Chapter 7, Containers and Iteration

John M. Morrison

October 14, 2017

Contents

0 Introduction and Orientation

Let us review the capabilities we have so far.

1. We have learned that JavaScript has objects, which are regions of memory that store data (state) and which can be sent messages that cause them to do work (methods).
2. JavaScript has variables, which are names you can attach to objects, which are actual items stored in memory. JavaScript keeps track of variables via dictionaries called symbol tables. Three of these are of importance to us. The global symbol table contains JavaScript's built-in infrastructure. This is where objects such as strings, numbers, and booleans are defined. It is also where always-visible functions such as `parseInt` and `parseFloat` are kept. When you define a object outside of any function, this becomes part of the user-defined global symbol table. These variables are visible from the time of their creation until the program ends execution.
3. JavaScript has a special type of object called a function. Functions allow us to store a procedure under a name. Functions keep their local variables secret from the rest of the program, alleviating namespace clutter and obviating the need for keeping track of their internal workings. They free us to think about *what* they do and not *how* they do it. When a function is called, its local symbol table is visible. This symbol table is destroyed as soon as the function returns. Functions you define in your program are part of the user-defined global symbol table.
4. JavaScript has conditional logic, which allows programs to make decisions based on information in their visible symbol tables.

0.1 What is next?

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. Our first container object will be the *array*, which allows us to store a list of objects under one name. Arrays and iteration, as we shall soon see, are natural partners for handling volumes of data.

1 The JavaScript Array Object

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. 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; both of these are examples of array literals. Notice that you must enclose the contents of an array literal 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`.

Access to array entries works in a manner similar to that of strings; we use the `[]` operator to do this. The mechanism of indexing and the fact that indices lurk between elements is identical to that of strings.

```
> stuff[0]
"Moe"
> stuff[1]
"Larry"
```

Here is an important difference. Look at what happens when we try to alter a string object.

```
> var bovine = "cow"
undefined
> bovine[0] = "C"
"C"
> bovine
"cow"
```

Ooh! How passive-aggressive! JavaScript just takes our request to do this illegal thing and drops in right on the floor. Remember, we said that strings are immutable; once created their state, i.e. their character sequence cannot be changed.

Now let us do this with our array.

```
stuff[2] = "Schempp";
"Schempp"
stuff
(5) ["Moe", "Larry", "Schempp", 5.6, true]
```

The entries in an array are lvalues, i.e. they are assignable, just as any other variable is. Arrays are mutable; you can change any of their entries. In your interactive session, there is a grey triangle to the left of the array. Pop it open by clicking on it to reveal this.

```
(5) ["Moe", "Larry", "Schempp", 5.6, true]
0 : "Moe"
1 : "Larry"
2 : "Schempp"
3 : 5.6
4 : true
length : 5
__proto__ : Array(0)
```

The line `length:5` tells us an array knows its length as a property.

```
stuff.length
5
```

If you click on the triangle to the left of `__proto__`, you will see all array methods listed. We show this here.

```
concat :  concat()
constructor :  Array()
copyWithin :  copyWithin()
entries :  entries()
every :  every()
fill :  fill()
filter :  filter()
find :  find()
findIndex :  findIndex()
forEach :  forEach()
includes :  includes()
indexOf :  indexOf()
join :  join()
keys :  keys()
lastIndexOf :  lastIndexOf()
length : 0
map :  map()
pop :  pop()
```

```
push :  push()
reduce :  reduce()
reduceRight :  reduceRight()
reverse :  reverse()
shift :  shift()
slice :  slice()
some :  some()
sort :  sort()
splice :  splice()
toLocaleString :  toLocaleString()
toString :  toString()
unshift :  unshift()
Symbol(Symbol.iterator) :  values()
Symbol(Symbol.unscopables) : {copyWithin: true, entries: true,
    fill: true, find: true, findIndex: true, }
__proto__ : Object
```

This reveals the entire list of array methods. We will explore some of them and you can learn about more in the documentation. This exploration will be the contents of the next section.

2 Array Methods

We saw at the end of the last section that there is a large number of array methods. We will explore some basic ones you will use frequently. We already know that an array keeps its length as a property and that its entries are lvalues.

Let us begin with `push` and `pop`. These methods both act on the end of the array. If you have closed your console session, fire it back up and enter this.

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

Now let us get pushy.

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

We see two things here. The items we pushed onto the array are appended to the end of the array. The method `push` also has a return value: this is the size of the array after the new item has been added to the array. 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
```

Notice that not only can we change array entries by assignment, but we can add and delete entries as well. Arrays are very mutable, indeed. Do these exercises now; they will introduce another interesting pair of methods that do work on arrays. Notice how they behave in a manner similar to push and pop.

Programming Exercise There is a pair of methods shift and unshift that act on the start of an array (at index 0). Do an exploration and answer the following questions.

1. What does stuff.shift() do to the array stuff?
2. What does the shift operator return?
3. What argument does stuff.unshift() require?
4. What does it do to the array stuff?
5. What does the unshift operator return?

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 loooooove 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.

2.1 A Brief Side Trip `toString`

While we are here, let us take a moment to discuss `toString()`. Builtin JavaScript objects have a `toString` method.

```
> var b = true
undefined
> b.toString()
"true"
> var a = 45
undefined
> a.toString()
"45"
> var s = "Hey I am a string!"
undefined
> s.toString()
"Hey I am a string!"
> function f(x){return x};
undefined
> f.toString()
"function f(x){return x}"
```

The `toString()` method for a number has an optional second argument. If you pass an argument 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"
num.toString(5)
"1331"
```

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.

2.2 Rose Mary Woods

Now we shall discuss a method that would warm the cockles of this lady's heart.

The `splice` method allows us to replace a segment of an array with new entries. Now try this.

```
> var tape = ["zero", "one", "two", "three", "four", "five"]
undefined
> tape.splice(4)
(2) ["four", "five"]
> tape
(4) ["zero", "one", "two", "three"]
```

If you pass a single integer to this method, it will truncate the array starting at that integer, and it will return an array containing any evictees.

Now let's put 'em back.

```
tape.splice(tape.length, 0, "four", "five");
[]
```

```
tape
(5) ["zero", "one", "two", "three", "four", "five"]
```

Since nobody got evicted, an empty list `[]` is returned. Now watch this.

```
> tape.splice(2, 2, "TWO", "THREE", "WHEE");
(2) ["two", "three"]
tape
(7) ["zero", "one", "TWO", "THREE", "WHEE", "four", "five"]
```

What happened? The evictees got returned. They got replaced by the three entries we passed in. So, if we have an array `foo` and we call `splice`, its actions are as follows.

1. `splice(n, howMany)` will evict `howMany` entries starting at index `n`.
2. `splice(n, howMany, item1, item2, ....)` will evict `howMany` entries starting at index `n` and insert the items `item1`, `item2` ... there.
3. `splice(n)` will evict all entries after index `n` and return the evictees in an array.

Programming Exericise The method `splice` hs a friend named `slice`. Compare and contrast these methods. You will find `slice` to be quite useful.

2.3 A Few Other Useful Array Methods

Youu can check for the presence of an object in an array using `includes`.

```
> var x = ["a","b", "c", "d", "e", "f", "g", "h"];
undefined
(8) ["a", "b", "c", "d", "e", "f", "g", "h"]
> x.includes("cows")
false
> x.includes("b")
true
```

You can locate an object in an array using `indexOf`.

```
> x.indexOf("c")
2
```

If the item is not present, a sentinal value of -1 is returned.

```
> x.indexOf("goose chase")
-1
```

Now let's add a few more entries to demonstrate searching from the right.

```
x.splice(x.length, 0, "g", "f", "e", "d","c", "b", "a")
[]
>x
(15) ["a", "b", "c", "d", "e", "f", "g",
    "h", "g", "f", "e", "d", "c", "b", "a"]
x.lastIndexOf("a")
14
x.indexOf("a")
0
```

We leave you to do a little spelunking on your own. Do not limit your explorations to these problems here. Try some of the methods we have not discussed.

Programming Exercises

1. Both of these methods `indexOf` and `lastIndexOf` have an optional second argument that is an integer. What does it do? Experiment and deduce its action.
2. Strings have a method `includes`. Experiment and figure out what it does.
3. There is an array method called `join` which takes a string as an argument. What does it do?
4. 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.

2.4 Command Line Arguments for Node

Create this program named `commandLine.js`.

```
console.log(process.argv);
```

Now run it in a terminal.

```
$ node commandLine.js a b c d e f
```

```
[ '/opt/local/bin/node',
  '/Users/morrison/book/webdev/wd7NEW/commandLine.js',
  'a',
  'b',
  'c',
  'd',
  'e',
  'f' ]
$
```

The object returned by `process.argv` is an array of strings. Enty 0 is the location of the `node` program in your system, and entry is the absolute path of your program. The rest of the entries are command line arguments. Here is an example of how we can use this mechanism.

```
var input = process.argv
var number = parseFloat(input[2])
console.log("The square of " + input[2] + " is " + (number*number) + ".");
```

```
$ node squarer.js 13
The square of 13 is 169.
```

Now run it like so

3 Iteration with `while`

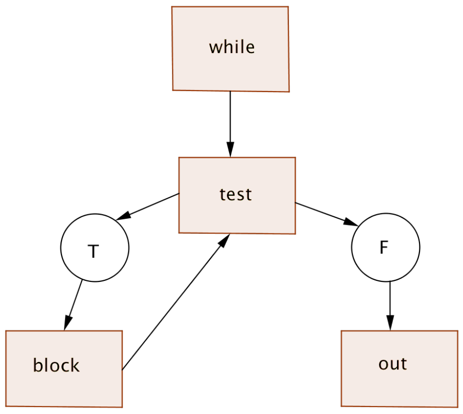
Now we roll out a powerful new tool that allows us to do something repeatedly; this is called `iteration`.

We begin with the `while` loop. Its code looks like this

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

The object `test` is a boolean-valued expression (predicate). The `block` represents a block of code. The statement `out` is just the next statement beyond the loop.

This *loop* diagram illustrates its action.



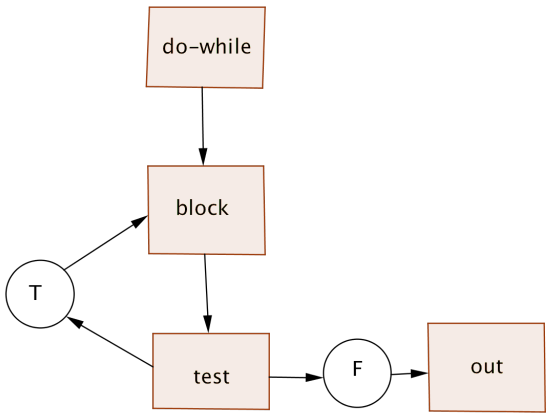
So let's walk through this. The loop is encountered and the predicate `test` runs. If it evaluates to `false`, you are out of the loop; otherwise, the block runs. The cycle of `test-block` is repeated. As soon as `test` becomes false, you are out of the loop. You are guaranteed that at the end of a `while` 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 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 test evaluates 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 nearly all 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.

4 Iterating with `for`

The `for` loop in JavaScript is a protean beast that appears in several guises. We will discuss the oldest one, the *C-style for loop* first. Here is its usage.

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

When the `for` loop is first encountered, 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)

.
.
.
```

A simple example we can write with the benefit of a `for` loop is a function to check and see if a string is a legitimate decimal integer. To do this we do the following.

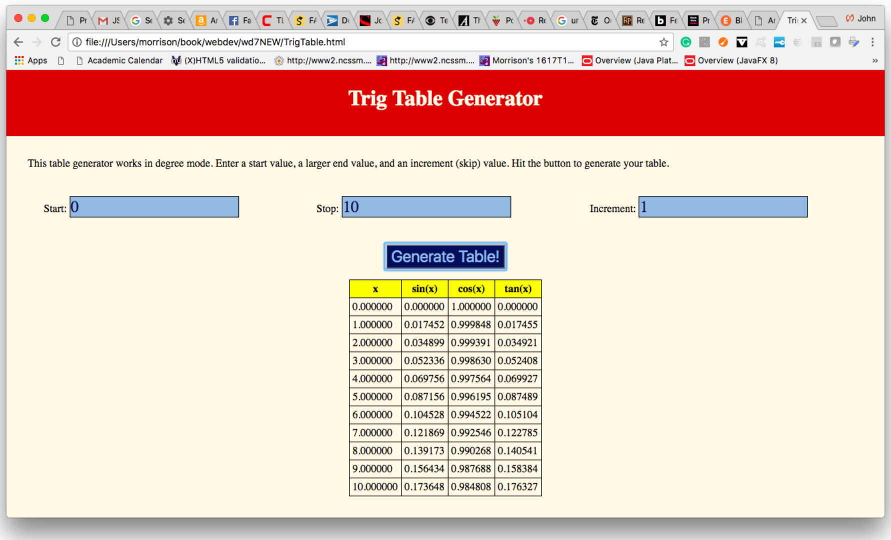
1. The first character could be a + or a -; if it is skip over it.
2. Every subsequent character must be a digit 0-9.

So we code as follows.

```
function isDecimalIntegerString(s)
{
    let k = 0;
    if(s[0] === "+" || s[0] === "-")
    {
        k++;
    }
    /the intializer can be empty here, since we already have k.
    for(; k < s.length; k++)
    {
        //If we find a non-digit, bail and say "fail."
        if(s[k] < "0" || s[k] > "9")
        {
            return false;
        }
    }
    //Phew! We made it!
    return true;
}
```

5 Case Study: A Trig Table Page

One thing loops do very nicely is to build tables dynamically. We are now going to make a web app that generates a trigonometric table in degrees; the user will specify a start value, and end value, and an increment. When a button is pushed a table with those values is generated on the page. Here is what the product will look like


file:///Users/morrison/book/webdev/wd7NEW/TrigTable.html

Trig Table Generator

This table generator works in degree mode. Enter a start value, a larger end value, and an increment (skip) value. Hit the button to generate your table.

Start: 0 Stop: 10 Increment: 1

x	sin(x)	cos(x)	tan(x)
0.000000	0.000000	1.000000	0.000000
1.000000	0.017452	0.999848	0.017455
2.000000	0.034899	0.999391	0.034921
3.000000	0.052336	0.998630	0.052408
4.000000	0.069756	0.997564	0.069927
5.000000	0.087156	0.996195	0.087489
6.000000	0.104528	0.994522	0.105104
7.000000	0.121869	0.992546	0.122785
8.000000	0.139173	0.990268	0.140541
9.000000	0.156434	0.987688	0.158384
10.000000	0.173648	0.984808	0.176327

5.1 The Skeleton

So, let's start to make this happen. We will begin with the HTML elements on the page. What do we see?

1. The page has a header and a main element.
2. There are three input boxes. Each is labeled.
3. The header of the table is present before the button is ever pushed.
4. There is a button you push, once you enter values, to generate the table.

So, here is our first cut at the HTML.

```
<!doctype html>
<!--Author: Morrison-->

<html>
<head>
<title>TrigTable</title>
<meta charset="utf-8"/>
```

```
<link rel="stylesheet" href="TrigTable.css"/>
<script type="text/javascript" src="TrigTable.js">
</script>
</head>
<body>
<header>
    <h1>Trig Table Generator</h1>
</header>

<main>

<p>This table generator works in degree mode.
Enter a start value, a larger
end value, and an increment (skip) value.
Hit the button to generate your table.  </p>

<p>Start: <input type="text" id="startValue"></input></p>
<p>Stop: <input type="text" id="stopValue"></input></p>
<p>Increment: <input type="text" id="skipValue"></input></p>

<p><button>Generate Table!</button></p>

<table>
<tr><th>x</th><th>sin(x)</th><th>cos(x)</th>
    <th>tan(x)</th></tr>
<tbody>
</tbody>
</table>

</main>
</body>
</html>
```

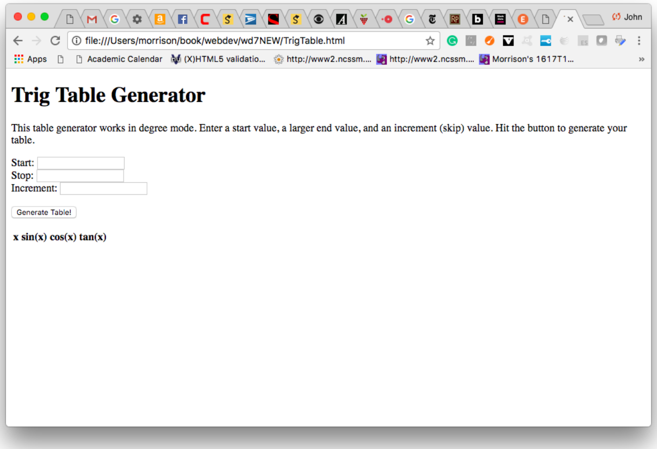
We do need to give some elements an `id`. We are going to insert table rows in the `tbody` element, so lets' do this

```
<tbody id="entrails">
</tbody>
```

The button will need an `onclick` attribute, so let's block that in.

```
<p><button onclick="">Generate Table!</button></p>
```

Here is our page's sad appearance appearance without any CSS.



Trig Table Generator

This table generator works in degree mode. Enter a start value, a larger end value, and an increment (skip) value. Hit the button to generate your table.

Start:
Stop:
Increment:

Generate Table!

x sin(x) cos(x) tan(x)

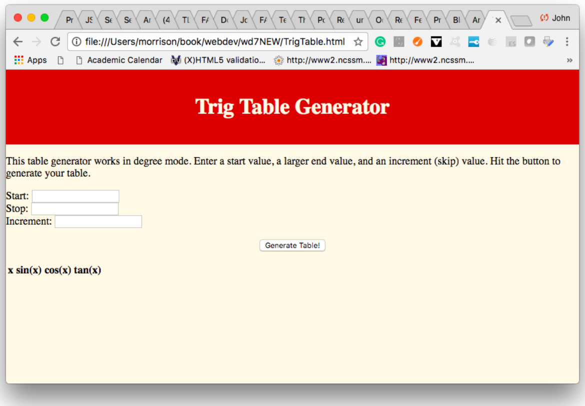
`The Skin` Now we begin adding style. Let's get the header looking right and make a provision for centering the button.

```
h1, h2, .display
{
    text-align:center;
}
header
{
    padding:1em;
    background-color:#CC0000;
    color:#FFF8E7;
}
```

To defeat the annoying margins imposed by the user agent, let's do this. We will also give the body a background color

```
body, html
{
    margin:0;
    padding:0;
}
body
{
    background-color:#FFF8E7;
}
```

Our page now has an improved appearance.



Trig Table Generator

This table generator works in degree mode. Enter a start value, a larger end value, and an increment (skip) value. Hit the button to generate your table.

Start:
Stop:
Increment:

x sin(x) cos(x) tan(x)

One other item needs doing. Put all of the document outside of the `header` element inside of a `main` element. Then give `main` a style rule so it has a margin.

```
main
{
    margin:2em;
}
```

Your content will no longer crowd at the edges of the page.

The next order of business is to format the input boxes and to center them on the page. For this we will use `div` elements and the table display apparatus. We begin by wrapping our boxes in the appropriate `div`s.

```
<div class="table">
    <div class="row">
        <div class="cell">
            Start: <input type="text" id="startValue"></input>
        </div>
        <div class="cell">
            Stop: <input type="text" id="stopValue"></input>
        </div>
        <div class="cell">
            Increment: <input type="text" id="skipValue"></input>
        </div>
    </div>
</div>
```

Now we will build the styles that make this work. First, we handle the

`.table` class as follows. You can fiddle with `border-spacing` if you want to adjust the closeness of the three input boxes.

```
.table
{
    display:table;
    border-spacing:1.5em;
    margin-left:auto;
    margin-right:auto;
}
```

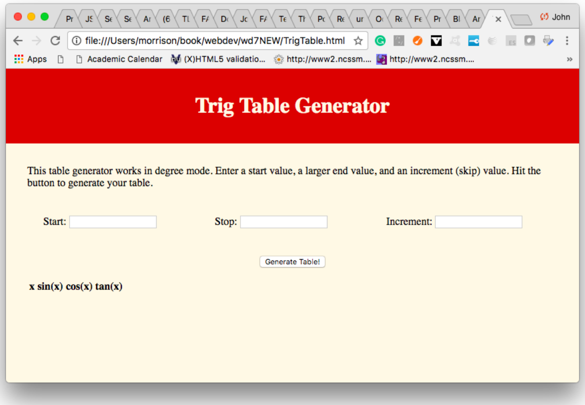
Now we put in the table row style.

```
.row
{
    display:table-row;
}
```

Finally we style the table cells. You can adjust the width if you wish.

```
.cell
{
    display:table-cell;
    width:20%;
}
```

Now let us look at what we have so far.



Trig Table Generator

This table generator works in degree mode. Enter a start value, a larger end value, and an increment (skip) value. Hit the button to generate your table.

Start: Stop: Increment:

x sin(x) cos(x) tan(x)

Absent styles, our tables are not going to look so great. So, let's give them some basic style so they look good. This code centers the table, puts rules around the cells, and makes the header yellow.

```
table
{
    margin-left:auto;
    margin-right:auto;
}
table, th, td
{
    border:solid 1px black;
    padding:.3em;
    border-collapse:collapse;
}
th
{
    background-color:yellow;
}
```

Next, we will prettify the input boxes and the buttons.

```
button
{
    font-size:1.5em;
    color:#99BADD;
    background-color:#001A57;
}
input
{
    font-family:mononspaced;
    background-color:#99BADD;
    color:#001A57;
    font-size:1.5em;
    border:solid 1px black;
}
```

All of the styles are now in place. Our table has the desired appearance.

5.2 All of this is pretty, but how about this working?

The next step in this game is to write the JavaScript file that connects it all together. It would be convenient to have trig functions that compute in degrees. One thing we have to watch out for is the awful tan(90°). Sine and cosine are easy.

```
function sinDeg(x)
{
    return Math.sin(x*Math.PI/180);
}
function cosDeg(x)
{
    return Math.cos(x*Math.PI/180);
}
```

We will exploit the periodicity and symmetry of the tangent function to tame its evil propensity. We know this

$$\tan(-x) = -\tan(x),$$

so let's do this.

```
function tanDeg(x)
{
    var sgn = 1;
    if(x < 0)
    {
        x = -x;
        sgn = -1;
    }
}
```

Since tan repeats every 180°, we can do this

```
function tanDeg(x)
{
    var sgn = 1;
    if(x < 0)
    {
        x = -x;
        sgn = -1;
    }
    x = x % 180;
}
```

If we are too close to 90 degrees, we return `NaN` to avoid aesthetical excrescences in our table.

```
function tanDeg(x)
{
    var sgn = 1;
    if(x < 0)
    {
        x = -x;
        sgn = -1;
    }
    x = x % 180;
    if(Math.abs(x - 90) < 1e-10)
        return NaN;
    return sgn*Math.tan(x*Math.PI/180);
}
```

We now have our trig functions in place. The next order of business is to get the values out of the input boxes and to convert them to numbers. Remember, the values come back as strings, so we need to convert them. Here we show getting the value of the start box.

```
function getStart()
{
    var out = document.getElementById('startValue').value;
    return parseFloat(out);
}
```

Deal similarly with the others.

```
function getStop()
{
    var out = document.getElementById('stopValue').value;
    return parseFloat(out);
}
function getSkip()
{
    var out = document.getElementById('skipValue').value;
    return parseFloat(out);
}
```

Now we build the table. We know we will need a loop for this and a variable to accumulate the HTML code.

```
function makeTable(start, stop, skip)
{
    let out = ""
    for(var k = start; k < stop + skip/2; k += skip)
    {
        out += (each table row)
    }
    document.getElementById("entrails").innerHTML = out;
}
```

Now we format the table rows. We need to put the values of `k` and then their sines, cosines and tangents in a table row. So we append to `out` as follows.

```
out += "<tr><td>" + k.toFixed(6)
    + "</td><td>" + sinDeg(k).toFixed(6)
    + "</td><td>" + cosDeg(k).toFixed(6)
    + "</td><td>" + tanDeg(k).toFixed(6) + "</td></tr>";
```

We use `toFixed(6)` to determine the number of decimal places tht will show.

Programming Exercises

1. Add a button to change to radian mode and alter the code to produce tables in radians.
2. Add an input to allow the user to specify how many decimal places to show.