

Ændrew Rininsland, Swizec Teller

D3.js 4.x Data Visualization

Third Edition

Learn to visualize your data with JavaScript



Packt

Title Page

D3.js 4.x Data Visualization

Third Edition

Learn to visualize your data with JavaScript

Ændrew Rininsland
Swizec Teller

Packt

BIRMINGHAM - MUMBAI

Copyright

D3.js 4.x Data Visualization

Third Edition

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2013

Second edition: April 2016

Third edition: April 2017

Production reference: 1250417

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

ISBN 978-1-78712-035-8

www.packtpub.com

Credits

Authors Ændrew Rininsland Swizec Teller	Copy Editor Dhanya Baburaj
Reviewer Gerardo Furtado	Project Coordinator Devanshi Doshi
Commissioning Editor Ashwin Nair	Proofreader Safis Editing

Acquisition Editor Shweta Pant	Indexer Tejal Daruwale Soni
Content Development Editors Johann Barretto Onkar Wani	Graphics Jason Monterio
Technical Editor Rashil Shah	Production Coordinator Shraddha Falebhai

About the Authors

Andrew Rininsland is a developer and journalist who has spent much of the last half a decade building interactive content for newspapers such as *The Financial Times*, *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. He is currently a senior developer with the interactive graphics team at the *Financial Times*.

I would like to thank my amazing partner, Naomi, for the countless times she encouraged and motivated me over the course of this book's writing. I'd also like to thank everyone on the D3.js Slack for an incredible amount of guidance while writing, particularly the members of the #v4-migration channel for all the help adjusting to the myriad of changes in v4.

About the Author2

Swizec Teller, author of *Data Visualization with d3.js*, is a geek with a hat. He founded his first start-up at the age of 21 years and is now looking for the next big idea as a full-stack Web generalist focusing on freelancing for early-stage start-up 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

Gerardo Furtado is a biology teacher, science communicator and writer, and author of a book on evolutionary biology published by the University of Brasilia, as well as fiction books.

His second academic passion is data visualization, which ultimately led him to D3.js, the most powerful (and complex) JavaScript library to create visualizations based on data.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

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 service@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://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at "<https://www.amazon.com/dp/178712035X>".

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

What this book covers

What you need for this book

Who this book is for

Conventions

Reader feedback

Customer support

Downloading the example code

Errata

Piracy

Questions

1. Getting Started with D3, ES2017, and Node.js

What is D3.js?

What happened to all the classes?

What's new in D3 v4?

What's ES2017?

Getting started with Node and Git on the command line

A quick Chrome Developer Tools primer

The obligatory bar chart example

Loading in data

Twelve (give or take a few) bar blues

Summary

2. A Primer on DOM, SVG, and CSS

DOM

Manipulating the DOM with d3-selection

Selections

Let's make a table

What exactly did we do here?

Selections example

Manipulating content

Joining data to selections

An HTML visualization example

Scalable Vector Graphics

Drawing with SVG

Manually adding elements and shapes

Text

Shapes

Transformations

CSS

Summary

3. Shape Primitives of D3

Using paths

Line

Area

Arc

Symbol

Chord/Ribbon

Axes

Summary

4. Making Data Useful

Thinking about data functionally
Built-in array functions

Data functions of D3

Managing objects with d3-collection
What about ES6 Maps and Sets?

Scales

Ordinal scales

Quantitative scales

Continuous range scales

Discrete range scales

Time

Formatting

Time arithmetic

Loading data
The core

Flow control
Promises

Generators

Observables

Geography
Getting geodata

Drawing geographically

Using geography as a base

Summary

 5. Defining the User Experience - Animation and Interaction

Animation

Animation with transitions

Interpolators

Easing

Timers

Putting it all together - sequencing animations

Interacting with the user

Basic interaction

Behaviors

Drag

Brushes

Zoom

Do you even need interaction?

Summary



6. Hierarchical Layouts of D3

What are layouts and why should you care?

Built-in layouts

Hierarchical layouts

Tree the whales!

Muster the cluster!

Money for nothing, treemaps for free (maps)

Smitten with partition

Pack it up, pack it in, let me begin...

Bonus chart! Sunburst radial partition joy!

Summary

7. The Other Layouts

Hoorah for modular code

When the moon hits your eye (chart), like a big pizza pie (chart)

Histograms, Herstograms, Yourstograms, and Mystograms

Striking a chord

May the force (simulation) be with you

Got mad stacks

Bonus chart - Streamalicious!

Summary

8. D3 on the Server with Canvas, Koa 2, and Node.js

Readyng the environment

All aboard the Koa train to servertown!

Proximity detection and the Voronoi geom

Rendering in Canvas on the server

Deploying to Heroku

Summary

9. Having Confidence in Your Visualizations

Linting everything!

Static type checking: TypeScript versus Tern.js
Code analysis with Tern.js

TypeScript - D3 powertools

Behavior-driven development with Mocha and Chai
Setting up your project with Mocha

Testing behaviors - BDD with Mocha

Summary

10. Designing Good Data Visualizations

Choosing the right dimensions, choosing the right chart

Clarity, honesty, and a sense of purpose

Helping your audience understand scale

Using color effectively

Understanding your audience

Some principles for designing for mobile and desktop
Columns are for desktops, rows are for mobiles

Be sparing with animations on mobiles

Realizing similar UI elements react differently between platforms

Avoiding mystery meat navigation

Be wary of the scroll

Summary

Preface

Welcome to *D3.js 4.x Data Visualization, Third 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 don't stop there. By the end of our time together, you'll have all the skills you need to become a total D3 ninja, and will be able to do everything from build visualizations from scratch straight through to using it on the server and writing automated tests. If you haven't leveled up your JavaScript skills in a while, you're in for a treat--this book endeavors to use the latest features currently being added to the language, all the while explaining why they're cool and how they differ from "old school" JavaScript.

What this book covers

[Chapter 1](#), *Getting Started with D3, ES2017, and Node.js*, covers 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](#), *Shape Primitives of D3*, identifies and creates the basic shapes that comprise a data visualization.

[Chapter 4](#), *Making Data Useful*, teaches how to transform data so that D3 can visualize it.

[Chapter 5](#), *Defining the User Experience – Animation and Interaction*, helps you use animation and user interactivity to drive your data visualizations.

[Chapter 6](#), *Hierarchical Layouts of D3*, focuses on how hierarchical layouts can take your D3 skills to the next level by providing reusable patterns for creating complex charts.

[Chapter 7](#), *The Other Layouts*, discusses the non-hierarchical layouts that speed the creation of many addition complex chart types.

[Chapter 8](#), *D3 on the Server with Canvas, Koa 2, and Node.js*, outlines how to build and deploy a Node.js-based web service that renders D3 using Koa.js and Canvas.

[Chapter 9](#), *Having Confidence in Your Visualizations*, showcases how to improve the quality of your code by introducing linting, static type checking, and automated testing to your projects.

[Chapter 10](#), *Designing Good Data Visualizations*, compares and contrasts differing approaches to data visualization while building a set of best practices.

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 you 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 it.

You should also have 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 also try to work in Safari or Internet Explorer/Edge, or Opera, or any number of other browsers, but I find Chrome's developer tools to be 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: "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."

A block of code is set as follows:

```
"babel": {  
  "presets": [  
    "es2017"  
  ],  
  "main": "lib/main.js",  
  "scripts": {  
    "start": "webpack-dev-server --inline",  
  },
```

Any command-line input or output is written as follows: **\$ brew install n**
\$ n lts

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: "We will mostly use the Elements and Console tabs, Elements to inspect the DOM and Console to play with JavaScript code and look for any problems."



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

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/D3.js-4.x-Data-Visualization>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

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.

Getting Started with D3, ES2017, and Node.js

Data-Driven Documents (D3), developed by Mike Bostock and the D3 community in 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 popular jQuery JavaScript library. Much like jQuery, in D3, you operate on elements by selecting and then manipulating them 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 to start with. After finishing with this book, you should be able to understand D3 well enough to figure out the examples, tweaking them to fit your needs. If you want to follow the development of D3 more closely, check out the source code hosted on GitHub at <https://github.com/d3>.

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 2017 (ES2017)**--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 v4 to render a basic chart.

What is D3.js?

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 may 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>.

D3 is ultimately based around functional programming principles, which is currently experiencing a renaissance in the JavaScript community. This book isn't really about functional programming, but a lot of what we'll do will seem really familiar if you've ever used functional programming principles before. If you haven't, or come from an **Object-Oriented (OO)** background like I do, don't worry, I'll explain the important bits as we get to them, and the revised section on functional programming at the beginning of [Chapter 4, Making Data Useful](#), will hopefully give you some insight into why this paradigm is so useful, especially for data visualization and application construction.

What happened to all the classes?

The second edition of this book contained quite a number of examples using the class feature that is new in ES2015. The revised examples in this edition use *factory functions* instead, and the class keyword never appears. Why is this, exactly?

ES2015 classes are essentially just *syntactic sugaring* for factory functions. By this I mean that they ultimately transpile down to factory functions *anyway*. Although classes can provide a certain level of organization to a complex piece of code, they ultimately hide what is going on underneath it all. Not only that, using OO paradigms, such as classes, is effectively avoiding one of the most powerful and elegant aspects of JavaScript as a language, which is its focus on first-class functions and objects. Your code will be simpler and more elegant using functional paradigms than OO, and you'll find it less difficult to read examples in the D3 community, which almost never use classes.

There are many, much more comprehensive arguments against using classes than I'm able to make here. For one of the best, refer to Eric Elliott's excellent *The Two Pillars of JavaScript* pieces at:

www.medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3.

What's new in D3 v4?

One of the key changes to D3 since the last edition of this book is the release of version 4.

Among its many changes, the most significant one is a complete overhaul of the D3 namespace. This means that none of the examples in this book will work with D3 3.x, and the examples from *Learning D3.js Data Visualization, Second Edition* will not work with D3 4.x. This is quite possibly the cruelest thing Mr. Bostock could ever do to educational authors such as myself (I am joking here!). Kidding aside, it also means many of the *block* examples in the D3 community are out of date and may appear rather odd if this book is your first encounter with the library. For this reason, it is very important to note the version of D3 an example uses -- if it uses 3.x, it might be worth searching for a 4.x example just to prevent this cognitive dissonance.

You can usually find the version of D3 in an example block online by looking at the script tags near the top of the code. If it resembles:



<script src="https://d3js.org/d3.v3.min.js"></script>

...You're looking at a D3 v3 example. If it says:

<script src="https://d3js.org/d3.v4.min.js"></script>

...You're looking at a modern, v4 example and are good to go.

Related to this is how D3 has been broken up from a single library into many smaller libraries (or micro-libraries). There are two approaches you can take:

- You can use D3 as a single library (*a monolib*) in much the same way as version 3
- You can selectively use individual (*microlib*) components of D3 in your project

This book takes the former route. While learning D3, using micro-libraries takes a lot more effort, even if it helps reduce the size of the final bundle that people who view your graphics will have to download. That said, I will try to signpost which package a particular piece of functionality resides in; so, once you

become more comfortable with D3, you can start using the microlibs instead of including everything and the kitchen sink.

What's ES2017?

One of the main changes to this book since its first edition is the emphasis on modern JavaScript; in this case, ES2017. Formerly known as ES6 (Harmony), it pushes the language features of JavaScript forward significantly, allowing for new usage patterns that simplify code readability and increased 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. However, 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 to be something of a 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 ES2015-17, check out <https://babeljs.io/docs/learn-es2015/>, a nice guide by the folks at Babel.js, which we will use extensively throughout this book.

Before I go any further, let me clear some confusion about what ES2017 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 (2017) now. The big release was ES2015, which more or less maps to ES6. ES2016 was ratified in June 2016, and builds on the previous year's standard, while adding a few fixes and two new features. ES2017 is currently in the draft stage, which means proposals for new features are being considered and developed until it is ratified sometime in 2017. As a result of this book being written while these features were in draft, they may not actually make it to ES2017 and thus may need to wait until a later standard to be officially added to the language.

You don't really need to worry about any of this, however, because we use Babel.js to transpile everything down to ES5 anyway, so it runs the same in

Node.js and in the browser. I will try to refer to the relevant specification where a feature is added, when I introduce it for the sake of accuracy (for instance, modules are an ES2015 feature), but when I refer to JavaScript, I mean all modern JavaScript, regardless of which ECMAScript specification it originated in.

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 macOS X), but you will 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 macOS X without Homebrew, use the installer at <https://nodejs.org/en/>. If you're on macOS 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 lts
```



If you're in Windows, the `$` above might be confusing. In UNIX-based operating systems, regular users see a `$` on the command prompt, and the root administrator user sees a `#`. By indicating that, I mean that you should run the above commands as a regular user and not a super-user.

Regardless of how you do it, when you have finished, verify by running the following lines:

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

If it displays the versions of `node` and `npm`, it means you're good to go.



I'm using 6.5.0 and 3.10.3, respectively, though yours might be slightly different--the key is making sure that node is at least version 6.0.0.

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.

Throughout this book, we will use a combination of Babel and Webpack to turn our fancy modular modern JavaScript into something even the crummiest old browsers (Hi, Internet Explorer 9!) can run.

First, create a `package.json` file to store the version of each dependency that we want. Do this by first creating a new folder and then running `npm init`:

```
| $ mkdir d3-projects  
| $ cd d3-projects  
| $ npm init -y
```

The `-y` flag tells `npm init` to use all the default settings and not ask you any questions.

Next, install our stack, using `npm`:

```
| $ npm install "babel-core@^6" "babel-loader@^6" "babel-preset-es2017@^6" "babel-preset-
```

This installs v2 of Webpack, v6 of Babel, and a boatload of presets and plugins for both. It then saves these to `package.json` so that you can reinstall them as easily as running:

```
| $ npm install
```

You'll also need to install D3 by typing the following command:

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

Next, we need to create a configuration for Webpack. I won't get into detail around what each of the directives do; for that, look at the config supplied with the book repository; save this as `webpack.config.js`:

```
const path = require('path');  
module.exports = [{  
  entry: {  
    app: ['./lib/main.js'],  
  },  
  output: {  
    path: path.resolve(__dirname, 'build'),  
    publicPath: 'assets',  
    filename: 'bundle.js',  
  },  
}];
```

```
  },
  devtool: 'inline-source-map',
  module: {
    rules: [
      { test: /\.js$/,
        exclude: /(node_modules|bower_components)/,
        loader: 'babel-loader',
      },
      { test: /\.json$/,
        loader: 'json-loader',
      },
      { test: /\.css$/,
        loader: 'style-loader!css-loader',
      },
    ],
  },
};
```

Lastly, we need to edit `package.json` to have a few shortcuts to make our lives easier. After the line that starts with `name`, put the following:

```
"babel": {
  "presets": [
    "es2017"
  ]
},
"main": "lib/main.js",
"scripts": {
  "start": "webpack-dev-server --inline",
},
```

This is what you need to do if you're starting a project from scratch.

Alternatively, you can clone the book's repository from GitHub. GitHub is where most of the world stores open source and other code. I've done a lot of configuration for you, in addition to supplying all of the examples and sample data. I'll move forward under the assumption you've cloned the book repository and are working out of the book repository directory. To do this, run the following command:

```
$ git clone https://github.com/aendrew/learning-d3-v4
$ cd learning-d3-v4
```

This will clone the development environment and all the samples in the `learning-d3-v4/` directory, as well as switching you into it and installing all of the dependencies via `npm`.

Another option is to fork the repository on GitHub and then clone your fork instead of mine as was just shown in the preceding code. This will allow you to easily publish your work on the cloud,



enabling you to more readily 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-v4` by clicking the "fork" button in GitHub, 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 on the master branch 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.

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 `lib/` folder. You'll notice it contains just a `main.js` file; almost always, we'll be working in `main.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>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learning Data Visualization with D3.js</title>
  </head>
  <body>
    <script src="assets/bundle.js"></script>
  </body>
</html>
```

There's also an empty style sheet in `styles/index.css`, which we'll add to in a bit.

Next, start the development server by typing the following line:

```
| $ npm start
```

This starts up the Webpack development server, which will transform our **new-fangled** ES2017 JavaScript into backward-compatible ES5 and serve it to the browser.

Now, point Chrome (or whatever, I'm not fussy--so long as it's not Internet Explorer!) 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:

localhost:8080/src/

Elements Network Sources Timeline Profiles »

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <script src="/assets/bundle.js"></script>
  </body>
</html>
```

html body script

Styles Event Listeners DOM Breakpoints Properties \$scope

Filter

```
element.style { }  
script { display: none; }
```

margin -
border -
padding -
auto x auto

Console Emulation Rendering

<top frame> ▾ Preserve log

Filter Regex Hide network messages

All Errors Warnings Info Logs Debug Handled

>

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 just Chrome here for the sake of simplicity. Feel free to use a different browser. Firefox's Developer Edition is particularly nice, and--yeah yeah, I hear you guys at the back--Opera is good too!

We will mostly 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:

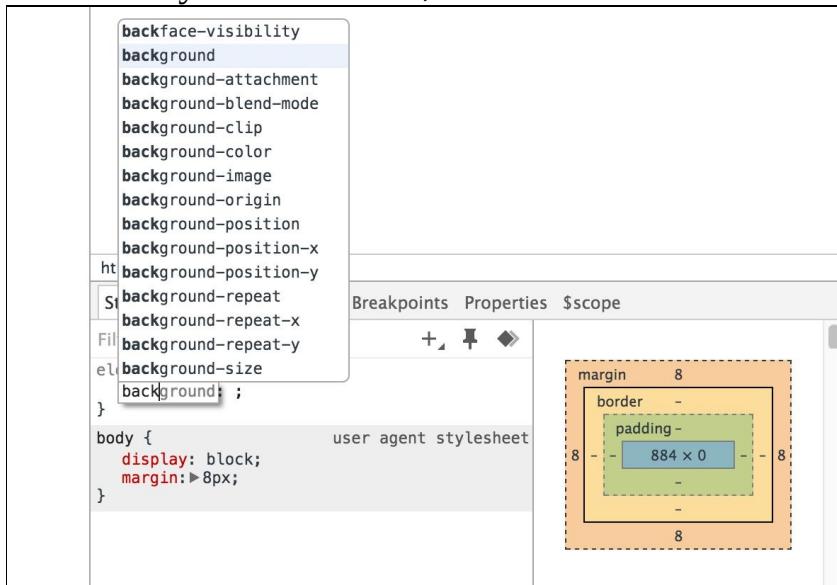


The Network tab will let you know how long files will take to load and help you inspect the Ajax requests. The Profiles tab will help you profile JavaScript for performance. The Resources tab is good for inspecting client-side data. Timeline and Audits are useful when you have a global variable that 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.

The main one you want to focus on, however, is **Sources**, which shows all the source code files that have been pulled in by the web page. Not only is this useful in determining whether your code is actually loading, it also contains a

fully functional JavaScript debugger, which few mortals dare to use. Although explaining how to debug code is kind of boring and not at the level of this chapter, learning to use breakpoints instead of perpetually using `console.log` to figure out what your code is doing, is a skill that will take you far in the years to come. For a good overview, visit <https://developers.google.com/web/tools/chrome-devtools/debug/breakpoints/step-code?hl=en>

Most of what you'll do with Developer Tools, however, involves looking at the CSS inspector at the right-hand side of the Elements tab. It can tell you what CSS rules are impacting 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% 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 *Financial Times* or *The Economist* includes in their articles--they frequently summarize the entire piece with a simple line chart or histogram. Coming from a newsroom development background (Full disclosure: I work on the Interactive Graphics desk at the *Financial Times*), 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 the UK Electoral Commission's result set from the UK Brexit referendum. We will draw a bar chart depicting voter turnout by region.

The source for this data is at <http://www.electoralcommission.org.uk/find-information-by-subject/elections-and-referendums/past-elections-and-referendums/eu-referendum/electorate-and-count-information>.

We'll create a bar for each region in the UK. The first step is to get a basic container setup, which we can then be populated with all of our delicious new JavaScript code. We can either jump straight into the code, or we can set up some stuff to make life easier for us later on. Let's go with the first route for now. There's a lot of new-fangled JavaScript stuff coming at you really soon, so let's keep it light for the moment.

First open `lib/main.js` and write your very first line of D3:

```
const chart = d3.select('body')
  .append('svg')
  .attr('id', 'chart');
```

This selects the HTML `<body>` element, appends a `<svg>` element, and gives it the ID of `#chart`. We'll be using this pattern a lot throughout the book.

Before we get any further, it's worth pointing out our first new-fangled modern JavaScript feature:

The `const` keyword is used to define a variable that won't change dramatically. By dramatically I mean that you can still modify it somewhat (for instance, adding elements to an array or modifying an object), but you'll throw an error if you try to reassign it. It will also throw an error if you try to use it before it's declared. Unlike constants in other languages, JavaScript constants only apply to the current function scope (they're not global unless you make them that). This is really useful when using a functional programming style, as it prevents weird bugs caused by variable hoisting (an unusual JavaScript language feature, whereby variables are ultimately interpreted at the top of each function closure instead of where you actually define them). For more on `const`, visit <http://mdn.io/const>.



Another new way to define variables in JavaScript is using `let`, which is like `var` but, like `const`, has block scope, meaning that it is limited to the block, statement, or expression where it's used. This also helps prevent weird bugs. For more information, visit <http://mdn.io/let>.

When should you use each? Use `const` if you're not going to reassign a variable and use `let` if you will. I try to avoid reassigning variables, so I will usually use `const` in this book. However, while you still can use the `var` keyword to assign variables in modern JavaScript, you really shouldn't--always use `let` or `const` instead, defaulting to `let` if you're not sure which will work in a given situation.

Yeah! Let's open this up in the browser; ensure that the development server is running (`npm start` if it isn't) and go to `http://localhost:8080`

Uncaught Error: Cannot find module "d3"

Whups. Okay, that could have gone better...

You're getting this error because we haven't imported D3, yet. If you've used D3 before, you might be more familiar with it attached to the `window` global object. This is what happens when you include `d3.js` via a `<script>` tag. We're not doing

that, however--we're JavaScript **rockstarninjaciraptors**; we use the new hotness, ES2015 module imports!

Go back to `main.js`. At the top of the file, type this:

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

Let's unpack this a bit. Import statements must be at the top of every file (so no sneaky Node.js-style `require()` calls inside your functions), because it allows for static analysis. This lets new JavaScript tools be more effective. They always start with the `import` token.

Next is the curly bracket bit. In an ES2015 module, there are two types of exports:

- **Named:** This is where you give the export a title that needs to be imported specifically (though it can be renamed), and it is inside curly brackets.
- **Default:** There can be only one of these per module, and it can be referred to as anything when importing. We'll see this a bit later on.

What we do above is import all of the D3 microlibs under the namespace `d3`.

If you go back to your browser and switch quickly to the Elements tab, you'll notice a new SVG with an ID of `#chart` at the bottom of the page. There's progress!

Loading in data

Go back to `main.js`. We need to get our data in somehow, and I'll show you far better ways of doing this later on, but let's work through the pain and do this the bad, old way--using `XMLHttpRequest`:

```
const req = new window.XMLHttpRequest();
req.addEventListener('load', mungeData);
req.open('GET', 'data/EU-referendum-result-data.csv');
req.send();
```

This instantiates a new `XMLHttpRequest` object, tells it to load the data from the data/directory and then passes it to the soon-to-be-written `mungeData()` function once loaded.

Note how we had to use the ugly `new` keyword to instantiate it? Note how it took four lines and a new variable declaration? Note how we have to handle our response in a callback? Eww! We'll improve upon this in later chapters. The only advantage of doing things this way is that it works in nearly any browser without polyfilling, but there are so many better ways of doing this, all of which we will touch upon in [Chapter 4, Making Data Useful](#).

The CSV file we're loading in has a row for each constituency in the UK, containing everything from what percentage voted for what to what the voter turnout was to how many ballots were invalid or spoiled. What we will do is turn that into an array of objects depicting the mean percentage for each broader region that voted for leaving.

It's time to create our `mungeData()` function. We will use `d3.csvParse()` (from the `d3-dsv` microlib) to parse our CSV data string in an object and then use some features from the `d3-array` microlib to manipulate that data:

```
function mungeData() {
  const data = d3.csvParse(this.responseText);
  const regions = data.reduce((last, row) => {
    if (!last[row.Region]) last[row.Region] = [];
    last[row.Region].push(row);
    return last;
  }, {});
  const regionsPctTurnout = Object.entries(regions)
    .map(([region, areas]) => ({
      region,
```

```
    meanPctTurnout: d3.mean(areas, d => d.Pct_Turnout),  
});  
} renderChart(regionsPctTurnout);
```

 Hey, there's another ES2015 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. This won't impact us very much because we're using a functional style of programming; but if we were using classes, this would be a lifesaver. For more on this, visit http://mdn.io/Arrow_functions.

We transform our data in three steps here:

1. First, we convert it into an array of objects using `d3.csvParse()` and assign the result to `data`.
2. Then, we transform the array into an object keyed by the region, such that the object's keys are the regions, and the values are an array of associated constituencies.
3. Lastly, `Object.entries` converts an object into a multidimensional array consisting of elements comprising key-value pairs, which we can then reduce into a new object comprising each region's name and the mean of each constituency's voter turnout percentage.

You may have noted that the function signature for the call to `Array.prototype.map` is a little unusual:

```
| .map(([region, areas]) => {
```

Here, we use a new ES2015 feature called *destructuring assignment* to give each element in our array a temporary name. Normally, the callback signature is the following:

```
| function(item, index, array) {}
```

However, because we know `item` is an array with two elements, we can give each of them a nickname, making our code easier to read (we don't use the `index` or `array` arguments this particular time, but if we did, we'd just put those arguments after the destructuring bit).

Lastly, we pass our fully munged data to an as-of-yet-unwritten function, `renderChart()`, which we'll add next.

We can also simply add the above to this:

```
| const regionsPctTurnout = d3.nest()  
|   .key(d => d.Region)  
|   .rollup(d => d3.mean(d, leaf => leaf.Pct_Turnout))  
|   .entries(data);
```

`d3.nest()` is part of the `d3-collection` microlib, which we'll cover in--you guessed it--[Chapter 4, *Making Data Useful*](#). D3 is a very un-opinionated library, which means you can accomplish many tasks in a variety of ways--there often really isn't a proper way of doing things. I will try to expose a variety of ways to accomplish tasks throughout the book; feel free to choose whichever you prefer.

Twelve (give or take a few) bar blues

With that done, let's render some data.

Create a new function in `main.js`, `renderChart()`:

```
function renderChart(data) {
  chart.attr('width', window.innerWidth)
    .attr('height', window.innerHeight);
}
```

All this does is take our earlier `chart` variable and set its width and height to that of the window. We're almost at the point of getting some bars onto that graph; hold tight!

First, however, we need to define our scales, which decide how D3 maps data values to pixel values. Put another way, a scale is simply a function that maps an input range to an output domain. This can be annoying to remember, so I'm going to shamelessly steal an exercise from Scott Murray's excellent tutorial on scales from *Interactive Data Visualization for the Web*:

When I say "input," you say "domain." Then I say "output," and you say "range."
Ready? Okay:

Input! Domain!

Output! Range!

Input! Domain!

Output! Range!

Got it? Great.

It seems silly, but I frequently find myself muttering the above when I have a deadline and am working on a chart late at night. Give it a go!

Next, add this code to `renderChart()`:

```
const x = d3.scaleBand()
  .domain(data.map(d => d.region))
  .rangeRound([50, window.innerWidth - 50])
  .padding(0.1);
```

The `x` scale is now a function that maps inputs from a domain composed of our

region names to a range of values between 50 and the width of your viewport (minus 50), with some spacing defined by the `0.1` value given to `.padding()`. What we've created is a band scale, which is like an ordinal scale, but the output is divided into sections. We'll talk more about scales later on in the book.



In this example, we use a uniform value of 50 for our margins, which we pass to our scales and elsewhere. Any arbitrary number passed in code is often referred to as a magic number, insomuch that, to anyone reading your code, it just looks like a random value that magically makes it work. This is bad; don't do this--it makes your code harder to read, and it means that you have to find and replace every value if you want to change it. I only do so here to demonstrate this fact. Throughout the rest of the book, we'll define things, such as margins more intelligently; stay tuned!

Still inside `renderChart()`, we define another scale named `y`:

```
| const y = d3.scaleLinear()  
|   .domain([0, d3.max(data, d => d.meanPctTurnout)])  
|   .range([window.innerHeight - 50, 0]);
```

Similarly, the `y` scale is going to map a linear domain (which runs from zero to the max value of our data, the latter of which we acquire using `d3.max`) to a range between `window.innerHeight` (minus our 50 pixel margin) and 0. Inverting the range is important because D3 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 in one of your scales.

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

```
| const xAxis = d3.axisBottom().scale(x);  
| const yAxis = d3.axisLeft().scale(y);
```

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. 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 will draw the axes on our graph:

```
chart.append('g')
  .attr('class', 'axis')
  .attr('transform',
    `translate(0, ${window.innerHeight - 50})`)
  .call(xAxis);
```



Hot new ES2015 feature alert! Above, the transform argument is in backticks (`), which are template literal strings. They're just like normal strings, except for two differences: you can use newline characters in them, and you can also run arbitrary JavaScript expressions in them via the \${} syntax. Above, we merely echo out the value of window.innerHeight, but you can write any expression that returns a string-like value, for instance, using Array.prototype.join to output the contents of an array; it's really handy!

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:

```
chart.append('g')
  .attr('class', 'axis')
  .attr('transform', 'translate(50, 0)')
  .call(yAxis);
```

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

```
chart.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => x(d.region))
  .attr('y', d => y(d.meanPctTurnout))
  .attr('width', x.bandwidth())
  .attr('height', d =>
    (window.innerHeight - 50) - y(d.meanPctTurnout));
```

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 to it



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 nonexistent 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 `bandwidth()` 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`).

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:

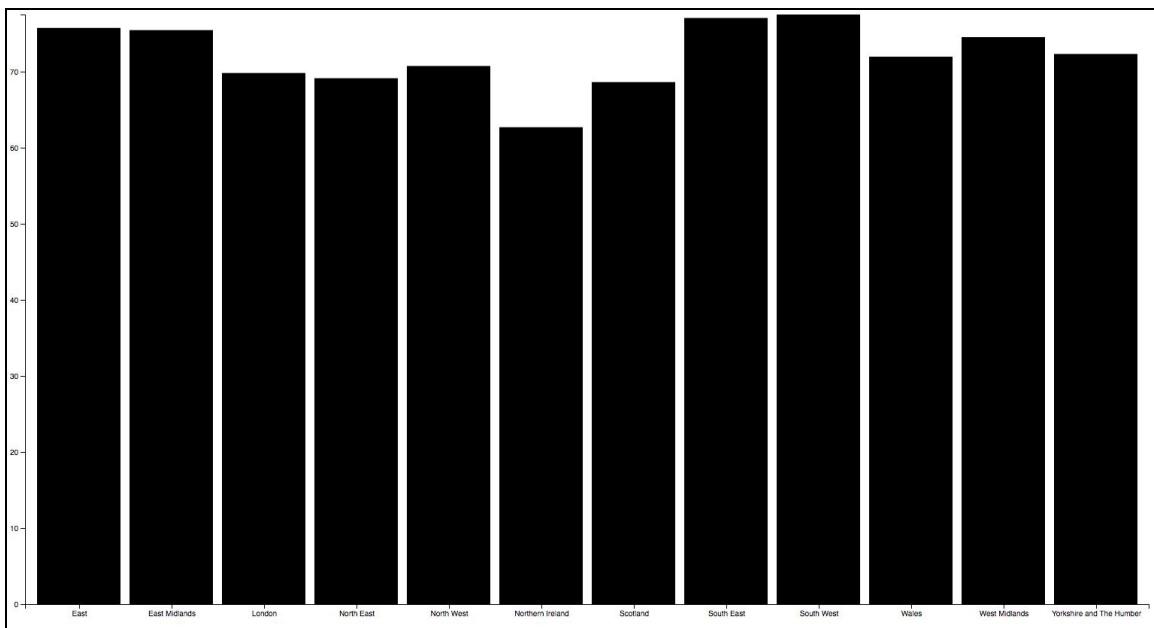
```
chart.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => x(d.region))
.attr('y', window.innerHeight - 50)
  .attr('width', x.bandwidth())
  .attr('height', 0)
  .transition()
  .delay((d, i) => i * 20)
  .duration(800)
  .attr('y', d => y(d.meanPctTurnout))
  .attr('height', d =>
    (window.innerHeight - 50) - y(d.meanPctTurnout));
```

The difference is that we statically put all bars at the bottom (`window.innerHeight - 50`) with a height of zero 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 datum 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. If we were using, say, classes, this last point would be fairly important; otherwise, we'd be evaluating the `rect` `SVGElement` object instead of whatever context we actually want to use. However, because we're mainly going to use factory functions for everything, figuring out which context is assigned to this is far less of a worry.

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 refresh. If everything went according to plan, you should have a chart that looks like the following:



According to this, voter turnout was fairly high during the EU referendum, with the south-west having the highest turnout. 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-v4/tree/chapter1> if you didn't get something

similar to the preceding screenshot.

We still need to do just a bit more, mainly 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. Also, where's the fun in writing HTML when we're learning some newfangled JavaScript?

First, create an `index.css` file in your `styles/` 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 the top of `main.js`:

```
| import * as styles from 'styles/index.css';
```

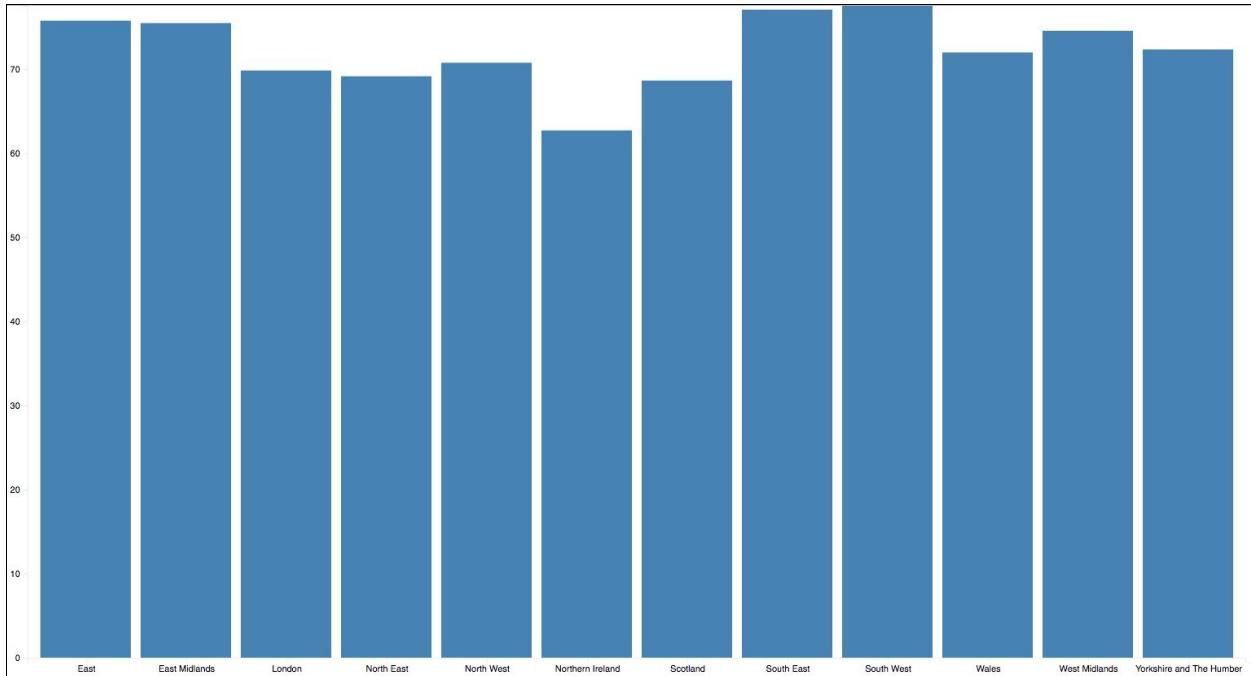
I know. Crazy, right? No `<style>` tags needed!



It's worth noting anything involving `require()` or `import` that isn't a JS file is the result of a Webpack loader. Although the author of this text is a fan of Webpack, all we're doing is importing the styles into `main.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 <https://github.com/webpack-contrib/css-loader>.

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 this and wait for the page to refresh. We've made our first D3 chart:



I recommend fiddling with the values passed to `.width` and `.height` 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 of 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, that we use a combination of CSS and SVG to make things beautiful. We also did a lot of fancy stuff with ES2017, 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 --soft origin/chapter1`, replacing 1 with whichever chapter you're on.

Next, we'll be looking at all this in 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.

A Primer on DOM, SVG, and CSS

You might already be used to manipulating DOM and CSS with libraries such as jQuery; if so, much of this will seem very familiar, as D3 has a full suite of manipulation tools. If not, don't worry, as this chapter exists to get everyone up to speed.

Very similar to HTML's DOM is the SVG namespace, which we'll use for most of the examples in this book. SVG is at the core of building truly great visualizations, so we'll take special care to understand it, starting out by manually drawing shapes and then doing the same using D3's path generators in [Chapter 3, *Shape Primitives of D3*](#).

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

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

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, which 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. Consider a simple HTML file to be like the following:

```
<!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>
```

Note how we don't have the `<html>`, `<head>` or `<body>` tags. Chrome will parse the preceding code to DOM, as follows:

A paragraph of text

- List item
- List item 2, *italic*

Type `document` into 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 relating to the specified element when you hover over it in the console.

You can also test random selections by typing `$('.some-selector')` into



the console. Even if jQuery isn't included in this page, it will still work because it's built into Chrome's console as an alias for `document.querySelector('.some-selector')` (and is then overridden by jQuery if it's included in the page). Additionally, `$('.some-selector')` acts as a shortcut to `document.querySelectorAll('.some-selector')`, if you want to return more than just the first element in the selection.

Manipulating the DOM with d3-selection

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

Take, for instance, the HTML code in our preceding example. If we want to change the word *italic* to make it underlined as well as bold and italic (the result of the `` and `` tags), we would do it using the following code:

```
document.querySelector('strong').style  
.setProperty('text-decoration', 'underline')
```

`document.querySelector` finds the first element of a CSS selection and then returns it. We then set underline as the `text-decoration` property.

Using the modern DOM API is a lot nicer than it used to be, but it's a bit fraught with peril unless you *polyfill* various functionalities into the browser. We could use jQuery, which gives a nice API for this style of development, or we could use D3, which comes with a similar set of tools for selecting and manipulating the DOM (all contained within the `d3-selection` microlibrary).

With `d3-selection`, we can treat HTML as just another type of data visualization. Let that one sink in--we can use our regular old HTML for 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 preceding example using `d3-selection`:

```
| import * as d3 from '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 will come in handy later.

What we just did is called a selection. Selections are the core of everything we do with D3--`d3.select` is used to select a single element, while `d3.selectAll` creates an array-like selection with multiple elements. Let's take a look at selections 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, to a tag name. We can even use pseudo-selectors, allowing us to do things, such as selecting every other paragraph tag using `p:nth-child(n+1)`.



Pseudo-selectors are really powerful when used with D3 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 cursor is above an element.

Using CSS selectors to create a chained series of manipulations gives us a simple language for defining elements in the document. It's actually the same as you're used to in jQuery and CSS.

To get the first element with ID as `graph`, we use `d3.select('#graph')`. To get all the elements with the class `blue`, we write `d3.selectAll('.blue')`. To get all the paragraphs in a document, we use `d3.selectAll('p')`.

We can combine these to get a more complex matching. You can perform a Boolean `AND` operation using the selector `.llama.duck`; 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`. However, what if you want to select child elements? Then, nested selections to the rescue!

You can do it with a simple selector, such as `tbody td`, or you can chain two `d3.selectAll()` calls as `d3.selectAll('tbody').selectAll('td')`. Both will select all the cells in a table body. Keep in mind that nested selections maintain the hierarchy of selected elements, which gives us some interesting capabilities.

Let's make a table

It's time for an example. Start by creating a new folder called `chapter2/` in your `lib/` directory, and then create a blank file in that called `table-factory.js`. We will not do much directly inside `main.js` from here on--instead, we will create modules that get loaded by `main.js`. This allows us to keep our code clean and split it into manageable pieces. Working with modules is a key skill both for organizing your code and reusing useful code you've written--getting into the practice of breaking your project into smaller components is something that will serve you well as you grow as a JavaScript developer.

Since we don't want to play around with HTML inside of our codebase too much, we will write a bunch of functions to build our tables. We're going to make the following 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 just copy that into `index.html`. However, remember, we're trying to avoid writing any HTML here? It's time to make a table using D3.

Open up `table-factory.js` and create the following factory function:

```
import * as d3 from 'd3';
export default function tableFactory(_rows) {
```

```

const rows = Array.from(_rows);
const header = rows.shift(); // Remove the first row for the header
const data = rows; // Everything else is a normal data row

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

return {
  table,
  header,
  data,
};
}

```

This will give us the outer container. First, we create a copy of our `_rows` argument variable using `Array.from`, which we'd prefixed with an underscore so that we don't have two variables with the same name. We could have technically just done everything with `_rows`, but that would have mutated the data going into our function, which is known as a *side effect*. When writing code using a functional style, you try to avoid side effects where possible, as they make your code harder to debug.

We then pull off the first row for our header, instantiate our table, and return the table, header, and remaining rows as an object.



ES2015 language feature alert! You may have noted the object we returned isn't comprised of the usual key-value pairs--in ES2015, if creating an object that has a key the same as the name of the variable being assigned to that key, you no longer have to specify the key and repeat yourself when supplying the variable. In other words, Babel is smart enough to know that `{"somevariable": somevariable}` is the same as `{someVariable: someVariable}`.

In order to see this work, we need to tell `main.js` to load our new class. Open that up now, and you'll note that all of our code from the preceding chapter is still there. How messy! Let's start by cleaning that up using ES2015 modules. Move all that code into `chapter1/index.js`, leaving `main.js` empty.

You should now have an empty `main.js` and all your code from [Chapter 1, Getting Started with D3, ES2017 and Node.js](#), in its own file. This is how we'll organize our code from now on, and this 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 extensible manner.



If you checked out the `origin/chapter1` branch in Git at any point in time before now, this is where you'll be--with the earlier code in its own file. If I lost you at some point along the way, you can catch up by typing the `git stash save && git checkout origin/chapter1` commands inside the `learning-d3-v4/` directory.

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

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

```
import tableFactory from './chapter2/table-factory';
const header = ['one', 'two', 'three', 'four', 'five', 'six'];
const rows = [
  header,
  ['q', 'w', 'e', 'r', 't', 'y'],
];
const table = tableFactory(rows);
```

Go to the command line and type the following:

```
| $ 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:

```
<script src="/assets/bundle.js"></s
...
<table class="table"></table>
</body>
</html>
```

Woo! A table element!

Let's go back to our `tableFactory` function and add the rest of the table:

```
export default function tableFactory(_rows) {
  const rows = Array.from(_rows);
  const header = rows.shift(); // Remove the first row for the header
  const data = rows; // Everything else is a normal data row

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

  const tableHeader = table.append('thead')
    .append('tr');

  const tableBody = table.append('tbody');

  // Each element in "header" is a string.
  header.forEach(value => {
    tableHeader.append('th')
```

```

        .text(value);
    });

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

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

return {
  table,
  header,
  data,
};
}

```

Now, your table should look like this:

The screenshot shows a browser's developer tools with the 'Elements' tab selected. On the left, there is a preview of a table with six columns labeled 'one' through 'six'. The first column contains the letter 'q', and the last column contains 'y'. On the right, the generated HTML code is displayed:

```


| one | two | three | four | five | six |
|-----|-----|-------|------|------|-----|
| q   | w   | e     | r    | t    | y   |


```

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`) element with each value into the `thead` row. There are then 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 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 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 bunch 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 could even extend your `tableFactory` function to do different things than it does now, without ever altering the code you just wrote.

Those nested `for...each` loops are really ugly though. Thankfully, d3-selection provides a mechanism for adding, updating, and removing data points, which we'll get into really quickly. Instead of tracking where everything is in your project, we just need to tell D3 to manage our data and it will update any attached elements. The key part of this is selections and building functions that intelligently manipulate them.

Selections example

A big part of prototyping JavaScript code is playing around in the browser, using the JavaScript console to manipulate values. Instead of hitting refresh and logging the output a whole bunch, you can do things in the browser and instantly see the result. To do this, you attach functions to the `window` global object.

In `main.js`, replace all its contents with the following:

```
import tableFactory from './chapter2/table-factory';
import * as d3 from 'd3';

window.d3 = d3;
window.tableFactory = tableFactory;
```

This attaches the `tableFactory` function and D3 library to the global `window` object, so we can now use both freely in the console.

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

```
d3.selectAll('.table').remove();
tableFactory([
  [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 that you don't actually need to do this; you can type the above 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 it in the meantime) and adds a new table.

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

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

The text will promptly turn red. Now, let's make everything in the table head bold by chaining two `d3.selectAll()` calls:

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

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

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

The two `d3.selectAll()` calls gave us all the instances of `<td>` in the body, separated by rows, giving us an array of three `NodeList` objects with six elements each: `[NodeList[6], NodeList[6], and NodeList[6]]`. These are in the `_groups` array returned by `d3.selectAll()`, whereas the `<tr>` selections are stored in the `_parents` array. We then used `.style()` to change the color of every selected element.

Using a function instead of a static property gave us the fine-grained control we needed. The function is called with a **datum argument** (we'll discuss more on that later) and an index of the column it's in; that is, the `i` variable. 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 playing around with selections and changing the properties of the elements. We can change the properties of elements on the page.

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

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

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

We selected all the table rows (which returned a new selection object) and then appended a new cell to each using the `.append()` selection. All D3 actions return the current selection, so we can chain actions or assign the new selection to a variable `newCol` 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 a column is present. However, that's kind of pointless, so let's follow the pattern set by other columns:

```
| newCol.text((d, i) => ['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()`. For instance, to get rid of the last row in the table, you'd write something as follows:

```
| d3.selectAll('tbody tr:last-child').remove()
```

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 array, and optionally a key function, telling D3 how to differentiate between various parts of the data.

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

- There is more data than there are selected elements (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
- New in D3 v4 is the `.merge()` function, which allows you to perform operations on selections containing both new and updating elements

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

You must be wondering, 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, not their number. If you shifted an array (that is, removed its first element) and then pushed a new value (or, added a new element to the end of an array), the previous first item would go to the `.exit()` reference and the new addition would go to the `.enter()` reference. This is tremendously powerful, as it allows D3 to track data as it changes in our projects without us having to write a lot of repetitive code.

Datum is the singular of data. You know that `d` argument we usually



pass in the callback functions? That's what it stands for! It's also worth noting that there's a `selection.datum()` method that can accept a single object that you want to apply to all the items in a selection.

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

An HTML visualization example

We will visualize World Health Organization (WHO) global life expectancy data in a simple table.

This is available at <http://apps.who.int/gho/data/node.main.688> by selecting JSON simplified structure. Save it to your `data/` directory (or, if you did `git checkout` at the start of the chapter, will be in your `data/` directory already).

Start off by creating a new file called `index.js` inside `lib/chapter2` and replacing all the code in `main.js` with the following:

```
| import lifeExpectancyTable from './chapter2/index';
| lifeExpectancyTable();
```

Add the following to `chapter2/index.js`:

```
import tableFactory from './table-factory';

export default async function lifeExpectancyTable() {
  const getData = async () => {
    try {
      const response = await fetch('data/who-gho-life-expectancy.json');
      const raw = await response.json();
    } catch (e) {
      console.error(e);
      return undefined;
    }
  };
}
```

We're getting kind of crazy in here using one of the best ES7 features ever, which I want to introduce now because it's fantastic.

If you look where we defined the `lifeExpectancyTable()` and `getData()` functions, you will notice that we put a new keyword, `async`. This marks a function as asynchronous, meaning that it will always return a `Promise`. What's a `Promise`? We get into that later on in [Chapter 3, Shape Primitives of D3](#), but for now, think of it as a special variable that may eventually have a value. This is cool, because that means `async` functions can then use the fancy-schmancy `await` keyword, which means *wait until this variable has a value and then proceed*. We also get to use the new Fetch API, which already returns a promise itself, meaning we can

await *all the things!*

We'll generally use this pattern to load in data throughout the book (though we will look at a few other options in [Chapter 3](#), *Shape Primitives of D3*). Begone, foul-smelling XMLHttpRequest of the previous chapter!

With data in hand, we can start constructing our epic Table... Of Life! (-expectancy values).

Add the following after const raw and before the catch block:

```
return raw.fact.filter(d => d.dim.GHO === 'Life expectancy at birth (years)'  
  && d.dim.SEX === 'Both sexes' && d.dim.YEAR === '2014')  
  .map(d => [  
    d.dim.COUNTRY,  
    d.Value,  
  ]);
```

This changes the bizarre format used by the WHO into something much simpler, and gives us only the *life expectancy at birth* values, for both sexes, from 2014. Time to use our brand new `tableBuilder`.

Add the following at the end of `lifeExpectancyTable`, before the closing curly bracket:

```
const data = await getData();  
data.unshift(['Country', 'Life expectancy (years from birth)']);  
return tableFactory(data);
```

This adds a header row to our table and passes it in to our `tableFactory` function. Your web browser should now have a really dull, unsorted table:

Country	Life expectancy (from birth)
Comoros	63.2
Cuba	79.0
Cyprus	80.3
Burundi	59.1
Bosnia and Herzegovina	77.2
Switzerland	83.2
Australia	82.7
Belgium	80.9
Bahrain	76.8
Bolivia (Plurinational State of)	70.4
Central African Republic	50.8
Angola	51.7
Cabo Verde	73.0
Austria	81.4
Chile	80.3
Cote d'Ivoire	52.8

This is super gross, though the whole point of tables is being able to rapidly look up information, which this definitely does not allow you to do at present.

Let's try sorting it alphabetically. To do this, however, we need to update our `tableFactory()` function to be a little bit smarter. Let's try using some data joins like those we learned above.

Rewrite your `tableFactory()` function to resemble the following:

```
export default function tableFactory (_rows) {
  const rows = Array.from(_rows);
  const header = rows.shift(); // Remove the first element for the header

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

  table.append('thead')
    .append('tr')
    .selectAll('td')
    .data(header)
    .enter()
    .append('th')
    .text(d => d);

  table.append('tbody')
    .selectAll('tr')
    .data(rows)
    .enter()
    .append('tr')
    .selectAll('td')
    .data(d => d)
    .enter()
    .append('td')
    .text(d => d);

  return {
    table,
    header,
    rows,
  };
}
```

Here, we use `selection.data` to create our table rows and all of our cells. Since D3 is now tracking all of our data and how it's attached to things on the screen, we can manipulate the data and have our table reflect it.

Change the last line of `lifeExpectancyTable` to the following:

```
return tableFactory(data).table
  .selectAll('tr')
  .sort();
```

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. As 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 (arguments `a` and `b`) and must decide how to order them, returning `-1` for `a` being less than `b`, `0` for both `a` and `b` being equal, and `1` for `a` being greater than `b`. If `a` and `b` are both numbers, you can merely subtract `b` from `a` to get ascending order (or vice versa to get descending). You can also use the `d3ascending` and `d3descending` comparators of D3, which saves some typing and has the added benefit of being much more immediately apparent about what you're trying to accomplish.

Think about somebody looking at our table in a newspaper or magazine, though. What is the reader more likely to be interested in? Finding the average life expectancy of a particular country, or finding which countries have the highest or lowest life expectancy? Particularly, in news media, readers tend to be concerned with the biggest and smallest numbers--who's the fastest, what's the biggest, and which company made the most money? In this instance, it probably makes more sense to sort by life expectancy, which we can do by replacing our `.sort()` line with the following:

```
| .filter(i => i)
| .sort(([countryA, yearsA], [countryB, yearsB]) => yearsA - yearsB);
```

There's some funky ES2015 stuff going on here, but it's really pretty simple. First, we filter it by *identity* to ensure that we don't have any undefined rows. Then, we write a sorting comparator function that takes two values--in our case, we know each datum is an array containing the country name and its average life expectancy. We use a new ES2015 feature called *array destructuring assignment* to assign the first element of our first argument to `countryA`, our first element of our second argument to `countryB`, our second element of our first argument to `yearsA`, and so on. We then subtract `yearsB` from `yearsA` to sort our values in an ascending order. If we wanted to sort in a descending order, we'd merely subtract `yearsA` from `yearsB` instead.

Boom! Awesome table, ahoy!:

Country	Life expectancy (years from birth)
Sierra Leone	48.1
Central African Republic	50.8
Angola	51.7
Lesotho	52.1
Chad	52.6
Cote d'Ivoire	52.8
Nigeria	53.6
Somalia	54.3
South Sudan	56.6
Mozambique	56.7
Cameroon	56.7
Malawi	57.6
Mali	57.8
Equatorial Guinea	57.9
Guinea	58.1
Liberia	58.1
Guinea-Bissau	58.4
Swaziland	58.4
Burundi	59.1
Zimbabwe	59.2
Burkina Faso	59.3
Democratic Republic of the Congo	59.3
Benin	59.7
Togo	59.7
Afghanistan	59.9
United Republic of Tanzania	60.7
Gambia	60.8
Zambia	61.1
Niger	61.4
Uganda	61.5
South Africa	62.0

Scalable Vector Graphics

Scalable Vector Graphics (SVG) is a vector graphics format that describes images with XML. It has been around since 1999 and is supported by all major browsers nowadays.

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 (for example, lines and circles) defined in the standard. The format itself represents shapes with XML elements and attributes. Since it's an XML-style standard like HTML, quite a lot of what you may already know about HTML also applies to SVG.

As such, SVG code is just a bunch of text 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, such as Preview or Adobe Illustrator.

Another consequence is that browsers can consider SVG to be a normal part of the document. You can use CSS for styling, listen for mouse events on specific shapes, and even move things around using animation. Owing to this, SVG is one of the most powerful tools in the web data visualization arsenal.

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.

Most of what we'll be doing in the rest of this book uses SVG. Pay attention, because this is all going to become very familiar, very quickly.

There's a bunch of stuff that is common to most D3 charts, and we're going to start off by saving ourselves a boatload of time by creating a new factory function that sets everything up for us all. Create a new folder called `common/` inside `lib/` and create a new file called `index.js` in that folder. Add the following to the file to `import d3: import * as d3 from 'd3';`

We will set up a prototype that we can override later if we need to:

```
const protoChart = {  
  width: window.innerWidth,  
  height: window.innerHeight,  
  margin: {  
    left: 10,  
    right: 10,  
    top: 10,  
    bottom: 10,  
  },  
};
```

This is just a plain old object with a bunch of values we'll use later on--consistent 10 pixel margin on each side, with width and height set to the interior dimensions of the browser window.

Next, we will create a factory that returns a basic chart, using the values used in the preceding code and allowing them to be overridden, as necessary: `export default function chartFactory(opts, proto = protoChart) { const chart = Object.assign({}, proto, opts);`

```
chart.svg = d3.select('body')  
.append('svg')  
.attr('id', chart.id || 'chart')
```

```
.attr('width', chart.width - chart.margin.right)
.attr('height', chart.height - chart.margin.bottom);

chart.container = chart.svg.append('g')
.attr('id', 'container')
.attr('transform', &grave;translate(${chart.margin.left},
${chart.margin.top})&grave;);

return chart;
}
```

Here, we assign our `protochart` object from before as the default value of the `proto` function argument. In this way, we can either override individual properties in the first argument or supply a whole new prototype, if we need to do something really funky. We then use `d3-selection` to add an SVG element to the page body, use our `width`, `height`, and `margin` values to size that appropriately and get bottom/right margins, and then add a group element `g` that we shift right and down in order to get our top/left margins.

It's simple, but it will do a lot of heavy lifting for us over the next few chapters.

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:

- Straight lines (a path with two points)
- Rectangles (a path with four points and right angles)
- Circles (a round path)
- Ellipses (an oblong path)
- Polylines (a path comprising straight lines)
- Polygons (a path comprising straight lines that closes in on itself)
- Text (the only one that isn't a path)

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

Text

As mentioned, text is the only SVG shape that isn't really a path. Let's look at it first, so the rest of this chapter is about shapes. Add the following to the top of `chapter2/index.js`:

```
| import * as d3 from 'd3';
| import chartFactory from '../common/index';
```

Then, create a new function below our trusty old `lifeExpectancyTable` function:

```
| export async function renderSVGStuff() {
|   const chart = chartFactory();
|   const text = chart.container.append('text')
|     .text("Ceci n'est pas un trajet!")
|     .attr('x', 50)
|     .attr('y', 200)
|     .attr('text-anchor', 'start');
| }
```

We took our `chartFactory` function and used it to create a new chart. We then appended a `text` element to the `container` property, which corresponds to our group element with all the margins. Then, we defined its actual text and added some attributes to position the text at the `(x, y)` point while anchoring the text at the start of a line.

The `text-anchor` attribute defines the horizontal positioning of the rendered text in relation to the anchor point defined by `(x, y)`. The positions it understands are the `start`, the `middle`, and the `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 we adjust the text margin and baseline relative to the font size because it understands the `em` unit.

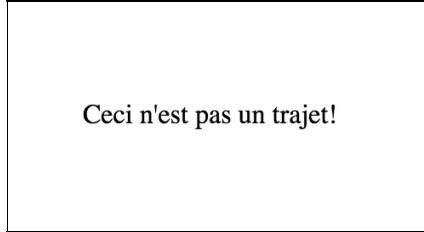
We now need to go back to `lib/index.js` and import our new function. Replace the first `import` statement in `lib/main.js` with this:

```
| import './chapter2/index';
```

Then, add this at the end of `chapter2/index.js`:

```
| renderSVGStuff();
```

Then, check it out:



Ceci n'est pas un trajet!

Since we instantiated our function in `chapter2/index.js`, all we need to do is import it in `lib/main.js`, and it will run. The rest of these SVG examples will simply be appended to `rendersvgstuff` so that we can focus on using D3 and not just write JavaScript enclosures all day. We're not really making anything useful in this chapter, you can disregard what I said earlier about modularity until the end of this section.

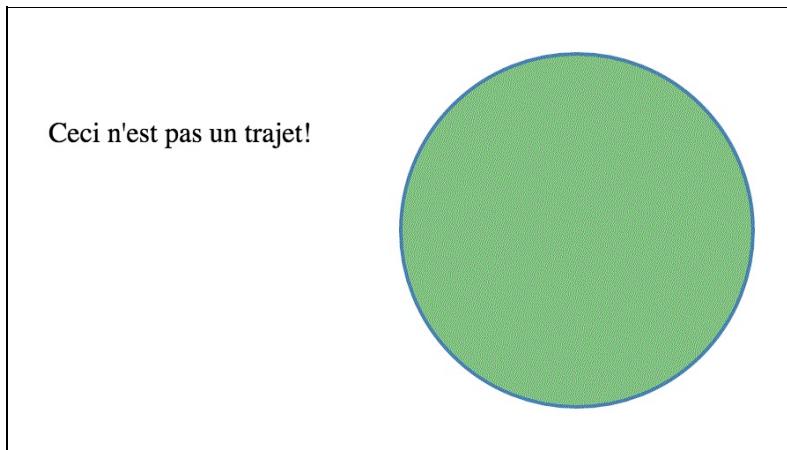
Shapes

Now that `text` is out of the way, let's look at something much more interesting--shape primitives, which we use to create everything from the bars in a bar chart to complex renders of geographic spaces.

Let's start by talking about things where you need only one point:

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



Mathematically speaking, a circle is just a special form of ellipse. You can make a less-round circle by setting an ellipse element's `rx` and `ry` attributes; it otherwise works the same:

```
const ellipses = chart.container.append('ellipse')
  .attr('cx', 350)
  .attr('cy', 250)
  .attr('rx', 150)
  .attr('ry', 70)
  .attr('fill', 'green')
  .attr('fill-opacity', 0.3)
  .attr('stroke', 'steelblue')
  .attr('stroke-width', 0.7);
```

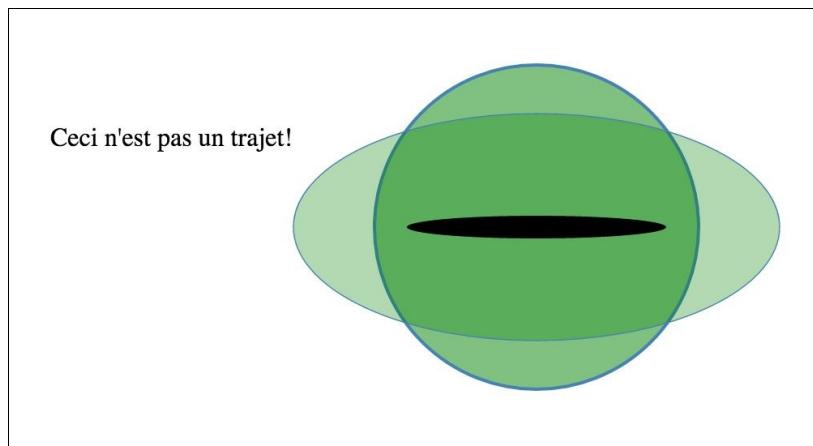
We added an `ellipse` element and defined some well-known attributes. The ellipse shape needs a central point (`cx`, `cy`) and the two radii, `rx` and `ry`. Setting a low fill-opacity attribute makes the circle visible under the ellipse:

Let's add another one using the following code:

```
chart.container.append('ellipse')
  .attr('cx', 350)
  .attr('cy', 250)
  .attr('rx', 80)
  .attr('ry', 7)
```

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

Smashing! We've made some kind of weird green eye thing!:



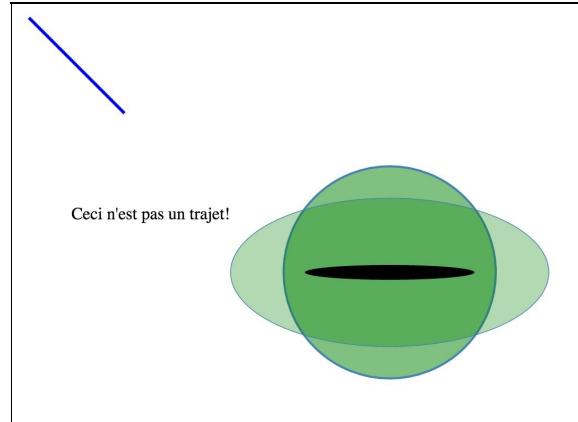
However, we want more. We want a diagonal blue line. Clearly, that's all that's needed to tie this amazing piece of art together.

To draw a straight line, use the following code:

```
const line = chart.container.append('line')
  .attr('x1', 10)
  .attr('y1', 10)
  .attr('x2', 100)
  .attr('y2', 100)
  .attr('stroke', 'blue')
  .attr('stroke-width', 3);
```

As before, we took our chart container element, appended a line, and defined some attributes. We're now diving into creating elements with multiple points--a line is drawn between two, (x_1, y_1) and (x_2, y_2) . To make the line visible, we have to define the `stroke` color and `stroke-width` attributes as well--otherwise it'd

simply be a black line with no stroke and no fill, and thus not able to be seen:



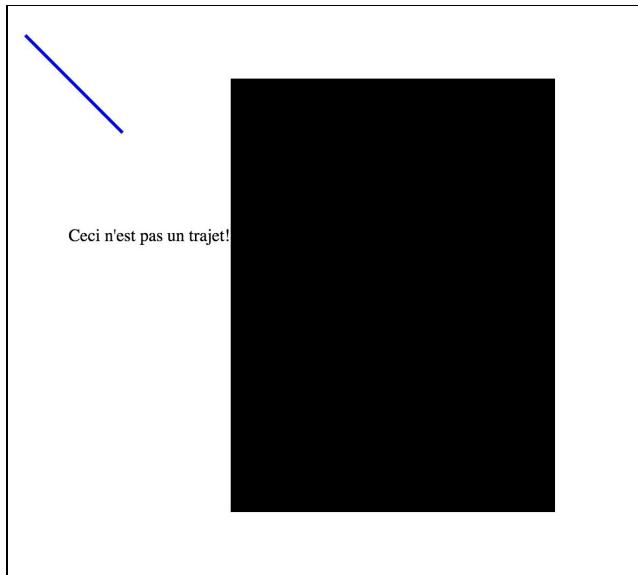
Our line points downward, even though y_2 is bigger than y_1 . That's because the origin in most image formats lies in the top-left corner. This means that $(x=0, y=0)$ defines the top-left corner of the image.

Hm. Still missing something... Let's move on to elements with four vertices!

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

```
const rect = chart.container.append('rect')
  .attr('x', 200)
  .attr('y', 50)
  .attr('width', 300)
  .attr('height', 400);
```

We appended a `rect` element to the `container` element and defined its core attributes. A rectangle is defined by its upper-left corner (`x`, `y`), `width`, and `height`. Our image now looks as follows:



We have a rather-large black rectangle. Shapes will render like this if you don't define the fill and stroke properties--by default, SVG shapes are borderless and with a 100% opacity black fill. Not only that, but it's blocking our weird green eye thing!

SVG stacking is defined in the order an element appears in the markup, and SVG doesn't have anything like z-index in HTML to arbitrarily change this order.

To place the rectangle behind or before the eye, we need to modify the following line:

```
| const rect = chart.container.append('rect')
```

Modify it and use `insert()` instead of `append()`:

```
| const rect = chart.container.insert('rect', 'circle')
```

The first argument is the type of element to be inserted, with the second argument a CSS selector for an item to insert before. In this case, we want to insert behind our earlier circle, so we supply that. Any CSS selector works though--we could have also put `:first-child` at the top of the stack, so behind everything (including our first text element).

Another way we could accomplish this, which is new in D3 v4, is to use `selection.raise()` and `selection.lower()`. `selection.raise()` will



recreate the selection as the last/bottom element in their parent container, and `selection.lower()` will recreate it at the top of its parent. This is really useful if you want to create a series of elements at the bottom of an SVG group, then move the elements to the top of the group after somebody clicks or taps on them.

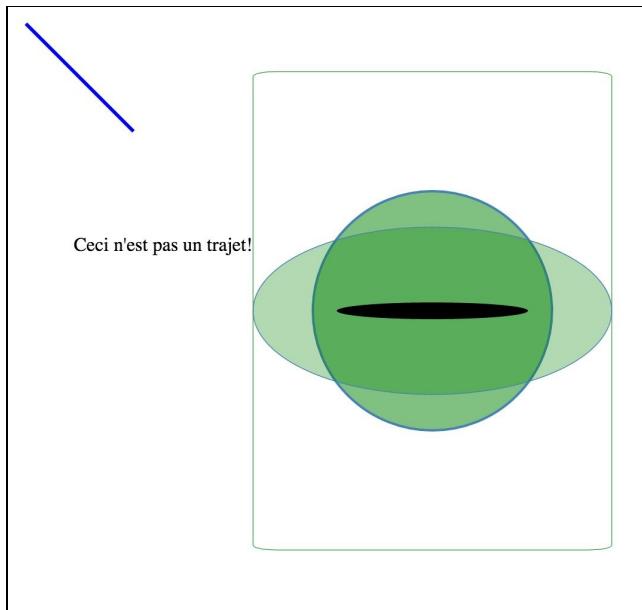
Let's set fill and stroke by defining three more properties like this:

```
| rect.attr('stroke', 'green')  
|   .attr('stroke-width', 0.5)  
|   .attr('fill', 'white')
```

Then, mess it up a little bit by rounding the corners an arbitrary amount:

```
|   .attr('rx', 20)  
|   .attr('ry', 4);
```

The output will look somewhat like this:



That's kind of interesting, I guess? 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.

The generated SVG is in an XML form, as follows; you can see the same by right-clicking the image and navigating to Inspect Element, which will select the element in Developer Tools:

```

<svg id="chart" width="801" height="726">
  <g id="container" transform="translate(10, 10)">
    <rect x="200" y="50" width="300" height="400"/>
    <text x="50" y="200" text-anchor="start">
      Ceci n'est pas un trajet!</text>
    <circle cx="350" cy="250" r="100" fill="green"
      stroke="steelblue" fill-opacity="0.5" 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="7"/>
    </ellipse>
    <line x1="10" y1="10" x2="100" y2="100" stroke="blue"
      stroke-width="3"/>
  </g>
</svg>

```

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

However, 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 is really useful when trying to debug problems. With something like Canvas (which we'll touch upon briefly in [Chapter 8, D3 on the server with Canvas, Koa 2 and Node.js](#)), all you see in the rendered output is a single element, which can be output as a base64-encoded string of indecipherable noise. This is both SVG's biggest strength and also its Achilles' heel--while it's incredibly easy to work with and debug, having to keep a complex tree of elements in memory really tends to slow down the browser after a certain point (at present, the general wisdom is that it's time to start thinking about Canvas when your chart renders over a thousand SVG elements).

We've looked at 1, 2, 3, and 4-vertex shapes now (circles, lines, ellipses, and rectangles, respectively). I mentioned earlier that polylines and polygons are also basic SVG elements--however, I will not cover them just yet. Although you can render them using D3 selections like the above, they're far easier to work with if using some of the path generators that come with D3. We'll cover those in just a second.

Transformations

Before jumping into more complicated things, let's take a look at transformations. For now, think of transformations as manipulations of whole SVG shapes.

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 they can be defined as matrix multiplications, making them very efficient to compute--it's much more performant for your graphics card to animate a shape moving from left to right using a transformation than it is to recalculate the position of each point in a shape and modify the shape directly.

However, unless your brain is made out of linear algebra, using transformations as matrices can get very tricky. SVG helps 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 that 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 note this in the following examples.

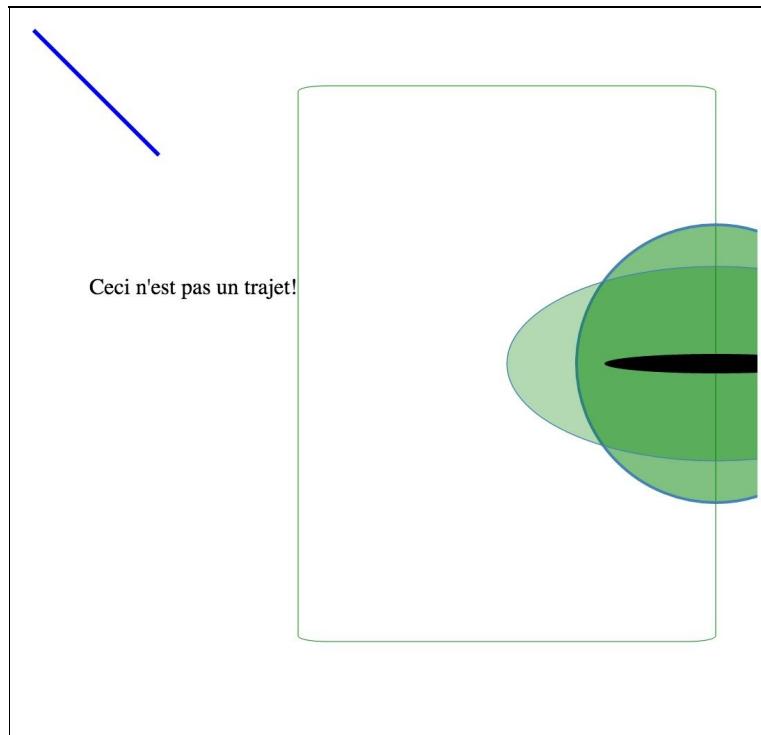
Let's move our eye to the edge of the rectangle:

```
| chart.container.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 note that new transformations are applied according to the original state of our shape. That's because there can only be one `transform` attribute per shape.

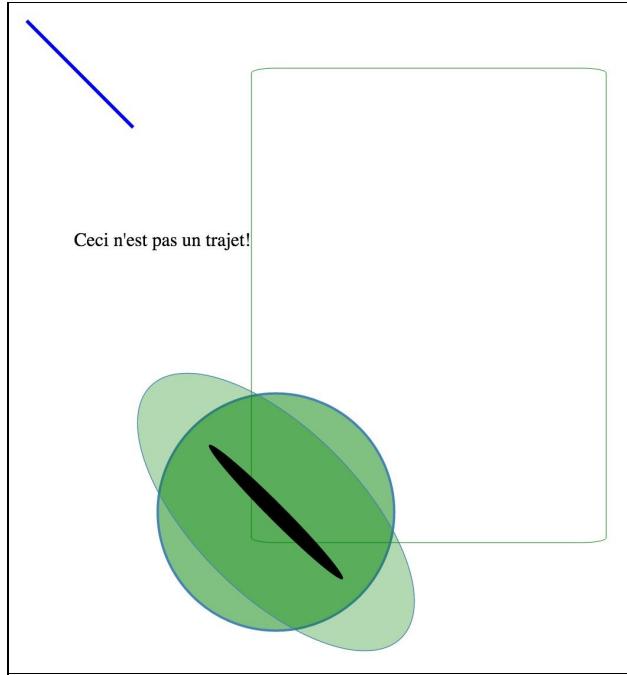
Our picture looks as follows:



Let's rotate the eye by 45 degrees:

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

The output will look something like this:

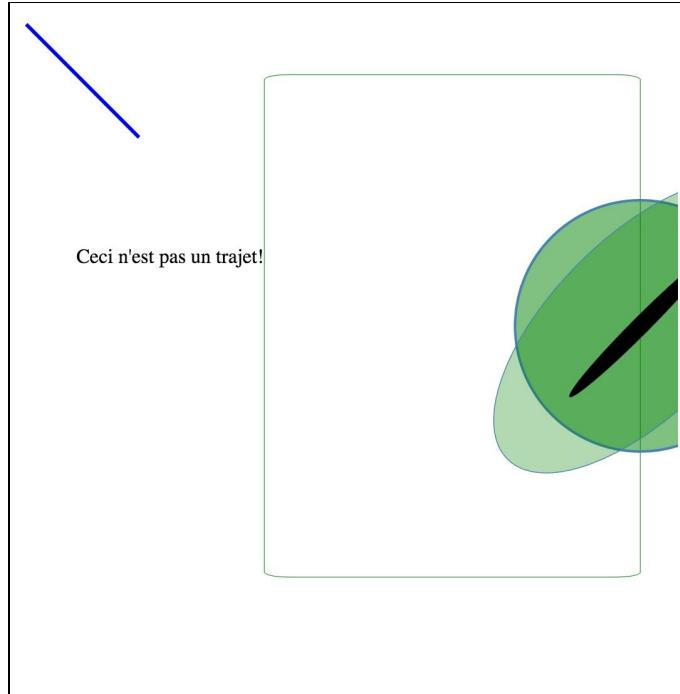


That's not what we wanted at all!

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

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

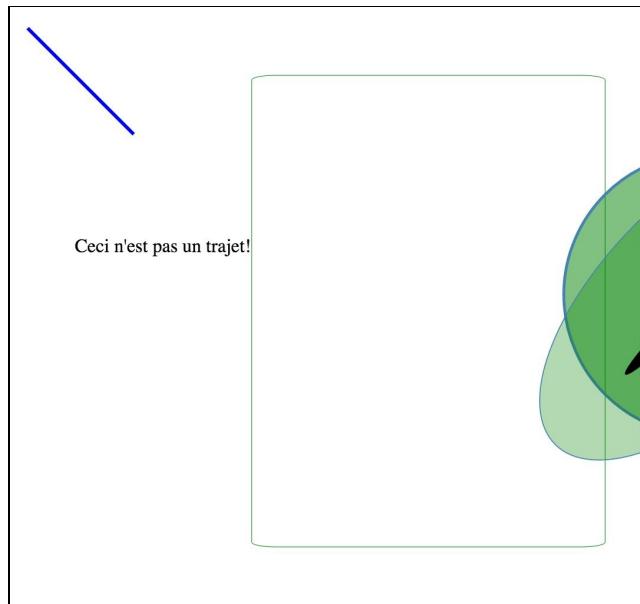
By adding two more arguments to `rotate()`, we defined the rotation axis and achieved the desired result:



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

```
chart.container.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:

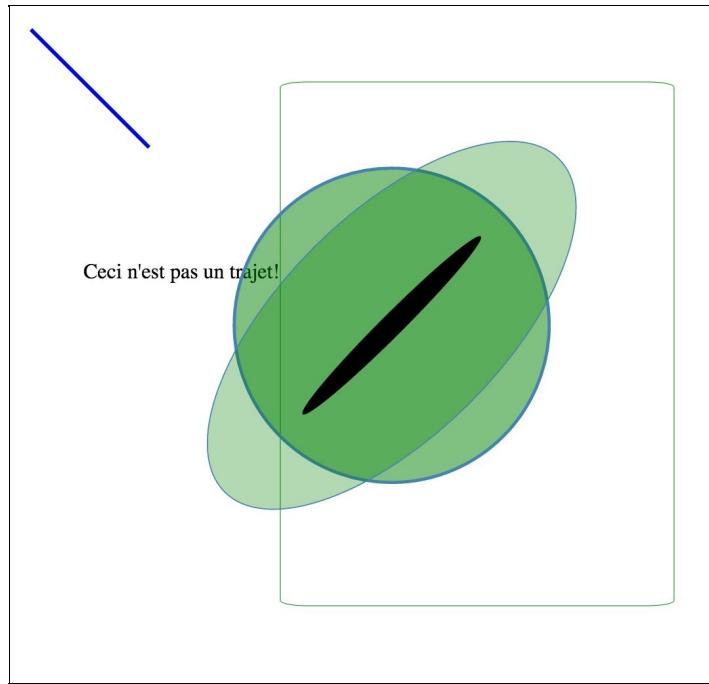


Once again, we pushed the position of the eye because scaling is anchored at the zeroth point of the whole image. We have to use another translate to move it back. However, the coordinate system we're working on is now rotated by 45 degrees and scaled. This makes things tricky. We need to translate between 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 cosine and sine at an angle of 45.

However, that's just messy. The number looks funny, and we worked way too much for something so simple. We're also falling off the page. Let's change the order of operations instead:

```
chart.container.selectAll('ellipse, circle')
  .attr('transform', &grave;translate(150, 0)
    scale(1.2)
    translate(-250, 0)
    rotate(-45, ${350 1.2}, ${250 1.2})&grave;);
```

Much better! We got exactly what we wanted:



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

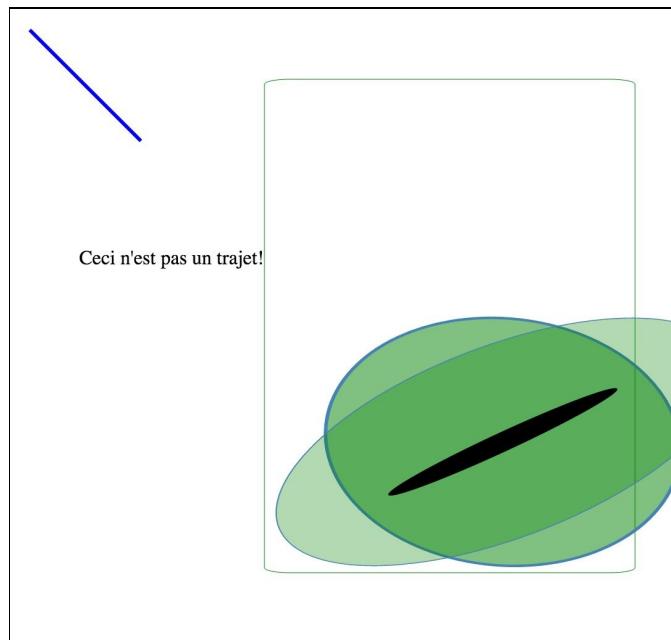
First, we translate to our familiar position and then scale by 1.2, pushing the eye out of position. We fix this by translating back to the left by 250 pixels, and then

finally 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:

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

We just bolted `skewY(20)` on 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 multiplication. In fact, you can define any transformation you want with the `matrix()` function, which the other methods are really just shortcuts for. To master transformations, take a look at what kind of matrix produces each of the preceding effects via the W3C specification that is available at <http://www.w3.org/TR/SVG/coords.html#EstablishingANewUserSpace>.

CSS

Cascading Stylesheets (CSS) have been with us since 1996, making them one of the oldest staples of the Web, even though they only reached widespread popularity 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 weird-looking SVG stuff.

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

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

That describes CSS better than I can--you use selectors to modify properties using values. Although there's a bit more to it, particularly in terms of how properties cascade down the DOM tree, the above is pretty much it.

We've been using selectors all this time. A selector is any string that describes one or more elements in a DOM tree.



Although you can get fancy with selectors, there's been a lot written about how to intelligently name classes recently so that most CSS is just simply just declarative. One such approach is BEM, standing for Block Element Modifier, wherein you write classes in the .block_element-modifier format, so that they tend to look like .some-module_submit-button--large. This modular approach reduces CSS conflicts and having to dig through the Developer Tools Element pane trying to figure out why something is styled a particular way. For more information, visit <http://getbem.com/>.

To review:

- `.path`: Selects all the `<path>` element
- `.axis`: Selects all the elements with a `class="axis"` attribute
- `.axis line`: Selects all the `<line>` elements that are children of the `class="axis"`

elements

- `.axis, line`: Selects all the `class="axis"` and `<line>` elements

D3 selections are a subset of CSS selectors; so, any selection you can make with `d3.selectAll`, you can specify using CSS.

We can invoke CSS with D3 in three ways:

- Defining a class attribute with the `.attr()` method, which can be brittle
- Using the `.classed()` method, the preferred way to define classes
- Defining styling directly with the `.style()` method

Let's throw readability to the wind for a second and make our axes totally trippy.

Go into `index.css` and add the following at the end:

```
.trippy {  
  animation: wheee 3s infinite;  
  fill: green !important;  
}  
  
@keyframes wheee {  
  0%   {stroke: red;}  
  25%  {stroke: yellow;}  
  50%  {stroke: blue;}  
  75%  {stroke: green;}  
  100% {stroke: red;}  
}
```



We use `!important` to override the fill color of our axes (which--if you read the last infobox, you'll remember--is defined in the `style` attribute, meaning that it generally takes precedence over CSS, unless `!important` is added to the CSS declaration), and then set the `stroke` (which is not defined anywhere yet) to a series of colors that cycles every three seconds on an infinity loop. This animation is defined using keyframes that indicate the various stages of the animation. I assure you, this shall not be the last time we get `.trippy` in this book!

Now, ensure that the following line is in `lib/main.js` to load the CSS into your HTML file; you can also use the `<link>` tags in your HTML, but again, that's boring:

```
| import '../styles/index.css';
```

Now, we amend the drawing loop from earlier to look as follows:

```
axes.forEach((axis, i) =>
  chart.container.append('g')
    .data(d3.range(0, amount))
    .classed('trippy', i % 2)
    .attr('transform', &grave;translate(0, ${i * 50} + chart.margin.top}&grave;)
    .call(axis)
);
```

We'll have none of that foolishness with specifying the same values five times in a row. Using the `.classed()` function, we add the `.trippy` class to every second axis. `.classed()` adds the specified class if the second argument is true or missing, and removes it otherwise. All the numbers on the second and fourth axes now change color, and it's amazing and totally life reaffirming.

I hope that you're getting excited, because we're *finally* through this terribly boring review chapter, and we'll soon move on to all kinds of supreme awesomeness! We'll be talking more about animation in [Chapter 5, *Defining the User Experience - Animation and Interaction*.](#)

Summary

That 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 to get through, feel free to come back later if you need a refresher.

We've gone through DOM manipulation and looked at SVG in great detail, covering everything from drawing shapes manually to using transformations. Finally, we looked at CSS as an 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 to draw anything you can think of.[footnote]

Shape Primitives of D3

In this chapter, we will draw some shapes like we did in the last chapter, but we will use D3's built-in generators to simplify things and make life easier for us. Most of these are included in the `d3-shape` package.

Using paths

In [Chapter 2](#), *A Primer on DOM, SVG, and CSS*, we created a few sundry shapes with a maximum of four points. Although those will get you pretty far drawing basic charts, for complex shapes we'll need to draw 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.

Wait: what about `polyline` and `polygon`? While those are also multipoint SVG primitives, they're really pretty much identical to `path`.

Most of the path's magic stems from the `d` attribute; it uses a mini language (in programming terms, a *domain-specific language*, or **DSL**) of three basic commands:

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

To create a path, we might write something like the following:

Create a new folder in `lib/` called `chapter3/` and put an `index.js` in it. In `lib/main.js`, change the following line:

```
| import './chapter2/index';
```

Change it to read:

```
| import './chapter3/index';
```

Then, create a new function like so:

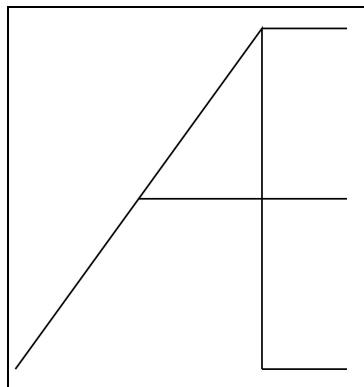
```
export function yayPaths() {
  const path = chart.container.append('path')
    .attr('d', 'M 10 500 L 300 100 L 300 500 M 300 100 l 100 0 M 155 300 l 245 0 M 300 50
      .attr('stroke', 'black')
      .attr('stroke-width', 2)
      .attr('fill', 'transparent');
}
yayPaths();
```

We appended a new element to our SVG and then defined some attributes. The interesting bit is the `d` attribute:

```
| M 10 500 L 300 100 L 300 500 M 300 100 l 100 0 M 155 300 l 245 0 M 300 500 l 100 0
```

What's going on here? SVG path directions always start with a capital `M`, meaning *move the pen here*--in this case, $(10, 500)$. Then, we draw a line using capital `L` to $(300, 100)$, then again to $(300, 500)$. We then move the pen to $(300, 100)$, and draw a line using a lowercase `l`--lowercase letters mean to use relative values; in this case, draw 100 pixels horizontally to the right so that the pen rests at $(400, 100)$. We then move the pen to $(155, 300)$, draw a 245 pixel-length line to the right, then move to $(300, 500)$ and draw another 100-pixel line.

Reload your browser to see what we drew. Hmm, this looks familiar:



The power of paths doesn't stop there, though. Commands beyond the `M`, `L`, `l` combination give us tools to create curves and arcs. As you now know, creating complex shapes by hand is incredibly tedious.

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

Let's get rid of that glorious diphthong you just drew by replacing all the code inside `chapter3.js` with the following:

```
import * as d3 from 'd3';
import chartFactory from '../common/index';
export function yayPaths() {
  const chart = chartFactory();
}
yayPaths();
```

This instantiates our new function, which generates a new chart using our factory; we've done this before.

To start things off, we'll draw the humble sine function. First, we need some data, which we'll generate using the built-in JavaScript sine function, `Math.sin`:

```
const 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 2-length array representing the maxima and minima of the curve. You might remember from your math class that sine starts at $(0, 0)$, then goes to $(\frac{\pi}{2}, 1)$, $(\pi, 0)$, $(\frac{3\pi}{2}, -1)$, and so on.

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

Path generators are really the meat of D3's magic. They're a big topic, and we will discuss them in more detail in [Chapter 6, Hierarchical Layouts of D3](#). They are essentially a function that takes some data (joined to elements) and produces 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.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.



In D3 v3, these generators were found under the `d3.svg` namespace.

In v4, they're all in the top-level namespace, or in `d3-shape`. If you get errors, such as `Cannot read property 'line' of undefined` when replicating an example you found online, the example quite possibly is still using D3 v3.

We begin by defining two scales. If you remember from [Chapter 1, Getting Started with D3, ES2017, and Node.js](#), scales are functions that map from a domain to a range; we'll talk more about them in the next chapter:

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

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

With those in place, we now use them to define our path generator:

```
const line = d3.line()
  .x(d => x(d[0]))
  .y(d => y(d[1]))
```

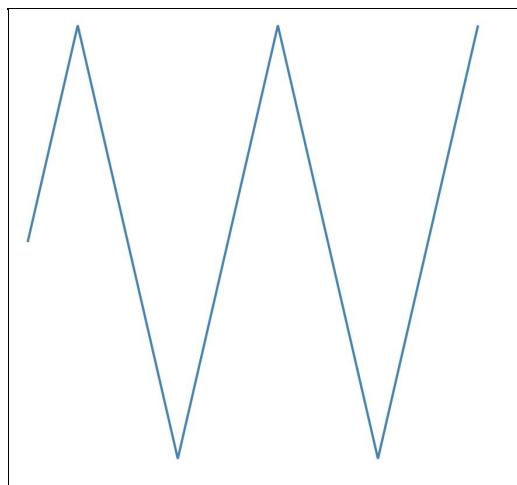
It is just a matter of taking the basic line generator and attaching some accessors to it. We told the generator to use our `x` scale on the first element and the `y` scale on the second element of every array. By default, it assumes our dataset is a collection of arrays defining points directly so that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

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

```
const g = chart.container.append('g');
g.append('path')
  .datum(sine)
  .attr('d', line)
  .attr('stroke', 'steelblue')
  .attr('stroke-width', 2)
  .attr('fill', 'none');
```

This creates a new group element, to which it appends a path, and adds 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 now looks as follows:



If you look at the resulting code in Chrome Dev Tools, you see something resembling the following gibberish:

```
| <path d="M0,185L42.833333333333,0L85.6666666666666,184.999999999997L128.5,370L171.
```

Although it's shorter than our JavaScript, our JavaScript is much more readable and much easier to reason about.

Our sine function is very jagged, nothing like what our math teachers used to draw during high school. We can make it better using interpolation.

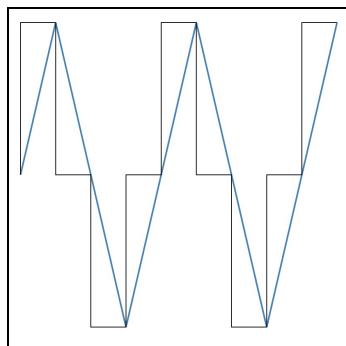
Interpolation is the act of guessing where unspecified points of a line should appear, considering the points we do know. By default, we're using the linear interpolator that just draws straight lines between points.

Our linear interpolator is kind of boring. Let's fix that!

Add the following to our function:

```
g.append('path')
  .datum(sine)
  .attr('d', line.curve(d3.curveStepBefore))
  .attr('stroke', 'black')
  .attr('stroke-width', 1)
  .attr('fill', 'none');
```

It is the same code as before, but we used the `d3.curveStepBefore` interpolator and changed the styling to produce this:



All we've done is set the curve factory on our generator to one of D3's built-in interpolators. The `.curve` method of our line generator takes an interpolation function (here, we've supplied the *step before* curve method built into D3), and returns the line generator. This returned line generator is then supplied to the `d` function of the path, so we get a different sort of shape than we did before.



This is a very good example of the style of functional programming that D3 embraces, and you'll see it frequently throughout the book.

D3 offers 18 line interpolators in total, which are all listed with an example in the official wiki page for d3-shape, <https://github.com/d3/d3-shape/blob/master/README.md#curves>.

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



New in D3 v4, `line.interpolate` has been renamed `line.curve`, and instead of supplying it a string, you supply it a function attached to either the `d3` namespace or the `d3-shape` module, depending on which



you're using. See the preceding URL for all available built-in interpolation functions.

Area

An area is the colored part between two lines, think of an area chart, which shows you the space beneath a curve.

Similar to how we define a line, we create 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`, because our area has a top and a bottom. Often the bottom `y` accessor will just be zero because the `y`-axis frequently starts at zero. This isn't always the case, however, and D3 gives you the power to handle any particular case you might run into.

We'll compare different generators side by side. Start by adding a new group element, which we'll render inside the same SVG element:

```
const g2 = chart.container.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:

```
const area = d3.area()
  .x(d => x(d[0]))
  .y0(chart.height / 2)
  .y1(d => y(d[1]))
  .curve(d3.curveBasis);

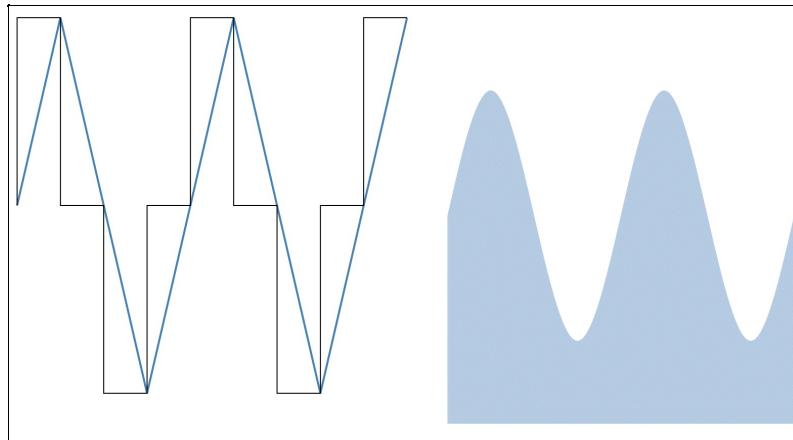
g2.append('path')
  .datum(sine)
  .attr('d', area)
  .attr('fill', 'steelblue')
  .attr('fill-opacity', 0.4);
```

We took the `d3.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.



D3 v4 change: Yep, you guessed it--`d3.svg.area` is now just simply `d3.area`. It's part of the `d3-shapes` microlib.

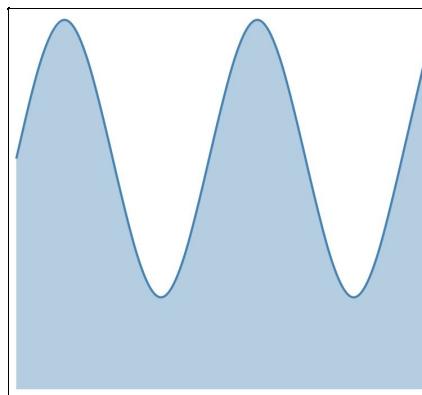
To draw the bottom edge, we defined y_0 as the bottom of our graph and produced a colored sine approximation:



One often uses an area in conjunction with a line. We can either give the path a border using its stroke property, or we could use our line generator to render another line atop the top edge of our area. Let's try the latter:

```
g2.append('path')
.datum(sine)
.attr('d', line.curve(d3.curveBasis))
.attr('stroke', 'steelblue')
.attr('stroke-width', 2)
.attr('fill', 'none');
```

Reusing the same line generator as before, we merely change the interpolator for the area:



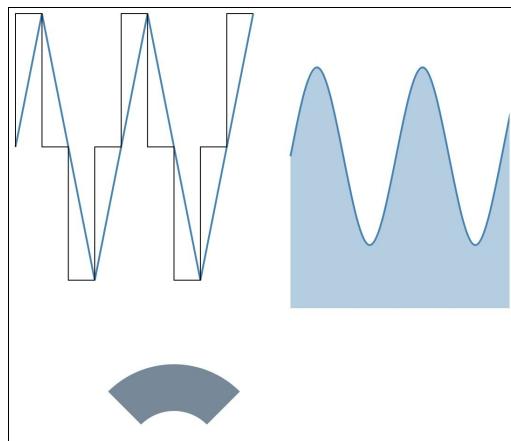
Arc

An arc is a circular path with an inner and outer radius, starting 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 the 2-value arrays we've gotten used to:

```
const arc = d3.arc();
const g3 = chart.container.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 could get away with using the default `d3.arc()` generator. Instead of using data, we calculated the angles by hand and also nudged the arc toward the center:



How fascinating!

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



Once again, `d3.svg.arc` in D3 v3 is now `d3.arc` in v4, and is part of the `d3-shapes` package. I think you get the idea at this point-- anything that was in `d3.svg` is now just in the top-level D3 namespace.

Symbol

Sometimes when visualizing data, we need a simple way to mark data points. That's where symbols come in, which are tiny glyphs we can use as an alternative to circles.

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

As always, we start with a path generator:

```
const symbols = d3.symbol()
  .type(d => (d[1] > 0 ? d3.symbolTriangle : d3.symbolDiamond))
  .size((d, i) => (i % 2 ? 0 : 64));
```

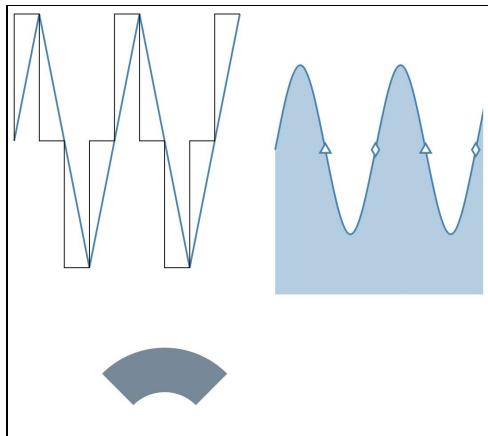
We've given the `d3.symbol()` generator a type accessor, telling it to draw a triangle when the y coordinate is positive and a diamond when not positive. This works because our sine data isn't mathematically perfect due to `Math.PI` not being infinite and due to floating point precision; we get infinitesimal numbers close to zero whose signedness depends on whether the `Math.sin` argument 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. Since 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')
  .attr('stroke-width', 2)
  .attr('fill', 'white')
  .attr('transform', d => `translate(${x(d[0])},${y(d[1])})`);
```

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

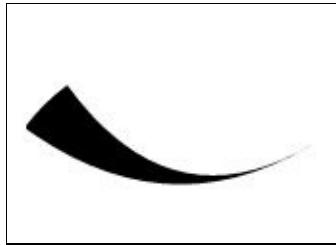


You can see other available symbols by inspecting `d3.symbols` or visiting <https://github.com/d3/d3-shape/blob/master/README.md#symbols>.

Chord/Ribbon

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

Put another way, a basic chord looks similar to half a villain's mustache:



Chords have changed quite a lot since D3 v3. Now in their own package (`d3-chord`), instantiating `d3.chord` creates a new layout, with the actual shapes drawn by `d3.ribbon`.

We just want to draw a ribbon here at the moment, so we're not going to use the entire chord layout. We pass `d3.ribbon()` to create a new ribbon generator, which draws our path:

```
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.ribbon());
```

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

The data itself works fine with the default accessors, so we can just hand it off to `d3.ribbon`. If we need to change the placement of each end of the chord, we can use `ribbon.source()` to define where the chord begins and `ribbon.target()` to set where it ends. Both are fed to another set of accessors, specifying the arc's radius, `startAngle`, and `endAngle`. As with the arc generator, angles are defined using radians.

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

```
| const data = d3.zip(d3.range(0, 12), d3.shuffle(d3.range(0, 12)));
| const colors = ['linen', 'lightsteelblue', 'lightcyan', 'lavender', 'honeydew', 'gainsb
```

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 suffice 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:

```
| const ribbon = d3.ribbon()
|   .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 the arrays define the generator. We're going to divide a circle into sections and connect random pairs with chords.

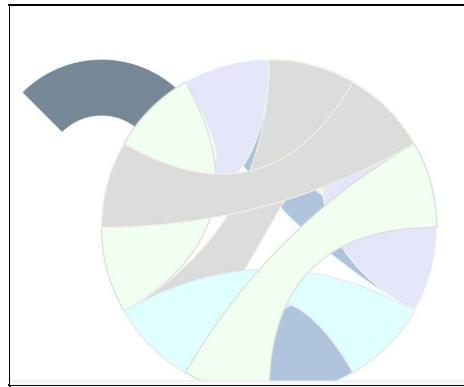
The `.source()` and `.target()` accessors tell us the first item in every pair is the source and the second is the target. For `startAngle`, we remember a full circle is 2π and divide it by the number of sections. Finally, to pick a section, we multiply by the current data. The `endAngle` accessor is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

```
| g3.append('g')
|   .attr('transform', 'translate(300, 200)')
|   .selectAll('path')
|   .data(data)
|   .enter()
|   .append('path')
|   .attr('d', ribbon)
|   .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 data. We use the ribbon generator from earlier to give each ribbon a shape, draw chords from each source to target, and add some color for fun.

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



We'll discuss the chord layout itself in [Chapter 6, Hierarchical Layouts of D3](#).



There used to be a section about `d3.svg.diagonal` here, but that's been removed in D3 v4. Suffice to say that drawing individual diagonals wasn't that useful or easy and all that functionality is wrapped up into the hierarchical chart layouts, part of the d3-hierarchy module. We'll get to that in [Chapter 6, Hierarchical Layouts of D3](#).

Axes

Now that we can make all these different shapes, let's use them to create something actually useful. One way we can do that 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.

*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.*



Also, because we will be using scales a lot from now on, let's do that Scott Murray exercise from Chapter 1, Getting Started with D3, ES2017, and Node.js, again: Input! Domain! Output! Range! Input! Domain! Output! Range!

Honestly, it really never gets old.

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

Let's clear our screens by removing the `yayPaths();` line at the bottom of `chapter3/index.js` and replacing it with the following:

```
export function axisDemos() {
  const chart = chartFactory({
    margin: { top: 30, bottom: 10, left: 50, right: 50 },
  });
  const amount = 200;
}
axisDemos();
```

Here, we override the default margins with some bigger ones to help you see these easier. We also have a maximum value assigned to a constant.

We also need a linear scale:

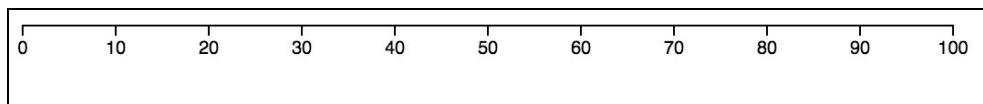
```
const x = d3.scaleLinear()  
  .domain([0, amount])  
  .range([  
    0,  
    chart.width - chart.margin.right - chart.margin.left - 20,  
  ]);
```

We set the maximum output value to the width of our chart (which, if you recall, is the entire width of the window), minus our two 10-pixel margins, minus another 20, because a horizontal axis needs about 20 pixels of padding on the right side to accommodate a three-digit final label.

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

```
const axis = d3.axisBottom()  
  .scale(x);  
  
chart.container.append('g')  
  .data(d3.range(0, amount))  
  .call(axis);
```

We told the `d3.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. It now looks like this:



In D3 v3, you had to style the axes yourself before they looked even borderline decent. D3 v4 now includes a reasonable set of defaults for axes, so you don't have to worry about them as much. In V4, these defaults are known as `style` attributes, so you'll need to use `!important` if you want to override them in CSS.

If you play around with the `amount` variable, you'll note that axes are smart enough to always pick the perfect amount of ticks to fit the space. This is an

incredibly useful feature that allows us to do some cool stuff with animation later on.

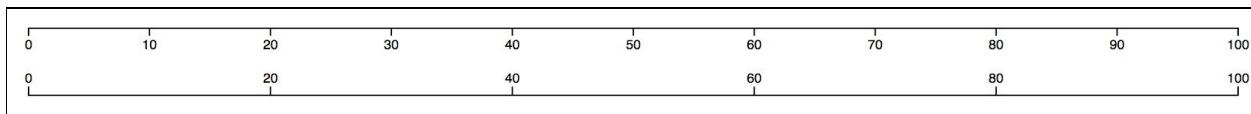
Let's compare what the different settings do to these axes. We will loop through several axes and render the same data.

Replace your call to `chart.container.append('g')` with the following:

```
const axes = [
  d3.axisBottom().scale(x),
  d3.axisTop().scale(x).ticks(5),
];

axes.forEach((axis, i) =>
  chart.container.append('g')
    .data(d3.range(0, amount))
    .attr('transform',
      `translate(0, ${(i * 50) + chart.margin.top})`)
    .call(axis)
);
```

Let's limit it to just two axes for now: one is the plain vanilla version and the other will render with exactly 5 ticks:

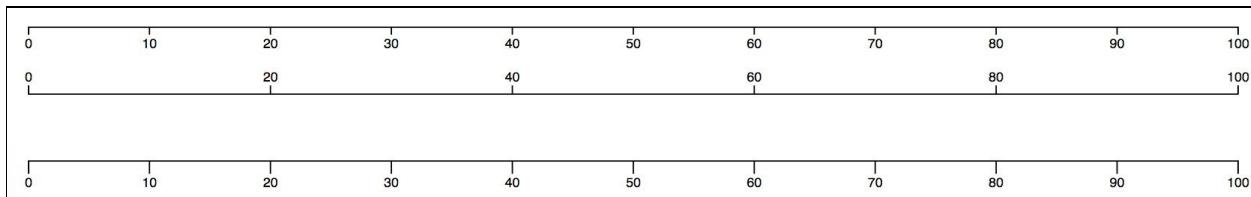


It worked! The axis generator figured out which ticks are best left off and relabeled everything without us doing too much.

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

```
| d3.axisBottom().scale(x).tickSize(10, 5, 10),
```

With `.tickSize()` we can make the minor ticks smaller. The arguments are `major`, `minor`, and `end` tick size:



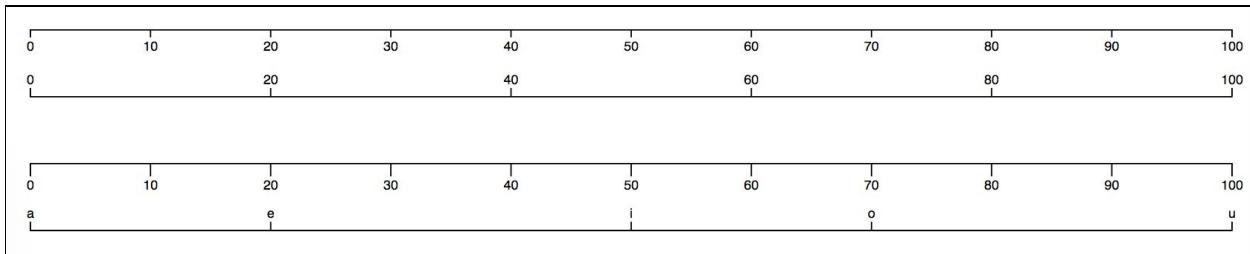
For our final trick, let's define some custom ticks and place them above the axis. We'll add another axis to the array:

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

Two things happen here: `.tickValues()` exactly defines which values should have a tick, and `.tickFormat()` specifies how to render the labels. We set it as `axisTop` to render it above the line.



In D3 v3, you had to set the axis direction using the `axis.orient()` method during creation time of the axis. In D3 v4, you choose `d3.axisTop`, `d3.axisBottom`, `d3.axisLeft`, or `d3.axisRight` to create an axis oriented in a particular way. This emphasizes the fact an axis' orientation can't be changed after it's initially drawn.



Summary

Hey, that wasn't so bad! We're starting to really make use of D3 and draw things a bit more complex than a circle or a rectangle. Just wait until you see the sort of things we make in the upcoming chapters!

We started by drawing a SVG path element manually, using its weird drawing DSL. Then, we drew some lines, made them curvalicious, drew some areas, axes and more--all using the path generators supplied by D3's `d3-shape` package.

We're about to start building some really interesting stuff pretty quickly, but first we need to go over how to transform data into any format you could possibly ever need. Stay tuned for [Chapter 4, *Making Data Useful*.](#)

Making Data Useful

When creating visualizations for the Web, chances are that the format your data comes in will not be the final format you use with D3. We will take a look at making our datasets useful with both D3 and regular JavaScript.

We start with a quick dip 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 write JavaScript in a functional style. Functional programming is a hot topic in JavaScript development right now, for good reason -- it makes your code easier to read, encourages good practices, such as not mutating variables, and leverages one of JavaScript's best features as a language -- the use of *first-class* functions. We'll take a look at what that means in a short while.

We will continue with loading external data in a variety of different ways, take a closer look at scales, and finish with some temporal and geographic data.

Thinking about data functionally

I've mentioned previously that D3 is very *functionally* designed, meaning that it uses some of the idioms JavaScript has adopted from functional programming. Although we can still approach D3 development in a very classical, object-oriented fashion, our lives will be much easier if we start thinking about our code and data with a functional mindset.

The good news is that JavaScript almost counts as a functional language; there are enough features to get the benefits of a 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.



Later on, in Chapter 9, Having Confidence in Your Visualizations, we'll look at TypeScript, which allows compilation of JavaScript using static types, and Tern.JS , which analyzes code in order to improve tooling. These efforts go a great deal toward improving confidence in how data moves through our visualizations, in addition to improving our tooling.

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

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

The biggest consequence is that we don't have to rely on state, which in turn gives us a 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 dataflow, that is, get data as an input, execute a sequence of functions that pass changed data down the chain,

and eventually get a result.

It also emphasizes immutability, whereby functions are written to prevent side-effects and changes to the underlying data as it passes through your code. In [Chapter 2, A Primer on DOM, SVG, and CSS](#), we used `Array.from()` to make a copy of an argument so that our function wouldn't mutate the data passed to it. We also generally assign variables using the `const` keyword, so we know to make a copy, and will get errors if we try to reassign a variable. We'll continue to do this to some extent though we mainly make use of ES6's concept of immutability, insomuch that it only protects against reassignment using the `const` keyword, we still have the ability to extend objects and interact with arrays using methods, such as `Array.prototype.pop()` and `Array.prototype.unshift()`, as per usual.

In the following examples, I give the full names of the native prototype methods so as to differentiate them from d3-selection's filter and map methods.

`Array.prototype.map`, thus, refers to the map method on the Array primitive's prototype object.



What's a prototype, though? JavaScript is a prototype-based language, meaning that everything is effectively an object that inherits from another object, its prototype. All arrays are descendants of the `Array` object, thus, they inherit its prototype methods, such as `.map()`, `.reduce()` and `.filter()`. The `Array` prototype itself inherits other prototypes, all the way up the chain to `Object.prototype`. D3 selections are also descendants of the `Array` prototype, but D3 then goes on to replace some functions, such as `.filter()` and `.sort()` with its own versions adapted for selections, in addition to adding a few other helpful methods, such as `.each`, or to come full circle with the whole naming thing, `d3.selection.prototype.each` (or simply `selection.each` as it's referred to in the documentation). Even ES2015 classes, which use a much different inheritance model and come from object-oriented programming, are ultimately still prototype-based.

You've already seen this functional approach 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 we 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.

Although I endeavor to take a somewhat functional approach throughout the rest of the book, there's no way I could adequately explain the nuances of what is a really interesting and important practice within modern JavaScript development. For a good series on the topic, I suggest Eric Elliott's excellent series that can be found at <https://medium.com/javascript-scene/the-rise-and-fall-and-rise-of-functional-programming-composable-software-c2d91b424c8c>.

```
> [1,2,3,4].map(d => d+1)<br/> [ 2, 3, 4, 5 ]  
  
> [1,2,3,4].reduce((acc, curr) => acc + curr, 0)<br/> 10  
  
> [1,2,3,4].filter(d => d%2) <br/> [ 1, 3 ]  
  
// Are all elements odd? <br/> [1,3,5,7,9].every(elem => elem % 2); // True <br/> [1,2,5,7,9].every(elem => elem % 2); // False <br/><br/> // Is at least one odd? <br/> [1,3,5,7,9].some(elem => elem % 2); // True <br/> [1,2,5,7,9].some(elem => elem % 2); // True <br/> [0,2,4,6,8].some(elem => elem % 2); // False
```

Also useful -- particularly for functional programming -- is `Array.from`, which creates a copy of an array.

Sometimes, it's tempting to use `Array.prototype.forEach` instead of `Array.prototype.map` because `.forEach` operates on the original array instead of creating a copy. Given, we're trying to adhere to a functional programming style in this book, we'll mainly use `.forEach` to run some logic and not do anything to the original array itself.

`Array.prototype.map`, `Array.prototype.reduce`, and `Array.prototype.filter` can be used to transform any data into a more useful format without mutating the original data. This is important because it makes troubleshooting things, such as why a value is not appearing as it should, much easier. One of the goals of functional programming is to write *idempotent* code that doesn't have any *side-effects*, or not-returning impact on the program.

Some of 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 ES2017 future, Babel polyfills them for us when it transpiles our bundle together.

Data functions of D3

D3 comes with a bunch of helper functions for manipulating arrays. They mostly have to do with handling data; things like calculating averages, ordering, bisecting arrays, and more. In D3 v4, they're found in the `d3-array` package.



Of all the D3 pieces of documentation that you'll read, the one for `d3-array` is definitely one of the most useful. We'll cover a few of `d3-array`'s more useful functions, but there is far more to it than we'll be able to reasonably go over in just one chapter. I even sometimes find it useful to have the README for `d3-array` open in a tab while I develop, because it also acts as a very good quick reference for each method's function signature; you can visit it at

<https://github.com/d3/d3-array>.

Briefly, some of the more useful functions in `d3-array` are as follows; most of them can take an accessor function, which is used when working with arrays of objects to specify which property should be used for the calculation:

- `d3.min` and `d3.max` return the smallest and largest values in the array respectively. `d3.extent` returns an array with both values.
- `d3.sum`, `d3.mean`, and `d3.median` are useful for doing these types of calculations on an array. It's important to use medians and means responsibly; displaying a mean on a dataset with a lot of extreme values at either end risks distorting the middle values. We'll discuss this a little bit more later on in [Chapter 10, Designing Good Data Visualizations](#).
- `d3.ascending` and `d3.descending` are pre-built comparator functions, so you don't have to remember which number you need to subtract from each number when sorting. It also ensures that numbers are sorted in natural order, and not alphabetical order (as is the default behavior of `Array.prototype.sort` if you sort numbers typed as strings, an issue we'll get into when we talk about static typing later on in the book).
- We've already used `d3.range` a few times already. Supply it two integer numbers and it will return an array with those values and every intermediary value between them; it includes the first number, not the last

number. It can be provided a third value that specifies how big the step between intermediary values should be, defaulting to 1. Note that this can be used on nonintegers, but with reduced accuracy; refer to the documentation for full details.

`d3-array` also contains `d3.histogram`, which we'll use later on, in [Chapter 7, *The Other Layouts*](#), to make histogram charts. I'm only mentioning it here because it's a bit more complex to explain; note that it's in the same package as all of the above.

Managing objects with d3-collection

I'm not going to dwell too long on `d3-collection`, as Babel provides objects, such as Maps and Sets, as part of the native JavaScript API. If you're developing for earlier browsers and not using Babel to transpile your code, `d3-collection` provides a good path to use some modern idioms without having to polyfill a bunch of stuff. All that said, it does have one incredibly useful utility, which is `d3.nest`. A nest is a structure similar to a NoSQL query result, in which documents have been aggregated into a group.

Say you get the following from a CSV document:

```
const sensorData = [
  {location: "a", status: "normal", average: "100", date: "2016-11-01"}, 
  {location: "a", status: "normal", average: "200", date: "2016-11-02"}, 
  {location: "a", status: "normal", average: "300", date: "2016-11-03"}, 
  [...] 
  {location: "b", status: "normal", average: "400", date: "2016-11-01"}, 
  {location: "b", status: "alarm", average: "500", date: "2016-11-02"}, 
  {location: "b", status: "alarm", average: "600", date: "2016-11-03"}, 
  [...] 
];
```

This might be from a spreadsheet that's a dump of a MySQL database somewhere, with a header row containing a location, an average value from a sensor, and the date on which the reading was taken. We can get by with a few complex `Map.prototype.reduce` calls or simply assign them to a nest and then roll it up, as follows:

```
d3.nest()
  .key(d => d.location)
  .rollup(d => d3.mean(d, v => v.average))
  .map(sensorData);

// >> {a: 200, b: 500}
```

What's happening here? We first create a map, which is an object that assigns one or more values to a particular key. In this case, we've chosen to group by the location. Then, we roll up the values, meaning that we run a function that assigns a value to the key -- in this case, we get the average of all items in each day (`d`, in this case, is an array containing all values associated with a key).

There are lots of things you can do with `d3.nest` and `nest.rollup` that I



simply just don't have space to accommodate. For a very good overview of the ways in which you can use `d3-collection`, visit LearnJSDa's excellent tutorial on it at http://learnjsdata.com/group_data.html.

What about ES6 Maps and Sets?

Maps and Sets are two new primitives that were introduced in ES2015. Both are iterable, meaning that you can loop through them as if they were an array.

Why would you use these? They ultimately provide a more structured way for assigning and accessing data, and can seem familiar if you've ever used these primitives in other languages. That all said, I don't make use of them in this book because `d3-selection` can't really do much with them, and it's just easier to use arrays. While `d3-collection` has its own variants of `Map` and `Set`, they are mainly used to deal with array data while working with `d3.nest` and are ultimately just a polyfill for browsers that don't support them natively (for example, Internet Explorer). What does this mean, given that Babel will polyfill Map and Set for us? Not much. With D3, all pieces of data tend to end up in arrays anyhow.

Although some interest has been expressed in making ES6 `Map` and `Set` work with `d3-selection`, there's an open issue from 2015 showing that there isn't much movement on that front. Keep an eye on <https://github.com/d3/d3/issues/2584> though, as this may change, particularly as browser support for Map and Set becomes more common.

Scales

We've already used scales many times -- we had a chant, if you remember; what was that again?

SURPRISE POP QUIZ:

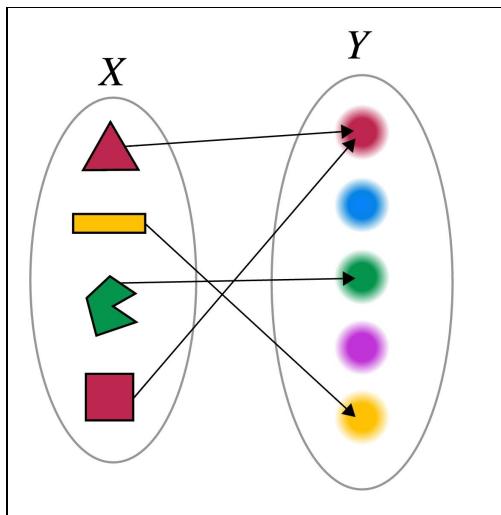
```
| INPUT!: [ ] DOMAIN! [ ] NOT DOMAIN!
| OUTPUT!: [ ] RANGE! [ ] NOT RANGE!
```

If you got `INPUT! = DOMAIN!` and `OUTPUT! = RANGE!`, you are totally correct!

The reason we use scales 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.

To reiterate a point I've hopefully been hammering home since [Chapter 1, Getting Started with D3, ES2017, and Node.js](#), a function's domains are those values that are defined (the input), and the range are those 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 could also do it with a dictionary, but `d3.scale` will always be more elegant and flexible:

```
const scale = d3.scaleOrdinal()
  .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. All scales are found in the `d3-scale` package, if you're going for the microlibrary approach or want to find its documentation.

Ordinal scales

Ordinal scales are the simplest; essentially just a dictionary, where keys are the domain and 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 was just `['red', 'yellow', 'green']`.

Let's try a few out. We will create band and point scales, which are ordinal scales with extra pizazz. We also need to create a color scale that repeats itself. First, though, we need somewhere to put our scales. Create a new Immediately Invoked Function Expression (IIFE) in `chapter4/index.js` named `scalesDemo`:

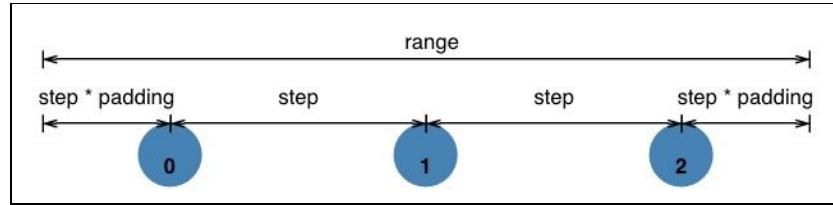
```
| const scalesDemo = (enabled => {
|   if (enabled) { // main block, put code here }
| })(true);
```

Next, we define the three scales we need and generate some data:

```
(function ordinalScales() {
  const data = d3.range(30);
  const colors = d3.scaleOrdinal(d3.schemeCategory10);
  const points = d3.scalePoint()
    .domain(data)
    .range([0, chart.height])
    .padding(1.0);
  const bands = d3.scaleBand()
    .domain(data)
    .range([0, chart.width])
    .padding(0.1);
}());
```

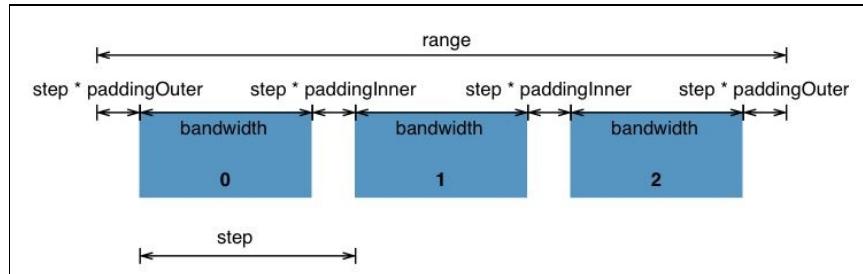
Our data is just a list of numbers going up to 30, and the colors scale is from [Chapter 2, A Primer on DOM, SVG, and CSS](#). In D3 v3, it used to be a predefined ordinal scale with an undefined domain and a range of 10 colors, but now it's just an array containing those same colors. For it to be an ordinal scale, it now needs to be assigned to one, such as we did in the preceding data.

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



This graphic from the d3-scale documentation explains how point scales are calculated.

We then use `.scaleBands()` to divide the width into 30 equal bands with a padding factor of 0.1 between bands. This time, we're setting the distance between bands, using `step * padding`, and a third argument would set edge padding using `step * outerPadding`:



This, also from the d3-scale documentation, depicts the factors influencing the calculation of a band scale.



This is a fairly big API change from D3 v3. Instead of `rangeBands()` and `rangePoints()` being methods attached to an ordinal scale, they're now scales unto themselves. For more information, refer to <https://github.com/d3/d3/blob/master/CHANGES.md#scales-d3-scale>.

We'll use code you already know from [Chapter 2, A Primer on DOM, SVG, and CSS](#), to draw two lines using these scales. Add the following to our IIFE:

```

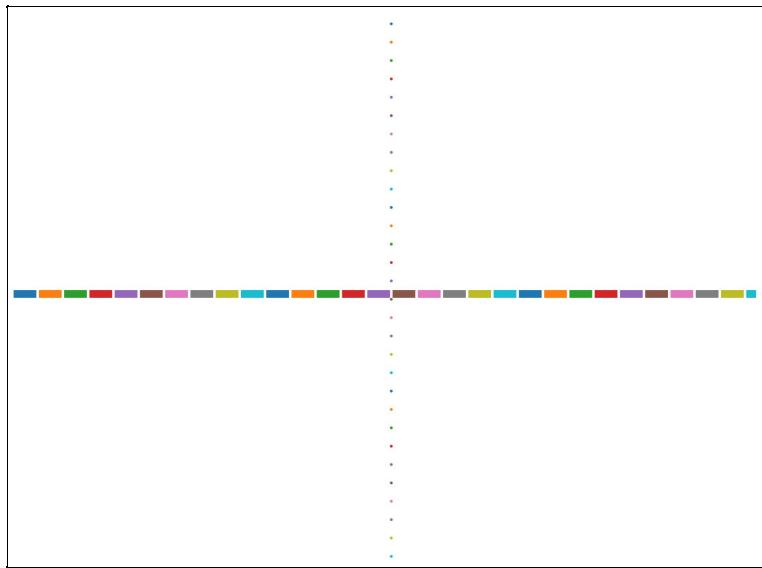
chart.container.selectAll('path')
  .data(data)
  .enter()
  .append('path')
  .attr('d', d3.symbol())
  .type(d3.symbolCircle)
  .size(10)
)
.attr('transform', d => `translate(${(chart.width / 2)}, ${points(d)})`)
.style('fill', d => colors(d));

chart.container.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('x', d => bands(d))
  .attr('y', chart.height / 2)
  .attr('width', bands.bandwidth)
  .attr('height', 10)
  .style('fill', d => colors(d));

```

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

The picture looks as follows:



You can get different effects by using different color schemes. Replace the whole bit starting with `chart.container.selectAll('rect')` with the following:

```

['10', '20', '20b', '20c'].forEach((scheme, i) => {
  const height = 10;
  const padding = 5;
  const categoryScheme = `category${scheme}`;
  const selector = `rect.${categoryScheme}`;
  const categoryColor = d3.scaleOrdinal(d3[categoryScheme]);

  chart.container.selectAll(selector)
    .data(data.slice())

```

```
.enter()
.append('rect')
.classed(selector, true)
.attr('x', d => bands(d))
.attr('y', (chart.height / 2) - ((i * height) +
  (padding * i)))
.attr('width', bands.bandwidth)
.attr('height', height)
.style('fill', d => categoryColor(d));
});
```

This renders all of the color schemes. Snazzy!



For more on D3's category scales, visit <https://github.com/d3/d3-scale#category-scales>.

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 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).

Add the following to `ScalesDemo`:

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

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

```
const data = d3.range(-100, 100).map(d => d / 200);
const extent = d3.extent(data.map(weierstrass));
const colors = d3.scaleOrdinal(d3.schemeCategory10);
const x = d3.scaleLinear()
  .domain(d3.extent(data))
  .range([0, chart.width]);
```

A drawing function will help us avoid code repetition:

```
const drawSingle = line => chart.container.append('path')
  .datum(data)
  .attr('d', line)
  .style('stroke-width', 2)
  .style('fill', 'none');
```

Continuous range scales

We can draw continuous range scales using the following code:

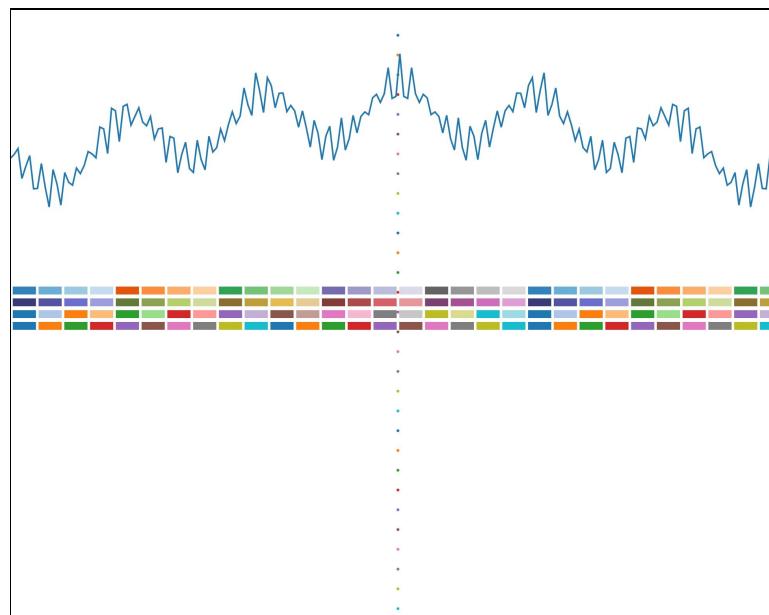
```
const linear = d3.scaleLinear()
  .domain(extent)
  .range([chart.height / 4, 0]);

const line1 = d3.line()
  .x(x)
  .y(d => linear(weierstrass(d)));

drawSingle(line1)
  .attr('transform', `translate(0, ${chart.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 output, and will even predict values that fall outside 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 the following:



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

```

const identity = d3.scaleIdentity()
  .domain(extent);

const line2 = line1.y(d => identity(weierstrass(d)));

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

const power = d3.scalePow()
  .exponent(0.2)
  .domain(extent)
  .range([chart.height / 2, 0]);

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

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

const log = d3.scaleLog()
  .domain(d3.extent(data.filter(d => (d > 0 ? d : 0))))
  .range([0, chart.width]);

const line4 = line1.x(d => (d > 0 ? log(d) : 0))
  .y(d => linear(weierstrass(d)));

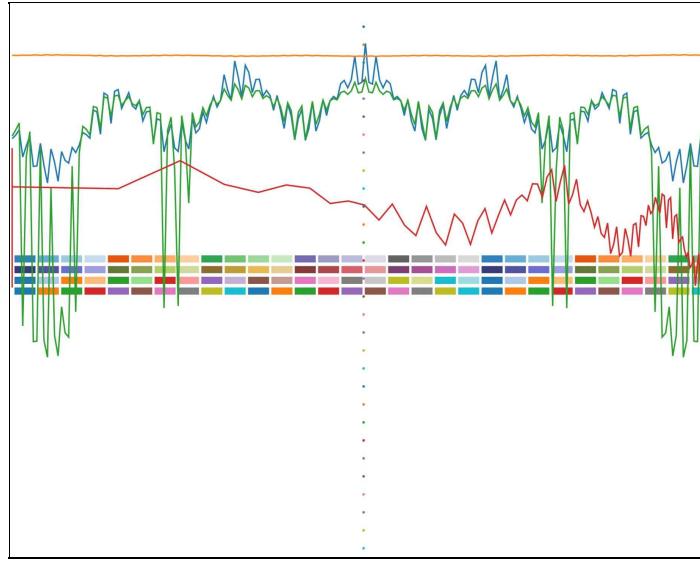
drawSingle(line4)
  .attr('transform', `translate(0, ${chart.height / 4})`)
  .style('stroke', colors(3));

```

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

Now, our picture looks as follows:



Starting to look like a New Order album cover or something up in here...
The identity scale (labeled 1) is orange and wiggles around by barely a pixel
because the data we feed into the function only goes from -0.5 to 0.5, the power
scale (labeled 2) is green, and the logarithmic scale (3) is red.

Discrete range scales

The interesting scales 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, whereas threshold scales let us map arbitrary domain sections to discrete values:

```
const offset = 100;
const quantize = d3.scaleQuantize()
  .domain(extent)
  .range(d3.range(-1, 2, 0.5))
  .map(d => d * 100);

const line5 = line1.x(x)
  .y(d => quantize(weierstrass(d)));

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

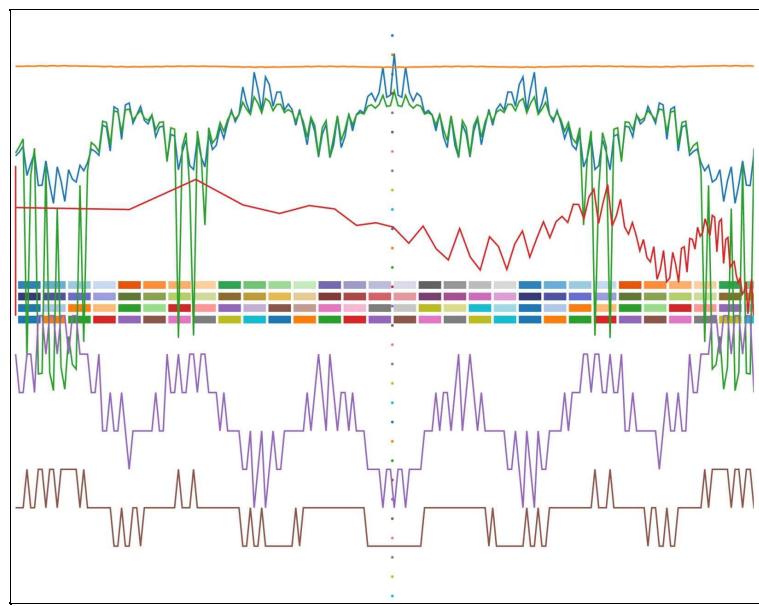
const threshold = d3.scaleThreshold()
  .domain([-1, 0, 1])
  .range([-50, 0, 50, 100]);

const line6 = line1.x(x)
  .y(d => threshold(weierstrass(d)));

drawSingle(line6)
  .attr('transform', `translate(0, ${chart.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 is as follows:



Time

Time is a complicated beast. An hour can last 3600 seconds or 3599 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 next time you want to add 3,600 seconds to a timestamp to advance it by an hour, or add $24*3600$ 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. They're all located in the `d3-time-format` package.

Formatting

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

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

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

We defined a new formatter with `d3.timeParse()` and provided a string mapping to year-month-day. We 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.

`d3.timeFormat()` works in another way:

```
|> format = d3.timeFormat('%Y-%m-%d')
|> format(new Date());
"2017-03-12"
```

You can find the full ISO standard time formatter at `d3.isoFormat`. This often comes in handy, given that most databases use it by default, as does the `toString()` method of `Date`.

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.timeInterval`, where "**Interval**" can be a second, a minute, an hour, and so on. It returns a new time interval. For instance, `d3.timeHour` will be an hour long.
- `d3.timeInterval(Date)` 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)` will move the date by a specified number of steps to the correct unit.
- `interval.range(Date_start, Date_stop)` will return every interval between the two specified dates.

For instance, if you wanted to find the time an hour from now, you'd do this:

```
| > d3.timeHour.offset(new Date(), 1)
| Sun Mar 19 2017 02:44:30 GMT+0100 (CET)
```

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



Want to do even more with time? `Moment.js` is a terrific library for accurately calculating things like time zones and the differences between two datetimestamps: <http://momentjs.com>.

Loading data

Quite often, you won't have the benefit of being able to create data using the Math random number generator functions, you'll need to load it from an external source. 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.

You can get data in a number of ways. I'll cover the main ones here:

- **Make some sort of manual HTTP request:** We already did this in earlier chapters. We supply a URL to a function that causes the browser to make a request. Both `XMLHttpRequest` and `Fetch` fall into this category. To import JSON, the server needs to have Cross-Origin Resource Sharing (CORS) enabled, meaning that it sends a header resembling `Access-Control-Allow-Origin: *`. This is due to the browser's security model, but it doesn't apply if we're loading data off the same domain, so we're able to do that without any extra work.
- **Import it as a module:** Some datasets are available as modules via npm. A good example of this is `earth-topojson`, which we'll use when we start making maps later on. This is super convenient, as it means you just have to import the module before being able to assign it to variables. Unfortunately, the static analysis requirements of ES2015+ means that you need to have a finite dataset that can be imported at, and only at build time.
- **Connect to a websocket:** Websockets are a new technology that enable the browser to subscribe to a stream and get new data pushed in real time. It's super useful for live data, but it's very memory intensive and you generally have to implement the server yourself.
- **Insert a script tag that loads in data via JSONP and attaches to a global:** JSONP is essentially a variant of JSON that is wrapped in a function callback specified by the URL. This is what people used before CORS became popular. You don't see it much any more.

The core

D3 provides its own system for making requests, which is found in the `d3-request` module. It uses `XMLHttpRequests` via 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 -- making the request once ready.

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

```
let xhr = d3.xhr('http://www.example.com/test.json');
  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 tell the server that we're a web browser named `SuperAwesomeBrowser`. One way to shorten this is by defining a callback immediately, but then you can't define custom headers or listen for other request events.

Another way is D3 convenience functions. They include the following:

`d3.text` to import a text file as a string:

```
d3.text('datatranscript.txt', (error, text) => {
  console.log(text);
});
```

`d3.csv` to load a comma-separated value spreadsheet. Use `d3.tsv` for tab-separated:

```
d3.csv('datafinance_data.csv', (error, data) => {
  console.log(data);
});
```

`d3.json` to load a JSON file:

```
| d3.json('http://www.aendrew.com/hello.json', (error, data) => {  
|   console.log(data);  
|});
```

`d3.xml` to load in an XML file:

```
| d3.xml('probablymade/by/java.xml', (error, data) => {  
|   console.log(data);  
|});
```

Note that you don't have to use any of these if you've gotten your data some other way; for instance, if you've loaded a CSV file as a string variable, you can use `d3.csvParse` from the `d3-dsv` module to turn that into an object or array.

Similarly, you can use the browser's built-in `JSON.parse()` to convert a JSON string into a usable object or `DOMParser` for XML.

Callbacks are used for flow control in all of the mentioned convenience methods, which can be annoying if you need to make multiple requests. Let's talk about that now.

Flow control

JavaScript is a single-threaded asynchronous language, meaning that it doesn't really *fork* in the same way you would expect a multi-threaded language like C++ to, and thus a function blocking the main thread causes everything to grind to a halt. Luckily, the *asynchronous* part of that means functions generally won't do this, and functions can be written so that the next one can fire before the first one finishes.

On the one hand, this is one of the most interesting and powerful aspects of JavaScript as a language. On the other, it makes things somewhat more difficult to reason about, and adds a degree of complexity to organizing one's code. While you can do pretty much anything without getting too bogged down in the JavaScript event model, the one place where the asynchronous nature of JavaScript is particularly visible is when you make a request -- no matter how fast broadband Internet gets, so long as it conforms to the standard model of physics, there will always be some degree of latency while data moves from one place to another. If JavaScript blocked the rest of your application while it started and waited for every HTTP request to finish, web pages would become sluggish to the point of being unusable. This is particularly true with modern JavaScript-based web applications, which potentially make dozens of requests in the background while you browse a site.

The traditional way of handling a request is to begin the request and then supply a function that is run once the request is complete, usually as a callback:

```
function done() {
  console.log(this.responseText); // Finished!
}

const req = new XMLHttpRequest();
req.addEventListener('load', done)
req.open('get', 'http://www.packt.com');
req.send();
```

Here, we use a callback function named *done* that prints the contents of the request to the console as a string. While it's pretty easy to understand what's going on in such a trivial example, it becomes much more difficult when you have to chain requests:

```

const req1 = new XMLHttpRequest();

req1.addEventListener('load', function done1() {
  const response1 = JSON.parse(this.textContent);
  const req2 = new XMLHttpRequest();

  req2.addEventListener('load', function done2() {
    console.log(this.textContent); // finished!
  });

  req2.get(response1.newEndpoint);
  req2.send();
});

req1.get('http://www.aendrew.com/api/1.json');
req1.send();

```

Here, we make one request, wait for it to finish, get a value from the JSON data we received, then make another request based on that value. If we wanted to make a third request using a value from the second request, we'd have to nest everything even further. This is commonly referred to in JavaScript development circles as *Callback Hell*, because it's really hard to read and reason about. Unfortunately, while D3's request methods are a bit less verbose than XMLHttpRequest, they still use the old school callback syntax:

```

d3.json('http://www.aendrew.com/api/1.json', (error1, data1) => {
  if (data1) {
    d3.json(data1.newEndpoint, (error2, data2) => {
      if (data2) {
        // ...and so on...
      }
    });
  }
});

```

After only two requests, we're already four levels of indentation deep. We could rewrite this in a few ways to make this more readable, but it's already starting to become apparent how annoying it can be to organize code in this fashion.

Fortunately, you are in no way chained to D3's request functions in order to acquire data! In fact, there are very many ways you can get data and structure your code in modern JavaScript, some of which include the following:

- Promises
- Generators
- Observables

We'll go through each of these momentarily, and how to use them with D3. Note

that you can use them for far more than simply waiting for data, as you'll see when we talk about generators.

Promises

Of all the methods we'll cover in this chapter to organize code, promises are probably my favorite. They're really simple to use and form the foundation of the `await/async` pattern I use all over this book. In short, a promise is an object that represents a value that may be available now, never, or sometime between those two extremes. Here's a short example:

```
const p = new Promise((resolve, reject) => {
  d3.json('http://www.aendrew.com/api/1.json', (err, data1) => {
    if (err) {
      reject(err);
    } else {
      resolve(data1);
    }
  });
};

p.then(data => console.log(data.newEndpoint))
  .catch(err => console.error(err));
```

We instantiate a new promise and assign it to `p`. The Promise constructor takes a callback containing two functions: `resolve` and `reject`. We pass errors to `reject` and data to `resolve`. Then, outside the Promise callback, we can use the `Promise.prototype.then()` method of `p` to do something once the promise is resolved, or handle errors using `Promise.prototype.catch()`. This is a pretty silly example though, because `d3-request` doesn't return promises, and you have to program around its callback style of design. Let's use `Fetch` from now on.

With that in mind, what if you have two promises you want to resolve before continuing? With callbacks, you'd probably still want to nest the requests or do something clever with events even if the second doesn't need any data from the first to succeed. With promises, we can resolve multiples of them using one call to `Promise.all`:

```
Promise.all([fetch('../data1.json'), fetch('../data2.json')])
  .then(([data1, data2]) => [data1.json(), data2.json()])
  .then(([data1, data2]) => {
    console.log(data1); // Resolved data1.json
    console.log(data2); // Resolved data2.json
 });
```

If you remember from way back in [Chapter 1, Getting Started with D3, ES2017](#),

and *Node.js*, we can use `fetch` to return a promise containing a request, which we can turn into another promise containing the resolved data parsed a particular way (text, binary blob, and JSON). All we're doing here is combining promises using `Promise.all` and resolving them as we would before.

But then again, why are we even bothering with the old promise syntax when we have the new hotness known as `async/await`?

```
async function getDataAsync() {
  const data1 = await (await fetch('./data1.json')).json();
  const data2 = await (await fetch(`./data${data1}.json`)).json();

  console.log(data1);
  console.log(data2);
}
```

Holy hell, that's utterly gorgeous! What's going on here?

Any function marked with `async` will ultimately return a promise of some variety. Even if you don't specify a value to return (such as in our preceding example), a function defined with the `async` keyword will still return a promise. What's cool, however, is that you can then use the `await` keyword to wait for a promise to resolve instead of having to use `Promise.prototype.then()`. As a result, the use of the Promise API is very transparent and lets you write asynchronous code in a style very familiar to anyone with experience of non-asynchronous languages.

I will refrain from giving an in-depth example of how to use Promises here because we've used them already and will continue to do so. By the end of this book, you'll be well-versed in how they work.

Generators

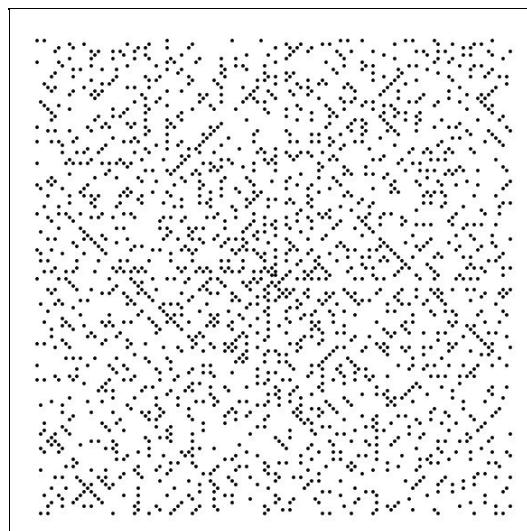
Newly added in ES7, generators are functions that can be paused while data is awaited.

When you instantiate a generator function, it returns an object containing a `next()` method and a `finished` property. Every time you call `next()`, the generator will *yield* the next value in the sequence. If and when you reach the end of the sequence, the `finished` property will equal true. 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.

For more on generators, visit http://mdn.io/Iterators_and_Generators.

Let's play with them a little by rendering 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; plot natural numbers in a spiraling pattern, and then remove all non-primes. Instead of numbers, we'll draw dots using the SVG `circle` element. The first stage in our experiment will look as follows:



Doesn't look like much, but that's only the first 2,000 primes in a spiral. 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 function in `lib/chapter3/index.js`. At the bottom of the file, turn our previous enclosure to `off` by setting the bit at the end to false, like so:

```
| })(false);
```

Next, create a new IIFE, as before:

```
| const ulamSpiral = ((enabled) => {
|   if (!enabled) return;
| 
|   const chart = chartFactory();
| })(true);
```

None of mentioned what's been should look very new at this point; we're just scaffolding a new chart via our `chartFactory` factory and instantiating it.

Next, we define the algorithm that generates a list of numbers and their spiraling coordinates on a grid. We start by creating the spiral algorithm. Create a new function inside the code we just wrote:

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

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

Next, we add the algorithm itself to the bottom of our new `generateSpiral` function:

```
d3.range(1, n).forEach((i) => {
  spiral.push({ x, 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;
```

`d3.range()` generates an array of numbers between the two arguments that 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 using `min` and `max` to change the direction once we bump into a corner.

Now we get to draw stuff; go back to our parent function and add the following to it:

```
const dot = d3.symbol().type(d3.symbolCircle).size(3);
const center = 400;
const l = 2;
const x = (x, l) => center + (l * x);
const y = (y, l) => center + (l * y);
```

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

Next, we need to calculate primes. We can always just get a big list of them online, but that wouldn't be as fun as using generators.

Create a new method called `generatePrimes` in our `ulamSpiral` function in `chapter3/index.js`:

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

Put the following generators inside this function.

Our first generator will simply 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 by the function keyword simply denotes it as a generator function. Once we're into the `while` loop (which doesn't ever finish, so we could 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 will 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;
|   }
| }
```

It shouldn't be that difficult to understand now. We assign our numbers generator to `seq` and start it on 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:

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

Now, put the following at the end of `generatePrimes`:

```
| for (let prime of getPrimes(n, primes())) {
```

```
|     console.log(prime);  
| }
```

Then, add this under our `generatePrimes` function:

```
| const primes = generatePrimes(2000);
```

If you start the development server via the following command:

```
| $ npm start
```

Then, go to <http://localhost:8080/>, you'll see an array of 2,000 sequential integers in 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 (let num of seq) {  
|         if (num % prime !== 0) {  
|             yield num;  
|         }  
|     }  
| }
```

This just goes through all the numbers in the sequence so far and checks whether there are any remainders when they're divided by a possible prime. If any of the numbers in the sequence have zero remainders when divided by a potential prime, it means 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`, add 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 we just created.

Go to the following:

```
| for (let 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;
```

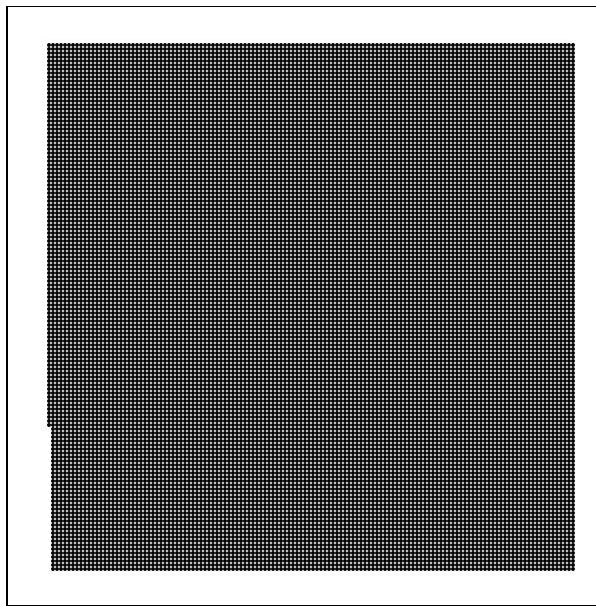
Now all of our primes are in an array; it's time to tie all this together! Go back to the constructor and add the following:

```
const primes = generatePrimes(2000);
const sequence = 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 pages ago:

```
chart.container.selectAll('path')
  .data(sequence)
  .enter()
  .append('path')
  .attr('transform', d => `translate(${x(d.x, 1)}, ${y(d.y, 1)})`)
  .attr('d', dot);
```

The output will look like this:



Hmm. Okay, not quite there, but it's 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:

```
| const sequence = 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 we'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 that result (and, indeed, generating more than 2,500 or so tends to cause Chrome to hit its stack size limit). Earlier, it was mentioned that we could 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 an approach such as this.

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. Squares will be red when there are fewer primes than 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 continuing where you left off in the constructor:

```
const scale = 8;
const 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.

We use `d3.nest()` 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
    }
}

```

Not very self-explanatory, but that's a collection of columns containing rows. The (0, 0) square contains 5 primes, (-1, 0) contains 2, and so on.

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

```

const values = d3.merge(d3.keys(regions)
    .map(_x => d3.values(regions[_x])));
const median = d3.median(values);
const extent = d3.extent(values);
const 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.

`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.

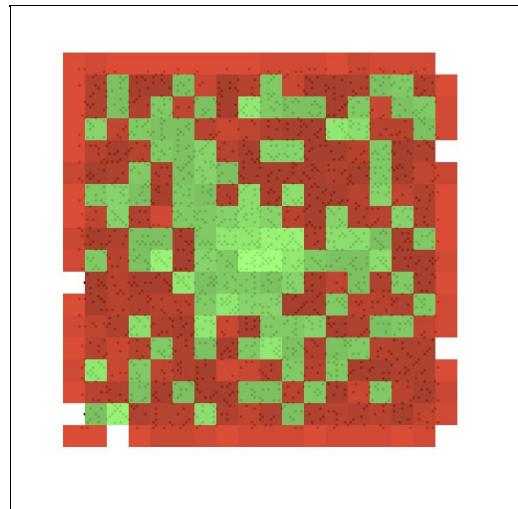
Finally, we walk the coordinates again to color the new grid:

```

regions.each((_x, _xKey) => {
    _x.each((_y, _yKey) => {
        let color,
            red = '#e23c22',
            green = '#497c36';
        if (_y > median) {
            color = d3.rgb(green).brighter(_y / shades);
        } else {
            color = d3.rgb(red).darker(_y / shades);
        }
        chart.container.append('rect')
            .attr('x', x(_xKey, 1 * scale))
            .attr('y', y(_yKey, 1 * scale))
            .attr('width', 1 * scale)
            .attr('height', 1 * scale)
            .style('fill', color)
            .style('fill-opacity', 0.9);
    });
});

```

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



Observables

Observables are a new way of approaching flow control, whereby you subscribe to a data source and then run functions based on events. While you can use them for anything, they're best suited to situations where the data does a lot of updating - for instance, in live stock market charts. We can also use Observables to run updates based on *push* events, such as when connected to a WebSocket stream, which reduces the need to constantly poll a single endpoint.

The most common library for working with Observables right now is RxJS. Whereas, with Promises, we'd do something like: const data = await (await fetch(url)).json()

With RxJS, we would instead do something like the following:

```
Rx.Observable
  .fromPromise(fetch('data/cultural.json'))
  .flatMap(v => v.json())
  .subscribe(
    (data) => {
      console.dir(data);
    },
    (error) => {
      console.dir(error);
    },
    () => {
      console.log('complete!');
    }
);
```

There's no way we can possibly cover anywhere near the level of depth necessary to adequately learn anything useful about Observables here, they're a really complex topic best suited to when one needs to respond to live data. For now, if your interest has been piqued, try taking a look at the RxJS book, available free online at <https://xgrommx.github.io/rx-book/>.

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 (and still isn't really *solved* in any way if the huge number of projections that ship with `d3-geo-projection` is any indication).

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 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 instead encodes basic features with arcs and re-uses 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

Compared with other types of data, geodata generally comes in formats less conducive to online presentation, and you'll need to convert it to TopoJSON. We'll find some data in Shapefile or GeoJSON formats, and then use the `topojson` command-line utilities to transform them into TopoJSON. Finding detailed data can be difficult, but is not impossible - if wanting to make a map of your country, google your country's census bureau. For instance, the US Census Bureau has many useful datasets available at <https://www.census.gov/geo/maps-data/>, the equivalent for the UK is at <https://geoportal.statistics.gov.uk/geoportal/>, and Canada's is: <http://www12.statcan.gc.ca/census-recensement/2011/geo/bound-limit/bound-limit-2011-eng.cfm>.

Natural Earth is another magnificent resource for geodata at different levels of detail. The biggest advantage 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>, or install the political boundaries pre-converted to TopoJSON via npm using the `world-atlas` package.

We're going to convert Natural Earth to TopoJSON because the `world-atlas` package doesn't give us things like terrain or rivers.

First, download everything from http://naciscdn.org/naturalearth/packages/natural_earth_vector.zip.

Unzip the archive and navigate to it in the console. Then, install the following dependencies globally:

```
| npm install --global topojson shapefile
```

We install it `topojson` globally here, because we need its accompanying `geo2topo` command-line tool to convert GeoJSON to TopoJSON. Assuming you've extracted the aforementioned zip file and you're currently residing in the resulting directory, run:

```
| $ mkdir output
| $ geo2topo water=<(shp2json 50m_physical/ne_50m_rivers_lake_centerlines.shp) >
water.json $ geo2topo land=<(shp2json 50m_physical/ne_50m_land.shp) > land.json
```

```
| $ geo2topo -n ne_50m_admin_0_boundary_lines_land=<(shp2json -n 50m_cultural/ne_50m_admi
```

Copy the resulting data files from build into your project's `data/` directory.

Drawing geographically

`d3.geoPath()` is going to be the workhorse of our geographic drawings.

It's similar to the SVG path generators we learned about earlier, except 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 a 2D image, `d3.geoPath()` uses projections. Different kinds of projections are designed to showcase different things about the data, but the end result is you can completely change what the map looks like just by changing the projection or moving its focal point.

Let's draw a map of the world, centered on and zoomed into Europe. We'll make our map navigable in [Chapter 5, Defining the User Experience – Animation and Interaction](#).

First though, we need to install the TopoJSON files into your project.

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

Now require topojson at the top of `lib/chapter4/index.js`.

```
| import * as topojson from 'topojson';
```

Let's create a new enclosure for all of this in `chapter4/index.js`:

```
| const geoDemo = ((enabled) => {
|   if (!enabled) return;
|   const chart = chartFactory();
| })(true);
```

Next, after `const chart`, we define a geographic projection in the constructor:

```
| const projection = d3.geoEquirectangular()
|   .center([8, 56])
|   .scale(500);
```

The equirectangular projection is one of the hundred-odd projections in the d3-geo-projection package, and is perhaps the most common projection we're used to seeing ever since high school.



The problem with equirectangular is that it doesn't preserve areas or represent the earth's surface all that well. The debate of how to best project a sphere onto a two dimensional surface has a fascinating history, and D3's enormous collection of geographic projections shows the many ways people have attempted to do so over the years: <https://github.com/d3/d3-geo-projection>.

The next two lines define where our map is centered and how zoomed in it is. By fiddling, I got all three values, latitude of 8, longitude of 56, and a scaling factor of 500. Play around to get a different look.

Now we load our data, using Fetch:

```
const world = await Promise.all([
  (await fetch('data/water.json')).json(),
  (await fetch('data/land.json')).json(),
  (await fetch('data/cultural.json')).json(),
]);
```

We're using ES2015 promises to run the three loading operations simultaneously. 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 returns its resolved value to `await` once all the promises are accounted for.

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

```
const addToMap = (collection, key) => chart.container.append('g')
  .selectAll('path')
  .data(topojson.feature(collection, collection.objects[key]).features)
  .enter()
  .append('path')
  .attr('d', d3.geoPath().projection(projection));
```

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

Whether it's more efficient to transform to GeoJSON than transferring 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.geoPath()` and tell it to use our projection. Other than generating the SVG path string, `d3.geoPath()` can also calculate different properties of our feature, such as the area (`.area()`) and the bounding box (`.bounds()`).

Now we can start drawing:

```
const draw = (worldData) => {
  const [sea, land, cultural] = worldData;
  addToMap(sea, 'water').classed('water', true);
  addToMap(land, 'land').classed('land', true);
  addToMap(cultural, 'ne_50m_admin_0_boundary_lines_land').classed('boundary', true);
  addToMap(cultural, 'ne_50m_urban_areas').classed('urban', true);
  chart.svg.node().classList.add('map');
};
```

Our draw function takes the error returned from loading data and the three datasets then lets `addToMap` do the heavy lifting. We use a new ES2015 feature here - argument destructuring - to assign each element of the array to a new variable.



What's all this now 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: `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.

Add some styling to styles/index.css:

```
.river {
  fill: none;
  stroke: #759dd1;
  stroke-width: 1;
}

.land {
  fill: #ede9c9;
  stroke: #7b5228;
  stroke-width: 1;
}

.boundary {
  stroke: #7b5228;
  stroke-width: 1;
  fill: none;
}

.urban {
  fill: #e1c0a3;
```

```
| }  
| .map {  
|   background: #79bcd3;  
| }
```

And then make sure it's still required at the top of `lib/main.js`:

```
| import '../styles/index.css';
```

Make sure `webpack-dev-server` is running and look at your page. It should resemble this:



We now have a slowly-rendering world map zoomed into Europe, displaying the world's urban areas as blots.

There are many reasons why it's 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've traded flexibility for rendering speed in this instance. That said, we could dramatically speed up our rendering by reducing the size of our TopoJSON files by passing them through `toposimplify` from the `topojson-simplify` package.

I've written a more detailed guide to the TopoJSON command-line tools in this post: <https://medium.com/@aendrew/creating-topojson-using-d3-v4-10838d1a9538>. For a very thorough walk-through of all the tools, direct from Mike Bostock, read his

series, starting at <https://medium.com/@mbostock/command-line-cartography-part-1-897aa8f8ca2c>.

Using geography as a base

Geography isn't just about drawing maps. A map is usually a base 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 US. 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/pdfl/XLS%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/master/chapter4/data/airports.dat> and <https://github.com/aendrew/learning-d3/blob/master/chapter4/data/renditions.csv> respectively.

For the renditions dataset, you'll need to open it in Excel and save as CSV. I've done that for you if you grab it from GitHub. We also need to install the `d3-dsv` package for parsing our CSV files:

```
| npm install d3-dsv --save
```

Now import `parseCSV` and `csvParseRows` from that by putting the following line at the top of `chapter4/index.js`:

```
| import { csvParseRows, parseCSV } from 'd3-dsv';
```

Make sure the files are named `airports.dat` and `renditions.csv` in your data directory, then add two more calls to `Promise.all()` to load them in. Add a call to `addRenditions()` after `draw()`.

```
| addRenditions(
|   await (await fetch('data/airports.dat')).text(),
|   await (await fetch('data/renditions.csv')).text()
| );
```

The function loads the two datasets and then calls (the as yet nonexistent) `addRenditions` to draw them.

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(airportData, renditions) {
  const airports = csvParseRows(airportData)
    .reduce((obj, airport) => {
      obj[airport[4]] = {
        lat: airport[6],
        lon: airport[7],
      };
      return obj;
    }, {});
  const routes = csvParse(renditions).map((v) => {
    const dep = v['Departure Airport'];
    const arr = v['Arrival Airport'];
    return {
      from: airports[dep],
      to: airports[arr],
    };
  })
    .filter(v => v.to && v.from)
    .slice(0, 100);
}
```

We used `d3.csvParseRows` 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, "
-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` are missing, which are likely cases where our map function wasn'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 100 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. Still inside our `drawRenditions` function, add the following:

```
chart.container.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 to `index.css`:

```
| .route {  
|   stroke-width: 2px;  
|   stroke: goldenrod;  
| }
```

The following screenshot displays the result:



Huh. That doesn't have much, beyond the one route represented by the black line near the middle. We're probably zoomed in too much. Let's tweak it a little. Go back to where we defined projection and set the scale to 200, with -50, 56 set as the center:

```
const projection = d3.geoEquirectangular()  
  .center([-50, 56])  
  .scale(200);
```

Hey! There we go! This is suddenly looking like the start of a piece of interactive news content!



We solved what's 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 map data (possibly coding departure airports one color and arrival airports another) or provide UI elements to toggle aspects of the dataset. We'll look at interactivity in the coming chapters; hold onto 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. We've looked at the various types of JavaScript flow control, resolved more promises than we care to remember and converted some shapefiles to TopoJSON -- at this point, your ES2015 skills should be approaching (if they aren't already over) 9,000.

Lastly, we made a cool map that showed a particular set of data, where we mashed up multiple datasets to create something cool. You'll find most of the more interesting data visualizations you build involve combining multiple datasets -- with your new-found skills with promises, you're well on your way to mashing up *all the data*.

In the next chapter, we'll look at making our stuff pop visually by adding animations. Now things are starting to get pretty exciting!

Defining the User Experience - Animation and Interaction

Animation is like salt. A little bit goes a really long way and can really help to make a graphic more digestible while leading the viewer through the content; too much, and it's all anyone notices. Good **User Experience (UX)**, the computer-use idioms you employ throughout your projects--is more like guacamole. If it's good, it's a nice subtle touch that improves the overall quality of your output and everyone's happy; if it's bad, it totally taints everything and ruins 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 the map from the last chapter super awesome. Throughout, we'll discuss why or why not animation or interactivity should be used in a particular scenario.

The ability to creatively display data with D3 is one of the best reasons for using it; interaction and animation allow you to not just 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 the like, or an *explanatory* graphic, where minimal interactivity guides the user through the relevant data. In reality, you'll probably mix both the approaches, but understanding which type of interaction you want to have with the reader at what point can be helpful in planning your projects.

We'll discuss the differences between these two approaches in this chapter.

Animation

The first question worth asking is: *Why would animation improve this project?*

If you're making something that isn't really intended 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. Don't let me discourage you from running rainbow color interpolators through your charts if you think it'd be fun (because, speaking from personal experience, creating crazy animated art with D3 is a rather 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 upward 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 animation with transitions. Transitions use the familiar principle of changing a selection's attributes, except that the changes are applied over time. Transitions, easings, and the like are provided in the `d3-transition` package.

To slowly turn all rectangles on the page into red, we will use the following line of code:

```
| d3.selectAll('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 an internal timer. The easiest way to create a sequence of transitions is to nest transition calls--each delay will be relative to the most recent transition:

```
| d3.select('rect')
|   .style('fill', 'green')
|   .transition()
|   .delay(2000)
|     .style('fill', 'red')
|     .transition()
|     .delay(1000)
|       .style('fill', 'blue')
```

This will fill the selected rectangle; wait two seconds and transition it to being red, then wait another second and transition it to being blue.



Delays being relative to the last transition is a significant change from D3 v3! Previously, delays were relative to the first transition in the chain and were a bit of a pain to orchestrate.

If you want to do something before a transition begins, or want to listen for it to end, you can use `.on()` with the appropriate event type. Add the following to the

preceding code:

```
| .style('fill', 'red')
|   .on('start', () => { console.log("I'm turning red!"); })
|   .on('end', () => { console.log("I'm all red now!"); })
```

This is a fairly major change in D3 v4; previously, you would use `transition.each()` to listen for transition events. In D3 v4, `transition.each()` works identically to `selection.each()`, that is, it takes a callback containing the datum, index, and context of the current element and allows you to invoke arbitrary code on each element.

This is handy for 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 this state or that.

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, number, or 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 the `b` argument. `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 notations. 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 will parse it for numbers, and then use `.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)
.attr('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 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`, 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.

We will start putting everything in separate files because it helps us manage our code when we start extending our charts. First, create `lib/chapter5/index.js` and add the following:

```
| import easingChart from './easingChart';
| easingChart(true);
```

Next, create a new file, `lib/chapter5/easingChart.js`, and add a new function enclosure:

```
| import * as d3 from 'd3';
| import chartFactory from '../common';
| function easingChart(enabled) {
|   if (!enabled) return;
|   const chart = chartFactory();
| }
| export default easingChart;
```

Next, we need an array of easing functions and a scale for placing them along the vertical axis. Put this after `const chart`:

```
| const easings = ['easeLinear', 'easePolyIn(4)', 'easeQuadIn', 'easeCubicIn', 'easeSinIn'
| const y = d3.scaleBand()
| .domain(easings)
| .range([50, 500]);
| const svg = chart.container;
```

You'll note that `easePolyIn`, `easeElasticIn`, and `easeBackIn` take arguments; since these are just strings, we'll have to manually change them into real arguments later. The `easePolyIn` easing function is a polynomial, so `easePolyIn(2)` is equal to `easeQuadIn` and `poly(3)` is equal to `easeCubicIn`. Alternatively, for those of us who stopped paying attention to math toward the end of our secondary school, 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 following code snippet). The higher the number, the more dramatic the delay. Have a play with it, do what feels right.

The `easeElasticIn` easing function simulates an elastic and the two arguments control tension. I suggest playing 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 will be.

The easings at the end include an `-out` modifier as well as an `-in` modifier. These mean the following:

- `-in`: Normal behavior
- `-out`: Reverses the easing direction
- `-inOut`: Copies and mirrors the easing function from `[0, 0.5]` and `[0.5, 1]`
- `-outIn`: 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 will render a bunch of circles animated using each easing:

```
easings.forEach((easing) => {
  const transition = svg.append('circle')
    .attr('cx', 130)
    .attr('cy', y(easing))
    .attr('r', (y.bandwidth() / 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.bandwidth()` for circle size. This way, we can add or remove examples easily. Transitions will start with a delay of just under half a second to give us a chance to see what's going on. A duration of 1,500 milliseconds and a final position of 400 should give enough time and space to see the easing.

We define the easing at the end of this function, before the `});` bit:

```
if (easing.indexOf('(') > -1) {
  const args = easing.match(/[\d-]+/g);
  const funcName = easing.match(/^[\w-]+/i).shift();
```

```

    const type = d3[funcName];
    transition.ease(type, args[0], args[1]);
} else {
  const type = d3[easing];
  transition.ease(type);
}

```

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.



Note that in D3 v4, we use the symbols attached to the D3 object, for instance, `d3.easeCubicIn`. Previously, you'd supply a string to `transition.ease()`, such as '`cubicIn`'.

Let's add some text so that we can tell the examples apart:

```

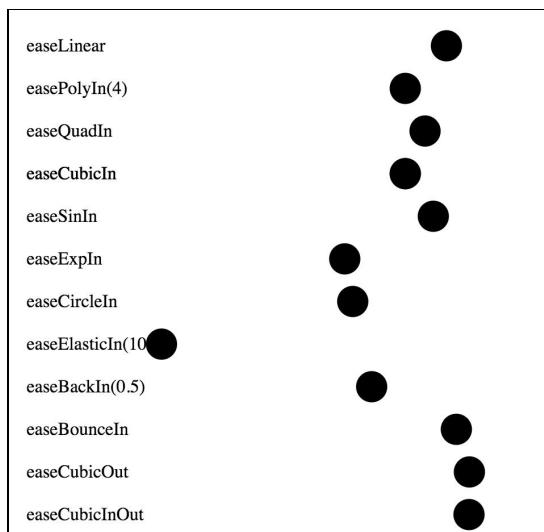
svg.append('text')
  .text(easing)
  .attr('x', 10)
  .attr('y', y(easing) + 5);

```

Ensure that the server's running (`$ npm start` if not) and visit <http://127.0.0.1:8080>.

In order to prevent having to repeat the preceding four lines over and over, I'll take it on faith that you know how to import and instantiate the modules we're creating at this point, in addition to knowing how to load the development server. I'll, thus, leave out that part from here on.

The visualization is a cacophony of dots:



The preceding figure doesn't quite showcase the animation, so you should really try this one in the browser. Alternatively, you can 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 most important thing is to think about how stuff should logically move instead of just slapping a 1-second linear fade-in on everything.

Timers

To schedule transitions, D3 uses timers. 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 not familiar with the animation terminology, keyframes define the start or end of a smooth transition.

To create a timer, we use `d3.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 epoch (`Date.getTime()` will do), 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.

Pass `false` to `easingChart()` in `chapter5/index.js` and add the following to the top:

```
| import spirograph from './spirograph';
| spirograph(true);
```

Next, create a new file in `lib/chapter5/`, called `spirograph.js`, and add this to the top:

```
| import * as d3 from 'd3';
| import chartFactory from '../common';

function spirograph(enabled) {
  if (!enabled) return;

  const chart = chartFactory();
}
export default spirograph;
```

We will construct a parametric equation. Here's a good function adapted from Wikipedia's article on those at http://en.wikipedia.org/wiki/Parametric_equations:

```

const position = (t) => {
  const a = 80;
  const b = 1;
  const c = 1;
  const 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 will return 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; examples are there in the same Wikipedia article.

This function returns positions between -2 and 2, so we need some scales to make it visible on the screen:

```

const tScale = d3.scaleLinear()
  .domain([500, 25000])
  .range([0, 2 * Math.PI]);

const x = d3.scaleLinear()
  .domain([-2, 2])
  .range([100, chart.width - 100]);

const y = d3.scaleLinear()
  .domain([-2, 2])
  .range([chart.height - 100, 100]);

```

`tScale` will translate time into parameters for the function; `x` and `y` will calculate the final position of the image.

Now we need to define `brush` that flies around and draws the lines. We also need a variable to hold the `previous` position:

```

const brush = chart.container.append('circle').attr('r', 4);
let previous = position(0);

```

Next, we need to define an animation `step` function that moves the brush and draws a line between the previous and current points:

```

const step = (time) => {
  if (time > tScale.domain()[1]) {
    return true;
  }

  const t = tScale(time);
  const pos = position(t);

  brush
    .attr('cx', x(pos.x))

```

```
    .attr('cy', y(pos.y));

chart.container.append('line')
  .attr('x1', x(previous.x))
  .attr('y1', y(previous.y))
  .attr('x2', x(pos.x))
  .attr('y2', y(pos.y))
  .attr('stroke', 'steelblue')
  .attr('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 to 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 steelblue line between the previous and current position (`pos`).

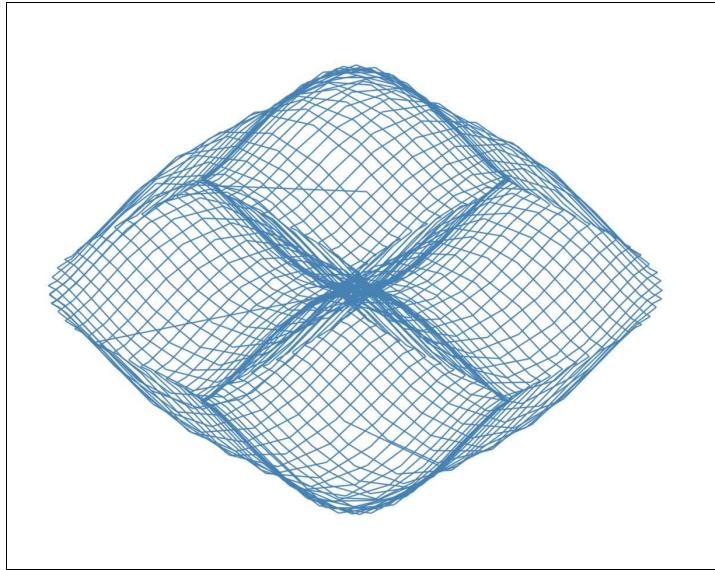
We conclude by setting a new value for the previous position.

All that's left now is to create a timer:

```
| d3.timer(step, 500);
```

That's it. Half a second after a page refresh, the code will begin drawing a shape and finish 25 seconds later.

After a while, it should look like this:



Putting it all together - sequencing animations

Time to put what we've learned to good use. Let's stop creating toys for a second and instead, create an actual chart.

We're going to use a dataset of UK prison population from 1900 to 2015, which is available at `uk_prison_population_1900-2015.csv` in this `data/` folder of the book's repository. I used this data to create a similar series of charts for *The Times* in 2016, with my team trying out several variations similar to the charts you're about to create, before arriving at the style of interaction you'll see in the following section, "Interacting with the user".

In `chapter5/index.js`, comment everything out and add the following at the top:

```
import prisonChart from './prisonChart';
(async (enabled) => {
  if (!enabled) return;
  await prisonChart.init();
  const data = prisonChart.data;
  function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
  }
  const randomChart = () => {
    try {
      const from = getRandomInt(0, data.length - 1);
      const to = getRandomInt(from, data.length - 1);
      prisonChart.update(data.slice(from, to));
    } catch (e) {
      console.error(e);
    }
  };
  prisonChart.update(data);
  setInterval(randomChart, 5000);
})(true);
```

We will create a chart that randomly displays a subset of our data, animating between each state. You wouldn't do this in production, but setting up tests like this can be useful to ensure that your animations will work no matter what you throw at them. We haven't written `prisonchart.init()` yet; we'll do so shortly.

Create a new file in `lib/chapter5/`, called `prisonchart.js`, and add the following:

```

import * as d3 from 'd3';
import { csvParse } from 'd3-dsv';
import chartFactory from '../common';

const prisonChart = chartFactory({
  margin: { left: 50, right: 0, top: 50, bottom: 50 },
  padding: 20,
  transitionSpeed: 500,
});

```

We instantiate a new chart using our good old `chartFactory` function, setting a few margins and default values.

Next, we need to define some methods on our new chart object:

```

prisonChart.resolveData = async function resolveData() {
  return csvParse(await (await fetch('data/uk_prison_data_1900-2015.csv')).text());
};

```

Here, we create a new asynchronous method to resolve our data. All this does is grab the file as a text string and then parse it using `d3-dsv`.

Next, we create another `async` function to do things like initializing scales. We make this asynchronous so that we can `await` the promise created by `resolveData()`:

```

prisonChart.init = async function init() {
  this.data = this.data || await this.resolveData();
  this.innerHeight = () =>
    this.height - (this.margin.bottom + this.margin.top + this.padding);
  this.innerWidth = () =>
    this.width - (this.margin.right + this.margin.left + this.padding);

  this.x = d3.scaleBand()
    .range([0, this.innerWidth()])
    .padding(0.2);

  this.y = d3.scaleLinear()
    .range([this.innerHeight(), 0]);

  this.x.domain(this.data.map(d => d.year));
  this.y.domain([0, d3.max(this.data, d => Number(d.total))]);

  this.xAxis = d3.axisBottom().scale(this.x)
    .tickValues(this.x.domain().filter((d, i) => !(i % 5)));
  this.yAxis = d3.axisLeft().scale(this.y);

  this.xAxisElement = this.container.append('g')
    .classed('axis x', true)
    .attr('transform', `translate(0, ${this.innerHeight()})`);
  this.yAxisElement = this.container.append('g')
    .classed('axis y', true)
    .call(this.yAxis);

  this.barsContainer = this.container.append('g')

```

```
| }, .classed('bars', true);
```

You may have noted that we use both the full not-fat-arrow `function` keyword here, and use the `this` keyword quite a lot. Using fat-arrow sets the `this` context to the parent block, which we don't want as we're wanting to set internal properties of our `prisonChart` object. This will come in handy later as we extend our chart to do new and different things.

Essentially, all we do in the preceding code is set up the scales and axes and assign them to our object's internal state using its `this` object. We also create a container to put our bars in, which we assign to `this.barsContainer`. Most notably, we use our `resolveData` function to get our data for us, which we assign to the `data` property of `prisonChart`.

We will do something different and create a whole method to update data in our chart. When we do more interesting things with interactivity later on in the chapter, we'll rely on this to change our chart's state.

I will go through this one step at a time as it expands upon what we learned about data joints earlier in the book while adding transitions. First, create a function and assign it to `prisonChart`, like so:

```
prisonChart.update = function update(_data) {
  const data = _data || this.data;
  const TRANSITION_SPEED = this.transitionSpeed;
};
```

All we do here is set a constant for our transitions, which we'll use throughout. We have a local `data` variable that we populate with either the subset of the data passed through as the argument or, if that's undefined, we grab the data we set to the object's internal `data` property in the `init()` function.

Continuing to add to that our `update()` function, insert the following lines:

```
// Update
const bars = d3.select('.bars').selectAll('.bar');
const barsJoin = bars.data(data, d => d.year);

// Update scales
this.x.domain(data.map(d => +d.year));
this.xAxis.tickValues(this.x.domain()
  .filter((d, i, a) => (a.length > 10 ? !(i % 5) : true)));

// Call this.xAxis after double the TRANSITION_SPEED timeout value
```

```
| d3.timeout(this.xAxis.bind(this, this.xAxisElement), TRANSITION_SPEED * 2);
```

Here, we select our `.bars` container, and then all the items with class `.bar` inside it, which we save as our `bars` local variable. We then join our new dataset to that, which we assign to `barsJoin`. Note how we use the second argument in `selection.data()` to tell D3 which pieces of data correspond to which array element; normally, D3 keys objects by array index, but if we're adding and subtracting elements to our data array, we can't rely on that. We update our x scale's domain to be all the years supplied in the dataset, and set the x axis to display every fifth tick value if the number of items in the array is greater than 10. We then set a timer to fire once after waiting twice the value of our transition speed constant, which causes our x axis to update (if confused, running `this.axisElement.call(this.xAxis)` would update the axis immediately, while `this.xAxis.bind(this, this.xAxisElement)` returns a function that updates our x axis when it is run--in this way, we leave `d3.timeout` to run the function once the timeout function fires).

Continuing to add to `prisonchart.update()`:

```
// Remove
barsJoin.exit()
  .transition()
  .duration(TRANSITION_SPEED)
  .attr('height', 0)
  .attr('y', this.y(0))
  .remove();
```

Here, we take our new data and join and set up its `exit()` state. We set up a new transition, which we set to the length of our transition speed constant. We then tell it to transition each bar's height to zero as well as move its y value to the chart x axis. This will cause all of our bars to shrink towards the x axis as they're leaving the scene.

Next, we set our `enter()` state:

```
const newBars = barsJoin.enter() // Enter
  .append('rect')
  .attr('x', d => this.x(+d.year))
  .classed('bar', true);
```

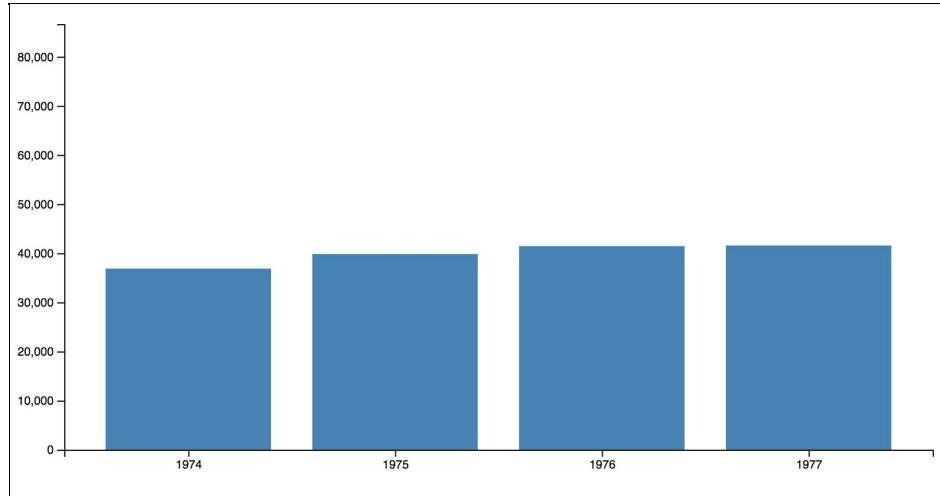
This sets a new local variable to the new bars that were created due to the data join. We set their x value straightaway and give them a `.bar` class.

Here's the last bit of `prisonChart.update()`, which handles both the new bars and the remaining old bars:

```
barsJoin.merge(newBars) // Update
  .transition()
  .duration(TRANSITION_SPEED)
  .attr('height', 0)
  .attr('y', this.y(0))
  .transition()
  .attr('x', d => this.x(+d.year))
  .attr('width', this.x.bandwidth())
  .transition()
  .attr('x', d => this.x(+d.year))
  .attr('y', d => this.y(+d.total))
  .attr('height', d => this.innerHeight() - this.y(+d.total));
```

Here, we use D3 v4's new `merge()` function to update our existing bars. We take the selection of our old bars and merge it with the selection of our new bars and then run operations on all of them. First, we initiate a transition lasting one transition speed interval, which sets all of our bars to a height of zero and a scaled y value of zero; this is exactly what we did while removing bars. Then, we set another transition lasting the default of 250 ms, which moves all the bars to their new location and sets their width appropriately, given their size. A final transition after that one sets each bar to its proper height.

Refresh and you'll have a hyperactive, rapidly-refreshing chart!:



The bars grow, the data changes, the bars shrink, the axes update, and the bars grow again. Splendid!

That does it for animation using D3 transitions. Next, we'll be looking at adding

interactions using the various behaviors that come with D3.

What are the other ways 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 deprecated it, which means you never have to worry about learning it. For the morbidly curious, refer to https://mdn.io/SVG_animation_with_SMIL.

*Another way of doing transitions is using **CSS transitions**. For simple animations, these are terrific as they are able to use your graphics card GPU to render things, ensuring that they don't block and thus improving performance. In fact, I used these for the last example in the previous edition of the book. However, they don't work with canvas and are difficult to sequence when you're also using d3-transition in your project, which means that they're probably best left to things such as animating hover states on buttons and other UI-related animations.*

For more information, visit: https://developer.mozilla.org/Web/CSS/CSS_Transitions/Using_CSS_transitions.



*In the future, another alternative to animate using JavaScript will be the **Web Animation 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: '&grave;translate(${x(i)}, ${y(d)})&grave;'}  
  ], {  
    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. Again, this is all really early days; if you use the Web Animations API with D3 in a project, drop me a line and let me know how it went!

Interacting with the user

This is it. This is where all of the UX tidbits I've been dropping throughout the chapter and all the functional programming ideas you've been learning come together; let's make a simple explanatory graphic that uses interaction to walk the viewer through some data.

The first step to 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 and what's the best way to tell it?

In the prison population dataset, we have the numerical product of over a century of incarceration in a western country. There are many ways we can look at this data. We can look at how the prison population has risen versus 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. When planning complex sequential interactions like this, it helps to write down these things in either bullet point or paragraph form, possibly even drawing a flow chart, 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 will look at a few notable historical points. The graphic will have five states, which will be navigated through a series of five buttons:

- **Initial view:** Years 1900 to 2015. This provides a general overview of how the prison population has risen over time.
- **Zoom 1900 to 1930:** Highlight 1914-1918. This text will explain that population rose due to the end of World War I.
- **Zoom 1930 to 1960:** Highlight 1939-1945. This text will explain why population rose after World War II.
- **Zoom 1960 to 1990:** Discusses the rise of consumer society and its impact

on crime.

- **Zoom 1990 to 2015:** Highlight 1993 and explains the sharp rise after the murder of James Bulger was used by politicians to increase penalties.

We're keeping the user interface deliberately simple, but remember that simple is often better, particularly when building for an audience on mobile (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 that our listener is called first and all the 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 that there will only 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 for avoiding unpredictable behavior.

Just like other functions acting on element selections, event listeners get the current datum and index and set this context to the DOM element. The global `d3.event` will let you access the actual event object.

Let's create a function to add our buttons and wire this all up.

First, comment out everything in `lib/chapter5/index.js` and add the following:

```
import buttonPrisonChart from './buttonChart';
(async (enabled) => {
  if (!enabled) return;
  await buttonPrisonChart.resolveData();
  await buttonPrisonChart.init();
  buttonPrisonChart.addUIElements();
})(true);
```

This is like a simplified version of the last IIFE we wrote in `index.js`. We await the data and then initialize.

Create a new file, `lib/chapter5/buttonChart.js`. Add the following:

```
import * as d3 from 'd3';
import scenes from '../../data/prison_scenes.json';
import PrisonPopulationChart from './prisonChart';
```

```

| const buttonPrisonPopulationChart = Object.create(PrisonPopulationChart);
| buttonPrisonPopulationChart.scenes = scenes;
| buttonPrisonPopulationChart.addUIElements = function addUI() {}

```

We import D3, a JSON file containing our descriptions for each chart state, and our last chart. Instead of creating a brand new chart with `chartFactory`, we use `Object.create()` to create a new object, using `prisonChart` as its prototype. This means that we have all methods and values of `prisonChart` available to us, and we can interact with its internal API by adding new methods or overloading the existing methods. We also instantiate a new function, `addUIElements()`, that we referenced earlier in `index.js`.

Add the following to `addUIElements()`:

```

// Add some room for buttons
this.height -= 100;

// Needs to update y scale/axis
this.y.range([this.innerHeight(), 0]);
this.yAxisElement.call(this.yAxis);

// ...and the x scale/axis
this.xAxisElement.attr('transform', &grave;translate(0, ${this.innerHeight()}&grave;,

```

We set the height to 100 pixels smaller than what we used for the last chart (in essence, the entire screen height), and then update all the scales and axes accordingly. Now add the following:

```

this.buttons = d3.select('body')
.append('div')
.classed('buttons', true)
.selectAll('.button');

this.buttons.data(this.scenes)
.enter()
.append('button')
.classed('scene', true)
.text(d => d.label)
.on('click', d => this.loadScene(d))
.on('touchstart', d => this.loadScene(d));

this.words = d3.select('body').append('div');
this.words.classed('words', true);
this.loadScene(this.scenes[0]);
}; // This is the end of the addUI() method

```

This adds buttons to the page and sets the `click` and `touchstart` events to fire a method for loading in each scene.

We also create a `div` element to house our descriptions and assign that to the

words' internal property. Lastly, we load the first entry state, which has nothing selected.

We're done with `addUIElements()`. It's time to add a few more functions--we want some functions to select individual bars, and we want some way of clearing that selection:

```
buttonPrisonPopulationChart.clearSelected = function clearSelected() {
  d3.timeout(() => {
    d3.selectAll('.selected').classed('selected', false);
  }, this.transitionSpeed);
};

buttonPrisonPopulationChart.selectBars = function selectBars(years) {
  this.clearSelected();
  d3.timeout(() => {
    d3.select('.bars').selectAll('.bar')
      .filter(d => years.indexOf(Number(d.year)) > -1)
      .classed('selected', true);
  }, this.transitionSpeed);
};
```

Here, we simply remove the `selected` class from all bars in our `clearSelected()` function and assign the same class to the bars when we pass a date range in `selectBars()`; pretty straightforward. We select the bars in a `d3.timeout()` call so that we're synchronized with the entry and exit bar transitions in `prisonChart.update()`.

Lastly, we need to write our `loadScene()` method:

```
buttonPrisonPopulationChart.loadScene = function loadScene(scene) {
  const range = d3.range(scene.domain[0], scene.domain[1]);
  this.update(this.data.filter(d => range.indexOf(Number(d.year)) > -1));

  this.clearSelected();

  if (scene.selected) {
    const selected = scene.selected.range
      ? d3.range(...scene.selected.range) : scene.selected;
    this.selectBars(selected);
  }

  this.words.html(scene.copy);

  d3.selectAll('button.active').classed('active', false);
  d3.select((d3.event && d3.event.target) ||
    this.buttons.node()).classed('active', true);
};
```

In the preceding code, we first calculate the range of bars that we'll pass to `update` to set the new bars and axes. Then, if the selected `scene` property has a `range` property, we get an array of values using `d3.range`; otherwise we assume that it's

already an array of values to be selected, which we then pass to `selectBars()`. If there's no selected bars, then it clears all the selected bars. We set the description to the copy provided by our JSON file, set all the buttons to inactive and, lastly, set our selected button to active.

Finally, we export our chart as such:

```
| export default buttonPrisonPopulationChart;
```

Let's do our HTML layout using CSS. Create a new file, `lib/chapter5/prisonChart.css`. Add the following:

```
.selected {
  fill: red;
}

.buttons {
  display: flex;
}

.buttons button {
  flex-grow: 1;
  color: white;
  background: black;
  border: 2px solid white;
  transition: .5s background;
}

.buttons button.active {
  background: steelblue;
}

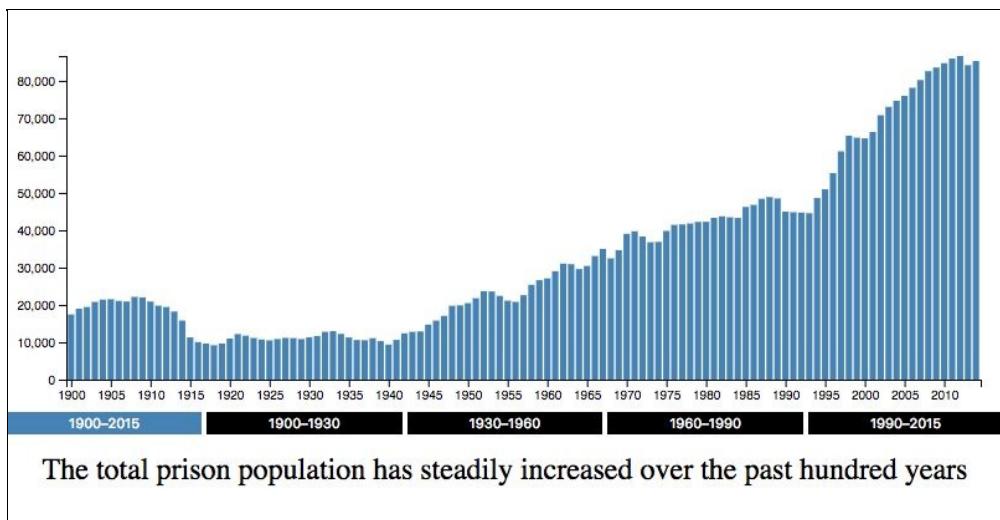
.words {
  padding: .5em;
  text-align: center;
  font-size: 1.5em;
}
```

This makes them stretch across the page and adds a basic CSS transition to when a button is selected. Import this into our `prisonchart.js` file at the top, as follows:

```
| import './prisonChart.css';
```

Once again, we use Webpack's CSS and style loaders to pull our styles into our JavaScript module.

Your chart should now look like this:



And there you have it, your first interactive data visualization!

Behaviors

In the last section, we made an *exploratory* 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 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 with both a mouse and touch device. The two currently supported behaviors are drag and zoom, 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 change? It involves a bit more work on the user's behalf, but it also gives them the ability to freely navigate through the chart, which may be desirable in some circumstances.

Let's extend our `prisonChart` object again. Comment out everything in `chapter5/index.js` and add the following:

```
import draggablePrisonChart from './draggableChart';
(async (enabled) => {
  if (!enabled) return;
  await draggablePrisonChart.init();
  draggablePrisonChart.addDragBehavior();
})(true);
```

Now, create a new file in `lib/chapter5`, called `draggableChart.js`:

```
import * as d3 from 'd3';
import PrisonPopulationChart from './prisonChart';

const draggablePrisonPopulationChart = Object.create(PrisonPopulationChart);
draggablePrisonPopulationChart.addDragBehavior = function addDrag() {};
```

This is the same way we started the other chart.

Inside of our `addDrag` function, add the following:

```
this.x.range([0, this.width * 4]);
this.update();

const bars = d3.select('.bars');
bars.attr('transform', 'translate(0,0)');
```

We set the `x` scale range to four times the screen size and update our chart to get the initial draw as well as set our axes. We also set an initial translate value on the bars, as we will use this value to ensure that it moves correctly.

You may have noted that there's an unpleasant flash while the axes update. There are a few ways we can get around that--we can update `prisonChart` to call our axis functions outside of the `init()` method, or we could hide the chart initially using CSS, making it



visible once the axes have updated. I'll leave this as an exercise for the reader.

Next, add the following, still inside our `addDrag` function:

```
const dragContainer = this.container.append('rect')
    .classed('bar-container', true)
    .attr('width', this.svg.node().getBBox().width)
    .attr('height', this.svg.node().getBBox().height)
    .attr('transform', `translate(${this.margin.left}, ${this.margin.top})`);
    .attr('x', 0)
    .attr('y', 0)
    .attr('fill-opacity', 0);

const xAxisTranslateY = d3.select('.axis.x').node().transform.baseVal[0].matrix.f;
```

We add a new, invisible element on top of everything, and we get the x axis translate value so that we can keep it at a constant Y position when we drag it. To do that, we get the x axis group element and then look at its `transform.baseVal[0].matrix` properties. The `e` property corresponds to y translate, and the `f` property corresponds to x translate.



There used to be a helpful function called `d3.transform` that would make this really easy. Alas, it has been removed in D3 v4.

Next, we set up our actual drag behavior and call it:

```
const drag = d3.drag().on('drag', () => {
  const barsTranslateX = bars.node().transform.baseVal[0].matrix.e;
  const barsWidth = bars.node().getBBox().width;
  const xAxisTranslateX = d3.select('.axis.x').node()
    .transform.baseVal[0].matrix.e;
  const dx = d3.event.dx;

  if (barsTranslateX + dx < 0 && barsTranslateX + dx >
    -barsWidth + this.innerWidth()) {
    bars.attr('transform', `translate(${barsTranslateX + dx}, 0)`);
    d3.select('.axis.x').attr('transform',
      `translate(${xAxisTranslateX + d3.event.dx}, ${xAxisTranslateY})`);
  }
});

dragContainer.call(drag);
```

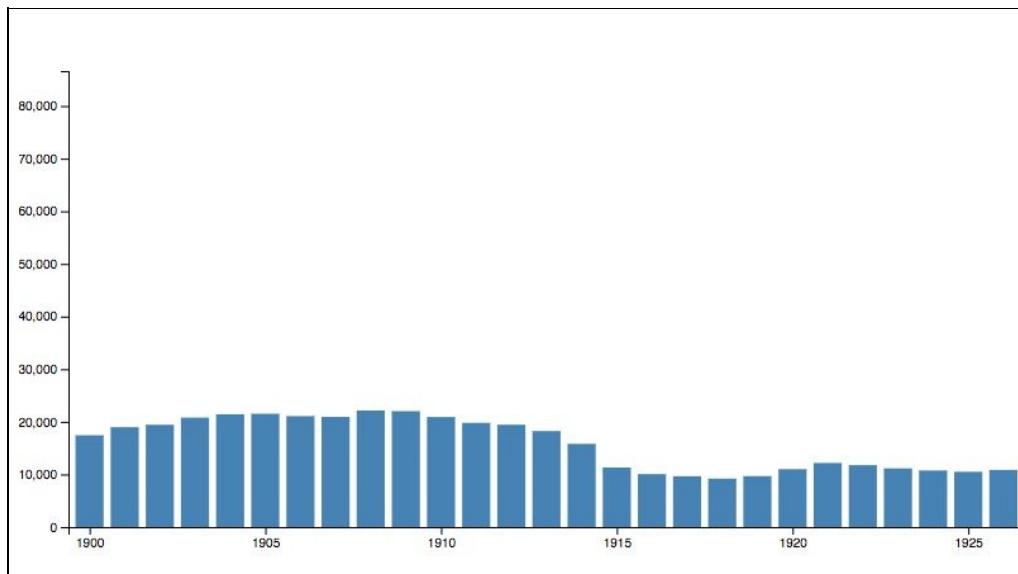
We attach the drag behavior to our `dragContainer` element. When a user drags a top of that element, it fires a drag event, which we use to set translate our bars and x axis horizontally. The `d3.event` object contains `dx` and `dy`, which are the horizontal and vertical distances dragged, respectively. The if-statement is used to prevent

us from dragging past or ahead of our bars, meaning that they'll stop one dragged to the y axis or the last bar is dragged to the right of the x axis.

Outside of `addDrag()`, add the following to export our chart:

```
| export default draggablePrisonPopulationChart;
```

Save, hit refresh, and you'll see your new chart:



Brushes

A brush behavior is similar to drag, but it is used to capture selections. Instead of moving items, a brush draws a box that selects items that it touches. You can find documentation in the `d3-brush` package.

To create a new brush, we'd call `d3.brush()` and then call it on an element. It emits start, brush, and end events, each receiving `selection`, `sourceEvent`, and `target` objects as part of `d3.event`.

It's time for an example!

We're going back yet again to our all-singing, all-dancing prison population graph. This is the last example to use it, I promise. We will let the user zoom in on a group of bars by selecting them with a brush, zooming out on rightclick.

Begin by commenting out everything in `chapter5/index.js` and adding the following, just like the last few examples:

```
import brushableChart from './brushableChart'; (async (enabled) => {
  if (!enabled) return;
  await brushableChart.init();
  brushableChart.addBrushBehavior();
})(true);
```

Create a new file at `lib/chapter5/brushablechart.js`. Add the following, like before:

```
import * as d3 from 'd3';
import interactivePrisonChart from './buttonChart';

const brushablePrisonPopulationChart = Object.create(interactivePrisonChart);
brushablePrisonPopulationChart.addBrushBehavior = function () {
  this.brush = d3.brush();
  this.container.append('g')
    .classed('brush', true)
    .call(this.brush
      .on('brush', this.brushmove.bind(this))
      .on('end', this.brushend.bind(this)));
  this.update();
};
```

The first part is just like before; the `addBrushBehavior()` method then sets up the brush behavior. We create a new group element for our brush, append it to our

chart, and attach event listeners. We then call `update()` to populate the chart.

Create a new function after `addBrushBehavior()`:

```
brushablePrisonPopulationChart.brushmove = function () {
  const e = d3.event.selection;
  if (e) {
    d3.selectAll('.bar').classed('selected', d =>
      e[0][0] <= this.x(d.year)
      && this.x(d.year) <= e[1][0]
    );
  }
};
```

Here, we get the dimensions and coordinates of the area that was brushed.

`d3.event.selection` contains two arrays, one containing the points where the brush started and another array containing the coordinates where it ended. We then grab the x-values from each point to get the range of bars we've selected. We then add the `.selected` class to each of them.

When the user finishes dragging the brush, the end event is fired; this callback handles that:

```
brushablePrisonPopulationChart.brushend = function () {
  if (!d3.event.sourceEvent) return; // Only transition after input.
  if (!d3.event.selection) return; // Ignore empty selections.
  const selected = d3.selectAll('.selected');
  const data = selected.data();

  // Clear brush object
  d3.select('g.brush').call(d3.event.target.move, null);

  // Zoom to selection
  if (data.length >= 2) {
    const start = data.shift();
    const end = data.pop();

    this.update(this.data.filter(d =>
      d3.range(start.year, end.year + 1).indexOf(Number(d.year)) > -1));

    const 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. First, we clear the brush overlay by calling `brush.move()` with `null` as its second argument, then we figure out the first and last elements

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 `buttonchart`. Lastly, we add a hit box, which we'll use to listen to the `contextmenu` event (`contextmenu` being the rightclick variant of the `click` event).

Lastly, we need an event handler for when the user rightclicks on the chart. Add this to the end of `brushableChart.js`:

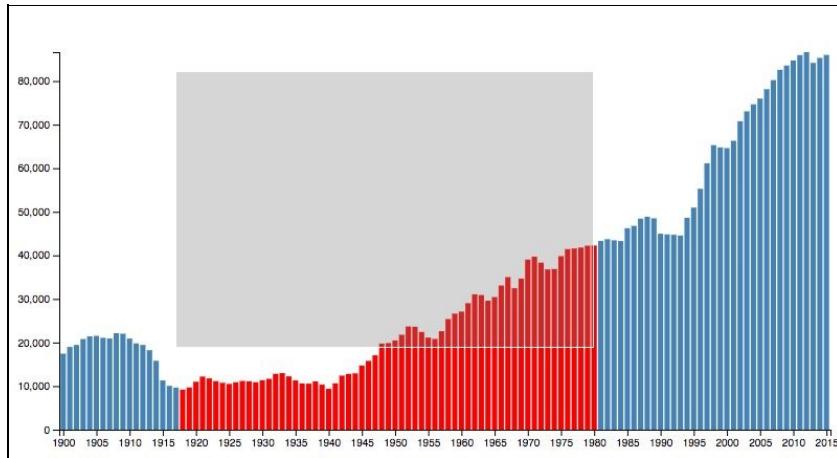
```
brushablePrisonPopulationChart.rightclick = function () {
  d3.event.preventDefault();
  this.clearSelected();
  this.update();
  this.svg.select('.hitbox').remove();
};
```

All this does is prevents the normal context menu from popping out, clears the selected bars, and runs an update using the entire dataset. We also remove the hitbox.

Finally, export it and save:

```
| export default brushablePrisonPopulationChart;
```

When you select some elements, the chart will look like this:



It'll then *zoom* in to those bars by running `update()`, with the data of the bars selected.

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, that includes pinch-to-zoom!

Remember that map from *Chapter 4, Making Data Useful*--the one with the rendition flights? Let's add the zoom and pan behavior to it!

You attach the zoom behavior to an element, which then fires an event when the user interacts with that element in a zoom-y way. This results in a transform value, which you can use to either apply a transform to the container element, or use for reprojecting geometry. We'll start with the latter.

Comment out everything in `chapter5/index.js` and add the following:

```
import { addZoomBehavior } from './zoomMap';

(async (enabled) => {
  if (!enabled) return;
  addZoomBehavior();
})(true);
```

Next, create `lib/chapter5/zoomMap.js` and add the following:

```
import * as d3 from 'd3';
import '../chapter4';

const zoomMap = {};

const NE_SCALE = 200;
const projection = d3.geoEquirectangular()
  .center([-50, 56])
  .scale(NE_SCALE);
```

We import the entirety of `chapter4/index.js` here, which only had the `geoDemo` IIFE remaining. In this instance, we designed things pretty badly, because it means that we don't have access to any of our chart object's internal state anymore. We can either go back and have `chapter4/index.js` `export geoDemo`, or we can throw caution to the wind and just work with what we have, applying the zoom behavior to an existing chart after it's rendered. Even though the first

option is far better in terms of general maintenance, let's instead do the latter - it's pretty straightforward and demonstrates that you can use D3 to manipulate SVG elements that have been created elsewhere.

That said, we need to replicate our projection from earlier so that we can accurately reproject our updated paths. The preceding projection is the same as we have in `chapter4/index`.

Let's add the zoom behavior now. Insert the following in `zoomMap.js`:

```
zoomMap.addZoomBehavior = () => {
  const chart = d3.select('#chart');

  const zoom = d3.zoom()
    .scaleExtent([0.5, 2])
    .on('zoom', zoomMap.onZoom);

  const center = projection(projection.center());

  chart.call(zoom)
    .call(zoom.transform, d3.zoomIdentity
      .translate(center[0], center[1])
    );
};
```

We instantiate the zoom behavior and define the minimum and maximum scale values using `scaleExtent()` before attaching a zoom event callback. We then calculate the center of the projection by supplying the original center values to our projection, which we'll use in the next line when we `call()` the zoom behavior on our chart. Note how we actually use `.call()` twice--the second time, we use our zoom behavior's transform method to set the initial zoom transform, to which we supply the center of our projection. If we don't do this, the initial interaction will be jumpy as the initial reprojection will not be calibrated to the chart's initial projection.



Note that in D3 v3, you would set the initial zoom vector using `zoom.scale()` and `zoom.translate()`; these have been removed in favor of the transform syntax used previously.

Next, let's flesh out the event callback:

```
zoomMap.onZoom = () => {
  const { x, y, k } = d3.event.transform;
  projection
    .scale(k * NE_SCALE)
    .translate([x, y]);
```

```

    d3.selectAll('path')
      .attr('d', d3.geoPath().projection(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 zoom transform properties are supplied via `d3.event.transform`; we destructure them into constants `x` (x position), `y` (y position), and `s` (scale). We update our projection to reflect the values provided by the behavior, and then use our updated projection to update the `d` value for all the paths and the start and end values of all of our flights.

Hit save and you should have a ludicrously-slow zooming and panning map! Wow, that's really unperformant and awful!

Doing what we did with complex Natural Earth geometry and reprojecting on every tick is awful for performance. You will, quite infrequently, need to do everything you did in the last example to get this behavior working on a chart, but it's good to know because it exposes the ability of `d3-zoom` to programmatically zoom via transforms. Let's change our approach and just transform the parent container.

In `chapter5/index`, comment out the following:

```
| zoomMap.addZoomBehavior();
```

Now, add the following:

```
| zoomMap.addZoomBehaviorTransform();
```

Save and head back to `zoomMap.js`:

```

zoomMap.addZoomBehaviorTransform = () => {
  const chart = d3.select('#chart');

  const zoom = d3.zoom()
    .scaleExtent([0.5, 2])
    .on('zoom', zoomMap.onZoomTransform);

  chart.call(zoom);
};

```

Wow, that's quite a lot easier, isn't it? Same as before, just not having to set up

projections or initial transforms. Let's move on to our event handler:

```
zoomMap.onZoomTransform = () => {
  const container = d3.select('#container');
  container.attr('transform', d3.event.transform);
  container.selectAll('path')
    .style('stroke-width', 1 / d3.event.transform.k);
};
```

This is even simpler! We select the container and set its transform property to the contents of `d3.event.transform` (which outputs as a value you can pass to the transform property when output as a string). We even get a bit fancy and scale the stroke width of our geometry to reflect the scale value; in reality, it can be a one-liner.

Hit save and try your map again. That's miles better, isn't it? D3 gives you a lot of freedom in implementing things; understanding which is the best route for any particular project comes from practice and knowing your audience.



However, generally, it's better not to reproject if you don't need to, it's awful for performance.

Do you even need interaction?

To end on a somewhat pedantic note--before adding any interaction to any project, ask whether it actually needs it. If you are in the cohort of folks using D3 as a means to generate data visualizations for news media, realize that the number of people who actually use interactive graphic functionality is depressingly small; most readers will simply flick right past the elaborately-designed interactive graphics you spent all that time on. New York Times Deputy Graphics Editor Archie Tse provides three rules for visual storytelling:

- If you make the reader click or do anything other than scroll, something spectacular has to happen.
- If you make a tooltip or rollover, assume that no one will ever see it. If content is important for readers to see, don't hide it.
- When deciding whether to make something interactive, remember that getting it to work on all platforms is expensive.

The source is at <https://github.com/archietse/malofiej-2016/blob/master/tse-malofiej-2016-slides.pdf>.

In fact, my team at the Financial Times very seldom make our graphics interactive as it tends to detract from what the graphic is actually trying to convey:

FT Interactive News Head Martin Stabe says information visualization is meant to clarify data, but too much interactivity hinders understanding by transferring responsibility from the designer to the reader to work out the important points.

Source (paywalled) is at <https://www.ft.com/content/c62b21c6-7feb-11e6-8e50-8ec15fb462f4>.

One solution around this is to attach interactivity to browser scroll events, leading the user through the data as they scroll, but even this is somewhat fraught with peril as it's somewhat difficult to get right, and some people just resent having their scrolling hijacked. The WSJ's scroll-watcher library might be one possibility if you intend to implement a "scrollyteller"-style interactive graphic. See <https://github.com/WSJ/scroll-watcher>.

Lastly, even if you're not designing for a news audience, realize that it's worth being very careful about how you build interfaces. Although you, as a developer, have quite high computational skills, the vast majority of people don't - in fact, a recent study looking into computing skills among OECD nations found that a full 24% are totally unable to use a computer effectively, with another 16% having only the most basic of skills (Refer to <https://www.nngroup.com/articles/computer-skills/>). Particularly if you're working in government and using D3 to convey information important to your constituents, keeping interaction to a minimum is often a very good idea.

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 a supremely awesome bar graph. Well done!

In this chapter, we've animated with transitions, interpolators, and timers. We then learned the difference between explanatory and exploratory visualizations, and used interactivity to create the former. We then made some exploratory visualizations 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 hierarchical layouts. Combining both the skills you've learned in this chapter and the next one will mean that you're able to produce some truly fantastic charts. Hope you're ready--this is when stuff starts getting really cool!

Hierarchical Layouts of D3

Part of the process of learning to do cool things with D3 is looking at examples on bl.ocks.org. This is great, as you have a living, forkable code base that you can modify into something specific to your use case, but part of what makes learning D3 difficult is that quite often, these rely on layouts, which are effectively algorithms that restructure data in a certain way. These can seem really opaque if you don't know how they work.



D3 v4 alert! Everything in this and the next chapters has changed significantly with D3 v4. Now more than ever, make sure that you pay attention to which version of D3 an example uses when looking at them online.

Over the course of the next two chapters, we'll be diving into layouts and building a ludicrous number of quick charts. First, we start with hierarchical layouts, which assumes a data structure with parent and child nodes; in the next chapter, we'll look at some of the other layouts, which include things such as pie charts and force-directed diagrams. Let's begin.

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.

However, we usually use layouts for more complex visualizations, such as drawing a force-directed graph or a tree, for instance. In these cases, layouts help us to separate calculating coordinates from putting pixels on the screen. This not only makes our code cleaner, but it also lets us reuse the same layouts for vastly different visualizations.

Built-in layouts

By default, D3 comes with around a dozen 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, whereas hierarchical layouts generally present data in a tree-like structure.

The normal (non-hierarchical) layouts are as follows:

- Histogram
- Pie
- Stack
- Chord
- Force

The hierarchical layouts are as follows:

- Tree
- Cluster
- Tree map
- Partition
- Pack

We will create a boatload of examples over the next two chapters. We start with hierarchical layouts (located in the `d3-hierarchy` module) because they're the most consistent in terms of data structure, which looks something like this:

```
{ "name": "Tywin Lannister",
  "children": [
    { "name": "Jamie Lannister",
      "children": [
        { "name": "Joffery Baratheon" },
        { "name": "Myrcella Baratheon" },
        { "name": "Tommen Baratheon" }
      ],
      "name": "Tyrion Lannister",
      "children": []
    }
  ]
}
```

An hierarchy can include lots of other data, but the main thing is it needs to be a nested series of objects, with this implicit hierarchy defined via an attribute (by

default, named `children`). We can also use a helper function in d3-hierarchy, `d3.stratify()`, to convert a flat hierarchy like the following to the structure needed by our hierarchical layouts:

```
[  
  {"name": "Westeros", "parent": ""},  
  {"name": "Tywin Lannister", "parent": "Westeros"},  
  {"name": "Jamie Lannister", "parent": "Tywin Lannister"},  
  {"name": "Tyrion Lannister", "parent": "Tywin Lannister"},  
  {"name": "Joffery Baratheon", "parent": "Jamie Lannister"},  
]
```

Note that each of these need one (and only one) root node, or rather, a node that has no parent and everything stems from. If you don't have a root node (or multiple root nodes), D3 will throw an exception, and nothing will work. In this case, we call our parent node `Westeros` because that's where all these folks live.



Be forewarned that we're using a dataset containing a boatload of data from a particular epic fantasy television show, up to and including season 6. It's possible that you might totally spoil a few of the surprises in the show if you dig too deeply into the data, so I'd recommend that you get caught up beforehand if this concerns you.

If you find Game of Thrones annoying, I'm really, really sorry.

We will use a third-party library here to generate legends. Install it using the following command:

```
| npm install d3-svg-legend --save
```

We could make a bunch of legends ourselves and it'd be a fun exercise and stuff, but really you can probably figure out what's going on already -- our new library uses a scale to create a bunch of SVG squares and text labels. Yawn, right?

It's been a while since we added anything to `lib/common/index.js` -- we're going to put a lot of stuff in there because we share a lot of functionalities across charts; add the following:

```
export const colorScale = d3.scaleOrdinal()  
  .range(d3.schemeCategory20);  
export const heightOrValueComparator =  
  (a, b) => b.height - a.height || b.value - a.value;  
export const valueComparator = (a, b) => b.value - a.value;
```

```
export const fixateColors = () => {};
export const addRoot = () => {};
export const tooltip = () => {};
export const descendantsDarker = () => {};
```

We start by creating a color scale using the `category20` 20-color scheme, as well as a few custom comparators we'll use over and over. We also create a bunch of empty functions, which we'll flesh out in just a second.

Time to start building charts! We will use `chartFactory` to create a base chart that we then will use throughout. Create a new folder in `lib/` called `chapter6/` and then create a new file in `chapter6/` called `index.js`. Add the following to it:

```
import * as d3 from 'd3';
import * as legend from 'd3-svg-legend';
import chartFactory, {
  fixateColors,
  addRoot,
  colorScale as color,
  tooltip,
  heightOrValueComparator,
  valueComparator,
  descendantsDarker,
} from '../common';
```

In the preceding code, we import D3, our 3rd-party legend tool, and a boatload of stuff from our common functionality module.

Next, add the following:

```
const westerosChart = chartFactory({
  margin: { left: 50, right: 50, top: 50, bottom: 50 },
  padding: { left: 10, right: 10, top: 10, bottom: 10 },
});
```

This adds a padding object to our chart's prototype, which we'll use to make things a bit easier. Next, we will add a data loading function, which differentiates between a CSV or JSON file:

```
westerosChart.loadData = async function loadData(uri) {
  if (uri.match(/\.csv$/)) {
    this.data = d3.csvParse(await (await fetch(uri)).text());
  } else if (uri.match(/\.json$/)) {
    this.data = await (await fetch(uri)).json();
  }
  return this.data;
};
```

Yay, more `async/await` fun; nice and straightforward. Parsed as CSV or JSON,

assign it to the local context, then return that value. We will add a function that we'll use to initialize our charts and pass arguments to them:

```
westerosChart.init = function initChart(chartType, dataUri, ...args) {  
  this.loadData(dataUri).then(data => this[chartType].call(this, data, ...args));  
  this.innerHeight = this.height - this.margin.top - this.margin.bottom - this.padding  
  this.innerWidth = this.width - this.margin.left - this.margin.right -  
  this.padding.left - this.padding.right;  
};
```

As our `loadData()` function is marked as `async`, it always returns a promise. We could make `init()` an `async` function and just await the results from `loadData()`, but let's just keep it simple and use `.then()` to resolve the promise returned by `loadData()`. This way the rest of `init()` doesn't have to wait until the data's resolved.



*The `...args` bit in the constructor's arguments is a new feature in ES2015 called the **rest parameter**. It collects every argument after the ones specifically defined as an array. We then use another new ES2015 feature, 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 rest of `init()` just sets up two properties, `innerHeight` and `innerWidth`, that we'll use to make adding legends and axes easier and less verbose.

Hierarchical layouts

All hierarchical layouts are based on an abstract hierarchy layout designed for representing hierarchical data: a recursive parent-child structure. As mentioned earlier, imagine a tree or an organization chart.

All the code for the `partition`, `tree`, `cluster`, `pack`, and `treemap` layouts are defined in the `d3-hierarchy` module, and they all follow similar design patterns. The layouts are very similar and share lots of common aspects; so, to avoid repeating ourselves a whole bunch, we'll look at the common stuff first, and then focus on the differences.

First of all, we need some hierarchical data. In the book repository, I've provided a file called `GoT-lineages-screentimes.json` that contains all the character lineages (by father), the amount of time spent on screen, and the number of episodes a character is in. We'll use genealogies of each character as a way of creating a hierarchy, then time on screen to size various page elements in layouts that necessarily need them (I don't bother in the tree or cluster examples to keep them shorter and easier to read).

Each item in `GoT-lineages-screentimes.json` is an object in the following format:

```
{  
  "itemLabel": "Lysa Arryn",  
  "fatherLabel": "Hoster Tully",  
  "screentime": 16.3,  
  "episodes": 5  
},
```

As our data isn't already in a tree format, we will use `d3.stratify()` to create a hierarchy from it, using `itemLabel` for each node's unique ID, and `fatherLabel` as the ID of each node's parent.

Tree the whales!

Let's start with the most basic of hierarchical charts -- a tree! Create a new function and fill it with the following:

```
westerosChart.tree = function Tree(_data) {
  const data = getMajorHouses(_data);
  const chart = this.container;
  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);
  const root = stratify(data);
  const layout = d3.tree()
    .size([
      this.innerWidth,
      this.innerHeight,
    ]);
}
```

We use our next-to-be-written `getMajorHouses()` function to filter out characters who don't have a `fatherLabel` property and whose `itemLabel` isn't set as anybody's `fatherLabel` property. We then create a new stratify object and set its `parentId()` accessor function to each item's `fatherLabel` and the `id()` accessor to each item's `itemLabel`. We're able to do the latter with this dataset because we know that each `itemLabel` is distinct; if this was not the case (for instance, if you had a dataset where there were a few people named John Smith in it), you'd need to use a property other than the person's name as the unique identifier tying everything together.

We then pass our data to our newly defined stratify function to create our root. We also instantiate our tree layout, setting the size of our inner dimensions.

Time to write our data munging functions. First, we will write the `addRoot` function we imported from common. Go back to common and update `addRoot` to resemble the following:

```
export function addRoot(data, itemKey, parentKey, joinValue) {
  data.forEach((d) => { d[parentKey] = d[parentKey] || joinValue; });
  data.push({
    [parentKey]: '',
    [itemKey]: joinValue,
  });
  return data;
}
```

The `addRoot()` function takes four arguments: the data, each item's ID property name, each item's parent ID property name, and a value to set the latter to if one isn't found. It does that, then adds this root element to the data array before returning it.

While we're here, let's fill out `fixateColors` and create a new function called `uniques()`:

```
export const uniques = (data, name) => data.reduce(
  (uniqueValues, d) => {
    uniqueValues.push(
      (uniqueValues.indexOf(name(d)) < 0 ? name(d) : undefined));
    return uniqueValues;
  }, [])
  .filter(i => i); // Filter by identity

export function fixateColors(data, key) {
  colorScale.domain(uniques(data, d => d[key]));
}
```

This sets the domain of our exported color scale to whatever unique values we pass to it. Our `uniques()` function works by creating an array, adding items as defined by the function passed as the second argument, or skipping the item if another already exists in the results array.

Let's also add a function to get the name of a house by walking backward up the tree and grabbing the old ancestor's last name:

```
export const getHouseName = (d) => {
  const ancestors = d.ancestors();
  let house;
  if (ancestors.length > 1) {
    ancestors.pop();
    house = ancestors.pop().id.split(' ').pop();
  } else {
    house = 'Westeros';
  }
  return house;
};
```

Also, let's add a function to return a list of all the major house names:

```
| export const houseNames = root =>      root.ancestors().shift().children.map(getHouseName
```

Next, we'll remove any items that don't have much in terms of lineage data. After our imports section in `chapter6/index`, add the following:

```
| const getMajorHouses = data =>
```

```

    addRoot(data, 'itemLabel', 'fatherLabel', 'Westeros')
    .map((d, i, a) => {
      if (d.fatherLabel === 'Westeros') {
        const childrenLen = a.filter(
          e => e.fatherLabel === d.itemLabel).length;
        return childrenLen > 0 ? d : undefined;
      } else {
        return d;
      }
    })
    .filter(i => i);

```

We pass our data to `addRoot` in order to add `Westeros` as a root node, and from the results of that, filter out any nodes that have `Westeros` as a father and haven't set themselves as anyone's father.

Now, we need to draw stuff using the tree layout. Add this to `westeroschart.tree`:

```

const links = layout(root)
  .descendants()
  .slice(1);

fixateColors(getHouseNames(root), 'id');
const line = d3.line().curve(d3.curveBasis);

chart.selectAll('.link')
  .data(links)
  .enter()
  .append('path')
  .attr('fill', 'none')
  .attr('stroke', 'lightblue')
  .attr('d', d => line([
    [d.x, d.y],
    [d.x, (d.y + d.parent.y) / 2],
    [d.parent.x, (d.y + d.parent.y) / 2],
    [d.parent.x, d.parent.y]],
  )));

```

Okay, this is where stuff starts to get interesting. We pass our root object created earlier to the layout generator. This gives us our layout data object. This has a method called `descendants()` that returns all the descendants of the root node, starting with the root node, and continuing down in topological order (we slice off the root node here because it doesn't have links). We then fixate our colors using the functions we wrote a bit earlier. We also create a new line generator with all the default settings and the `curveBasis` interpolator to make it look nice.

We then create a new selection, join our data to it, and append path elements for each connection. We then use each datum to manually draw paths between the child and parent nodes, putting in two midway points for the curve.

The way these links are generated is sort of less intuitive than it



could be, which the D3 maintainers seem to realize needs improvement. Watch issue #27 on d3-shape at github.com/d3/d3-shape/issues/27, as this may change.

Next, let's draw circles for each node:

```
const nodes = chart.selectAll('.node')
  .data(root.descendants())
  .enter()
    .append('circle')
    .attr('r', 4.5)
    .attr('fill', getHouseColor)
    .attr('class', 'node')
    .attr('cx', d => d.x)
    .attr('cy', d => d.y);
```

We get all the nodes using `root.descendants()` again and pass that to a new selection, appending circles to each node. We fill them using `getHouseColor` (which we'll define in a moment) and setting the radius to 4.5. Let's define `getHouseColor` now, at the top of `chapter6/index`, right after `getMajorHouses()`:

```
const getHouseColor = (d) => {
  const ancestors = d.ancestors();
  let house;
  if (ancestors.length > 1) {
    ancestors.pop();
    house = ancestors.pop().id.split(' ').pop();
  } else {
    house = 'Westeros';
  }
  return color(house);
};
```

We take the ID, split it into an array using the space character, then take the last item in the array. We're assuming here that the person's last name has a space before it and is at the end of the person's name (*Sammy Davis Jr.*, for instance, would cause issues). If the item only has one ancestor (itself), we set it to `westeros`, as that's our root node. We then take whatever value we have and supply it to our color scale.

Next, we'll add our legend. Go back to `westeroschart.tree` and add the following:

```
const legendGenerator = legend
  .legendColor()
  .scale(color);

this.container
  .append('g')
  .attr('id', 'legend')
```

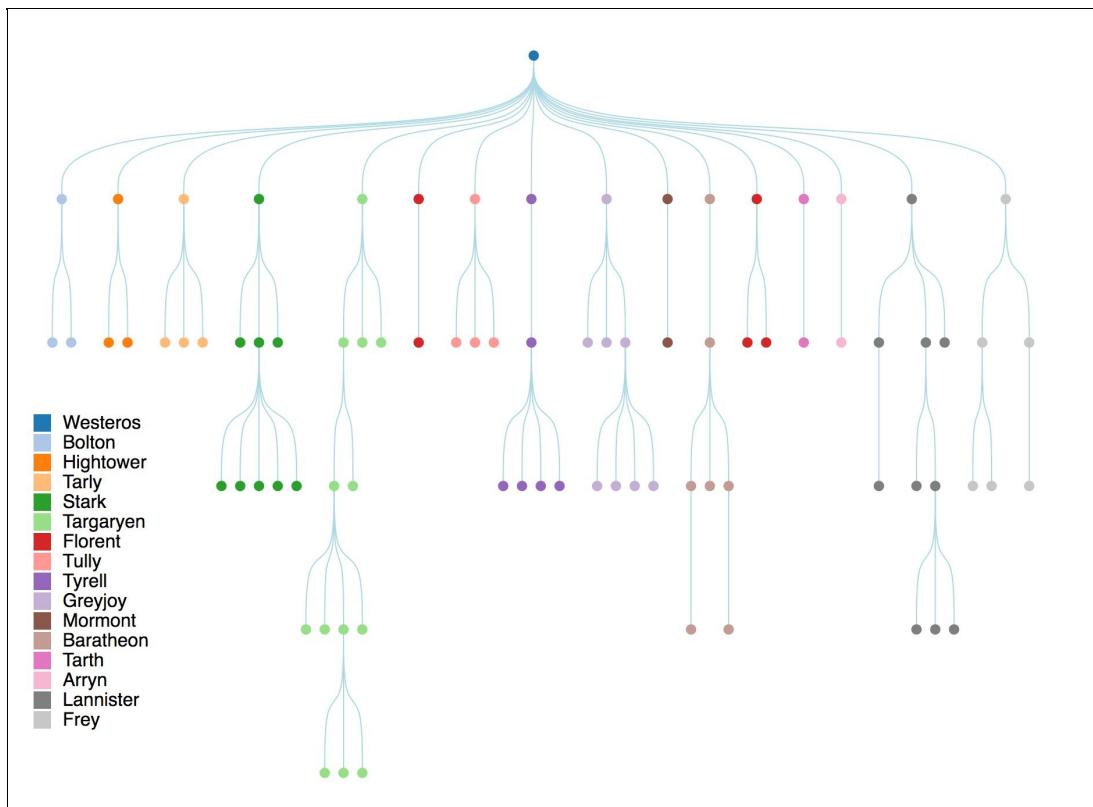
```
|     .attr('transform', &grave;translate(0, ${this.innerHeight / 2})&grave;)
|     .call(legendGenerator);
```

This creates our legend generator and appends it to our container object, halfway down the height of the chart. We supply it our color scale so that it knows what items to put in the legend.

Go back to `lib/main.js` and modify it to resemble the following:

```
| import '../styles/index.css';
| import westerosChart from './chapter6/index';
| westerosChart.init('tree', 'data/GoT-lineages-screentimes.json');
```

Switch to your browser (run `npm start` beforehand if your local development server isn't running), and you should see something like the following:



We're not done yet though, there aren't any labels!

We could do something like append text labels to each node (which is easy, considering each datum contains the node's `itemLabel` property), but there are a lot of nodes and the labels will likely collide into each other on most screen sizes. Instead, it's time to trundle to tooltip town! We will use tooltips to show our

users what each item in our charts represents when they hover over it.

Go to `common/index.js` and update the `tooltip()` function with the following:

```
export function tooltip(text, chart) {
  return (selection) => {
    function mouseover(d) {
      const path = d3.select(this);
      path.classed('highlighted', true);

      const mouse = d3.mouse(chart.node());
      const tool = chart.append('g')
        .attr('id', 'tooltip')
        .attr('transform',
          `translate(${mouse[0] + 5},${mouse[1] + 10})`);

      const textNode = tool.append('text')
        .text(text(d))
        .attr('fill', 'black')
        .node();

      tool.append('rect')
        .attr('height', textNode.getBBox().height)
        .attr('width', textNode.getBBox().width)
        .style('fill', 'rgba(255, 255, 255, 0.6)')
        .attr('transform', 'translate(0, -16)');

      tool.select('text')
        .remove();

      tool.append('text').text(text(d));
    }

    function mousemove() {
      const mouse = d3.mouse(chart.node());
      d3.select('#tooltip')
        .attr('transform',
          `translate(${mouse[0] + 15},${mouse[1] + 20})`);
    }

    function mouseout() {
      const path = d3.select(this);
      path.classed('highlighted', false);
      d3.select('#tooltip').remove();
    }

    selection.on('mouseover.tooltip', mouseover)
      .on('mousemove.tooltip', mousemove)
      .on('mouseout.tooltip', mouseout);
  };
}
```

What we do here is create a factory function that returns a new function taking a selection argument. We then attach a bunch of event handlers to each state of the cursor hovering over an element -- when it's over the element, the tooltip element is populated with the specified text and the coordinates of the mouse. When it leaves the targeted element, it hides the tooltip. When it moves around

atop of the targeted element, the tooltip's coordinates are updated.

Go back to `westeroschart.tree` and add the following to the bottom:

```
| nodes.call(tooltip(d => d.data.itemLabel, this.container));
```

If you remember, our tooltip generator takes two arguments, an accessor function and a target container. We supply this to the generator, then call it on each of our nodes.

There you have it! Tooltips!

Muster the cluster!

Another type of similar diagram is a dendrogram, which uses D3's `cluster` layout and puts all leaf nodes of a tree at the same depth. Let's create that now.

Comment out the `westerosChart.init()` line in `main.js` and add this beneath it:

```
| westerosChart.init('cluster', 'data/GoT-lineages-screentimes.json');
```

Go back to `chapter6/index` and add the following:

```
westerosChart.cluster = function Cluster(_data) {
  const data = getMajorHouses(_data);
  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);

  const root = stratify(data);

  fixateColors(houseNames(root), 'id');

  const layout = d3.cluster()
    .size([
      this.innerWidth - 150,
      this.innerHeight,
    ]);
  const links = layout(root)
    .descendants()
    .slice(1);
}
```

This should look familiar already--we get our data, create a stratify generator, then use it on our data. We then create a cluster layout, give it a size (though, here we subtract 150 pixels for the legend), then generate the links using `layout.descendants()`.

Still in `westerosChart.cluster`, add the following:

```
const line = d3.line().curve(d3.curveBasis);

this.container.selectAll('.link')
  .data(links)
  .enter()
  .append('path')
  .attr('fill', 'none')
  .attr('stroke', 'lightblue')
  .attr('d', d => line([
    [d.y, d.x],
    [(d.y + d.parent.y) / 2, d.x],
    [(d.y + d.parent.y) / 2, d.parent.x],
  ]));
```

```
|   [d.parent.y, d.parent.x]],  
|});
```

This is the same path, drawing bit of code from before, except we've swapped the x and y values so that it displays horizontally instead of vertically.

Next, add circles for the nodes:

```
const nodes = this.container.selectAll('.node')  
  .data(root.descendants())  
  .enter()  
  .append('circle')  
  .classed('node', true)  
  .attr('r', 5)  
  .attr('fill', getHouseColor)  
  .attr('cx', d => d.y)  
  .attr('cy', d => d.x);
```

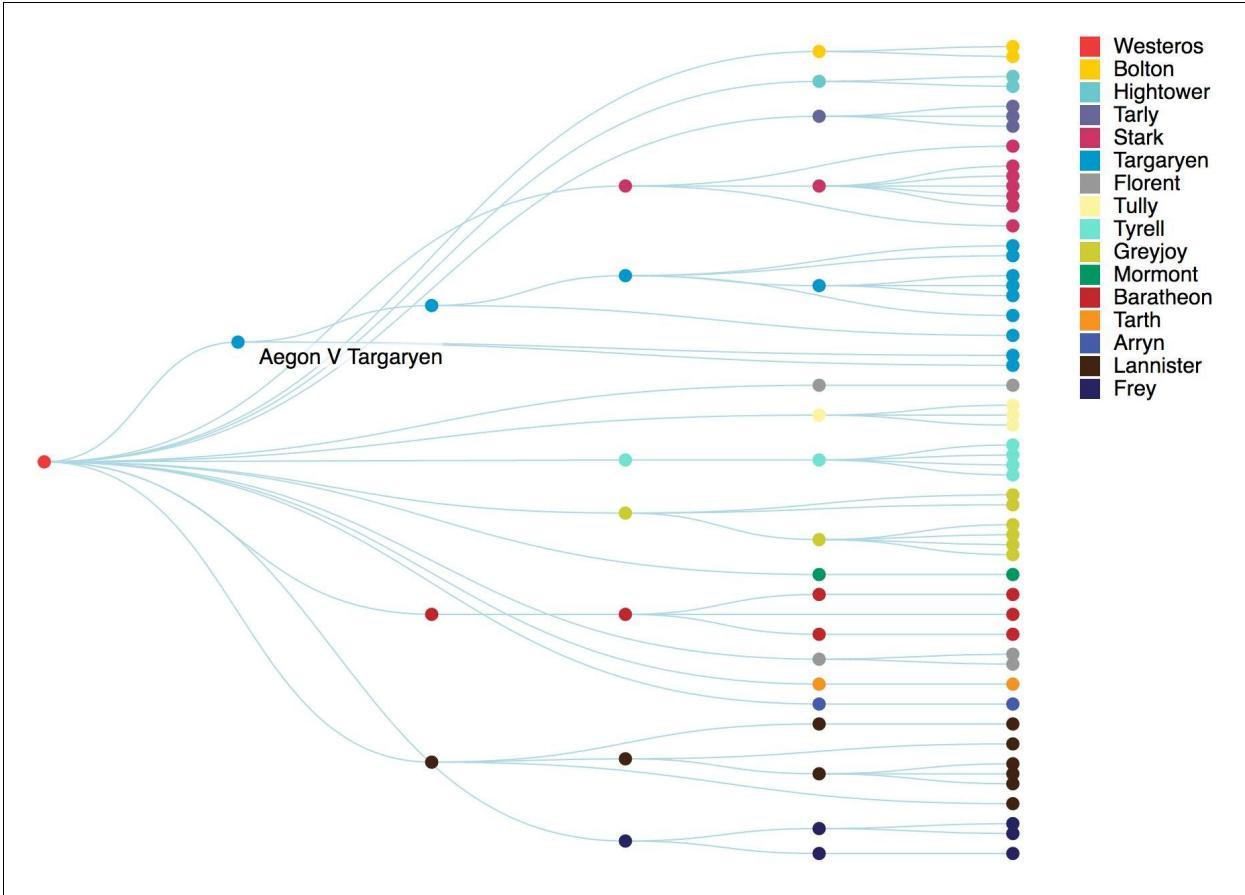
Then, add a legend:

```
const l = legend  
  .legendColor()  
  .scale(color);  
  
this.container  
  .append('g')  
  .attr('id', 'legend')  
  .attr('transform', `translate(${this.innerWidth - 100}, 0)`)  
  .call(l);
```

Lastly, call the tooltip factory, and we're done!:

```
| nodes.call(tooltip(d => d.data.itemLabel, this.container));
```

Click on save, and you'll get something like this:



Money for nothing, treemaps for free (maps)

Despite the similar name, treemaps bear little visual resemblance to the tree layout we used earlier; instead, they divide a tree into rectangular regions. This requires us to have a dimension to our data; in this case, we'll size the treemap regions based on screen time, nesting each region into its parent. As such, the size of each parent will be the sum of its children, plus its own value.

This is all going to start looking really similar, so we will start writing all of our common functions now. In `common/index`, add this function, which will give us increasingly deeper gradations based on a hierarchy:

```
| export const descendantsDarker = (d, color, invert = false, dk = 5) =>
|   d3.color(
|     color(
|       d.ancestors()[d.ancestors().length - 2].id.split(' ').pop()
|     )[invert ? 'brighter' : 'darker'](d.depth / dk);
```

This takes a datum, a color scale, and then three options: a Boolean to make the scale go brighter instead of darker, a numerical multiplier for the strength of the effect, and a string denoting the property that contains the ID field, defaulting it to `itemLabel`.

Brilliant! Let's do the standard set up we did in the last few charts. Go to `lib/main.js` and comment out the last `westerosChart.init()` call while adding this new one:

```
| westerosChart.init('treemap', 'data/GoT-lineages-screentimes.json');
```

Navigate back to `chapter6/index.js` and add this:

```
| westerosChart.treemap = function Treemap(_data) {
|   const data = getMajorHouses(_data);
|   const stratify = d3.stratify()
|     .parentId(d => d.fatherLabel)
|     .id(d => d.itemLabel);
|
|   const root = stratify(data)
|     .sum(d => d.screentime)
|     .sort(heightOrValueComparator);
```

```
|   const cellPadding = 10;
|   const houseColors = color.copy().domain(houseNames(root));
| }
```

This does everything we did before, except now we sort by the `heightOrValue` comparator we created when I had you paste those empty functions into common/index. If you need a refresher, all it does is this:

```
| (a, b) => b.height - a.height || b.value - a.value
```

This means that it works regardless of whether a height or a value is supplied. We also get the sum of each person's screentime and also all of their children's screentime. You must sort and sum your root before passing it to your treemap layout generator; otherwise, it won't know how to calculate the size of the regions.

We also define a constant for our cell padding (which we set to 10 because I think it helps to have really big region borders, given this dataset) and create a new color scale just for house names. We need to have two color scales here because otherwise our legend will get really big and unwieldy -- we just want it to display the categorical colors associated with each house.

Next, we add our treemap layout generator:

```
| const layout = d3.treemap()
|   .size([
|     this.innerWidth - 100,
|     this.innerHeight,
|   ])
|   .padding(cellPadding);
```

This should look pretty similar to the other ones. We just add our cell padding and make it a hundred pixels less wide so that we have room for the legend.

We now call our layout on our data, which adds the relevant properties to it; this mutates our hierarchy, so we don't need to assign the result to anything:

```
| layout(root);
```

I know, the functional programmer in me is crying on the inside too.

Now, we draw all the nodes, which are just rectangles:

```
| const nodes = this.container.selectAll('.node')
```

```

    .data(root.descendants().slice(1))
    .enter()
    .append('g')
      .attr('class', 'node');

  nodes.append('rect')
    .attr('x', d => d.x0)
    .attr('y', d => d.y0)
    .attr('width', d => d.x1 - d.x0)
    .attr('height', d => d.y1 - d.y0)
    .attr('fill', d => descendantsDarker(d, color, true, 3));

```

Since we don't need the root node in this instance, we slice it off when we call `root.descendants()`. We then append a new group for each node. Then, to each of those, we append a rectangle, setting its positioning and sizing accordingly. We then use our newly created `descendantsDarker` function to give us a sequence of similar colors.

Here's both the legend and tooltip in one go:

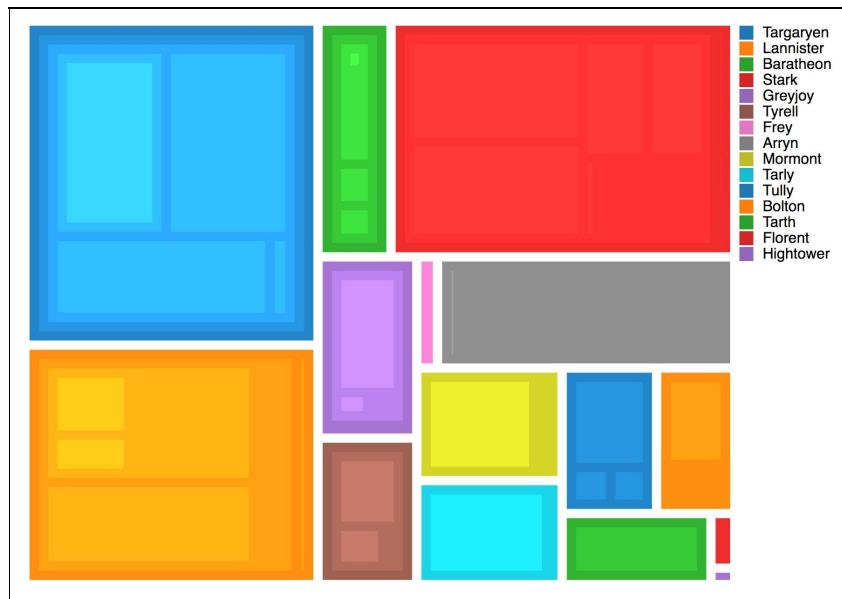
```

this.container
  .append('g')
    .attr('id', 'legend')
    .attr('transform',
      `translate(${this.innerWidth - 100}, ${cellPadding})`)
    .call(legend.legendColor().scale(houseColors));

  nodes.call(tooltip(d => d.data.itemLabel, this.container));

```

Click on save and we're done with our third chart. We're now halfway there!



Smitten with partition

These next few charts are just going to fly by; they're really similar and we don't have anything else in terms of common functions to write.

Adjacency diagrams are similar to treemaps, but are intended to fill the available space. They tend to be useful for visualizing things such as disk usage. They are created by the partition layout.

Here's our example code in full:

```
westerosChart.partition = function Partition(_data) {
  const data = getMajorHouses(_data);
  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);

  const root = stratify(data)
    .sum(d => d.screentime)
    .sort(heightOrValueComparator);

  const layout = d3.partition()
    .size([
      this.innerWidth - 100,
      this.innerHeight,
    ])
    .padding(2)
    .round(true);

  layout(root);

  const nodes = this.container.selectAll('.node')
    .data(root.descendants().slice(1))
    .enter()
    .append('g')
    .attr('class', 'node');

  nodes.append('rect')
    .attr('x', d => d.x0)
    .attr('y', d => d.y0)
    .attr('width', d => d.x1 - d.x0)
    .attr('height', d => d.y1 - d.y0)
    .attr('fill', d => descendantsDarker(d, color, false));

  this.container
    .append('g')
    .attr('id', 'legend')
    .attr('transform', `translate(${this.innerWidth - 100}, 0)`)

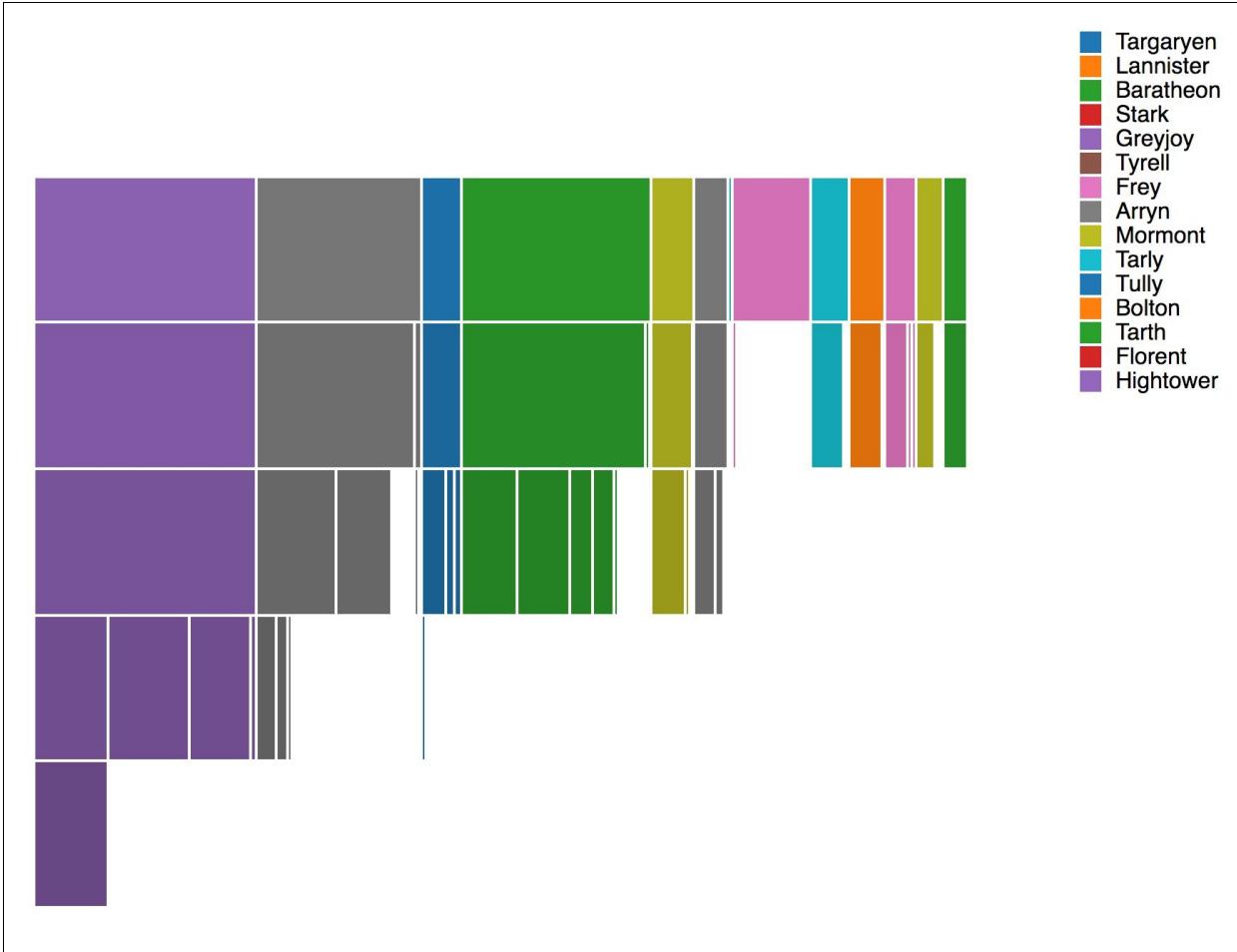
    .call(legend.legendColor().scale(houseColors));

  nodes.call(tooltip(d => d.data.itemLabel, this.container));
};
```

Any of this look new or unfamiliar to you? I am guessing not by now! Like treemaps, we slice off the root node and get rectangle sides by subtracting the top left corner from the bottom right corner. Add this to `main.js` as before, commenting out the other times we've initialized `westerosChart`:

```
westerosChart.init('partition', 'data/GoT-lineages-screentimes.json');
```

It comes out looking like this:



Kinda cool, I guess? I like treemaps more, personally.

Pack it up, pack it in, let me begin...

The pack layout produces charts similar to treemaps, but using round nodes. This is probably the best of the last three charts in this section for representing this hierarchy -- remember how we needed to use a ludicrous amount of padding in our treemap to make the parent nodes more visible? Since circle packing diagrams use more space, the parent nodes are more visible and that relationship is more pronounced.

Like before, comment out the other `westerosChart.init()` lines in `main.js` and add this: `westerosChart.init('partition', 'data/GoT-lineages-screentimes.json');`

Next, add the following to `chapter6/index.js`:

```
westerosChart.pack = function Pack(_data) {
  const data = getMajorHouses(_data);

  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);

  const root = stratify(data)
    .sum(d => d.screentime)
    .sort(valueComparator);

  const houseColors = color.copy().domain(houseNames(root));
  fixateColors(data, 'itemLabel');

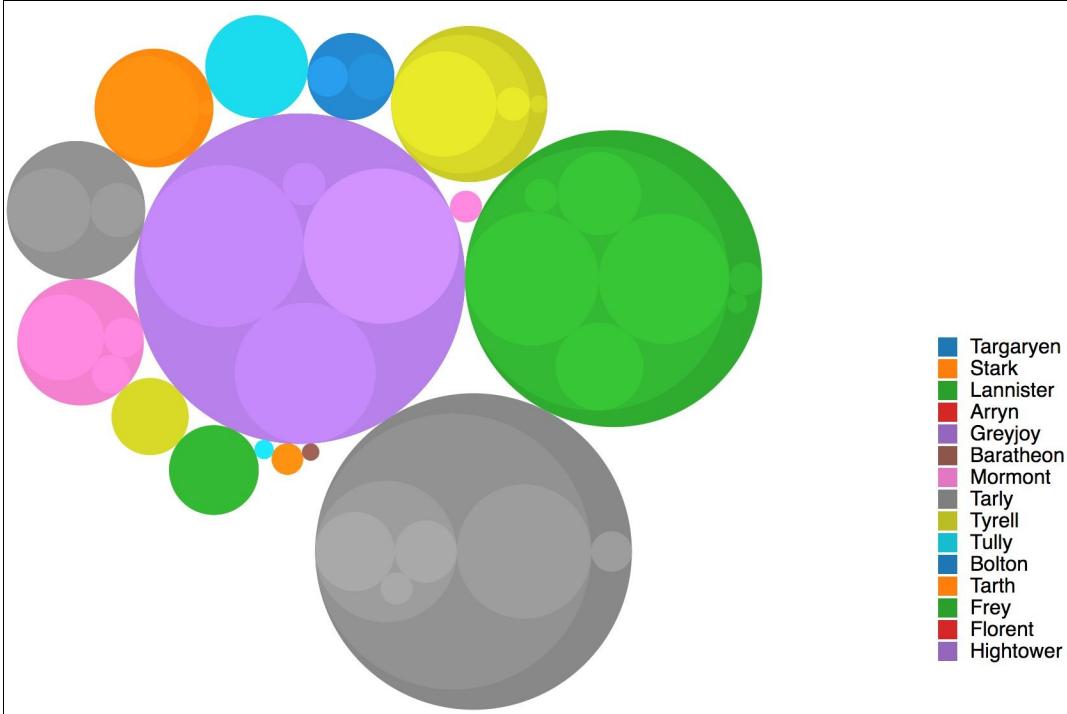
  const layout = d3.pack()
    .size([
      this.innerWidth - 100,
      this.innerHeight,
    ]);
  layout(root);

  const nodes = this.container.selectAll('.node')
    .data(root.descendants().slice(1))
    .enter()
    .append('circle')
      .attr('class', 'node')
      .attr('cx', d => d.x)
      .attr('cy', d => d.y)
      .attr('r', d => d.r)
      .attr('fill', d => descendantsDarker(d, color, true, 5));

  this.container
    .append('g')
      .attr('id', 'legend')
      .attr('transform',
        `translate(${this.innerWidth - 100}, ${this.innerHeight / 2})`)
```

```
.call(legend.legendColor().scale(houseColors));
  nodes.call(tooltip(d => d.data.itemLabel, this.container));
};
```

This is pretty much the exact same as the last chart, but we put the legend closer to the bottom and are appending circles instead of rectangles. You should now have something like this:



Bonus chart! Sunburst radial partition joy!

Oh, you thought we were done? Let's squeeze one more chart out of `d3-hierarchy` before moving on.

If you remember, the partition chart was a bit funky, largely because it's mainly used in datasets where only the leaf nodes (that is, the outermost nodes without children) have a value. Since some of the parents in our dataset have `screentime` values, this sort of distorts it and makes it look odd. We will re-render that, but make it all cool and circular this time.

You know the drill. `main.js`:

```
| westerosChart.init('radialPartition', 'data/GoT-lineages-screentimes.json');
```

In `chapter6/index.js`:

```
| westerosChart.radialPartition = function RadialPartition(_data) {
  const data = getMajorHouses(_data)
    .map((d, i, a) => Object.assign(d, {
      screentime: a.filter(v =>
        v.fatherLabel === d.itemLabel).length ? 0 : d.screentime,
    }));
  const radius = Math.min(this.innerWidth, this.innerHeight) / 2;
};
```

We start by creating a radius that's half of the smallest dimension and mapping our data so that any nodes that are ancestors get their `screentime` set to zero.

Continue by creating our root and house color scale as per before:

```
| const stratify = d3.stratify()
  .parentId(d => d.fatherLabel)
  .id(d => d.itemLabel);

  const root = stratify(data)
    .sum(d => d.screentime)
    .sort(null);

  const houseColors = color.copy().domain(root.ancestors().shift()
    .children.map(d => d.id.split(' ')[d.id.split(' ').length - 1]))
  );
```

Next, we create our partition layout, but set it to half the width and height of parent chart. This is because we're calculating it radially, so the output will be double whatever value we set here:

```
const layout = d3.partition()
  .size([
    this.innerWidth / 2,
    this.innerHeight / 2,
  ])
  .padding(1)
  .round(true);
```

Next, we create an x-scale and an arc generator:

```
const x = d3.scaleLinear()
  .domain([0, radius])
  .range([0, Math.PI * 2]);

const arc = d3.arc()
  .startAngle(d => x(d.x0))
  .endAngle(d => x(d.x1))
  .innerRadius(d => d.y0)
  .outerRadius(d => d.y1);
```

We set our domain to our radius and our range to double the value of PI. Again, this is a standard circle stuff.

Populate the layout and create your data join:

```
layout(root);

const nodes = this.container
  .append('g')
  .attr('class', 'nodes')
  .attr('transform',
    `translate(${this.innerWidth / 2}, ${this.innerHeight / 2})`)
  .selectAll('.node')
  .data(root.descendants().slice(1))
  .enter()
  .append('g')
  .attr('class', 'node');
```

Next, append paths, supplying the arc generator for the `d` value:

```
nodes.append('path')
  .attr('d', arc)
  .attr('fill', d => descendantsDarker(d, color, false));
```

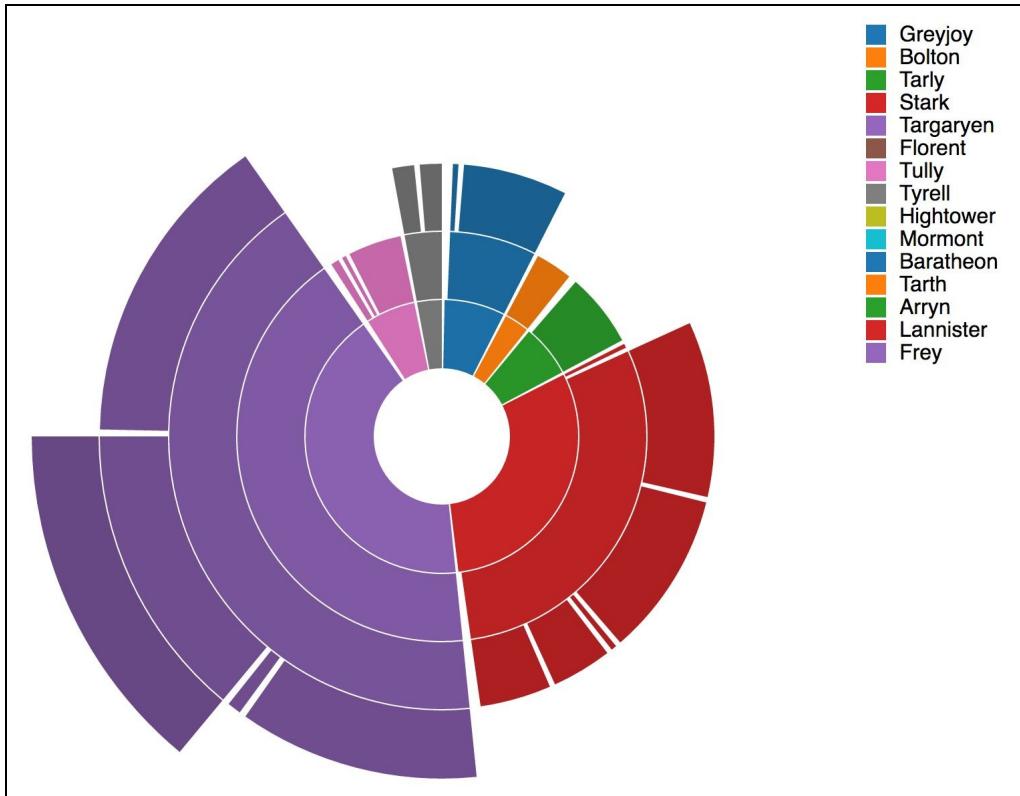
Lastly, we set up the legend and tooltip:

```
this.container
  .append('g')
  .attr('id', 'legend')
```

```
.attr('transform', &grave;translate(${this.innerWidth - 100}, 0)&grave;)
    .call(legend.legendColor().scale(houseColors));

nodes.call(tooltip(d => d.data.itemLabel, this.container));
```

Finally! Let's see how it looks:



Must say, that's definitely an improvement over the rectangular version.

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. We've used the hierarchical layouts to create a boatload of different charts, without much more than a few lines of code differing between implementations.

These charts are so simple, we could have really extended our base object a lot to be far more abstracted; so by doing things such as fixating colors and adding the legend during the `init()` method -- between `chapter6/index` and `common/index` -- we've added probably around 600 lines, and we could probably reduce that by a couple of hundred. However, it's also useful to create each chart separately to reinforce the workflow needed by the hierarchical layouts, as I've done in this chapter.

In the next chapter, we'll look at a few more layouts. They'll look pretty familiar to the hierarchical ones in terms of how we write them, but we'll be flipping around the data a fair bit more to accommodate the different styles output each nonhierarchical layout provides. By the end of this book, you'll be through all the basics. Hopefully, you're starting to get excited by all the new things you can do.

The Other Layouts

In the preceding chapter, we used the hierarchical layouts in `d3-hierarchy` to create a nice selection of charts. In this chapter, we will look at other layouts in D3 v4, which are spread across a few different modules. Instead of calling them *normal* or *nonhierarchical* layouts, let's just call them *the other* layouts because there's not a lot of similarities between them.

Hoorah for modular code

Let's get to it. Go to `lib/main.js` and add the following line to the imports section:

```
import westerosChart from './chapter7/index';
```

Create a folder called `chapter7` and create a file called `index.js`. Add the following code to it:

```
import * as d3 from 'd3';
import * as legend from 'd3-svg-legend';
import baseChart from '../chapter6/';
import './chapter7.css';
import {
  colorScale as color,
  tooltip,
  connectionMatrix,
  uniques,
} from './common';
```

We import our `westerosChart` object from the last chapter as `baseChart`, so we can just extend it here.

That's all there is to it; we're ready for our first chart.

When the moon hits your eye (chart), like a big pizza pie (chart)

Pie charts are a very common way of presenting simple quantitative data, but they have their own limitations--people have greater difficulty perceiving the size of an area when it's in a circular shape, and you really need to make sure that the wedges are ordered descending clockwise from the top in order to be able to adequately compare them. That said, they're common enough that knowing how to make them is an incredibly useful skill as anyone doing data visualization work will at some point be asked for one.

D3's pie chart layout resides in the `d3-shape` package and is somewhere between the layouts of the last chapter and the line generators of [Chapter 3, Shape Primitives of D3](#). We create a pie chart layout and pass it an array of numbers and then pass that to an arc generator to create our pie chart. Let's get to it.

We start by filtering out people who have less than 60 minutes of screen time and creating a pie generator, using the `value()` method to let the pie generator know which field to use to size the pie chart slices. We then create an arc generator, setting the outer radius to a quarter of the screen size. We then create a group and translate it into the center of the screen:

```
westerosChart.pie = function Pie(_data) {
  const data = _data.filter(d => d.screentime > 60);
  const pie = d3.pie().value(d => +d.screentime);
  const arc = d3.arc()
    .outerRadius(this.innerWidth / 4)
    .innerRadius(null);

  const chart = this.container.append('g')
    .classed('pie', true)
    .attr('transform', `translate(${this.innerWidth / 2}, ${this.innerHeight / 2}`);
};

export default westerosChart;
```

Next, we render the slices. Add the following to our `pie()` function:

```
const slices = chart.append('g')
  .attr('class', 'pie')
  .selectAll('.arc')
  .data(pie(data).sort((a, b) => b.data.screentime - a.data.screentime))
  .enter()
```

```

    .append('path')
    .attr('d', arc)
    .classed('arc', true)
    .attr('fill', d => color(d.data.itemLabel));

```

Here, we pass our data to the pie generator, and then sort the object it returns using the `screentime` property (which now resides in each datum's data property since we're looking at the result of the layout). We then pass that to the `arc` generator and use that to generate our paths. We then pass our `itemLabel` (also in the `data` property) to the color scale.

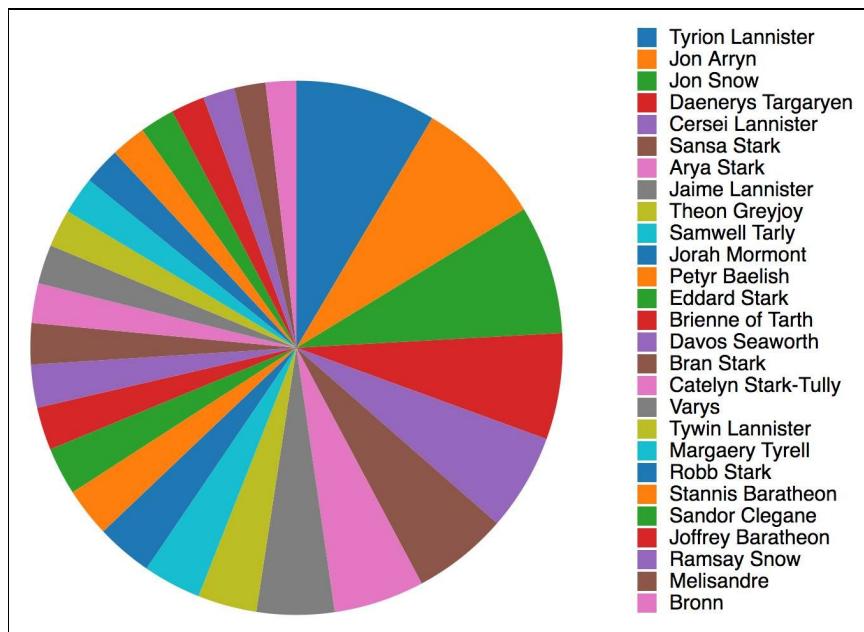
Let's add the tooltips and legend in one fell swoop:

```

slices.call(tooltip(d => d.data.itemLabel, this.container));
this.container
.append('g')
.attr('id', 'legend')
.attr('transform',
translate(${this.innerWidth - 150}, ${this.innerHeight / 2} - 250))
.call(legend.legendColor().scale(color));

```

Here's how it looks:



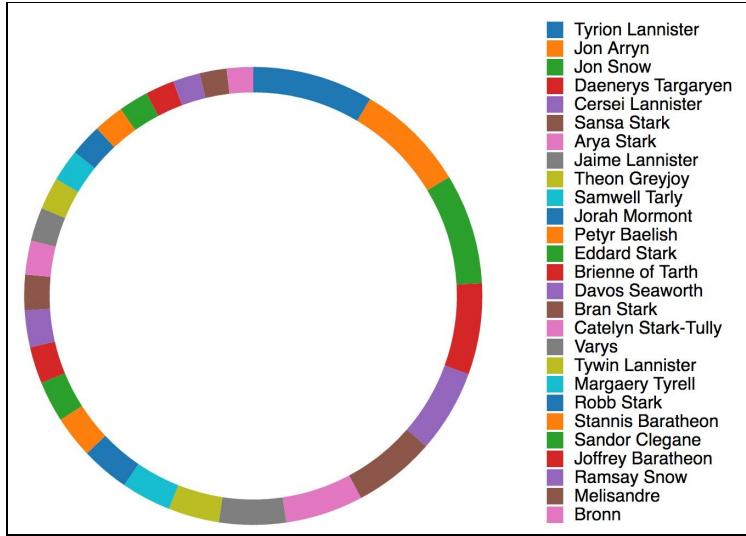
If you want to create a donut chart (that is, a pie chart with a hole in it), simply set the `arc.innerRadius` property to a value, as follows:

```

    .innerRadius(this.innerWidth / 4.5);

```

You will get something like this:



That's pretty much all there is to the pie layout. This is easy, right? You got this!

You might have noticed that this chart repeats a few colors as a result of our ordinal scale. This is because it has more than 10 items, and so our color scheme repeats itself. This wasn't a problem in the last chapter, because we relied on the data's natural hierarchy to provide depth to our color.



To get around this, we have a few options. We could try to use a 20-color categorical color scheme like `d3.schemeCategory20`, but that still wouldn't work because we have more than 20 items. We could approach it in a similar fashion to the last chapter, and color code individual characters based on their house color, but in the process make it more difficult to discern which pie segment corresponds to whom. In reality, the best approach is also the simplest--limit the number of pie segments to the ten characters with the most screen time, creating a separate "other" slice to aggregate everybody who didn't make the cut. Which approach is most appropriate depends on your audience and your data--just remember that the more slices you have in a pie chart, the harder it is for the reader to interpret the size of any given segment.

I'll leave fixing this as an exercise for the reader given we have a boatload of charts to get through this chapter and pies are by far the least interesting of the bunch.

Histograms, Herstograms, Yourstograms, and Mystograms

Another simple layout is the histogram, which simplifies creating bar charts when they're in a continuous series. We can also use it for binning a series of values so that we don't have to do as many gymnastics with `Array.prototype.reduce` and `Array.prototype.map`.

In this instance, we will create an ordinal scale of episodes and seasons and use that to create a histogram. In doing so, we're going to use a new dataset, which I've included in the `data/` directory, `GoT-deaths-by-season.json`. This includes all the deaths in the show in the following format:

```
{
  "name": "Will",
  "role": "Ranger of the Night's Watch",
  "death": {
    "season": 1,
    "episode": 1
  },
  "execution": "Beheaded for desertion by Ned Stark",
  "likelihoodOfReturn": "0%"
},
```

The only data we're really concerned with here is the `death` object, which we'll use to create an ordinal scale.

Start by resetting `main.js` by commenting out all the `westeroschart` lines, then add the following:

```
| westerosChart.init('histogram', 'data/GoT-deaths-by-season.json');
```

Go back to `chapter7/index.js` and add the following:

```
westerosChart.histogram = function histogram(_data) {
  const data = _data.data.map(d =>
    Object.assign(d, { death: (d.death.season * 100) + d.death.episode }))
  .sort((a, b) => a.death - b.death);
  const episodesPerSeason = 10;
  const totalSeasons = 6;
  const allEpisodes = d3.range(1, totalSeasons + 1).reduce((episodes, s) =>
    episodes.concat(d3.range(1, episodesPerSeason + 1).map(e => (s * 100) + e)), []);
```

This replaces the death object with a string where the season is multiplied by 100 and added to the episode number, then sorts all the data first by season, then by episode. It also creates an array of 60 elements of the format detailed above.



This last step is optional, but we want to show all the episodes (even if they don't have any deaths in them), so we populate the x-scale as per the following. This is a bit different to how histograms are generally used, which is with a continuous series. Alas, we don't really have any of those in this dataset, so this will have to suffice.

Next, we instantiate our x-scale:

```
const x = d3.scaleBand()  
  .range([0, this.innerWidth])  
  .domain(allEpisodes)  
  .paddingOuter(0)  
  .paddingInner(0.25);
```

We use an ordinal band scale here and set the domain to the `allEpisodes` array we have just created.

Next, we create our histogram layout generator, and supply it with the following data:

```
const histogram = d3.histogram()  
  .value(d => d.death)  
  .thresholds(x.domain());  
const bins = histogram(data);
```

We set the `value` accessor to return our `death` value and use the `histogram.thresholds()` method to set the extents of each bin. The `histogram.thresholds()` method expects an array containing a series of values defining the edges of the bin--the first bin is situated between the first and second array elements, second bin between the second and third elements, and so on.

This returns us an array of bins. Each *bin* is an array with all the corresponding data assigned to it, with a `length` property corresponding to the number of array elements, and `x0` and `x1` properties, corresponding to the edges of the bin as explained above. The lower bound (`x0`) is inclusive, whereas the upper bound (`x1`) is exclusive (except for the last bin).

Next, we create a y-scale:

```
const y = d3.scaleLinear()
  .domain([0, d3.max(bins, d => d.length)])
  .range([this.innerHeight - 10, 0]);
```

This is pretty straightforward; we get the maximum length of an item in our bins by running `d3.max()` on them, and set the range to 10 pixels less than the `innerHeight` (our x axis will be double-decked to accommodate the number of labels, and each line is 10 pixels high).

Time to add the bars!:

```
const bar = this.container.selectAll('.bar')
  .data(bins)
  .enter()
  .append('rect')
  .attr('x', d => x(d.x0))
  .attr('y', d => y(d.length))
  .attr('fill', 'tomato')
  .attr('width', () => x.bandwidth())
  .attr('height', d => (this.innerHeight - 10) - y(d.length));
```

We make everything 10 pixels shorter to accommodate the legend; in every other respect, it's like any bar chart. We use `bandwidth()` to set the width--if we weren't using an ordinal `bandScale` here, we could subtract `x0` from `x1` and pass that to the x-scale to get the width of each bar.

Almost done! We're going to add the x axis next:

```
const xAxis = this.container.append('g')
  .attr('class', 'axis x')
  .attr('transform', `translate(0, ${this.innerHeight - 10})`)
  .call(d3.axisBottom(x).tickFormat(
    d => `${(d - (d % 100)) / 100}E${d % 100}`);
  xAxis.selectAll('text')
  .each(function (d, i) {
    const yVal = d3.select(this).attr('y');
    d3.select(this).attr('y', i % 2 ? yVal : (yVal * 2) + 2);
  });
  xAxis.selectAll('line')
  .each(function (d, i) {
    const y2 = d3.select(this).attr('y2');
    d3.select(this).attr('y2', i % 2 ? y2 : y2 * 2)
  });
```

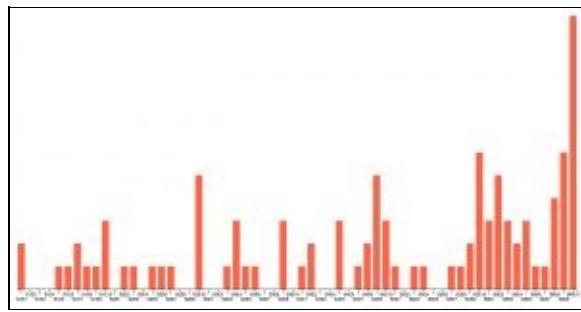
This is like any other x axis, except we're going a bit crazy here and making it double-decked by moving every other tick label down slightly more than twice its y-value (note that it's inside a group, so this value is relative to that) and extending its tick line accordingly. We use `d3.each` to run this operation on each

element in the selection, which sets the `this` context to the current element; because of this, we use regular functions instead of fat arrow, which preserves the context provided by `d3.each`.

Lastly, we set up our trusty tooltip generator:

```
| bar.call(tooltip((d) =>
|   &grave;${d.x0}: ${d.length} deaths&grave;, this.container));
```

Click on save, and you should have a nifty bar chart like this!:



Wow, season six goes in pretty hard, doesn't it?



Wait a minute, was this a proper use of a histogram?

Well, no, not really; a histogram is normally used to take a bunch of samples and break them into discrete chunks across a specified output range. It could be argued this would have made more sense, as a simple bar chart with an ordinal scale for the x-axis. However, in at least this particular case, using a histogram is perhaps a bit easier, as it does all the totaling for us, even if we did have to do a bit of contrived data wrangling to get the x-axis to make any sense. As a rule of thumb, if you're not entirely sure what your x-axis ticks should be, and you have a lot of data you want to display in a bar-chart-like manner, you should consider a histogram. If your data is already classified such that you could get away with using an ordinal scale for your x-axis, do that instead.

Striking a chord

The `chord` layout creates a circular diagram showing relations in a dataset. We will use yet another dataset here, found in the `data/` directory `stormofswords.csv`, from the Network of Thrones dataset available at <https://www.macalester.edu/~abeverid/thrones.html>.

This dataset was created by looking at the proximity of character names in the text of the book series in order to find the weight of each character's connection to the other characters. It is an ideal dataset for the next two examples, which look at arbitrary nonhierarchical connections between data.

Start by doing the *comment out the last example and add this* dance in `main.js`:

```
| westerosChart.init('chord', 'data/stormofswords.csv');
```

Go back to `chapter7/index` and scaffold out the new chart method:

```
| westerosChart.chord = function Chord(_data) {};
```

Nothing new. We will create an array of sources and links between them, assuming the strength of the connection is greater than `20`. Add this to the `chord` function:

```
| const minimumWeight = 20;
|   const majorLinks = _data.filter(d => +d.Weight > minimumWeight);
|   const majorSources = uniques(majorLinks, d => d.Source);
|   const data = majorLinks.filter(d =>
|     majorSources.indexOf(d.Target) > -1);
```

We filter out links that don't have a target to get our data array.

We will pass this to a function to get a connection matrix, which is an array of arrays. Each element in the array's arrays references the other values in a circular fashion. Let's fill out `connectionMatrix` in `common.js`:

```
| export function connectionMatrix(data, sourceKey = 'source', targetKey = 'target', valueKey = 'Weight') {
|   const nameIds = nameId(allUniqueNames(data, 'Source', 'Target'), d => d);
|   const uniqueIds = nameIds.domain();
|   const matrix = d3.range(uniqueIds.length).map(() => d3.range(uniqueIds.length).map(() => 0));
|   data.forEach((d) => {
|     matrix[nameIds(d[sourceKey])][nameIds(d[targetKey])] += Number(d[valueKey]);
```

```

    });
}

return matrix;
}

```

We create a scale of all the unique names in the data array. We then get an array of unique names by getting its domain. We then use `d3.range()` twice to get a matrix filled with arrays filled with 1s. We then replace each 1 with the value of the connection weight from the dataset, then return the matrix.

Go back to `chapter7/index` and add the following to the `chord` function:

```

const matrix = connectionMatrix(data, 'Source', 'Target', 'Weight');
const outerRadius = (Math.min(this.width, this.height) * 0.5) - 40;
const innerRadius = outerRadius - 30;

```

We create our matrix, then define an outer and inner radius.

Next, we create our chord layout generator, then an arc generator, then a ribbon generator:

```

const chord = d3.chord()
  .padAngle(0.05)
  .sortSubgroups(d3.descending);

const arc = d3.arc()
  .innerRadius(innerRadius)
  .outerRadius(outerRadius);

const ribbon = d3.ribbon()
  .radius(innerRadius);

```

Let's get this show on the road. First, append a group to our container, then pass our matrix to the chord layout:

```

const chart = this.container
  .append('g')
  .attr('class', 'chord')
  .attr('transform', `translate(${this.innerWidth / 2}, ${this.innerHeight / 2})`)
  .datum(chord(matrix));

```

We translate it to half the width and height because otherwise it would be in the top-left corner.

Next, we create groups for each of the items in our dataset:

```

const group = chart.append('g').attr('class', 'groups')
  .selectAll('g')
  .data(chords.groups)
  .enter()

```

```

    .append('g');

group.append('path')
  .style('fill', d => color(d.index))
  .style('stroke', d => d3.color(color(d.index)).darker())
  .attr('d', arc);

```

Lastly, we add our ribbons linking each of the groups:

```

const ribbons = chart.append('g').attr('class', 'ribbons')
  .selectAll('path')
  .data(chords => chords)
  .enter()
    .append('path')
    .attr('d', ribbon)
    .style('fill', d => color(d.target.index))
    .style('stroke', d => d3.color(color(d.index)).darker());

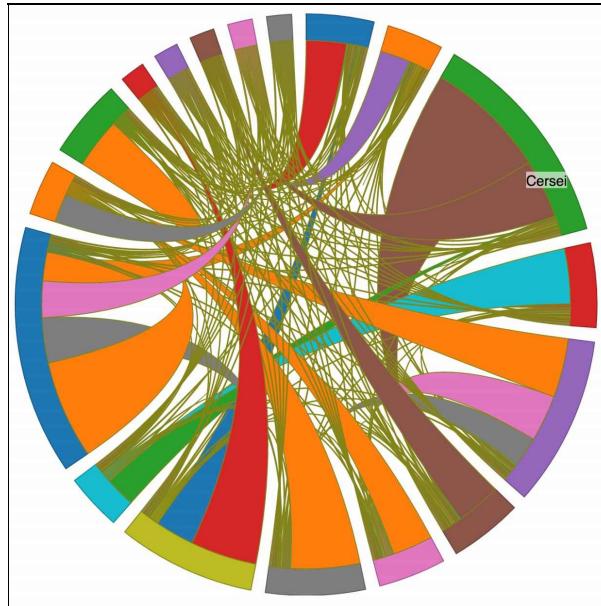
```

Finally, call the tooltip on both the ribbons and the groups:

```

ribbons.call(tooltip(d => majorSources[d.target.index], this.container));
group.call(tooltip(d => majorSources[d.index], this.container));

```



That's actually pretty cool!

May the force (simulation) be with you

One of the cooler layouts provided by D3 (in the `d3-force` package) is force simulation. This is actually really useful, as it allows us to position elements on the screen using simulated physical properties. This can be really useful when the positioning of an element doesn't have anything to do with its data, such as in a network diagram, where the linkages between items is more important than when they are on screen.

We will use the network dataset from the last example to create a network of connections in the show. Go to `main.js`, comment out the last example, and add this line:

```
| westerosChart.init('force', 'data/stormofswords.csv');
```

Go back to `chapter7/index` and add a new function enclosure:

```
westerosChart.force = function Force(_data) {
  const nodes = uniqueness(
    _data.map(d => d.Target)
      .concat(_data.map(d => d.Source)),
    d => d)
      .map(d =>
        ({ id: d, total: _data.filter(e => e.Source === d).length }));
  fixateColors(nodes, 'id');
}
```

We create an array of unique nodes of the format:

```
| {id: <character-name>, total: <total linkages>}
```

There isn't much more in terms of data in the dataset, and it's hard to join it to another dataset because it lacks character surnames. If we had more data we wanted to add (for instance, we created a lookup table for each node ID in the network dataset and then mapped that to a value in the other dataset), we could do it at this stage and attach it to each node. In this instance, we get a count of the number of linkages each node has, which we'll use to size our nodes.

We also then fixate our color scale and create an array of links connecting all the nodes together, with the weight depicting the strength of the relationship. We'll use this weight to define the thickness of the lines connecting each node.

Continuing to add to `westerosChart.force`:

```
| const links = _data.map(d =>
|   { source: d.Source, target: d.Target, value: d.Weight }));
```

We will first append all the nodes and links, then set up the simulation to move stuff around the screen. Add the following to `westerosChart.force` beneath the last line:

```
| const link = this.container.append('g').attr('class', 'links')
|   .selectAll('line')
|     .data(links)
|       .enter()
|         .append('line')
|           .attr('stroke', d => color(d.source))
|             .attr('stroke-width', d => Math.sqrt(d.value));
```

This puts a line on the screen linking each node. We take the square root of the value to get a value for the link thickness, and color the link so that it reflects the connection's source. Just beneath, add the following code:

```
| const radius = d3.scaleLinear().domain(
|   d3.extent(nodes, d => d.total)).range([4, 20]);
| const node = this.container.append('g').attr('class', 'nodes')
|   .selectAll('circle')
|     .data(nodes)
|       .enter()
|         .append('circle')
|           .attr('r', radius)
|             .attr('fill', d => color(d.id))
|               .call(d3.drag()
|                 .on('start', dragstart)
|                   .on('drag', dragging)
|                     .on('end', dragend));
```

We define a scale to size the radii, linearly scaling nodes between a radius of 4 and 20. We then add a bunch of circles to the screen and a bunch of event callbacks that we'll write in a little bit.

Put the tooltip on the nodes:

```
| node.call(tooltip(d => d.id, this.container));
```

Let's finally set up the simulation. This is really basic, and `d3-force` has a lot of properties we can tweak, but this will suffice for now:

```

| const sim = d3.forceSimulation()
|   .force('link', d3.forceLink().id(d => d.id).distance(200))
|   .force('charge', d3.forceManyBody())
|   .force('center',
|     d3.forceCenter(this.innerWidth / 2, this.innerHeight / 2));

```

This sets up the force simulation layout, with three forces: `link`, `charge` and `center`.

- The `link` force depicts the tension caused by a vector linking two bodies. This can be used to do things such as add elasticity to links, or use them to pull nodes closer together. We set the distance at 200 so that the nodes are a decent distance away from each other. If we wanted to rely more on the other forces, we wouldn't set distance because it's pretty powerful here.
- The `charge` force means that items on the screen have a simulated electric charge, using a *many body* simulation to calculate this. This is really useful, because it means we can cause elements to either attract or repulse each other, the latter of which is an effective way of ensuring that all data points are on screen and not overlapping each other. We instantiate the simulation with the default options here, so there's a slight negative charge causing nodes to repel each other a bit.
- Lastly, the `center` force causes an attraction to a central point as defined by the values in the constructor.

Next, add this:

```

| sim.nodes(nodes).on('tick', ticked);
|   sim.force('link').links(links);

```

This adds the nodes and links to the simulation.

`.nodes()` sets the following properties on each object passed to it:

- `index`: the node's zero-based index
- `x`, `y`: the current x-and y-position of the node
- `vx`, `vy`: the current v-and y-velocity of the node

You can also set the `fx` and to ensure the node is at a fixed position. We use this in our drag event callbacks to be able to move a node.

We're not done yet. We need to write all the callbacks we defined earlier. Let's start with what happens when a simulation *tick* occurs. Add the following, still

inside of the `Force()` function:

```
function ticked() {
  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);

  node.attr('cx', d => d.x)
    .attr('cy', d => d.y);
}
```

The force simulation just updates the values on the data; it doesn't actual cause them to be re-rendered. We have to do that ourselves, by setting the specific properties we need on the link and node elements.

Just after that, add the following three functions to define how the nodes should interact with the mouse events:

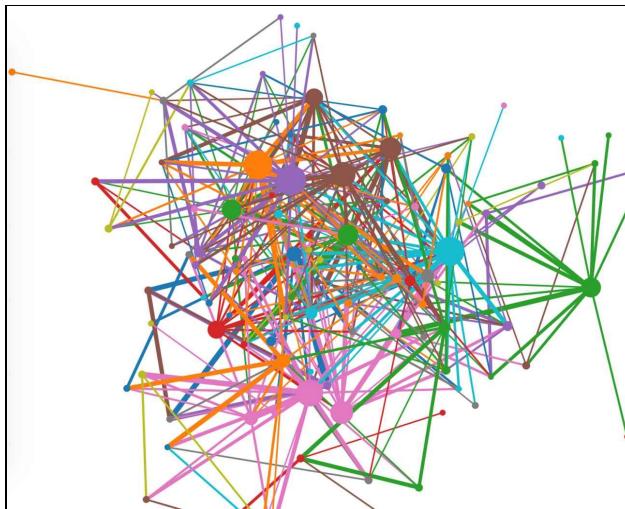
```
function dragstart(d) {
  if (!d3.event.active) sim.alphaTarget(0.3).restart();
  d.fx = d.x;
  d.fy = d.y;
}

function dragging(d) {
  d.fx = d3.event.x;
  d.fy = d3.event.y;
}

function dragend(d) {
  if (!d3.event.active) sim.alphaTarget(0);
  d.fx = null;
  d.fy = null;
}
```

We set the fixed position properties to the x-and y-coordinates of the mouse event. After we stop dragging, we set that to `null` so that it springs back to where it was.

Click on save, and you should have this awesomeness:



There's a lot more to `d3-force` than what we've gone through here. Check out the documentation for more on all the different styles of forces and values you can set.

Got mad stacks

The stack layout allows us to stack regions in a chart, which is useful for things such as stacked area charts and what have you. It's in the `d3-shape` package.

Go back to `main.js` and comment out the last example and add the following:

```
| westerosChart.init('stack',
|   'data/GoT-deaths-by-season.json', false);
```

We add another option to this because we're going to create two charts from this example. This time we're using our `deaths-by-season` dataset.

In `chapter7/index.js` add the following to the bottom of the file:

```
westerosChart.stack = function Stack({ data }, isStream = false) {
  const episodesPerSeason = 10;
  const totalSeasons = 6;
  // Create a nest containing deaths per episode
  const seasons = d3.nest()
    .key(d => d.death.episode)
    .key(d => d.death.season)
    .entries(data.filter(d => !d.death.isFlashback))
    .map(v => {
      return d3.range(1, totalSeasons + 1)
        .reduce((item, episodeNumber) => {
          const deaths = v.values.filter(d =>
            +d.key === episodeNumber)
          .shift() || 0;
          item[`season-${episodeNumber}`] = deaths ?
            deaths.values.length : 0;
          return item;
        }, { episode: v.key });
    })
    .sort((a, b) => +a.episode - +b.episode);
}
```

We create a function that has two arguments, one expecting an object with a `data` attribute, and another a Boolean deciding whether this is a stream chart or not. We then create a nest of items and filter out anyone who died in a flashback. We get an array of episodes, with each having a certain mortality rate per season, defined by the `season-n` key.

Next, set up our stack layout generator and map it to our `season-n` key:

```
| const stack = d3.stack()
|   .keys(d3.range(1, totalSeasons + 1)
```

```
|     .map(key => `season-${key}`));
```

Add this line to make it so that this instantly becomes a stream chart just by providing a truthy value to the function constructor:

```
| if (isStream) stack.offset(d3.stackOffsetWiggle);
```

Next, we set up our x-and y-scales:

```
const x = d3.scaleLinear()
  .domain([1, episodesPerSeason])
  .range([this.margin.left, this.innerWidth - 20]);

const y = d3.scaleLinear()
  .domain([
    d3.min(stack(seasons), d => d3.min(d, e => e[0])),
    d3.max(stack(seasons), d => d3.max(d, e => e[1]))
  ])
  .range([this.height -
    (this.margin.bottom + this.margin.top + 30), 0]);
```

Our stack layout creates an array of arrays, with each of those containing an upper and lower y-value. We can pass this to an area generator to create an area. Let's do that now:

```
const area = d3.area()
  .x(d => x(d.data.episode))
  .y0(d => y(d[0]))
  .y1(d => y(d[1]))
  .curve(d3.curveBasis);
```

Next, append the paths using our area generator:

```
const stream = this.container.append('g')
  .attr('class', 'streams')
  .selectAll('path')
  .data(stack(seasons))
  .enter()
  .append('path')
  .attr('d', area)
  .style('fill', (d, i) => color(i));
```

We provide our seasons data object to stack, which we pass to `.data()`. We then use our area generator to set the region shape.

Next, we append an x axis:

```
this.container.append('g')
  .attr('class', 'axis')
  .attr('transform',
    `translate(0,${this.height - (this.margin.bottom +
      this.margin.top + 30)})`)
```

```
|     .call(d3.axisBottom(x));
```

And then a legend:

```
const legendOrdinal = legend.legendColor()
  .orient('horizontal')
  .title('Season')
  .labels(d => d.i + 1)
  .scale(color);

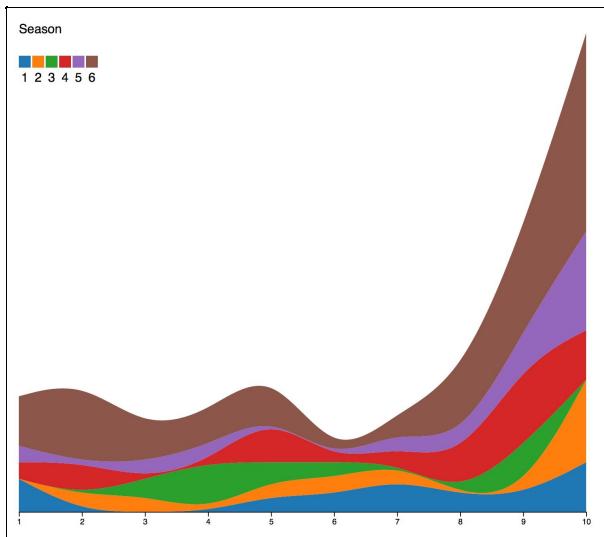
const legendTransform = isStream ?
  `translate(50,${this.height -
    (this.margin.bottom + this.margin.top + 130)})` :
  `translate(50,0)`;

this.container.append('g')
  .attr('class', 'legend')
  .attr('transform', legendTransform)
  .call(legendOrdinal);
```

Finally, it's tooltip time!:

```
| stream.call(tooltip(d => `Season ${d.index + 1}`), this.container));
```

There we go!



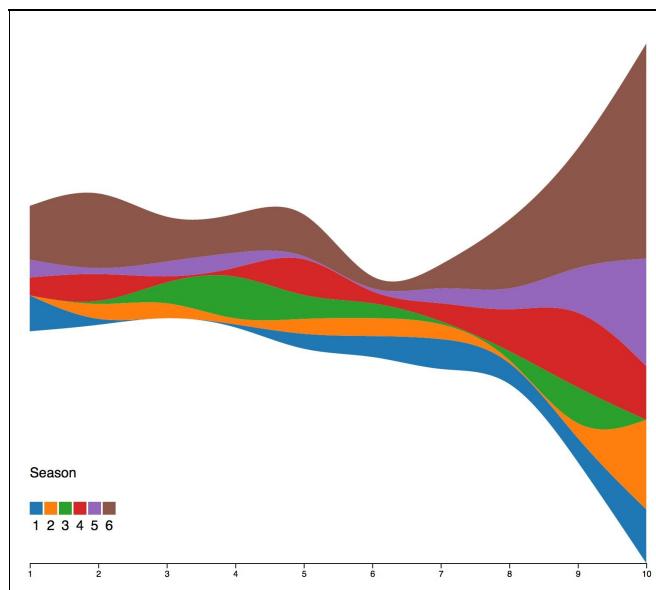
Bonus chart - Streamalicious!

Our bonus chart in this chapter is easy; we've already made it!

Go back to `main.js`, comment out the last line, and add this one:

```
westerosChart.init('stack',
'data/GoT-deaths-by-season.json', true);
```

All we've done is pass `true`. This gives us a stream chart, because we set the offset in the last example to `d3.stackOffsetWiggle`. Kind of cool looking, huh?



Summary

That's pretty much all the layouts, and we've done a pretty decent dip into each of them. I've provided the package names for each so that you can peruse the documentation. Each has a lot of options you can tweak, which is helpful for producing a very wide array of charts.

After going full out with these examples, we used almost every trick we've learned so far. We even wrote so much code that we pretty much have our own utility library! With a bit of generalization, some of those functions could be layouts of their own. There's a whole world of community that developed layouts for various types of charts. The `d3-plugins` repository on GitHub (<https://github.com/d3/d3-plugins>) is a good way 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 the bare bones, and use it to render things that aren't even SVG.

D3 on the Server with Canvas, Koa 2, and 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% of situations, so don't sweat it if you want to stick to the frontend and SVG for now. This chapter will still be here for that rainy afternoon when you want to try to figure out how **Heroku** works.

Readyng 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)** web hosting 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 will just test everything locally.



What is Heroku, and do you have to use it? Heroku is a way of deploying applications that use 12factor app principles (for the specifics, visit <http://12factor.net>). Without going too deep into the 12factor app philosophy, the idea is that you try to create stateless applications that use web services in the place of a large, monolithic piece of server infrastructure (for instance, a Linux-based virtual server running both the web server and database processes). I use Heroku, in this instance, because it's simple to deploy 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. Both Google Chrome and Node.js use the V8 JavaScript engine, 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 koa@next koa-bodyparser@next canvas-prebuilt --save
```

`Koa` is one of the leading Node.js web server libraries; it nicely abstracts Node's

ability to open ports and serve content into an easy-to-use API that is very light and fast. It's conceptually similar to another Node web server library, Express, but tries to be a bit more cutting-edge. We're using the `next` version of both `koa` and `koa-bodyparser` because the 2.x branch leverages `async/await` for flow control, which we're all about in this book. We also install a prebuilt version of Automatic's `node-canvas` library, which will allow us to render Canvas elements in Node.

Since Node uses the CommonJS module loading standard, we need to add a new set of build instructions to our `webpack.config.js` to transpile our ES2015 import statements into CommonJS `require()` statements. To do this, we need to make the object being exported in `webpack.config.js` into an array (so that it contains both configurations), so make it look as follows (the first configuration has been truncated for space reasons):

```
const path = require('path');

module.exports = [ // Change module.exports into an array!
  { ... }, // This is the first config; leave it be!
  {
    entry: './lib/chapter8/index.js',
    target: 'node',
    output: {
      path: path.resolve(__dirname, 'build'),
      filename: 'server.js'
    },
    externals: {
      'canvas-prebuilt': 'commonjs canvas-prebuilt'
    },
    devtool: 'inline-source-map',
    module: {
      loaders: [
        {
          test: /\.js$/,
          exclude: 'node_modules',
          loader: 'babel'
        },
        {
          test: /\.json$/,
          loader: 'json-loader'
        }
      ]
    }
  }
]; // Don't forget the closing bracket!
```

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 prebuilt `node-canvas` (which we'll use later in the chapter) as an external library and not try to bundle it. We build our bundle to `build/server.js`.

Note that this will still emit all your other code during a build; if it's going slow, feel free to temporarily disable the first `config` by commenting it out.

Should you use Babel for your server-side projects? It depends on your server environment, but probably not.

Node.js is pretty up to date in terms of what it supports with regards to modern JavaScript syntax, and you don't really need to transpile your server-side code anymore. Nevertheless, `async/await` support only dropped in Node version 7, (and only then by providing the `--harmony` flag to the interpreter in versions before 7.7.0), so in the spirit of not requiring everyone to upgrade to the latest version of Node, we're passing our code through Babel and Webpack. We also use ES6 module syntax throughout the book, which you'll need to transpile in all current versions of Node. Just be aware that you can ignore all the Webpack and Babel stuff if you use `require()` instead of `import` and Node version 7+.



All aboard the Koa train to servertown!

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 in `lib/chapter8` called `index.js`:

```
import * as Koa from 'koa';
import * as bodyParser from 'koa-bodyparser';
const app = new Koa();
const port = process.env.PORT || 5555;
app.use(bodyParser());
app.listen(port, () => console.log(`Listening on port ${port}`));
export default app;
```

Here, we import Koa and Koa Body Parser, instantiate Koa, assign a port number (defaulting to 5555), tell Koa to use the Body Parser middleware (which mainly just interprets `POST` request bodies into JavaScript objects for us) and then tell Koa to listen on that port.

What is `process.env.PORT`? The `process.env` is simply a global object of all environment variables from the shell that spawned the NodeJS script. We listen for requests on either port 5555 or *whatever the PORT environment variable is set to*. I've emphasized that last line because it's a key tenant in building web applications such as this -- keep all configuration in environment variables, so you never have to worry about storing plain text passwords in your source code (or in this case, the port number, which Heroku will set for us).

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 in 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 you instead 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 to say making your applications configurable through environment variables is a key aspect of writing good JavaScript server applications.

Add the following line:

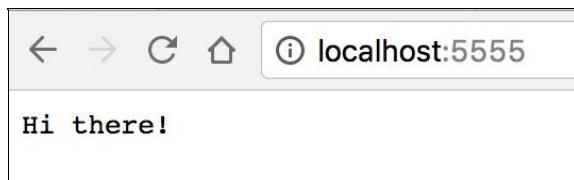
```
| app.use(ctx => ctx.body = 'Hi there!');
```

This creates a brand new piece of middleware and adds it to the stack. Our middleware here takes one argument, `ctx`, which is the Koa `context` object. Every time somebody makes a request to your application, a new context is created, and it navigates down through your middleware. Unlike with Express, a Koa context handles both the request (the incoming message from the user's browser to your application) and the response (what your application sends back to the browser). What we do here is set the response body to simply *Hi there!* and continue onward. Eventually, we reach the end of the middleware chain, and the response is sent back to the browser.

Now, go back to the command line and type the following:

```
| $ npm run server
```

Then, go to `http://localhost:5555`, 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 instructions with callbacks in a closure execute simultaneously, just like in the browser. In [Chapter 4, Making Data Useful](#), we discussed how JavaScript is said to be *asynchronous* -- when you write your server code, 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, this has really exciting ramifications for server code, making it very fast and easy to scale horizontally.

Let's do something cooler than displaying a static message on the screen, and let's break out a brand new D3 feature while we're at it.

Proximity detection and the Voronoi geom

A **Voronoi geom** (found in the `d3-voronoi` package) 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 any other point. It's effectively another layout, but it's used more for utility than for display -- for instance, a grid of Voronoi regions is often used to increase the mouseover area of data in a chart so that your cursor will always be highlighting the nearest point. We will use the Voronoi geom to figure out what the closest major airport is to your current location, which we'll supply via the HTML5 Location API.

Replace the last line we wrote with the following:

```
async function renderView(ctx, next) {
  if (ctx.method === 'GET') {
    ctx.body =
      `<!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(pos) {
        var latlng = pos.coords.latitude + ',' + pos.coords.longitude;
        document.querySelector('[name="location"]').value = latlng;
      });
    </script>
  </body>
</html>`;
  } else if (ctx.method === 'POST'){
    await next(); // This ensures all the other middleware runs first!
  }
  const airport = ctx.state.airport.data;
  ctx.body = `<!DOCTYPE html>
<html>
  <head>
    <title>Your nearest airport is: ${airport.name}</title>
  </head>
  <body style="text-align: center;">
    <h1>${airport.name}</h1>
    <p>${airport.address}</p>
    
</html>`;
}
```

```
<h1>
  The airport closest to your location is: ${airport.name}
</h1>
<table style="margin: 0 auto;">
  <tr>
    ${Object.keys(airport).map(v => `<th>${v}</th>`).join('')}
  </tr>
  <tr>
    ${Object.keys(airport)
      .map(v => `<td>${airport[v]}</td>`).join('')}
  </tr>
</table>
</body>
</html>;
}
app.use(renderView);
```

Holy moly, that's a lot of code. What's all going on here?

We first create a function and then assign it as a piece of middleware to our Koa app. All middleware functions receive a context (`ctx`) and `next()` argument; the former is used to pass instructions around the application, whereas the latter is used to control flow. In our `renderView()` function, first we create an HTML page template for when we make a `GET` request to our web server -- in other words, when we first load the site. We then make another page template for when we `POST` to our web server. This is where things get interesting. You'll notice in our `else` statement that before we do anything else, we call `next()` and `await` it to return. What this means is, "wait until the downstream middleware runs, then continue onward in this function." This means our function can return our `GET` request without invoking the other middleware, or in the case of a `POST` request, invoke our other middleware, have those add data to our context's `state` object, then continue with the rest of the function.



You'll notice that we put view code inside of our web server logic, which isn't great, but it works for our purposes. If you're creating multipage applications with multiple routes, you might want to use something like `Koa-Router` and `Koa-Views`. The former allows you to define URL routes for requests, and the latter allows you to use template languages, such as Handlebars, to define HTML output.

If you haven't recently, restart the server by press `Ctrl+C` and then run the following command:

```
| $ 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 the user to fill in latitude and longitude as a comma-separated value. Alternatively, if the user accepts the browser's request to use the HTML5 location API, it will auto-populate.

Next, we need to create a new function for the Voronoi calculations. Add the following at the top of `import` section of `chapter8/index.js`:

```
import { readFileSync } from 'fs';
import * as d3 from 'd3';
import * as boundaries from '../../../../../data/cultural.json';
import * as land from '../../../../../data/land.json';
```

This imports D3 and Node's built-in synchronous filesystem loading function. We also import our TopoJSON files from [Chapter 4, Making Data Useful](#), and because we have access to the filesystem in Node, we can happily just import them like any old JavaScript file instead of doing an XHR request.

After that, add this:

```
const airportData = readFileSync('data/airports.dat',
  { encoding: 'utf8' });
const points = d3.csvParseRows(airportData)
  .filter(airport => !airport[5].match(/N/) && airport[4] !== '')
  .map(airport => ({
    name: airport[1],
    location: airport[2],
    country: airport[3],
    code: airport[4],
    latitude: airport[6],
    longitude: airport[7],
    timezone: airport[11],
  }));
});
```

This will parse all of our airport data into an object that we can use throughout our application. Next, we create a function that creates a `voronoi` layout used to calculate which of the airports is closest to our location:

```
export function nearestLocation(location, points) {
  const coords = location.split(/,/s?/);
  const voronoi = d3.voronoi()
    .x(d => d.latitude)
    .y(d => d.longitude);

  return voronoi(points).find(coords[0], coords[1]);
}
```

We assume our location is in a comma-separated string, which we convert into an array. We then instantiate our `d3.voronoi` layout generator, telling it to get each datum's latitude and longitude properties to calculate the x and y-values, respectively. We then pass our object containing all airports as points, then use the new `.find()` method to calculate which point is the closest, which we then return.



The Voronoi geom has changed a fair bit in D3 v4, most notably with the addition of `layout.find()`. Earlier, one had to do a lot of fiddly math stuff, and this chapter was quite a bit more complex. That is no more! Thanks, Mike!

This is useful, but in order to tie it into our Koa app, we need to write a new piece of middleware. Add this function to `lib/chapter8/index.js`:

```
function nearestAirport(ctx, next) {
  if (ctx.request.body.location) {
    const { location } = ctx.request.body;
    ctx.state.airport = nearestLocation(location, points);
    next();
  }
}
```

This is pretty straightforward. We get the location value from the request body, use our `nearestLocation()` function to determine the nearest airport, then assign that to the context `state` object. We then call `next()` to invoke the rest of the downstream middleware and pass our updated `state` object to it. Our `renderView` middleware will await `nearestAirport` to call `next()` and then use the updated `state` object to render a table of information about the nearest airport.

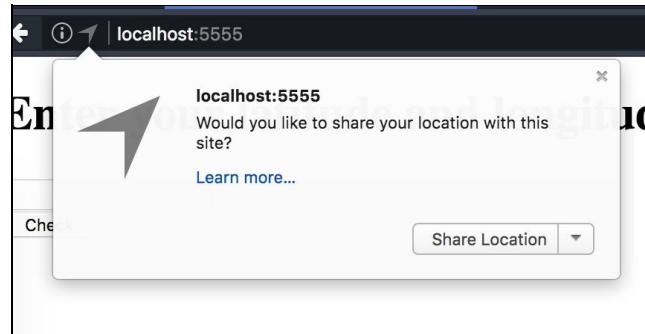
Find the lines where we instantiate our middleware using `app.use()`. Add our middleware to the stack, after `bodyParser` and `renderView`, so it looks like the following:

```
app.use(bodyParser())
  .use(renderView)
  .use(nearestAirport);
```

Click on save, press `Ctrl+C` in your terminal to kill the server and then type this command:

```
| npm run server
```

This will restart it. Go to `localhost:5555`, and it should ask you for permission to geolocate:



This is actually from Firefox, but Chrome should look similar:

Enter your latitude and longitude, or allow your browser to check.

51.5048775,-0.1145936

Click on the check button, and you should have a screen like the following:

name	location	country	code	latitude	longitude	timezone
City	London	United Kingdom	LCY	51.505278	0.055278	Europe/London

Hey, that's actually kind of cool. Also, 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 in 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 tasks 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.

We haven't really talked about it much, but Canvas is another cool way D3 can be used to render data. Instead of writing a bunch of elements into a DOM tree format like SVG, Canvas renders pixels in a 2D plane, which can be much better for performance. It's a much different workflow than what we're used to, and I don't focus on it too much in this book because it tends to be harder to debug for beginners, but let's give it a shot and deep dive into it for one example.

By the way, it's worth noting 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.

Create a new function in `chapter8/index.js`:

```
function canvasMap(ctx, next) {
  const { airport } = ctx.state;
  const scale = ctx.query.scale || 1200;
  const projectionName = d3.hasOwnProperty(ctx.query.projection) ?
    ctx.query.projection : 'geoStereographic';
  const canvas = new Canvas(960, 500);
  const canvasCtx = canvas.getContext('2d');
  const projection = d3[projectionName]()
    .center([airport.data.longitude, airport.data.latitude])
    .scale(scale);
}
```

We start by getting the airport object from context's `state` object. We want to be able to use query arguments to set the scale and projection of the output, so we assign them from context's `query` object, offering a default value if that argument doesn't exist. We then instantiate a new 960px by 500px Canvas object, get its drawing context, and create a new projection (defaulting to `d3.geoStereographic`) centered on the user's location. In order to differentiate the Canvas context (which is where and how you do your drawing) from the Koa context (which is

all about how the request is handled), we're going to call the former `canvasCtx`.

Next, add the following to `canvasMap`:

```
const path = d3.geoPath()
  .projection(projection)
  .context(canvasCtx);

canvasCtx.beginPath();
path(topojson.mesh(land));
path(topojson.mesh(boundaries));
canvasCtx.stroke();
```

This creates a path generator using our projection, and tells it to output to the Canvas context. We then create a path, supplying our land and boundaries as Topojson meshes, which we stroke with the default values. Unlike what we normally do in D3, we don't chain these methods -- we run them one right after the other, or *procedurally*.

If we were doing this in the browser, we'd now have a map of the world centered on the user. Let's add an indicator showing where the user's nearest airport is:

```
const airportProjected = projection([airport.data.longitude,
  airport.data.latitude]);
canvasCtx.fillStyle = '#f00';
canvasCtx.fillRect(airportProjected[0] - 5,
  airportProjected[1] - 5, 10, 10);
```

This gives us the projected x and y values for the airport, which we provide to the Canvas context's `fillRect` method, which simply draws a filled rectangle (the arguments supplied are *x position*, *y position*, *width*, and *height*).

We're not done yet, though -- even though our server has rendered the Canvas drawing, we can't yet see it because it currently only exists in an ephemeral form, inside the web server's memory. In order to display it, we need to convert it to a base64-encoded string, which can be given to an `img` tag to render the Canvas element. Add the following to the end of `drawCanvas()`:

```
ctx.state.canvasOutput = canvas.toDataURL();
next();
```

We convert the Canvas element to a base64 and attach that to the `state` object. We then call `next()` to invoke the other downstream middleware.

Let's keep our original page rendering middleware intact and just write a new

one. Copy and paste `renderView` below the original `renderView`, rename it `renderViewCanvas`, and add an image tag in the second template. I've highlighted the parts you need to worry about in the following code:

```
async function renderViewCanvas(ctx, next) {
  if (ctx.method === 'GET') {
    ctx.body =
      &grave;<!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.querySelector('[name="location"]').value =
              position.coords.latitude + ',' + position.coords.longitude;
          });
        </script>
      </body>
    </html>&grave;;
  } else if (ctx.method === 'POST') {
    await next(); // This ensures the other middleware runs first!
    const airport = ctx.state.airport.data;
    const { canvasOutput } = ctx.state;
    ctx.body = &grave;<!doctype html>
    <html>
    <head>
      <title>Your nearest airport is: ${airport.name}</title>
    </head>
    <body style="text-align: center;">
      <h1>
        The airport closest to your location is: ${airport.name}
      </h1>
      
      <table style="margin: 0 auto;">
        <tr>
          ${Object.keys(airport).map(v => &grave;<th>${v}</th>&grave;).join('')}
        </tr>
        <tr>
          ${Object.keys(airport).map(v =>
            &grave;<td>${airport[v]}</td>&grave;).join('')}
        </tr>
      </table>
    </body>
  </html>&grave;;
}
}
```

In addition to our airport, we also now pull the `canvasOutput` property from `state`, and output it as an image. The key bit is here:

```
| 
```

Note that Canvas produces *raster* images -- unlike SVG, which can be scaled as big or tiny as you want without quality degradation -- the size of a canvas element is dictated by the resolution it's rendered at, and you will start to see pixels if you scale it past the usual image thresholds. We use a popular trick here to ensure that our image renders nicely even on Retina displays -- when we instantiated the Canvas object back in `drawCanvas`, we made it double the size we wanted it to eventually render as. Here, in `renderViewCanvas`, we explicitly set the width and height to half of the original values, which results in a sharper image.

Go down to your `app.use` section and replace it with the following:

```
| app.use(bodyParser())
|   .use(renderViewCanvas)
|   .use(nearestAirport)
|   .use(canvasMap);
```

Press `Ctrl+C` if you still have Node running and then restart it by typing:

```
| $ npm run server
```

Open up `localhost:5555`, enter a latitude/longitude pair (or let the browser geolocate it), and now your results page has a handy little static map!:



If we add query arguments, our URL looks as follows:

<http://localhost:5555/?scale=400&projection=geoMercator>

It will change the scale to 400 and the projection to Mercator. 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 suffers less from 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 550 (plus an additional 450 if you verify your account with a credit card) *dyno hours* distributed between all of your servers, with machines down cycling when they aren'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 (otherwise it defaults to `node start`, but we use that to launch our Webpack development server in this book instead). Add the following contents:

```
| web: node build/server.js
```

Save it and then make sure that you have the latest bundle built:

```
| $ npm run server
```

Press `ctrl+c` after it builds; we don't need to run it any more.

Lastly, add everything to a commit:

```
| $ git add . && git commit -am "Time for Heroku"
```

We're now going to deploy. Assuming that 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 chapter8 branch in Git, type the following instead:

```
$ git push heroku chapter8:master
```

Visit the URL provided by `heroku create --` 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 Koa 2 that used D3's Voronoi geom to find the nearest airport to a particular latitude/longitude pair. We then upgraded our server app to draw a map using D3 and Canvas, which we then outputted to the user as a PNG via a base64-encoded string.

Wasn't that all really pretty weird, but also kind of fun? Writing server-side code is like that. However, it can also be really rather cathartic after spending a bunch of time doing frontend development, which tends to be really finicky due to having to support so many devices. Furthermore, although purely client-side *single-page applications* were popular for a while, webapps are increasingly pre-rendered via a server, then *rehydrated* once they arrive for the user. It's never been more important for frontend developers to have at least a passable understanding of what's going on behind the scenes.

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 *at all* into scalability (which is super important when building things for large audiences, such as in the newsroom), nor into how to properly architect a nontrivial 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 Viăduțu's Mastering Web Application Development with Express* (Packt, 2014). Koa is very similar to Express, and embraces a lot of its idioms (in some ways, it's even Express' spiritual successor), so you'll get a lot of mileage out of Express tutorials (alas! There aren't yet that many for Koa 2).

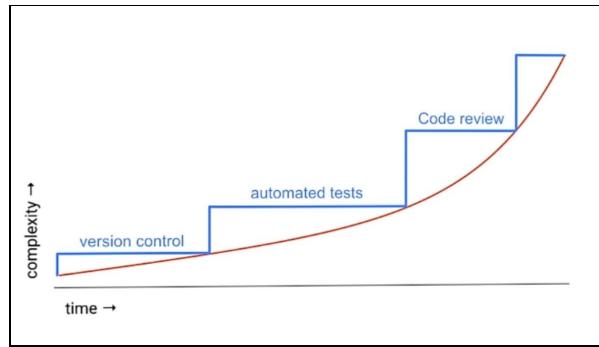
You now have a pretty full toolbox for confronting a very wide array of data visualization challenges. In the next chapter, we'll add a few more tools to it -- linting, unit testing, and static typing -- in order to help you have more confidence in the work you produce.

Having Confidence in Your Visualizations

When you're building things for the huge audience the Web allows, a very real fear that you have is a software glitch that prevents data from displaying correctly. When developing projects on a tight timeline, testing is an aspect that often gets neglected as the deadline gets closer and closer, and often things need to be viewed by other people (editors, managers, and the like) even earlier, emphasizing output over process in turn. Let me be clear--if you want to ensure that your visualizations are high-quality, you need to take steps to ensure that they are well-tested and functioning properly. On some level, doing this is an exercise in managing complexity.

The following chart depicts project complexity over time. As you can see, complexity increases somewhat exponentially as time goes on. 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, often it's better to do so incrementally in response to the project's demands. For instance, generally everyone will start with version control because it helps collaboration, can revert mistakes, and provides a project with history (plus, it's really easy to get set up). Then, say, you add a few more team members or open-source a project. Now, you have code flying at you from all angles; having some way to automatically test whether a change will break anything starts to become incredibly useful. A bit later, imagine that a lot of bad code is still getting past the automated testing; having manual code reviews might further help.

Each of these improvements to process takes time to both implement and use, and whether they'll benefit your project is hugely dependent upon how big your team is and how well you trust every member of it (or, if you are working alone, how well you trust yourself not to introduce errors):



As time goes on, complexity increases; tooling increases, however, 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 on a lot of the same topics as this chapter and it's worth watching; you can find it at <https://www.youtube.com/watch?v=yy4c0hzNXKw>.

In this chapter, we'll focus on a few technologies to 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.
- We then move to **static type checking**, which is a tool to ensure that you're correctly passing the right kinds of variable between functions at build time.
- **Automated testing** is what it says on the tin; you write tests for your code, all of which it must pass. Failing tests can be used to diagnose bugs or other issues with the code.

The aforementioned tools all try to do two things -- help reduce bug density and improve the development experience. Primarily, they accomplish this by providing feedback to developers while coding, ensuring that things such as misspelled variable names or incorrectly passed arguments are caught in development instead of in production.

Linting everything!

A **linter** is a piece of software that runs source code through a set of rules and then causes a stink if your code breaks any of those rules. On the one hand, this is intended to make code look consistent across a project, but on another, it flags up potential code issues while developing--particularly obvious mistakes, such as misnamed variables.

Linting rules are often based on industry best practices, and most open source projects have a customized ruleset corresponding to their community guidelines. This simultaneously ensures that code looks consistent even when delivered by a multitude of people, and also lets contributors know when they're doing something that's 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 repo for this book, perhaps you've noticed a hidden file called `.eslintrc`, or noticed `eslint` in `package.json`. ESLint is the current best linter for ES2017 code, and it works very similarly to its predecessors: JSHint and JSCS. It's usually configured via the `.eslintrc` file, which both contains the specific rules to be checked against (or a set of defaults to extend--I'm a huge fan of Airbnb's configuration and use a slightly modified version of their ruleset throughout this book) 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 as follows:

```
| $ npm run lint
```

Although there won't be any linting errors in the repo by the time this book gets to print, here's an example of how the output looked at this stage of the book:

```
/Users/aendrew/Sites/Learning-d3-v4/lib/common/index.js
 31:25 error Assignment to property of function parameter 'd'  no-param-reassign
 41:10 error Unexpected string concatenation  prefer-template
 41:29 error Strings must use singlequote  quotes
 42:9 error Strings must use singlequote  quotes
 42:29 error Multiple spaces found before '", "'  no-multi-spaces
 42:29 error Strings must use singlequote  quotes
 42:33 error Unexpected mix of '+' and '/'  no-mixed-operators
 42:61 error Unexpected mix of '+' and '/'  no-mixed-operators
 43:9 error Strings must use singlequote  quotes
 43:28 error Strings must use singlequote  quotes
 43:32 error Unexpected mix of '+' and '/'  no-mixed-operators
 43:60 error Unexpected mix of '+' and '/'  no-mixed-operators
 44:9 error Strings must use singlequote  quotes
 44:28 error Strings must use singlequote  quotes
 48:10 error Unexpected string concatenation  prefer-template
 48:10 error Strings must use singlequote  quotes
 48:29 error Strings must use singlequote  quotes
 49:9 error Strings must use singlequote  quotes
 49:13 error Unexpected mix of '+' and '/'  no-mixed-operators
 49:41 error Unexpected mix of '+' and '/'  no-mixed-operators
 49:47 error Strings must use singlequote  quotes
 50:9 error Strings must use singlequote  quotes
 50:13 error Unexpected mix of '+' and '/'  no-mixed-operators
 50:41 error Unexpected mix of '+' and '/'  no-mixed-operators
 50:47 error Strings must use singlequote  quotes
 51:9 error Strings must use singlequote  quotes
 51:28 error Strings must use singlequote  quotes
 94:1 warning Line 94 exceeds the maximum line length of 100  max-len
223:1 warning Line 223 exceeds the maximum line length of 100  max-len
224:1 warning Line 224 exceeds the maximum line length of 100  max-len
303:1 warning Line 303 exceeds the maximum line length of 100  max-len
```

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:

```
| $ node .node_modules.bin/eslint 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 use it all the time during 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 in the module section of each build item:

```
module: {
  rules: [
    {
      test: /\.js$/,
      loader: 'eslint-loader',
      exclude: 'node_modules',
    },
  ],
}
```

Run `webpack-dev-server`:

```
| $ npm start
```

Ta-da! Now you can get instant feedback about how messy your code is every time you hit Save! You're utterly thrilled by this, I can just feel it!

Linting is a very light way to manage 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. However, it's this lightness that makes it one of the easiest pieces of tooling to implement -- even if the rest of your team thinks linting is a godforsaken nuisance, you can still add an `.eslintrc` file to your home folder and get the benefits of having a linter yourself.



*What's even more useful than getting linting errors in your console? Getting linting errors directly in your IDE! Pretty much every text editor now has a plugin for ESLint, with the ones for Atom and Sublime Text being quite excellent. Do yourself a **huge** favor and install one of these; they'll give you immediate feedback when you've mistyped a variable name or are using a variable outside of its scope. This alone will save you countless hours of checking your browser logs. You don't even need to have ESLint installed in your project for this to work!*

Static type checking: TypeScript versus Tern.js

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 it looks at the *type* of each variable and uses either *type annotations* (bits of text defining what type a variable is when the variable itself is defined) or *type inference* (figuring out what the type is from how you first used it) to ensure that functions don't mutate a variable in an unexpected way. This is called *static typing* and 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. As we're using a transpiler to transform our JavaScript anyway (throughout the book this has been Babel, though many other transpilers exist), we can introduce static type checking to JavaScript at the same time if we want to.

This is only a part of why static type analysis is useful - the other half is how it integrates into your IDE, which allows things such as easy access to documentation and extremely accurate autocompletion. It also gives immediate feedback on developer errors, which means errors make it to production far less frequently, and are thus much faster to fix. It also helps to internally document your code, which may make it easier for others to pick up.

TypeScript works really well with D3 because it has a very high-quality type definition, allowing you quick access to documentation within your code editor. This, combined with the ability of the TypeScript compiler to provide immediate feedback on malfunctioning code, means writing high-quality D3 code is much easier.

First, a caveat: is it even worth bothering with TypeScript?

I've done a few projects in TypeScript recently and while I've really

tried to like it and think that it does add a certain extra level of quality to my code, I can't argue there aren't trade-offs to using it. If you do TypeScript with "implicit any" turned off (which we'll discuss later), a lot of time can be spent chasing (or writing) the right interface to satisfy a particular function.

While this extra metadata makes code a bit easier to read and ultimately more predictable in production, there's some evidence indicating that it doesn't have anywhere near as big an effect in preventing bugs as automated testing and TDD (Test-Driven Development). It can also really slow down using some fluent styles of programming if you're already fairly proficient in a library, meaning that you may find it more of a drag than anything. For a more thorough discussion, read Eric Elliott's "You Might Not Need TypeScript (Or Static Types)" at <https://medium.com/javascript-scene/you-might-not-need-typescript-or-static-types-aa7cb670a77b>.



I personally really like writing TypeScript and using the TypeScript plugin for Atom, even if it does occasionally slow me down somewhat (in all fairness, the newer versions of TypeScript are far less verbose and much better at inferring what a particular type should be without having to explicitly annotate it). It's definitely worth trying out, but don't do so at the expense of tests, which I will address in the next section. Again, as mentioned in the introduction, how useful any of this stuff is depends entirely upon your team and the ultimate complexity of the project - if you're creating one-off static HTML and JavaScript pages for a news media publication, you can probably get away without ever having to learn TypeScript. If you're creating massive, multi-view applications with a lot of user interactivity and dozens of modules, TypeScript will be significantly more useful.

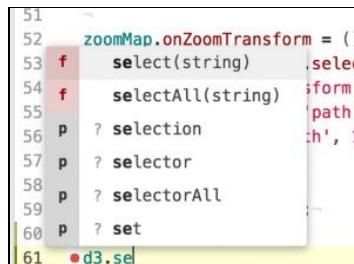
What if you want the internal documentation improvements without having to switch a boatload of tooling and what not? One solution is `tern.js`, which provides TypeScript-like intellisense without the static typing. It's definitely only useful if you have IDE support for it, so check whether your editor has a plugin (most decent ones do). We'll cover that first because it's easiest to get up and running and you don't have to do much else.

Code analysis with Tern.js

As mentioned earlier, build-time type checking is helpful for finding errors in your code while you write it as it flags them while you develop. Although this is helpful, it carries with it a fair bit of extra effort and configuration. Meanwhile, the other half of why things such as type definitions are helpful is that they improve the developer experience by providing API documentation and code analysis.

You can get quite a lot of this functionality without using static typing through a piece of tooling called `Tern.js`. Tern is a code analysis engine that scans your modules and provides feedback to other tooling. You generally don't use it on its own; you install a plugin for your code editor of choice and it talks to Tern.

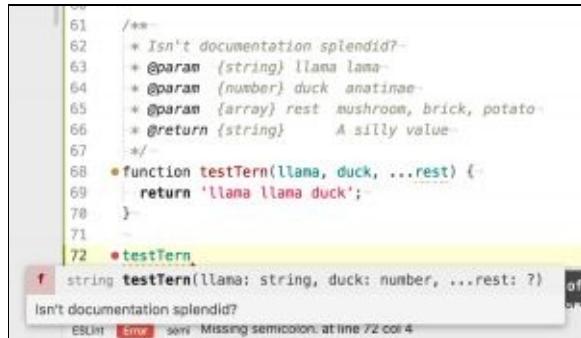
Atom, Sublime, and other IDEs have plugins -- how you install them is dependent upon your IDE. I personally use Atom with `atom-ternjs`:



When Tern's all configured in Atom, it provides code completion, such as the preceding. Note how, just by typing `d3.se`, I've brought up both `d3.select` and `d3.selectAll`, plus their function signatures? This is helpful for ensuring that you pass the correct types to various functions. Tern can do more, though--if you have a documentation comment as shown:

```
/**  
 * Isn't documentation splendid?  
 * @param {string} llama lama  
 * @param {number} duck anatinae  
 * @param {array} rest mushroom, brick, potato  
 * @return {string} A silly value  
 */  
function testTern(llama, duck, ...rest) {  
  return 'llama llama duck';  
}
```

It will autocomplete with far more information, like this:



A screenshot of a code editor showing a file named `index.js`. The code contains a function `testTern` with JSDoc-style documentation. The cursor is at the end of the line `string testTern(string, number, ...rest: ?)`. A tooltip appears over the cursor with the text "Isn't documentation splendid?". Below the code editor, there is a status bar with the text "ESLint" and "Error: semi Missing semicolon at line 72 col 4".

To install Tern globally, run the following:

```
| $ npm install -g tern
```

You generally need to do this before Tern starts working in your IDE. You also need to create a `tern` config file. Add the following to a file called `.tern-project` in the root of your project directory:

```
{  
  "ecmaversion": 7,  
  "libs": [  
    "browser"  
,  
  "plugins": {  
    "doc_comment": {  
      "fullDocs": true,  
      "strong": false  
    },  
    "node": {  
      "dontLoad": "",  
      "load": "",  
      "modules": ""  
    },  
    "modules": {  
      "dontLoad": "",  
      "load": "",  
      "modules": ""  
    },  
    "es_modules": {}  
  }  
}
```

Install a Tern plugin to your IDE of choice and you should start seeing rich code completion provided by Tern.

Unfortunately, D3.js doesn't have anything in terms of internal documentation and so Tern.js is far more useful when you document your own code well. In the next section, we'll use TypeScript, which has an excellent type definition for D3,

in essence pulling a lot of documentation into your editor.

TypeScript - D3 powertools

There are many upsides to using TypeScript, which I'll get into in just a moment. Before that, be forewarned that there's bit more effort involved with getting TypeScript set up because it's a transpiler in its own right, which means we won't really use Babel with it 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, you probably won't notice the difference between the two. Further, we will actually use both simultaneously, importing TypeScript modules into Babel and vice versa.

First, let's install some more stuff. Modern JavaScript development is such as 50% coding, 20% being confused by what libraries to use, and 30% installing those libraries. So, let's get to it:

```
| $ npm install typescript ts-loader --save-dev
```

This will install both the TypeScript transpiler and the Webpack loader. Next, run the following:

```
| $ npm install @types/d3 d3-sankey aendrew/d3-sankey --save
```

This installs the D3 type definitions provided by the community. Previously, you used to need a separate utility to install definitions, but with TypeScript 2.0, they're all now available via the `@types` npm organization. We save them as a normal dependency because your definitions are part of your main code with TypeScript. It also installs `d3-sankey`, which isn't part of the main D3 monolib package, and the `aendrew/d3-sankey` TypeScript definition.

Tom Wanzek is the superhero who spearheaded creating the excellent TypeScript definitions for D3 v4. I cannot express how grateful I am that he did, because that would have been a crazy amount of work.



The one exception to this is `aendrew/d3-sankey`, which is a TypeScript



definition for `d3-sankey` I wrote in preparation for this chapter. Writing TypeScript definitions is a bit beyond the scope of this chapter, but it doesn't take an enormous degree of familiarity with TypeScript to start contributing to community projects, such as DefinitelyTyped, which is where the `@types npm` organization gets all of its definitions from.

Next, let's update our Webpack config. Under rules, add the following:

```
{  
  test: /\.ts$/  
  loader: 'ts-loader'  
},
```

This will use TypeScript to load all the files ending in `.ts`. You can go back to all your old files and rename them to `.ts`, or you can just do like we did 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 will mash up both ES2015+ and TypeScript for great awesomeness.

Not done yet! Create a file called `.tsconfig` in the root of your project, and fill it with the following:

```
{  
  "compilerOptions": {  
    "target": "ESNext",  
    "module": "ES6",  
    "moduleResolution": "node",  
    "isolatedModules": false,  
    "jsx": "react",  
    "experimentalDecorators": true,  
    "emitDecoratorMetadata": true,  
    "declaration": false,  
    "noImplicitAny": false,  
    "noImplicitUseStrict": false,  
    "removeComments": true,  
    "noLib": false,  
    "preserveConstEnums": true,  
    "suppressImplicitAnyIndexErrors": true,  
    "allowJs": true,  
    "lib": [  
      "es2015",  
      "es2015.iterable",  
      "es2015.symbol",  
      "es2015.promise",  
      "dom"  
    ]  
  },  
  "exclude": [  
    "node_modules"  
  ],  
  "compileOnSave": false,
```

```
|   "buildOnSave": false,  
|   "atom": {  
|     "rewriteTsconfig": false  
|   }  
| }
```

To start, we'll comment out everything in `lib/main.js` and add the following:

```
| import sankeyChart from './chapter9/index.ts';  
| sankeyChart();
```

Next, create a folder called `lib/chapter9` and a new file in that, called `index.ts`:

```
| import {  
|   sankey,  
|   SankeyLink,  
|   SankeyNode,  
|   SankeyData } from 'd3-sankey';  
| import * as d3 from 'd3';  
| import chartFactory from '../common/index';
```

This includes both your TypeScript definitions and imports D3 from `node_modules`. From here, we can start using TypeScript like we would normally.

If you didn't get the hint from all the stuff we just installed, we will create a Sankey diagram. Much like the force directed diagram we made earlier, a Sankey diagram depicts relations in a graph format; however, its main use is to depict the magnitude of those links. The Sankey diagram we make will depict where seats went in the last UK general election. I suppose the 2016 US presidential election would be slightly more topical, but Sankey diagrams are more fun when we have more nodes than just two to work with.

Let's start using our old friend, `chartFactory`, to scaffold out our chart:

```
| const chart = chartFactory({  
|   margin: { left: 40, right: 40, top: 40, bottom: 40 },  
|   padding: { left: 10, right: 10, top: 10, bottom: 10 },  
| });
```

Next, we create an object containing the various colors representing each major party, grouping a bunch of the smaller parties into the "Other" category.

```
| const partyColors = {  
|   CON: '#0087DC',  
|   LAB: '#DC241F',  
|   SNP: '#FFFF00',  
|   LIB: '#FDBB30',  
|   UKIP: '#70147A',  
|   Green: '#6AB023',  
|   Other: '#CCCCCC',
```

```
|};
```

I've done a bunch of data parsing and reformatting to make this example nice and straightforward; it's in the book repo `data/` directory as `uk-election-sankey.json`. Let's create a new `async` function and load in our data using `fetch`:

```
async function typescriptSankey() {
  const sankeyData: SankeyData = await (
    await fetch('data/uk-election-sankey.json')).json();
  const width = chart.width;
  const height = chart.height;
  const svg = chart.container;
}
```

If you want to see how the data was generated, the script I used is `getSankeyData.ts` in `data/scripts/` in the book repository. It's in TypeScript too!

We will not annotate many of our variables in this chapter because our type definition does a lot of the hard work for us. We do, however, need to annotate our output from `fetch` because otherwise TypeScript will just see it as an untyped object. To do this, we use the `SankeyData` interface we imported earlier, which is part of the `d3-sankey` TypeScript definition. An *interface* is essentially just a grouping of type declarations, sort of like an object of types. In its entirety, our `SankeyData` interface looks as follows:

```
export interface SankeyData {
  nodes: Array<SankeyNode>;
  links: Array<SankeyLink>;
}
```

Despite the weird syntax, it's pretty straightforward--we're essentially creating an object with two properties: `nodes` and `links`--that consist of arrays containing `SankeyNodes` and `SankeyLinks`, respectively. Although it's not hard to take a look at what these two additional interfaces look like (if you've installed the excellent **atom-typescript** plugin for working in this section, right-click on the import and select *Go to declaration* from the context menu), suffice to say they're pretty straightforward and similar to data structures you've already used many times throughout this book (specifically, `SankeyNode` has a `name` and a `value` property, and the `SankeyLink` interface has `source`, `target`, and `value` properties). These, in turn, extend another object that contains all the properties `d3-sankey` decorates our data with, but again, a lot of this is behind the scenes and you don't really interact with it, except when you get something wrong and the TypeScript compiler

complains that you're missing a property or whatever. In many ways, this replaces having a linter--TypeScript presently isn't lintable by ESLint and instead has its own linter--tslint--but that's mainly to ensure code formatting and less to flag up problems with your code (the TypeScript compiler itself lets you know when something is amiss).

We haven't done anything with SVG gradients yet, so let's play with those in our last real example. To create a gradient with SVG, we create a gradient definition object and store it in a hidden tag called *defs*. We then reference that by ID later on. Bear with me here, it's a bit confusing but it'll all make sense in a second.

Still inside `typescriptSankey()`, add the following:

```
const defs = svg.append('defs');
Object.keys(partyColors).forEach((d, i, a) => {
  a.forEach(v => {
    defs.append('linearGradient')
      .attr('id', `&grave;${d}-$v`)&grave;;
    .call((gradient) => {
      gradient.append('stop')
        .attr('offset', '0%')
        .attr('style',
          `&grave;stop-color:${partyColors[d]}; stop-opacity:0.8&grave;`);
      gradient.append('stop')
        .attr('offset', '100%')
        .attr('style',
          `&grave;stop-color:${partyColors[v]}; stop-opacity:0.8&grave;`);
    });
  });
});
```

This will create a structure resembling the following:

```
<defs>
  <linearGradient id="CON-LAB" >
    <stop
      offset="0%"
      style="stop-color: #0087DC; stop-opacity: 0.8"
    />
    <stop
      offset="100%"
      style="stop-color: #0087DC; stop-opacity: 0.8"
    />
  </linearGradient>
</defs>
```

A linear gradient can have an arbitrary number of color stops set along a continuum. We transition from the originating party color to the target party color and give it an ID reflective of this relationship.

Oh snap waddup--it's finally time to rock out with our layout generator! Again, add the following inside `typescriptSankey`:

```
const sankeyGenerator = sankey()
  .size([width - 100, height - 100])
  .nodeWidth(15)
  .nodePadding(10)
  .nodes(sankeyData.nodes)
  .links(sankeyData.links)
  .layout(1);

const path = sankeyGenerator.link();
```

This creates a Sankey generator set to the size of our browser window, sets the width of each node to 15, pads each node by 10, and assigns all of our links and nodes to the layout generator. We then create a path generator using `Sankey.link()`, which will draw all of our beautiful swoopy paths between nodes.

With all of that set up, finally we're at the *let's actually render stuff on the page* part of our example. Draw our links, like so:

```
const link = svg.selectAll('.link')
  .data(sankeyData.links)
  .enter()
  .append('g')
  .classed('link', true);

link.append('path')
  .attr('d', path)
  .attr('fill', 'none')
  .attr('stroke', d => {
    const source = d.source.name.replace(/(2010|2015)/, '');
    const target = d.target.name.replace(/(2010|2015)/, '');
    return `url(#${source}-${target})`;
  })
  .style('stroke-width', d => Math.max(1, d.dy));
```

The only things that are somewhat confusing here are the `stroke-width` and `stroke` attributes--for our stroke color, we generate a string reflective of our source and target party names, and use that to reference the gradients we made earlier. To go back to the earlier example, our gradient transitioning from blue to red (indicating support changing from the Conservative Party to Labour) will be `#CON-LAB`. To make it reference the earlier `linearGradient` definition, we wrap this ID in `url()`.

Lastly, we set the `stroke-width` to the relative y value of each link.

Now that we have all of our links set up, let's render our nodes and labels!

First, we set up a group to contain our rectangles and labels:

```
const node = svg.selectAll('.node')
  .data(sankeyData.nodes)
  .enter()
  .append('g')
  .classed('node', true)
  .attr('transform', d => `translate(${d.x},${d.y})`);
```

It's pretty straightforward. Let's add our rectangles:

```
node.append('rect')
  .attr('height', d => d.dy)
  .attr('width', sankeyGenerator.nodeWidth())
  .style('fill', d =>
    partyColors[d.name.replace(/(2010|2015)/, '')])
  .append('title')
  .text(d => `${d.name} nSeats: ${d.value}`);
```

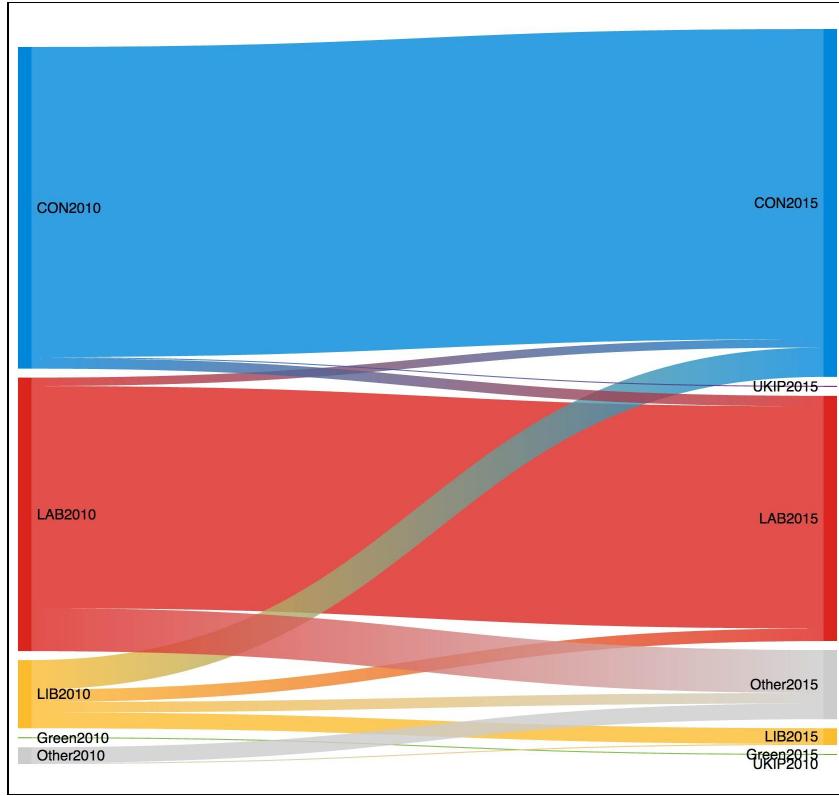
We get the width of each rectangle using the `sankey.nodeWidth()` method, which is the same method we used earlier but without an argument, so it returns our earlier width. We set the fill using our `partyColor` object from earlier, but we remove the year from each name using `String.prototype.replace()`, so it matches our object.

Lastly, we'll quickly add some labels so that we don't have to identify our parties based on color alone:

```
node.append('text')
  .attr('x', -6)
  .attr('y', d => d.dy / 2)
  .attr('dy', '.35em')
  .attr('text-anchor', 'end')
  .attr('transform', null)
  .text(d => d.name)
  .filter(d => d.x < width / 2)
  .attr('x', 6 + sankeyGenerator.nodeWidth())
  .attr('text-anchor', 'start');
```

Nothing too weird here; we add `6` to the width of our nodes to offset them a little bit, I've just chosen `6` because it seems to work well. Yay for magic numbers!

Click on Save and check your browser; you should have a Sankey diagram, as illustrated:



Notice the weird "UKIP 2010" label hanging out in the bottom right-hand corner? Remember that, it'll be important later on in this chapter when we do unit testing.

Pretty nice, huh? From this, we can see, at a glance, that both the Liberal Democrat and Labour parties lost a lot of support, with many of the Liberal Democrat voters going to the Labour and Conservative parties, and a lot of Labour voters going to the *other* category (although it's not obvious, given that we've grouped national and alternative parties into a single *other* group, the vast bulk of Labour losses in 2015 were in Scotland to the Scottish Nationalist Party, which promised to vote similarly to Labour but with a preference for Scottish interests during the election). And even though the Conservatives were immensely concerned about the effect UKIP would have, only one of their seats went to them. I'm sure former Prime Minister David Cameron wishes he had known that result before he promised that whole *Brexit* referendum prior to the election!

Okay, enough jabbering on about British politics. Hopefully, what I've demonstrated with this example is that TypeScript looks very similar to normal

ES2017 code, with a few minor exceptions here and there. If you're using popular libraries with high-quality definitions, the added development effort to incorporate TypeScript might be worth the instant feedback and internal documentation it gives you.



What if you're using a library that doesn't have a TypeScript definition? You have two options: either declare it using the `any` type, or turn `noImplicitAny` to `false` in `tsconfig.json`. This is probably easier than writing your own definition for a library, but it also loses the benefits that come with TypeScript. It's worth considering what libraries you'll use before you incorporate TypeScript, as dealing with an untyped library can add a lot of headache to a project, as it did for me when I was initially creating this example!

This was a ludicrously shallow overview of TypeScript that barely scratches the surface, but hopefully you've given it a go using a good editor and plugin combo and are already sensing how powerful it can be. To be able to use TypeScript effectively, you should know at least how to typecast each variable and create things such as interfaces. If TypeScript interests you as a technology, I highly recommend reading the handbook at <http://www.typescriptlang.org/Handbook>.

Behavior-driven development with Mocha and Chai

All of these will get you pretty far toward, with more confidence in your visualizations, but another step you can take to be even more of a rockstar is to add automated testing to your projects.

There are many reasons to write automated tests: if you have a product that has to reliably render charts, and the chart is rendering merely a part of a much larger application, you 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 that, automated tests are great if you want to be able to show your editor proof that your chart is working and accurate, or if you merely want to have more confidence in your data visualization work.

There are fundamentally two ways 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 should be seen as the hallmark of having reached some degree of skill with JavaScript. An extension of that 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 nicer to read and write, and that's what I'll use here, via the Chai library.

There are several types of automated test you can do, but we will 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 4, 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 that 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, you switch to your command line's test runner instead, which will explicitly tell you whether something worked or not.

This can often be much faster, which helps to offset the time cost of writing the tests before anything else. You can even set up your test runner (in our examples, we'll be using Mocha) so that it runs inside your editor!

Functional testing, on the other hand, is more comparable to looking at how the application behaves in a consumer context. Imagine that you buy a new phone. The phone has had each of its components tested to 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. Comparatively, a functional test will evaluate whether you can do certain activities successfully--can you open a web browser and navigate to your favorite website? If you press the *increase volume* button, does the volume increase? Are you able to successfully navigate to the clock app and set an alarm for the next morning? These end-user structured workflows are what make up good functional tests.

BDD is more geared toward 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

We will use two things to write our unit tests: Mocha and Chai. Mocha is the process that makes all of your tests run, and Chai is a library that allows us to write nice-looking assertions. An assertion is effectively a statement saying that a behavior works as intended; the most basic of these is *equals*, which takes the *actual equals expected* format. Everything else is effectively just syntactic sugar for this assertion and makes your tests (debatably) nicer to read.

Install all of them to your project first, along with a few other goodies:

```
| $ npm install mocha chai mocha-loader --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 will also create a new Webpack config file for testing. Create a file called `webpack.test.config.js` in your project root and add the following to it:

```
const path = require('path');
module.exports = [
  {
    output: {
      path: path.resolve(__dirname, 'build'),
      publicPath: 'assets',
      filename: 'bundle.js',
    },
    devtool: 'inline-source-map',
    module: {
      rules: [
        {
          test: /\.ts$/,
          exclude: [(node_modules|bower_components)],
          loader: 'ts-loader',
        },
        {
          test: /\.js$/,
          exclude: (node_modules|bower_components),
          loader: 'babel-loader',
        },
      ],
    },
  ],
];
```

Let's get a feel for BDD-style development with Mocha and Chai by updating our TypeScript class from earlier with a few new behaviors. Also, add the following to the scripts stanza of `package.json`:

```
| "start-tests": "webpack-dev-server --config webpack.test.config.js 'mocha-loader!./lib/
```

Now, you can run and develop tests with the server running in the background, by typing the following:

```
| $ npm run start-tests
```

Testing behaviors - BDD with Mocha

We really will not do this BDD-style because we have very little space left in the book and writing about automated testing to any significant degree can fill way more pages than we really have left. We will update our `typescriptsankey` function to filter by a particular relationship when the user clicks on a node; however, do so in a BDD fashion, which should hopefully give you a taste of how all this works. Just remember that testing is generally best when you write your tests *before* you write any other code!

To start, create a new file in `lib/chapter9`, called `index.spec.js`. The convention is to name your test file 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 reduce the opacity of irrelevant segments when a node is clicked on
- The relevant segments should remain at full opacity
- Clicking on a node again should return opacity to full for all parts of the chart

As you'll see, our assertions in Mocha will resemble these statements quite closely. Open up `index.spec.js` and add the following:

```
import * as d3 from 'd3';
import chai from 'chai';
import sankey from './index.ts';

chai.should();

describe('functional tests for UK election sankey', () => {
  describe('select() method', () => {
    it('should set change opacity when supplied an argument',
      async () => {
        const chart = await sankey();
      });
  });
});
```

First, we import our chart library and our data and set up our parent `describe` block. The `describe` tag 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 the assertions 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 to verify what the test implies.

We will exercise each part of the chart to ensure that it returns the right data. We'll create a helper function first to save us some typing. Put this inside the `it` block:

```
function getOpacities(source = null) {
  chart.select(source);

  const links = d3.selectAll('.link[opacity="1"]')
    .data()
    .map(d => d.target.name)
    .sort();
  const nodes = d3.selectAll('.node[opacity="1"]')
    .data()
    .map(d => d.name)
    .sort();

  return {
    links,
    nodes,
  };
}
```

It's time to write our tests. As our chart is an `async` function, we'll use a few more `async` functions to test each part of what we're doing:

After `getOpacities()`, add the following:

```
await (async () => {
  const { links, nodes } = getOpacities('CON2010');
  links.should.eql(['CON2015', 'LAB2015', 'UKIP2015']);
  nodes.should.eql(['CON2010', 'CON2015', 'Green2015',
    'LAB2015', 'LIB2015', 'Other2015', 'UKIP2015']);
})();
```

This returns two arrays from `getOpacities()`, which we compare against the expected results. Note that we sorted our values in `getOpacities()`, so our *expected* values should reflect that.

Here are the rest of the tests; put them directly after the last one:

```
await (async () => {
  const { links, nodes } = getOpacities('LAB2010');
  links.should.eql(['CON2015', 'LAB2015', 'Other2015']);
```

```

    nodes.should.eql(['CON2015', 'Green2015', 'LAB2010',
                      'LAB2015', 'LIB2015', 'Other2015', 'UKIP2015']);
})();

await (async () => {
  const { links, nodes } = getOpacities('LIB2010');

  links.should.eql(['CON2015', 'LAB2015', 'LIB2015',
                    'Other2015']);
  nodes.should.eql(['CON2015', 'Green2015', 'LAB2015',
                    'LIB2010', 'LIB2015', 'Other2015', 'UKIP2015']);
})();

await (async () => {
  const { links, nodes } = getOpacities('Green2010');

  links.should.eql(['Green2015']);
  nodes.should.eql(['CON2015', 'Green2010', 'Green2015',
                    'LAB2015', 'LIB2015', 'Other2015', 'UKIP2015']);
})();

await (async () => {
  const { links, nodes } = getOpacities('Other2010');

  links.should.eql(['LIB2015', 'Other2015']);
  nodes.should.eql(['CON2015', 'Green2015', 'LAB2015',
                    'LIB2015', 'Other2010', 'Other2015', 'UKIP2015']);
})();

```

It's pretty similar all around. Let's add a final test to exercise what happens when we deselect everything:

```

await (async () => {
  const { links, nodes } = getOpacities(null);
  links.should.have.length(13);
  nodes.should.have.length(11);
})();

```

Here, we're being a bit lazy and just checking the length of the result array. We can check the actual versus expected values again if we wanted, though.

Now that we've built our tests, it's time to make our code pass them! Start up our server by running the following:

```
| $ npm run start-server
```

You'll see a message about our tests failing, like this:

passes: 0 failures: 1 duration: 0.22s (100%)

functional tests for UK election sankey

select() method

* should set change link and node opacity when supplied an argument

```
TypeError: Cannot read property 'select' of undefined
at getOpacities (assets/bundle.js:23127:14)
at assets/bundle.js:23139:34
at Generator.next (<anonymous>)
at step (assets/bundle.js:23541:30)
at assets/bundle.js:23559:14
at F (assets/bundle.js:2801:28)
at assets/bundle.js:23538:12
at Context.<anonymous> (assets/bundle.js:23143:9)
at Generator.next (<anonymous>)
at step (assets/bundle.js:23541:30)
```

This is actually a good thing; it shows that we will not get false positives when our tests actually start passing. False positives are a hazard of testing; if you don't do an asynchronous test properly, it will still pass. Luckily, we're using `async/await` here, which abstracts away a lot of the problematic bits. Normally, you pass a function as the first argument in your `it()` declarations called `done()`, which you then call once your `async` operation and assertions are complete. Still, always ensure that your tests fail properly while you're writing them!

Another upside to these tests failing is that 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.

Let's fill in the rest of our chart. Add the following to the bottom of `typescriptSankey()` in `index.ts`:

```
const select = (item: string|null) => {
  if (item) {
    const filteredLinks = sankeyData.links.filter(d =>
      d.source.name === item);
    const filteredNodes = sankeyData.nodes
      .filter(d => d.name === item || d.name.match(/2015$/));

    svg.selectAll('.link')
      .attr('opacity', d => d.source.name === item ? 1 : .3);
    svg.selectAll('.node')
      .attr('opacity', d => (d.name === item ||
        d.name.match('2015')) ? 1 : .3);
  } else {
    svg.selectAll('.link')
      .attr('opacity', 1);

    svg.selectAll('.node')
```

```
| }     .attr('opacity', 1);  
| } }
```

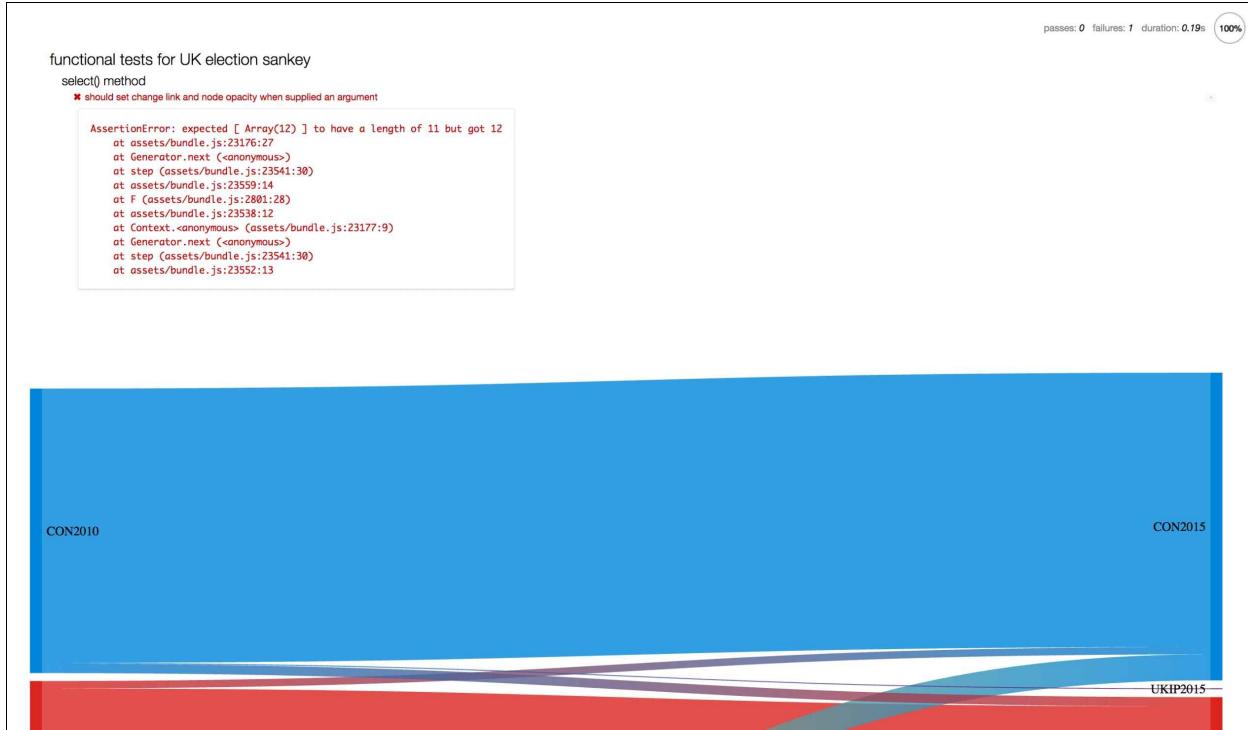
Now, we add our event handlers:

```
let current = null;  
node.on('click', e => {  
  if (current === e.name) {  
    current = null;  
  } else {  
    current = e.name;  
  }  
  
  select(current);  
});
```

Finally, return our methods to resolve the `async` function promise:

```
return {  
  select,  
  node,  
  link,  
  data: sankeyData,  
};
```

Click on Save and we're good to go!:

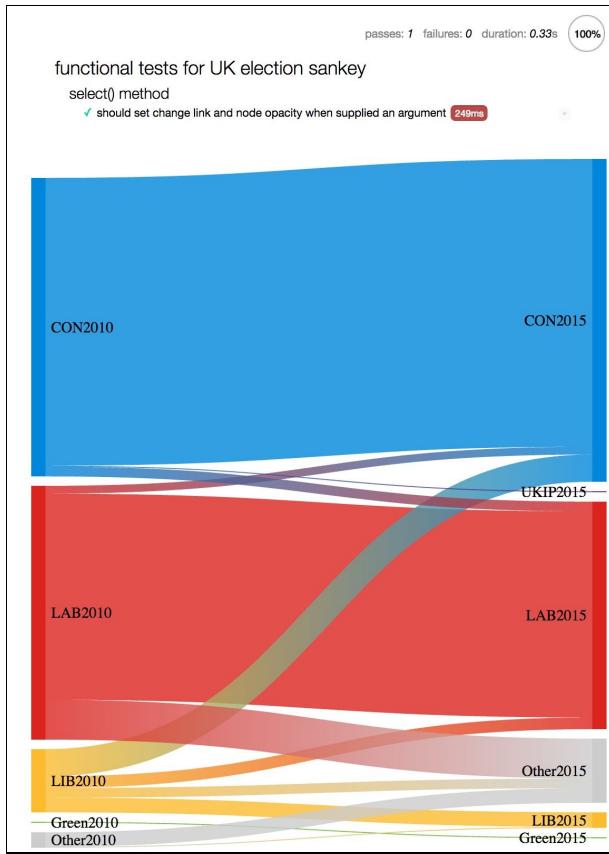


Ah, rats! We still have a failing test. It's due to that superfluous *Ukip2010* node we had earlier; I kind of messed up when creating the dataset and couldn't figure out how to remove it. Let's make it work, though!

Add this bit before our `select()` method in `index.ts`:

```
node.selectAll('rect[height="0"]')
  .each(function(d){
    this.parentNode.remove();
 });
```

This is kind of a hack, but let's work with it. We select all the rectangle nodes without any height (such as our "UKIP2010" node), and remove the parent, which effectively gets rid of the misplaced "UKIP 2010" label. Hit Save:



All of our tests are now passing and we are now Tony Doneza!

Once again, there's absolutely no way I can go into a topic this dense 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 of 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 such as 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 feeling like you're an awesome automated test writer, do this fun 2-minute interactive quiz from the NYT and then, if you get it wrong, write more negative test assertions. You can find the quiz at <http://www.nytimes.com/interactive/2015/07/03/upshot/a-quick-puzzle-to-test-your-problem-solving.html>.



Summary

This chapter was all about tooling. We started with linting using ESLint, then moved up to static typing using TypeScript, then finished with some testing in Mocha. That's a lot of ground to cover! Don't feel bad if you're a bit overwhelmed by it all right now, this was meant to be a really shallow overview of these tools and how to use them with D3. There are lots of resources you can pursue now that you know the basics; I highly recommended reading a little more about testing because that's really probably the biggest and most important of all of them.

We will close this book by looking at some particularly impressive examples of data visualization. Don't worry, we're done with coding now; sit back, relax, and enjoy the show as we take a quick look at what makes for a quality data visualization.

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 that 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 similarly 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 such as *The Financial Times*, *The Times*, *The Economist*, and *The Guardian*. These are very fast-paced newsrooms, and the goal 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 visual journalist, 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.

That caveat out of the way, let's go on with discussing a few of the finer points of what exactly comprises good data visualization.

Choosing the right dimensions, choosing the right chart

The first step toward making a solid piece of data visualization is choosing the right type of chart for a project. Quite frequently, data will contain many, many dimensions. It's your job to choose which dimensions to visualize. Similarly, it's your job to choose *how* to visualize those dimensions.



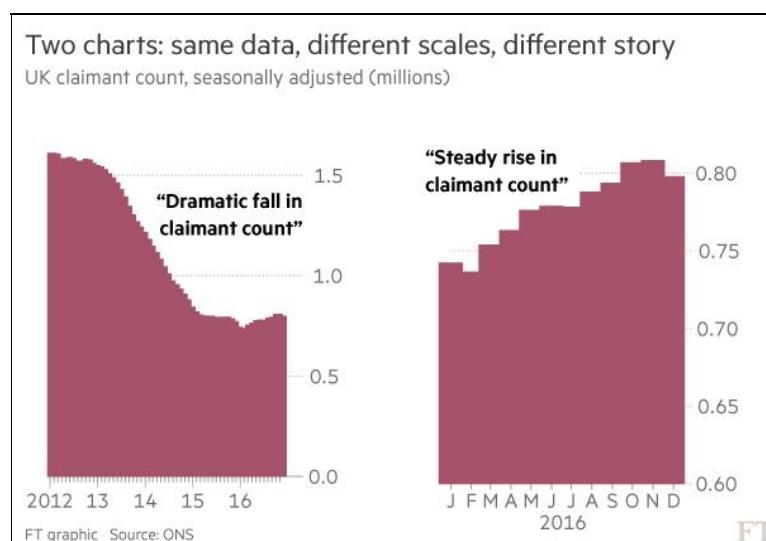
It's very easy to get sucked into thinking that you need a map for everything. Nowhere is this more true than in the creation of content for elections; everyone thinks of the giant choropleths behind newscasters as they watch results come in, each region turning the hue of the party it voted for. However, maps are primarily beneficial for depicting geographic data and the proximity of things in a physical space. In an American election, Idaho and New Hampshire both have four electoral college seats, but the latter is a fraction of the former's size. New York state is geographically much smaller than Montana, but it has 29 electoral seats to Montana's three. Unless the intent is to show that states in a particular section of the country all vote a particular way, there really isn't much value to displaying that data as a map (beyond the familiarity a map brings, in terms of displaying, an entire country in a recognizable format).

My team at the *Financial Times* uses a tool called the Visual Vocabulary, which is inspired by Jon Schwabish and Severino Ribecca's Graphic Continuum. To use it, you choose the dimensions of data you want to depict and it suggests several types of charts you can use to do so. Not only that, but the GitHub repo also contains D3 examples of each chart type. To use it, visit ft-interactive.github.io/visual-vocabulary and fork the examples repo at github.com/ft-interactive/visual-vocabulary.

Clarity, honesty, and a 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 that 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.

Axes and scales are one of the easiest things to get wrong. You should usually start them at zero, because not doing so can dramatically distort the shape of the chart and hide information from the viewer. For time series data, the amount of data you show can also impact how a chart is perceived. Here's an example from Alan Smith's *Chart Doctor* series:

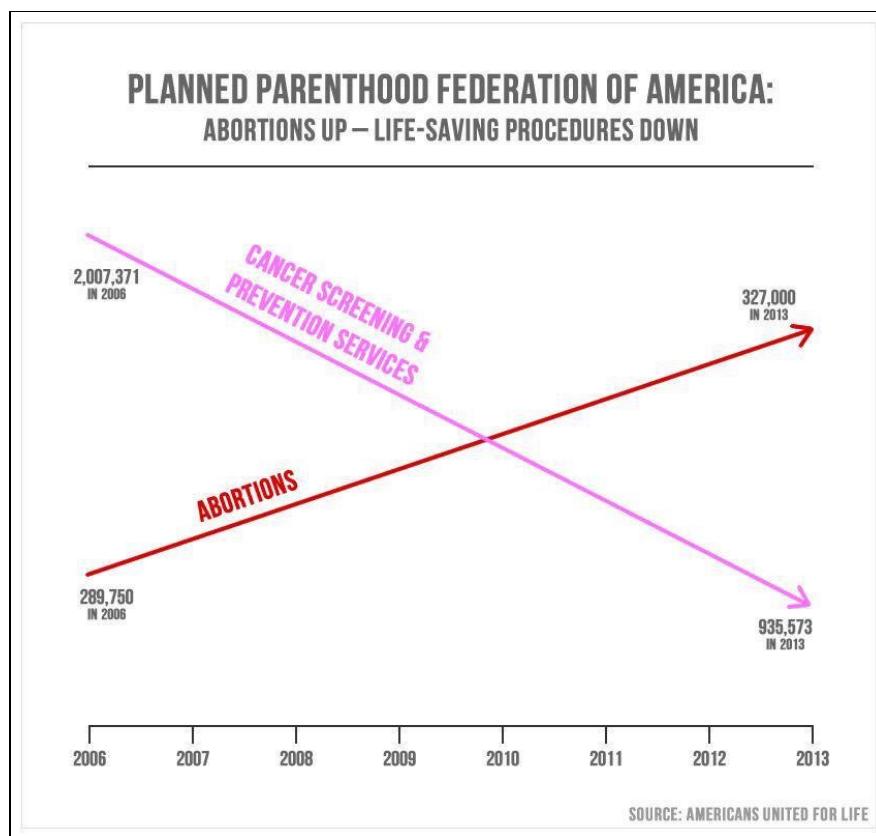


Taken from *How alternative facts rewrite history* by Alan Smith: <https://www.ft.com/content/3062d082-e3d>

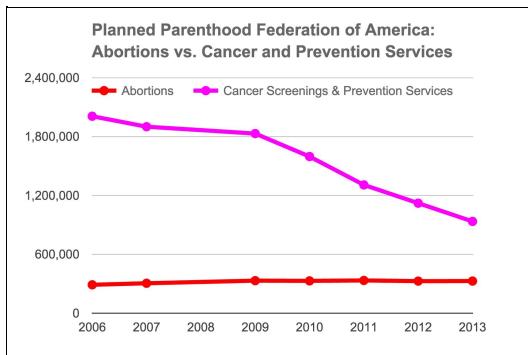
a-11e6-8405-9e5580d6e5fb

The preceding screenshot shows how simply changing the duration of a chart can impact what it's saying. They both show the same dataset, which is the number of people in the UK claiming job seeker's allowance and can prove that they're looking for work, which to some degree is a proxy for the unemployment rate. The chart to the right is particularly misleading because the y axis doesn't start at zero, which exaggerates the 0.2 million increase in 2016. The chart on the left, meanwhile, starts at zero and while one can see the rise depicted in the second chart, the added perspective of portraying the change since 2012 shows that the trend has been effectively flat since 2015, and far less than it was just a few years earlier, in 2012.

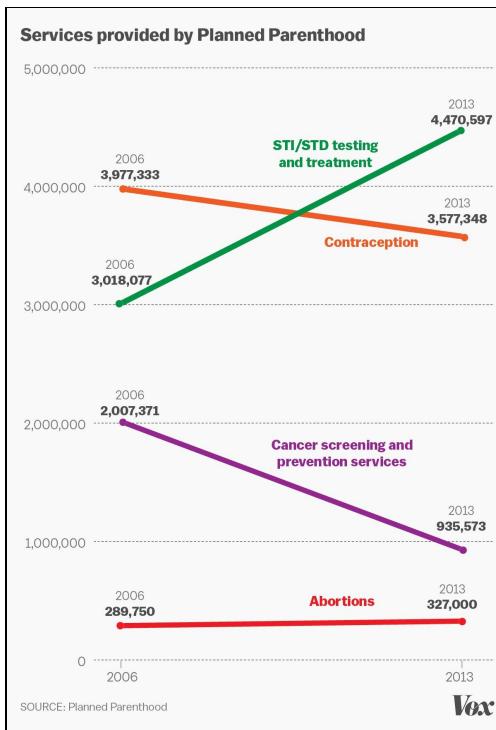
Here's another contemporary example. In September 2015, the U.S. Congress held a hearing on Planned Parenthood, the American reproductive and women's health group. During the hearing, Republican congressman Jason Chaffetz showed the following chart created by anti-abortion group, Americans United For Life:



There are many problems with this chart, not least the complete lack of a y axis. 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 had 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 had 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 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 explain to 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. Whatever you do, be clear with your intentions and never mislead.

Sometimes, however, you need to pay attention to what the data is saying and where the drama lies. A good example of when not to start a chart at zero is when the data never goes near zero. If you build a stacked area chart depicting votes by party in U.S. federal



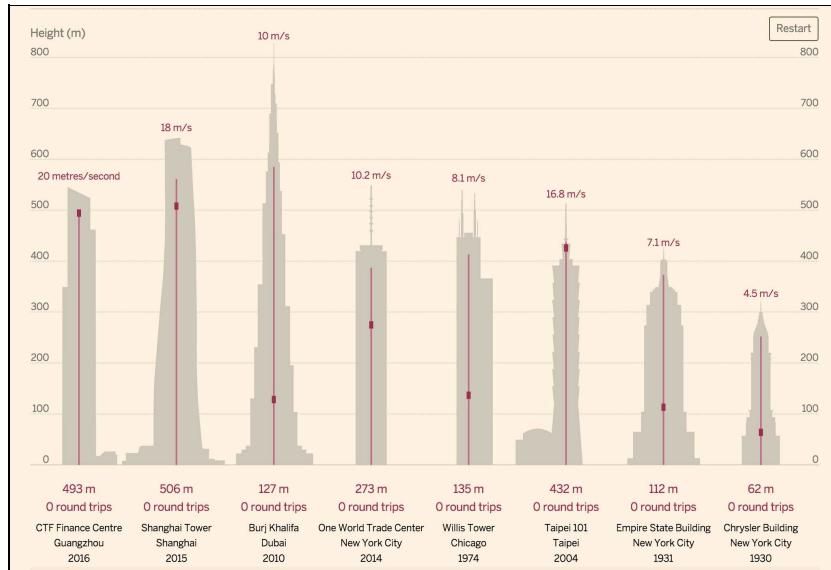
elections over two decades, starting the y axis at zero will effectively show a flat line. Similarly, if you plot the Dow Jones Industrial Average since 1900 starting at zero using a linear scale, the entirety of the Great Depression is basically flattened into oblivion (in fact, when depicting data that scales multiple orders of magnitude, it's quite possible that you will want to use a logarithmic scale; for a good overview of when and why to use log scales, see the Chart Doctor piece mentioned at the beginning of the chapter).

Helping your audience understand scale

A big part of visualizing data is conveying scale and differences in magnitude. The following examples do this particularly well.

To start with, view John Burn-Murdoch's graphic on high-speed elevators for the Financial Times at <http://www.ft.com/cms/s/2/1392ab72-64e2-11e4-ab2d-00144feabdc0.html>.

The following screenshot doesn't really do it justice:

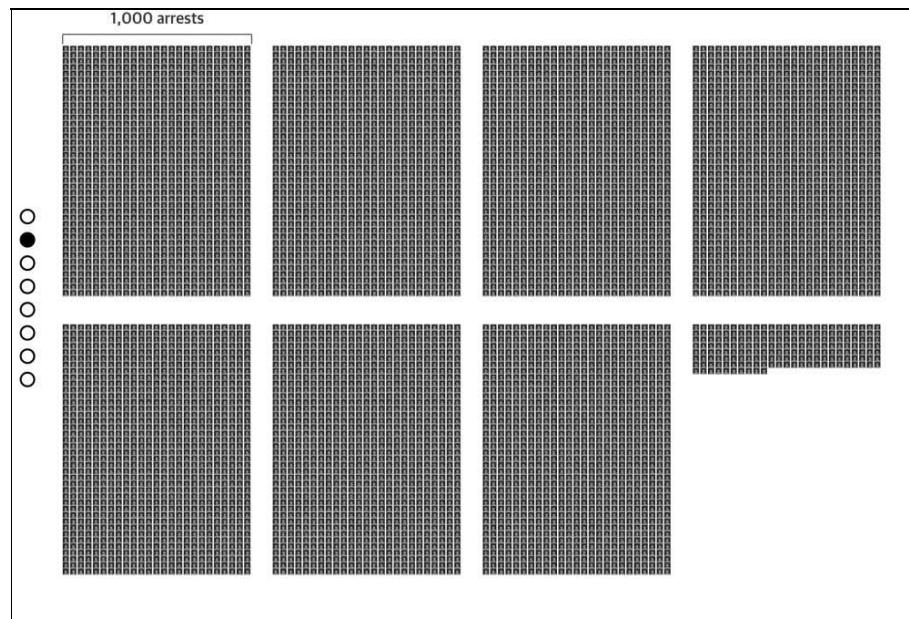


If the preceding screenshot were a live visualization, you would see the elevators at each building endlessly rise and fall, with a counter beneath tracking how many times the elevator has gone up and down while looking at the page. A nice bit of easing at the top and bottom makes you feel that the little magenta square travelling 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--the speed of the elevator. In this, one can really see the power of using the digital medium to communicate data in ways a

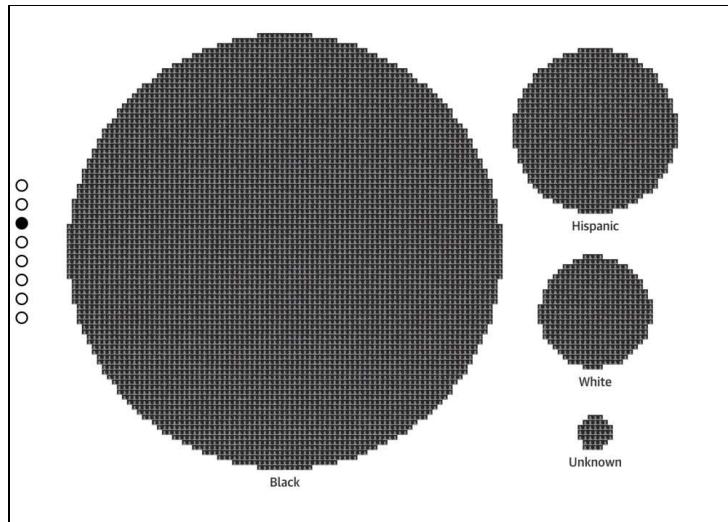
static print version will never be able to. Scale is demonstrated by tying visual content to time; in this case, merely saying *the elevator can do a return trip in two minutes* would not be nearly as effective as demonstrating what two minutes feel like to the reader.

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. If you haven't seen it, visit [http://www.theguardian.com/us-news/ng-i](http://www.theguardian.com/us-news/ng-interactive/2015/oct/19/homan-square-chicago-police-detainees)nteractive/2015/oct/19/homan-square-chicago-police-detainees and be prepared to have your mind utterly blown.

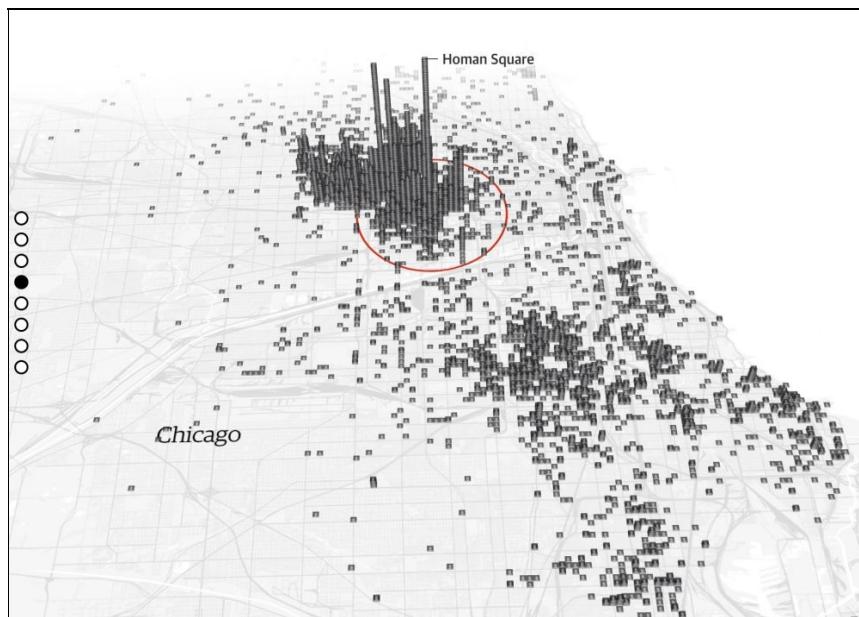
It starts with a big collection of faceless grey silhouettes, each representing a single person in custody at a secretive police warehouse in Chicago:



As you scroll, the faces fly around the screen to make up different configurations, such as this bubble chart:



They can also make this isomorphic map:

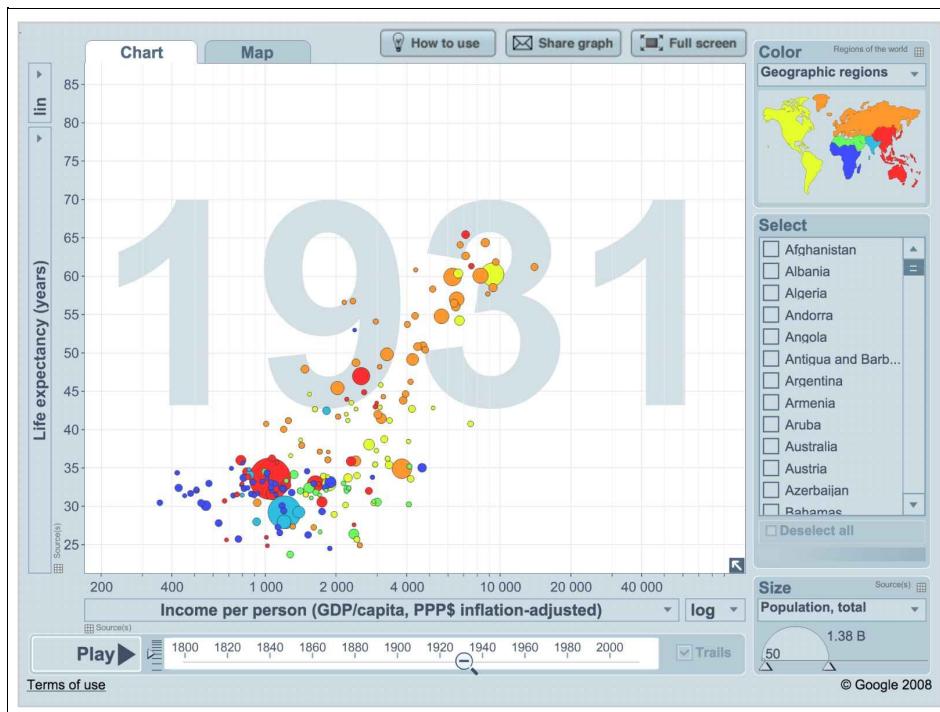


The Guardian U.S. interactive team did a fantastic Q&A about the process behind this visualization that is well worth reading; you can find it at <https://source.opennews.org/en-US/articles/how-we-made-homan-square-portrait/>.

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).

Your projects don't need to be as complex as *Homan Square* to be compelling, however. Another example demonstrating good use of animation to explain scale is the late, great *Hans Rosling's Gapminder project* (<http://www.gapminder.org/world/>), which allows viewers to see how the world has changed over time. In his many excellent talks about understanding the world from a data-driven perspective, Rosling discussed how, when 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:

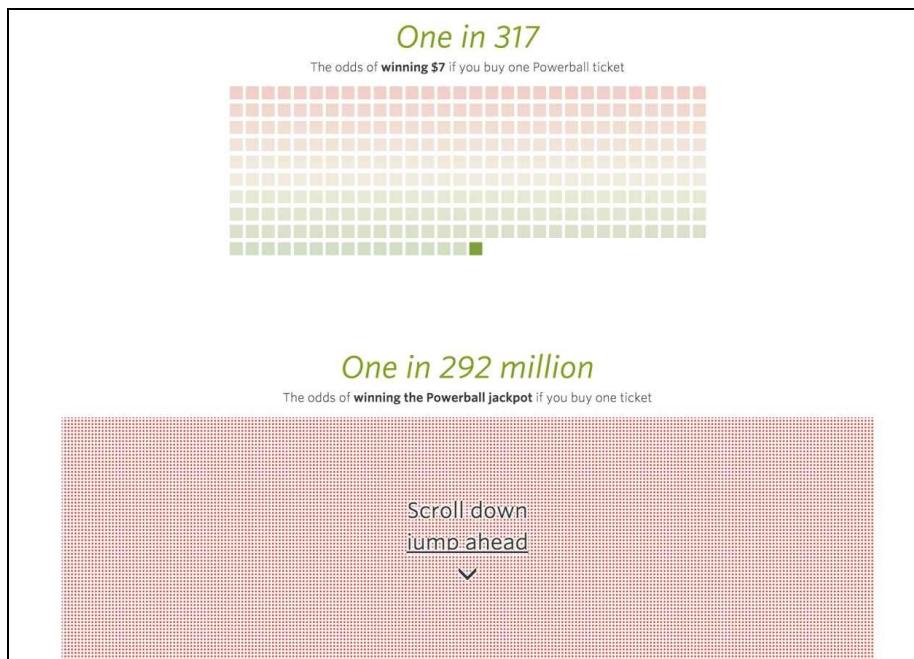


As time goes on, you can watch as the bubbles in the chart migrate from the bottom-left quadrant to the upper-right, showing increased life expectancy and income per person across the board. It's worth comparing the fairly-dry exploratory visualization to Rosling's TED talk (https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen). 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 what can be rendered on a screen.



You can see an example of the preceding chart recreated using D3 by visiting Mike Bostock's implementation at <http://bostocks.org/mike/nations/>.

As a final example, please visit this graphic by the Wall Street Journal that attempts to visualize what your chances of winning the Powerball lottery jackpot are; it's another one where the following screenshots really won't do it justice: <http://graphics.wsj.com/lottery-odds/> Also, once there, try not to hit the *jump ahead* button for at least a little while:



After a good deal of continuous scrolling, the screen looks 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 that 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 speed interactive earlier), in as much that the physical effort involved to scroll just 1% to the end (much less count how many dots that is) very effectively demonstrates how mind-bogglingly big a number like 292 million is in the context of comparing probabilities.

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 Chaffetz chart in the last 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 on screen.

Using color effectively

One of the biggest choices you'll make when building data visualizations is choosing which 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 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, you'll definitely 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, 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 such as 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 dataset? Colors are very 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 prebuilt 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! The scale in `d3-color`, `d3.schemeCategory20c`, contains the colors it uses.

An easier way to use these is via the `colorbrewer` package:

```
$ npm install colorbrewer --save
```

Then, `require('colorbrewer')` as normal. For the preceding 7-Class BuGn scale (look next to the export pane for the scheme's name in the Colorbrewer2 UI), you'd write:

```
import BuGn from 'colorbrewer';
const sevenColorBlueGreen = 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 similar 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

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 the topic at hand? 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 the one hand, it never hurts to figure out the rough shape of your data before committing to a particular visualization style. Frequently, I've been asked for pie charts with a few small outlier values highlighted, which totally 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 at this stage, before committing the idea to code.

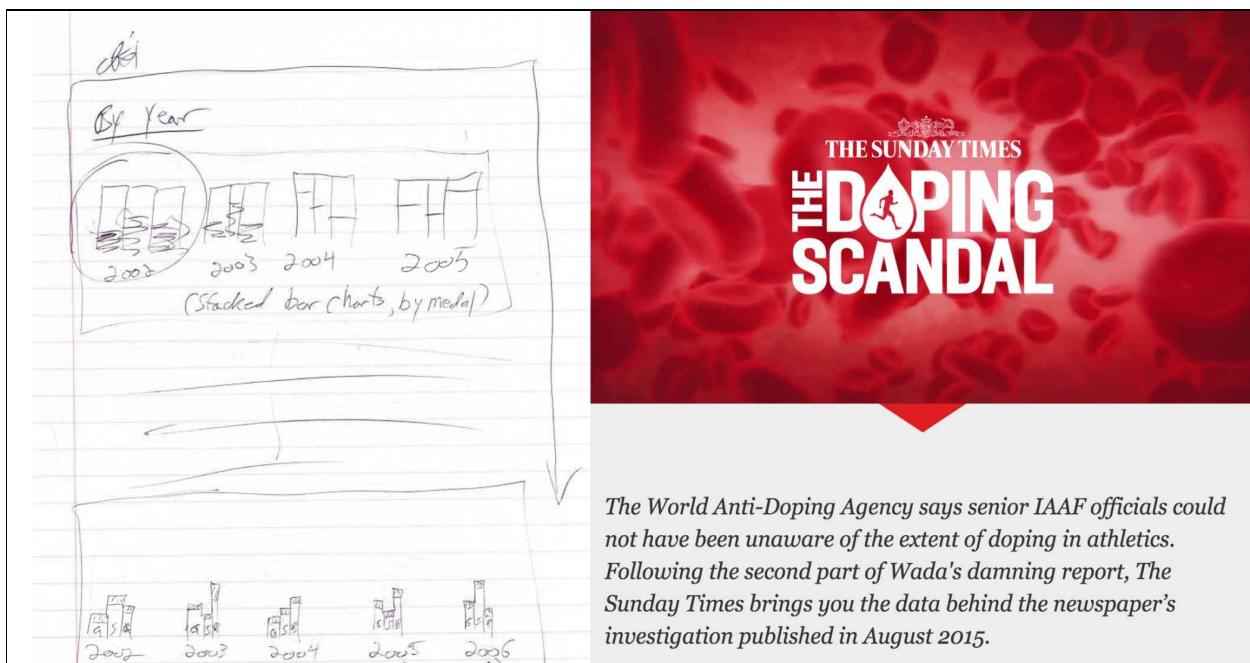
The second way you should sketch out your visualization is on a component or an interaction level. This is where understanding which devices your audience uses is important. If you have previous work out there, look at its analytics. To see what percentage of readers use a specific browser in *Google Analytics*, look under `Audience > Technology`. Pay attention to `Audience > Mobile > Overview` also, which will tell you what proportion of your audience comprises mobile and desktop. If you have no analytic data to work with (for instance, 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 are begun 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 shrinking 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 page designed first and out of the way, and can let things grow as screen resolution increases.

Whether you need to do either depends on your audience--it's quite possible that it predominantly uses one platform or the other, reducing the need to accommodate both. Again, check your analytics. That said, while ensuring that things work well, cross-device and cross-browser take 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, drop-down boxes, radio buttons, and so on) or chart to go. How you distinguish each element from the others is entirely up to you just ensure 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:



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 very key ways, and understanding these is crucial for building data visualizations that are effective on both the platforms.

Mobile

- Has multiple screen orientations
- Has uniformly-small screen dimensions
- Does not have a pointer; interaction is derived from touch gestures and has no *hover* state
- Often relies on 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 and 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; can also 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 mobiles

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 that one column is in a full row extending perpendicularly to the column's direction. Paragraphs should always flow vertically (instead of horizontally in horizontal columns), and it's not an awful idea to rotate bar charts 90° so that your bars aren't squished in one narrow horizontal row. Much like the preceding WSJ lottery probabilities graphic, make use of the infinite vertical space you have to scroll.

On the 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 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, at <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, at <http://flexboxgrid.com/>.

Be sparing with animations on mobiles

Animation on mobiles 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 mobiles, try to only use CSS transitions as these are more performant than iterating through properties via JavaScript. When in doubt, disable animation on mobiles.

Realizing similar UI elements react differently between platforms

Things such as radio buttons, sliders, and checkboxes are available both on the mobile and desktop, but some are easier to use on mobiles 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 drop-downs), make individual form elements stretch the entire device width on mobiles, or in the case of things such as checkboxes and radio buttons, use the `for` property of the `<label>` element to make labels tapable, and scale these horizontally.

Avoiding 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 desktops, it's a really terrible anti-pattern referred to as *mystery meat navigation* when on mobiles. When you hover over a button, you're not committing to clicking it. However, because mobile devices lack a hover state, users must necessarily commit to a user interaction to understand what a button even does. Particularly, 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 mobiles.



For some good mobile icons, try Font Awesome (<http://fontawesome.github.io>) and Google's Material Icons (<https://design.google.com/icons>).

Be wary of the scroll

A common user interaction idiom on desktops (popularized by the New York Times's *Snowfall* long-form piece) is to tie animations to the browser's scroll event. This is frequently fraught with peril on mobiles 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 not safe to assume that a scroll-dependent animation will play properly and won't feel unperformant and *janky* on mobiles. 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 mobiles.

Summary

Hopefully, by now you feel that 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 in doing so. We also discussed ensuring that your work functions well on mobile.

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 10 chapters; writing a textbook is a fine balance of getting to the quick and dirty learning bits while also having some fun along the way. Regardless, if you somehow read this thing from beginning to end, my hat's 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 DOM and CSS. Then, we delved into SVG and learned how to build super pretty web vector graphics, after which we started doing some really neat stuff with D3--remember that chapter on layouts, when we made like a gazillion charts?! We did some cool stuff with Node.js and Canvas, we made a boatload of cool maps; 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 stuff with all this technology, and being confident in our work. Whether it's looking to others for inspiration or wiring up a solid test suite to ensure that our work is accurate, data visualization is a craft, one that will only continue to grow in importance as our world becomes ever-more data rich, but one that also requires a degree of precision. Don't worry if you get some things wrong, but do try your best to get everything right.

We are in a truly amazing era of the web where the restrictions that were holding developers back 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's on the cutting-edge. There is no other field in computing where you can build things that work so *universally*, 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 your possession, a massive warchest of tools that you can try to use. You clearly need to learn a lot more about some of them (particularly the ones I sprinted through, such as Node, Canvas, and TypeScript) to use them 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 somewhere 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 around who's been there and is willing to help.