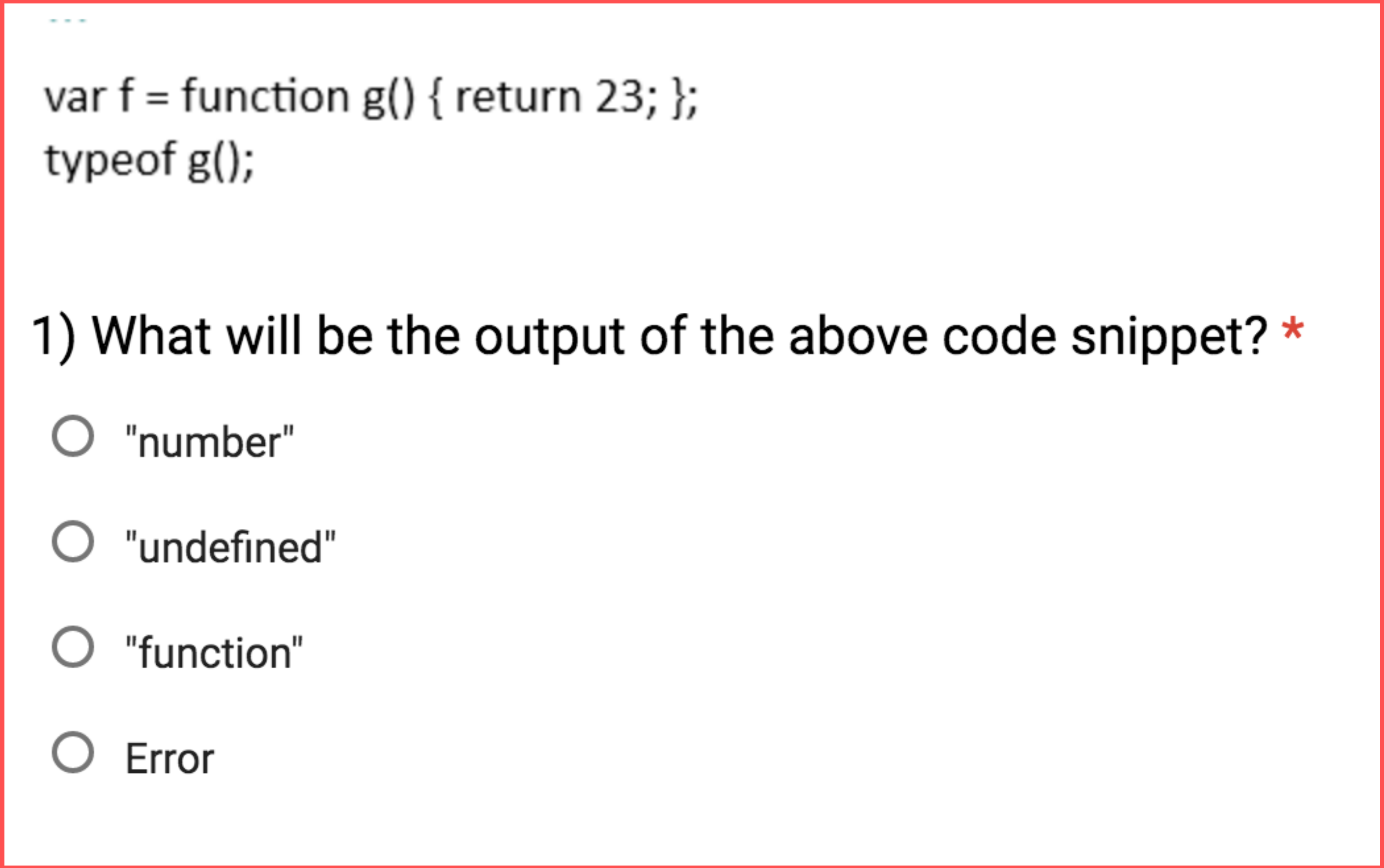
JAVASCRIPT QUIRKS

QUIZ #3

#1

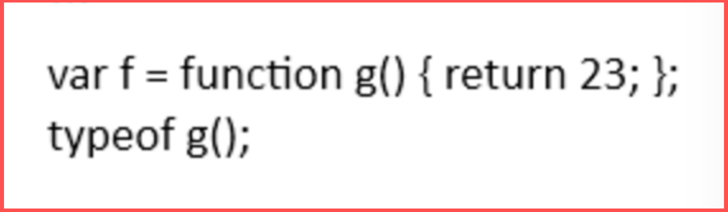


```
var f = function g() { return 23; };
typeof g();
```

1) What will be the output of the above code snippet? *

- ○ "number"
- ○ "undefined"
- ○ "function"
- ○ Error

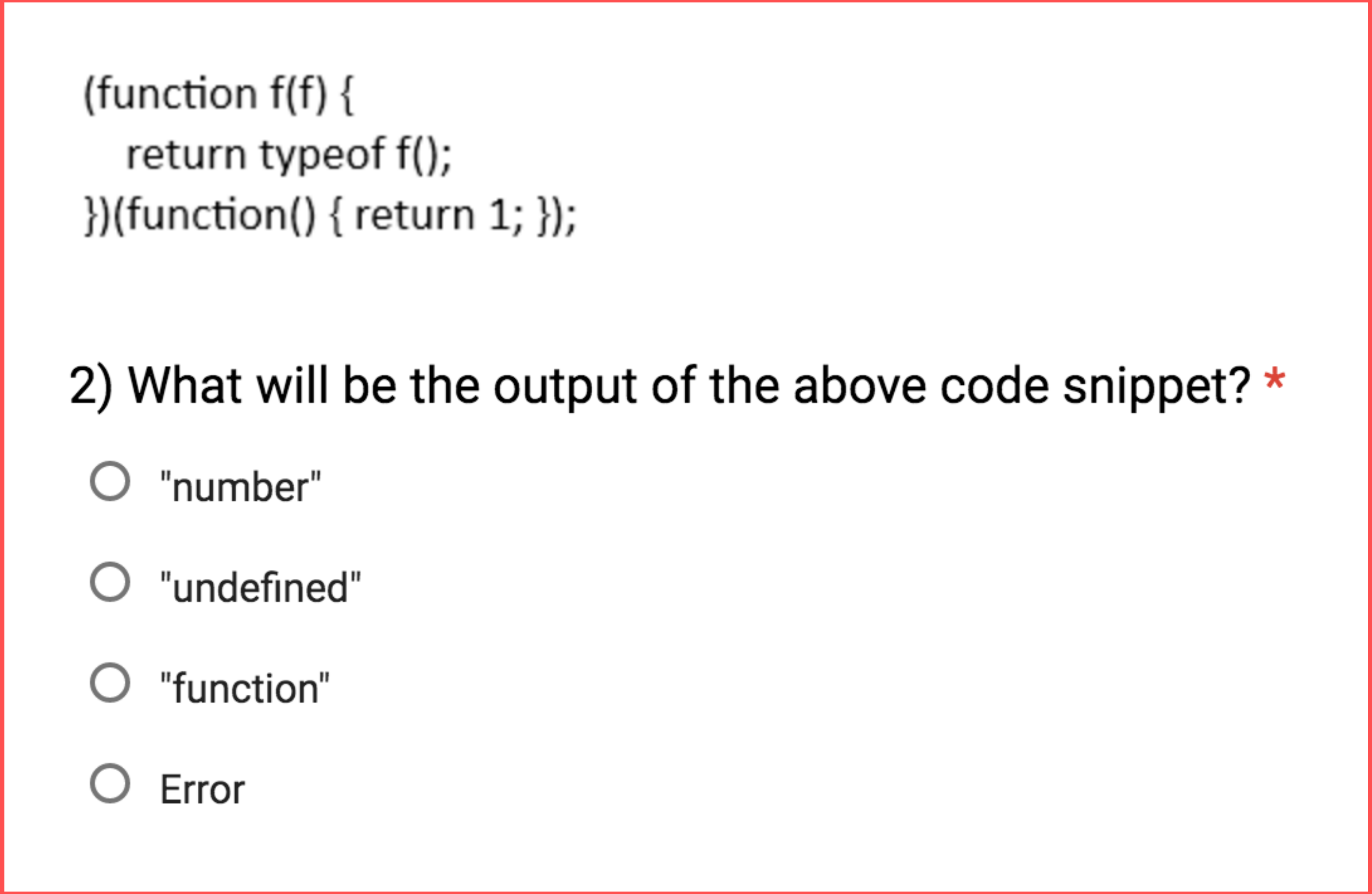
REFERENCE ERROR: G IS NOT DEFINED



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

- function g(){ return 23; } is a function expression (a named one, in fact), not a function declaration.
- The function is actually bound to the variable f, not g

#2



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

2) What will be the output of the above code snippet? *

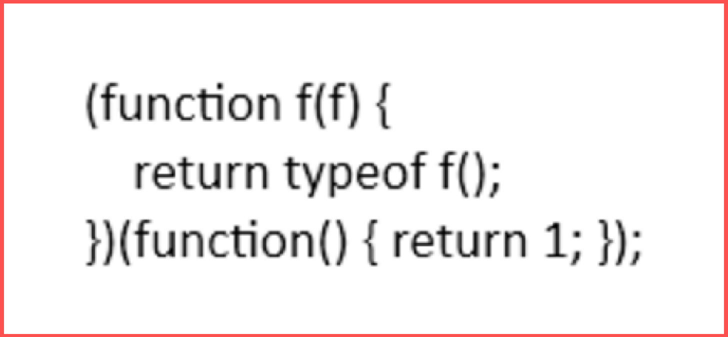
○ "number"

○ "undefined"

○ "function"

○ Error

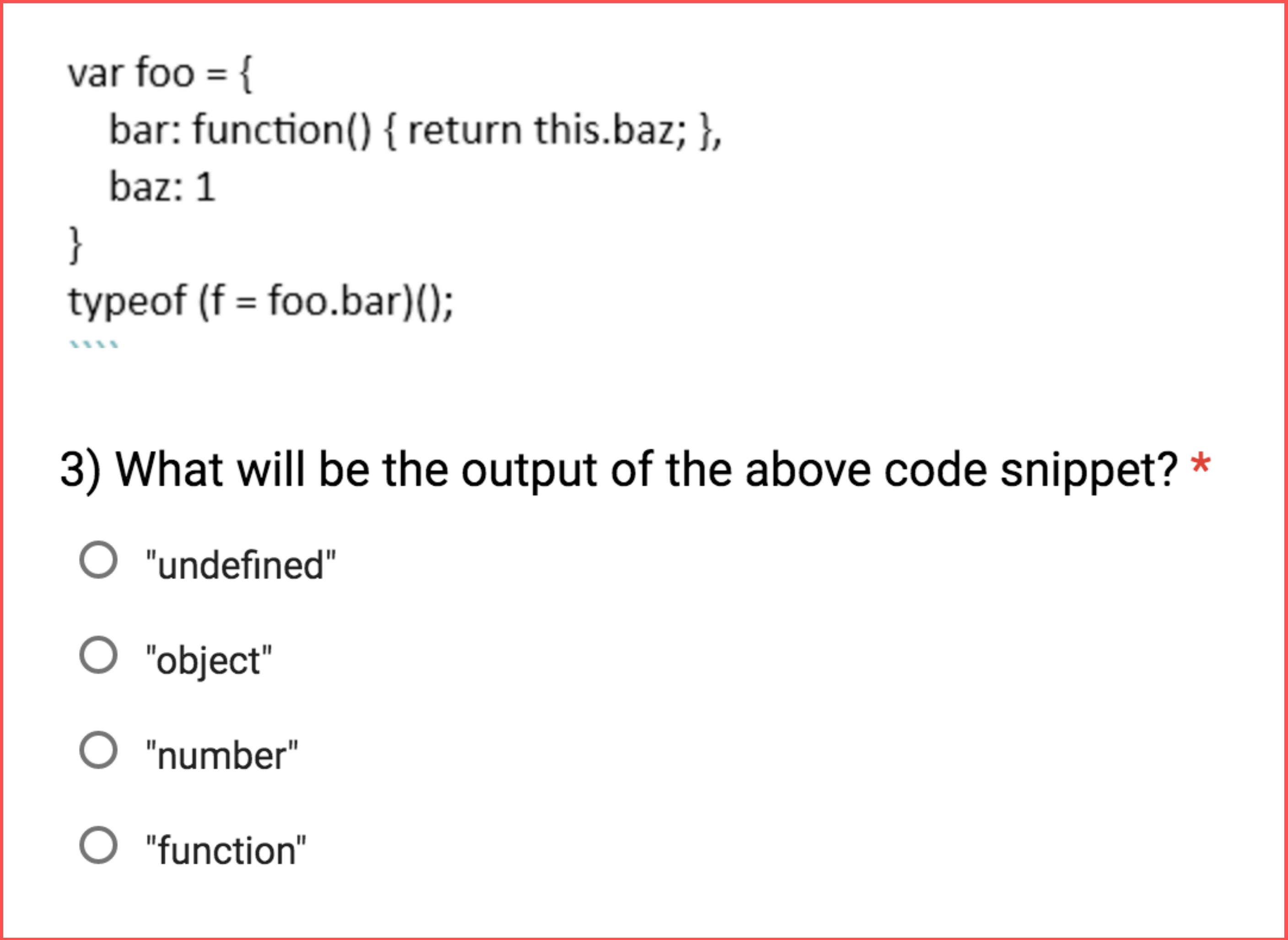
'NUMBER’



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

- The variable f inside the function refers to the argument
- Output of invoking the argument is 1
- typeof 1 === "number"

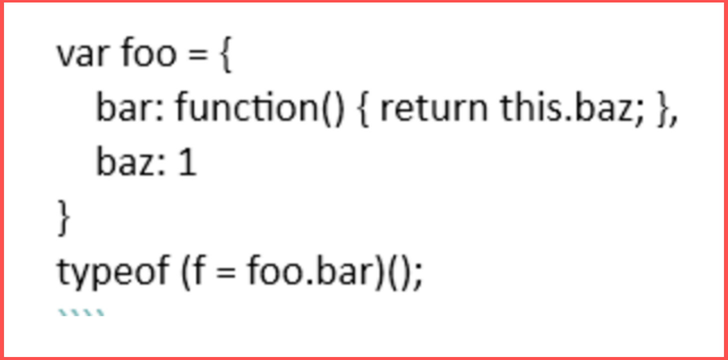
#3



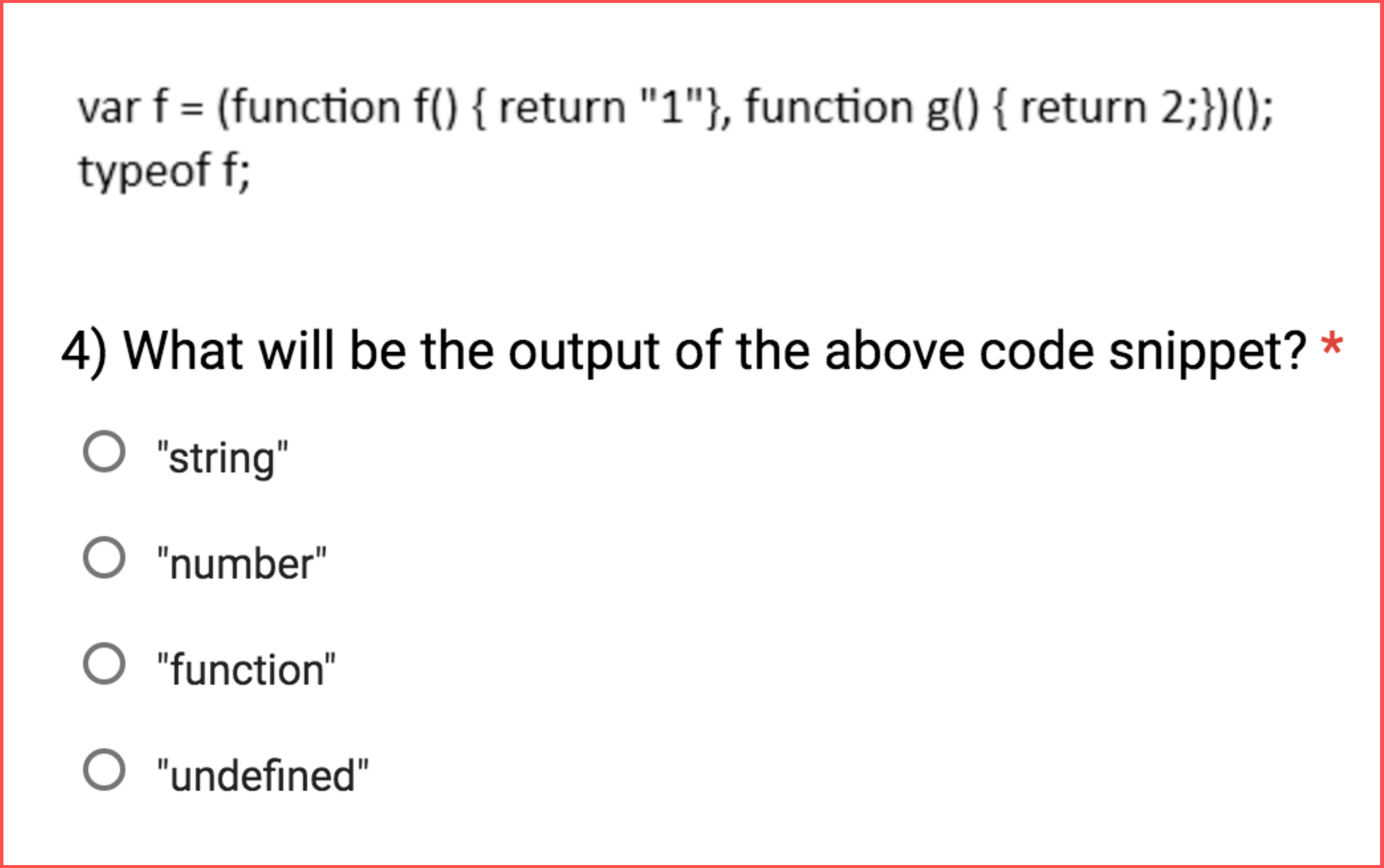
```
var foo = {
   bar: function() { return this.baz; },
   baz: 1
}
typeof (f = foo.bar)();
```

````

3) What will be the output of the above code snippet? *

- ○ "undefined"
- ○ "object"
- ○ "number"
- ○ "function"
````

'UNDEFINED’


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

- foo.bar is assigned to a global variable f
- f evaluates to function definition, which is immediately invoked
- **this** is no longer bound to foo but to the global object
- this.baz is undefined
```

#4



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

4) What will be the output of the above code snippet? *

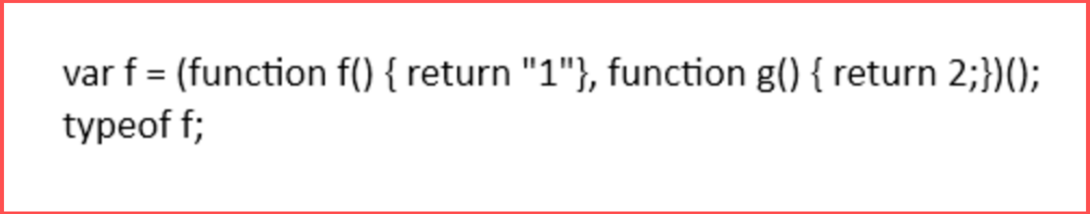
○ "string"

○ "number"

○ "function"

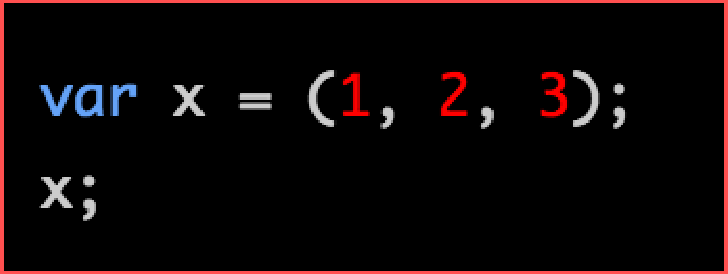
○ "undefined"

'NUMBER'



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

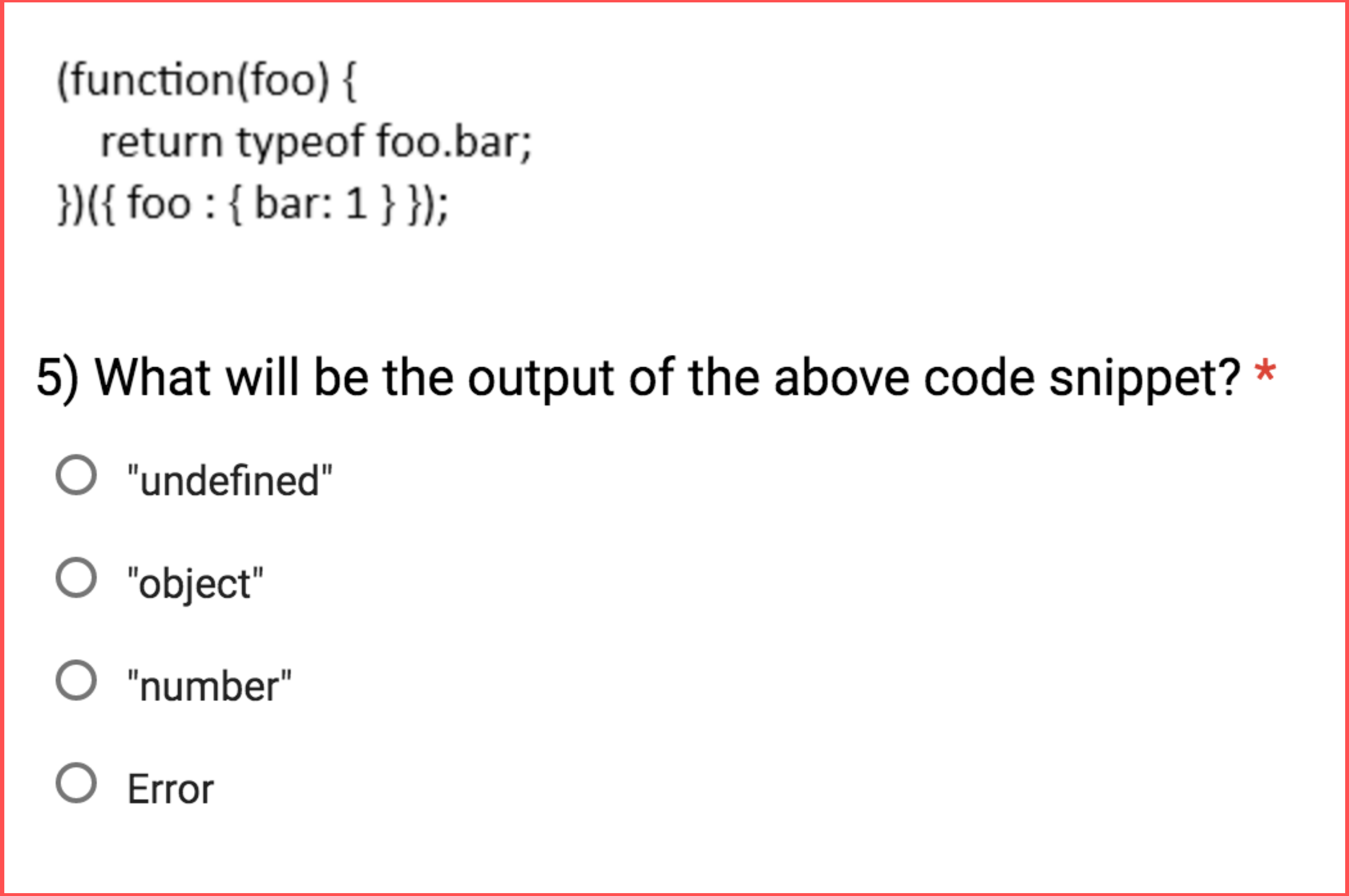
- When you have a series of expressions grouped together and separated by commas, they are evaluated from left to right, but only the last expression's result is preserved.



```
var x = (1, 2, 3);
x;
```

- x evaluates to 3

#5

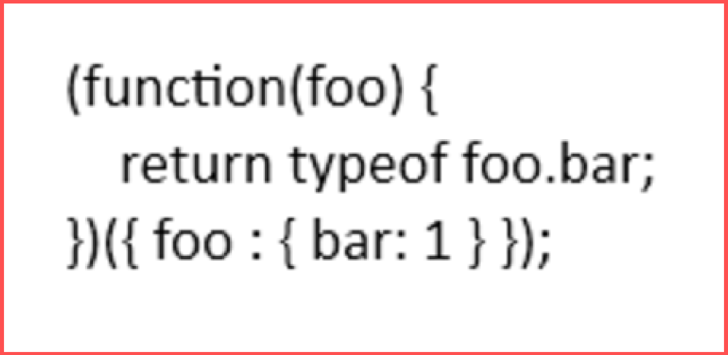


```
(function(foo) {
   return typeof foo.bar;
})({ foo : { bar: 1 } });
```

5) What will be the output of the above code snippet? *

- "undefined"
- "object"
- "number"
- Error

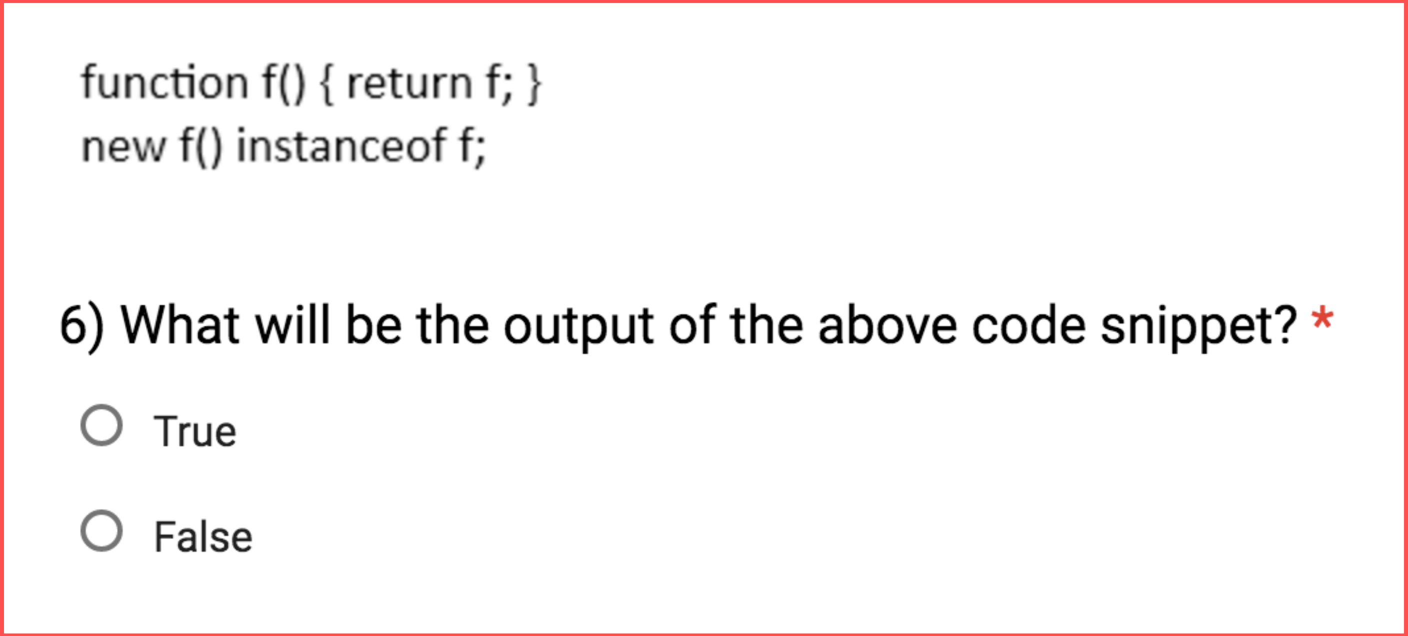
'UNDEFINED'



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

- The *argument* foo equals the Object {foo: {bar:1}}
- foo.bar is not defined (but foo.foo is)

#6



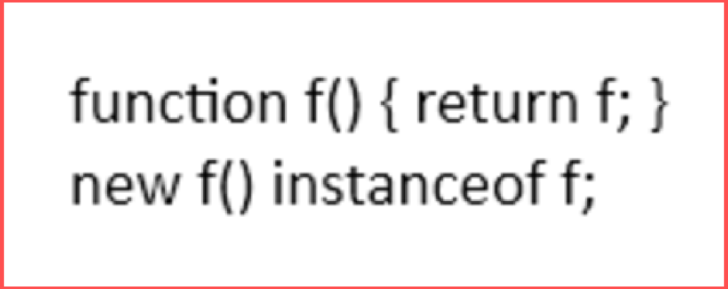
```
function f() { return f; }
new f() instanceof f;
```

6) What will be the output of the above code snippet? *

○ True

○ False

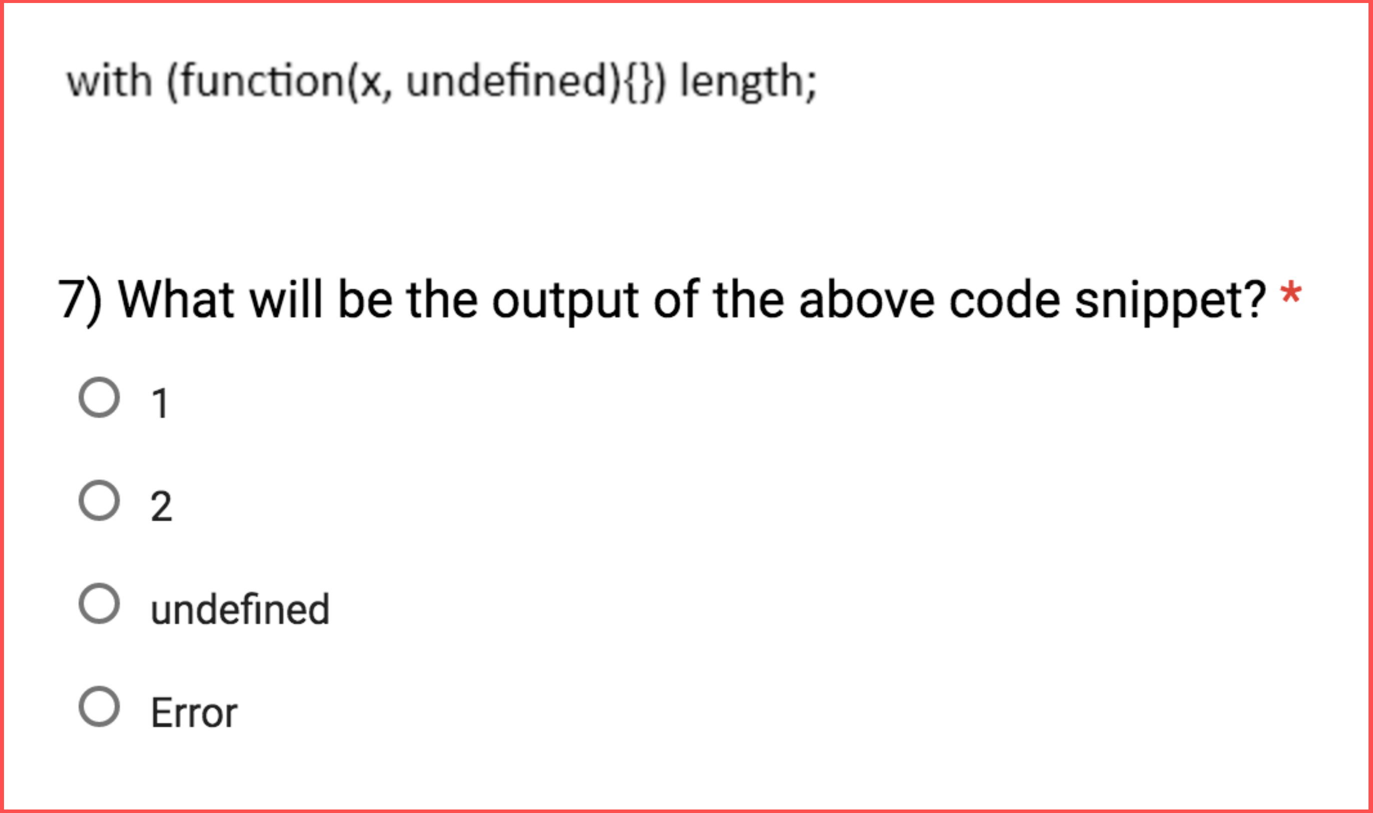
FALSE



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

- When the constructor returns an object, the new operator will yield the returned object
- new f() === f
- We know that f itself is a function, so it is an instance of the Function object
- f instanceof Function === true, but f instanceof f === false

#7



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

7) What will be the output of the above code snippet? *

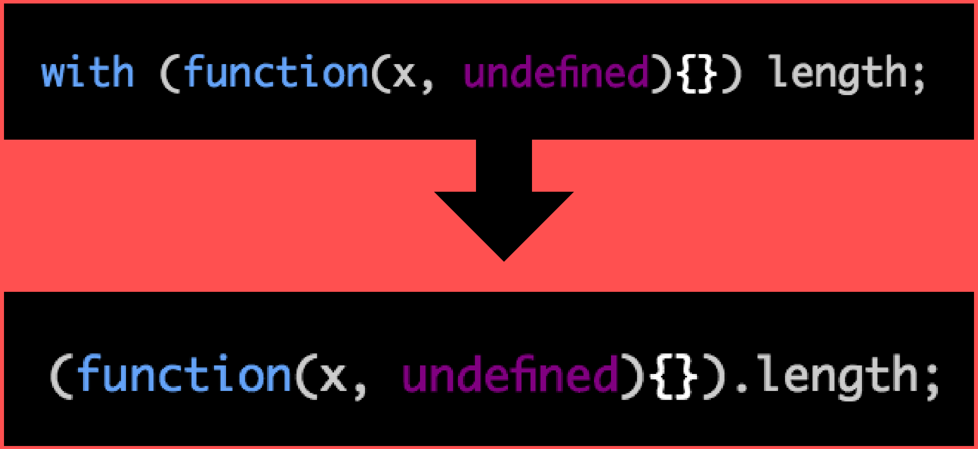
○ 1

○ 2

○ undefined

○ Error

ANSWER IS 2

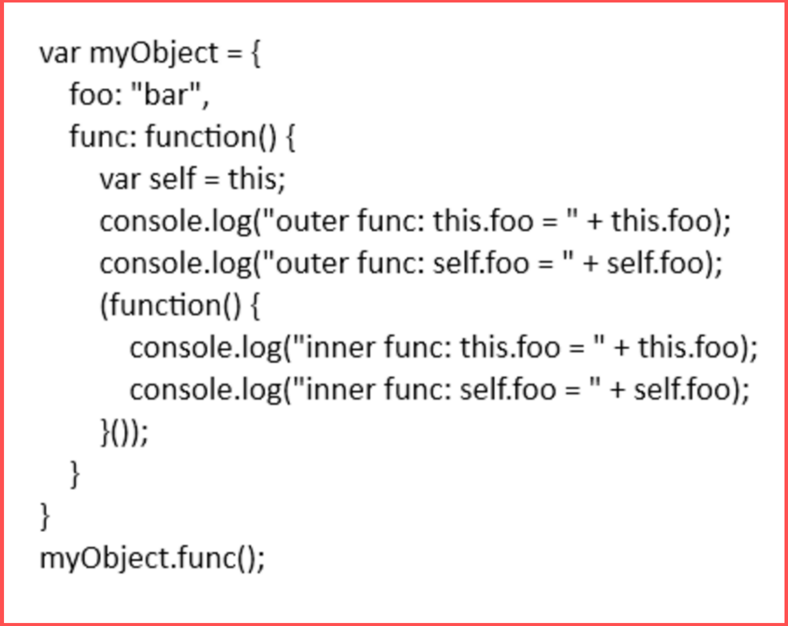


```
with (function(x, undefined){}) length;
```

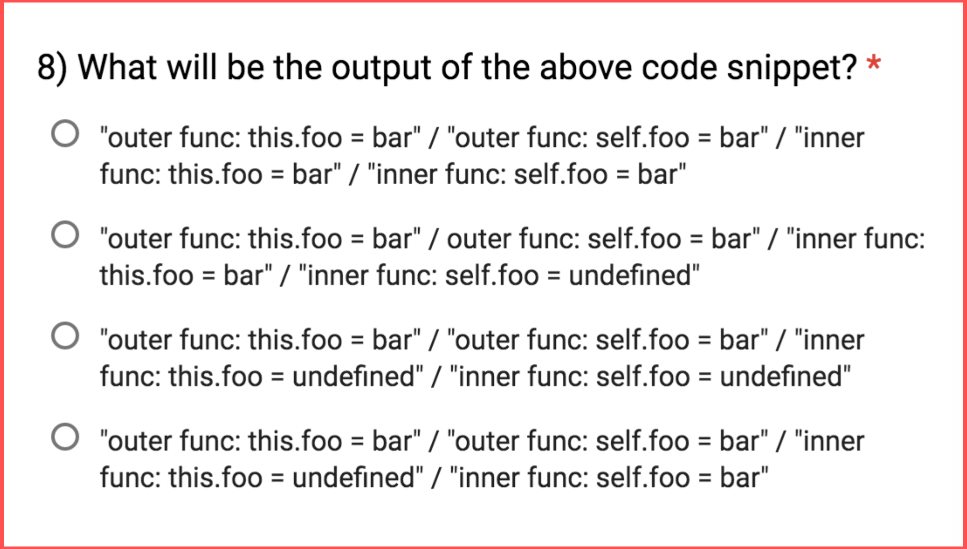
```
(function(x, undefined){}).length;
```

- The with operator allows us to use an arbitrary object as the scope (deprecated in modern JavaScript)
- Each function has a length property that indicates its arity, i.e. the number of arguments it takes.

#8

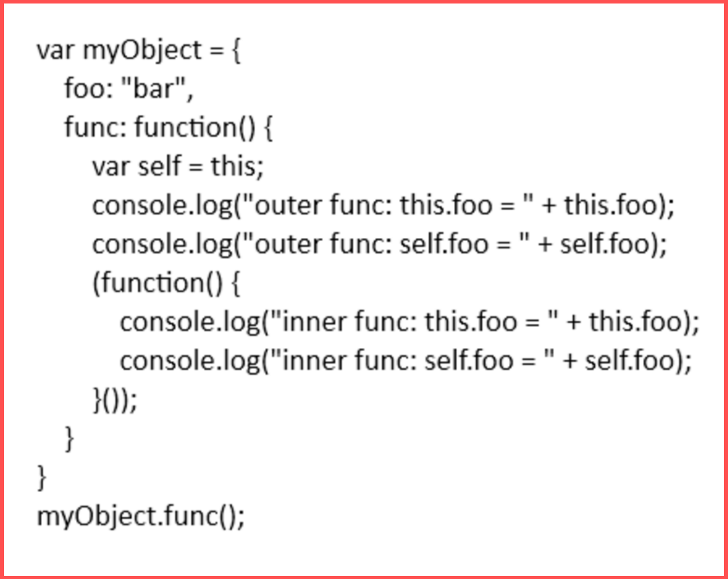


```
var myObject = {
   foo: "bar",
   func: function() {
      var self = this;
      console.log("outer func: this.foo = " + this.foo);
      console.log("outer func: self.foo = " + self.foo);
      (function() {
         console.log("inner func: this.foo = " + this.foo);
         console.log("inner func: self.foo = " + self.foo);
      }());
   }
}
myObject.func();
```



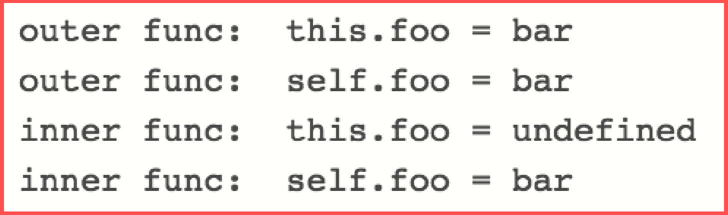
8) What will be the output of the above code snippet? *

- ○ "outer func: this.foo = bar" / "outer func: self.foo = bar" / "inner func: this.foo = bar" / "inner func: self.foo = bar"
- ○ "outer func: this.foo = bar" / outer func: self.foo = bar" / "inner func: this.foo = bar" / "inner func: self.foo = undefined"
- ○ "outer func: this.foo = bar" / "outer func: self.foo = bar" / "inner func: this.foo = undefined" / "inner func: self.foo = undefined"
- ○ "outer func: this.foo = bar" / "outer func: self.foo = bar" / "inner func: this.foo = undefined" / "inner func: self.foo = bar"



```
var myObject = {
    foo: "bar",
    func: function() {
        var self = this;
        console.log("outer func: this.foo = " + this.foo);
        console.log("outer func: self.foo = " + self.foo);
        (function() {
            console.log("inner func: this.foo = " + this.foo);
            console.log("inner func: self.foo = " + self.foo);
        }());
    }
}
myObject.func();
```

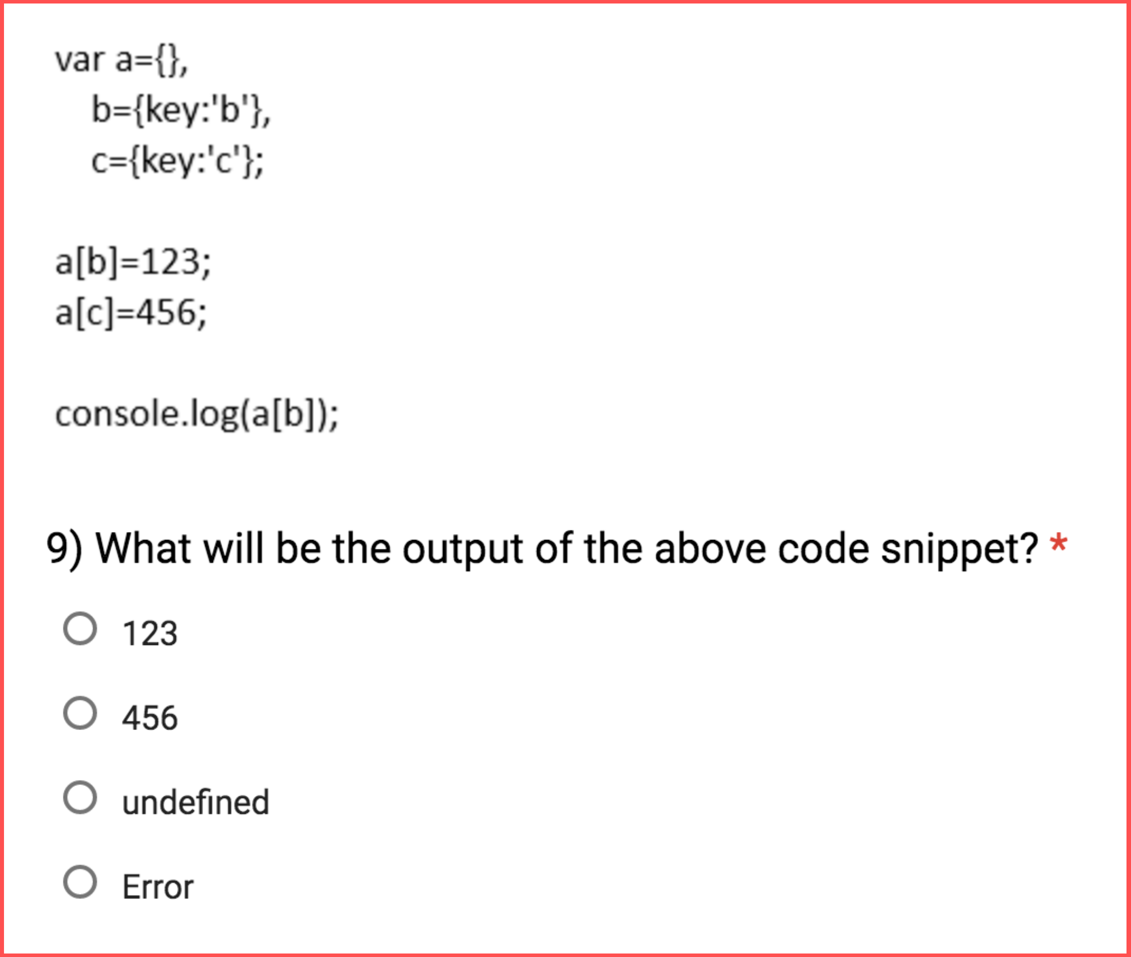
ANSWER



```
outer func:  this.foo = bar
outer func:  self.foo = bar
inner func:  this.foo = undefined
inner func:  self.foo = bar
```

- In the outer function, both 'this' and 'self' refer to myObject and therefore both can properly reference and access foo.
- In the inner function, 'this' no longer refers to myObject
- The reference to the variable 'self' remains in scope (closure!) and is accessible there

#9

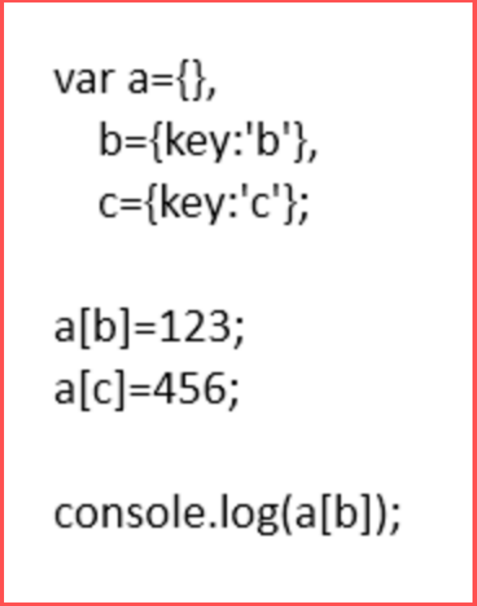


```
var a={},
   b={key:'b'},
   c={key:'c'};

a[b]=123;
a[c]=456;

console.log(a[b]);
```

9) What will be the output of the above code snippet? *

- ○ 123
- ○ 456
- ○ undefined
- ○ Error

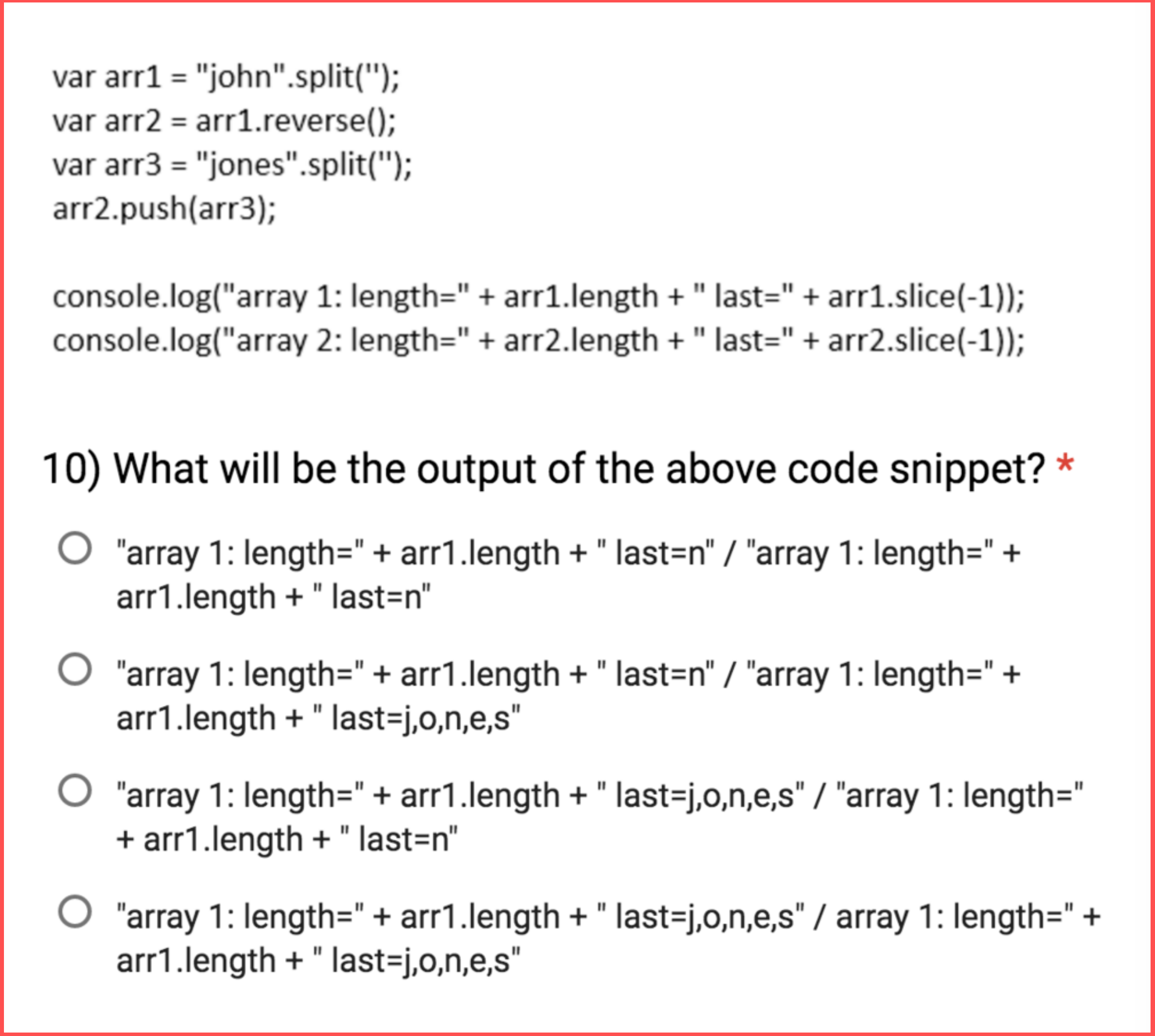
456



```
var a={},
    b={key:'b'},
    c={key:'c'};

a[b]=123;
a[c]=456;

console.log(a[b]);
```

- When setting an object property, JavaScript will implicitly stringify the parameter value.
- since b and c are both objects, they will both be converted to "[object Object]"
- a["[object Object]"] === 456

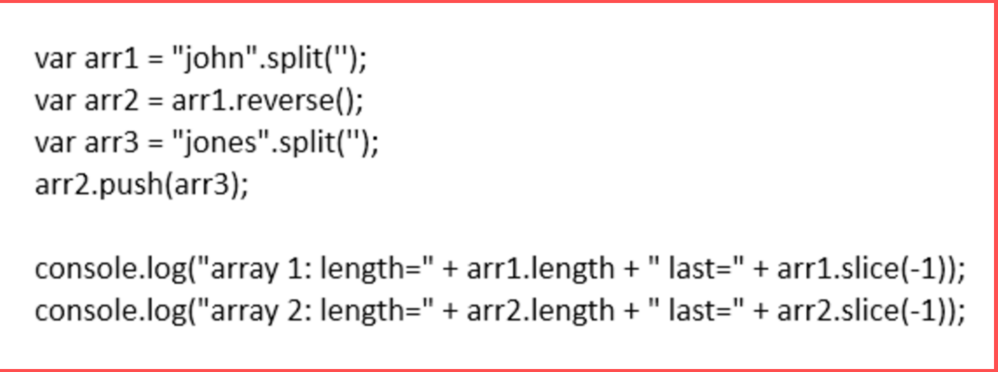
#10



```
var arr1 = "john".split('');
var arr2 = arr1.reverse();
var arr3 = "jones".split('');
arr2.push(arr3);

console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1));
console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1));
```

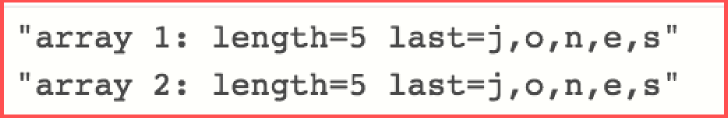
10) What will be the output of the above code snippet? *

- "array 1: length=" + arr1.length + " last=n" / "array 1: length=" + arr1.length + " last=n"
- "array 1: length=" + arr1.length + " last=n" / "array 1: length=" + arr1.length + " last=j,o,n,e,s"
- "array 1: length=" + arr1.length + " last=j,o,n,e,s" / "array 1: length=" + arr1.length + " last=n"
- "array 1: length=" + arr1.length + " last=j,o,n,e,s" / array 1: length=" + arr1.length + " last=j,o,n,e,s"



```
var arr1 = "john".split('');
var arr2 = arr1.reverse();
var arr3 = "jones".split('');
arr2.push(arr3);

console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1));
console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1));
```

ANSWER



```
"array 1: length=5 last=j,o,n,e,s"
"array 2: length=5 last=j,o,n,e,s"
```

- reverse() method reverses an array ***in place*** (i.e., arr1)
- reverse() method returns a reference to the array itself
  - As a result, arr2 is simply a reference to arr1
  - Therefore, when anything is done to arr2, arr1 will be affected

- Passing an array to the push() method of another array pushes that entire array as a single element onto the end of the array.
- Negative subscripts in calls to array methods like slice() references elements at the end of the array
  - e.g., a subscript of -1 indicates the last element in the array