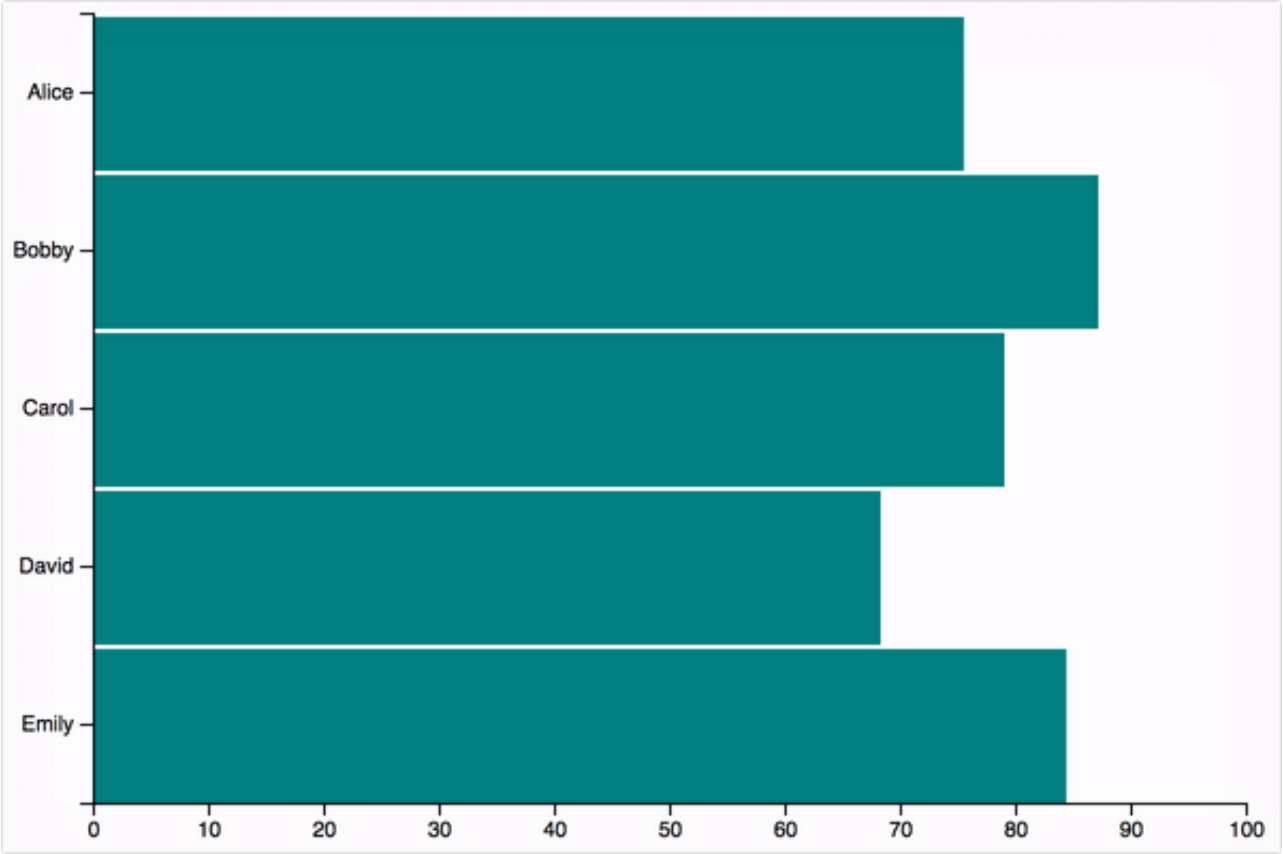
D3 Day 1: Building Blocks

D3.js is the de facto JavaScript library for creating data visualizations on the web. Companies like The New York Times, FiveThirtyEight, Netflix, and countless others use it extensively to visualize complex relationships and trends, examine geographic changes over time, and to gain insights into mountains of internal data.

Unfortunately, D3 also has a bit of a reputation as being hard to learn, but it doesn't have to be. D3 is built on a small handful of core concepts and once you understand them you can begin working toward mastery. It just so happens this course is all about those core concepts.

You'll learn those foundational topics while creating a slick, animated bar chart like this one.



Prerequisites

D3 is built on the fundamental web standards of HTML, CSS, SVG, and JavaScript. For the purposes of this course it is only assumed you have a basic understanding of HTML, CSS, and JavaScript. SVG will be taught from first principles as needed.

HTML

HTML and its DOM model, of course, provide the foundation on which any web based project must run. D3 provides terse APIs for interacting with the DOM, and some D3 projects are built entirely with native HTML elements. That said, most D3 visualizations create SVG output, which we'll discuss briefly below.

CSS

Whether your project targets the DOM or SVG, you will use CSS to style it. D3 provides APIs for adding and removing CSS classes as well as manipulating individual style properties, so as long as you know the basics you shouldn't have any trouble.

SVG

SVG is a "language for describing two-dimensional vector and mixed vector/raster graphics in XML". Really rolls off the tongue, doesn't it?

All you really need to know is that vectors provide resolution independence, allowing you to create graphics that look good everywhere from phones to giant desktop monitors. Additionally, SVG graphics do not participate in the browser's standard layout flow.

What does that mean? Well, when you create an SVG element using markup like `<svg width="300" height="300"></svg>`, that 300x300 block will appear in the browser's layout flow just like any other element. If the tag was defined after a `<div>` tag, the SVG will be rendered below the div. Tags defined after the SVG tag will be rendered below it. *Within that block*, however, there is no automatic layout to prevent items from overlapping. If you create two circles with the markup `<circle r="20" /><circle r="20" />` they will be rendered in the same exact spot, right on top of one another. While this may seem inconvenient at first, it actually provides an incredible amount of flexibility. Avoiding overlap requires explicit coordinates but SVG graphics are capable of extraordinary expressiveness through layering, opacity, interactivity, and fluid animation.

D3 Core Concepts

D3 stands for Data-Driven Documents, which really does describe its underlying philosophy. It provides a declarative API for creating and transforming a document's structure based on arbitrary underlying data. Whether your data is a simple array of numbers or a complex graph of JSON objects and arrays, D3 enables you to transform that data into a standard web document using declarative APIs.

When we say D3 is declarative, what we mean is that you tell D3 what you want, but not how you want it done. You will not write loops to create a collection of elements, you will simply describe a relationship between a set of data points and a set of UI elements, and D3 will handle the implementation details. For example, to turn all of the paragraph text on your page white, you could use the code `d3.selectAll('p').style('color', 'white')`. D3 will handle the mechanics of iterating over DOM elements and the updating of attribute values for you.

While this abstraction alone is helpful in terms of productivity and avoiding browser inconsistencies, the real power comes from the integration of dynamic data. From the D3 site, "styles, attributes, and other properties can be specified as *functions of data* in D3, not just simple constants". What does this look like in practice?

The following code will apply the number values being passed to the `data()` method as the font size to existing paragraph elements on the page.

```
d3.selectAll('p')
  .data([4, 8, 15, 16, 23, 42])
  .style('font-size', (d) => { return d + 'px' })
```

Selections

Selections are one of D3's core concepts, and understanding them will go a long way to making you feel comfortable working in D3. The next lesson is devoted entirely to selections, but these syntax examples will give you an idea of how things work and what to expect.

```
d3.select('#foo')        // selects the element with an id="foo" attribute
d3.selectAll('.primary') // selects all elements with CSS class primary
d3.selectAll('div')      // select all divs on the page
```

`d3.select` and `d3.selectAll` operate on the page as a whole, but they return selection objects that allow you to narrow the scope of subsequent selections. For instance, if we were to change the first selector above to `d3.select('#foo').selectAll('p')` we would only get back the paragraphs that live *inside* the `#foo` element. That means you can do things like turn all your nav links red with a one-liner like `d3.select('nav').selectAll('a').style('color', 'red')`.

The rabbit hole can go pretty deep, but we'll stop here for now. The next lesson is all about selections!

Scales

Data visualization is ultimately about transformation. Quarterly revenue becomes column height, the winning political party becomes a state's fill color on a map, and on and on. In D3 these transformations are done using scales, and they're another one of D3's core concepts.

Linear scales are the most straightforward as they directly translate values from one context to another.

```
const percentToDecimal = d3.scaleLinear()
    .domain([0, 100])
    .range([0, 1])
```

Calling `d3.scaleLinear()` will instantiate a new linear scale. We then call two methods on the scale, passing each of them a two-element array representing the minimum and maximum values. The `domain()` method is used to specify the possible input values while `range()` specifies the corresponding output values. The return value of this code is a *scale function* named `percentToDecimal`. This function will accept a percentage value and return the corresponding decimal.

```
percentToDecimal(0)   // 0
percentToDecimal(50)  // 0.5
percentToDecimal(100) // 1
```

Again, this is just the smallest tip of the iceberg. In lesson four we'll see how to use scales for numbers, strings, dimensions, and even colors!

Community and Resources

Last but not least, there are some great resources out there for learning, sharing, and exploring D3.

One of the oldest and absolute best is https://bl.ocks.org/ by Mike Bostock, the creator of D3 itself. It's a treasure trove of cool demos where the code used to create them is displayed right below the output. Each demo is backed by a GitHub Gist, so you can go to the original source, fork it, tweak the code and play with your own version.

http://blockbuilder.org/ was created by Ian Johnson as a sort of next generation https://bl.ocks.org/ and will even load the same examples by simply changing the domain in the URL. It has the added benefit of a code editor and live preview right there in the browser so you can tinker with a very tight feedback loop. Do not underestimate the power of changing things and seeing what happens. It can be a powerful way to learn by example!

Lastly, there is a D3.js Slack group that is pretty active and very welcoming. There are channels dedicated to getting help when you're stuck, the broader effort of learning D3, and many more.

Whew, that was a lot!

We covered a lot of ground for a first lesson! Next we'll go deep on selections, then build a real data-driven chart, followed by scales, margins, and axes.

If you have any questions at all, about something in this lesson or anything else related to D3 just hit reply. I'm here to help!

D3 Day 2: Selections

Selections are at the heart of D3. So much so that it's literally impossible to use D3 without them. We saw a few basic examples of selections in the previous lesson.

```
d3.select('#foo')        // select the element with an id="foo" attribute
d3.selectAll('.primary') // select all elements on the page with a CSS class
of primary
d3.selectAll('div')      // select all divs on the page
```

We also saw how selections can be scoped by calling `select` and `selectAll` on a selection object rather than directly on `d3` itself. `d3.select('#foo').selectAll('p')` would only select paragraphs inside the `#foo` element, `d3.select('nav').selectAll('a')` would only select links within the `nav` element, and so on.

Believe it or not, the examples above cover most of the ground when it comes to purely selecting elements on a page.

If you want to select	One Item	Multiple Items
Anywhere on the page	d3.select	d3.selectAll
Within an existing selection	selection.select	selection.selectAll

The choices in that matrix will cover 95% of your use cases when working with D3. If you've worked with `document.querySelector` and `document.querySelectorAll` before you should feel right at home.

You may notice some similarities to jQuery as you read through the code samples below. Trust me when I say the differences will be crystal clear after the next lesson.

Element CRUD

This is where the real fun begins. This is also where we'll learn about a common pattern in D3 APIs.

selection.attr(name[, value])

selection.attr is for working with element attributes. When calling `selection.attr(name)` the method will act as a getter and return the `name` attribute's value. `selection.attr(name, value)` will cause the method to act as a setter, assigning `value` to the `name` attribute.

```
d3.select('a').attr('href') // get the href value of the first link on the
page
d3.select('a').attr('href', 'http://google.com') // set the first link's href
to Google
```

Methods as dual purpose getters and setters is something you will see and use a lot.

The `value` parameter can also be specified as a function. This is useful when you need to set a dynamic value and is also extremely common in D3 code. Additionally, the functions you pass to D3 methods will almost always take the same exact form:

```
d3.selectAll('p').attr('href', function(d, i, nodes) {
  // the first parameter is named d and is the node's data point, or datum
  // the second parameter is named i and is the node's index in the selection
  // the third parameter is an array of the nodes in the selection
  // lastly, `this` === nodes[i]
  // the function is bound to the DOM element, so `this` refers to the actual
node
})
```

I repeat, *you will see this everywhere*.

It's worth noting that, because of the differences in lexical scoping, `this` will not refer to the DOM element if you use an arrow function. If you use an arrow function you will need to use `nodes[i]` to get a reference to the element.

selection.style(name[, value])

Look familiar? selection.style is essentially a mirror image of `selection.attr`, it just deals with styles rather than attributes.

```
d3.select('a').style('color') // get the text color of the first link
d3.select('a').style('color', 'red') // set the first link's text color to red
d3.select('a').style('color', function(d, i, nodes) {
  return getColorByURL(this.href) // this is the native DOM element
})
```

selection.property(name[, value])

selection.property is for boolean properties like `checked` on checkboxes, and special properties like `value` on an `input` field. It is otherwise identical to `selection.attr` and `selection.style`.

selection.classed(names[, value])

selection.classed is for getting and setting CSS class names, and yet another method that uses the pattern we keep seeing. It has a slight wrinkle, but the basics hold truthy.

The first parameter is `names` rather than `name`, because you can specify a space-separated list of class names. Since CSS classes are either applied or they're not, `value` should be coercible to a boolean, or a function that returns something coercible to a boolean.

```
d3.select('#submitBtn').classed('btn-lg') // does submitBtn's classList
include btn-lg?
d3.select('#submitBtn').classed('btn btn-lg', true) // apply the btn and btn-
lg classes
d3.selectAll('button').classed('btn', function(d, i, nodes) {
  return i < 3 // apply the btn class to the first 3 buttons
})
```

Note that most of the time you will see classes applied using the `selection.attr('class', 'btn btn-lg')` syntax. This is functionally identical to `selection.classed('btn btn-lg', true)`, but `selection.classed` is generally reserved for cases where a class is being conditionally applied or toggled on and off.

selection.text([value]) && selection.html([value])

These methods are essentially aliases. selection.text for `innerText` and selection.html for `innerHTML`.

```
d3.select('#someDiv').text() // get the innerText of #someDiv
d3.select('#someDiv').text('Hello World!') // set the innerText of #someDiv
d3.select('#someDiv').html() // get the innerHTML of #someDiv
d3.select('#someDiv').html('<h2>Hello World!</h2>') // set the innerHTML of
#someDiv
```

selection.append(type) && selection.insert(type[, before])

selection.append accepts a tag name or a function, and will create a new element and attach it as the last child of each node in the selection. For example, `d3.select('body').append('p')` will add a paragraph to the end of the page, but `d3.selectAll('p').append('span')` will add a `span` to every paragraph on the page.

Similarly, selection.insert takes an optional `before` parameter, allowing you to pass selector strings like `:first-child` to have the new item inserted before the first child of each item in the selection. If you don't provide the `before` parameter, `selection.insert` will behave just like `selection.append`.

selection.remove()

Can you guess what selection.remove does? That's right, it removes any elements in the selection from the DOM.

Handling Events

Last but not least there is selection.on(type, listener). This allows you to create simple event handlers by passing a DOM event type and a handler function.

```
d3.selectAll('button').on('click', handleButtonClicks)
d3.select('body').on('mousemover', trackMousePosition)
```

There's a lot more about event handling in the docs, but just knowing the `on` method will get you well on your way.

Are We Done Yet?!

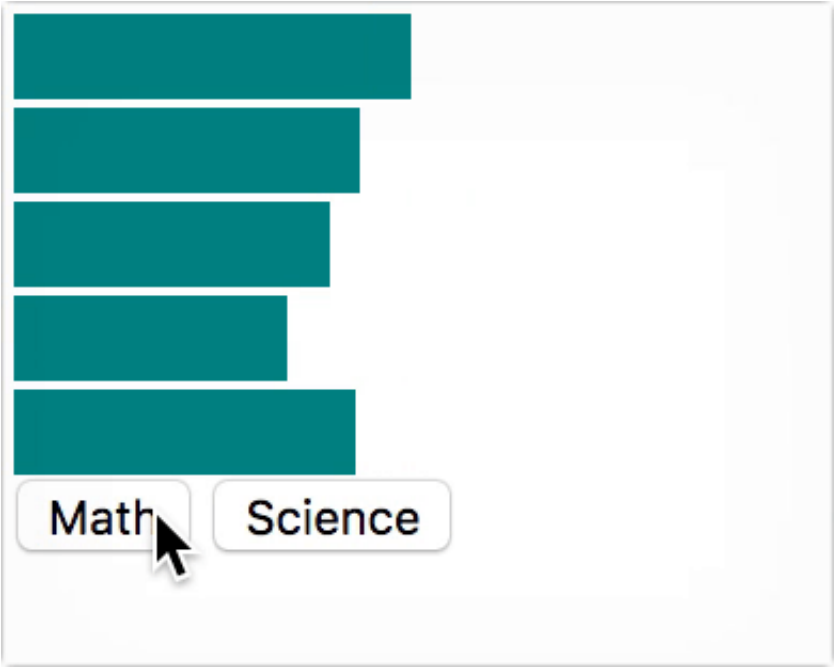
Yes, we are done for today. Sit back, relax, and ruminate on your newfound knowledge! Also, don't worry if this all feels a bit overwhelming. It's hard to digest this much information in a single sitting, but my hope is that by having seen it you will be better equipped to review and tinker with code samples around the web.

Worried that something isn't clicking like you think it should? Feel like something above was explained poorly? Please let me know! This is a lot of new stuff, and I want to help you make sense of it.

More next time!

D3 Day 3: Building a real bar chart

Fundamentals are important but let's face it, you're learning D3 so you can build cool stuff, right? Maybe an animated bar chart like this one?



Well do I have good news for you! 😆 By the end of today's lesson you'll have done just that, and you'll actually *understand how it works*.

Driving with data

In the previous lesson I mentioned that some D3 APIs resemble jQuery, and that's true when you are getting and setting simple values. But when it comes to controlling a document or visualization based on underlying data, D3's power and expressiveness are truly unparalleled.

Remember, D3 stands for Data-Driven Documents. The way you do that is by *joining data and selections*. The fundamental API for this is selection.data which accepts an array of data items and joins them to the DOM elements in the selection.

Take the following bit of HTML:

```
<div id="chart">
  <div></div>
  <div></div>
  <div></div>
  <div></div>
  <div></div>
</div>
```

We can populate those empty divs by selecting them and then joining data.

```
const data = [ 90, 70, 50, 30, 10 ]

d3.select('#chart')
  .selectAll('div')
  .data(data) // pair each number in the array with an empty div
    .attr('class', 'bar') // color, height, and spacing via CSS
    .style('width', function(d) {
      return d + 'px' // use the data items as pixel widths
    })
```

Check it out for yourself and you'll see this:

What has happened is that D3 has assigned each item in our `data` array to one of the empty divs in the selection. By default, items will be matched based on index. That's why the first div was given a width of 90, the second 70, etc.

Sweet! You built your first bar chart with D3!

What's with those empty divs?

OK, so, we kind of cheated a little bit. In the example above, there just so happened to be five empty divs in the `#chart` container and five items in our data array. I know, what are the odds, right?

Let's talk about how things will work in the real world. D3 has a concept called the enter selection that represents the data items that could not be paired with existing DOM elements.

A more realistic starting point for the HTML would be `<div id="chart"></div>`, a single, empty div. In that case, our existing code of `d3.select('#chart').selectAll('div').data(data)` would return an empty selection since there are no divs inside `#chart`. However, that new selection has an `enter()` method which returns the enter selection. Since the enter selection represents data items in need of DOM elements, we can use it to create exactly that.

Let's update our previous example to take advantage of the enter selection, allowing us to target an empty container like `<div id="chart"></div>`.

```
d3.select('#chart')
  .selectAll('div')
  .data(data)
  .enter() // gimme the enter selection and those poor orphaned data items
    .append('div') // create and append a new div for each data item
    .attr('class', 'bar')
    .style('width', function(d) {
      return d + 'px'
    })
```

Looks familiar, eh? The only changes from our first example are the two commented lines.

Every single chart you create will include this pattern. `selection.data` will pretty much always be followed by a call to `enter()`, and that will always be followed by a call to `append()`. In fact, if your data doesn't change after your page loads, you're basically done.

By the way, remember how we talked about declarative code and telling D3 *what you want*, but not *how to do something*? That is exactly what we're doing here. We are not looping over the data, we are not telling D3 how many elements we want. We are simply saying "The divs and data go together. You figure out the details."

Dynamic charts

What if your chart's data changes after the initial render? In that case we can follow the same basic pattern as above, we'll just need to reset the container each time we render new data. For this example we'll use a slightly more complex set of data. Here we have an array of students who each have a name and two different scores.

```
const data = [
  { name: 'Alice', math: 93, science: 84 },
  { name: 'Bobby', math: 81, science: 97 },
  { name: 'Carol', math: 74, science: 88 },
  { name: 'David', math: 64, science: 76 },
  { name: 'Emily', math: 80, science: 94 }
]
```

We'll also put a couple of buttons on the page to use for switching which subject is being rendered.

```
<button onclick="render('math')">Math</button>
<button onclick="render('science')">Science</button>
```

Notice that the strings we pass to the `render` function match the property names in our data objects. Now all that's left is to implement the render function, which should look pretty familiar.

```
function render(subject) {
  d3.select('#chart') // within the #chart container...
    .selectAll('div') // select all the divs
    .remove()         // and remove them, clearing the #chart div

  // now just repeat our creation pattern from the first example
  d3.select('#chart')
  .selectAll('div')
  .data(data)
  .enter()
    .append('div')
    .attr('class', 'bar')
    .style('width', function(d) {
      return d[subject] + 'px' // use the subject passed in to grab the
correct property
    })
}

render('math') // render the math data when the page first loads
```

You can check out the live example and click the buttons to switch subjects! No page reload necessary, D3 simply updates the necessary DOM elements and you have a dynamically changing bar chart.

Uhh, that's not animated

I know, I know. We're getting there. Thankfully, there are only a few tweaks we need to make to animate our chart updates.

In our previous example we just deleted the chart and redrew it each time the data changed. That's the quickest and easiest way to do things, but it's not compatible with animated transitions. After all, if we want a bar to animate from one size to another, deleting it is not exactly a helpful first step.

The key to reusing elements instead of deleting them is actually called a *key function*, and it's the optional second argument when calling `selection.data`. The purpose of a key function is to ensure the same DOM element will render the same data item between renders, overriding the default match-by-index behavior we've previously seen. In practice this is done by referencing a property that is consistent between updates, which in our case is the name property.

```
- .data(data)
+ .data(data, function(d) { return d.name })
```

Let's look at an updated version of our `render` function.

```
function render(subject) {
  // store a reference to the bars already on the chart
  const bars = d3.select('#chart')
    .selectAll('div') // this won't be empty after the first time this
function runs
    .data(data, function(d) {
      return d.name // use the name property to match across updates
    })

  const newBars = bars.enter() // use the enter selection
    .append('div') // to add new bars for any data items without an existing
DOM element
      .attr('class', 'bar')
      .style('width', 0) // set the initial width to 0

  // combine the selections of new and existing bars
  // so you can act on them together
  newBars.merge(bars)
      .transition() // animate everything that comes after this line!
      .style('width', function(d) {
        return d[subject] + 'px' // set the width like normal!
      })
}
```

That's it! Sit back and bask in the glory of creating an animated bar chart in D3!

Additional Resources

We covered a lot of ground today. Also, full disclosure, I glossed over some things to avoid going too far down the rabbit hole. If you feel comfortable with where we ended up that's fantastic. In the next two lessons we'll just be fleshing out this example to make it more flexible and polished, but you're over the biggest hurdle.

On the other hand, if you're the type of person that likes to really dig in and learn the underlying mechanics, I've collected a few links for you. These resources are a great way to dive deeper, but you definitely shouldn't feel obligated to do so.

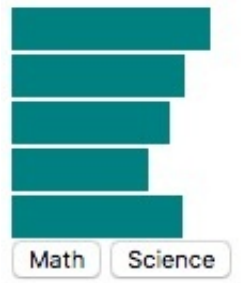
- Thinking with Joins
- Joining Data
- Object Constancy

Either way, if you have any questions at all, hit reply and let me know if there's something I can clear up!

D3 Day 4: Scales

In the previous lesson we got our first taste of data-driven documents. We took a set of objects representing students and scores and we turned them into bars on a web page. The final output can take a million different forms, but the fundamental tasks of transformation and representation are the core of data visualization.

In our example, however, we mostly relied on representation. We directly used the test scores to set our bar widths, resulting in this tiny chart hiding in the corner. It looks so lonely! Why not stretch out and relax, little bar chart buddy?



Lucky for us, another one of D3's fundamental concepts is devoted entirely to transformations. Today we're going to learn about scales, and we'll use them to turn our meek little bar chart into a hulking beast capable of growing as big as you tell it.

Before we start bulking things up though, I want to walk you through the basics. Scales will be a key aspect of every project you create, so it's worth the effort to make sure you really understand them.

Anatomy of a scale

D3 provides more than 10 types of scales, but they all have two common, fundamental parts: a domain and a range. We'll start with linear scales because they do a direct, proportional mapping from one context to another.

```
// instantiate a new linear scale
// we expect the input values to be between 0 and 100
// and we want to transform them to values between 0 and 1
const percentToDecimal = d3.scaleLinear()
    .domain([0, 100])
    .range([0, 1])
```

When you create a scale the return value is a *scale function*. This function will accept a value from the configured `domain` and return the corresponding value from the `range`.

```
percentToDecimal(0)   // 0
percentToDecimal(50)  // 0.5
percentToDecimal(100) // 1
```

Reversible

What if we wanted to go the other way? One option would be to create another scale, with the domain and range swapped.

```
const decimalToPercent = d3.scaleLinear()
    .domain([0, 1])
    .range([0, 100])

// now we can go the other way
decimalToPercent(0.5) // 50
decimalToPercent(1)   // 100
```

Another way is to use the invert method. The invert method will accept a value from the range and return the corresponding value from the domain. Using our original scale function `percentToDecimal`, we can now convert decimals to percents as well.

```
percentToDecimal.invert(0)   // 0
percentToDecimal.invert(0.5) // 50
percentToDecimal.invert(1)   // 100
```

What about outliers?

When we defined `percentToDecimal`, we said the domain (possible input values) would fall between 0 and 100. But there is nothing stopping anyone from passing values outside those bounds to the function, so let's see what happens.

```
percentToDecimal(-10) // -0.1
percentToDecimal(150) // 1.5
percentToDecimal(200) // 2
```

Hmm. Nothing blew up, but now the return values are outside the range we defined. Depending on what you're using your scale for, that might be just fine. But what if your scale is determining the width of bars on a chart? A bar can't have a negative width, and you don't want it running off the right side of your chart either.

In these scenarios you can call the clamp method on your scale to prevent it from returning anything outside the defined range.

```
const percentToDecimal = d3.scaleLinear()
    .domain([0, 100])
    .range([0, 1])
    .clamp(true)

percentToDecimal(-10) // 0
percentToDecimal(150) // 1
percentToDecimal(200) // 1
```

That's it.

This is not an exhaustive look at scales, of course, but those are the fundamentals. If the idea of converting from one context to another, in either direction, and handling outlier data makes sense, congratulations, you understand scales.

Scales in practice

The most common use for scales is adapting charts to fill the available space. In the previous lesson we saw code that used test scores as column widths: `.style('width', function(d) { return d[subject] + 'px' })`. This certainly works, but it produces a pretty small chart. What if we wanted our chart to be 600 pixels wide? The test scores we're plotting will always be between 0 and 100, but we need our bars to potentially be much wider. Scales to the rescue!

```
// specify chart dimensions
const width = 600
const height = 400

// create a scale to map scores to bar widths
// test scores are stored as percentages
// a score of 100 should create a full-width bar
const xScale = d3.scaleLinear()
  .domain([0, 100])
  .range([0, width])

d3.select('body')
  .selectAll('div')
  .data(data)
  .enter()
    .append('div')
    .style('width', function(d) {
      // pass the score through the linear scale function
      return xScale(d[subject]) + 'px'
    })
```

Now instead of our bars always being 0 to 100 pixels wide they will be from 0 to 600 pixels wide. A score of 50 will produce a width of 300, etc.

What about that height?

To calculate the height of our bars we need to use a special type of scale called a band scale. Band scales are specifically for bar and column charts, and ours looks like this.

```
const yScale = d3.scaleBand()
  .domain(data.map(function(d) { return d.name }))
  .range([0, height])
```

The domain of band scales takes the full list of values that will be present, so we create an array of the student names in our dataset. Since we want our bands to fill the specified height we use that value to set the range.

We can then set our bar heights by calling a method directly on the band scale.

```
.style('height', function(d) {
  return yScale.bandwidth() + 'px'
})
```

The scale knows how many values there are and how much space is available. That means it can calculate how big each element should be, and we get that value by calling the `bandwidth()` method.

Let's see it

You can see all of these things applied to our bar chart here. Check it out and play with the `width` and `height` definitions and see if you can anticipate how the output will change.

That's basically it

Not too bad, right? There are many more scale types in D3 (and you can roll your own!) but they all follow this same basic formula. You pick a scale type, define what type of data will go in and what should come out, and in return you get a function that will convert values in the manner you specified.

Once you understand these fundamental building blocks you are well on your way to being productive with D3 scales! As always, let me know if you have questions!

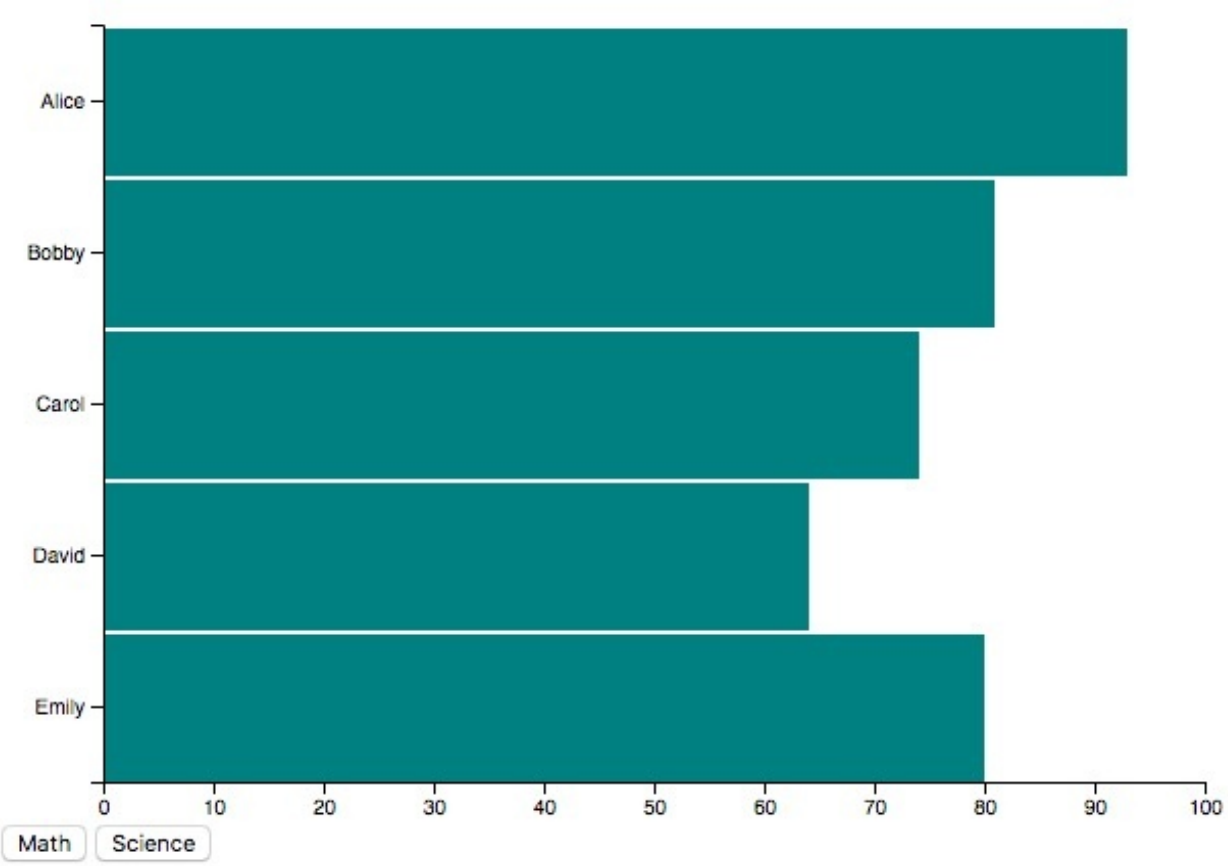
D3 Day 5: Adding context with axes

Welcome to Day 5! Today's lesson is the last in our series. 😢 On the bright side, we get to put the finishing touches on our bar chart by adding some proper axes!

To recap our progress up to this point, we learned the fundamental concept of D3 selections, put it into practice to create a bar chart, and then learned about scales, another core concept in D3.

At the end of our last lesson we had a nice bar chart that can be sized to any dimensions we please. It was a great step forward, but we're still lacking context. Based on the output alone there is no way to know what the bars represent. Are we looking at countries and their populations? Factories and their production capacities?

Avoiding those types of questions is the job of axes. By the time we're done today, our chart will look like this gem.



Axes come from scales

In the previous lesson we defined our dimensions and created scales to control the size of our chart. As a reminder, this is what that code looked like.

```
// specify chart dimensions
const width = 600
const height = 400

// create a scale to map scores to widths
const xScale = d3.scaleLinear()
  .domain([0, 100])
  .range([0, width])

// create a scale to calculate bar height
const yScale = d3.scaleBand()
  .domain(data.map(function(d) { return d.name }))
  .range([0, height])
```

The good news is that creating axes is just a matter of passing our scales into some built-in D3 methods. The not-exactly-good-but-don't-sweat-it news is that D3's axes must be built using SVG, and we've not really covered that yet.

Welcome to your SVG crash course, it's really not that bad. The first thing we'll do is create the SVG tag itself inside our `#chart` container div.

```
const svg = d3.select('#chart')
  .append('svg')
  .attr('width', width)
  .attr('height', height)
  .style('position', 'absolute')
  .style('top', 0)
```

Given our dimensions defined earlier, this will give us the following markup.

```
<svg width="600" height="400"></svg>
```

Notice that we're also using styles to absolutely position the SVG element. We need to do this so it's not laid out further down the page, after the bar divs. Absolute positioning will cause our SVG element to effectively be laid directly on top of the bars. This wouldn't be necessary if we had built a pure SVG chart, which is actually more common (but also more difficult) than using divs like we did. Should we could cover that in a future course?

Next we can create the axes using the built-in methods D3 provides.

```
svg.append('g').call(d3.axisBottom(xScale))
svg.append('g').call(d3.axisLeft(yScale))
```

Each line starts with `svg.append('g')`, which creates an SVG group element to hold each axis. You can think of `g` elements as the SVG equivalent to divs. They're just a container with no inherent size or appearance of their own.

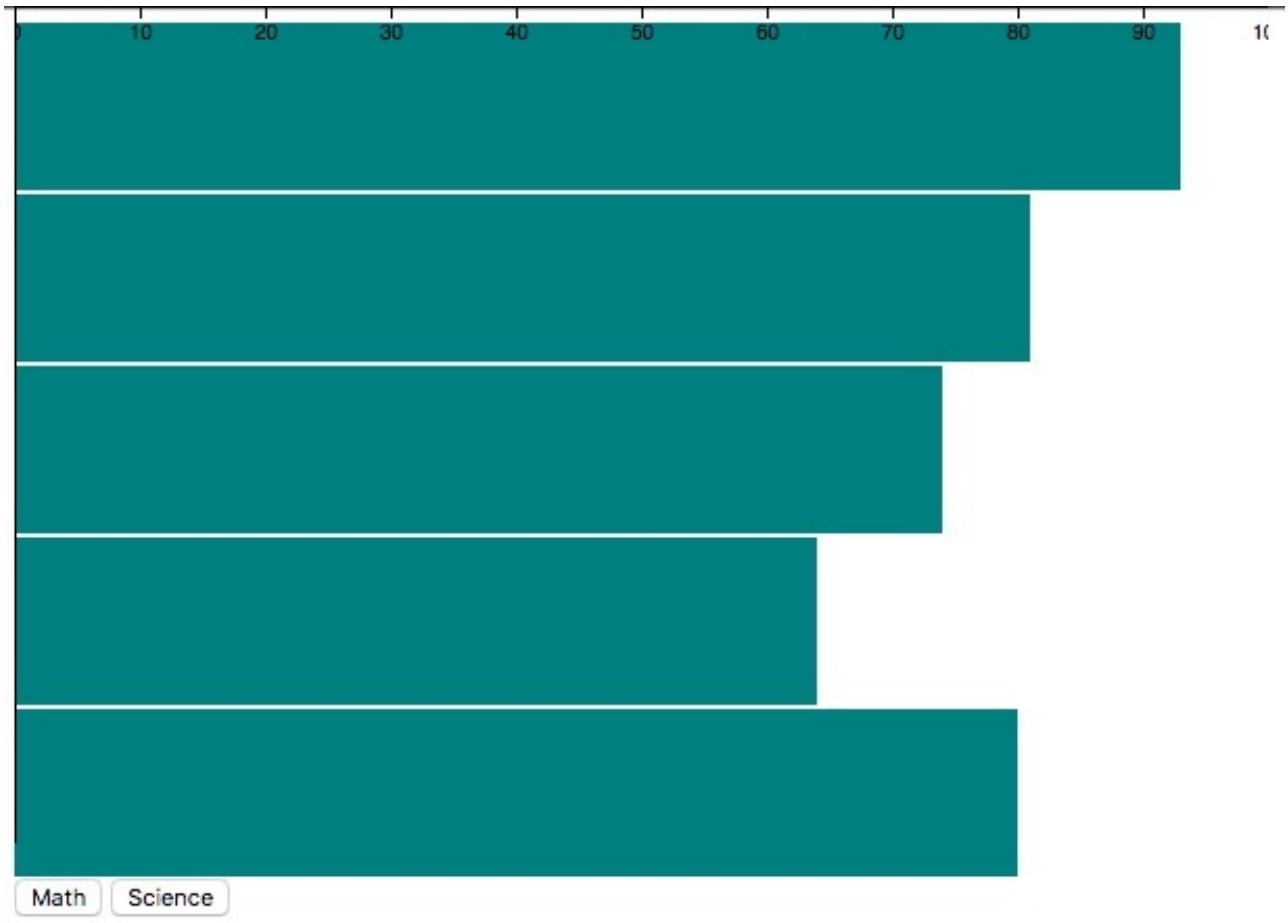
Inside each of those containers we've called a D3 axis method, passing in one of our scales. We passed our X scale to `d3.axisBottom` since we want the labels and ticks below the axis line itself. That said, you can probably guess why we used `d3.axisLeft` for our Y scale.

You haven't seen selection.call before. You don't really need to understand it right now, but it's basically just a way to enable chained operations on a selection. (If you'd like to know more just hit reply and let me know.)

We now have markup that looks something like this.

```
<svg width="600" height="400" style="position: absolute; top: 0px; left:
0px;">
  <g><!-- D3 creates the X axis in here --></g>
  <g><!-- D3 creates the Y axis in here --></g>
</svg>
```

Sweet, we've created our axes! Let's take a look!



Hmm, that's not exactly what we want. The X axis is sitting at the top of our chart, and we can only see the tiniest sliver of Y axis on the left edge of the chart.

Moving the axes into view

Like most UI coordinate systems, SVG's default position is the top left corner. This is why the starting point for both of our axes were drawn there, and they're not appearing where we want them.

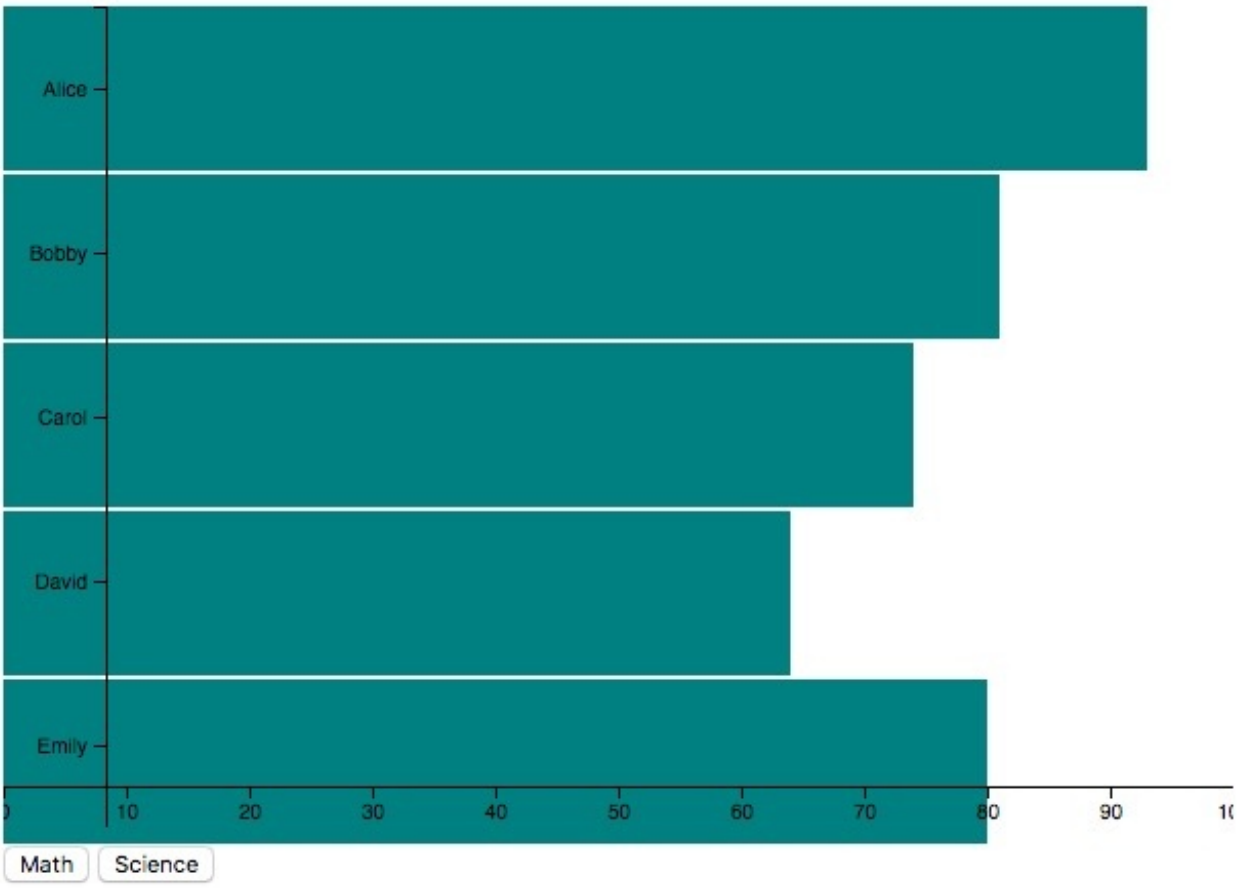
Somewhat unique to SVG is that positioning is not done by setting `top` or `left` styles. To move our `g` elements, and therefore the axes constructed inside them, we need to set their `transform` attribute. What's more, the value of that attribute is a call to a `translate` function that accepts X and Y coordinates, similar to CSS transforms you may have used before. You can think of `translate` as an alias for "move", which is what it really does. Let's update our code to move our axes where we want them.

```
svg.append('g')
  .attr('transform', `translate(0, ${height - 20})`)
  .call(d3.axisBottom(xScale))

svg.append('g')
  .attr('transform', `translate(50, 0)`)
  .call(d3.axisLeft(yScale))
```

We've now specified that, before drawing our X axis, its containing element should be moved down to 20 pixels above the bottom of our chart as defined by our `height` property. We've also moved our Y axis 50 pixels to the right. (If you've not seen template literals before they're just a way to avoid string concatenation in JavaScript.)

OK, what does our chart look like now?



Hmm, that's definitely better, but we still have some problems. The axes are right on top of the bars, the X axis is getting cut off on the sides, and the ends of the axes don't meet in a single point like you'd expect.

D3 Margin Convention to the rescue

Since this is such a common problem, the D3 community has settled on a repeatable way to solve it, referred to as the margin convention.

When creating a D3 chart with axes, convention dictates that we use a *margin object* to define the amount of space to reserve on each side of the chart for axes.

```
// we know most of the space we need is below and to the left of the chart
const margin = { top: 10, right: 10, bottom: 20, left: 50 }
```

You then subtract the margins when defining your `width` and `height` properties. This is done so their values will hold the actual usable space for the chart.

```
const width = 600 - margin.left - margin.right
const height = 400 - margin.top - margin.bottom
```

Here we've updated our existing dimension definitions to include our margins. Given the margin object we defined above, `width` will now be `540` and `height` will be `370`. This is the amount of space actually available for the bars of our chart.

(Note that our chart div needs a corresponding amount of padding which we can apply via CSS: `padding: 10px 10px 20px 50px;`. This is another thing we could avoid with a pure SVG chart.)

The next step is to create a container that will house both of our axes. This container is positioned according to our `left` and `top` margin values, ensuring everything "starts" from the correct position.

```
const axisContainer = svg.append('g')
  .attr('transform', `translate(${margin.left}, ${margin.top})`)
```

This will give us the following markup as our starting point.

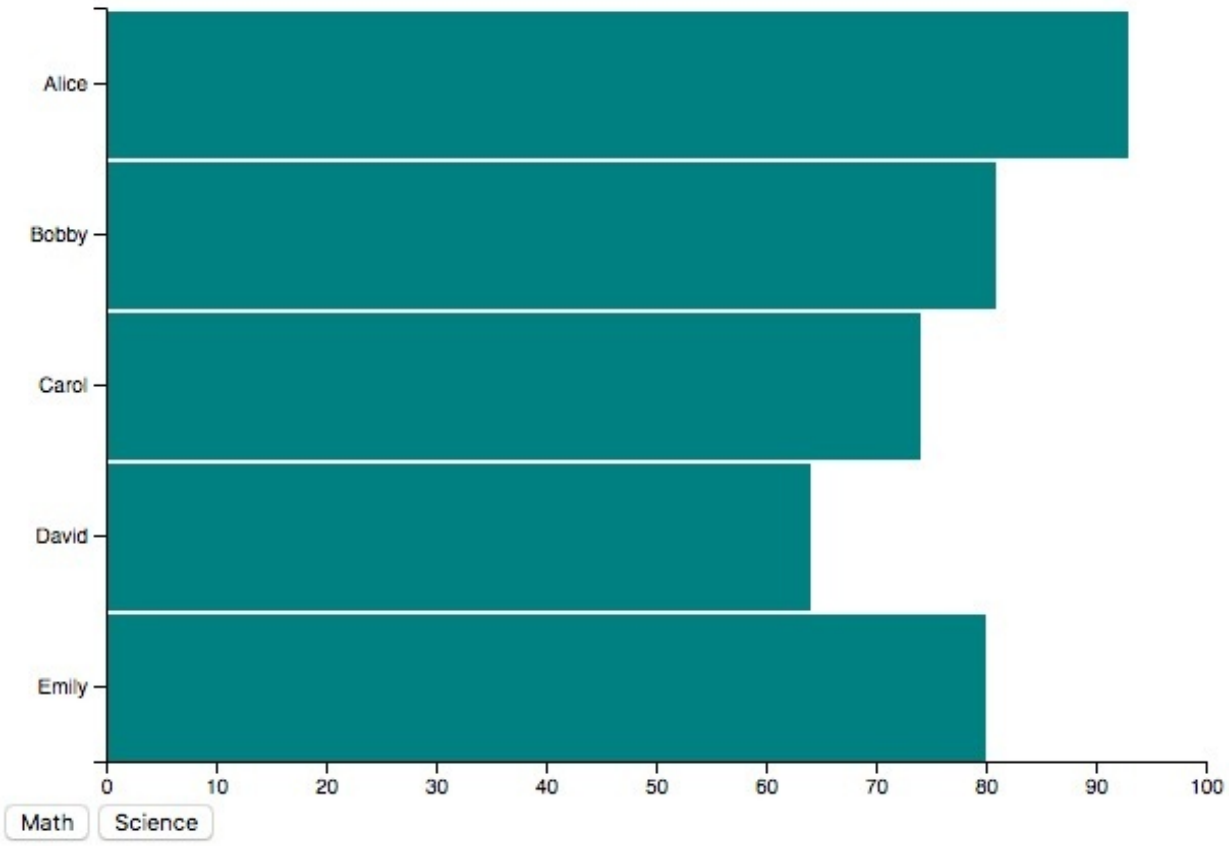
```
<svg width="600" height="400">
  <g transform="translate(50, 10)"></g>
</svg>
```

With the container in place we can update our axis creation code. We'll create the axes within the container and use our newly calculated `height` to position the X axis right at the bottom of the chart.

```
axisContainer.append('g')
  .attr('transform', `translate(0, ${height})`)
  .call(d3.axisBottom(xScale))

axisContainer.append('g')
  .call(d3.axisLeft(yScale)) // we don't have to move this at all now
```

OK, let's take another look.



Now that's what I'm talking about! Nothing is overlapping, the axes meet in the corner, and it looks like a proper bar chart!

🎉🎉 You've just built a legitimate D3 data visualization! 🎉🎉

Give the final product another spin and tinker with it. Change the data, the margins, dimensions, scales, or anything else and see if you can anticipate the output. Tweak the transition durations and delays, maybe specify a different easing function. You'll notice quickly how robust your code is. You can very easily change any of the presentational aspects of your chart and it remains intact.

This is the power of driving with data.

That's it! But could I ask a tiny favor?

I would LOVE to hear what you thought about this course, *good or bad*. My goal is to create the best introduction to D3 in existence, and I can only do that with lots of feedback. It doesn't have to be eloquent or fully thought out! Was there something in particular that was confusing? Did you have any lightbulb moments? Any and all feedback would be massively appreciated.

Thank you!