Javascript Basics

Scope

Now that we covered functions, we can cover scope. If you're already comfortable with how scope works in Javascript, you can skip this session.

What is Scope

MDN describes scope as:

> The current context of execution. The context in which values and **expressions** are "visible," or can be referenced. If a **variable** or other expression is not "in the current scope," then it is unavailable for use. Scopes can also be layered in a hierarchy, so that child scopes have access to parent scopes, but not vice versa.

So what does this have to do with functions?

A function body exists in its own "scope", or context. A variable declared in the context of a function will not be visible to the outer "scope", or context.

```
function add2 (number) {
  var two = 2
  return number + two
}
add2(1)
```

In this example, the "two" variable that is declared in the add2 function is not visible outside of the function. So if I did this:

```
function add2 (number) {
  var two = 2
  return number + two
}
add2(1)
console.log(two) // two isn't defined in this scope
```

an error would be thrown.

A scope of a function is also known as a closure in Javascript.

The outer scope cannot reference the inner scope, but the inner scope can see what's in the outer scope. So if the two was put outside the function, the console log would be fine:

```
var two = 2
function add2 (number) {
  return number + two // inner scope can see the outer scope
}
add2(1)
console.log(two) // two is defined in this scope
```

What defines whether a variable can be seen in a particular scope is where it is declared, not defined. So if the code was rewritten to define the two variable inside of the scope, it would also work:

```
var two // declared, but not defined
function add2 (number) {
  two = 2
  return number + two // inner scope can see the outer scope
}
add2(1) // two won't be set unless this function is called
console.log(two) // two is defined in this scope
```

So why is scope important?

The value in scope is that you can separate concerns and isolate logic inside an inner scope and not worry about affecting the outer scope. That inner function could declare hundreds of variables and we would never need to know about them or worry about them in the outer scope.

The inner function can even declare and define a variable that already exists in the outer scope, and it won't affect what's in the outer scope. For example, let's declare and define two in the outer scope to equal "50", but then declare and define our own value for two in the inner function:

```
var two = 50
function add2 (number) {
  var two = 2
  return number + two // inner scope two is used
}
add2(1)
console.log(two) // outputs 50
```

By the way, the function's argument or arguments, in this case "number", are considered variables inside of the function scope, and are not visible to the outside scope.

Scopes can be nested. I'll rewrite this function to add 22, putting a function within a function (and changing the variables around a bit):

```
var amountToAdd = 50

function add22 (number) {
```

```
  var amountToAdd = 20

  function add2 (number) {
    var amountToAdd = 2
    return number + amountToAdd // number + 2
  }

  return add2(number) + amountToAdd // number + 20
}
console.log('1 + 22 = ' + add22(1))
console.log('amountToAdd: ' + amountToAdd) // still 50
```

So the inner most function declares and defines the amount to add as 2, which does not affect the parent function's scope, which set it to 20 earlier, which doesn't affect the global scope, which declared and defined amountToAdd to 50.

By the way, the add2 function inside the add22 function is not visible in the outer scope either. Consider add2 a variable in the inner scope.

Defining functions within functions in Javascript is very common, and is used to isolate variables and logic within certain scopes. We can rewrite the add22 body to execute the add2 function without defining it on a separate line. To pull this off, surround the function definition in parentheses, and then added open/close parentheses at the end.

```
var amountToAdd = 50

function add22 (number) {
  var amountToAdd = 20

  // (function () {})(function arguments)
  return (function add2 (number) {
    var amountToAdd = 2
    return number + amountToAdd // number + 2
  })(number) + amountToAdd // number + 20
}
console.log('1 + 22 = ' + add22(1))
console.log('amountToAdd: ' + amountToAdd) // still 50
```

Since the function is being defined and called at the same time, there's no need to name it:

```
var amountToAdd = 50

function add22 (number) {
  var amountToAdd = 20

  // (function () {})(function arguments)
  return (function (number) {
    var amountToAdd = 2
    return number + amountToAdd // number + 2
  })(number) + amountToAdd // number + 20
}
console.log('1 + 22 = ' + add22(1))
console.log('amountToAdd: ' + amountToAdd) // still 50
```

When a function has no name, it is called an anonymous function, or a function expression. Anonymous functions are useful when defining self executing functions like this, or when passing functions as arguments into another function.

Let me show you an example of passing a function in as an argument. Remember when we went over the sort functions for arrays? Let's say you want to sort an array of numbers from smallest to largest:

```
var numericArray = [1, 28, -1, 4, -70]
numericArray.sort()
console.log(numericArray)
```

The default behavior of the sort function is to sort based off the string representation of each array value, which more often than not is actually not what you want to sort by. As you can see, sorting this array returns numbers in the wrong order.

To sort in the right order, you can add an optional argument to the sort call: a function that compares two of the values passed in as arguments. If the function passed in returns a negative number, then the value that was the first argument is placed earlier in the array. If the function returns zero, the order is left unchanged. If the function returns a number greater than zero, the second argument is placed earlier in the array.

Knowing this, a common quick way to sort numbers is to return the result of the first argument minus the second one. For example, given numbers 4 and 28, since 4 minus 28 is a negative number, the sort algorithm will know to put 4 before 28.

```
var numericArray = [1, 28, -1, 4, -70]
function compareNumbers (numberA, numberB) {
  return numberA - numberB
}
numericArray.sort(compareNumbers)
console.log(numericArray)
```

Now sort does what we'd expect.

But since we're only using the compareNumbers function in the sort call, there's no reason to declare it as a separate function. We can just do this:

```
var numericArray = [1, 28, -1, 4, -70]
numericArray.sort(function compareNumbers (numberA, numberB) {
  return numberA - numberB
})
console.log(numericArray)
```

And remove the name of the function:

```
var numericArray = [1, 28, -1, 4, -70]
numericArray.sort(function (numberA, numberB) {
  return numberA - numberB
})
console.log(numericArray)
```

That covers the basic scopes and anonymous functions for now, at least as far as variables defined with the var keyword are concerned.

What about properties on objects though? We could do this:

```
var myObject = {
  innerProperty: 'hello',
  getInnerProperty: function () {
    return myObject.innerProperty
  }
}
console.log(myObject.innerProperty)
console.log(myObject.getInnerProperty())
```

But there's some limitations with this approach.

Remember that variables are just references to values in memory? Let's say we made a variable that references the function that returns the object's inner property:

```
var myObject = {
  innerProperty: 'hello',
  getInnerProperty: function () {
    return myObject.innerProperty
  }
}
var getInnerProperty = myObject.getInnerProperty
console.log(myObject.innerProperty)
console.log(getInnerProperty())
```

and then changed the reference to myObject:

```
var myObject = {
  innerProperty: 'hello',
  getInnerProperty: function () {
    return myObject.innerProperty
  }
}
var getInnerProperty = myObject.getInnerProperty

// Make a new object, but use the same reference
var myObject = {
  innerProperty: 'something completely different'
}
console.log(myObject.innerProperty)
console.log(getInnerProperty())
```

The original myObject value hasn't changed even though nothing is directly referencing it, but because the function is returning whatever the current "myObject.innerProperty" value is, it's going to return the new object's innerProperty.

Or even more rudimentary than that, what if you just want to change the name of the myObject variable?

```
var differentName = {
  innerProperty: 'hello',
  getInnerProperty: function () {
    return differentName.innerProperty // need to change it here too!
  }
}
console.log(differentName.innerProperty)
console.log(differentName.getInnerProperty())
```

You'll need to change the name of the variable everywhere within the object. Which means the inner object context needs to know about the outside state, which is typically not good programming practice.

So how do we reference the former object's innerProperty without needing to memorize the name, or without risking the object references changing from outside the object?

We can replace the inner name with the "this" keyword:

```
var myObject = {
  innerProperty: 'hello',
  getInnerProperty: function () {
    return this.innerProperty
  }
}
console.log(myObject.innerProperty)
console.log(myObject.getInnerProperty())
```

The `this` keyword is a reference to the current context of a scope. In the global scope, the this keyword will reference the window object in the browser:

```
console.log(window === this)
```

When used in a function defined inside an object, it references the object that contains the function. The this keyword applies even if the function is defined outside of the object definition:

```
function getInnerProperty () {
  return this.innerProperty
}
var myObject = {
  innerProperty: 'hello',
  getInnerProperty: getInnerProperty
}
console.log(myObject.innerProperty)
console.log(myObject.getInnerProperty())
```

That's the basics of `this` anyway, but there are lots of quirks to how it works, especially when you start involving object prototypes and functions that can change the context that a function is called in, like `bind`, `call`, and `apply`.

For that reason, while I would encourage you to play around with the this keyword to familiarize yourself with it, I would recommend using it sparingly and only when it is very clear what its intention is in your professional development. There are often better ways to structure code so that the this keyword is not necessary.

If I were to restructure the provided code, I would make it so the inner property is not directly accessible, and the only way to get it is through the get function. That way, outside code wouldn't be able to modify it and change the behavior. This can be done in a couple of ways with anonymous functions. For example:

```
var myObject = (function() {
  var innerProperty = 'hello'
        return {
    getInnerProperty: function () {
            return innerProperty
    }
  }
})()
console.log(myObject.innerProperty) // undefined because it isn't exposed
console.log(myObject.getInnerProperty())
```

So myObject is the result of a self executing function that defines its own scoped variables and returns an object for accessing those variables.

Another very common approach for those with OOP backgrounds is to define a named function that doesn't self execute and create an object with the `new` operator, like this:

```
var MyObjectConstructor = function() {
  var innerProperty = 'hello'
        return {
    getInnerProperty: function () {
            return innerProperty
    }
  }
}
var myObject = new MyObjectConstructor()
console.log(myObject.innerProperty) // undefined because it isn't exposed
console.log(myObject.getInnerProperty())
```

When a function is invoked via `new`, if the function doesn't return an object, the "this" object is returned. Otherwise, it returns the object specified as the return.

So if you want to use the this keyword in this case, you can also do this:

```
var MyObjectConstructor = function() {
  var innerProperty = 'hello'
  this.getInnerProperty = function () {
          return innerProperty
  }
  // return this
}
var myObject = new MyObjectConstructor()
console.log(myObject.innerProperty) // undefined because it isn't exposed
console.log(myObject.getInnerProperty())
```

And it will do the same thing.

I try to avoid using `new` for the most part as Javascript is not fundamentally an OOP language, but there are some use cases where it makes sense to.

And that's most of what you'll need to know for scope and the this keyword. Scope is a mechanism for concealing variable declarations within a smaller context so that it can't be affected by outside code. And the `this` keyword is a reference to whatever the current scope is.