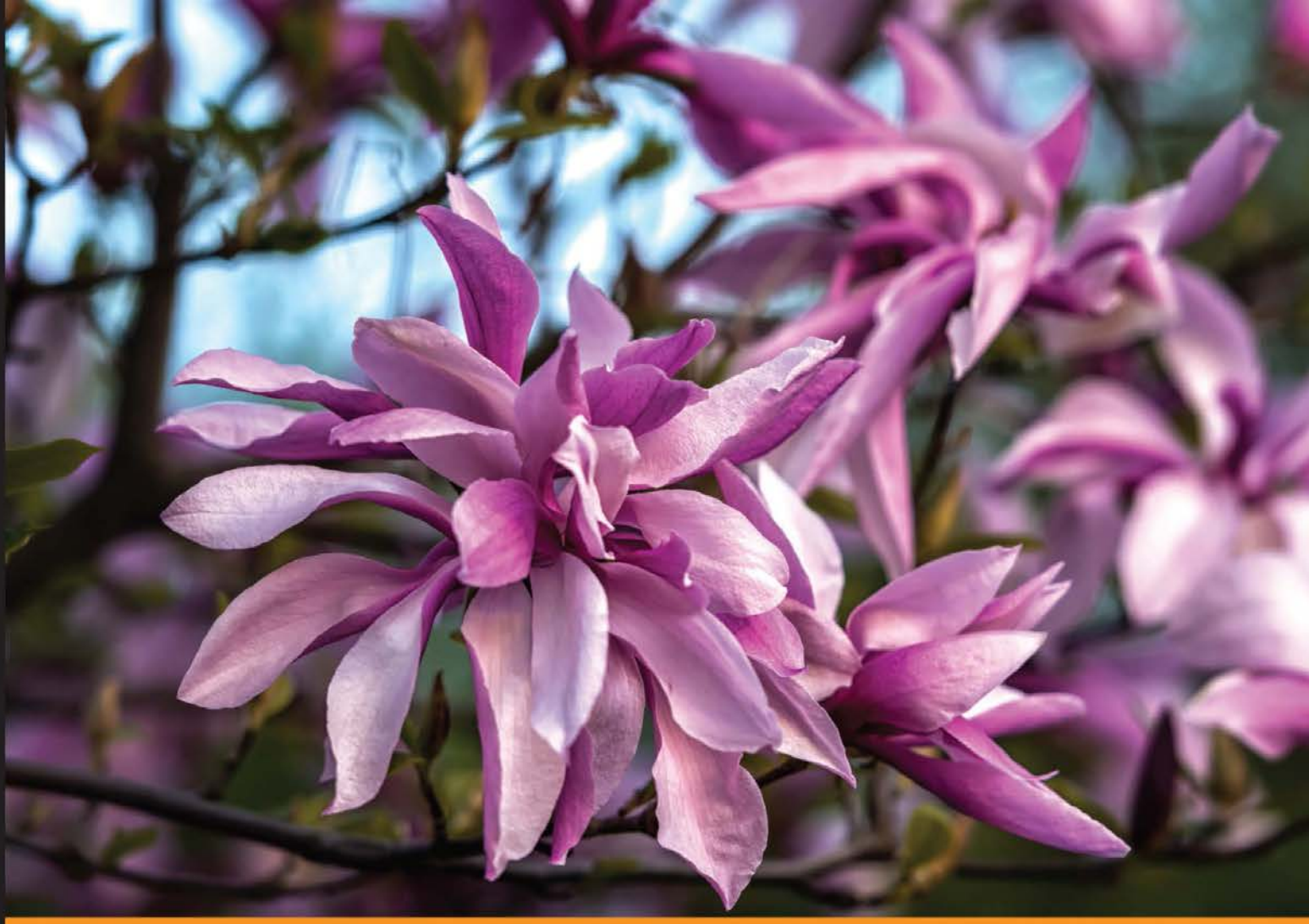


Community Experience Distilled

Learning d3.js Data Visualization

Second Edition

Inject new life into your data by creating compelling visualizations with d3.js

Ændrew Rininsland
Swizec Teller

[PACKT] PUBLISHING open source* community experience distilled

Learning d3.js Data Visualization

Second Edition

Inject new life into your data by creating compelling visualizations with d3.js

Ændrew Rininsland

Swizec Teller

[PACKT] PUBLISHING

open source
community experience distilled

BIRMINGHAM - MUMBAI

Learning d3.js Data Visualization
Second Edition

First published: October 2013

Second edition: April 2016

Production reference: 1220416

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-904-2

www.packtpub.com

Credits

Authors
Ændrew Rininsland
Swizec Teller

Reviewer
Elliot Bentley

Commissioning Editor
Wilson D'souza

Acquisition Editor
Smeet Thakkar

Content Development Editor
Onkar Wani

Technical Editor
Tanmayee Patil

Copy Editor
Vikrant Phadke

Project Coordinator
Bijal Patel

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Production Coordinator
Nilesh Mohite

Cover Work
Nilesh Mohite

About the Authors

Ændrew Rininsland is a developer and journalist who has spent much of the last half-decade building interactive content for newspapers such as *The Times*, *Sunday Times*, *The Economist*, and *The Guardian*. During his 3 years at *The Times* and *Sunday Times*, he worked on all kinds of editorial projects, ranging from obituaries of figures such as Nelson Mandela to high-profile data-driven investigations such as *The Doping Scandal*, the largest leak of sporting blood test data in history.

A prolific open source developer, Ændrew has released many kinds of projects, ranging from small utility libraries such as Doctop (which allow the creation of live-updating datasets using Google Docs) to big charting tools such as AxisJS. He is also a co-maintainer of C3.js, a widely used abstraction of D3 that greatly speeds up the creation of several basic chart types, and GitHub.js, a library that makes it easy to interact with the GitHub API.

You can follow him on Twitter at `@aendrew` and on GitHub at `github.com/aendrew`.

I'd like to thank my family for their continued encouragement, and also my partner Siyuan for being incredibly supportive while I worked on this, my first book.

I'd also like to thank (in no particular order) `@marvin.richter` on the Polymer Slack, snover on `##typescript`, `@arrayjam`, `@sebastian`, `@seemantk`, everyone else on the D3.js Slack, and especially Elliot Bentley for his fantastic technical reviewing.

Swizec Teller author of *Data Visualization with d3.js*, is a geek with a hat. Founding his first startup at 21, he is now looking for the next big idea as a full-stack web generalist focusing on freelancing for early-stage startup companies.

When he isn't coding, he's usually blogging, writing books, or giving talks at various non-conference events in Slovenia and nearby countries. He is still looking for a chance to speak at a big international conference.

In November 2012, he started writing *Why Programmers Work At Night*, and set out on a quest to improve the lives of developers everywhere.

I want to thank `@gandalfar` and `@robertbasic` for egging me on while writing and being my guinea pigs for the examples. I also want to send love to everyone at `@psywerx` for keeping me sane and creating one of the best datasets ever.

About the Reviewer

Elliot Bentley is a graphics editor at the *Wall Street Journal*, where he works on interactive graphics and newsroom tools. He runs Journocoders, a meetup group in London for people who like journalism and code. In his free time, he works on oTranscribe, an open source web app for transcribing interviews.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface

Welcome to *Learning d3.js Data Visualization, Second Edition*. Over the course of this book, you'll learn the basics of one of the world's most ubiquitous and powerful data visualization libraries, but we won't stop there. By the end of our time together, you'll have all the skills you need to become a total D3 ninja, able to do everything from building visualizations from scratch to using it on the server and writing automated tests. As well, if you haven't leveled up your JavaScript skills for a while, you're in for a treat—this book endeavors to use the latest features that are currently being added to the language, all this while explaining why they're cool and how they differ from old-school JavaScript.

What this book covers

Chapter 1, *Getting Started with D3, ES2016, and Node.js*, updated and expanded in the second edition, gets you up and running with the latest tools for building data visualizations using D3.

Chapter 2, *A Primer on DOM, SVG, and CSS*, reviews the underlying web technologies that D3 can manipulate.

Chapter 3, *Making Data Useful*, teaches you how to transform data so that D3 can visualize it.

Chapter 4, *Defining the User Experience – Animation and Interaction*, updated and expanded in the second edition, is where you use animation and user interactivity to drive your data visualizations.

Chapter 5, *Layouts — D3's Black Magic*, updated and expanded in the second edition, teaches you how layouts can take your D3 skills to the next level by providing reusable patterns for creating complex charts.

Chapter 6, *D3 on the Server with Node.js*, new in the second edition, helps you build and deploy a Node.js-based web service that renders D3 using Canvas.

Chapter 7, *Designing Good Data Visualizations*, updated and expanded in the second edition, compares and contrasts differing approaches to data visualization while building a set of best practices.

Chapter 8, *Having Confidence in Your Visualizations*, new in the second edition, shows you how to improve the quality of your code by introducing linting, static type checking, and unit testing to your projects.

What you need for this book

You will need a machine that is capable of running Node.js. We will discuss how to install this in the first chapter. You can run it on pretty much anything, but having a few extra gigabytes of RAM available will probably help while developing. Some of the mapping examples later in the book are kind of CPU-intensive, though most machines produced since 2014 should be able to handle them.

You will also need the latest version of your favorite web browser; mine is Chrome, and I use it in the examples, but Firefox also works well. You can try to work in Safari, Internet Explorer/Edge, Opera, or any other browser, but I feel that Chrome's Developer Tools are the best.

Who this book is for

This book is for web developers, interactive news developers, data scientists, and anyone interested in representing data through interactive visualizations on the Web with D3. Some basic knowledge of JavaScript is expected, but no prior experience with data visualization or D3 is required to follow this book.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other code using the `require` directive."

A block of code is set as follows:

```
d3.selectAll('.bars')
  .data(dataset)
  .enter()
  .append('rect')
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
d3.selectAll('.bars')
  .data(dataset)
  .enter()
  .append('rect')
```

Any command-line input or output is written as follows:

```
$ npm install d3 --save
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The **Resources** tab is good for inspecting client-side data."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

All the code for the book is hosted on GitHub. We talk about how to clone the repo and switch between branches (which are used to separate the code into chapters) in *Chapter 1*, *Getting Started with D3, ES2016, and Node.js* but you can take a look ahead of time by visiting `https://github.com/aendrew/learning-d3`.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/Learningd3jsDataVisualizationSecond_Edition_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started with D3, ES2016, and Node.js

In this chapter, we'll lay the foundations of what you'll need to run all the examples in the book. I'll explain how you can start writing **ECMAScript 2016** (**ES2016**) today—which is the latest and most advanced version of JavaScript—and show you how to use Babel to transpile it to ES5, allowing your modern JavaScript to be run on any browser. We'll then cover the basics of using D3 to render a basic chart.

What is D3.js?

D3 (**Data-Driven Documents**), developed by Mike Bostock and the D3 community since 2011, is the successor to Bostock's earlier Protovis library. It allows pixel-perfect rendering of data by abstracting the calculation of things such as scales and axes into an easy-to-use **domain-specific language** (**DSL**), and uses idioms that should be immediately familiar to anyone with experience of using the massively popular jQuery JavaScript library. Much like jQuery, in D3, you operate on elements by selecting them and then manipulating via a chain of modifier functions. Especially within the context of data visualization, this declarative approach makes using it easier and more enjoyable than a lot of other tools out there. The official website, `https://d3js.org/`, features many great examples that show off the power of D3, but understanding them is tricky at best. After finishing this book, you should be able to understand D3 well enough to figure out the examples. If you want to follow the development of D3 more closely, check out the source code hosted on GitHub at `https://github.com/mbostock/d3`.

The fine-grained control and its elegance make D3 one of the most powerful open source visualization libraries out there. This also means that it's not very suitable for simple jobs such as drawing a line chart or two—in that case you might want to use a library designed for charting. Many use D3 internally anyway. For a massive list, visit `https://github.com/sorrycc/awesome-javascript#data-visualization`.

As a data manipulation library, D3 is based on the principles of functional programming, which is probably where a lot of confusion stems from. Unfortunately, functional programming goes beyond the scope of this book, but I'll explain all the relevant bits to make sure that everyone's on the same page.

What's ES2016?

One of the main changes in this edition is the emphasis on ES2016, the most modern version of JavaScript currently available. Formerly known as ES6 (Harmony), it pushes the JavaScript language's features forward significantly, allowing for new usage patterns that simplify code readability and increase expressiveness. If you've written JavaScript before and the examples in this chapter look pretty confusing, it means you're probably familiar with the older, more common ES5 syntax. But don't sweat! It really doesn't take too long to get the hang of the new syntax, and I will try to explain the new language features as we encounter them. Although it might seem a somewhat steep learning curve at the start, by the end, you'll have improved your ability to write code quite substantially and will be on the cutting edge of contemporary JavaScript development.

For a really good rundown of all the new toys you have with ES2016, check out this nice guide by the folks at Babel.js, which we will use extensively throughout this book:
`https://babeljs.io/docs/learn-es2015/`.

Before I go any further, let me clear some confusion about what ES2016 actually is. Initially, the ECMAScript (or ES for short) standards were incremented by cardinal numbers, for instance, ES4, ES5, ES6, and ES7. However, with ES6, they changed this so that a new standard is released every year in order to keep pace with modern development trends, and thus we refer to the year (2016) now. The big release was ES2015, which more or less maps to ES6. ES2016 is scheduled for ratification in June 2016, and builds on the previous year's standard, while adding a few fixes and two new features. You don't really need to worry about compatibility because we use Babel.js to transpile everything down to ES5 anyway, so it runs the same in Node.js and in the browser. For the sake of simplicity, I will use the word "ES2016" throughout in a general sense to refer to all modern JavaScript, but I'm not referring to the ECMAScript 2016 specification itself.

Getting started with Node and Git on the command line

I will try not to be too opinionated in this book about which editor or operating system you should use to work through it (though I am using Atom on Mac OS X), but you are going to need a few prerequisites to start.

The first is Node.js. Node is widely used for web development nowadays, and it's actually just JavaScript that can be run on the command line. Later on in this book, I'll show you how to write a server application in Node, but for now, let's just concentrate on getting it and npm (the brilliant and amazing package manager that Node uses) installed.

If you're on Windows or Mac OS X without Homebrew, use the installer at `https://nodejs.org/en/`. If you're on Mac OS X and are using Homebrew, I would recommend installing "n" instead, which allows you to easily switch between versions of Node:

```
$ brew install n
$ n latest
```

Regardless of how you do it, once you finish, verify by running the following lines:

```
$ node --version
$ npm --version
```

If it displays the versions of node and npm (I'm using 5.6.0 and 3.6.0, respectively), it means you're good to go. If it says something similar to `Command not found`, double-check whether you've installed everything correctly, and verify that Node.js is in your `$PATH` environment variable.

Next, you'll want to clone the book's repository from GitHub. Change to your project directory and type this:

```
$ git clone https://github.com/aendrew/learning-d3
$ cd $ learning-d3
```

This will clone the development environment and all the samples in the `learning-d3/` directory as well as switch you into it.

Another option is to fork the repository on GitHub and then clone your fork instead of mine as was just shown. This will allow you to easily publish your work on the cloud, enabling you to more easily seek support, display finished projects on GitHub Pages, and even submit suggestions and amendments to the parent project. This will help us improve this book for future editions. To do this, fork `aendrew/learning-d3` and replace `aendrew` in the preceding code snippet with your GitHub username.

Each chapter of this book is in a separate branch. To switch between them, type the following command:

```
$ git checkout chapter1
```

Replace `1` with whichever chapter you want the examples for. Stay at `master` for now though. To get back to it, type this line:

```
$ git stash save && git checkout master
```

The `master` branch is where you'll do a lot of your coding as you work through this book. It includes a prebuilt `package.json` file (used by npm to manage dependencies), which we'll use to aid our development over the course of this book. There's also a `webpack.config.js` file, which tells the build system where to put things, and there are a few other sundry config files. We still need to install our dependencies, so let's do that now:

```
$ npm install
```

All of the source code that you'll be working on is in the `src/` folder. You'll notice it contains an `index.html` and an `index.js` file; almost always, we'll be working in `index.js`, as `index.html` is just a minimal container to display our work in. This is it in its entirety, and it's the last time we'll look at any HTML in this book:

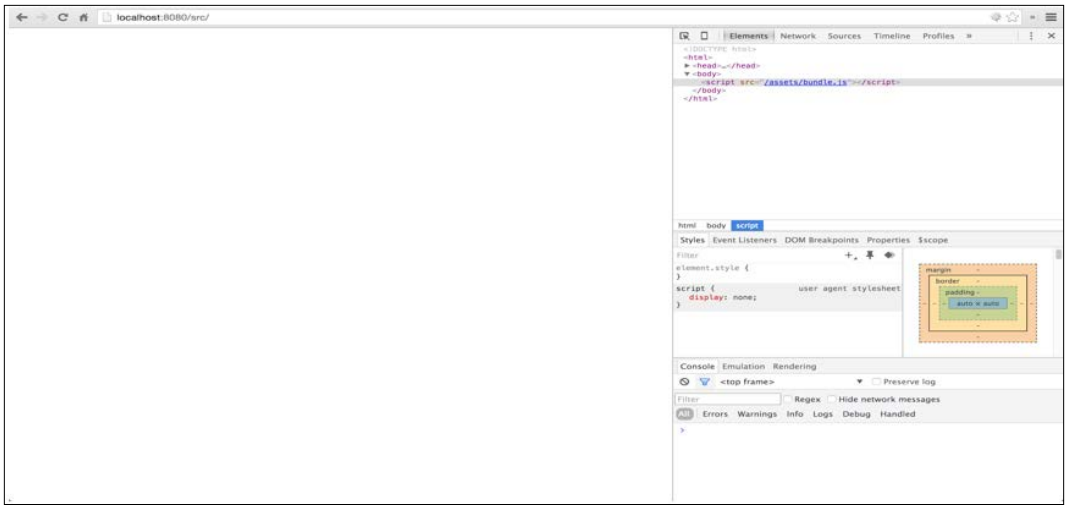
```
<!DOCTYPE html>
<div id="chart"></div>
<script src="/assets/bundle.js"></script>
```

To get things rolling, start the development server by typing the following line:

```
$ npm start
```

This starts up the Webpack development server, which will transform our ES2016 JavaScript into backwards-compatible ES5, which can easily be loaded by most browsers. In the preceding HTML, `bundle.js` is the compiled code produced by Webpack.

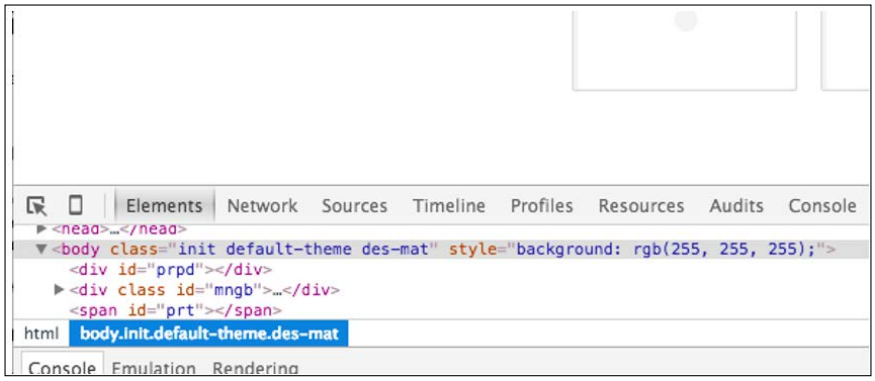
Now point Chrome to `localhost:8080` and fire up the developer console (*Ctrl* +*Shift* + *J* for Linux and Windows and *Option* + *Command* + *J* for Mac). You should see a blank website and a blank JavaScript console with a Command Prompt waiting for some code:



A quick Chrome Developer Tools primer

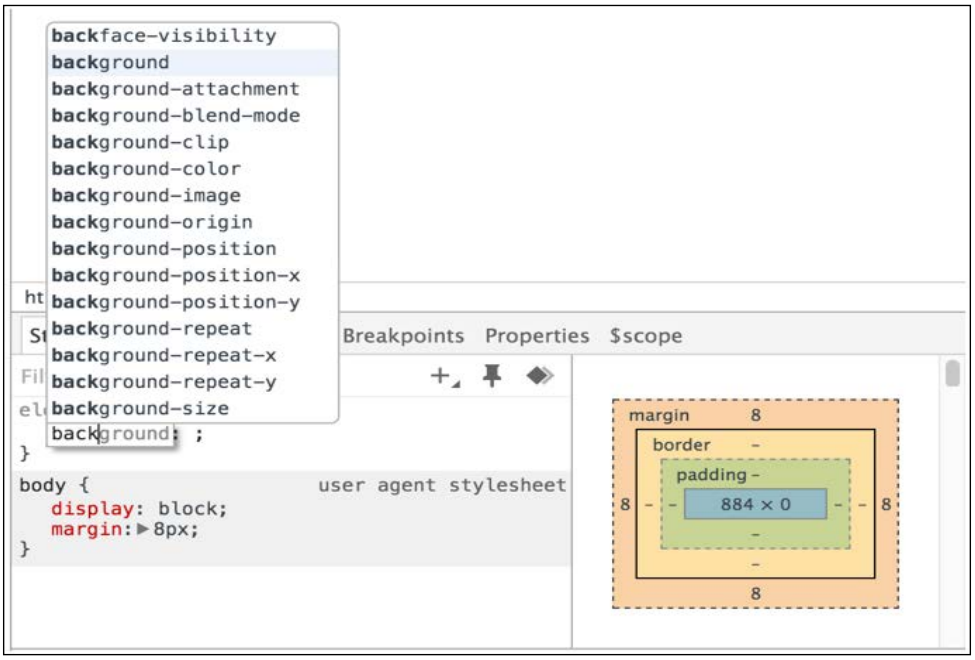
Chrome Developer Tools are indispensable to web development. Most modern browsers have something similar, but to keep this book shorter, we'll stick to Chrome here for the sake of simplicity. Feel free to use a different browser. Firefox's Developer Edition is particularly nice.

We are mostly going to use the **Elements** and **Console** tabs, **Elements** to inspect the DOM and **Console** to play with JavaScript code and look for any problems. The other six tabs come in handy for large projects:



The **Network** tab will let you know how long files are taking to load and help you inspect the Ajax requests. The **Profiles** tab will help you profile JavaScript for performance. The **Resources** tab is good for inspecting client-side data. **Timeline** and **Audits** are useful when you have a global variable that is leaking memory and you're trying to work out exactly why your library is suddenly causing Chrome to use 500 MB of RAM. While I've used these in D3 development, they're probably more useful when building large web applications with frameworks such as React and Angular.

One of the favorites from Developer Tools is the CSS inspector at the bottom of the screen. It can tell you what CSS rules are affecting the styling of an element, which is very good for hunting rogue rules that are messing things up. You can also edit the CSS and immediately see the results, as follows:



The obligatory bar chart example

No introductory chapter on D3 would be complete without a basic bar chart example. They are to D3 as "Hello World" is to everything else, and 90 percent of all data storytelling can be done in its simplest form with an intelligent bar or line chart. For a good example of this, look at the kinds of graphics *The Economist* includes with their articles—they frequently summarize the entire piece with a simple line chart. Coming from a newsroom development background, many of my examples will be related to some degree to current events or possible topics worth visualizing with data. The news development community has been really instrumental in creating the environment for D3 to flourish, and it's increasingly important for aspiring journalists to have proficiency in tools such as D3.

The first dataset that we'll use is UNHCR's regional population data.

The documentation for this endpoint is at `data.unhcr.org/wiki/index.php/Get-population-regional.html`.

We'll create a bar for each population of displaced people. The first step is to get a basic container set up, which we can then populate with all of our delicious new ES2016 code. At the top of `index.js`, put the following code:

```
export class BasicChart {
  constructor(data) {
    var d3 = require('d3'); // Require D3 via Webpack
    this.data = data;
    this.svg = d3.select('div#chart').append('svg');
  }
}
var chart = new BasicChart();
```

If you open this in your browser, you'll get the following error on your console:

Uncaught Error: Cannot find module "d3"

This is because we haven't installed it yet. You'll notice on line 3 of the preceding code that we import D3 by requiring it. If you've used D3 before, you might be more familiar with it attached to the `window` global object. This is essentially the same as including a script tag that references D3 in your HTML document, the only difference being that Webpack uses the Node version and compiles it into your `bundle.js`.

To install D3, you use npm. In your project directory, type the following line:

```
$ npm install d3 --save
```

This will pull the latest version of D3 from `npmjs.org` to the `node_modules` directory and save a reference to it and its version in your `package.json` file. The `package.json` file is really useful; instead of keeping all your dependencies inside of your Git repository, you can easily redownload them all just by typing this line:

```
$ npm install
```

If you go back to your browser and switch quickly to the **Elements** tab, you'll notice a new SVG element as a child of `#chart`.

Go back to `index.js`. Let's add a bit more to the constructor before I explain what's going on here:

```
export class BasicChart {
  constructor(data) {
    var d3 = require('d3'); // Require D3 via Webpack
    this.data = data;
    this.svg = d3.select('div#chart').append('svg');
    this.margin = {
      left: 30,
      top: 30,
      right: 0,
      bottom: 0
    };
    this.svg.attr('width', window.innerWidth);
    this.svg.attr('height', window.innerHeight);
    this.width = window.innerWidth - this.margin.left -
this.margin.right;
    this.height = window.innerHeight - this.margin.top -
this.margin.bottom;
    this.chart = this.svg.append('g')
      .attr('width', this.width)
      .attr('height', this.height)
.attr('transform', `translate(${this.margin.left},
${this.margin.top})`);
  }
}
```

Okay, here we have the most basic container you'll ever make. All it does is attach data to the class:

```
    this.data = data;
```

This selects the `#chart` element on the page, appending an SVG element and assigning it to another class property:

```
    this.svg = d3.select('div#chart').append('svg');
```

Then it creates a third class property, `chart`, as a group that's offset by the margins:

```
    this.width = window.innerWidth - this.margin.left -
this.margin.right;
    this.height = window.innerHeight - this.margin.top -
this.margin.bottom;
    this.chart = svg.append('g')
      .attr('width', this.width)
      .attr('height', this.height)
      .attr('transform', `translate(${this.margin.left},
${this.margin.top})`);
```

Notice the snazzy new ES2016 string interpolation syntax—using `` `backticks` ``, you can then echo out a variable by enclosing it in `${` and `}`. No more concatenating!

The preceding code is not really all that interesting, but wouldn't it be awesome if you never had to type that out again? Well! Because you're a total boss and are learning ES2016 like all the cool kids, you won't ever have to. Let's create our first child class!

We're done with `BasicChart` for the moment. Now, we want to create our actual bar chart class:

```
export class BasicBarChart extends BasicChart {
  constructor(data) {
    super(data);
  }
}
```

This is probably very confusing if you're new to ES6. First off, we're extending `BasicChart`, which means all the class properties that we just defined a minute ago are now available for our `BasicBarChart` child class. However, if we instantiate a new instance of this, we get the constructor function in our child class. How do we attach the data object so that it's available for both `BasicChart` and `BasicBarChart`?

The answer is `super()`, which merely runs the constructor function of the parent class. In other words, even though we don't assign data to `this.data` as we did previously, it will still be available there when we need it. This is because it was assigned via the parent constructor through the use of `super()`.

We're almost at the point of getting some bars onto that graph; hold tight! But first, we need to define our scales, which decide how D3 maps data to pixel values. Add this code to the constructor of `BasicBarChart`:

```
let x = d3.scale.ordinal()
  .rangeRoundBands([this.margin.left, this.width -
this.margin.right], 0.1);
```

The `x` scale is now a function that maps inputs from an as-yet-unknown domain (we don't have the data yet) to a range of values between `this.margin.left` and `this.width - this.margin.right`, that is, between 30 and the width of your viewport minus the right margin, with some spacing defined by the `0.1` value. Because it's an ordinal scale, the domain will have to be discrete rather than continuous. The `rangeRoundBands` means the range will be split into bands that are guaranteed to be round numbers.

Hoorah! Another fancy new ES2016 feature!

The `let` is the new `var`—you can still use `var` to define variables, but you should use `let` instead because it's limited in scope to the block, statement, or expression on which it is used. Meanwhile, `var` is used for more global variables, or variables that you want available regardless of the block scope. For more on this, visit `http://mdn.io/let`.

If you have no idea what I'm talking about here, don't worry. It just means that you should define variables with `let` because they're more likely to act as you think they should and are less likely to leak into other parts of your code. It will also throw an error if you use it before it's defined, which can help with troubleshooting and preventing sneaky bugs.

Still inside the constructor, we define another scale named `y`:

```
    let y = d3.scale.linear().range([this.height,
this.margin.bottom]);
```

Similarly, the `y` scale is going to map a currently unknown linear domain to a range between `this.height` and `this.margin.bottom`, that is, your viewport height and 30. Inverting the range is important because D3.js considers the top of a graph to be `y=0`. If ever you find yourself trying to troubleshoot why a D3 chart is upside down, try switching the range values.

Now, we define our axes. Add this just after the preceding line, inside the constructor:

```
    let xAxis = d3.svg.axis().scale(x).orient('bottom');
    let yAxis = d3.svg.axis().scale(y).orient('left');
```

We've told each axis what scale to use when placing ticks and which side of the axis to put the labels on. D3 will automatically decide how many ticks to display, where they should go, and how to label them.

Now the fun begins!

We're going to load in our data using Node-style `require` statements this time around. This works because our sample dataset is in JSON and it's just a file in our repository. In later chapters, we'll load in CSV files and grab external data using D3, but for now, this will suffice for our purposes—no callbacks, promises, or observables necessary! Put this at the bottom of the constructor:

```
let data = require('./data/chapter1.json');
```

Once or maybe twice in your life, the keys in your dataset will match perfectly and you won't need to transform any data. This almost never happens, and today is not one of those times. We're going to use basic JavaScript array operations to filter out invalid data and map that data into a format that's easier for us to work with:

```
let totalNumbers = data.filter((obj) => {
return obj.population.length;
  })
  .map(
    (obj) => {
      return {
        name: obj.name,
        population: Number(obj.population[0].value)
      };
    }
  );
```

This runs the data that we just imported through `Array.prototype.filter`, whereby any elements without a population array are stripped out. The resultant collection is then passed through `Array.prototype.map`, which creates an array of objects, each comprised of a name and a population value.

We've turned our data into a list of two-value dictionaries. Let's now supply the data to our `BasicBarChart` class and instantiate it for the first time. Consider the line that says the following:

```
var chart = new BasicChart();
```

Replace it with this line:

```
var myChart = new BasicBarChart(totalNumbers);
```

The `myChart.data` variable will now equal `totalNumbers`!

Go back to the constructor in the `BasicBarChart` class.

Remember the `x` and `y` scales from before? We can finally give them a domain and make them useful. Again, a scale is a simply a function that maps an **input range** to an **output domain**:

```
x.domain(data.map((d) => { return d.name }));
y.domain([0, d3.max(data, (d) => { return d.population; })]);
```

Hey, there's another ES2016 feature! Instead of typing `function() {}` endlessly, you can now just put `() => {}` for anonymous functions. Other than being six keystrokes less, the "fat arrow" doesn't bind the value of `this` to something else, which can make life a lot easier. For more on this, visit `http://mdn.io/Arrow_functions`.

Since most D3 elements are objects and functions at the same time, we can change the internal state of both scales without assigning the result to anything. The domain of `x` is a list of discrete values. The domain of `y` is a range from `0` to the `d3.max` of our dataset—the largest value.

Now we're going to draw the axes on our graph:

```
this.chart.append('g')
    .attr('class', 'axis')
    .attr('transform', `translate(0, ${this.height})`)
    .call(xAxis);
```

We've appended an element called `g` to the graph, given it the `axis` CSS class, and moved the element to a place in the bottom-left corner of the graph with the `transform` attribute.

Finally, we call the `xAxis` function and let D3 handle the rest.

The drawing of the other axis works exactly the same, but with different arguments:

```
this.chart.append('g')
    .attr('class', 'axis')
    .attr('transform', `translate(${this.margin.left}, 0)`)
    .call(yAxis);
```

Now that our graph is labeled, it's finally time to draw some data:

```
this.chart.selectAll('rect')
    .data(data)
    .enter()
    .append('rect')
    .attr('class', 'bar')
    .attr('x', (d) => { return x(d.name); })
```

```
        .attr('width', x.rangeBand())
        .attr('y', (d) => { return y(d.population); })
        .attr('height', (d) => { return this.height -
y(d.population); });
```

Okay, there's plenty going on here, but this code is saying something very simple. This is what it says:

- For all rectangles (`rect`) in the graph, load our data
- Go through it
- For each item, append a `rect`
- Then define some attributes

Ignore the fact that there *aren't* any rectangles initially; what you're doing is creating a selection that is bound to data and then operating on it. I can understand that it feels a bit weird to operate on non-existent elements (this was personally one of my biggest stumbling blocks when I was learning D3), but it's an idiom that shows its usefulness later on when we start adding and removing elements due to changing data.

The `x` scale helps us calculate the horizontal positions, and `rangeBand` gives the width of the bar. The `y` scale calculates vertical positions, and we manually get the height of each bar from `y` to the bottom. Note that whenever we needed a different value for every element, we defined an attribute as a function (`x`, `y`, and `height`); otherwise, we defined it as a value (`width`).

Keep this in mind when you're tinkering.

Let's add some flourish and make each bar grow out of the horizontal axis. Time to dip our toes into animations!

Modify the code you just added to resemble the following. I've highlighted the lines that are different:

```
this.chart.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', (d) => { return x(d.name); })
  .attr('width', x.rangeBand())
  .attr('y', () => { return y(this.margin.bottom); })
  .attr('height', 0)
  .transition()
```

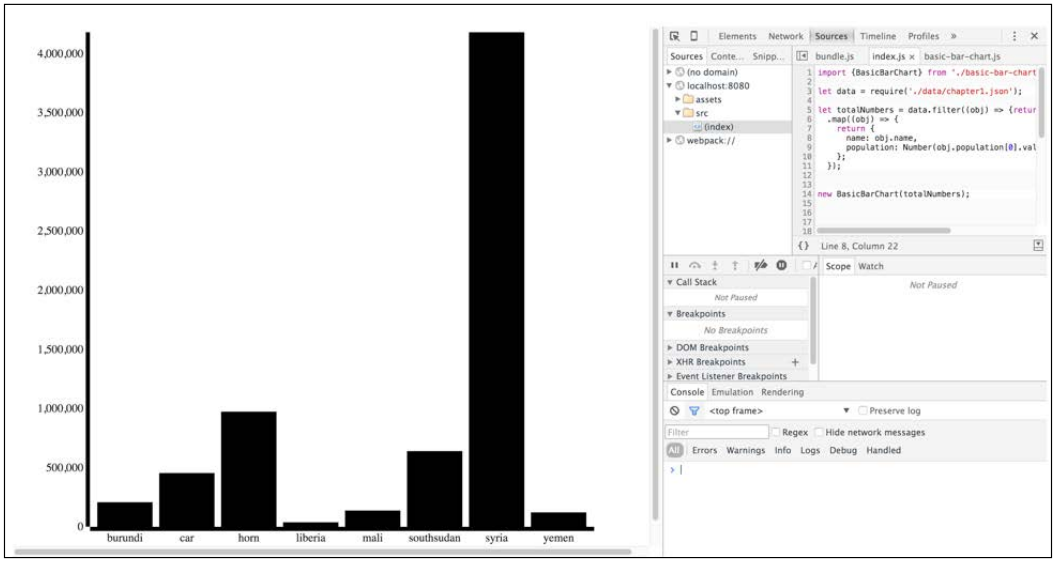
```
.delay((d, i) => { return i*20; })
.duration(800)
.attr('y', (d) => { return y(d.population); })
.attr('height', (d) => {
     return this.height - y(d.population);
  });
```

The difference is that we statically put all bars at the bottom (`margin.bottom`) and then entered a transition with `.transition()`. From here on, we define the transition that we want.

First, we wanted each bar's transition delayed by 20 milliseconds using `i*20`. Most D3 callbacks will return the datum (or "whatever data has been bound to this element," which is typically set to `d`) and the index (or the ordinal number of the item currently being evaluated, which is typically `i`) while setting the `this` argument to the currently selected DOM element. Because of this last point, we use the fat arrow—so that we can still use the class `this.height` property. Otherwise, we'd be trying to find the height property on our `SVGRect` element, which we're midway to trying to define!

This gives the histogram a neat effect, gradually appearing from left to right instead of jumping up at once. Next, we say that we want each animation to last just shy of a second, with `.duration(800)`. At the end, we define the final values for the animated attributes—`y` and height are the same as in the previous code—and D3 will take care of the rest.

Save your file and the page should auto-refresh in the background. If everything went according to plan, you should have a chart that looks like the following:



According to this UNHCR data from June 2015, by far the largest number of displaced persons are from Syria. Hey, look at this—we kind of just did some data journalism here! Remember that you can look at the entire code on GitHub at `http://github.com/aendrew/learning-d3/tree/chapter1` if you didn't get something similar to the preceding screenshot.

We still need to do just a bit more, mainly by using CSS to style the SVG elements.

We could have just gone to our HTML file and added CSS, but then that means opening that yucky `index.html` file. And where's the fun in writing HTML when we're learning some newfangled JavaScript?!

First, create an `index.css` file in your `src/` directory:

```
html, body {
  padding: 0;
  margin: 0;
}

.axis path, .axis line {
  fill: none;
  stroke: #eee;
  shape-rendering: crispEdges;
}

.axis text {
  font-size: 11px;
}

.bar {
  fill: steelblue;
}
```

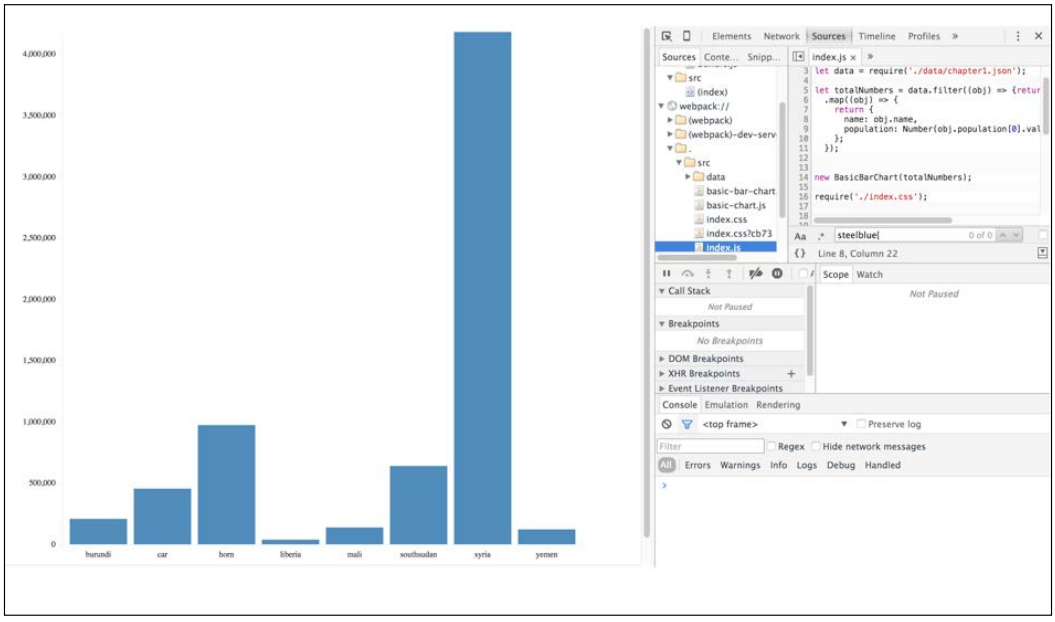
Then just add the following line to `index.js`:

```
require('./index.css');
```

I know. Crazy, right?! No `<style>` tags needed!

It's worth noting that anything involving `require` is the result of a Webpack loader; in this chapter, we've used both the CSS/Style and JSON loaders. Although the author of this text is a fan of Webpack, all we're doing is compiling the styles into `bundle.js` with Webpack instead of requiring them globally via a `<style>` tag. This is cool because instead of uploading a dozen files when deploying your finished code, you effectively deploy one optimized bundle. You can also scope CSS rules to be particular to when they're being included and all sorts of other nifty stuff; for more information, refer to `github.com/webpack/css-loader#local-scope`.

Looking at the preceding CSS, you can now see why we added all those classes to our shapes—we can now directly reference them when styling with CSS. We made the axes thin, gave them a light gray color, and used a smaller font for the labels. The bars should be light blue. Save and wait for the page to refresh. We've made our first D3 chart!



I recommend fiddling with the values for `width`, `height`, and `margin` inside of `BasicChart` to get a feel of the power of D3. You'll notice that everything scales and adjusts to any size without you having to change other code. Smashing!

Summary

In this chapter, you learned what D3 is and took a glance at the core philosophy behind how it works. You also set up your computer for prototyping ideas and to play with visualizations. This environment will be assumed throughout the book.

We went through a simple example and created an animated histogram using some of the basics of D3. You learned about scales and axes, that the vertical axis is inverted, that any property defined as a function is recalculated for every data point, and that we use a combination of CSS and SVG to make things beautiful. We also did a lot of fancy stuff with ES2016, Babel, and Webpack and got Node.js installed. Go us!

Most of all, this chapter has given you the basic tools so that you can start playing with D3.js on your own. Tinkering is your friend! Don't be afraid to break stuff—you can always reset to a chapter's default state by running `$ git reset --hard origin/chapter1`, replacing `1` with whichever chapter you're on.

Next, we'll be looking at all this a bit more in depth, specifically how the DOM, SVG, and CSS interact with each other. This chapter discussed quite a lot, so if some parts got away from you, don't worry. Just power through to the next chapter and everything will start to make a lot more sense!

2
A Primer on DOM, SVG, and CSS

In this chapter, we'll take a look at the core technologies that make D3 tick, and they are as follows:

- **Document Object Model** (**DOM**)
- **Scalable Vector Graphics** (**SVG**)
- **Cascading Style Sheets** (**CSS**)

You're probably used to manipulating DOM and CSS with libraries such as jQuery or MooTools, but D3 has a full suite of manipulation tools as well.

SVG is at the core of building truly great visualizations, so we'll take special care to understand it—everything from manually drawing shapes to transformations and path generators.

DOM

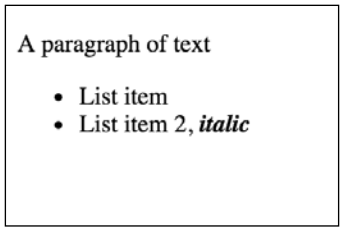
The **Document Object Model** (**DOM**) is a language-agnostic model for representing structured documents built in HTML, XML, or similar standards. You can think of it as a tree of nodes that closely resembles the document parsed by the browser.

At the top, there is an implicit document node; this node represents the `<html>` tag. Browsers create this tag even if you don't specify it and then build the tree off this root node according to what your document looks like. Suppose you have a simple HTML file like this:

```
<!DOCTYPE html>
<title>A title</title>
<div>
```

```
<p>A paragraph of text</p>
</div>
<ul>
<li>List item</li>
<li>List item 2, <em><strong>italic</strong></em></li>
</ul>
```

Then Chrome will parse the preceding code to DOM as follows:



Type `document` in the Chrome JavaScript console to get this tree view. You can expand it by double-clicking. Chrome will then highlight the section of the page that relates to the specified element when you hover over it in the console.

Manipulating the DOM with D3

Every node in a DOM tree comes with a slew of methods and properties that you can use to change the look of the rendered document.

Take for instance the HTML code in our previous example. If we want to change the word `italic` to make it underlined as well as bold and italic (the result of the `<em>` and `<strong>` tags), we can do it using the following code:

```
document.getElementsByTagName('strong')[0].style
  .setProperty('text-decoration', 'underline')
```

Whoa, that's a lot of code!

We took the root `document` node and found every node created from a `<strong>` tag. Then we took the first item in this array and added a `text-decoration` property to its style property.

The sheer amount of code it took to do something this simple in a document with only 11 nodes is why few people today use the DOM API directly— not to mention all the subtle differences between browsers. Since we'd like to keep our lives simple and avoid using the DOM directly, we need a library. We can use jQuery; or we can use D3, which comes with a similar set of tools for selecting and manipulating the DOM.

This means we can treat HTML as just another type of data visualization. Let that one sink in. HTML is data visualization!

In practice, this means that we can use similar techniques to present data as a table or an interactive image. Most of all, we can use the same data.

Let's rewrite the previous example in D3:

```
d3.select('strong').style('text-decoration', 'underline')
```

Much simpler! We selected the `strong` element and defined a `style` property. Job done!

By the way, any property you set with D3 can be dynamic, so you can assign a function as well as a value. This is going to come in handy later.

What we just did is called a selection. Since selections are the core of everything we do with D3, let's take a look at them in more detail.

Selections

A selection is an array of elements pulled from the current document according to a particular CSS selector—this can be anything from a class to an ID, or a tag name. It can even be a funny-looking pseudo-selector that allows us to do things like selecting every other paragraph tag, and it is written as `p:nth-child(n+1)`.

Pseudo-selectors are really powerful when used with D3.js and can often be used in place of a loop or some really difficult math. They're the types of selectors that have a colon in front and describe an element's state. A good example is `:hover`, which is active when the mouse arrow is above a particular element.

Using CSS selectors to decide which elements to work on gives us a simple language for defining elements in the document. It's actually the same as you're used to from jQuery and CSS itself.

To get the first element with ID as a graph, we use `.select('#graph')`. To get all the elements with the `blue` class, we write `.selectAll('.blue')`. To get all the paragraphs in a document, we use `.selectAll('p')`.

We can combine these to get a more complex matching. Think of it as set operations. You can perform a Boolean `AND` operation by using the `.llama.duck` selector; it will get elements that have both the `.llama` and `.duck` classes. Alternatively, you might perform an `OR` operation with `.llama, .duck` to get every element that is either a `llama` *or* a `duck`. But what if you want to select children elements? Nested selections to the rescue!

You can do it with a simple selector such as `tbody td`, or you can chain two `selectAll` calls as `.selectAll('tbody').selectAll('td')`. Both will select all the cells in a table body. Keep in mind that nested selections maintain the hierarchy of the selected elements, which gives us some interesting capabilities. Let's look at a short example.

Let's make a table!

Start by creating a new file in your `src/` directory, and call it `table-builder.js`. We're not going to work directly inside `index.js` from here on. Instead, we're going to create modules that are loaded by `index.js`. This allows us to keep our code clean and split it into manageable pieces.

Because we don't want to play around with HTML inside of our code base too much, we're going to write a bunch of functions to build our tables for us. We'll be making this table:

```
<table class="d3-table">
  <thead>
    <tr>
      <td>One</td>
      <td>Two</td>
      <td>Three</td>
      <td>Four</td>
      <td>Five</td>
      <td>Six</td>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>q</td>
      <td>w</td>
      <td>e</td>
      <td>r</td>
      <td>t</td>
      <td>y</td>
    </tr>
  </tbody>
</table>
```

Now, we could've just copied that into `index.html`, but remember something? We're trying to avoid writing any HTML here! Time to make a table using D3!

Open up `table-builder.js` and create the following class:

```
export class TableBuilder {
  constructor(rows) {
```

```
        var d3 = require('d3');

        this.header = rows.shift(); // Remove the first element for
    the header
        this.data = rows; // Everything else is a normal data row

        var table = d3.select('body').append('table').attr('class',
    'table');
        return table;
      }
    }
```

This will give us the outer container. In order to see this work, we need to tell `index.js` to load our new class. Open that up now. You'll notice that all of our code from the last chapter is still there. How messy! Let's start by cleaning it up using ES2016 modules. Move the code for the `BasicChart` class into `basic-chart.js`, and move `BasicBarChart` into `basic-bar-chart.js`. Lastly, you need to let `BasicBarChart` know where the `BasicChart` class is, so put the following line at the top of `basic-bar-chart.js`:

```
import {BasicChart} from './basic-chart';
```

What's all this now? This almost looks like *Python* or something. Are you *sure* I'm still teaching you JavaScript here…?

Behold, dear reader! For this is the fabulous new ES2016 module loading syntax! Speaking as an open source developer, I think the lack of a canonical module loading spec has been one of the most irritating and frustrating aspects of JavaScript development for a very, *very* long time. ES2016 goes tremendously far to fix this state of affairs by introducing the preceding syntax.

Why use both Webpack's `require()`, that is, CommonJS format and the new ES2016 format?

In this book, we use import for ES2016 code we've written that must be transpiled by Babel, and require to load in dependencies that are already ES5 CommonJS modules. You can also import D3 using ES2016 syntax:

```
import * as d3 from 'd3';
```

There's no particular advantage to using require to get D3, it's just shorter.

In D3 4.0, each major component of D3 is available as a separate ES2016 module, which can help when reducing code size. We use 3.5.x in this book because it's kind of nice to have the entire library available to you while learning.

There's still a bit of code left for getting the first chapter's example set up. If you care to save that first chart for your reference, put it in another file, `chapter1.js`, and add the following `import` statement at the top of that:

```
import {BasicBarChart} from './basic-bar-chart';
```

You should now have an empty `index.js` and all your classes from *Chapter 1, Getting Started with D3, ES2016, and Node.js*, in their own files. This is how we'll organize our code from now on, and it is a good practice to get into by making your code modular. It not only helps you reuse your old code but also trains you to think in a more object-oriented manner.

If you've checked out the `origin/chapter1` branch in Git at any point in time before now, this is where you'll be with the earlier classes in their own files and the data loading code in `chapter1.js`. Alternately, if I lost you at some point along the way, you can catch up by typing the following command inside the `learning-d3/` directory:

`git stash save && git checkout origin/chapter1`

Let's go back to `index.js` and finally get back to creating that table!

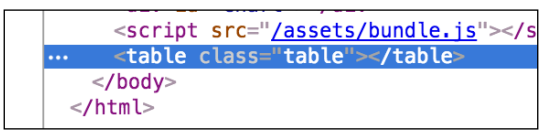
In `index.js`, add the following code:

```
import {TableBuilder} from './table-builder';

let header = ['one', 'two', 'three', 'four', 'five', 'six'];

let rows = [
  header,
  ['q', 'w', 'e', 'r', 't', 'y']
];

let table = new TableBuilder(rows);
```

Go to the command line and type this:

```
$ npm start
```

Then go to `http://127.0.0.1:8080` in your browser. Right-click on the page, go to **Inspect Element**, and you'll see our `table` element:

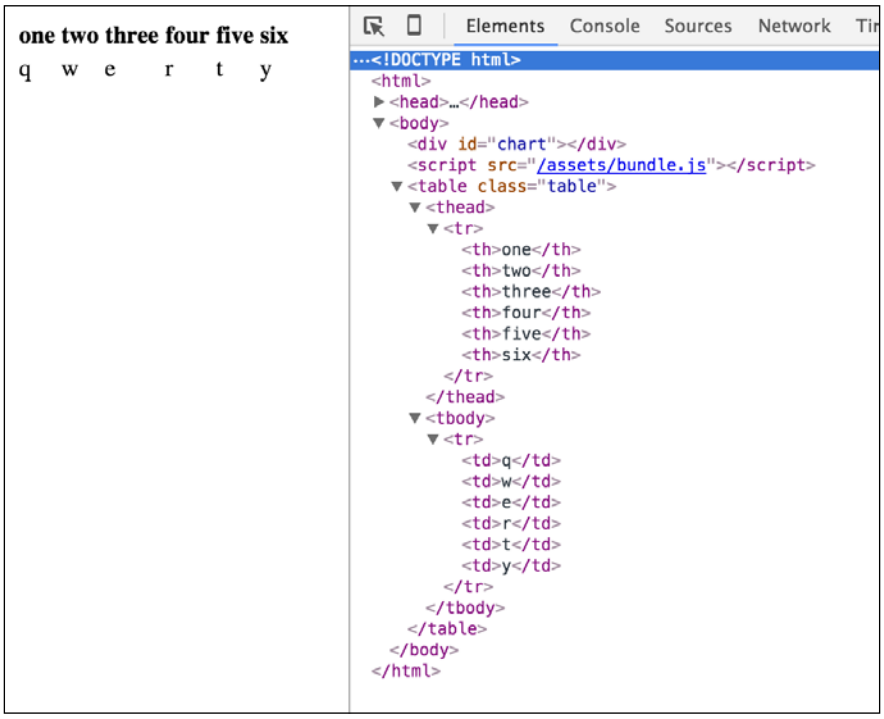


Woo! A `table` element!

Let's go back and add the rest of the table:

```
export class TableBuilder {
  constructor(rows) {
    let d3 = require('d3');

    // Remove the first element for the header
    this.header = rows.shift();
    this.data = rows; // Everything else is a normal row.

    let table = d3.select('body')
      .append('table').attr('class', 'table');

    let tableHeader = table.append('thead').append('tr');
    let tableBody = table.append('tbody');

    // Each element in "header" is a string.
    this.header.forEach(function(value){
      tableHeader.append('th').text(value);
    });

    // Each element in "data" is an array
    this.data.forEach((row) => {
      let tableRow = tableBody.append('tr');

      row.forEach((value) => {
        // Now, each element in "row" is a string
        tableRow.append('td').text(value);
      });
    });

    return table;
  }
}
```

Now your table should look like this:



What exactly did we do here?

The key is in the three `for-each` statements that we used. One loops through the array of table header strings and appends a table cell (`td`, or `th` for header cells) element with each value to the `thead` element's row. Then there are two nested `.forEach` statements that do the same for each row in the body. We technically only have one row in the body right now, so we probably didn't need that messy double `for-each`, but now all we have to do to add another row to the table is simply append another data array to the `rows` variable. We'll talk a bunch more about `Array.prototype.forEach` and other array functions in the next chapter.

This might seem like a lot of work for such a simple table, but the advantages of doing it this way are huge. Instead of wasting a lot of time typing out a totally static table that you'll never use again, you've effectively created a basic JavaScript library that will produce a basic table for you whenever you need it. You can even extend your `TableBuilder` class to do different things—other than what it does now—without ever altering the code you just wrote. We'll do a bit of that in the next few chapters.

Okay, time to finally play with some selections!

Selections example

Let's not mess up our `index.js` file any more than we have to, so replace all its contents with the following:

```
import {TableBuilder} from './table-builder';
window.TableBuilder = TableBuilder;
window.d3 = require('d3');
```

This assigns the `TableBuilder` object to the global `window` object, so we can now use it freely in the console.

In Chrome's Developer console, type the following two lines:

```
d3.selectAll('.table').remove();
new TableBuilder([
  [1,2,3,4,5,6],
  ['q', 'w', 'e', 'r', 't', 'y'],
  ['a', 's', 'd', 'f', 'g', 'h'],
  ['z', 'x', 'c', 'v', 'b', 'n']
]);
```

Psst! If you need to add a newline character in Chrome's Developer console, hold *Shift* while pressing *return*. Note, however, that you don't actually need to do this; you can type the preceding words all as one line if it's easier. I've only presented it this way for clarity.

This removes the old table (if you didn't refresh in the meantime) and adds a new table.

Now, let's make the text in all the table cells red!

```
d3.selectAll('td').style('color', 'red')
```

The text will promptly turn red. Next, let's make everything in the table head bold by chaining two `selectAll` calls:

```
d3.selectAll('thead').selectAll('td').style('font-weight', 'bold')
```

Great! Let's take nested selections a bit further and make the table body cells in the second column and the fourth column green:

```
d3.selectAll('tbody tr').selectAll('td')
  .style('color', (d, i) => { return i%2 ? 'green' : 'red'; })
```

The two `selectAll` calls gave us all the instances of `td` in the body, separated by rows, giving us an array of three arrays with five elements: `[ Array[5], Array[5], Array[5] ]`. Then we used `.style()` to change the color of every selected element.

Using a function instead of a static property gave us the fine-grained control that we needed. The function is called with a `data` attribute (we'll discuss more on this later) and an index of the column it's in, that is, the `i` variable. Since we're using nested selections, a third parameter would give us the row. Then we simply return either "green" or "red" based on the current index.

One thing to keep in mind is that chaining selections can be more efficient than `OR` selectors when it comes to very large documents. This is because each subsequent selection only searches through the elements matched previously.

Manipulating content

We can do far more with D3 than just play around with selections and change the properties of elements. We can manipulate things.

With D3, we can change the contents of an element, add new elements, or remove elements that we don't want.

Let's add a new column to the table from our previous example:

```
var newCol = d3.selectAll('tr').append('td')
```

We selected all the table rows and then appended a new cell to each using `.append()`. All D3 actions return the current selection of new cells in this case, so we can chain actions or assign the new selection to a `newCol` variable for later use.

We have an empty, invisible column on our hands. Let's add some text to spruce things up:

```
newCol.text('a')
```

At least now that it's full of instances of `a`, we can say that a column is present. But that's kind of pointless, so let's follow the pattern set by other columns:

```
newCol.text( (d, i) => { return ['Seven', 'u', 'j', 'm'][i] })
```

The trick of dynamically defining the content via a function helps us pick the right string from a list of values depending on the column we're in, which we identify by the index `i`. Figured out the pattern yet?

Similarly, we can remove elements using `.remove()`. To get rid of the last row in the table, you'd write something as follows:

```
d3.selectAll('tr')[0][3].remove()
```

You have to use `[0][3]` instead of just `[3]` because selections are arrays of arrays.

Joining data to selections

We've made it to the fun part of our DOM shenanigans. Remember when I said HTML is data visualization? Joining data to selections is how that happens.

To join data with a selection, we use the `.data()` function. It takes a data argument in the form of a function or an array, and optionally a function telling D3 how to differentiate between various parts of data.

When you join data to a selection, one of the following three things will happen:

- There is more data than was already joined (the length of the data is longer than the length of a selection). You can reference the new entries with the `.enter()` function.
- There is exactly the same amount of data as before. You can use the selection returned by `.data()` itself to update element states.
- There is less data than before. You can reference these using the `.exit()` function.

You can't chain `.enter()` and `.exit()` because they are just references and don't create a new selection. This means that you will usually want to focus on `.enter()` and `.exit()` and handle the three cases separately. Mind you, all three can happen at once.

You must be wondering, "But how's it possible for there to be both more and less data than before?" That's because selection elements are bound to each individual datum and not their number. If you shift an array and then push a new value, the previous first item would go to the `.exit()` reference and the new addition would go to the `.enter()` reference.

"Datum" is the singular of "data." You know the `d` argument that we usually pass in the callback functions, right? That's what it stands for!

Let's build something cool with data joins and HTML.

An HTML visualization example

Start off by creating a new file called `chapter2.js` inside `src/` and replacing all of the code in `index.js` with this:

```
import {renderDailyShowGuestTable} from './chapter2';
renderDailyShowGuestTable();
```

Then add the following code to `chapter2.js`:

```
import {TableBuilder} from './table-builder';
export function renderDailyShowGuestTable() {
  let url =
      'https://cdn.rawgit.com/fivethirtyeight/data/master/daily-show-
guests/daily_show_guests.csv';

  let table = new TableBuilder(url);
}
```

This creates a new function that instantiates `TableBuilder`. We then run this function in `index.js`.

For this example, we're going to visualize *FiveThirtyEight's* dataset of every guest who was ever on *The Daily Show* with Jon Stewart. This is available at `https://github.com/fivethirtyeight/data/blob/master/daily-show-guests/`.

We're going to use our fancy new `TableBuilder` class to visualize this data in a useful way.

Let's start by taking another look at our `TableBuilder` class. Open it up and rewrite it so that it looks like this:

```
let d3 = require('d3');

export class TableBuilder {
  constructor(url) {
    this.load(url);
    this.table = d3.select('body').append('table')
      .attr('class', 'table');
    this.tableHeader = this.table.append('thead');
    this.tableBody = this.table.append('tbody');
  }

  load(url) {
    d3.csv(url, (data) => {
      this.data = data;
      this.redraw();
```

```
    });
  }

  redraw() {
    // Redraw code will be here
  }
}
```

We've gotten rid of those nasty `for-each` loops and added a few class methods, one for loading in data and another for updating the data. Let's quickly look at `d3.csv()`:

```
    d3.csv(url, (data) => {
      this.data = data;
      this.redraw();
    });
```

Here, we supply `d3.csv()` with a URL (though it can also be a local path) pointing at a CSV file; in this case, it's our `The` *Daily Show* data. Once `d3.csv()` retrieves the data, it fires the callback in the next argument, wherein the retrieved data is attached to the class object and `redraw` is called.

We'll be messing with the dataset later, so it's handy to have a function that we can call when we want to reload the data without having to refresh the page.

Because our dataset is in CSV format, we use the `csv` function of D3 to load and parse it. D3 is smart enough to understand that the first row in our dataset is not data but a set of labels, so it populates the `data` variable with an array of objects, as follows:

```
{
  GoogleKnowlege_Occupation: "actor",
  Group: "Acting"
  Raw_Guest_List: "Michael J. Fox",
  Show: "1/11/99",
  YEAR: "1999"
}
```

Our next step is to make `redraw()` actually do something. Update `redraw()` to resemble the following code:

```
  redraw() {
    this.rows = this.tableBody.selectAll('tr').data(this.data);
    this.rows.enter().append('tr');
    this.rows.exit().remove();
  }
```

The code is divided into three parts. The first part selects all the table rows (of which none exist yet) and joins our data using the `.data()` function. The resulting selection is saved in the `rows` class property.

Next, we create a table row for every new datum in the dataset using the `.enter()` reference. Right now, this is for all of them.

The last part of this code doesn't do anything yet but will remove any `<tr>` element in the `.exit()` reference once we change the data later.

After execution, the `rows` property will hold an array of `<tr>` elements, each bound to its respective place in the dataset. The first `<tr>` element holds the first datum, the second holds the second datum, and so on.

Rows are useless without cells. Let's add some by relying on the fact that data stays joined to elements even after a new selection:

```
this.rows.selectAll('td')
  .data(d => d3.values(d))
  .enter()
  .append('td')
  .text(d => d);
```

More fun with double-arrow functions! Notice how, in the preceding code, I've done away with a couple of parentheses and curly brackets, not to mention a `return` statement. This is an expression body, and if you just want to return the value of an object or function, you don't need stinkin' `return` statements or curly brackets! This alone results in code that is so much shorter and so much more readable that you'll wonder why you took so long to update?

We selected all the `<td>` children of each row (none exist yet). We then had to call the `.data()` function with the same data transformed into a list of values using `d3.values()`. This gave us a new chance to use `.enter()`.

From then on, it's more of the same. Each new entry gets its own table cell, and the text is set to the current datum.

Save everything and switch to Chrome. You will now have a simple but effective table detailing every *The Daily Show* guest from when Jon Stewart took over hosting the show in 1999 up until his final show in 2015.

1999 actor	1/11/99	Acting	Michael J. Fox
1999 Comedian	1/12/99	Comedy	Sandra Bernhard
1999 television actress	1/13/99	Acting	Tracey Ullman
1999 film actress	1/14/99	Acting	Gillian Anderson
1999 actor	1/18/99	Acting	David Alan Grier
1999 actor	1/19/99	Acting	William Baldwin
1999 Singer-lyricist	1/20/99	Musician	Michael Stipe
1999 model	1/21/99	Media	Carmen Electra
1999 actor	1/25/99	Acting	Matthew Lillard
1999 stand-up comedian	1/26/99	Comedy	David Cross
1999 actress	1/27/99	Acting	Yasmine Bleeth
1999 actor	1/28/99	Acting	D. L. Hughley
1999 television actress	10/18/99	Acting	Rebecca Gayheart
1999 Comedian	10/19/99	Comedy	Steven Wright
1999 actress	10/20/99	Acting	Amy Brenneman
1999 actress	10/21/99	Acting	Melissa Gilbert
1999 actress	10/25/99	Acting	Cathy Moriarty
1999 comedian	10/26/99	Comedy	Louie Anderson
1999 actress	10/27/99	Acting	Sarah Michelle Gellar
1999 Singer-songwriter	10/28/99	Musician	Melanie C
1999 actor	10/4/99	Acting	Greg Proops
1999 television personality	10/5/99	Media	Maury Povich
1999 actress	10/6/99	Acting	Brooke Shields
1999 Comic	10/7/99	Comedy	Molly Shannon
1999 actor	11/1/99	Acting	Chris O'Donnell
1999 actress	11/15/99	Acting	Christina Ricci
1999 Singer-songwriter	11/16/99	Musician	Tori Amos
1999 actress	11/17/99	Acting	Yasmine Bleeth
1999 comedian	11/18/99	Comedy	Bill Maher
1999 actress	11/2/99	Acting	Jennifer Love Hewitt
1999 rock band	11/29/99	Musician	Goo Goo Dolls
1999 musician	11/3/99	Musician	Dave Grohl
1999 Film actor	11/30/99	Acting	Stephen Rea
1999 Model	11/4/99	Media	Roshumba Williams

Let's try sorting by some arbitrary property, for instance, the interviewee's occupation. To do so, add this code to the bottom of `redraw()`:

```
this.tableBody.selectAll('tr')
  .sort((a, b) => d3.ascending(a.Group, b.Group));
```

Without doing anything else, this code will redraw the table with the new ordering—no refreshing the page and no manually adding or removing elements. Because all our data is joined to the HTML, we didn't even need a reference to the original `tr` selection or the data. Pretty nifty, if you ask me!

The `.sort()` function takes only a comparator function. The comparator is given two pieces of data and must decide how to order them: `-1` for being less than `b`, `0` for being equal, and `1` for being more than `b`. You can also use the `d3.ascending` and `d3.descending` comparators of D3.

That's still pretty unclear though. Let's group by interviewee name in order to remove duplicates. Rewrite `redraw()` to resemble the following:

```
  redraw() {
    let nested = d3.nest()
      .key(d => d['Raw_Guest_List'])
      .entries(this.data);

    this.data = nested.map(d => {
      let earliest = d.values.sort((a, b) => d3.ascending(a.YEAR,
b.YEAR)).shift();

      return {
        name: d.key,
        category: earliest.Group,
        'earliest appearance': earliest.YEAR
      }
    });

    this.rows = this.tableBody.selectAll('tr').data(this.data);
    this.rows.enter().append('tr');
    this.rows.exit().remove();

    this.rows.selectAll('td')
      .data(d => d3.values(d))
      .enter()
      .append('td')
      .text(d => d);
  }
```

The last part looks familiar, but what's happening up top?

The first thing we do is create a nest, which is a really terrific feature that D3 has for grouping objects in arrays. This results in an array of objects grouped by the `Raw_Guest_List` key (that is, the interviewee's name); we then rebuild the array further using `Array.prototype.map`. Inside the `map` function, we merge the `values` object that was created by `d3.nest` by first sorting appearances by year and then using `Array.prototype.shift()` to pull off the first item in the array. In the `return` statement for the `map` function, we then cherry-pick the attributes we want to finally display in the table.

D3 has a ridiculous number of array helper functions that are all conveniently located in a rather obtuse and hard-to-read piece of documentation. You probably won't ever need to memorize or use them all, but if you ever have a hankerin' for a round of code golf on a Friday afternoon…:

`https://github.com/mbostock/d3/wiki/Arrays`

That's a much nicer table!

Michael J. Fox	Acting	1999
Sandra Bernhard	Comedy	1999
Tracey Ullman	Acting	1999
Gillian Anderson	Acting	1999
David Alan Grier	Acting	1999
William Baldwin	Acting	1999
Michael Stipe	Musician	1999
Carmen Electra	Media	1999
Matthew Lillard	Acting	1999
David Cross	Comedy	1999
Yasmine Bleeth	Acting	1999
D. L. Hughley	Acting	1999
Rebecca Gayheart	Acting	1999
Steven Wright	Comedy	1999
Amy Brenneman	Acting	1999
Melissa Gilbert	Acting	1999
Cathy Moriarty	Acting	1999
Louie Anderson	Comedy	1999
Sarah Michelle Gellar	Acting	1999
Melanie C	Musician	1999
Greg Proops	Acting	1999
Maury Povich	Media	1999
Brooke Shields	Acting	1999
Molly Shannon	Comedy	1999
Chris O'Donnell	Acting	1999
Christina Ricci	Acting	1999
Tori Amos	Musician	1999
Bill Maher	Comedy	1999
Jennifer Love Hewitt	Acting	1999
Goo Goo Dolls	Musician	1999

Scalable Vector Graphics

Scalable Vector Graphics (**SVG**) is a vector graphics format that describes images with XML. It's been around since 1999 and is supported by all major browsers these days (Internet Explorer only introduced it in IE9, but at the time of writing this book, 96.5 percent of Internet users can render SVG in their browsers as per `caniuse.com/#feat=svg`). Vector images can be rendered in any size without becoming fuzzy. This means that you can render the same image on a large retina display or a small mobile phone, and it will look great in both cases.

SVG images are made up of shapes you can create from scratch using paths, or put together from basic shapes defined in the standard, for example, a line or a circle. The format itself represents shapes with XML elements and some attributes.

As such, SVG code is just a bunch of text that you can edit manually, inspect with your browser's normal debugging tools, and compress with standard text compression algorithms. Being text-based also means that you can use D3 to create an image in your browser, then copy and paste the resulting XML to a `.svg` file, and open it with any SVG viewer.

Another consequence is that browsers can consider SVG to be a normal part of the document. You can use CSS for styling, listening for mouse events on specific shapes, and even scripting the image to make animations where images are interactive.

Drawing with SVG

To draw with D3, you can add shapes manually by defining the appropriate SVG elements, or you can use helper functions that help you create advanced shapes easily.

Now we're going to go through the very core of what D3 does. Everything else builds from this, so pay attention.

Let's start by importing our old friend, `BasicChart`, and rearranging `chapter2.js` a bit:

```
import {TableBuilder} from './table-builder';
import {BasicChart} from './basic-chart';

let d3 = require('d3');

export default function() {
  let svg = new BasicChart().chart;
}

export function renderDailyShowGuestTable() {
```

```
  let url =
      'https://cdn.rawgit.com/fivethirtyeight/data/master/daily-show-
guests/daily_show_guests.csv';

  let table = new TableBuilder(url);
}
```

Nothing too surprising here. You now have an SVG element that expands to the entire screen size, with a group object inside defining marginalia. This has been assigned to the `svg` variable in our default function. Replace `index.js` with the following:

```
import ch2 from './chapter2';
ch2();
```

You may have noticed that the preceding `import` statement in `index.js` doesn't have curly brackets around `ch2` and… hey, wait a minute! Where are we getting this `ch2` nonsense from anyhow?!

One thing that ES2016 modules allow is exporting a default function. This allows somebody who's importing the module to call the imported class or function whatever they like. In this case, we've simply called it `ch2`. The same works for classes. You can also export both named and default items, which can be useful if you are writing a class that has a lot of independently acting pieces (for instance, a library comprised of a bunch of math functions).

Manually adding elements and shapes

An SVG image is a collection of elements rendered as shapes and comes with a set of seven basic elements. Almost all of these are just an easier way to define a path:

- Text (the only one that isn't a path)
- Straight lines
- Rectangles
- Circles
- Ellipses
- Polylines (a set of straight lines)
- Polygons (a set of straight lines closing in on itself)

You build SVG images by adding these elements to the canvas and defining some attributes. All of them can have a stroke style defining how the edge is rendered and a fill style defining how the shape is filled. Also, all of them can be rotated, skewed, or moved using the `transform` attribute.

Text

Text is the only element that is neither a shape nor translates to a path in the background like the others. Let's look at it first so that the rest of this chapter can be about shapes. Add the following code at the bottom of your default function in `chapter2.js`:

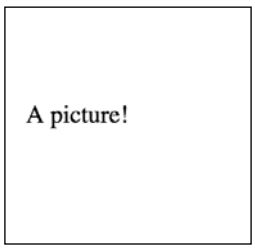
```
svg.append('text')
  .text('A picture!')
  .attr({x: 10,
    y: 150,
    'text-anchor': 'start'});
```

We took our `svg` element and appended a `text` element. Then we defined its actual text, added some attributes to position the text at the `(x, y)` point, and anchored the text at the start.

The `text-anchor` attribute defines the horizontal positioning of rendered text in relation to the anchor point defined by `(x, y)`. The positions it understands are `start`, `middle`, and `end`.

We can also fine-tune the text's position with an offset defined by the `dx` and `dy` attributes. This is especially handy when adjusting the text margin and baseline relative to the font size because it understands the `em` unit.

Our image looks like this:



Shapes

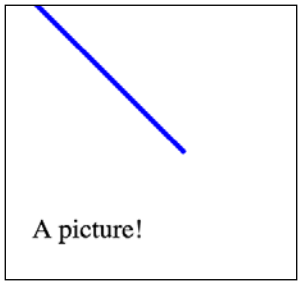
Now that `text` is out of the way, let's look at something useful—shapes, the heart of the rest of this book!

We begin by drawing a straight line using the following code:

```
svg.append('line')
  .attr({x1: 10,
    y1: 10,
    x2: 100,
```

```
    y2: 100,
    stroke: 'blue',
    'stroke-width': 3
  });
```

As before, we took the `svg` element, appended a line, and defined some attributes. A line is drawn between two points: `(x1, y1)` and `(x2, y2)`. To make the line visible, we have to define the `stroke` color and `stroke-width` attributes as well.



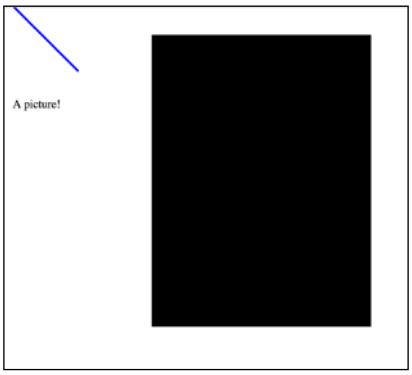
Our line points downwards even though y2 is bigger than y1. That's because the origin in most image formats lies in the top-left corner. This means that (x=0, y=0) defines the top-left corner of the image.

To draw a rectangle, we can use the `rect` element:

```
svg.append('rect')
  .attr({x: 200,
    y: 50,
    width: 300,
    height: 400
  });
```

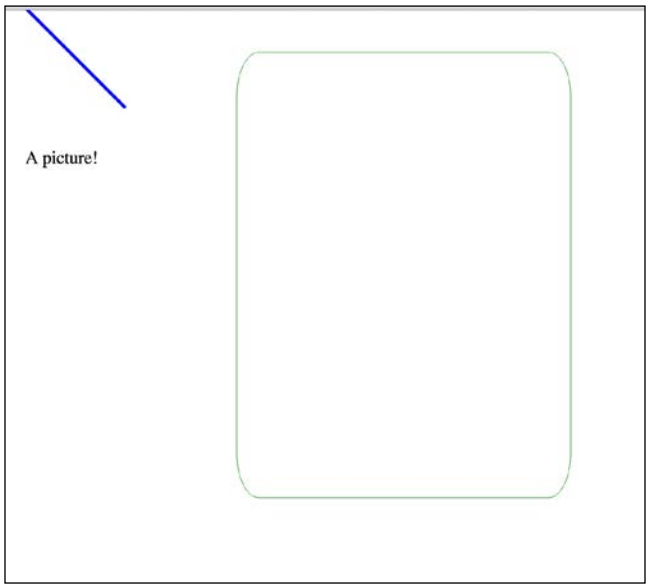
We appended a `rect` element to the `svg` element and defined some attributes. A rectangle is defined by its top-left corner (`(x, y)`), `width`, and `height`.

Our image now looks like this:



We have an unwieldy black rectangle. We can make it prettier by defining three more properties, as follows:

```
svg.select('rect')
  .attr({stroke: 'green',
    'stroke-width': 0.5,
    fill: 'white',
    rx: 20,
    ry: 40
  });
```

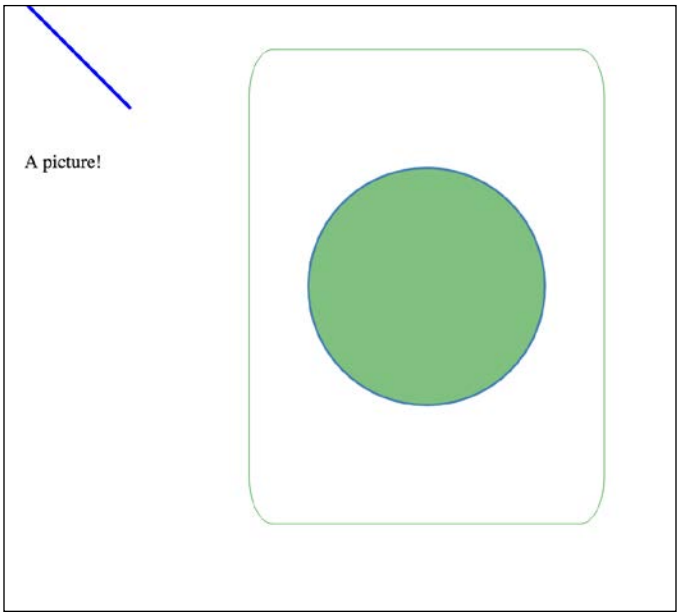


This is much better. Our rectangle has a thin green outline. Rounded corners come from the `rx` and `ry` attributes, which define the corner radius along the *x* and *y* axes.

Let's try adding a circle:

```
svg.append('circle')
  .attr({cx: 350,
    cy: 250,
    r: 100,
    fill: 'green',
    'fill-opacity': 0.5,
    stroke: 'steelblue',
    'stroke-width': 2
  });
```

A circle is defined by a central point, `(cx, cy)`, and a radius, `r`. In this instance, we get a green circle with a steel blue outline in the middle of our rectangle. The `fill-opacity` attribute tells the circle to be slightly transparent so that it doesn't look too strong against the light rectangle:



Mathematically speaking, a circle is just a special form of ellipse. By adding another radius and changing the element, we can draw one of these:

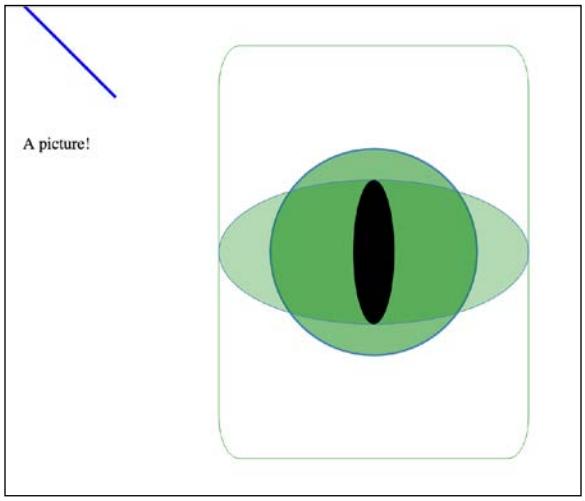
```
svg.append('ellipse')
  .attr({cx: 350,
    cy: 250,
    rx: 150,
    ry: 70,
    fill: 'green',
    'fill-opacity': 0.3,
    stroke: 'steelblue',
    'stroke-width': 0.7
  });
```

We added an `ellipse` element and defined some well-known attributes. The ellipse shape needs a central point (`(cx, cy)`) and two radii (`rx` and `ry`). Setting a low `fill-opacity` attribute makes the circle visible under the ellipse:

That's nice, but we can make it more interesting using the following code:

```
svg.append('ellipse')
  .attr({cx: 350,
    cy: 250,
    rx: 20,
    ry: 70
  });
```

The only trick here is that `rx` is smaller than `ry`, creating a vertical ellipse. Lovely!



A strange green eye with a random blue line is staring at you, all thanks to the manual addition of basic SVG elements to the canvas and the defining of some attributes.

The generated SVG looks as follows in XML form. You can see the same by right-clicking on the image and going to **Inspect Element**, which will select the element in Developer Tools:

```
<svg width="1000" height="1008">
    <g width="908" height="958">
        <text x="10" y="150" text-anchor="start">A picture!</text>
        <line x1="10" y1="10" x2="100" y2="100" stroke="blue" stroke-
width="3"/>
        <rect x="200" y="50" width="300" height="400" stroke="green"
stroke-width="0.5" fill="white" rx="20" ry="40"/>
        <circle cx="350" cy="250" r="100" fill="green" fill-
opacity="0.5" stroke="steelblue" stroke-width="2"/>
        <ellipse cx="350" cy="250" rx="150" ry="70" fill="green" fill-
opacity="0.3" stroke="steelblue" stroke-width="0.7"/>
```

```
        <ellipse cx="350" cy="250" rx="20" ry="70"/>
    </g>
</svg>
```

Yeah, I wouldn't want to write that by hand either!

But you can see all the elements and attributes we added before. Being able to look at an image file and understand what's going on might come in handy someday. It's certainly cool. Usually, when you open an image in a text editor, all you get is binary gobbledygook.

Now, I know I mentioned earlier that polylines and polygons are also basic SVG elements. The only reason I'm leaving off the explanation of these basic elements is that with D3, we have some great tools to work with them. Trust me, you don't want to do them manually.

Transformations

Before jumping onto more complicated things, we'll have to look at transformations.

Without going into too much mathematical detail, it suffices to say that transformations, as used in SVG, are affine transformations of coordinate systems used by shapes in our drawing. The beautiful thing is that they can be defined as matrix multiplications, making them very efficient to compute.

But unless your brain is made out of linear algebra, using transformations as matrices can get very tricky. However, SVG helps us out by coming with a set of predefined transformations, namely `translate()`, `scale()`, `rotate()`, `skewX()`, and `skewY()`.

According to Wikipedia, an affine transformation is any transformation that preserves points, straight lines, and planes, while keeping sets of parallel lines parallel. They don't necessarily preserve distances but do preserve ratios of distances between points on a straight line. This means that if you take a rectangle, you can use affine transformations to rotate it, make it bigger, and even turn it into a parallelogram; however, no matter what you do, it will never become a trapezoid.

Computers handle transformations as matrix multiplication because any sequence of transformations can be collapsed into a single matrix. This means they only have to apply a single transformation that encompasses your sequence of transformations when drawing the shape, which is handy.

We will apply transformations with the `transform` attribute. We can define multiple transformations that are applied in order. The order of operations can change the result. You'll notice this in the following examples.

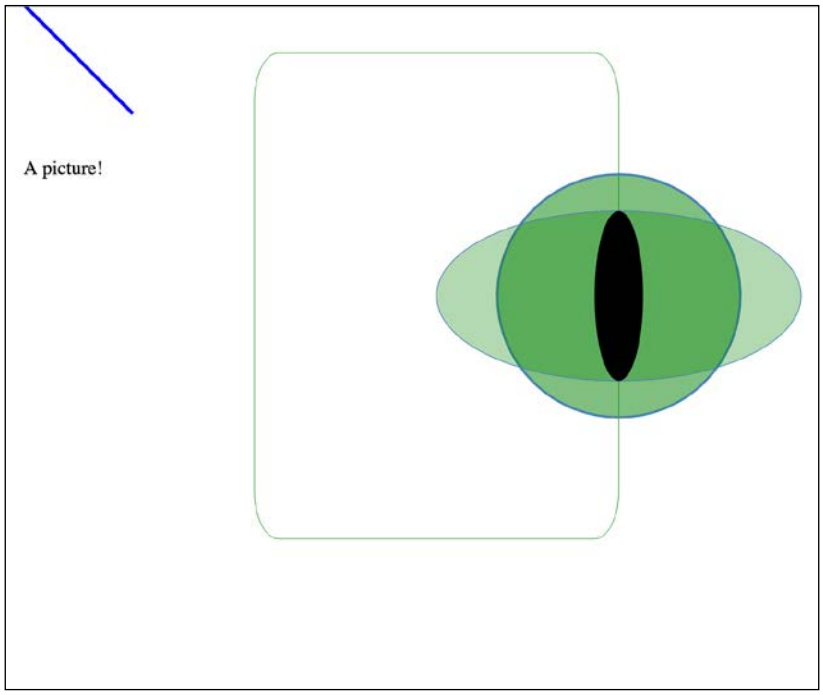
Let's move our eye to the edge of the rectangle:

```
svg.selectAll('ellipse, circle')
  .attr('transform', 'translate(150, 0)');
```

We selected everything our eye is made of (two ellipses and a circle) and then applied the `translate` transformation. It moved the shape's origin along the `(150, 0)` vector, moving the shape 150 pixels to the right and 0 pixels down.

If you try moving it again, you'll notice that new transformations are applied according to the original state of the shape. That's because there can only be one `transform` attribute per shape.

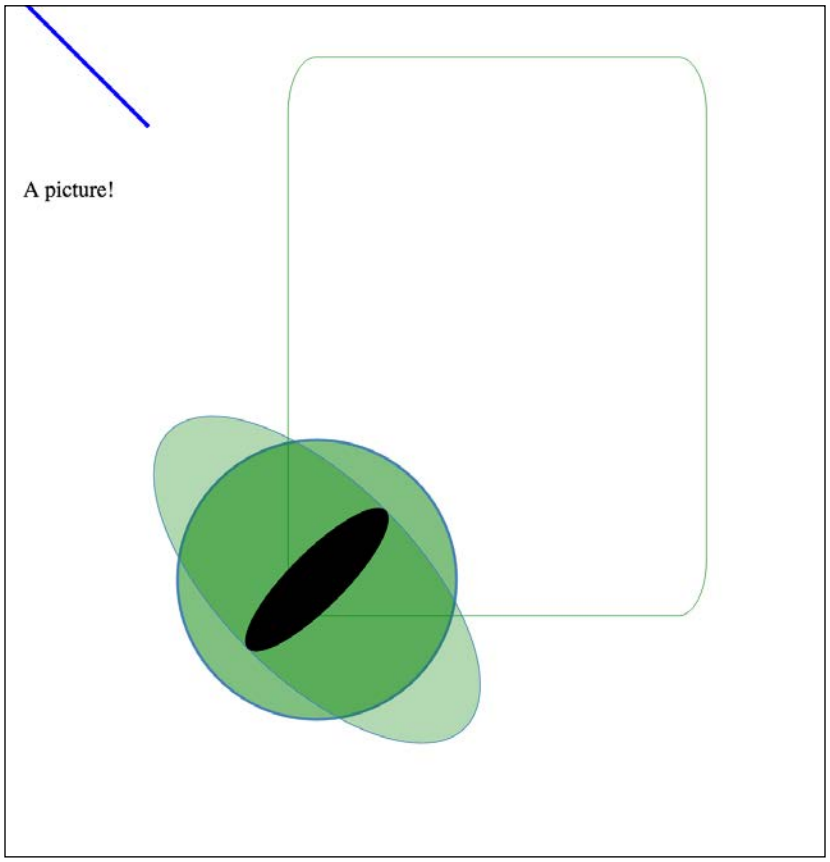
Our picture looks like what is shown here:



Let's rotate the eye by 45 degrees:

```
svg.selectAll('ellipse, circle')
  .attr('transform', 'translate(150, 0) rotate(45)');
```

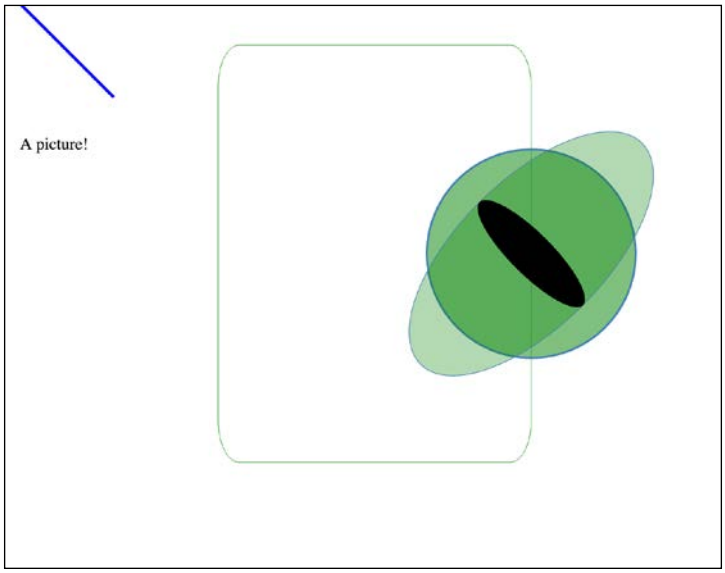
That's not what we wanted at all!



What tricked us is that rotations happen around the origin of the entire image and not the shape. We have to define the axis of rotation ourselves:

```
svg.selectAll('ellipse, circle')
  .attr('transform', 'translate(150, 0) rotate(-45, 350, 250)');
```

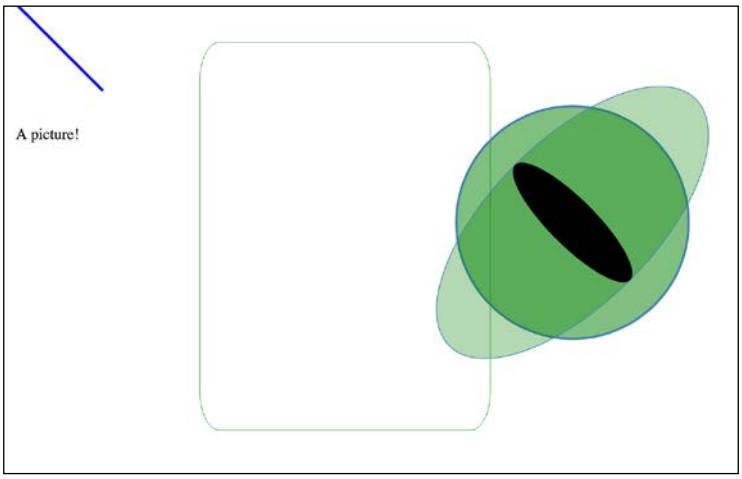
By adding two more arguments to `rotate()`, we defined the rotation axis and achieved the desired result:



Let's make the eye a little bigger with a `Scale()` transformation:

```
svg.selectAll('ellipse, circle')
  .attr('transform', 'translate(150, 0) rotate(-45, 350, 250)
scale(1.2)');
```

This will make our object 1.2 times bigger along both the axes; two arguments would have scaled by different factors along the *x* and *y* axes.

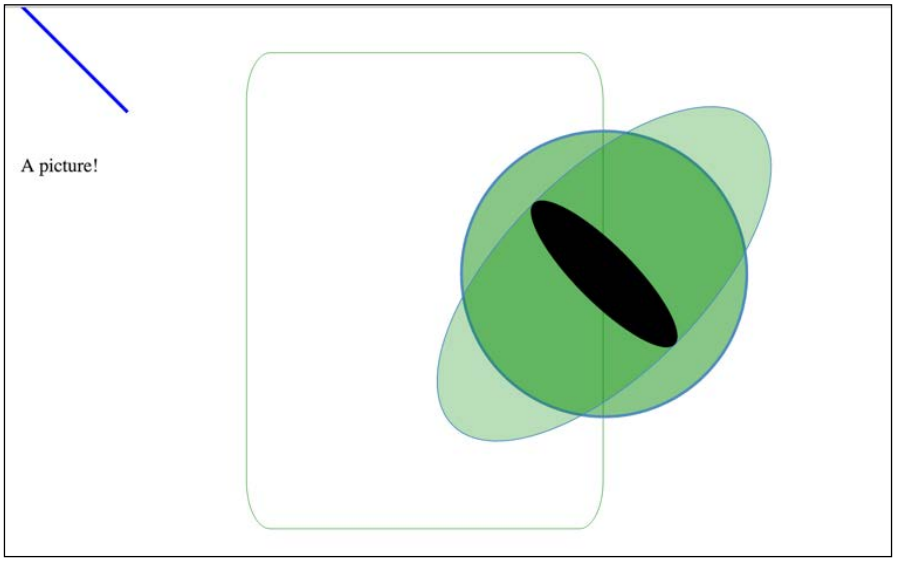


Once again, we have pushed the position of the eye because scaling is anchored at the zeroth point of the whole image. We have to use another translate operation to move it back. But the coordinate system we're working on is now rotated by 45 degrees and scaled. This makes things tricky. We need to translate between the two coordinate systems to move the eye correctly. To move the eye 70 pixels to the left, we have to move it along each axis by `70*sqrt(2)/2` pixels, which is the result of the cosine and sine at an angle of 45 degrees.

But that's just messy. The number looks funny, and we've worked way too much for something so simple. Let's change the order of operations instead:

```
svg.selectAll('ellipse, circle')
  .attr('transform',
    'translate(150, 0) scale(1.2) translate(-70, 0) rotate(-45, '
+
    (350/1.2) + ', ' + (250/1.2) + ')');
```

Much better! We get exactly what we wanted:



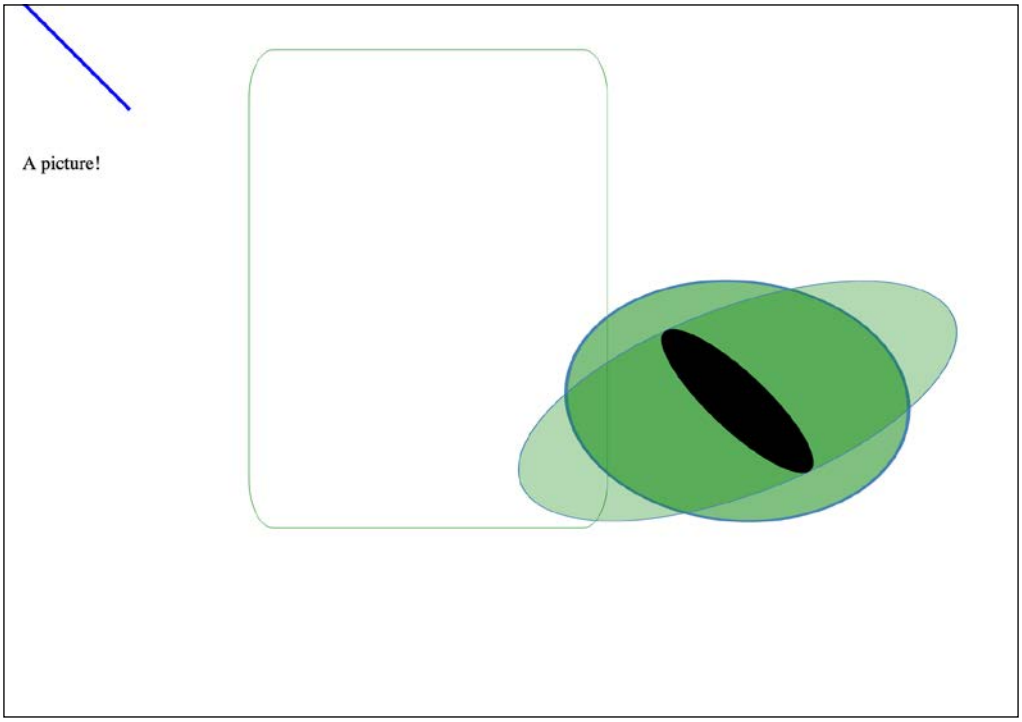
A lot has changed, so let's take a look at it.

First, we translate to our familiar position and then scale by `1.2`, pushing the eye out of position. We fix this by translating back to the left by `70` pixels and then performing the 45-degree rotation, making sure to divide the pivot point by `1.2`.

There's one more thing we can do to the poor eye; skew it. Two skew transformations exist: `skewX` and `skewY`. Both skew along their respective axis:

```
svg.selectAll('ellipse, circle')
  .attr('transform',
    `translate(150, 0) scale(1.2) translate(-70, 0) rotate(-45,
${350/1.2}, ${250/1.2}) skewY(20)`);
```

We've just bolted `skewY(20)` to the end of the `transform` attribute:



We have once more destroyed our careful centering; fixing this is left as an exercise for the reader

All said, transformations really are just matrix multiplications. In fact, you can define any transformation you want with the `matrix()` function. I suggest taking a look at exactly what kind of matrix produces each of the preceding effects. The W3C specification is available at `http://www.w3.org/TR/SVG/coords.html#EstablishingANewUserSpace` can help.

Using paths

Path elements define outlines of shapes that can be filled, stroked, and so on. They are generalizations of all other shapes and can be used to draw nearly anything.

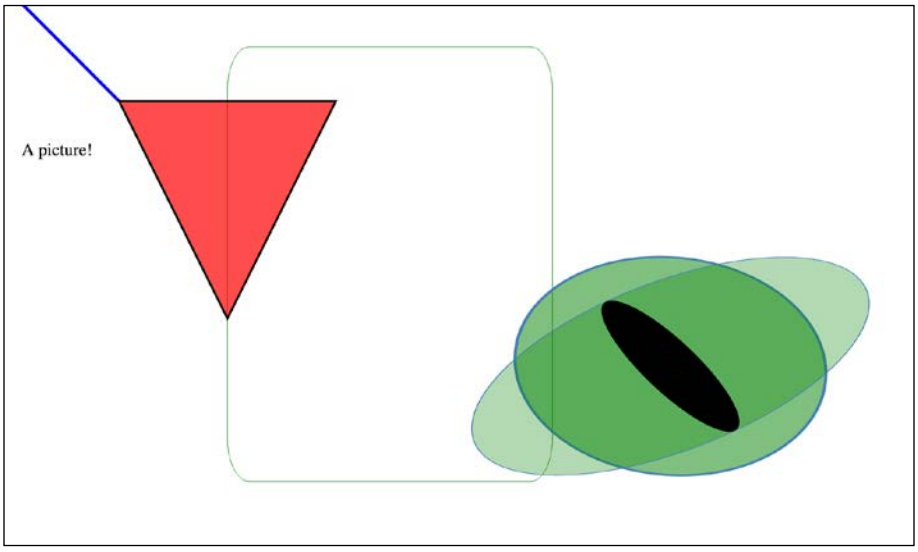
Most of the path's magic stems from the `d` attribute; it uses a mini language of three basic commands:

- `M`, meaning `moveto`
- `L`, meaning `lineto`
- `Z`, meaning `closepath`

To create a rectangle, we might write something as follows:

```
svg.append('path')
  .attr({d: 'M 100 100 L 300 100 L 200 300 z',
    stroke: 'black',
    'stroke-width': 2,
    fill: 'red',
    'fill-opacity': 0.7});
```

We appended a new element to our `svg` and then defined some attributes. The interesting bit is the `d` attribute, with the value `M 100 100 L 300 100 L 200 300 z`. Breaking this down, you can see that we moved to `(100, 100)`, drew a line on `(300, 100)`, drew another line on `(200, 300)`, and then closed the path:



The power of paths doesn't stop there though. Commands beyond the `M, L, Z` combination give us tools to create curves and arcs. However, creating complex shapes by hand is beyond tediousness.

D3 comes with some helpful path generator functions that take JavaScript and turn it into path definitions. We'll be looking at them next.

Our image is getting pretty crowded, so let's restart the environment.

To start things off, we'll draw the humble `sine` function. Once again, we begin by preparing the drawing area. Chuck all the code from the last section into another function in `chapter2.js`, call it `myWeirdSVGDrawingMyWeirdSVGDrawing` or something like that, and return the default function to the following state:

```
export default function() {
  let chart = new BasicChart();
  let svg = chart.svg;
}
```

Previously we called the `BasicChart` function's constructor and then immediately took its `chart` property, which is the SVG group element we've done all our work in up to now. However, `BasicChart` also gives us a bunch of more information about our chart that we'll use momentarily, which is why we've initially assigned it to the `chart` local variable.

Next, we need some data, which we'll generate using the built-in JavaScript `sine` function, `Math.sin`:

```
let sine = d3.range(0,10).map(
  (k) => [0.5*k*Math.PI, Math.sin(0.5*k*Math.PI)]
);
```

Using `d3.range(0,10)` gives us a list of integers from zero to nine. We map over them and turn each into a tuple, actually a 2-length array representing the maxima, minima, and zeros of the curve. You might remember from your math class that sine starts at `(0,0)`, then goes to `(Pi/2, 1)`, `(Pi, 0)`, `(3Pi/2, -1)`, and so on.

We'll feed these as data into a path generator.

Path generators are really the meat of D3's magic. We'll discuss the gravy of the magic in *Chapter 5*, *Layouts – D3's Black Magic*. They are essentially functions that take some data (joined to elements) and produce a path definition in SVG's path mini language. All path generators can be told how to use our data. We also get to play with the final output a great deal.

Line

To create a line, we use the `d3.svg.line()` generator and define the `x` and `y` accessor functions. Accessors tell the generator how to read the `x` and `y` coordinates from data points.

We begin by defining two scales. Scales are functions that map from a domain to a range; we'll talk more about them in the next chapter:

```
  let x =
      d3.scale.linear()
          .range(
              [0, chart.width / 2 - (chart.margin.left +
chart.margin.right)])
          .domain(d3.extent(sine, (d) => d[0]));

  let y =
      d3.scale.linear()
          .range(
              [chart.height / 2 - (chart.margin.top +
chart.margin.bottom), 0])
          .domain([-1, 1]);
```

Now we get to define a simple path generator:

```
  let line = d3.svg.line()
    .x((d) => x(d[0]))
    .y((d) => y(d[1]));
```

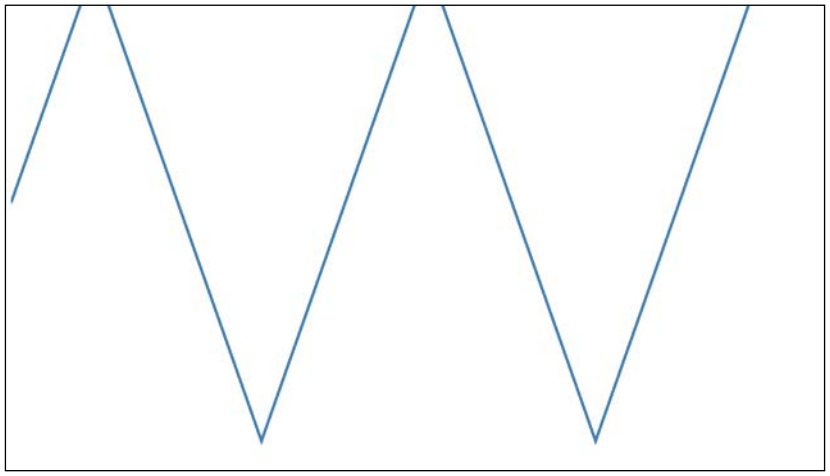
It is just a matter of taking the basic line generator and attaching some accessors to it. We told the generator to use our `x` scale on the first element and the `y` scale on the second element of every tuple. By default, it considers our dataset as a collection of arrays defining points directly so that `d[0]` is `x` and `d[1]` is `y`.

All that's left now is drawing the actual line:

```
let g = svg.append('g');
g.append('path')
  .datum(sine)
  .attr('d', line)
  .attr({stroke: 'steelblue',
    'stroke-width': 2,
    fill: 'none'});
```

Append a path and add the sine data using `.datum()`. Using this instead of `.data()` means that we can render the function as a single element instead of creating a new line for every point. We let our generator define the `d` attribute. The rest just makes things visible.

Our graph looks as follows:



If you look at the generated code, you'll see this sort of gobbledygook:

```
d="M0,220L48.888888888888886,0L97.77777777777777,219.99999999999994L14
6.66666666666666,440L195.55555555555554,220.00000000000006L244.4444444
4444446,0L293.3333333333333,219.99999999999991L342.22222222222223,440L
391.1111111111111,220.00000000000009L440,0"
```

See! I told you. Nobody wants to write that by hand!

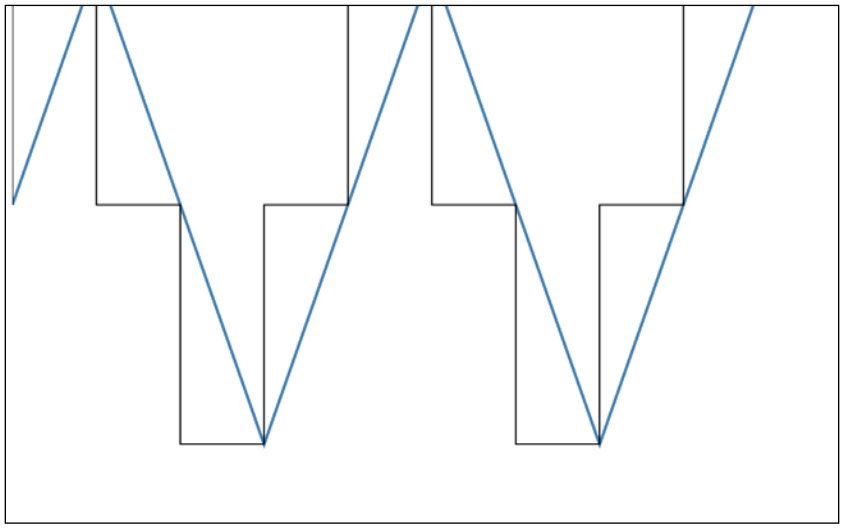
It's a very jagged `sine` function we've got here, nothing similar to what the math teacher used to draw in high school. We can make it better with interpolation.

Interpolation is the act of guessing where unspecified points of a line should appear by considering the points we do know. By default, we've used the linear interpolator that just draws straight lines between points.

Let's try something else:

```
g.append('path')
  .datum(sine)
  .attr('d', line.interpolate('step-before'))
  .attr({stroke: 'black',
  'stroke-width': 1,
  fill: 'none'});
```

It is the same code as before, but we used the `step-before` interpolator and changed the styling to produce this:



D3 offers 12 line interpolators in total, which I am not going to list here. You can look at them on the official wiki page at `https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-line_interpolate`.

I suggest trying out all of them to get a feel of what they do.

Area

An area is the colored part between two lines, a polygon really.

We define an area similar to how we define a line, so we take a path generator and tell it how to use our data. For a simple horizontal area, we have to define one `x` accessor and two `y` accessors, `y0` and `y1`, for both the bottom and the top.

We'll compare different generators side by side, so let's add a new graph, which we'll render inside the same SVG element:

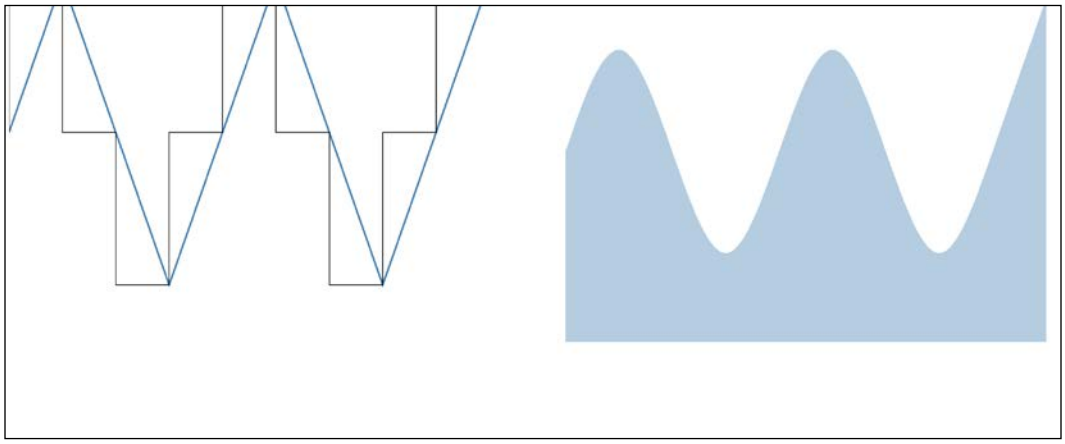
```
let g2 = svg.append('g')
  .attr('transform',
    'translate(' + (chart.width / 2 +
    (chart.margin.left + chart.margin.right)) +
    ', ' + chart.margin.top + ')');
```

Now we define an area generator and draw an area:

```
let area = d3.svg.area()
  .x((d) => x(d[0]))
  .y0(chart.height / 2)
  .y1((d) => y(d[1]))
  .interpolate('basis');

g2.append('path')
  .datum(sine)
  .attr('d', area)
  .attr({fill: 'steelblue', 'fill-opacity': 0.4});
```

We took a vanilla `d3.svg.area()` path generator and told it to get the coordinates through the `x` and `y` scales we defined earlier. The basis interpolator will use a B-spline to create a smooth curve from our data.

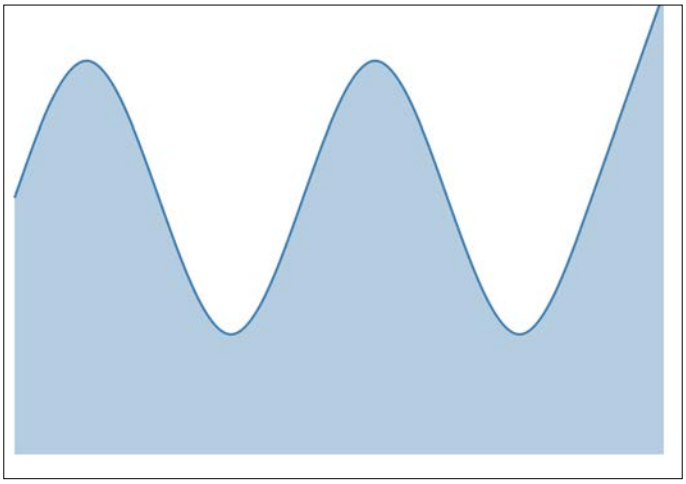
To draw the bottom edge, we defined `y0` as the bottom of our graph and produced a colored sine approximation:



Areas are often used together with lines that make an important edge stand out. Let's try that:

```
g2.append('path')
  .datum(sine)
  .attr('d', line.interpolate('basis'))
  .attr({stroke: 'steelblue',
    'stroke-width': 2,
    fill: 'none'});
```

We could reuse the same line generator as before; we just need to make sure that we use the same interpolator as that for the area. This way, the image looks much better:



Arc

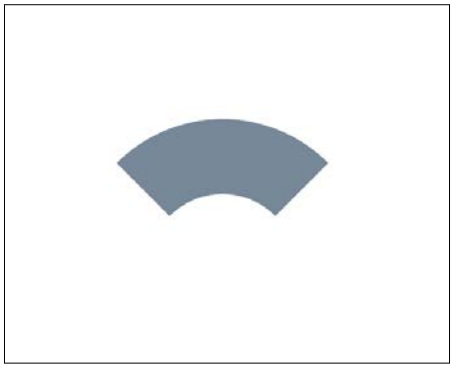
An arc is a circular path with an inner radius and an outer radius, going from one angle to another. They are often used for pie and donut charts.

Everything works as before; we just tell the base generator how to use our data. The only difference is that this time the default accessors expect named attributes instead of two-value arrays we've gotten used to.

Let's draw an arc:

```
    let arc = d3.svg.arc();
    let g3 = svg.append('g')
      .attr('transform', 'translate(' +
        (chart.margin.left + chart.margin.right) +
        ',' +( chart.height / 2 + (chart.margin.top +
chart.margin.bottom) ) +
    ')');

    g3.append('path')
      .attr('d', arc({outerRadius: 100,
        innerRadius: 50,
        startAngle: -Math.PI*0.25,
        endAngle: Math.PI*0.25}))
      .attr('transform', 'translate(150, 150)')
      .attr('fill', 'lightslategrey');
```

This time, we were able to get away with using the default `d3.svg.arc()` generator. Instead of using data, we calculated the angles manually and also nudged the arc towards the center:



Huzzah, a simple arc. Rejoice!!

Even though SVG normally uses degrees, the start and end angles use radians. The zero angle points upwards towards the 12 o'clock position, with negative values going anticlockwise and positive values going the other way. With every `2Pi`, we come back to zero.

Symbol

Sometimes when visualizing data, we need a simple way to mark data points. That's where symbols come in—tiny glyphs used to distinguish between data points.

The `d3.svg.symbol()` generator takes a type accessor and a size accessor, and leaves the positioning to us. We are going to add some symbols to our area chart showing where the function goes when it crosses zero.

As always, we start with a path generator:

```
let symbols = d3.svg.symbol()
  .type((d) => d[1] > 0 ? 'triangle-down' : 'triangle-up')
  .size((d, i) => i%2 ? 0 : 64);
```

We've given the `d3.svg.symbol()` generator a type accessor, telling it to draw a downward-pointing triangle when the `y` coordinate is positive and an upward one when it is not positive. This works because our sine data isn't mathematically perfect due to `Math.PI` not being infinite and due to floating-point precision; we get infinitesimal numbers close to zero whose "signedness" depends on whether the argument provided to `` `Math.sin` `` is slightly less or slightly more than the perfect point for `sin=0`.

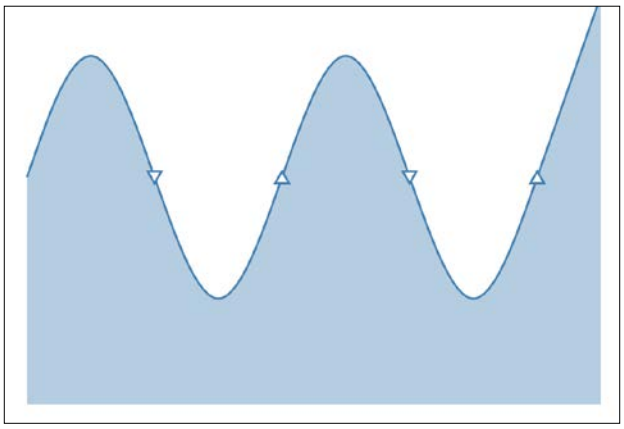
The size accessor tells `symbol()` how much area each symbol should occupy. Because every other data point is close to zero, we hide the others with an area equal to zero.

Now we can draw some symbols:

```
g2.selectAll('path')
  .data(sine)
  .enter()
  .append('path')
  .attr('d', symbols)
  .attr({stroke: 'steelblue', 'stroke-width': 2, fill: 'white'})
  .attr('transform', (d) => `translate(${x(d[0])},${y(d[1])})`);
```

You'll notice that I haven't used the shiny new ES2016 backtick template string syntax before now in this chapter, even though it makes `translate` strings much more compact. This is mainly because the additional dollar sign and curly brackets can sometimes make these less readable, and it helps to present them in as basic a fashion as possible initially. From here on, we'll use the backtick template syntax, however.

Go through the data, append a new path for each entry and turn it into a symbol moved into position. The result looks like this:



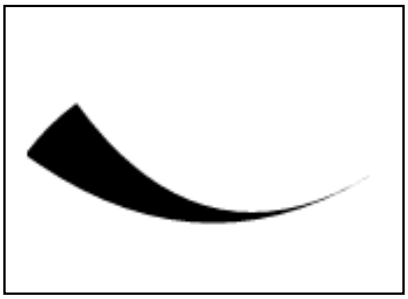
You can see other available symbols by printing `d3.svg.symbolTypes` or visiting `https://github.com/mbostock/d3/wiki/SVG-Shapes#symbol_type`.

Chord

Good news! We are leaving the world of simple charts and entering the world of magic.

Chords are most often used to display relations between group elements when arranged in a circle. They use quadratic Bézier curves to create a closed shape connecting two points on an arc.

If you don't have a strong background in computer graphics, this tells you nothing. A basic chord looks similar to half a villain's moustache:



To draw that, we use the following piece of code:

```
g3.append('g')
.selectAll('path')
.data([{
  source: {
    radius: 50,
    startAngle: -Math.PI*0.30,
    endAngle: -Math.PI*0.20
  },
  target: {
    radius: 50,
    startAngle: Math.PI*0.30,
    endAngle: Math.PI*0.30}
  }])
  .enter()
  .append('path')
  .attr('d', d3.svg.chord());
```

This code adds a new grouping element, defines a dataset with a single datum, and appends a path using the default `d3.svg.chord()` generator for the `d` attribute.

The data itself works fine with the default accessors, so we can just hand it off to `d3.svg.chord()`. The `source` defines where the chord begins and `target` defines where it ends. Both are fed to another set of accessors, specifying the arc's radius, start angle, and end angle. As with the arc generator, angles are defined using radians.

Let's make up some data and draw a chord diagram:

```
let data = d3.zip(d3.range(0, 12), d3.shuffle(d3.range(0, 12)));
let colors = ['linen', 'lightsteelblue', 'lightcyan', 'lavender',
'honeydew', 'gainsboro'];
```

Nothing too fancy. We defined two arrays of numbers, shuffled one, and merged them into an array of pairs. We will look at the details in the next chapter, but it suffices to say that `d3.range` gives you an array of values between two numbers, `d3.shuffle` randomizes the order of an array, and `d3.zip` gives you an array of arrays. We then defined some colors:

```
let chord = d3.svg.chord()
  .source((d) => d[0])
  .target((d) => d[1])
  .radius(150)
  .startAngle((d) => -2*Math.PI*(1/data.length)*d)
  .endAngle((d) => -2*Math.PI*(1/data.length)*((d-1)%data.length));
```

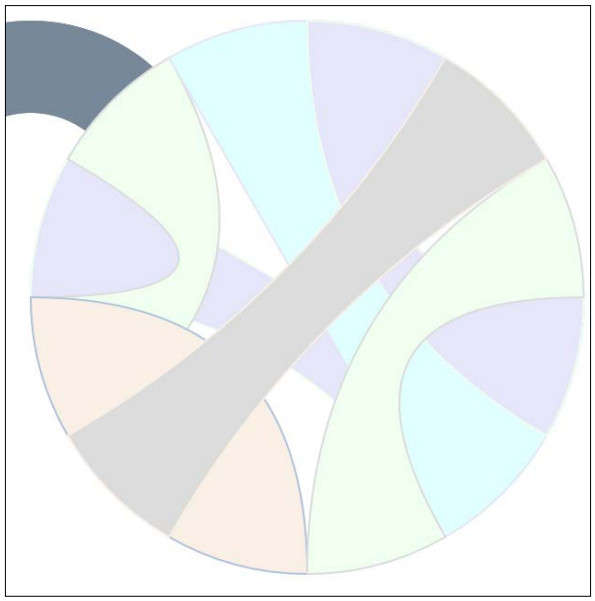
All of this just defines the generator. We're going to divide a circle into sections and connect random pairs with chords.

The `.source()` and `.target()` accessors tell us that the first item in every pair is the source and the second is the target. For `startAngle`, we remember that a full circle is `2Pi` and divide it by the number of sections. Finally, to pick a section, we multiply by the current datum. The `endAngle` accessor is more of the same, except with the datum offset by 1:

```
g3.append('g')
  .attr('transform', 'translate(300, 200)')
  .selectAll('path')
  .data(data)
  .enter()
  .append('path')
  .attr('d', chord)
  .attr('fill', (d, i) => colors[i%colors.length])
  .attr('stroke', (d, i) => colors[(i+1)%colors.length]);
```

To draw the actual diagram, we create a new grouping, join the dataset, and then append a path for each datum. We use the chord generator from earlier to give each chord a shape, draw chords from each source to target, and add some color for fun.

The end result changes with every refresh, but it looks something like this:



Diagonal

The diagonal generator creates cubic Bézier curves—smooth curves between two points. It is very useful for visualizing trees with a node-link diagram.

Once again, the default accessors assume that your data is a dictionary with keys named after the specific accessor. You need source and target, which are fed into projection. It then projects Cartesian coordinates into whatever coordinate space you like. By default, it just returns Cartesian coordinates.

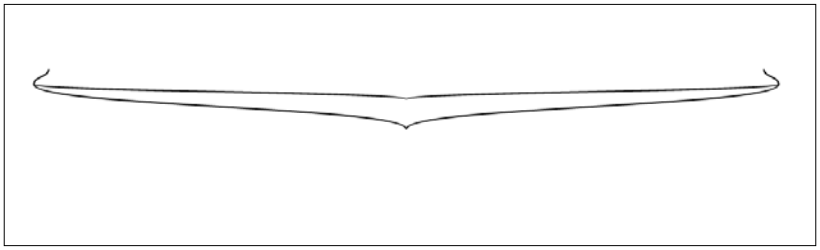
Let's draw a moustache. Trees are hard without `d3.layouts` and we'll do those later:

```
let g4 = svg.append('g')
  .attr('transform', `translate(${chart.width/2},${chart.height/2})`);

let moustache = [
  {source: {x: 250, y: 100}, target: {x: 500, y: 90}},
  {source: {x: 500, y: 90}, target: {x: 250, y: 120}},
  {source: {x: 250, y: 120}, target: {x: 0, y: 90}},
  {source: {x: 0, y: 90}, target: {x: 250, y: 100}},
  {source: {x: 500, y: 90}, target: {x: 490, y: 80}},
  {source: {x: 0, y: 90}, target: {x: 10, y: 80}}
];
```

We started off with a fresh graph on our drawing area and defined some data that should create a sweet 'stache!

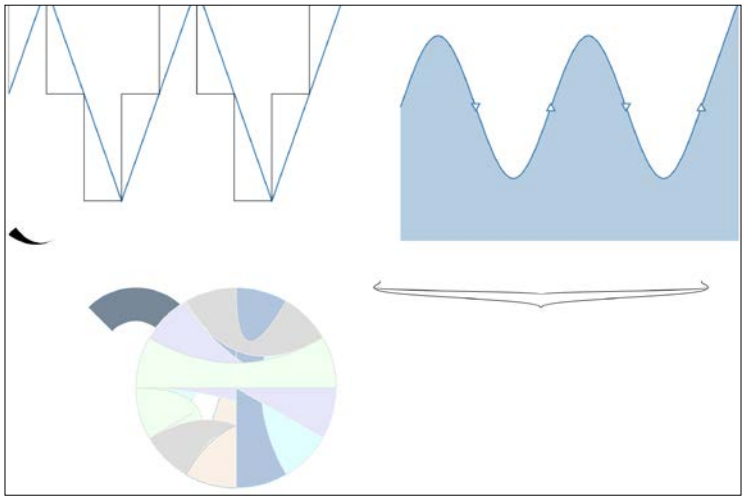
```
g4.selectAll('path')
  .data(moustache)
  .enter()
  .append('path')
  .attr('d', d3.svg.diagonal())
  .attr({stroke: 'black', fill: 'none'});
```

The rest is just a matter of joining data to our drawing and using the `d3.svg.diagonal()` generator for the `d` attribute:



Okay, it's a bit Daliesque. It may be, but it doesn't really look anything like a moustache. That's because the tangents that define how Bézier curves bend are tweaked to create good-looking fan-out in tree diagrams. Unfortunately, D3 doesn't give us a simple way of changing these, and manually defining Bézier curves through SVG's path mini language is tedious at best.

Either way, we have created a side-by-side comparison of path generators:



Axes

But we haven't done anything useful with our paths and shapes yet. One way we can do so is by using lines and text to create graph axes. It would be tedious though, so D3 makes our lives easier with axis generators. They take care of drawing a line, putting on some ticks, adding labels, evenly spacing them, and so on.

A D3 axis is just a combination of path generators configured for awesomeness. All we have to do for a simple linear axis is create a scale and tell the axis to use it. That's it!

In D3, it's worth remembering that a **scale** is a function that maps an input range to an output domain, whereas an **axis** is merely a visual representation of a scale.

For a more customized axis, we might have to define the desired number of ticks and specify the labels, perhaps something even more interesting. There are even ways to make circular axes.

We begin with a drawing area. Move all your code from your default function to another function named `funkyD3PathRenders`, and reset your default function so that it looks like this:

```
export default function() {
  let chart = new BasicChart();
  let svg = chart.chart;
}
```

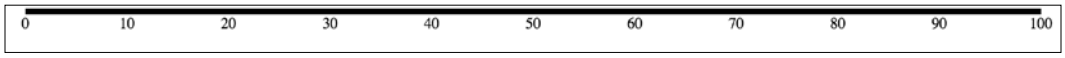
We also need a linear scale:

```
let x = d3.scale.linear()
  .domain([0, 100])
  .range([chart.margin.left, chart.width - chart.margin.right]);
```

Our axis is going to use the following to translate data points (domain) to coordinates (range):

```
let axis = d3.svg.axis()
  .scale(x);

let a = svg.append('g')
  .attr('transform', 'translate(0, 30)')
  .data(d3.range(0, 100))
  .call(axis);
```

We told the `d3.svg.axis()` generator to use our `x` scale. Then, we simply created a new grouping element, joined some data, and called the axis. It's very important to call the axis generator on all of the data at once so that it can handle appending its own element.



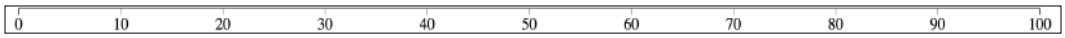
The result doesn't look good at all.

Axes are complex objects, so fixing this problem is convoluted without CSS, which comes in the next section.

For now, adding this code will be sufficient:

```
a.selectAll('path')
  .attr({fill: 'none',
    stroke: 'black',
    'stroke-width': 0.5});

a.selectAll('line')
  .attr({fill: 'none',
    stroke: 'black',
    'stroke-width': 0.3});
```

An axis is a collection of paths and lines; we give them some swagger and get a nice-looking axis in return:



If you play around with the amount, make sure that the scale's domain and the range's max value match, and you'll notice that axes are smart enough to always pick the perfect number of ticks.

Let's compare what the different settings do to axes. We're going to loop through several axes and render the same data.

Wrap your axis-drawing code in a loop by adding this line just above `svg.append('g')`. Don't forget to close the loop just after the last stroke-width:

```
axes.forEach(function (axis, i) {
```

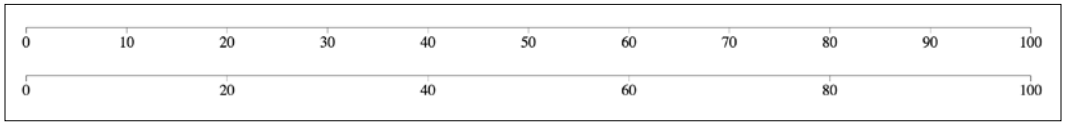
You should also change the `.attr('transform', …)` line to put each axis 50 pixels below the previous one:

```
.attr('transform', `translate(0,${i*50+chart.margin.top})`)
```

Now that's done, so we can start defining an array of axes:

```
let axes = [
  d3.svg.axis().scale(x),
  d3.svg.axis().scale(x).ticks(5)
];
```

Two for now: one is the plain vanilla version and the other will render with exactly five ticks:

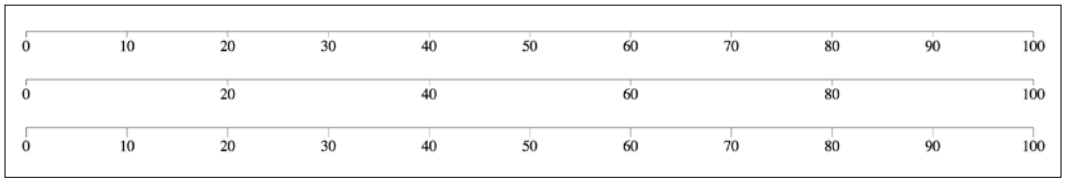


It worked! The axis generator figured out which ticks are best left off and relabeled everything without us doing much.

Let's add more axes to the array and see what happens:

```
d3.svg.axis().scale(x).tickSubdivide(3).tickSize(10, 5, 10)
```

With `.tickSubdivide()`, we instruct the generator to add some subdivisions between the major ticks; `.tickSize()` tells it to make the minor ticks smaller. The arguments are `major`, `minor`, and `end` tick size:

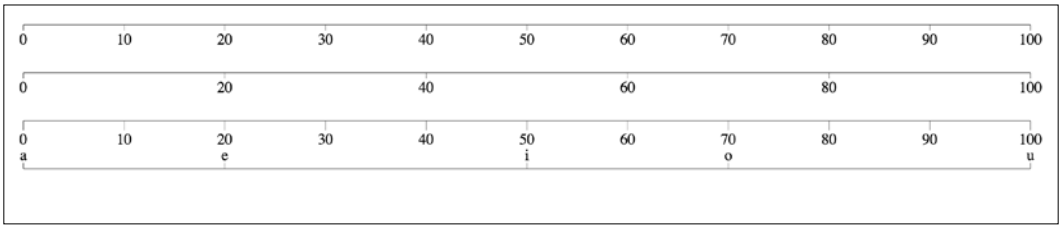


For our final trick, let's define some custom ticks and place them above the axis. We'll add another axis to the array:

```
    d3.svg.axis().scale(x).tickValues([0, 20, 50, 70, 100])
      .tickFormat((d, i) => ['a', 'e', 'i', 'o',
'u'][i]).orient('top')
```

Three things happen here: `.tickValues()` exactly defines which values should have a tick, `.tickFormat()` specifies how to render the labels, and finally `.orient('top')` puts the labels above their axis.

You might have guessed that the default orient is `bottom`. For a vertical axis, you can use `left` or `right`, but don't forget to assign an appropriate scale:



CSS

Cascading Style Sheets (**CSS**) have been with us since 1996, making them one of the oldest staples of the Web, even though they reached widespread popularity only with the tables versus CSS wars of the early 2000s.

You're probably familiar with using CSS for styling HTML. So, this section will be a refreshing breeze after all that SVG stuff.

My favorite thing about CSS is its simplicity; refer to the following code:

```
selector {
  attribute: value;
}
```

And that's it! Everything you need to know about CSS in three lines!

The selectors can get fairly complicated and are beyond the scope of this book. I suggest looking around the Internet for a good guide. We just need to know some basics:

- `path`: This selects all the `<path>` elements
- `.axis`: This selects all the elements with a `class="axis"` attribute
- `.axis line`: This selects all the `<line>` elements that are children of `class="axis"` elements
- `.axis, line`: This selects all the `class="axis"` and `<line>` elements

Right now, you might be thinking, "Oh hey! That's the same as the selectors for D3 selections." Yes! It is exactly the same. D3 selections are a subset of CSS selectors.

We can invoke CSS with D3 in three ways:

- Define a class attribute with the `.attr()` method, which can be brittle
- Use the `.classed()` method, which is the preferred way of defining classes
- Define styling directly with the `.style()` method

Let's improve the axes example from before and make the styling less cumbersome.

Go to `index.css` and replace it with the following code:

```
.axis path,
.axis line {
  fill: none;
  stroke: black;
  stroke-width: 1px;
  shape-rendering: crispEdges;
}

.axis text {
  font-size: 11px;
}

.axis.dotted line,
.axis.dotted path {
  stroke-dasharray: 0.9;
}
```

Now add `require('./index.css')` to your default function to load it in your HTML file. You can also use `<link>` tags in your HTML, but again, that's boring.

Modifying SVG via CSS is very similar to changing SVG attributes directly via D3. We used `stroke` and `fill` to define the shape of the line and set `shape-rendering` to `crispEdges`. This will make things better.

We've also defined an extra type of axis with dotted lines using the SVG `stroke-dasharray` property.

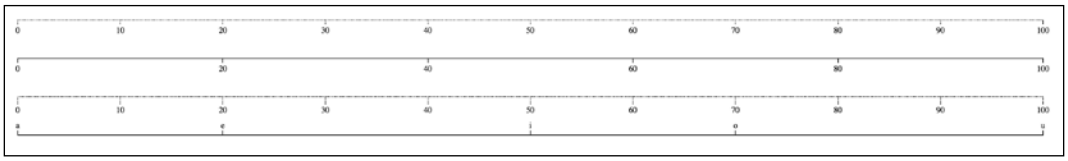
You can do more than just dotted lines with `stroke-dasharray`. You can also have dashed lines and animate the property to get a trailing or growing line effect. Visit this article on the Mozilla Developer Network for more examples:

`https://mdn.io/stroke-dasharray`

Now we amend the drawing loop from earlier to look like this:

```
axes.forEach(function (axis, i) {
  svg.append('g')
  .classed('axis', true)
  .classed('dotted', i%2 == 0)
  .attr('transform', `translate(0, ${i*50+(chart.margin.top)})`)
  .data(d3.range(0, 100))
  .call(axis);
});
```

None of that foolishness with specifying the same looks five times in a row. Using the `.classed()` function, we add the `axis` class to each axis, and every second axis is red. The `.classed()` adds the specified class if the second argument is true and removes it otherwise:



Colors

Beautiful visualizations often involve color beyond the basic names you can think of off the top of your head. Sometimes, you want to play with colors depending on what the data looks like.

D3 has us covered with a slew of functions devoted to manipulating colors in four popular color spaces: *RGB*, *HSL*, *HCL*, and *L*a*b*. The most useful for us are going to be **red green blue** (**RGB**) and **hue saturation lightness** (**HSL**), which is secretly just another way of looking at RGB. Either way, all color spaces use the same functions, so you can use what fits your needs best.

To construct an RGB color, we use `d3.rgb(r, g, b)`, where `r`, `g`, and `b` specify the channel values for red, green, and blue, respectively. We can also replace the triplet with a simple CSS color argument. Then we get to make the color darker or brighter, which is much better than shading by hand.

Time to play with colors in a fresh environment! We'll draw two color wheels with their brightness changing from the center towards the outside.

As always, we begin with some variables and a drawing area. Rename your last bunch of stuff to `axisDemo` and set up a new default function as follows:

```
export default function() {
  let chart = new BasicChart();
  let svg = chart.chart;

  let rings = 15;
  let slices = 20;
}
```

The main variable henceforth will be `rings`; it will tell the code how many levels of brightness we want. The number of pieces in each wheel is represented by the `slices` variable. We also need some basic colors and a way to calculate angles:

```
let colors = d3.scale.category20b();
let angle = d3.scale.linear().domain([0, slices]).range([0,
2*Math.PI]);
```

The `colors` is technically a scale, but we'll use it as data. The `.category20b` is one of the four predefined color scales that come with D3—an easy way to get a list of well-picked colors. Although you can set the number of pieces by changing the `slices` values, note that the maximum is `20`, because we only have that many colors in the `.category20b` scale.

To calculate angles, we're using a linear scale that maps the `[0, slices]` domain to a full circle (`[0, 2*pi]`).

Next, we need an arc generator and two data accessors to change the color shade for every ring:

```
let arc = d3.svg.arc()
  .innerRadius((d) => d*50/rings)
  .outerRadius((d) => 50+d*50/rings)
  .startAngle((d, i, j) => angle(j))
  .endAngle((d, i, j) => angle(j+1));

let shade = {
  darker: (d, j) => d3.rgb(colors(j)).darker(d/rings),
  brighter: (d, j) => d3.rgb(colors(j)).brighter(d/rings)
};
```

The arc will calculate the inner and outer radii from a simple ring counter, and the angles will use the angle scale, which will automatically calculate the correct radian values. We're ultimately creating a bunch of concentric arcs in order to create a gradient feel. If you decrease the `rings` variable earlier to something like 3 or 4, you can get a better idea of what we're doing here. In the preceding data accessors, `d` is the current ring and `j` is the current slice. We feed the latter into our `angle` scale to get the relevant angles.

Since we're making two pictures, we can simplify the code by using two different shaders from a dictionary.

Each shader will take a `d3.rgb()` color from the colors scale and then darken or brighten it by the appropriate number of steps, depending on which ring it is drawing. Once again, the `j` argument tells us which arc section we're in and the `d` argument tells us which ring we're at.

Finally, we draw the two color wheels:

```
[
  [100, 100, shade.darker],
  [300, 100, shade.brighter]
].forEach(function (conf) {
  svg.append('g')
  .attr('transform', `translate(${conf[0]}, ${conf[1]})`)
  .selectAll('g')
  .data(colors.range())
  .enter()
  .append('g')
  .selectAll('path')
  .data((d) => d3.range(0, rings))
  .enter()
  .append('path')
  .attr('d', arc)
  .attr('fill', (d, i, j) => conf[2](d, j));
});
```

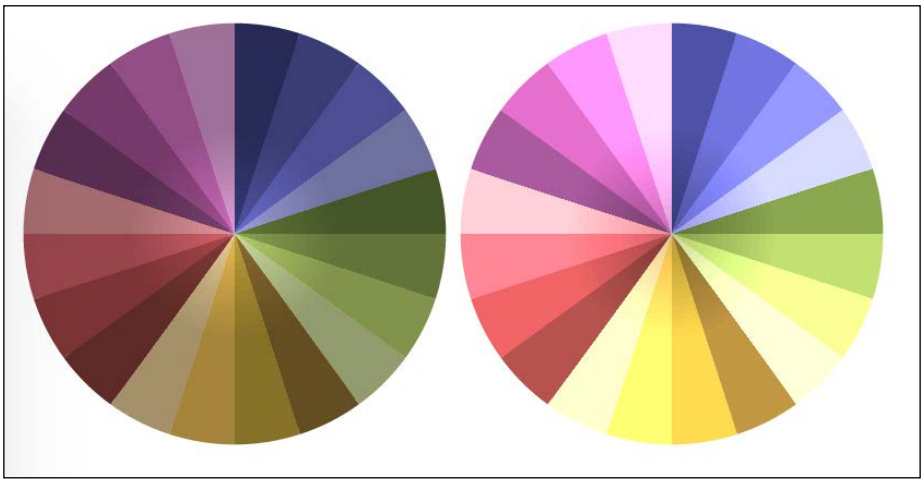
Wow! That's quite a bit of code.

We take two arrays with three values (also called a "triplet"), each defining the color wheel's position and which shader to use; then we call a function that draws a shiny, colorful circle with each.

For each circle, we append a `<g>` element and move it into position. Then we use `colors.range()` to get a full list of colors and join it as data. For every new color, we create another `<g>` element and select all the `<path>` elements it contains.

Here, things get magical. We join more data but just an array of numbers going from 0 to the number of rings this time. D3 remembers the first time we joined data and supplies it as the third argument, `j`. For every element in this array, we append a `<path>` element and use the `arc` generator we added earlier to draw the arc shape. Finally, we calculate the `fill` attribute with an appropriately shaded color.

The result looks as follows:



Depending on whether this is a black-and-white page, it might appear slightly more colorful than the preceding figure.

Our main trick was that joining a second dimension of data retains the knowledge of the first dimension via the third attribute supplied to the data accessors.

Summary

This was an intense chapter. Go have a nice beverage of some kind if you're still with me; you deserve it. If not, don't get too discouraged—that was a lot of material! We'll be using a lot of it throughout the rest of the book, and why any of this is useful will become gradually more apparent as we use some of these tools in practice.

You should now have a firm grasp of the basics that go into great visualizations.

We went through DOM manipulation and looked at SVG in great detail—everything from drawing shapes manually to path generators. Finally, we looked at CSS as a better alternative for making things pretty.

Everything we look at from now on is going to build on these basics, but you now have the tools needed to draw anything you can think of. The rest of this book just shows you more elegant ways of doing it.

3
Making Data Useful

At its core, D3 is a data manipulation library. We're going to take a look at making our datasets useful with both D3 and plain old JavaScript.

We start with a quick dive into functional programming to bring everyone up to speed. A lot of this will be self-evident if you use Haskell, Scala, or Lisp, or have already been writing JavaScript in functional style, but it's worth reviewing so that we can contrast it with the object-oriented style we use in our classes.

We will continue loading external data and taking a closer look at scales, and finish with some temporal and geographic data.

Thinking about data functionally

Due to the functional design of D3, we have to start thinking about our code and data with a functional mindset. This is fundamentally different from the object-oriented approach used by classes, which we've mainly used to give the basic structure to our code.

The good news is that JavaScript almost counts as a functional language; there are enough features for us to get the benefits of functional style, and it also provides enough freedom to do things imperatively or in an object-oriented way. The bad news is that unlike real functional languages, the environment gives no guarantee about our code.

Two projects that address this are Facebook's Flow and Microsoft's TypeScript projects, which allow the compilation of JavaScript using static types. Another is immutable.js, which allows the creation of immutable objects in JavaScript. These efforts go a great deal towards improving confidence in how data moves through our visualizations, in addition to improving our tooling.

We'll talk about Flow and TypeScript in *Chapter 8*, *Having Confidence in Your Visualizations*.

In this section, we'll go through the basics of functional-style coding and look at wrangling data so that it's easier to work with. If you want to try proper functional programming, I recommend looking at Haskell and *Learn You a Haskell for Great Good!*, which is free to read at `http://learnyouahaskell.com/`.

The idea behind functional programming is simple—compute by relying only on function arguments. It's simple, but the consequences are far reaching.

The biggest of them is that we don't have to rely on state, which in turn gives us referential transparency. This means that functions executed with the same parameters will always give the same results regardless of when or how they're called.

In practice, this means that we design the code and data flow, that is, get data as the input, execute a sequence of functions that pass changed data down the chain, and eventually get a result.

You've already seen this in previous examples, particularly in *Chapter 2*, *A Primer on DOM, SVG, and CSS*. Our dataset started and ended as an array of values. We performed some actions for each item and relied only on the current item when deciding what to do. We also had the current index so that we could cheat a little with an imperative approach by looking ahead and behind in the stream.

Built-in array functions

JavaScript comes with a slew of array manipulation functions. We'll focus on those that are more functional in nature—the iteration methods.

Map, reduce, and filter (or `Array.prototype.map`, `Array.prototype.reduce`, and `Array.prototype.filter` to be specific) are hugely useful for remodeling data. In fact, map/reduce is a core pattern in NoSQL databases, and the ability to parallelize these functions grants them a huge degree of scalability.

In the following examples, I will give the full names of the native array methods so as to differentiate them from D3's own filter and mapping methods. `Array.prototype.map` thus refers to the map method on the `Array` primitive's prototype.

But what's a "prototype"? JavaScript is a prototype-based language, which means that everything is effectively an object that inherits from another object, or its prototype. All arrays are descendants of the `Array` primitive. Thus, they inherit its prototype methods, such as `map`, `reduce`, and `filter`. D3 selections are also descendants of the `Array` primitive, but D3 then goes on to replace some functions, such as `filter` and `sort`, with its own versions adapted for selections, other than adding a few other helpful methods such as `.each`, or to come full circle with the whole naming thing, `d3.selection.prototype.each`.

Even ES2015 classes, which use a very different inheritance model and come from object-oriented programming, are ultimately prototype-based.

Let's look at the built-in functions in detail. It's worth noting that none of these are *mutative*. In other words, they return a copy of the source array and leave it unchanged:

- `Array.prototype.map` applies a function to every element of an array and returns a new array with changed values:

```
> [1,2,3,4].map((d) => d+1)
[ 2, 3, 4, 5 ]
```

- `Array.prototype.reduce` uses a combining function and a starting value to collapse an array into a single value:

```
> [1,2,3,4].reduce(
(accumulator, current) => accumulator+current, 0)
10
```

- `Array.prototype.filter` goes through an array and keeps those elements for which the predicate returns true:

```
> [1,2,3,4].filter((d) => d%2)
[ 1, 3 ]
```

- Two more useful functions are `Array.prototype.every` and `Array.prototype.some`, which are true if all or some items in the array are true:

```
// Are all elements odd?
[1,3,5,7,9].every(elem => elem % 2); // True
[1,2,5,7,9].every(elem => elem % 2); // False

// Is at least one odd?
[1,3,5,7,9].some(elem => elem % 2); // True
[1,2,5,7,9].some(elem => elem % 2); // True
[0,2,4,6,8].some(elem => elem % 2); // False
```

Sometimes, using `Array.prototype.forEach` instead of `Array.prototype.map` is better because `.forEach` operates on the original array instead of creating a copy, which is important for working with large arrays and is mainly used for the side effect. The `.forEach` is also useful when you want each element in an array to run some logic and don't want to necessarily do anything to the original array itself.

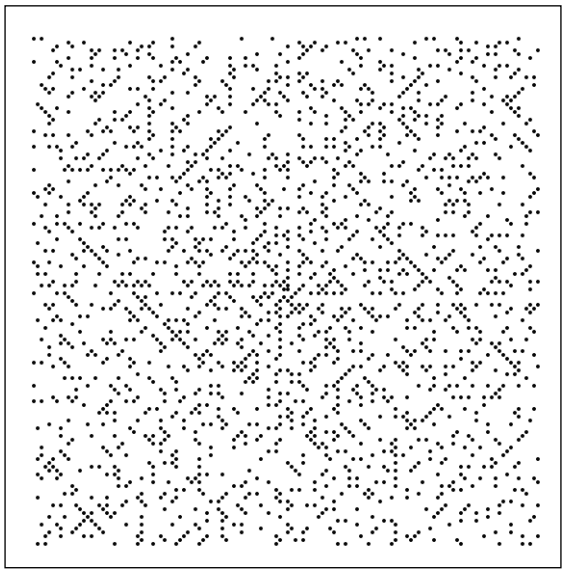
These functions are relatively new to JavaScript, whereas `.map` and `.filter` have existed since JavaScript 1.7 and `.reduce` since 1.8. In the bad old days, you'd have to use either `es6-shim` or something like `Underscore.js` to be able to use them, but since we're now in the bright shiny ES6 future, Babel polyfills them for us when it transpiles our bundle together.

Data functions of D3

D3 comes with plenty of its own array functions. They mostly have to do with handling data; it comprises calculating averages, ordering, bisecting arrays, and many helper functions for associative arrays.

Let's play with some data functions and draw an unsolved mathematical problem called the Ulam spiral. Discovered in 1963, it reveals patterns in the distribution of prime numbers on a two-dimensional plane. So far, nobody has found a formula that explains them.

We'll construct the spiral by simulating Ulam's pen-and-paper method; we'll write natural numbers in a spiraling pattern and then remove all non-primes, but instead of numbers, we'll draw dots. The first stage of our experiment will look like this:



It doesn't look like much, but that's only the first 2,000 primes in a spiral. Did you notice the diagonal rows of dots? Some can be described with polynomials, which brings interesting implications about predicting prime numbers and, by extension, the safety of cryptography.

Let's start by creating a new file called `chapter3.js` in our `src/` directory. Add a new class as follows:

```
let d3 = require('d3');

import {BasicChart} from './basic-chart';

export class UlamSpiral extends BasicChart {
 constructor(data) {
    super(data);
  }
}
```

None of this should look very new at this point; we've just scaffolded a new chart based on our `BasicChart` class.

Next, we define an algorithm that generates a list of numbers and their spiraling coordinates on a grid. We start by creating the spiral algorithm. Create a private class method by putting the following inside the class, beneath the constructor function:

```
  generateSpiral(n) {
    let spiral = [],
      x = 0, y = 0,
      min = [0, 0],
      max = [0, 0],
      add = [0, 0],
      direction = 0,
      directions = {
        up : [ 0, -1 ],
        left : [ -1, 0 ],
        down : [ 0, 1 ],
        right : [ 1, 0 ]
      };
```

We have defined a spiral function that takes a single upper-bound argument, `n`. This function starts with four directions of travel and some variables for our algorithm. The combination of the `min` and `max` known coordinates will tell us when to turn, `x` and `y` will be the current position, and `direction` will tell us which part of the spiral we're tracing.

Next, we add the algorithm itself at the bottom of our function:

```
      d3.range(1, n).forEach((i) => {
        spiral.push({x : x, y : y, n : i});
        add = directions[[ 'up', 'left', 'down', 'right'
][direction]];
        x += add[0], y += add[1];

        if (x < min[0]) {
          direction = (direction + 1) % 4;
          min[0] = x;
        }
        if (x > max[0]) {
          direction = (direction + 1) % 4;
          max[0] = x;
        }
        if (y < min[1]) {
          direction = (direction + 1) % 4;
          min[1] = y;
        }
        if (y > max[1]) {
          direction = (direction + 1) % 4;
          max[1] = y;
        }
      });

      return spiral;
```

The `d3.range()` generates an array of numbers between the two arguments, which we then iterate with `.forEach`. Each iteration adds a new `{x: x, y: y, n: i}` triplet to the spiral array. The rest is just the use of `min` and `max` to change the direction once we bump into a corner.

Now we'll get to draw stuff. Go back to the constructor function and add the following code under the call to `super`:

```
      let dot = d3.svg.symbol().type('circle').size(3),
        center = 400,
        l = 2,
        x = (x, l) => center + l * x,
        y = (y, l) => center + l * y;
```

So, we've defined a dot generator and two functions to help us turn grid coordinates from the spiral function into pixel positions. Here, `l` is the length and width of a square in the grid.

Next, we need to calculate primes. We could have, of course, got a big list of them online, but that wouldn't be as much fun as using generators, a new technology in ES2015.

What are generators and why should you care? Generators are effectively factories for iterators, which are functions that are able to access items from a collection one at a time while keeping track of their internal position. We could use a bunch of `forEach` loops, but this would be more computationally heavy and not as extensible. That said, you don't need to use generators or iterators at all—I do so here merely to expose a new feature of ES2016, which you might find useful if you frequently find yourself processing the individual items of a collection in a certain way. For more information about iterators and generators, visit `http://mdn.io/Iterators_and_Generators`.

To start, we need to make sure that the Babel polyfill is available. Generators are so new that even modern browsers need a polyfill for them. To do this, go to `index.js` and replace its contents with the following line:

```
import 'babel-polyfill';
```

While you're here, add these two lines as well:

```
import {UlamSpiral} from './chapter3';
new UlamSpiral();
```

Next, we create a new method called `generatePrimes` in our `UlamSpiral` class in `chapter3.js`:

```
generatePrimes(n) {}
```

Put the following generators inside this function.

Our first generator is simply going to be a function that we call over and over, each time giving us the next cardinal number:

```
function* numbers(start) {
  while (true) {
    yield start++;
  }
}
```

Wow, this looks all new and confusing! Let's break it down a bit. The asterisk with the `function` keyword simply denotes it as a generator function. Once we're into the `while` loop (which never finishes, so we can keep asking for new numbers until the cows come home), we use the `yield` keyword to return the next number. We'll see how this is used in just a moment.

Now we're going to create our `primes` generator, which will continually call our new `numbers` generator:

```
function* primes() {
  var seq = numbers(2); // Start on 2.
  var prime;

  while (true) {
    prime = seq.next().value;
    yield prime;
   }
}
```

This shouldn't be that difficult to understand now. We assign our `numbers` generator to `seq` and start it from `2`. Then we use `.next()` to have it yield the next result value from the `number` generator.

We need another generator to iterate through our primes. Add the following code:

```
function* getPrimes(count, seq) {
  while (count) {
    yield seq.next().value;
    count--;
  }
}
```

Now, at the end of `generatePrimes`, put these lines:

```
for (var prime of getPrimes(n, primes())) {
  console.log(prime);
}
```

And then, this comes under our dot generator in the class's constructor function:

```
let primes = this.generatePrimes(2000);
```

Suppose you start the development server via the following line:

```
$ npm start
```

Then you go to `http://localhost:8080/`. You'll see an array of 2,000 sequential integers on your console.

We still need to add a filter for prime numbers. Go back to `generatePrimes` and add this new generator:

```
function* filter(seq, prime) {
  for (var num of seq) {
    if (num % prime !== 0) {
      yield num;
    }
  }
}
```

This one just goes through all the numbers in the sequence thus far and checks whether there are any remainders when they're divided by a possible prime. If any of the numbers in the sequence has `0` as the remainder when divided by a potential prime, it means that the number isn't in fact a prime. We'll use this to filter out non-prime numbers.

Next, in the `primes` generator, after `yield prime`, put the following line:

```
seq = filter(seq, prime);
```

This will run the entire sequence up to the present iteration against the prime in the `filter` function that we just created.

Consider this code:

```
for (var prime of getPrimes(n, primes())) {
  console.log(prime);
}
```

Change it to the following:

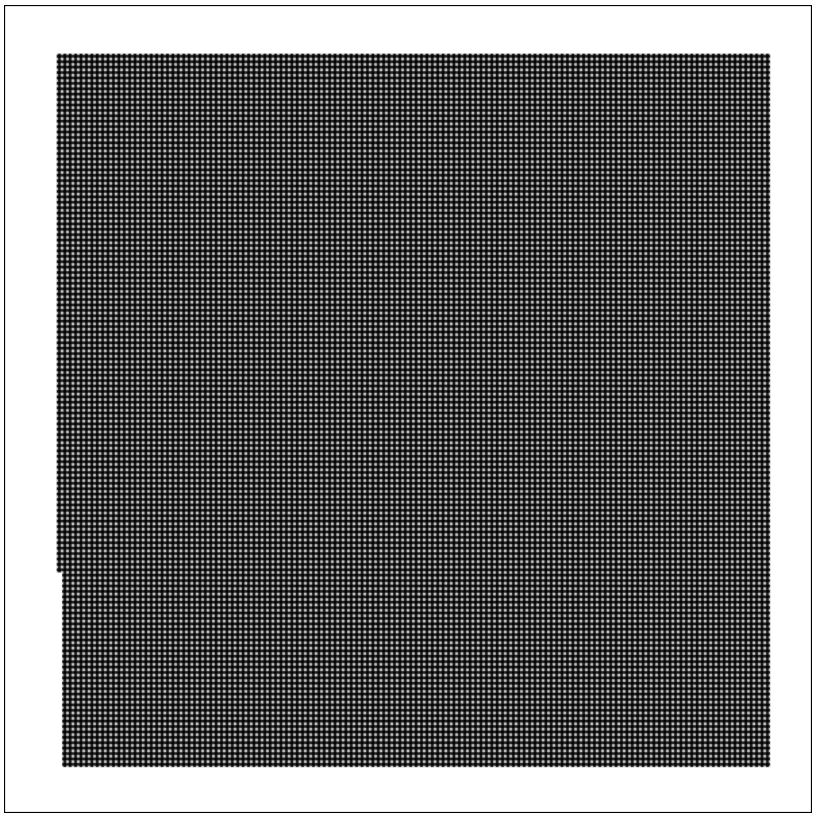
```
let results = [];
for (let prime of getPrimes(n, primes())) {
  results.push(prime);
}

return results;
```

Now all of our primes are in an array. Time to tie this all together! Go back to the constructor and add the following lines:

```
let primes = this.generatePrimes(2000);
let sequence = this.generateSpiral(d3.max(primes));
```

This creates an array of 2,000 primes using our generator and runs our spiral generator on the maximum value of those primes. Now, let's combine this with our dot generator to finally get the example we had all those many pages ago!

```
this.chart.selectAll('path')
    .data(sequence)
    .enter()
    .append('path')
    .attr('transform',
          d => `translate(${ x(d['x'], l) }, ${ y(d['y'], l) })`)
    .attr('d', dot);
```



Hmm… okay, not quite there, but it's still the right idea!

What we need to do next is filter out the non-prime numbers from that dot matrix. Change your `let sequence` line to the following:

```
let sequence = this.generateSpiral(d3.max(primes))
                   .filter((d) => primes.indexOf(d['n']) > -1);
```

If you have a slower or older computer, this might take a while because you're asking a lot of your poor web browser!

Clearly, there are some performance implications at play here. Although our project is super cool and able to generate however many prime numbers we want, in reality this is a brutally inefficient way of arriving at the result (and indeed, generating more than 2,500 or so numbers tends to cause web browsers to hit stack size limits).

Earlier, it was mentioned that we can always get a list of several thousand primes online and it would only be another kilobyte or two to load. In most circumstances, this would be the correct way forward. Throughout the rest of the book, we will generally take this approach.

Let's make it more interesting by visualizing the density of primes. We'll define a grid with larger squares and then color them depending on how many dots they contain. A square will be red when there are fewer primes than the median and green when there are more. The shading will tell us how far they are from the median.

First, we'll use the nest structure of D3 to define a new grid. Let's continue from where we left off in the constructor:

```
let scale = 8;
let regions = d3.nest()
  .key((d) => Math.floor(d['x'] / scale))
  .key((d) => Math.floor(d['y'] / scale))
  .rollup((d) => d.length)
  .map(sequence);
```

We scale by a factor of `8`; that is, each new square contains 64 of the old squares.

The `d3.nest()` is handy for turning data into nested dictionaries according to a key. The first `.key()` function creates our columns; every `x` is mapped to the corresponding `x` of the new grid. The second `.key()` function does the same for `y`. We then use `.rollup()` to turn the resulting lists into a single value, a count of the dots.

The data goes in with `.map()`, and we get a structure as follows:

```
{
  "0": {
    "0": 5,
    "-1": 2
  },
  "-1": {
      "0": 3,
      "-1": 4
  }
}
```

It's not very self-explanatory, but that's a collection of columns containing rows. The `(0, 0)` square contains five primes, `(-1, 0)` contains two, and so on.

To get the median and the number of shades, we need those counts in an array:

```
let values = d3.merge(
  d3.keys(regions).map((_x) => d3.values(regions[_x]));
let median = d3.median(values),
  extent = d3.extent(values),
  shades = (extent[1]-extent[0])/2;
```

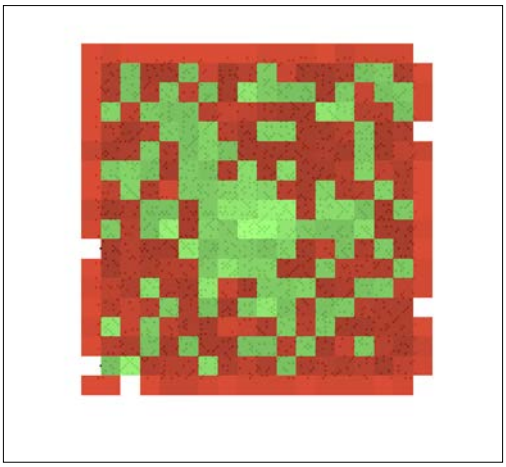
We map through the keys of our regions (*x* coordinates) to get a list of values for each column, and then use `d3.merge()` to flatten the resulting array of arrays.

The `d3.median()` gives us the middle value of our array, and `d3.extent()` gives us the lowest and highest number, which we used to calculate the number of shades we needed.

Finally, we walk the coordinates again to color the new grid:

```
d3.keys(regions).forEach((_x) => {
  d3.keys(regions[_x]).forEach((_y) => {
    let color,
        red = '#e23c22',
        green = '#497c36';

    if (regions[_x][_y] > median) {
      color = d3.rgb(green).brighter(regions[_x][_y] / shades);
    } else {
      color = d3.rgb(red).darker(regions[_x][_y] / shades);
    }
    this.chart.append('rect')
        .attr({
          x : x(_x, 1 * scale),
          y : y(_y, 1 * scale),
          width : 1 * scale,
          height : 1 * scale
        })
        .style({fill : color, 'fill-opacity' : 0.9});
  });
});
```

Our image looks like one of those randomly-generated WordPress avatars:



Loading data

One of the best features of D3 is that it has a bunch of great helper functions for loading data. While sometimes it's easier to have your code generate your dataset, most of the time, you'll be mapping real data to what you create with D3.

The reason we want to load data externally is that bootstrapping large datasets into the page with predefined variables isn't very practical. Loading hundreds of kilobytes of data takes a while, and doing so asynchronously lets the rest of the page render in the meantime. Plus, who wants all of that data smack-dab in the middle of your code anyway?

To make HTTP requests, D3 uses `XMLHttpRequests` (XHR for short). This limits us to loading data off the same domain as the script because of the browser's security model, but we can make cross-domain requests if the server sends a header resembling `Access-Control-Allow-Origin: *` (commonly known as a **Cross-Origin Resource Sharing** or **CORS** header).

The core

At the core of all this loading is the humble `d3.xhr()`, the manual way of issuing an XHR request.

It takes a URL and an optional callback. If supplied with a callback, it will immediately trigger the request and receive the data as an argument once the request finishes.

If there's no callback, we get to tweak the request; everything from the headers to the `request` method, later making the request once ready.

To make a request, you might have to write the following code:

```
let xhr = d3.xhr('<a_url>');
xhr.mimeType('application/json');
xhr.header('User-Agent', 'SuperAwesomeBrowser');
xhr.on('load', function (request) { ... });
xhr.on('error', function (error) { ... });
xhr.on('progress', function () { ... });
xhr.send('GET');
```

This will send a `GET` request, expecting a JSON response, and will tell the server that we're a web browser named `SuperAwesomeBrowser`. One way of shortening this is by defining a callback immediately, but then you can't define custom headers or listen for other request events.

Another way is convenience functions. We'll be using these throughout the book.

Convenience functions

D3 comes with several convenience functions that use `d3.xhr()` behind the scenes and parse the response before giving it back to us. This lets us limit our workflow to calling the appropriate function and defining a callback, which takes an error and a data argument. D3 is also nice enough to let us throw caution to the wind and use callbacks with a single data argument that will be undefined in case of an error.

We have a choice of data formats such as TXT, JSON, XML, HTML, CSV, and TSV. JSON and CSV/TSV are used the most, JSON for small, structured data and CSV/TSV for large data dumps, where we want to conserve space.

All of these follow this pattern:

```
d3.json('a_dataset.json', function (err, data) {
     // draw stuff
});
```

Unfortunately, this syntax makes it a bit annoying to use promises and async-wait, two new features in ES2015. We'll generally use these instead of the normal D3 way of doing things because they improve code flow and allow intelligent loading of multiple resources. Hopefully, D3's convenience functions will return promises by default sometime in the future. I've opened an issue for this reason, and you can track its progress at `https://github.com/mbostock/d3/issues/2684`.

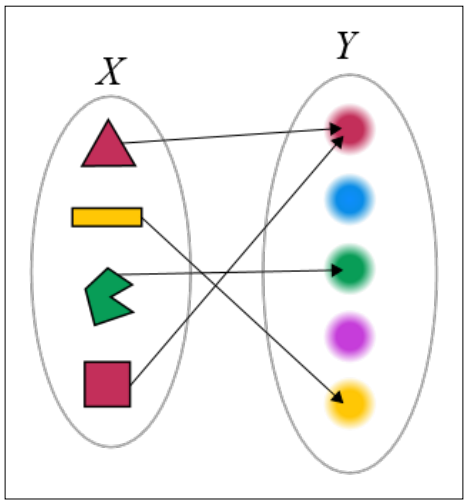
Scales

Scales are functions that map a domain to a range. I know I keep saying that, but there really isn't much more to say.

The reason we use them is to avoid math. This makes our code shorter, easier to understand, and more robust, as mistakes in high-school mathematics are some of the hardest bugs to track down.

If you haven't spent 4 years just listening to mathematics at school, note that a function's domain includes the values for which it is defined (the input), and its range includes the values it returns.

The following figure is borrowed from Wikipedia:



Here, **X** is the domain, **Y** is the range, and the arrows are the functions. We need a bunch of code to implement this manually:

```
let shape_color = (shape) => {
  if (shape == 'triangle') {
    return 'red';
  }else if (shape == 'line') {
    return 'yellow';
  }else if (shape == 'pacman') {
    return 'green';
  }else if (shape == 'square') {
    return 'red';
  }
};
```

You can also do it with a dictionary, but `d3.scale` will always be more elegant and flexible:

```
let scale = d3.scale.ordinal()
      .domain(['triangle', 'line', 'pacman', 'square'])
      .range(['red', 'yellow', 'green', 'red']);
```

Much better!

Scales come in three types; ordinal scales have a discrete domain, quantitative scales have a continuous domain, and time scales have a time-based continuous domain.

Ordinal scales

Ordinal scales are the simplest, essentially just a dictionary where the keys are the domain and the values are the range.

In the preceding example, we defined an ordinal scale by explicitly setting both the input domain and the output range. If we don't define a domain, it's inferred from use, but that can give unpredictable results.

A cool thing about ordinal scales is that having a range smaller than the domain makes the scale repeat values once used. Furthermore, we'd get the same result if the range were just `['red', 'yellow', 'green']`.

Let's try one. Create a new class in `chapter3.js` named `ScalesDemo`, as shown in this code:

```
export class ScalesDemo extends BasicChart {
  constructor() {
    super();
    this.ordinal();
   }

  ordinal() {
  }
}
```

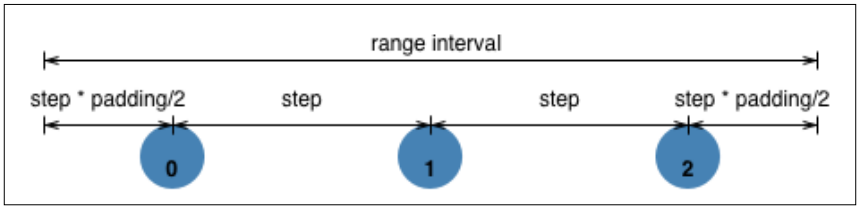
Inside the `ordinal()` method, we define the three scales that we need and generate some data:

```
  ordinal() {
    let data = d3.range(30),
      colors = d3.scale.category10(),
      points = d3.scale.ordinal().domain(data)
        .rangePoints([0, this.height], 1.0),
```

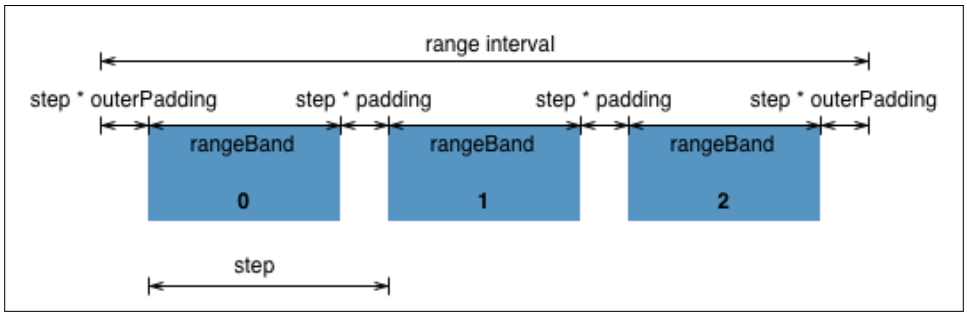
```
    bands = d3.scale.ordinal().domain(data)
      .rangeBands([0, this.width], 0.1);
}
```

Our data is just a list of numbers going up to 30, and the `colors` scale is from *Chapter 2*, *A Primer on DOM, SVG, and CSS*. It is a predefined ordinal scale with an undefined domain and a range of 10 colors.

Then we defined two scales that split our drawing into equal parts. The `points` uses `.rangePoints()` to distribute 30 equally spaced points along the height of our drawing. We set the edge padding with a factor of `1.0`, which sets the distance from the last point to the edge to half the distance between the points. The end points are moved inwards from the range edge using `point_distance*padding/2`.



Our `bands` scale uses `.rangeBands()` to divide the width into 30 equal bands with a padding factor of `0.1` between the bands. This time, we're setting the distance between bands using `step*padding`, and a third argument will set the edge padding using `step*outerPadding`, as you can see here:



We'll use the code you already know from *Chapter 2*, *A Primer on DOM, SVG, and CSS*, to draw two lines using these scales:

```
let data = d3.range(30),
  colors = d3.scale.category10(),
  points = d3.scale.ordinal().domain(data)
    .rangePoints([0, this.height], 1.0),
 bands = d3.scale.ordinal().domain(data)
```

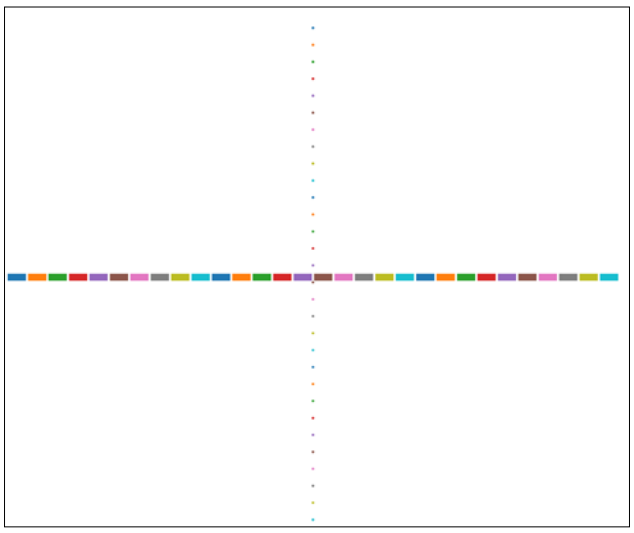
```
          .rangeBands([0, this.width], 0.1);

        this.chart.selectAll('path')
          .data(data)
          .enter()
          .append('path')
          .attr({d: d3.svg.symbol().type('circle').size(10),
              transform: (d) => `translate(${(this.width / 2)},
${points(d)})`
           })
          .style('fill', (d) => colors(d));

        this.chart.selectAll('rect')
          .data(data)
          .enter()
          .append('rect')
          .attr({x: (d) => bands(d),
             y: this.height / 2,
             width: bands.rangeBand(),
             height: 10})
          .style('fill', (d) => colors(d));
```

Now, update `index.js` to resemble this:

```
import {ScalesDemo} from './chapter3';
new ScalesDemo();
```

To get the positions for each dot or rectangle, we have called the scales as functions, and used `bands.rangeBand()` to get the rectangle width.

The picture looks like this:



Quantitative scales

Quantitative scales come in a few different flavors, but they all share a common characteristic in that the input domain is continuous. Instead of a set of discrete values, a continuous scale can be modeled with a simple function. The seven types of quantitative scales are linear, identity, power, log, quantize, quantile, and threshold. They define different transformations of the input domain. The first four have a continuous output range while the latter three map to a discrete range.

To see how they behave, we'll use all of these scales to manipulate the *y* coordinate when drawing the `weierstrass` function, the first discovered function that is continuous everywhere but differentiable nowhere. This means that even though you can draw the function without lifting your pen, you can never define the angle you're drawing at (calculate a derivative).

Create a new method in `ScalesDemo` called `quantitative` and fill it with the following code:

```
quantitative() {
  let weierstrass = (x) => {
    let a = 0.5,
      b = (1+3*Math.PI/2) / a;
    return d3.sum(d3.range(100).map((n) => {
      return Math.pow(a, n)*Math.cos(Math.pow(b, n)*Math.PI*x);
    }));
  };
}
```

A drawing function will help us avoid code repetition:

```
  let drawSingle = (line) => {
    return svg.append('path')
      .datum(data)
      .attr('d', line)
      .style({'stroke-width': 2,
      fill: 'none'});
  };
```

We generate some data, get the extent of the `weierstrass` function, and use a linear scale for `x`:

```
var data = d3.range(-100, 100).map(function (d) { return d/200; }),
  extent = d3.extent(data.map(weierstrass)),
  colors = d3.scale.category10(),
  x = d3.scale.linear().domain(d3.extent(data)).range([0, width]);
```

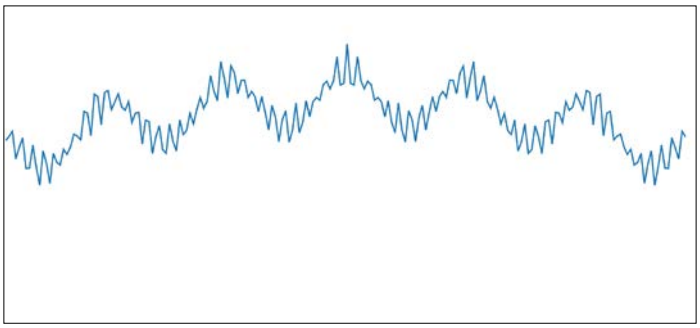
Continuous range scales

We can draw using the following code:

```
    let linear =
d3.scale.linear().domain(extent).range([this.height/4, 0]),
      line1 = d3.svg.line()
        .x(x)
        .y((d) => linear(weierstrass(d)));

    drawSingle(line1)
      .attr('transform', `translate(0, ${this.height / 16})`)
      .style('stroke', colors(0));
```

We defined a linear scale with the domain encompassing all the values returned by the `weierstrass` function, and a range from zero to the drawing width. The scale will use linear interpolation to translate between the input and the output, and will even predict values that fall outside of its domain. If we don't want that to happen, we can use `.clamp()`. Using more than two numbers in the domain and range, we can create a polylinear scale, where each section behaves like a separate linear scale.

The linear scale creates what you can see in the following screenshot:



Let's add the other continuous scales in one fell swoop:

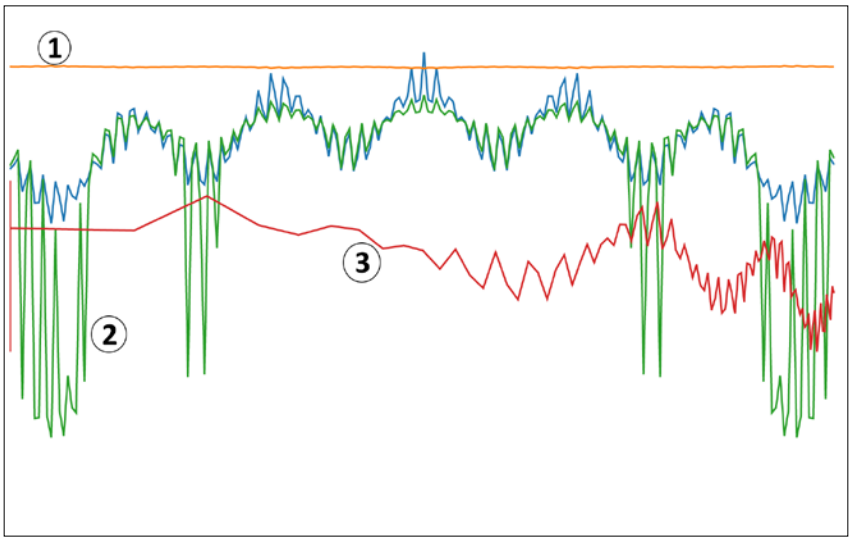
```
    let identity = d3.scale.identity().domain(extent),
      line2 = line1.y((d) => identity(weierstrass(d)));

    drawSingle(line2)
      .attr('transform', `translate(0, ${this.height / 12})`)
      .style('stroke', colors(1));

    let power =
d3.scale.pow().exponent(0.2).domain(extent).range([ this.height /
2, 0]),
```

```
  line3 = line1.y((d) => power(weierstrass(d)));

drawSingle(line3)
  .attr('transform', `translate(0, ${this.height / 8})`)
  .style('stroke', colors(2));

var log = d3.scale.log().domain(
  d3.extent(data.filter((d) => d > 0 ? d : 0)))
  .range([0, this.width]),
  line4 = line1.x((d) => d > 0 ? log(d) : 0)
    .y((d) => linear(weierstrass(d)));

drawSingle(line4)
  .attr('transform', `translate(0, ${this.height / 4})`)
  .style('stroke', colors(3));
```

We keep reusing the same line definition, changing the scale used for `y`, except for the power scale, because changing `x` makes a better example.

We also took into account the fact that log is only defined on positive numbers, but you usually wouldn't use it for periodic functions anyway. It's much better for showing large and small numbers on the same graph.

Now our picture looks as follows:



The identity scale (labeled **1**) is orange and wiggles around by barely a pixel. This is because the data we feed into the function only ranges from `-0.5` to `0.5`. The power scale (labeled **2**) is green and the logarithmic scale (labeled **3**) is red.

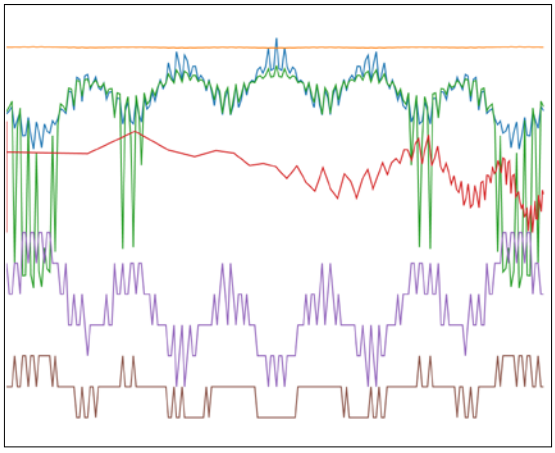
Discrete range scales

The scales that are interesting for our comparison are quantize and threshold. The quantize scale cuts the input domain into equal parts and maps them to values in the output range, while the threshold scale lets us map arbitrary domain sections to discrete values:

```
let quantize = d3.scale.quantize().domain(extent)
  .range(d3.range(-1, 2, 0.5).map((d) => d*100)),
  line5 = line1.x(x).y((d) => quantize(weierstrass(d))),
  offset = 100

drawSingle(line5)
 .attr('transform',
   `translate(0, ${this.height / 2 + offset})`)
 .style('stroke', colors(4));

var threshold = d3.scale.threshold()
.domain([-1, 0, 1]).range([-50, 0,50, 100]),
  line6 = line1.x(x).y((d) => threshold(weierstrass(d)));

drawSingle(line6)
 .attr('transform',
   `translate(0, ${this.height / 2 + offset * 2})`)
 .style('stroke', colors(5));
```

The quantize scale will divide the `weierstrass` function into discrete values between `1` and `2` with a step of `0.5` (`-1`, `-0.5`, `0`, and so on), and `threshold` will map values smaller than `-1` to `-50`, `-1` to `0`, and so on.

The result looks like this:



Time

Time is a complicated beast. An hour can last 3,600 seconds or 3,599 seconds if there's a leap second. Tomorrow can be 23 to 25 hours away, months range from 28 to 31 days, and a year can be 365 or 366 days. Some decades have fewer days than others.

Keep this in mind the next time you want to add 3,600 seconds to a timestamp to advance it by an hour, or by adding 24*3600 to a timestamp to get the same time one day into the future.

Considering that many datasets are closely tied to time, this can become a big problem. Just how do you handle time?

Luckily, D3 comes with a bunch of time-handling features.

Formatting

You can create a new formatter by giving `d3.time.format()` a format string. You can then use it to parse strings into Date objects and vice versa.

The whole language is explained in the documentation of D3, but let's look at a few examples:

```
> format = d3.time.format('%Y-%m-%d')
> format.parse('2015-12-14')
Mon Dec 14 2015 00:00:00 GMT+0100 (CET)
```

We defined a new formatter with `d3.time.format()` (year-month-day) and then parsed a date as they often appear in datasets. This gave us a proper date object with default values for hours, minutes, and seconds.

The same formatter works the opposite way as well:

```
> format(new Date())
"2013-02-19"
```

You can find the complete ISO standard time formatter at `d3.time.format.iso`. That often comes in handy.

Time arithmetic

We also get a full suite of time arithmetic functions that work with JavaScript's Date objects and follow a few simple rules:

- `d3.time.interval`: Here, `interval` can be a second, minute, hour, and so on. It returns a new time interval. For instance, `d3.time.hour` will be an hour long.
- `d3.time.interval(Date)`: This is an alias for `interval.floor()`, which rounds `Date` down so that more specific units than the interval are set to zero.
- `interval.offset(Date, step)`: This will move the date by a specified number of steps to the correct unit.
- `interval.range(Date_start, Date_stop)`: This will return every interval between the two specified dates.
- `d3.time.intervals`: Here, an interval is seconds, minutes, hours, and so on. These are helpful aliases for `interval.range`.

For instance, if you want to know the time an hour from now, you will have to do this:

```
> d3.time.hour.offset(new Date(), 1)
Mon Dec 15 2015 00:06:30 GMT+0100 (CET)
```

And find out that it's getting really late and you should stop writing books about JavaScript and go to bed!

Want to do more with time? Moment.js is a terrific library for accurate calculations of things such as time zones and the difference between two datestamps:

`http://momentjs.com`.

Geography

Geospatial data types are used for weather or population data—anything where you want to draw a map. Converting real-world coordinates into something representable on a 2D plane is a complex mathematical problem that has spanned centuries of human history.

D3 gives us three tools for geographic data:

- **Paths** produce the final pixels
- **Projections** turn sphere coordinates into Cartesian coordinates
- **Streams** speed things up

The main data format that we'll use is TopoJSON, a more compact extension of GeoJSON, created by Mike Bostock. In a way, TopoJSON is to GeoJSON what DivX is to video. While GeoJSON uses the JSON format to encode geographical data with points, lines, and polygons, TopoJSON encodes basic features with arcs and reuses them to build more and more complex features. As a result, files can be as much as 80 percent smaller than when we use GeoJSON.

Getting geodata

Now, unlike many other datasets, geodata can't be found just lying around the Internet, especially not in a fringe format such as TopoJSON. We'll find some data in Shapefile or GeoJSON formats, and then use the `topojson` command-line utility to transform it into TopoJSON. Finding detailed data can be difficult, but is not impossible. Look for your country's census bureau. For instance, the US Census Bureau has many useful datasets available at `https://www.census.gov/geo/maps-data/`, and the equivalent for the UK is at `https://geoportal.statistics.gov.uk/geoportal/`.

Natural Earth is another magnificent resource for geodata at different levels of detail. The biggest advantage of it is that different layers (oceans, countries, roads, and so on) are carefully made to fit together without discrepancies and are frequently updated. You can find the datasets at `http://www.naturalearthdata.com`.

Let's prepare some data for the next example. Go to `http://www.naturalearthdata.com` and download the ocean, land, rivers, and lake centerlines and land boundary lines datasets at the 50m detail level, and the urban areas dataset at 10m. You'll find them under the **Downloads** tab. The files are also in the examples on GitHub available at `https://github.com/aendrew/learning-d3/tree/chapter3/src/data`.

Unzip the five files. We'll combine them into three TopoJSON files to save the request time—three big files are quicker than five small files—and we prefer TopoJSON because of the smaller file size.

We'll merge categorically so that we can reuse the files later: one for water data, another for land data, and the third for cultural data.

You'll need to install `topojson`, which is a command-line utility written in NodeJS. On the command line, type this line:

```
$ npm install -g topojson
```

If it gives you errors about permissions, try it again as a super user:

```
$ sudo npm install -g topojson
```

Next, we transform the files with three simple commands:

```
$ topojson -o water.json ne_50m_rivers_lake_centerlines.shp ne_50m_ocean.
shp
$ topojson -o land.json ne_50m_land.shp
$ topojson -o cultural.json ne_50m_admin_0_boundary_lines_land.shp
ne_10m_urban_areas.shp
```

The `topojson` library transforms shape files into TopoJSON files and merges the files that we want. We specified where to put the results with `-o`; the other arguments were source files.

We've generated three files: water.json, land.json, and cultural.json. Feel free to look at them, but they aren't very "human friendly."

Drawing geographically

The `d3.geo.path()` is going to be the work horse of our geographic drawings.

It's similar to the SVG path generators that you learned about earlier, except that it draws geographic data and is smart enough to decide whether to draw a line or an area.

To flatten spherical objects, such as planets, into 2D images, `d3.geo.path()` uses projections. Different kinds of projections are designed to showcase different things about the data, but the end result is that you can completely change what the map looks like just by changing the projection or moving its focal point.

With the right projection, you can even make the data of Europe look like that of the U.S. Rather unfortunately then, the default projection is `albersUsa`, designed specifically to draw the standard map of the U.S.

Let's draw a map of the world, centered and zoomed into Europe because that's where I'm from. We'll make it navigable in *Chapter 4*, *Defining the User Experience — Animation and Interaction*.

We first need to add some things to our standard HTML file.

We need to install TopoJSON in our project. Note that this is different from installing it with `-g`; in this case, we want to use it as a dependency and not as a command-line utility:

```
$ npm install topojson --save
```

Now, we require `topojson` at the top of `chapter3.js`:

```
let topojson = require('topojson');
```

Let's create a new class for all of this in `chapter3.js`:

```
export class GeoDemo extend BasicChart {
  constructor(){
    super();
    let chart = this.chart;
  }
}
```

Note that this time around, we're assigning the parent class's `chart` property to a local variable—`chart`. We could keep referring to it as `this.chart`, but then we would need to do some ugly stuff with `Function.prototype.call`; I'd rather not get into that.

Next, we define a geographic projection in the constructor:

```
let projection = d3.geo.equirectangular()
    .center([8, 56])
    .scale(800);
```

The equirectangular projection is one of the 12 projections that come with D3, and it is perhaps the most common projection we're used to seeing ever since high school.

The problem with equirectangular projection is that it doesn't preserve areas or represent the Earth's surface all that well. A full discussion of projecting a sphere onto a two-dimensional surface would take too much time, so I recommend looking at the Wikipedia page of D3 and the visual comparison of all the projections implemented in the projection plugin. It is available at `https://github.com/mbostock/d3/wiki/Geo-Projections`.

The next two lines define where our map is centered and how much zoomed in it is. By fiddling, I got all three values: latitude 8, longitude 56, and a scaling factor of 800. Play around to get a different look.

Now we load our data using ES2016 promises:

```
let p1 = new Promise((resolve, reject) => {
  d3.json('data/water.json', (err, data){
    err ? reject(err) : resolve(data);  });
});

let p2 = new Promise((resolve, reject) => {
  d3.json('data/land.json', (err, data){
    err ? reject(err) : resolve(data);  });
```

```
});

let p3 = new Promise((resolve, reject) => {
  d3.json('data/cultural.json', (err, data){
    err ? reject(err) : resolve(data);
  });
});

Promise.all([p1, p2, p3]).then((values) => {
  let [land, sea, cultural] = values; // OMG ES2016 DESTRUCTURING
});
```

We're using ES2016 promises to run the three loading operations in sequence. Each will use `d3.json()` to load and parse the data, either rejecting (if there's an error) or resolving the promise (if the error function argument is undefined or null). The promises are then collected in `Promise.all()`, which fires its `.then()` method once all the promises are resolved and accounted for. We then use a new ES2016 feature—destructuring—to assign each element of the array to a new variable.

Now, what's all this about destructuring? To quote the Mozilla Developer's Network, destructuring assignment syntax allows the "*extract[ion of] data from arrays or objects using a syntax that mirrors the construction of array and object literals.*"

The equivalent code in ES5 would be as follows:

```
var land = values[0];
var sea = values[1];
var cultural = values[2];
```

For more on destructuring and how it can make your code awesome, check out `https://mdn.io/Destructuring_assignment`.

We need one more thing before we start drawing. We need a function that adds a feature to the map, which will help us reduce code repetition:

```
    function addToMap(collection, key) {
      return chart.append('g')
        .selectAll('path')
          .data(topojson.feature(collection,
collection.objects[key]).features)
        .enter()
        .append('path')
        .attr('d', d3.geo.path().projection(projection));
    }
```

This function takes a collection of objects and a key for choosing which object to display. The `topojson.object()` translates a TopoJSON topology into a GeoJSON one for `d3.geo.path()`.

Whether it's more efficient to transform to GeoJSON than to transfer data in the target representation depends on your use case. Transforming data takes some computational time, but transferring megabytes instead of kilobytes can make a big difference in responsiveness.

Finally, we create a new `d3.geo.path()` and tell it to use our projection. Other than generating the SVG path string, `d3.geo.path()` can also calculate different properties of our feature, such as the area (`.area()`) and the bounding box (`.bounds()`).

Now we can start drawing:

```
function draw (sea, land, cultural) {
  addToMap(sea, 'ne_50m_ocean')
    .classed('ocean', true);
}
```

Our `draw` function takes the error returned from loading data, and the three datasets then let `addtoMap` do the heavy lifting.

We add some styling to `index.css`:

```
.ocean {
  fill: #759dd1;
}
```

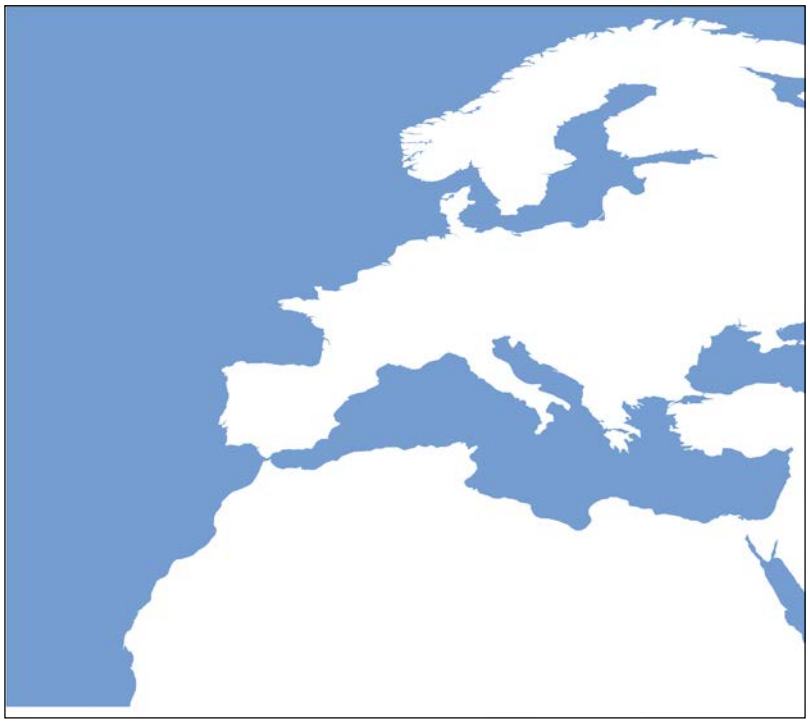
And then we require it at the top of `chapter3.js`:

```
require('./index.css');
```

Lastly, we call `draw` inside of our promise callback:

```
Promise.all([p1, p2, p3]).then((values) => {
  let [sea, land, cultural] = values;
  draw(sea, land, cultural);
});
```

Refreshing the page, we'll be in ocean town!



We add four more `addToMap` calls to the `draw` function to fill in the other features, as follows:

```
addToMap(land, 'ne_50m_land')
   .classed('land', true);
addToMap(sea, 'ne_50m_rivers_lake_centerlines')
   .classed('river', true);
addToMap(cultural, 'ne_50m_admin_0_boundary_lines_land')
   .classed('boundary', true);
addToMap(cultural, 'ne_10m_urban_areas')
   .classed('urban', true);
```

Add some more style definitions as follows:

```
.river {
  fill: none;
  stroke: #759dd1;
  stroke-width: 1;
}
```

```
.land {
     fill: #ede9c9;
     stroke: #79bcd3;
     stroke-width: 2;
}

.boundary {
     stroke: #7b5228;
     stroke-width: 1;
     fill: none;
}

.urban {
     fill: #e1c0a3;
}
```

We now have a slowly rendering world map zoomed into Europe, displaying the world's urban areas as blots:

There are many reasons for it to be so slow. We transform between TopoJSON and GeoJSON on every call to `addToMap`. Even when using the same dataset, we're using data that's too detailed for such a zoomed-out map, and we render the whole world to look at a tiny part. We have traded flexibility for rendering speed.

Using geography as a base

Geography isn't just about drawing maps. A map is usually a base that we build to show some data.

Let's turn this into a map of the world's airports. Actually, scratch that! Let's do something cooler. Let's make a map of CIA rendition flights out of the U.S. To do this, we'll still need the world's airports, as the airport values in the Rendition Project's dataset use the airport short codes, not latitude and longitude.

The first step is fetching the `airports.dat` dataset from `http://openflights.org/data.html` and the Rendition Project's U.S. flights data from `http://www.therenditionproject.org.uk/pdf/XLS%201%20-%20Flight%20data.%20US%20FOI%20resp.xls`. You can also find it in the examples on GitHub at `https://github.com/aendrew/learning-d3/blob/chapter3/src/data/airports.dat` and `https://github.com/aendrew/learning-d3/blob/chapter3/src/data/renditions.csv`, respectively. For the renditions dataset, you'll need to open in Excel and save as CSV. I've done that for you if you are going to grab it from GitHub.

First add two new promises, and update the `Promise.all()` call to complete once the two new datasets are available. Add a call to `addRenditions()` after `draw()`:

```
let p4 = new Promise((resolve, reject) => {
  d3.text('data/airports.dat', (err, data) => {
    err ? reject(err) : resolve(data);
  });
});

let p5 = new Promise((resolve, reject) => {
  d3.csv('data/renditions.csv', (err, data) => {
    err ? reject(err) : resolve(data);
  });
});

Promise.all([p1, p2, p3, p4, p5]).then((values) => {
  let [sea, land, cultural, airports, renditions] = values;
  draw(sea, land, cultural);

  addRenditions(airports, renditions);
});
```

The function loads the two datasets and then calls (the yet-nonexistent) `addRenditions` to draw them. We use `d3.text` instead of `d3.csv` for `airports.dat` because it doesn't have a header line, so we have to parse it manually.

In `addRenditions`, we first wrangle the data into JavaScript objects—airports into a dictionary by ID—and use that to get the latitude and longitude of each destination and arrival airport:

```
function addRenditions(_airports, renditions) {
  let airports = {},
    routes;

  d3.csv.parseRows(_airports).forEach(function (airport) {
    var id = airport[4];
    airports[id] = {
      lat: airport[6],
      lon: airport[7]
    };
  });

  routes = renditions.map((v) => {
    let dep = v['Departure Airport'];
    let arr = v['Arrival Airport'];
    return {
      from: airports[dep],
      to: airports[arr]
    };
  }).filter((v) => v.to && v.from).slice(0, 50);
}
```

We used `d3.csv.parseRows` to parse CSV files into arrays and manually turned them into dictionaries. The array indices aren't very legible, unfortunately, but they make sense when you look at the raw data:

```
1,"Goroka","Goroka","Papua New Guinea","GKA","AYGA",
    -6.081689,145.391881,5282,10,"U"
2,"Madang","Madang","Papua New Guinea","MAG","AYMD",
    -5.207083,145.7887,20,10,"U"
```

We then map each rendition flight so that we just have a dictionary of arrival and departure coordinates. We filter out any results where either `to` or `from` is missing, which are likely cases when our map function isn't able to match the airport short codes. Also, because it's a really big dataset and drawing all of it looks a bit messy, we've limited it to the first 50 objects in the array using `Array.prototype.slice`.

Next, we'll actually draw the lines, using our projection to translate the latitude and longitude coordinates into something that can fit on our screen:

```
let lines = chart.selectAll('.route')
  .data(routes)
  .enter()
    .append('line')
    .attr('x1', (d) => projection([d.from.lon, d.from.lat])[0])
    .attr('y1', (d) => projection([d.from.lon, d.from.lat])[1])
    .attr('x2', (d) => projection([d.to.lon, d.to.lat])[0])
    .attr('y2', (d) => projection([d.to.lon, d.to.lat])[1])
    .classed('route', true);
```

The routes won't show up until we style them. Add the following code to `index.css`:

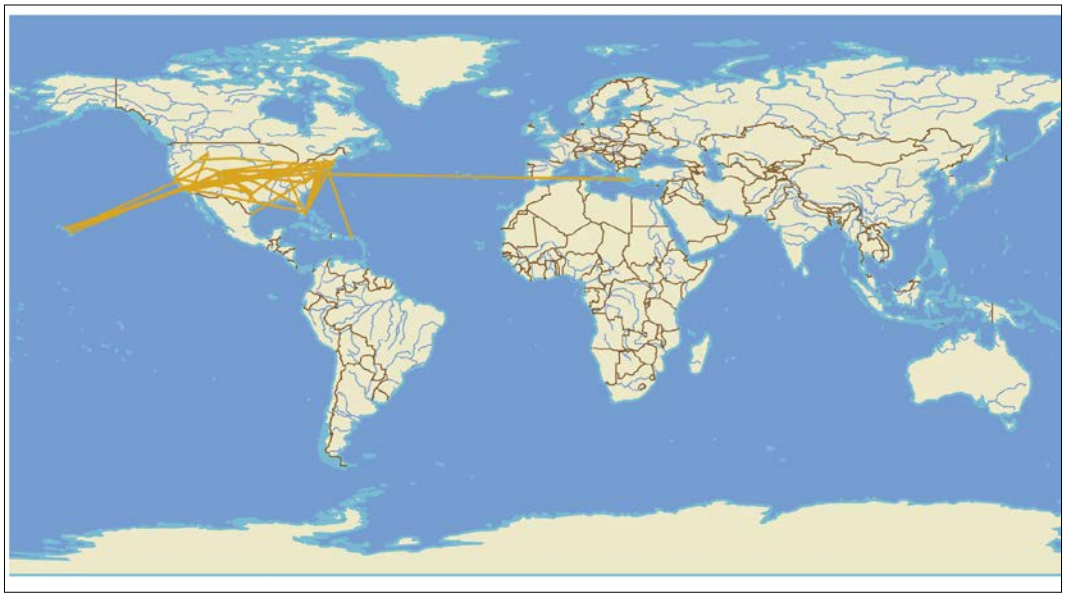
```
.route {
  stroke-width: 2px;
  stroke: goldenrod;
}
```

This screenshot displays the result:

Huh! That doesn't have much, beyond the one route represented by the black line near the middle. We've probably zoomed in too much. Let's tweak it a little. Go back to where we defined `projection` and set the scale to `200`:

```
let projection = d3.geo.equirectangular()
.center([8, 56])
.scale(200);
```

Hey! There we go! This is suddenly looking like the start of a piece of interactive news content!



We solved what is commonly referred to as the "too many markers" problem—that is, when zoomed out, data on a map looks cluttered—by simply limiting the amount of data that can be shown. This is admittedly a pretty cheap way out; a better workaround is to either cluster the map data (which wouldn't be all that easy with lines like the ones we have) or provide UI elements to toggle aspects of the dataset. We'll look at interactivity in the coming chapters; hold on to your hats!

Summary

You've made it through the chapter on data!

We really got to the core of what D3 is about—that is, data wrangling. While learning about data wrangling, you saw some interesting properties of prime numbers, learned all about loading external data, and effectively used scales to avoid calculations. We played with promises and generators along the way.

Lastly, we made a cool map to learn how simple geographic data can be once you get a hand on a good source and transform it into a better format.

4

Defining the User Experience – Animation and Interaction

Animation is like chilli sauce. A little goes a really long way and can really help to spice up a graphic while leading the viewer through the content; if there is too much, it's all anyone will notice. Good UX — short for user experience, which is one of the computer idioms you employ throughout your projects — is more like guacamole. If it's good, it's a nice subtle touch which will improve the overall quality of your output and make everyone happy; if it is bad, it will taint everything and ruin the whole burrito.

In this chapter, we'll discuss both animation and user interaction, with an eye towards using both to improve the quality of your data visualizations. We'll also use D3's behaviors to make that map from the last chapter look awesome. Throughout the chapter, we'll discuss why or why not animation or interactivity should be used in a particular scenario.

The ability to display data creatively with D3 is one of the best reasons for using it; interaction and animation allow you to not only display data but also *explain* data. How you use UX throughout your interface design determines whether you are building an *exploratory* graphic, wherein the user is given access to all of the data and has the ability to change how it's displayed through sorting, filtering and so on, or an *explanatory* graphic, where minimal interactivity guides the user through the relevant data. In reality, you'll probably mix both approaches, but understanding which type of interaction you want to have with the reader at what point is helpful when planning your projects.

We'll discuss the differences between these two approaches throughout this chapter.

Animation

The first question to ask is, why would animation improve this project?

If you're making something that isn't really designed to communicate data and is just designed to trip people out at your local warehouse rave, then "because it would make it look cool" is a totally valid response. Please don't let me discourage you from running rainbow color interpolators through that spiral in the last chapter if you think it'd be fun (because, speaking from personal experience, creating crazy animated art with D3 is a highly enjoyable use of a Saturday afternoon).

If, however, you're rendering data, a bit more consideration is probably necessary. What is your data doing? If it's a value increasing over time, animating a line going upwards from left-to-right makes more sense than fading in the line all at once.

Previously, we set attributes on our various SVG objects as we wanted them to appear once the image was finally rendered. Now, we'll use animation to guide viewers through our graphic, using the narrative focus it provides as a way of helping them interpret the data we're displaying. To do this, we need to animate the relevant properties of each SVG object.

Animation with transitions

D3 transitions are one way of accomplishing this. Transitions use the familiar principle of changing a selection's attributes, except that changes are applied over time.

To slowly turn a rectangle red, we use the following line of code:

```
d3.select('rect').transition().style('fill', 'red');
```

We start a new transition with `.transition()` and then define the final state of each animated attribute. By default, every transition takes 250 milliseconds; you can change the timing with `.duration()`. New transitions are executed on all properties simultaneously unless you set a delay using `.delay()`.

Delays are handy when we want to make transitions happen in sequence. Without a delay, they are all executed at the same time, depending on the internal timer. For single objects, nested transitions are much simpler than carefully calibrated delays.

We've already used these way back in *Chapter 1*, *Getting Started with D3, ES2016, and Node.js* — remember how our bar chart bars grew to accommodate the data? I've reproduced the relevant section from the following `BasicBarChart.js`:

```
        this.chart.selectAll('rect')
            .data(data)
            .enter()
```

```
        .append('rect')
        .attr('class', 'bar')
        .attr('x', (d) => x(d.name))
        .attr('width', x.rangeBand())
        .attr('y', () => y(this.margin.bottom))
        .attr('height', 0)
      .transition()
        .delay((d, i) => i*200)
        .duration(800)
        .attr('y', (d) => y(d.population))
        .attr('height', (d) => this.height - y(d.population))
```

We initialize the transition, set each datum's delay to be 200ms later than the last and then run the 800ms transition, increasing the `y` value and bar height to the bar's value.

If you want to do something before a transition begins or want to listen for it to end, you can use `.each()` with the appropriate event type. Add the following to the preceding code:

```
.style('fill', 'red')
.each('start', () => { console.log("I'm turning red!"); })
.each('end', () => { console.log("I'm all red now!"); })
```

This is handy when making instant changes before or after a transition. Just keep in mind that transitions run independently and you cannot rely on transitions outside the current callback being in any particular state.

Whoops, we've actually done something silly with our animation here. Even though it was a nifty demonstration of how to stagger animations using `.delay()`, you generally shouldn't do this with ordinal scale charts. Why? Staggering gives the impression that the *x* axis is changing over time — in fact, we're not charting time series data at all in this so it makes more sense for all the bars to rise at the same time. It's really subtle but these are the sorts of things worth considering as you begin to use animation.

Interpolators

To calculate values between the initial and final states of a transition, D3 uses a type of function called an interpolator, which maps the `[0,1]` domain to a target range, which can be a color, a number, or a string. These make it easy to blend between two values, because the interpolator will return the iterations between the values supplied to it. Under the hood, scales are based on these same interpolators.

D3's built-in interpolators can interpolate between almost any two arbitrary values, most often between numbers or colors, but also between strings. This sounds odd at first but it's actually pretty useful. To let D3 pick the right interpolator for the job, we just write `d3.interpolate(a, b)` and the interpolation function is chosen depending on the type of `b`. `a` is the initial value, and `b` is the final value.

If `b` is a number, `a` will be coerced into a number and `.interpolateNumber()` will be used. You should avoid interpolating to or from a zero value because values will eventually be transformed into a string for the actual attribute and very small numbers might turn into scientific notation. CSS and HTML don't quite understand 1e-7 (the digit 1 with seven zeroes before the decimal place), so the smallest number you can safely use is 1e-6.

If `b` is a string, D3 checks whether it's a CSS color, in which case it is transformed to a proper color, just like the ones in *Chapter 2*, *A Primer on DOM, SVG, and CSS*. `a` is transformed into a color as well and then D3 uses `.interpolateRgb()` or a more appropriate interpolator for your color space.

Something even more amazing happens when the string is not a color. D3 can handle that too! When it encounters a string, D3 parses it for numbers, then uses `.interpolateNumber()` on each numerical piece of the string. This is useful for interpolating mixed style definitions.

For instance, to transition a font definition, you might do something like this:

```
d3.select('svg')
.append('text')
.attr({x: 100, y: 100})
.text("I'm growing!")
.transition()
.styleTween('font', () =>
 d3.interpolate('12px Helvetica', '36px Comic Sans MS'));
```

We used `.styleTween()` to manually define a transition. It is most useful when we want to define the starting value of a transition without relying on the current state. The first argument defines which style attribute to transition and the second is the interpolator.

You can use `.tween()` to do this for attributes other than the style.

Every numerical part of the string was interpolated between the starting and ending values and the string parts changed to their final state immediately. An interesting application of this is interpolating path definitions—you can make shapes change in time. How cool is that?

Keep in mind that only strings with the same number and location of control points (numbers in the string) can be interpolated. You can't use interpolators for everything. Creating a custom interpolator is as simple as defining a function that takes a single `t` parameter and returns the start value for `t = 0` and the end value for `t = 1` and blends values for anything in between.

For example, the following code shows the `interpolateNumber` function of D3:

```
function interpolateNumber(a, b) {
  return function(t) {
    return a + t * (b - a);
  };
}
```

It's as simple as that!

You can even interpolate whole arrays and objects, which work like compound interpolators of multiple values. We'll use those soon.

Easing

Easing tweaks the behavior of interpolators by controlling the time (`t`) argument. We use this to make our animations feel more natural, to add some bounce elasticity, and so on. Mostly, we use easing to avoid the artificial feel of linear animation.

Let's make a quick comparison of the easing functions provided by D3 and see what they do.

First create the file `chapter4.js` and a new class that extends `BasicChart`. You know the drill.

```
import {BasicChart} from './basic-chart';

export class chapter4 extends BasicChart {
  constructor(data) {
    super(data);

  }
}
```

Next, we need an array of easing functions and a scale to place them along the vertical axis. Put this in the constructor under `super(data)`:

```
let eases = ['linear', 'poly(4)', 'quad', 'cubic', 'sin', 'exp',
'circle', 'elastic(10, -5)', 'back(0.5)', 'bounce', 'cubic-in',
'cubic-out', 'cubic-in-out', 'cubic-out-in'],
y = d3.scale.ordinal().domain(eases).rangeBands([50, 500]);
```

You'll notice that `poly`, `elastic`, and `back` take arguments since these are just strings so we'll have to change them into real arguments manually later. The `poly` easing function is just a polynomial, so `poly(2)` is equal to **quad** and `poly(3)` is equal to **cubic**. Or, for those of us who stopped paying attention towards the end of our secondary school math, the higher the `poly` argument value, the deeper the curve — for instance, `poly(4)` (equivalent to **quart**) has a fair bit of delay at the beginning, the end or both, depending on where you set the easing (see below). The higher the number, the more dramatic the delay. Have a play with it, do what feels right.

The `elastic` easing function simulates a rubber band and the two arguments control tension. You should play with the values to get the effect you want. The `back` easing function is supposed to simulate backing into a parking space. The argument controls how much overshoot there's going to be.

The easings at the end (`cubic-in`, `cubic-out`, and so on) are functions that we create ourselves by combining the following modifiers:

- `-in`: It does nothing
- `-out`: It reverses the easing direction
- `-in-out`: It copies and mirrors the easing function from `[0, 0.5]` and `[0.5, 1]`
- `-out-in`: It copies and mirrors the easing function from `[1, 0.5]` and `[0.5, 0]`

You can add these to any easing function so play around. Now, we're going to render a bunch of circles animated using each easing:

```
eases.forEach((ease) => {
  let transition = svg.append('circle')
    .attr({cx: 130, cy: y(ease), r: y.rangeBand()/2-5})
    .transition()
    .delay(400)
    .duration(1500)
    .attr({cx: 400});
});
```

We loop over the list with an iterator that creates a new circle and uses the `y()` scale for vertical placement and `y.rangeBand()` for the circle size. In this way, we can easily add or remove examples. Transitions start with a delay of just under half a second to give us a chance to see what's going on. A duration of `1500` milliseconds and a final position of `400` should give us enough time and space to see the easing.

We define the easing at the end of this function, before the `});` section, as shown:

```
if (ease.indexOf('(') > -1) {
  let args = ease.match(/[0-9]+/g),
    type = ease.match(/^[a-z]+/);
  transition.ease(type, args[0], args[1]);
} else {
  transition.ease(ease);
}
```

This code checks for parentheses in the `ease` string, parses out the easing function and its arguments, and feeds them to `transition.ease()`. Without parentheses, `ease` is just the easing type.

Let's add some text so that we can tell the examples apart:

```
svg.append('text')
.text(ease)
.attr({x: 10, y: y(ease)+5});
```

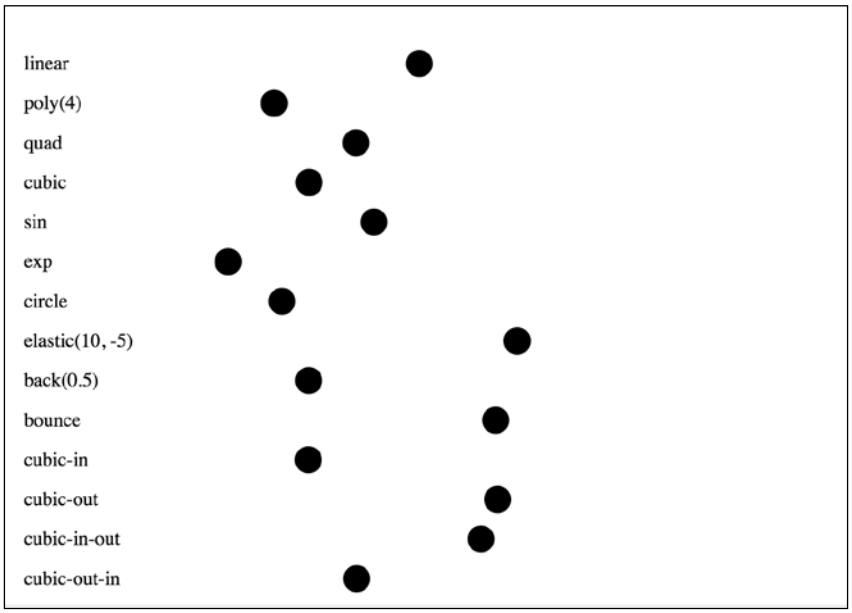
Replace `index.js` with the following:

```
import {chapter4} from './chapter4';
new chapter4();
```

Ensure that the server is running (`$ npm start` if not) and visit `http://127.0.0.1:8080`.

In order to avoid repeating the preceding four lines over and over again, I'll take it on faith that you know how to import and instantiate the classes we're creating at this point, in addition to knowing how to load the developer server. I'll therefore leave out that part from here on in.

The visualization is a cacophony of dots:



The screenshot doesn't quite showcase the animation so you should try this one in the browser. You could also take a look at the easing curves at `http://easings.net/`.

Easings are a nice finishing touch to put on most animations. Most things in the real world don't have constant acceleration; a good rule of thumb is to match whichever element you're animating with an easing appropriate for its size in relation to the page. In other words, small elements should generally move faster than large elements and have tighter `-in` and `-out` easings. The key thing is to think about how stuff should logically move instead of just slapping a 1-second fade-in on everything.

Timers

D3 uses timers to schedule transitions. Even an immediate transition will start after a delay of 17ms.

Far from keeping timers all to itself, D3 lets us use timers so that we can take animation beyond the two-keyframe model of transition. If you are not familiar with animation terminology, keyframes define the start or end of a smooth transition.

We use `d3.timer()` to create a timer. It takes a function, a delay, and a starting mark. After the set delay (in milliseconds) from the mark, the function will be executed repeatedly until it returns `true`. The mark should be a date converted into milliseconds since Unix timestamp (`Date.getTime()` is sufficient), or you can let D3 use `Date.now()` by default.

Let's animate the drawing of a parametric function to work just like the Spirograph toy you might have had as a kid.

We'll create a timer, let it run for a few seconds, and use the millisecond mark as the parameter for a parametric function.

Create a new class in `chapter4.js`:

```
export class Spirograph extends BasicChart {
  constructor(data) {
    super(data);
  }
}
```

Here's a good function from Wikipedia's article on parametric equations at `http://en.wikipedia.org/wiki/Parametric_equations`:

```
let position = (t) => {
  let a = 80, b = 1, c = 1, d = 80;
  return {x: Math.cos(a*t) - Math.pow(Math.cos(b*t), 3),
          y: Math.sin(c*t) - Math.pow(Math.sin(d*t), 3)};
};
```

This function returns a mathematical position based on the parameter, going from zero up. You can tweak the Spirograph by changing the `a`, `b`, `c`, and `d` variables — there are examples in the same Wikipedia article.

This function returns positions between `-2` and `2`, so we need scales to make it visible on the screen:

```
let tScale = d3.scale.linear().domain([500, 25000])
  .range([0, 2*Math.PI]),
  x = d3.scale.linear().domain([-2, 2]).range([100, this.width-100]),
  y = d3.scale.linear().domain([-2, 2]).range([this.height-100,100]);
```

`tScale` translates time into parameters for the function; `x` and `y` calculate the final position on the image.

Now we need to define `brush` to fly around and pretend that it's drawing and also a variable to hold the `previous` position so that we can draw straight lines:

```
let brush = chart.append('circle')
  .attr({r: 4}),
  previous = position(0);
```

Next, we need to define an animation `step` function to move the brush and draw a line between the previous and current points:

```
let step = (time) => {
  if (time > tScale.domain()[1]) {
    return true;
  }

  let t = tScale(time),
    pos = position(t);

  brush.attr({cx: x(pos.x), cy: y(pos.y)});

  this.chart.append('line')
      .attr({x1: x(previous.x),
        y1: y(previous.y),
        x2: x(pos.x),
        y2: y(pos.y),
        stroke: 'steelblue',
        'stroke-width': 1.3});

  previous = pos;
};
```

The first condition stops the timer when the current value of the `time` parameter is beyond the domain of `tScale`. Then, we use `tScale()` to translate the time into our parameter and get a new position for the brush.

Then, we move the brush — there is no transition because we are performing the transition ourselves already — and draw a new steel blue line between the previous position and the current position (`pos`).

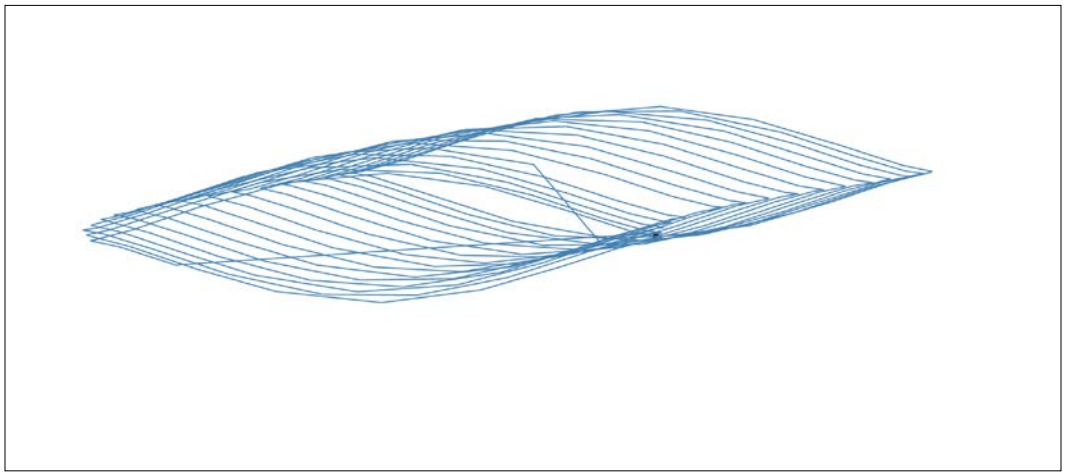
We conclude by setting a new value for the previous position.

All that's left now is to create a timer:

```
let timer = d3.timer(step, 500);
```

That's it. Half a second after a page refresh, the code will begin drawing a beautiful shape and finish 25 seconds later.

Starting out, it looks like this:



Getting the whole picture takes a while so this isn't the best way to draw Spirographs. Since we're using time as a parameter, a smoother curve (with more points) takes more time.

Animation with CSS transitions

Another way of animating things with D3 is by using CSS transitions. If you simply wish to animate a `transform` on an element (particularly if you don't need a lot of control over sequencing), CSS transitions are easier. They have the added benefit of being much better for performance due to not forcing the browser to repaint on every tick, which means that the GPU does all the work and everything runs much smoother. Lastly, you can use CSS media queries to target how an animation works on different devices.

Let's make a basic bar chart and use CSS transitions to do the heavy lifting.

Create a new class in `chapter4.js`, as shown here:

```
export class PrisonPopulationChart extends BasicChart {
  constructor(data) {
    super(data);
  }
}
```

And fill it with the following basic bar chart code, It seems like a lot but is actually really similar to what we used way back in `chapter1.js`:

```
export class PrisonPopulationChart extends BasicChart {
  constructor(path) {
    super();

    this.margin.left = 50;
    let d3 = require('d3');

    let p = new Promise((res, rej) => {
      d3.csv(path, (err, data) => err ? rej(err) : res(data));
    });

    require('./index.css');

    this.x = d3.scale.ordinal().rangeBands([this.margin.left,
this.width], 0.1);

    p.then((data) => {
      this.data = data;
      this.drawChart();
    });

    return p;
  }

  drawChart() {
    let data = this.data;
    data = data.filter((d) => d.year >= d3.min(data, (d) =>
d.year) && d.year <= d3.max(data, (d)=> d.year));

    this.y = d3.scale.linear().range([this.height,
this.margin.bottom]);
    this.x.domain(data.map((d) => d.year));
    this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

    this.xAxis =
d3.svg.axis().scale(this.x).orient('bottom').
tickValues(this.x.domain().filter((d, i) => !(i % 5)));
    this.yAxis = d3.svg.axis().scale(this.y).orient('left');

    this.chart.append('g')
        .classed('axis x', true)
        .attr('transform', `translate(0, ${this.height})`)
```

```
            .call(this.xAxis);

        this.chart.append('g')
            .classed('axis y', true)
            .attr('transform', `translate(${this.margin.left}, 0)`)
            .call(this.yAxis);

        this.bars = this.chart.append('g').classed('bars',
    true).selectAll('rect')
            .data(data)
            .enter()
            .append('rect')
            .style('x', (d) => {
              return this.x(d.year);
            })
            .style('y', () => this.y(0))
            .style('width', this.x.rangeBand())
            .style('height', 0);

        // Run CSS animation
        setTimeout(()=> {
          this.bars.classed('bar', true)
          .style('height', (d) => this.height - this.y(+d.total) )
            .style('y', (d) => this.y(+d.total));
        }, 1000);
      }
    }
```

The only thing that's a bit surprising this time is that we load the data using a promise and then return it from the constructor. This is a bit tricky — what's being returned isn't the resolved promise data but, rather, the promise object itself. The resolved data is taken care of in the constructor when we use `.then()` and attached to the class as a property. We will use this ability to return unresolved promises later on in the chapter to let our child classes know when the data has resolved.

We're using a dataset of the UK prison population from 1900 to 2015, which is available at `https://github.com/aendrew/learning-d3/blob/chapter4/src/data/uk_prison_data_1900-2015.csv` or in the `src/` data folder of the book's repo.

Change `index.js` to look like this:

```
import {PrisonPopulationChart} from './chapter4';
new PrisonPopulationChart('data/uk_prison_data_1900-2015.csv');
```

You might have noticed that we now only use the constructor to load in data; all the D3 work is in the `drawChart()` class method. Although the earlier chapters were pretty light on object-oriented programming concepts, we're going to start writing tighter classes now that we're dealing with user interaction. Breaking down a project into smaller functions is important for several reasons: it helps prevent you repeating yourself when constructing the stages of a user's journey and it allows each piece of the project to be more easily tested. We'll get into automated testing later but, for now, start thinking of ways you can construct your classes so that each major operation is in its own method.

As a taste of some of the cool things ES2015 classes allow you to do, did you see how I set the margin to a new value in the constructor? Our `BasicChart` class is smart enough to pick up the new value and adjust the chart created by our parent class accordingly.

After the constructor, we move onto `drawChart()`. The first part is setting up the scales and axes in the same way as before:

```
let data = this.data;

this.y = d3.scale.linear().range([this.height,
  this.margin.bottom]);
this.x.domain(data.map((d) => d.year));
this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

this.xAxis = d3.svg.axis().scale(this.x).orient('bottom')
  .tickValues(this.x.domain().filter((d, i) => !(i % 5)));

this.yAxis = d3.svg.axis().scale(this.y).orient('left');

this.chart.append('g')
    .classed('axis x', true)
    .attr('transform', `translate(0, ${this.height})`)
    .call(this.xAxis);

this.chart.append('g')
    .classed('axis y', true)
    .attr('transform', `translate(${this.margin.left}, 0)`)
    .call(this.yAxis);
```

If you hadn't noticed, we defined the `x` scale back up in the constructor:

```
this.x = d3.scale.ordinal().rangeBands([this.margin.left,
this.width], 0.1);
```

This is so that we get immediate access to it from a child class that we will create later on in the chapter. It's a standard ordinal scale using `.rangeBands`. We also filter the *x* axis ticks so we get a tick every 5 years.

Pop-quiz! Why did we use an ordinal scale for years instead of a time or linear scale?

While we could have used either, an ordinal scale makes the most sense because we're creating a bar chart of years — time and linear scales will render the tick on the left edge of the bar, which looks silly.

Then, we set all the bars:

```
this.bars = this.chart.append('g').classed('bars', true)
  .selectAll('rect')
    .data(data)
    .enter()
    .append('rect')
    .style('x', (d) => this.x(d.year))
    .style('y', () => this.y(0))
    .style('width', this.x.rangeBand())
    .style('height', 0);
```

This puts them into a group element with the class `.bars`. I've set the `y` value to zero as well as the height. This sets the initial state or the first keyframe. Next, we added the following right after that:

```
// Run CSS animation
setTimeout(()=> {
  this.bars.classed('bar', true)
  .style('height', (d) => this.height - this.y(d.total) )
    .style('y', (d) => this.y(d.total));
}, 1000);
```

This adds the `.bar` class to our bars and sets both its height and `y` value to their end keyframe values. Note how we've used `.style()` instead of `.attr()` — this is so that the new values are recorded in the `style` attribute tag, which means that our CSS transition will affect it. We do this separately in a `setTimeout` delay because we don't want the transition to be applied on the initial render and, without the delay, the user won't see the initial state.

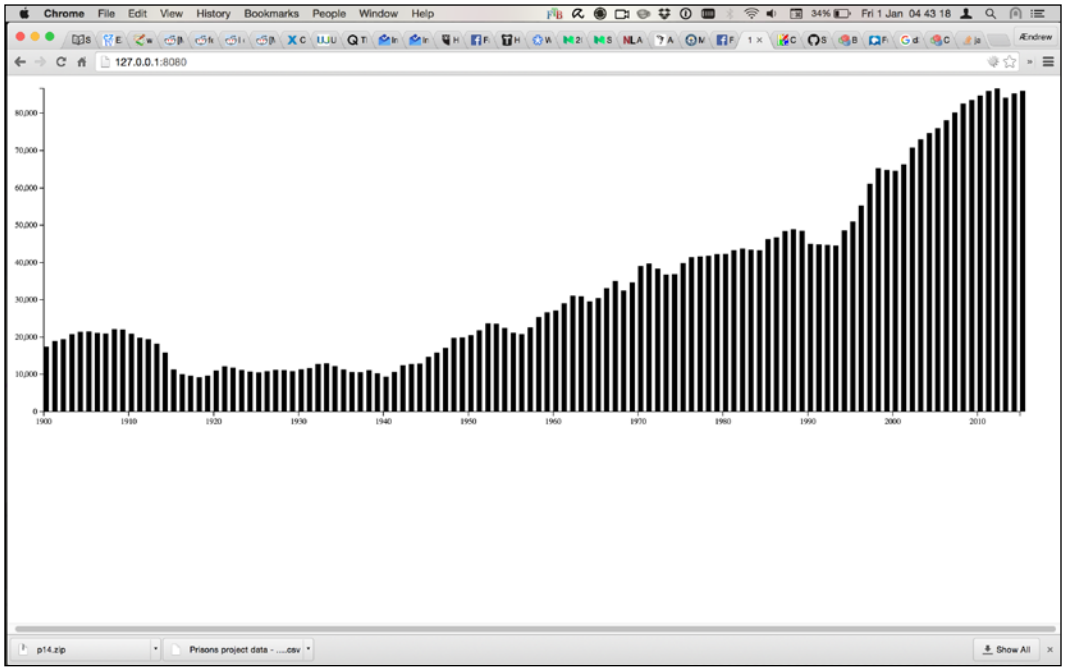
Lastly, add the following to `index.css`:

```
.bar {
  transition: all 1s ease-out;
}
```

This simply tells the web browser to animate any style property changes over one second and use the `ease-out` easing.

You should now have a CSS-driven bar chart!

For more on the CSS transitions API, visit this Mozilla Developer Network article: `https://developer.mozilla.org/Web/CSS/CSS_Transitions/Using_CSS_transitions`.



When should you use CSS transitions instead of the D3 transition API? There are two big use cases where it really makes a lot of sense: when you have to animate a lot of SVG elements and when your audience is viewing your work on a cellphone. It's often really difficult to create smooth animations using JavaScript-based interpolations on mobile devices because several things affect execution — most notably page scroll, which has made making fancy long form-style articles particularly nightmarish to get working on a mobile device in the last few years (though, luckily, this has changed as of iOS 8).

Generally, D3's transition API is easier to use, more powerful and doesn't mix animation logic with styling logic. It's also pretty fast — you shouldn't be afraid to use it by default, you can switch particular features to CSS transitions if you feel the performance benefit would improve the user experience.

Lastly, it makes a lot of sense to animate user interface interactivity using CSS transitions as UI elements tend to be simpler than charts and their animations shouldn't impact on performance when animating the SVG graphic.

What other ways are there to animate SVG? So glad you asked!

In the beginning, there was **SMIL** or **Synchronized Multimedia Integration Language**. To use SMIL, you use `<animate>` tags, which you put in your SVG markup. If this sounds gross already, it is. Luckily, Chrome 45 depreciated it, which means you never have to worry about learning it. The morbidly curious can visit: `https://mdn.io/SVG_animation_with_SMIL`.

In the future, an alternative to CSS transitions will be the Web Animations API. It's currently not supported by anything from Microsoft or Apple but it's been in Chrome since version 36 and Firefox since version 41. It has the *nicest* syntax, is all based on JavaScript and works well with D3. Here's an example:

```
d3.selectAll('.bar').each(function(d, i){
  this.animate([
    {transform: `translate(${x(i)}, ${y(d)})`}
  ],{
    duration: 1000,
    iterations: 5,
    delay: 100
  });
});
```

Alas, as always, web development is a toy chest full of things you can't reliably use *just quite yet*. If you want to play with it while it's still being standardized, check out the fantastic polyfill at `https://github.com/web-animations/web-animations-js` that enables the use of web animations in most modern browsers. I'll hopefully be able to write more about the Web Animations API in the next edition of this book!

Interacting with the user

This is it. This is where all of the UX tidbits that I've been dropping throughout the chapter and all the ES2015 ideas that you've been learning come together — let's make a simple explanatory graphic that uses interaction to walk the viewer through data.

The first step in any visualization involving user interactivity is to plan exactly what you want the visualization to do, how you want your viewers to interact with it, and what you want to say about the data. What is the data's story? What's the best way to tell it?

We have the numerical product of over a century of incarceration in a western country in the prison population dataset. There are many ways we can look at this data. We can look at how the prison population has risen versus the overall population growth or we can look at how the prison population has risen or fallen in relation to known historical events. Often, you'll need more than one chart — for instance, when I used this data in a project for *The Times*, the piece had no less than five charts and one map, with the reader being walked through each graphic in sequence. This is where we start to get into actual data journalism territory, which is far beyond the scope of this short section. Suffice to say, however, that it helps to write down these things in either bullet point or paragraph form before you start writing any code. The real work is often done long before the first line of JavaScript is ever written.

In this particular instance, because we have a century of data, we're going to look at a few notable historical points. The graphic will have five states which will be navigated through a series of five buttons:

1. Initial view — years 1900 to 2015. This provides a general overview of how the prison population has risen over time.
2. Zoom 1900 to 1930. Highlights 1914–1918. The text explains how the population rose due to the end of World War I.
3. Zoom 1930 to 1960. Highlights 1939-1945. The text explains how the population rose after World War II.
4. Zoom 1960 to 1990. Discusses the rise of the consumer society.
5. Zoom 1990 to 2015. Highlights 1993 and explains the sharp rise after the murder of James Bulger.

We're keeping the user interface deliberately simple but remember that simpler is often better, particularly when building for an audience on mobile devices (sliders are much harder to use on touch devices than buttons, for instance).

Basic interaction

Much like elsewhere in JavaScript land, the principle for interaction is simple—attach an event listener to an element and do something when it's triggered. We add and remove listeners to and from selections with the `.on()` method, an event type (for instance, `click`), and a listener function that is executed when the event is triggered.

We can set a capture flag which ensures our listener is called first and all other listeners wait for our listener to finish. Events bubbling up from children elements will not trigger our listener.

You can rely on the fact there will only ever be a single listener for a particular event on an element because old listeners for the same event are removed when new ones are added. This is very useful when trying to eliminate unpredictable behavior.

Just like other functions acting on element selections, event listeners get the current datum and index and set the `this` context to the DOM element. The global `d3.event` lets you access the actual event object.

We're going to create a new class that extends our last chart:

```
export class InteractivePrisonPopulationChart extends
PrisonPopulationChart {
  constructor(path) {
    let p = super(path);
    this.scenes = require('./data/prison_scenes.json');
    this.scenes.forEach(
      (v, i) => v.cb = this['loadScene' + i].bind(this));

    p.then(() => this.addUIElements());
  }
}
```

Creating a scaffold like this is really useful when starting a more involved project like this. Additionally, I have pulled out the chapter data from a JSON file and assigned a callback function to each scene. You can often get away with having just one `draw` function that does something clever to load each scene but it's often easier to start out thinking in terms of discrete segments so that you don't end up with a single function with a ton of conditional logic.

Pay attention to the `.bind(this)` call in that ugly `forEach` loop I wrote. Without that, we wouldn't have access to our class methods via `this` in the scene functions!

A good rule of thumb when writing ES2015 classes is: when in doubt and throwing a lot of `TypeErrors`, check whether `this` is what you think it should be!

We gain the advantage of having all that work we just did relegated to the `super()` call because we're extending `PrisonPopulationChart`, in effect concentrating all the rendering code in a separate class. You don't necessarily need to create a totally separate class for interaction; I did it here to simplify everything. Also, note how we assign whatever the parent constructor returns to a variable? This is because we returned a promise in the parent constructor, which allows us to use `.then()` to ensure that the data is loaded and attached to `this.data` before we continue.

Let's start by giving ourselves some space underneath the chart. Add the following to the end of the constructor:

```
this.height = window.innerHeight / 2;
this.chart.attr('height', this.height);
this.svg.attr('height', this.height + 50);
this.margin.right = 10;
this.margin.bottom = 10;
```

This gives us roughly half of the window to work with. You'll want to experiment a bit to see what the best combination is or possibly set the element size using CSS media queries.

Next, let's create our UI elements — in this case, five buttons. Add a function called `addUIElements`, shown as follows:

```
addUIElements() {
  this.buttons = d3.select('#chart')
    .append('div')
    .classed('buttons', true)
      .selectAll('.button')
      .data(this.scenes).enter()
      .append('button')
      .classed('scene', true)
      .text((d) => d.label)
      .on('click', d => d.cb())
      .on('touchstart', d => d.cb());

  this.words = d3.select('#chart').append('div');
  this.words.classed('words', true);
}
```

There is nothing new here — create a new button for each element in the chapters array and run its callback function when any button is clicked or tapped. We also drop a plain ol' `div` into the chart area, which is where we'll put all of our text describing each scene.

Haven't run into `touchstart` before? Think of it as the `mousedown` event of touch. Other useful touch events are `touchmove`, `touchend`, `touchcancel`, and `tap`. Mozilla's documentation explains touch events in more detail at `https://developer.mozilla.org/Web/Guide/API/DOM/Events/Touch_events`.

We need a function to clear selected bars — let's do something a little bit different in terms of sequencing animation and use promises. Although D3 doesn't use promises for animation natively (or for much in general, really), it's an idiom becoming widely used throughout the JavaScript world, particularly by the Angular 2 community. Add the following method to your class:

```
clearSelected() {
  return new Promise((res, rej) => {
    d3.selectAll('.selected').classed('selected', false);
    res();
  });
}
```

This returns a new promise, which resolves after we've removed the `.selected` class from all the bar elements.

We also need a method to select specific bars:

```
selectBars(years) {
  this.bars.filter((d) => years.indexOf(
   Number(d.year)) > -1).classed('selected', true);
}
```

We now need to create our chart's update function. Let's make it return a promise like our animation functions:

```
updateChart(data = this.data) {
  return new Promise((res, rej) => {
    let bars = this.chart.selectAll('.bar').data(data);

    this.x.domain(data.map((d) => d.year));
    this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

    this.chart.selectAll('.axis.x').call(
      d3.svg.axis().scale(this.x).orient('bottom')
     .tickValues(this.x.domain().filter((d, i) => !(i % 5))));
    this.chart.selectAll('.axis.y')
      .call(this.yAxis);
```

```
    // Update
    bars.style('x', (d) => this.x(d.year))
      .style('width', this.x.rangeBand())
      .style('height', (d) => this.height - this.y(+d.total) )
      .style('y', (d) => this.y(+d.total))

    // Add
    bars.enter()
      .append('rect')
      .style('x', (d) => this.x(+d.year))
      .style('width', this.x.rangeBand())
      .style('height', (d) => this.height - this.y(+d.total) )
      .style('y', (d) => this.y(+d.total))
      .classed('bar', true);

    // Remove
    bars.exit().remove();

    res();
  });
}
```

We do the standard D3 update, add and remove routine, setting the scales to the selected data. Finally, we resolve the promise by calling `res()` at the end. Note how we take an argument which defaults to the class `data` property.

Default argument values are a new feature in ES2016! This is such a nice addition to the language, as somebody who has written so much existence checking logic for basic functions it'd make you cry.

Let's look at that again:

```
updateChart(data = this.data)
```

This means the `data` argument will equal the class' internal `data` property if no argument is supplied. The next time that you feel yourself reaching for `typeof argumentVar !== 'undefined'`, give default arguments a try.

Lastly, let's set up our states. The first one is easy:

```
loadScene0() {
  this.clearSelected().then(() => this.updateChart());
  this.words.html('');
}
```

We clear the selected bars and then update the chart. We set `words` to be an empty string. We need to clear both the selected bars and the words on the first scene even if they originate in that way because we're allowing the viewer to navigate the scenes in a non-linear order — they can always come back to the first scene, which means we have to clear anything created by another scene:

```
loadScene1() {
  let scene = this.scenes[1];
  this.clearSelected().then(() => {
    this.updateChart(this.data.filter((d) =>
      d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.
year)) > -1))
        .then(() => this.selectBars(d3.range(1914, 1918)));
  });
  this.words.html(scene.copy);
}
```

This is somewhat more interesting. We do the same thing as we did in the first scene but, in the `then()` callback, we create a range from the domain given in the scene array. This gives us an array of the years with which we can filter the bars. Lastly, we set the text in the interactive dialog (helpfully written by my colleague, Sam Joiner) to form a meaningful sentence explaining the change.

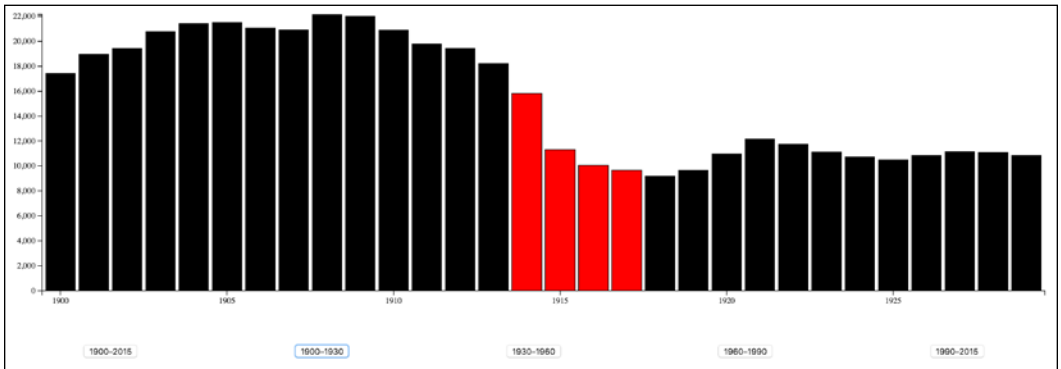
Copy is what people in the newspaper and advertising businesses call text content. I don't mean it in the duplicate sense here!

The next bit is rather repetitive:

```
loadScene2() {
  let scene = this.scenes[2];
  this.clearSelected().then(
    () => {
      this.updateChart(this.data.filter((d) =>
d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.year))
> -1))
        .then(() => this.selectBars(d3.range(1939, 1945)));
  });
  this.words.html(scene.copy);
}

loadScene3() {
  let scene = this.scenes[3];
  this.clearSelected().then(
```

```
      () => this.updateChart(this.data.filter((d) =>
d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.year))
> -1))
    );
    this.words.html(scene.copy);
}

loadScene4() {
    let scene = this.scenes[4];
    this.clearSelected().then(
      () => {
        this.updateChart(this.data.filter((d) =>
d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.year))
> -1))
          .then(() => this.selectBars([1993]));
      }
    );
    this.words.text(scene.copy);
}
```

And there you have it — your first interactive data visualization!



Prison numbers fall, and then – as men return from the battlefields of the First World War – start to rise to new heights.

Behaviors

In the last section, we created an *explanatory* graphic that used interaction to guide the user through the data. Often, however, the goal is just to make a dataset interactive and give the user some way of manipulating it, in other words, an *exploratory* graphic.

D3's behaviors save a boatload of time in setting up the more complex interactions in a chart. Additionally, they're designed to handle differences in input devices so you only have to implement a behavior once to have it work both with a mouse and on touch devices. The two currently supported behaviors are drag and zoom, both of which will get you pretty far.

Drag

Instead of having the user click buttons in the last example, what if we just let them drag the chart area to see the UK's prison population increase? It involves a bit more work from the user but it also gives them the ability to navigate freely through the chart — which may be desirable in some circumstances.

Let's extend the last chart we created and override the `.addUIElements` method. Create a child class that extends `InteractivePrisonPopulationChart` and set it up as shown below, thus overriding the parent `.addUIElements` method and avoiding a bunch of unnecessary buttons.

```
export class DraggableInteractivePrisonChart extends
InteractivePrisonPopulationChart {
  constructor(path) {
    let p = super(path);
    this.x.rangeBands([this.margin.left, this.width * 4]);
  }
  addUIElements() {}
}
```

The first thing that we need to do is create a hit box — we want dragging on the bars area to result in the bars being dragged. Alas, `g` container elements aren't clickable so we need to get the dimensions of `g.bars`, placing an invisible `rect` element of that size on top of it. Yet further alas, the dimensions of `g.bars` won't make any sense until the CSS transition has finished. Luckily, we can listen to the `transitionend` event to see if this has occurred. Add this to the `addUIElements` function, as shown here:

```
  let bars = d3.select('.bars').on('transitionend', ()=> {
    let dragContainer = this.chart.append('rect')
      .classed('bar-container', true)
      .attr('width', bars.node().getBBox().width)
      .attr('height', bars.node().getBBox().height)
      .attr('x', 0)
      .attr('y', 0)
      .attr('fill-opacity', 0);

});
```

`SVGElement.getBBox()` is a tremendously useful function that gives you the `x`, `y`, height and width of a particular element. Note, however, that you can't use this on D3 selections — only SVG elements! That's why we get the underlying element out of the selection by using `selection.node()`.

Now we need to set up the drag behavior. Still inside the `transitionend` callback, add the following:

```
let drag = d3.behavior.drag().on('drag', () => {
  let barsTransform = d3.transform(bars.attr('transform'));
  let xAxisTransform = d3.transform(
    d3.select('.axis.x').attr('transform'));
  bars.attr('transform',
    `translate(${barsTransform.translate[0] + d3.event.dx}, 0)`);
  d3.select('.axis.x').attr('transform',
    `translate(${xAxisTransform.translate[0] + d3.event.dx},
               ${xAxisTransform.translate[1]})`);
});
```

Firstly, we get the current transform values of the *x* axis and the bars themselves. `d3.transform()` is a very helpful function for getting a transform matrix into a useful object form so we use that twice. We then translate both the axes and bars their current translation distance, plus the distance the user drags across our hit box, provided by `d3.event.x`. We also provide the `y` translate of the axis since we set that way back up in our parent class.

Finally, after our call above but still inside the `transitionend` event callback, add the following to instantiate the drag behavior on our container.

```
dragContainer.call(drag);
```

Now you can drag!

You may have noticed that the bars drag behind the *x* axis which looks a bit unsightly. You can't add backgrounds to `g` elements as you can with `div` so you'll have to append a white `rect` element to the axis. The technique for getting the `rect` width and height is the same as it is for the hit box — I'll leave fixing this as an exercise for you.

Zoom

Despite the name, the zoom behavior lets you do more than just zoom—you can also pan! Like the drag behavior, zoom automatically handles both mouse and touch events and then triggers the higher-level zoom event. Yes, this means pinch-to-zoom works! That's pretty awesome, if you ask me.

Remember that map from *Chapter 3*, *Making Data Useful*, the one with the rendition flights?

Let's commit a crime against computational efficiency and make it zoom and pan.

I am warning you that this will be very rudimentary and painfully slow. This is not how you'd make a real explorable map, just an example to let us play with zooming. In real life, you would use tiling, progressive detailing, and other tricks. You also wouldn't write everything in the constructor, as we did in this example.

After even one example organizing all the functions into class methods, doesn't this already feel atrociously messy? Hopefully, you can see why it's so much better to organize intelligently, plan and organize your classes — you can always revisit a project and being able to pick your code back up and understand it as quickly as possible is both enormously important and very much facilitated by writing clean and extensible code.

Let's go back to the map. Jump to the end of the `GeoDemo` `draw` function in `chapter3.js` and add a call to `zoomable`; we'll define this next.

```
zoomable();
```

While you're in `draw`, turn off the river and cities layers to help with performance.

`zoomable` sets up the behavior on the chart so put this at the end of the constructor. Next, we'll define what the behavior actually is:

```
function zoomable() {
  chart.call(
    d3.behavior.zoom()
    .translate(projection.translate())
    .scale(projection.scale())
    .on('zoom', () => onzoom())
  );
}
```

We defined a zoom behavior with `d3.behavior.zoom()` and immediately called it on the whole image.

We set the current `.translate()` vector and `.scale()` to whatever the projection was using. The `zoom` event calls our `onzoom` function.

Let's define it as follows:

```
function onzoom() {
  projection
    .translate(d3.event.translate)
    .scale(d3.event.scale);

  d3.selectAll('path')
  .attr('d', d3.geo.path().projection(projection));

}
```

Firstly, we told our projection that the new translation vector was in `d3.event.translate`.

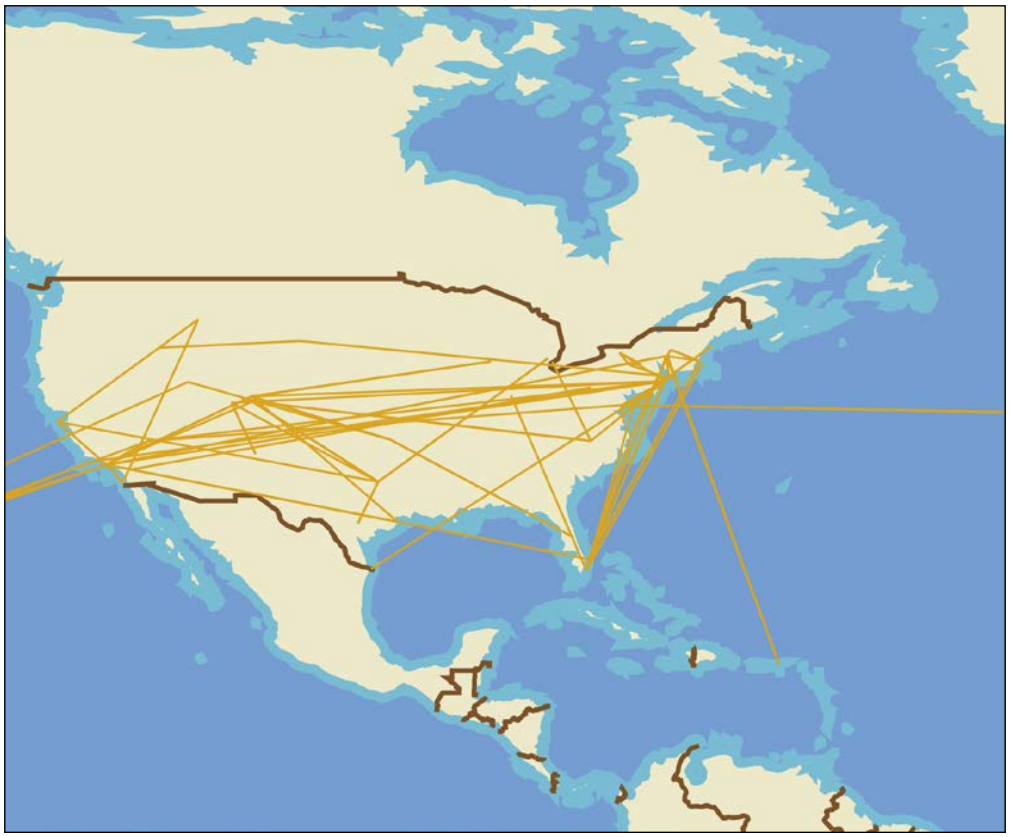
The translation vector pans the map with a transformation as in *Chapter 2*, *A Primer on DOM, SVG, and CSS*. `d3.event.scale` is just a number the projection uses to scale itself, effectively zooming the map.

Next, we recalculate all the routes using the changed projection:

```
d3.selectAll('line.route')
  .attr('x1', (d) => projection([d.from.lon, d.from.lat])[0])
  .attr('y1', (d) => projection([d.from.lon, d.from.lat])[1])
  .attr('x2', (d) => projection([d.to.lon, d.to.lat])[0])
  .attr('y2', (d) => projection([d.to.lon, d.to.lat])[1]);
```

The positioning function is exactly the same as in `addRenditions` because geographic projections handle panning out of the box. The thickness stays more or less the same throughout.

You can now explore the world!



Have patience, though, it's *reaaaaal* slow. Redrawing everything on every move does that. For a more performant and zoomable map, we'd have to use data with less detail when zoomed out, draw a sensible number of lines, and possibly avoid drawing parts of the map that fall out of the image anyway. Another way in which we could make it more performant is by transforming the entire map area instead of reprojecting on each event. This not only requires less computing but it also allows the GPU to do some of the work. Let's give that a shot!

Replace the entirety of `zoomable` with the following snippet:

```
function zoomable() {
  chart.call(d3.behavior.zoom()
    .center([chart.attr('width') / 2, chart.attr('height') / 2])
    .scale(projection.scale())
    .on('zoom', () => onzoom()));
}
```

All we've done here is replace the translate line with a line telling it to always zoom from the center of the chart — it works without doing this but it's a bit janky. Next, replace `onzoom` with the following:

```
function onzoom() {
  let scaleFactor = d3.event.scale / projection.scale();
  chart.attr('transform', `translate(${d3.event.translate})
scale(${scaleFactor})`);
  d3.selectAll('line.route').each(function() {
    d3.select(this).style('stroke-width', `${ 2 /
scaleFactor}px`);
  });
}
```

We're doing two things here: we're getting the real scale factor by dividing the scale given by the zoom event by the initial scale as it was defined for the projection (in this case, 150). We then use this to scale and translate the `g` element holding all of our geometry. We also divide the initial stroke width (2px, set in `index.css`) by the scaling factor to prevent the route lines from getting big and blocky as we zoom in.

That's miles better, isn't it? D3 gives you a lot of freedom in how you implement things; understanding which is the best method for any particular project comes from practice and knowing your audience.

Brushes

Brushes are similar to zoom and drag and are a simple way to create complex behavior— they enable users to select a part of the canvas.

Strangely, they aren't considered as a behavior but fall under the `.svg` namespace, perhaps because they are mostly meant for visual effects.

To create a new brush, we call `d3.svg.brush()` and define its `x` and `y` scales using `.x()` and `.y()`. We can also define a bounding rectangle.

Time for an example!

We're going back yet again to our all-singing, all-dancing prison population graph. This is the last example in which I will use it, I promise. We are going to let the user zoom in on a group of bars by selecting them with a brush, zooming out on a right-click.

Begin by creating a new class that extends `InteractivePrisonPopulationChart`, like so:

```
export class SelectableInteractivePrisonChart extends
InteractivePrisonPopulationChart {
  constructor(path) {
    super(path);
  }

  addUIElements() {}
  brushstart() {}
  brush() {}
  brushend() {}
  rightclick() {}
}
```

This is almost the same setup that we used for the drag example. Add the following to `addUIElements`:

```
    this.chart.append('g')
      .classed('brush', true)
      .call(d3.svg.brush().x(this.x).y(this.y)
      .on('brushstart', this.brushstart.bind(this))
      .on('brush', this.brushmove.bind(this))
      .on('brushend', this.brushend.bind(this)));
```

We made a new grouping element for the brush and called a freshly constructed `d3.svg.brush()` with both scales defined. The `.brush` class helps with styling. We also `bind` the local context to each event callback so that we still have access to our class methods.

Next, we define listeners for the `brushstart`, `brush`, and `brushend` events as part of our class. We're not doing anything with `brushstart` so let's skip straight to `brushmove`:

```
  brushmove() {
    let e = d3.event.target.extent();
    d3.selectAll('.bar').classed('selected', (d) =>
      e[0][0] <= this.x(d.year)
      && this.x(d.year) <= e[1][0]
    );
  }
```

`brushmove` is where the real magic happens. Firstly, we find the selection's boundaries by using `d3.event.target.extent()`.

`d3.event.target` returns the current brush and `.extent()` returns a set of two points — the upper-left and bottom-right corners.

Then, we go through all the bars and turn the `.selected` class on or off, depending on whether the bar's position lies within the bounding box.

Note that the values returned by `extent` are wholly reliant on the type of scale you provide to it — if you give it a linear scale, you'll compare the unscaled datum value against the extent value. In an ordinal scale like we have here, we have to scale the datum value before comparing.

Next, we define what happens when the mouse button is released:

```
brushend() {
  let selected = d3.selectAll('.selected');

  // Clear brush object
  d3.event.target.clear();
  d3.select('g.brush').call(d3.event.target);

  // Zoom to selection
  let first = selected[0][0];
  let last = selected[0][selected.size() - 1]
  let startYear = d3.select(first).data()[0].year;
  let endYear = d3.select(last).data()[0].year;
  this.clearSelected().then(() => {
    this.updateChart(this.data.filter((d) =>
      d3.range(startYear, endYear).indexOf(Number(d.year)) > -1));
  });

  let hitbox = this.svg
    .append('rect')
    .classed('hitbox', true)
    .attr('width', this.svg.attr('width'))
    .attr('height', this.svg.attr('height'))
    .attr('fill-opacity', 0);

  hitbox.on('contextmenu', this.rightclick.bind(this));
}
```

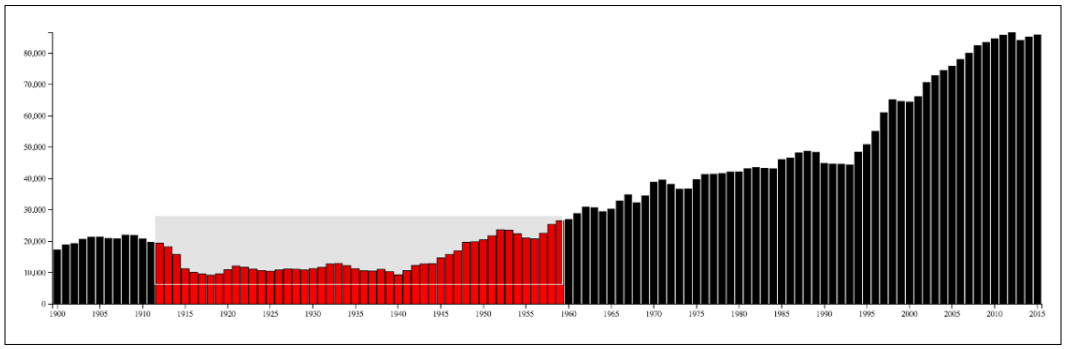
Quite a lot happens here. Firstly, we clear the brush overlay, then we figure out the first and last element selected by the brush. From that, we get the start and end years, which we supply to `d3.range`, redrawing the chart just like we did in `InteractivePrisonPopulationChart`. Lastly, we add a hit box, which we'll use to listen to the `contextmenu` event (`contextmenu` being the right-click variant of the click event).

Our HTML needs some more styling definitions, like this:

```
.brush .extent {
  stroke: #fff;
  fill-opacity: .125;
  shape-rendering: crispEdges;
}

.bar.selected {
  stroke: black;
}
```

In this case, we're still using the CSS transitions from the first example. It's preferable to use CSS transitions rather than what D3 can do in this case — brushes sometimes have problems with D3 transitions and change properties immediately.

When you select some elements, the image will look like this:



Summary

Wow, what a fun chapter!

You've made things jump around the page, almost killed your computer and patience with a zoomable map, and made one supremely awesome bar graph. Well done!

In this chapter, we've animated with transitions, interpolators and timers, and then we learned how to do some of that with CSS. We then learned the difference between explanatory and exploratory visualizations, and used interactivity to create the former. We then made some exploratory visualizations by using behaviors with some of our previous projects. Things are starting to look pretty snazzy, aren't they?

In the next chapter, we'll be looking at creating a whole boatload of really pretty charts using D3's black magic — layouts. Combining the skills you've learned in this chapter and the next one will mean that you're able to produce some truly fantastic charts. I hope that you're ready — this is where stuff starts getting really cool!

5
Layouts – D3's Black Magic

Most of us look at the Internet for inspiration and code samples. You find something that looks great, you look at the code, and your eyes glaze over. It doesn't make any sense.

The usual culprit is D3's reliance on layouts for anything remotely complicated. The black magic of taking some data, calling a function, and—voilà—visualization! This elegance makes layouts look deceptively difficult, but they make things a lot easier when you get the hang of them.

In this chapter, we'll go in, guns blazing, with everything you've learned so far to create 11—count 'em! 11!—visualizations of the same dataset.

What are layouts and why should you care?

D3 layouts are modules that transform data into drawing rules. The simplest layout might only transform an array of objects into coordinates, like a scale.

But we usually use layouts for more complex visualizations, drawing a force-directed graph or a tree for instance. In these cases, layouts help us separate calculating coordinates from putting pixels on a page screen. This not only makes our code cleaner but also lets us reuse the same layouts for vastly different visualizations.

Theory is boring. Let's dig in.

Built-in layouts

By default, D3 comes with 12 built-in layouts that cover most common visualizations. They can be split roughly into **normal** and **hierarchical** layouts. Normal layouts represent data in a flat hierarchy, while hierarchical layouts generally present data in a tree-like structure. The normal layouts are as follows:

- Histogram
- Pie
- Stack
- Chord
- Force

The hierarchical layouts are as follows:

- Partition
- Tree
- Cluster
- Pack
- Tree map

To see how they behave, we're going to make an example for each type. We'll start with the humble pie chart and histogram and then progress to force-directed graphs and fancy trees. We'll be using the same dataset for all examples so that we can get a feel of how different presentations affect the perception of data.

We're getting pretty good at this by now, so we're going to make these examples particularly magnificent. That's going to create a lot of code, so every time we come up with something reusable, we'll put it in a **helpers.js** file as a function. We'll be exporting this as a big ol' bucket of functions instead of a class this time, mainly because we will have to deal with much less weirdness due to this changing context.

Let's create an empty `helpers.js` file. We'll be adding functions to this file willy-willy and then importing them under the same namespace elsewhere. We could have done this in a bunch of different ways, for instance, exporting an object with a bunch of different methods, or creating a class or a factory function. However, doing it this way means that we can either selectively choose functions from the module without importing them all, or import all of them during development with a single statement. For now, just add this line at the top:

```
let d3 = require('d3');
```

Meanwhile, let's once again extend `BasicChart`, creating a class that will create all our different representations of the same chart. Add the following code to a new file called `chapter5.js`:

```
import * as helpers from './helpers';
import {BasicChart} from './basic-chart';
let d3 = require('d3');

export class PoliticalDonorChart extends BasicChart {
  constructor(chartType, ...args) {
    super();
    require('./chapter5.css');

    let p = new Promise((res, rej) => {
      d3.csv('data/uk_political_donors.csv',
        (err, data) => err ? rej(err) : res(data));
    });


    p.then((data) => {
      this.data = data;
      this[chartType].call(this, ...args);
    });

    return p;
  }
}
```

You'll notice that we're importing everything from `helpers.js` under the `helpers` namespace. As we add helper functions, they'll automatically be added to the `helpers` object we're importing from that file. Otherwise, there is nothing unusual:

1. First, we load the CSS file that we'll make in a moment.
2. Then we call `super`.
3. We create a new `Promise`.
4. We have D3 load and parse the CSV data.
5. Then we resolve the promise if everything is okay, calling the method specified by the constructor's first argument.

The rest of the constructor's arguments are then passed as arguments to the method we're calling.

The `...args` bit in the constructor's arguments is a new feature in ES2016, called the **rest parameter**. It collects every argument after the ones specifically defined as an array. We then use another new ES2016 feature, called the **spread operator**, in `this[chartType].call(this, ...args)` to destructure the array into its individual values. This lets us add as many arguments as we want to each chart method we're writing.

The dataset

The dataset that we are going to use in all of these examples is a list of donations to various political entities of the UK: who received the donation, the amount, whom it was from, the type of donor, and the date it was made. Data like this is important, as it can indicate individual actors influencing the political process. This alone might not be enough for a story, but at the very least, it can point you towards groups or individuals worth pursuing further. It's in the book's repository at `src/data/uk_political_donors.csv`, or you can get it from `https://github.com/leilahaddou/graph-data/blob/master/pef.csv`.

It's worth noting that I got this dataset from Leila Haddou's totally awesome tutorial on Neo4j, a network graph database. If you like the force-directed graph later on in this chapter, Neo4j might just totally blow your mind. Give it a shot! It is at `http://leilahaddou.github.io/neo4j-tutorial.html`.

Normal layouts

Time to draw! As mentioned, we'll begin with normal layouts. They display data in a flat hierarchy.

Using the histogram layout

We are going to use the `histogram` layout to create a bar chart of the number of donations received. The layout itself will handle everything from collecting values in bins to calculating heights, widths, and positions of the bars.

Histograms usually represent a probability distribution over a continuous numerical domain, but the names of donation recipients are ordinal. To bend the `histogram` layout to our will, we will have to turn names into numbers—we'll use a scale.

Since it feels like this could be useful in other examples, we'll put the code in `helpers.js`:

```
export function uniques(data, name) {
  let uniques = [];
  data.forEach((d) => {
    if (uniques.indexOf(name(d)) < 0) {
      uniques.push(name(d));
    }
  });
  return uniques;
}

export function nameId(data, name) {
  let uniqueNames = uniques(data, name);
  return d3.scale.ordinal()
  .domain(uniqueNames)
  .range(d3.range(uniqueNames.length));
}
```

These are two simple functions:

- `uniques`: This goes through the data and returns a list of unique names. We help it with the `name` accessor.
- `nameId`: This creates an ordinal scale that we'll be using to convert names into numbers. Expect to see both of these a lot in this chapter.

Now we can tell the histogram how to handle our data with `nameId`. Add the following method to your `PoliticalDonorChart` class:

```
histogram() {
  let data = this.data;

  let nameId = helpers.nameId(data, (d) => d.EntityName);
  let histogram = d3.layout.histogram()
                    .bins(nameId.range())
                    .value((d) => nameId(d.EntityName))(data);

  this.margin = {top : 10, right : 40, bottom : 100, left : 50};
}
```

Using `d3.layout.histogram()`, we create a new histogram and use `.bins()` to define the upper threshold for each bin. Given `[1,2,3]`, values under `1` go into the first bin, values between `1` and `2` into the second, and so on.

The `.value()` accessor tells the histogram how to find values in our dataset.

Another way to specify bins is by specifying the number of bins you want and letting the histogram uniformly divide a continuous numerical input domain into bins. For such domains, you can even make probability histograms by setting `.frequency()` to `false`. You can limit the range of considered bins with `.range()`.

Finally, we use the layout as a function on our data to get an array of objects with the following schema:

```
{
    0: {
        DonorName: String, // e.g. "Ændrew Rininsland"
        DonorStatus: String, // e.g. "Individual"
        ECRef: String, // e.g. "c1234567"
        EntityName: String, // e.g. "Green Party"
        ReceivedDate: Date, // e.g. "27/01/15" (dd/mm/yy)
        Value: String // e.g. "£1,234"
    },
    dx: Number,
    x: Number,
    y: Number
}
```

It's worth reiterating that we define the histogram and then immediately run it on our dataset. Look at the following line:

```
let histogram = d3.layout.histogram()
  .bins(nameId.range())
  .value((d) => nameId(d.EntityName))(data);
```

It can also be written like this:

```
let histogramLayout = d3.layout.histogram()
  .bins(nameId.range())
  .value((d) => nameId(d.EntityName));
let histogram = histogramLayout(data);
```

We just avoid the temporary variable for something we use only once here by immediately executing the layout and then storing the output in a variable.

The bin width is in the `dx` property, `x` is the horizontal position, and `y` is the height. We access elements in bins with normal array functions. Note that they're all strings, even the amount and date. We'll have to handle this in our code using `d3.format`.

Using this data to draw a bar chart should be easy by now. We'll define a scale for each dimension, label both axes, and place some rectangles for bars.

Next, add two scales:

```
let x = d3.scale.linear()
  .domain([0, d3.max(histogram, (d) => d.x)])
  .range([this.margin.left, this.width-this.margin.right]);

let y = d3.scale.log().domain([1, d3.max(histogram, (d) => d.y)])
  .range([ this.height - this.margin.bottom, this.margin.top ]);
```

The use of a logarithmic scale for the vertical axis will make it such that the number of donations received by the biggest entities doesn't totally flatten everything else.

Next, put a vertical axis on the left:

```
let yAxis = d3.svg.axis()
     .scale(y)
     .tickFormat(d3.format('f'))
     .orient('left');

this.chart.append('g')
 .classed('axis', true)
 .attr('transform', 'translate(50, 0)')
 .call(yAxis);
```

We create a grouping element (the `'g'`) for every bar and its label:

```
    let bar = this.chart.selectAll('.bar')
      .data(histogram)
      .enter()
      .append('g')
      .classed('bar', true)
      .attr('transform', (d) => `translate(${x(d.x)},${y(d.y)})`);
```

Moving the group into position, as shown in the following code, means less work when positioning the bar and its label:

```
bar.append('rect')
  .attr({
    x: 1,
    width: x(histogram[0].dx) - this.margin.left - 1,
    height: (d) => this.height - this.margin.bottom - y(d.y)
  })
  .classed('histogram-bar', true);
```

Because the group is in place, we can put the bar a pixel from the group's edge. All bars will be `histogram[0].dx` units wide, and we'll calculate heights using the `y` position of each datum and the total graph height. Lastly, we create the labels:

```
bar.append('text')
  .text((d) => d[0].EntityName)
  .attr({
    transform: (d) =>
     `translate(0, ${this.height - this.margin.bottom - y(d.y)
      + 7}) rotate(60)`
  });
```

We move labels to the bottom of the graph, rotate them by 60 degrees to avoid overlap, and set their text to the `EntityName` property of the datum.

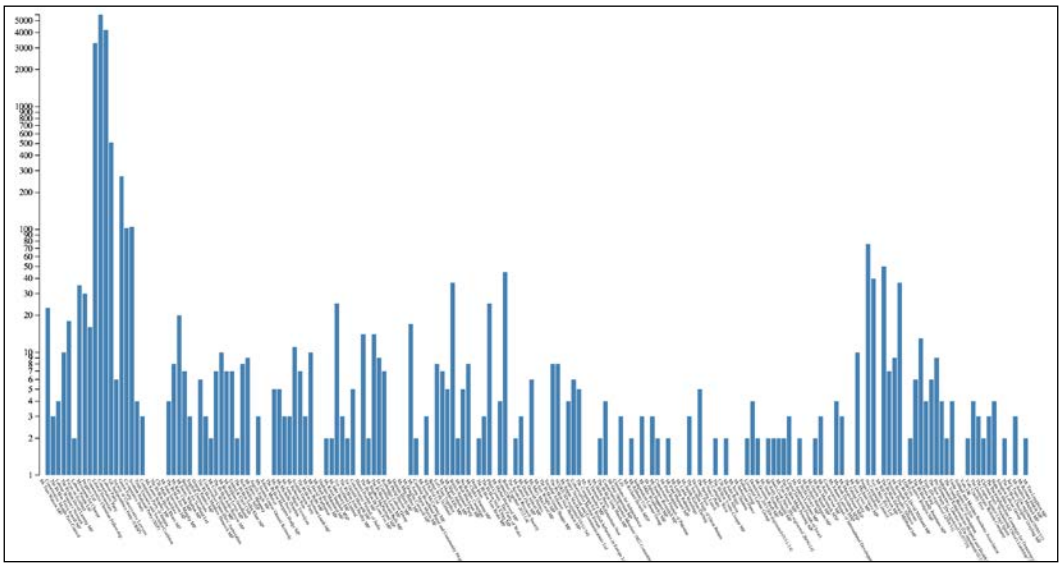
Now create a new file, `chapter5.css`, and add this code:

```
.axis path, .axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
.axis text {
  font-size: 0.75em;
}
rect.histogram-bar {
  fill: steelblue;
  shape-rendering: crispEdges;
}
.bar text {
  font-size: 0.4em;
}
```

Don't forget to `require` the CSS file in your JavaScript. Replace `index.js` with the following line:

```
require('./index.css');
import {PoliticalDonorChart} from './chapter5';
new PoliticalDonorChart('histogram');
```

Our bar chart looks like this:

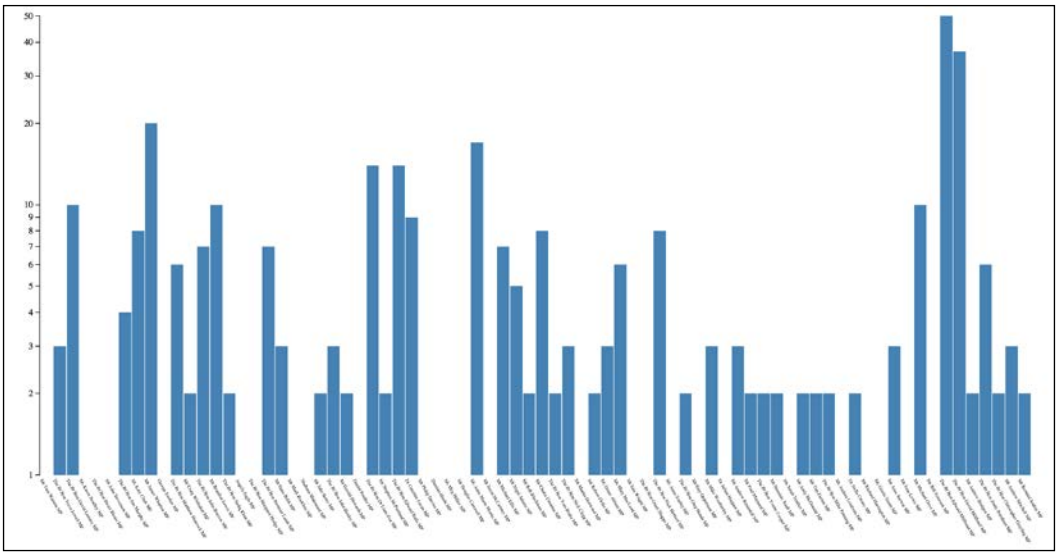


Hmm… It seems that the incumbent party got the most donations. This is totally unsurprising. Let's filter for only Members of Parliament (MPs, for those outside the Commonwealth).

Replace the first line of your `histogram` method with the following:

```
let data = this.data.filter((d) => d.EntityName.match(' MP'));
```

Much better! Note that we start at `1` instead of `0` as we're using a logarithmic scale. This means that MPs with only one donor appear as if they have none. You'd want to explain that to your audience if sticking to the log scale in such an instance, or more likely filter out individuals with only one donation:



Ed Miliband, then-leader of the opposition Labour Party, had the most donations in this particular set of data

Baking a fresh 'n' delicious pie chart

The preceding bar chart reveals that Ed Miliband received the most donations during this period. Let's find out who's making him so popular.

We are going to use the pie chart layout to cut the donations to *The Rt Hon Edward Miliband MP* into slices, showing how many donations he got from each donor. After filtering the dataset for donations going to Miliband, we have to categorize the entries by givers, and finally we feed them into the pie chart layout to generate a pie chart.

We can use the `histogram` layout to put data into bins depending on the `DonorName` property. Let's add a function to `helpers.js`:

```
export function binPerName (data, name) {
   let nameIds = nameId(data, name);
   let histogram = d3.layout.histogram()
    .bins(nameIds.range())
    .value((d) => nameIds(name(d)) );
   return histogram(data);
}
```

Similar to the `uniques` and `nameId` functions, `binPerName` takes the data and a `name` accessor and returns histogram data.

Now create a `pie` method in your `PoliticalDonorChart` class:

```
pie(name) {
  let filtered = this.data.filter((d) => d.EntityName === name);
  let perDonor = helpers.binPerName(filtered,
    (d) => d.DonorName);
}
```

Entries in the `perDonor` variable will tell us exactly how many donations were received by the name specified by the function argument.

To bake a pie, we call the `pie` layout and give it a `value` accessor:

```
  let pie = d3.layout.pie()
    .value((d) => d.length)(perDonor);
```

The `pie` layout is now full of slice objects, each holding the `startAngle` and `endAngle` values and the original value.

The entries look like this:

```
{
  data: Array[135],
  endAngle: 2.718685950221936,
  startAngle: 0,
  value: 135
}
```

We could have specified a `.sort()` function to change how slices are organized and a `.startAngle()` or `.endAngle()` function to limit the pie's size.

All that's left to do now is drawing a pie chart. We'll need an `arc` generator (just like the ones in *Chapter 2*, *A Primer on DOM, SVG, and CSS*) and some colors to tell the slices apart.

Finding 24 distinct colors that look great together is hard; luckily for us, `@ponywithhiccups` has jumped into the challenge and made the pick. Thank you!

Let's add these colors to `helpers.js`:

```
export const color = d3.scale.ordinal().range(['#EF3B39',
'#FFCD05', '#69C9CA', '#666699', '#CC3366',
  '#0099CC', '#999999', '#FBF5A2', '#6FE4D0', '#CCCB31',
'#009966', '#C1272D', '#F79420', '#445CA9',
  '#402312', '#272361', '#A67C52', '#016735', '#F1AAAF','#A0E6DA',
'#C9A8E2', '#F190AC', '#7BD2EA',
  '#DBD6B6']);
```

The `color` scale is an ordinal scale without a domain. We export it as a constant (that is, a variable that won't change) using the `const` keyword. Exporting variables that aren't constants isn't allowed.

If you try to redefine the `color` constant, Babel will throw this error:
`Module build failed: SyntaxError: ../learning-d3/src/helpers.js: Line 39: "color" is read-only`
Use constants for variables that really shouldn't ever be modified. Note, however, that you can still update your color scale by using `.range` and `.domain` as setters (that is, by supplying them a new array to replace the one you defined in `helpers.js`).

To make sure that the donors always get the same color, a function in `helpers.js` will help us fixate the domain, as shown in the following code:

```
export function fixateColors (data) {
  color.domain(uniques(data, (d) => d.DonorName));
}
```

Now, we can define the `arc` generator in our pie method and fixate the colors:

```
let arc = d3.svg.arc()
  .outerRadius(150)
  .startAngle((d) => d.startAngle)
  .endAngle((d) => d.endAngle);

helpers.fixateColors(filtered);
```

A group element will hold each arc and its label, as shown in the following code:

```
let slice = this.chart.selectAll('.slice')
           .data(pie)
           .enter()
           .append('g')
           .attr('transform', 'translate(300, 300)');
```

To make positioning simpler, we move every group to the center of the pie chart. Creating slices works the same as in *Chapter 2, A Primer on DOM, SVG, and CSS*:

```
slice.append('path')
  .attr({
    d: arc,
    fill: (d) => helpers.color(d.data[0].DonorName)
  });
```

We get the color for a slice with `d.data[0].DonorName`. The original dataset is in `.data`, and all the `.DonorName` properties in it are the same. That's what we grouped by.

Labeling your pie chart

Labels take a bit more work. They need to be rotated into place and sometimes flipped so that they don't appear upside down. Labeling an arc will be handy later as well, so let's make a general function in `helpers.js`:

```
export function arcLabels(text, radius) {
  return function (selection) {
    selection.append('text')
      .text(text)
      .attr('text-anchor', (d) =>
         tickAngle(d) > 100 ? 'end' : 'start')
      .attr('transform', (d) => {
        let degrees = tickAngle(d);
        let turn = `rotate(${degrees})
          translate(${radius(d) + 10}, 0)`;
        if (degrees > 100) {
          turn += `rotate(180)`;
        }
        return turn;
      });
  }
}
```

Here, we're creating a "factory"-style function that generates another function operating on a D3 selection. This means we can use it with `.call()` while still defining our own parameters.

We'll give `arcLabels` a `text` accessor and a `radius` accessor, and it will return a function that we can use with `.call()` on a selection to make labels appear in just the right places. The meaty part appends a text element, tweaks its `text-anchor` element depending on whether or not we're going to flip it, and rotates the element to a particular position with the help of a `tickAngle` function.

Let's add the contents of the `tickAngle` function:

```
export function tickAngle (d) {
  let midAngle = (d.endAngle - d.startAngle) / 2;
  let degrees = (midAngle + d.startAngle) / Math.PI * 180 - 90;
  return degrees;
}
```

The `helpers.tickAngle` calculates the middle angle between `d.startAngle` and `d.endAngle` and transforms the result from radians to degrees so that SVG can understand it.

This is basic trigonometry, so I won't go into details, but your favorite high schooler should be able to explain the math.

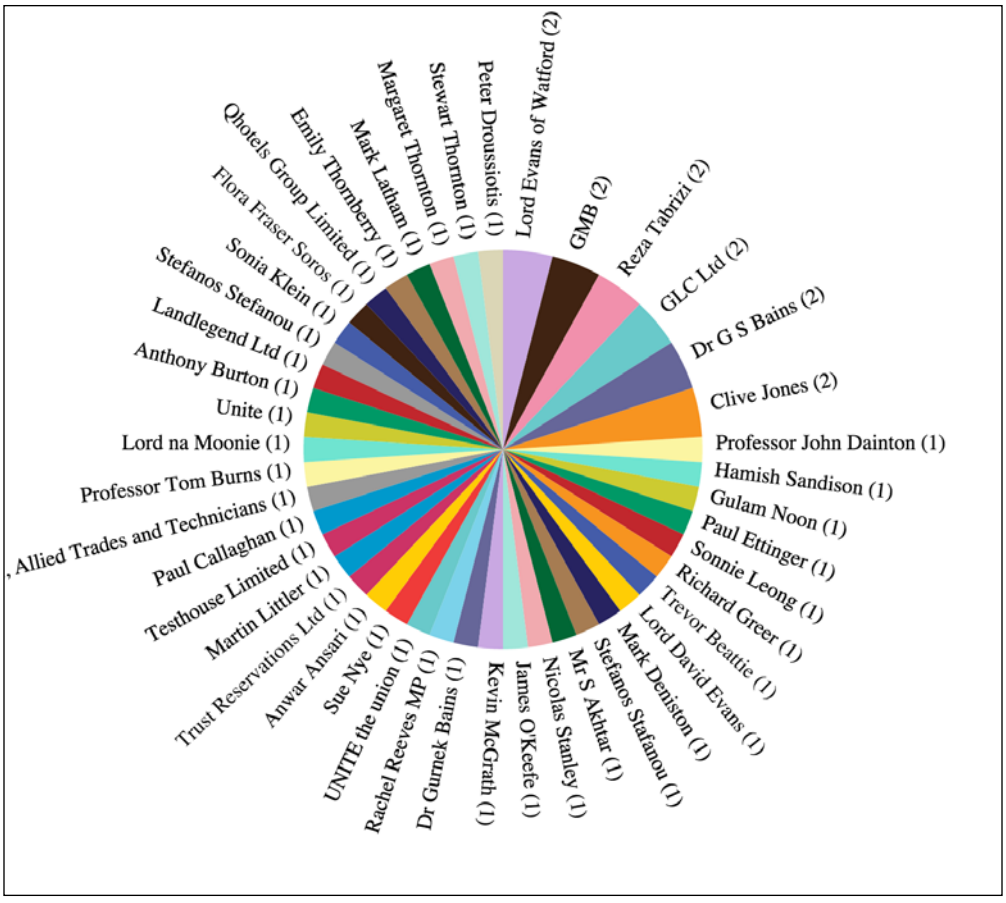
We use `arcLabels` back in the `pie` method:

```
slice.call(helpers.arcLabels((d) =>
  `${d.data[0].DonorName} (${d.value})`, arc.outerRadius()));
```

Lastly, we initialize the constructor in `index.js`:

```
new PoliticalDonorChart('pie', 'The Rt Hon Edward Miliband MP');
```

And our delicious pie is done, as shown in the following screenshot:



Showing popularity through time with stack

D3's official docs say:

> "*The stack layout takes a two-dimensional array of data and computes a baseline; the baseline is then propagated to the above layers, so as to produce a stacked graph.*"

Not clear at all, but I am hard pressed to come up with anything better. The `stack` layout calculates where one layer ends and another begins. An example should help.

We're going to make a layered timeline of donations over the 5-year sample, stretching as far back as 2010, with the width of each layer telling us how many donations went where at a certain time. This timeline is called a **streamgraph**.

To label layers, we're going to create a `mouseover` behavior that highlights a layer and shows a tooltip with the donation recipient's name.

Let's begin the binning. Create a new method called `streamgraph` in `PoliticalDonorChart`:

```
streamgraph() {
  let time = d3.time.format('%d/%m/%y');
  let data = this.data;
  let extent = d3.extent(data.map((d) =>
    time.parse(d.ReceivedDate)));
  let timeBins = d3.time.days(extent[0], extent[1], 12);
}
```

To parse timestamps into `Date` objects, we specified a format for strings such as `27/04/15`. Then, we used this format to find the earliest and latest time with `d3.extent`. Telling `d3.time.days()` to go from start to finish with a step of 14 days creates a list of bins.

We use the `histogram` layout to munge our dataset into a more useful form:

```
  let perName = helpers.binPerName(data, (d) => d.EntityName);
  let timeBinned  = perName.map((nameLayer) => {
    return {
      to: nameLayer[0].EntityName,
      values: d3.layout.histogram()
            .bins(timeBins)
            .value((d) => time.parse(d.ReceivedDate))(nameLayer)
    }
  });
```

You already know what `helpers.binPerName` does.

To bin data into time slots, we mapped through each layer of the `name` accessors and turned it into a two-property object. The `.to` property tells us whom the layer represents, and `.values` is a histogram of time slots where entries tell us how much karma the user many donations somebody got in a certain 12-day period.

Time for a `stack` layout:

```
let layers = d3.layout.stack()
  .order('inside-out')
  .offset('wiggle')
  .values((d) => d.values)(timeBinned);
```

The `d3.layout.stack()` creates a new `stack` layout. We told it how to order layers with `.order('inside-out')` (you should also try `default` and `reverse`) and decided how the final graph looks with `.offset('wiggle')`. The `wiggle` minimizes changes in slope. Other options include `silhouette`, `zero`, and `expand`. Try them!

Once again, we told the layout how to find values with the `.values()` accessor. Our `layers` array is now filled with objects like this one:

```
{to: " The Rt Hon Edward Miliband MP",
   values: Array[50]}
```

The `values` is an array of arrays. Entries in the outer array are time bins that look like this:

```
{dx: 1036800000,
   length: 1,
   x: Object(Thu Oct 13 2011 00:00:00 GMT+0200 (CEST)),
   y: 1,
   y0: 140.16810522517937}
```

The important parts of this array are as follows: `x` is the horizontal position, `y` is the thickness, and `y0` is the baseline. The `d3.layout.stack` will always return these.

To start drawing, we need some margins and two scales:

```
this.margin = {
  top: 220,
  right: 50,
  bottom: 0,
  left: 50
};

let x = d3.time.scale()
```

```
    .domain(extent)
    .range([this.margin.left, this.width - this.margin.right]);

  let y = d3.scale.linear()
    .domain([0, d3.max(layers,
      (layer) => d3.max(layer.values, (d) => d.y0 + d.y))])
    .range([this.height - this.margin.top, 0]);
```

The tricky thing was finding the vertical scale's domain. We found it by going through each value of every layer, looking for the maximum `d.y0+d.y` value—baseline plus thickness.

We'll use an `area` path generator for the layers:

```
let offset = 100;
let area = d3.svg.area()
  .x((d) => x(d.x))
  .y0((d) => y(d.y0) + offset)
  .y1((d) => y(d.y0 + d.y) + offset);
```

Nothing too fancy. The baselines define bottom edges and adding the thickness gives the top edge. Fiddling determined that both should be pushed down by 100 pixels.

Let's draw an axis first:

```
let xAxis = d3.svg.axis()
  .scale(x)
  .tickFormat(d3.time.format('%b %Y'))
  .ticks(d3.time.months, 2)
  .orient('bottom');

this.chart.append('g')
  .attr('transform', `translate(0, ${this.height - 100})`)
  .classed('axis', true)
  .call(xAxis)
  .selectAll('text')
    .attr('y', 5)
    .attr('x', 9)
    .attr('dy', '.35em')
    .attr('transform', 'rotate(60)')
    .style('text-anchor', 'start');
```

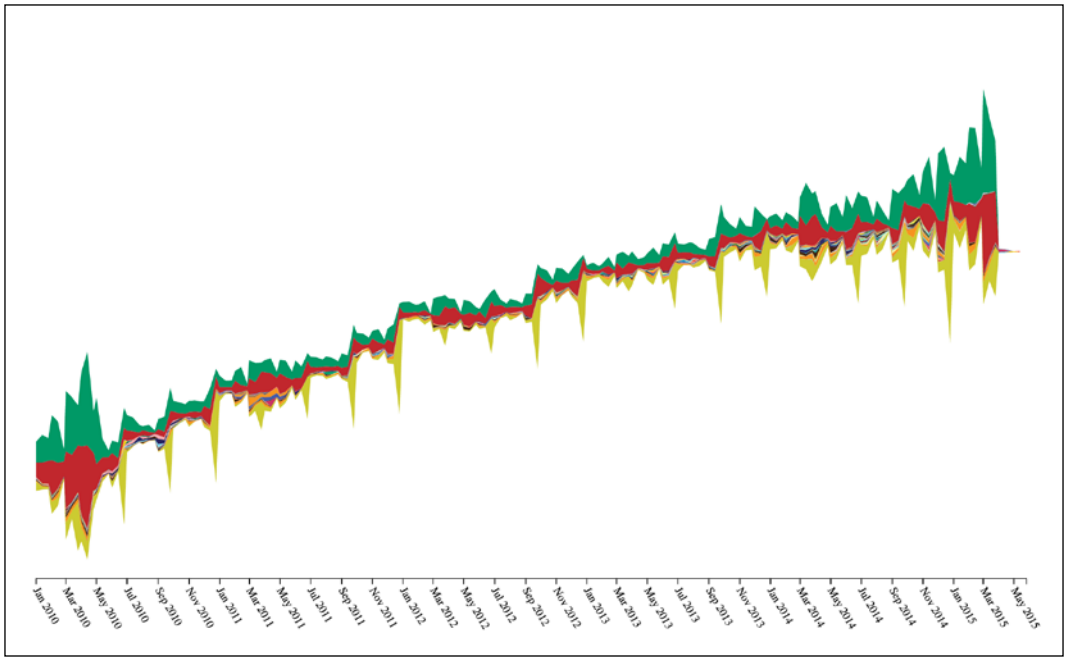
Same as usual—we defined an axis, called it on a selection, and let D3 do its thing. We made it more readable with a custom `.tickFormat()` function and used `.ticks()` to say that we want a new tick every 2 months.

Okay, now for the streamgraph, add the following code:

```
this.chart.selectAll('path')
  .data(layers)
  .enter()
  .append('path')
  .attr('d', (d) => area(d.values))
  .style('fill', (d, i) => helpers.color(i))
  .call(helpers.tooltip((d) => d.to, this.chart));
```

Not much is going on. We used the `area` generator to draw each layer, defined colors with `helpers.color`, and called a `tooltip` function, which we'll define in `helpers.js` later.

The streamgraph looks like this:



Adding tooltips to our streamgraph

It looks pretty, but it is useless. Let's add that `tooltip` function to the `helpers.js` tooltip:

```
export function tooltip (text, chart) {
  return function (selection) {
    selection.on('mouseover.tooltip', mouseover)
```

```
        .on('mousemove.tooltip', mousemove)
        .on('mouseout.tooltip', mouseout);
    }
  }
```

We defined event listeners with a `.tooltip` namespace so that we can define multiple listeners on the same events.

The `mouseover` function will highlight streams and create tooltips, `mousemove` will move tooltips, and `mouseout` will put everything back to normal.

Let's put the three listeners inside the inner function of our tooltip helper:

```
    function mouseover(d) {
      let path = d3.select(this);
      path.classed('highlighted', true);
    }
```

That's the simple part of `mouseover`. It selects the current area and changes its class to `highlighted`. This will make it lighter and add a red outline.

In the same function, add the meaty part:

```
      let mouse = d3.mouse(chart.node());
      let tool = chart.append('g')
        .attr({
          'id': 'nameTooltip',
          transform: `translate(${mouse[0] + 5},${mouse[1] + 10})`
        });

      let textNode = tool.append('text')
        .text(text(d)).node();

      tool.append('rect')
        .attr({
          height: textNode.getBBox().height,
          width: textNode.getBBox().width,
          transform: 'translate(0, -16)'
        });

      tool.select('text')
        .remove();

      tool.append('text').text(text(d));
```

It is longer and with a dash of magic, but not scary at all!

First, we find the mouse's position. Then we create a group element and position it down and to the right of the mouse. We add a text element to the group and call SVG's `getBBox()` function on its node. This gives us the text element's bounding box and helps us size the background rectangle.

Finally, we remove the text, because it's covered by the background, and add it again. We might be able to avoid all this trouble by using HTML *divs*, but I wanted to show you pure SVG tooltips. Hence, consider the following code:

```
function mousemove () {
    let mouse = d3.mouse(chart.node());
    d3.select('#nameTooltip')
    .attr('transform', `translate(${mouse[0] + 15},
      ${mouse[1] + 20})`);
}
```

The `mousemove` listener in the following code is much simpler. It just finds the `#nameTooltip` element and moves it to follow the cursor:

```
function mouseout () {
    let path = d3.select(this);
    path.classed('highlighted', false);
    d3.select('#nameTooltip').remove();
}
```

The `mouseout` function selects the current path, removes its `highlighted` styling, and removes the tooltip.

Voilà! Tooltips!

Very rudimentary—they don't understand edges and they won't break any hearts with their looks, but they get the job done. Let's add some CSS to `chapter5.css`:

```
#nameTooltip {
  font-size: 1.3em;
}

#nameTooltip rect {
  fill: white;
}

#nameTooltip text {
  fill: #000;
  stroke: #000;
  color: #000;
}
```

```
path.highlighted {
  fill-opacity: 0.5;
  stroke: red;
  stroke-width: 1.5;
}
```

And suddenly, we have a potentially useful streamgraph on our hands!

Highlighting connections with chord

We've seen how many donations people have and when they got it, but there's another gem hiding in the data—connections. We can visualize who is donating to whom using the `chord` layout.

We're going to draw a chord diagram—a circular diagram of connections. Chord diagrams are often used in genetics and have even appeared on covers of magazines (`http://circos.ca/intro/published_images/`).

Ours is going to have an outer ring showing how much money is being donated and chords showing where that money is going.

First, we need a matrix of connections for the chord diagram, and then we'll go the familiar route of path generators and adding elements. The matrix code will be useful later, so let's put it in `helpers.js`:

```
export function connectionMatrix (data) {
  let nameIds = nameId(allUniqueNames(data), (d) => d );
  let uniques = nameIds.domain();
  let matrix = d3.range(uniques.length).map(
    () => d3.range(uniques.length).map(() =>  0));
  data.forEach((d) => {
    matrix[nameIds(d.DonorName)][nameIds(d.EntityName)] +=
      Number(d.Value.replace(/[^\d\.]*/g, ''));
  });

  return matrix;
}
```

Let's also create a function that returns unique names:

```
export function allUniqueNames(data) {
  let donors = uniques(data, (d) => d.DonorName);
  let donees = uniques(data, (d) => d.EntityName);
  return uniques(donors.concat(donees), (d) => d);
}
```

We begin with the familiar `uniques` list and the `nameId` scale. Then we create a zero matrix and loop through the data to increment by the value of each donation (which we quickly clean up by removing the pound symbol and the comma before changing it to a `Number`). Rows are *from whom*, columns are *to whom*. For example, if the fifth cell in the first row holds `10`, it means the first person or organization has given £10 to the fifth person. This is called an **adjacency matrix**.

Meanwhile, back in `PoliticalDonorChart`, create a `chord` method:

```
chord(filterString) {
    let filtered = this.data.filter((d) =>
      d.EntityName.match(filterString || ' MP') );
    let uniques = helpers.uniques(filtered, (d) => d.DonorName);
    let matrix = helpers.connectionMatrix(filtered);
}
```

We create the matrix from our data, which we've filtered for individual MPs. We can also provide an argument in the constructor to filter based on it. Moreover, we create another array of unique names, this time for MPs. This allows us to have something to connect to.

We're going to need `uniques` for labels, and it would be nice to have the `innerRadius` and `outerRadius` variables handy:

```
    let innerRadius = Math.min(this.width, this.height) * 0.3;
    let outerRadius = innerRadius * 1.1;
```

Time to make the `chord` layout do our bidding:

```
    let chord = d3.layout.chord()
      .padding(.05)
      .sortGroups(d3.descending)
      .sortSubgroups(d3.descending)
      .sortChords(d3.descending)
      .matrix(matrix);
```

It is a little different from others. The `chord` layout takes data via the `.matrix()` method and can't be called as a function.

We started with `d3.layout.chord()` and put a `.padding()` method between groups, which improves readability. To improve readability further, everything is sorted. The `.sortGroups` sorts groups on the edge, `.sortSubgroups` sorts chord attachments in groups, and `.sortChords` sorts the chord drawing order so that smaller chords overlap bigger ones.

In the end, we feed data into the layout with `.matrix()`:

```
let diagram = this.chart.append('g')
      .attr('transform',
         `translate(${this.width / 2},${this.height / 2})`);
```

We add a centered group element so that all our coordinates are relative to the center from now on.

The drawing of the diagram happens in three steps—arcs, labels, and chords—as shown in the following code:

```
let group = diagram.selectAll('.group')
  .data(chord.groups)
    .enter().append('g');

let arc = d3.svg.arc()
  .innerRadius(innerRadius)
  .outerRadius(outerRadius);

group.append('path')
  .attr('d', arc)
  .attr('fill', (d) => helpers.color(d.index));
```

This creates the outer ring. We use `chord.groups` to get group data from the layout, create a new grouping element for every chord group, and then add an arc. We use `arc_labels` from the pie example to add the labels:

```
group.call(helpers.arcLabels(
  (d) => uniques[d.index], () => outerRadius + 10));
```

Even though the radius is constant, we have to define it as a function using the following code because we didn't make `arcLabels` flexible enough for constants. Nobody ain't got time for that, though—we still have seven charts to make in this chapter!

```
diagram.append('g')
       .classed('chord', true)
       .selectAll('path')
       .data(chord.chords)
       .enter()
       .append('path')
       .attr('d', d3.svg.chord().radius(innerRadius))
       .attr('fill', (d) => helpers.color(d.target.index));
```

We get chord data from `chord.chords` and use a `chord path` generator to draw the chords. We pick colors with `d.target.index` because the graph looks better, but chord colors are *not* informative.

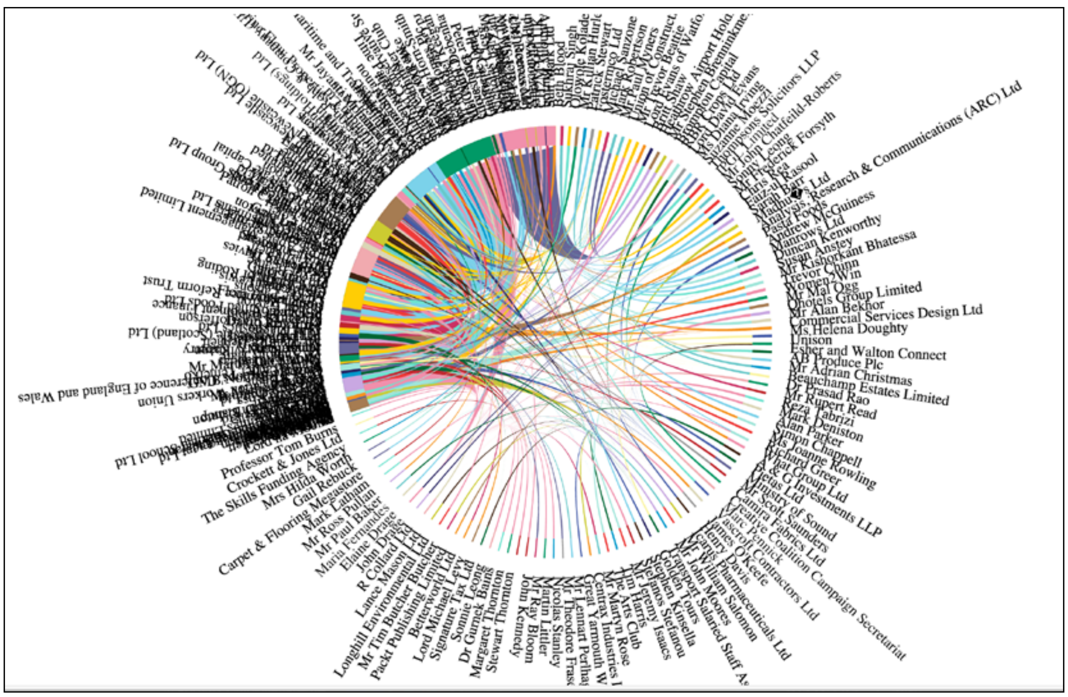
We add some CSS to `chapter5.css` to make the chords easier to follow:

```
.chord path {
      stroke: black;
      stroke-width: 0.2;
      opacity: 0.6;
}
```

Finally, replace the new call in `index.js` with the following:

```
new PoliticalDonorChart('chord');
```

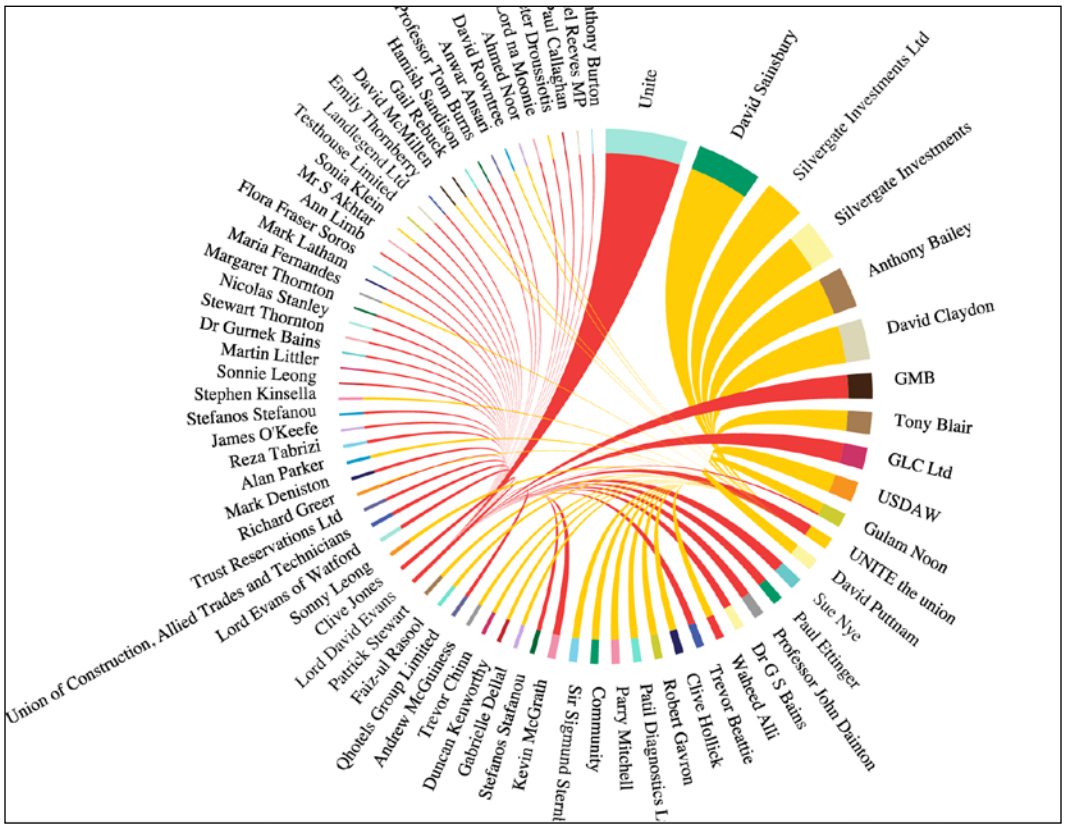
And now we have a very busy and hard-to-read diagram!



Pretty cool! If only it was more readable! Change the constructor to the following:

```
new PoliticalDonorChart('chord', 'Miliband');
```

This will get us Ed Miliband and his brother, David Miliband, both Labour MPs:



First of all, chord colors don't mean anything! They just make it easier to distinguish chords. Furthermore, this graph shows how big each donation is by the size of the chord.

A chord chart is suboptimal for this type of data because donations are pretty unidirectional—MPs typically don't give money back to donors. In the preceding chart, you see your arcs kind of tapering towards nowhere; this is because we don't have an item on the chart for where all the donations are going. Fixing it's a bit hacky, but easy enough. We put this code block before the line where we define `uniques` in our `chord` method:

```
let uniqueMPs = helpers.uniques(filtered, (d) =>
  d.EntityName);
uniqueMPs.forEach((v) => {
  filtered.push({
    DonorName: v,
    EntityName: v,
    Value: '9001'
  });
});
```

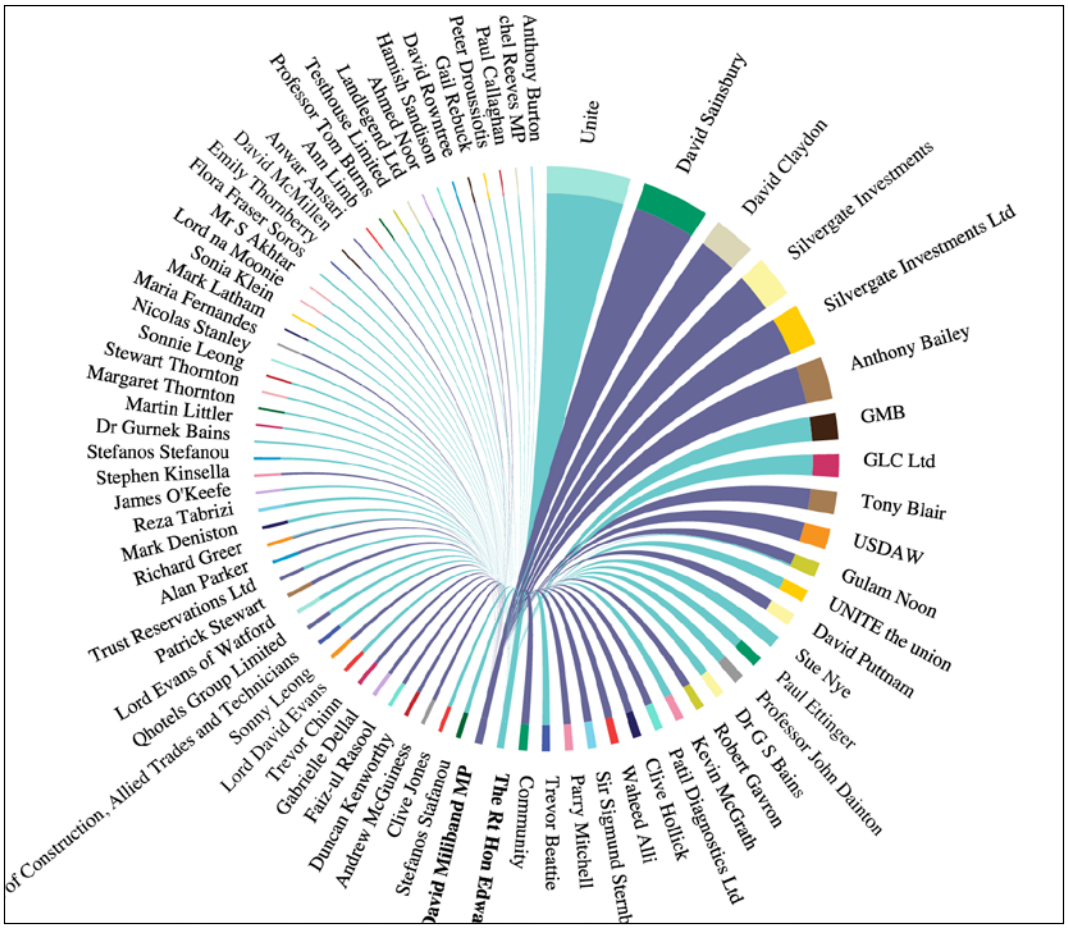
This just adds another entry and gives it a value that's over nine thousand (nine thousand?!), which is just an arbitrary number I chose that gives the entries a bit of size but not enough to distort the graphic. Feel free to play around with it.

Lastly, add the following code at the bottom of `chord` to make the entries you just added bold:

```
this.chart.selectAll('text').filter(function(d) {
  return d3.select(this).text().match(filterString)})
  .style('font-weight', 'bold');
```

There we go! It makes slightly more sense now! From this, you can tell that *Unite* (Britain's largest union, for those outside the UK) gave a lot of money to Ed Miliband's campaign, and *Silvergate Investments* gave a lot to David Miliband. You'll notice that both are listed twice; clearly this data needs some cleanup. As you build things in D3, you'll often find situations like this. You can do either one of two things: handle all these edge cases in your code, or clean up the data using something like OpenRefine. It's much better to clean up messy data before it gets to D3 as it results in cleaner and easier-to-read code with less hacky conditional logic.

For a good tutorial on using OpenRefine, visit `http://schoolofdata.org/handbook/recipes/cleaning-data-with-refine/`.



Drawing with force

The `force` layout is the most complicated of the non-hierarchical layouts. It lets you draw complex graphs using physical simulations—force-directed graphs if you will. Everything you draw will have built-in animation.

We're going to draw a graph of connections between donors and politicians. Every donor and politician will be a node, the size of which will correspond to the user's total donations. Links between nodes will tell us who is donating to whom.

To make things clearer, we're going to add tooltips and make sure that hovering over a node highlights the connected nodes.

Let's begin!

As in the chord example, we begin with a matrix of connections. We aren't going to feed this directly to the `force` layout, but we will use it to create the kind of data it enjoys. Start by changing the constructor in `index.js` to the following:

```
new PoliticalDonorChart('force');
```

Then add a `force` method to your chart class:

```
  force(filterString =  ' MP ') {
    let filtered = this.data.filter(
      (d) => d.EntityName.match(filterString) );
    let nameId = helpers.nameId(
      helpers.allUniqueNames(filtered), (d) => d);
    let uniques = helpers.allUniqueNames(filtered);
    let matrix = helpers.connectionMatrix(filtered);
  }
```

D3's `force` layout expects an array of nodes and links. Let's make them:

```
let nodes = uniques.map((name) => new Object({
  name: name,
  totalDonated: matrix[nameId(name)].reduce(
    (last, curr) => last + curr, 0),
  totalReceived: matrix.reduce((last, curr) =>
    last + curr[nameId(name)], 0)
}));

let links = filtered.map((d) => {
  return {
    source: nameId(d.DonorName),
    sourceName: d.DonorName,
    target: nameId(d.EntityName),
    targetName: d.EntityName,
    amountDonated:
matrix[nameId(d.DonorName)][nameId(d.EntityName)]
  }
});
```

We're defining the bare minimum of what we need, and the layout will calculate all the hard stuff.

The `nodes` tell us who they represent, and `links` connect a `source` object to a `target` object with an index into the `nodes` array. The layout will turn them into proper references, as shown in the following code. Every link also contains a `count` object that we'll use to define its strength:

```
    let force = d3.layout.force()
      .nodes(nodes)
```

```
        .links(links)
        .charge((node) => node.totalDonated ? -50 : 0)
        .gravity(0.05)
        .size([this.width, this.height]);

      force.start();
```

We create a new `force` layout with `d3.layout.force();` just like the `chord` layout, it isn't a function either. We feed in the data with `.nodes()` and `.links()`. The `charge` is set so that nodes that are donators repel each other, which will help prevent everything from just clumping in the center. The `gravity` setting pulls the graph towards the center of the image; we defined its strength with `.gravity()`. We tell the `force` layout the size of our picture with `.size()`. No calculation happens until `force.start()` is called, but we need the results to define a few scales for later.

There are a few more parameters to play with: the overall `.friction()` (the smallest `.linkDistance()` value the nodes stabilize to) and `.linkStrength()` for link stretchiness. Play with them. The `nodes` members now look like this:

```
{index: 0,
  name: "The Rt Hon Edward Miliband MP",
  px: 497.0100389553633,
  py: 633.2734045531992,
  weight: 100,
  x: 499.5873097327753,
  y: 633.395804766377}
```

The `weight` tells us how many links connect with this node, `px` and `py` state its previous position, and `x` and `y` state the current position.

The `links` members are a lot simpler:

```
{count: 2
  source: Object
  target: Object}
```

Both the `source` and `target` objects are a direct reference to the correct node. Now that the layout has made its first calculation step, we have the data needed to define some scales:

```
let distance = d3.scale.linear()
   .domain(d3.extent(d3.merge(matrix)))
   .range([300, 100]);

let given = d3.scale.linear()
    .domain(d3.extent(matrix, (d) => d3.max(d)))
   .range([2, 35]);
```

We're going to use the `given` scale for node sizes and `distance` for link lengths. Nodes that either receive or donate more will appear bigger, and nodes representing donors will be closer to the nodes they donated to based on the donated amount:

```
force.linkDistance((d) => distance(d.amountDonated));
force.start();
```

We use `.linkDistance()` to dynamically define link lengths according to the `.count` property. To put the change in effect, we restart the layout with `force.start()`.

Finally! Time to put some ink on paper—well—pixels on screen!

```
let link = this.chart.selectAll('line')
  .data(links)
  .enter()
    .append('line')
    .classed('link', true);
```

Links are simple. Go through the list of links and draw a `line`.

Next, draw a circle for every donor node and a square for every recipient:

```
let node = this.chart.selectAll('.node')
  .data(nodes)
  .enter()
    .append((d) => {
      return document.createElementNS(
        'http://www.w3.org/2000/svg', d.totalDonated > 0 ?
        'circle' : 'rect');
    })
    .classed('node', true);

this.chart.selectAll('circle.node')
  .attr({
    r: (d) => given(d.totalDonated),
    fill: (d) => helpers.color(d.index),
    class: (d) => 'name_' + nameId(d.name)
  })
  .classed('node', true);

this.chart.selectAll('rect.node')
  .attr({
    width: (d) => given(d.totalReceived),
    height: (d) => given(d.totalReceived),
    fill: (d) => helpers.color(d.index),
    class: (d) => 'name_' + nameId(d.name)
  })
  .classed('node', true);
```

The only unusual bit in the preceding code is us using a callback for the `.append` method, which requires a new element to be returned. Because we aren't able to use `.append` itself when we override its accessor (since we're conditionally making the element a circle or square depending on whether the node is a donor or recipient of a donation), we have to set the namespace and everything…

We add tooltips with the familiar `helpers.tooltip` function, and `force.drag` will automatically make the nodes draggable:

```
node.call(helpers.tooltip((d) => d.name, this.chart));
node.call(force.drag);
```

After all that work, we still have to do the updating on every tick of the `force` layout animation:

```
force.on('tick', () => {
  link.attr('x1', (d) => d.source.x)
  .attr('y1', (d) => d.source.y)
  .attr('x2', (d) => d.target.x)
  .attr('y2', (d) => d.target.y);

  this.chart.selectAll('circle.node').attr('cx', (d) => d.x)
  .attr('cy', (d) => d.y);

  this.chart.selectAll('rect.node').attr('x', function(d) {
    return d.x - this.getBBox().width / 2;
  })
  .attr('y', function(d) {
    return d.y - this.getBBox().height / 2
  });
});
```

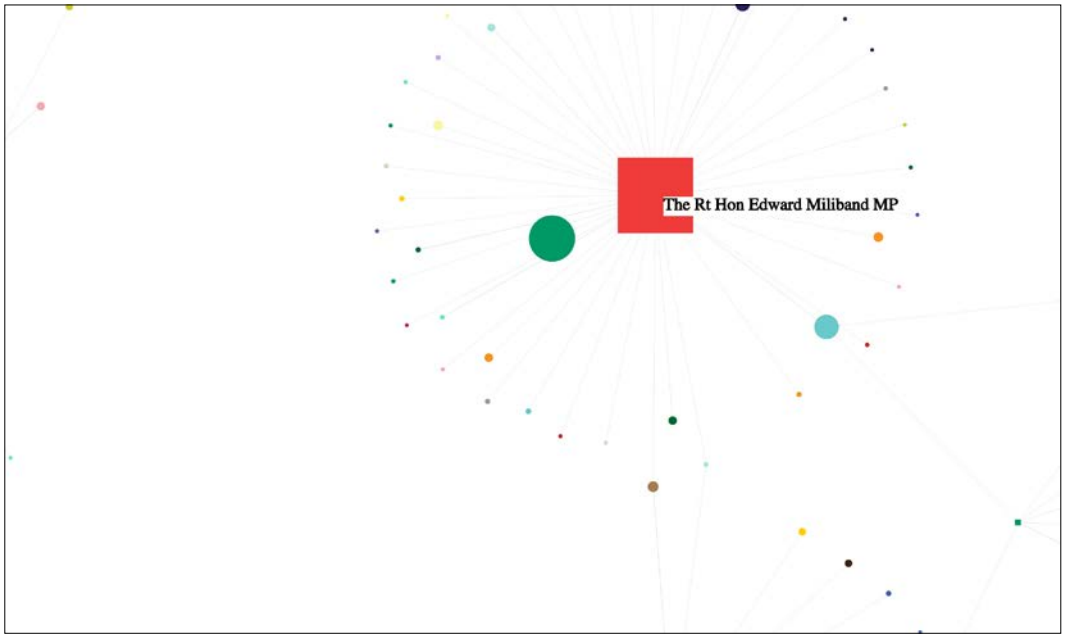
On a `tick` event, we move every `link` endpoint and `node` to its new position. Simple!

Add some styling to `chapter5.css`:

```
.link {
  stroke: lightgrey;
  stroke-width: 0.3;
}
```

And voilà! We get a force-directed graph of election donations.

Running this example looks silly because it spins around a lot before settling down. But once it stabilizes, the graph looks something like this:



Drag and pull it around the screen to see different connections.

We should have added some code to print names next to the highlighted nodes, but the example was long enough. Let's say that's left as an exercise for the reader.

We will now move on to hierarchical layouts!

Hierarchical layouts

All hierarchical layouts are based on an abstract hierarchy layout designed for representing hierarchical data—data within data within data within data within.... all the way down. As mentioned earlier, imagine a tree, or an org chart if you don't go outside very much.

All of the common code for the `partition`, `tree`, `cluster`, `pack`, and `treemap` layouts is defined in `d3.layout.hierarchy()`, and they all follow similar design patterns. The layouts are so similar that the official documentation very obviously copy-pastes most of its explanations which are practically identical. Let's avoid that by looking at the common stuff first, and then we will focus on the differences.

First of all, we need some hierarchical data. There's an implicit hierarchy in our data, insomuch that donations flow downwards, but it's not as well-defined as it would be for, say, organizational units of most businesses or the filesystem tree of a hard drive. The result is a scheme that works well with three of the layouts but looks slightly contrived for the other two—apologies in advance for that, but hopefully, it gives you the right idea.

We'll have a root node called `donations`, which will contain the names of people or groups that have donated in the last election cycle. For the `tree` and `cluster` layouts, each of those will contain nodes for everyone they have donated to. For the `partition`, `pack`, and `treemap` layouts, child nodes will tell us who contributed to the parent's total received donations.

The final data structure will look like this:

```
{
    "name": "donations",
    "children": [
        {
            "name": "Unite",
            "count": 5,
            "children": [
                {
                    "name": "Ed Balls",
                    "count": 2,
                    "children": []
                },
                {
                    "name": "Ed Miliband",
                    "count": 6,
                   "children": []
                }
}
```

While it could potentially go on forever, that wouldn't make sense in our case. We have only three levels in this example, but you can go down as many levels as you like.

The default accessor expects a `.children` property, but we could have easily done something crazy, such as dynamically generating a fractal structure in a custom accessor.

As usual, there's a `.value()` accessor that helps layouts to find data in a node. We'll use it to set the amount either donated or received.

To run a hierarchical layout, we call `.nodes()` with our dataset. This immediately returns a list of nodes that you can't get to later. For a list of connections, we call `.links()` with a list of our nodes. Nodes in the returned list will have some extra properties calculated by the layout. Most layouts tell us where to put something with `.x` and `.y`, and then use `.dx` and `.dy` to tell us how big the layout should be.

All hierarchical layouts also support sorting with `.sort()`, which takes a sorting function, such as `d3.ascending` or `d3.descending`.

Enough of theory! Let's add a data munging function to `helpers.js`:

```
export function makeTree(data, filterByDonor, name1, name2) {
  let tree = {name: 'Donations', children: []};
  let uniqueNames = uniques(data, (d) => d.DonorName)

  tree.children = uniqueNames.map((name) => {
    let donatedTo = data.filter((d) => filterByDonor(d, name));
    let donationsValue = donatedTo.reduce((last, curr) => {
          let value = Number(curr.Value.replace(/[^\d\.]*/g, ''));
          return value ? last + value : last;
    }, 0);

    return {
      name: name,
      donated: donationsValue,
      children: donatedTo.map((d) => {
        return {
          name: name2(d),
          count: 0,
          children: []
        };
      })
    };
  });

  return tree;
}
```

Wow, there's a lot going on here! We avoided recursion because we know our data will never nest more than two levels deep.

The `tree` holds an empty root node at first. We use `helpers.uniques` to get a list of names. Then we map through the array and define the children of the root node by counting everyone's donations and using `helpers.binPerName` to get an array of children.

The code is wibbly-wobbly because we use `filterByDonor`, `name1`, and `name2` for data accessors, but making this function flexible makes it useful in all hierarchical examples.

Drawing a tree

The `tree` layout displays data in a tree using the Reingold-Tilford tidy algorithm. We'll use it to display our dataset in a large circular tree, with every node connected to its parent by a curvy line.

We begin the method by fixating colors, turning data into a tree, and defining a way to draw curvy lines:

```
  tree(filterString = ' MP') {
    let filtered = this.data.filter(
      (d) => d.EntityName.match(filterString) );
    helpers.fixateColors(filtered);

    let tree = helpers.makeTree(filtered,
      (d, name) => d.DonorName === name,
      (d) => d.EntityName,
      (d) => d.EntityName || '');

    tree.children = tree.children.filter(
      (d) => d.children.length > 1)

    let diagonal = d3.svg.diagonal.radial()
      .projection((d) => [d.y, d.x / 180 * Math.PI]);
}
```

You know `fixateColors` from before. We defined `makeTree` about one page ago, and we talked about the `diagonal` generator in *Chapter 2, A Primer on DOM, SVG, and CSS*:

```
    let layout = d3.layout.tree()
        .size([360, this.width / 5]);
    let nodes = layout.nodes(tree);
    let links = layout.links(nodes);
```

We create a new tree layout by calling `d3.layout.tree()`. Defining its size with `.size()` and executing it with `.nodes().size()` tells the layout how much room it's got—in this case, we're using `x` as an angle (`360` degrees) and `y` as a radius, though the layout itself doesn't really care about that.

To avoid worrying about centering later on, we put a grouping element center stage:

```
let chart = this.chart.append('g')
    .attr('transform',
      `translate(${this.width / 2},${this.height / 2})`);
```

First, we are going to draw the links, and then the nodes and their labels:

```
    let link = chart.selectAll('.link')
      .data(links)
        .enter()
        .append('path')
        .attr('class', 'link')
        .attr('d', diagonal);
```

You should be familiar with this by now; go through the data and append new paths shaped with the `diagonal` generator:

```
    let node = chart.selectAll('.node')
      .data(nodes)
      .enter().append('g')
      .attr('class', 'node')
      .attr('transform',
        (d) => `rotate(${d.x - 90})translate(${d.y})`);
```

For every node in the data, we create a new grouping element and move it into place using `rotate` for angles and `translate` for radius positions.

Now it's just a matter of adding a circle and a label:

```
    node.append('circle')
      .attr('r', 4.5)
      .attr('fill', (d) => helpers.color(d.name))
      .on('mouseover', function(d) {
        d3.select(this.nextSibling).style('visibility',
'visible');
      })
      .on('mouseout', function(d) {
        d3.select(this.nextSibling).style('visibility', 'hidden');
      });
    node.append('text')
      .attr('dy', '.31em')
      .attr('text-anchor', (d) => d.x < 180 ? 'start' : 'end')
```

```
      .attr('transform', (d) => d.x < 180 ? 'translate(8)' :
'rotate(180)translate(-8)')
      .text((d) => d.depth > 1 ? d.name : d.name.substr(0, 15) +
(d.name.length > 15 ? '...' : ''))
      .style({
        'font-size': (d) => d.depth > 1 ? '0.6em' : '0.9em',
        'visibility': (d) => d.depth > 0 ? 'hidden' : 'visible'
      });
```

Every node is colored with the user's native color, and the text is transformed similarly to the earlier pie and chord examples. Finally, we've made leaf nodes' text smaller to avoid overlap.

Our tree looks something like this:

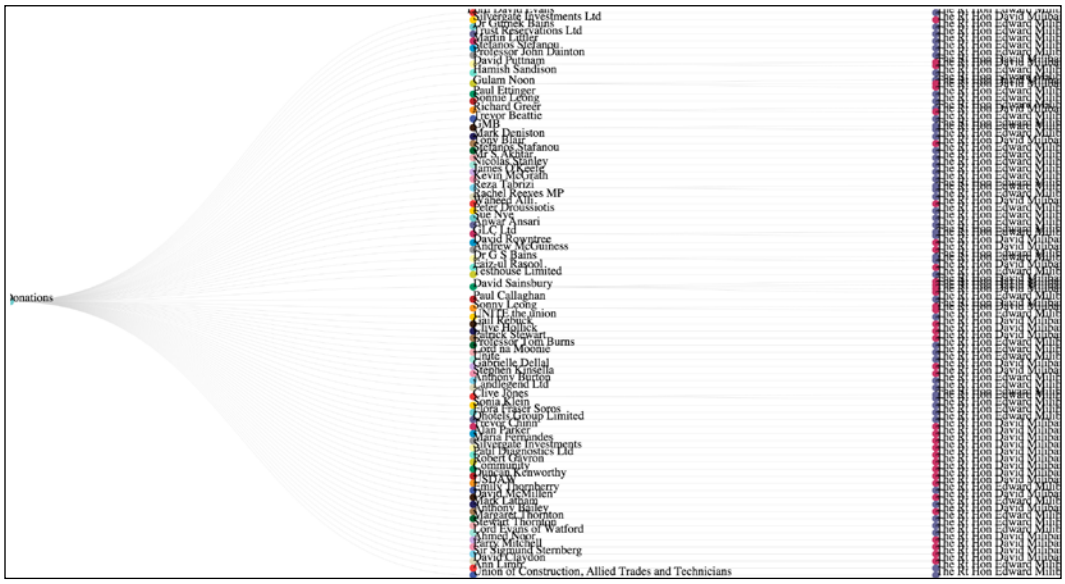
It's rather big, so you should try it out in the browser. Just remember that the inner ring represents users, people, or groups giving karma donations to the politicians in the outer ring.

Showing clusters

The `cluster` layout is the same as the `tree` layout, except that the leaf nodes line up.

Code-wise, this example is the same as the last, so we won't go through it again. Really, the only difference is that we don't have to flip labels at certain angles. You can look at the code in `chapter5.js` of the book's repo.

We end up with a very tall graph that looks something like this:



Partitioning a pie

Now we're getting somewhere! The next three layouts fit our data perfectly—we're taking three looks at how our core politicians' donations are structured.

The `partition` layout creates adjacency diagrams, where you don't draw nodes with links between them but next to each other so that it looks as if the children partition the parent.

We are going to draw a two-layer donut chart. Users will go on the first layer and the layer on top will show us where the donations are coming from.

We begin by munging the dataset and fixating colors; it's the same as before:

```
    partition(filterString = ' MP') {
      let filtered = this.data.filter(
        (d) => d.EntityName.match(filterString) );
      let tree = helpers.makeTree(filtered,
        (d, name) => d.DonorName === name,
        (d) => d.EntityName,
        (d) => d.EntityName || '');

      helpers.fixateColors(filtered);
  }
```

Then we use the `partition` layout:

```
      let partition = d3.layout.partition()
        .value((d) => d.parent.donated)
        .sort((a, b) =>
          d3.descending(a.parent.donated, b.parent.donated))
        .size([2 * Math.PI, 300]);

      let nodes = partition.nodes(tree);
```

We used `.value()` to tell the layout that we care about the `.donated` values, and we'll get a better picture if we `.sort()` the output. Similar to the `tree` layout, `x` will represent angles—this time in radians—and `y` will be radii.

We need an `arc` generator as well, as shown in the following code:

```
      let arc = d3.svg.arc()
        .innerRadius((d) => d.y)
        .outerRadius((d) => d.depth ? d.y + d.dy / d.depth : 0);
```

The generator will use each node's `.y` property for the inner radius and add `.dy` for the outer radius. Fiddling shows that the outer layer should be thinner. Hence, we are dividing it by the tree depth.

Notice that there's no accessor for `.startAngle` and `.endAngle`, which are stored as `.x` and `.dx`. It's easier to just fix the data:

```
      nodes = nodes.map((d) => {
        d.startAngle = d.x;
        d.endAngle = d.x + d.dx;
        return d;
      });
      nodes = nodes.filter((d) => d.depth);
```

It is as simple as mapping the data, defining angle properties, and then filtering the data to make sure that the root isn't drawn.

We use the familiar grouping trick to center our diagram:

```
let chart = this.chart.attr('transform',
  `translate(${this.width / 2}, ${this.height / 2})`);
```

The preparation work is done. It's drawing time!

```
    let node = chart.selectAll('g')
      .data(nodes)
      .enter()
      .append('g');

    node.append('path')
    .classed('partition', true)
    .attr({
      d: arc,
      fill: (d) => helpers.color(d.name)
    });
```

An arc is drawn for every node. The color is chosen as usual. Lastly, we add tooltips because adding labels is a bit too messy on this one:

```
node.call(helpers.tooltip(function (d) { return d.nick; }));
```

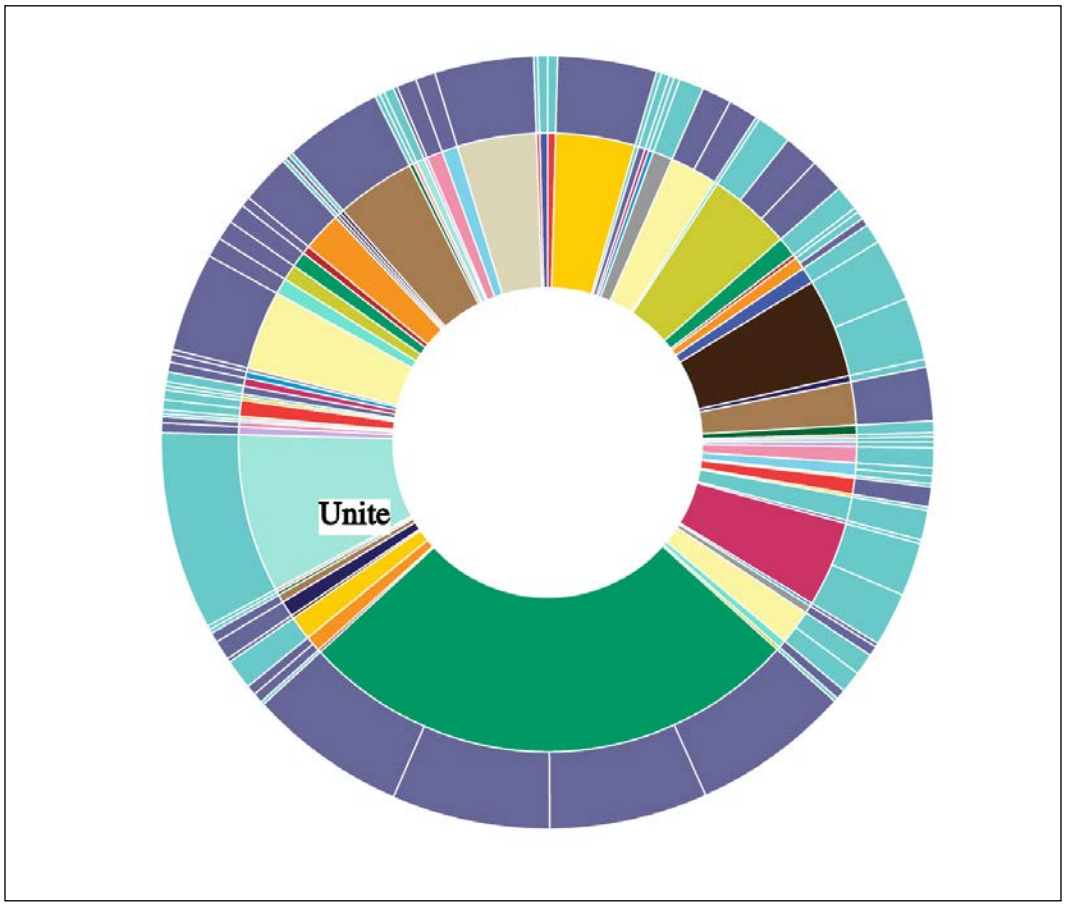
Add some more CSS to `chapter5.css`:

```
path.partition {
  stroke: white;
  stroke-width: 1;
}
```

Finally, instantiate in `index.js`. We're going to continue filtering by the `Miliband` brothers:

```
new PoliticalDonorChart('partition', 'Miliband');
```

The adjacency diagram looks like this:

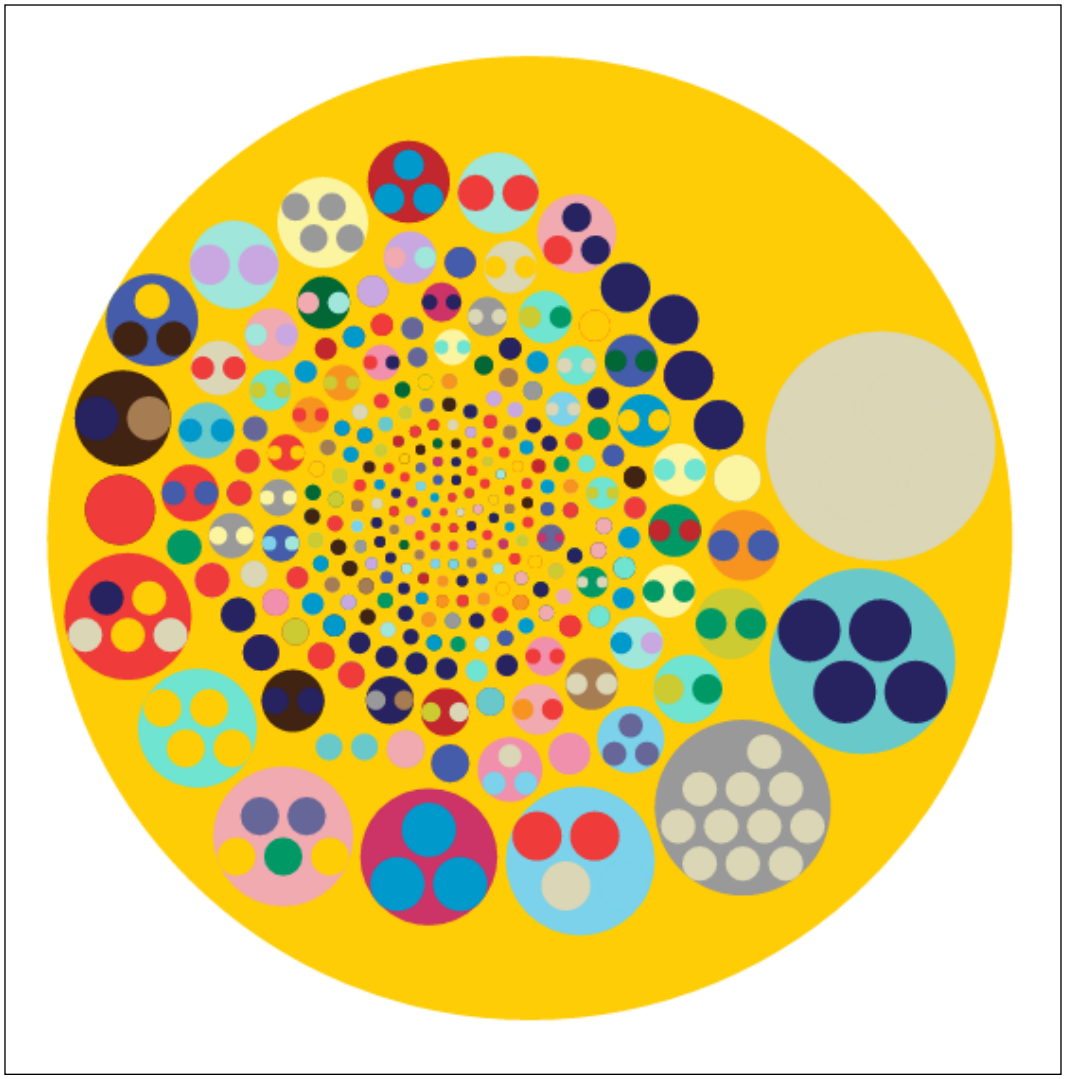


The outer segments represent either Ed or David Miliband (depending on the color), and the inner segments represent the donors

Packing it in

The `pack` layout uses packing to visually represent hierarchies. It stuffs children nodes into their parents, trying to conserve space and sizing each node so that it's the cumulative size of its children.

Conceptually, it's very similar to the `treemap` layout, so I'm going to skip all of the code and just show you the image. To see the code that generated it, check out the `pack` method in `chapter5.js`:



It looks nice, but the pack layout probably isn't the best way to visualize this particular dataset

The code is rather familiar—generate a tree, fixate the colors, create the layout, tweak a few parameters, get computed nodes, draw the nodes, and add tooltips. Simple!

It looks very pretty, but it's not too informative. Adding labels wouldn't help much either because most of the nodes are too small.

Subdividing with treemap

The `treemap` layout subdivides nodes with horizontal and vertical slices, essentially packing children into their parents, just like the `pack` layout, but using rectangles. As a result, node sizes on every level can be compared directly, making this one of the best layouts for analyzing cumulative effects of subdivisions.

We are going to have some fun with this example. Tooltips will name the parent—parents are almost completely obscured by the children—and moving the mouse arrow over a node will make unrelated nodes become lighter, making the graph less confusing (at least in theory).

It's also a cool effect and a great way to end this chapter on layouts. But we begin with the boring stuff—prepare data and fixate the colors:

```
  treemap(filterString = ' MP') {
    let filtered = this.data.filter(
      (d) => d.EntityName.match(filterString) );
    let tree = helpers.makeTree(filtered,
      (d, name) => d.DonorName === name,
      (d) => d.EntityName,
      (d) => d.EntityName || '');

    helpers.fixateColors(filtered);
}
```

Creating the `treemap` layout follows familiar patterns:

```
    let treemap = d3.layout.treemap()
      .size([this.width, this.height])
      .padding(3)
      .value((d) => d.parent.donated)
      .sort(d3.ascending);

    let nodes = treemap.nodes(tree)
      .filter((d) => d.depth);
```

We added some padding with `.padding()` to give the nodes room to breathe. Every node will become a group element holding a rectangle. The leaves will also hold a label:

```
    let node = this.chart.selectAll('g')
      .data(nodes)
      .enter()
      .append('g')
      .classed('node', true)
```

```
    .attr('transform', (d) => `translate(${d.x},${d.y})`);

  node.append('rect')
    .attr({
      width: (d) => d.dx,
      height: (d) => d.dy,
      fill: (d) => helpers.color(d.name)
    });
```

Now, for the first fun bit. Let's fit labels into as many nodes as they can possibly go:

```
  let leaves = node.filter((d) => d.depth > 1);

  leaves.append('text')
    .text((d) => {
      let name = d.name.match(/([^\s]+\s[^\s]+) MP$/)
        .shift().split(' ');
      return `${name[0].substr(0, 1)}. ${name[1]}`;
    })
    .attr('text-anchor', 'middle')
    .attr('transform', function (d) {
      let box = this.getBBox();
      let transform = `translate(${d.dx / 2},
        ${d.dy / 2 + box.height / 2})`;

      if (d.dx < box.width &&
          d.dx > box.height && d.dy > box.width) {
        transform += 'rotate(-90)';
      } else if (d.dx < box.width || d.dy < box.height) {
        d3.select(this).remove();
      }

      return transform;
    });
```

Finally! That was some interesting code!

We found all the leaves and started adding text. To fit labels into nodes, we get their size with `this.getBBox()`. Then we move them to the middle of the node, and check for fit. We also do a bit of regex and array manipulation to the text so that it removes everything except the last name and the initial before that. This is not the most sure-fire way to sanitize those strings—it's much better to use something like OpenRefine to clean up your data beforehand and put it in the format you ultimately want to present it as—but it works for our purpose right now.

If the label is too wide but fits vertically, we rotate it; otherwise, we remove the label after verifying again that it doesn't fit. Checking the height is important because some nodes are very thin.

We add tooltips with `helpers.tooltip`:

```
leaves.call(helpers.tooltip((d) => d.parent.name, this.chart));
```

Another fun bit—partially hiding nodes from different parents:

```
leaves.on('mouseover', (d) => {
  let belongsTo = d.parent.name;
  this.chart.selectAll('.node')
    .transition()
    .style('opacity', (d) => {
      if (d.depth > 1 && d.parent.name !== belongsTo) {
        return 0.3;
      }

      if (d.depth == 1 && d.name !== belongsTo) {
        return 0.3;
      }

      return 1;
    });
})
.on('mouseout', () => {
  d3.selectAll('.node')
    .transition()
    .style('opacity', 1);
})
.on('click', (d) => alert(d.name));
```

We used two mouse event listeners: one creates the effect, another removes it. The `mouseover` listener goes through all the nodes and lightens those with a different parent or those that aren't parents (that is, `d.parent.name` and `d.name` are different). This listener removes all changes. We also have it do an alert popup with the full label text if we click on any of the `rect` elements.

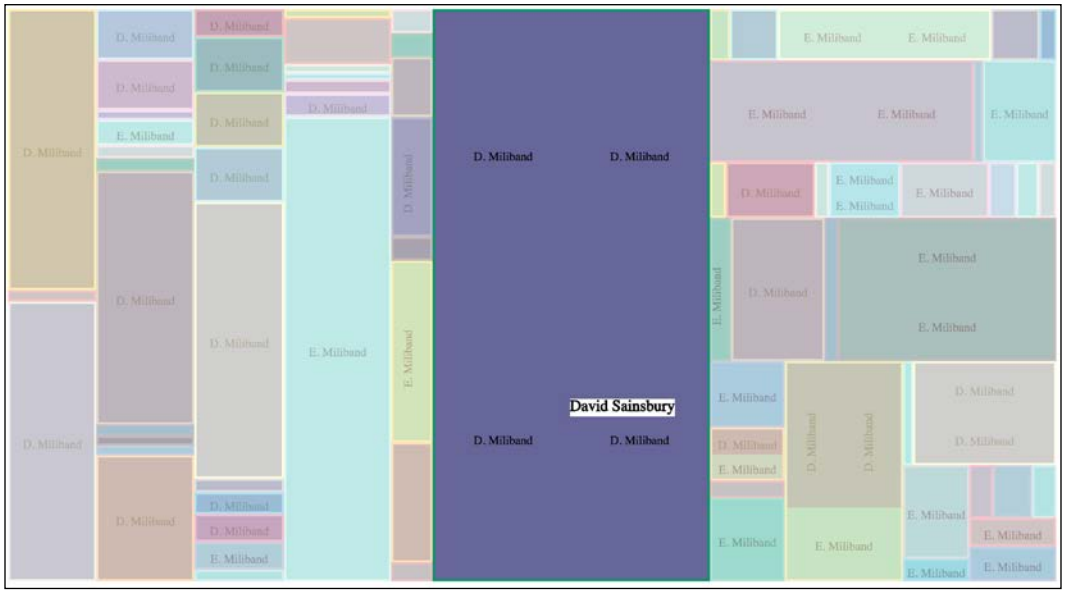
The `alert()` is without a doubt the most disgusting way of presenting information to the user. Even after ignoring the indescribably obnoxious ways in which this browser "feature" was used by Internet marketers in the 90s, it should still never be used, given their `alert()` method's visual similarity to operating system alerts, which can confuse the recipient. Furthermore, you can't style them, and they steal focus from whatever else is going on in the browser. You shouldn't even use them for debugging; `console.log` and `console.dir` are far more descriptive for dumping variables.

I used them in this example purely for demonstrative purposes; in all actuality, a much better way to handle click events would be to update a text element beneath the treemap. I'll leave this as an exercise for the reader.

After this, add a few more lines of CSS to `chapter5.css`:

```
.name text {
  font-size: 1.5em;
}

.name rect {
  fill: white;
}
```

The end result looks like an abstract painting:



Each grouping is a single donor, with the size of its contents reflecting the size of the donation.

Summary

Despite the near-mythical power of D3 layouts, they turn out to be nothing more than helpers that turn your data into a collection of coordinates.

After going all-out with these examples, we used almost every trick we've explained so far. We even wrote so much code that we had to make a separate library! With a bit of generalization, some of those functions could be layouts of their own. There's a whole world of community-developed layouts for various types of charts. The `d3-plugins` repository on GitHub (`https://github.com/d3/d3-plugins`) is a good place to start exploring.

You now understand what all the default layouts are up to, and I hope you're already thinking about using them for purposes beyond the original developers' wildest dreams.

In the next chapter, you'll learn how to use D3 *outside* of the browser. That's right folks, we're headed to Server Town! In doing so, we'll strip D3 right down to its bare bones and use it to render things that aren't even SVG!

6
D3 on the Server with Node.js

Here's where we start to get really funky with D3. Not only can it render beautiful charts on the frontend, but we can also use D3 to generate things before they even get to the user's browser. This is really the cutting edge of D3, so realize that the skills you've learned in the preceding chapters will serve you in 95 percent of situations and don't sweat it if you want to stick to the frontend for now. This chapter will still be here for that rainy afternoon when you want to try to figure out how **Heroku** works.

We've added this chapter to the second edition because it uses D3 in a really abstract sense and we can start to tie up a lot of the stray concepts we've started discussing in the book

Readying the environment

Ever written server apps in PHP? If so, you're in for a treat. JavaScript web applications are a million times easier to deploy thanks to **Platform as a Service** (**PaaS**) webhosting providers, and you can manage an entire fleet of servers using a few simple tools. Instead of fighting with a huge monolithic Apache or Nginx configuration, you can deploy a new instance for every app you create, which sandboxes them and allows for much more compact infrastructure. We'll discuss how to deploy to Heroku later in the chapter; for now, we're just going to test everything locally.

What is Heroku and do you have to use it? Heroku is a way of deploying applications that use "12-factor app" principles (for the specifics, visit `http://12factor.net`). Without going too deep into the 12-factor app philosophy, the idea is that you try to create *stateless* applications that use web services in place of a large, monolithic piece of server infrastructure (for instance, a Linux-based virtual server running both the webserver and database processes). I use Heroku in this instance because it's simple to deploy to and free to use in limited capacities, but you can also deploy the code we'll write in this chapter on any server infrastructure that has Node.js installed.

You may not know it, but practically everything you need to write a server application resides in our project directory. If you've never done Node.js development before, it's very similar to what we've done in the preceding chapters with the browser, but with its own APIs and concepts. The JavaScript engine running on both Google Chrome and Node.js is called "V8", so you don't have to learn a totally different set of languages or skills in order to start immediately building server applications.

We're going to install a few more dependencies via `npm`:

```
$ npm install express body-parser --save
```

`express` is the leading Node.js web server library; it nicely abstracts Node.js's ability to open ports and serve content into an easy-to-use API that is very light and fast. However, because Node.js uses the CommonJS module loading standard, we need to add a new set of build instructions to our `webpack.config.js`. We need to make the object being exported an array (so both configurations run), so make it look as follows (the first config has been truncated for space reasons):

```
var path = require('path');

module.exports = [
  { ... }, // This is the first config; leave it be!
  {
      name: 'server',
      entry: './src/chapter6.js',
      target: 'node',
      output: {
          path: path.resolve(__dirname, 'build'),
          filename: 'server.js'
      },
      externals: {
        canvas: 'commonjs canvas'
```

```
    },
    module: {
      loaders: [
        {
          test: /\.js?$/,
          exclude: 'node_modules',
          loader: 'babel'
        },
        {
          test: /\.json$/,
          loader: 'json-loader'
        }
      ]
    }
  }
];
```

It's pretty much the same; we've just changed the target to `node`. The only thing that's different is we've told Webpack to treat `node-canvas` (which we'll use later in the chapter) as an external library instead of trying to bundle it.

Note that this will still emit all your other code during a build; if it's going slowly, feel free to temporarily disable the first `config` by commenting it out.

All aboard the Express train to Server Town!

Okay, let's get into the nitty-gritty right away and I'll explain what's going on. Add all of this into a new file called `chapter6.js`:

```
import express from 'express';
import bodyParser from 'body-parser';
import d3 from 'd3';
import {readFile} from 'fs';

import {nearestVoronoi} from './helpers';

let app = express();

app.use(bodyParser.urlencoded());
```

Here we're just importing all of our libraries; nothing to see here... Oh wait — didn't I say we have to use CommonJS because we're in Nodeville? No — because we're using Babel and Webpack to transpile all of our code, we can still use the lovely ES2015 module loading syntax without issue — even for Node.js modules that aren't themselves in ES6!

Anyway, we essentially instantiate Express, assign it to `app` and tell it to use the `urlencoded` form data body parser for `POST` requests (we installed this at the same time as `express`).

Add the following code:

```
app.get('/', (req, res) => { res.send('Hi there!'); });

app.listen(process.env.PORT || 8081, () => {
  console.log("We're up and running on " +
http://localhost:(process.env.PORT || 8081);)
});
```

This sets up a route for `GET` (that is, a normal visit to a page, as opposed to a form submission, for instance) on the application's root path. Then it starts listening for requests on either port `8081` or *whatever that $PORT environment variable is set to*. I've emphasized that last line because it's a key tenant in building web applications like this — keep all configuration in environment variables, so you never have to worry about storing plaintext passwords in your source code.

What are these "environment variables" I keep mentioning?

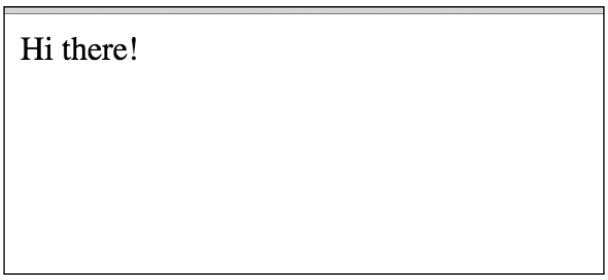
Your shell keeps a number of variables persistent that it uses to do things — you might be familiar with `$PATH`, which is a list of directories the shell looks in for executable code. Environment variables can also be used by web servers to hold configuration details, which can then be consumed by web applications (such as the one we're making!).

One example of where this is useful is database connection details — you generally don't want to version control sensitive details such as database passwords, so instead you provide them to your web server as environment variables, which your application then uses to connect to the database. There are a ton of other benefits to this, but suffice it to say, making your applications configurable through environment variables is a key aspect of writing good JavaScript server applications.

Now go back to the command line and type the following:

```
$ npm run server
```

Then visit `http://localhost:8081` and you should be greeted with the following text:



Congrats, you are now a bonafide web applications developer!

Well, not quite, but, *rapidly getting there!*

One thing worth noting at this point is that all instructions in a closure execute simultaneously, just like in the browser. In this way, JavaScript is said to be "asynchronous" — by executing everything simultaneously, you can't rely on something to be blocking like you can in PHP, for instance. Although it's a bit hard to wrap your head around this when starting in Node.js, this has really exciting ramifications for server code, making it very easy and very quick to scale horizontally.

Let's do something way less lame, and let's break out a brand new D3 feature while we're at it.

Proximity detection and the Voronoi geom

A **Voronoi geom** chops a geographic shape into discrete regions around points, such that no section overlaps and the entirety of the area is covered. Anything within a particular point's section is closer to that point than to any other point. We're going to use this to figure out what the closest major airport is to your current location, which we'll supply via the HTML5 Geolocation API.

Replace your call to `app.get` in `chapter6.js` with the following:

```
app.get('/', (req, res) => {
  res.send(`<!doctype html>
    <html>
```

```
    <head>
      <title>Find your nearest airport!</title>
    </head>
    <body>
      <form method="POST" action="/">
        <h1>Enter your latitude and longitude, or allow your
browser to check.</h1>
        <input type="text" name="location" /> <br />
        <input type="submit" value="Check" />
      </form>
      <script type="text/javascript">
      navigator.geolocation.getCurrentPosition(function(position)
{
        document.getElementById('latlon').value =
position.coords.latitude + ',' + position.coords.longitude;
      });
      </script>
    </body>
    </html>`);
});
```

You'll notice we put view code inside of our web server logic, which isn't great, but it works for our purposes. You'll want to use Express's views system for anything more elaborate than what we're doing.

If you haven't recently, restart the server by press *Ctrl* + *C* and then running the following:

```
$ npm run server
```

It's worth noting that, unlike in the frontend, we need to restart the server every time there's a change. If something isn't working as expected, try restarting it.

This sends the web browser a basic HTML document that asks them to fill in latitude and longitude as a comma-separated value. Or, if they accept the browser's request to use the HTML5 Geolocation API, it will auto-populate.

Next we need to create a new function for the Voronoi calculations. Inside of `helpers.js`, add the following:

```
export function nearestVoronoi(location, points) {
  let nearest = {};
  let projection = d3.geo.equirectangular();
```

```
    location = location.split(/,\s?/);

    let voronoi = d3.geom.voronoi(
      points.map((point) => {
        let projected = projection([point.longitude,
          point.latitude]);
        return [projected[0], projected[1], point];
      }))
      .filter((d) => d);

    voronoi.forEach((region) => {
      if (isInside(projection([location[1], location[0]]), region))
      {
        nearest = {
          point: region.point[2],
          region: region
        };
      }
    });

    if (nearest === {}) throw new Error('Nearest not findable');
    else return nearest;
  }
```

Lots and lots going on here; let's unpick it a bit.

We have two function arguments, `location` and `points: location` is a comma-separated latitude/longitude pair, `points` will be an array with all of our airports when we supply it from our server code in `chapter6.js`.

We then start things off by splitting our latitude/longitude string into an array, and setting up an equirectangular projection like we did with our first map all the way back in *Chapter 3*, *Making Data Useful*.

Finally, we get to configuring our Voronoi geom. Despite how complex this might look, it's pretty simple underneath — you ultimately just supply it with a bunch of points and it calculates the max size of each region. We do this via an `Array.prototype.map`, where we first project the longitude and latitude, then return these as an array. We also include the original `airport` object in the array so we know which point corresponds to which airport.

You can include as many additional array elements as you want and they'll all show up in each Voronoi region's `.point` object. The Voronoi geom only reads the first two elements of the array when constructing the Voronoi regions.

We also filter by the datum's identity, as follows:

```
      .filter((d) => d);
```

This is because the Voronoi geom will return a point as `undefined` if there are any duplicates (such as there are in this particular dataset), which will mess up other stuff (particularly if you want to then use a path generator to draw your Voronoi regions). This strips out all the undefined items.

Our Voronoi regions in hand, we then do a `.forEach` loop to check whether our user's location is inside of each region. Once we find which Voronoi region our point is interior to, we return that. If none are found, we throw an error.

The last thing we need to do is write the `isInside` function, which we'll also put in `helpers.js`. This is taken from `substack/point-in-polygon` on GitHub, which you can install via npm if you want to save yourself some typing:

```
export function isInside(point, polygon) {
  let x = Number(point[0]), y = Number(point[1]);

  let inside = false;
  for (let i = 0, j = polygon.length - 1; i < polygon.length; j =
i++) {
      let xi = polygon[i][0], yi = polygon[i][1];
      let xj = polygon[j][0], yj = polygon[j][1];

      let intersect = ((yi > y) != (yj > y))
          && (x < (xj - xi) * (y - yi) / (yj - yi) + xi);
      if (intersect) inside = !inside;
  }

  return inside;
}
```

Without getting too deep into the preceding code, it ultimately draws lines out from a test point and looks at how many boundaries it crosses. It's based on the code from the following URL, which goes into much greater detail: `https://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html`.

Now we just need to wire this all up back in `chapter6.js`. Instead of a `GET` request, this time we're going to handle a `POST` request (which will contain our current location) to the same address.

```
app.post('/', (req, res) => {
  let location = req.body.location;
```

```
  let airportPromise = new Promise((res, rej) =>
    readFile('src/data/airports.dat', 'utf8',
      (err, data) => err ? rej (err) : res(data)));
});
```

Pretty straightforward — when Express receives a `POST` request (such as when you submit a form), `body-parser` will intercept the form data and assign it to `req.body`. Next, we create a new `Promise`, where we use `fs.readFile` to load in our airport database.

We use `fs.readFile` because Node doesn't know how to `use d3.csv`, which uses `XMLHttpRequest`, which is only in the browser. Because the file we want is on the same computer as our server application, we can just load it in from the disk instead.

Let's resolve that promise and feed our helper functions some data:

```
airportPromise.then((airportData) => {
  let points = d3.csv.parseRows(airportData)
    .filter((airport) => !airport[5].match(/\N/)
      && airport[4] !== '')
    .map((airport) => {
      return {
        name: airport[1],
        location: airport[2],
        country: airport[3],
        code: airport[4],
        latitude: airport[6],
        longitude: airport[7],
        timezone: airport[11]
      }
    });

  let airport = nearestVoronoi(location, points);
})
.catch((err) => console.log(err));
```

This is pretty straightforward; we're just reformatting our data so it's a bit nicer to work with. Note that we filter out airports without both the airport codes, which means we'll only get large international airports.

It's worth noting the `airports.dat` data set not only includes airports ranging in size from tiny Moose Jaw Municipal Airport to the gargantuan Beijing International, but also includes some international rail stations — for instance, London St. Pancras International. Which, *I guess* kind of makes sense? Trains are *sort* of like ground planes, right?

After that, we put both our location and point arrays into our `nearestVoronoi` function, which we assign to `airport`. We also log any errors to the console via `Promise.catch`, because otherwise we won't have any idea what's going on if it fails.

The only thing left to do now is return a HTML document to the web browser if we're successful. Add this after `let airport`:

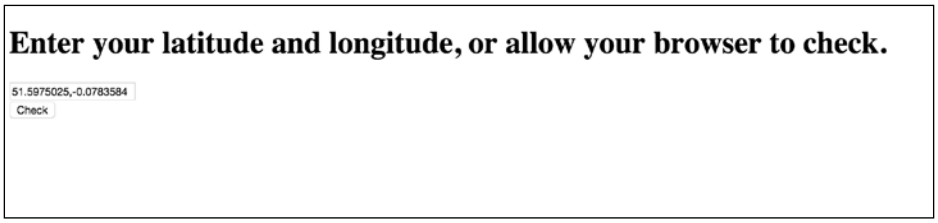
```
res.send(`<!doctype html>
<html>
<head>
  <title>Your nearest airport is: ${airport.point.name}</title>
</head>
<body style="text-align: center;">
  <h1>
    The airport closest to your location is: ${airport.point.name}
  </h1>
  <table style="margin: 0 auto;">
  <tr>
    ${Object.keys(airport.point).map((v) =>
     `<th>${v}</th>`).join('')}
  </tr>
  <tr>
    ${Object.keys(airport.point).map((v) =>
      `<td>${airport.point[v]}</td>`).join('')}
    </tr>
  </body>
  </html>`);
```

It's not going to win any prizes for beauty, but it does summarize the data with a minimal amount of code.

Okay, go back to your terminal. Hit *Ctrl + C* to kill off Node.js if it was still running from earlier, and then run the following from the root of your project directory:

```
$ npm run server
```

It'll take a second for Webpack to churn through everything, but once it's done, visit `http://localhost:8081` to see a screen like the following:



Either enter in a latitude/longitude pair, or let your web browser take your current position instead. Click `Submit` to see the results:

Hey, that's actually kind of cool! And we've figured something out using D3 that didn't even involve drawing anything!

By now you should be seeing D3 less as a bunch of magical tools that turn data into pretty visual things, and more just as a collection of functions that output mathematics a particular way. While D3 is certainly the most interesting when it's drawing things in a web browser, it's such a powerful library that you can use it for things far removed from its original use case of rendering SVG in the DOM.

Rendering in Canvas on the server

How about we do another one of those things right now? As mentioned before, the output from our little server app is pretty dull. Let's render a map using Canvas!

For this to work on the server, we're going to need to install `node-canvas`, which uses Cairo as a dependency. Assuming you're in Mac OS X and have Homebrew installed, run the following:

```
$ brew install pkg-config cairo libpng jpeg giflib
```

If you're not an OS X user with Homebrew installed, I suggest visiting `https://github.com/Automattic/node-canvas` and following the instructions there.

Alternatively, you can skip this entirely if you don't care to test locally, as we'll be deploying this all to Heroku later on in the chapter.

Next, add `node-canvas` to our app's dependencies:

```
$ npm install canvas earth-topojson --save
```

It's worth noting that this is a super weird way to use Canvas compared to how we can in the browser; normally we'd just rely on the browser's built-in Canvas renderer, but we don't have that luxury on the server. Note however, that the Canvas code we're writing for `node-canvas` will work the same way in the browser, in case you want to use it there.

We also install `earth-topojson`, which is a quick and dirty way of getting some basic political boundaries into a project.

Create a new function in `chapter6.js`:

```
function drawCanvasMap(location, airports) {
  let Canvas = require('canvas');
  let topojson = require('topojson');

  let canvas = new Canvas(960, 500);
  let ctx = canvas.getContext('2d');
  let projection = d3.geo.mercator()
    .center([location.split(/,\s?/)[1],
      location.split(/,\s?/)[0]])
    .scale(500);
}
```

This sets up `Canvas` and creates a new Mercator projection centered on the user's location. `ctx` is your canvas context; it's where and how you do your drawing.

Next, add the following to `drawCanvasMap`:

```
  let boundaries = require('earth-topojson/110m.json');
  let airport = nearestVoronoi(location, airports);
  let airportProjected = projection([airport.point.longitude,
airport.point.latitude]);

  let path = d3.geo.path()
    .projection(projection)
```

```
      .context(ctx);

  ctx.beginPath();
  path(topojson.feature(boundaries,
boundaries.objects.countries));
  ctx.stroke();

  ctx.fillStyle = '#f00';
  ctx.fillRect(airportProjected[0] - 5 , airportProjected[1] - 5,
10, 10);

  return canvas.toDataURL();
```

The first three lines are what you'd expect — first we load in our boundaries, then we use our Voronoi function from earlier to calculate the nearest airport. From that we get a point, which we project into our coordinate system.

Then it's time to draw: we create a geo path generator, and supply `ctx` to it using the `.context` method. This is really pretty cool, because D3 does all the hard work in drawing paths in Canvas for us.

With our path generator doing its thing, we tell `ctx` to start drawing a path. We then run the path generator on our geometry, and add a stroke to it. Unlike what we normally do in D3, we don't chain these methods — we run them one right after the other, or *procedurally*.

We then draw a tiny little square on our closest airport, because that's how we do!

Finally, we use one of Canvas' super awesome features, which is outputting directly to a base64-encoded PNG data string. We return this data URL, because we'll be updating our `POST` method to output it.

Inside the call to `app.post`, add the following line above the call to `res.send`:

```
let canvasOutput = drawCanvasMap(location, points);
```

Then replace the call to `res.send` with the following:

```
    res.send(`<!doctype html>
      <html>
      <head>
        <title>Your nearest airport is:
${airport.point.name}</title>
      </head>
      <body style="text-align: center;">
        <h1>The airport closest to your location is:
${airport.point.name}</h1>
```

```
          <img style="width: 480px; height: 250px;"
src="${canvasOutput}" />
          <table style="margin: 0 auto;">
          <tr>
            ${Object.keys(airport.point).map((v) =>
`<th>${v}</th>`).join('')}
          </tr>
          <tr>
            ${Object.keys(airport.point).map((v) =>
`<td>${airport.point[v]}</td>`).join('')}
          </tr>
        </body>
        </html>`);
```

This simply adds an image tag, with the `src` attribute set to our data string. If you look at it by right-clicking and selecting **Inspect Element**, you can see what our function is actually outputting, as follows:

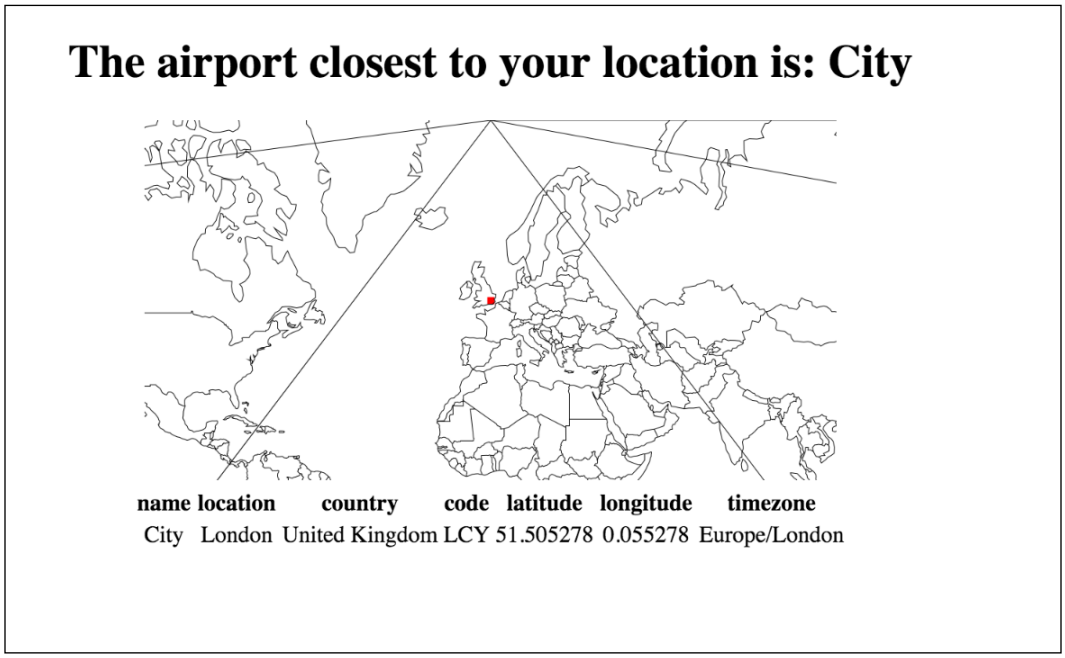
```
<!doctype html>
      <html>
      <head>
        <title>Your nearest airport is: City</title>
      </head>
      <body style="text-align: center;">
        <h1>The airport closest to your location is: City</h1>
        <img style="width: 480px; height: 250px;"
```

This is what an image rendered to a base64 string looks like

Hit *Ctrl* + *C* if you still have Node.js running and then restart it by typing the following:

```
$ npm run server
```

Open up `localhost:8081`, enter a lat/long pair, and now your results page has a handy little static map!



Pretty cool, huh? Canvas, despite being a bit weird to use in D3, is a super powerful technology, particularly when rendering huge amounts of data (any DOM-based display language tends to get really slow after about 1000 elements; Canvas is effectively just a 2D drawing plane, so never has that problem).

Deploying to Heroku

A server app isn't very useful without a server!

Luckily, **Heroku** provides free plans for limited use and is super easy to deploy to. At the moment, they allow 18 hours of uptime per day on the free plan, with your machine downcycling when it isn't active (in effect, this means that your server generally won't ever run out of uptime hours provided it isn't being hit with traffic constantly).

Start by creating an account at `http://www.heroku.com` and install the Heroku Toolbelt from `http://toolbelt.heroku.com`. Once you've done so, go to the root of your project folder and type the following:

```
$ heroku create
```

This will create a new Git remote and set up your app at a random URL, like `https://calm-dusk-16214.herokuapp.com/`.

Next, create a new file named `Procfile`. Heroku looks at this when you deploy to know how to run your app. Add the following contents:

```
web: node build/server.js
```

Save, then make sure you have the latest bundle built, as follows:

```
$ npm run server
```

Hit *Ctrl* + *C* after it builds; we don't need to run it any more.

Because the servers you get from Heroku are really basic by default, we need to find a way of installing Cairo and any other dependencies node-canvas might have. Run the following:

```
$ heroku config:add BUILDPACK_URL=https://github.com/mojodna/heroku-buildpack-multi.git#build-env
```

This tells Heroku to use a custom buildpack when deploying. A buildpack is effectively just a recipe for configuring a server a certain way. We define our buildpacks in a new file called `.buildpacks` (notice the dot at the beginning). Add the following to it:

```
https://github.com/mojodna/heroku-buildpack-cairo.git
https://github.com/heroku/heroku-buildpack-nodejsNode.js.git
```

Finally, add everything to a commit:

```
$ git add . && git commit -am "Time for Heroku"
```

We're now going to deploy! Assuming you're working from the master branch, type the following:

```
$ git push heroku master
```

Heroku always deploys from the master branch. If you've checked out the `chapter6` branch in Git, type the following instead:

```
$ git push heroku chapter6:master
```

Visit the URL provided by `heroku create` — and you should see your app in all its glory, online and accessible to anyone on the Internet! Congratulations, you've just written a pretty awesome webapp!

Didn't think you'd get a crash course on writing backend code in a book about data visualization, did you?!

Summary

In this chapter, first we set up Webpack to produce a separate bundle for the server, then we wrote a simple webapp using Express and D3's Voronoi geom to find the nearest airport to a user. We then upgraded our server app to draw a map using D3 and Canvas, which we then outputted to the user as a PNG.

Wasn't that all really pretty weird but also kind of fun? Writing server-side code is like that, but it can also be really nicely cathartic after spending a bunch of time doing frontend development, which tends to be really finicky due to having to support so many devices. If nothing else, hopefully you've begun to see how processing on the backend can improve the performance of your projects by offloading some of the processing work from the user's machine.

There's clearly quite a lot more you could learn about this topic that I just simply don't have room to cover here. We didn't go into scalability at all (which is super important when building things for large audiences, such as in the newsroom), nor into how to properly architect a non-trivial application — for that I'd recommend installing a few Express-based Yeoman generators and seeing how they scaffold projects (I'm particularly a fan of `generator-angular-fullstack`), or checking out Alexandru Vlăduțu's *Mastering Web Application Development with Express*, *Packt Publishing*, 2014.

You now have a pretty full toolbox for confronting a very wide array of data visualization challenges. In the next chapter, we'll add two more tools to it — unit testing and strong typing — in order to help you have more confidence in the work you produce.

7
Designing Good Data Visualizations

Data visualization is a tool that can be used in many ways. As you've seen while building examples throughout the book, data visualization is sometimes used to communicate information in a novel or interesting way; sometimes data visualization provides clarity, other times it's just used to make cool things.

Regardless of whether you're a journalist wanting to highlight a change in GDP, a scientist needing to communicate the results of an experiment, or a software engineer looking to integrate visualization into a product, chances are you'll want data visualization that is clear, concise, and does not mislead. Although the examples in this chapter will mainly be from a news media context, many of the points we'll discuss apply in a similar way to data visualization in general.

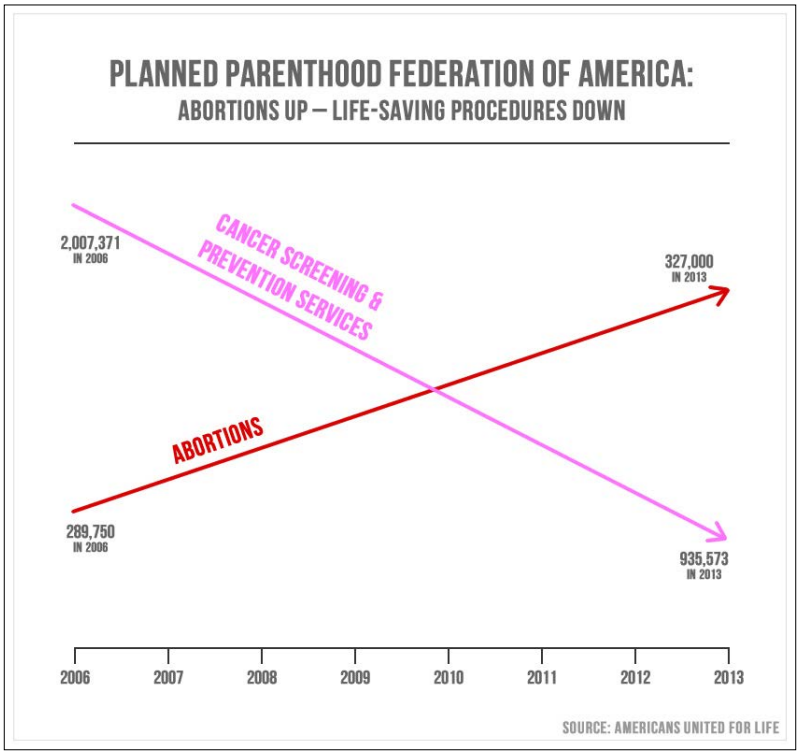
In this chapter, we'll look at a few general principles to keep in mind while building data visualizations, and I'll give some examples of good data visualization as well. Note that I'm in no way a data visualization expert, per se — I'm a developer and a journalist with a degree of learned design experience, and my thoughts on what constitute "good data visualization" are very much influenced by my background in building explanatory data-driven graphics for titles like *The Times*, *The Economist*, and *The Guardian*. These are very fast-paced newsrooms, and the goal when visualizing data is generally to communicate the important bits of a dataset to an audience instead of letting them explore the data. Although you might not have the same demands in your use of D3 as that of a newsroom developer, much of what I'll be discussing also applies if you're using D3 in academia, publishing, or elsewhere — the skill of being able to quickly and succinctly communicate information is incredibly valuable no matter your profession.

With that caveat out of the way, let's get on with discussing what exactly comprises good data visualization.

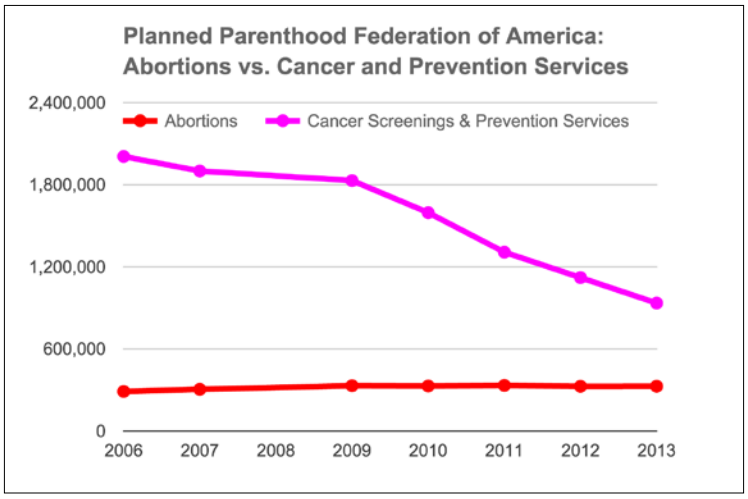
Clarity, honesty, and sense of purpose

There are two big schools of thinking in terms of data visualization at the moment: there's the ultra-minimalist philosophy espoused by Alberto Cairo and Edward Tufte, where the primary goal of data visualization is to reduce confusion, and then there are those who use data to create beautiful things that uphold design over communication. If you couldn't tell by the title of this section, I generally believe the former is far more appropriate in most cases. As somebody wishing to visually communicate data, the absolute worst thing you can do is mislead an audience, whether intentionally or not — not only do you lose credibility with your audience once they discover how they've been misled, but you also increase public skepticism over the ability of data to communicate the truth.

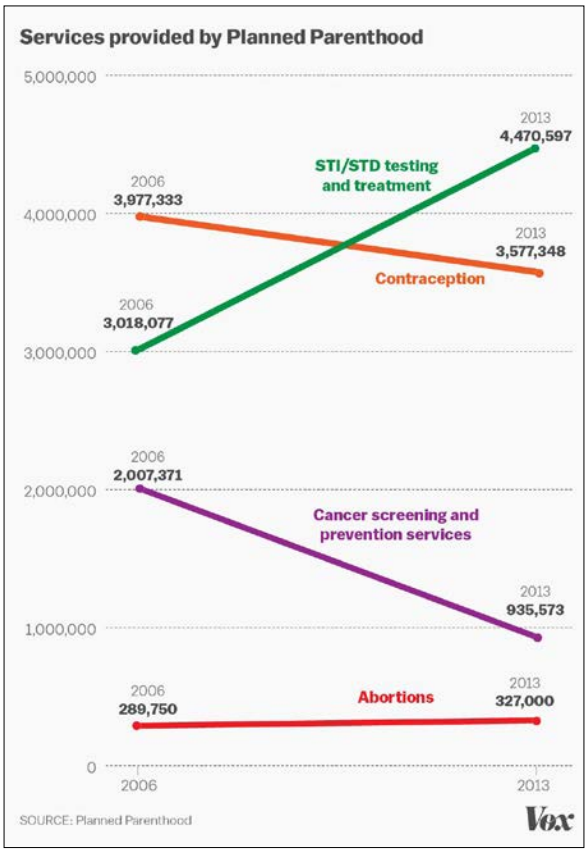
Here's a contemporary example. In September 2015, the U.S. Congress held a hearing on Planned Parenthood, the American reproductive and women's health group. During the hearing, Republican congressman Jason Chaffetz showed the following chart, created by the anti-abortion group Americans United for Life:



There are many problems with this chart, not least the complete lack of *y* axes. PolitiFact redrew the chart with corrected axes and it came out like this:



Vox took it a step further and drew the rest of Planned Parenthood's services:



As you can see, while there has definitely been a decrease in cancer screenings and prevention services (as well as contraceptives, for that matter), and a slight rise in the number of abortions, there has also been a dramatic increase in spending for STI/STD treatment and prevention. As Alberto Cairo commented on the original chart:

> "*That graphic is a damn lie … Regardless of whatever people think of this issue, this distortion is ethically wrong.*"

The public backlash about this one misleading chart led it to being named "2015's Most Misleading Chart" by Quartz. Regardless of what the creators of the chart originally intended to demonstrate with it, any hope of achieving that goal was obliterated once viewers felt they were being misled.

For a more thorough discussion of everything wrong with Chaffetz's chart, I highly recommend the commentary by both PolitiFact and Vox, at `http://www.politifact.com/truth-o-meter/statements/2015/oct/01/jason-chaffetz/chart-shown-planned-parenthood-hearing-misleading-/` and `http://www.vox.com/2015/9/29/9417845/planned-parenthood-terrible-chart` respectively.

In the preceding quote, Cairo makes an interesting point in that communicating data carries with it certain fundamental ethical requirements. This is how *data visualization* differs from *data art* — in the latter, what's ethically required of the *artist* is to purposefully and honestly communicate their emotions, beliefs, fears, and so on. This is a long way off from the ethical requirement of the *visualizer*, which is to communicate specific qualities of the data through visual methods. Taking this a step further, the ethics of data journalism compel the journalist to tell the audience what the data really means and how it relates to that audience.

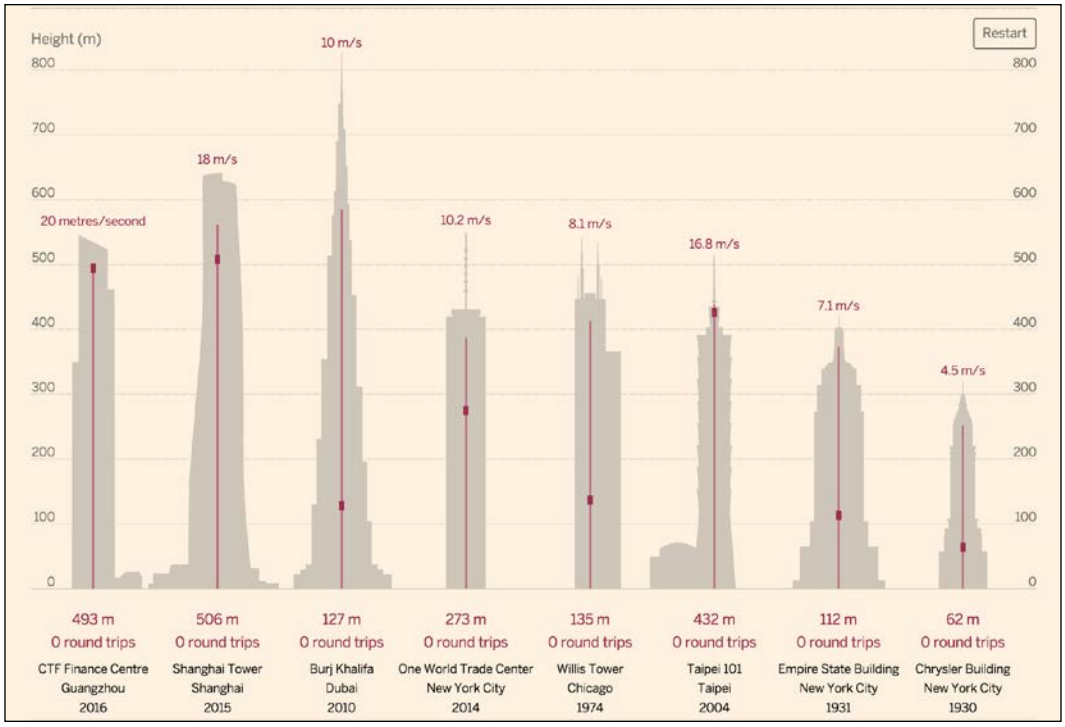
In your projects, decide where your scope lies. Are you acting in the role of a data journalist, with a desire to walk the reader through a specific bunch of numbers and figures? Are you acting as a data visualizer, perhaps creating a dashboard designed to quickly and effectively summarize a very large, multivariate dataset? Or do you want to build something fun and entertaining that leverages data merely as a method by which to achieve that aim? All three of these roles are perfectly acceptable, and there is room for work ranging from incisively explained line-charts all the way through to *objets d'art* that give us a better understanding of our size and place in the universe. But whatever you do, be clear with your intentions and never mislead.

Helping your audience understand scale

A big part of visualizing data is conveying scale and differences in magnitude. The following few examples do this particularly well.

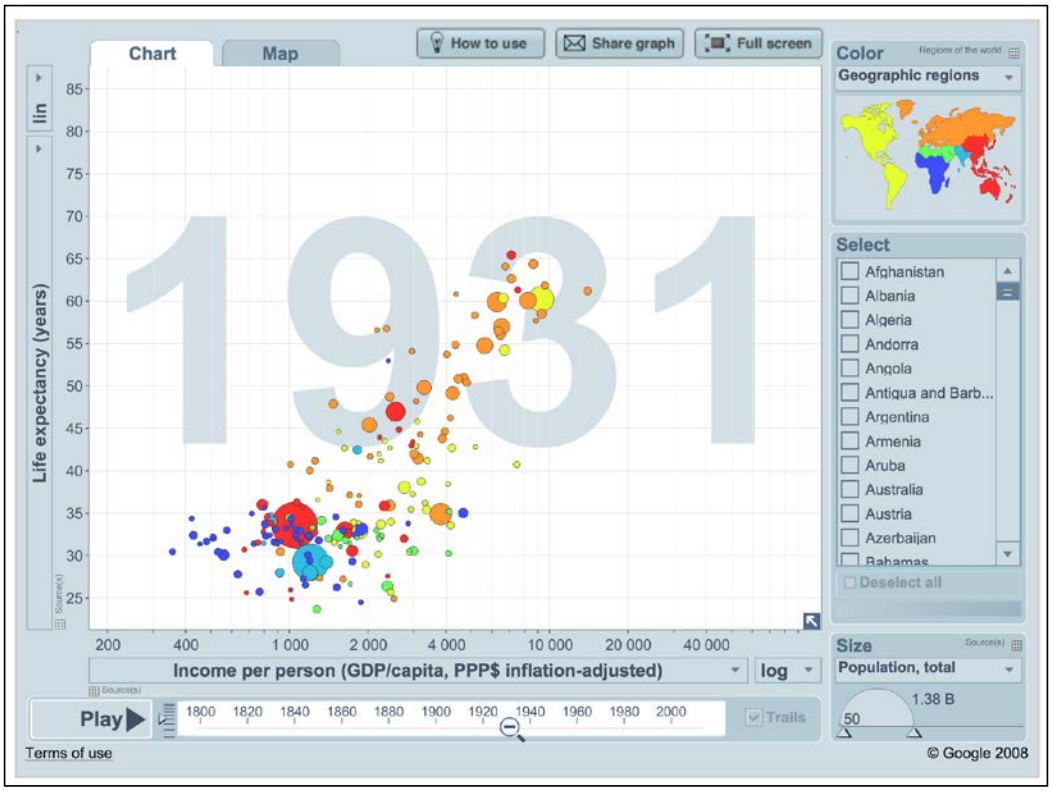
To start with, please view John Burn-Murdoch's graphic on high-speed elevators for the at `http://www.ft.com/cms/s/2/1392ab72-64e2-11e4-ab2d-00144feabdc0.html`.

The following screengrab doesn't really do it justice:



If the above were the live visualization, you would see the elevators in each building endlessly rise and fall, with a counter beneath tracking how many times the elevator has gone up and down while you were looking at the page. A nice bit of easing at the top and bottom makes you feel like the little magenta square traveling along the line is a real elevator, subject to physics in the same way a big metal cage rapidly moving up and down the world's tallest buildings would be. Although this printed version only communicates one dimension — the relative heights of each elevator and building — the interactive version is able to use animation to convey a second quality; that is, the speed of the elevator. In this, one can really see the power of using the digital medium to communicate data in ways a static print version will never be able to. Scale is demonstrated by tying visual content to time; in this case, merely saying "the elevator can do a return trip in 2 minutes" would not be nearly as effective as demonstrating what "2 minutes" feels like to the reader.

Another good use of animation to explain scale is Hans Rosling's Gapminder project (`http://www.gapminder.org/world/`), which allows viewers to see how the world has changed over time. In his excellent TED talk about understanding the world from a data-driven perspective, Rosling discusses how, if looking at measures such as life expectancy and GDP per capita, the world is getting substantially better as time goes on, and that our perceptions of other countries are often rooted in a degree of ignorance as to how similar we generally are.

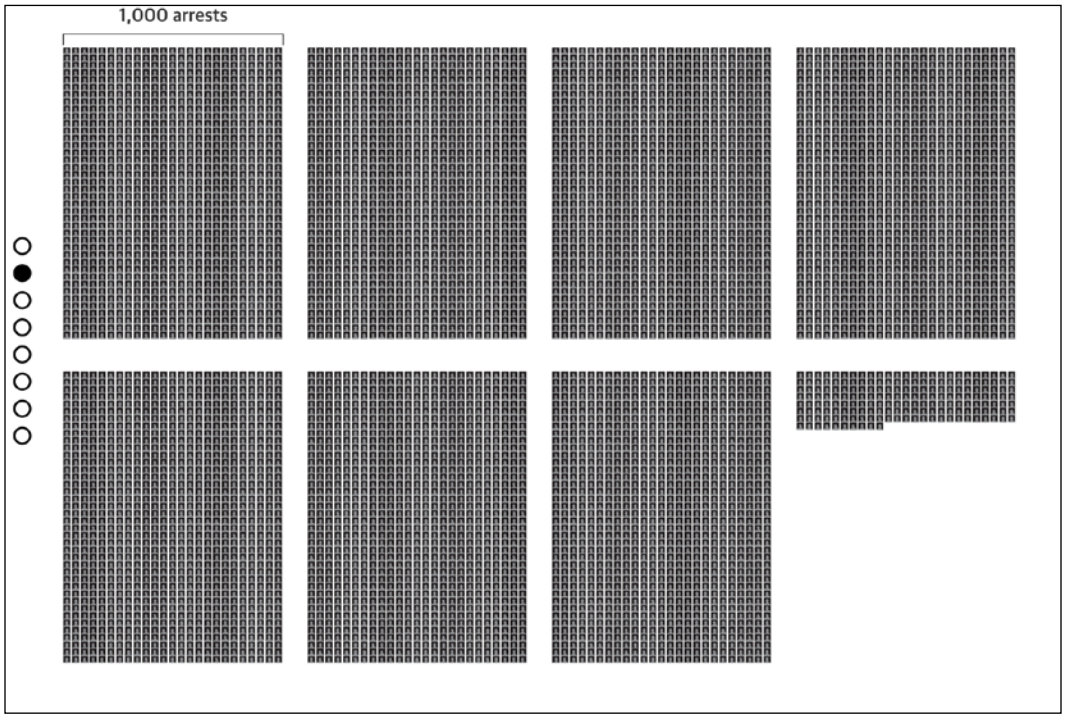


As time goes on, you can watch as the bubbles in the chart migrate from the bottom-left quadrant to the upper-right. It's worth comparing the fairly-dry exploratory visualization to the TED talk (`https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen`). Although depicting similar data, the latter is far more enjoyable simply due to Rosling's narration, with excitement evident in his voice as he explains how the world is improving and changing as societies develop. Good explanatory data visualization doesn't have to be limited to text output.

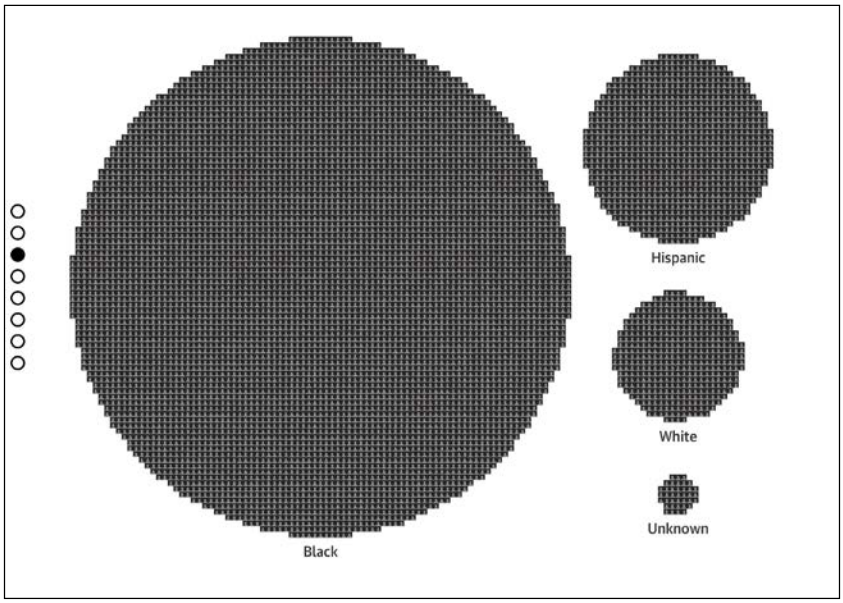
You can see an example of the preceding chart recreated using D3 by visiting Mike Bostock's implementation at `http://bost.ocks.org/mike/nations/`.

A much more elaborate example of embracing the digital medium to help convey scale is The Guardian's *Homan Square: A Portrait of Chicago's Detainees* interactive by Spencer Ackerman, Zach Stafford and the Guardian US interactive team. If you haven't seen it, please visit and prepare to have your mind utterly blown: `http://www.theguardian.com/us-news/ng-interactive/2015/oct/19/homan-square-chicago-police-detainees`.

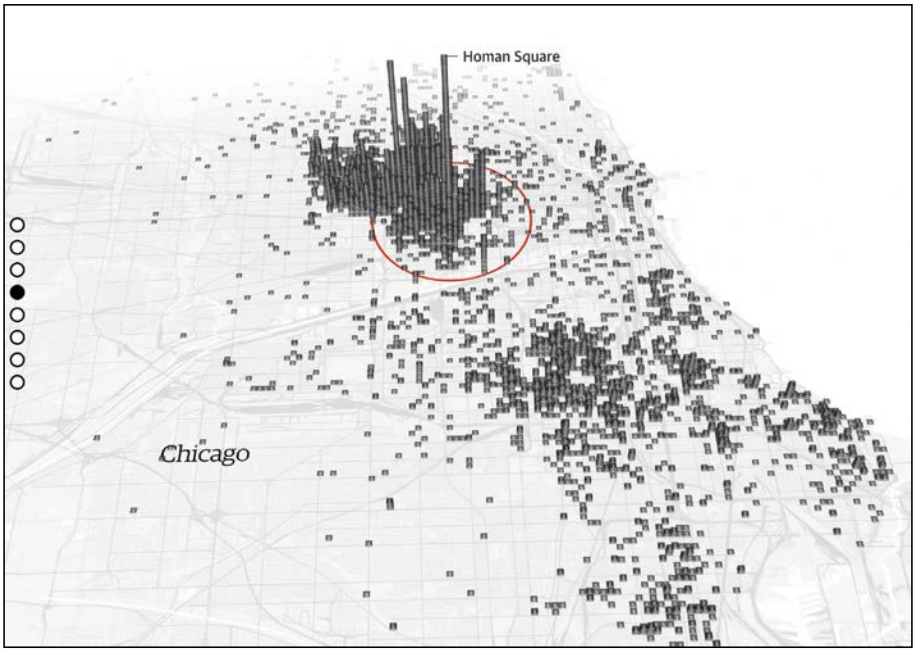
It starts with a big collection of faceless grey silhouettes, each representing a single person in custody at a secretive police warehouse in Chicago:



As you scroll, the faces fly around the screen to make up different configurations, such as this bubble chart:



And even this isomorphic map:

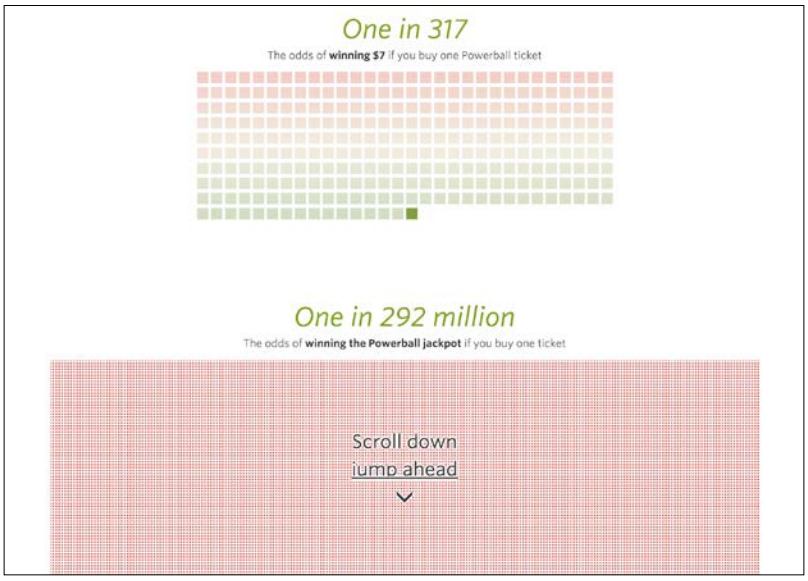


The Guardian's U.S. interactive team did a fantastic Q&A about the process behind this visualization that is well worth reading — `https://source.opennews.org/en-US/articles/how-we-made-homan-square-portrait/`.

Although the data morphs in many different ways, you still feel an attachment to each point, remembering how they were originally displayed — they're not merely pixels on a screen; each portrait represents another person whose life has been impacted (a feeling reinforced when they single out individual portraits as case studies later on in the piece).

These examples are phenomenal for communicating scale to the reader. One of the biggest reasons data visualization is such a powerful tool is that it helps answer the questions "how big is 'big'?" and "how small is 'small'?" As reporting becomes increasingly reliant on data, it becomes very easy to mislead the reader by over- or under-emphasizing scale (indeed, this is a big reason why the chart in the preceding section was considered so dishonest). This can often be mitigated by designing visual output in such a way that the viewer feels some attachment to the visual stimuli onscreen.

As a final example, please visit this graphic by Ana Becker of *The Wall Street Journal* that attempts to visualize what your chances are of winning the Powerball lottery jackpot — it's another one where the following screenshots really won't do it justice. Also, try not to hit the **jump ahead** button for at least a little while, `http://graphics.wsj.com/lottery-odds/`:



After a while of continuous scrolling, the screen starts to look like this:

At this point, you realize there's no chance that you're going to ever physically scroll to the bottom, and so either click the **jump ahead** link or grab the scrollbar. If at that point you're still convinced you can plan your retirement based on your lottery earnings, probably nothing will change your mind.

In a sense, scale is emphasized through comparing the initial "coin-toss" probability to the probability of picking the winning Powerball numbers. Comparing the coin-toss's 1 in 2 with the Powerball's 1 in 292,201,337 is impossible to do in one screen (and especially so in print), because no matter what sort of scale you use, the latter completely dwarfs the former. Making use of the browser's practically unlimited vertical screen real estate is a very effective way of tying visual elements to a physical property (much like the elevator speeds interactive earlier), insomuch that the physical effort involved in scrolling down to just 1 percent of the page (much less, count how many dots that is) very effectively demonstrates how mind-bogglingly big a number like 292m is in the context of comparing probabilities.

Using color effectively

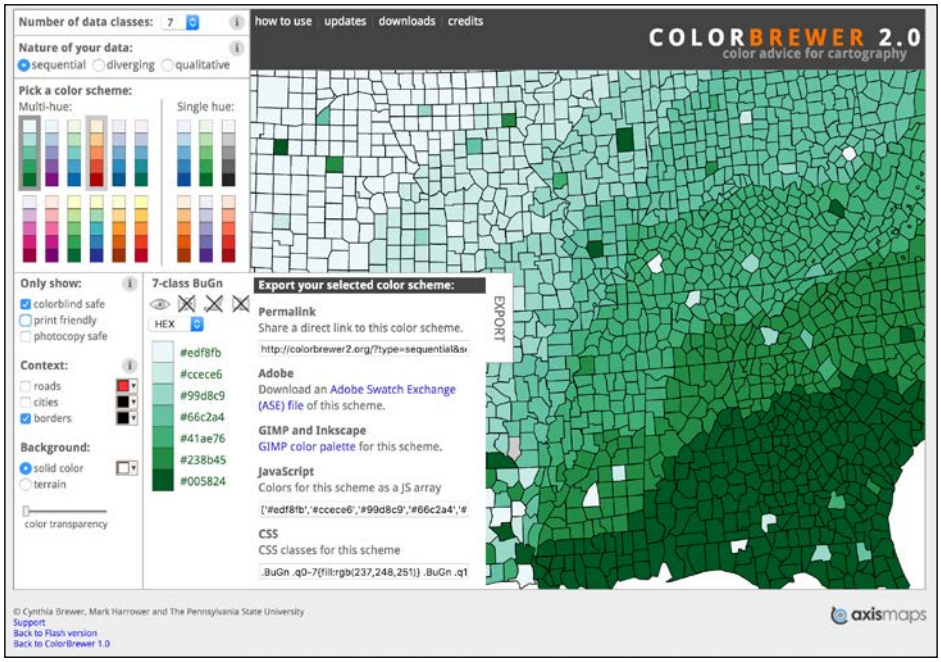
One of the biggest choices you'll make when building data visualizations is choosing what colors to use to represent what. While it's certainly possible to convey a lot of information through purely monochromatic charts, using the wide range of color representable through a digital display can be a very effective way of depicting another property dimension of the data you're visualizing.

If you plan to use color to communicate information, there are a few things to consider. The first is whether a user will be able to discern the pattern necessary to understand what the colors mean. If you have a legend explaining what color corresponds to what, try turning it off and thinking about whether you're still able to understand what the color combination means. Is it intending to show increasing intensity? Diverging values? Or just that it has a particular quality (in which case, do you really need a legend)?

Secondly, pay some attention to people who are colorblind. The most common color combination to use for choropleth maps is generally the "stoplight" color scheme: green for low values, yellow for medium values, and red for high values (or the inverse, depending on whether a high value is a good thing or not). There's a problem with this, though — for people who are red-green colorblind, the highest and lowest values look nearly identical.

Also, while still on the topic of stoplight colors, when discussing sensitive topics like immigration, a fair degree of care should be taken when color-coding anything green or red — if red means "bad" or "severe", is it actually that way? Or is it displaying red merely because the color scale has been constrained to the values in the data set? Colors are much more emotive, and it's easy to unintentionally present an opinion when using them. In general, using a sequential color scheme (scaled to the average of what one would expect that value to be) is a much safer bet.

A very good way of finding a color scheme for a map (and quite a lot else) is using ColorBrewer (`http://colorbrewer2.org`), a tool built by Cynthia Brewer and Mark Harrower at Penn State. It provides a bunch of different pre-built color schemes for representing data, and you can choose the type of relationship you want to depict. Additionally, it helpfully allows filtering out color schemes that aren't colorblind-friendly, or even printer- and photocopier-friendly.



You can use ColorBrewer schemes directly in D3! There's a file in the D3 repo at `lib/colorbrewer/colorbrewer.js`, but at the time of writing it has been removed in releases and isn't really easy for us to use with modular JavaScript. Instead, install it separately via:

```
$ npm install colorbrewer --save
```

And then `require` it as normal. For the above **7-class BuGn** scale (look next to the **EXPORT** pane for the scheme's name), you'd write:

```
let BuGn = require('colorbrewer').BuGn[7];
```

Lastly, listen to natural cues from your data: if you're doing a visualization of political parties, it makes sense to color-code the data at least somewhat similarly to the party colors (as boring as it is to always color-code political data to party colors, the novelty of not doing it this way is easily outweighed by the cognitive dissonance it causes).

Understanding your audience (or "trying not to forget about mobile")

Your audience is one of the most critical things to consider when beginning a new data visualization project. This has two parts: the first is from an editorial perspective (what is the audience's background knowledge of this topic? What types of charts will the audience be able to recognize and properly read? How do these charts work within the broader contexts of this story and other work published?), while the second is technological (what platforms and devices will be used to consume this content?).

It's really important to tentatively sketch out any bespoke data visualization before you start writing code, and this can take many forms. On one hand, it never hurts to figure out the rough shape of your data before committing to a particular visualization style — frequently I get asked for pie charts with a few small outlier values highlighted, which doesn't work (the rest of the chart dwarfs the outliers). You don't necessarily need to get a pencil and paper out for this — pasting your data into Excel and playing with its default charts often helps before committing the data idea to code.

The second way you should sketch out your visualization is on a component or an interaction level. This is where understanding which devices your audience use is important. If you have previous work out there, look at its analytics. To see what percentage of readers use a specific browser in *Google Analytics*, look under **Audience | Technology**. Pay attention also to **Audience | Mobile | Overview**, which will tell you what proportion of your audience is comprised of mobile and desktop. If you have no analytic data to work with (for instance, if you're launching a new project), it's generally a smart bet to assume that half of your audience will be on mobile, and so you should design your project accordingly.

Developing for an array of screen resolutions is called **responsive design**, and there are two major workflows: mobile-first and desktop-first. Traditionally, designs begin at desktop size and are then scaled down for mobile, but this often means that mobile support is an afterthought and it's often problematic to shrink down large elements later on. Mobile-first design starts at the smallest possible size and scales up, which is often easier as you get the more difficult versions of the site designed first and out of the way, and you can let things grow as screen resolution increases. Whether you need to do either depends on your audience — it's quite possible it predominantly uses one platform or the other, reducing the need to accommodate both; again, check your analytics. That said, while ensuring things work well cross-device and cross-browser takes a lot of additional work, it should be a standard of excellence that you strive towards as you create things for broader consumption.

If creating a mobile-first design, take out some paper and a black marker and draw a bunch of phone-sized shapes. Draw squares where you want each user-interface element (buttons, dropdown boxes, radio buttons, and so on) or chart to go — how you distinguish each element from the others is entirely up to you, just make sure that whatever vocabulary you use is shared by those you work with. You don't need to be super artistic with it, just be detailed enough to express how you think each user interaction (clicking/tapping, swiping, dragging, and so on) should *feel* within the project. Sketch out roughly how you think each element should *flow* on the different devices. Then do the same for a larger desktop screen size.

A rough drawing versus the finished page. Prototypes are meant to be discussion pieces you can use to solicit feedback from colleagues and refine throughout the development process. Don't worry if the end result looks nothing like your original sketches, or if (like me) you can't draw a straight line to save your life.

If possible, run your pen-and-paper prototypes past a few people to see whether your user interactions feel natural.

Some principles for designing for mobile and desktop

Mobile and desktop computing differ in some key ways, and understanding these is crucial for building data visualizations that are effective on both platforms.

Mobile:

- Has multiple screen orientations.
- Has uniformly-small screen dimensions.
- Does not have a pointer; interaction is derived from touch gestures. There is no "hover" state.
- Relies on often inconsistent data availability.

- Has significantly less computing power than comparable laptops.
- Keyboard is visible on-screen when in use; it is not used for navigation.

Desktop:

- Generally has only one screen orientation.
- Has variably-large screen dimensions (and is often connected to huge screens).
- Uses a pointer (and keyboard) to interact. The "hover" state is usable.
- Generally has reliable data availability
- Has significantly more computational power than comparable phones and tablets.
- Keyboard can be used without impacting what's visible on the screen; additionally, can be used for navigation.

I really don't have the space to go into a full course about how to do responsive design right here, but a few basic principles to keep in mind will get you started.

Columns are for desktops, rows are for mobile

On some level, it's fairly safe to assume that most people will view your work in portrait mode on mobile. This means that you effectively have one column, and every element in your one column is in a full row extending perpendicularly to the column's direction. Paragraphs should always flow vertically in a single column (instead of in multiple columns positioned left to right), and it's not an awful idea to rotate bar charts 90^{o} so that your bars aren't squished together in one narrow horizontal row. Much like the *WSJ* lottery probabilities graphic above, make sure use of the infinite vertical space you have to scroll.

On desktop, however, a single paragraph spanning the entire horizontal width of the display is very hard to read, as it requires the eye to move back and forth quite a lot. Using columns is often preferable, as it not only makes better use of the horizontal space, but it also improves readability. Form elements in particular often look much better when grouped into columns.

The good news is that Flexbox makes it really easy to switch the orientation of groups of elements, provided you don't have to support older versions of Internet Explorer. If you aren't using flexbox already, take the effort to do so — it will make your life so much easier (at least, once you figure out how to use it).

Want to learn flexbox the fun way? Try out Flexbox Froggy, a game that teaches you how to position elements using Flexbox and Frogs. `http://flexboxfroggy.com/`.

Still find flexbox hideously frustrating? You're in good company. Try out Flexbox Grid, which uses Twitter Bootstrap-like classes to create responsive grids using flexbox. `http://flexboxgrid.com/`.

Be sparing with animations on mobile

Animation on mobile is really tricky, not only because you have reduced graphics processing power, but also because scroll events have traditionally messed with JavaScript execution timing. If you don't totally disable animation on mobile, try to only use CSS transitions, as these are more performant than iterating through properties via JavaScript. When in doubt, disable animation on mobile.

Realize similar UI elements react differently between platforms

Things like radio buttons, sliders, and checkboxes are available both on mobile and desktop, but some are easier to use on mobile than others. In general, web browsers draw all of these elements slightly too small for comfort on most mobile devices. Where possible (for instance, select dropdowns), make individual form elements stretch the entire device width on mobile, or in the case of things such as checkboxes and radio buttons, use the `for` property of the `<label>` element to make labels tap-able and scale these horizontally.

Avoid "mystery meat" navigation

Pointer-based devices have the benefit of being able to use the cursor's "hover" state to reveal information (for instance, labels on buttons). While this can allow for more minimalist-looking interfaces on desktop, it's a really terrible anti-pattern referred to as "mystery meat navigation" when on mobile. When you hover over a button, you're not committing to clicking it. However, because mobile devices lack a hover state, users must commit to a user interaction to understand what a button actually does. Given how slow the mobile reading experience can be, users tend to be a lot more cautious before committing to any action that might cause the page to reload.

The solution is simple: unless using incredibly clear iconography, label your buttons on mobile.

For some good mobile icons, try Font Awesome (`https://fortawesome.github.io/Font-Awesome/`) and Google's Material Icons (`https://design.google.com/icons`).

Be wary of the scroll

A common user interaction idiom on desktop (popularized by the *New York Times*' "Snowfall" long-form piece) is to tie animations to the browser's scroll event. This is frequently fraught with peril on mobile, because scrolling triggers a memory-intensive redraw that often blocks JavaScript execution. While newer mobile operating systems handle this better than before (I'm looking mainly at you, iOS), it's often unsafe to assume that a scroll-dependent animation will play properly and not feel non-performant and "janky" on mobile. Again, this has improved quite a lot with newer devices, but it's still never a bad idea to tie animation states to tap events (possibly delivered via a button) on mobile.

Summary

Hopefully by now, you feel like you have a secure understanding of the work you want to do using D3 to visualize data. We went through some examples and laid some good ground rules for building high-quality data visualizations that not only inform an audience, but also look pretty spectacular while doing so. We also discussed how to make sure your work functions well on mobile.

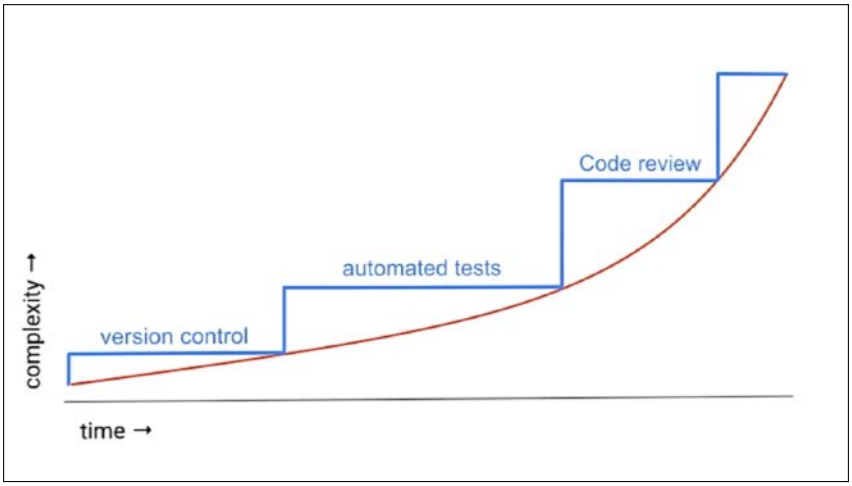
In the next chapter, we'll tie things up by helping you feel more confident with the visualizations you produce by using type checking and automated testing to ensure that nothing goes wonky at the worst possible time. Stay tuned…

8
Having Confidence in Your Visualizations

When you're building things for as big an audience as the Web provides, a very real fear is that a software glitch prevents data from displaying correctly. When developing projects on a tight timeline, testing is one aspect that often gets neglected as the deadline gets closer and closer, and often things need to be viewed by other people even earlier (editors, managers, and other people further on down the line — possibly even lawyers) in turn emphasizing output over process. Let me be clear—if you want to ensure that your visualizations are of a high quality, you need to take steps to make sure that they are well tested and functioning properly. On some level, doing this is an exercise in managing complexity.

The next chart depicts project complexity over time. As you can see, complexity increases somewhat exponentially as time passes by. Adding more team members, more lines of code, and/or more dependencies increases the project's complexity dramatically. Meanwhile, the step chart depicts how tooling processes improve in response to complexity. Although it's possible to implement all of these at the beginning of every project, it's often better to do so incrementally in response to the project's demands. For instance, generally everyone will start with version control because it helps in collaboration, can revert mistakes, and provides a project with history (plus it's really easy to set up). Then, say you add a few more team members, or make a project open source. Now you have code flying at you from all directions. Having a way to automatically test whether that code will break anything or not starts to become incredibly useful.

A bit later, imagine a lot of bad code still getting past the automated testing; having manual code reviews might help this further. Each of these improvements to the process take time to both implement and use, and whether they'll benefit your project or not is hugely dependent on how big your team is and how well you trust every member of it (or, if working alone, how well you trust yourself not to introduce errors).



As time passes by, complexity increases somewhat exponentially, but tooling increases in a step pattern; source: Martin Probst and Alex Eagle, https://youtu.be/yy4c0hzNXKw?t=245

The preceding chart is from Martin Probst and Alex Eagle's talk on TypeScript at AngularConnect 2015. They touch upon a lot of the same topics as this chapter and it's worth watching (`https://www.youtube.com/watch?v=yy4c0hzNXKw`):

- In this chapter, we'll focus on a few technologies that help manage project complexity. We start by talking about linting. **Linting** is when you run your code against preset rules to ensure that it conforms to a set of standards.
- Then we will move on to **static** type checking. This is a tool for ensuring that the data flowing through your application doesn't act unexpectedly.
- We will end this chapter with **automated** testing. This is what it says on the tin: you write tests for your code that must all pass. Failing tests can be used to diagnose bugs or other issues with the code.

You honestly don't need to do any of these things to make stuff with D3, but getting into the habit of using these tools when necessary will both improve the quality of your code and make you a better developer. Let's dive in!

Linting all the things

A **linter** is a piece of software that runs source code past a set of rules and then causes a stink if your code breaks any of those rules. Now, I know what you're thinking: "my boss/manager/editor/significant other is *already* giving me more feedback than I'd care for. Why do I need *yet another* thing to do that?" Glad you asked!

Linting rules are often based on industry best practices, and most open source projects have a customized rule set corresponding to their community guidelines. This simultaneously ensures that code looks consistent even when delivered by a multitude of people and lets contributors know when they're doing something that is a little confusing or error prone in their code. Note, however, that all of these rules are just opinions—you don't *have* to write your code following them, but it tends to help everyone else out if you do.

If you've been following along with the GitHub repository for this book, then, perhaps, you've noticed a hidden file called `.eslintrc`, or noticed `eslint` in `package.json`. ESLint is currently the best linter for ES62016 code, and it works very similarly to its predecessors, JSHint and JSCS. It is configured via the `.eslintrc` file, which contains both the specific rules to be checked against (or a set of defaults to extend, such as what we've used) and information about the code's environment (for instance, NodeJS has different global variables than the browser).

To run ESLint against the current project, type this line:

```
$ npm run lint
```

Although there won't be any linting errors in the repository by the time this book goes to print, here's an example of what the output looked like at the writing stage of the book:

```
> learning-d3@1.0.0 lint /Users/aendrew/Sites/learning-d3
> ./node_modules/eslint/bin/eslint.js src/*.js


/Users/aendrew/Sites/learning-d3/src/chapter2.js
  44:12  error  "d" is defined but never used  no-unused-vars

/Users/aendrew/Sites/learning-d3/src/chapter3.js
  139:11  error  "lines" is defined but never used  no-unused-vars
  379:9   error  Unexpected constant condition      no-constant-condition
  387:9   error  Unexpected constant condition      no-constant-condition

/Users/aendrew/Sites/learning-d3/src/chapter4.js
   80:9   error  "timer" is defined but never used  no-unused-vars
  178:30  error  "rej" is defined but never used    no-unused-vars
  185:30  error  "rej" is defined but never used    no-unused-vars

/Users/aendrew/Sites/learning-d3/src/chapter5.js
  227:53  error  Irregular whitespace not allowed  no-irregular-whitespace
  338:9   error  "link" is defined but never used  no-unused-vars
  354:33  error  "d" is defined but never used     no-unused-vars
  357:32  error  "d" is defined but never used     no-unused-vars

/Users/aendrew/Sites/learning-d3/src/voronoi-airports.js
   3:25  error  "location" is defined but never used  no-unused-vars
  13:7   error  "path" is defined but never used      no-unused-vars

✖ 13 problems (13 errors, 0 warnings)
```

Tsk-tsk! What a lot of errors we have here!

We use npm to run ESLint in this instance, which is effectively an alias for the following line:

```
$ node ./node_modules/eslint/bin/eslint.js src/*.js
```

You can also install ESLint globally and then just run it anywhere.

Although this is helpful, linting is way more useful when you see it all the time while developing. Let's make ESLint scream at us while we're using `webpack-dev-server`. First, install `eslint-loader`:

```
$ npm install eslint-loader --save-dev
```

Next, we add `eslint-loader` as a preloader to our Webpack config, so it runs on our code *before* Babel does its thing with it. In `webpack.config.js`, add the following code in the module section of each build item:

```
preLoaders: [
  {test: /\.js$/, loader: "eslint-loader",
   exclude: /node_modules/}
]
```

Then run `webpack-dev-server`:

```
$ npm start
```

Ta-da! Now, you can get instant feedback about how messy all of your code is every time you hit save! You're utterly thrilled by this. I can feel it!

Linting is a very light way of managing complexity. Generally, a linting failure won't necessarily cause anything other than a message to appear on the developer's screen. It's up to you and your team to maintain a sense of discipline in terms of not committing code until it passes linting.

Static type checking with TypeScript and Flow

Static type checking is where you have a process that looks at how variables are being used, and then throws a wobbly if you do something weird. By this, I mean that it looks at the *type* of each variable and uses *type annotations* (bits of text defining what type a variable is when the variable itself is defined) to ensure that functions don't mutate a variable in an unexpected way. This is called **static typing**, and it is a feature built into many robust languages, such as C++ and Java. While JavaScript's dynamic typing (also shared by lots of other web languages, such as PHP and Ruby) is helpful in some ways and enables a certain style of programming, it can also be incredibly frustrating due to its ability to introduce silent errors. Because we're using a transpiler to transform our JavaScript anyway (throughout the book, this has been Babel, though that doesn't necessarily have to be the case), we can introduce static type checking to JavaScript at the same time if we so want.

There are two big players in statically typed JavaScript right now: Facebook's Flow project and Microsoft's TypeScript. Which you use is somewhat dependent upon when and how you plan to employ type checking, though my personal favorite is TypeScript for reasons that I'll get into later.

Although it's a level of tooling complexity above both version control and linting, in all honesty, to get the most from any of those three tools, it really helps to use them from the outset and be really disciplined with their use (particularly if you have two or more developers on your project). Flow is nice in that it is much easier to work into a Babel-based workflow than TypeScript, but TypeScript has a ton of benefits that extend beyond just ensuring that types don't change, and is particularly well-suited for working with D3.

Warning!

Lots of boring details about configuring stuff ahead! Wait, though! I have a solution!

The following sections will describe how to get started with either TypeScript or Flow, and give a bit of light config info, so you can get a sense of how it all fits together. Alas! Configuring things is really dull and should be avoided. If you want to dive right into playing with this stuff, you have two options. You can simply check out the `chapter8` branch of the book repository by doing the following:

```
$ git stash save && git checkout origin/chapter8
```

Alternatively, you can install my handy `strong-d3` Yeoman generator, which will quickly scaffold out a brand new D3 project using either Flow or TypeScript. To install it via npm, run this line:

```
$ npm install --global generator-strong-d3 yo
```

Create a new directory for your project, switch into it, and run the generator, replacing `myProject` with whatever you want to call what you're working on:

```
$ mkdir myProject && cd myProject && yo strong-d3
myProject
```

Answer the questions it asks and you're on your way! For full usage instructions, visit `github.com/aendrew/generator-strong-d3`.

The new kid on the block – Facebook Flow

If you have an existing ES2016 project and you want to introduce type checking into it retroactively, Flow might be the ticket. Unlike TypeScript, which totally replaces Babel, Flow lets you selectively add type checking to your projects. In many cases, this is all you'll need.

We're not actually going to use Facebook's normal Flow implementation (which starts up a whole server and is way too heavy for our uses); instead, we're going to use a Babel plugin that uses Flow-style syntax, `babel-plugin-typecheck`. Install that first:

```
$ npm install --save-dev babel-plugin-typecheck@2
```

Next, change your `.babelrc` file to the following:

```
{
  plugins: ["typecheck"]
}
```

Let's test this out. Start up `webpack-dev-server` with this command:

```
$ npm start
```

Then open `index.js`. Clear it out and add the following code:

```
function giveMeANumber(base: number): number {
  return base * Math.random();
}
alert(giveMeANumber('hi!'));
```

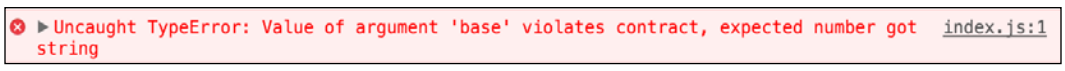
What we do here is create a function that multiplies a random number by whatever number the function is supplied. As you can see, the function declaration looks a bit different:

```
function giveMeANumber(base: number): number {
```

What we're doing here is saying that the argument base should be a number and the function should always return a number. We then try to completely disregard what we've just said it should be and try to supply the function with a string:

```
alert(giveMeANumber('hi!'));
```

If you open this up in your web browser, your console should let you know you've messed up:



Although this is a truly trivial example, imagine you had a massive project with tons of code, and this was your first day as a new developer on the project. Instead of possibly introducing a bug while learning the ins and outs of the code, Flow will smack you down straightaway, letting you know that you're using the code wrong. Plus, because the type definitions are right in the code, you can go straight to that function, look at the type annotation, and realize that you need to give it a number instead of a string. Combining it with a modern editor that lets you do this in a keystroke results in a very nice development experience.

TypeScript – the current heavyweight champion

That development experience is made even better with TypeScript, which is somewhat more stable and more widely used than Flow (at least at the time of writing this book). There are many upsides to using TypeScript, which I'll get into in just a moment. But before that, be forewarned that there's slightly more effort involved with getting TypeScript set up. This is because it's a transpiler in its own right (instead of just a type checker), which means we can't use Babel anymore.

The good news, however, is that TypeScript compiles modern JavaScript very similarly to how Babel does, and we just need to tweak a few things before we can begin using it. In practice, unless you're using a lot of Babel plugins, you probably won't notice the difference between the two. Further, we're actually going to use both simultaneously, importing TypeScript modules into Babel and vice versa like it ain't no *thang*.

First, let's install some more stuff. Modern JavaScript development is like 50 percent coding, 20 percent being confused about what libraries to use, and 30 percent installing those libraries. So let's get to it:

```
$ npm install typescript@1.8 ts-loader --save-dev
```

This will install both the TypeScript transpiler and the Webpack loader. Next, we're going to install a utility called `typings`, which helps us download prewritten type definitions for libraries such as D3:

```
$ npm install --g typings
```

Next, generate a `typings.json` file:

```
$ typings init
```

Then install the D3 TypeScript definition:

```
$ typings install d3 --save
```

Next, let's update our Webpack config. Under `loaders`, add the following:

```
{
  test: /\.ts$/,
  loader: 'ts-loader'
}
```

This will use TypeScript to load all files ending in `.ts`. You can go back through all your old files and rename them to `.ts`, or you can just do as we are doing here and use *both* Babel and TypeScript. Once you get used to TypeScript, you'll probably want to start with it from the beginning, but this time around, we're going to mash up both ES6 and TypeScript for great awesomeness.

First, let's add some lines to `index.js` to import our forthcoming TypeScript class and instantiate it with some data:

```
import {TypeScriptChart} from './chapter8.ts';
let data = require('./data/chapter1.json');
new TypeScriptChart(data);
```

Importing a TypeScript file into Babel is now as difficult as specifying the `.ts` extension.

Next, create a new file in `src/` called `chapter8.ts` and add this code at the top:

```
/// <reference path="../typings/main.d.ts" />
import * as d3 from 'd3';
```

This both includes your TypeScript definitions and imports D3 from `node_modules`. From here, we can start using TypeScript as we would normally.

One worthwhile thing to do at this point is to create a `tsconfig.json` file, which is a TypeScript config file. With this, you can prevent a number of annoying behaviors from the TypeScript transpiler, and some TypeScript IDE integrations make extensive use of it (particularly `atom-typescript`). For an example `tsconfig.json` file, look at the `chapter8` branch of the book's repository.

Let's create a brand new chart, using our old friend `BasicChart` as a base. "But wait a minute, that's not in TypeScript?" you ask. Eh, give it a shot anyway!

```
import {BasicChart} from './'basic-chart';
```

Run Webpack:

```
$ webpack
```

And you'll get the following error:

```
ERROR in ./src/chapter8.ts
(2,26): error TS2307: Cannot find module './basic-chart'.
```

Eh, was worth a try! How do we solve this?

Create a new file in `src/` named `basic-chart.d.ts`. We are going to write a really simple, ambient type definition for our `BasicChart` class, which will allow us to add type checking to our existing class code without changing it or converting it to TypeScript. This is similar to how we consume D3 with TypeScript—we use the `Typings` utility to install D3's TypeScript definition, which gives us all the advantages of TypeScript with D3, all without any extra effort on Mr. Bostock's behalf (the definition is maintained separately by the fantastic folks at a project called `DefinitelyTyped`).

Here it is in one go:

```
import * as d3 from 'd3';
export declare class BasicChart {
  constructor(data?: any);
  data: Array<any>|Object;
  svg: d3.Selection<SVGElement>;
  chart: d3.Selection<SVGGElement>;
  width: number;
  height: number;
  margin: {
    left: number;
    top: number;
    right: number;
    bottom: number;
  }
}
```

This probably looks a bit different from how we've been doing things, so bear with me.

First off, we import everything from the `d3` module and call it `d3`. This is how you import D3 3.5.x as an ES2016 module, by the way. Although it didn't matter whether we used the shorter `require()` when our code was transpiled with Babel earlier in the book, in TypeScript you really want to use the ES6 module syntax.

Next, we declare the `BasicChart` class and export it. All we do then is list its properties (and methods if it had any), noting what types each property should take. All the numerical properties at the end of it are pretty self-explanatory. Let's take a quick look at the first four:

```
constructor(data?: any);
data: any;
```

We put our constructor and the arguments it takes. It has one optional argument (specified by the question mark), and we're not really prescriptive in terms of what it should be, so we use the `any` type. Because we then directly assign the argument to the class' `data` property, I've set that as `any`.

If you set `compilerOptions.noImplicitAny` in your `tsconfig.json` file to `false`, any variable without a type definition will be given the `any` type. If you're having trouble getting your code to work with types initially and just want to add types to existing code on an ad hoc basis, setting `noImplicitAny` to `false` might be something worth trying.

If we did some transformation on the data first—changing it to, say, an array of numbers—we can write something like this:

```
data: Array<number>
```

We supply the number argument to the `Array` interface, which is what we're doing with these weirdo angled brackets.

Similarly, we define the `svg` and `chart` properties using the D3 selection interface, containing `SVGElements`:

```
svg: d3.Selection<SVGElement>;
chart: d3.Selection<SVGGElement>;
```

Where on earth did I get `d3.Select<SVGElement>` from? Although we could have just specified these as the `any` type as an easy way out, TypeScript is more powerful when you give variables as specific a type annotation as you possibly can. For instance, the D3 definition declares several interfaces corresponding to the various types of variables created by D3, `Selection` being only one.

Throughout this book, I've attempted to be very neutral on the topic of editing environments and IDEs, not caring whether you use **Atom**, **Sublime Text**, **vi**, or something awful like Notepad. However, you *really really* should use an IDE if you're going to work with TypeScript. I use Atom (with the `atom-typescript` plugin). My environment tells me what interfaces the D3 module has while I'm typing, and I can just browse through them. Not only that, the autocomplete will tell me what arguments to specify for the functions I'm using, because a TypeScript definition effectively acts as machine-readable API documentation for the editor. It is super helpful!

Save and run Webpack again. It should compile without issues. Go back to `chapter8.js`, and we'll start filling out our new chart class. Let's kick it old school and redo the first chart that we did in this book as TypeScript.

```
import * as d3 from 'd3';
import {BasicChart} from './basic-chart';
export class TypeScriptChart extends BasicChart{
  constructor(data: Array<ITypeScriptChartDatum>) {
    super(data);
  }
}
```

Okay, this looks familiar. But what's that `ITypeScriptChartDatum` thing? Things that start with a capital `I` in TypeScript are generally interfaces, or reusable sets of typings. As you can see, our constructor takes an array of these. Let's create that interface now. Put this code at the bottom of the file, outside of your class:

```
interface ITypeScriptChartDatum {
  population: Array<IPopulation>;
  name: string;
}
```

If you remember, our data is comprised of an array of objects containing a name string and an array of population measurements. We don't need to create a new interface for the population array items, but it makes things a bit more readable if we do. Add the following code afterwards:

```
interface IPopulation {
  module_name: Array<any>;
  module_type: string;
  value: string;
  demography: {
    "04M": string;
    "04F": string;
    "511M": string;
    "511F": string;
    "1217M": string;
    "1217F": string;
    "1859M": string;
    "1859F": string;
    "60M": string;
    "60F": string;
  }
}
```

We go back to our constructor, and let's start fleshing out our class a bit more. Let's define our class' `data` property as an array of objects that are strings and numbers. Below the constructor, add this line:

```
data: Array<{name: string; population: number}>
```

It's good to do this because otherwise TypeScript will see the `any` type we used in our parent class and not know how the data is structured (as a result, it will need a bunch of type annotations in things such as D3 data accessors, which gets annoying).

Let's try to directly assign our constructor argument to `this.data` inside our constructor, as follows:

```
this.data = data;
```

If you run the TypeScript compiler now, you'll get the following error:

```
    ERROR in ./src/chapter8.ts
    (15,7): error TS2322: Type 'ITypeScriptChartDatum[]' is not
assignable to type '{ name: string; population: number; }[]'.
      Type 'ITypeScriptChartDatum' is not assignable to type '{
name: string; population: number; }'.
        Types of property 'population' are incompatible.
          Type 'IPopulation[]' is not assignable to type 'number'.
```

Note that TypeScript, even if it's screaming at you for whatever reason, will still compile the JS files, which is why sometimes it works even if the compiler throws errors.

But not today! Let's munge that data like it's going out of style! Replace that last line with:

```
this.data = data.filter((obj) => obj.population.length > 0)
  .map((obj) => {
    return {
      name:  obj.name,
      population: Number(obj.population[0].value)
    };
  });
```

No more errors! Alright! Let's set some happy little scales inside the constructor:

```
this.x = d3.scale.ordinal().rangeRoundBands(
  [this.margin.left, this.width - this.margin.right], 0.1);
this.y = d3.scale.linear().range(
  [this.height, this.margin.bottom]);
this.x.domain(this.data.map((d) => d.name ));
this.y.domain([0, d3.max(this.data, (d) => d.population)]);
```

Nothing new here, except that now we're throwing a TypeScript error, saying that the `x` and `y` properties do not exist. What?!

What we need to do here is define the types of these class properties. Just as with `data`, we need to tell TypeScript what form these should take. Add the following two lines beneath the constructor but still inside the class:

```
x: d3.scale.Ordinal<string, number>;
y: d3.scale.Linear<number, number>;
```

What we do here is use the `Ordinal` and `Linear` interfaces for D3 to let TypeScript know what kinds of scales these are supposed to be. Because our *x* axis is ultimately going to be a bunch of place names mapped to a location on the screen, we provide `string` as the domain argument and `number` as the range argument (the argument signature for the `Linear` interface helpfully provided by my IDE's autocompletion). Meanwhile, the *y* axis is a linear scale that maps numbers to numbers. Cool, no more errors!

Let's add the axes:

```
let xAxis = d3.svg.axis().scale(this.x).orient('bottom');
let yAxis = d3.svg.axis().scale(this.y).orient('left');

this.chart.append('g')
    .attr('class', 'axis')
    .attr('transform', `translate(0, ${this.height})`)
    .call(xAxis);

this.chart.append('g')
    .attr('class', 'axis')
    .attr('transform', `translate(${this.margin.left}, 0)`)
    .call(yAxis);
```

Nothing surprising here at all! Finally, let's draw the bars:

```
this.bars = this.chart.selectAll('rect')
    .data(this.data)
    .enter()
    .append('rect')
    .attr('class', 'bar')
    .attr('x', (d) => this.x(d.name))
    .attr('width', this.x.rangeBand())
    .attr('y', () => this.y(this.margin.bottom))
    .attr('height', 0);

this.bars.transition()
```

```
            .delay((d, i) => i*200)
            .duration(800)
            .attr('y', (d) => this.y(d.population))
            .attr('height', (d) => this.height -
    this.y(d.population));
```

Because we're saving the reference to our bars selection as a class property, we need to define what it is. Beneath the two scale type annotations, add the following code:

```
bars: d3.Selection<{name: string; population: number}>;
```

Woo! We've built our first strictly typed D3 chart using TypeScript!

This was a ludicrously shallow overview of TypeScript that barely scratched the surface, but hopefully, you're using a good editor and are already feeling how powerful it can be. To be able to use TypeScript effectively, you should know at least how to typecast each variable as well as create things such as interfaces. If TypeScript interests you as a technology, I highly recommend the handbook at `http://www.typescriptlang.org/Handbook`.

Behavior-driven development with Karma and Mocha Chai

All of these will take you pretty far towards having more confidence in your visualizations, but another step you can take to be even more of a rock star is adding automated testing to your projects.

There are many reasons to write automated tests. If you have a product that needs to render charts reliably and the chart rendering is merely a part of a much larger application, you will likely want to use automated testing to ensure that changes to the application don't break your charts. Likewise, if you've created an open source project that receives a lot of pull requests from various people who use your library, you might want tests to ensure that none of this outside code causes regressive bugs. Beyond this, automated tests are great if you want to be able to show your editor proof that your chart is working and is accurate, or if you merely want to gain more confidence in your data visualization work.

There are fundamentally two ways by which you can approach testing—you can build your project and add testing after the fact, possibly needing to refactor parts so that it's more easily testable; or you can write your tests at the very beginning of your project, before any code, and *then* build your project, ensuring that it passes the tests you've created at each step of the way.

The latter approach is called **test-driven development** (**TDD**), and it should be seen as the hallmark of having reached some degree of skill with JavaScript. An extension of it is called **behavior-driven development** (**BDD**), which tends to be more focused on user interface interactions. BDD tests tend to be less brittle, as they focus more on how a feature is functioning than how it works.

I personally find the syntax of BDD-style testing frameworks much easier to read and write, which is what I'll use here, via the Mocha Chai library.

There are many types of automated tests you can do, but we're going to mainly focus on unit and functional testing.

Unit testing is when you test each of your project's functions in an isolated fashion, which requires you to write your code in such a way that side effects are minimized. If you remember from *Chapter 3*, *Making Data Useful*, one aim of functional programming is to not have your functions produce side effects, and unit tests are a way of both ensuring and verifying that this is in fact the case. TDD focuses on writing unit tests for each part of your application as part of the initial development process—you're simultaneously quality-assuring your code as you write it. One upshot of this is you can be less reliant on ad hoc testing methods; that is, instead of switching between the web browser and your code editor every time you hit *save* to see whether something worked or not, you can switch to your command line's test runner, which will explicitly tell you whether that thing worked or not. This can often be much faster, which helps offset the time spent on writing the tests before anything else.

Functional testing, on the other hand, is more comparable to looking at how the application behaves in a consumer context. Imagine you buy a new phone. The phone has had each of its components tested at a very high level at the factory, and that whole process is opaque to you; you assume that it passed all tests because it made it out of the factory. However, you still test it in your own ways to make sure you like it: how does it feel in your hand? Is it light and flimsy-feeling or do the materials used make it feel like a premium product? Is the touch interface responsive? How do the buttons feel? Is the screen bright enough? How long does the battery last?

BDD is more geared towards testing the latter, and it's particularly helpful with data visualizations. In some ways, D3 does some of the unit testing work for you. Because your project is using a release of D3 that passes all of its tests (that is, "the phone has left the factory"), you don't need to worry so much about D3 doing anything wrong. Rather, your bigger concerns are preventing silent errors due to dynamic typing (that is, concatenating the numbers `"1"` and `"1"` as strings and getting `"11"` instead of adding them together to get `"2"`) and ensuring that user manipulation of data doesn't cause errors. This is where BDD comes in.

Setting up your project with Mocha and Karma

We're going to use two things to write our unit tests: Karma and Mocha. Karma is the process that makes all our tests run inside of a browser, and Mocha is the test framework we're going to use. Chai is an add-on to Mocha that allows us to write BDD-style test assertions.

Install all of them in your project first, along with a few other goodies:

```
$ npm install mocha chai karma karma-webpack karma-chrome-launcher karma-
nyan-reporter karma-sourcemap-loader karma-mocha--save-dev
```

Why do we keep using `--save-dev` instead of just `--save` in this chapter? We use `--save-dev` because these are developer tools that only run in the developer environment. They shouldn't be part of the distributed code.

We need to create a config file for Karma. You can either run through the wizard by running this:

```
$ ./node_modules/karma/bin/karma init
```

And then spend the next four hours reading documentation about how to set up Karma, or you can just grab the prebuilt `karma.conf.js` file in the `chapter8` of the book repo (which has lots of comments explaining what I've done). Alternatively, you can use `generator-strong-d3` to scaffold things out really easily—generally, saving time by using known good build environments is a sound strategy. Configuring build tools is a massive time sink that can usually be avoided by using good boilerplate code.

Let's get a feel of BDD-style development with Mocha and Chai by updating our TypeScript class from earlier with a few new behaviors.

Testing behaviors first – BDD with Mocha

We really aren't going to write a full test suite because we have very little space left in the book and writing about automated testing to any significant degree could fill way more pages than we really have left together. But what we'll do is update our `TypeScriptChart` class to sort the bar chart either by population or alphabetically. However, we will do so in a BDD fashion.

To start, create a new folder called `test/` and create a file called `chapter8.spec.js`. The convention is to name your test files the same as the file it's testing, but with `.spec` before the file extension. Before we write any code, it's often helpful to write out our goals in plain English:

- The chart should sort data alphabetically ascending by region name
- The chart should sort data alphabetically descending by region name
- The chart should sort data ascending by population
- The chart should sort data descending by population

As you'll see, our assertions in Mocha will resemble these statements quite closely. Open up `chapter8.spec.js` and add the following code:

```
import {TypeScriptChart} from '../src/chapter8.ts';
let data = require('../src/data/chapter1.json');
let chart = new TypeScriptChart(data);
describe('ordering a TypeScriptChart', () => {
  describe('alphabetically', () => {
    it('should sort data ascending', () => {});
    it('should sort data descending', () => {});
  });

  describe('by population', () => {
    it('should sort data ascending', () => {});
    it('should sort data descending', () => {});
  });
});
```

First, we import our chart library and our data and set up our parent `describe` block. The `describe` block is used to organize your tests. Having one parent `describe` block per `.spec` file that then contains more logical groupings is a good convention to follow. We then scaffold our assertions using `it`. Similar to `describe`, `it` groups a number of tests together (the assertions that we'll soon write in each `it` statement's callback function). Think of each `it` statement as a test and the assertions contained therein as the parts of the test needed for it to verify what the test says.

In our tests here, we're using the actual data that the chart will display. Because these are behavior tests, we're just testing to ensure that the behavior that we expect actually occurs. Often, instead of loading in a dataset, you'll create a mock data sample, which you'll use to test each function or method. Using mock data is often much faster and helps ensure that the tests are functionally correct; in other words, we ensure that they're not passing merely because of an edge-case bug resulting from, for instance, a dirty dataset. Which testing method you use is entirely up to you and how detailed you want to be with your testing.

Before we add assertions, we need to make our code do the behavior we're testing. Update your test to resemble this:

```
describe('ordering a TypeScriptChart', () => {
  describe('alphabetically', () => {
    it('should sort data ascending', () => {
      chart.order('alphabetical', 'asc');
    });
    it('should sort data descending', () => {
      chart.order('alphabetical', 'desc');
    });
  });

  describe(' by population', () => {
    it('should sort data ascending', () => {
      chart.order('population', 'asc');
    });
    it('should sort data descending', () => {
      chart.order('population', 'desc');
    });
  });
});
```

As you can see, our chart now seems to have an `order` method, which takes two arguments: ordering `type` and `direction`. Our humble `TypeScriptChart` class doesn't have one of these yet, but we'll get to that shortly. First, we need to write our assertions. Update the first nested `describe` block to resemble the following:

```
  describe('alphabetically', () => {
    it('should sort data ascending', () => {
      chart.order('alphabetical', 'asc');
      chart.x.domain().should.have.length(8);
      chart.x.domain()[0].should.equal('burundi');
      chart.x.domain()[7].should.equal('yemen');
    });
    it('should sort data descending', () => {
      chart.order('alphabetical', 'desc');
      chart.x.domain().should.have.length(8);
      chart.x.domain()[0].should.equal('yemen');
      chart.x.domain()[7].should.equal('burundi');
    });
  });
```

We look at the `x` scale's domain to see whether the sort has worked or not because that's what we use to update things such as the *x* axis. From querying our dataset, we know that the first item alphabetically is Burundi and the last is Yemen. In each test, we first verify that the domain has the right number of items, and then verify that the first item is what it should be and the last is what it should be. If we wanted, we could have checked the entire array, but that would be more work and make the test slightly more brittle. Let's do the second block now:

```
describe('by population', () => {
  it('should sort data ascending', () => {
    chart.order('population', 'asc');
    chart.x.domain().should.have.length(8);
    chart.x.domain()[0].should.equal('liberia');
    chart.x.domain()[7].should.equal('syria');
  });
  it('should sort data descending', () => {
    chart.order('population', 'desc');
    chart.x.domain().should.have.length(8);
    chart.x.domain()[0].should.equal('syria');
    chart.x.domain()[7].should.equal('liberia');
  });
});
```

It suffices to say that these really aren't the best tests. Because you're not using mock data, it's not entirely obvious where strings such as `syria` and `liberia` are coming from. Our dataset is small enough that we can get away with it, though.

From the preceding lines, you can get a taste for what Mocha Chai BDD syntax is like. There are also a few other types of syntax you can use, depending on your preference and testing style, but I find the Chai BDD style easiest to read. As well, I use the "Should" variant; the other variant, "Expect", would look like this:

```
expect(chart.x.domain()).to.have.length(8);
```

Cool! Now that we've written our assertions, it's time to write some *actual* code! Open up `chapter8.ts` and add the following function to your `TypeScriptChart` class:

```
  order(type, direction) {
    this.data = this.data.sort((a, b) => {
      switch(type) {
        case 'population':
          return direction.indexOf('asc') ?
            b.population - a.population : a.population -
b.population;
```

```
          case 'alphabetical':
            return direction.indexOf('asc') ?
              (a.name < b.name ? 1 : a.name > b.name ? - 1 : 0) :
              (a.name > b.name ? 1 : a.name < b.name ? - 1 : 0);
        }
      });
    this.redraw();
  }
```

We still need to write the `redraw` function to update all the scales; put this below the preceding code:

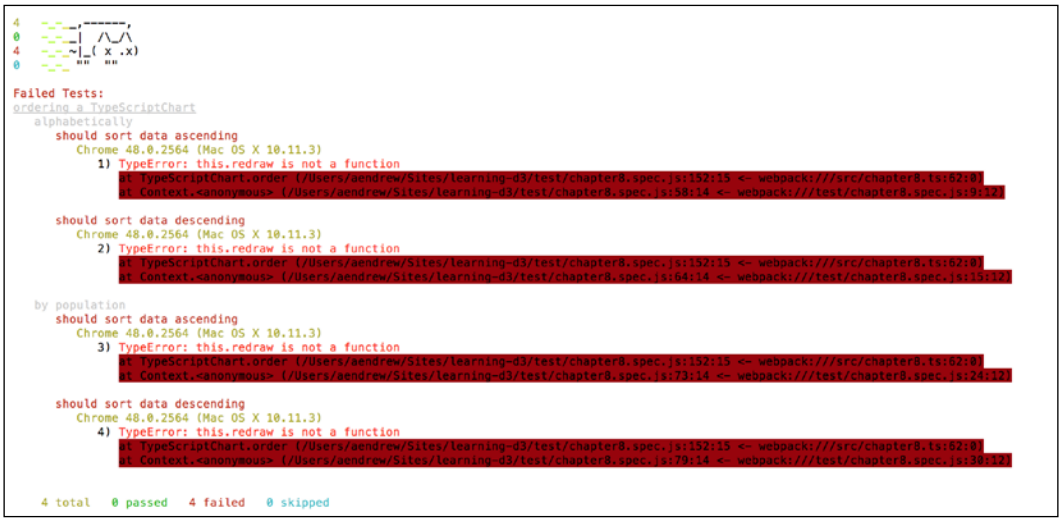
```
  redraw() {
    this.bars.data(this.data)
        .enter()
        .append('rect')
        .attr('class', 'bar')
        .attr('x', (d) => this.x(d.name))
        .attr('width', this.x.rangeBand())
        .attr('y', () => this.y(this.margin.bottom))
        .attr('height', 0);

    this.bars.transition()
      .delay((d, i) => i*200)
      .duration(800)
      .attr('y', (d) => this.y(d.population))
      .attr('height', (d) => this.height - this.y(d.population));

    this.x.domain(this.data.map((d) => d.name ));
    let xAxis = d3.svg.axis().scale(this.x).orient('bottom');
    this.chart.select('.x.axis').call(xAxis);
  }
```

We are finally done! Jump back to your terminal and run Karma:

```
$ ./node_modules/karma/bin/karma start
```

Look at that! 4/4 passed! Nyan Cat loves us!

It doesn't always look like that though. If we had run Karma before we wrote our `redraw` function, the output would have looked like this:



Yikes, Nyan Cat isn't looking so good now.

In some ways, the latter is far more useful. It tells us where our tests are failing, which we can use to quickly debug problems with our code. This is the whole point of tests—to be able to pinpoint exactly where something's broken when something goes wrong during development.

Once again, there's absolutely no way I can go into a topic this big in the space of one chapter, but hopefully, you get a sense of how you can start building testing into your projects. If this is something that interests you, I highly recommend doing quite a bit more reading about the philosophy behind automated testing and BDD. Although writing tests is fairly easy, writing *good* tests takes quite a lot of skill to master.

Even though automated testing is a good tool for giving yourself confidence in your data visualizations, don't trust it absolutely. It's possible to write a whole lot of tests that don't really test anything, and things like confirmation bias have a tendency to creep into test writing. That aside, it can be a tremendously helpful tool for producing world-class code.

Next time you're feelin' like you're an awesome automated test writer, do this fun 2-minute interactive quiz from NYT:

`http://www.nytimes.com/interactive/2015/07/03/upshot/a-quick-puzzle-to-test-your-problem-solving.html`.

Then, if you get it wrong, write more negative test assertions.

Summary

As much as it pains me, I believe it is nearly time to bid each other adieu. I truly hope you've learned some things and enjoyed the preceding eight chapters—writing a textbook is a fine balance of getting to the quick 'n' dirty learning bits while also having some fun along the way. Regardless, if you have somehow read this thing from beginning to end, my hats off to you, as we have covered an absolutely mind-boggling array of technologies and approaches to software development.

We started off with some super basic stuff, talking about the DOM and CSS. Then we deep-dived into SVG and you learned how to build super pretty web vector graphics. After that, we started doing some really neat stuff with D3—remember that chapter on layouts when we made, like, a gazillion charts?!

We've done some cool stuff with Node.js and Canvas, we've made a boatload of cool maps, and we even put some stuff on Heroku because we're web development ninjas! Hee-yah! So much stuff!

Lastly, we talked about making truly great projects with all of this technology and being confident in our work. Whether it's looking to others for inspiration or wiring up a solid test suite to make sure that our work is accurate, data visualization is a craft. It is one craft that will only continue to grow in importance as our world becomes all the more "data rich," but one that requires a degree of precision. Don't worry if you get some things wrong, but do try your hardest to get everything right.

We are in a truly amazing era of the Web, where the restrictions that were once preventing developers from building brilliant things are slowly being eliminated, by smarter approaches, bolder decision-making, and better tooling. It is all moving *frighteningly* fast, but don't let that hold you back from trying new things and playing with code that is on the cutting edge. There is no other field in computing where you can build things that work so *universally* and so *instantaneously*. Although web development can be really difficult (hello, responsive design!), never has any one set of technologies been so utterly crucial to the way the world consumes information. Having finished this book, you now have in possession a massive war chest of tools you can try to use. Some of them (particularly the ones I sprinted through, that is, Node, Canvas, TypeScript, and Mocha) you clearly need to learn a lot more about to use effectively, but hopefully, I've given you a few things to try out in your journey to understand and make use of all this stuff. At the very least, you should have everything you need to start visualizing your world with D3 and be able to share it with the world through *the magic of the Internet*.

Finally, if you get frustrated and need either a hand or a place to express your annoyance that this *thing* you've been working on is still broken two days later, come join us on the D3 Slack channel—there's usually somebody or the other around who's been there and can help.

-Æ.

Index

Symbols

A

B

C

D

E

F

G

H

I

T

U

V

W

Z