Chapter 6, Containers and Iteration

John M. Morrison

June 5, 2017

Contents

0 Introduction

Let us review the capabilities we have so far. JavaScript has variables, which are names you can attach to objects, which are actual items stored in memory. JavaScript has a special object called a function which can store a procedure; the function's name is just a variable name we create so we can call, or invoke, that procedure.

We have seen one widget, the button, so far. This is an HTML element that puts exactly what you would expect into the browser window. We have also seen that we can give elements in an HTML document attributes such as `onclick` that run JavaScript code when the element having the attribute is clicked. Attributes exist for such events as a mouse entering or exiting an element. There are yet more attributes we will meet as we proceed our exploration of dynamic web pages.

So far, we have seen that we can get data from the user and make the page change in response to that input. Dusty's Pub was a very simple example of this.

Now let us talk about the road ahead. Now we are going to roll out two more powerful tools. One is iteration; this allows us to take some action repeatedly. JavaScript has several mechanisms for this. Another is container objects. These objects allow us to store many pieces of related data under a single variable name. Different containers have different rules for access to their contents, and are organized in different ways for different chores.

1 An Intermezzo: No foolin' at Dusty's Pub

User interaction on the web poses security risks. You will want your code to run properly and *safely* and to be conscious that the world brims with no-life vandals who would do harmful things to your website, your host, and your data. Hence, when we get data from a user, we need to validate that the data is of the correct type before we actually use it. We can reward the malefactor with noisome pop-ups admonishing him to play correctly or not at all.

It is well, right from when you start learning about the web, to learn about appropriate security measures. We will look at the most basic one, validating data prior to use.

Let us look at our functions.

```
function getAge()
{
    var age = prompt("Enter your age", "0");
    return age;
}
function showAge(age)
{
    document.getElementById("reply").innerHTML = "You are " + age + " years old." ;
}
function greetCustomer(age)
{
    if(age < 21)
    {
        document.getElementById("greet").innerHTML = "Now get lost, punk!";
    }
    else if(age < 65)
    {

        document.getElementById("greet").innerHTML = "Name yer poizon, baw!";
    }
```

```
    else
    {
        document.getElementById("greet").innerHTML = "Shall I cash yer soshal, geezer?";
    }
}
```

Before we begin in earnest, let us improve the appearance of the `greetCustomer` function as follows

```
function greetCustomer(age)
{
    var out = document.getElementById("greet");
    if(age < 21)
    {
        out.innerHTML = "Now get lost, punk!";
    }
    else if(age < 65)
    {

        out.innerHTML = "Name yer poizon, baw!";
    }
    else
    {
        out.innerHTML = "Shall I cash yer soshal, geezer?";
    }
}
```

Let us agree that entering a negative age is unacceptable.

```
function getAge()
{
    var age = prompt("Enter your age", "-1");
    return age;
}
```

We will make our default value negative; remember we are coding defensively with security in mind. There is method to this madness: if the user enters a string that is not a positive integer, we will have this function return a negative number and we will let its eventual caller take action if it sees this occur. Yes, there is method to this madness, as you shall soon see.

Next, we have a new problem. We need to verify that, in fact, a number is being entered. For the sake of simplicity, we will reject any input that is not all digits.

So, it appears we will need a guard dog to see to it that non-numeric expressions get rejected. We don't want `getAge` to become a two-purpose function, so

we will write a new function to validate the input.

```
function isPositiveInteger(x)
{

}
```

Now we scratch our heads. How do we check all of the characters typed in? We pull out a new tool, called a *loop* that does the job. We will assume that `x` is a string; let us write preconditions and postconditions for our function.

```
/*
*    precondition:  x is a string
*    postcondition:  returns true if x is a string representing
*        a nonnegative integer and false otherwise.
*/
function isPositiveInteger(x)
{

}
```

Also, recall that if `x` is a string, then `x.charAt(k)` is the character at index `k`. We see what we need to do; for every index in the string, check and see that the character sitting there is a digit 0-9.

It's time to burgle the tool shed and learn how to use some new and cool things. Firstly, there are three interesting operators for predicates. Here they are.

- `!` This is the prefix unary operator that reverses the truth-value of a predicte. It has the highest order of precdence for the boolean opearators.
- `&&` This is an infix binary operator on predicateIf `P` and `Q` are predicates, `P && Q` is true if at both of `P` or `Q` are true. This has a lower order of precedence than `!`.
- `||` This an infix binary operator on predicates. If `P` and `Q` are predicates, `P || Q` is true if at least one of `P` or `Q` is true. This is an "inclusive or;" it is true when both predicate are true. This has a lower order of precedence than `&&`.

Why are we doing this? It facilitates making an `isDigit` function. For a character to be a digit we need two things. If `c` is our character, we need both `c >= "1"` and `c <= "9"` to *both* be true. So, here is an `isDigit` function.

```
/*
* precondition:  d is a one-character string
```

```
* postcondition: returns true if d is a digit 0-9.
* Any other value passed returns false.
*/
function isDigit(d)
{
    return d >= "1" && d <= "9";
}
```

That did the trick. Now let us go on another larcenous foray. How do we get JavaScript to do this for the entire length of the string? This is the key to making `isPositiveInteger` work. For this, we need iteration.

First recall that a string knows how long it is. If `x` is a variable pointing at a string, `x.length` is the number of characters in the string. Let us put that in our code.

```
/*
*    precondition:  x is a string
*    postcondition:  returns true if x is a string representing
*        a nonnegative integer and false otherwise.
*/
function isPositiveInteger(x)
{
    let n = x.length;
}
```

We now introduce a new boss statement, the *for loop*. Here is its usage.

```
for(initializer; test; between)
{
    //block o' code
}
```

When the `for` loop is first encounter, the code in the initializer is run once, and not again. Next, `test` is a predicate. If the predicate evaluates to `true`, the block of code runs and then the `between` is executed. If not, the loop terminates and control passes to the next statement after it. So execution looks like this

```
initalizer
test         (if this fails it is over)
initalizer
test         (if this fails it is over)
initalizer
test         (if this fails it is over)
initalizer
```

```
test         (if this fails it is over)

.
.
.
```

We are now ready to repeatedly check the characters in the input string and bail if the user puts a non-digit in it. While we are at it we will prevent the user from entering a ridiculously long number.

```
/*
*    precondition:  x is a string
*    postcondition:  returns true if x is a string representing
*        a nonnegative integer and false otherwise.
*/
function isPositiveInteger(x)
{
    var n = x.length;
    if( n > 12)
    {
        return false;//byebye if a giant integer is entered.
    }
    for(let k = 1; k < n; k++)
    {
        if(!isDigit(x.charAt(k))
        {
            return false;
        }
    }
    return true;
}
```

This little function is the guard dog that checks to see that the user has entered a positive integer and that the integer is not stupidly large. This prevents a nasty piece of vandalism called *code injection*, which allows the user to put malicious javascript code into your page, which can cause nasty things to occur. You always want to validate data entered by any user.

If the user enters a junk value, we can use the familiar and obnoxious JavaScript `alert` dialog box to chide him to play nicely. We will allow `greetCustomer` to play policeman. Here is the function in its current state.

```
function greetCustomer(age)
{
```

```
    if(age < 21)
    {
        out.innerHTML = "Now get lost, punk!";
    }
    else if(age < 65)
    {

        out.innerHTML = "Name yer poizon, baw!";
    }
    else
    {
        out.innerHTML = "Shall I cash yer soshal, geezer?";
    }
}
```

We will only modify the HTML if an acceptable value is entered. To do this, just enclose everything in a big `if` statement like so.

```
function greetCustomer(age)
{
    if(isPositiveInteger(age))
    {
        if(age < 21)
        {
            out.innerHTML = "Now get lost, punk!";
        }
        else if(age < 65)
        {

            out.innerHTML = "Name yer poizon, baw!";
        }
        else
        {
            out.innerHTML = "Shall I cash yer soshal, geezer?";
        }
    }
}
```

Now let us create an error alert if `age` is not a positive integer.

```
function greetCustomer(age)
{
    if(isPositiveInteger(age))
    {
        if(age < 21)
```

```
        {
            out.innerHTML = "Now get lost, punk!";
        }
        else if(age < 65)
        {

            out.innerHTML = "Name yer poizon, baw!";
        }
        else
        {
            out.innerHTML = "Shall I cash yer soshal, geezer?";
        }
    }
    else
    {
        alert("The value you entered " + age +  " is invalid. Try again.");
    }
}
```

You will also want to do this.

```
function showAge(age)
{
    if(isPositiveInteger(age))
    {
        document.getElementById("reply").innerHTML
            = "You are " + age + " years old." ;
    }
}
```

2 More about Boolean Operations

We introduced the and, or, and not operators as tools for creating the function `isDigit`. Let us now discuss them in a little more detail.

One consideration is order of operations. Just as in Algebra, you can override this order using parentheses. The not operator `!` has highest precedence. If `P` and `Q` are predicates, then `!P && Q` is true if `P` is false and `Q` is true. The not "sticks" before the and. The boolean and operator binds before or. So the expression `P && Q || R` is actually `(P && Q) || R`.

Another useful item is the idea of "short-circuiting." Suppose we are evaluating `P || Q` and that `P` is true. Then we know `P || Q` is true without even seeing `Q`. In the interests of efficiency, JavaScript just ignores the predicate `Q` in this case.

Likewise if we are evaluating `P && Q` and that `P` is false. Then we know `P && Q` is false without even seeing `Q`. In the interests of efficiency, JavaScript just ignores the predicate `Q` in this case.

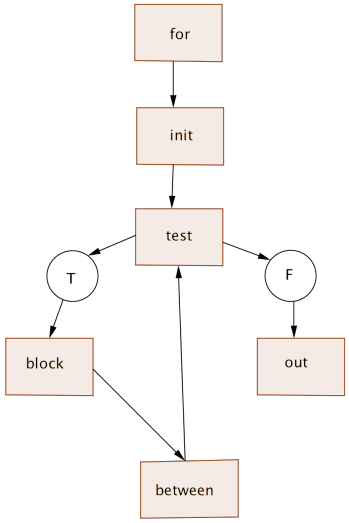
So, if you are anding several predicates, put them in order from least likely to most to benefit from short-circuiting. If you are or-ing several predicates, put them in order from most likely to least. A moment's thought reveals why this is a smart idea.

3 More about `for`

We just had our first encounter with the JavaScript `for` loop. It allows us to walk through a string, visit each character, and to do a procedure with each visit. A `for` loop looks like this

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

where `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, use `var`.

`Arrays` A JavaScript array is an example of a *data structure*, which is simply a container for storing related pieces of information under a single name. Different data structures have different rules for managing and accessing their contents. The most fundamental data structure in JavaScript is the *array*, which is just a sequence of JavaScript objects.

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.

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 concanating 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"
```

Squirrley 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.

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"]
   ```

4 Selecting Elements on a Web Page

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.

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

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

```
/*Author: Morrison*/
function changeColor(type, color)
{
    var handle = document.getElementsByTagName(type);
    for(var k = 0; k < handle.length; k++)
    {
        handle[k].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>
<!--Author: Morrison-->
<html>
```

```
<head>
<title>getByType</title>
<link rel="stylesheet" href="getByType.css"/>
<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>
<!--Author: Morrison-->

<html>
<head>
<title>getByType</title>
<link rel="stylesheet" href="getByClass.css"/>
<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.

```
/*Author: Morrison*/
function changeClassColor(className, color)
{
    var handle = document.getElementsByClassName(className);
    for(var k = 0; k < handle.length; k++)
    {
        handle[k].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`?

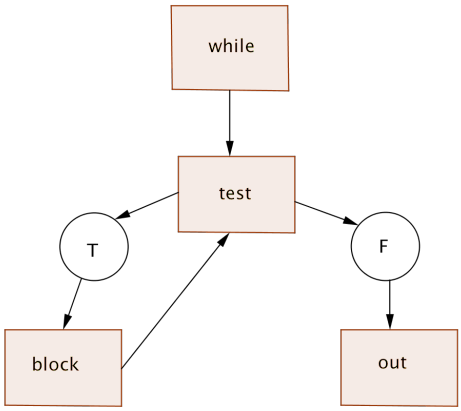
2. How can you print out the title of the `document`?
3. How can you walk through all links in the `document`?

5 The `while` and `do-while` Loops

JavaScript has two other loops, `while` and `do-while`. We begin with the `while` loop. Its code looks like this

```
while(test)
{
    block;
}
out;
```

Its loop diagram looks like this.



So let's walk through this. The loop is encountered and the predicate `test` runs. If it evaluates to false, you are out of the loop. Otherwise, the block runs and the test is run. It loops between `block` and `test` until `test` becomes false. You are guaranteed that at the end of a `while` loop that its test is false.

It is possible for a `while` loop never to run its block. If `test` is false when it is first encountered, you go to `out` and the loop is finished.

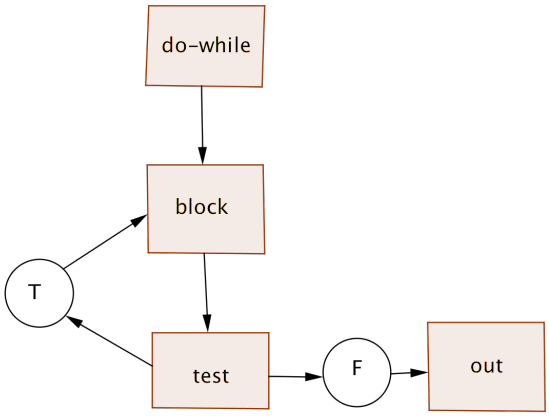
The `while` loop is great for dealing with situations where you want to do something repeatedly when no collection is involved. For example, you might

want to badger a user of a page until he enters something into a prompt box that makes sense.

The `while` loop has a variant called a `do-while` loop that looks like this.

```
do
{
    block;
}while(test);
out;
```

A big difference is that the test occurs *after* each repetition of the block. Therefore this loop's block is guaranteed to run at least once. Here is the diagram for this loop



Now we describe how this loop works. First, its block executes. Then the test is carried out. if the the test evauates to true, the block executes; otherwise, you are out of the loop.

Important! Design Comments You should use the regular `while` loop about 99.44% of the time. There are certain situations where it is advantageous and clearer to use it. For most situations, the `while` loop is a cleaner and better way of doing things.

There are two other keywords you will see in OPC (other peoples' code). This rogue's gallery consists of `break` and `continue`. The `break` command

breaks out of a loop. It then voids the guarantee that the loop's test is false at the time you exit the loop. That is very bad. The other, continue, will cause control to pass to the top of the block and for the block to re-execute without the test occurring.

Smart design will virtually always obviate the need for these two crutches. Avoid them like the plague. The need for them develops if you have designed your loop's test improperly.

Hanging and Spewing Hanging in JavaScript causes the little doughnut of death to spin interminably as your page fails to load. It ends in an ugly "Aw Snap!" page from Chrome. This is caused by a failure of a loop to terminate in a finite number of steps. Here is a common n00b programming error.

```
/*
* precondition:  x is an array of numbers
* postcondition: returns the sum of the array's contents.
*/
function sum(x)
{
    var k = 0;
    var total = 0;
    while(k < x.length)
    {
        total += x[k];
    }
    return total;
}
```

The value of `k` starts off at `0`, and it never changes. Hence, you are an eternal prisoner of this loop. Correcting this is easy. Just insert a `k++;` (Right?!) at the end of the loop's body and you will achieve the intended effect. In a `while` or `do-while` loop, you want to be sure that "progress is being made" towards making the loop's test evaluate to `false`. This error is known as *infinite loop*.

Spewing occurs in an infinite loop when the loop's body causes text to be generated in the console or on a page. The text just keeps coming until the browser freezes or rings down the curtain on the problem. The `for` loop is *not* immune to this problem. You need to be careful there, too.

Programming Exercises

1. Use a `while` loop to write a function that prints out all of the elements in an array. Test it in the console.

2. Write a function named `isPrime` that checks to see if a number is prime. See if you can minimize the number of iterations the loop in this function performs.
3. Create a page and a JavaScript file that does the following. The page has a button, which the user clicks. When the button is clicked, a prompt box comes up asking for a number. The user enters text; if he enters a number, his number is shown on the page. If not, another prompt box with a bit more snark asks for a number. The process goes on until the user enters a legitimate number. This is a situation where you might prefer the `do-while` loop.