The hitchhiker’s guide to web development

Table of Contents

1. Week 1: Variables, scopes, hoisting

Goals

- Explore scoping in JavaScript
- See how nested scopes work
- Understand how variable definitions are preparsed

1.1. Initialising vs. defining

When we create variables, we can do so by either `defining` or `initialising` them or both:

Listing 1: initialisation and definition

```
var a ①
var b = 2 ②

a = 1 ③
```

① We **define** variable `a`, but it still has no value (i.e. is `undefined`).

② We **define** and **initialise** variable `b`.

③ We **initialise** variable `a`.

1.2. Variables in JavaScript

In JavaScript (as well as other programming languages) have a lifetime, usually referred to as **scope**. The scope defines when a variable starts and ceases to exist.

For example: In which lines do the variables `a`, `b` and `c` start and end to exist?

Listing 2: global and local variables

```
var a = 1
function x() {
  var b = 2
  console.log(a, b)
  c = 3
  console.log(a, b, c)
}

x()
console.log(a, c)
console.log(window.a, window.c)
```

When running the program we can see that `a` exists for the entire program, `b` only exists inside the function `x` and `c` exists from the moment we initialised it to the end of the program.

Here are three things visible:

1. a `function` creates a new scope and variables defined with `var` only exist in this scope. We call them **local** to the scope.
2. a variable that is defined without `var`, like `c` in our example, will exist in the **global** scope.
3. global variables will be added to the `window` object.

1.3. Scopes

Like many other programming languages, JavaScript has blocks (a list of statements grouped by curly braces):

Listing 3: Scopes

```
{ ①
  console.log('Hello')
  var x = 1
  console.log(x)
}

if(true) { ②
  console.log('Yes')
  isTrue = true
}
```

① Blocks can be used without additional statements…

② … or as part of certain keywords, such as `if`, `for`, `while` etc.

Now, what is the scope of the variable `b`?

Listing 4: function scopes

```
function x() {
  var a = 1
  if(true) {
    var b = 2
  }

  console.log(a, b) // what do we get?
}

x()
```

We get `1 2` as the output!

Even though `b` was initialised in the block following the `if`, it was added to the scope of the function. This happens, because JavaScript - unlike other programming languages - does not create a new scope for a block, so the surrounding scope is the function, not the block.

1.4. Shadowing

Scopes are semi-isolated from each other. Let's see what that means:

Listing 5: Shadowing

```
function outer() {
  var a = 1
  var b = 2

  function inner() {
    a = 10
    var b = 20
    console.log(a, b)
  }

  inner()
  console.log(a, b)
}

outer()
```

The output is this:

```
10, 20
10, 2
```

Here we see an example of **shadowing** and **nested scopes**. Each function creates its own scope. A scope has access to all variables from its surrounding scopes - that's why we can change `a` from within `inner`.

However, scopes can create variables with the same name as variables in the surrounding scope. *These variables will be separate variables*. Thanks to this, we can have a local `b` variable inside `inner` that only exists within the `inner` scope and does not harm the `outer` variable `b`

1.5. Hoisting

Let's look at the following code:

Listing 6: Hoisting

```
var a = true
function x() {
  if(a === undefined) {
    var a = 10
  }
  console.log(a)
}

x()
console.log(a)
```

The output is `10` and `true`. But how did that happen? We initialised `a` in the global scope as `true`, so why was it `undefined` in the function?

This happens because the parser will go through a new scope once it is being created and define all variables used in the scope right at the beginning of the scope, so the previous coude is identical to the following:

```
var a = true
function x() {
  var a ①
  if(a === undefined) { ②
    a = 10 ③
  }
  console.log(a)
}

x()
console.log(a) ④
```

① The parser has moved the definition of `a` out of the `if` to the beginning of the scope.

② As the scope now has a shadowing variable `a` that is still `undefined`, we execute the `if` block.

③ The shadowing `a` is initialised with 10 for the scope.

④ Outside of the scope, `a` has remained intact as is usual when shadowing happens.

This process is called **hoisting**. As this can be difficult to see in larger codebases, it is recommended to define local variables at the beginning of a scope.

1.6. Closures

Let's see how scoping can be put to great use. Assume we have the following code:

```
for(var i=0; i<10; i++) {
  setTimeout(function() { ①
    console.log(i)
  }, 1000)
}
```

① This can be anything that is asynchronous, e.g. fetching data from the network and waiting for the response.

So our code runs ten times with `i` changing its value from 0 to 9 each time. Each time we tell the browser that, after one second, it should print the value of `i`.

But due to the fact that `i` is in the global scope and `for` is synchronous, the loop will finish before `setTimeout` can run the function. Luckily, scopes help us here:

```
function print(i) { ①
  setTimeout(function() {
    console.log(i) ②
  }, 100)
}

for(var i=0; i<10; i++) {
  print(i) ③
}
```

① Parameters such as `i` in this case belong to the new scope of the function.

② This is now using the shadowed `i` from the scope of `print`.

③ We pass the current value of `i` into the scope of the `print` function, which will shadow it.

Of course you can name the parameter for `print` anything you want, the principle stays the same. But I wanted to point out that shadowing allows us to make sure that even if the variable name is identical we will not get in conflict with the parent scope here.

We can also use scopes to hide variables from the outside:

```
function makeGreeting(message) { ①
  return function greet(name) { ②
    return message + name ③
  }
}

var welcome = makeGreeting('Hello there, ')
welcome('Alice') ④
welcome.message = 'Yo, ' ⑤
```

① This function takes a `message` parameter into its scope.

② It returns a `greet` function that has access to the parent scope and takes a `name` parameter.

③ The `greet` function uses the `message` from its parent scope and the `name` from its own.

④ We can pass in a name from the outside…

⑤ But we do not have access to the `message` from the scope of `makeGreeting` anymore!

This concept is called a `closure` as we use an intermediary scope to hold variables for an inner scope.

2. Week 2: Primitives and objects

The ECMAScript specification knows the following types of data:

- `undefined`
- `null`
- `Boolean` (`true` and `false`)
- `Number`
- `String`
- `Object`
- `Symbol` (since ECMAScript 6)

All of them *except for Object* are so-called **primitives**. Objects aren't primitives but a collection of properties (i.e. key-value pairs). For example:

```
var obj = {
  name: 'Bob', ①
  friends: [ ②
    { name: 'Alice' } ③
  ]
}
```

① Here `name` is the **key** and `'Bob'` is the associated value.

② Values can be complex, such as an Array.

③ Objects can also hold other objects.

There is an important difference between primitives and objects as the following sample shows:

```
var a = 1
var c = { x: 1 }

var b = a
var d = c

b++
d.x++

console.log(a, b, c.x, d.x)
```

1. We created two variables `a` with a primitive value and `c` with an object value.
2. We then created two more variables `b` and `d` and assigned them the values of `a` and `d`, respectively.
3. We then incremented the value of `b` and the value of the `x` property in `d`.

The output is `1 2 2 2` and not `1 2 1 2` as we might have expected. This happens because primitive values are being **copied** on assignment (`var b = a` copied the value) while objects are being **referenced** on assignment (`var d = c` points `d` to the object that `c` holds).

In order to avoid this, we can sometimes use the detour via `JSON.stringify` and `JSON.parse`:

```
var a = { x: 1 }
var b = JSON.parse(JSON.stringify(a))

b.x++
console.log(a.x, b.x)
```

3. Glossary