ATOMIC OBJECT

October 20, 2014 by: Kaitlin Davis

13 Comments

What You Should Already Know about JavaScript Scope

If you are a novice JavaScript programmer, or if you've been messing around with JQuery to pull off a few animations on your website, chances are you're missing a few vital chunks of knowledge about JavaScript.

One of the most important concepts is how scope binds to "*this*".

For this post, I'm going to assume you have a decent understanding of JavaScript's basic syntax/objects and general terminology when discussing scope (block vs. function scope, this keyword, lexical vs. dynamic scoping).

Lexical Scoping

First off, JavaScript has *lexical scoping* with *function scope*. In other words, even though JavaScript looks like it should have block scope because it uses curly braces { }, a new scope is created only when you create a new function.

```
var outerFunction  = function(){

   if(true){
      var x = 5;
      //console.log(y); //line 1, ReferenceError: y not defined
   }

   var nestedFunction = function() {

      if(true){
         var y = 7;
         console.log(x); //line 2, x will still be known prints 5
      }

      if(true){
         console.log(y); //line 3, prints 7
       }
   }
   return nestedFunction;
}

var myFunction = outerFunction();
myFunction();
```

In this example, the variable *x* is available everywhere inside of `outerFunction()`. Also, the variable *y* is available everywhere within the `nestedFunction()`, but neither are available outside of the function where they were defined. The reason for this can be explained by lexical scoping. The scope of variables is defined by their position in source code. In order to resolve variables, JavaScript starts at the innermost scope and searches outwards until it finds the variable it was looking for. Lexical scoping is nice, because we can easily figure out what the value of a variable will be by looking at the code; whereas in dynamic scoping, the meaning of a variable can change at runtime, making it more difficult.

Closures

The fact that we can access the variable *x* might still be confusing, because, normally, a local variable inside a function is gone after a function finishes executing. We called `outerFunction()` and assigned its result, `nestedFunction()`, into `myFunction()`. How does the variable *x* still exist if `outerFunction()` has already returned?

Merely accessing a variable outside of the immediate scope (no return statement is necessary) will create something called a *closure*. Mozilla Development Network(MDN) gives a great definition:

> *"A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created."*

Since *x* is a member of the environment that created `nestedFunction()`, `nestedFunction()` will have access to it. Straightforward enough? It's about to get more interesting, because *this* doesn't behave like a regular variable. Try this example, which has nested functions inside of an object with some properties:

```
var cat = {

   name: "Gus",
   color: "gray",
   age: 15,

   printInfo: function() {
      console.log("Name:", this.name, "Color:", this.color, "Age:", this.age); //line 1, prints correctly

      nestedFunction = function() {
         console.log("Name:", this.name, "Color:", this.color, "Age:", this.age); //line 2, loses cat scop
      }

   nestedFunction();
   }
}
cat.printInfo(); //prints Name: window Color: undefined Age: undefined
```

Why are color and age undefined in line 2? You might be thinking, “the cat object properties are clearly defined above and are in an outer more global scope aren’t they?” More importantly, where did “window” come from?

JavaScript loses scope of *this* when used inside of a function that is contained inside of another function. When it is lost, by default, *this* will be bound to the global window object. In our example, it just so happens that the window object also has a “name” property with a value of “window”.

Controlling Context

So what now?

We can’t alter how lexical scoping in JavaScript works, but we can control the *context* in which we call our functions. Context is decided at runtime when the function is called, and it’s always bound to the object the function was called within. The only exception to this rule is the nested function case above.

By saying, change the “context”, I mean, we’re changing what “*this*” actually is. In the example below, what do “line 1” and “line 2” print out?

```
var obj1 = {
   printThis: function() {
      console.log(this);
   }
};

var func1 = obj1.printThis;
obj1.printThis(); //line 1
func1(); //line 2
```

Line 1 prints out obj1. Line 2 prints out the window. The context of line 1 is obj1, because we called `printThis()` on it immediately. However, in `func1()`, we first store a reference to the `printThis()` function, then call it in context of the global object; as a result, it prints the window. If `func1()` had been nested within a different function, that would have been its context.

Call, Bind, and Apply

There are multiple ways to control the value of *this*, including the following:

> storing a reference to *this* in another variable
>
> .call()
>
> .apply()
>
> .bind()

The first one:

```
var cat = {
   name: "Gus",
   color: "gray",
   age: 15,

   printInfo: function() {
      var that = this;
      console.log("Name:", this.name, "Color:", this.color, "Age:", this.age); //prints correctly

      nestedFunction = function() {
         console.log("Name:", that.name, "Color:", that.color, "Age:", that.age); //prints correctly
      }
   nestedFunction();
   }
}
cat.printInfo();
```

Because we bound *this* to a variable *that*, it will be available just like any other variable. Let’s take a look now at `call()`, `apply()`, and `bind()`, which, I think, are a bit cleaner.

```
var cat = {
   name: "Gus",
   color: "gray",
   age: 15,

   printInfo: function() {
      console.log("Name:", this.name, "Color:", this.color, "Age:", this.age);
      nestedFunction = function() {
         console.log("Name:", this.name, "Color:", this.color, "Age:", this.age);
      }
      nestedFunction.call(this);
      nestedFunction.apply(this);

      var storeFunction = nestedFunction.bind(this);
         storeFunction();
      }
}
cat.printInfo();
```

Focus on the first argument of each function: *this*. It refers to the cat object. Now, `nestedFunction()`'s *this* refers to the cat object. If, instead of passing in *this* as an argument, we passed in a "dog" object, then `nestedFunction()`'s *this* refers to dog. Basically, whatever the first argument is becomes the nestedFunction()'s *this.*

So what's the difference between the three?

The main difference between `call()` and `apply()` lies within how to pass extra arguments. `Call()` takes in multiple arguments, separated by commas, which allows the `nestedFunction()` to utilize them. `Bind()` also takes extra arguments in this way. However, `apply()` takes in a single array of arguments, rather than multiple arguments.

It is important to remember that using `call()` and `apply()` actually invoke your function — so this would be incorrect (notice the () on nestedFunction):

```
nestedFunction().call(this);
```

`Bind()`, on the other hand, is nifty because it allows you to change what *this* references and then store a reference to the altered function in a variable to be used at a later time (see storeFunction in the code example above). Meanwhile, because `call()` and `apply()` immediately run the function, they return the result of calling that function.

Practical Applications

Thus far, we have seen closures, `call()`, `apply()`, and `bind()`, but we have not discussed any practical applications and when to use each one to get the correct binding of *this.*

1. Closures are best used when you have nested functions similar to the first cat example above, which used `var that = this`. This method is also nice because you don't have to worry about cross-platform issues. For instance, `bind()` is a recent addition to ECMAScript5 and isn't always supported.

2. `Call()` and `apply()` are useful for when you want to borrow a method from one object and use it in a completely separate object. For example, using our cat

example above, we could reuse its `printInfo()` function to print information about a dog.

3. `Bind()` is useful for maintaining context in asynchronous callbacks and events.

Great! I hope you now have a little more insight regarding how scope in JavaScript is handled. We have covered the basics of lexical scoping with function scope, closures, ways to control the context through closures, `call()`, `apply()`, `bind()`, and lastly, some practical applications.

Related Posts