Scope in JavaScript

Thomas H. Austin
San José State University
thomas.austin@sjsu.edu

March 15, 2020

Abstract

Today, we are going to learn the rules for scoping in JavaScript, and cover many of the strange corner cases that we need to be aware of. We will also review the `apply`, `call`, and `bind` methods.

1 Introduction

Scoping in JavaScript has many subtle corner cases, and is one of the issues that can lead to a lot of head-scratching questions. The focus of today's module is to review this area in depth.

First, let's look at a sample program:

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  }
}

// Usage
var addOne = makeAdder(1);
console.log(addOne(10));
```

Read over the code and determine what it should print. Run the code to verify.

Exercise 1. For a quick warm-up exercise, create a `makeListOfAdders` function that works similarly to the last example. It should take in a list of numbers and return a list of functions. Sample usage is given below:

```
var adders = makeListOfAdders([1,5,22]);

adders[0](42); // This function adds 1 to 42 to get 43.
adders[1](42); // This function adds 5 to 42 to get 42.
adders[2](42); // This function adds 22 to 42 to get 64.
```

In your solution, use anonymous functions where possible.
When you have a solution that works, or if you have a solution that *should* work but doesn't, take a look at the next page for one possible approach.

2 Handling Free Variables in JavaScript

One approach that you might have taken to the previous problem is shown below. This function creates an empty array (`arr`) on line 2 to hold the adder functions. It then iterates over the input list (`lst`) and stores each number in the local variable `n`.

After that, it creates a new, anonymous function that takes in one parameter `x` and returns the value of `x + n`. (As a reminder, JavaScript functions are *closures*; these functions will retain access to the *free variable* `n`.) The anonymous function is then added to the array of functions. Finally, the array of functions is returned.

```
 1  function makeListOfAdders(lst) {
 2    var arr = [];
 3    for (var i=0; i<lst.length; i++) {
 4      var n = lst[i];
 5      arr[i] = function(x) { return x + n; }
 6    }
 7    return arr;
 8  }
 9
10  var adders = makeListOfAdders([1,3,99,21]);
11  adders.forEach(function(adder) {
12    console.log(adder(100));
13  });
```

Looking at the code, you might expect the example to print out ‘101’, ‘103’, ‘199’, and ‘121’. But what it *actually* prints out is:

```
121
121
121
121
```

Even stranger, a seemingly innocuous change in the code can produce a completely different output. The code below is revised from the above version to eliminate the need for the local variable `n`.

```
function makeListOfAdders(lst) {
  var arr = [];
  for (var i=0; i<lst.length; i++) {
    arr[i] = function(x) { return x + lst[i]; }
  }
  return arr;
}

var adders = makeListOfAdders([1,3,99,21]);
adders.forEach(function(adder) {
  console.log(adder(100));
});
```

This version instead prints:

```
NaN
NaN
NaN
NaN
```

So what is going on here?

2.1 Variable Hoisting

The issue in the previous codes examples is that JavaScript does *not* have block scope[1], unlike languages such as Java. Any variable you declare with `var` is *hoisted* to the top of the function. So while you might write code like:

```javascript
function(lst) {
  for (var i=0; i<lst.length; i++) {
    var n = lst[i];
  }
}
```

The interpreter treats your code as if you had written:

```javascript
function(lst) {
  var i, n;
  for (i=0; i<lst.length; i++) {
    n = lst[i];
  }
}
```

Exercise 2. Rewrite the two versions of `makeListOfAdders` to show how the variables are hoisted. Explain why each function prints out the results that it does.

[1] Actually, more recent versions of JavaScript do have block scoping. We will review this feature another day.

2.2 Faking Block Scope

The following version of `makeListOfAdders` works correctly.

```
 1  function makeListOfAdders(lst) {
 2    var arr = [];
 3    for (var i=0; i<lst.length; i++) {
 4      (function() {
 5        var n = lst[i];
 6        arr[i] = function(x) { return x + n; }
 7      })(); // The anonymous function is invoked immediately
 8    }
 9    return arr;
10  }
11
12  var adders = makeListOfAdders([1,3,99,21]);
13  adders.forEach(function(adder) {
14    console.log(adder(100));
15  });
```

Note that an anonymous function is created on line 4, which is then immediately invoked on line 7. The result is that `n` is hoisted to the top of the anonymous function, rather than to the top of the `makeListOfAdders` function. Effectively, this is a hack to create block scope.

This design pattern shows up with some frequency in older JavaScript code, but is fading from use as better alternatives are available in newer versions of JavaScript. We will review those alternatives another day.

3 Problems with JavaScript Constructors

Consider the following JavaScript code.

```
1  name = "Monty";
2  function Rabbit(name) {
3    this.name = name;
4  }
5  var r = Rabbit("Python");
6
7  console.log(r.name);
8  console.log(name);
```

Looking at the code, you might expect it to first print out ‘Python’, and then ‘Monty’.

Exercise 3. Run the above code. What does it *actually* print out?
Comment out line 7. Now what does it print out?
What caused the problem?

3.1 Forgetting `new` Causes Strange Results

The issue with the code on the previous page is that `new` was never called. As a result, `Rabbit` is treated like a normal function. Nothing is explicitly returned from the function, so `r` is set to `undefined`. Line 7 then causes an error, since `undefined` does not have a `name` property.

The more bizarre part is that after you comment out line 7, line 8 prints out 'Python' rather than 'Monty'.

The reason for this strange result is how `this` is handled in JavaScript. For constructors and methods, `this` works the way that you would expect. However, for other functions, `this` is instead bound to the global object. So on the update to `this.name` on line 3, the global variable name is changed.

Moral of the story: do not forget `new` in JavaScript.

3.2 More Problems with Rabbits

Here is another use of a JavaScript constructor to consider. (I did not forget `new` this time.)

```
1  function Rabbit(name, favFoods) {
2    this.name = name;
3    this.myFoods = [];
4    favFoods.forEach(function(food) {
5      this.myFoods.push(food);
6    });
7  }
8
9  var bugs = new Rabbit("Bugs", ["carrots", "lettuce", "souls"]);
10 console.log(bugs.myFoods);
```

This code also has an error. In this case, the error happens on line 5:

```
TypeError: Cannot read property 'push' of undefined
```

And yet, the `myFoods` property is set 2 lines earlier. So what is causing this error message?

The issue is that `this` uses *dynamic scoping* in JavaScript. Unlike other variables in JavaScript, which use *lexical scoping*.

The `forEach` loop on line 4 accepts a function as its argument. Since this function is not a constructor or a method, `this` refers to the global object within this anonymous function.

A quick fix is to capture a reference to `this` outside of the `forEach` loop. The following code shows that approach, and correctly prints out "carrots"[2], "lettuce", and "souls".

```
function Rabbit(name, favFoods) {
  this.name = name;
  this.myFoods = [];
  var that = this; // Capturing the current value of 'this'
  favFoods.forEach(function(food) {
    that.myFoods.push(food); // Uses outer 'this' from Rabbit.
  });
}
var bugs = new Rabbit("Bugs", ["carrots", "lettuce", "souls"]);
console.log(bugs.myFoods);
```

[2]Public service announcement: don't feed your rabbit carrots. Clark Gable ate carrots in a famous movie, which is why Bugs Bunny ate them, but they should only be occasional treats for non-animated rabbits.

3.3 Execution Contexts

In JavaScript, scoping information is stored in an *execution context.* It is made up of:

- A variable object
- A scope chain
- A context object

The variable object stores local variables and parameters. For free variables, i.e. a variable that is *not* a local variable or parameter, the JavaScript engine needs to know where to find these variables. The scope chain is effectively a link to another variable object, which itself may link to another variable object, and so on. The *global object* is the top link in this chain. If a variable is not found here, the value of that variable is `undefined`.

The variable object also contains a special `arguments` object, which is an array-like structure (but not a true array) that holds all of the arguments passed to the function.

The last piece is the *context object.* The context object determines what `this` refers to in the current scope. The rules for `this` are a bit complex.

When a function is invoked with `new`, it is treated as a constructor. `this` is bound to a newly created object, which will be implicitly returned from the function. If a function is attached to an object, the function is considered a method, and `this` refers to the object it belongs to. The behavior of `this` in both of these cases matches our expectations coming from Java.

For functions that are not methods or constructors, or where we forgot to use `new` with our constructor, `this` is bound to the global object. I have rarely found this behavior useful, but it is the way that the language works.

There are some additional special cases to be aware of. Within a browser, in-line event handlers on DOM elements bind `this` to the DOM element. There are also 3 special functions where you can explicitly bind `this`: `apply`, `call`, and `bind`.

The following code shows examples of how to use `apply`, `call`, and `bind`.

```
 1  x = 3;
 2
 3  function foo(y) {
 4    console.log(this.x + y);
 5  }
 6  foo(100);
 7
 8  foo.apply(null, [100]);    // Array passed for args
 9  foo.apply({x:4}, [100]);
10
11  foo.call({x:4}, 100);      // No array needed
12
13  var bf = foo.bind({x:5}); // Create a new function
14  bf(100);
```

Line 6 will print ‘103’, since `this` refers to the global object.

Lines 8 and 9 show how `apply` is used. The first argument passed to the `apply` function explicitly sets what `this` refers to. On line 8, since `null` is specified, `this` refers to the global object. The arguments of the function are passed in as an array, so `y` is bound to ‘100’. Therefore, the result of this line is also ‘103’.

On line 9, the first argument is an object where the property `x` is set to ‘4’. For this function call, `this` refers to the object passed in. The result is therefore ‘104’.

Line 11 shows how `call` is used. The only difference from apply is how the arguments are passed in. For `apply`, the arguments must be passed in as an array. For `call`, they are instead passed in "loose".

The `bind` function is different from the other two. It returns a new function, where some of the arguments are already specified. In line 13, `bf` is set to a function where `this` is bound to the object passed in to bind. Therefore, on line 14, the call to `bf` will print out '105'.

Exercise 4. (Note that this exercise might be helpful for homework #3).

Write a `makeDebugWrapperApply` function that takes in a function and returns another function. The new function should print out the arguments passed to it, call the original function, print out the result, and then return that result. Use `apply` to make this function work. You will likely need to use the built-in `arguments` object. Sample usage is shown below:

```
function makeDebugWrapperApply(f, thisObj) {
  return function() {
    // YOUR CODE HERE
  }
}

function add(x, y) {
  return x+y;
}

// Prints 7
console.log(add(3,4));

var addWrapped = makeDebugWrapperApply(add);

// First prints "Passing 8 14",
// then prints "Returning 22",
// and then prints 22.
console.log(addWrapped(8, 14));
```

In Exercise 4, you can specify `null` as the first argument passed to `apply`. If you are working with methods, however, you need to make sure that `this` is bound correctly. The next exercise explores this piece.

Exercise 5. Continuing on from Exercise 4, add support for objects with methods. Sample code is shown below.

```
function Employee(name, salary) {
  this.name = name;
  this.salary = salary;
}
Employee.prototype.displayWages = function(bonus=0) {
  console.log(`Annual wages for ${this.name}: ${this.salary
      + bonus}`);
}

var emp1 = new Employee("Nellanus Glacanus", 100000);
var emp2 = new Employee("Joe Bob Briggs", "$20 and half a
    case of Budweiser ");

wages1 = makeDebugWrapperApply(emp1.displayWages, emp1);
wages2 = makeDebugWrapperApply(emp2.displayWages, emp2);

// First prints "Passing 5000",
// then prints "Annual wages for Nellanus Glacanus:
    105000",
// then prints "No return value".
wages1(5000);

// First prints "No arguments passed",
// then prints "Annual wages for Nellanus Glacanus:
    100000",
// then prints "No return value".
wages1();

// First prints "Passing 18",
// then prints "Annual wages for Joe Bob Briggs: $20 and
    half a case of Budweiser 18",
// then prints "No return value".
wages2(18);
```

4 Comments?

This module is a bit of an experiment. As a final (optional) exercise, let me know your thoughts.

Exercise 6. Are any concepts from today's module still unclear? What portions would you like me to review in more depth?

Does the mix of this module and a lecture-free Zoom session seem to work? Did you encounter any issues that I should try to address for next time?

Were you able to get through all of the exercises during class time? If not, how far did you get?

Lastly, did you spot any typos in the text?