. 아무런 속성이 정의되지 않은 새로운 객체를 생성한다.
2. 1에서 만든 객체 Prototype 내부 속성(__proto__ 속성)에 Identify.prototype 값을 설정한다. 만약 여기서 Identify.prototype 값에 지정할 객체가 없다면 Object.prototype값이 설정된다.
3. Identify를 호출한다. 이때 this 값은 1에서 만든 .객체로 할당되고 new 연산자와 함께 전달된 arguments는 그대로 설정된다.
4. 3의 반환값이 객체이면 그것을 반환하고 그렇지 않으면 1에서 만든 개체를 반환한다.

위의 과정을 이해하고 코드를 살펴보면 new f( )의 코드 평가의 결과는 f 함수가 호출되면서 자기 자신 function f( ) { return f; } 를 반환한다.

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

결과적으로 new f( )는 f 함수가 되고 new f( ) instanceof f 는 f instanceof f 가 되며 이것은 false 이다. 자기 자신에 프로토타입 체인에 자신은 존재하지 않는다.

만약 new f( ) instanceof Function 혹은 new f( ) instanceof Object 의 경우에는 true이다.

참고

1. JavaScript Definitive Guide 5th
   A. 5.10.3 The Object-Creation Operator (new)
2. MDC
   A. https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Operators/Special_Operators/new_Operator (new Operator)

사양

1. ECMA-262
    A. 11.2.2 The new Operator
    B. 13.2.2 [[Construct]]

정답
false

문제

```
with( function( x, undefined ) { } ) length;
```

설명

이것은 with 구문과 함수 리터럴에 대한 이해를 필요하는 문제이다.

<script> ... </script> 내에서 수행되는 변수, 함수등의 선언된 것들은 암묵적으로 구문 환경을 window 개체를 갖게 된다. 이 처럼 with 구문은 코드블록 내에서 암묵적인 구문 환경 개체를 지정하기 위한 것이다.

```
var foo = { bar : 12 };
with( foo ) {
        alert( bar );       // 12
        alert( zoo );       // ReferenceError: zoo is not defined
}
```

위의 간단한 예시처럼 with구문에 지정한 foo 개체는 코드블록 내에서 명시적으로 접근하는 것이 아닌 암묵적인 구문 환경이 생성되어 진다.

즉 위의 문제는 with 구문에 주어진 함수 리터럴의 length property를 접근하는 문제인데 function literal은 Function 클래스로 생성되어지는 것이기 때문에 Function의 속성과 Function.prototype 개체의 속성에도 접근할 수 있다.

참고

1. MDC
   A. https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Function#Properties_2

사양

1. ECMA-262 5th
   A. 12.10 The with Statement
   B. 15.3.3 Properties of the Function Constructor

정답

2

참고자료

1. ECMAScript Language Specification 5th edition & 3rd edition
   http://www.ecma-international.org/news/PressReleases/PR_Ecma%20approves%20new%20Patent%20Policy%20Document.htm

   http://www.ecma-international.org/publications/standards/ECMA-262.HTM

2. JavaScript The Definitive Guide, 5th edition by David Flanagan - http://www.amazon.com/JavaScript-Definitive-Guide-David-Flanagan/dp/0596101996/ref=sr_1_1?ie=UTF8&qid=1267784311&sr=1-1-spell

3. Mozila Developer Center - https://developer.mozilla.org
4. Kangax Blog – http://perfectionkills.com/

함께 읽어보기

1. JavaScript Quiz and Explanations of its Answers - http://kourge.net/node/130
2. Kangax 의 JavaScript Quiz 내용 파악하기 - http://blog.outsider.ne.kr/430