JavaScript

What is it?

He sent dispatches to all parts of the kingdom, to each province in its own *script* and to each people in their own *language*, proclaiming that every man should be ruler over his own household, using his native tongue. Esther 1:22 NIV

JavaScript is a programming language. Even though the name contains *Java*, it has little to do with this programming language.

It was developed by Brendan Eich (GITHub - https://github.com/BrendanEich, Blog - https://brendaneich.com/) and first introduced as JavaScript in December of 1995. Brendan Eich was working for Netscape at the time. Netscape's strategy had been that dynamic content in web pages should be delivered as Java Applets, but Brendan Eich felt strongly that a programming language for non-programmers was needed.

He co-founded the Mozilla Project - https://www.mozilla.org/ and was it's CEO until April of 2014. Mozilla is known for developing the Firefox browser.

Why should I care?

JavaScript is the programming language of a Chrome Application. It is JavaScript that is executed when we use the mouse or the keyboard to interact with our application.

JavaScript

Details

Loading from an HTML5 page

A JavaScript file is loaded from an HTML5 page using the `script` element.

```
<html>
  <head>
  </head>
  <body>
    <script type="text/javascript" src="main.js"></script>
  </body>
</html>
```

The `type` attribute indicates the type of script. It is necessary to specify this because browsers support other script languages. Internet Explorer for example has its own types; JScript, and VBScript.
The `src` attribute gives the name of the file that contains the JavaScript source code.

Statements

Statements describe things to do. Each statement is ended with a semi-colon (`;`). Multiple statements can be combined and enclosed in a set of curly brackets `{}` called a block statement. A block statement is not ended with a semi-colon.

JavaScript

```
statement1;
statement2;
{
  statement3;
  statement4;
}
```

Please note that block statements do not provide a scope. In other languages such as Java, something inside curly brackets are considered private to that area, but this is not the case in JavaScript.

Variable Statement

Variables must be declared with the `var` keyword followed by the name of the variable. You can optionally also assign a value when you declare variables. Once declared, the `var` keyword must not be used.

```
var some;
var other = 5;
other = 6;
```

Names of variables must not be one of the keywords or built-in global variables.

If...else Statement

The if...else statement is used to execute some statements if a condition is `true` and something else if it is `false`.

```
var a = 5;
if(a < 6) {
  statement1;
  statement2;
} else {
  statement3;
  statement4;
}
```

Since the variable `a` has the value `5` which is less than `6`, statements `statement1` and`statement2` will be executed.
The following values are considered `false`:

> `undefined`, `null`, `0`, `NaN`, `""`

Everything else is considered `true`.

Everything that you have not yet defined will have the value `undefined`. The value `null` is different from `undefined`, but it is a subtle difference. The value `null` means *nothing*.

The value `NaN` is something that is *Not a Number*. It will be the result of an impossible calculation, like something divided by zero.

The pair of double-quotes means an empty string. A string is a set of characters.

For Statement

The for statement creates a loop. The statement takes
three expressions separated by semicolon enclosed in parentheses followed by the statement to execute.

JavaScript

The first expression is the initial statement to execute.

Then comes a condition that has to be `true` in order for the loop to continue. The final expression is the statement to execute after each iteration of the loop.

```
for(initialStatement; condition; afterIteration)
  loopStatement
```

So the order of execution is:

1. initialStatement
2. condition
3. loopStatement
4. afterIteration
5. condition

Steps 3-5 will be repeated as long as the condition is `true`. If the condition is initially `false`, they will not be executed at all.

Example:

```
for(var i=1; i<10; i++) {
  console.log(i);
}
```

Outputs:

```
1
2
3
4
```

JavaScript

The `console` is a global object that can output information to the browser's log.
To see the log, use the JavaScript Console in Chrome browser developer tools. You open the console by pressing <Ctrl+Shift+J>.

Operators

The **assignment operator** `=` is used to give a name to something. The name will be on the left side, and the thing to name will be on the right side of the equal sign. We have already seen the assignment operator used in the variable statement.
You can prefix the assignment operator with another operator such as `+` as a shorthand:

```
var a = 1;
a += 5;
console.log(a);
```

The console output will be 6. `a += 5` is short for `a = a + 5`.

Arithmetic operators are plus (`+`), minus (`-`), multiply (`*`), divide (`/`), and remainder (`%`). You should be familiar with the first 4, but maybe not with remainder, so we will give that one a little more attention.

JavaScript

The remainder operator is used to find what is left after division. If you divide `9` by `2`, the result is `4` with a remainder of `1`. That is why we say the result is `4` and `1/2`. The remainder becomes the numerator and the number we originally divided with; `2` becomes the denominator of the remaining fraction. Example:

```
var a = 9;
var remainder = 9 % 2;
console.log(remainder);
```

The console output becomes `1`.
This is very convenient to find out if a number is an even or an odd number for instance. Example:

```
var a = 9;
if(a % 2) {
  console.log(a + " is an odd number");
} else {
  console.log(a + " is an even number");
}
```

Remember that `0` is considered `false`.

Functions

Functions is a way of grouping statements together and giving the group a name. This is convenient if you want to execute the same statements from different areas of your application.

A function can be created in two different ways. Either as an expression or a declaration.

Expression:

```
var log = function(argument) {
  console.log(argument);
}
```

Declaration:

```
function log(argument) {
  console.log(argument);
}
```

Objects

An object in JavaScript is a set of functions and set of variables grouped together with a name. We say that the functions defines the behavior, and the variables define the state of the object. The variables in an object are called *properties*, and the functions are called *methods*.

One special thing in JavaScript that is different from many other languages is that a function is also an object. This means that you do cool stuff like passing a function as an argument to a function, or return it as the result of a function.

To create an object, you can just define it literally like this:

```
var person = {};
```

You can then add properties and methods to your object like this:

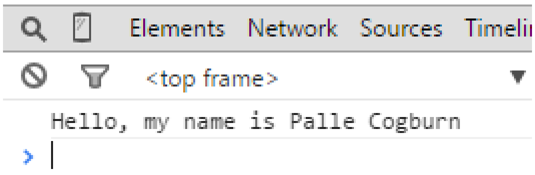
JavaScript

```
person.name = "Palle Cogburn";
person.speak = function() {
  console.log("Hello, my name is " + this.name);
}
```

Then to call the `speak` function on the person object, you will write this:

```
person.speak();
```

Did you notice that I wrote `this.name` in the `speak` function? In this case, `this` references the current object, so what is actually output to the `console` looks like this:

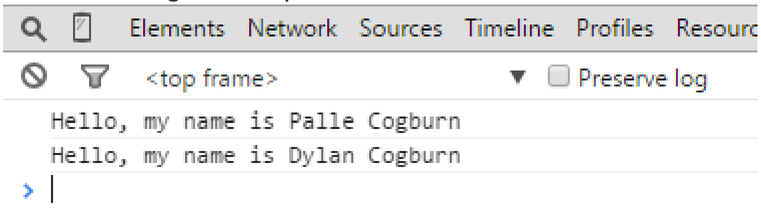


```
Hello, my name is Palle Cogburn
```

What if you want to create two people? You can create a function that returns an object like this:

```
var person = function(name) {
  var that = {
    name: name,
    speak: function() {
      console.log("Hello, my name is " + that.name);
    }
  };
  return that;
}
var person1 = person("Palle Cogburn");
var person2 = person("Dylan Cogburn");
person1.speak();
person2.speak();
```

The following is now printed to the console:



```
Hello, my name is Palle Cogburn
Hello, my name is Dylan Cogburn
```

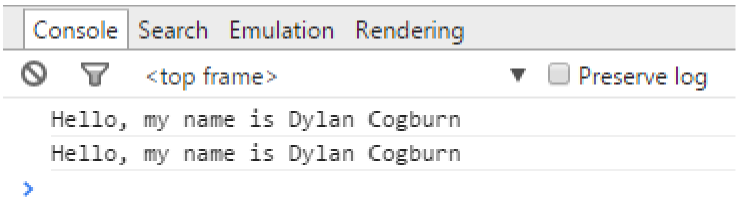
We specified the properties and methods as a comma-separated list of name, colon, and value.

Notice how I used `that` instead of `this`. In some cases `this` refers to the global scope, which makes it dangerous and it should be used with caution, or not at all.

If I had written the code like this:

```
var person = function(name) {
  this.name = name;
  this.speak = function() {
    console.log("Hello, my name is " + this.name);
  }
  return this;
}
var person1 = person("Palle Cogburn");
var person2 = person("Dylan Cogburn");
person1.speak();
person2.speak();
```

The result would have been:



```
Hello, my name is Dylan Cogburn
Hello, my name is Dylan Cogburn
```

It is also possible to use the keyword `new` in front of a function call that returns an object like this:

```
var Person = function(name) {
  var that = {
    name: name,
    speak: function() {
      console.log("Hello, my name is " + that.name);
    }
  };
  return that;
}
var person1 = new Person("Palle Cogburn");
var person2 = new Person("Dylan Cogburn");
person1.speak();
person2.speak();
```

By convention the function name now starts with an upper-case letter.

Amazingly when we use the `new` keyword, we could have written the code like this:

```
var Person = function(name) {
  this.name = name;
  this.speak = function() {
      console.log("Hello, my name is " + this.name);
  }
}
```

JavaScript

```
var person1 = new Person("Palle Cogburn");
var person2 = new Person("Dylan Cogburn");
person1.speak();
person2.speak();
```

That works fine! The value of `this` is now the object being created, and not the global object. As I said, it is dangerous and confusing to use `this`. Just don't do it, it is not necessary. You need to know about this though in order to read and understand other peoples code.

Built-in Objects

Some objects are built into the language and belong to the global scope. One of these is Array.

An array is a list of things basically.

An Array object can be constructed literally like this:

```
var months = [undefined
, "January", "February", "March"
, "April", "May", "June"
, "July", "August", "September"
, "October", "November", "December"];
```

You can also use the `new` keyword like this:

```
var months = new Array(undefined
, "January", "February", "March"
, "April", "May", "June"
, "July", "August", "September"
, "October", "November", "December");
```

JavaScript

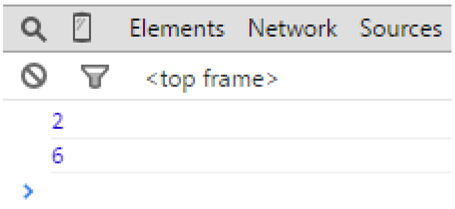
An Array has a `length` property that contains the size of the Array, which is not necessarily the number of objects in the Array.

You can assign a value to a particular index in the Array like this:

```
months[12] = "December; the month of Christmas";
```

If you assign a value to an index that is higher than the currently highest index, the Array `length` increases and the Array is expanded to include all the elements up to the new highest index.

```
var students = ["Peter", "John"];
console.log(students.length);
students[5] = "Matthew";
console.log(students.length);
```

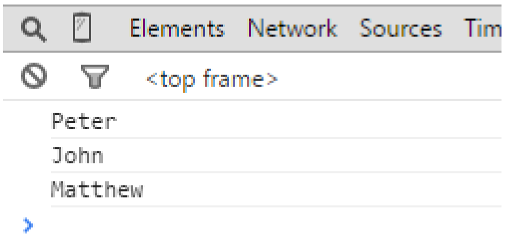


```
2
6
```

When I said before that the size of the Array was not necessarily the same as the number of objects in the Array it is because the indexes that have not been assigned now contains `undefined` which is not an object.

Since `undefined` is one of the values that are considered `false`, you can write code such as this:

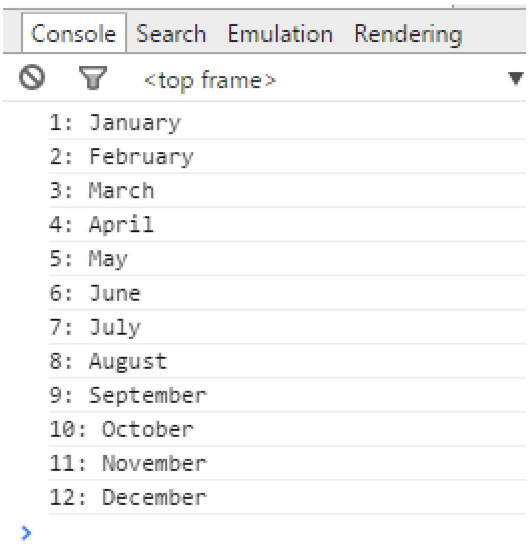
```
var students = ["Peter", "John"];
students[5] = "Matthew";
for(var i=0; i<students.length; i++) {
  var student = students[i];
  if(student) {
    console.log(student);
  }
}
```



```
Peter
John
Matthew
```

Did you notice in the months array earlier, I assigned `undefined` to the first index. Indexes into an Array start with `0`. This Array is convenient to convert between month number and name like this:

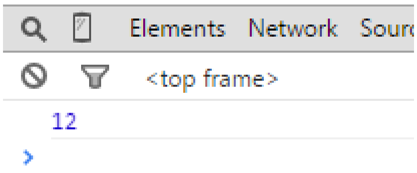
```
for(var i=1; i<months.length; i++) {
  console.log(i + ": " + months[i]);
}
```



```
1: January
2: February
3: March
4: April
5: May
6: June
7: July
8: August
9: September
10: October
11: November
12: December
```

When you create an Array, if there is only one argument and that is a Number, it indicates the size of the Array, and not the first element, but only if you use the `new` keyword to construct it like this:

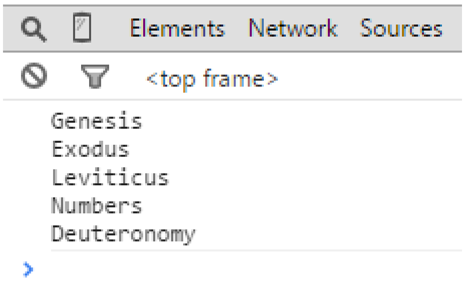
```
var months = new Array(12);
console.log(months.length);
```



```
12
```

An Array also has methods. One of these methods is `join`. It can be used to take all values in the array and put them together like this:

```
var pentateuch = ["Genesis", "Exodus", "Leviticus", "Numbers",
"Deuteronomy"];
console.log(pentateuch.join("\n"));
```



```
Genesis
Exodus
Leviticus
Numbers
Deuteronomy
```

The argument to the `join` method is the string to *glue* the array elements together with. The special string “\n” is a line-break.

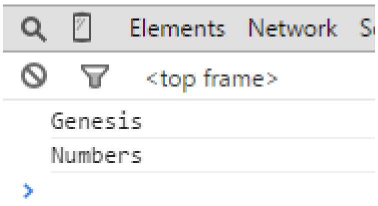
Other interesting methods of the Array object are: `pop`, `push`, `shift`, `concat`, `contains`. You can find the full description of the Array object here:

JavaScript

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array.

To create an array from a `String`, you can use the `split` method of the `String` object like this:

```
var pentateuch = "Genesis, Exodus, Leviticus, Numbers, Deuteronomy";
var pentateuchArray = pentateuch.split(", ");
console.log(pentateuchArray[0]);
console.log(pentateuchArray[3]);
```



```
Genesis
Numbers
```

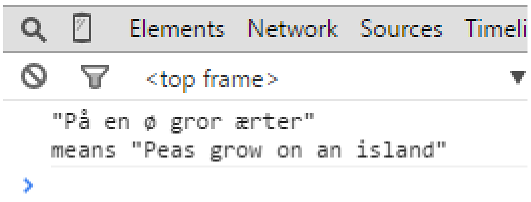
The argument to the `split` method is the `String` to match for to separate the elements.

You can create a `String` literally as a number of characters enclosed in either a set of double-quotes or single-quotes like this:

```
var name = "Palle Cogburn";
var job = 'Software Developer';
```

Special characters in a `String` can be encoded by setting a back-slash in front like this:

```
var danish = "P\u00E5 en \u00F8 gror \u04D5rter";
var english = "Peas grow on an island";
console.log("\""+danish+"\" \nmeans \""+english+"\"");
```

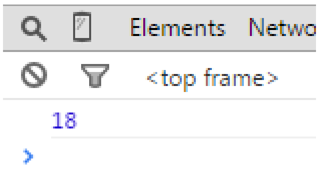


```
"På en ø gror ærter"
means "Peas grow on an island"
```

In this way you can write characters like double-quotes with `\"`, line-breaks with `\n`, and even characters that are not part of the English alphabet with `\u`. The 4 characters following `\u` are the unicode hexadecimal number. You can find all the defined characters here: http://www.unicode.org/charts.
These sequences of characters starting with a back-slash are called *escape sequences*.

Like an Array a String object has a `length` property. It indicates the number of characters in the String. Note that this excludes escaped sequences like `\u00F8`:

```
var danish = "P\u00E5 en \u00F8 gror \u04D5rter";
console.log(danish.length);
```



```
18
```

You can find the full description of the String object here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String.

JavaScript

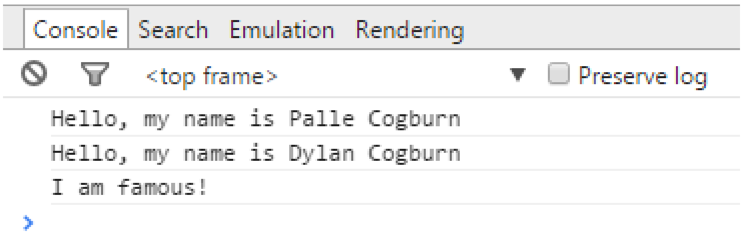
Inheritance

Inheritance is used when you have an object, and you want to change it just a little. You will be creating a new object that has all the behavior and state of an existing one, but maybe one of the methods should do something different. The functions of an object are called methods, and define the behavior. The variables in an object are called properties and defined the state.

Let's imagine that we want to create an object for a famous person. A famous person does also have a `speak` method, but they will speak more boastfully. Here is how you could do it:

```
var person = function(name) {
  var that = {
    name: name,
    speak: function() {
      console.log("Hello, my name is " + that.name);
    }
  }
  return that;
}
var famousPerson = function(name) {
  var that = person(name);
  var base = {
    speak: that.speak
  }
  that.speak = function() {
    base.speak();
    console.log("I am famous!");
  }
  return that;
}
var person1 = person("Palle Cogburn");
var person2 = famousPerson("Dylan Cogburn");
person1.speak();
person2.speak();
```

The result looks like this:

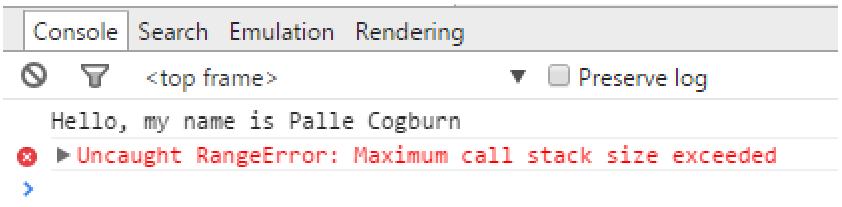


```
Hello, my name is Palle Cogburn
Hello, my name is Dylan Cogburn
I am famous!
```

Notice how `base.speak` is assigned the `that.speak` value. This means that it is a for the speak method in `that`. When `base.speak` is called inside the `speak` method of `that`, it calls this method. You might think it could have been written like this instead:

```
  that.speak = function() {
    that.speak();
    console.log("I am famous!");
  }
```

The result would have been this:



```
Hello, my name is Palle Cogburn
▶ Uncaught RangeError: Maximum call stack size exceeded
```

It is because it is actually calling itself instead of the `base` object. Since it can never get out, it throws an error.

The type of inheritance described here is called functional inheritance. There are other types of inheritance you can do in JavaScript, but they are generally more confusing and dangerous.

Name Collision

Your JavaScript code might just be a part of all the code running on one web page. It is therefore likely that you would have named some of your functions the same as something else. This will not work, because a declaration is only allowed once.

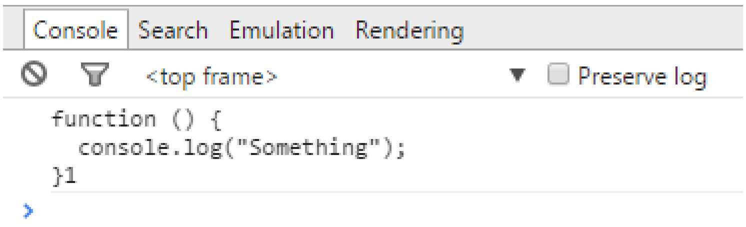
```
var something = 1;

var something = function() {
  console.log("Something");
}

var result = something + 1;
console.log(result);
```

Imagine that one script file will set `something` to the value `1`. Another script file will define `something` as a function and the first file comes along and adds `1` to `something` and prints out the result. The output will be this, which is not what was expected:

JavaScript



```
function () {
  console.log("Something");
}1
```

You can avoid this by putting your code inside an anonymous function declaration like this:

```
(function() {
  // You code goes here
}());
```

Since the function doesn't have a name, there is no other code that can get to it.

Assignments

Look at the following code:

```
(function(){
  var a = 1;
  var b = a;
  b = 2;
  console.log(a);
}());
```

First the variable `a` is assigned the value `1`, then the variable `b` is assigned the value of `a`. We then assign the value `2` to `b`, and output `a`.

The console output is `1`.

JavaScript

Now look at this code:

```
(function(){
  var a = "1";
  var b = a;
  b = "2";
  console.log(a);
}());
```

First variable `a` gets assigned the `String` of `1`, the variable `b` gets assigned the value of `a`. We then assign the value `2` to `b`, and output `a`.

The console output is still `1`.

When you assign a value to a variable, you are really just giving the value a name, or a reference. When we assign the value of `a` to `b`, we are really just saying that `b` is another name for `a`. When we then assign the value `2` to variable `b`, we are just saying that now `b` is the name for `2`.

Now look at this code:

```
(function(){
  var a = {value: 1};
  var b = a;
  b.value = 2;
  console.log(a.value);
}());
```

Now the console output becomes `2`. Why?

Well, when we assign `a` to `b`, `b` becomes another name for `a`, so when we assign `2` to`b.value` it is just another way of saying that `a.value` is `2`.

Both `a` and `b` are just alternative names or references to the object that was initialized as `{value: 1}`.

JavaScript

Even experienced programmer will tell you that there is a difference between assigning primitive values such as `1` and assigning object values such as `{value: 1}`, but there isn't. Assignments in JavaScript are always by reference.

If we actually wanted `b` to be referencing another object than `a`, but with the same state and behavior, we can use the `create` method of `Object` like this:

```
(function(){
  var a = {value: 1};
  var b = Object.create(a);
  b.value = 2;
  console.log(a.value);
}());
```

Now the console output becomes `1`.

Equality

Sometimes you need to find out if two variables have the same value, or I should say whether they both reference something that has the same value. Look at this code:

```
(function(){
  var a = {value: 1};
  var b = a;
  b.value = 2;
  if(a === b) {
    console.log("a and b are equal");
  } else {
    console.log("a and b are different");
  }
}());
```

We can use `===` to find out if two variables both reference something that is equal. Now we need to be careful, because there really is a difference here between value types and reference types.

Look at this code:

```
(function(){
  var a = 1;
  var b = 1;
  if(a === b) {
    console.log("a and b are equal");
  } else {
    console.log("a and b are different");
  }
}());
```

The console output is `a and b are equal`. But we just learned that `a` and `b` in this case are referencing two different things. They just happen to have the same value.

This is because numbers, strings, and booleans are considered value types when they are compared for equality. They are still assigned by reference.

Let's now compare two objects for equality:

```
(function(){
  var a = {value: 1};
  var b = {value: 1};
  if(a === b) {
    console.log("a and b are equal");
  } else {
    console.log("a and b are different");
  }
}());
```

The console output now says `a and b are different`.

This is quite confusing, and it can lead to errors in your code, if you don't understand this. Make sure you get this! Make up your own examples, and test them out.

The explanation is that `a` and `b` are references to two different objects that happen to have the same state and behavior. Objects other than numbers, strings, and booleans are compared by reference, not value.

Since `a` and `b` are referencing two different objects, they are considered to be not equal.

But what if you really wanted to know if the objects have the same state and behavior? Well, you would have to examine them more closely. Look at this example:

```
(function(){
  var a = {value: 1};
  var b = {value: 1};
  for(var property in a) {
    if(a[property] === b[property]) {
      console.log("a." + property + " and b." + property + " are
equal");
    } else {
      console.log("a." + property + " and b." + property + " are
different");
    }
  }
}());
```

The console output will now be

```
 a.value and b.value are equal.
```

Now since properties of objects can be objects also, this is not enough to compare any two objects. Look at this example:

```
(function(){
  var a = {value: 1, other: {value: 1}};
  var b = {value: 1, other: {value: 1}};
  for(var property in a) {
    if(a[property] === b[property]) {
      console.log("a." + property + " and b." + property + " are
equal");
    } else {
      console.log("a." + property + " and b." + property + " are
different");
    }
  }
}());
```

The variables `a` and `b` still look like they have exactly the same state, but the console output says:

```
a.value and b.value are equal

a.other and b.other are different
```

We can use something called *recursion* to solve this problem. Look at this example:

```
if(!NSMscsbend) {
  var NSMscsbend = {};
}
```

JavaScript

```
NSMscsbend.equals = function(one, other) {
  var equals = function(one, other) {
    console.log("Comparing types");
    if(typeof(one) !== typeof(other)) {
      return false;
    }
    console.log("Comparing properties");
    var property;
    for(property in one) {
      console.log("Comparing " + property + " " + one[property] +
        " with " + other[property]);
      if(other[property] === undefined) {
        console.log("Missing " + property);
        return false;
      }
      if(typeof(one[property]) !== typeof(other[property])) {
        return false;
      }
      if(typeof(one[property]) === "function") {
        console.log("We are only comparing state, not behavior");
        continue;
      }
      if(typeof(one[property]) === "object") {
        if(!equals(one[property], other[property]) ||
          !equals(other[property], one[property])) {
            return false;
        }
        continue;
      }
      if(one[property] !== other[property]) {
        return false;
      }
    }
    return true;
  }
  return equals(one, other) && equals(other, one);
};

(function(){
  var a = {value: 1, other: {value: 2}, something: function(){}};
  var b = {value: 1, other: {value: 2}, something: function(){}};
  if(NSMscsbend.equals(a, b)) {
    console.log("a and b are equal");
  } else {
    console.log("a and b are different");
  }
}());
```

Notice how there is an `equals` function inside the `NSMscsbend.equals` function. When we come across a property that is an object, we have to compare those two objects considering all their properties, so we call back into the same function.

If you need to compare two objects for equality, please consider if that is *really* what you need, or if your code can be written in a simpler way, it usually can. When *testing* your code it can be useful to know if two objects have the same state. We will look more at testing in the next chapter.

Where can I find more information?

Mozilla JavaScript Reference - https://developer.mozilla.org/en-

JavaScript

US/docs/Web/JavaScript/Reference