D3 on AngularJS

Create Dynamic Visualizations with AngularJS

FULLSTACK.io

Ari Lerner | Victor Powell

D3 on AngularJS

Create Dynamic Visualizations with AngularJS

Ari Lerner and Victor Powell

This book is for sale at http://leanpub.com/d3angularjs

This version was published on 2014-02-14

Tweet This Book!

Please help Ari Lerner and Victor Powell by spreading the word about this book on Twitter!

The suggested hashtag for this book is #d3angular.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#d3angular

Contents

Introduction

About the authors

Ari Lerner is a developer with more than 20 years of experience, and co-founder of Fullstack.io. He co-runs ng-newsletter, speaks at conferences, and recently released Riding Rails with AngularJS. He also teaches in-person classes at Hack Reactor and online with airpair.

Victor Powell is a freelance data visualization developer. Prior to freelance, Victor built data visualization tools with YinzCam, Inc. used by NFL, NHL, and NBA sports teams. In his free time, Victor enjoys finding ways to explaining unintuitive or complex mathematical concepts visually. Victor also guest instructs at Hack Reactor.

About this book

The D3 on AngularJS book is packed with the solutions you need to be a D3[1] and AngularJS[2] expert. AngularJS is an advanced front-end framework released by the team at Google[3]. It enables you to build a rich front-end experience, quickly and easily and D3 is an advanced data visualization framework released by Mike Bostock.

The D3 on AngularJS gives you the cutting-edge tools you need to get you up and running on AngularJS and creating impressive web experiences in no time. The goal of this book is not only to give you a deep understanding of how D3 works, but how to integrate it properly into your own AngularJS apps.

With these tools and tests, you can dive into making your own dynamic web applications and visualizations with AngularJS while being confident in understanding the technology.

Audience

We have written this book for those who have never used AngularJS to build a web application and are curious about how to get started with an awesome JavaScript framework. We assume that you have a working knowledge of HTML and CSS and a familiarity with basic JavaScript (and possibly other JavaScript frameworks). We also assume that you've never written in D3 before, but are interested in learning.

[1]http://d3js.org/
[2]http://angularjs.org
[3]http://google.com

Organization of this book

The first half off the book focuses exclusively on the basics of D3 so we'll start off assuming you have not used the Library before. In the second half of the book, we'll cover Angular and how it can be used to make reusable data visualization components. We'll also assume you have not used Angular but at the same time, we won't get into too many details of the Library and only cover the concepts that specifically aid in creating reusable and interactive data visualizations.

Additional resources

Since this book does not cover AngularJS in-depth, we urge you to check out the Complete Book on AngularJS at ng-book.com[4].

We suggest that you take a look at the AngularJS API documentation, as it gives you direct access to the recommended methods of writing Angular applications. It also gives you the most up-to-date documentation available.

Conventions used in this book

Throughout this book, you will see the following typographical conventions that indicate different types of information:

In-line code references will look like: `<h1>Hello</h1>`.

A block of code looks like so:

```
var App = angular.module('App', []);

function FirstController($scope) {
  $scope.data = "Hello";
}
```

Any command at the command line will look like:

```
$ ls -la
```

Any command in the developer console in Chrome (the browser with which we will primarily be developing) will look like:

[4] https://www.ng-book.com

```
> var obj = {message: "hello"};
```

Important words will be shown in **bold**.

Finally, tips and tricks will be shown as:

> Tip: This is a tip

Development environment

In order to write any applications using AngularJS or D3, we first need to have a comfortable development environment. Throughout this book, we'll be spending most of our time in two places: our text editor and our browser. We recommend you download the Google Chrome browser, as it provides a great environment to develop in with its convenient and built in developer tools suit. It's also the browser we used to create the examples. In theory, there should be no differences in the way the examples run on other standards compliant browsers but we can't ever be absolutely sure.

conclusion.md

Introducing D3. A simple example

In this chapter, we'll go over what D3 is and what makes it such a powerful tool for data visualization. We'll also introduce a simple 'Hello World' style example that shows how to get quickly get setup and running with D3.

What is it?

D3 (or Data-Driven Documents) is a library written by Mike Bostock[5] for "manipulating documents based on data." That is to say, D3's primary job is to take data and produce structured documents such as HTML or SVG. Unlike most visualization libraries, D3 is not a ready-made collection of common graphs and widgets. It's easy to use D3 to make common graphs, like bar charts and pie charts. but its real power is in its flexibility and fine grain control over the final result.

Because D3 is primarily responsible for manipulating the structure of the DOM based on data, it works well with other with, not against, established web technologies like CSS and SVG as we'll see later.

If you're just looking for a particular graph type, say, a bar chart, and don't care how exactly it ends up looking, D3 might not be the right library for the job. Several other ready made libraries exist for creating simple bar charts, such as HighCharts[6] or Chart.js[7] or Google Charts API[8]. If, on the other hand, you have strong requirements for how your graph should look and function, D3 would be a great pick.

To illustrate this point, we'll walk though a little thought experiment. Imagine we're working with a ready-made visualization library. Typical visualization libraries might have a `BarChart` class to create a new bar chart which works fine until we want to do something the library didn't allow to be configured. For example, say we wanted to change the background color of the legend in our bar chart. Well, we could take their code and try and modify it to add our needed feature, but that can quickly get very messy and cumbersome. We might have to create a new subclass of the `BarChart` or it could be that the original BarChart class wasn't written in a way to be easily extensible. Alternatively, we can use D3 to quickly, and only in a few lines, create our own custom bar chart were we can do whatever we'd like to our legend or any other component, for that matter.

As another example, say we wanted to create a new type of visualization that doesn't even exist yet (or at least not yet in JavaScript.) This is a perfect example of when you would want to use

[5]http://bost.ocks.org/mike/

[6]http://www.highcharts.com/

[7]http://www.chartjs.org/

[8]https://developers.google.com/chart/

D3. In this sense, D3 is a sort of “meta-library”; the kind of library one would want to have if we were creating a library of new data visualizations. It does this by using a new way to think about data visualizations using selectors and joins (but more on that, later.) In short, if we’re going to be creating data visualizations, we’ll typically be writing an order of magnitude less code if we use D3 than without.

When it comes to configuring the look and feel our visualization, we can very easily utilize our existing knowledge of CSS, so long as we use classes when creating the components that make up our bar chart. continuing our though experiment, something along these lines would be enough to change the background color of our legend.

```
.graph .legend{
  color: white;
}
```

D3’s functional, declarative style permits us to write less code. Less code allows changes to make changes faster and reduces the cognitive load required to remember all the code we’ve written. When code is shorter, we can remember it better.

Consider the following example that changes all `<circle>` nodes in an `<svg>` to be positioned horizontally occurring to the `data` array.

Don’t worry if you don’t understand all that’s happening yet as we’ll walk through much of how this works in the course of this book. This sample is only to demonstrate D3’s brevity.

Using straight JavaScript, we’ll need to select the `<svg>` element and select all of the `<circle>` elements (assuming they are on the DOM already) and modify their `cx` attribute, like so:

```
var data = [ 10, 20, 30, 40];
var svg = document
  .getElementsByTagName('svg')
  .item(0);
var circles = svg.getElementsByTagName('circle');
for(var i = 0; i < circles.length; i++){
  var circle = circles.item(i);
  circle.setAttribute('cx', data[i]);
}
```

Using D3 allows us to accomplish same exact method using much less code:

```
d3.select('svg').selectAll('circle')
  .data([10, 20, 30, 40])
  .attr('cx', function(d){ return d; });
```

One advantage of the d3-style is that it will add the `<circle>` element on the DOM for us if it's not there for each data point, unlike the JavaScript version above.

'Hello World' D3 style

To dive right in, bellow is a simple 'Hello World' style D3 example which simply appends an `<h1>` with the text `Hello World!` to the `<body>` using D3.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
  </head>
  <body>
    <script>
      d3.select('body')
        .append('h1')
        .text('Hello World!');
    </script>
  </body>
</html>
```

live demo http://jsbin.com/uhEmuJI/1/edit[9]

We just created our first D3 app.

Although this example does not do very much, it highlights the structure we'll build with our D3 apps; the foundation for most of the examples that follow.

The resulting HTML after our D3 code has executed looks like:

[9]http://jsbin.com/uhEmuJI/1/edit

```
<html>
  <head>...</head>
  <body>
    <script>...</script>
    <h1>Hello World!</h1>
  </body>
</html>
```

We can inspect the source of the resulting HTML using the fantastic Chrome developer tools[10] `Elements` tab. We'll use these tools throughout the book to deeply inspect our running code.

Common Gotcha:

Common Gotcha

When not using Angular or jQuery, make sure to put any code that depends on the `<body>` inside the `<body>` and *not* before it (ie., in the `<head>`.) If the code executes before the body element, then it will simply fail as the `<body>` will not have been created yet.

[10] https://developers.google.com/chrome-developer-tools/

Selections And Data Binding

This chapter covers D3's "selections" and how they can be combined with data to expressively create and manipulate HTML documents. We'll use selectors all throughout our D3 code so the content in this chapter is fundamental to using D3.

Those reads familiar with **jQuery** are strongly encouraged *not* to skip this chapter. Even though D3's selectors are similar, they deviate in a few key areas.

Selections

Selectors are objects that represent collections (or sets) of DOM elements. This allows us to modify properties on the selector and have those changes applied to all the DOM elements in it. This is an extremely powerful idea! It's now no longer our job to be explicit in *how* the program should go about changing all the elements. Instead, we only need to specify which elements should change and their associated changes. Say we wanted to change the background of a bunch of `<div>` tags. Instead of having an array of DOM elements and modifying each one directly:

```
// `divs` is just an array of DOM elements
for(var i = 0; i < divs.length; i++){
  divs[i].style.backgroundColor = 'blue';
}
```

Live version: http://jsbin.com/UdiDiQu/2/edit[11]

We can apply modifications to each `<div>` element using the selector.

```
// `divs` is a selector object
divs.style('background-color', 'blue');
```

[11]http://jsbin.com/UdiDiQu/2/edit

Live version: http://jsbin.com/ADeReRo/1/edit[12]

The last two example snippets perform exactly the same task. The latter example also introduced our first selector method, `style`, which changes the background color of all the divs in the selection to blue. We'll talk about this method more below. For now, just understand that selectors give us a vocabulary for talking about groups of DOM elements and that their methods apply to each containing DOM element within the selector.

So how do we create new selectors? Fortunately, selectors in D3 follow the same convention used in CSS. (This should also feel very familiar if you have ever used jQuery.) In the same way a CSS rule applies to a set of DOM elements, we can collect DOM elements to put in our selector using the same rules. Here's an example of a snippet of CSS which applies to all `<div>` tags that have the class 'foo':

```
div.foo{
  /* CSS applied to all <div>'s of class "foo" */
}
```

With selectors, how we specify which elements we want to select is exactly the same. We can still just use `div.foo` by passing it to `d3.selectAll`. The result, is a new selector object.

```
var fooDivs = d3.selectAll('div.foo');
```

The selector object `fooDivs` now contains all the divs on the page that have class `foo`. It's helpful to think about the `d3.selectAll` searching the entire DOM for all the elements that are "divs" *and* of the class the class "foo". As a refresher, here's a few CSS rules and their associated meaning.

CSS rule	**Meaning**
.foo	select every DOM element that has class `foo`
#foo	select every DOM element that has id `foo`
div.foo	select all the `<div>` tags that have class `foo`
div#foo	select all the `<div>` tags that have id `foo`
div .foo	select every DOM element that has class `foo` that's inside a `<div>` tag
div #foo	select every DOM element that has id `foo` that's inside a `<div>` tag
div p .foo	select every DOM element that has class `foo` that's inside a `<p>` tag that's inside a `<div>` tag

[12] http://jsbin.com/ADeReRo/1/edit

Unlike CSS specified in stylesheets, a selector object only contains the elements that matched the selection rule when the selection was first created. In the example above, if new `<div>` tags were added to the DOM with class `foo` they would not automatically be in the `fooDivs` selector.

Selector methods

The methods on selectors apply to all its containing DOM elements, minus a few exceptions like the selector `size()` method which returns the number of elements in the selector.

style

Take a look at the following example that changes the background color of every `<div>` on the page to blue using the `style()` selector method:

```
// change all of the divs to have a background color of blue
d3.selectAll('div')
  .style('background-color', 'blue');
```

If the code in our `<body>` tag was the following before running the code.

```
<body>
  <div>Hello</div>
  <div>World</div>
  <div>!</div>
</body>
```

After the DOM loads and our JavaScript runs, the DOM now looks like this:

```
<body>
  <div style="background-color: blue;">Hello</div>
  <div style="background-color: blue;">World</div>
  <div style="background-color: blue;">!</div>
</body>
```

Live version: http://jsbin.com/iYASaLoX/1/edit[13]

[13]http://jsbin.com/iYASaLoX/1/edit

Quick testing

A quick way to test out small snippets of D3 code is to pull up the Chrome Developer Tools while visiting d3js.org[14]. the global `d3` object will be available there.

As it stands all our divs will become blue, but what if we wanted to apply a different style to each div? The `style()` method (along with most other selector methods) can be passed an accessor function instead of a value. This accessor function will be called for each element in the selector and *its* result will be used to set the 'background-color' (or whatever other style property we specify.)

Working from our previous example, we can randomly change the color of every `<div>` to `red` or `blue` using this technique. We're just passing the `style` method a function instead of a string.

```
// blue and red the divs!
d3.selectAll('div')
  .style('background-color', function(d, i) {
    if(Math.random() > 0.5) {
      return 'red';
    }else{
      return 'blue';
    }
  });
```

Live version: http://jsbin.com/agibaYo/1/edit[15]

The accessor we pass to style also gets called with two arguments, here named `d` and `i`. `i` is the index of the current element in the selector. If we wanted to instead color only the first div `blue` and the rest `green` we can use the `i` property to check the index and apply the color accordingly.

```
d3.selectAll('div')
  .style('background-color', function(d, i){
    // make only the first div `blue`, the rest `green`
    if(i === 0){
      return 'blue';
    }else{
      return 'green';
    }
  });
```

[14]http://d3js.org
[15]http://jsbin.com/agibaYo/1/edit

Live version: http://jsbin.com/uTOQAtIT/2/edit[16]

The first `d` argument stands for `datum` we'll come back to this when we talk about data binding, but for the time being, let's just ignore it.

You can make these two arguments whatever you like but `d` and `i` are the common convention in the d3 community.

attr

Another selector method that operates almost identically to `style()` is `attr()`. It follows the same calling conventions but is used to modify attributes of the selected DOM elements (instead of only working on the style attribute like the `style()` method).

Given a `<body>` with the following content:

```
<body>
  <div>Hello</div>
  <div>World</div>
  <div>!</div>
</body>
```

We can set the width of all of the `<div>` elements to `100%` by using the `attr()` method, like so:

```
d3.selectAll('div')
  .attr('width', '100%');
```

Applying this to the previous HTML would result in the following DOM:

```
<body>
  <div width="100%">Hello</div>
  <div width="100%">World</div>
  <div width="100%">!</div>
</body>
```

[16]http://jsbin.com/uTOQAtIT/2/edit

Live version: http://jsbin.com/uNiTovuJ/1/edit?html,output[17]

For convenience, selector methods (like `style()` and `attr()`) can be passed an object hash's instead of individual key/value pairs.

This allows us to shrink the following code into a single call:

```
var divs = d3.selectAll('div');
divs.attr('width', '100%');
divs.attr('height', '100%');
divs.attr('background-color', function(d, i) {
    i % 2 === 0 ? 'red' : 'blue'
});
```

The above code becomes just this:

```
d3.selectAll('div').style({
  'width': '100%',
  'height': '100%',
  'background-color': function(d, i){
    return i % 2 === 0 ? 'red' : 'blue';
  }
});
```

Applying this to the previous DOM from above would result in the following:

```
<body>
  <div style="width: 100%; height: 100%; background-color: red;">Hello</div>
  <div style="width: 100%; height: 100%; background-color: blue;">World</div>
  <div style="width: 100%; height: 100%; background-color: red;">!</div>
</body>
```

Live version: http://jsbin.com/uNiTovuJ/2/edit[18]

Most selector methods return a reference to the original selector. That is, in JavaScript they return the `self` object. This allows us to easily chain selector method calls:

[17]http://jsbin.com/uNiTovuJ/1/edit?html,output
[18]http://jsbin.com/uNiTovuJ/2/edit

```
d3.selectAll('divs')
  .style(...).attr(...);
```

Selectors are *not* simply arrays

Although it is convenient to think of selectors as arrays, they are not and we cannot simply use the `[]` API to access individual elements in the selector.

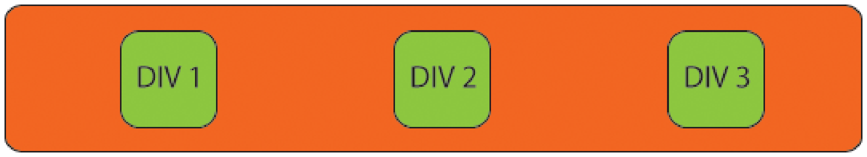
Data binding

Data binding and joining with selections is probably one of the hardest concepts to wrap your brain around in D3 so don't get discouraged if you have to read through this section a few times to make sure you really get how they work. It will also provide the foundation for almost everything that comes later.

Data binding might sound like a complicated term, but it really only means updating what gets shown based on some piece of information (or data.) As a simple example, image I was building a football scoreboard application. The act of updating a teams score on the score board when a player makes a point would be an example of data binding. Some part of the user interface was updated based on some piece of data.

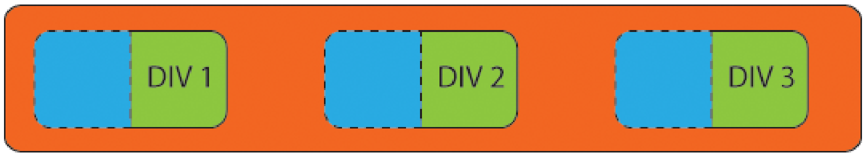
When talking about data binding within HTML, we usually mean changing the contents of some visible property of a DOM element based on some variable. This could mean updating the text within a DOM element or changing one of its style properties, like its opacity. Since this task is so common in data visualization, D3 uses a new way of performing data binding. Unlike with jQuery's selections, D3 lets us easily associate data with DOM elements which makes modifying those elements a lot easier and, more importantly, take less code to write. The trade off is that we have to *really* understand how selections operate under the hood to get their benefit.

Up until now, what we've described of D3 selections has been a bit of a white lie. Selections are slightly more than just collections of DOM elements. They are actually collections of data and DOM element pairs. Here's what a selector looks like in jQuery. It's just a collection of DOM elements.



jQuery style selector

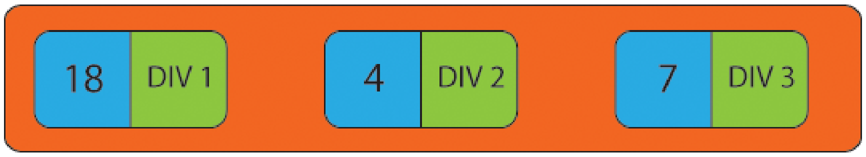
Here's what a D3 selector looks like.



D3 style selector

The dashed blue lines represent empty data elements for each of the three `<div>` tags.

If we were to then “bind” a new array of data to those data/element pairs, say `[18, 4, 7]`, we would now have a selector that looks like the following.



And here’s the d3 code that would produce this selector

```
var selector = d3.selectAll('div').data([18, 4, 7]);
```

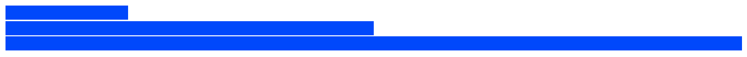
`18` is now associated with the first `<div>` tag, `4` with the second, and `7` with the third.

As you’ve just seen, the selector `data()` method takes an array of data and binds each data item in that array to each DOM element in selector. To demonstrate this, lets build a simple bar chart. We’ll start of with this HTML.

```
<body>
  <style>
    .bar{
      height: 10px;
      background-color: blue;
      border-bottom: 1px solid white;
    }
  </style>
  <div class="bar"></div>
  <div class="bar"></div>
  <div class="bar"></div>
</body>
```

And the JavaScript:

```
var data = [10, 30, 60];
d3.selectAll('.bar').data(data)
  .style('width', function(d){
    return d + '%';
  });
```



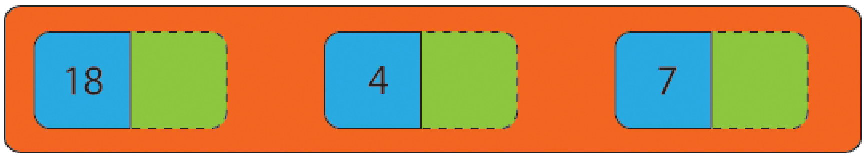
Live version: http://jsbin.com/AlIruJuW/1/edit[19]

Remember that `style` and `attr` will apply style or attribute changes to each DOM element in the selector. If they're passed a function, that function will be run for each element. Now that we know selectors are *really* collections of data/element pairs, it's more appropriate to say these accessor functions are actually getting called for each data/element pair. The first argument to this function (often labeled `d`) is the data item (or datum) for the current data/element pair. The second, recalling from the previous section, is the current index within the selector.

Awesome, but what if we don't know ahead of time how many bar's we're going to have in our bar chart? This is were the `enter()` method comes in.

`.enter()` method

So you may have noticed in the previous example, `data()` seemed to be returning the original selector. Although it is updating the data/element pairs in the original, it is also returning an entirely new selector. The old selector and the new selector both happen to have the same number of data/element pairs but they wont always. We could have an empty selector and create a new one by binding a data array of length 3 to it. Our data selector would now look like the following:

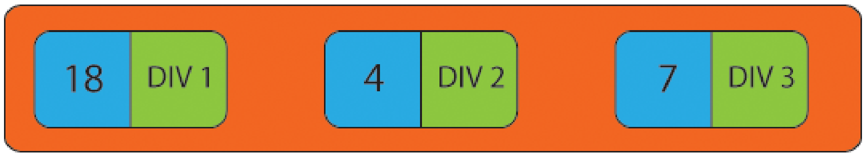


But our original selector would still be empty. Selections returned from calling `data()` come with a method called `enter()` which also returns a new selector. This new selector will contain all the data/element pairs that do not yet contain elements. In our case, it will seem identical to the selector originally returned from `data()` since in the beginning *all* the data/element pairs contain no elements. As we'll see a bit later, this wont always be the case. Our original selector might have

[19]http://jsbin.com/AlIruJuW/1/edit

more or less data/element pairs than the selector returned from `data()`. If there were fewer, calling `enter()` would return an empty selector.

We can now add our missing DOM elements using `append('div')` on the selector returned from `enter()`. Our selector now looks like the following.

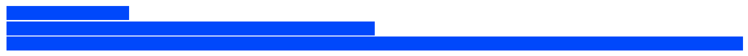


To summarize, we can use `.enter()` and `.append()` to create new DOM elements for lonely data items. Let's update our bar chart example above to incorporate this new feature. Since we'll be creating the DOM elements, we don't need to have them hard coded in the `<body>` tag anymore.

```
<body>
  <style>
    .bar{
      height: 10px;
      background-color: blue;
      border-bottom: 1px solid white;
    }
  </style>
</body>
```

```
var data = [10, 30, 60];
d3.select('body').selectAll('.bar').data(data)
  .enter().append('div')
    .style('width', function(d){ return d + '%'; })
    .attr('class', 'bar');
```

Notice in the code we also had to select the body tag first. Selections can be performed relative to other selections. In this case, we need to perform the selection relative to the body so that D3 knows where any new `<div>` tags will be added. We also need to specify the CSS class so that our CSS rule for each `.bar` can take effect.

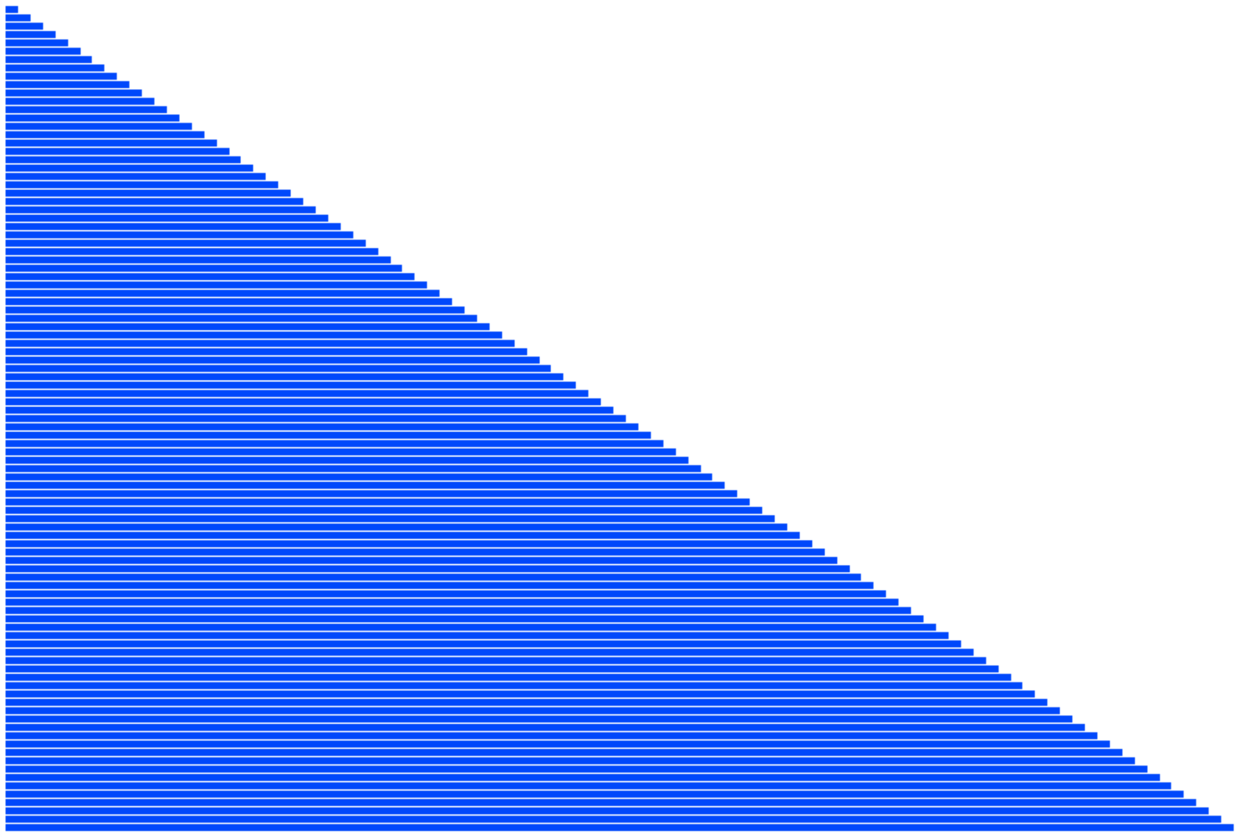


Live version: http://jsbin.com/eNOCuTeL/1/edit[20]

[20]http://jsbin.com/eNOCuTeL/1/edit

The result looks the same but now our code is more flexible. To illustrate this, let's walk through an example of creating a bar chart with an arbitrary number of bars. In this example, we'll use `d3.range(n)` which simply produces an array of the specified length `n`; `d3.range(3)` returns the array `[0, 1, 2]` `d3.range(4)` return the array `[0, 1, 2, 3]`, and so on.

```
var data = d3.range(100);
d3.select('body').selectAll('.bar').data(data)
  .enter().append('div')
    .style('width', function(d){ return d + '%'; })
    .attr('class', 'bar');
```



Live version: http://jsbin.com/eNOCuTeL/3/edit[21]

Remember that `enter()` produces a new selector of all the data/elements pairs that have no element in the selector returned from `data()`. So, if we reran the code above by copying and pasting it bellow itself, the result would appear the same and no new `<div>` tags would be added. This is because the

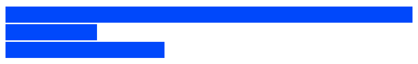
[21] http://jsbin.com/eNOCuTeL/3/edit

`d3.select('body').selectAll('.bar')` would return 100 data/element pairs. Binding them to a new array of length 100 would simple replace all the old data values in the data/element pairs and since none of the data/element pairs would be missing an element, the `enter()` would return an empty selector.

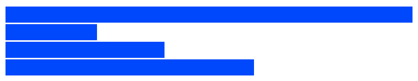
Lets walk through a simply example of rerunning an `.enter()`. Notice how running the second selection with different data updated the first 3 `<div>` tags created in the first selection and then adds one more.

```
d3.select('body').selectAll('.bar').data([18, 4, 7])
  .enter().append('div')
    .attr('class', 'bar')
    .style('width', function(d){ return d + '%'; });

d3.select('body').selectAll('.bar').data([18, 4, 7, 11])
  .enter().append('div')
    .attr('class', 'bar')
    .style('width', function(d){ return d + '%'; });
```

Here's what the document looks like after the first selection but before the second.

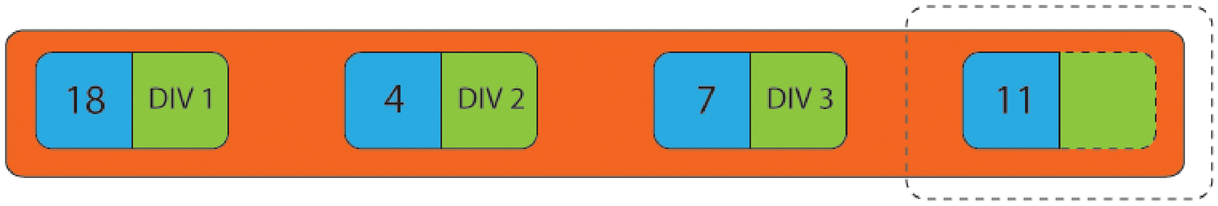


Here's the document after the second selection.



Live version: http://jsbin.com/iHeQoXuP/1/edit[22]

The second `data()` selection looks like the following before the `.enter()` occurs. The dotted border represents what the `.enter()` selection will be.



[22]http://jsbin.com/iHeQoXuP/1/edit

Root DOM elements

If we do not include the first line of `d3.select('body')`, D3 won't know where to place our elements, so it picks the topmost DOM element (or the root element), which is the `<html>` element.

We need to specify where we want to place our D3 chart. This does **not** need to be the `<body>` element and can be any element in the DOM.

exit()

`exit()` works similar to `enter()` but unlike `.enter()` returns a new selector of data/element pairs without elements.

Here's a quick example of using `exit()` assuming the DOM currently has three `<div class="bar"></div>` tags.

```
d3.select('body').selectAll('.bar').data([8, 3]).exit().remove();
```

Live version: http://jsbin.com/UpIgiwah/1/edit[23]

In this example we've also introduced the `remove()` selector method which, like `append()`, simply removes each DOM elements in the selector. Here the DOM originally had 3 `<div>` tags with the `.bar` class. Those elements are then paired with an array with only two data items. Because the 3rd element is now no longer paired with a data item `exit()` will create a new selector with just it. `remove()` then removes it.

General update pattern

Let's walk through a slightly more complicated example of using `enter()`, `exit()`, and regular `selectAll()` selections to create a dynamically updating bar chart. Assuming we've taken care of the styles, we'll first need to create a selector for our future bars. This tells D3 where within the DOM to add new elements if we call `append()`.

```
var bars = d3.select('body').selectAll('.bar');
```

Next, we'll create a function called `update()` that will be responsible for periodically updating our bar chart with new data.

[23] http://jsbin.com/UpIgiwah/1/edit

```
function update(){
  // our update code will go here!
}
setInterval(update, 800);
```

Inside of it, we'll use `d3.range(n)` again to create a new array of length `n` as our data. So we can demonstrate adding or removing bars, we'll give `n` a random value between `0 and 100` each time `update()` is called.

```
function update(){
  var n = Math.round(Math.random() * 100);
  var data = d3.range(n);
  bars = bars.data(data);
}
```

Next, we need to add or remove elements depending on if there's a data item for each.

```
// if data.length is now larger, add new divs for those data items
bars.enter().append('div')
  .attr('class', 'bar');
// if data.length is now smaller, remove the divs that would no longer
// have a data item.
bars.exit().remove();
```

The very last step is to update all the still remaining `<div>` tags. This includes the `<div>` tags that didn't get removed and any recently added `<div>` tags.

```
bars
  .style('width', function(d){ return d / (n-1) * 100 + '%' })
  .style('height', 100 / n + '%');
```

Live version: http://jsbin.com/UpIgiwah/2/edit[24]

[24]http://jsbin.com/UpIgiwah/2/edit

SVG basics

The first section of this chapter provides an introduction to SVG, a powerful open web standard for creating vector graphics in the browser. In the second half of this chapter, we'll highlight the D3 helper functions that make working with SVG fun and manageable.

Scalable Vector Graphics

Although D3 can easily help us create and draw `div` elements, our visualizations generally require much more horsepower. We'll use Scalable Vector Graphics (or SVG for short) to give us a much more expressive interface for drawing in the DOM.

SVG is, like HTML, is an XML-based markup language created specifically for creating vector-based two-dimensional graphics that has support for interactivity and animation.

Traditional graphics systems, or raster graphics, rely on manipulating pixels which often ends up being what's called a destructive progress. This means once we commit to modifying the pixel information in a particular way, it's very hard and often impossible to alter or undo those changes.

With vector graphics, instead of simply recording all the pixels in the image, just the necessary shapes and colors are recorded and recounted just before the graphic is shown on the screen. This last step, of taking the geometry and converting it to pixels on the screen happens transparently and automatically for us.

This process has the benefit of being easily scalable, as well as taking up much less space to store. It allows for us to create SVG images from any standard text editor, we can search, script them, and scale them indefinitely.

Vector graphics do have some disadvantages compared to raster graphics. For example, it's much more convenient to store photos are raster graphics since most photos are not composed of simple geometric shapes. On the other hand, most data visualizations are composed of simple geometric shapes, so vector graphics makes them a perfect fit for use with D3.

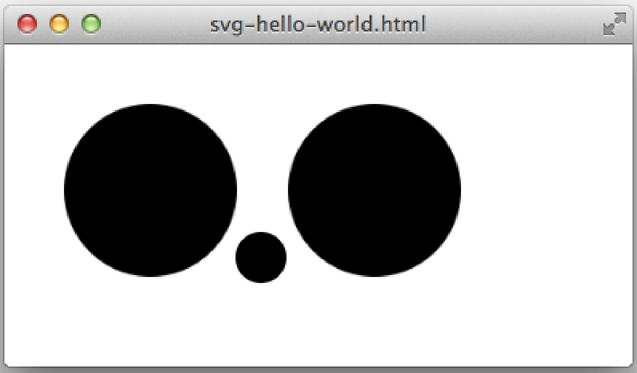
Almost every D3 data visualization we'll come across was likely created using SVG. The fact that we're storing geometric information and not pixel data directly also means it's very easy for us to go about making changes to the SVG graphics we create later.

Possibly the most convenient part about SVG is that we can use it directly inside of our HTML. It will *feel* like we are working with any other DOM element but the result will be a graphic.

Getting started

To get us started, lets take a look at the following 'Hello World' style SVG example:

```
<!DOCTYPE HTML>
<html>
  <body>
    <svg>
      <circle cx="223" cy="83" r="54"/>
      <circle cx="83" cy="83" r="54"/>
      <circle cx="152" cy="125" r="16"/>
    </svg>
  </body>
</html>
```



Panda eyes

Play with the HTML above at http://jsbin.com/ahOZAtEj/1/edit[25].

In order to actually place any SVG elements in our DOM, we'll need to put them inside a parent `<svg>` element.

A new SVG element is easy to make, like so:

```
<svg height="200" width="200"></svg>
```

The `<svg>` element above is simply a DOM element, just like a `<div>` element. Just like any other HTML element, it will take as much room as it possibly can take unless we specify the dimensions.

[25]http://jsbin.com/ahOZAtEj/1/edit

This element won't look like anything on the page quite yet because we haven't placed anything inside of it.

SVG edges

Note that when we are using an SVG element positioned beyond the border of its parent `<svg>` element, it will be clipped. To be sure that our elements don't get cut off, we'll either have to ensure the `<svg>` element is large enough or by making sure our inner SVG elements never exceed the border of their root `<svg>` element.

There are a number of SVG elements that we can use to draw visual elements inside of our `<svg>` element on the DOM as well as we have the opportunity to build our own. These include simple shapes, such as circles, ellipses, lines, rectangles, and rounded rectangles.

We can draw text with many different fonts or paths that can be straight, curves filled in, outlined and more. We have the full color spectrum available for us to use to create gradients, patterns, visual effects on images, compositing, and alpha effects.

SVG also allows us to define animation for different elements.

Furthermore, many popular illustration programs export graphics to SVG, so it's possible to draw paths using an external tool that we can draw the graphics from on the web.

Differences between SVG and HTML

Even though SVG feels like HTML, it differs in a few, not-so-obvious ways. Simply because a property or style works in HTML doesn't mean it will work for SVG elements.

For example, `width` and `height` properties make no sense for a circle. The only three things that describe the geometry of a circle are precisely the three we mentioned, its x and y position from its center (`cx` and `cy`), and its radius (`r`). Notice also that these are properties and not styles. We ***cannot* effect position of SVG elements by using CSS styles like we can with HTML elements**.

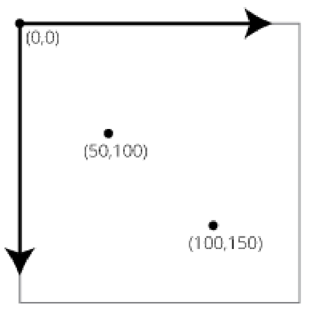
Common Gotcha:

Just because a property or style works in HTML doesn't mean it will work for SVG. Common pitfalls involve using attributes or styles such as `width` `height`, `position`, `left`, `top`, `right`, `bottom`, or `float` on SVG elements witch don't support them.

We will look at a few SVG types now so as to highlight a few of the most commonly used SVG types.

SVG coordinates

Computer-based graphics have traditionally set their coordinate system starting from the top left corner as defined as the base `0, 0` coordinate system. Larger y values move the coordinate downward, while increasing x values move it right. The SVG coordinate system is no different.



Coordinate system

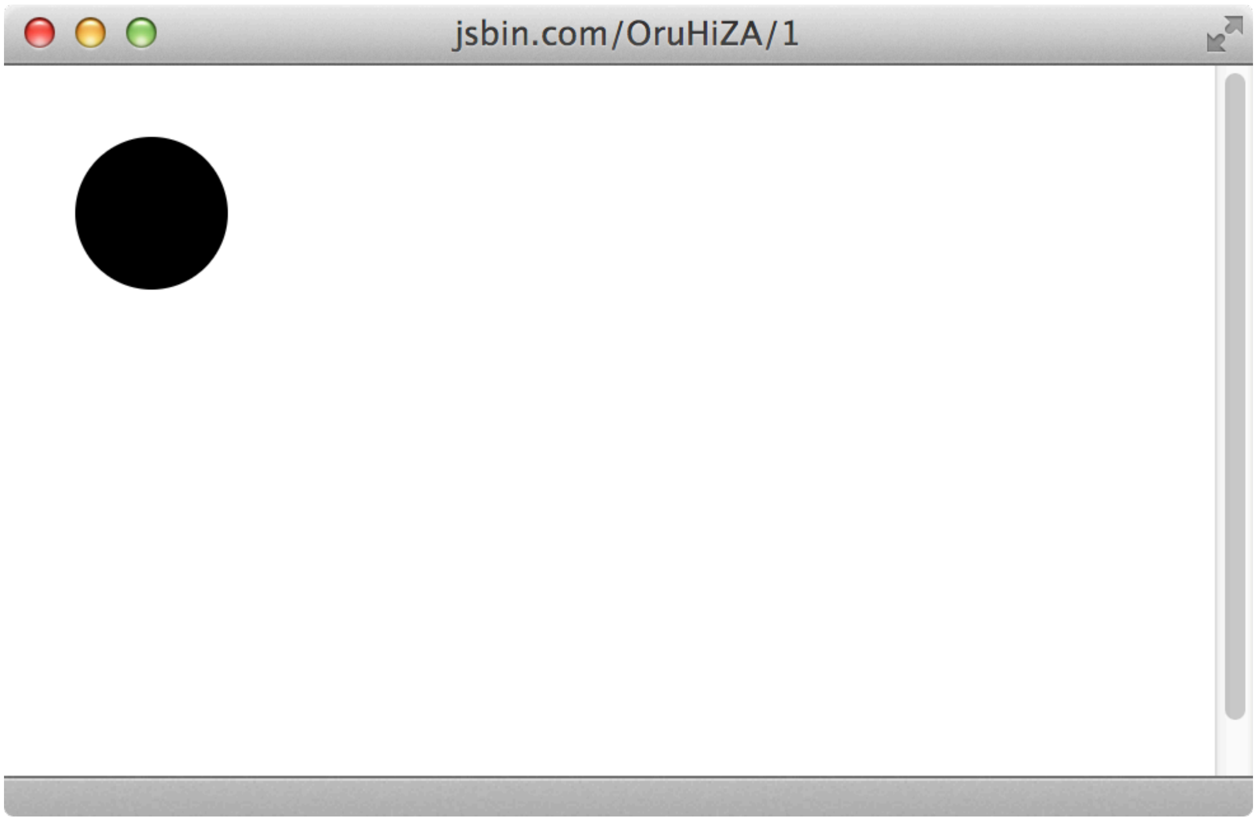
When we place an element inside of our root `<svg>` element, we'll need to respect the the location based on these coordinates.

SVG circle

The first SVG element we just introduced is the `<circle>` which, unsurprisingly creates a circle. Notice how we set the properties `cx` and `cy` corresponding to the offset of the circle from the center along the `x` and `y` axis and the `r`, the radius of the circle.

These are not HTML properties we're using. They are properties specific to circles and not other SVG shapes (like rectangles) or other HTML elements.

```
<circle cx="50" cy="50" r="30">
</circle>
```



Circle

This sets the center of the circle to be placed at the coordinate `(50, 50)` with a radius of `30`.

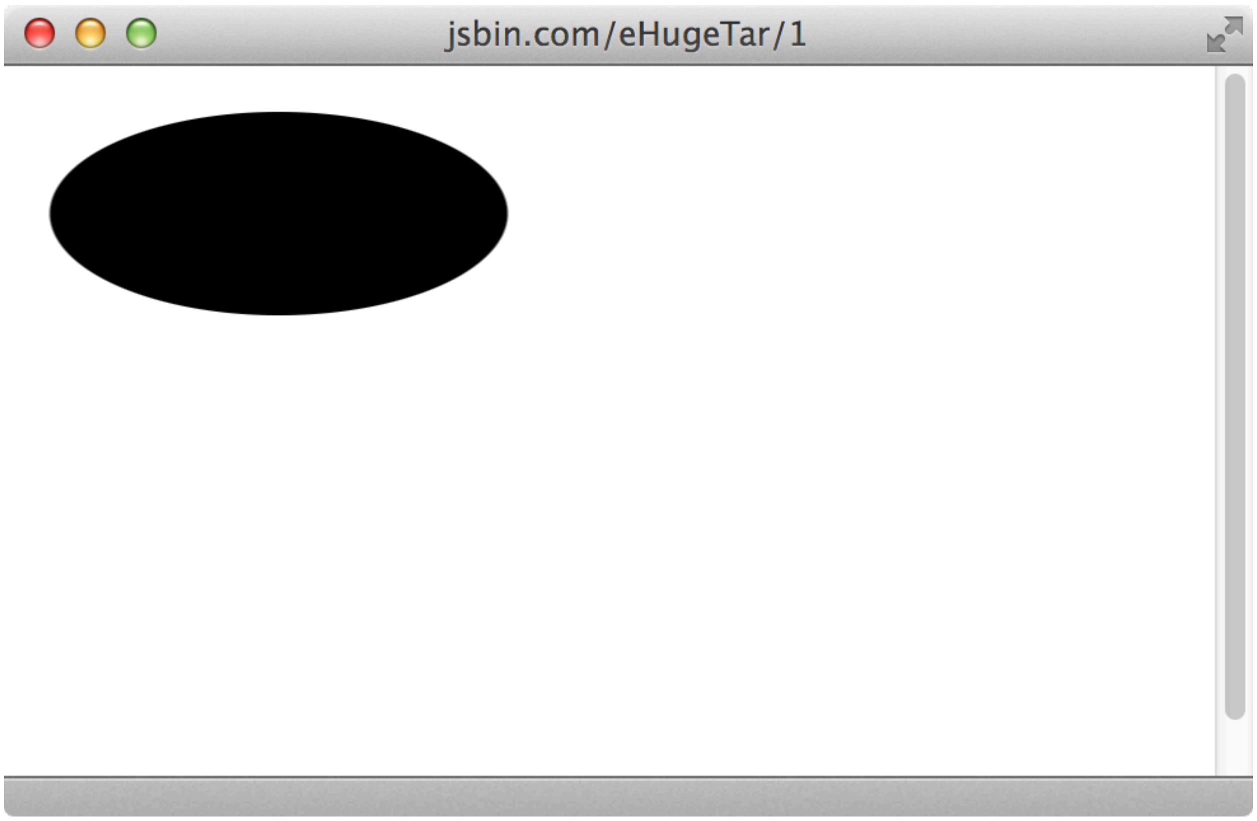
See the live version at http://jsbin.com/OruHiZA/1/edit[26].

SVG ellipse

Like the circle, we can also draw an ellipse that has a different expectation of a different radius for each dimension than the circle.

[26]http://jsbin.com/OruHiZA/1/edit

```
<ellipse cx="50" cy="50" rx="40" ry="20">
</ellipse>
```



Ellipse

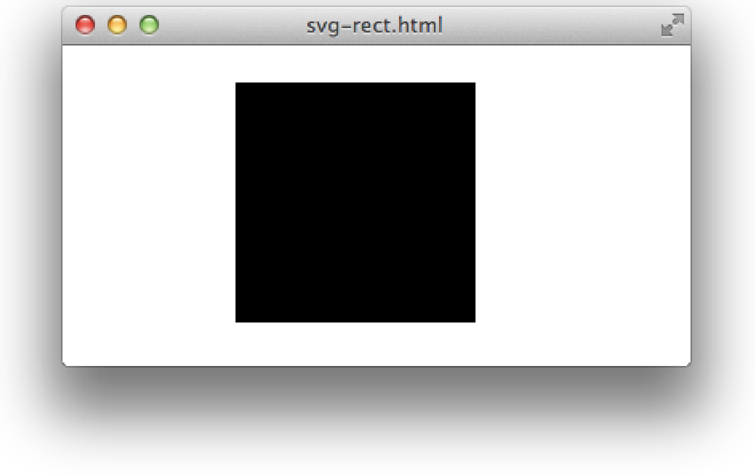
See the live version at http://jsbin.com/eHugeTar/1/edit[27].

SVG rect

The `<rect>` is a straightforward SVG element that simply creates a rectangle. To specific the geometry, we'll need to set the properties `x`, `y`, `width`, and `height`. We'll set the `x` and `y` values to specify the coordinates of the top left corner, while we use the width and the height to specify the rectangle's dimensions, relative to `x` and `y`.

[27]http://jsbin.com/eHugeTar/1/edit

```
<!DOCTYPE HTML>
<html>
  <body>
    <svg>
      <rect x="100" y="15" width="150" height="150"></rect>
    </svg>
  </body>
</html>
```



Black Square on a White SVG, 2013

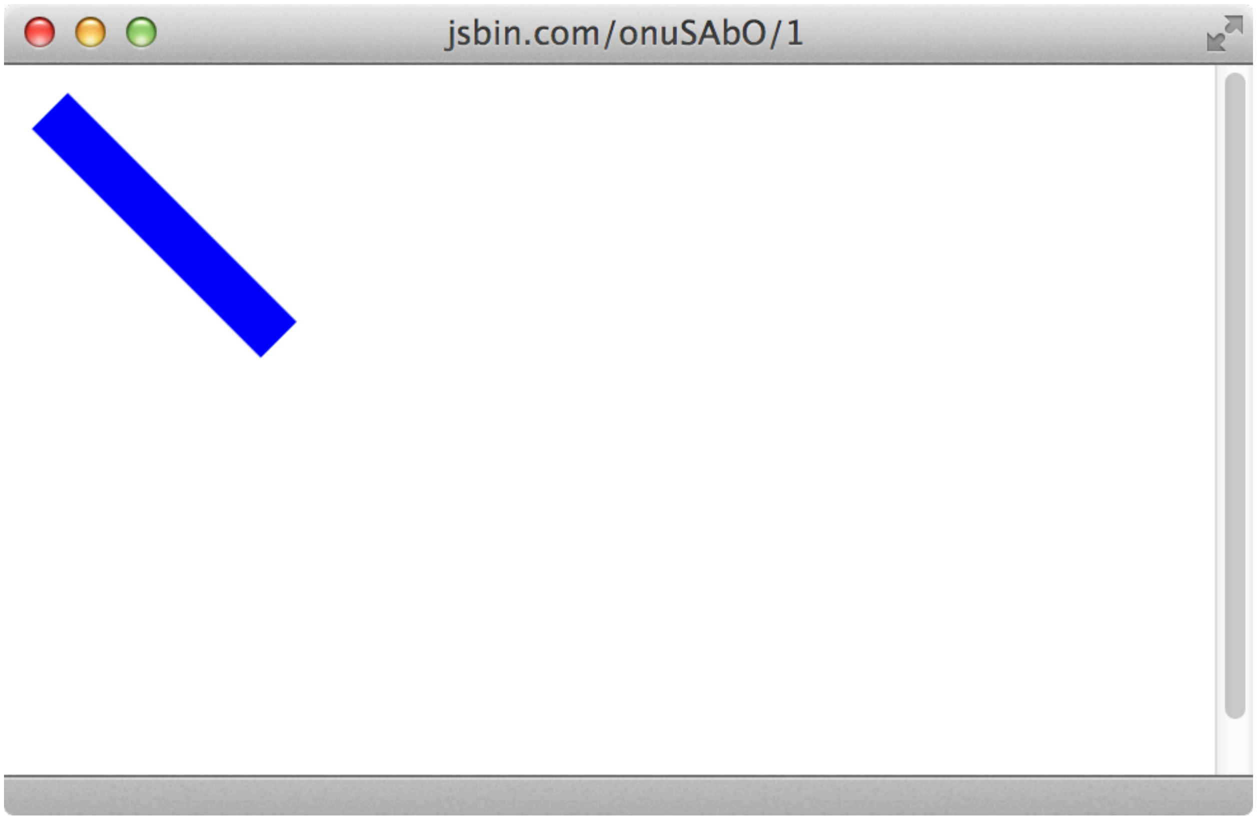
See the live version at http://jsbin.com/etISEYuY/1/edit[28].

SVG lines

SVG also supports drawing simple lines. Lines require four attributes, the starting (`x1` and `y1`) and ending (`x2` and `y2`) coordinates of the line segment that would pass through both points.

```
<line x1="10" y1="10"
      x2="100" y2="100"
      stroke="blue"
      stroke-width="100"></line>
```

[28]http://jsbin.com/etISEYuY/1/edit



Line

See the live version at http://jsbin.com/onuSAbO/1/edit[29].

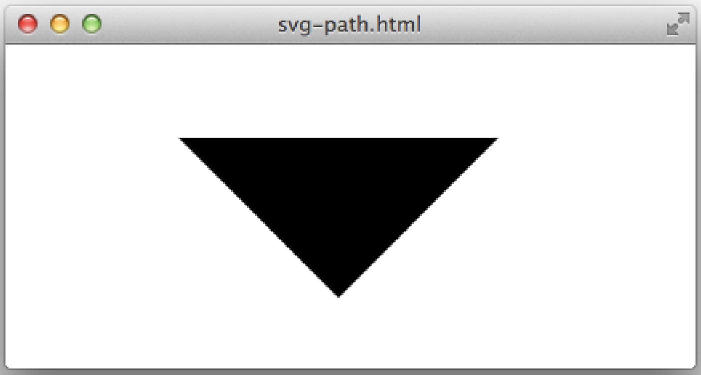
SVG path

The `<path>` is a deceptively cumbersome tag to work with because we have to specific the entire geometry of the path in only a single attribute named simply `d`.

But don't fret! As we'll learn later in this chapter, D3 comes with a wide collection of helper functions that make generating this `d` attribute from our data effortless. It's helpful to know how to create path's so we'll go over the different commands that constitute the path geometry in the `d` attribute. Here's a simple example we'll dissect:

[29]http://jsbin.com/onuSAbO/1/edit

```
<!DOCTYPE HTML>
<html>
  <body>
    <svg>
      <path d="M 100 50 L 300 50 L 200 150 z"/>
    </svg>
  </body>
</html>
```



insane clown triangle

The commands that are passed to the `d` are described using turtle graphics[30] style commands. Think of these commands as how you would tell someone to draw a picture blindfolded, issuing commands such as `lift pen off paper`, `put pen on paper`, `move down`, `move up and to the left`, etc. the commands we pass to `d` are expressed in the same imperative way. The path `d` attribute versions are:

`M` for `move to`

`L` for `line to` ### `z` which stands for `close path`

Lets break down the commands in the previous example in more detail:

[30] http://en.wikipedia.org/wiki/Turtle_graphics

```
M 100 50    - move to position (x=100, y=50),
              without drawing a line
L 300 50    - draw a line from the previous
              position (x=100, y=50) to
              (x=300, y=50)
L 200 150   - draw a line from (x=300, y=50)
              to (x=200, y=150)
z           - connect the start with the end by
              drawing a line from (x=200, y=150)
              to (x=100, y=50)
```

The `<path>`'s `d` attribute can take a few other commands but we wont get into those here since the ones we just examined are the most frequently used.

With that said, we can find further information and learn about the others commands by reading MSDN's extensive documentation on the SVG `<path>`[31]. These include commands for creating smooth lines using Bezier curves.

SVG text

The `<text>` tag is a straightforward way of including text into our SVG but we might question it's existence.

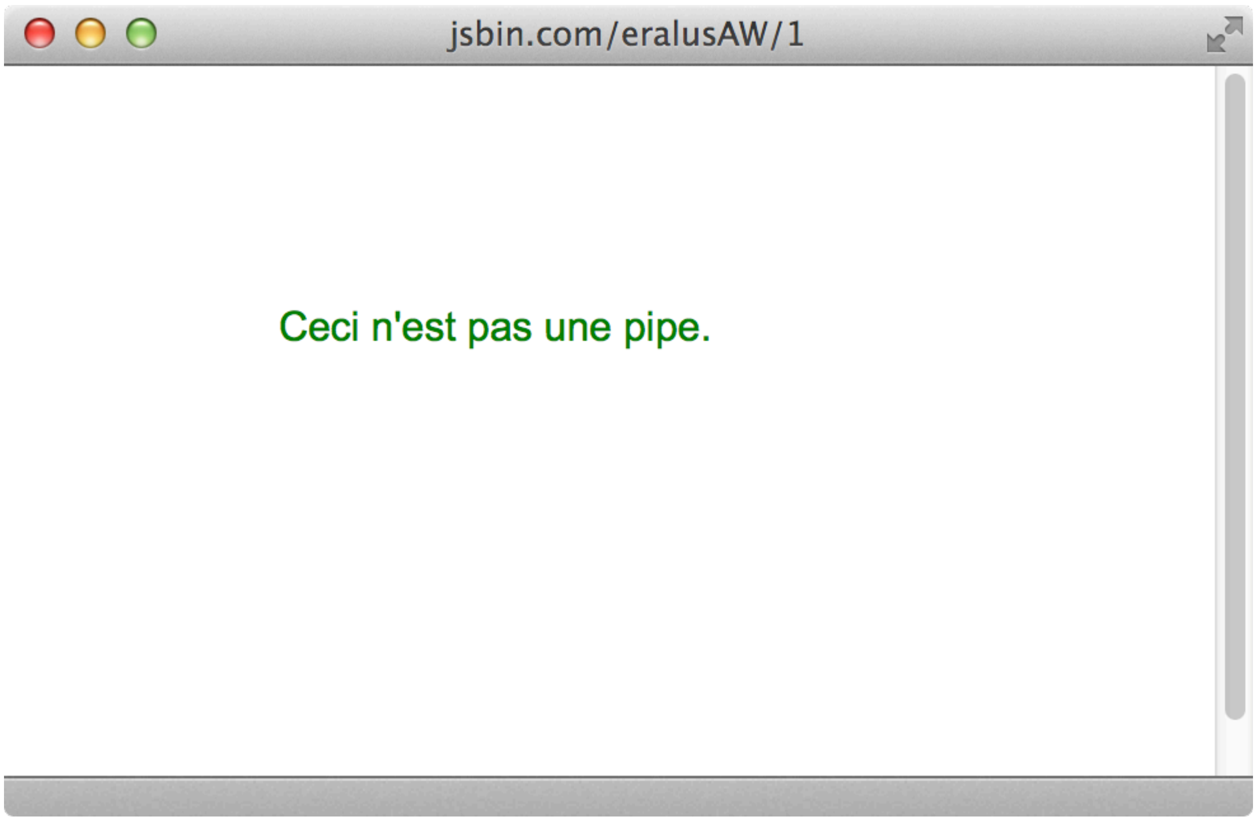
Why couldn't we just use a simple `<p>`, `<div>` or `<span>` tag? Why do we need a separate tag to include text in our drawings?

The main reason is that we cannot (easily) include HTML into an SVG the way we can include SVG into HTML. This is also to allow SVG images to exist on their own, outside of a browser allowing SVG images to be edited in image editors like Adobe Illustrator. We can set the location of a `<text>` tag using its `x` and `y` attributes and set the text by editing its inner text content.

That said, the `<text>` element inherits from it's CSS-specified font styles unless it's directly set on the `<text>` element itself.

[31]https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths

```
<!DOCTYPE HTML>
<html>
  <body>
    <svg>
      <text x="100" y="100" fill="green" font-family="arial" font-size="16">
        Ceci n'est pas une pipe.
      </text>
    </svg>
  </body>
</html>
```



Ceci n'est pas une pipe.

text element in SVG

See this live at http://jsbin.com/eralusAW/1/edit[32].

[32]http://jsbin.com/eralusAW/1/edit

SVG group

The `<g>` tag is an easy way to group SVG elements. Any transformations (positioning, scaling, rotating) that are applied to the group element will also be applied to each of the child elements.

Using groups, we could, for example, create a collection of SVG elements that draw a car using circles for wheels, and rectangles for axles. If we then wanted to rotate the car, we could simply apply the transformation to the `<g>` element containing all the components.

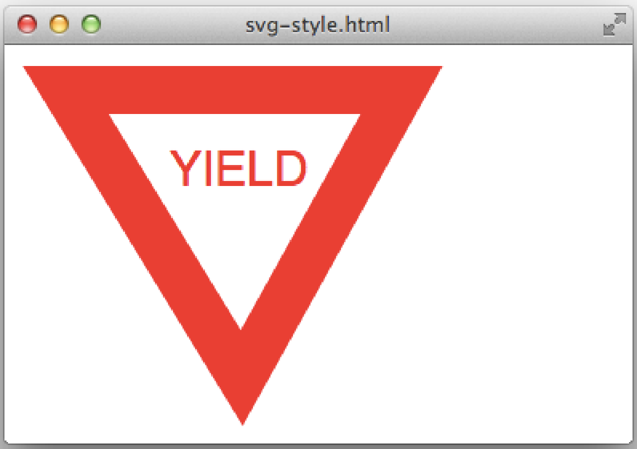
SVG style

Styling SVG elements is almost exactly the same as styling HTML elements. We can still use CSS with `id` and `class` attributes to connect our CSS with our SVG elements or simply apply the style directly using the `style` attribute.

The only difference is that a large number of the styles differ from what we except from experience with HTML.

Take for example the HTML `background-color: orange;` style. The closest equivalent for an SVG element is `fill: orange;`. Similarly, what we expect as of `border-color: blue` equivalent in SVG is `stroke: blue;` or `border-width: 10px;` and `stroke-width: 10;` This is an important detail and a common gotcha when working with SVG coming from an HTML background.

```
<!DOCTYPE HTML>
<html>
  <head>
    <style>
      svg path {
        stroke: red;
        stroke-width: 30px;
        fill: white;
      }
      svg text {
        font-size: 30px;
        font-family: sans-serif;
        fill:red;  /* not `color` or `font-color` */
      }
    </style>
  </head>
  <body>
    <svg>
      <path d="M 30 20 L 140 200 L 240 20 z"></path>
      <text x="95" y="80">YIELD</text>
    </svg>
  </body>
</html>
```



See the live version at http://jsbin.com/iXigIFe/1/edit[33].

It's possible to place these styles directly on the elements themselves. Using CSS to do so is generally much easier and more recommended as it is easier to maintain and read.

Common gotcha:

Keep in mind that for SVG elements, transformations are applied using the `transform` attribute, unless HTML elements, which use a `transform` CSS property.

More on SVG

Transparency

SVG supports drawing with transparency, which we can do one of two ways. We can either set the `opacity` attribute directly on an element or we can set the `rgba` as a color, where the fourth argument is the transparency (also known as the alpha value).

We can set the opacity of an SVG element just like we set any other attribute:

[33] http://jsbin.com/iXigIFe/1/edit

```
d3.select('body').append('svg')
  .attr('width', '300px')
  .attr('height', '200px')
  .data([1,2,3])
  .enter().append('circle')
    .attr('fill', 'blue')
    .attr('opacity', '0.4');
```

See the live version at http://jsbin.com/UzIvodE/2/edit[34].

As we previously mentioned, it's easy to combine the two attributes of `opacity` and `fill` using the `rgba()` function, where we set both the color and the transparency in one go.

```
d3.select('body').append('svg')
  .attr('width', '300px')
  .attr('height', '200px')
  .data([1,2,3])
  .enter().append('circle')
    .attr('fill', 'rgba(100, 200, 300, 0.4');
```

See the live version at http://jsbin.com/UzIvodE/2/edit[35].

Strokes

There are more options at our disposal when it comes to the appearance of shape strokes than just stroke thickness and color. There's also `stroke-linejoin` which effects our the corners of line segments, `stroke-linecap` which effects the appearance of line ends in a similar way `stroke-linejoin` and also `stroke-dasharray` which can be used to create a dashed line effect instead of the defualt solid like.

[34]http://jsbin.com/UzIvodE/2/edit
[35]http://jsbin.com/UzIvodE/2/edit

```
<svg>
  <line x1="20" y1="20"
        x2="200" y2="200"
        stroke="blue"
        stroke-linecap="round"
        stroke-dasharray="4,30"
        stroke-width="20">
  </line>
</svg>
```

See the live version at http://jsbin.com/EBuciXuZ/1/edit[36].

Layering

You may run into situations where you have overlapping SVG elements were one element should be behind another instead of in front. This can happen when the order in which you add your SVG elements is not the same as their visual depth order, bottom to top. With HTML we can fix this using the `z-index` CSS property but that doesn't work for SVG elements. Instead, we'll have to remove and re-add any elements we'd like to be on the every top of their parent element. Luckly, D3 selections come with a nice helper method called `sort` for just this problem. We can pass it our sort function and D3 will resort our selection and update the DOM elements accordingly.

```
d3.selectAll('circles').sort(function(a, b){
  // assuming we've added our own `zIndex` data property
  return a.zIndex - b.zIndex;
});
```

SVG and D3

SVG can be somewhat tedious to use on it's own but after the previous introduction, we'll rarely, if ever, need to write any SVG directly.

Instead, we'll use D3 to create these elements and apply style for us.

We'll still need to know how SVG works to do master D3, but we will let the library take care of the heavy lifting.

Now that we've covered some of the basics of SVG, we can begin to talk more about using it in combination with D3.

[36]http://jsbin.com/EBuciXuZ/1/edit

From nothing to SVG

When we first start up the page, we'll need a place to place our SVG elements. As stated above, we can only place our elements inside of an `<svg>` tag. Sometimes it's more convenient (especially when we integrate with using directives) to not place this directly on the page.

We can use `d3` to make our svg, like so:

```
var svg = d3.select("body")
            .append("svg");
```

With the `svg` variable, we can now reference the `<svg>` element and interact with our D3 stage at any time. It's possible to use the same methods on the `<svg>` element that we'll use on the D3 elements to set attributes on it.

For instance, let's set a height and a width:

```
d3.select('body').append('svg')
  .attr('width', 300)
  .attr('height', 200);
```

Generators

One of the nicest parts of D3, is it's focus on making SVG element creation manageable through D3's generators.

Generators play the part of both objects and functions. Generators are functions, like any other function that has additional methods to allow us to change it's behavior. Just like any other D3 class, most generator methods return themselves so we can chain methods on top of them.

Once we've configured our generators for our needs, we'll use them to *generate* the necessary SVG commands to create our shape.

Let's look at a very simple example, the line generator:

The `line` generator takes any data that we pass it and generators the necessary SVG `<path>` `d` attribute commands. In order to use the path generator, we need to tell it a little about the format of our data. We do this by defining `accessor` functions for the `x` and `y` coordinates of the line across our data. This essentially says "hey line generator, when you create a line, use this for the `x` and this for the `y` of each data point."

If our data was a much of points where a single point was an array of the form `[x, y]` and the array of points looked like this:

```
var data = [ [x1, y1], [x2, y2], [x3, y3], etc... ];
```

Our line generator would look like this:

```
var line = d3.svg.line()
  .x(function(d) { return d[0]; })
  .y(function(d) { return d[1]; });
```

Lets say our data instead was an array of coordinate objects of the form `{ x: x1, y: y1 }` and the array of points looked like this:

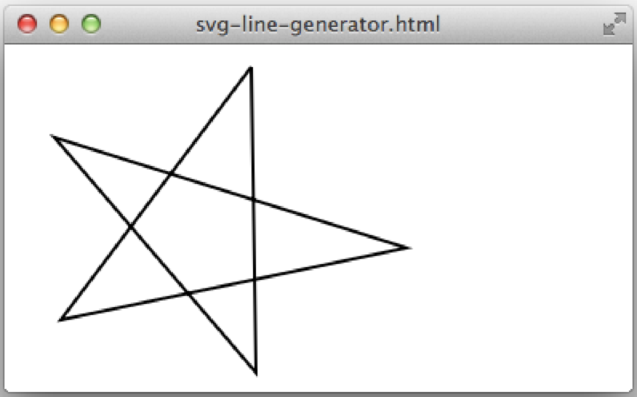
```
var data = [ { x: x1, y: y1 }, { x: x2, y: y2 }, etc... ];
```

Then our line generator `x` and `y` accessors would look like this:

```
var line = d3.svg.line()
  .x(function(d) { return d.x; })
  .y(function(d) { return d.y; })
```

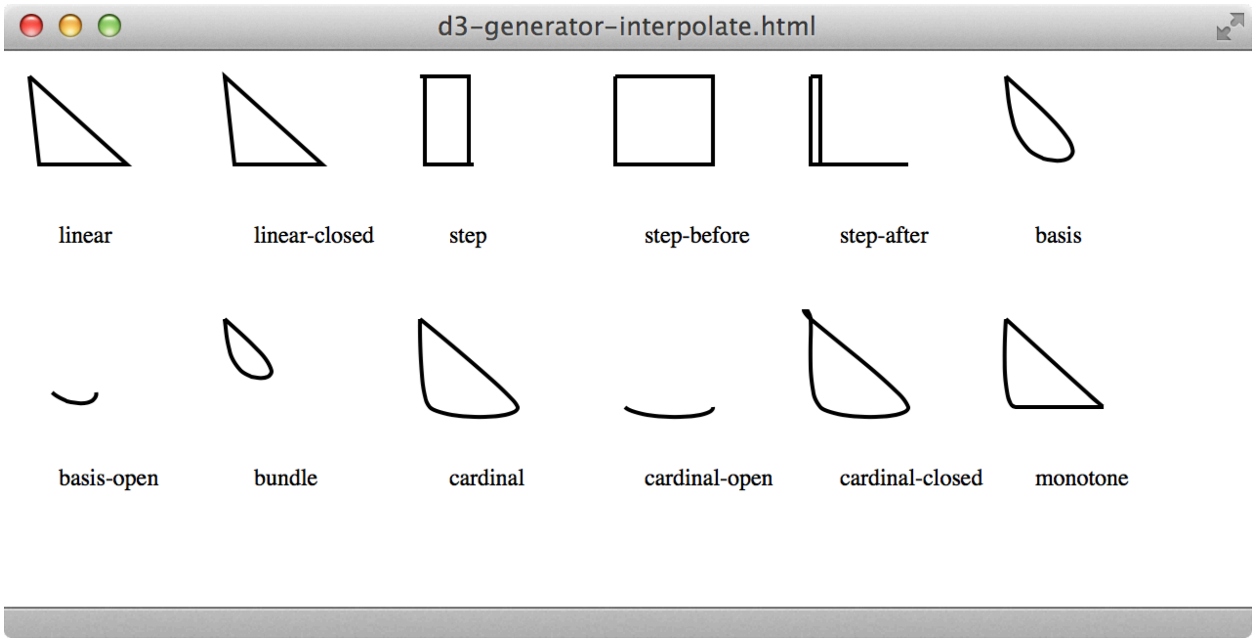
Here's a complete line generator example

```
<!DOCTYPE HTML>
<html>
  <head>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <style>
      body{ margin: 0; }
      path { stroke: black; stroke-width: 2px; fill: none;}
    </style>
  </head>
  <body>
    <script>
var lineGenerator = d3.svg.line()
  .x(function(d) { return d[0]; })
  .y(function(d) { return d[1]; });
// the line data
var lineData = [[154, 14], [35, 172], [251, 127], [31, 58], [157, 205],
  [154, 14]];
// create `<svg>` and inner `<path>` element
d3.select('body').append('svg')
  // use the generator and line data to create the `d` attribute for our path.
  .append('path').datum(lineData).attr('d', lineGenerator);
    </script>
  </body>
</html>
```



With our generator, we can also set an interpolation mode that tells D3 how to render the lines between each point.

There are quite a few interpolation modes available to us.

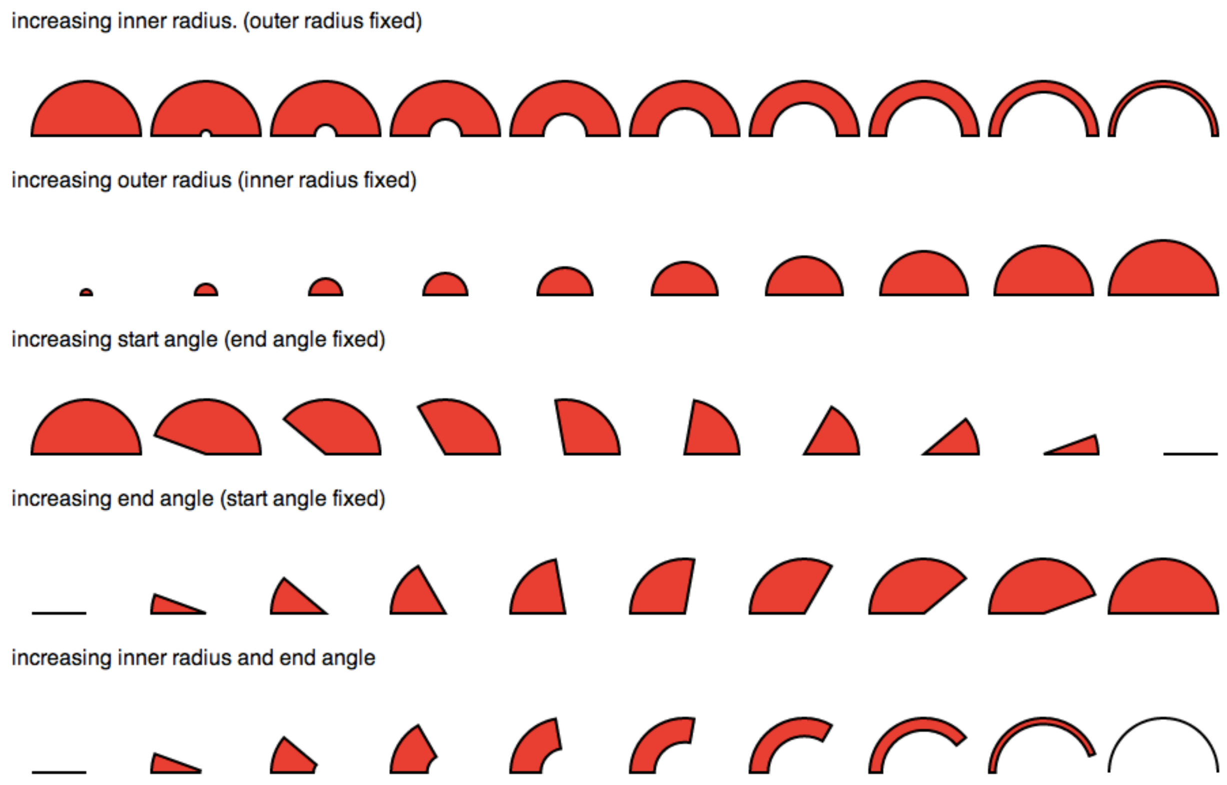


Interpolation modes for d3.svg.line()

And don't forget, generators can be used more than once, for multiple elements so if our data changes but still has the same general format, we can just re-apply the generator.

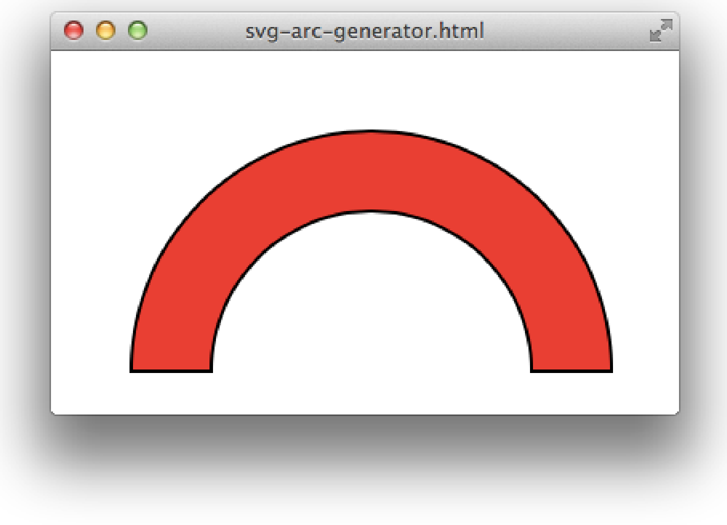
d3.svg.arc

We can also produce an arc or a circle using the `d3.svg.arc()` generator. The four parameters that describe an arc are its inner and outer radii, as well as the start and end angles (where an angle of zero points up and -Ï€, left.)


increasing inner radius. (outer radius fixed)
increasing outer radius (inner radius fixed)
increasing start angle (end angle fixed)
increasing end angle (start angle fixed)
increasing inner radius and end angle

If we wanted our arc generator to always produce the same arc, we could fix each parameter to a constant value by calling all of the arc generator accessors by passing each a single number value. Our arc is also created at (0,0) so we'll also translate it left and down by 200 pixels to we can see the entire arc.

```
var arc = d3.svg.arc()
  .innerRadius(100).outerRadius(150)
  .startAngle(-Math.PI / 2).endAngle(Math.PI / 2)

d3.select('body').append('svg')
    .append('path').attr('d', arc).attr('transform', 'translate(200, 200)')
```



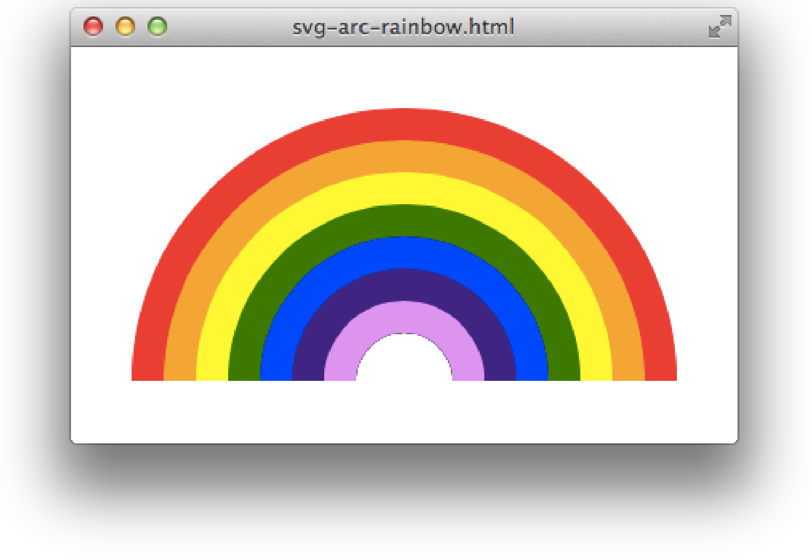
live version[37]

In addition, the accessors can be passed functions, instead of number values, which will be called for each element of the selectors data. For every data item in the selector, D3 will call our accessor and use its result to set that parameter for the current arc being generated. Let's create a rainbow to show this off.

```
<!DOCTYPE HTML>
<html>
  <body>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <script>

var rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];
var arc = d3.svg.arc()
  .innerRadius(30)
  // the outer radius of the rainbow is set from the index in the rainbow array
  .outerRadius(function(d, i){ return (rainbow.length - i) * 20 + 30})
  .startAngle(-Math.PI / 2).endAngle(Math.PI / 2);

d3.select('body').append('svg')
    .selectAll('path').data(rainbow).enter().append('path')
    .attr({
      d: arc,
      transform: 'translate(200, 200)',
      fill: function(d){ return d }
```

[37] http://jsbin.com/uPEfeMaj/3/edit

```
    })
    </script>
  </body>
</html>
```



live version[38]

Here's an example of a donut chart that uses `d3.layout.pie` that handles converting our data to a format we can pass directly to the pie generator. Say we had the following data that we wanted to represent as a pie chart.

```
var data = [21, 32, 35, 64, 83];
```

We could convert that to the necessary array of start and end angles to be passed to your arc generator by calling the `pie()` method on the pie layout.

```
var pie = d3.layout.pie();
var arcData = pie(data);
```

Then, we'll create an arc generator with fixed inner and outer radius values.

```
var arc = d3.svg.arc().innerRadius(100).outerRadius(200);
```

We don't need to specify the `startAngle` and `endAngle` accessors because those, by default, access a property by the same name on their datum element so there's no need to add this snippet.

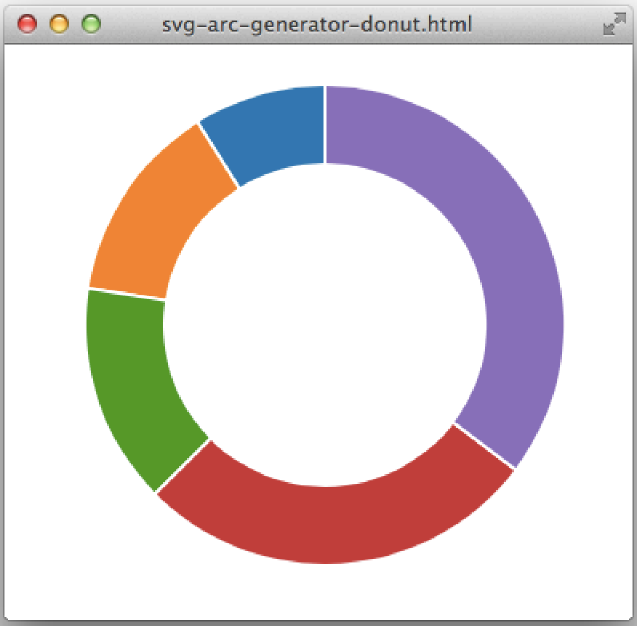
[38]http://jsbin.com/AbUGOZiX/1/edit

```
// default `startAngle` accessor
arc.startAngle(function(d){
  return d.startAngle;
});
// default `endAngle` accessor
arc.endAngle(function(d){
  return d.endAngle;
});
```

Now, the last step is to bind our `arcData` to newly created `path` elements in our selector.

```
var path = svg.append('path').data(arcData)
  .enter().append('path').attr('d', pie);
```

We'll need to translate all the paths a bit so they're not all drawn at (0,0) in the upper left of the screen. Then, after adding a some color using a color scale (which well see more of in a later chapter), the result should look like the following:



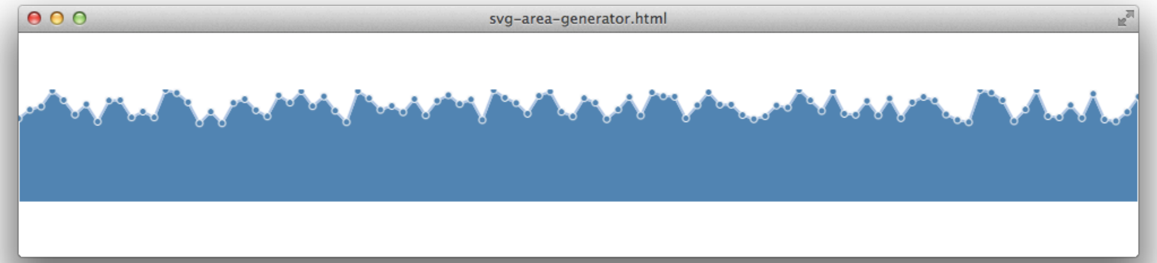
live version[39]

[39] http://jsbin.com/axICUjaH/1/edit

d3.svg.area

The area generator makes it easy to create line plots where the area bellow the plot line is filled in. We need to give the generator's accessors either functions or number values for `y0`, `y1` and `x`. The y* accessors tell the generator how lower or how high the area shape should extend. `x` should be the successive x values along the plot just like with `d3.svg.line()`.

```
<!DOCTYPE HTML>
<html>
  <head>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <style>
      path { stroke: white; stroke-width: 2; fill: steelblue;}
      .plot{ fill: none; stroke: #aec7e8; stroke-width: 2;}
      circle{ fill: steelblue; stroke: white;}
    </style>
  </head>
  <body>
    <script>

var svg = d3.select('body').append('svg')
// construct and configure the are generator
var area = d3.svg.area()
  .y0(100) // set the lower y bound of the area shape
  .y1(function(d){ return d.y })
  .x(function(d, i){ return d.x })

// generator some random data
var data = d3.range(100).map(function(){ return Math.random() * 30 + 0 })
  .map(function(d, i){ return { x: i * 10, y: d }})

// draw the area shape under the line of the plot
svg.append('path').datum(data).attr('d', area)

// give the area shape a border along its top edge
var line = d3.svg.line()
  .x(function(d){ return d.x})
  .y(function(d){ return d.y })
svg.append('path').datum(data).attr('d', line).attr('class', 'plot')

// the circles at all the points
svg.selectAll('circle').data(data).enter().append('circle')
  .attr('cx', function(d){ return d.x })
  .attr('cy', function(d){ return d.y })
  .attr('r', 3)

    </script>
  </body>
</html>
```



d3.svg.chord

The svg chord generator helps in creating connected arc segments which is the essential component in chord diagrams. Every other component in a chord diagram can be created using other svg generators. Two arc segments make up one arc connection. The first arc segment is called the `source`, the second, the `target`. Each arc segment is described by a `radius`, a `startAngle` and an `endAngle`. Conveniently, `d3.layout.chord()` can be used to take relational information in the form of a matrix and produce the necessary default chord layout that `d3.svg.chord` expects. For example, here's an arbitrary relational data set. Every cell describes the relationship among of two items (in our example, the items `A` `B` `C` and `D`). Maybe the relationship could be `likness`. "How much Alex likes Betty." and since Betty might like Alex more than Alex likes Betty, the connections have different magnitudes depending on their direction.

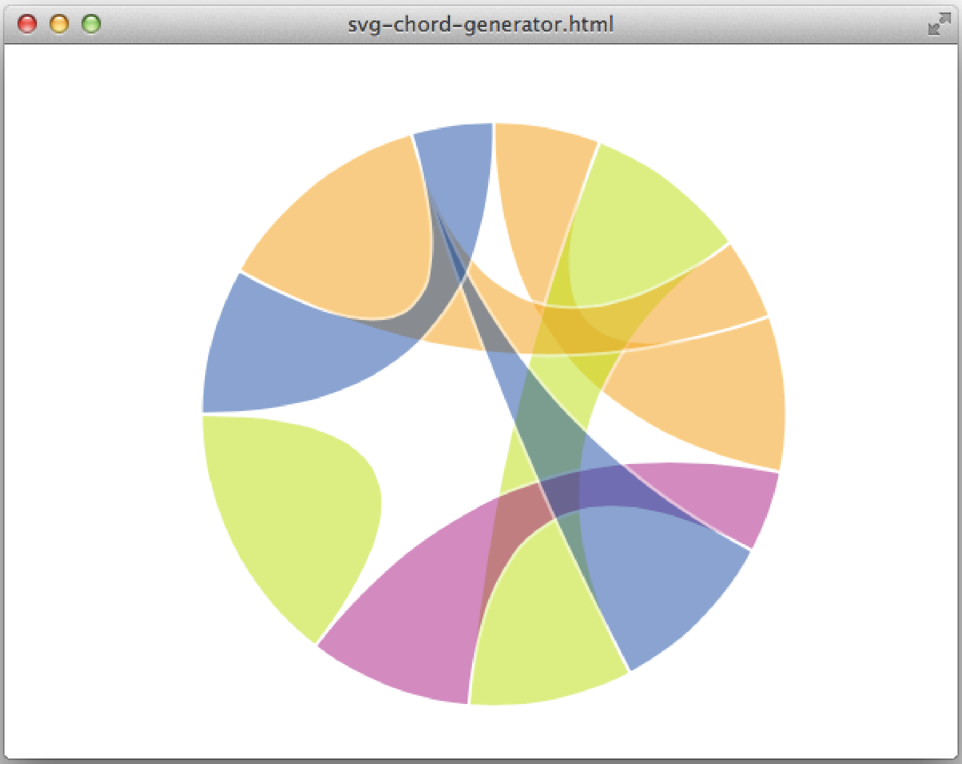
```
var matrix = [
// each cell value represents a single node->-node relationship (column->row)
//       A                  B                 C                  D
/*A*/ [ 0   /* A -> A */, 1.3 /* B -> A */, 2   /* C -> A */, 1   /* D -> A */],
/*B*/ [ 1.9 /* A -> B */, 0   /* B -> B */, 1   /* C -> B */, 2.1 /* D -> B */],
/*C*/ [ 2   /* A -> C */, 2   /* B -> C */, 3.2 /* C -> C */, 1.8 /* D -> C */],
/*D*/ [ 2.7 /* A -> D */, 0   /* B -> D */, 1   /* C -> D */, 0   /* D -> D */]
]; // `C` really likes herself
```

Passing this data to `d3.layout.chord().matrix(matrix).chord()` gives us back data that looks like this:

```
[
 {
  source:{ startAngle:1.2280, endAngle:1.7707, value:1.9 },
  target:{ startAngle:0, endAngle:0.3712, value:1.3 }
 },
 {
  source:{ startAngle:0.3712, endAngle:0.9424, value:2 },
  target:{ startAngle:2.6560, endAngle:3.2272, value: 2 }
 },
 // etc...
]
```

```
<!DOCTYPE HTML>
<html>
  <head>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <style>
      path { stroke: white; stroke-width: 2; fill: steelblue;}
      .plot{ fill: none; stroke: #aec7e8; stroke-width: 2;}
      circle{ fill: steelblue; stroke: white;}
    </style>
  </head>
  <body>
    <script>
var width = window.innerWidth;
var height = window.innerHeight;

var svg = d3.select('body').append('svg')
var matrix = [
// each cell value represents a single row-to-column relationship
  [ 0   /* A -> A */, 1.3 /* A -> B */, 2   /* A -> C */, 1   /* A -> D */],
  [ 1.9 /* B -> A */, 0   /* B -> B */, 1   /* B -> C */, 2.1 /* B -> D */],
  [ 2   /* C -> A */, 2   /* C -> B */, 3.2 /* C -> C */, 1.8 /* C -> D */],
  [ 2.7 /* D -> A */, 0   /* D -> B */, 1   /* D -> C */, 0   /* D -> D */]
];

var chord = d3.layout.chord().matrix(matrix);
var innerRadius = Math.min(width, height) * .41;
var outerRadius = innerRadius * 1.1;


var fill = d3.scale.ordinal()
```

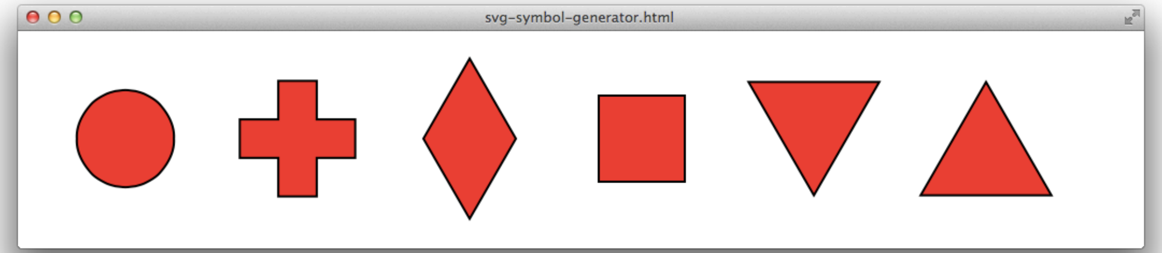
```
    .domain(d3.range(4))
    .range(['#FFA400', '#C50080', '#ABF000', '#1049A9']);

svg = svg.append('g')
    .attr('transform', 'translate(' + width / 2 + ',' + height / 2 + ')');

svg.append('g').selectAll('path').data(chord.chords)
  .enter().append('path')
    .attr('d', d3.svg.chord().radius(innerRadius))
    .style('fill', function(d) { return fill(d.target.index); })
    .style('opacity', 0.5);

    </script>
  </body>
</html>
```



d3.svg.symbol

The symbol generator creates a new symbol generating function for creating symbols. The default symbol is a circle with a `size` (area in pixels) of 64. This is the behavior you're most likely to want but if you'd like to specify the dimensions of say, a square, you'll have to compute the area given your desired dimensions and symbol.

```
var symbol = d3.svg.symbol().type('square').size(100)
var svg = d3.select('body').append('svg')
svg.append('path').attr({d: symbol}).attr('transform', 'translate(100,100)')
```

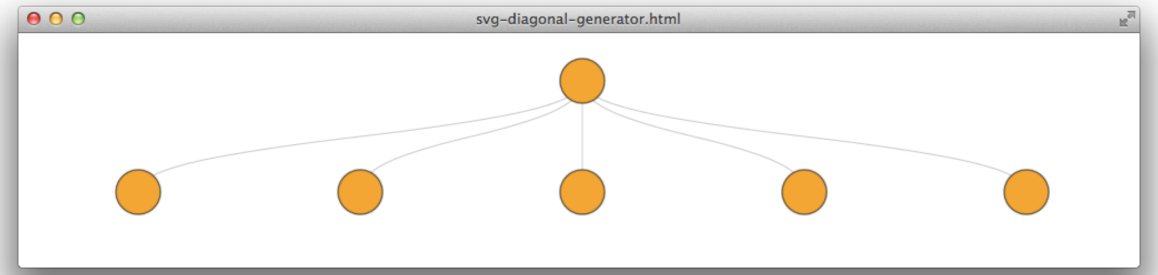


d3.svg.diagonal

The diagonal generator takes node pairs and produces the necessary path that would connect the two elements of the pair using a curved path. The default accessors assume each pair has the following form:

```
{ source: { x: 54, y: 12 }, target: { x: 83, y: 62 } }
```

```
<!DOCTYPE html>
<html>
  <head>
    <style>
    circle { fill: orange; stroke: #333; }
    path { fill: none; stroke: #ccc; }
    </style>
  </head>
  <body>
  <script src="http://d3js.org/d3.v3.min.js"></script>
  <script>

var svg = d3.select('body').append('svg');
var diagonal = d3.svg.diagonal();
var source = { x: 500, y: 50 };
var targets = [
  { x: 100, y: 150 },
  { x: 300, y: 150 },
  { x: 500, y: 150 },
  { x: 700, y: 150 },
  { x: 900, y: 150 }
];
// create the link pairs
var links = targets.map(function(target){
  return { source: source, target: target };
});

// use the diagonal generator to take our links and create the the curved paths
// to connect our nodes
var link = svg.selectAll('path').data(links).enter().append('path')
    .attr('d', diagonal);

// add all the nodes!
var nodes = targets.concat(source)
svg.selectAll('circle').data(nodes).enter().append('circle')
  .attr({
    r: 20,
    cx: function(d){ return d.x; },
    cy: function(d){ return d.y; }
  });

  </script>
  </body>
</html>
```



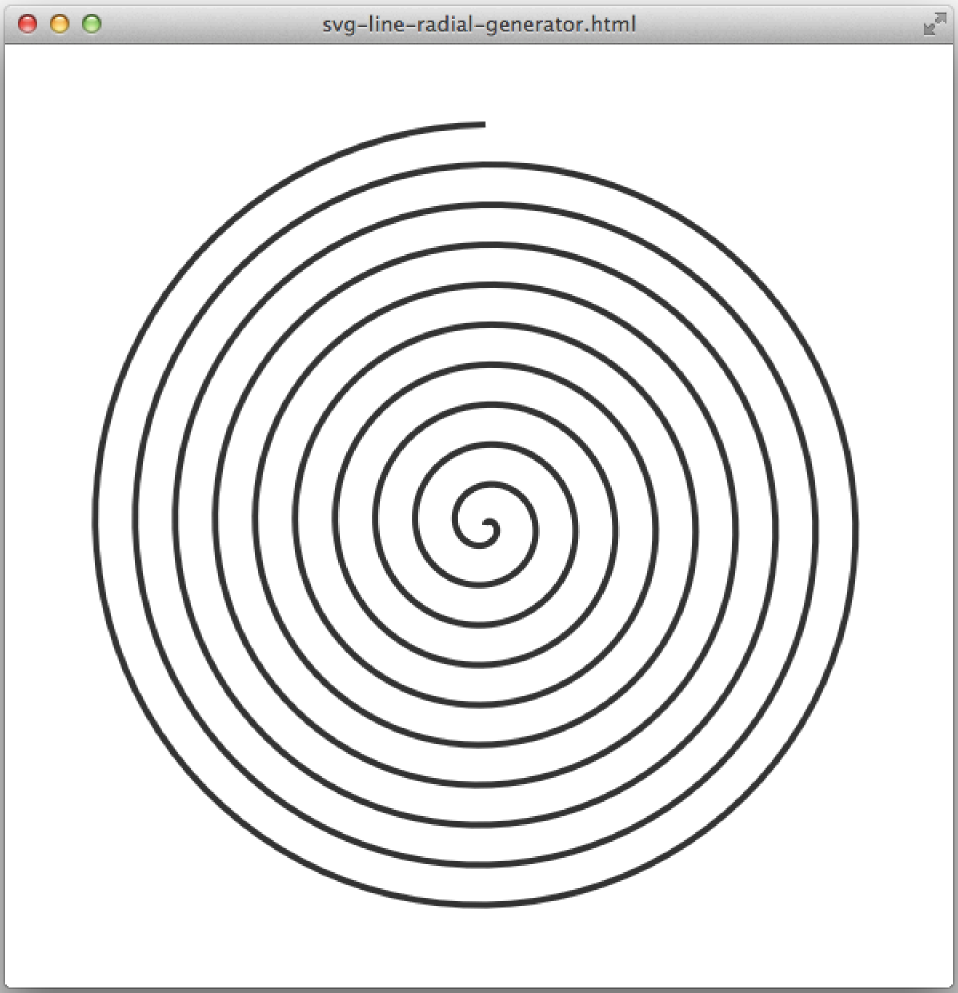
d3.svg.line.radial

The radial generator takes a data array of radius and angle pairs and produces the necessary SVG path that would connect them in series, starting with the first. The default accessors assume each pair has the following form, where the angle is measured in radians.

```
[43, Math.PI * 0.5]
```

```
<!DOCTYPE html>
<html>
  <head>
    <style>
    body{ margin: 0; }
    circle { fill: orange; stroke: #333; }
    path { fill: none; stroke: #333; stroke-width: 4; }
    </style>
  </head>
  <body>
  <script src="http://d3js.org/d3.v3.min.js"></script>
  <script>

var svg = d3.select('body').append('svg');
// our radial line generator
var lineRadial = d3.svg.line.radial();

var n = 1000, max_r = 250, rotations = 10;
var data = d3.range(n).map(function(d){
  var t = d / (n - 1);
  return [ t * max_r, t * Math.PI * rotations * 2 ];
});
```

```
var link = svg.append('path').datum(data).attr('d', lineRadial)
  .attr('transform', 'translate(300,300)');

  </script>
  </body>
</html>
```



Array helpers

D3 is packed with a bunch of object helpers that allow us to massage our data into more meaningful data structures. This process, often called "cleaning" our data is a common task for data visualizers.

> We'll discuss this process in-depth in chapter 7.

JavaScript itself includes many methods to help us modify our datasets in a meaningful way. We can reverse the order of the elements in our array, modify the contents of our arrays, append two together, filter through elements, etc.

The libraries `lodash` and `underscore` also provide convenient methods for handling many different cases for which we can modify our data.

We'll walk through common methods for how we can manipulate our data in this chapter.

Ranges

We'll often use a `range` in our dadta visualization tools. A range is an array containing an arithmetic progress of numbers. These numbers can be integers of floating point numbers. This is particularly useful when we're creating `axis` where we want to show a range of numbers in a graph, for instance.

We'll use the method `d3.range([start,] stop[, step])`, which takes up to three arguments to generate arrays of incrementing/decrementing sequences:

`start`

The `start` argument is defaults to `0` and is considered optional. This is where the range array will start.

`stop`

The `stop` argument is required and will determine where the range will stop. Note that the resulting array will exclude the stop number.

`step`

The `step` determines how much we'll increment or decrement our counter by. If it's positive, then it will increase the next value by the `step` amount. If it's negative, then it will decrease the next value by the `step` amount.

If it's omitted, then `step` defaults to `1`.

This enables us to generate arrays, such as the following:

```
d3.range(0, 5); // => [0, 1, 2, 3, 4]
d3.range(0, 6, 2); // => [0, 2, 4]
d3.range(5); // => [0, 1, 2, 3, 4]
d3.range(3, 1, -0.5); // => [3, 2.5, 2, 1.5]
```

Permutations

We can generate permutations of our data based upon their position in an array. This is useful for laying out our data in a table, for instance.

The `d3.permute(array, indexes)` takes two arguments:

`array`

The `array` is the original array that we'll be pulling the data for our `indexes`.

`indexes`

The `indexes` are a list of integers that represent the location from within the array that we're interested in.

```
// => ['c', 'b', 'a']
d3.permute(['a', 'b', 'c'], [2, 1, 0])
// => ['a', 'a', 'b']
d3.permute(['a', 'b', 'c'], [0, 0, 1])
// => ['c', 'b', undefined]
d3.permute(['a', 'b', 'c'], [2, 1, 4])
```

Accessing and manipulating simple arrays

array.forEach()

array.forEach()[40] is actually not a feature of D3 but commonly underused by developers.

The `array.forEach()` method is useful for iterating over an array of items without worrying about for-loops or needing to keep track of the current index.

It accepts a two possible arguments:

callback

The `callback` parameter is a function that will be called for every single element in the array. This function will be called with the arguments:

[40] https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/forEach

element The `element` is the element value of the array.

index The `index` is the integer location inside of the array.

originalArray The `originalArray` points to the original array being traversed.

thisArg

The `thisArg` is an optional argument that allows us to set the `this` value to be set inside the callback function.

If we pass the `thisArg` in the `array.forEach()` method, then this will be the first argument the callback function will be called with. For instance: `function(thisArg, element, index, originalArray)`

'LoopingwithoutforEach()'

```
var notes = ['Eb', 'Bb', 'F', 'C'],
    note, index;
for(index = 0; index < notes.length; index++)
{
  note = notes[index];
  console.log('beat', index, 'note: ', note);
}
```

'Andwith'

```
['Eb', 'Bb', 'F', 'C']
.forEach(function(note, index) {
  console.log('beat', index, 'note: ', note);
});
```

If we want to **stop** the iteration, we can simply return false from the function and it will halt it's execution.

array.map()

The array.map[41] function is not a feature of D3, but an often overlooked feature of JavaScript.

Similar to the `array.forEach()` function, we can use it to iterate over items in our array but it has the added feature of allowing us to easily create new arrays by returning the item we'd like to add at our current index.

The new array returned from `map` will always be the same length of our original array.

A common task in data visualization is to convert data in one format to data in another.

For instance, say we had an array of arrays of values that represent `x,y` positions at successive time steps, like so `[ [1, 2], [3, 4], ... ]`), but some other code expects instead to receive an array of objects of the form `[{x: 1, y: 2}, { x: 3, y: 4}, ...]`. We can concisely achieve this using `array.map()`:

```
// our original positions
var positions = [
  [1, 2],
  [3, 4],
  [5, 6],
  [7, 8],
  [9, 10]
];


// converting data without map()
var newPositions = [];
for(var i = 0; i < positions.length; i++)
{
  newPositions
  .push({
    x: positions[i][0],
    y: positions[i][1]
  });
}

// and with map()
var newPositions = positions.map(
  function(pos){
    return { x: pos[i][0], pos[i][1] };
});
```

[41]https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/map

array.filter()

TODO:

array.sort()

TODO:

d3.merge(arrays)

We can merge any number of arrays into a single array. This is similar to the built-in array concat method.

```
d3.merge([[1,2], [3,4], [5,6,7]]);
// [1,2,3,4,5,6,7]
d3.merge([[1,2], [3, [4, 5]]])
// [1, 2, 3, [4, 5]]
```

d3.zip(arrays)

The `d3.zip()` method returns an array of arrays where the ith array contains the same ith element from each of the argument arrays. If the *arrays* contains a single array, then the array contains a one-element array. If we don't pass any arrays, then the returned array is an empty array.

```
d3.zip([1, 2], [3, 4])
// [[1, 3], [2, 4]]
d3.zip([1, 2], [3, 4, 5])
// [[1, 3], [2, 4]]
d3.zip([1, 2])
// [[1], [2]]
d3.zip([1, 2], [3, 4], [5, 6])
// [[1, 3, 5], [2, 4, 6]]
```

d3.transpose()

We can run a transposition on an array (which is functionally equivalent to `d3.zip.apply(null, matrix)`) on a matrix. It turns out that transposition of matrices is incredibly useful for quick calculations and matrix math.

The `transpose(array)` method takes a single argument of a matrix (defined as an array).

```
d3.transpose([[1,2,3], [3,2,1]]);
// [[1,3], [2, 2], [3, 1]]
d3.transpose([[], [3,2]]);
// []
d3.transpose([[1,2, 3], [3]]);
// [[1, 3]]
```

d3.pairs()

We also can associated adjacent pairs of elements in an array to return tuples of elements using the `d3.pairs()` method. This method takes a single argument of an array.

```
d3.pairs([1, 2, 3, 4]);
// [ [1,2], [2,3], [3,4]]
d3.pairs([1])
// []
d3.pairs([1,2])
// [ [1,2] ]
```

d3.ascending()/d3.descending()

Often times, data visualizations will need to sort arrays. D3 includes two helpers functions to help us with this type of sorting.

The `d3.ascending()` and `d3.descending()` are helper functions that accept two arguments.

For the `d3.ascending(a,b)`, it will return a value of `-1` if a is less than b, return `1` if a is greater than b, and `0` if they are equal.

The `d3.descending(a,b)` function returns the opposite, where it will return `-1` if a is larger than b, `1` if a is less than b, and `0` if they are equal.

These functions can be used by passing them into the array `sort()` function, like so:

```
var arr  =[1,28,9,4];
arr.sort(d3.ascending);
// => [1, 4, 9, 28]
arr.sort(d3.descending);
// => [28, 9, 4, 1]
```

extent()

Extent will return an array of two elements, the minimum and the maximum of the array:

```
var arr = [1,28,9,4];
d3.extent(arr);
// => [1, 28]
```

min(), max(), sum(), mean(), median()

Often times we want to pull out metrics of our array. D3 gives us the ability to pull out key metrics of our arrays with some helper functions.

Each one of these functions takes up to two arguments:

array The `array` argument is the array of which we will run the function on.

accessor The `accessor` function is an optional function that will get run before the actual computation runs. We can use this function to compute values for which we are interested in, such as ignoring outliers in data.

sum()

We can get the sum of the array using the `d3.sum(array)` function, which returns `0` for an empty array. The `sum()` function ignores invalid values such as `NaN` and `undefined`.

```
var arr = [1,28,9,4];
d3.sum(arr)
// => 42
var obj = [
  {x: 1, y: 20},
  {x: 28, y: 5},
  {x: 9, y: 5},
  {x: 4, y: 2}
]
d3.sum(obj, function(d) { return d.x; })
// => 42
```

min()/max()

We can pick out the minimum and the maximum of the array by using the D3 `min()` and `max()` functions. These functions are both different from their built-in Math counterparts in that they ignore undefined values.

They also compare using natural ordering, rather than using numeric ordering. This means that the maximum of [“20”, “3”] is “3”, while [20, 3] the maximum is 20. The same is true for the minimum, but in reverse.

```
var arr = [1,28,9,4];
d3.min(arr)
// => 1
d3.max(arr)
// => 28
var obj = [
  {x: 1, y: 20},
  {x: 28, y: 5},
  {x: 9, y: 5},
  {x: 4, y: 2}
]
d3.min(obj, function(d) { return d.x; })
// => 1
d3.max(obj, function(d) { return d.x; })
// => 28
```

mean()

The `mean()` function returns us the mean of a given array. The mean is the average of the values in the array, or the *central* value of the array.

```
var arr = [1,28,9,4];
d3.mean(arr);
// => 10.5
var obj = [
  {x: 1, y: 20},
  {x: 28, y: 5},
  {x: 9, y: 5},
  {x: 4, y: 2}
]
d3.mean(obj, function(d) { return d.x; })
// => 10.5
```

median()

The `median()` function returns the median of the given array using the R-7 algorithm. The median is the "middle" of the array.

```
var arr = [1,28,9,4];
d3.median(arr);
// => 6.5
var obj = [
  {x: 1, y: 20},
  {x: 28, y: 5},
  {x: 9, y: 5},
  {x: 4, y: 2}
]
d3.median(obj, function(d) { return d.x; })
// => 6.5
```

There are a lot of other simple array helper functions available through D3. For more functions and deeper descriptions of their functionality, check out the documentation available at https://github.com/mbostock/d3/wiki/Arrays[42].

Associative array helpers

Associative arrays are like simple objects. They have a set of named properties. D3 offers a few helpers for converting these maps into standard arrays, which are generally more useful in visualizations.

keys()

The `keys()` method returns an array of the key names of the specific object.

```
d3.keys({x: 1, y: 2, z: 3})
// => ["x", "y", "z"]
```

values()

The `values()` method returns an array containing all of the values of a specific object.

```
d3.values({x: 1, y: 2, z: 3})
// => [1, 2, 3]
```

entries()

Sometimes we want the specific array to contain both the key name and the value name. This is particularly useful when we're interested in dynamic objects where we are interested in a single key by name.

[42]https://github.com/mbostock/d3/wiki/Arrays

```
d3.entries({x: 1, y: 2, z: 3})
[
  {key: "x", value: 1},
  {key: "y", value: 2},
  {key: "z", value: 3}
]
```

Maps

Maps are similar to objects in JavaScript, but when using any built-in object behavior and/or built-in keys (such as `__proto__` or `hasOwnProperty`), the object behavior will tend to act unexpectedly.

ES6 will implement simple maps and sets, but doesn't quite implement them yet. As sets and maps are incredibly useful in data visualization

ECMAScript 6 (ES6) is the upcoming sixth major release of the ECMAScript language specification. ECMAScript is the name for the language commonly referred to as JavaScript.

We can create a new D3 map object by using the `d3.map()` function. The function itself can take a single, optional argument of an object.

If an object is passed, then all of the properties of the object are copied into the map.

```
var m1 = d3.map(); // map object
var m2 = d3.map({x: 1});  // map object with
                          // x as a key and
                          // 1 as the value
```

map.has()

We can test if a map includes a key or not using the `map.has()` method. It takes a single argument of a *key* string.

```
var m = d3.map({x: 1})
m.has("x"); // true
m.has("y"); // false
var key = "x";
m.has(key); // true
```

map.get()

We can fetch the value of a key inside of our map by using the `map.get()` function. It takes a single argument of a key string to fetch.

```
var m = d3.map({x: 1})
m.get("x"); // 1
m.get("y"); // undefined
```

map.set()

We can set a value inside of our map as well using the `map.set()` method. This takes two arguments, the key (string) to set and the value to set it. If the map already *knows* about the key, it will replace the old value with the new value.

```
var m = d3.map({x: 1})
m.set("y", 2);
m.set("x", 2);
// m => {x: 2, y: 2}
```

map.remove()

We can also delete a key that is in our map by using the `map.remove()` function. This takes a single argument, the key (string) to remove out of the map.

```
var m = d3.map({x: 1})
m.set("y", 2);
m.remove("x");
// m => {y: 2}
```

map.keys()

We can fetch an array of all the keys in the map by using the `map.keys()` function.

```
var m = d3.map({x: 1})
m.set("y", 2);
m.keys(); // => ["x", "y"]
```

map.values()

Similarly, we can fetch an array of all the values in the map using the `map.values()` function.

```
var m = d3.map({x: 1})
m.set("y", 2);
m.values(); // => [1,2]
```

map.entries()

We can also return an array of key-value based objects with two keys of `key` and `value`. This is useful for picking out certain objects inside of a map.

```
var m = d3.map({x: 1})
m.set("y", 2);
m.entries();
// [{key:"x", value:1}, {key:"y",value:2}]
```

map.forEach()

Finally, we can iterate over every entry in the map by using the `map.forEach()` function. This takes a single argument of a function that will be called for every single entry in the map. This function will be called with two arguments, the `key` and the `value` like so:

```
var m = d3.map({x: 1})
m.set("y", 2);
m.forEach(function(key, value) {
  console.log(value);
});
// 1
// 2
```

> The `this` context inside of the function points to the map itself

Sets

Sets are very similar to maps, with the exception that they can only have a single value associated in the set.

We can create a set by calling the `d3.set()` method. This method takes a single, optional argument of an array.

If an array is passed in as an argument, every value in the array will be returned as a member of the set.

```
var s = d3.set([1,2,"foo","bar"])
// s = [1,2,"foo","bar"]
```

set.has()

We can test if the set has the value set as an entry by using the `set.has()` method. This method takes a single argument of a `value`. The method will return true if and only if the set has an entry for the value string.

```
var s = d3.set([1,2,"foo","bar"])
s.has(1); // true
s.has("donald_duck"); // false
```

set.add()

We can add a value to the set by using the `set.add()` function. It takes a single argument of `value`, the value to append to the set.

```
var s = d3.set([1,2,"foo","bar"])
s.add(3);
s.has(3); // true
```

set.remove()

We can remove a value using the `set.remove()` method. This remove method takes a single argument of the value to remove from the set. If the value is in the set, it will remove the value (and return true). If the value is *not* in the array, then it will do nothing and return false.

```
var s = d3.set([1,2,"foo","bar"])
s.remove(1); // true
s.remove("space"); // false
s.has(1); // false
```

set.values()

Just like in arrays, we can get all of the values in the set. This will return an array that includes all of the “unique” values in the set.

```
d3.set([1,2,1,3]).values(); // ["1","2","3"]
```

set.forEach()

We can run a function for every single value in our set as well by using the `set.forEach()` function. The `set.forEach(fn)` method takes a single argument of the function to run for every element.

The function will be run with a single argument of the value in the set.

```
var s = d3.set([1,2,"foo","bar"])
s.forEach(function(val) {
  console.log("Value: " + val);
});
// Value: 1
// Value: 2
// Value: foo
// Value: bar
```

The `this` context inside the function points to the set itself.

Nests

A nest allows elements in an array to be grouped into a hierarchical tree structure. Nests allows us to neatly format our data into meaningful grouping contexts.

Nests allow us to format data in groupings that makes sense for our use. For instance, if we have the population of countries in the world, the continent they are on and their population, it would be useful for us to be able to view the population based upon continent.

We can set up our data in a nest and then populate the nest data using a key of the data:

```
var raw_country_data = [{
      "countryName": "Andorra",
      "continent": "EU",
      "languages": "ca",
      "areaInSqKm": "468.0",
      "population": "84000"
    },
    {"countryName": "United Arab Emirates",
      "continent": "AS",
      "languages": "ar-AE,fa,en,hi,ur",
      "areaInSqKm": "82880.0",
      "population": "4975593"
    },
    {"countryName": "Antigua and Barbuda",
      "continent": "NA",
      "languages": "en-AG",
      "areaInSqKm": "443.0",
      "population": "86754"
    },
    {"countryName": "Anguilla",
      "continent": "NA",
      "languages": "en-AI",
      "areaInSqKm": "102.0",
      "population": "13254"
    },
    {"countryName": "Albania",
      "continent": "EU",
      "languages": "sq,el",
      "areaInSqKm": "28748.0",
      "population": "2986952"
    },
    {"countryName": "Armenia",
      "continent": "AS",
      "languages": "hy",
      "areaInSqKm": "29800.0",
      "population": "2968000"
    }
  ]
var data = d3.nest()
    .key(function(d) { return d.continent })
    .sortKeys(d3.ascending)
    .entries(raw_country_data);
// [
//   key: "AS", values: [
//     {continent: "AS", ...
//   ]
// ]
```

The `d3.nest()` function creates a new nest operator. It takes no arguments, but allows us to call all nests functions on it.

nest.key()

To set up the key functions, we'll use the `nest.key()` method. This method takes a single function that will be invoked for each element in the input array and it is expected to return a string identifier that's used to assign the element to its group.

In the above example, we set up the `key()` function to return the string identified by the `d.continent` accessor. These are usually keys to set up accessors.

We can create multiple keys functions to set up multiple accessors (which will result in an additional hierarchy level).

```
var nest = d3.nest()
  .key(function(d) { return d.continent })
  .key(function(d) { return d.languages });
```

nest.sortKeys()

Often times we'll want to sort the entries so that we'll look at values ascending or descending. In the case of our example from above, we are sorting on the countryName, so that we get an alphabetical listing of our countries in our data.

```
var nest = d3.nest()
  .key(function(d) { return d.continent })
  .sortKeys(d3.ascending);
```

nest.sortValues()

We can also sort the values of our elements using the `nest.sortValues()` function. This is similar to sorting the input array before applying the nest operator, except that it tends to be more efficient as the groupings tend to be much smaller than the entire dataset.

```
var nest = d3.nest()
  .key(function(d) { return d.continent })
  .sortValues(d3.ascending);
```

nest.rollup()

We can *rollup* data that we are interested in, rather than simply returning the raw sorted data.

For instance, if we want to return the area in square miles for each continent that we know about, we can create a *rollup* function using the `nest.rollup(function)`. The method takes a single argument that will be called on each of the elements.

```
var data = d3.nest()
    .key(function(d) { return d.continent })
    .sortKeys(d3.ascending)
    .rollup(function(d) {
      return {
        area: d3.sum(d, function(g) {
          return +g.areaInSqKm;
        })
      }
    })
    .entries(raw_country_data);
// [{
//    "key":"AS",
//    "values":{
//      "area":112680
//    }
//  }, {
//    "key":"EU",
//    "values":{
//      "area":29216
//    }
//  }, {
//    "key":"NA",
//    "values":{
//      "area":545
//    }
//  }]
```

nest.map()

We can apply the nest operator to a specific array using the `nest.map()` function. This will allow us to create a map of the data, rather than simply returning an array (as we've seen above).

The `map()` method takes up to two arguments. It accepts the array, which is the input array whose data will be applied in the nest. It can also take an optional `mapType` function. If this is specified, then it can return a different type, such as a `d3.map`, rather than a simple object.

```
var data = d3.nest()
    .key(function(d) { return d.continent })
    .sortKeys(d3.ascending)
    .rollup(function(d) {
      return {
        area: d3.sum(d, function(g) {
          return +g.areaInSqKm;
        })
      }
    })
    .map(raw_country_data);
// {
//  AS: { area: 112680 },
//  EU: { area: 29216 },
//  NA: { area: 545 }
// }
```

nest.entries

As we've done in the above examples, we can apply data to our nest using the `nest.entries(arr)` function. This simply takes a single array as it's input to pull data for the nest.

The `entries()` function runs on all levels of the hierarchy.

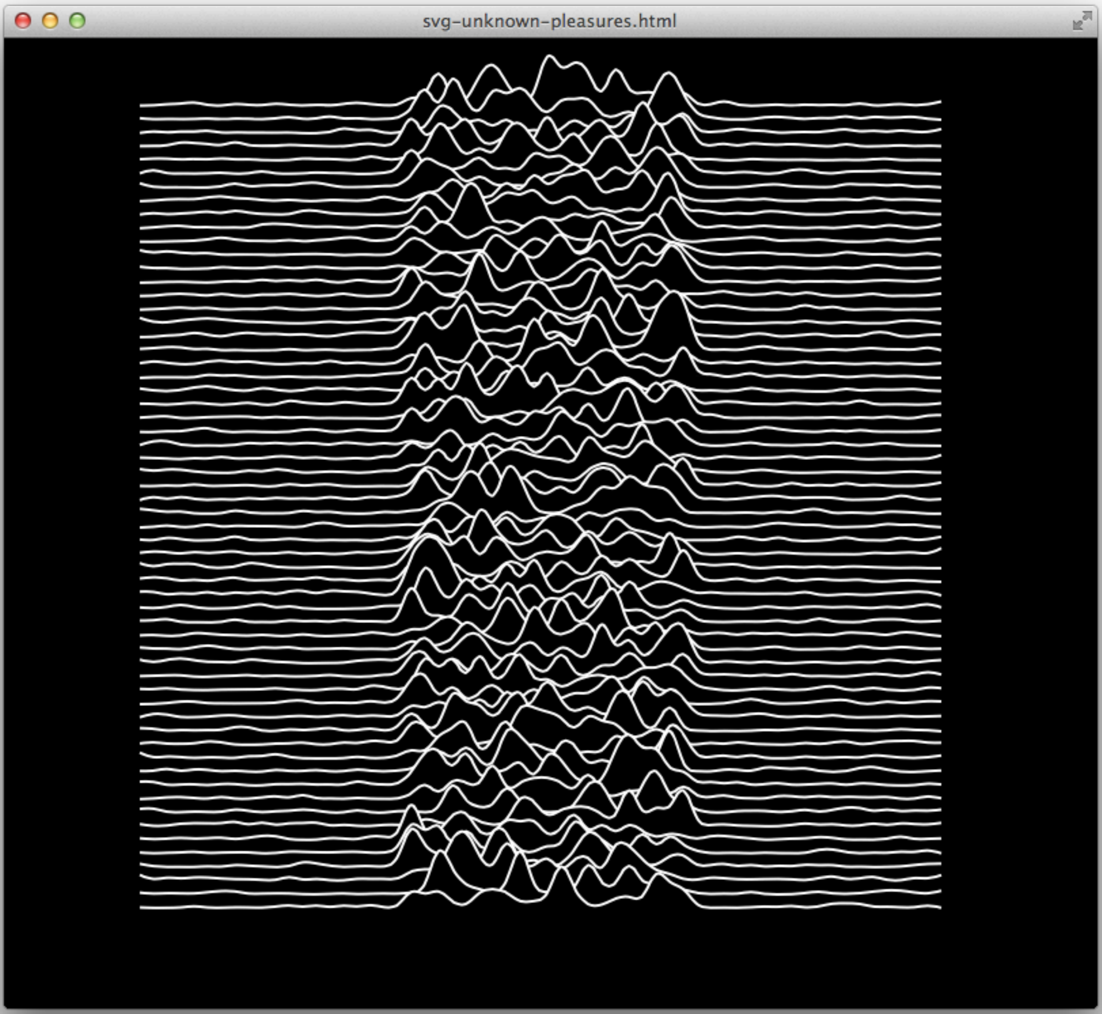
```
var data = d3.nest()
    .key(function(d) { return d.continent })
    .sortKeys(d3.ascending)
    .entries(raw_country_data);
```

Applying our knowledge

We'll combine all we've learned so far to recreate the iconic album cover of Unknown Pleasures by Joy Division using just `d3.range`, `array.map` and the line generator `d3.svg.line()`.

```
<!DOCTYPE HTML>
<html>
  <head>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <style>
      /* make paths 2px wide and white, and give them a black fill. */
      path { stroke: white; stroke-width: 2; fill: black; }
      /* make the svg and body black */
      body, svg { background-color: black; }
    </style>
  </head>
  <body>
    <script>
var svg = d3.select('body').append('svg');
// generate random data that squiggles a lot in the middle
var data = d3.range(50, 650, 10).map(function(y_offset){
  // generate a line of the form [[6,7],[22,34],...]
  return d3.range(100, 700, 10).map(function(d){
    var y = d;
    if(y < 300 || y > 500) y = 50; else y = 500;
    return [d, y_offset - Math.random() * Math.random() * y / 10];
  });
});
// create our line generator
var line = d3.svg.line()
```

```
  .x(function(d) { return d[0]; })
  .y(function(d) { return d[1]; })
  .interpolate("basis");
// create several path elements using our line generator and squiggle data
svg.selectAll('path').data(data).enter()
  .append('path').attr('d', line);
    </script>
  </body>
</html>
```



Scales

As we build our datasets and start to try to apply them to our visual canvas, we'll need to set up scales so that our data can sit nicely on our page.

It's very unlikely that any of the data that we'll work with will ever translate to our pixel coordinates in our visualizations. We'll want to scale our data so that our data reflects the relationships we have to our pixel data.

For this purpose, D3 includes a convenient helper object that we'll use to create a scaled relationship between our data and the relative pixel coordinates to represent that data.

When we talk about scales, we're talking about relative data points that we can use to map to data relative to other points.

Why scales

For instance, what if wanted to plot the number of viewers[43] for each episode of the third season of the AMC television show, *The Walking Dead* with a bar chart?

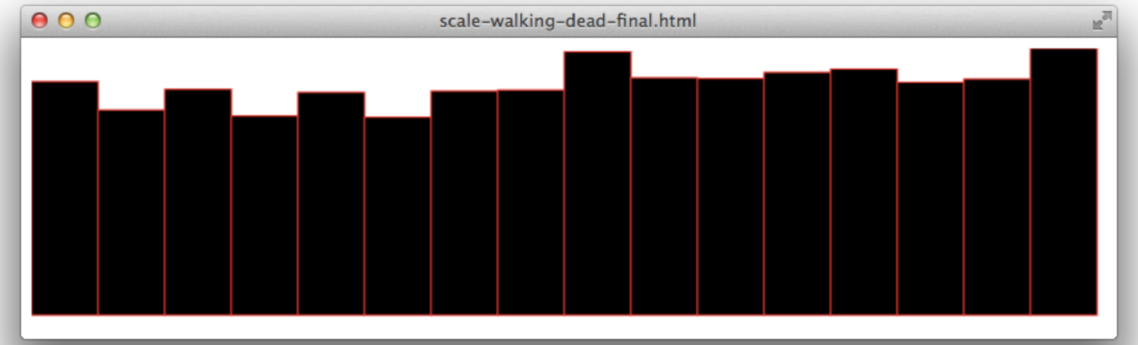
Each bar chart would not stop until it was *way* outside of the visible screen space. We just don't have 10 million pixels, one to represent each viewer!

One solution would be to step through and apply a scaling factor to each data point so that the maximum point wouldn't go past the top of are chart.

```
<!DOCTYPE HTML>
<html>
  <head>
    <script src="http://d3js.org/d3.v3.min.js"
      charset="utf-8"></script>
    <style> rect { stroke: red; stroke-width: 1; fill: black; } </style>
  </head>
  <body>
    <script>
      // viewers for each episode
      var data = [ 10870000, 9550000, 10510000,
                  9270000, 10370000, 9210000,
                  10430000, 10480000, 12260000,
                  11050000, 11010000, 11300000,
                  11460000, 10840000, 10990000,
                  12420000 ];
```

[43] http://en.wikipedia.org/wiki/List_of_The_Walking_Dead_episodes

```
      // scaling data
      var height = 200;
      // the episode with the most views
      var max = d3.max(data);
      // Our scale factor
      var scale = height / max;
      var scaled_data =
        data.map(function(d){
          return d * scale;
        });

      var svg = d3.select('body')
                .append('svg')
                .selectAll('rect')
                .data(scaled_data)
        // add all the <rect> tags
        .enter().append('rect')
        .attr({
          width: 50,
          height: function(d){ return d },
          x: function(d, i){ return i * 50 },
          // y pixels count down
          y: function(d){ return height - d }
        });
    </script>
  </body>
</html>
```

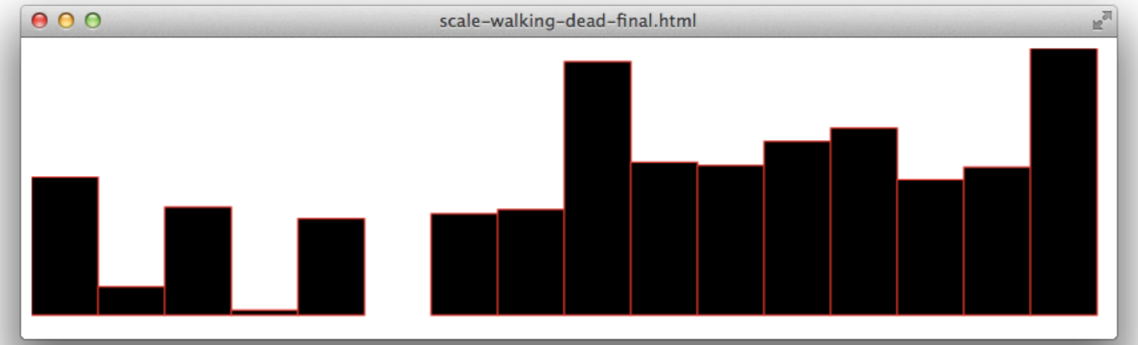


Not bad, but we have to take into account a lot of extra manual work. Now, we have to worry about this extra array `scaled_data`.

We also joined our data with this `scaled_data` array instead of the original, which makes it difficult to add features to the visualization, like adding tool-tips to each bar chart.

Our data is also bunched up at the top so it's not easy to see the differences. These differences would be more apparent if the data was plotted from the minimum episode to the highest, to emphases these differences.

```
// scaling data
var height = 200;
var min = d3.min(data);
var range = d3.max(data) - min;
var scale = height / range;
var scaled_data = data.map(function(d){
  return (d - min) * scale;
});
```



That looks a lot better, but took more code than it needed and has the side effect of overriding our original data.

D3's linear scale generator eliminates both these corners by giving us a function we can call anytime we want to convert an input range (viewers in our case) to an output domain (most often, pixels.) Scales, essentially allow us to describe the relationship between our data and pixels.

This allows us to be expressive about the range of possible input data values and limit the possible output domain to translate the data into a new domain.

Creating a scale

We can easily create a scale of different types by using functions given to us by the `d3.scale` object. To create a linear scale, we'll use the `d3.scale.linear()` function:

```
var scale = d3.scale.linear();
```

This scale, although relatively useless will map input values to output values at a 1:1 dimension. That is, calling it with 2, we'll get an output of 2.

We have not actually applied any domains or ranges on this scale object, so it only assumes `1:1`.

Setting input domain

We can set an input domain by using the `.domain()` function.

```
var scale = d3.scale.linear()
        .domain([0, 100]);
```

This will set the input domain to the start from 0 and go to 100. This will set the minimum number of our domain to be at `0` and our max to scale to `100`.

If we don't pass numbers into the array, then the values will be coerced into numbers. For instance, this happens with Date objects (although D3 does have a `d3.time.scale` object we can use for dates which is often more convenient).

Often times, we'll want the input domain to match our dataset. Rather than set the domain values manually, more often than not, we'll set these using the min and max of our dataset.

We can do this using the `d3.min()` and `d3.max()` functions or by using the `d3.extent()` function.

```
var color = d3.scale.linear()
    // Using straight-forward min and
    // max with two functions
    .domain([d3.min(data), d3.max(data)])
    // OR we can use d3.extent() function:
    .domain(d3.extent(data))
```

We can also pass in a *polylinear* scale by passing in more than a single value into the array.

```
var scale = d3.scale.linear()
      .domain([-1, 0, 1]);
```

With a multi similar input value for multiple output range, we can define diverging quantitative scales. For instance, this is useful for scaling between colors for different ranges of our input.

```
var color = d3.scale.linear()
    .domain([-1, 0, 1])
    .range(["red", "white", "green"]);
```

Setting output range

We can set the output *range* by using the `.range([values])` function:

```
var scale = d3.scale.linear()
      .domain([0, 1])
      .range([10, 100]);
```

The `range()` function sets the output range to span between `10` and `100` for output values. It accepts a single array of values to match the cardinality of the input domain. That is, the same amount of numbers as the domain.

One feature of the `range()` function is that it does not need to be numeric inputs, so long as the underlying interpolation (depending upon which type of scale we're working with, i.e. linear, time, etc) supports it.

This enables us to scale between colors, for instance.

```
var color = d3.scale.linear()
    .domain([-1, 0, 1])
    .range(["red", "white", "green"]);
```

Using our scale

When it comes time to translate our data to pixels, we can just use `scale(d)`

For instance, let's say that our domain ranges from `0` to `2` that we want to map on to a chart of between `0` to `500`.

```
var scale = d3.scale.linear()
        .domain([0, 2])
        .range([0, 500]);
```

If we pass in `0` to our resulting `scale()` function, we'll get an output value of `0`. If we pass in `2`, then we will get our maximum output of `500`.

```
scale(0); // 0
scale(2); // 500
```

We can use this scale to provide a gradient between our values of `0` and `2`. For instance:

```
scale(0.5); // 125
scale(1.1); // 275
scale(1.9); // 475
```

We'll most often use this `scale()` function when setting an attribute for

As we can see, our scales output relative ranges for each datum. This process is called *normalization*, which is mapping a numeric value to all possible minimum and maximum values on a given scale.

If it takes 100 licks to get to the center of a tootsie roll, then 90 licks in and we are at 90% of the way done, while at 10 licks we're only at 10%. This is called normalization.

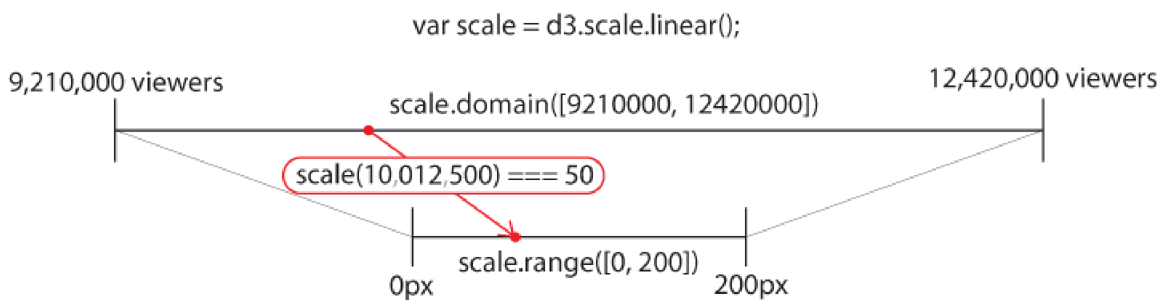
Back to our mapping of viewership, we can create a scale that maps on to our entire dataset:

```
var scale = d3.scale.linear()
  .domain(d3.extent(data)])
  .range([0, 200]);
```

This will map the entire range of our input values from the minimum values to the maximum viewership values.

We can now use this scale to set the height of our bar chart:

```
// count up from height,
// since y counts down
svg.selectAll('rect')
  .data(data).enter()
  .append('rect')
  .attr('y', function(d){
    return height - scale(d);
  })
```



Scales can be used to map our data to pixels

We've covered how to use the `d3.scale.linear()` scale in this chapter, but there are quite a few more options for us in terms of using scales when mapping input domain data to output ranges.

Other non-linear scales

D3 includes other scales than simply linear mappings. For instance, we can create a logarithmic scale using the `d3.scale.log()` function.

We can create a new power scale by using the `d3.scale.pow()` method.

The logarithmic scale is useful for translating data that increases exponentially. The power scale, like the logarithmic scale is useful for translating data that increases exponentially. These two scales output format are slightly different in their computation of the output range.

All of these scales work very similar to each other, with slight differences. Now that we know how they work, we can examine the D3 documentation[44] for more detailed explanation of each of the functions.

Axises

In any visualization where we're trying to convey meaning, having the viewer understand the scale of data in comparison to other data is necessary. One easy way we can simply convey this type of data is by using Axises in our visualizations.

D3 provides fantastic helper functions that help us construct the necessary annotations that combine to form an axis, such as tick marks, labels and reference lines.

As long as we properly create our scale and use it to build our axis, D3 can take care of constructing the building function.

Creating an axis

To create an axis, we can use the `d3.svg.axis()` method, like so:

[44] http://d3js.org

```
var axis = d3.svg.axis()
```

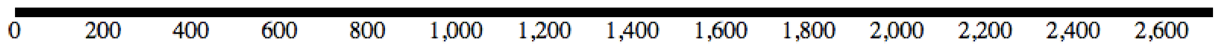
That's it. We'll discuss ways to customize this shortly.

We'll only need to add the axis to our SVG element. It's possible to call our new `axis()` function directly, ie., `axis(svg)`.

Although this does technically work, it doesn't allow us to chain our methods together, like the rest of D3.

To support chaining, we'll instead use the form `svg.call(axis)` to create the axis on our svg:

```
<!DOCTYPE html>
<html>
  <body>
    <script
      src="http://d3js.org/d3.v3.min.js"
      charset="utf-8"></script>
    <script>
      var width = 800;
      // the heights of the worlds latest buildings, in ft.
      var data = [2717, 2073, 1971,
            1776, 1670];
          // create our svg and add it
          // to the `<body>`
      var svg = d3.select('body')
          .append('svg');
          // create a scale that goes from
          // [0, max(data)] -> [10, width]
      var scale = d3.scale.linear()
            .domain([0, d3.max(data)])
            .range([10, width]);
          // create a new axis that
          // uses this new scale
      var axis = d3.svg.axis()
            .scale(scale);
         // add the new axis to the svg.
         // same as `axis(svg);` except
         // that it returns `svg`
      svg.call(axis);
    </script>
  </body>
</html>
```



hello axis

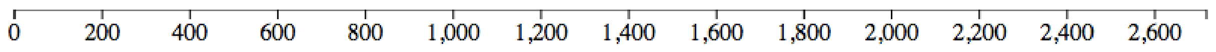
```
<!-- the resulting SVG -->
<svg>
  <!-- first tick mark and label -->
  <g class="tick"
      transform="translate(10,0)"
      style="opacity: 1;">

    <line y2="6" x2="0"/>
    <text y="9" x="0" dy=".71em"
      style="text-anchor: middle;">0</text>
  </g>
  <!-- second tick mark and label -->
  <g class="tick"
    transform='translate(68.15237394184763,0)'
    style="opacity: 1;">
    <line y2="6" x2="0"/>
    <text y="9" x="0" dy=".71em"
          style="text-anchor: middle;">
            200
    </text>
  </g>
  <!-- ...etc... -->
  <path class="domain" d="M10,6V0H800V6"/>
</svg>
```

TODO:

It's fairly common to add additional styles for the axis as the default axis isn't very attractive. To do this, we'll add a class to a containing `<g>` element which we'll then add the axis component to.

```
svg.append('g').call(axis).attr('class', 'x axis');
```

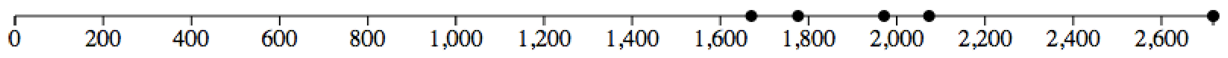
```
.x.axis path, .x.axis line{
  fill: none; stroke: black;
}
```



axis with style

Now lets add some data points, to represent the heights of the worlds top 5 tallest buildings. They'll just be circles of radius, `r=4`, positioned at their respective locations along the axis. We'll also translate the down 100 pixels so they don't get clipped off the edge of the screen.

```
var height = 100;
// move the axis down 100 pixels
svg.append('g').call(axis).attr('class', 'x axis')
  .attr('transform', 'translate(0,' + height + ')');


// add some circles along the axis that present the building heights
svg.selectAll('circle').data(data)
  .enter().append('circle').attr('r', 4).attr('transform', function(d){
    return 'translate(' + scale(d) + ', ' + height + ')'
  });
```



add some data points

Sick! Now how about we add a few labels? We'll need more data for that. Specifically, we'll need to relate building heights to building names.

```
var data = [
  [ "Burj Khalifa"                 , 2717],
  [ "Shanghai Tower"               , 2073],
  [ "Makkah Royal Clock Tower Hotel", 1971],
  [ "One World Trade Center"       , 1776],
  [ "Taipei 101"                   , 1670]
];
```

This is the data we want but adding it directly into our existing project, without making any other changes, will break our code. This is because the position of the circles depends on the data being a flat array of integers, as well as our call to `d3.max()`. We'll need to change `scale(d)` to `scale(d[1])` since each data item is now itself an array, with the second element the height of the building. Similarly when we specify our scales `domain`, we'll need to give d3.max an accessor function, so it knows what part of our data item (datum) should be used to find the max of. In our case, this is the second element of our array.

```
var scale = d3.scale.linear()
  .domain([0, d3.max(data, function(d){ return d[1]; })])
  .range([10, width]);
```

Now that our code is working again, let's add those labels. Since each element of the original data array is now an array itself, we'll use `d[0]` to reference the building name and `scale(d[1])` to map the building height to the x position in pixels along the axis. We'll also give it a slight counter-clockwise rotation using `rotate(-20)` so the labels don't overlap. Finally, we'll apply one last translation, `translate(5,-5)`, so the label isn't bumped up against the data point.

```
// add the labels
svg.selectAll('text').data(data)
  .enter().append('text').text(function(d){ return d[0] })
  .attr('transform', function(d){
    return 'translate(' + scale(d[1]) + ', ' + height + ') '
      + 'rotate(-20) translate(5,-5)';
  });
```

Lets also format the axis labels using the axis `tickFormat()` method so people can tell what units we're using. D3 offers a variety of different formatters we can use to pass into tick-Format for our axis. If you're interested in reading about these, check out d3.format()[45] and d3.time.format()[46]. Our formatter, `d3.format(',.0f')`, rounds off decimals and add commas. ie., `d3.format(',.0f')(1000.04) === "1,000"`.

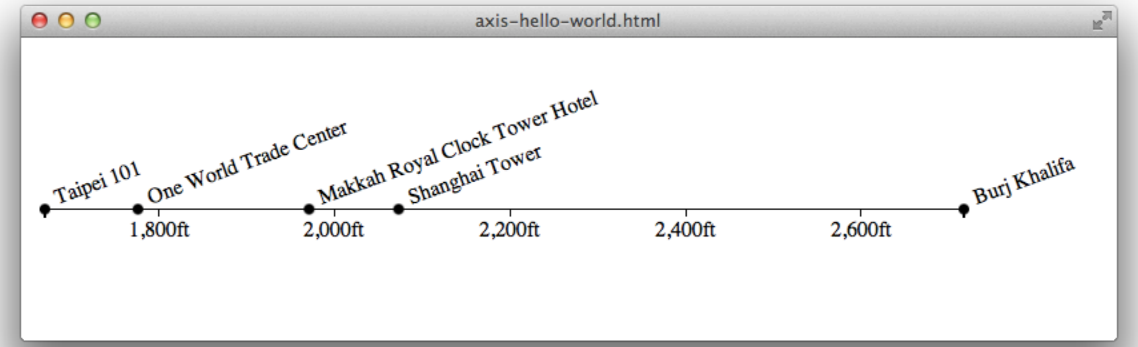
```
// create our axis and format the ticks
var axis = d3.svg.axis().scale(scale)
  .tickFormat(function(d){ return d3.format(',.0f')(d) + 'ft' }).ticks(5);
```

As a last touch, say we decide there's too much empty space before the start of the first data point. Lets update our scale to instead start at the first data point instead of zero. That change is as simple as changing our scales domain from `[0, d3.max(...)]` to `d3.extent(data, ...)`

[45]https://github.com/mbostock/d3/wiki/Formatting
[46]https://github.com/mbostock/d3/wiki/Time-Formatting

```
var scale = d3.scale.linear()
  // .domain([0, d3.max(data, function(d){ return d[1]; })])
  .domain(d3.extent(data, function(d){ return d[1]; }))
  .range([10, width]);
```

Here's what the final version looks like:



worlds tallest buildings

```
<!DOCTYPE html>
<html>
  <head>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <style>
      .x.axis path, .x.axis line{
        fill: none; stroke: black;
      }
    </style>
  </head>
  <body>
    <script>
var width = 700, height = 100;
var svg = d3.select('body').append('svg');
// worlds latest buildings
var data = [
  ["Burj Khalifa"                  , 2717],
  ["Shanghai Tower"                , 2073],
  ["Makkah Royal Clock Tower Hotel", 1971],
  ["One World Trade Center"        , 1776],
  ["Taipei 101"                    , 1670]
];
var scale = d3.scale.linear()
  .domain(d3.extent(data, function(d){ return d[1]; }))
  .range([10, width]);
// add the data points
svg.selectAll('circle').data(data)
  .enter().append('circle').attr('r', 4).attr('transform', function(d){
    return 'translate(' + scale(d[1]) + ', ' + height + ')'
  });
// add the labels
svg.selectAll('text').data(data)
  .enter().append('text').text(function(d){ return d[0] })
  .attr('transform', function(d){
    return 'translate(' + scale(d[1]) + ', ' + height + ') '
      + 'rotate(-20) translate(5,-5)';
  });
// create the axis
var axis = d3.svg.axis().scale(scale)
  .tickFormat(function(d){ return d3.format(',.0f')(d) + 'ft' }).ticks(5);
// add the axis inside a new `g`
svg.append('g').call(axis).attr('class', 'x axis')
  .attr('transform', 'translate(0,' + height + ')');
    </script>
  </body>
</html>
```

Animation and interaction

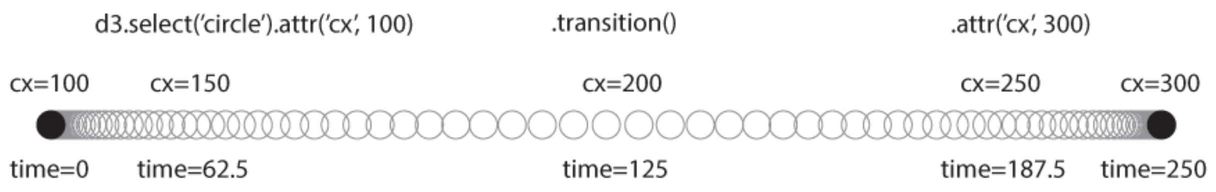
We'll cover the basics of animation and other interactive concepts in this chapter. These techniques lay the foundation for things like live updating, real-time data visualizations.

Transitions

At the hart of almost all animated data visualizations built with D3 our transitions. Transitions make it effortless for us to apply a smooth, animated, transition over a collection of selected elements. It's easier to show how this works so lets dive right in with a simple example. Reload the page if the animation wasn't noticed the first time.

```
<!DOCTYPE html>
<html>
  <body>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <script>
d3.select('body').append('svg')
  .append('circle').attr({r: 10, cx: 100, cy: 100})
  .transition().attr('cx', 300);
    </script>
  </body>
</html>
```

Wow, how cool is that? We took a `circle` and animated with the addition of only a single line of code! Let's break down what's happening here. Selectors have an additional method called `transition()` which will return *another selection*, specifically, a special "transition[47] selection", that will gradually update all the DOM element attributes and styles in the first selection with their new values, in the second selection. This all occurs over a duration of 250 milliseconds (1/4 of a second), the default duration for a transition.



hello transition

[47] https://github.com/mbostock/d3/wiki/Transitions

As a reminder, the object returned from calling `transition()` on a selector, it itself a selector, so we can easily use this same technique for selectors with multiple elements. If a we don't change a particular property or attribute on both selections, (the selector before the transition and the one after), that property will *not* be transitioned.

We can adjust how long a transition takes by using the `.duration()` method that exists only on transition selectors. Working from the above example, we can adjust the animation to take 4 seconds (4000ms), instead of 250ms, by using `transition().duration(4000)`.

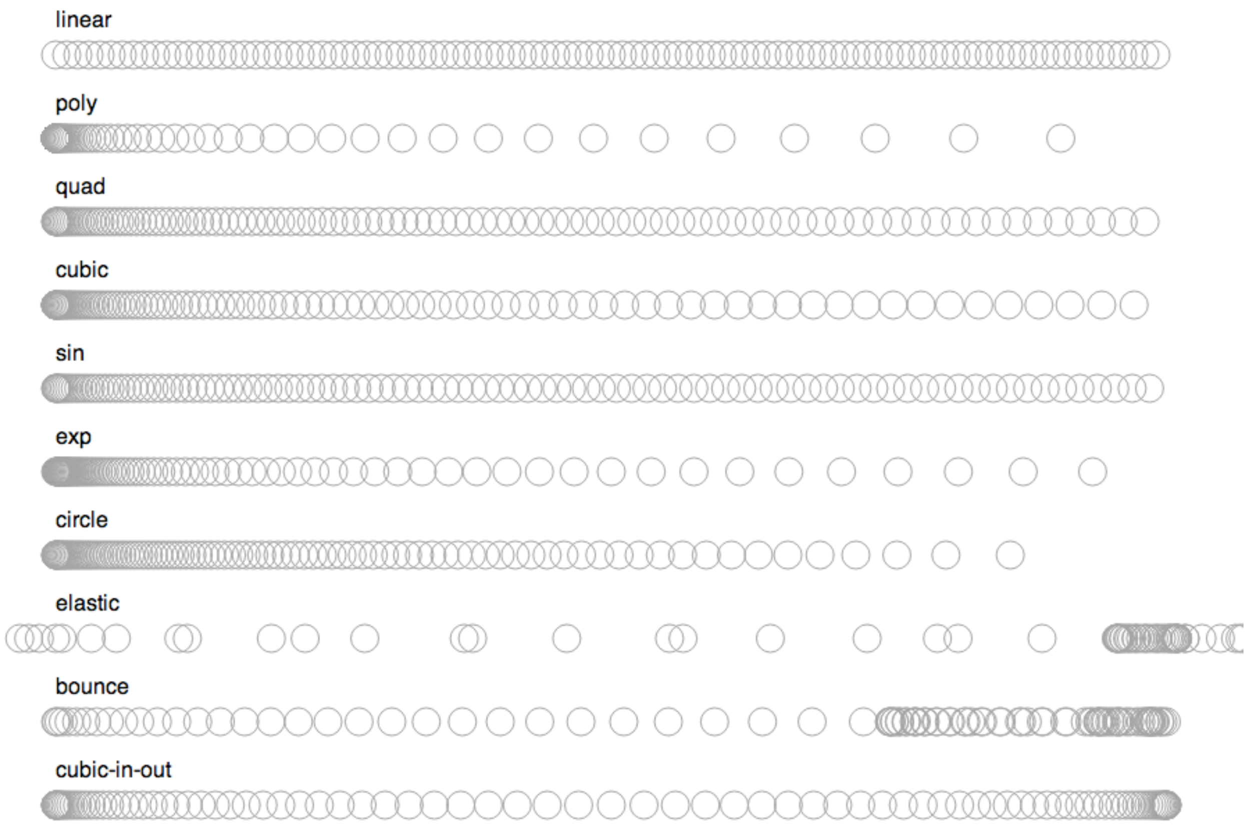
```
d3.select('body').append('svg')
  .append('circle').attr({r: 10, cx: 100, cy: 100})
  .transition().duration(4000).attr('cx', 300);
```

Another useful feature of transitions is the ability to specify an easing type. Essentially, easing let us fiddle with how the transition should accelerate. For example, if we drop a feather 5 feet, it will fall at about a constant rate. This would be a `"linear"` easing type. If, instead, we drop a hammer, it will accelerate down, going faster and faster (unless we're on the moon were there's no atmosphere to slow down the feather.) This would be an `"exp"` easing type. By default, D3 uses `"cubic-in-out"` which is how most physical objects move in the real world. Most objects don't instantaneously start moving at a constant speed. They first speed up, then, when they approach their target, they slow down, like a car speeding up after a stop sign and then slowing down as it approaches a red light at an intersection. This is the same transition illustrated in the above example. Here's an example of how we can specify a different easing type with a duration of 4 seconds.

```
d3.select('body').append('svg')
  .append('circle').attr({r: 10, cx: 100, cy: 100})
  .transition().duration(4000).ease(d3.ease('bounce')).attr('cx', 300);
```

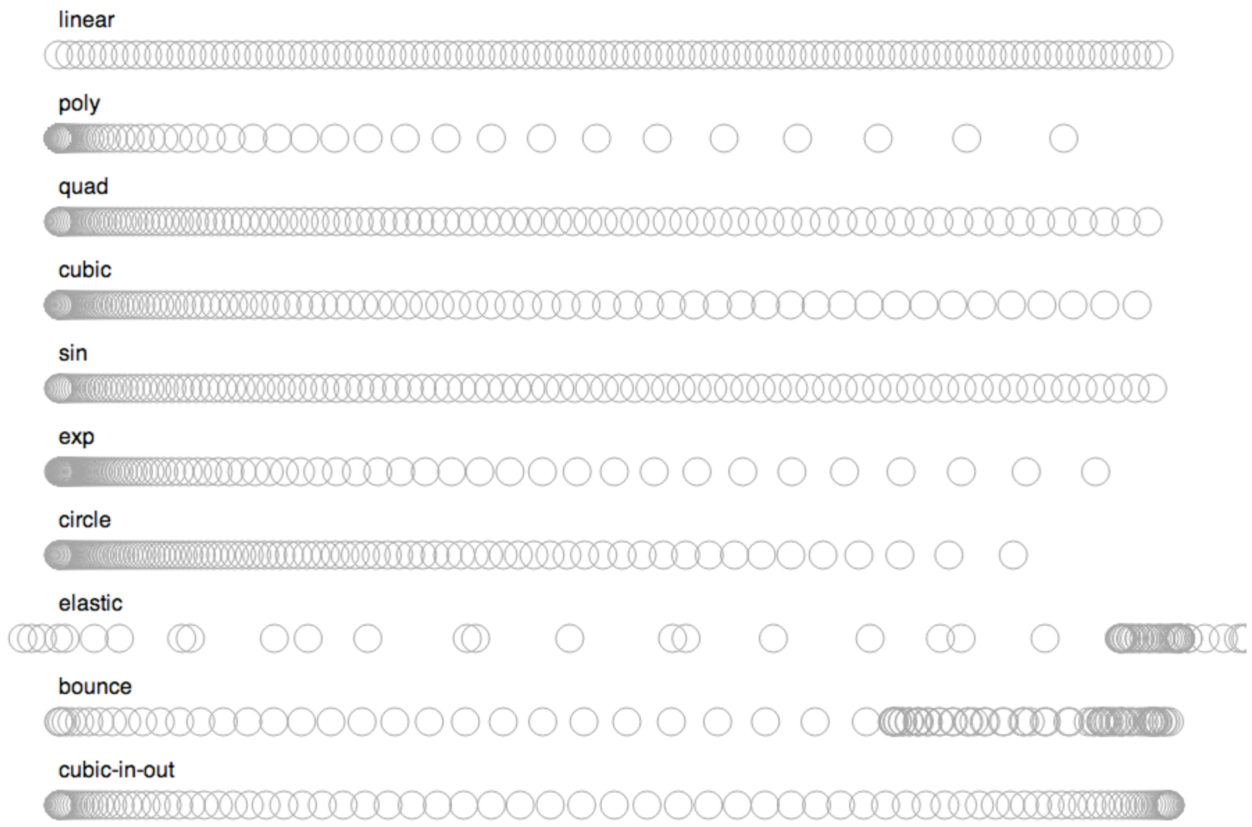
Here's a visualization[48] of other easing types.

[48]http://blog.vctr.me/experiments/transition-tweens.html



Easing types

Notice how all the transitions above gain speed accept for the last, `"cubic-in-out"`. `"cubic-in-out"` is actually the same as `"cubic"` except it has its easing applied equally to both ends (the `in` and `out` portions.) By default, all easing types are `in` so just `cubic` is the same as `cubic-in` and we can fiddle with the easing type by appending either `-out` or `-in-out`. Here's an example of the 3 different `poly` easing variations.



Transition easing variations

Events

Events allow our visualizations to be interactive. Using them, we specify what should happen when a user performs an action, like clicking, or moving the mouse, or pressing the keyboard. HTML on its own has a variety of different interactive events we can opt to be notified of using callbacks. D3 simply makes it easier to setup them up using selections instead of having to deal with individual DOM elements directly. We can do all this just using the `on()` selection method, passing it the name of the event we want to listen for and a callback function which should be called when that event occurs. Lets have a look at a simple 'hello world' style event example that compares the native HTML way of setting up an event listener and the D3 way.

```
<!DOCTYPE html>
<html>
  <body>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <select>
      <option value="hello">hello</option>
      <option value="world">world</option>
    </select>
    <script>
// // the non-D3 way
document.getElementsByTagName('select')[0]
  .addEventListener('change', function(){ alert(this.value); });
// the D3 way
d3.select('select').on('change', function(){ alert(this.value); });
    </script>
  </body>
</html>
```

Notice how the callback gets called with the `select` element as the `this` variable. Using `on` instead of `addEventListener` might sound like a trivial change but there's more going on here than just that. Different event listeners are being applied to all the elements in the selection. Just like most other selection methods, our callback will get called with the first two arguments set to the clicked elements `datum` and `index` in the selection, respectively. Event listener will only be setup once per element, event if we re-setup the listener. If we copied and pasted the non-D3 version, we'd get two `alerts`.

```
document.getElementsByTagName('select')[0]
  .addEventListener('change', function(){ alert(this.value); });
document.getElementsByTagName('select')[0]
  .addEventListener('change', function(){ alert(this.value); });
// the second event listener is added after the first
```

If we did the same with the D3 version, we'd get only one.

```
d3.select('select').on('change', function(){ alert(this.value); });
d3.select('select').on('change', function(){ alert(this.value); });
// the second event listener overrides the first
```

This is convenient since most of the time, we'll want to simply re-apply our event listeners for all selected elements instead of having to explicitly remove our old listeners. If, for some reason, we want to remove all the event listeners for a selection, we can call `on()` with `null` as the second argument.

```
d3.select('select').on('change', function(){ alert(this.value); });
d3.select('select').on('change', null);
// nothing is alerted!
```

Some other common events we'll be interested in are the mouse events `click`, `mousedown`, `mousemove`, and `mouseup`. A click events are fired when a user presses and then releases the mouse button. It's a combination of `mousedown` and `mouseup`. mouse move is fired everytime the mouse moves.

Putting together all our knowledge of transitions and events, lets build a simple video game. The objective is to avoid moving asteroids (`circles`) from colliding with our spaceship (mouse.) We'll start off by defining our current level, and the width and height of our game, which will take up the entire window.

```
var level = 0, width = window.innerWidth, height = window.innerHeight;
```

Now lets create the svg that will hold our game objects.

```
var svg = d3.select('body').append('svg');
```

Lets add some asteroids. `20` sounds like a good number. Lets make the pretty big, with a radius of `20`

```
var circles = svg.selectAll('circle').data(d3.range(20)).enter().append('circle')
  .attr('r', 50)
```

We'll randomly position all the asteroids across the screen at every level and transition all the asteroids as go between levels so lets create an update method we can call periodically using `setInterval`.

```
function update(){
  circles.transition().duration(2000).attr({
    cx: function(){ return Math.random() * width },
    cy: function(){ return Math.random() * height } });
  level++;
}
```

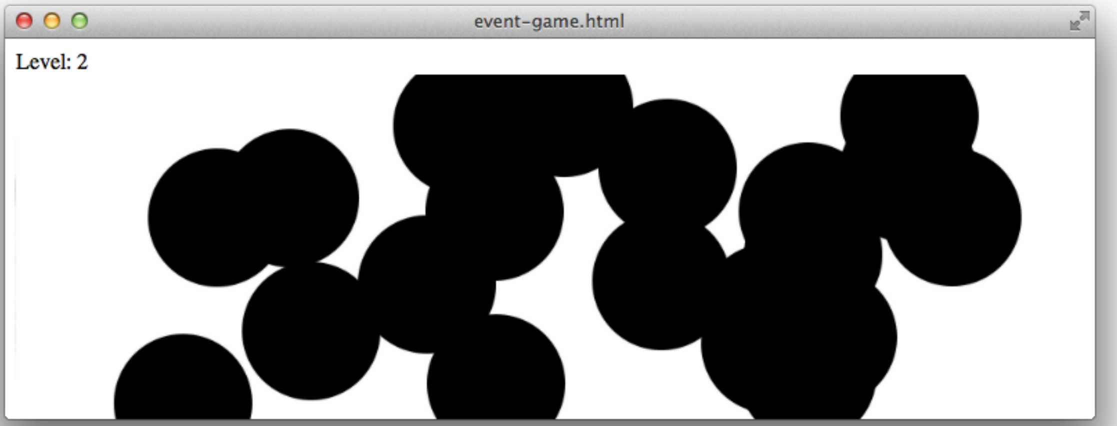
So there's something on the screen when the user first visits our game, lets call update() right off the bat. Then we'll setup our interval to call `update()` every 2 seconds.

```
update();
setInterval(update, 2000);
```

When an asteroid (circle) collides with our spaceship (mouse), game over! Alert the user that they lost and reset the level counter.

```
circles.on('mouseover', function(){
  alert('Nice! You made it to level ' + level);
  level = 0;
});
```

And we're done! We just made a video game in 15 lines of JavaScript. Here's what the final version looks like.



Asteroids, D3 style

```
<!DOCTYPE html>
<html>
  <body>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <script>
var level = 0, width = window.innerWidth, height = window.innerHeight;
var svg = d3.select('body').append('svg');
var circles = svg.selectAll('circle').data(d3.range(20)).enter()
  .append('circle').attr('r', 50)
  .on('mouseover', function(){
    alert('Nice! You made it to level ' + level);
    level = 0;
  });
function update(){
  circles.transition().duration(2000).attr({
    cx: function(){ return Math.random() * width },
    cy: function(){ return Math.random() * height } });
  level++;
}
update();
setInterval(update, 2000);
    </script>
  </body>
</html>
```

Data Parsing and Formatting

This chapter covers some of the common ways to get and clean data for use with D3.

Getting data into the browser

All the examples up until this point used hard-coded data values directly in our code. This is fine when working with very small, unchanging, data sets but quickly becomes unmanageable when the amount of data increases or our data changes frequently. We don't want to have to keep copying and pasting data into our code. This problem can be solved with a few D3 helper functions that allow us to load data in different formats from a separate files, instead of being in our code. Specifically, we can use `d3.json()` to load JSON[49] formatted data, or `d3.csv()` to load CSV[50] data.

All we need to give each is the name of the file to load, and a callback to be called once the file has been loaded.

Loading CSV data

episodes.csv

```
Title, Air date, U.S. viewers (in millions)
"Winter Is Coming", "April 17, - 2011", 2.22
"The Kingsroad", "April 24, - 2011",  2.20
"Lord Snow", "May 1, - 2011", 2.44
```

source: http://en.wikipedia.org/wiki/List_of_Game_of_Thrones_episodes[51]

```
d3.csv('data.csv', function(err, data){
  if(err) throw err;
  console.log(data[0].Title); // => "Winter is Coming"
  console.log(data[1].Title); // -> "The kingsroad"
  // etc...
});
```

Loading JSON data

[49] http://secretgeek.net/json_3mins.asp
[50] http://en.wikipedia.org/wiki/Comma-separated_values
[51] http://en.wikipedia.org/wiki/List_of_Game_of_Thrones_episodes

episodes.csv

```
[
  {
    "Title": "Winter Is Coming",
    "Air date": "April 27, - 2011",
    "U.S. viewers (in millions)": 2.22
  },{
    "Title": "The Kingsroad",
    "Air date": "April 24, - 2011",
    "U.S. viewers (in millions)": 2.20
  },{
    "Title": "Lord Snow",
    "Air date": "May 1, - 2011",
    "U.S. viewers (in millions)": 2.44
  }
]
```

```
d3.csv('data.json', function(err, data){
  if(err) throw err;
  console.log(data[0].Title); // => "Winter is Coming"
  console.log(data[1].Title); // -> "The kingsroad"
  // etc...
});
```

Common Gotcha: Loading data from local file

Most modern browsers impose security restrictions that prevent HTML files loaded from your computer from make requests for other local file resources. This means if you've just been double clicking on HTML files and opening them in your browser, those HTML pages wont be able to make requests for our locally stored data files. This is to prevent normal browser users from opening a downloaded HTML file by mistake and having it snoop other files off of their hard drive. To circumvent this problem, the simplest solution is to run a web server using `python -m SimpleHTTPServer` from the command line in the directory from which you want to run your server. Then we can access `our-file.html` in that directory from `localhost:8080/our-file.html` in the browser.

Common Gotcha: JSON is a subset of JavaScript

JSON is a subset of JavaScript. This means we could copy and paste JSON into a JavaScript program without problems, but this *does not* mean we can copy and paste anything from JavaScript into JSON. One key difference is that JSON object keys must be in double quotes. The same is true for string values. ie., { foo : 'bar' } is valid JavaScript but not valid JSON. { "foo" : "bar" } is valid JSON *and* valid JavaScript. Valid JSON cannot contain `function` or `Date` objects.

Working with data

Often the hardest part of data visualization is just getting the data we wan to visualize into a meaningful format. Notice how in the above example the dates are formatted as strings and contain a -? We'll have a hard time using those dates in scales or graphing them along a time line if we can't work with them as numbers. To convert them cover to numbers, we can use D3's time formatting helper method, `d3.time.parse()`. With it, we can create a new date format object to parse our funky dates.

```
var format = d3.time.format('%B %e, - %Y');
format.parse("April 27, - 2011"); // => a new date object
```

There's a ton of different date formatting tokens you can use. Checkout the full list on the D3 github wiki page[52].

To reformat our data set with so it contains actual date objects, we can call our formatter on each "Air date" property of every element in our data array. In this step, we'll also create a `viewers` property instead of `viewers (in millions)` since it's best practice to keep our data in the most common and/or obvious units by default.

```
var format = d3.time.format('%B %e, - %Y');
data.forEach(function(datum){
  datum['Air date'] = format.parse(datum['Air date']);
  // add another property, `viewers`
  datum['U.S. Viewers'] = datum['viewers (in millions)'] * 1000000;
  // remove the `U.S. viewers (in millions)` property
  delete datum['U.S. viewers (in millions)'];
});
```

[52]https://github.com/mbostock/d3/wiki/Time-Formatting#wiki-format

What next?

This chapter will cover additional resources and stepping off points for continued exploration of D3.

Layouts

D3 Offers an extensive collection of pre-existing layout helper[53] functions that make building the typical cookie-cutter graphs like pie, histogram, and stack chart ease. D3 does **not** provide methods like `createPieChart`. Instead, D3 offers functions that make creating them from scratch easier. Lets walk though a quick pie chart example to show this in action. We'll combine to two new methods, `d3.svg.arc()`, which creates a `arc` generator, and `d3.layout.pie()`. The `pie()` layout simply takes data and produces the necessary parameters that `arc()` expects.

```
var data = [4,5,6];
var pie = d3.layout.pie();
console.log(pie(data)); // =>
[
  { data:4, value:4, startAngle:4.61, endAngle:6.28 },
  { data:5, value:5, startAngle:2.51, endAngle:4.61 },
  { data:6, value:6, startAngle:0, endAngle:2.51 }
]
```

Now we can use an arc generator to take the start and end angles above and produce the necessary `d` attribute values to draw arc paths in SVG.

```
// great an arc generator function, setting the outer radius
// of the created arcs to `100` and inner radius to `0`
var arc = d3.svg.arc().outerRadius(100).innerRadius(0);
```

[53]https://github.com/mbostock/d3/wiki/Layouts

```
var color = d3.scale.category10();
var arcs = vis.selectAll('path').data(pie([4,5,6]))
  .enter().append('path')
    .attr({ d: arc }) // create the arc paths using the arc generator
    .style({
      // style each arc slice with a different color and white border
      fill: function(d, i){ return color(i) },
      stroke: 'white'
    });
```

Here's what it all looks like together. Note the containing `g` element and `(200,200)` translation so that the center of the pie chart isn't stuck in the top, left corner of the svg.

```
<!DOCTYPE html>
<html>
  <body>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <script>
      var vis = d3.select('body').append('svg').append('g')
          .attr('transform', 'translate(200,200)');
      var color = d3.scale.category10();
      var arc = d3.svg.arc().outerRadius(100).innerRadius(0);
      var pie = d3.layout.pie();
      var arcs = vis.selectAll('path').data(pie([4,5,6]))
        .enter().append('path').attr({
            d: arc,
            fill: function(d, i){ return color(i) },
            stroke: 'white'
        });
    </script>
  </body>
</html>
```

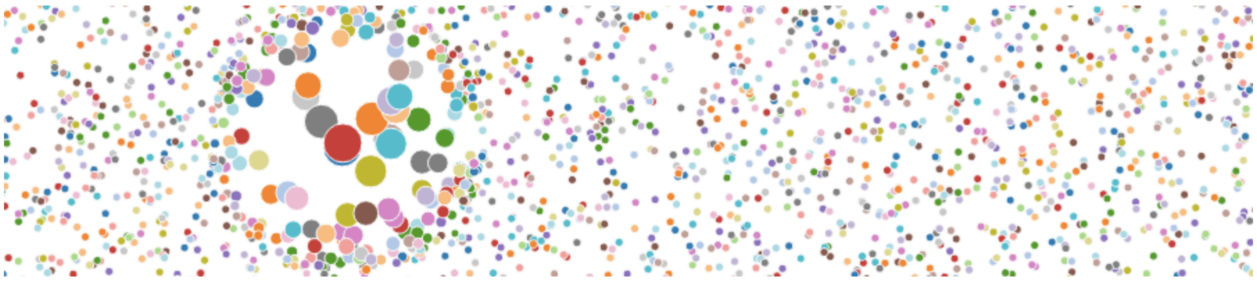
There are many more layouts like this one so if you ever find yourself creating a pretty common graph type, be sure to look for possible pre-existing layout helpers first.

Plugins and other tools

The venerable Mike Bostock and others have collected several common D3 plugins[54] that haven't seemed necessary enough to be put into D3 core but are, on occasion, extremely useful. One of the

[54]https://github.com/d3/d3-plugins

most interesting among them is the `d3.fisheye` plugin which applies a distorted fisheye lense effect which is useful when focusing in a particular portion of a larger collection of elements.


fisheye plugin

There are a few common tools that are often used with D3. Chief among them may be Crossfilter[55]. Think of Crossfilter as offering faster alternatives to D3's array helpers. With it, it's possible to quickly sort and filter extremely large amounts (millions of rows) of data.

[55]http://square.github.io/crossfilter/

Intro to Angular

Up until this point, we've focused only on D3 but in the next chapters, we'll look at some of the interesting ways in can be combined with Angular to make our visualizations more modular, reusable, and interactive. This also had a the benefit of making them easier to conceptualize in our heads as well as making it easier for us to come back to several months later after we've forgotten all the nuances of our visualization.

About Angular

Angular is a framework that makes it easier to create (web) applications but it lends it self well to interactive visualization. It does this by forcing us to break up our application into pieces that perform specific types of tasks. Splitting our code up this way allows components to be easily swapped out for other components. It also helps with connecting our components to other components allowing one visualization to be driven by each other or update themselves automatically when their underlying data changes.

Angular has a lot of features but most aren't directly related to data visualization. For a more in depth deep dive into Angular check out Ari's ng-book[56].

Angular for interactive and reusable data visualization

There are a lot of reason's to use Angular when doing data visualization but chief among them all directives. Directives make it easier to create reusable visualizations. They are essentially a way for us to create our own HTML elements that act just like the browsers built in HTML tags. Imagine we just created a pie chart visualization we'd like to reuse again. If we had created a directive that contained our pie chart, this would be simple. We can just copy and paste the `<pie-chart>` element in our html without having to write any JavaScript.

```
<pie-chart></pie-chart>
<pie-chart></pie-chart>
```

This also makes it easier for people that didn't write our pie chart, since they don't need to know the pie chart works to use it.

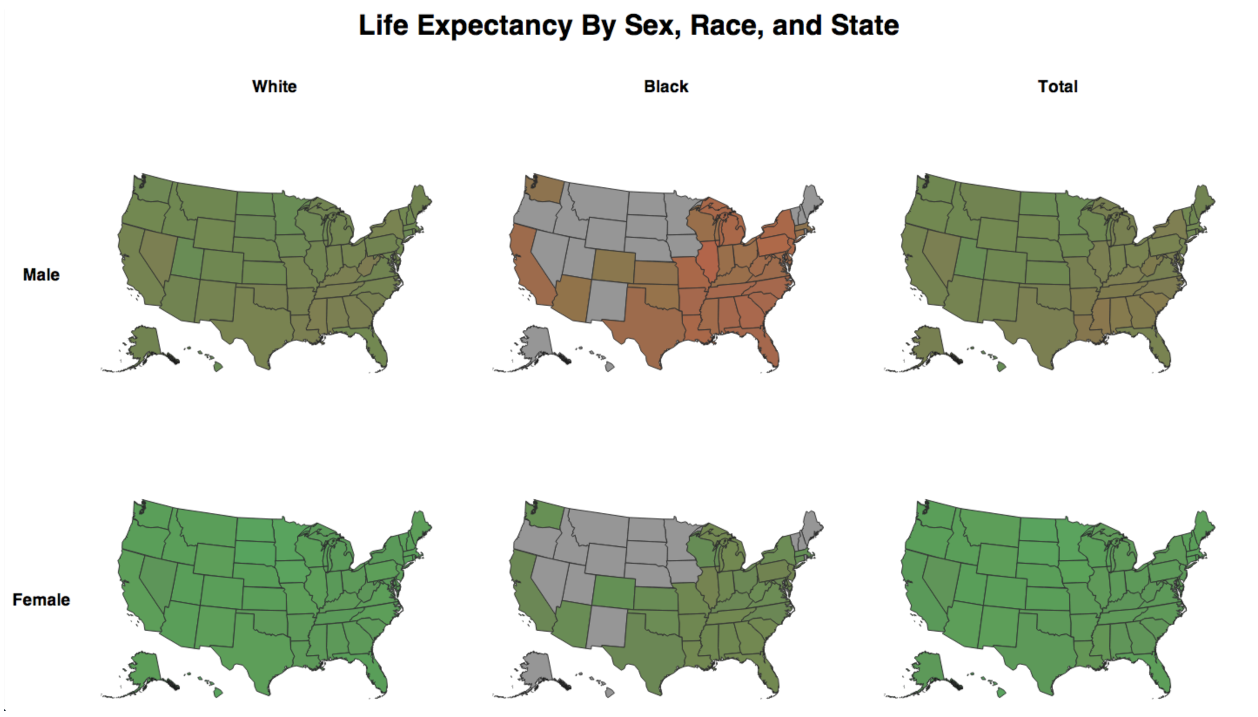
If we write our visualizations in directives, we also get a lot of other advantages for free. Namely, we can use Angular's other built in directives like `ng-switch` to conditionally show different visualizations. In this example we're conditionally showing either a pie chart or bar chart view of the data depending on the state of the `selection` scope variable.

[56]https://www.ng-book.com/

```
<div ng-switch on="selection">
  <div ng-switch-when="pie">
    <pie-chart></pie-chart>
  </div>
  <div ng-switch-default>
    <bar-chart></bar-chart>
  </div>
</div>
```

Using another built in Angular directive, `ng-repeat`, we can easily create the effective data visualization technique of small multiples allowing us to convey more information more effectively. It does this by allowing us to reuse the same visualization a variable number of times.

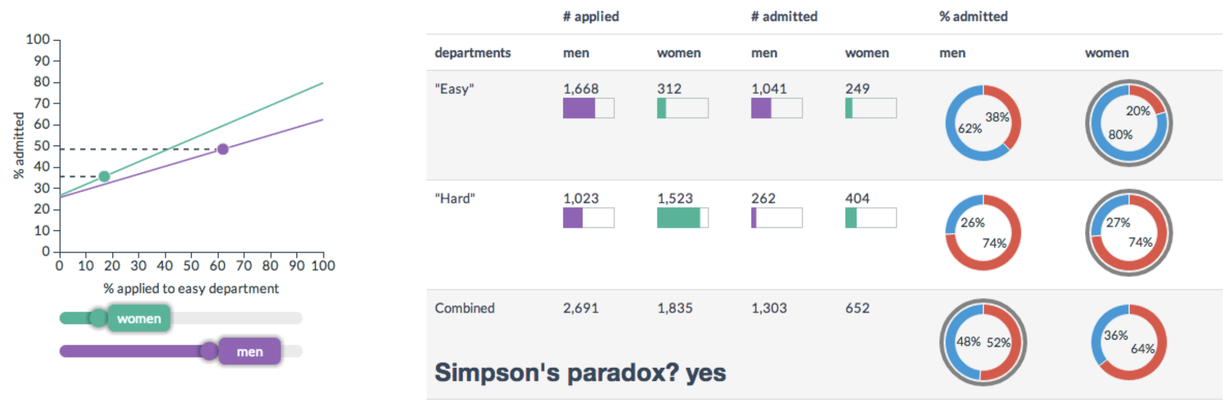
```
<div ng-repeat="races in incomes">
    <map ng-repeat="race in races"></map>
</div>
```



Live version: http://vicapow.github.io/angular-d3-talk/slides/demos/life-expectancy/index.html[57]

[57]http://vicapow.github.io/angular-d3-talk/slides/demos/life-expectancy/index.html

Lastly, Angular makes it easy to create dynamic visualizations by handling updates on the scope. This is often useful when creating visualizations that depend on the state of some other visualization or when, say, you wanted two different visualizations to update whenever a slider was moved.



	# applied		# admitted		% admitted	
departments	men	women	men	women	men	women
"Easy"	1,668	312	1,041	249	62% 38%	20% 80%
"Hard"	1,023	1,523	262	404	26% 74%	27% 74%
Combined	2,691	1,835	1,303	652	48% 52%	36% 64%

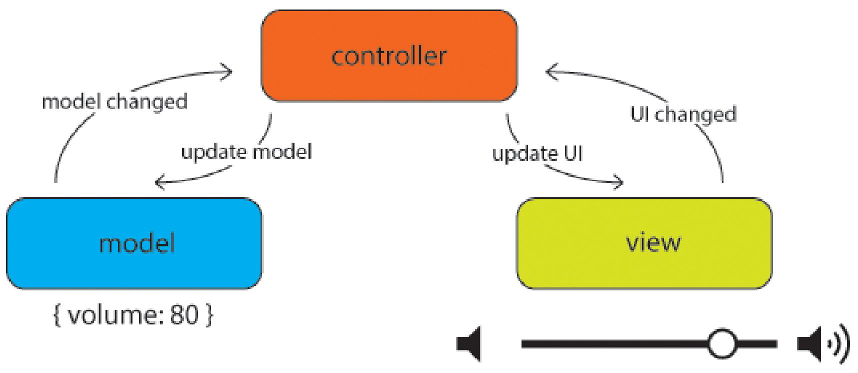
Simpson's paradox? yes

Live version: http://vudlab.com/simpsons/[58]

Model, View, Whatever

Most web frameworks encourage developers to break up their code into ether “models”, “views”, or “controllers” or (MVC). We can think of them in the following way:

Models: The information our application wants to show or allow to be modified. Views: The components responsible for showing or modifying that information or a “view” of the information. Controllers: the logic responsible for properly connecting the two.

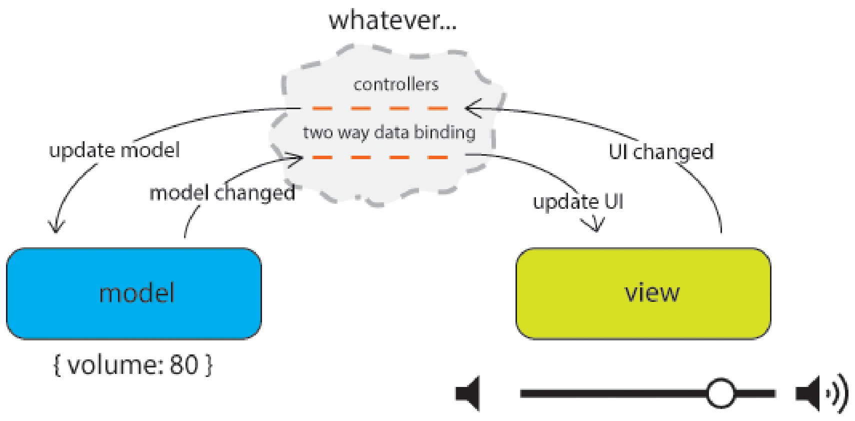


Remember to keep in mind that there’s a difference between this general, informal categorization of the parts of any application, and the types of components that actually exist in our application.

58 http://vudlab.com/simpsons/

Some frameworks might share some of these responsibilities across components or might call them something completely different. In Angular specifically, there is no formal “view” class or function. The “view” is just the DOM.

In Angular, for the most part, our viewers and models are automatically connected. This automatic (or “two-way”) data binding is the default and occurs on what’s called the “scope”. Just by changing a variable on the scope (aka, the model), it’s change will automatically be sent to the DOM. Because this isn’t strictly MVC, the Angular community refers to this pattern as “Model, View, Whatever” because you can loose a lot of time[59] contemplating exactly what pattern Angular falls into.



Hello Angular!

To get familiar with how to install and setup an Angular app, let’s walk through creating a ‘Hello World’ style application. This app will update a template whenever the value of an input tag changes.

Installing Angular

The easiest way to install Angular is to include the latest version from Google’s CDN by copy and pasting the below snippet into a new HTML file. (At the time of writing, the newest version is 1.2.10).

[59]https://plus.google.com/+AngularJS/posts/aZNVhj355G2

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.10/angular.m\
in.js"></script>
  </head>
  <body></body>
</html>
```

Next, we'll need to tell angular where our app begins using `ng-app`. Most of the time, this will just be the entire `<body>` tag.

```
<body ng-app>
  <!-- yo angular, everything in here is our app -->
</body>
```

Now we'll add an input tag that created a variable `person` on the scope. The snippet `ng-model="person"` is responsible doing this. It also handles checking for changes to the `<input>` tag and updating the scope, or vice versa. In MVC speak, this "wires up" the "model" `person` variable on scope to the `<input>` tag (our "view"). Don't worry too much *how* the scope works. Just know that it contains the information of our application. In this case, the "person" we'd like to say hello to. Angular will take care of wiring everything up for us automatically using two way data binding.

```
<body ng-app>
  <input type="text" ng-model="person"></input>
</body>
```

The `person` variable now updates whenever the value of the `<input>` tag changes but there's no way to prove it since we're not using that variable anywhere else. Let's fix that by using a simple Angular template.

```
<body ng-app>
  <input type="text" ng-model="person"></input>
  hello {{person}}!
</body>
```

Live version: http://jsbin.com/izimoQoL/1/edit[60]

Great! Now when we update the `<input>` text field, the change is automatically sent to the scope which then causes the template to get rendered. Here's the full version.

[60]http://jsbin.com/izimoQoL/1/edit

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.10/angular.min\
.js"></script>
  <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
</head>
<body ng-app>
  <input type="text" ng-model="person"></input>
  hello {{person}}!
</body>
</html>
```

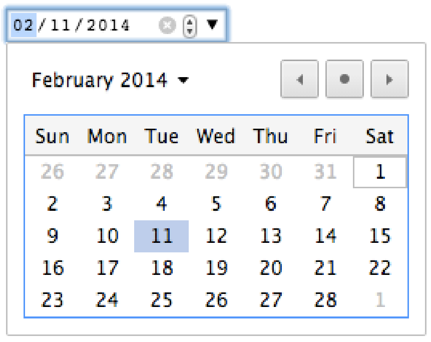
Directives for reusable visualizations

Understanding directives

Directives give us a way to encapsulate and reuse our visualizations by allowing us to essentially create our own HTML tags. In a way, you've already been using directives. The HTML5 `<input>` tags like `range` and `date` are essentially the browsers own directives. They're constructed from a combination of several UI element but expose a simple API. We just don't get access to their code.

```html
<input type="range" min="1" max="1000" value="0" style="width:100%">
```

```html
<input type="date" value="1989-01-23">
```



Creating a directive

Now that you understand what directives are, let's walk through creating our first with an example. Most directives do something useful but to keep it simple, we'll create a directive that just says "hello world!".

We'll start off with this HTML:

```html
<!DOCTYPE html>
<html>
  <head>
    <script src="angular.js"></script>
  </head>
  <body ng-app>
    <!-- our app! -->
  </body>
</html>
```

Next we need to make a new app module to hold all of our directives. All of are directives will live on this module. We only have one so it seems pointless but when we have many, this helps us group related directives.

```javascript
var myApp = angular.module('myApp', []);
```

And in our app we'll need to change `ng-app` to `ng-app="myApp"` to tell Angular we'd like to get fancy and use our new app module instead of the default.

```html
<body ng-app="myApp">
  <!-- our app! -->
</body>
```

Now we can actually create our directive. First, we give it a name. If we want to be able to use our directive in HTML and call it `<hello-world></hello-world>`, we need to name it "helloWorld". Angular will take care of converting that to the proper HTML tag name with `-` dashes instead of capital letters.

```javascript
myApp.directive('helloWorld', function(){
  // TODO: finish directive
});
```

Directives have a `link` function which is essentially a "constructor" if you're familiar with Object Oriented Programming. It contains everything that should happen every time the element appears in the HTML. Angular will call our `link` method with a few arguments. The first is the `scope` (or "model"). Next is the `element` the directive on. The third is an object hash of all the elements properties.

The `restrict: 'E'` option tells angular our directive should be used using the element name, (ie. `<hello-world></hello-world>`). We could also indicate something is a directive using its class name by using `'C'` instead of `'E'`. Then when we would use our directive using the `<div class="hello-world"></div>` style instead.

```
myApp.directive('helloWorld', function(){
  function link(scope, element, attr){
    // TODO: finish directive
  }
  return {
    link: link,
    restrict: 'E'
  }
});
```

Our directive doesn't do anything at the moment so lets change that by adding the text "hello world!" to its contents. Because the `link` method gets called for every new directive instance, our code for modifying the directive will go inside of the `link` method.

```
function link(scope, element, attr){
  element.text("hello world!");
}
```

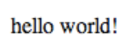
The directive is finished but we haven't used it yet in our HTML. Here we'll add it to the `<body>` tag.

```
<body ng-app="myApp">
  <hello-world></hello-world>
</body>
```

We're all done. Now when Angular loads our app, it will check our HTML for any directives and invoke all the `link` functions on them. The resulting `<body>` DOM looks like the following:

```
<body ng-app="myApp" class="ng-scope">
  <hello-world>hello world!</hello-world>
  <script>...</script>
</body>
```

Live version: http://jsbin.com/ABUmAqiR/1/edit[61]

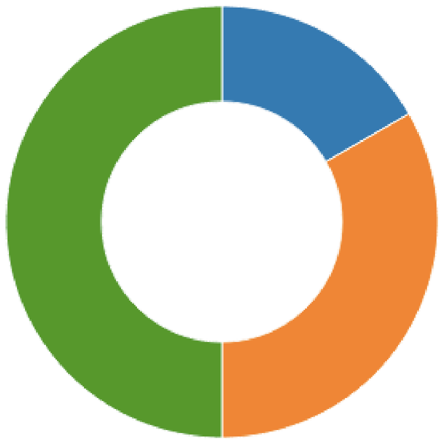


hello world!

[61]http://jsbin.com/ABUmAqiR/1/edit

A donut chart directive

Our `<hello-world>` directive was quiet but not very useful. Let's expand on it by creating a donut chart directive that we can reuse.



To get us started, we'll work from this existing D3 code that creates a donut chart.

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    <script src="angular.js"></script>
    <script src="d3.js"></script>
    <script>
    var color = d3.scale.category10();
    var data = [10, 20, 30];
    var width = 300;
    var height = 300;
    var min = Math.min(width, height);
    var svg = d3.select('body').append('svg');
    var pie = d3.layout.pie().sort(null);
    var arc = d3.svg.arc()
      .outerRadius(min / 2 * 0.9)
      .innerRadius(min / 2 * 0.5);

    svg.attr({width: width, height: height});
    var g = svg.append('g')
      // center the donut chart
      .attr('transform', 'translate(' + width / 2 + ',' + height / 2 + ')');

    // add the <path>s for each arc slice
    g.selectAll('path').data(pie(data))
      .enter().append('path')
        .style('stroke', 'white')
        .attr('d', arc)
        .attr('fill', function(d, i){ return color(i) });
    </script>
  </body>
</html>
```

Live version: http://jsbin.com/zinu/1/edit[62]

Just like in the "hello world!" example above, we'll need to setup our new directive on the app module.

```
myApp.directive('donutChart', function(){
  function link(scope, element, attr){
    // put D3 code here
  }
  return {
    link: link,
    restrict: 'E'
  }
});
```

The only difference is changing the directive name from 'helloWorld' to 'donutChart' so we'll be able to use our directive by the tag name `<donut-chart>`.

In the link function, we can copy and paste in all of the original pie chart code. The only change we'll need to make is updating the `d3.select('body')` selection to be relative to the directive using `d3.select(el[0])` instead of the entire DOM.

[62] http://jsbin.com/zinu/1/edit

```
// var svg = d3.select('body').append('svg'); // old version
var svg = d3.select(el[0]).append('svg'); // new version
```

The reason we have to use `el[0]` instead of just `el` is because `el` "is" a jQuery wrapped selection and not an ordinary DOM object. Doing `el[0]` gives us just the DOM element. (I say "is" in quotes because it's technically a `jqlite` wrapped DOM element. jqlite is essentially a heavily slimmed down version of jQuery.)

Live version: http://jsbin.com/niwe/1/edit[63]

It might not look different but we're well on our way to making it reusable. As a simple demo, we can easily create several donut charts just by copying and pasting the `<donut-chart>` tag.

```
<donut-chart></donut-chart>
<donut-chart></donut-chart>
<donut-chart></donut-chart>
```

Live version: http://jsbin.com/yili/1/edit[64]

Isolate scope

Our donut chart could really stand to be improved in a few ways. Specifically, it would be nice if it they didn't all show the same content. We'd prefer to be able to pass each its own data to display instead of hard coding the same data right in the link function of the directive. To do this, we can use the what's called an isolate scope. An isolate scope will allow us to pass our data into the directive using an attribute. (In our case, we'll just call it `data` but we could have called it whatever we like.)

```
<donut-chart data="[8, 3, 7]"></donut-chart>
<donut-chart data="[2, 5, 9]"></donut-chart>
<donut-chart data="[6, 2, 3]"></donut-chart>
```

To tell Angular our directive should have an isolate scope, we just add a `scope` property to the object returned from our directive declaration.

[63]http://jsbin.com/niwe/1/edit
[64]http://jsbin.com/yili/1/edit

```
return {
    link: link,
    restrict: 'E',
    scope: { data: '=' }
  }
```

Inside of our `link` method, we'll just reference `scope.data` and it will contain the data we passed to it in the `data` attribute in our template above.

```
function link(scope, element, attr){
  // our data for the current directive instance!
  var data = scope.data;
  // ...
}
```

Live version: http://jsbin.com/yili/6/edit[65]

If, you still wanted to have the same data shared between all the pie charts, we could now simply define a variable on the scope and pass it to all the individual donut charts.

```
<body ng-app="myApp" ng-init="ourSharedData=[8, 2, 9]">
  <donut-chart data="ourSharedData"></donut-chart>
  <donut-chart data="ourSharedData"></donut-chart>
  <donut-chart data="ourSharedData"></donut-chart>
```

Live vesion: http://jsbin.com/yili/7/edit[66]

Giving our donut chart an isolate scope also gives us the ability to use Angular's built in `ng-repeat` directive to create an arbitrary number of donut charts given an array of chart data arrays.

```
<body ng-app="myApp" ng-init="charts=[[8, 3, 7],[2, 5, 9],[6, 2, 3]]">
  <donut-chart data="chart" ng-repeat"chart in charts"></donut-chart>
```

Live version: http://jsbin.com/yili/8/edit[67]

[65]http://jsbin.com/yili/6/edit
[66]http://jsbin.com/yili/7/edit
[67]http://jsbin.com/yili/8/edit

Dynamic visualizations

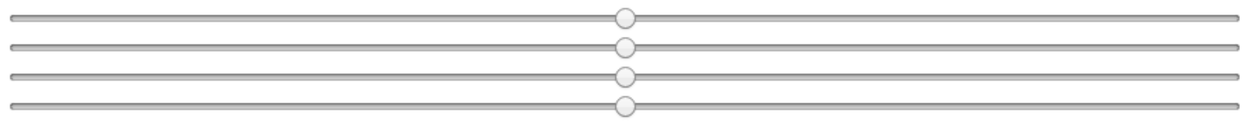
As we saw in the last chapter, directives are a great tool for making our visualizations self contained and reusable across different parts of our project or different projects entirely. Another great advantage to using Angular in combination with directives for data visualization is Angular's support for automatic two way data binding. That sounds like a mouth full but it's a actually a simple idea. It just means that Angular takes care of updating the DOM whenever data changes. Similarly, it automatically updates our data when the DOM changes. All we need to specify is the relationship between our data and DOM elements. The big advantage to all of this is that we'll get to write a lot less code but the disadvantage is that we'll have to spend the effort to wrap our heads around this new way of thinking about data binding and develop a strong understanding of what all Angular is doing automatically for us.

For the first half of this chapter we'll be working from simple toy examples that illustrate the fundamentals. In the second part, we'll walk through using these new concepts to make dynamic data visualizations.

Two way data binding

To illustrate and motivate the advantages of two way data binding, we'll start of with a very simple example. Here, several range `<input>` sliders are all bound to the same data model. When we drag one, all the others are updated automatically. This saves us the trouble of having to manually set up event listeners on all of the sliders as well as taking care of updating their values. It might help to think of Angular as having written all that code for us.

```
<body ng-app>
  <input type="range" ng-model="ourRangeValue">
  <input type="range" ng-model="ourRangeValue">
  <input type="range" ng-model="ourRangeValue">
  <input type="range" ng-model="ourRangeValue">
</body>
```



Live version: http://jsbin.com/tobuq/1/edit[68]

The scope

All of this automatic updating happens for variables on what's called the "scope". In the example above, the `ng-model` directive creates a variable on the scope called `ourRangeValue` and handles updating the value of the `<input>` range slider anytime `ourRangeValue` changes. The scope works a lot like the DOM tree. If we create a new scope on a DOM element, all of the child elements will have access to the variables on that new scope. When we told Angular the root of our app was the `<body>` tag it also created a default scope on the `<body>` tag that any inner elements can access. This is the scope the `ourRangeValue` is created on. This top level scope is called the "root scope".

There's a few ways new scopes get created but we can't simply just create a new scope on its own. The simplest way to create a new scope is to first create a "controller". Controller's are normally used to modify scope variables problematically in JavaScript. Whenever we create and use a new controller, a new scope is also created for it. Working from our previous example, lets see what it looks like to create and use a controller to create new scopes.

We first need to create a module to put our controller on.

```
var app = angular.module('myApp', []);
```

Next, we'll create a controller called 'HelloController' that just creates a variable on its new scope called `ourRangeValue`.

```
app.controller('HelloController', function($scope){
  $scope.ourRangeValue = 50;
});
```

To use it, we need to remember to use our module in the `<body>` tags `ng-app`.

```
<body ng-app="myApp">
```

Next, we'll use our new controller twice using `ng-controller="HelloController"` on a `<div>` tag. Now the scopes inside the `<div>` will inherit from the new controllers created for each `HelloController`.

[68] http://jsbin.com/tobuq/1/edit

```
<body ng-app="myApp">
  <div ng-controller="HelloController">
    <input type="range" ng-model="ourRangeValue">
    <input type="range" ng-model="ourRangeValue">
  </div>
  <div ng-controller="HelloController">
    <input type="range" ng-model="ourRangeValue">
    <input type="range" ng-model="ourRangeValue">
  </div>
</body>
```

Live version: http://jsbin.com/hite/1/edit[69]

Two scopes were created inside our apps root scope. One for the first `HelloController` and another one for the second. `ourRangeValue` will be a different property on each of these two inner scopes. This has the effect of binding the first two range slides to the value of `ourRangeValue` on the first scope and the second two range sliders to the value of `ourRangeValue` on the second scope.

Inner scopes inherit from their outer scopes. This means that if the `ourRangeValue` wasn't found on the `HelloController` scope, its outer scope would have been checked for a property named `ourRangeValue` and so on put until the root scope was checked.

Scope inheritance

`ng-init` is used to initialize properties on the scope of the current element. In the following example, a property called `foobar` on the root scope is given the value `50`. (The `HelloController` is unchanged from the previous examples.) We might expect that all of the range sliders would now be bound to the same `foobar` element in the outer scope. But they actually are not. Instead a new `foobar` property is created on each of the inner `HelloController` scopes.

[69]http://jsbin.com/hite/1/edit

```
<body ng-app="myApp" ng-init="foobar=50">
  <div ng-controller="HelloController">
    <input type="range" ng-model="foobar">
    <input type="range" ng-model="foobar">
  </div>
  <div ng-controller="HelloController">
    <input type="range" ng-model="foobar">
    <input type="range" ng-model="foobar">
  </div>
</body>
```

Live version: http://jsbin.com/vibon/1/edit[70]

This can be somewhat unexpected behavior but makes sense when considering that scope inheritance works the same as regular JavaScript inheritance. The the following snippet illustrates what's happening inside of Angular.

```
var rootScope = { foobar: 50 };
var scope1 = Object.create(rootScope); // scope1 inherits from rootScope
var scope2 = Object.create(rootScope); // scope2 inherits from rootScope
// scope1.foobar is 50
// scope2.foobar is 50
rootScope.foobar = 100; // changing `rootScope.foobar` works as expected
// scope1.foobar is 100
// scope2.foobar is 100
scope1.foobar = 2; // but assigning a value to `foobar` creates a new
// property on the child scope which shadows `rootScope.foobar`.
// rootScope.foobar is 100 instead of 2
```

If this is still somewhat confusing, a longer form review of JavaScript prototype chaining and inheritance may be helpful. We recommend this blog post by Sebastian Porto[71]

One way to fix this is to use another object as a wrapper to prevent shadowing properties of the parent scope. In this way, we're never assigning a new value to the `foobar` property.

[70]http://jsbin.com/vibon/1/edit

[71]http://sporto.github.io/blog/2013/02/22/a-plain-english-guide-to-javascript-prototypes/

```
var rootScope = { foobar: { value: 50 } };
var scope1 = Object.create(rootScope); // scope1 inherits from rootScope
var scope2 = Object.create(rootScope); // scope2 inherits from rootScope
// scope1.foobar.value is 50
// scope2.foobar.value is 50
rootScope.foobar.value = 100;
// scope1.foobar.value is 100
// scope2.foobar.value is 100
scope1.foobar.value = 2;
// rootScope.foobar.value is 2
```

Using this knowledge of scope inheritance, let's update our the previous example.

```
<body ng-app="myApp" ng-init="foobar = { value: 50 }">
  <div ng-controller="HelloController">
    <input type="range" ng-model="foobar.value">
    <input type="range" ng-model="foobar.value">
  </div>
  <div ng-controller="HelloController">
    <input type="range" ng-model="foobar.value">
    <input type="range" ng-model="foobar.value">
  </div>
</body>
```

Live version: http://jsbin.com/vibon/1/edit[72]

Now whenever an `<input>` range slider is adjusted, the same root scope `foobar.value` property is updated and shared across all of the inner controller scopes.

Making visualizations dynamic with `$watch`

`scope.$watch`

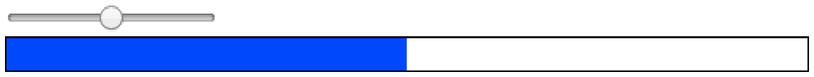
Using Angular's built in directives makes it easy to do simple things like binding the changes of a range slider to another but for complicated visualizations, we'll need to create our own directives that watch for changes on the scope and update their contents in response to these changes. Let's walk through a simple example of a visualization directive that updates its contents in response to changes

[72]http://jsbin.com/vibon/1/edit

to its data. Specifically we'll create a `<progress-bar>` directive. It's `progress` scope property will be a number that represents its progression from 0 to 100 percent. Whenever this `progress` property changes, we'll need to update the progress bar `<rect>`.

To allow for this type of task, scope objects have a function called `$watch()` that we'll pass two arguments. The first argument is the scope variable we'd like to watch, the second is the function that should be called whenever a change is detected on that property. For our `<progress-bar>` directive this might look the following:

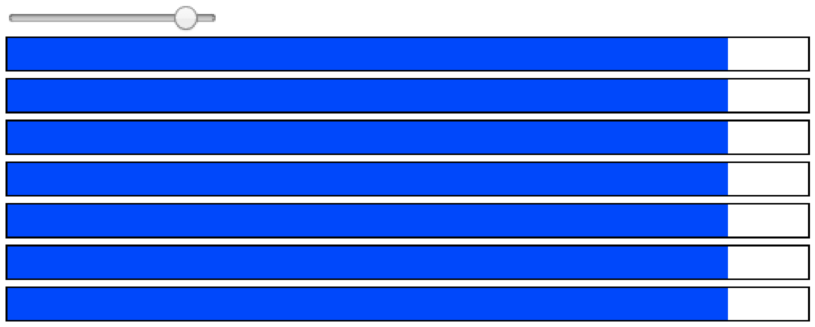
```
scope.$watch('progress', function(progress){
  rect.attr({x: 0, y: 0, width: width * progress / 100, height: height });
});
```



Live version: http://jsbin.com/vagiq/3/edit[73]

Everytime the slider is dragged, the `ng-model` direct on the range slider updates the scope variable `progress`. Then, the `scope.$watch('progress', ...)` callback is called in the progres-bar directive.

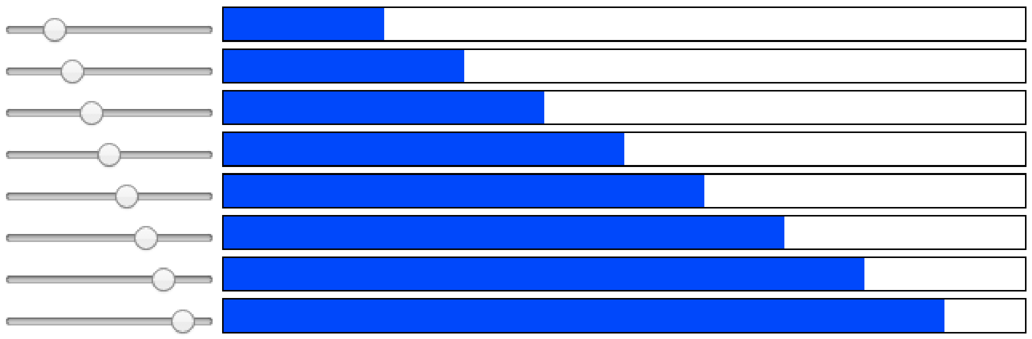
This might not feel very "automatic". We still have to be manually watch for changes to the scope when we're *building* our own directives. The advantage is that our directive can now be placed into other places and all of this "wiring up" with happen automatically. For example, we could copy and paste our directive a few times and watch as changes to the range slider updates all of the progress bars.



73 http://jsbin.com/vagiq/3/edit

Live version: http://jsbin.com/hadi/1/edit[74]

Or we could create an array of progresses and use an `ng-repeat` to create several slider and progress bars for each.



Live version: http://jsbin.com/manon/1/edit[75]

Let’s make a dynamic donut chart

Now that we understand how we can make directives that update dynamically, let’s work through a more complicated example of a dynamically updating visualization. We’ll take the donut chart directive from the previous chapter and make it dynamic so that we can change its data and the visualization will update accordingly.

Live version: http://jsbin.com/yili/6/edit[76]

To start off, we first need to be able to modify the data the chart is displaying. Three `<input>` range sliders will do the trick but we could have used any method to update the data. The directive doesn’t know or care how its data gets modified.

[74]http://jsbin.com/hadi/1/edit
[75]http://jsbin.com/manon/1/edit
[76]http://jsbin.com/yili/6/edit

```
<body ng-app="myApp" ng-init="chart=[10, 20, 30]">
  <input type="range" ng-model="chart[0]">
  <br>
  <input type="range" ng-model="chart[1]">
  <br>
  <input type="range" ng-model="chart[2]">
  <br>
  <donut-chart data="chart"></donut-chart>
</body>
```

Live version: http://jsbin.com/waqep/1/edit[77]

The directive doesn't update yet because it's not watching for changes to its `data` scope variable. To fix that, we'll just use the `$watch` function like before.

```
scope.$watch('data', function(data){
  console.log("an element within `data` changed!");
  console.log(data);
}, true);
```

Live version: http://jsbin.com/waqep/3/edit[78]

One important difference is that we're now giving `$watch` an extra third variable `true`. This tells `$watch` to also watch for changes within `data` itself. Otherwise, changing the first element of data by doing `data[0]=45` would not trigger the `$watch` callback because `data` is still the same array after the change. It just has a different value for its first element.

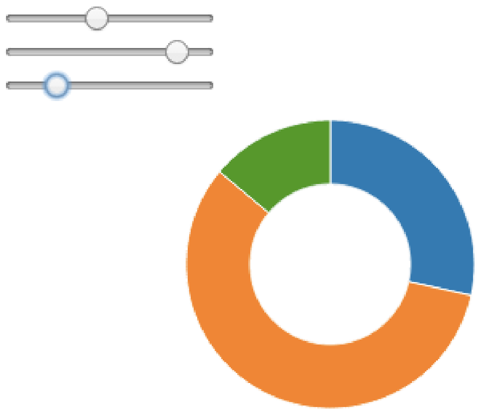
`$watch` callbacks will not be fired for changes to the elements of arrays unless you pass `true` as the third argument. Otherwise, the `$watch` callback will only fire if the array being watched becomes an entirely new array.

The last step is to update the arc `<path>` tags when the data changes since they are the only part of our visualization that depend on the data. Since we'll be updating the arcs in the `$watch` we need to give the `<path>` arc selector a name to refer to later. We also don't need to set the arcs `d` property anywhere expect when the data changes in the `$watch`.

[77]http://jsbin.com/waqep/1/edit
[78]http://jsbin.com/waqep/3/edit

```
var arcs = g.selectAll('path').data(pie(data))
  .enter().append('path')
    .style('stroke', 'white')
    .attr('fill', function(d, i){ return color(i) });

scope.$watch('data', function(){
  arcs.data(pie(data)).attr('d', arc);
}, true);
```

Live version: http://jsbin.com/jiqoz/1/edit[79]



Our Donut chart is lookin' good! But it always has to have the same number of slices as it started with. It would be better if we could instead add or remove elements from the `chart` array and have the donut chart appropriately add or remove slices. We can do this by having our `$watch` be responsible for adding or removing the necessary arc `<path>` tags and combination with d3's `.enter()` `.exit()` selections. If some of this syntax is still a bit unclear, refer back to chapter 2 for a refresher on how selections work in D3.

[79]http://jsbin.com/jiqoz/1/edit

```
var arcs = g.selectAll('path');
scope.$watch('data', function(data){
  arcs = arcs.data(pie(data));
  arcs.exit().remove(); // remove path tags that no longer have data
  arcs.enter().append('path') // or add path tags if there's more data
    .style('stroke', 'white')
    .attr('fill', function(d, i){ return color(i) });
  arcs.attr('d', arc); // update all the `<path>` tags
}, true);
```

Next, we need somewhat to increase or decrease the elements in `chart`. For this, we can use two buttons, one for adding elements to `chart`, another for removing elements.

```
<body ng-app="myApp" ng-init="chart=[10, 20, 30]">
  <button ng-click="chart.push(50)">add slice</button>
  <button ng-click="chart.pop()">remove slice</button>
  <!-- ... -->
</body>
```

We also need some way to add the `<input>` range sliders for each slice (or element) in the `chart` array. For this, we can use an `ng-repeat`.

```
<input type="range" ng-model="slice" ng-repeat="slice in chart">
```

However, when you run this code, you'll notice a problem. Angular throws an error:

> Error: [ngRepeat:dupes] Duplicates in a repeater are not allowed. Use 'track by' expression to specify unique keys. Repeater: slice in chart, Duplicate key: number:20

Live version: http://jsbin.com/leco/1/edit[80]

What Angular is telling us is that every element in an `ng-repeat` needs to be unique. However, we can tell Angular to use the elements index within the array instead to determine uniqueness by adding `track by $index` after `slice in chart`.

```
<input type="range"
  ng-model="slice" ng-repeat="slice in chart track by $index">
```

[80]http://jsbin.com/leco/1/edit

Adding this stops any errors from being through and adding or removing slices seems to work but dragging a slider has no effect on the visualization. The `chart` array is not being modified by the slider values. This is another reincarnation of scope inheritance we mentioned earlier in this chapter. `ng-repeat` creates a new scope for each `<input>` tag. And when a range value gets updated, its updating its `slice` property which isn't shared with its parent.

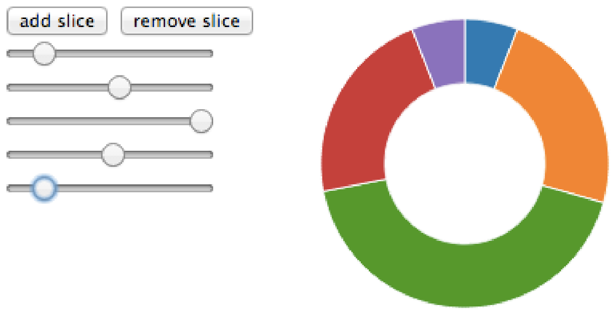
Live version: http://jsbin.com/wadev/1/edit[81]

We can fix it the same way as before by using a wrapper object. Doing so, we can safely modify its properties without having to worry about shadowing properties from the parent scope.

```
<body ng-app="myApp" ng-init="chart=[{value: 10}, {value: 20}, {value: 30}]">
  <donut-chart data="chart" style="float:right"></donut-chart>
  <button ng-click="chart.push({value: 10})">add slice</button>
  <button ng-click="chart.pop()">remove slice</button>
  <input type="range" ng-model="slice.value"
    ng-repeat="slice in chart track by $index">
```

Our directive now has to be updated so that it can access the slice value of each object in the `data` array. Because the `pie` layout is the only thing that uses `data` directly, we can give `pie` a `value` accessor and it will take care of plucking the pie chart value out of each object in the `data` array.

```
pie.value(function(d){ return d.value; });
```



Live version: http://jsbin.com/momiq/1/edit[82]

[81]http://jsbin.com/wadev/1/edit
[82]http://jsbin.com/momiq/1/edit