≡ MENU

Eloquent JavaScript: Laying out a Table

16 APRIL 2015

Introduction

When reading the 2nd Edition of Eloquent JavaScript, I noticed that the "Laying out a Table" section in Chapter 6 seemed like it could be a sticking point for many for beginners trying to learn Javascript. After poking around a bit online, I saw a couple comments that confirmed this assumption.

I think the only way to show what is going on is to trace the drawTable function from begining to end. I used the rows array that was built with the checkerboard code to go trace the function. I documented all of the steps in hopes that it helps anyone learning from this book get unstuck.

I am assuming you already have a firm grasp of the map and reduce methods. Those are necessary to understanding much of the drawTable function.

Hopefully this guide is not too confusing and helps clear some things up!

Step 1: Build the rows array

For the sake of simplicity, lets go through the checkerboard problem but only make it a 2 x 2 board.

```
var rows = [];
for (var i = 0; i < 2; i++) {
    var row = [];
    for (var j = 0; j < 2; j++) {
        if ((j + i) % 2 == 0)
           row.push(new TextCell("##"));
        else
            row.push(new TextCell("  "));
    }
    rows.push(row);
}
console.log(drawTable(rows));
```

This code should be relatively easy to follow. Basically it is creating an array of arrays where the outer array (`rows`) is a container for all of the rows in the array. Each inner array represents one row of the checkerboard.

The line:

```
if ((j + i) % 2 == 0)
```

is the part of the code that creates the checker pattern. It takes advantage of the fact that if we fill in (with `“##”`) every cell where the sum of the row number and column number is even, a checkerboard pattern will appear. Keep in mind to start counting from zero, not one, when checking this for yourself.

If the “if statement” we just described above is true, then the code will push an object that is built with the `TextCell` constructor to the end of the current `row` array. Let’s take a look at the `TextCell` constructor:

```
function TextCell(text) {
  this.text = text.split("\n");
}
```

The `TextCell` constructor only has one parameter, text. We can see that the parameter is meant to be a string that will be the content for one of the cells in the table we are building. The string has the split method applied to it, which will turn it into an array. If there are any newlines present in the string that was used as an argument, then that is where the string will be separated into array elements when the array is being formed. For example:

```
var testObj = new TextCell(“This is a\nTEST”);
```

Will create an object called `testObj` that looks like:

`{text: [“This is a”,”TEST”]}`

Getting back to the checkerboard. When the conditional is true, the `TextCell` constructed object that is pushed to the row array will look like:

`{text: [“##”]}`

and when the conditional is false, the object that is pushed will look like:

`{text: [“  ”]}`

As we see in the checkerboard code, when the inner for loop completes itself, the array that has just been built, which represents a single row, will then be pushed to the outer rows array. With our 2 x 2 checkerboard, our complete rows array should look like:

`[[{text: [“##”]},{text: [“  ”]}],[{text: [“  ”]},{text: [“##”]}]]`

It is an array that contains two inner arrays. Each inner array contains two objects. Each object represents one cell or “box” in our checkerboard.

It is important to realize that the `rows` array is our input which the `drawTable` function will use to build a formatted checkerboard.

Step 2: Pass the rows array into the drawTable function

Now that we understand what rows looks like, we can move on to the hard part. We can see in the final line of the checkerboard code above, that rows is passed into the function, drawTable. When it is logged to the console, our checkerboard should look like:

```
// → ##
//      ##
```

a 2 x 2 checkerboard where the upper left and bottom right boxes are filled in with `##`.

Let's look at the complete `drawTable` function to see how this is created:

```
function drawTable(rows) {
  var heights = rowHeights(rows);
  var widths = colWidths(rows);

  function drawLine(blocks, lineNo) {
    return blocks.map(function(block) {
      return block[lineNo];
    }).join(" ");
  }

  function drawRow(row, rowNum) {
    var blocks = row.map(function(cell, colNum) {
      return cell.draw(widths[colNum], heights[rowNum]);
    });
    return blocks[0].map(function(_, lineNo) {
      return drawLine(blocks, lineNo);
    }).join("\n");
  }

  return rows.map(drawRow).join("\n");
}
```

Step 3: Find the `heights` value with the `rowHeights` function

First, the `drawTable` function creates two variables, `heights` and `widths`, using the `rowHeights` and `colWidths` functions. We can see above that `rowHeights` and `colWidths` each takes in the argument, `rows`, which is the array that we just built and passed into `drawTable`. Lets take a look at `rowHeights`:

```
function rowHeights(rows) {
  return rows.map(function(row) {
    return row.reduce(function(max, cell) {
      return Math.max(max, cell.minHeight());
    }, 0);
  });
}
```

Lets break it down:

Line 1: rowHeights takes in the rows array as an argument. As a reminder, this is our rows array:

`[[{text: [“##”]},{text: [“  ”]}],[{text: [“  ”]},{text: [“##”]}]]`

Line 2: An array that is created with the Array.prototype.map method is being returned. The map method is being used on our `rows` array that we passed in. The callback function will be transforming the two inner row arrays and returning them in a new array. To clarify, the map method will first look at the first inner array:

`[{text: [“##”]},{text: [“  ”]}]`

followed by the second inner array:

`[{text: [“  ”]},{text: [“##”]}]`

They will be transformed by the body of the callback function we will see on line 3 and 4 and returned as a new array.

Line 3 & 4: We can see the callback function’s body is a reduce method what will go through each of the objects in the inner array and return a the height (in lines) of the tallest cell in the row we are looking at. It does this by checking each `cell.minHeight` and returning the largest one.

`cell.minHeight` comes from the `TextCell` prototype object. Lets take a look:

```
TextCell.prototype.minHeight = function() {
  return this.text.length;
};
```

By looking at the reduce function, we can see that cell represents each object in the inner arrays. So first, the reduce method is going to look at the top row (the first inner array) and see there are two objects in it. Its going to take the first object which is:

`{text: [“##”]}`

We see that the minHeight function simply returns the length of the array that is paired with the text property. It is an array with only one element so the length is 1. Don’t make the mistake of thinking it is asking for the length of the string in the array. That’s what the `colWidths` function is supposed to do as we will see in a moment.

Since both of the objects in the first row have arrays of length one, the map method is going to return 1. The exact same thing happens for the second row. Now we can see that the `rowHeights` function returns an array with the two elements we just found and it will look like this.

`heights == [1,1]`

Step 4: Find the `widths` value with the `colWidths` function

Now lets looks at the `colWidths` function:

```
function colWidths(rows) {
  return rows[0].map(function(_, i) {
    return rows.reduce(function(max, row) {
      return Math.max(max, row[i].minWidth());
    }, 0);
  });
}
```

Like `rowHeights`, the `rows` array is passed into function as an argument. Once again, this is what the entire `rows` array looks like:

`[[{text: [“##”]},{text: [“  ”]}],[{text: [“  ”]},{text: [“##”]}]]`

We see on line two of the `colWidths` function, it is returning an array that is generated by the map method on rows[0]. Looking at the `rows` array, rows[0] is this part:

```
[{text: ["##"]},{text: ["  "]}]
```

It is an array containing two objects. We can see that the callback function that is being used as an argument in map is only concerned with the index and not the element value. The underscore signifies that the element value is not needed. We can see that map's callback function returns a reduce method on the entire `rows` array (not just `rows[0]` like the map method that contains it) and returning the maximum width (in characters) for each column. We see that it specifically does this by looking at `row[i].minWidth`. This works because the first time the map method iterates, `i` will be 0. The reduce method will first look at the first row and then specifically look at `row[0]`, which contains:

```
{text: ["##"]}
```

It will use the `minWidth()` method (which is located in the `TextCell` prototype object) to return the length of the string in the array which is the `"##"`. Let's take a look at `minWidth`:

```
TextCell.prototype.minWidth = function() {
  return this.text.reduce(function(width, line) {
    return Math.max(width, line.length);
  }, 0);
};
```

We see that there is a reduce method being used because in certain cases, the array value that is being evaluated will have more than one string contained in it. That is not what is happening in this case. Since it is only the `“##”` contained in the `row[0].text` array, it will only return the length of that which is 2.

Next the reduce method looks at the second `row[0]` which we can see is:

`{text: [“  ”]}`

Following the same logic we see that this also returns 2 which is the same as what was in the previous row. And since it was the same, the reduce method will return a 2 to the first element in the array that is created by the map method.

Next the value of `i` iterates to 1. Since we can also see that both objects in the second column also only contain strings that have length two, we know that the second element in the array returned by the map method is going to be 2 as well. From this we conclude that `colWidths` returns the following array and sends it to the widths variable:

`widths == [2,2]`

Step 5: Find the `blocks` value

Now the the values for `heights` and `widths` are set, the next thing that the `drawTable` function does is looks at the last line which is:

```
return rows.map(drawRow).join("\n");
```

We see that the map method is being used on the `rows` array and it is taking in the `drawRow` function as a callback. Lets take a look at `drawRow` to see what it is doing.

```
function drawRow(row, rowNum) {
    var blocks = row.map(function(cell, colNum) {
      return cell.draw(widths[colNum], heights[rowNum]);
    });
    return blocks[0].map(function(_, lineNo) {
      return drawLine(blocks, lineNo);
    }).join("\n");
}
```

The first thing that is happening is that it is creating "blocks". Each row is being mapped by the draw function, and takes in `widths[colNum]` and `heights[rowNum]`. This is looking at the `heights` and `widths` variables (that contain arrays) that we went through in Step 3 and 4. Since `drawRow` is the callback function of a map method, the second parameter, `rowNum` refers to the index number that is currently being iterated. The same thing goes for `colNum`. Since `cell.draw` is in a map within a map (much like a loop within a loop) it is going to draw all of the cells in the first row, before moving on to the next row. Let's look at the `draw` function.

```
TextCell.prototype.draw = function(width, height) {
  var result = [];
  for (var i = 0; i < height; i++) {
    var line = this.text[i] || "";
    result.push(line + repeat(" ", width -
line.length));
  }
  return result;
};
```

Let's look at what is happening step by step. The `rows` array

`[[{text: ["##"]},{text: ["  "]}],[{text: ["  "]},{text: ["##"]}]]`

is being mapped with the `drawRow` function. `drawRow` is going to take the first `row`

`[{text: ["##"]},{text: ["  "]}]`

and the first `rowNum` index will start at 0. The `blocks` variable is being defined by mapping the first `row`. `colNum` will start at 0. We know by understanding the map method that `cell` is referring to the object inside of `row`. The first cell in the first row is

`{text: ["##"]}`

so we know that the `draw` method is going to be working on this object like this:

`{text: ["##"]}.draw(widths[0], heights[0])`

If we look at the `widths` variable we see that it is referencing the value 2, and the heights variable is referencing the value 1.

`{text: ["##"]}.draw(2, 1)`

Now, looking at `TextCell.prototype.draw`, we see that there is a for loop that loops through the `height` of each cell and assigns the content of each line in a cell to a variable called `line`. If the cell is empty, it will assign an empty string. Before `line` is pushed to the `result` array, it is concatenated with spaces until it is the length of `width`. Since in our case, all of the cells contain lines that are the same length, the `repeat` function will never be used. Here is the `repeat` function so you can see how it works.

```
function repeat(string, times) {
  var result = "";
  for (var i = 0; i < times; i++)
    result += string;
  return result;
}
```

After the for loop terminates, the `result` array is returned and eventually returned again as part of the map that is saved in the `blocks` variable. In our case, the first time `drawRow` iterates, the variable `blocks` will look like:

```
[[“##”],[“  ”]]
```

with each of the inner arrays being what was returned by the `draw` method.

The second, and final time it iterates, blocks will look like:

```
[["  "],["##"]]
```

Step 6: Create each line of the table/checkerboard with the `drawLine`

function

Next we see that `drawRows` returns:

```
return blocks[0].map(function(_, lineNo) {
  return drawLine(blocks, lineNo);
}).join("\n");
```

`blocks[0]` is the first inner array in `blocks` and only contains one element since the cell (along with all of the other cells in this example) is only one line in height. It uses the `drawLine` function that along with `drawRow`, that was defined in `drawTable`.

```
  function drawLine(blocks, lineNo) {
    return blocks.map(function(block) {
      return block[lineNo];
    }).join(" ");
  }
```

If we look at our first iteration of `blocks` we see that `drawLine` has mapped

`[[“##”],[“   ”]]`

and joining them with a space. So it will look like:

`“##____”`

where the underscores represent the spaces. Note that the `.join(\n)` at the end of `drawRow` will not be used because the row has only one line.

Step 7: Draw the entire checkerboard

Finally, looking at the last line of the `drawTable` function, we should have all of the `rows` mapped out looking like:

`[“##___”,”___##”]`

with the underscores representing spaces. We see that the last thing `drawTable` does before returning this is it joins it together with a newline character so it will actually return this:

`“##___\n___##”`

When this is logged to the console, we will get our 2 x 2 checkerboard!

```
// → ##
//      ##
```

Conclusion

One last thing to note is that the checkerboard doesn't show what the drawFunction table is fully capable of because all of the cells are identical in that they all have a height of 1 and a length of 2. Because of this the repeat function isn't utilized and the drawLine function doesn't have to deal with cells that contain multiple lines. Despite this, I hope you were able to follow along.

If you are still confused by this example, you may want to check-out Gordon Zhu's Eloquent JavaScript: The Annotated Version. Also, if you have any requests for future blog posts or clarifications to this post, let me know in the comments.

Good Luck!

23 Comments **tomi.io** 1 **Login**

Recommend 2 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Jiehan Aldicho • 2 months ago
Thank you!
I took two days contemplating on this code and barely understand anything.
A glance of your delightful explanation totally save my day
Reply • Share ›

Tomio Mod → Jiehan Aldicho • 2 months ago
Nice! I'm happy I was able to help you out :)
Reply • Share ›

swastik sharma • a year ago
Tomio it is quite clear that you put a lot of effort in simplifying this devil .
I thank you for that .
Though i havent been able to wrap it up but i guess after reading it again everything's gonna be clear.
Thanks man for the time and effort you put into this.:)
Reply • Share ›

Tomio Mod → swastik sharma • 2 months ago
Sure! It definitely takes at least a few tries before things start to click. Good luck!
Reply • Share ›

Sergiu • a year ago
It took me several days to get this around my head and I could not have done it without your article.

It's quite harder than I thought, but seems I can slowly get somewhere (or so I hope). Nearly gave up reading the book at this chapter, you helped me a lot!

Thank you so much!!
Reply • Share ›

Tomio Mod → Sergiu • a year ago
Hey Sergiu. I was in the exact same boat as you, believe me! It can be quite disheartening when it takes days to feel like you understand this one section. As you can see, your struggle with this section of the book isn't unusual. If I were to guess, I would imagine that this section of

the book trips up the majority of beginners. Unfortunately, I think this causes beginner JS learners to give up with the book at this point, as you almost did. I'm glad you persevered and now feel like you can move on. I feel that anyone who can say that they understand this table generator has taken a solid step in their learning journey. Keep up the great work!

1 ^ | v • Reply • Share ›

Shane Levine • a year ago

Thank you so much for writing this up! It's just what I needed.

Unfortunately I'm stuck near the end where the map method is called on blocks[0]. Why is only the first array in the row (blocks[0]) called? That whole part confuses me.

^ | v • Reply • Share ›

Tomio Mod ➦ Shane Levine • a year ago

Hi Shane. Yeah, no worries! It is super confusing and the reason why it is so confusing is because that is the part of the code that handles the situation where a cell may be more than one line in height, which my checkerboard example doesn't cover.

The reason why only the first array (think of it as the first cell) in the row is used is because the height of that cell is the same height as all of the other cells in that row. We know that the heights of all of the cells in the same row are the same because we determined what those heights are with the rowHeights function.

So again, we only need to map blocks[0] because block[0] is an inner array whose elements each represents a line in a cell, and we only want the map to iterate the same number of times as the height of the cell. So in the checkerboard example I use above, the height is only 1 for each blocks, so we only want to map once to because we only have one line to draw for each row of cells.

Now say instead our rows array looked like:

see more

^ | v • Reply • Share ›

Shane Levine ➦ Tomio • a year ago

Yes!! I get it!

Thank you so much--that was a really big "a ha" moment for me.

^ | v • Reply • Share ›

Tomio Mod ➦ Shane Levine • a year ago

Awesome! I love those ah ha moments!

^ | v • Reply • Share ›

Jewa ➦ Tomio • a year ago

Just a little confusion:

First off your 2*2 confuses me a bit. When drawLine is called inside drawRow it seems to be adding a space between [00] and [10] instead of [00] and [01] maybe this works because the array is symmetric but won´t if it is asymmetric. I guess I am muddling things up.

Lastly, you confuse me when you say the .join("\n") at the last line of the drawTable function does nothing (I totally agree with this) because I think your 2*2 is ready for printing the moment drawLine is returned under the blocks[0] statement. However in step 7 you say the last thing drawTable does is to return the array with the new line joined which I read to mean that the .join method in the last line of drawTable still executes I think this is not possible since there is no comma or am I reading it wrongly and what you refer to is the .join called inside the drawRow.

I think you did a good job. I also like thinking like a compiler but this was bit beyond me and I doubt I would have made it this far without your write-up.

• Reply • Share ›

Jewa ➦ Jewa • a year ago

update: I think I understand everything now and all the questions above were a product of total confusion. I had to take a break and read it a second time. I was about to ask a question as to why blocks[0] because blocks[0] might have a height of 2 and blocks[1] and height of 3 but this is not possible because heights uses max height.

Thanks once more for helping me solve this mystery.

• Reply • Share ›

Tomio Mod ➦ Jewa • a year ago

Hey Jewa. Sorry for the delayed reply. Life's been a bit busy.

It's a little funny that every time someone asks me a question about this, I actually do have to run through my writeup myself to understand it again. After a few months my understanding also kind of fades from memory!

There is always the possibility that I have made a mistake, and I haven't reviewed the example and checked it against your comment yet, but it does seems like you have figured it out yourself. I am very well aware how inefficient a tired brain is and how things can become immediately clearer after getting some sleep or taking a break. It happens to me on a daily basis!

If you want to try and make sure that you are

see more

Reply • Share ›

Glendon Philipp Baculio • 2 years ago

God, this is so hard, too hard to wrap in head in a day. what good books do you recommend pre-eloquent javascript? some says it's intermediate but I think it's Godlike level!

Reply • Share ›

Tomio Mod → Glendon Philipp Baculio • a year ago

Hey Glendon. Yeah, I agree that it is super hard. I was only able to figure it out the first time by forcing myself to think like a compiler and go through the code step by step and keep track of the state of everything with a pen and paper. The problem is that it's just plain complicated and it's really hard to keep track of everything in your head.

As far as books go, I highly recommend the "You Don't Know JS" series by Kyle Simpson. I think if I could choose only one resource for learning JavaScript it would be these books. Amazingly, Kyle has all of the content available for free on his GitHub here https://github.com/getify/Y..., but you can also get the books on Amazon if you prefer hard copies.

Best of luck with your JS journey! Let me know if there's anything else I can help you with.

Reply • Share ›

nickyNohands • 2 years ago

Thank you Tomio! I finally understand this exercise after almost 2 weeks of trying to wrap my head around it..

Reply • Share ›

Tomio Mod → nickyNohands • 2 years ago

Hey! I'm glad you were able to finally figure it out. It certainly is very tricky. Best of luck to you!

Reply • Share ›

L0dz • 2 years ago

Thank you so much Tomio. You saved my day !

Reply • Share ›

Revels0k • 2 years ago

Thank you. Helped so much! You're the hero.

Reply • Share ›

Daniel • 2 years ago

Thank you for the incredibly thorough explanation. Helps a lot!

Reply • Share ›

Tomio Mod → Daniel • 2 years ago

Awesome, Daniel! I'm happy you found it useful.

Reply • Share ›

Tomio Mod • 2 years ago

You're welcome! I'm glad I could help out.

Reply • Share ›

JBO • 2 years ago

Thank you SO much for this! I kept looking at this code like "wha . . ?". You cleared it all up for me!

Reply • Share ›

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Privacy**

Tomio Mizoroki

Read more posts by this author.

Share this post

tomicode © 2018 Proudly published with **Ghost**