#react #vue #angular #javascript #node #laravel #css #vs-code #python

Introduction

JavaScript has a feature called Scope. Though the concept of scope is not that easy to understand for many new developers, I will try my best to explain them to you in the simplest scope. Understanding `scope` will make your code stand out, reduce errors and help you make powerful design patterns with it.

What is Scope?

Table of Contents

Scope is the accessibility of variables, functions, and objects in some particular part of your code during runtime. In other words, scope determines the visibility of variables and other resources in areas of your code.

Why Scope? The Principle of Least Access

So, what's the point in limiting the visibility of variables and not having everything available everywhere in your code? One advantage is that scope provides some level of security to your code. One common principle of computer security is that users should only have access to the stuff they need at a time.

Think of computer administrators. As they have a lot of control over the company's systems, it might seem okay to grant full access user account to them. Suppose you have a company with three administrators, all of them having full access to the systems and everything is working smooth. But suddenly something bad happens and one of your system gets infected with a malicious virus. Now you don't know whose mistake that was? You realize that you should them basic user accounts and only grant full access privileges when needed. This will help you track changes and keep an account of who did what. This is called The Principle of Least Access. Seems intuitive? This principle is also applied to programming language designs, where it is called scope in most programming languages including JavaScript which we are going to study next.

As you continue on in your programming journey, you will realize that scoping parts of your code helps improve efficiency, track bugs and reduce them. Scope also solves the naming problem when you have variables with the same name but in different scopes. Remember not to confuse scope with context. They are both different features.

Scope in JavaScript

In the JavaScript language there are two types of scopes:

- » Global Scope
- » Local Scope

Variables defined **inside a function** are in local scope while variables defined outside of a function are in the global scope. Each function when invoked creates a new scope.

Global Scope

When you start writing JavaScript in a document, you are already in the Global scope. There is only one Global scope throughout a JavaScript document. A variable is in the Global scope if it's defined outside of a function.

JAVASCRIPT

```javascript
// the scope is by default global
var name = 'Hammad';
```

Variables inside the Global scope can be accessed and altered in any other scope.

JAVASCRIPT

```
var name = 'Hammad';

console.log(name); // logs 'Hammad'

function logName() {
    console.log(name); // 'name' is accessible here and everywher
}

logName(); // logs 'Hammad'
```

Local Scope

Variables defined inside a function are in the local scope. And they have a different scope for every call of that function. This means that variables having the same name can be used in different functions. This is because those variables are bound to their respective functions, each having different scopes, and are not accessible in other functions.

JAVASCRIPT

```
// Global Scope
function someFunction() {
    // Local Scope #1
    function someOtherFunction() {
        // Local Scope #2
    }
}

// Global Scope
function anotherFunction() {
    // Local Scope #3
}
// Global Scope
```

Block Statements

Block statements like `if` and `switch` conditions or `for` and `while` loops, unlike functions, don't create a new scope. Variables defined inside of a block statement will remain in the scope they were already in.

JAVASCRIPT

```
if (true) {
    // this 'if' conditional block doesn't create a new scope
    var name = 'Hammad'; // name is still in the global scope
}

console.log(name); // logs 'Hammad'
```


ECMAScript 6 introduced the `let` and `const` keywords. These keywords can be used in place of the `var` keyword.

JAVASCRIPT

```
var name = 'Hammad';

let likes = 'Coding';
const skills = 'Javascript and PHP';
```

Contrary to the `var` keyword, the `let` and `const` keywords support the declaration of local scope inside block statements.

JAVASCRIPT

```
if (true) {
    // this 'if' conditional block doesn't create a scope

    // name is in the global scope because of the 'var' keyword
    var name = 'Hammad';
    // likes is in the local scope because of the 'let' keyword
    let likes = 'Coding';
    // skills is in the local scope because of the 'const' keywor
    const skills = 'JavaScript and PHP';
}

console.log(name); // logs 'Hammad'
console.log(likes); // Uncaught ReferenceError: likes is not def
console.log(skills); // Uncaught ReferenceError: skills is not de
```

> *Global scope lives as long as your application lives. Local Scope lives as long as your functions are called and executed.*

Context

Many developers often confuse scope and context as if they equally refer to the same concepts. But this is not the case. Scope is what we discussed above and *Context* is used to refer to the value of `this` in some particular part of your code. Scope refers to the visibility of variables and context refers to the value of `this` in the same scope. We can also change the context using function methods, which we will discuss later. In the global scope context is always the Window object.

JAVASCRIPT

```
// logs: Window {speechSynthesis: SpeechSynthesis, caches: Caches
console.log(this);

function logFunction() {
    console.log(this);
}
// logs: Window {speechSynthesis: SpeechSynthesis, caches: Caches
// because logFunction() is not a property of an object
logFunction();
```

If scope is in the method of an object, context will be the object the method is part of.

JAVASCRIPT

```
class User {
    logName() {
        console.log(this);
    }
}

(new User).logName(); // logs User {}
```

> *(New User).logName() is a short way of storing your object in a variable and then calling the logName function on it. Here, you don't need to create a new variable.*

One thing you'll notice is that the value of context behaves differently if you call your functions using the `new` keyword. The context will then be set to the instance of the called function. Consider one of the examples above with the function called with the `new` keyword.

JAVASCRIPT

```
function logFunction() {
    console.log(this);
}
```

```
new logFunction(); // logs logFunction {}
```

> *When a function is called in **Strict Mode**, the context will default to undefined.*

Execution Context

To remove all confusions and from what we studied above, the word *context* in **Execution Context** refers to scope and not context. This is a weird naming convention but because of the JavaScipt specification, we are tied to it.

JavaScript is a single-threaded language so it can only execute a single task at a time. The rest of the tasks are queued in the Execution Context. As I told you earlier that when the JavaScript interpreter starts to execute your code, the context (scope) is by default set to be global. This global context is appended to your execution context which is actually the first context that starts the execution context.

After that, each function call (invocation) would append its context to the execution context. The same thing happens when an another function is called inside that function or somewhere else.

> *Each function creates its own execution context.*

Once the browser is done with the code in that context, that context will then be popped off from the execution context and the state of the *current* context in the execution context will be transferred to the parent context. The browser always executes the execution context that is at the top of the execution stack (which is actually the innermost level of scope in your code).

> *There can only be one global context but any number of function contexts.*

The execution context has two phases of creation and code execution.

Creation Phase

The first phase that is the creation phase is present when a function is called but its code is not yet executed. Three main things that happen in the creation phase are:

- » Creation of the Variable (Activation) Object,
- » Creation of the Scope Chain, and
- » Setting of the value of context (`this`)

Variable Object

The Variable Object, also known as the activation object, contains all of the variables, functions and other declarations that are defined in a particular branch of the execution context. When a function is called, the interpreter scans it for all resources including function arguments, variables, and other declarations. Everything, when packed into a single object, becomes the the Variable Object.

JAVASCRIPT

```
'variableObject': {
    // contains function arguments, inner variable and function d
}
```

Scope Chain

In the creation phase of the execution context, the scope chain is created after the variable object. The scope chain itself contains the variable object. The Scope Chain is used to resolve variables. When asked to resolve a variable, JavaScript always starts at the innermost level of the code nest and keeps jumping back to the parent scope until it finds the variable or any other resource it is looking for. The scope chain can simply be defined as an object containing the variable object of its own execution context and all the other execution contexts of it parents, an object having a bunch of other objects.

JAVASCRIPT

```
'scopeChain': {
    // contains its own variable object and other variable objec
```

```
}
```

The Execution Context Object

The execution context can be represented as an abstract object like this:

JAVASCRIPT

```
executionContextObject = {
    'scopeChain': {}, // contains its own variableObject and othe
    'variableObject': {}, // contains function arguments, inner v
    'this': valueOfThis
}
```

Code Execution Phase

In the second phase of the execution context, that is the code execution phase, other values are assigned and the code is finally executed.

Lexical Scope

Lexical Scope means that in a nested group of functions, the inner functions have access to the variables and other resources of their parent scope. This means that the child functions are lexically bound to the execution context of their parents. Lexical scope is sometimes also referred to as Static Scope.

JAVASCRIPT

```
function grandfather() {
    var name = 'Hammad';
    // likes is not accessible here
    function parent() {
        // name is accessible here
        // likes is not accessible here
        function child() {
            // Innermost level of the scope chain
            // name is also accessible here
            var likes = 'Coding';
        }
    }
}
```

The thing you will notice about lexical scope is that it works forward, meaning `name` can be accessed by its children's execution contexts. But it doesn't work backward to its parents, meaning that the variable `likes` cannot be accessed by its parents. This also tells us that variables having the same name in different execution contexts gain precedence from top to bottom of the execution stack. A variable, having a name similar to another variable, in the innermost function (topmost context of the execution stack) will have higher precedence.

Closures

The concept of closures is closely related to Lexical Scope, which we studied above. A Closure is created when an inner function tries to access the scope chain of its outer function meaning the variables outside of the immediate lexical scope. Closures contain their own scope chain, the scope chain of their parents and the global scope.

> *A closure can not only access the variables defined in its outer function but also the arguments of the outer function.*

A closure can also access the variables of its outer function even after the function has returned. This allows the returned function to maintain access to all the resources of the outer function.

When you return an inner function from a function, that returned function will not be called when you try to call the outer function. You must first save the invocation of the outer function in a separate variable and then call the variable as a function. Consider this example:

```javascript
function greet() {
    name = 'Hammad';
    return function () {
        console.log('Hi ' + name);
    }
}

greet(); // nothing happens, no errors

// the returned function from greet() gets saved in greetLetter
greetLetter = greet();

 // calling greetLetter calls the returned function from the gree
greetLetter(); // logs 'Hi Hammad'
```

The key thing to note here is `that greetLetter` function can access the name variable of the greet function even after it has been returned. One way to call the returned function from the `greet` function without variable assignment is by using parentheses `()` two times `()()` like this:

```javascript
function greet() {
    name = 'Hammad';
    return function () {
        console.log('Hi ' + name);
    }
}

greet()(); // logs 'Hi Hammad'
```

Public and Private Scope

In many other programming languages, you can set the visibility of properties and methods of classes using public, private and protected scopes. Consider this example using the PHP language:

```php
// Public Scope
public $property;
public function method() {
  // ...
}

// Private Sccpe
private $property;
private function methoa() {
  // ...
}

// Protected Scope
protected $property;
protected function methoa() {
  // ...
}
```

Encapsulating functions from the public (global) scope saves them from vulnerable attacks. But in JavaScript, there is no such thing as public or private scope. However, we can emulate this feature using closures. To keep everything separate from the global we must first encapsulate our functions within a function like this:

```javascript
(function () {
  // private scope
})();
```

The parenthesis at the end of the function tells the interpreter to execute it as soon as it reads it without invocation. We can add functions and variables in it and they will not

accessible outside. But what if we want to access them outside, meaning we want some of them to be public and some of them to be private? One type of closure, we can use, is called the Module Pattern which allows us to scope our functions using both public and private scopes in an object.

The Module Pattern

The Module Pattern looks like this:

```javascript
var Module = (function() {
    function privateMethod() {
        // do something
    }

    return {
        publicMethod: function() {
            // can call privateMethod();
        }
    };
})();
```

The return statement of the Module contains our public functions. The private functions are just those that are not returned. Not returning functions makes them inaccessible outside of the Module namespace. But our public functions can access our private functions which make them handy for helper functions, AJAX calls, and other things.

```javascript
Module.publicMethod(); // works
Module.privateMethod(); // Uncaught ReferenceError: privateMethod
```

One convention is to begin private functions with an underscore, and returning an anonymous object containing our public functions. This makes them easy to manage in a long object. This is how it looks:

```javascript
var Module = (function () {
    function _privateMethod() {
        // do something
    }
    function publicMethod() {
        // do something
    }
    return {
        publicMethod: publicMethod,
    }
})();
```

Immediately-Invoked Function Expression (IIFE)

Another type of closure is the Immediately-Invoked Function Expression (IIFE). This is a self-invoked anonymous function called in the context of window, meaning that the value of `this` is set `window`. This exposes a single global interface to interact with. This is how it looks:

```javascript
(function(window) {
    // do anything
})(this);
```

Changing Context with .call(), .apply() and .bind()

Call and Apply functions are used to change the context while calling a function. This gives you incredible programming capabilities (and some ultimate powers to **Rule The World**). To use the call or apply function, you just need to call it on the function instead

of invoking the function using a pair of parenthesis and pass the context as the first argument. The function's own arguments can be passed after the context.

```javascript
function hello() {
    // do something...
}

hello(); // the way you usually call it
hello.call(context); // here you can pass the context(value of th
hello.apply(context); // here you can pass the context(value of t
```

The difference between `.call()` and `.apply()` is that in Call, you pass the rest of the arguments as a list separated by a comma while apply allows you to pass the arguments in an array.

```javascript
function introduce(name, interest) {
    console.log('Hi! I\'m '+ name +' and I like '+ interest +'.')
    console.log('The value of this is '+ this +'.')
}

introduce('Hammad', 'Coding'); // the way you usually call it
introduce.call(window, 'Batman', 'to save Gotham'); // pass the a
introduce.apply('Hi', ['Bruce Wayne', 'businesses']); // pass the

// Output:
// Hi! I'm Hammad and I like Coding.
// The value of this is [object Window].
// Hi! I'm Batman and I like to save Gotham.
// The value of this is [object Window].
// Hi! I'm Bruce Wayne and I like businesses.
// The value of this is Hi.
```

> *Call is slightly faster in performance than Apply.*

The following example takes a list of items in the document and logs them to the console one by one.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Things to learn</title>
</head>
<body>
    <h1>Things to Learn to Rule the World</h1>
    <ul>
        <li>Learn PHP</li>
        <li>Learn Laravel</li>
        <li>Learn JavaScript</li>
        <li>Learn VueJS</li>
        <li>Learn CLI</li>
        <li>Learn Git</li>
        <li>Learn Astral Projection</li>
</ul>
<script>
    // Saves a NodeList of all list items on the page in list
    var listItems = document.querySelectorAll('ul li');
    // Loops through each of the Node in the listItems NodeL
    for (var i = 0; i < listItems.length; i++) {
      (function () {
        console.log(this.innerHTML);
      }).call(listItems[i]);
    }

    // Output logs:
    // Learn PHP
    // Learn Laravel
    // Learn JavaScript
    // Learn VueJS
    // Learn CLI
    // Learn Git
```

```
        // Learn Astral Projection
    </script>
</body>
</html>
```

The HTML only contains an unordered list of items. The JavaScript then selects all of them from the DOM. The list is looped over till the end of the items in the list. Inside the loop, we log the content of the list item to the console.

This log statement is wrapped in a function wrapped in parentheses on which the `call` function is called. The corresponding list item is passed to the call function so that the `the` keyword in the console statement logs the innerHTML of the correct object.

Objects can have methods, likewise functions being objects can also have methods. In fact, a JavaScript function comes with four built-in methods which are:

- » Function.prototype.apply()
- » Function.prototype.bind() (Introduced in *ECMAScript 5 (ES5)*)
- » Function.prototype.call()
- » Function.prototype.toString()

> *Function.prototype.toString() returns a string representation of the source code of the function.*

Till now, we have discussed `.call()`, `.apply()`, and `toString()`. Unlike Call and Apply, Bind doesn't itself call the function, it can only be used to bind the value of context and other arguments before calling the function. Using Bind in one of the examples from above:

JAVASCRIPT

```
(function introduce(name, interest) {
    console.log('Hi! I\'m '+ name +' and I like '+ interest +'.')
    console.log('The value of this is '+ this +'.')
}).bind(window, 'Hammad', 'Cosmology')();

// logs:
// Hi! I'm Hammad and I like Cosmology.
// The value of this is [object Window].
```

Bind is like the `call` function, it allows you pass the rest of the arguments one by one separated by a comma and not like `apply`, in which you pass the arguments in an array.

Conclusion

These concepts are radical to JavaScript and important to understand if you want to approach more advanced topics. I hope you got a better understanding of JavaScript Scope and things around it. If something just didn't click, feel free to ask in the comments below.

Scope up your code and till then, Happy Coding!

Like this article? Follow @shammadahmed on Twitter (https://twitter.com/shammadahmed)

Read next...