Chapter 6, JavaScript Containers

John M. Morrison

January 21, 2019

Contents

0 Introduction

Have you ever had a nocturnal itch to store a bunch of related items together? F'rinstance, if you have a sock drawer in your dresser, you can pull out a (hopefully clean) sock out of the drawer without undue rooting around in, or possibly under your dresser amongst the growling dust kittens? At another level of organization, you might even pair matching socks together when are finished laundering them so you can find pairs easily as you stumble about in the morning from a lack of sleep!

It is often a useful idea to store a group of related things in one place. Your dresser has drawers; hopefully you actually use them. If you do, you likely keep socks in one drawer, underwear in another (or in another part of the sock

drawer), shirts in another, etc. We can do the same sort of organizing of data in JavaScript: this will be accomplished with two new types, arrays and objects. These are both examples of *data structures*, which are containers that store related pieces of data under a single name.

We have seen how to store a glob of text in a single place; to do this we use a string. A string is a smart character sequence the knows its characters and which can perform tasks based on the characters it contains.

1 Arrays

We will begin our study of data structures with the array. Shortly, you will see that, like strings, arrays come with useful methods that allow us to easily accomplish a wide array of useful chores.

We begin by showing how to make an array. There are two principal ways. The code

```
var foo = [];
```

creates an empty array. The code

```
var stuff = ["Moe", "Larry", "Joe", 5.6, true];
```

creates an array holding the five objects listed. Notice that you must enclose the contents of an array in square brackets. Open a console session now and we will demonstrate how to work with arrays. Begin by entering the two examples cited here.

```
> var foo = [];
undefined
> var stuff = ["Moe", "Larry", "Joe", 5.6, true];
undefined
```

Do not worry about the `undefined`s. Assignment of an array is actually a function that does not return anything explicitly, so it just returns an`undefined`.

Now let us stuff `stuff` with some stuff.

```
> stuff.push(1)
6
> stuff.push("cows")
7
> stuff
["Moe", "Larry", "Joe", 5.6, true, 1, "cows"]
```

Observe that the original array had 5 elements. When we pushed `1` onto this array, the `1` got placed on the end of the array and the new size, 6, was returned. The same thing happened when we added `"cows"` to our array. From this you can quickly deduce that arrays are mutable objects; i.e, that it is possible to change the state of an array you have created. The state of the array is just the sequence of objects it is pointing at.

So the array method `push` places its argument on the end of the array, makes the array one bigger, and returns the new, larger size. Now let us drink in `pop`; this function needs no argument.

```
> stuff.pop()
"cows"
> stuff.pop()
1
> stuff
["Moe", "Larry", "Joe", 5.6, true]
```

The `pop` method removes the last item in the array and then it returns it. What happens if we try to pop from an empty array?

```
> var empty = []
undefined
> empty.pop()
undefined
```

It just returns an undefined and it has no additional effect. It is always smart to check prior to popping so you do not from an empty array. That can cause unexpected annoyances. You can use conditional logic to prevent popping from an empty array.

Arrays know their size and they provide access to their entries, just as strings do. We now demonstrate this.

```
> stuff.length
5
> stuff[0]
"Moe"
> stuff[1]
"Larry"
```

The square-brackets operator provides access to array entries. The `length` property tells you the size of your array. Now observe this nifty trick with a `for` loop.

```
> for(var k = 0; k < stuff.length; k++){console.log(stuff[k]);}
Moe
```

```
Larry
Joe
5.6
true
undefined
```

We have caused an array to list out its contents.

Can we concatenate arrays? Let's try with a +!

```
> [1,2,3] + [4,5,6]
"1,2,34,5,6"
```

Ooh, bitter disappointment. What did rotten old JavaScript do? JavaScript is a stringophile. It said, "Hey, + is great for concatenating strings and I looooooove strings, so I will just convert the operands to strings and concatenate." Crash and burn.

Now try this.

```
> [1,2,3].concat([4,5,6])
[1, 2, 3, 4, 5, 6]
```

Use `concat` to concatenate arrays. Don't confuse it with the telecom we all hate.

While we are here, let us take a moment to discuss `toString()`. Builtin JavaScript objects have a `toString` method. First we show `toString` at work on numbers.

If you pass an arugment to a number's `toString` method, it will give you a string representation for that number in that base. The default, of course, is 10.

```
> var num = 216
undefined
> num.toString()
"216"
> num.toString(16)
"d8"
> num.toString(8)
"330"
> num.toString(2)
"11011000"
```

Squirrely note: You can't use this method on a numerical literal but you can use it via a variable.

We now show `toString` working on an array.

```
> stuff.toString()
"Moe,Larry,Joe,5.6,true"
```

It's what we expect.

The `split` method can be useful for breaking input into pieces.

```
> let x =  "This is a test";
undefined
> x.split(" ")
(4) ["This", "is", "a", "test"]
```

You now have easy access to the individual words in the phrase. You can split on any character or string. This method always returns an array of strings. You will see below that some of these strings can be empty.

```
> let y = "ratatouille"
undefined
> y.split("a")
(3) ["r", "t", "touille"]
> y.split("e")
(2) ["ratatouill", ""]
```

Programming Exercises For these problems, visit `http://www.w3schools.com/jsref/jsref_obj_array.asp` While you are there see if there are other methods you might find handy.

1. Use `fill()` to zero out an array.
2. If you have an array of numbers, how do you find the first number in the array whose value is 5? How about the last index?
3. How do you figure out if an array of numbers contains a 5?
4. What do the methods `shift` and `unshift` do to an array?
5. Write a function named `spew` that takes an array as an argument, prints out all of its entries, and that leaves the array empty at the end. Can you do this printing in regular and reverse orders?
6. The *median* of a list of numbers is computed as follows. You sort the list. Then, if the list has odd length, the median is the middle number in the list. If the list is of even length,the median is the average of the middle two numbers. Write a function named `median` that, when given a numerical array, computes the median of the numbers in the array.
7. Write a function named `range(a,b)` which returns an array of numbers `[a, a + 1, a + 2, ....  b - 1]` If `a >= b`, just return an empty array. Here some examples of `range`'s action.

```
range(0, 5) -> [0, 1, 2, 3, 4]
range(3,2) -> []
range(2,3) -> [2]
```

8. Write a function named `explode` that takes a string and "explodes" it into an array of one-character strings. Here are examples of its action.

   ```
   explode("foo") -> ["f", "o", "o"]
   explode("") -> []
   explode("cats") -> ["c", "a", "t", "s"]
   ```

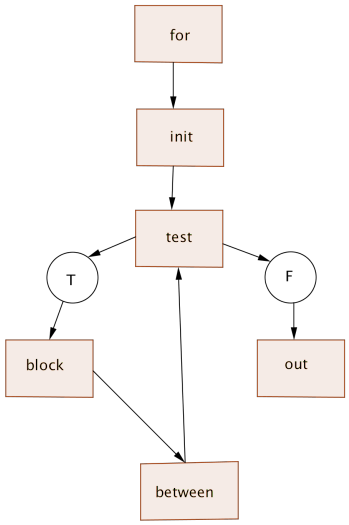
9. Create an array and then convert it to a string by concatenating it with an empty string. Then split on the character `","`. What happens?

2 A new loop: the `for` loop

This loop comes in several flavors. We will first look at the one that most resembles the `while` loop, the C-style `for` loop. The loop looks like this.

```
for(init; test; between)
{
    block;
}
out;
```

The text `block` stands for the block of code and `out` stands for the first (possibly empty) statement beyond the loop. In this diagram, we show the action of the `for` loop.



Here is what the diagram is saying. When the `for` loop is first encountered, the `for` loop is first encountered, the `init` runs. This can have more than one statement; if so just separate with a comma and not a semicolon. The `init` code is never seen again. The `test` is a predicate. If the predicate is true, you progress from `test` to `block` and the `block` executes. If it evaluates to false, you go to `out` and it is all over.

Once the block executes, `between` executes and you go back to the `test`. You have the progression `test`, `block`, `between` until the test fails (`test` is false). At that time, you go to `out` and it is all over.

One especially useful role for the `for` loop is for dealing with collections of objects. We shall now turn to studying this.

Style Point When creating a `for` loop, use `let` in the initializer if you don not need the loop variable after the loop ends. If you do want the loop variable after the loop ends, declare it just before the loop and it will be visible inside of the loop's enclosing block.

Programming Exercises

1. Convert this pseudocode for a C style for loop into a `while` loop

```
for(init; test; between)
{
    block;
```

```
}
out
```

2. Use the `for` loop to write a function named `range(a,b)` which returns an array of numbers `[a, a + 1, a + 2, .... b - 1]` If `a >= b`, just return an empty array. Here some examples of `range`'s action.

   ```
   range(0, 5) -> [0, 1, 2, 3, 4]
   range(3,2) -> []
   range(2,3) -> [2]
   ```

3. Use the `for` loop to write a function named `explode` that takes a string and "explodes" it into an array of one-character strings. Here are examples of its action.

   ```
   explode("foo") -> ["f", "o", "o"]
   explode("") -> []
   explode("cats") -> ["c", "a", "t", "s"]
   ```

3 Two more `for` loops

One must be careful in using the C-style `for` loop when traversing (visiting every element) in an array. You must pay attention to the end of the array, or the result of your computation could be gibberish. Let's build a little array.

```
> let x = []
undefined
> x.push(1);
1
> x.push(2);
2
> x.push(3);
3
> x.push(4);
4
> x.push(5);
5
x
(5) [1, 2, 3, 4, 5]
```

Now watch this.

```
>  for(let k in x)
   {
       console.log(k*k);
```

```
   }
0
1
4
9
16
undefined
> k
VM157:1 Uncaught ReferenceError: k is not defined
    at <anonymous>:1:1
(anonymous) @ VM157:1
> for(let k of x)
{
    console.log(k*k);
}
1
4
9
16
25
undefined
```

So, what happened? The `for-in` loop gave us each of the indices of the array in succession. The `for-of` loop gave us each of the values stored in the array in index order. Notice that the loop variable is out of scope once the loop is over.

Now observe this.

```
for(let k in x)
{
    k = 0;
}
0
x
(5) [1, 2, 3, 4, 5]
for(let k of x)
{
    k = 0;
}
0
x
(5) [1, 2, 3, 4, 5]
```

In neither event did the array change. This is because you are shown *copies* of the items you are iterating through, not the originals. To add to this hubbub, now see this.

```
> for(let k in x)
{
    x[k] = 0;
}
0
> x
(5) [0, 0, 0, 0, 0]
```

This might seem a paradox, but it is not. You are getting copies of the indices of the array, then using them to access the actual array. Hence, the array gets zeroed out.

4 Selecting Elements on a Page using Nodelists

So far, we have used the method `document.getElementById` to select a single element from a page by its `id`. Elements with `id`s can be selected using CSS or JavaScript.

CSS can select elements by tag type or by class. Can we make JavaScript do the same thing? Happily, yes.

Recall that the type of object returned by `document.getElementById` is called a *node*. The entire document is a node. So is every element. Four types of nodes exist. They include

- All attributes live in attribute nodes.
- All text lives in text nodes.
- Comments live in comment nodes.
- Element nodes are just elements. They contain attribute and text nodes.

We will deal largely with element nodes.

The `docuement` object offers three very useful functions, `getElementsbyTagName`, `getElementsbyClassName`, and `querySelectorAll`. These items return an object called a *nodelist*. This is like an array in that it provides read entry access and it knows its length. Nodelists do not have any other array properties or methods. You can iterate through them with a `for` loop.

Warning, Will Robinson! Note the presence of the plural `s` in `getElementsByTagName` and `getElementsbyClassName`. Omitting that letter causes vexatious errors.

We now create a JavaScript function that changes colors of elements by tag type. Place this code in `getByType.js`

```
function changeColor(type, color)
{
    var handle = document.getElementsByTagName(type);
    for(let elem of handle)
    {
        elem.style.backgroundColor = color;
    }
}
```

We can loop through the node list returned to us by `document.getElementsByTagName` and change the background color to each element to the color we specify. Now create this HTML document. Notice how we are using an `onload` attribute to delay the running of JavaScript code until the document is loaded and all of the elements it specifies exist.

```
<!doctype html>
<html>
<head>
<title>getByType</title>
<script type="text/javascript" src="getByType.js">
</script>
</head>
<body onload="changeColor('li', 'green');">
<p>Here is a couple of lists.</p>
<ul>
    <li>one</li>
    <li>two</li>
    <li>three</li>
</ul>

<p>and</p>

<ul>
    <li>one</li>
    <li>two</li>
    <li>three</li>
</ul>

</body>
</html>
```

This will turn all list items green. Now try adding these to the body's `onload` attribute.

```
changeColor('p', 'red');
```

```
changeColor('body', 'yellow');
```

Now let us do a similar thing by class. Here is the HTML file we will use

```
<!doctype html>
<html>
<head>
<title>getByType</title>
<script type="text/javascript" src="getByClass.js">
</script>
</head>
<body onload = "changeClassColor('foo', 'blue');">
<p class = "foo">Here is a couple of lists.</p>
<ul>
<li class = "foo">one</li>
<li>two</li>
<li>three</li>
</ul>
<p class = "foo">and</p>
<ul>
<li>one</li>
<li>two</li>
<li>three</li>
</ul>

</body>
</html>
```

Now for the JavaScript.

```
function changeClassColor(className, color)
{
    var handle = document.getElementsByClassName(className);
    for(let elem of handle)
    {
        elem.style.backgroundColor = color;
    }
}
```

Load and run. These turn all elements marked with class `foo` blue. You should try adding both JavaScript files shown here to your page and see how calling these and changing order affects the page's appearance.

This nodelist thing is cool! Can I select other nodelists from my pages? Happily yes! Suppose you want to get all elements that are a list item inside of an ordered list. The CSS selector for this is `ol li`.

To select by a CSS selector, usse `document.querySelectorAll`. In this instance use `document.querySelectAll("ol li")`. To select all paragraphs with class `"foo"`, use `document.querySelectorAll("p.foo")`. As an exercise, go into the previous two examples and try turning all list items inside of an ordered list orange and turning all paragraphs with class `foo` cyan.

Programming Exercises To do these, visit this refrence page, `http://www.w3schools.com/jsref/dom_obj_document.asp`.

1. How can you print out the URL of the `document`?
2. How can you print out the title of the `document`?
3. How can you walk through all links in the `document`?

5 Creating JavaScript Objects

JavaScript has a data structure called an *object*; these allow you to package data (properties) and functions (methods) into a single package of code. JavaScript provides a variety of ways to create objects; we will look at the simplest ones first.

One way to create objects is to stick-build them right at the scene. Let us demonstrate with a little console session. Line numbers have been added for convenience here

```
1       > obj = {}
2       Object {}
3       > obj.x = 3
4       3
5       > obj.y = 4
6       4
7       > obj
8       Object {x: 3, y: 4}
```

On line 1, we create an empty object. The console echoes back on line 2 that we, in fact, have an empty object. On lines 3 and 5, we give properties `x` and `y` to our object. On line 8, we see that we now have an object with property `x` set to 3 and `y` set to 4.

You can also create an object with an object literal in this manner.

```
other = {x:3, y:4};
Object {x: 3, y: 4}
```

You can even attach a function to an object as follows

```
obj.toOrigin = function(){return Math.hypot(this.x, this.y);}
```

Now call this function.

```
obj.toOrigin()
5
```

Here is something to puzzle over.

```
other == obj
false
```

This is so because `==` tests for equality of identity of objects, not equality of value. The two objects hold the same values, but they are separate copies, so they are not the same object.

Notice that, if we want to create another object like this one, that we must repeat the entire stick-building process or we must create another object literal to do the job.

For example, we might want an object to represent a point being clicked on on a canvas. We could return an object literal such as `{x: 3, y: 4}`, but it might be nicer if we knew this was an actual point object, which we could document and give some handy methods as well.

What if our object is more complex than just being a container for two numbers and a single function (method)? What if we are using this object frequently, leaving duplicate code everywhere?

Then answer to this problem is ECMA6 classes. Let us see what a class for points in the plane might look like. Let us begin by creating an HTML file so we can open our JavaScript code in the console.

```
<!DOCTYPE HTML>
<html>
   <head>
       <title>Class Test</title>
       <meta charset='utf-8'/>
       <script type="text/javascript"  src="objects.js"></script>
   </head>
   <body>
```

```
        <h2>Class Test</h2>
    </body>
</html>
```

We now introduce two new keywords, `class`, and `constructor`. A class is a blueprint for the creation of objects. So, let us begin by creating a very simple class for points.

```
class Point
{
    constructor(x,y)
    {
        this.x = x;
        this.y = y;
    }
}
```

We now inspect this in the console.

```
> var p = new Point(3,4);
undefined
> p
Point {x: 3, y: 4}
> typeof p
"object"
```

So, we can stamp out as many objects of this new type as we like by making the call `new Point(a,b)`. When we do this, the following chain of events occurs.

1. A new, empty object is created.
2. The variable `this` is set to this new, empty object.
3. The function `constructor` is called. Its purpose is to give our object properties (state).

`Point`s are born with the `x` and `y` we specify.

```
> p.x
3
> p.y
4
```

Our code becomes more readable. We are not just creating an *ad hoc* object; we are making a point.

We can also attach methods to objects created by classes. Let us make a nice string representation for a point. We will also make a method for calculating the distance to another point, and give default values to `x` and `y`.

```
class Point
{
    constructor(x = 0,y = 0)
    {
        this.x = x;
        this.y = y;
    }
    toString()
    {
        return "(" + this.x + ", " +  this.y + ")";
    }
    distanceTo(p)
    {
        return Math.hypot(this.x - p.x, this.y - p.y);
    }
}
```

Let us now take this for a test drive. You can now see how we attached methods to our point objects.

```
> var p = new Point(3,4);
undefined
> var q = new Point();
undefined
> p
Point {x: 3, y: 4}
> q
Point {x: 0, y: 0}
> p.toString()
"(3, 4)"
> q.toString()
"(0, 0)"
> p.distanceTo(q)
5
```

6 I demand my equal rights!

We have noticed that `==` for objects is just equality of identity. What if we want to test to see if two points are equal? We can attach a method for that. We introduce a new keyword `instanceof`.

We now add this method to our class.

```
equals(that)
{
    if( !(that instanceof Point))
    {
        return false;
    }
    return (this.x == that.x) && (this.y == that.y);
}
```

The predicate in the `if` statement is checking to see if the object we are comparing to is a `Point`. If not, we return `false` and bail. This is called the *species test*. We reject equality if the object we are comparing to is of a different species. Once the species test is over, we then know the object we are comparing to is a point and we carry out the comparison. Let us now test-drive this.

```
> var p = new Point(3,4)
undefined
> var q = new Point(3,4);
undefined
> p.equals(q)
true
> var r = new Point();
undefined
> p.equals(r)
false
> p.equals("caterwaul")
false
> p.equals(56)
false
```

Notice that the last two tests show the species test at work. Comparing a point to a number or a string returns `false`.