

Ændrew Rininsland, Michael Heydt,
Pablo Navarro Castillo

D3.js: Cutting-edge Data Visualization Learning Path

Turn your raw data into real knowledge by creating and deploying complex data visualizations with D3.js



Packt

D3.js: Cutting-edge Data Visualization

Turn your raw data into real knowledge by creating and deploying complex data visualizations with D3.js

A course in three modules

Packt

BIRMINGHAM - MUMBAI

D3.js: Cutting-edge Data Visualization

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course 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 course to ensure the accuracy of the information presented. However, the information contained in this course 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 course.

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

Published on: December 2016

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

ISBN 978-1-78728-177-6

www.packtpub.com

Credits

Authors

Ændrew H. Rininsland

Michael Heydt

Pablo Navarro Castillo

Reviewers

Elliot Bentley

Patrick Cason

Pablo Núñez Navarro

William Sankey

Andrew Berls

Simon Heimler

Lars Kotthoff

Nathan Vander Wilt

Content Development Editor

Samantha Gonsalves

Graphics

Jason Monteiro

Production Coordinator

Shraddha Falebhai

Preface

D3 is an amazing library. On its website, there are hundreds of beautiful examples, visualizations, and charts created mainly with D3. Looking at the examples, we soon realize that D3 allows us to create an uncanny variety of visuals. We can find everything from simple bar charts to interactive maps.

The ability to create almost anything with D3 comes at a price; we must think about our charts at a more abstract level and learn how to bind data elements with elements in our page. This association between properties of our data items and visual attributes of the elements in our chart will allow us to create complex charts and visualizations.

In real-life projects, we will have to integrate components and charts created with D3 with other components and libraries. In most of the examples in this module, we will cover how to integrate D3 with other libraries and tools, creating complete applications that leverage the best of each library.

D3 is a great tool to experiment with visuals and data!

What this learning path covers

Module 1, Learning d3.js Data Visualization, Second Edition, Over the course of this module, you'll learn the basics of one of the world's most ubiquitous and powerful data visualization libraries, but we won't stop there. By the end of our time together, you'll have all the skills you need to become a total D3 ninja, able to do everything from building visualizations from scratch to using it on the server and writing automated tests. As well, if you haven't leveled up your JavaScript skills for a while, you're in for a treat – this module endeavors to use the latest features that are currently being added to the language, all this while explaining why they're cool and how they differ from old-school JavaScript.

Module 2, D3.js By Example, This module uses examples that take you right from the beginning, with the basic concepts of D3.js, using practical examples that progressively build on each other both within a specific chapter and also with reference to previous chapters. We will focus on the examples created for this module as well as those found online that are excellent but could use some additional explanation. Each example will explain how the example works either line by line or by comparison with other examples and concepts learned earlier in the module.

Module 3, Mastering D3.js, In this module, we will cover reusable charts using external data sources, thereby creating user interface elements and interactive maps with D3. At the end, we will implement an application to visualize topics mentioned on Twitter in real time.

What you need for this learning path

Module 1:

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

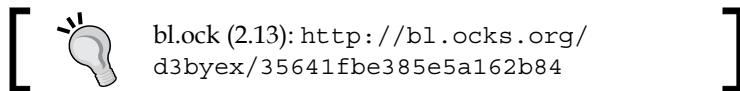
You will also need the latest version of your favorite web browser; mine is Chrome, and I use it in the examples, but Firefox also works well. You can try to work in Safari, Internet Explorer/Edge, Opera, or any other browser, but I feel that Chrome's Developer Tools are the best.

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

Module 2:

All of the tools used in this module are available on the Internet free of charge. All that is required is a modern web browser to run the samples, and all code can be edited and run online within the browser. To be specific about what makes up a modern browser, this includes Firefox, Chrome, Safari, Opera, IE9+, Android, and iOS.

All examples in this text are available to review, execute, and edit online. References to code are referred to as a bl.ock and be referenced as follows:



This will take you to a page on <http://bl.ocks.org/> for the example. This page will also contain a link to take you to jsbin.com where you can interactively make changes to the code.

Module 3:

The code bundle of this module was created using Jekyll, which is a static website generator. To run most of the examples in the code bundle, you will need a static web server and a modern web browser. The following list summarizes the main dependencies:

- A modern web browser
- D3 3.4
- Jekyll or other static web servers
- Text editor

Some chapters require you to install additional frontend libraries, such as Backbone, TopoJSON, Typeahead, and Bootstrap. Additional instructions on installing these libraries can be found in the corresponding chapters. In other chapters, we will use additional software to compile assets or process files. In those cases, installing the software is optional (the compiled files will be present as well), but it might be useful for you to install them for your own projects:

- Node and Node packages
- Git
- Make
- TopoJSON
- GDAL

Instructions to install these applications can also be found in the corresponding chapters.

Who this learning path is for

Whether you are new to data and data visualization, a seasoned data scientist, or a computer graphics specialist, this Learning Path will provide you with the skills you need to create web-based and interactive data visualizations. Some basic JavaScript knowledge is expected, but no prior experience with data visualization or D3 is required.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – 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 course'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 course, 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 course from your account at <http://www.packtpub.com>. If you purchased this course 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 course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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 course is also hosted on GitHub at <https://github.com/PacktPublishing/D3js-Cutting-edge-Data-Visualization>. We also have other code bundles from our rich catalog of course 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 courses – 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 course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, 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 course 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 course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1	1
Chapter 1: Getting Started with D3, ES2016, and Node.js	3
What is D3.js?	3
What's ES2016?	4
Summary	19
Chapter 2: A Primer on DOM, SVG, and CSS	21
DOM	21
What exactly did we do here?	28
Scalable Vector Graphics	38
Summary	72
Chapter 3: Making Data Useful	73
Thinking about data functionally	73
Loading data	85
Scales	87
Geography	96
Summary	108
Chapter 4: Defining the User Experience – Animation and Interaction	109
Animation	110
Animation with transitions	110
Interacting with the user	125
Behaviors	132
Summary	142
Chapter 5: Layouts – D3's Black Magic	143
What are layouts and why should you care?	143
Normal layouts	146
Using the histogram layout	146

Table of Contents

Baking a fresh 'n' delicious pie chart	152
Showing popularity through time with stack	157
Highlighting connections with chord	163
Hierarchical layouts	174
Drawing a tree	177
Showing clusters	180
Summary	189
Chapter 6: D3 on the Server with Node.js	191
Readying the environment	191
All aboard the Express train to Server Town!	193
Proximity detection and the Voronoi geom	195
Rendering in Canvas on the server	201
Deploying to Heroku	205
Summary	207
Chapter 7: Designing Good Data Visualizations	209
Clarity, honesty, and sense of purpose	210
Helping your audience understand scale	212
Using color effectively	219
Understanding your audience (or "trying not to forget about mobile")	221
Summary	225
Chapter 8: Having Confidence in Your Visualizations	227
Linting all the things	229
Static type checking with TypeScript and Flow	231
Behavior-driven development with Karma and Mocha Chai	241
Summary	249
Module 2	251
Chapter 1: Getting Started with D3.js	253
A brief overview of D3.js	254
Tools for creating and sharing D3.js visualizations	259
Google Chrome and Developer tools	263
Hello World – D3.js style	264
Examining the DOM generated by D3.js	267
Summary	268
Chapter 2: Selections and Data Binding	269
D3.js selections	270
D3.js and data binding	275
Summary	291

Table of Contents

Chapter 3: Creating Visuals with SVG	293
Introducing SVG	293
The basic shapes provided by SVG	297
Applying CSS styles to SVG elements	300
Strokes, caps, and dashes	301
Applying SVG transforms	304
Groups	309
Transparency	310
Layers	311
Summary	312
Chapter 4: Creating a Bar Graph	313
The basic bar graph	313
Margins and axes	318
Summary	328
Chapter 5: Using Data and Scales	329
Data	330
Scales	337
Summary	346
Chapter 6: Creating Scatter and Bubble Plots	347
Creating scatter plots	348
Creating a bubble plot	355
Summary	358
Chapter 7: Creating Animated Visuals	359
Introduction to animation	360
Adding a fifth dimension to a bubble plot – time	369
Summary	374
Chapter 8: Adding User Interactivity	375
Handling mouse events	376
Using behaviors to drag, pan, and zoom	379
Enhancing a bar graph with interactivity	383
Highlighting selected items using brushes	387
Summary	394
Chapter 9: Complex Shapes Using Paths	395
An overview of path data generators	396
Drawing line graphs using interpolators	406
Summary	414

Table of Contents

Chapter 10: Using Layouts to Visualize Series and Hierarchical Data	415
Using stacked layouts	415
Visualizing hierarchical data	424
Representing relationships with chord diagrams	437
Techniques to demonstrate the flow of information	441
Summary	447
Chapter 11: Visualizing Information Networks	449
An overview of force-directed graphs	450
A simple force-directed graph	452
Using link distance to spread out the nodes	456
Adding repulsion to nodes for preventing crossed links	458
Summary	467
Chapter 12: Creating Maps with GeoJSON and TopoJSON	469
Introducing TopoJSON and GeoJSON	470
Creating a map of the United States	476
Styling the map of the United States	478
Creating a flat map of the world	482
Spicing up a globe	488
Adding interactivity to maps	492
Annotating a map	497
Summary	505
Chapter 13: Combining D3.js and AngularJS	507
An overview of composite visualization	508
Creating a bar graph using AngularJS	509
Adding a second directive for a donut	515
Adding a detail view and interactivity	518
Updating graphs upon the modification of details data	521
Summary	525
Module 3	527
Chapter 1: Data Visualization	529
Defining data visualization	531
Introducing the D3 library	538
Summary	542
Chapter 2: Reusable Charts	543
Creating reusable charts	543
Using the barcode chart	559

Table of Contents

Creating a layout algorithm	562
Summary	569
Chapter 3: Creating Visualizations without SVG	571
SVG support in the browser market	571
Visualizations without SVG	572
Polyfilling	584
Using canvas and D3	588
Summary	594
Chapter 4: Creating a Color Picker with D3	595
Creating a slider control	595
Creating a color picker	603
Summary	611
Chapter 5: Creating User Interface Elements	613
Highlighting chart elements	613
Creating tooltips	618
Selecting a range with brushing	623
Summary	630
Chapter 6: Interaction between Charts	631
Learning the basics of Backbone	632
The stock explorer application	634
Summary	660
Chapter 7: Creating a Charting Package	661
The development workflow	662
Creating the package contents	664
The project setup	679
Using the package in other projects	696
Summary	699
Chapter 8: Data-driven Applications	701
Creating the application	701
Hosting the visualization with GitHub Pages	726
Hosting the visualization in Amazon S3	727
Summary	729
Chapter 9: Creating a Dashboard	731
Defining a dashboard	731
Good practices in dashboard design	733
Making a dashboard	734
Summary	743

Table of Contents

Chapter 10: Creating Maps	745
Obtaining geographic data	747
Creating maps with D3	753
Summary	778
Chapter 11: Creating Advanced Maps	779
Using cartographic projections	779
Creating a rotating globe	785
Creating an interactive star map	790
Projecting raster images with D3	798
Summary	807
Chapter 12: Creating a Real-time Application	809
Collaborating in real time with Firebase	809
Creating a Twitter explorer application	813
Creating the streaming server	815
Creating the client application	830
Summary	847
Bibliography	849

Module 1

**Learning d3.js Data Visualization
Second Edition**

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

1

Getting Started with D3, ES2016, and Node.js

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

What is D3.js?

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

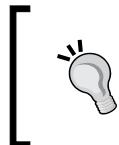
The fine-grained control and its elegance make D3 one of the most powerful open source visualization libraries out there. This also means that it's not very suitable for simple jobs such as drawing a line chart or two—in that case you might want to use a library designed for charting. Many use D3 internally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

As a data manipulation library, D3 is based on the principles of functional programming, which is probably where a lot of confusion stems from.

Unfortunately, functional programming goes beyond the scope of this book, but I'll explain all the relevant bits to make sure that everyone's on the same page.

What's ES2016?

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



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

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

Getting started with Node and Git on the command line

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

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

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

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

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

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

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

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

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

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



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

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

```
$ git checkout chapter1
```

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

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

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

```
$ npm install
```

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

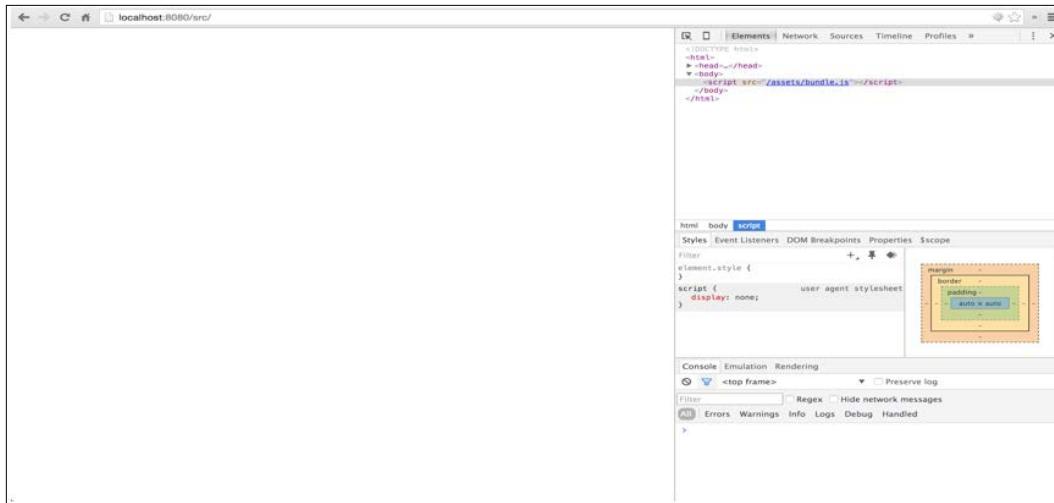
```
<!DOCTYPE html>
<div id="chart"></div>
<script src="/assets/bundle.js"></script>
```

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

```
$ npm start
```

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

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



A quick Chrome Developer Tools primer

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

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



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

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



The obligatory bar chart example

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

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

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

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

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

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

Uncaught Error: Cannot find module "d3"

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

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

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

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

```
$ npm install
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Hoorah! Another fancy new ES2016 feature!

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



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

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

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

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

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

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

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

Now the fun begins!

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

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

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

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

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

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

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

Replace it with this line:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

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

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

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

Keep this in mind when you're tinkering.

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

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

```

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

```

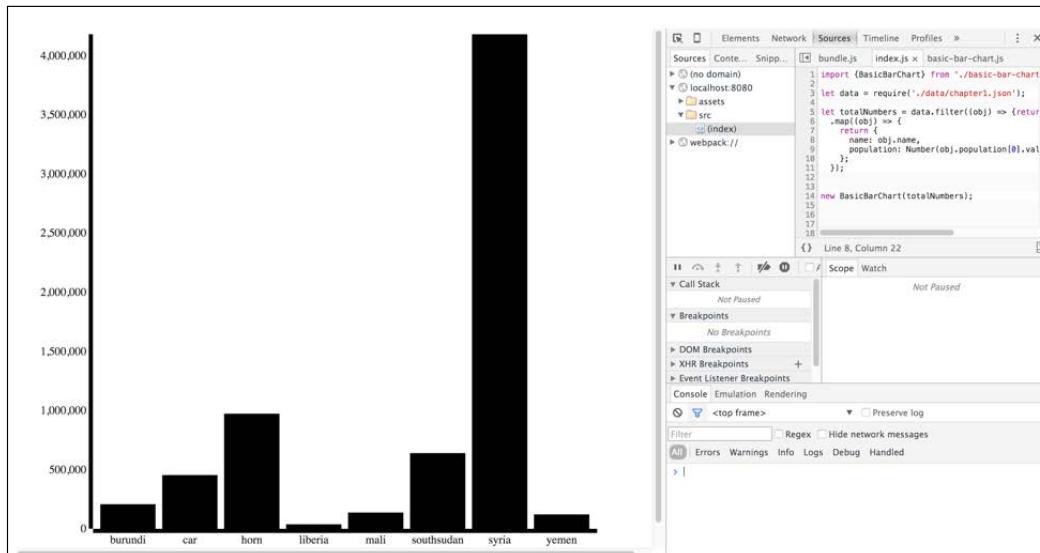
```
.delay((d, i) => { return i*20; })
.duration(800)
.attr('y', (d) => { return y(d.population); })
.attr('height', (d) => {
  return this.height - y(d.population);
});
```

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

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

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

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



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

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

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

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

```
html, body {  
    padding: 0;  
    margin: 0;  
}  
  
.axis path, .axis line {  
    fill: none;  
    stroke: #eee;  
    shape-rendering: crispEdges;  
}  
  
.axis text {  
    font-size: 11px;  
}  
  
.bar {  
    fill: steelblue;  
}
```

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

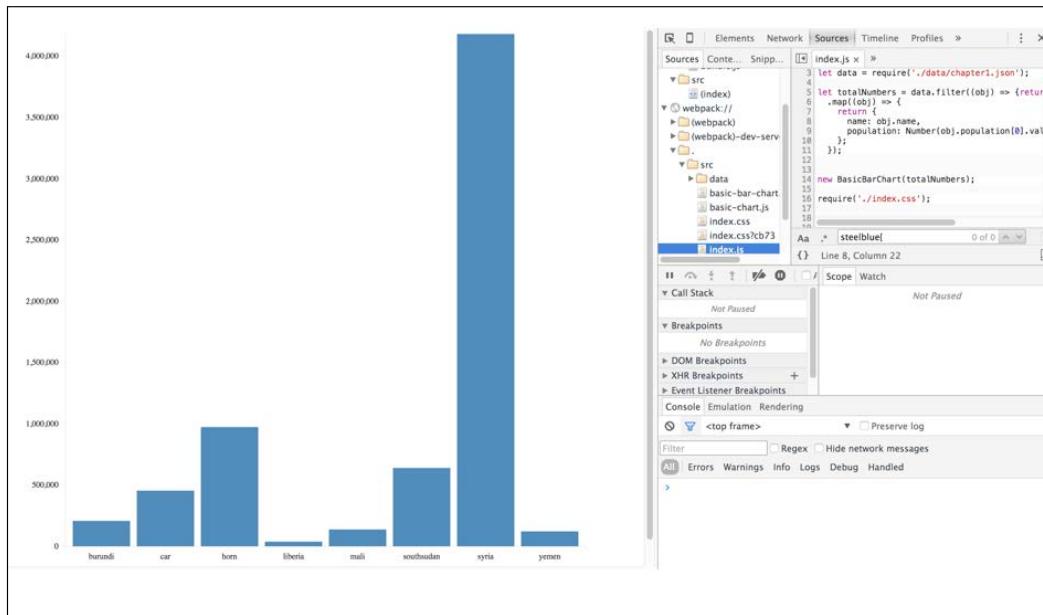
```
require('./index.css');
```

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



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

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



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

Summary

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

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

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

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

2

A Primer on DOM, SVG, and CSS

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

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

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

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

DOM

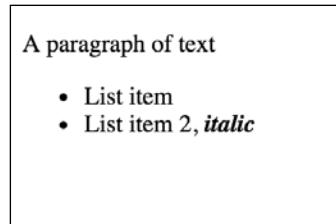
The **Document Object Model (DOM)** is a language-agnostic model for representing structured documents built in HTML, XML, or similar standards. You can think of it as a tree of nodes that closely resembles the document parsed by the browser.

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

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

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

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



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

Manipulating the DOM with D3

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

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

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

Whoa, that's a lot of code!

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

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

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

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

Let's rewrite the previous example in D3:

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

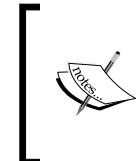
Much simpler! We selected the `strong` element and defined a `style` property. Job done!

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

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

Selections

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



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

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

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

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

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

Let's make a table!

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

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

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

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

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

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

```

var d3 = require('d3');

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

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

```

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

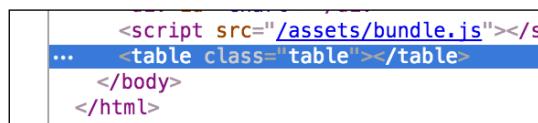
let rows = [
  header,
  ['q', 'w', 'e', 'r', 't', 'y']
];

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

Go to the command line and type this:

```
$ npm start
```

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



Woo! A `table` element!

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

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

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

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

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

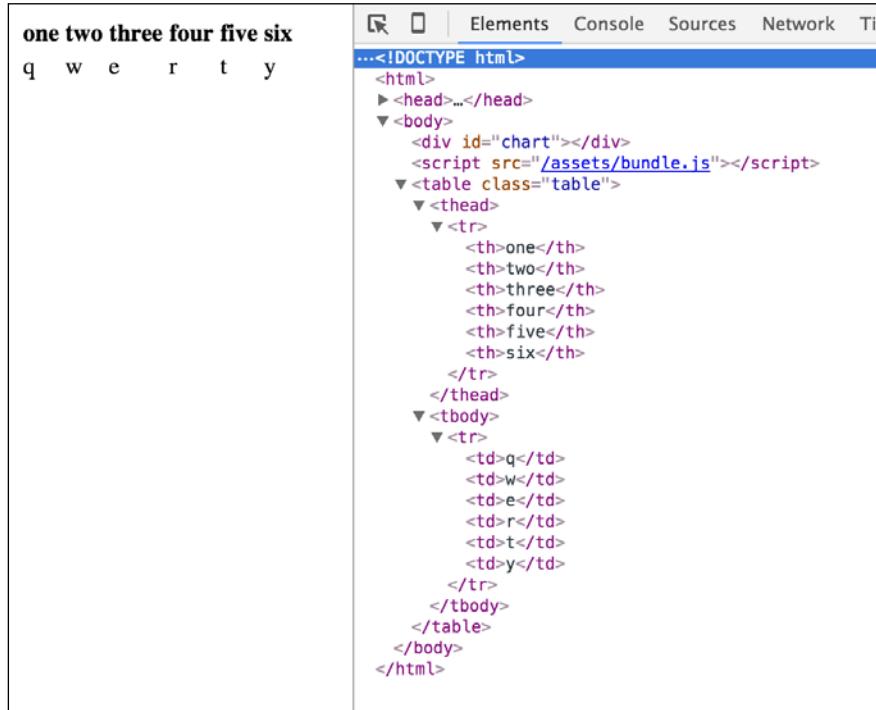
    // Each element in "header" is a string.
    this.header.forEach(function(value) {
      tableHeader.append('th').text(value);
    });

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

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

    return table;
  }
}
```

Now your table should look like this:



The screenshot shows the browser's developer tools with the "Elements" tab selected. On the left, there is a table with six columns labeled "one" through "six". On the right, the generated HTML code is displayed:

```
...<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <div id="chart"></div>
    <script src="/assets/bundle.js"></script>
    <table class="table">
      <thead>
        <tr>
          <th>one</th>
          <th>two</th>
          <th>three</th>
          <th>four</th>
          <th>five</th>
          <th>six</th>
        </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>
  </body>
</html>
```

What exactly did we do here?

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

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

Okay, time to finally play with some selections!

Selections example

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

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

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

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

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

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

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

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

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

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

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

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

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

The two `selectAll` calls gave us all the instances of `td` in the body, separated by rows, giving us an array of three arrays with six elements: [`Array[6]`, `Array[6]`, `Array[6]`]. Then we used `.style()` to change the color of every selected element.

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

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

Manipulating content

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

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

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

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

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

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

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

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

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

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

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

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



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



Joining data to selections

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

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

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

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

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

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



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



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

An HTML visualization example

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

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

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

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

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

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

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

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

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

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

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

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

```
        });
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The `.sort()` function takes only a comparator function. The comparator is given two pieces of data and must decide how to order them: `-1` for being less than `b`, `0` for being equal, and `1` for being more than `b`. You can also use the `d3ascending` and `d3descending` comparators of D3.

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

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

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

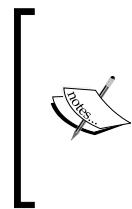
    return {
      name: d.key,
      category: earliest.Group,
      'earliest appearance': earliest.YEAR
    }
  });

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

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

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

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



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

<https://github.com/mbostock/d3/wiki/Arrays>

That's a much nicer table!

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

Scalable Vector Graphics

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

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

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

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

Drawing with SVG

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

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

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

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

let d3 = require('d3');

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

export function renderDailyShowGuestTable() {
```

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

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

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

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

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

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

Manually adding elements and shapes

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

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

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

Text

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

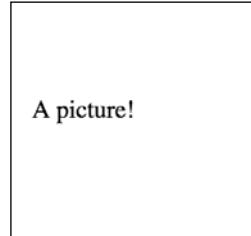
```
svg.append('text')
  .text('A picture!')
  .attr({x: 10,
         y: 150,
         'text-anchor': 'start'})
```

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

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

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

Our image looks like this:



Shapes

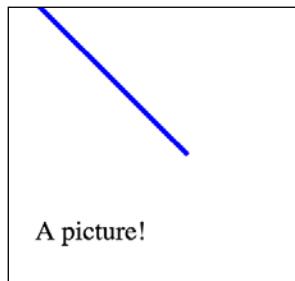
Now that `text` is out of the way, let's look at something useful—shapes, the heart of the rest of this book!

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

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

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

As before, we took the `svg` element, appended a line, and defined some attributes. A line is drawn between two points: (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.



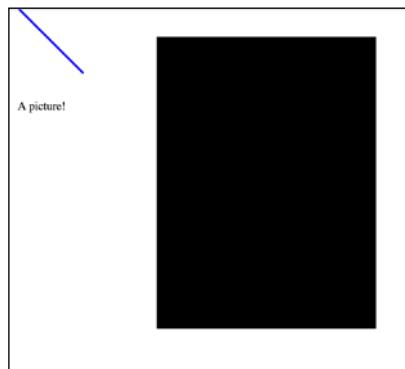
Our line points downwards even though y_2 is bigger than y_1 . That's because the origin in most image formats lies in the top-left corner. This means that $(x=0, y=0)$ defines the top-left corner of the image.

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

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

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

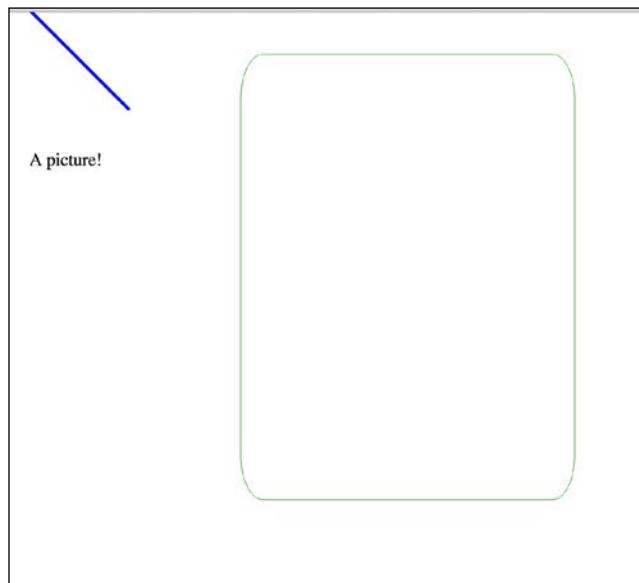
Our image now looks like this:



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

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

```



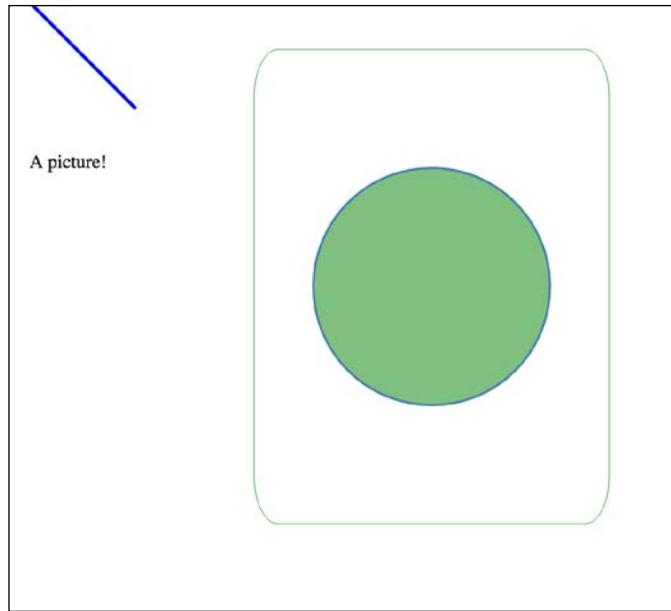
This is much better. Our rectangle has a thin green outline. Rounded corners come from the `rx` and `ry` attributes, which define the corner radius along the `x` and `y` axes.

Let's try adding a circle:

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

```

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



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

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

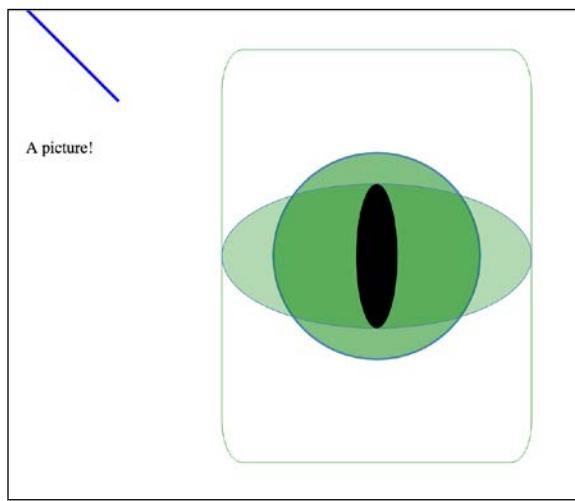
```

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

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

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

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



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

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

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

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

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

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

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

Transformations

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

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

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

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

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

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

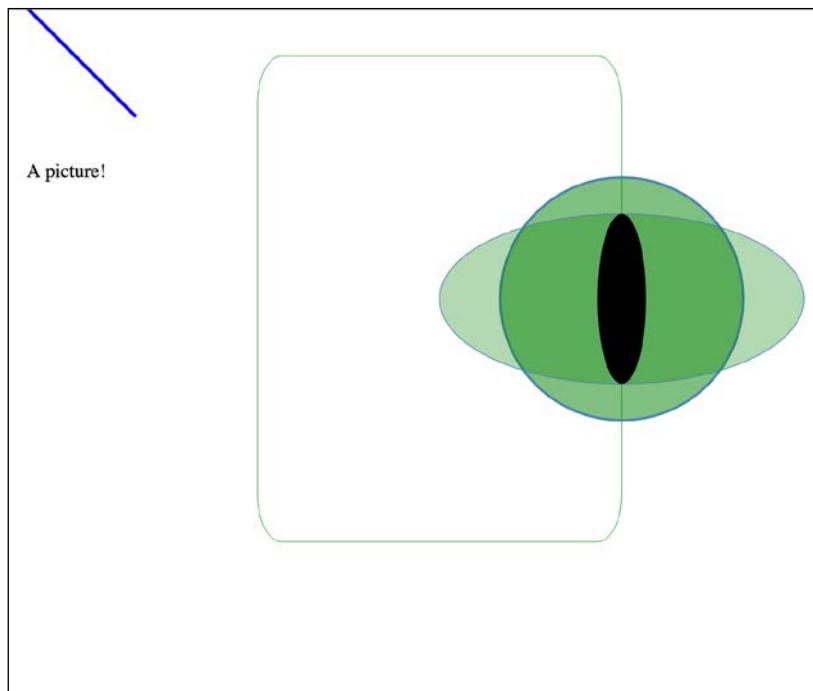
Let's move our eye to the edge of the rectangle:

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

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

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

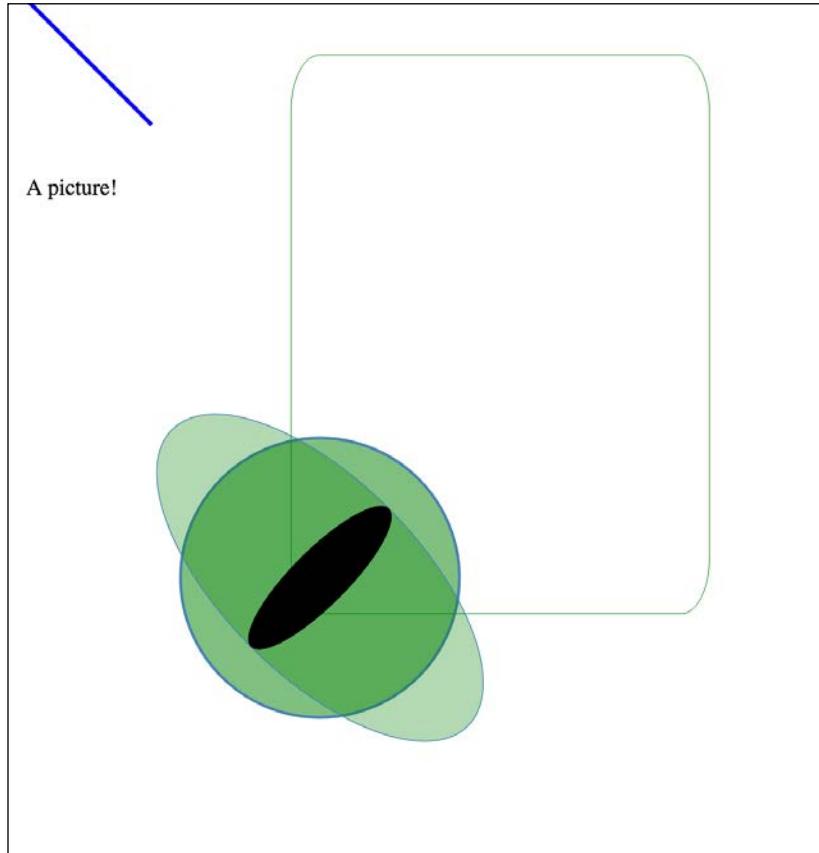
Our picture looks like what is shown here:



Let's rotate the eye by 45 degrees:

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

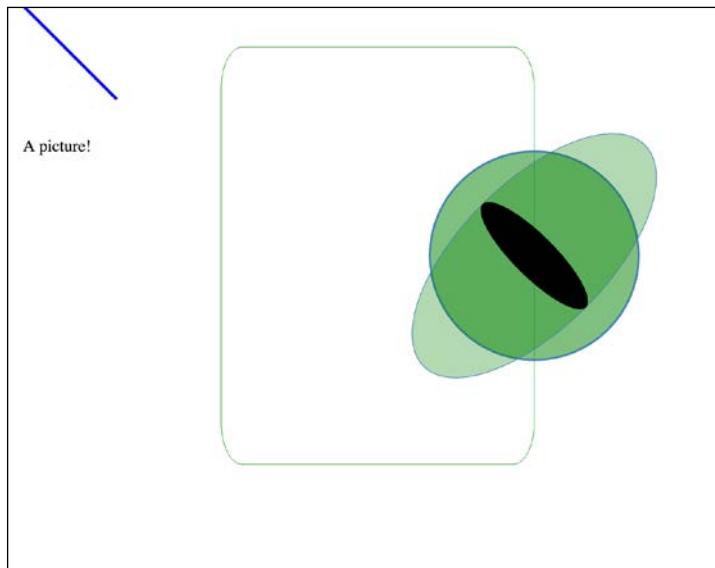
That's not what we wanted at all!



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

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

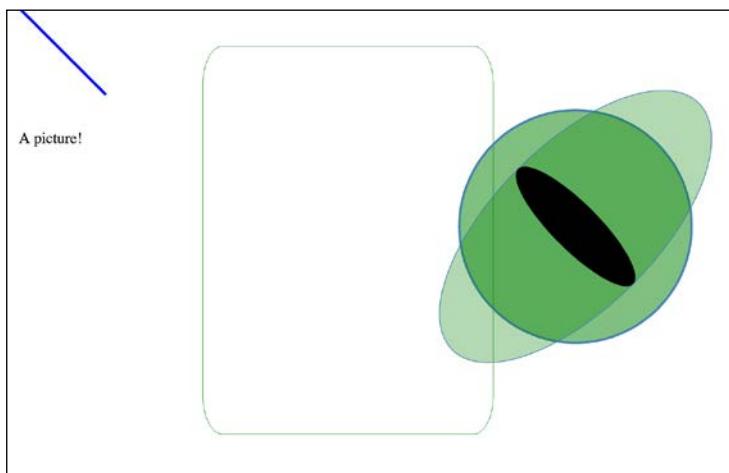
By adding two more arguments to `rotate()`, we defined the rotation axis and achieved the desired result:



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

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

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

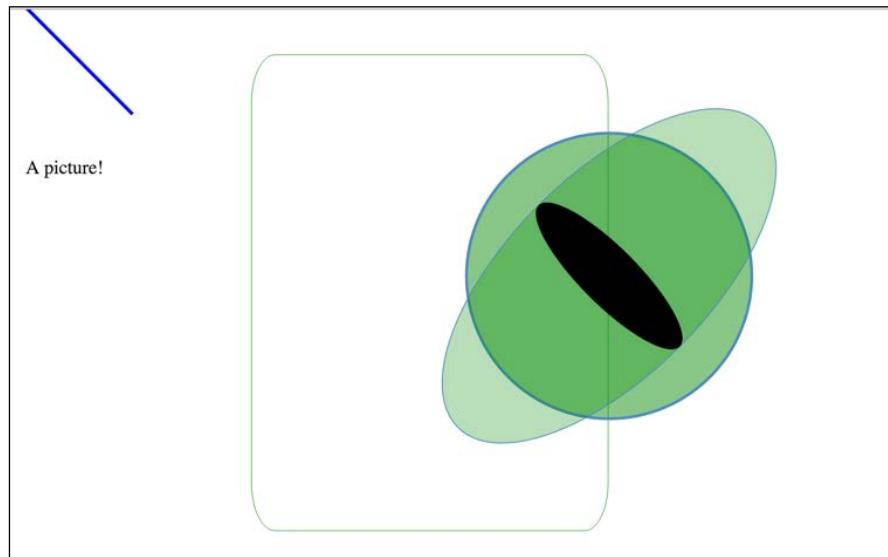


Once again, we have pushed the position of the eye because scaling is anchored at the zeroth point of the whole image. We have to use another translate operation to move it back. But the coordinate system we're working on is now rotated by 45 degrees and scaled. This makes things tricky. We need to translate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

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

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

Much better! We get exactly what we wanted:



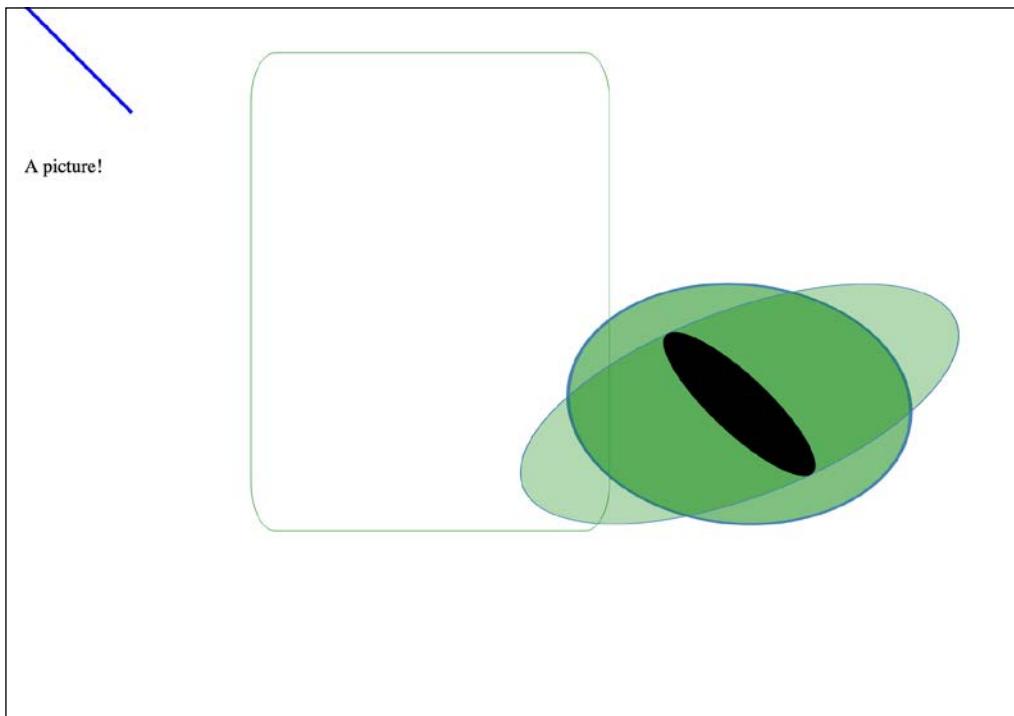
A lot has changed, so let's take a look at it.

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

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

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

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



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

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

Using paths

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

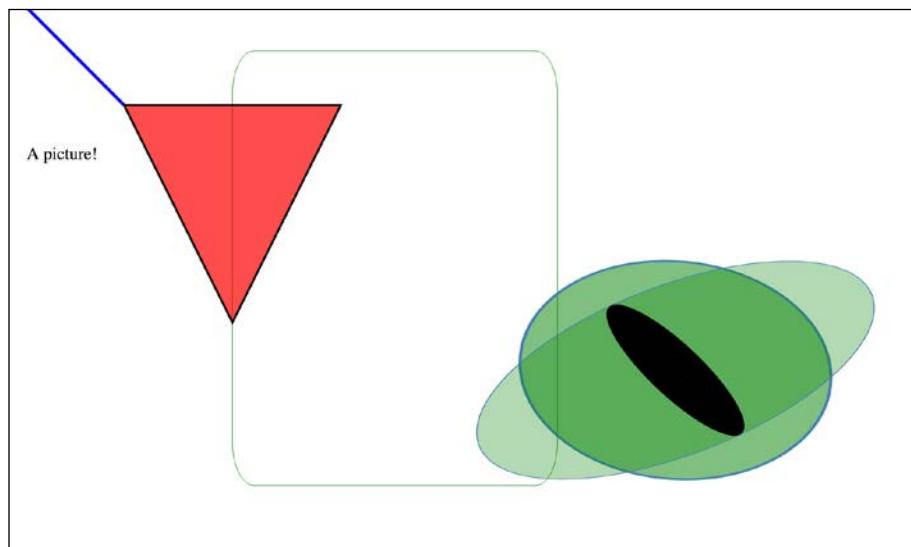
Most of the path's magic stems from the `d` attribute; it uses a mini language of three basic commands:

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

Line

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

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

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

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

Now we get to define a simple path generator:

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

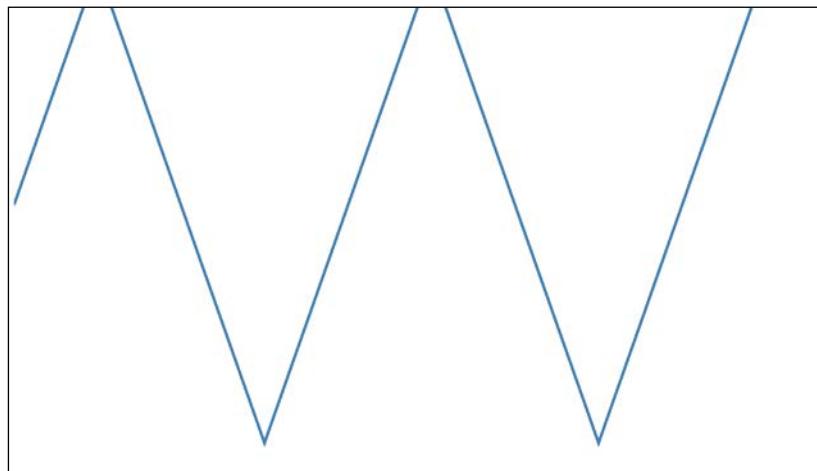
It is just a matter of taking the basic line generator and attaching some accessors to it. We told the generator to use our `x` scale on the first element and the `y` scale on the second element of every tuple. By default, it considers our dataset as a collection of arrays defining points directly so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

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

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

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

Our graph looks as follows:



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

```
d="M0,220L48.8888888888886,0L97.7777777777777,219.9999999999994L14  
6.6666666666666,440L195.555555555554,220.0000000000006L244.4444444  
4444446,0L293.33333333333,219.9999999999991L342.222222222223,440L  
391.111111111111,220.0000000000009L440,0"
```

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

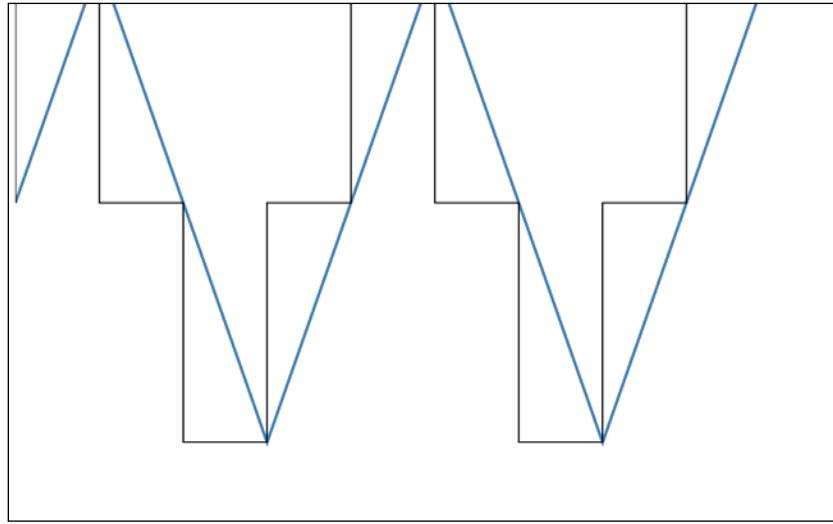
It's a very jagged `sine` function we've got here, nothing similar to what the math teacher used to draw in high school. We can make it better with interpolation.

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

Let's try something else:

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

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



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

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

Area

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

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

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

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

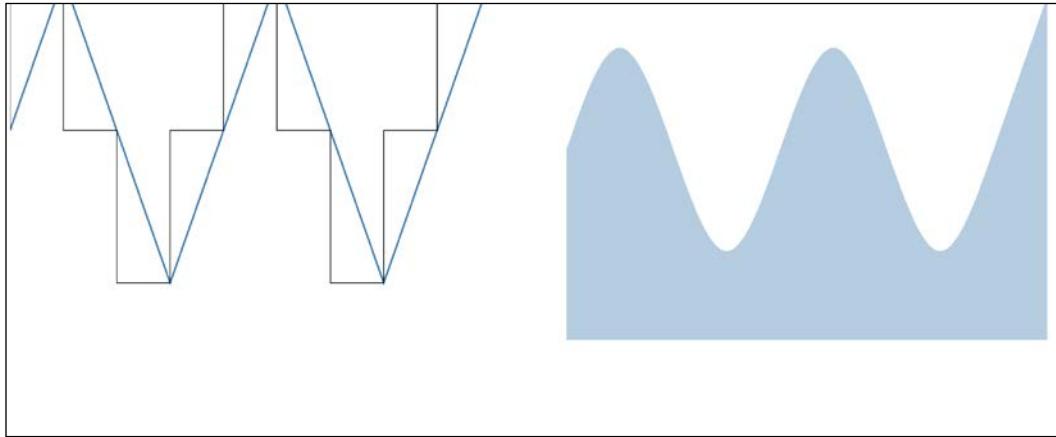
Now we define an area generator and draw an area:

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

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

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

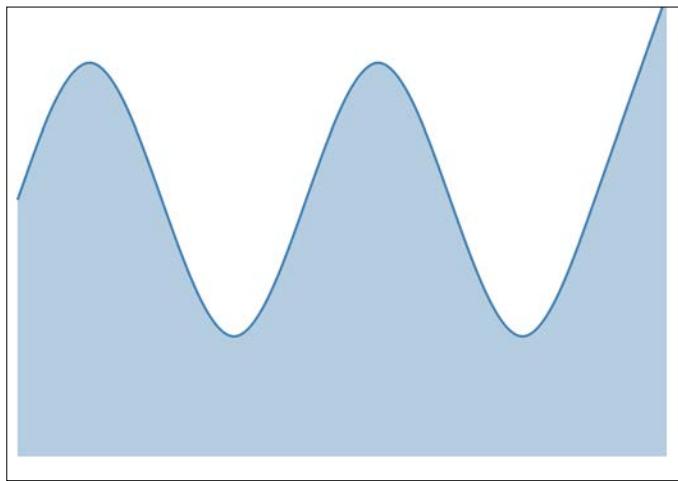
To draw the bottom edge, we defined `y0` as the bottom of our graph and produced a colored sine approximation:



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

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

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



Arc

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

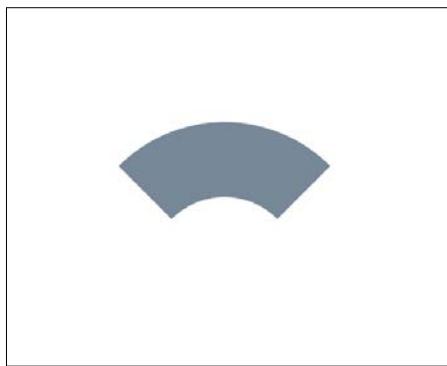
Everything works as before; we just tell the base generator how to use our data. The only difference is that this time the default accessors expect named attributes instead of two-value arrays we've gotten used to.

Let's draw an arc:

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

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

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



Huzzah, a simple arc. Rejoice!!

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

Symbol

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

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

As always, we start with a path generator:

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

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

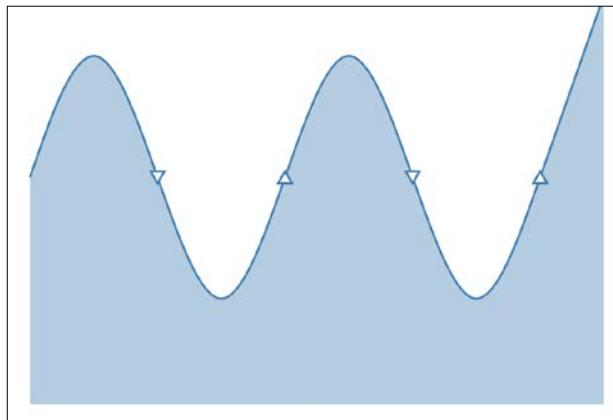
The size accessor tells `symbol()` how much area each symbol should occupy. Because every other data point is close to zero, we hide the others with an area equal to zero.

Now we can draw some symbols:

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

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

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



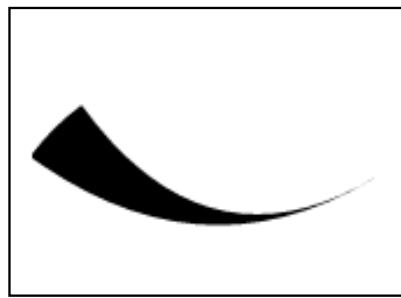
You can see other available symbols by printing `d3.svg.symbolTypes` or visiting https://github.com/mbostock/d3/wiki/SVG-Shapes#symbol_type.

Chord

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

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

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



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

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

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

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

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

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

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

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

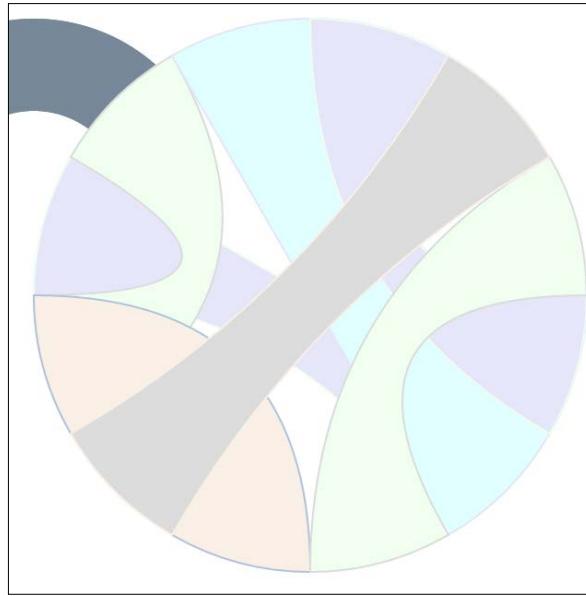
All of this just defines the generator. We're going to divide a circle into sections and connect random pairs with chords.

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

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

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

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



Diagonal

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

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

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

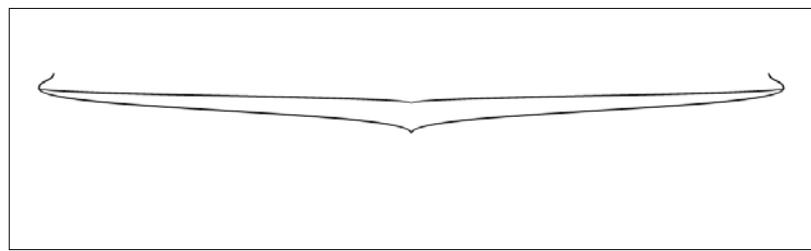
```
let g4 = svg.append('g')
  .attr('transform', `translate(${chart.width/2}, ${chart.height/2})`);

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

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

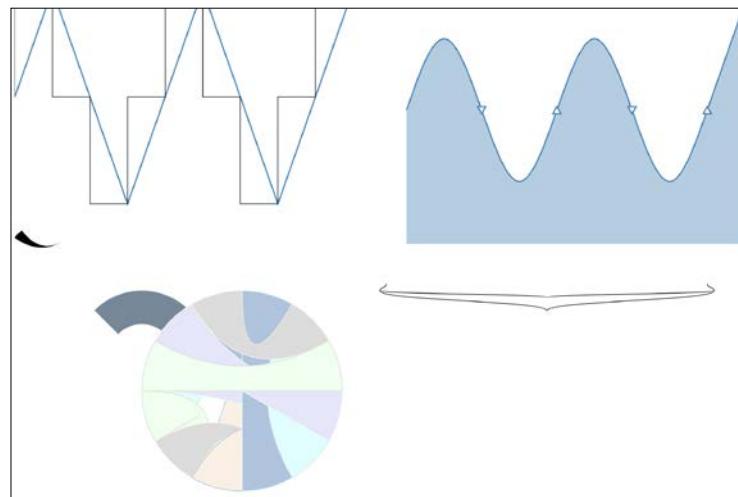
```
g4.selectAll('path')
  .data(moustache)
  .enter()
  .append('path')
  .attr('d', d3.svg.diagonal())
  .attr({stroke: 'black', fill: 'none'});
```

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



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

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



Axes

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

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



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

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

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

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

We also need a linear scale:

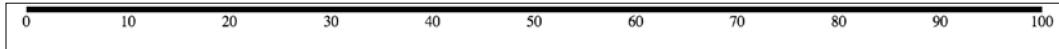
```
let x = d3.scale.linear()
  .domain([0, 100])
  .range([chart.margin.left, chart.width - chart.margin.right]);
```

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

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

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

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



The result doesn't look good at all.

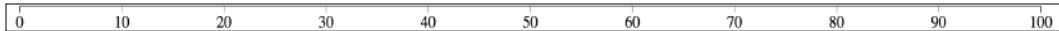
Axes are complex objects, so fixing this problem is convoluted without CSS, which comes in the next section.

For now, adding this code will be sufficient:

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

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

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



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

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

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

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

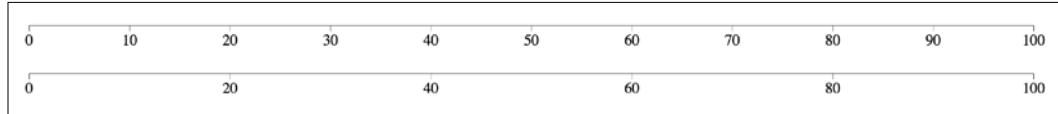
You should also change the `.attr('transform', ...)` line to put each axis 50 pixels below the previous one:

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

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

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

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

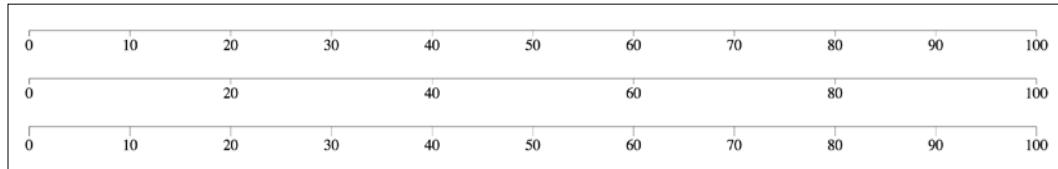


It worked! The axis generator figured out which ticks are best left off and relabeled everything without us doing much.

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

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

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

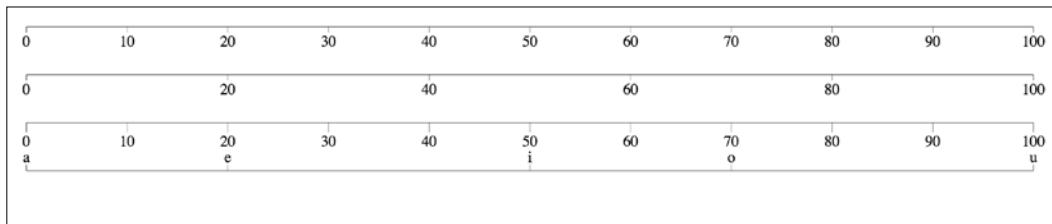


For our final trick, let's define some custom ticks and place them above the axis. We'll add another axis to the array:

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

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

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



CSS

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

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

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

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

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

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

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

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

We can invoke CSS with D3 in three ways:

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

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

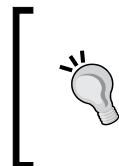
Go to `index.css` and replace it with the following code:

```
.axis path,  
.axis line {  
  fill: none;  
  stroke: black;  
  stroke-width: 1px;  
  shape-rendering: crispEdges;  
}  
  
.axis text {  
  font-size: 11px;  
}  
  
.axis.dotted line,  
.axis.dotted path {  
  stroke-dasharray: 0.9;  
}
```

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

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

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

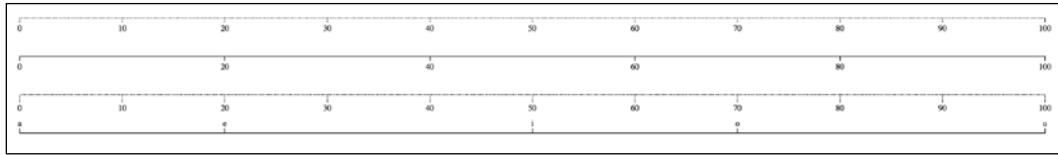


You can do more than just dotted lines with `stroke-dasharray`. You can also have dashed lines and animate the property to get a trailing or growing line effect. Visit this article on the Mozilla Developer Network for more examples:
<https://mdn.io/stroke-dasharray>

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

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

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



Colors

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Finally, we draw the two color wheels:

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

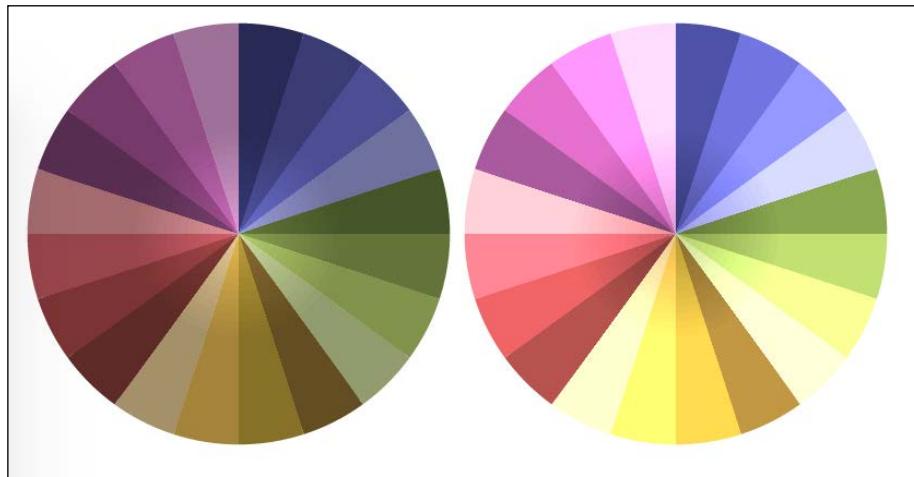
Wow! That's quite a bit of code.

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

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

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

The result looks as follows:



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

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

Summary

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

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

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

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

3

Making Data Useful

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

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

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

Thinking about data functionally

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

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

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

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

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

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

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

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

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

Built-in array functions

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

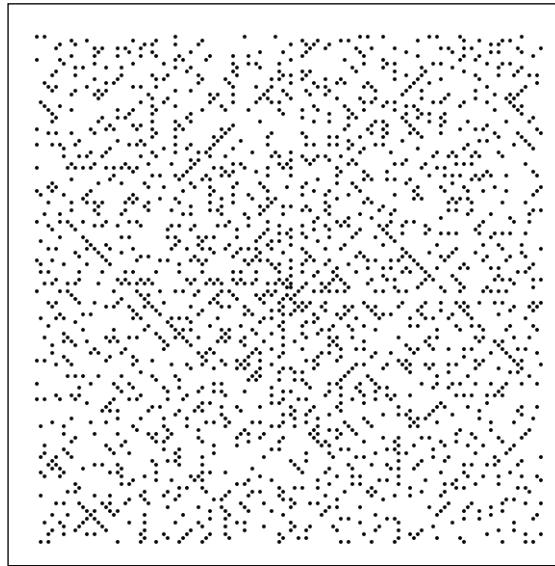
These functions are relatively new to JavaScript, whereas `.map` and `.filter` have existed since JavaScript 1.7 and `.reduce` since 1.8. In the bad old days, you'd have to use either `es6-shim` or something like `Underscore.js` to be able to use them, but since we're now in the bright shiny ES6 future,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

Data functions of D3

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

return spiral;
```

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

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

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

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

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

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

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

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

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

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

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

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

Put the following generators inside this function.

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

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

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

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

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

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

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

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

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

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

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

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

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

Suppose you start the development server via the following line:

```
$ npm start
```

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

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

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

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

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

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

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

Consider this code:

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

Change it to the following:

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

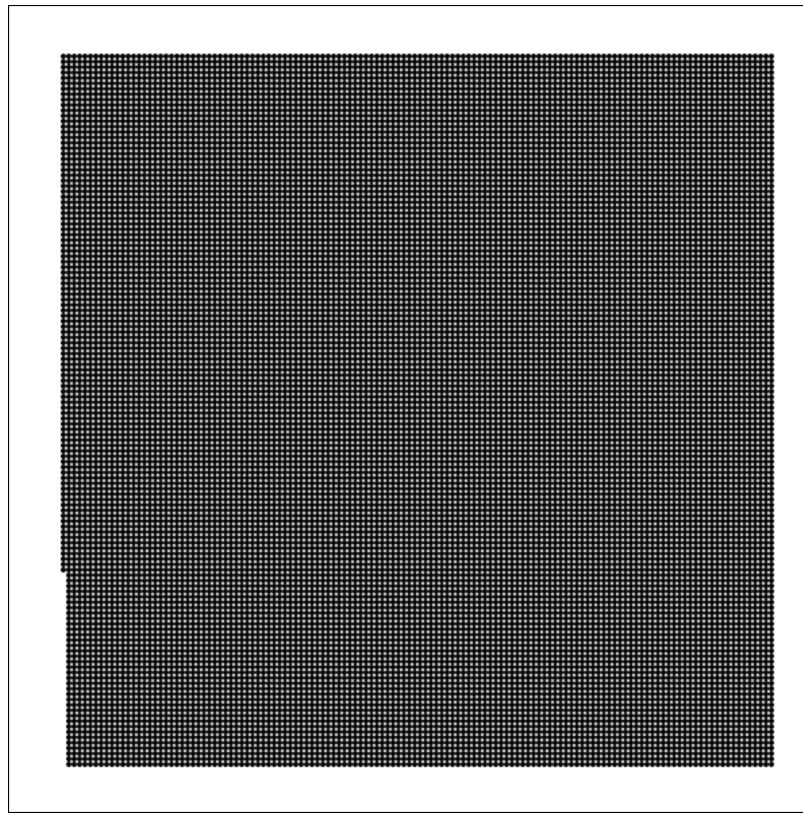
return results;
```

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

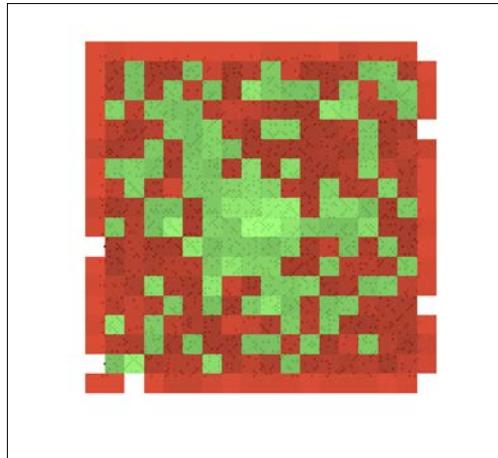
We map through the keys of our regions (*x* coordinates) to get a list of values for each column, and then use `d3.merge()` to flatten the resulting array of arrays.

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

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

```
d3.keys(regions).forEach(_x => {  
  d3.keys(regions[_x]).forEach(_y => {  
    let color,  
      red = '#e23c22',  
      green = '#497c36';  
  
    if (regions[_x][_y] > median) {  
      color = d3.rgb(green).brighter(regions[_x][_y] / shades);  
    } else {  
      color = d3.rgb(red).darker(regions[_x][_y] / shades);  
    }  
    this.chart.append('rect')  
      .attr({  
        x : x(_x, l * scale),  
        y : y(_y, l * scale),  
        width : l * scale,  
        height : l * scale  
      })  
      .style({fill : color, 'fill-opacity' : 0.9});  
  });  
});
```

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



Loading data

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

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

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

The core

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

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

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

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

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

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

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

Convenience functions

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

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

All of these follow this pattern:

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

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

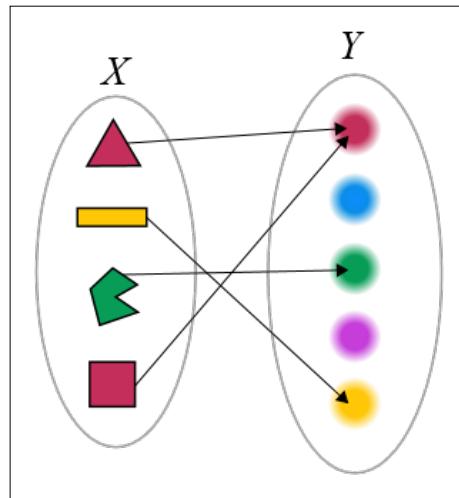
Scales

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

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

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

The following figure is borrowed from Wikipedia:



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

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

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

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

Much better!

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

Ordinal scales

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

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

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

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

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

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

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

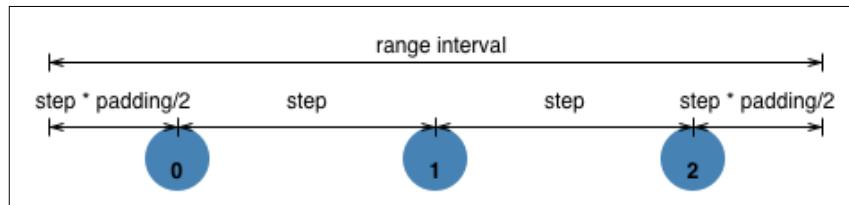
```

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

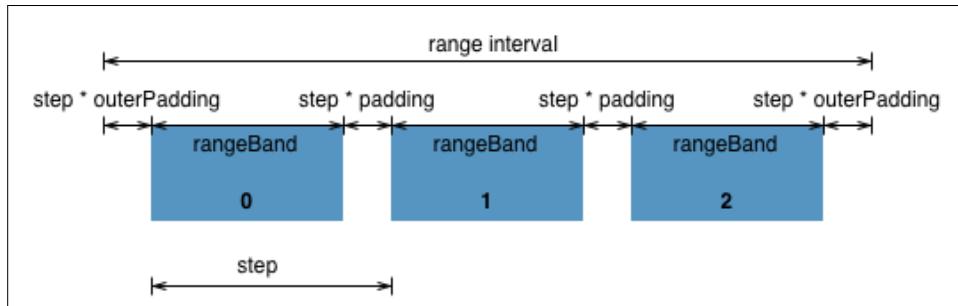
```

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

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



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



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

```

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

```

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

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

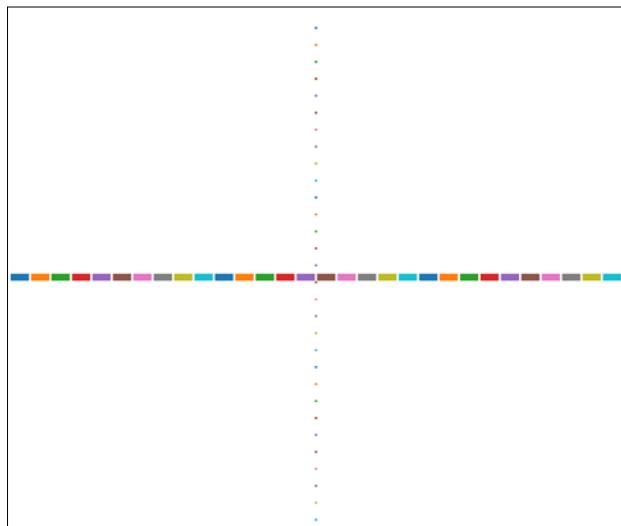
this.chart.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr({x: (d) => bands(d),
         y: this.height / 2,
         width: bands.rangeBand(),
         height: 10})
  .style('fill', (d) => colors(d));
```

Now, update index.js to resemble this:

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

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

The picture looks like this:



Quantitative scales

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

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

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

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

A drawing function will help us avoid code repetition:

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

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

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

Continuous range scales

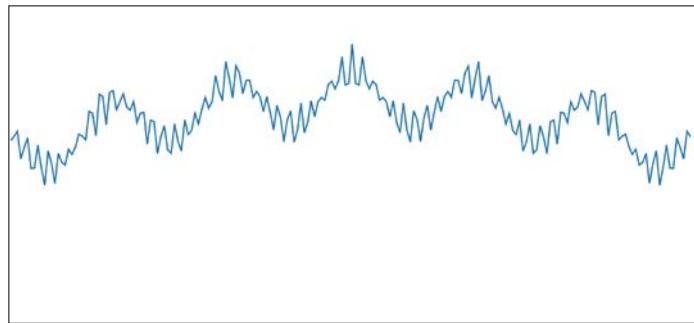
We can draw using the following code:

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

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

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

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



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

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

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

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

```

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

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

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

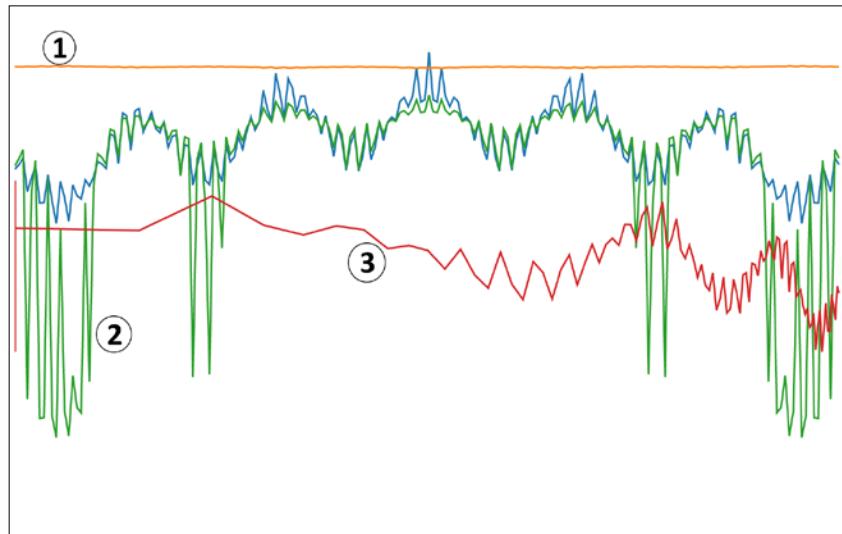
drawSingle(line4)
  .attr('transform', `translate(0, ${this.height / 4})`)
  .style('stroke', colors(3));

```

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

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

Now our picture looks as follows:



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

Discrete range scales

The scales that are interesting for our comparison are quantize and threshold.

The quantize scale cuts the input domain into equal parts and maps them to values in the output range, while the threshold scale lets us map arbitrary domain sections to discrete values:

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

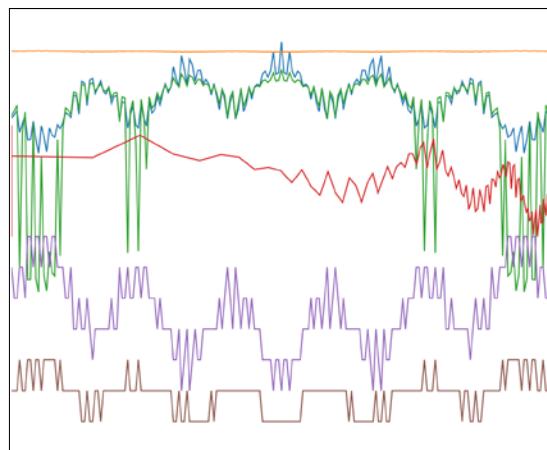
drawSingle(line5)
  .attr('transform',
    `translate(0, ${this.height / 2 + offset})`)
  .style('stroke', colors(4));

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

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

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

The result looks like this:



Time

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

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

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

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

Formatting

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

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

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

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

The same formatter works the opposite way as well:

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

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

Time arithmetic

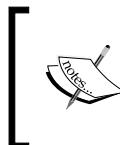
We also get a full suite of time arithmetic functions that work with JavaScript's Date objects and follow a few simple rules:

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

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

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

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



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

<http://momentjs.com>.



Geography

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

D3 gives us three tools for geographic data:

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

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

Getting geodata

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

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

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

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

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

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

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

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

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

Next, we transform the files with three simple commands:

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

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

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

Drawing geographically

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

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

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

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

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

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

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

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

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

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

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

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

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

Next, we define a geographic projection in the constructor:

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

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

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

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

Now we load our data using ES2016 promises:

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

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

```
) ;

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

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

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

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

The equivalent code in ES5 would be as follows:



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

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

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

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

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

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

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

Now we can start drawing:

```
function draw (sea, land, cultural) {  
    addToMap(sea, 'ne_50m_ocean')  
        .classed('ocean', true);  
}
```

Our `draw` function takes the error returned from loading data, and the three datasets then let `addToMap` do the heavy lifting.

We add some styling to `index.css`:

```
.ocean {  
    fill: #759dd1;  
}
```

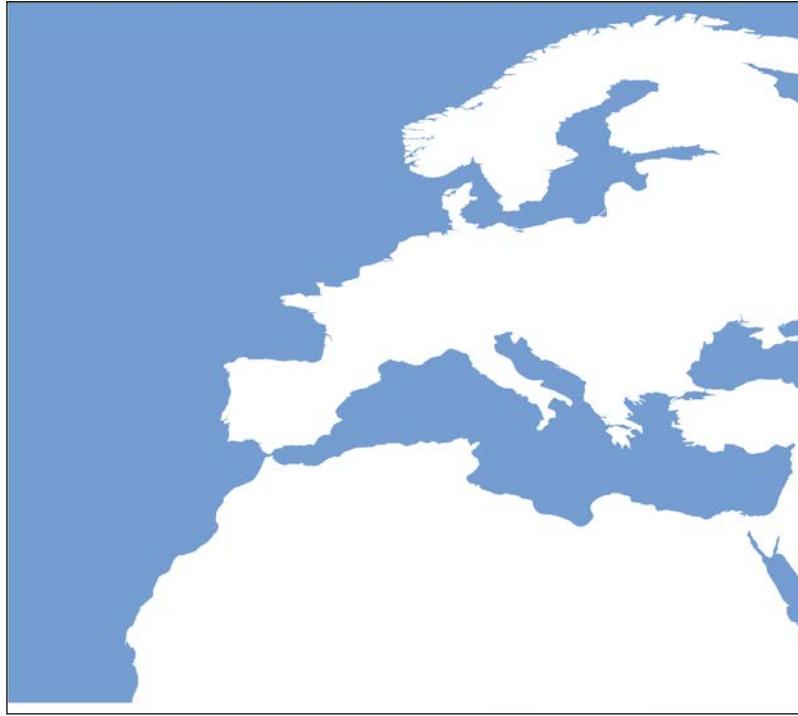
And then we require it at the top of `chapter3.js`:

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

Lastly, we call `draw` inside of our promise callback:

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

Refreshing the page, we'll be in ocean town!



We add four more `addToMap` calls to the `draw` function to fill in the other features, as follows:

```
addToMap(land, 'ne_50m_land')
    .classed('land', true);
addToMap(sea, 'ne_50m_rivers_lake_centerlines')
    .classed('river', true);
addToMap(cultural, 'ne_50m_admin_0_boundary_lines_land')
    .classed('boundary', true);
addToMap(cultural, 'ne_10m_urban_areas')
    .classed('urban', true);
```

Add some more style definitions as follows:

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

```
.land {  
    fill: #ede9c9;  
    stroke: #79bcd3;  
    stroke-width: 2;  
}  
  
.boundary {  
    stroke: #7b5228;  
    stroke-width: 1;  
    fill: none;  
}  
  
.urban {  
    fill: #e1c0a3;  
}
```

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



There are many reasons for it to be so slow. We transform between TopoJSON and GeoJSON on every call to `addToMap`. Even when using the same dataset, we're using data that's too detailed for such a zoomed-out map, and we render the whole world to look at a tiny part. We have traded flexibility for rendering speed.

Using geography as a base

Geography isn't just about drawing maps. A map is usually a base that we build to show some data.

Let's turn this into a map of the world's airports. Actually, scratch that! Let's do something cooler. Let's make a map of CIA rendition flights out of the U.S. To do this, we'll still need the world's airports, as the airport values in the Rendition Project's dataset use the airport short codes, not latitude and longitude.

The first step is fetching the `airports.dat` dataset from <http://openflights.org/data.html> and the Rendition Project's U.S. flights data from <http://www.therenditionproject.org.uk/pdf/XLS%201%20-%20Flight%20data.%20US%20FOI%20resp.xls>. You can also find it in the examples on GitHub at <https://github.com/aendrew/learning-d3/blob/chapter3/src/data/airports.dat> and <https://github.com/aendrew/learning-d3/blob/chapter3/src/data/renditions.csv>, respectively. For the renditions dataset, you'll need to open in Excel and save as CSV. I've done that for you if you are going to grab it from GitHub.

First add two new promises, and update the `Promise.all()` call to complete once the two new datasets are available. Add a call to `addRenditions()` after `draw()`:

```
let p4 = new Promise((resolve, reject) => {
  d3.text('data/airports.dat', (err, data) => {
    err ? reject(err) : resolve(data);
  });
});

let p5 = new Promise((resolve, reject) => {
  d3.csv('data/renditions.csv', (err, data) => {
    err ? reject(err) : resolve(data);
  });
});

Promise.all([p1, p2, p3, p4, p5]).then((values) => {
  let [sea, land, cultural, airports, renditions] = values;
  draw(sea, land, cultural);

  addRenditions(airports, renditions);
});
```

The function loads the two datasets and then calls (the yet-nonexistent) `addRenditions` to draw them. We use `d3.text` instead of `d3.csv` for `airports.dat` because it doesn't have a header line, so we have to parse it manually.

In `addRenditions`, we first wrangle the data into JavaScript objects—airports into a dictionary by ID—and use that to get the latitude and longitude of each destination and arrival airport:

```
function addRenditions(_airports, renditions) {
  let airports = {},
    routes;

  d3.csv.parseRows(_airports).forEach(function (airport) {
    var id = airport[4];
    airports[id] = {
      lat: airport[6],
      lon: airport[7]
    };
  });
}

routes = renditions.map((v) => {
  let dep = v['Departure Airport'];
  let arr = v['Arrival Airport'];
  return {
    from: airports[dep],
    to: airports[arr]
  };
}).filter((v) => v.to && v.from).slice(0, 50);
}
```

We used `d3.csv.parseRows` to parse CSV files into arrays and manually turned them into dictionaries. The array indices aren't very legible, unfortunately, but they make sense when you look at the raw data:

```
1,"Goroka","Goroka","Papua New Guinea","GKA","AYGA",
-6.081689,145.391881,5282,10,"U"
2,"Madang","Madang","Papua New Guinea","MAG","AYMD",
-5.207083,145.7887,20,10,"U"
```

We then map each rendition flight so that we just have a dictionary of arrival and departure coordinates. We filter out any results where either `to` or `from` is missing, which are likely cases when our map function isn't able to match the airport short codes. Also, because it's a really big dataset and drawing all of it looks a bit messy, we've limited it to the first 50 objects in the array using `Array.prototype.slice`.

Making Data Useful

Next, we'll actually draw the lines, using our projection to translate the latitude and longitude coordinates into something that can fit on our screen:

```
let lines = chart.selectAll('.route')
  .data(routes)
  .enter()
    .append('line')
    .attr('x1', (d) => projection([d.from.lon, d.from.lat])[0])
    .attr('y1', (d) => projection([d.from.lon, d.from.lat])[1])
    .attr('x2', (d) => projection([d.to.lon, d.to.lat])[0])
    .attr('y2', (d) => projection([d.to.lon, d.to.lat])[1])
    .classed('route', true);
```

The routes won't show up until we style them. Add the following code to `index.css`:

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

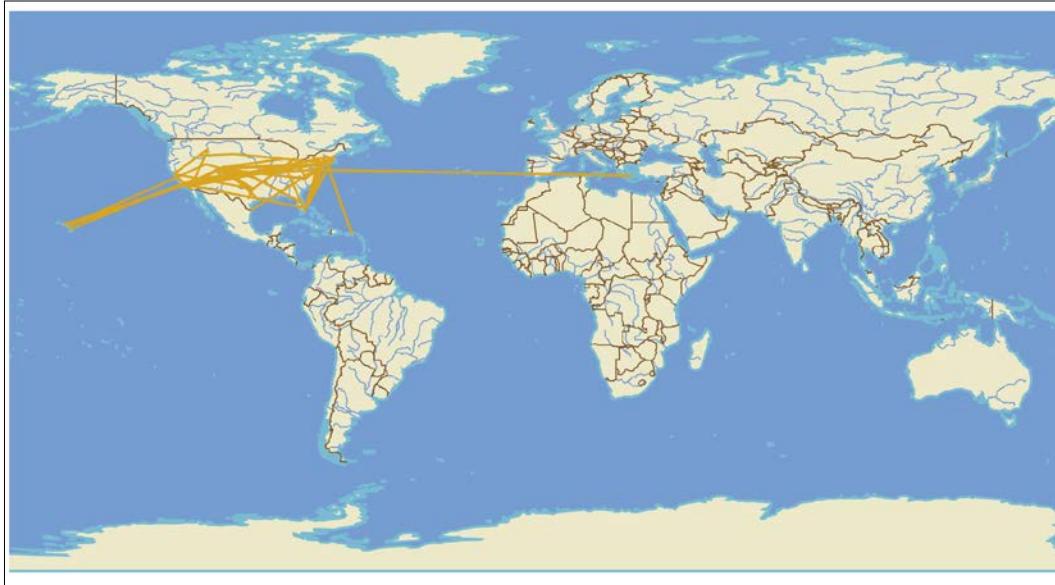
This screenshot displays the result:



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

```
let projection = d3.geo.equirectangular()  
.center([8, 56])  
.scale(200);
```

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



We solved what is commonly referred to as the "too many markers" problem—that is, when zoomed out, data on a map looks cluttered—by simply limiting the amount of data that can be shown. This is admittedly a pretty cheap way out; a better workaround is to either cluster the map data (which wouldn't be all that easy with lines like the ones we have) or provide UI elements to toggle aspects of the dataset. We'll look at interactivity in the coming chapters; hold on to your hats!



Summary

You've made it through the chapter on data!

We really got to the core of what D3 is about—that is, data wrangling. While learning about data wrangling, you saw some interesting properties of prime numbers, learned all about loading external data, and effectively used scales to avoid calculations. We played with promises and generators along the way.

Lastly, we made a cool map to learn how simple geographic data can be once you get a hand on a good source and transform it into a better format.

4

Defining the User Experience – Animation and Interaction

Animation is like chilli sauce. A little goes a really long way and can really help to spice up a graphic while leading the viewer through the content; if there is too much, it's all anyone will notice. Good UX – short for user experience, which is one of the computer idioms you employ throughout your projects – is more like guacamole. If it's good, it's a nice subtle touch which will improve the overall quality of your output and make everyone happy; if it is bad, it will taint everything and ruin the whole burrito.

In this chapter, we'll discuss both animation and user interaction, with an eye towards using both to improve the quality of your data visualizations. We'll also use D3's behaviors to make that map from the last chapter look awesome. Throughout the chapter, we'll discuss why or why not animation or interactivity should be used in a particular scenario.

The ability to display data creatively with D3 is one of the best reasons for using it; interaction and animation allow you to not only display data but also *explain* data. How you use UX throughout your interface design determines whether you are building an *exploratory* graphic, wherein the user is given access to all of the data and has the ability to change how it's displayed through sorting, filtering and so on, or an *explanatory* graphic, where minimal interactivity guides the user through the relevant data. In reality, you'll probably mix both approaches, but understanding which type of interaction you want to have with the reader at what point is helpful when planning your projects.

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

Animation

The first question to ask is, why would animation improve this project?

If you're making something that isn't really designed to communicate data and is just designed to trip people out at your local warehouse rave, then "because it would make it look cool" is a totally valid response. Please don't let me discourage you from running rainbow color interpolators through that spiral in the last chapter if you think it'd be fun (because, speaking from personal experience, creating crazy animated art with D3 is a highly enjoyable use of a Saturday afternoon).

If, however, you're rendering data, a bit more consideration is probably necessary. What is your data doing? If it's a value increasing over time, animating a line going upwards from left-to-right makes more sense than fading in the line all at once.

Previously, we set attributes on our various SVG objects as we wanted them to appear once the image was finally rendered. Now, we'll use animation to guide viewers through our graphic, using the narrative focus it provides as a way of helping them interpret the data we're displaying. To do this, we need to animate the relevant properties of each SVG object.

Animation with transitions

D3 transitions are one way of accomplishing this. Transitions use the familiar principle of changing a selection's attributes, except that changes are applied over time.

To slowly turn a rectangle red, we use the following line of code:

```
d3.select('rect').transition().style('fill', 'red');
```

We start a new transition with `.transition()` and then define the final state of each animated attribute. By default, every transition takes 250 milliseconds; you can change the timing with `.duration()`. New transitions are executed on all properties simultaneously unless you set a delay using `.delay()`.

Delays are handy when we want to make transitions happen in sequence. Without a delay, they are all executed at the same time, depending on the internal timer. For single objects, nested transitions are much simpler than carefully calibrated delays.

We've already used these way back in *Chapter 1, Getting Started with D3, ES2016, and Node.js* – remember how our bar chart bars grew to accommodate the data? I've reproduced the relevant section from the following `BasicBarChart.js`:

```
this.chart.selectAll('rect')
  .data(data)
  .enter()
```

```

.append('rect')
.attr('class', 'bar')
.attr('x', (d) => x(d.name))
.attr('width', x.rangeBand())
.attr('y', () => y(this.margin.bottom))
.attr('height', 0)
.transition()
.delay((d, i) => i*200)
.duration(800)
.attr('y', (d) => y(d.population))
.attr('height', (d) => this.height - y(d.population))

```

We initialize the transition, set each datum's delay to be 200ms later than the last and then run the 800ms transition, increasing the y value and bar height to the bar's value.

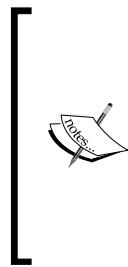
If you want to do something before a transition begins or want to listen for it to end, you can use `.each()` with the appropriate event type. Add the following to the preceding code:

```

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

```

This is handy when making instant changes before or after a transition. Just keep in mind that transitions run independently and you cannot rely on transitions outside the current callback being in any particular state.



Whoops, we've actually done something silly with our animation here. Even though it was a nifty demonstration of how to stagger animations using `.delay()`, you generally shouldn't do this with ordinal scale charts. Why? Staggering gives the impression that the `x` axis is changing over time — in fact, we're not charting time series data at all in this so it makes more sense for all the bars to rise at the same time. It's really subtle but these are the sorts of things worth considering as you begin to use animation.

Interpolators

To calculate values between the initial and final states of a transition, D3 uses a type of function called an interpolator, which maps the $[0, 1]$ domain to a target range, which can be a color, a number, or a string. These make it easy to blend between two values, because the interpolator will return the iterations between the values supplied to it. Under the hood, scales are based on these same interpolators.

D3's built-in interpolators can interpolate between almost any two arbitrary values, most often between numbers or colors, but also between strings. This sounds odd at first but it's actually pretty useful. To let D3 pick the right interpolator for the job, we just write `d3.interpolate(a, b)` and the interpolation function is chosen depending on the type of `b`. `a` is the initial value, and `b` is the final value.

If `b` is a number, `a` will be coerced into a number and `.interpolateNumber()` will be used. You should avoid interpolating to or from a zero value because values will eventually be transformed into a string for the actual attribute and very small numbers might turn into scientific notation. CSS and HTML don't quite understand `1e-7` (the digit 1 with seven zeroes before the decimal place), so the smallest number you can safely use is `1e-6`.

If `b` is a string, D3 checks whether it's a CSS color, in which case it is transformed to a proper color, just like the ones in *Chapter 2, A Primer on DOM, SVG, and CSS*. `a` is transformed into a color as well and then D3 uses `.interpolateRgb()` or a more appropriate interpolator for your color space.

Something even more amazing happens when the string is not a color. D3 can handle that too! When it encounters a string, D3 parses it for numbers, then uses `.interpolateNumber()` on each numerical piece of the string. This is useful for interpolating mixed style definitions.

For instance, to transition a font definition, you might do something like this:

```
d3.select('svg')
.append('text')
.attr({x: 100, y: 100})
.text("I'm growing!")
.transition()
.styleTween('font', () =>
  d3.interpolate('12px Helvetica', '36px Comic Sans MS'));
```

We used `.styleTween()` to manually define a transition. It is most useful when we want to define the starting value of a transition without relying on the current state. The first argument defines which style attribute to transition and the second is the interpolator.

You can use `.tween()` to do this for attributes other than the style.

Every numerical part of the string was interpolated between the starting and ending values and the string parts changed to their final state immediately. An interesting application of this is interpolating path definitions—you can make shapes change in time. How cool is that?

Keep in mind that only strings with the same number and location of control points (numbers in the string) can be interpolated. You can't use interpolators for everything. Creating a custom interpolator is as simple as defining a function that takes a single t parameter and returns the start value for $t = 0$ and the end value for $t = 1$ and blends values for anything in between.

For example, the following code shows the `interpolateNumber` function of D3:

```
function interpolateNumber(a, b) {
  return function(t) {
    return a + t * (b - a);
  };
}
```

It's as simple as that!

You can even interpolate whole arrays and objects, which work like compound interpolators of multiple values. We'll use those soon.

Easing

Easing tweaks the behavior of interpolators by controlling the time (t) argument. We use this to make our animations feel more natural, to add some bounce elasticity, and so on. Mostly, we use easing to avoid the artificial feel of linear animation.

Let's make a quick comparison of the easing functions provided by D3 and see what they do.

First create the file `chapter4.js` and a new class that extends `BasicChart`. You know the drill.

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

export class chapter4 extends BasicChart {
  constructor(data) {
    super(data);

  }
}
```

Next, we need an array of easing functions and a scale to place them along the vertical axis. Put this in the constructor under `super(data)`:

```
let eases = ['linear', 'poly(4)', 'quad', 'cubic', 'sin', 'exp',
  'circle', 'elastic(10, -5)', 'back(0.5)', 'bounce', 'cubic-in',
  'cubic-out', 'cubic-in-out', 'cubic-out-in'],
y = d3.scale.ordinal().domain(eases).rangeBands([50, 500]);
```

You'll notice that `poly`, `elastic`, and `back` take arguments since these are just strings so we'll have to change them into real arguments manually later. The `poly` easing function is just a polynomial, so `poly(2)` is equal to `quad` and `poly(3)` is equal to `cubic`. Or, for those of us who stopped paying attention towards the end of our secondary school math, the higher the `poly` argument value, the deeper the curve — for instance, `poly(4)` (equivalent to `quart`) has a fair bit of delay at the beginning, the end or both, depending on where you set the easing (see below). The higher the number, the more dramatic the delay. Have a play with it, do what feels right.

The `elastic` easing function simulates a rubber band and the two arguments control tension. You should play with the values to get the effect you want. The `back` easing function is supposed to simulate backing into a parking space. The argument controls how much overshoot there's going to be.

The easings at the end (`cubic-in`, `cubic-out`, and so on) are functions that we create ourselves by combining the following modifiers:

- `-in`: It does nothing
- `-out`: It reverses the easing direction
- `-in-out`: It copies and mirrors the easing function from `[0, 0.5]` and `[0.5, 1]`
- `-out-in`: It copies and mirrors the easing function from `[1, 0.5]` and `[0.5, 0]`

You can add these to any easing function so play around. Now, we're going to render a bunch of circles animated using each easing:

```
eases.forEach((ease) => {
  let transition = svg.append('circle')
    .attr({cx: 130, cy: y(ease), r: y.rangeBand()/2-5})
    .transition()
    .delay(400)
    .duration(1500)
    .attr({cx: 400});
});
```

We loop over the list with an iterator that creates a new circle and uses the `y()` scale for vertical placement and `y.rangeBand()` for the circle size. In this way, we can easily add or remove examples. Transitions start with a delay of just under half a second to give us a chance to see what's going on. A duration of 1500 milliseconds and a final position of 400 should give us enough time and space to see the easing.

We define the easing at the end of this function, before the `) ;` section, as shown:

```
if (ease.indexOf('(') > -1) {
  let args = ease.match(/\d+/g),
    type = ease.match(/\w+/);
  transition.ease(type, args[0], args[1]);
} else {
  transition.ease(ease);
}
```

This code checks for parentheses in the `ease` string, parses out the easing function and its arguments, and feeds them to `transition.ease()`. Without parentheses, `ease` is just the easing type.

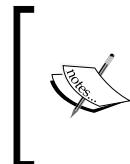
Let's add some text so that we can tell the examples apart:

```
svg.append('text')
  .text(ease)
  .attr({x: 10, y: y(ease)+5});
```

Replace `index.js` with the following:

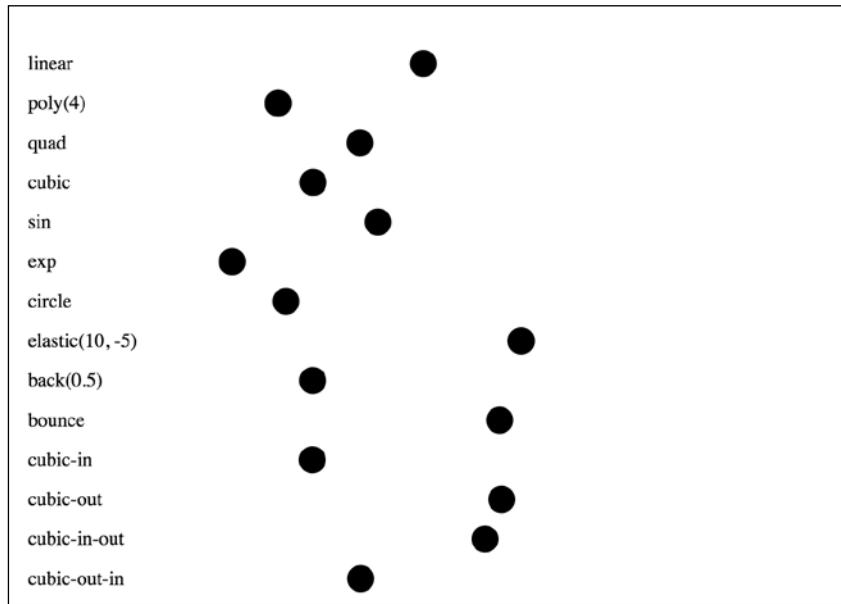
```
import {chapter4} from './chapter4';
new chapter4();
```

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



In order to avoid repeating the preceding four lines over and over again, I'll take it on faith that you know how to import and instantiate the classes we're creating at this point, in addition to knowing how to load the developer server. I'll therefore leave out that part from here on in.

The visualization is a cacophony of dots:



The screenshot doesn't quite showcase the animation so you should try this one in the browser. You could also take a look at the easing curves at <http://easings.net/>.

Easings are a nice finishing touch to put on most animations. Most things in the real world don't have constant acceleration; a good rule of thumb is to match whichever element you're animating with an easing appropriate for its size in relation to the page. In other words, small elements should generally move faster than large elements and have tighter -in and -out easings. The key thing is to think about how stuff should logically move instead of just slapping a 1-second fade-in on everything.

Timers

D3 uses timers to schedule transitions. Even an immediate transition will start after a delay of 17ms.

Far from keeping timers all to itself, D3 lets us use timers so that we can take animation beyond the two-keyframe model of transition. If you are not familiar with animation terminology, keyframes define the start or end of a smooth transition.

We use `d3.timer()` to create a timer. It takes a function, a delay, and a starting mark. After the set delay (in milliseconds) from the mark, the function will be executed repeatedly until it returns `true`. The mark should be a date converted into milliseconds since Unix timestamp (`Date.getTime()` is sufficient), or you can let D3 use `Date.now()` by default.

Let's animate the drawing of a parametric function to work just like the Spirograph toy you might have had as a kid.

We'll create a timer, let it run for a few seconds, and use the millisecond mark as the parameter for a parametric function.

Create a new class in `chapter4.js`:

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

Here's a good function from Wikipedia's article on parametric equations at http://en.wikipedia.org/wiki/Parametric_equations:

```
let position = (t) => {
  let a = 80, b = 1, c = 1, d = 80;
  return {x: Math.cos(a*t) - Math.pow(Math.cos(b*t), 3),
          y: Math.sin(c*t) - Math.pow(Math.sin(d*t), 3)};
};
```

This function returns a mathematical position based on the parameter, going from zero up. You can tweak the Spirograph by changing the `a`, `b`, `c`, and `d` variables – there are examples in the same Wikipedia article.

This function returns positions between -2 and 2, so we need scales to make it visible on the screen:

```
let tScale = d3.scale.linear().domain([500, 25000])
  .range([0, 2*Math.PI]),
  x = d3.scale.linear().domain([-2, 2]).range([100, this.width-100]),
  y = d3.scale.linear().domain([-2, 2]).range([this.height-100,100]);
```

`tScale` translates time into parameters for the function; `x` and `y` calculate the final position on the image.

Now we need to define brush to fly around and pretend that it's drawing and also a variable to hold the previous position so that we can draw straight lines:

```
let brush = chart.append('circle')
    .attr({r: 4}),
    previous = position(0);
```

Next, we need to define an animation step function to move the brush and draw a line between the previous and current points:

```
let step = (time) => {
  if (time > tScale.domain()[1]) {
    return true;
  }

  let t = tScale(time),
    pos = position(t);

  brush.attr({cx: x(pos.x), cy: y(pos.y)});

  this.chart.append('line')
    .attr({x1: x(previous.x),
           y1: y(previous.y),
           x2: x(pos.x),
           y2: y(pos.y),
           stroke: 'steelblue',
           'stroke-width': 1.3});

  previous = pos;
};
```

The first condition stops the timer when the current value of the time parameter is beyond the domain of tScale. Then, we use tScale() to translate the time into our parameter and get a new position for the brush.

Then, we move the brush — there is no transition because we are performing the transition ourselves already — and draw a new steel blue line between the previous position and the current position (pos).

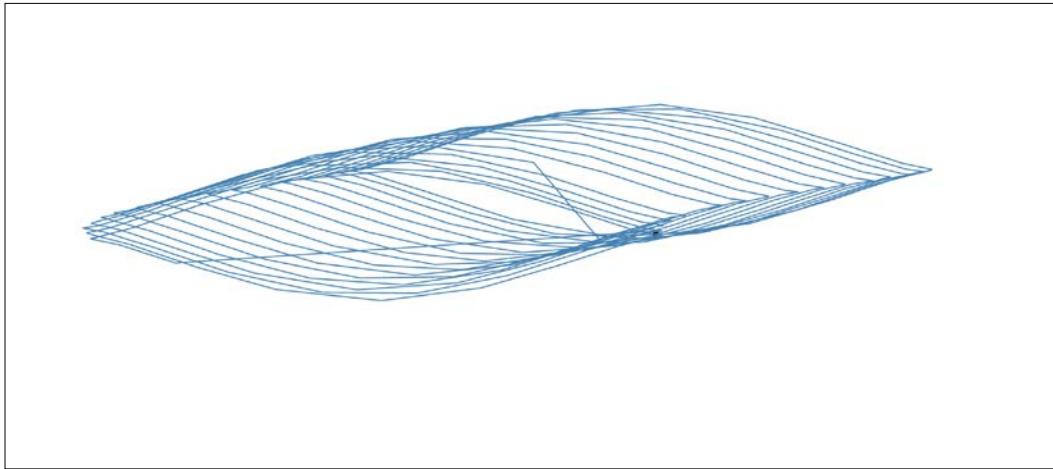
We conclude by setting a new value for the previous position.

All that's left now is to create a timer:

```
let timer = d3.timer(step, 500);
```

That's it. Half a second after a page refresh, the code will begin drawing a beautiful shape and finish 25 seconds later.

Starting out, it looks like this:



Getting the whole picture takes a while so this isn't the best way to draw Spirographs. Since we're using time as a parameter, a smoother curve (with more points) takes more time.

Animation with CSS transitions

Another way of animating things with D3 is by using CSS transitions. If you simply wish to animate a `transform` on an element (particularly if you don't need a lot of control over sequencing), CSS transitions are easier. They have the added benefit of being much better for performance due to not forcing the browser to repaint on every tick, which means that the GPU does all the work and everything runs much smoother. Lastly, you can use CSS media queries to target how an animation works on different devices.

Let's make a basic bar chart and use CSS transitions to do the heavy lifting.

Create a new class in `chapter4.js`, as shown here:

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

And fill it with the following basic bar chart code, It seems like a lot but is actually really similar to what we used way back in chapter1.js:

```
export class PrisonPopulationChart extends BasicChart {
  constructor(path) {
    super();

    this.margin.left = 50;
    let d3 = require('d3');

    let p = new Promise((res, rej) => {
      d3.csv(path, (err, data) => err ? rej(err) : res(data));
    });

    require('./index.css');

    this.x = d3.scale.ordinal().rangeBands([this.margin.left,
this.width], 0.1);

    p.then((data) => {
      this.data = data;
      this.drawChart();
    });
  }

  return p;
}

drawChart() {
  let data = this.data;
  data = data.filter((d) => d.year >= d3.min(data, (d) =>
d.year) && d.year <= d3.max(data, (d) => d.year));

  this.y = d3.scale.linear().range([this.height,
this.margin.bottom]);
  this.x.domain(data.map((d) => d.year));
  this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

  this.xAxis =
d3.svg.axis().scale(this.x).orient('bottom').
tickValues(this.x.domain().filter((d, i) => !(i % 5)));
  this.yAxis = d3.svg.axis().scale(this.y).orient('left');

  this.chart.append('g')
  .classed('axis x', true)
  .attr('transform', `translate(0, ${this.height})`)
}
```

```
.call(this.xAxis);

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

this.bars = this.chart.append('g').classed('bars',
true).selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .style('x', (d) => {
    return this.x(d.year);
  })
  .style('y', () => this.y(0))
  .style('width', this.x.rangeBand())
  .style('height', 0);

// Run CSS animation
setTimeout(()=> {
  this.bars.classed('bar', true)
  .style('height', (d) => this.height - this.y(+d.total) )
  .style('y', (d) => this.y(+d.total));
}, 1000);
}
```

The only thing that's a bit surprising this time is that we load the data using a promise and then return it from the constructor. This is a bit tricky — what's being returned isn't the resolved promise data but, rather, the promise object itself. The resolved data is taken care of in the constructor when we use `.then()` and attached to the class as a property. We will use this ability to return unresolved promises later on in the chapter to let our child classes know when the data has resolved.

We're using a dataset of the UK prison population from 1900 to 2015, which is available at https://github.com/aendrew/learning-d3/blob/chapter4/src/data/uk_prison_data_1900-2015.csv or in the `src/` data folder of the book's repo.

Change `index.js` to look like this:

```
import {PrisonPopulationChart} from './chapter4';
new PrisonPopulationChart('data/uk_prison_data_1900-2015.csv');
```

You might have noticed that we now only use the constructor to load in data; all the D3 work is in the `drawChart()` class method. Although the earlier chapters were pretty light on object-oriented programming concepts, we're going to start writing tighter classes now that we're dealing with user interaction. Breaking down a project into smaller functions is important for several reasons: it helps prevent you repeating yourself when constructing the stages of a user's journey and it allows each piece of the project to be more easily tested. We'll get into automated testing later but, for now, start thinking of ways you can construct your classes so that each major operation is in its own method.



As a taste of some of the cool things ES2015 classes allow you to do, did you see how I set the margin to a new value in the constructor? Our `BasicChart` class is smart enough to pick up the new value and adjust the chart created by our parent class accordingly.

After the constructor, we move onto `drawChart()`. The first part is setting up the scales and axes in the same way as before:

```
let data = this.data;

this.y = d3.scale.linear().range([this.height,
    this.margin.bottom]);
this.x.domain(data.map((d) => d.year));
this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

this.xAxis = d3.svg.axis().scale(this.x).orient('bottom')
    .tickValues(this.x.domain().filter((d, i) => !(i % 5)));

this.yAxis = d3.svg.axis().scale(this.y).orient('left');

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

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

If you hadn't noticed, we defined the `x` scale back up in the constructor:

```
this.x = d3.scale.ordinal().rangeBands([this.margin.left,
    this.width], 0.1);
```

This is so that we get immediate access to it from a child class that we will create later on in the chapter. It's a standard ordinal scale using `.rangeBands`. We also filter the `x` axis ticks so we get a tick every 5 years.



Pop-quiz! Why did we use an ordinal scale for years instead of a time or linear scale?

While we could have used either, an ordinal scale makes the most sense because we're creating a bar chart of years – time and linear scales will render the tick on the left edge of the bar, which looks silly.

Then, we set all the bars:

```
this.bars = this.chart.append('g').classed('bars', true)
  .selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .style('x', (d) => this.x(d.year))
  .style('y', () => this.y(0))
  .style('width', this.x.rangeBand())
  .style('height', 0);
```

This puts them into a group element with the class `.bars`. I've set the `y` value to zero as well as the height. This sets the initial state or the first keyframe. Next, we added the following right after that:

```
// Run CSS animation
setTimeout(()=> {
  this.bars.classed('bar', true)
  .style('height', (d) => this.height - this.y(d.total) )
  .style('y', (d) => this.y(d.total));
}, 1000);
```

This adds the `.bar` class to our bars and sets both its height and `y` value to their end keyframe values. Note how we've used `.style()` instead of `.attr()` – this is so that the new values are recorded in the `style` attribute tag, which means that our CSS transition will affect it. We do this separately in a `setTimeout` delay because we don't want the transition to be applied on the initial render and, without the delay, the user won't see the initial state.

Lastly, add the following to `index.css`:

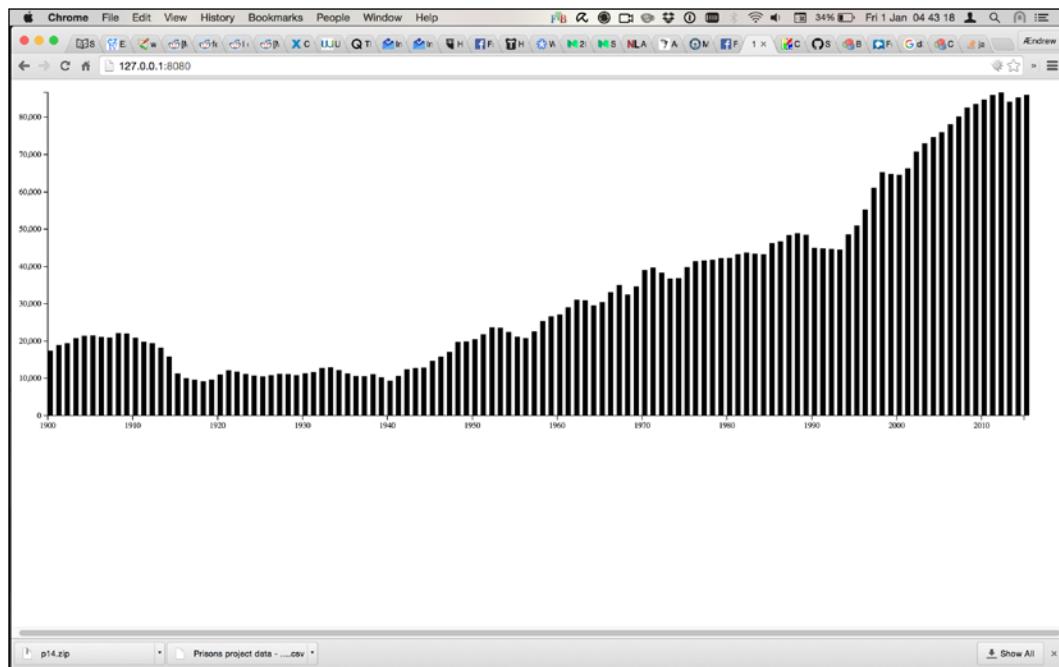
```
.bar {
  transition: all 1s ease-out;
}
```

This simply tells the web browser to animate any style property changes over one second and use the `ease-out` easing.

You should now have a CSS-driven bar chart!



For more on the CSS transitions API, visit this Mozilla Developer Network article: https://developer.mozilla.org/Web/CSS/CSS_Transitions/Using_CSS_transitions.



When should you use CSS transitions instead of the D3 transition API? There are two big use cases where it really makes a lot of sense: when you have to animate a lot of SVG elements and when your audience is viewing your work on a cellphone. It's often really difficult to create smooth animations using JavaScript-based interpolations on mobile devices because several things affect execution — most notably page scroll, which has made making fancy long form-style articles particularly nightmarish to get working on a mobile device in the last few years (though, luckily, this has changed as of iOS 8).

Generally, D3's transition API is easier to use, more powerful and doesn't mix animation logic with styling logic. It's also pretty fast — you shouldn't be afraid to use it by default, you can switch particular features to CSS transitions if you feel the performance benefit would improve the user experience.

Lastly, it makes a lot of sense to animate user interface interactivity using CSS transitions as UI elements tend to be simpler than charts and their animations shouldn't impact on performance when animating the SVG graphic.

What other ways are there to animate SVG? So glad you asked!

In the beginning, there was **SMIL** or **Synchronized Multimedia Integration Language**. To use SMIL, you use `<animate>` tags, which you put in your SVG markup. If this sounds gross already, it is. Luckily, Chrome 45 deprecated it, which means you never have to worry about learning it. The morbidly curious can visit: https://mdn.io/SVG_animation_with_SMIL.

In the future, an alternative to CSS transitions will be the Web Animations API. It's currently not supported by anything from Microsoft or Apple but it's been in Chrome since version 36 and Firefox since version 41. It has the *nicest* syntax, is all based on JavaScript and works well with D3. Here's an example:

```
d3.selectAll('.bar').each(function(d, i) {
  this.animate([
    {transform: `translate(${x(i)}, ${y(d)})`}
  ], {
    duration: 1000,
    iterations: 5,
    delay: 100
  });
});
```

Alas, as always, web development is a toy chest full of things you can't reliably use *just quite yet*. If you want to play with it while it's still being standardized, check out the fantastic polyfill at <https://github.com/web-animations/web-animations-js> that enables the use of web animations in most modern browsers. I'll hopefully be able to write more about the Web Animations API in the next edition of this book!



Interacting with the user

This is it. This is where all of the UX tidbits that I've been dropping throughout the chapter and all the ES2015 ideas that you've been learning come together — let's make a simple explanatory graphic that uses interaction to walk the viewer through data.

The first step in any visualization involving user interactivity is to plan exactly what you want the visualization to do, how you want your viewers to interact with it, and what you want to say about the data. What is the data's story? What's the best way to tell it?

We have the numerical product of over a century of incarceration in a western country in the prison population dataset. There are many ways we can look at this data. We can look at how the prison population has risen versus the overall population growth or we can look at how the prison population has risen or fallen in relation to known historical events. Often, you'll need more than one chart — for instance, when I used this data in a project for *The Times*, the piece had no less than five charts and one map, with the reader being walked through each graphic in sequence. This is where we start to get into actual data journalism territory, which is far beyond the scope of this short section. Suffice to say, however, that it helps to write down these things in either bullet point or paragraph form before you start writing any code. The real work is often done long before the first line of JavaScript is ever written.

In this particular instance, because we have a century of data, we're going to look at a few notable historical points. The graphic will have five states which will be navigated through a series of five buttons:

1. Initial view — years 1900 to 2015. This provides a general overview of how the prison population has risen over time.
2. Zoom 1900 to 1930. Highlights 1914–1918. The text explains how the population rose due to the end of World War I.
3. Zoom 1930 to 1960. Highlights 1939–1945. The text explains how the population rose after World War II.
4. Zoom 1960 to 1990. Discusses the rise of the consumer society.
5. Zoom 1990 to 2015. Highlights 1993 and explains the sharp rise after the murder of James Bulger.

We're keeping the user interface deliberately simple but remember that simpler is often better, particularly when building for an audience on mobile devices (sliders are much harder to use on touch devices than buttons, for instance).

Basic interaction

Much like elsewhere in JavaScript land, the principle for interaction is simple — attach an event listener to an element and do something when it's triggered. We add and remove listeners to and from selections with the `.on()` method, an event type (for instance, `click`), and a listener function that is executed when the event is triggered.

We can set a capture flag which ensures our listener is called first and all other listeners wait for our listener to finish. Events bubbling up from children elements will not trigger our listener.

You can rely on the fact there will only ever be a single listener for a particular event on an element because old listeners for the same event are removed when new ones are added. This is very useful when trying to eliminate unpredictable behavior.

Just like other functions acting on element selections, event listeners get the current datum and index and set the `this` context to the DOM element. The global `d3.event` lets you access the actual event object.

We're going to create a new class that extends our last chart:

```
export class InteractivePrisonPopulationChart extends
PrisonPopulationChart {
  constructor(path) {
    let p = super(path);
    this.scenes = require('../data/prison_scenes.json');
    this.scenes.forEach(
      (v, i) => v.cb = this['loadScene' + i].bind(this));

    p.then(() => this.addUIElements());
  }
}
```

Creating a scaffold like this is really useful when starting a more involved project like this. Additionally, I have pulled out the chapter data from a JSON file and assigned a callback function to each scene. You can often get away with having just one draw function that does something clever to load each scene but it's often easier to start out thinking in terms of discrete segments so that you don't end up with a single function with a ton of conditional logic.



Pay attention to the `.bind(this)` call in that ugly `forEach` loop I wrote. Without that, we wouldn't have access to our class methods via `this` in the scene functions!
A good rule of thumb when writing ES2015 classes is: when in doubt and throwing a lot of `TypeErrors`, check whether `this` is what you think it should be!

We gain the advantage of having all that work we just did relegated to the `super()` call because we're extending `PrisonPopulationChart`, in effect concentrating all the rendering code in a separate class. You don't necessarily need to create a totally separate class for interaction; I did it here to simplify everything. Also, note how we assign whatever the parent constructor returns to a variable? This is because we returned a promise in the parent constructor, which allows us to use `.then()` to ensure that the data is loaded and attached to `this.data` before we continue.

Let's start by giving ourselves some space underneath the chart. Add the following to the end of the constructor:

```
this.height = window.innerHeight / 2;
this.chart.attr('height', this.height);
this.svg.attr('height', this.height + 50);
this.margin.right = 10;
this.margin.bottom = 10;
```

This gives us roughly half of the window to work with. You'll want to experiment a bit to see what the best combination is or possibly set the element size using CSS media queries.

Next, let's create our UI elements — in this case, five buttons. Add a function called `addUIElements`, shown as follows:

```
addUIElements() {
  this.buttons = d3.select('#chart')
    .append('div')
    .classed('buttons', true)
    .selectAll('.button')
    .data(this.scenes).enter()
    .append('button')
    .classed('scene', true)
    .text(d => d.label)
    .on('click', d => d.cb())
    .on('touchstart', d => d.cb());

  this.words = d3.select('#chart').append('div');
  this.words.classed('words', true);
}
```

There is nothing new here — create a new button for each element in the `chapters` array and run its callback function when any button is clicked or tapped. We also drop a plain ol' `div` into the chart area, which is where we'll put all of our text describing each scene.



Haven't run into `touchstart` before? Think of it as the `mousedown` event of touch. Other useful touch events are `touchmove`, `touchend`, `touchcancel`, and `tap`. Mozilla's documentation explains touch events in more detail at https://developer.mozilla.org/Web/Guide/API/DOM/Events/Touch_events.

We need a function to clear selected bars — let's do something a little bit different in terms of sequencing animation and use promises. Although D3 doesn't use promises for animation natively (or for much in general, really), it's an idiom becoming widely used throughout the JavaScript world, particularly by the Angular 2 community. Add the following method to your class:

```
clearSelected() {
  return new Promise((res, rej) => {
    d3.selectAll('.selected').classed('selected', false);
    res();
  });
}
```

This returns a new promise, which resolves after we've removed the `.selected` class from all the bar elements.

We also need a method to select specific bars:

```
selectBars(years) {
  this.bars.filter((d) => years.indexOf(
    Number(d.year)) > -1).classed('selected', true);
}
```

We now need to create our chart's update function. Let's make it return a promise like our animation functions:

```
updateChart(data = this.data) {
  return new Promise((res, rej) => {
    let bars = this.chart.selectAll('.bar').data(data);

    this.x.domain(data.map((d) => d.year));
    this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

    this.chart.selectAll('.axis.x').call(
      d3.svg.axis().scale(this.x).orient('bottom')
      .tickValues(this.x.domain().filter((d, i) => !(i % 5))));
    this.chart.selectAll('.axis.y')
      .call(this.yAxis);
```

```
// Update
bars.style('x', (d) => this.x(d.year))
  .style('width', this.x.rangeBand())
  .style('height', (d) => this.height - this.y(+d.total) )
  .style('y', (d) => this.y(+d.total))

// Add
bars.enter()
  .append('rect')
  .style('x', (d) => this.x(+d.year))
  .style('width', this.x.rangeBand())
  .style('height', (d) => this.height - this.y(+d.total) )
  .style('y', (d) => this.y(+d.total))
  .classed('bar', true);

// Remove
bars.exit().remove();

res();
});
}
```

We do the standard D3 update, add and remove routine, setting the scales to the selected data. Finally, we resolve the promise by calling `res()` at the end. Note how we take an argument which defaults to the class `data` property.

 Default argument values are a new feature in ES2016! This is such a nice addition to the language, as somebody who has written so much existence checking logic for basic functions it'd make you cry.

Let's look at that again:

```
updateChart(data = this.data)
```

This means the `data` argument will equal the class' internal `data` property if no argument is supplied. The next time that you feel yourself reaching for `typeof argumentVar !== 'undefined'`, give default arguments a try.

Lastly, let's set up our states. The first one is easy:

```
loadScene0() {
  this.clearSelected().then(() => this.updateChart());
  this.words.html('');
}
```

We clear the selected bars and then update the chart. We set words to be an empty string. We need to clear both the selected bars and the words on the first scene even if they originate in that way because we're allowing the viewer to navigate the scenes in a non-linear order – they can always come back to the first scene, which means we have to clear anything created by another scene:

```
loadScene1() {
  let scene = this.scenes[1];
  this.clearSelected().then(() => {
    this.updateChart(this.data.filter((d) =>
      d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.year)) > -1));
    .then(() => this.selectBars(d3.range(1914, 1918)));
  });
  this.words.html(scene.copy);
}
```

This is somewhat more interesting. We do the same thing as we did in the first scene but, in the `then()` callback, we create a range from the domain given in the scene array. This gives us an array of the years with which we can filter the bars. Lastly, we set the text in the interactive dialog (helpfully written by my colleague, Sam Joiner) to form a meaningful sentence explaining the change.



Copy is what people in the newspaper and advertising businesses call text content. I don't mean it in the duplicate sense here!



The next bit is rather repetitive:

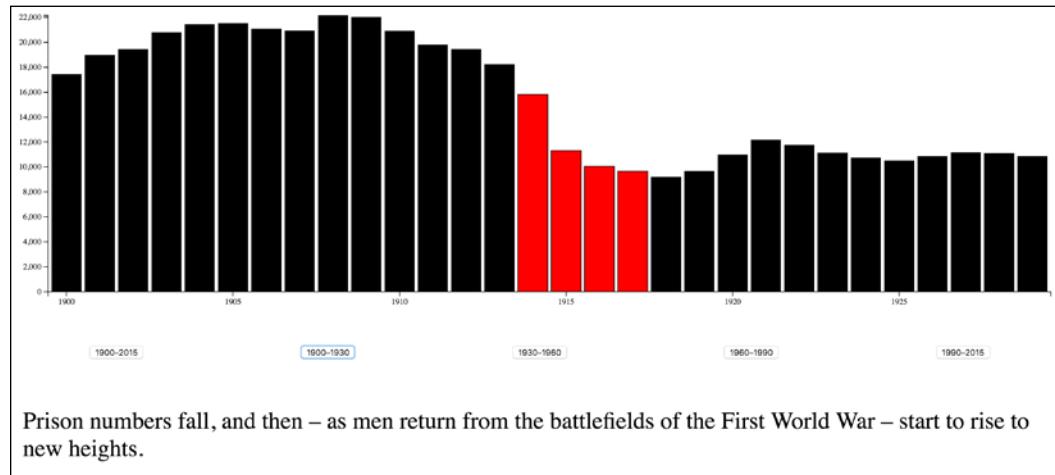
```
loadScene2() {
  let scene = this.scenes[2];
  this.clearSelected().then(
    () => {
      this.updateChart(this.data.filter((d) =>
        d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.year)) > -1));
      .then(() => this.selectBars(d3.range(1939, 1945)));
    });
  this.words.html(scene.copy);
}

loadScene3() {
  let scene = this.scenes[3];
  this.clearSelected().then(
```

```
( () => this.updateChart(this.data.filter((d) =>
d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.year)) 
> -1))
);
this.words.html(scene.copy);
}

loadScene4() {
let scene = this.scenes[4];
this.clearSelected().then(
() => {
this.updateChart(this.data.filter((d) =>
d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.year)) 
> -1))
.then(() => this.selectBars([1993]));
}
);
this.words.text(scene.copy);
}
```

And there you have it – your first interactive data visualization!



Behaviors

In the last section, we created an *explanatory* graphic that used interaction to guide the user through the data. Often, however, the goal is just to make a dataset interactive and give the user some way of manipulating it, in other words, an *exploratory* graphic.

D3's behaviors save a boatload of time in setting up the more complex interactions in a chart. Additionally, they're designed to handle differences in input devices so you only have to implement a behavior once to have it work both with a mouse and on touch devices. The two currently supported behaviors are drag and zoom, both of which will get you pretty far.

Drag

Instead of having the user click buttons in the last example, what if we just let them drag the chart area to see the UK's prison population increase? It involves a bit more work from the user but it also gives them the ability to navigate freely through the chart — which may be desirable in some circumstances.

Let's extend the last chart we created and override the `.addUIElements` method. Create a child class that extends `InteractivePrisonPopulationChart` and set it up as shown below, thus overriding the parent `.addUIElements` method and avoiding a bunch of unnecessary buttons.

```
export class DraggableInteractivePrisonChart extends
InteractivePrisonPopulationChart {
  constructor(path) {
    let p = super(path);
    this.x.rangeBands([this.margin.left, this.width * 4]);
  }
  addUIElements() {}
}
```

The first thing that we need to do is create a hit box — we want dragging on the bars area to result in the bars being dragged. Alas, `g` container elements aren't clickable so we need to get the dimensions of `g.bars`, placing an invisible `rect` element of that size on top of it. Yet further alas, the dimensions of `g.bars` won't make any sense until the CSS transition has finished. Luckily, we can listen to the `transitionend` event to see if this has occurred. Add this to the `addUIElements` function, as shown here:

```
let bars = d3.select('.bars').on('transitionend', ()=> {
  let dragContainer = this.chart.append('rect')
    .classed('bar-container', true)
    .attr('width', bars.node().getBBox().width)
    .attr('height', bars.node().getBBox().height)
    .attr('x', 0)
    .attr('y', 0)
    .attr('fill-opacity', 0);

}) ;
```

`SVGELEMENT.getbbox()` is a tremendously useful function that gives you the x, y, height and width of a particular element. Note, however, that you can't use this on D3 selections — only SVG elements! That's why we get the underlying element out of the selection by using `selection.node()`.

Now we need to set up the drag behavior. Still inside the `transitionend` callback, add the following:

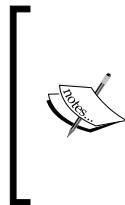
```
let drag = d3.behavior.drag().on('drag', () => {
  let barsTransform = d3.transform(bars.attr('transform'));
  let xAxisTransform = d3.transform(
    d3.select('.axis.x').attr('transform'));
  bars.attr('transform',
    `translate(${barsTransform.translate[0] + d3.event.dx}, 0)`);
  d3.select('.axis.x').attr('transform',
    `translate(${xAxisTransform.translate[0] + d3.event.dx},
      ${xAxisTransform.translate[1]})`);
});
```

Firstly, we get the current transform values of the x axis and the bars themselves. `d3.transform()` is a very helpful function for getting a transform matrix into a useful object form so we use that twice. We then translate both the axes and bars their current translation distance, plus the distance the user drags across our hit box, provided by `d3.event.x`. We also provide the y translate of the axis since we set that way back up in our parent class.

Finally, after our call above but still inside the `transitionend` event callback, add the following to instantiate the drag behavior on our container.

```
dragContainer.call(drag);
```

Now you can drag!



You may have noticed that the bars drag behind the x axis which looks a bit unsightly. You can't add backgrounds to `g` elements as you can with `div` so you'll have to append a white `rect` element to the axis. The technique for getting the `rect` width and height is the same as it is for the hit box — I'll leave fixing this as an exercise for you.

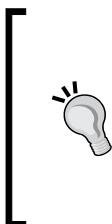
Zoom

Despite the name, the zoom behavior lets you do more than just zoom—you can also pan! Like the drag behavior, zoom automatically handles both mouse and touch events and then triggers the higher-level zoom event. Yes, this means pinch-to-zoom works! That's pretty awesome, if you ask me.

Remember that map from *Chapter 3, Making Data Useful*, the one with the rendition flights?

Let's commit a crime against computational efficiency and make it zoom and pan.

I am warning you that this will be very rudimentary and painfully slow. This is not how you'd make a real explorable map, just an example to let us play with zooming. In real life, you would use tiling, progressive detailing, and other tricks. You also wouldn't write everything in the constructor, as we did in this example.



After even one example organizing all the functions into class methods, doesn't this already feel atrociously messy? Hopefully, you can see why it's so much better to organize intelligently, plan and organize your classes — you can always revisit a project and being able to pick your code back up and understand it as quickly as possible is both enormously important and very much facilitated by writing clean and extensible code.

Let's go back to the map. Jump to the end of the `GeoDemo` draw function in `chapter3.js` and add a call to `zoomable`; we'll define this next.

```
zoomable();
```

While you're in `draw`, turn off the river and cities layers to help with performance.

`zoomable` sets up the behavior on the chart so put this at the end of the constructor. Next, we'll define what the behavior actually is:

```
function zoomable() {
  chart.call(
    d3.behavior.zoom()
      .translate(projection.translate())
      .scale(projection.scale())
      .on('zoom', () => onzoom())
  );
}
```

We defined a zoom behavior with `d3.behavior.zoom()` and immediately called it on the whole image.

We set the current `.translate()` vector and `.scale()` to whatever the projection was using. The `zoom` event calls our `onzoom` function.

Let's define it as follows:

```
function onzoom() {  
    projection  
        .translate(d3.event.translate)  
        .scale(d3.event.scale);  
  
    d3.selectAll('path')  
        .attr('d', d3.geo.path().projection(projection));  
  
}
```

Firstly, we told our projection that the new translation vector was in `d3.event.translate`.

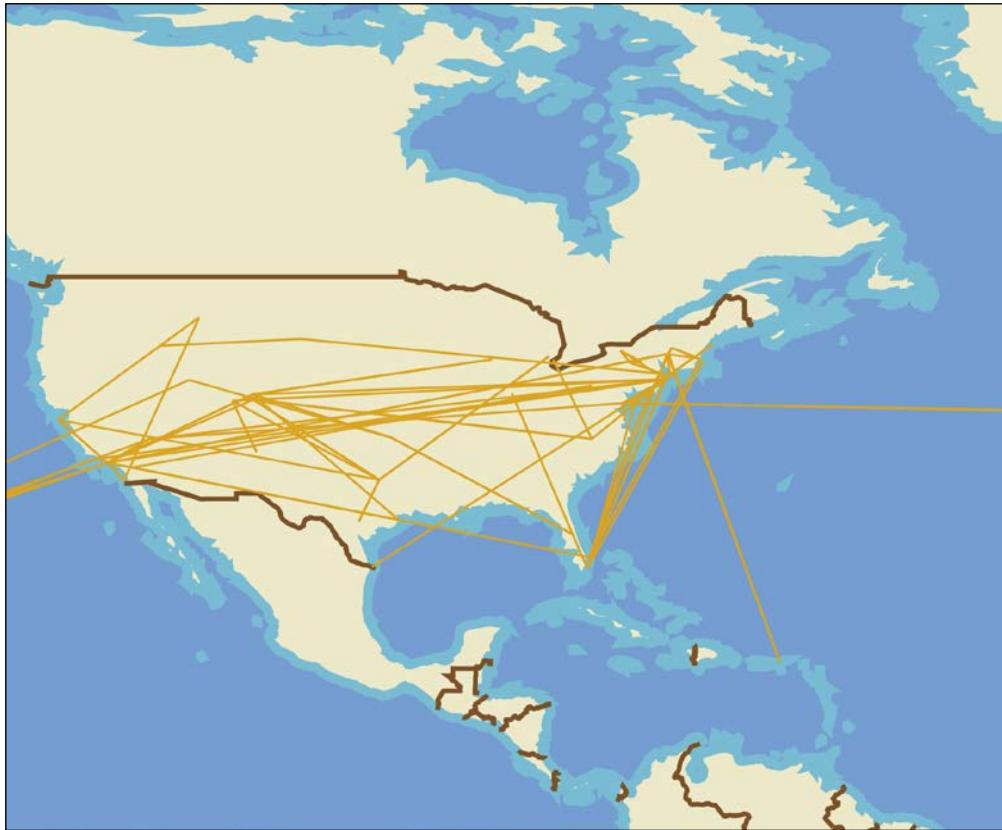
The translation vector pans the map with a transformation as in *Chapter 2, A Primer on DOM, SVG, and CSS*. `d3.event.scale` is just a number the projection uses to scale itself, effectively zooming the map.

Next, we recalculate all the routes using the changed projection:

```
d3.selectAll('line.route')  
    .attr('x1', (d) => projection([d.from.lon, d.from.lat])[0])  
    .attr('y1', (d) => projection([d.from.lon, d.from.lat])[1])  
    .attr('x2', (d) => projection([d.to.lon, d.to.lat])[0])  
    .attr('y2', (d) => projection([d.to.lon, d.to.lat])[1]);
```

The positioning function is exactly the same as in `addRenditions` because geographic projections handle panning out of the box. The thickness stays more or less the same throughout.

You can now explore the world!



Have patience, though, it's *reaaaaal* slow. Redrawing everything on every move does that. For a more performant and zoomable map, we'd have to use data with less detail when zoomed out, draw a sensible number of lines, and possibly avoid drawing parts of the map that fall out of the image anyway. Another way in which we could make it more performant is by transforming the entire map area instead of reprojecting on each event. This not only requires less computing but it also allows the GPU to do some of the work. Let's give that a shot!

Replace the entirety of `zoomable` with the following snippet:

```
function zoomable() {
  chart.call(d3.behavior.zoom()
    .center([chart.attr('width') / 2, chart.attr('height') / 2])
    .scale(projection.scale())
    .on('zoom', () => onzoom()));
}
```

All we've done here is replace the translate line with a line telling it to always zoom from the center of the chart — it works without doing this but it's a bit janky. Next, replace `onzoom` with the following:

```
function onzoom() {  
  let scaleFactor = d3.event.scale / projection.scale();  
  chart.attr('transform', `translate(${d3.event.translate})`)  
    scale(`${scaleFactor}`);  
  d3.selectAll('line.route').each(function() {  
    d3.select(this).style('stroke-width', `${2 /  
      scaleFactor}px`);  
  });  
}
```

We're doing two things here: we're getting the real scale factor by dividing the scale given by the zoom event by the initial scale as it was defined for the projection (in this case, 150). We then use this to scale and translate the `g` element holding all of our geometry. We also divide the initial stroke width (2px, set in `index.css`) by the scaling factor to prevent the route lines from getting big and blocky as we zoom in.

That's miles better, isn't it? D3 gives you a lot of freedom in how you implement things; understanding which is the best method for any particular project comes from practice and knowing your audience.

Brushes

Brushes are similar to zoom and drag and are a simple way to create complex behavior — they enable users to select a part of the canvas.

Strangely, they aren't considered as a behavior but fall under the `.svg` namespace, perhaps because they are mostly meant for visual effects.

To create a new brush, we call `d3.svg.brush()` and define its `x` and `y` scales using `.x()` and `.y()`. We can also define a bounding rectangle.

Time for an example!

We're going back yet again to our all-singing, all-dancing prison population graph. This is the last example in which I will use it, I promise. We are going to let the user zoom in on a group of bars by selecting them with a brush, zooming out on a right-click.

Begin by creating a new class that extends `InteractivePrisonPopulationChart`, like so:

```
export class SelectableInteractivePrisonChart extends
  InteractivePrisonPopulationChart {
  constructor(path) {
    super(path);
  }

  addUIElements() {}
  brushstart() {}
  brush() {}
  brushend() {}
  rightclick() {}
}
```

This is almost the same setup that we used for the drag example. Add the following to `addUIElements`:

```
this.chart.append('g')
  .classed('brush', true)
  .call(d3.svg.brush().x(this.x).y(this.y)
    .on('brushstart', this.brushstart.bind(this))
    .on('brush', this.brushmove.bind(this))
    .on('brushend', this.brushend.bind(this)));
```

We made a new grouping element for the brush and called a freshly constructed `d3.svg.brush()` with both scales defined. The `.brush` class helps with styling. We also bind the local context to each event callback so that we still have access to our class methods.

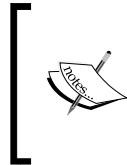
Next, we define listeners for the `brushstart`, `brush`, and `brushend` events as part of our class. We're not doing anything with `brushstart` so let's skip straight to `brushmove`:

```
brushmove() {
  let e = d3.event.target.extent();
  d3.selectAll('.bar').classed('selected', (d) =>
    e[0][0] <= this.x(d.year)
    && this.x(d.year) <= e[1][0]
  );
}
```

`brushmove` is where the real magic happens. Firstly, we find the selection's boundaries by using `d3.event.target.extent()`.

`d3.event.target` returns the current brush and `.extent()` returns a set of two points — the upper-left and bottom-right corners.

Then, we go through all the bars and turn the `.selected` class on or off, depending on whether the bar's position lies within the bounding box.



Note that the values returned by `extent` are wholly reliant on the type of scale you provide to it — if you give it a linear scale, you'll compare the unscaled datum value against the extent value. In an ordinal scale like we have here, we have to scale the datum value before comparing.



Next, we define what happens when the mouse button is released:

```
brushend() {
  let selected = d3.selectAll('.selected');

  // Clear brush object
  d3.event.target.clear();
  d3.select('g.brush').call(d3.event.target);

  // Zoom to selection
  let first = selected[0][0];
  let last = selected[0][selected.size() - 1];
  let startYear = d3.select(first).data()[0].year;
  let endYear = d3.select(last).data()[0].year;
  this.clearSelected().then(() => {
    this.updateChart(this.data.filter((d) =>
      d3.range(startYear, endYear).indexOf(Number(d.year)) > -1));
  });

  let hitbox = this.svg
    .append('rect')
    .classed('hitbox', true)
    .attr('width', this.svg.attr('width'))
    .attr('height', this.svg.attr('height'))
    .attr('fill-opacity', 0);

  hitbox.on('contextmenu', this.rightclick.bind(this));
}
```

Quite a lot happens here. Firstly, we clear the brush overlay, then we figure out the first and last element selected by the brush. From that, we get the start and end years, which we supply to `d3.range`, redrawing the chart just like we did in `InteractivePrisonPopulationChart`. Lastly, we add a hit box, which we'll use to listen to the `contextmenu` event (`contextmenu` being the right-click variant of the `click` event).

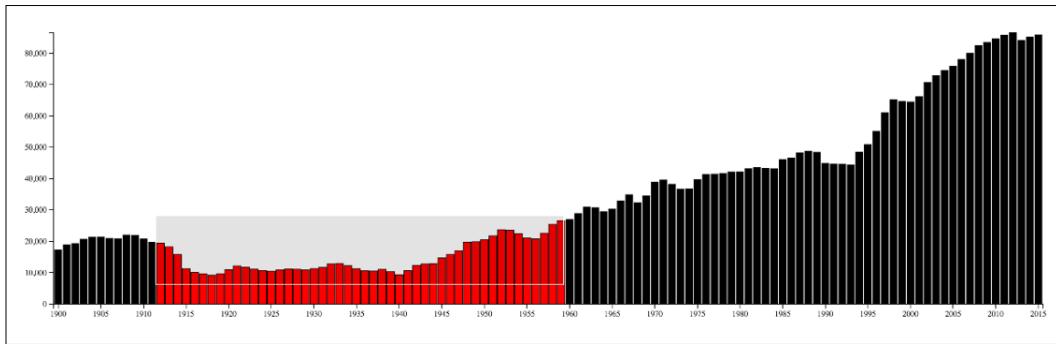
Our HTML needs some more styling definitions, like this:

```
.brush .extent {
  stroke: #fff;
  fill-opacity: .125;
  shape-rendering: crispEdges;
}

.bar.selected {
  stroke: black;
}
```

In this case, we're still using the CSS transitions from the first example. It's preferable to use CSS transitions rather than what D3 can do in this case — brushes sometimes have problems with D3 transitions and change properties immediately.

When you select some elements, the image will look like this:



Summary

Wow, what a fun chapter!

You've made things jump around the page, almost killed your computer and patience with a zoomable map, and made one supremely awesome bar graph. Well done!

In this chapter, we've animated with transitions, interpolators and timers, and then we learned how to do some of that with CSS. We then learned the difference between explanatory and exploratory visualizations, and used interactivity to create the former. We then made some exploratory visualizations by using behaviors with some of our previous projects. Things are starting to look pretty snazzy, aren't they?

In the next chapter, we'll be looking at creating a whole boatload of really pretty charts using D3's black magic – layouts. Combining the skills you've learned in this chapter and the next one will mean that you're able to produce some truly fantastic charts. I hope that you're ready – this is where stuff starts getting really cool!

5

Layouts – D3's Black Magic

Most of us look at the Internet for inspiration and code samples. You find something that looks great, you look at the code, and your eyes glaze over. It doesn't make any sense.

The usual culprit is D3's reliance on layouts for anything remotely complicated. The black magic of taking some data, calling a function, and—voilà—visualization! This elegance makes layouts look deceptively difficult, but they make things a lot easier when you get the hang of them.

In this chapter, we'll go in, guns blazing, with everything you've learned so far to create 11—count 'em! 11!—visualizations of the same dataset.

What are layouts and why should you care?

D3 layouts are modules that transform data into drawing rules. The simplest layout might only transform an array of objects into coordinates, like a scale.

But we usually use layouts for more complex visualizations, drawing a force-directed graph or a tree for instance. In these cases, layouts help us separate calculating coordinates from putting pixels on a page screen. This not only makes our code cleaner but also lets us reuse the same layouts for vastly different visualizations.

Theory is boring. Let's dig in.

Built-in layouts

By default, D3 comes with 12 built-in layouts that cover most common visualizations. They can be split roughly into **normal** and **hierarchical** layouts. Normal layouts represent data in a flat hierarchy, while hierarchical layouts generally present data in a tree-like structure. The normal layouts are as follows:

- Histogram
- Pie
- Stack
- Chord
- Force

The hierarchical layouts are as follows:

- Partition
- Tree
- Cluster
- Pack
- Tree map

To see how they behave, we're going to make an example for each type. We'll start with the humble pie chart and histogram and then progress to force-directed graphs and fancy trees. We'll be using the same dataset for all examples so that we can get a feel of how different presentations affect the perception of data.

We're getting pretty good at this by now, so we're going to make these examples particularly magnificent. That's going to create a lot of code, so every time we come up with something reusable, we'll put it in a **helpers.js** file as a function. We'll be exporting this as a big ol' bucket of functions instead of a class this time, mainly because we will have to deal with much less weirdness due to this changing context.

Let's create an empty **helpers.js** file. We'll be adding functions to this file willy-nilly and then importing them under the same namespace elsewhere. We could have done this in a bunch of different ways, for instance, exporting an object with a bunch of different methods, or creating a class or a factory function. However, doing it this way means that we can either selectively choose functions from the module without importing them all, or import all of them during development with a single statement. For now, just add this line at the top:

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

Meanwhile, let's once again extend `BasicChart`, creating a class that will create all our different representations of the same chart. Add the following code to a new file called `chapter5.js`:

```
import * as helpers from './helpers';
import {BasicChart} from './basic-chart';
let d3 = require('d3');

export class PoliticalDonorChart extends BasicChart {
  constructor(chartType, ...args) {
    super();
    require('./chapter5.css');

    let p = new Promise((res, rej) => {
      d3.csv('data/uk_political_donors.csv',
        (err, data) => err ? rej(err) : res(data));
    });

    p.then((data) => {
      this.data = data;
      this[chartType].call(this, ...args);
    });

    return p;
  }
}
```

You'll notice that we're importing everything from `helpers.js` under the `helpers` namespace. As we add helper functions, they'll automatically be added to the `helpers` object we're importing from that file. Otherwise, there is nothing unusual:

1. First, we load the CSS file that we'll make in a moment.
2. Then we call `super`.
3. We create a new `Promise`.
4. We have D3 load and parse the CSV data.
5. Then we resolve the promise if everything is okay, calling the method specified by the constructor's first argument.

The rest of the constructor's arguments are then passed as arguments to the method we're calling.

 The `...args` bit in the constructor's arguments is a new feature in ES2016, called the **rest parameter**. It collects every argument after the ones specifically defined as an array. We then use another new ES2016 feature, called the **spread operator**, in `this[chartType].call(this, ...args)` to destructure the array into its individual values. This lets us add as many arguments as we want to each chart method we're writing.

The dataset

The dataset that we are going to use in all of these examples is a list of donations to various political entities of the UK: who received the donation, the amount, whom it was from, the type of donor, and the date it was made. Data like this is important, as it can indicate individual actors influencing the political process. This alone might not be enough for a story, but at the very least, it can point you towards groups or individuals worth pursuing further. It's in the book's repository at `src/data/uk_political_donors.csv`, or you can get it from <https://github.com/leilahaddou/graph-data/blob/master/pef.csv>.

 It's worth noting that I got this dataset from Leila Haddou's totally awesome tutorial on Neo4j, a network graph database. If you like the force-directed graph later on in this chapter, Neo4j might just totally blow your mind. Give it a shot! It is at <http://leilahaddou.github.io/neo4j-tutorial.html>.

Normal layouts

Time to draw! As mentioned, we'll begin with normal layouts. They display data in a flat hierarchy.

Using the histogram layout

We are going to use the `histogram` layout to create a bar chart of the number of donations received. The layout itself will handle everything from collecting values in bins to calculating heights, widths, and positions of the bars.

Histograms usually represent a probability distribution over a continuous numerical domain, but the names of donation recipients are ordinal. To bend the `histogram` layout to our will, we will have to turn names into numbers—we'll use a scale.

Since it feels like this could be useful in other examples, we'll put the code in `helpers.js`:

```
export function uniques(data, name) {
  let uniques = [];
  data.forEach((d) => {
    if (uniques.indexOf(name(d)) < 0) {
      uniques.push(name(d));
    }
  });
  return uniques;
}

export function nameId(data, name) {
  let uniqueNames = uniques(data, name);
  return d3.scale.ordinal()
    .domain(uniqueNames)
    .range(d3.range(uniqueNames.length));
}
```

These are two simple functions:

- `uniques`: This goes through the data and returns a list of unique names. We help it with the `name` accessor.
- `nameId`: This creates an ordinal scale that we'll be using to convert names into numbers. Expect to see both of these a lot in this chapter.

Now we can tell the histogram how to handle our data with `nameId`. Add the following method to your `PoliticalDonorChart` class:

```
histogram() {
  let data = this.data;

  let nameId = helpers.nameId(data, (d) => d.EntityName);
  let histogram = d3.layout.histogram()
    .bins(nameId.range())
    .value((d) => nameId(d.EntityName))(data);

  this.margin = {top : 10, right : 40, bottom : 100, left : 50};
}
```

Using `d3.layout.histogram()`, we create a new histogram and use `.bins()` to define the upper threshold for each bin. Given `[1, 2, 3]`, values under 1 go into the first bin, values between 1 and 2 into the second, and so on.

The `.value()` accessor tells the histogram how to find values in our dataset.

Another way to specify bins is by specifying the number of bins you want and letting the histogram uniformly divide a continuous numerical input domain into bins. For such domains, you can even make probability histograms by setting `.frequency()` to `false`. You can limit the range of considered bins with `.range()`.

Finally, we use the layout as a function on our data to get an array of objects with the following schema:

```
{  
  0: {  
    DonorName: String, // e.g. "Andrew Rininsland"  
    DonorStatus: String, // e.g. "Individual"  
    ECRef: String, // e.g. "c1234567"  
    EntityName: String, // e.g. "Green Party"  
    ReceivedDate: Date, // e.g. "27/01/15" (dd/mm/yy)  
    Value: String // e.g. "£1,234"  
  },  
  dx: Number,  
  x: Number,  
  y: Number  
}
```

It's worth reiterating that we define the histogram and then immediately run it on our dataset. Look at the following line:

```
let histogram = d3.layout.histogram()  
  .bins(nameId.range())  
  .value((d) => nameId(d.EntityName))(data);
```

It can also be written like this:

```
let histogramLayout = d3.layout.histogram()  
  .bins(nameId.range())  
  .value((d) => nameId(d.EntityName));  
let histogram = histogramLayout(data);
```

We just avoid the temporary variable for something we use only once here by immediately executing the layout and then storing the output in a variable.

The bin width is in the `dx` property, `x` is the horizontal position, and `y` is the height. We access elements in bins with normal array functions. Note that they're all strings, even the amount and date. We'll have to handle this in our code using `d3.format`.

Using this data to draw a bar chart should be easy by now. We'll define a scale for each dimension, label both axes, and place some rectangles for bars.

Next, add two scales:

```
let x = d3.scale.linear()
  .domain([0, d3.max(histogram, (d) => d.x)])
  .range([this.margin.left, this.width-this.margin.right]);

let y = d3.scale.log().domain([1, d3.max(histogram, (d) => d.y)])
  .range([ this.height - this.margin.bottom, this.margin.top ]);
```

The use of a logarithmic scale for the vertical axis will make it such that the number of donations received by the biggest entities doesn't totally flatten everything else.

Next, put a vertical axis on the left:

```
let yAxis = d3.svg.axis()
  .scale(y)
  .tickFormat(d3.format('f'))
  .orient('left');

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

We create a grouping element (the '`g`') for every bar and its label:

```
let bar = this.chart.selectAll('.bar')
  .data(histogram)
  .enter()
  .append('g')
  .classed('bar', true)
  .attr('transform', (d) => `translate(${x(d.x)}, ${y(d.y)})`);
```

Moving the group into position, as shown in the following code, means less work when positioning the bar and its label:

```
bar.append('rect')
  .attr({
    x: 1,
    width: x(histogram[0].dx) - this.margin.left - 1,
    height: (d) => this.height - this.margin.bottom - y(d.y)
  })
  .classed('histogram-bar', true);
```

Because the group is in place, we can put the bar a pixel from the group's edge. All bars will be `histogram[0].dx` units wide, and we'll calculate heights using the `y` position of each datum and the total graph height. Lastly, we create the labels:

```
bar.append('text')
  .text((d) => d[0].EntityName)
  .attr({
    transform: (d) =>
      `translate(0, ${this.height - this.margin.bottom - y(d.y)} + 7) rotate(60)`
  });
});
```

We move labels to the bottom of the graph, rotate them by 60 degrees to avoid overlap, and set their text to the `EntityName` property of the datum.

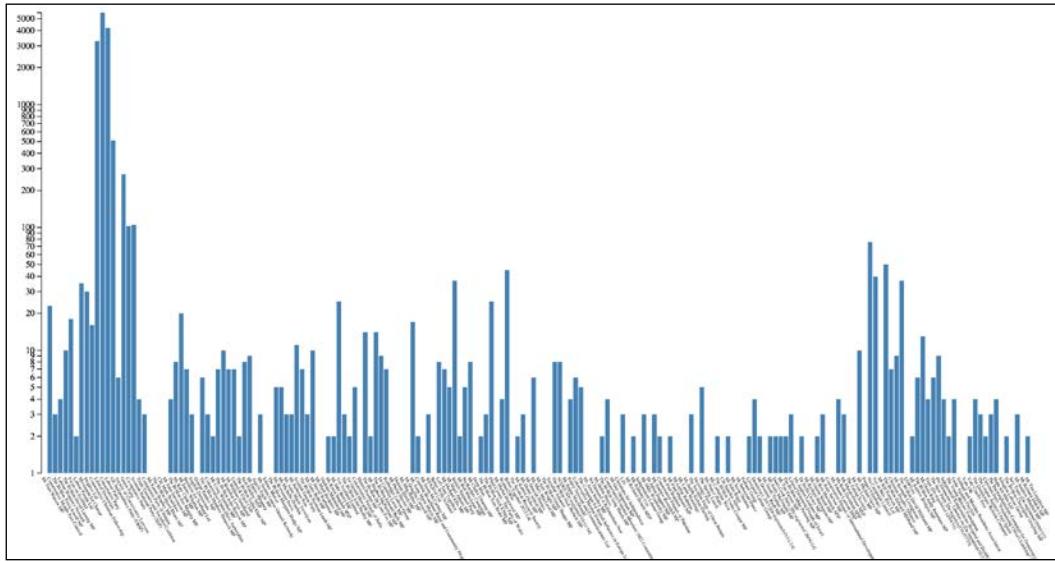
Now create a new file, `chapter5.css`, and add this code:

```
.axis path, .axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
.axis text {
  font-size: 0.75em;
}
rect.histogram-bar {
  fill: steelblue;
  shape-rendering: crispEdges;
}
.bar text {
  font-size: 0.4em;
}
```

Don't forget to require the CSS file in your JavaScript. Replace `index.js` with the following line:

```
require('./index.css');
import {PoliticalDonorChart} from './chapter5';
new PoliticalDonorChart('histogram');
```

Our bar chart looks like this:

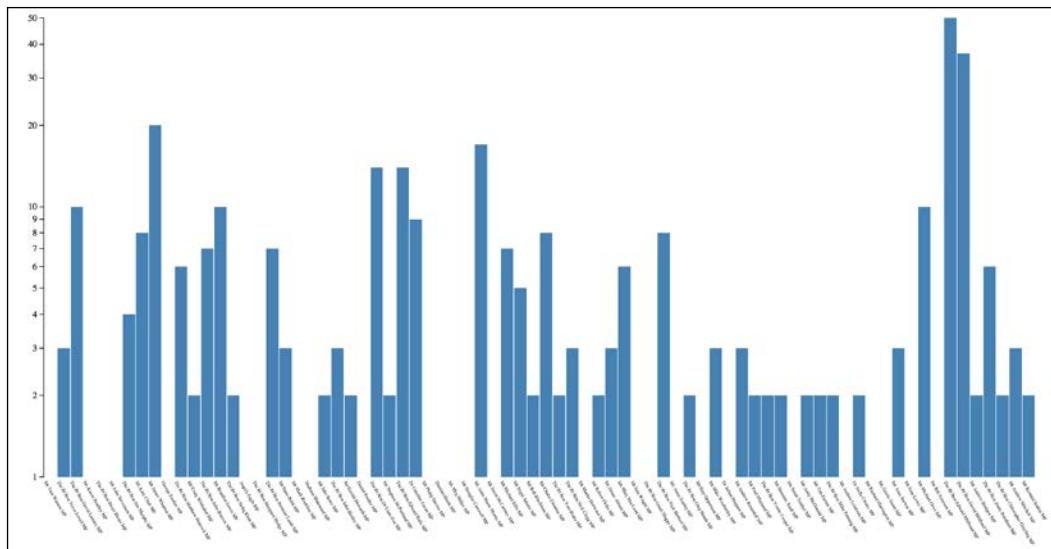


Hmm... It seems that the incumbent party got the most donations. This is totally unsurprising. Let's filter for only Members of Parliament (MPs, for those outside the Commonwealth).

Replace the first line of your `histogram` method with the following:

```
let data = this.data.filter((d) => d.EntityName.match(' MP'));
```

Much better! Note that we start at 1 instead of 0 as we're using a logarithmic scale. This means that MPs with only one donor appear as if they have none. You'd want to explain that to your audience if sticking to the log scale in such an instance, or more likely filter out individuals with only one donation:



Ed Miliband, then-leader of the opposition Labour Party, had the most donations in this particular set of data

Baking a fresh 'n' delicious pie chart

The preceding bar chart reveals that Ed Miliband received the most donations during this period. Let's find out who's making him so popular.

We are going to use the pie chart layout to cut the donations to *The Rt Hon Edward Miliband MP* into slices, showing how many donations he got from each donor. After filtering the dataset for donations going to Miliband, we have to categorize the entries by givers, and finally we feed them into the pie chart layout to generate a pie chart.

We can use the `histogram` layout to put data into bins depending on the `DonorName` property. Let's add a function to `helpers.js`:

```
export function binPerName (data, name) {
  let nameIds = nameId(data, name);
  let histogram = d3.layout.histogram()
    .bins(nameIds.range())
    .value((d) => nameIds(name(d)));
  return histogram(data);
}
```

Similar to the `uniques` and `nameId` functions, `binPerName` takes the data and a name accessor and returns histogram data.

Now create a `pie` method in your `PoliticalDonorChart` class:

```
pie(name) {
  let filtered = this.data.filter((d) => d.EntityName === name);
  let perDonor = helpers.binPerName(filtered,
    (d) => d.DonorName);
}
```

Entries in the `perDonor` variable will tell us exactly how many donations were received by the name specified by the function argument.

To bake a pie, we call the `pie` layout and give it a value accessor:

```
let pie = d3.layout.pie()
  .value((d) => d.length)(perDonor);
```

The `pie` layout is now full of slice objects, each holding the `startAngle` and `endAngle` values and the original value.

The entries look like this:

```
{
  data: Array[135],
  endAngle: 2.718685950221936,
  startAngle: 0,
  value: 135
}
```

We could have specified a `.sort()` function to change how slices are organized and a `.startAngle()` or `.endAngle()` function to limit the pie's size.

All that's left to do now is drawing a pie chart. We'll need an `arc` generator (just like the ones in *Chapter 2, A Primer on DOM, SVG, and CSS*) and some colors to tell the slices apart.

Finding 24 distinct colors that look great together is hard; luckily for us, `@ponywithhiccups` has jumped into the challenge and made the pick. Thank you!

Let's add these colors to `helpers.js`:

```
export const color = d3.scale.ordinal().range(['#EF3B39',
  '#FFCD05', '#69C9CA', '#666699', '#CC3366',
  '#0099CC', '#999999', '#FBF5A2', '#6FE4D0', '#CCCB31',
  '#009966', '#C1272D', '#F79420', '#445CA9',
  '#402312', '#272361', '#A67C52', '#016735', '#F1AAAF', '#A0E6DA',
  '#C9A8E2', '#F190AC', '#7BD2EA',
  '#DBD6B6']);
```

The color scale is an ordinal scale without a domain. We export it as a constant (that is, a variable that won't change) using the `const` keyword. Exporting variables that aren't constants isn't allowed.

If you try to redefine the `color` constant, Babel will throw this error:

`Module build failed: SyntaxError: .../learning-d3/src/helpers.js: Line 39: "color" is read-only`

Use constants for variables that really shouldn't ever be modified. Note, however, that you can still update your color scale by using `.range` and `.domain` as setters (that is, by supplying them a new array to replace the one you defined in `helpers.js`).

To make sure that the donors always get the same color, a function in `helpers.js` will help us fixate the domain, as shown in the following code:

```
export function fixateColors (data) {
  color.domain(uniques(data, (d) => d.DonorName));
}
```

Now, we can define the `arc` generator in our `pie` method and fixate the colors:

```
let arc = d3.svg.arc()
  .outerRadius(150)
  .startAngle((d) => d.startAngle)
  .endAngle((d) => d.endAngle);

helpers.fixateColors(filtered);
```

A group element will hold each arc and its label, as shown in the following code:

```
let slice = this.chart.selectAll('.slice')
  .data(pie)
  .enter()
  .append('g')
  .attr('transform', 'translate(300, 300)');
```

To make positioning simpler, we move every group to the center of the pie chart. Creating slices works the same as in *Chapter 2, A Primer on DOM, SVG, and CSS*:

```
slice.append('path')
  .attr({
    d: arc,
    fill: (d) => helpers.color(d.data[0].DonorName)
 });
```

We get the color for a slice with `d.data[0].DonorName`. The original dataset is in `.data`, and all the `.DonorName` properties in it are the same. That's what we grouped by.

Labeling your pie chart

Labels take a bit more work. They need to be rotated into place and sometimes flipped so that they don't appear upside down. Labeling an arc will be handy later as well, so let's make a general function in `helpers.js`:

```
export function arcLabels(text, radius) {
  return function (selection) {
    selection.append('text')
      .text(text)
      .attr('text-anchor', (d) =>
        tickAngle(d) > 100 ? 'end' : 'start')
      .attr('transform', (d) => {
        let degrees = tickAngle(d);
        let turn = `rotate(${degrees})`;
        translate(${radius(d) + 10}, 0)`;
        if (degrees > 100) {
          turn += `rotate(180)`;
        }
        return turn;
      });
  }
}
```

Here, we're creating a "factory"-style function that generates another function operating on a D3 selection. This means we can use it with `.call()` while still defining our own parameters.

We'll give `arcLabels` a `text` accessor and a `radius` accessor, and it will return a function that we can use with `.call()` on a selection to make labels appear in just the right places. The meaty part appends a text element, tweaks its `text-anchor` element depending on whether or not we're going to flip it, and rotates the element to a particular position with the help of a `tickAngle` function.

Let's add the contents of the `tickAngle` function:

```
export function tickAngle (d) {
  let midAngle = (d.endAngle - d.startAngle) / 2;
  let degrees = (midAngle + d.startAngle) / Math.PI * 180 - 90;
  return degrees;
}
```

The `helpers.tickAngle` calculates the middle angle between `d.startAngle` and `d.endAngle` and transforms the result from radians to degrees so that SVG can understand it.

This is basic trigonometry, so I won't go into details, but your favorite high schooler should be able to explain the math.

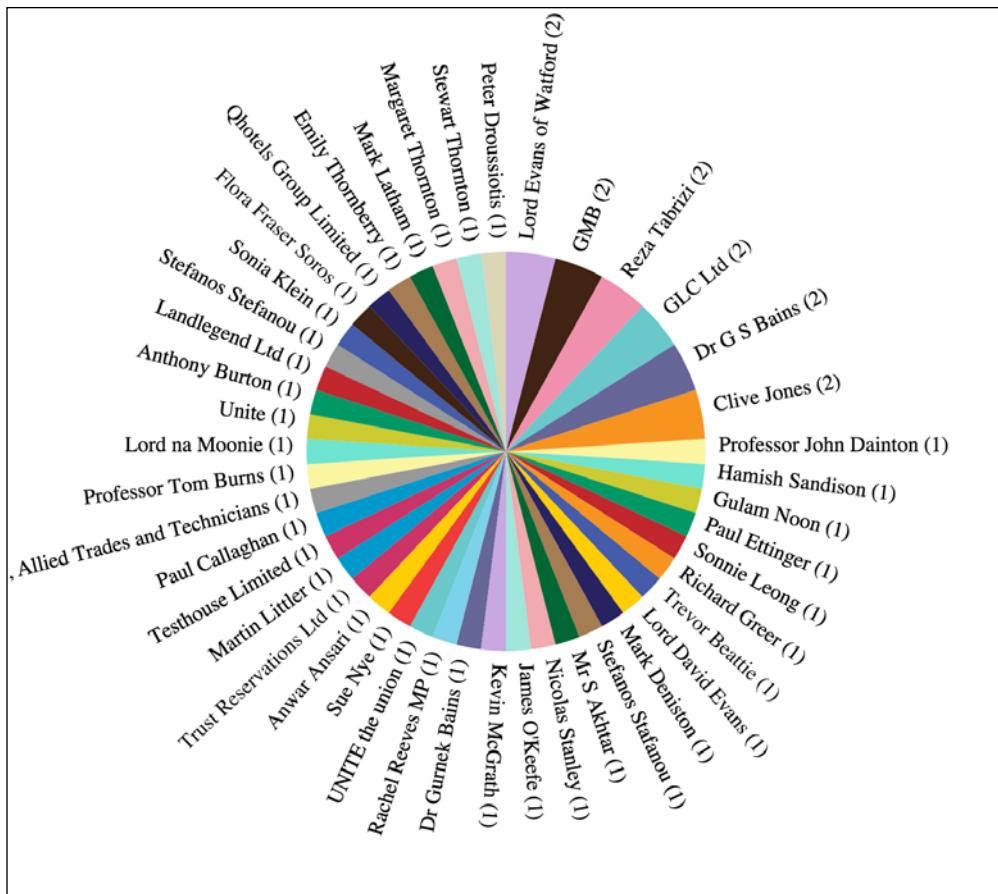
We use `arcLabels` back in the `pie` method:

```
slice.call(helpers.arcLabels((d) =>
` ${d.data[0].DonorName} (${d.value})`, arc.outerRadius()));
```

Lastly, we initialize the constructor in `index.js`:

```
new PoliticalDonorChart('pie', 'The Rt Hon Edward Miliband MP');
```

And our delicious pie is done, as shown in the following screenshot:



Showing popularity through time with stack

D3's official docs say:

"The stack layout takes a two-dimensional array of data and computes a baseline; the baseline is then propagated to the above layers, so as to produce a stacked graph."

Not clear at all, but I am hard pressed to come up with anything better. The `stack` layout calculates where one layer ends and another begins. An example should help.

We're going to make a layered timeline of donations over the 5-year sample, stretching as far back as 2010, with the width of each layer telling us how many donations went where at a certain time. This timeline is called a **streamgraph**.

To label layers, we're going to create a mouseover behavior that highlights a layer and shows a tooltip with the donation recipient's name.

Let's begin the binning. Create a new method called `streamgraph` in `PoliticalDonorChart`:

```
streamgraph() {
  let time = d3.time.format('%d/%m/%y');
  let data = this.data;
  let extent = d3.extent(data.map((d) =>
    time.parse(d.ReceivedDate)));
  let timeBins = d3.time.days(extent[0], extent[1], 12);
}
```

To parse timestamps into `Date` objects, we specified a format for strings such as `27/04/15`. Then, we used this format to find the earliest and latest time with `d3.extent`. Telling `d3.time.days()` to go from start to finish with a step of 14 days creates a list of bins.

We use the `histogram` layout to munge our dataset into a more useful form:

```
let perName = helpers.binPerName(data, (d) => d.EntityName);
let timeBinned = perName.map((nameLayer) => {
  return {
    to: nameLayer[0].EntityName,
    values: d3.layout.histogram()
      .bins(timeBins)
      .value((d) => time.parse(d.ReceivedDate))(nameLayer)
  }
});
```

You already know what `helpers.binPerName` does.

To bin data into time slots, we mapped through each layer of the `name` accessors and turned it into a two-property object. The `.to` property tells us whom the layer represents, and `.values` is a histogram of time slots where entries tell us how much karma the user many donations somebody got in a certain 12-day period.

Time for a `stack` layout:

```
let layers = d3.layout.stack()  
  .order('inside-out')  
  .offset('wiggle')  
  .values((d) => d.values)(timeBinned);
```

The `d3.layout.stack()` creates a new `stack` layout. We told it how to order layers with `.order('inside-out')` (you should also try `default` and `reverse`) and decided how the final graph looks with `.offset('wiggle')`. The `wiggle` minimizes changes in slope. Other options include `silhouette`, `zero`, and `expand`. Try them!

Once again, we told the layout how to find values with the `.values()` accessor. Our `layers` array is now filled with objects like this one:

```
{to: "The Rt Hon Edward Miliband MP",  
 values: Array[50]}
```

The `values` is an array of arrays. Entries in the outer array are time bins that look like this:

```
{dx: 1036800000,  
 length: 1,  
 x: Object(Thu Oct 13 2011 00:00:00 GMT+0200 (CEST)),  
 y: 1,  
 y0: 140.16810522517937}
```

The important parts of this array are as follows: `x` is the horizontal position, `y` is the thickness, and `y0` is the baseline. The `d3.layout.stack` will always return these.

To start drawing, we need some margins and two scales:

```
this.margin = {  
  top: 220,  
  right: 50,  
  bottom: 0,  
  left: 50  
};  
  
let x = d3.time.scale()
```

```
.domain(extent)
.range([this.margin.left, this.width - this.margin.right]);  
  
let y = d3.scale.linear()
.domain([0, d3.max(layers,
  (layer) => d3.max(layer.values, (d) => d.y0 + d.y))])
.range([this.height - this.margin.top, 0]);
```

The tricky thing was finding the vertical scale's domain. We found it by going through each value of every layer, looking for the maximum `d.y0+d.y` value—baseline plus thickness.

We'll use an area path generator for the layers:

```
let offset = 100;
let area = d3.svg.area()
.x((d) => x(d.x))
.y0((d) => y(d.y0) + offset)
.y1((d) => y(d.y0 + d.y) + offset);
```

Nothing too fancy. The baselines define bottom edges and adding the thickness gives the top edge. Fiddling determined that both should be pushed down by 100 pixels.

Let's draw an axis first:

```
let xAxis = d3.svg.axis()
.scale(x)
.tickFormat(d3.time.format('%b %Y'))
.ticks(d3.time.months, 2)
.orient('bottom');  
  
this.chart.append('g')
.attr('transform', `translate(0, ${this.height - 100})`)
classed('axis', true)
.call(xAxis)
.selectAll('text')
.attr('y', 5)
.attr('x', 9)
.attr('dy', '.35em')
.attr('transform', 'rotate(60)')
.style('text-anchor', 'start');
```

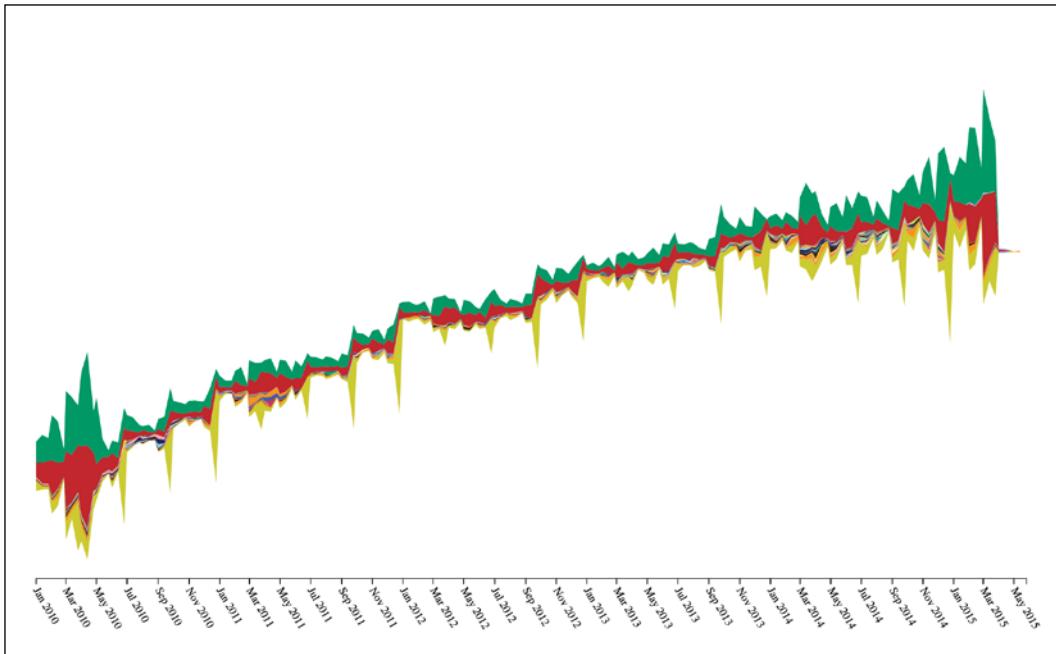
Same as usual—we defined an axis, called it on a selection, and let D3 do its thing. We made it more readable with a custom `.tickFormat()` function and used `.ticks()` to say that we want a new tick every 2 months.

Okay, now for the streamgraph, add the following code:

```
this.chart.selectAll('path')
  .data(layers)
  .enter()
  .append('path')
  .attr('d', (d) => area(d.values))
  .style('fill', (d, i) => helpers.color(i))
  .call(helpers.tooltip((d) => d.to, this.chart));
```

Not much is going on. We used the `area` generator to draw each layer, defined colors with `helpers.color`, and called a `tooltip` function, which we'll define in `helpers.js` later.

The streamgraph looks like this:



Adding tooltips to our streamgraph

It looks pretty, but it is useless. Let's add that `tooltip` function to the `helpers.js` tooltip:

```
export function tooltip (text, chart) {
  return function (selection) {
    selection.on('mouseover.tooltip', mouseover)
```

```
    .on('mousemove.tooltip', mousemove)
    .on('mouseout.tooltip', mouseout);
}
}
```

We defined event listeners with a `.tooltip` namespace so that we can define multiple listeners on the same events.

The `mouseover` function will highlight streams and create tooltips, `mousemove` will move tooltips, and `mouseout` will put everything back to normal.

Let's put the three listeners inside the inner function of our tooltip helper:

```
function mouseover(d) {
  let path = d3.select(this);
  path.classed('highlighted', true);
}
```

That's the simple part of `mouseover`. It selects the current area and changes its class to `highlighted`. This will make it lighter and add a red outline.

In the same function, add the meaty part:

```
let mouse = d3.mouse(chart.node());
let tool = chart.append('g')
  .attr({
    'id': 'nameTooltip',
    'transform': `translate(${mouse[0] + 5}, ${mouse[1] + 10})`});
}

let textNode = tool.append('text')
  .text(text(d)).node();

tool.append('rect')
  .attr({
    height: textNode.getBBox().height,
    width: textNode.getBBox().width,
    transform: 'translate(0, -16)'
});

tool.select('text')
  .remove();

tool.append('text').text(text(d));
```

It is longer and with a dash of magic, but not scary at all!

First, we find the mouse's position. Then we create a group element and position it down and to the right of the mouse. We add a text element to the group and call SVG's `getBBox()` function on its node. This gives us the text element's bounding box and helps us size the background rectangle.

Finally, we remove the text, because it's covered by the background, and add it again. We might be able to avoid all this trouble by using HTML *divs*, but I wanted to show you pure SVG tooltips. Hence, consider the following code:

```
function mousemove () {
    let mouse = d3.mouse(chart.node());
    d3.select('#nameTooltip')
        .attr('transform', `translate(${mouse[0] + 15},
        ${mouse[1] + 20})`);
```

The `mousemove` listener in the following code is much simpler. It just finds the `#nameTooltip` element and moves it to follow the cursor:

```
functionmouseout () {
    let path = d3.select(this);
    path.classed('highlighted', false);
    d3.select('#nameTooltip').remove();
```

The `mouseout` function selects the current path, removes its `highlighted` styling, and removes the tooltip.

Voilà! Tooltips!

Very rudimentary – they don't understand edges and they won't break any hearts with their looks, but they get the job done. Let's add some CSS to `chapter5.css`:

```
#nameTooltip {
    font-size: 1.3em;
}

#nameTooltip rect {
    fill: white;
}

#nameTooltip text {
    fill: #000;
    stroke: #000;
    color: #000;
}
```

```
path.highlighted {
  fill-opacity: 0.5;
  stroke: red;
  stroke-width: 1.5;
}
```

And suddenly, we have a potentially useful streamgraph on our hands!

Highlighting connections with chord

We've seen how many donations people have and when they got it, but there's another gem hiding in the data—connections. We can visualize who is donating to whom using the `chord` layout.

We're going to draw a chord diagram—a circular diagram of connections. Chord diagrams are often used in genetics and have even appeared on covers of magazines (http://circos.ca/intro/published_images/).

Ours is going to have an outer ring showing how much money is being donated and chords showing where that money is going.

First, we need a matrix of connections for the chord diagram, and then we'll go the familiar route of path generators and adding elements. The matrix code will be useful later, so let's put it in `helpers.js`:

```
export function connectionMatrix (data) {
  let nameIds = nameId(allUniqueNames(data), (d) => d);
  let uniques = nameIds.domain();
  let matrix = d3.range(uniques.length).map(
    () => d3.range(uniques.length).map(() => 0));
  data.forEach((d) => {
    matrix[nameIds(d.DonorName)][nameIds(d.EntityName)] +=
      Number(d.Value.replace(/[^\\d\\.]*$/g, ''));
  });

  return matrix;
}
```

Let's also create a function that returns unique names:

```
export function allUniqueNames (data) {
  let donors = uniques(data, (d) => d.DonorName);
  let donees = uniques(data, (d) => d.EntityName);
  return uniques(donors.concat(donees), (d) => d);
}
```

We begin with the familiar `uniques` list and the `nameId` scale. Then we create a zero matrix and loop through the data to increment by the value of each donation (which we quickly clean up by removing the pound symbol and the comma before changing it to a `Number`). Rows are *from whom*, columns are *to whom*. For example, if the fifth cell in the first row holds 10, it means the first person or organization has given £10 to the fifth person. This is called an **adjacency matrix**.

Meanwhile, back in `PoliticalDonorChart`, create a `chord` method:

```
chord(filterString) {
  let filtered = this.data.filter((d) =>
    d.EntityName.match(filterString || ' MP') );
  let uniques = helpers.uniques(filtered, (d) => d.DonorName);
  let matrix = helpers.connectionMatrix(filtered);
}
```

We create the matrix from our data, which we've filtered for individual MPs. We can also provide an argument in the constructor to filter based on it. Moreover, we create another array of unique names, this time for MPs. This allows us to have something to connect to.

We're going to need `uniques` for labels, and it would be nice to have the `innerRadius` and `outerRadius` variables handy:

```
let innerRadius = Math.min(this.width, this.height) * 0.3;
let outerRadius = innerRadius * 1.1;
```

Time to make the `chord` layout do our bidding:

```
let chord = d3.layout.chord()
  .padding(.05)
  .sortGroups(d3.descending)
  .sortSubgroups(d3.descending)
  .sortChords(d3.descending)
  .matrix(matrix);
```

It is a little different from others. The `chord` layout takes data via the `.matrix()` method and can't be called as a function.

We started with `d3.layout.chord()` and put a `.padding()` method between groups, which improves readability. To improve readability further, everything is sorted. The `.sortGroups` sorts groups on the edge, `.sortSubgroups` sorts chord attachments in groups, and `.sortChords` sorts the chord drawing order so that smaller chords overlap bigger ones.

In the end, we feed data into the layout with `.matrix()`:

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

We add a centered group element so that all our coordinates are relative to the center from now on.

The drawing of the diagram happens in three steps – arcs, labels, and chords – as shown in the following code:

```
let group = diagram.selectAll('.group')
  .data(chord.groups)
  .enter().append('g');

let arc = d3.svg.arc()
  .innerRadius(innerRadius)
  .outerRadius(outerRadius);

group.append('path')
  .attr('d', arc)
  .attr('fill', (d) => helpers.color(d.index));
```

This creates the outer ring. We use `chord.groups` to get group data from the layout, create a new grouping element for every chord group, and then add an arc. We use `arc_labels` from the pie example to add the labels:

```
group.call(helpers.arcLabels(
  (d) => uniques[d.index], () => outerRadius + 10));
```

Even though the radius is constant, we have to define it as a function using the following code because we didn't make `arcLabels` flexible enough for constants. Nobody ain't got time for that, though – we still have seven charts to make in this chapter!

```
diagram.append('g')
  .classed('chord', true)
  .selectAll('path')
  .data(chord.chords)
  .enter()
  .append('path')
  .attr('d', d3.svg.chord().radius(innerRadius))
  .attr('fill', (d) => helpers.color(d.target.index));
```

We get chord data from `chord.chords` and use a chord path generator to draw the chords. We pick colors with `d.target.index` because the graph looks better, but chord colors are *not* informative.

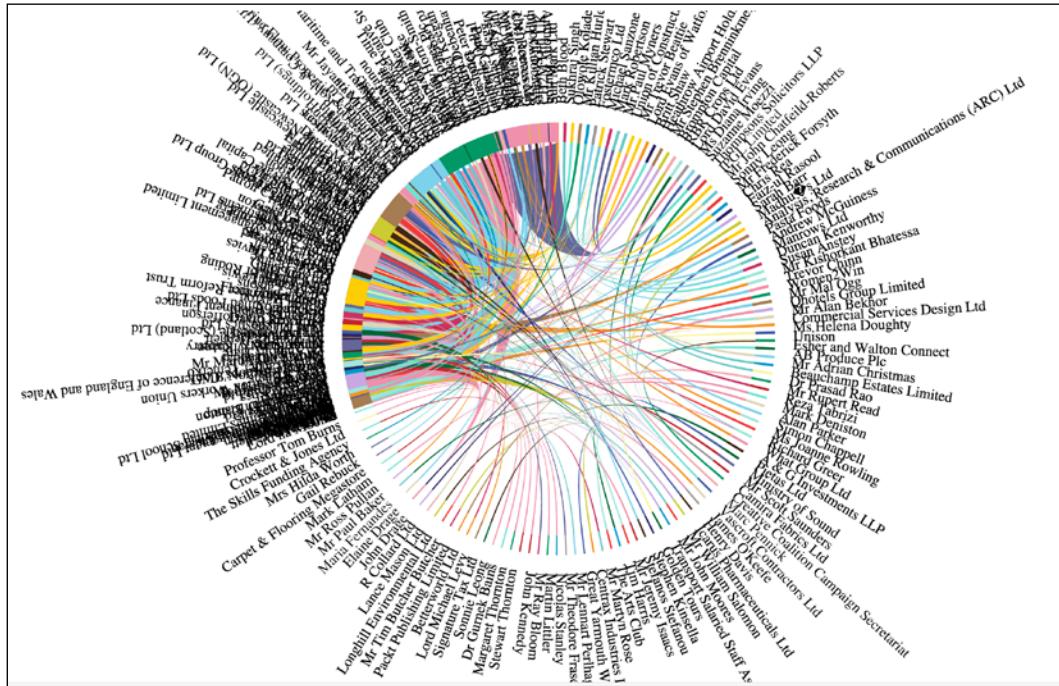
We add some CSS to `chapter5.css` to make the chords easier to follow:

```
.chord path {  
    stroke: black;  
    stroke-width: 0.2;  
    opacity: 0.6;  
}
```

Finally, replace the new call in `index.js` with the following:

```
new PoliticalDonorChart('chord');
```

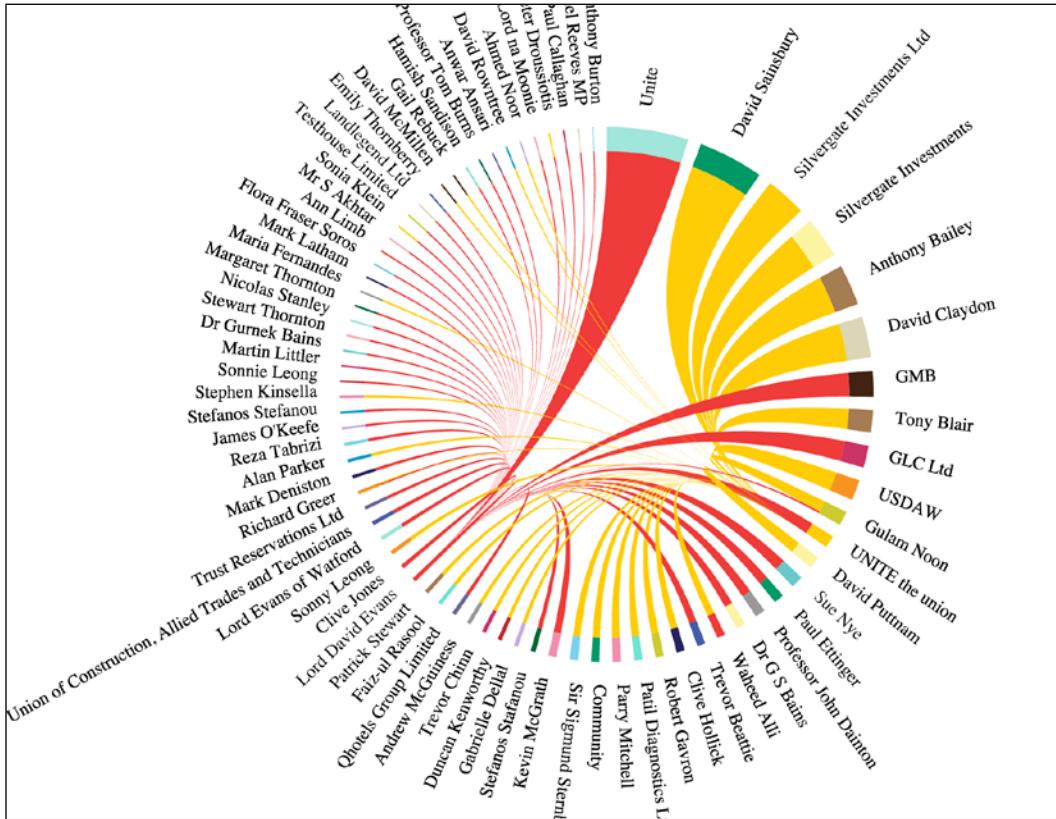
And now we have a very busy and hard-to-read diagram!



Pretty cool! If only it was more readable! Change the constructor to the following:

```
new PoliticalDonorChart('chord', 'Miliband');
```

This will get us Ed Miliband and his brother, David Miliband, both Labour MPs:



First of all, chord colors don't mean anything! They just make it easier to distinguish chords. Furthermore, this graph shows how big each donation is by the size of the chord.

A chord chart is suboptimal for this type of data because donations are pretty unidirectional—MPs typically don't give money back to donors. In the preceding chart, you see your arcs kind of tapering towards nowhere; this is because we don't have an item on the chart for where all the donations are going. Fixing it's a bit hacky, but easy enough. We put this code block before the line where we define `uniques` in our chord method:

```
let uniqueMPs = helpers.uniques(filtered, (d) =>
  d.EntityName);
uniqueMPs.forEach((v) => {
  filtered.push({
    DonorName: v,
    EntityName: v,
    Value: '9001'
  });
});
```

This just adds another entry and gives it a value that's over nine thousand (nine thousand?!), which is just an arbitrary number I chose that gives the entries a bit of size but not enough to distort the graphic. Feel free to play around with it.

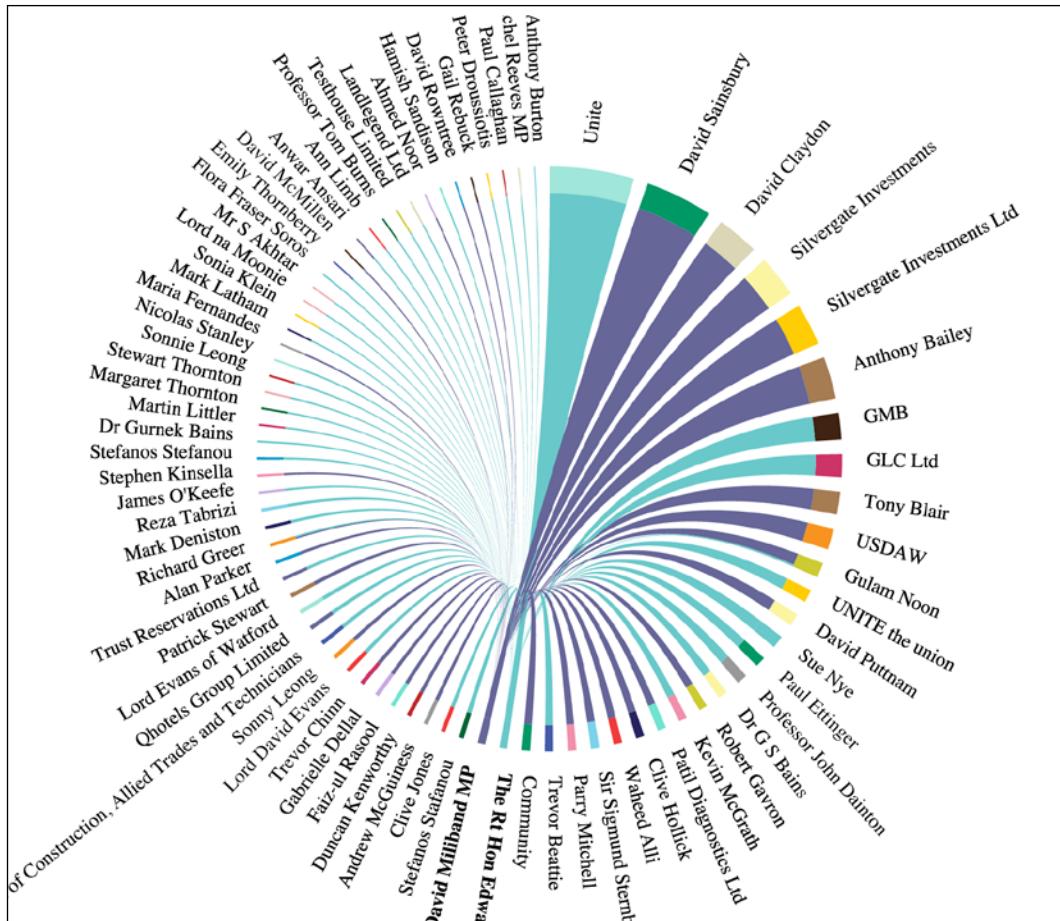
Lastly, add the following code at the bottom of `chord` to make the entries you just added bold:

```
this.chart.selectAll('text').filter(function(d) {
  return d3.select(this).text().match(filterString))
  .style('font-weight', 'bold');
```

There we go! It makes slightly more sense now! From this, you can tell that *Unite* (Britain's largest union, for those outside the UK) gave a lot of money to Ed Miliband's campaign, and *Silvergate Investments* gave a lot to David Miliband. You'll notice that both are listed twice; clearly this data needs some cleanup. As you build things in D3, you'll often find situations like this. You can do either one of two things: handle all these edge cases in your code, or clean up the data using something like OpenRefine. It's much better to clean up messy data before it gets to D3 as it results in cleaner and easier-to-read code with less hacky conditional logic.



For a good tutorial on using OpenRefine, visit <http://schoolofdata.org/handbook/recipes/cleaning-data-with-refine/>.



Drawing with force

The force layout is the most complicated of the non-hierarchical layouts. It lets you draw complex graphs using physical simulations—force-directed graphs if you will. Everything you draw will have built-in animation.

We're going to draw a graph of connections between donors and politicians. Every donor and politician will be a node, the size of which will correspond to the user's total donations. Links between nodes will tell us who is donating to whom.

To make things clearer, we're going to add tooltips and make sure that hovering over a node highlights the connected nodes.

Let's begin!

As in the chord example, we begin with a matrix of connections. We aren't going to feed this directly to the force layout, but we will use it to create the kind of data it enjoys. Start by changing the constructor in `index.js` to the following:

```
new PoliticalDonorChart('force');
```

Then add a `force` method to your chart class:

```
force(filterString = ' MP ') {
  let filtered = this.data.filter(
    (d) => d.EntityName.match(filterString));
  let nameId = helpers.nameId(
    helpers.allUniqueNames(filtered), (d) => d);
  let uniques = helpers.allUniqueNames(filtered);
  let matrix = helpers.connectionMatrix(filtered);
}
```

D3's force layout expects an array of nodes and links. Let's make them:

```
let nodes = uniques.map((name) => new Object({
  name: name,
  totalDonated: matrix[nameId(name)].reduce(
    (last, curr) => last + curr, 0),
  totalReceived: matrix.reduce((last, curr) =>
    last + curr[nameId(name)], 0)
}));

let links = filtered.map((d) => {
  return {
    source: nameId(d.DonorName),
    sourceName: d.DonorName,
    target: nameId(d.EntityName),
    targetName: d.EntityName,
    amountDonated:
      matrix[nameId(d.DonorName)][nameId(d.EntityName)]
  }
});
```

We're defining the bare minimum of what we need, and the layout will calculate all the hard stuff.

The nodes tell us who they represent, and links connect a source object to a target object with an index into the nodes array. The layout will turn them into proper references, as shown in the following code. Every link also contains a count object that we'll use to define its strength:

```
let force = d3.layout.force()
  .nodes(nodes)
```

```
.links(links)
  .charge((node) => node.totalDonated ? -50 : 0)
  .gravity(0.05)
  .size([this.width, this.height]);

force.start();
```

We create a new force layout with `d3.layout.force()`; just like the `chord` layout, it isn't a function either. We feed in the data with `.nodes()` and `.links()`. The `charge` is set so that nodes that are donators repel each other, which will help prevent everything from just clumping in the center. The `gravity` setting pulls the graph towards the center of the image; we defined its strength with `.gravity()`. We tell the force layout the size of our picture with `.size()`. No calculation happens until `force.start()` is called, but we need the results to define a few scales for later.

There are a few more parameters to play with: the overall `.friction()` (the smallest `.linkDistance()` value the nodes stabilize to) and `.linkStrength()` for link stretchiness. Play with them. The `nodes` members now look like this:

```
{index: 0,
  name: "The Rt Hon Edward Miliband MP",
  px: 497.0100389553633,
  py: 633.2734045531992,
  weight: 100,
  x: 499.5873097327753,
  y: 633.395804766377}
```

The `weight` tells us how many links connect with this node, `px` and `py` state its previous position, and `x` and `y` state the current position.

The `links` members are a lot simpler:

```
{count: 2
  source: Object
  target: Object}
```

Both the `source` and `target` objects are a direct reference to the correct node. Now that the layout has made its first calculation step, we have the data needed to define some scales:

```
let distance = d3.scale.linear()
  .domain(d3.extent(d3.merge(matrix)))
  .range([300, 100]);

let given = d3.scale.linear()
  .domain(d3.extent(matrix, (d) => d3.max(d)))
  .range([2, 35]);
```

We're going to use the given scale for node sizes and distance for link lengths. Nodes that either receive or donate more will appear bigger, and nodes representing donors will be closer to the nodes they donated to based on the donated amount:

```
force.linkDistance(d => distance(d.amountDonated));  
force.start();
```

We use `.linkDistance()` to dynamically define link lengths according to the `.count` property. To put the change in effect, we restart the layout with `force.start()`.

Finally! Time to put some ink on paper—well—pixels on screen!

```
let link = this.chart.selectAll('line')  
  .data(links)  
  .enter()  
  .append('line')  
  .classed('link', true);
```

Links are simple. Go through the list of links and draw a line.

Next, draw a circle for every donor node and a square for every recipient:

```
let node = this.chart.selectAll('.node')  
  .data(nodes)  
  .enter()  
  .append((d) => {  
    return document.createElementNS(  
      'http://www.w3.org/2000/svg', d.totalDonated > 0 ?  
      'circle' : 'rect');  
  })  
  .classed('node', true);  
  
this.chart.selectAll('circle.node')  
  .attr({  
    r: (d) => given(d.totalDonated),  
    fill: (d) => helpers.color(d.index),  
    class: (d) => 'name_' + nameId(d.name)  
  })  
  .classed('node', true);  
  
this.chart.selectAll('rect.node')  
  .attr({  
    width: (d) => given(d.totalReceived),  
    height: (d) => given(d.totalReceived),  
    fill: (d) => helpers.color(d.index),  
    class: (d) => 'name_' + nameId(d.name)  
  })  
  .classed('node', true);
```

The only unusual bit in the preceding code is us using a callback for the `.append` method, which requires a new element to be returned. Because we aren't able to use `.append` itself when we override its accessor (since we're conditionally making the element a circle or square depending on whether the node is a donor or recipient of a donation), we have to set the namespace and everything...

We add tooltips with the familiar `helpers.tooltip` function, and `force.drag` will automatically make the nodes draggable:

```
node.call(helpers.tooltip((d) => d.name, this.chart));
node.call(force.drag);
```

After all that work, we still have to do the updating on every tick of the `force` layout animation:

```
force.on('tick', () => {
  link.attr('x1', (d) => d.source.x)
    .attr('y1', (d) => d.source.y)
    .attr('x2', (d) => d.target.x)
    .attr('y2', (d) => d.target.y);

  this.chart.selectAll('circle.node').attr('cx', (d) => d.x)
    .attr('cy', (d) => d.y);

  this.chart.selectAll('rect.node').attr('x', function(d) {
    return d.x - this.getBBox().width / 2;
  })
    .attr('y', function(d) {
      return d.y - this.getBBox().height / 2
    });
});
```

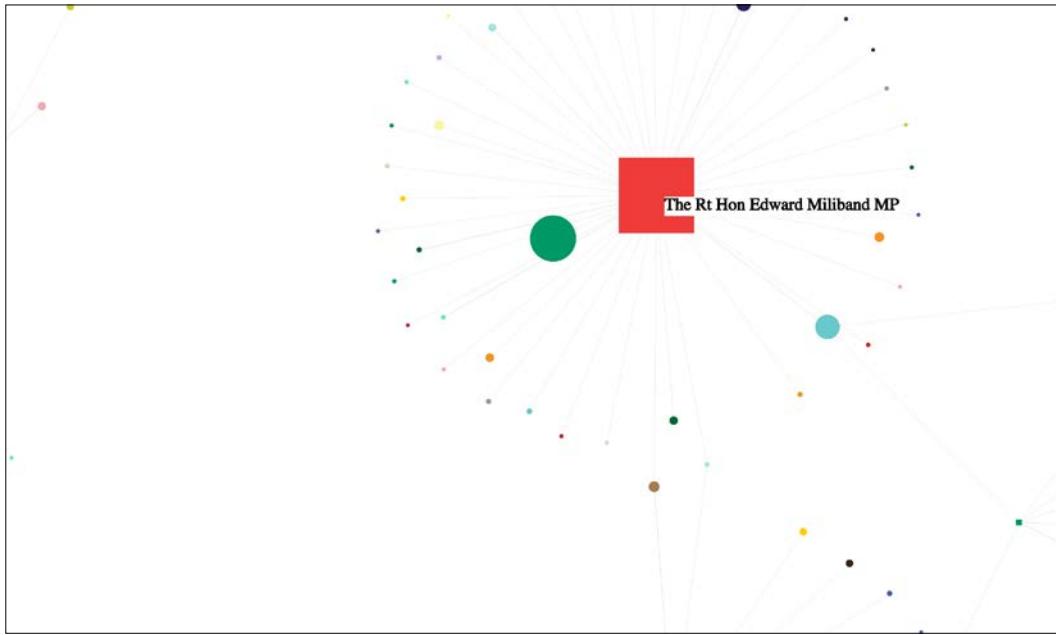
On a `tick` event, we move every `link` endpoint and `node` to its new position. Simple!

Add some styling to `chapter5.css`:

```
.link {
  stroke: lightgrey;
  stroke-width: 0.3;
}
```

And voilà! We get a force-directed graph of election donations.

Running this example looks silly because it spins around a lot before settling down. But once it stabilizes, the graph looks something like this:



Drag and pull it around the screen to see different connections.

We should have added some code to print names next to the highlighted nodes, but the example was long enough. Let's say that's left as an exercise for the reader.

We will now move on to hierarchical layouts!

Hierarchical layouts

All hierarchical layouts are based on an abstract hierarchy layout designed for representing hierarchical data – data within data within data within data within.... all the way down. As mentioned earlier, imagine a tree, or an org chart if you don't go outside very much.

All of the common code for the `partition`, `tree`, `cluster`, `pack`, and `treemap` layouts is defined in `d3.layout.hierarchy()`, and they all follow similar design patterns. The layouts are so similar that the official documentation very obviously copy-pastes most of its explanations which are practically identical. Let's avoid that by looking at the common stuff first, and then we will focus on the differences.

First of all, we need some hierarchical data. There's an implicit hierarchy in our data, insomuch that donations flow downwards, but it's not as well-defined as it would be for, say, organizational units of most businesses or the filesystem tree of a hard drive. The result is a scheme that works well with three of the layouts but looks slightly contrived for the other two—apologies in advance for that, but hopefully, it gives you the right idea.

We'll have a root node called `donations`, which will contain the names of people or groups that have donated in the last election cycle. For the `tree` and `cluster` layouts, each of those will contain nodes for everyone they have donated to. For the `partition`, `pack`, and `treemap` layouts, child nodes will tell us who contributed to the parent's total received donations.

The final data structure will look like this:

```
{
  "name": "donations",
  "children": [
    {
      "name": "Unite",
      "count": 5,
      "children": [
        {
          "name": "Ed Balls",
          "count": 2,
          "children": []
        },
        {
          "name": "Ed Miliband",
          "count": 6,
          "children": []
        }
      ]
    }
  ]
}
```

While it could potentially go on forever, that wouldn't make sense in our case. We have only three levels in this example, but you can go down as many levels as you like.

The default accessor expects a `.children` property, but we could have easily done something crazy, such as dynamically generating a fractal structure in a custom accessor.

As usual, there's a `.value()` accessor that helps layouts to find data in a node. We'll use it to set the amount either donated or received.

To run a hierarchical layout, we call `.nodes()` with our dataset. This immediately returns a list of nodes that you can't get to later. For a list of connections, we call `.links()` with a list of our nodes. Nodes in the returned list will have some extra properties calculated by the layout. Most layouts tell us where to put something with `.x` and `.y`, and then use `.dx` and `.dy` to tell us how big the layout should be.

All hierarchical layouts also support sorting with `.sort()`, which takes a sorting function, such as `d3.ascend`ing or `d3.descend`ing.

Enough of theory! Let's add a data munging function to `helpers.js`:

```
export function makeTree(data, filterByDonor, name1, name2) {
  let tree = {name: 'Donations', children: []};
  let uniqueNames = uniques(data, (d) => d.DonorName)

  tree.children = uniqueNames.map((name) => {
    let donatedTo = data.filter((d) => filterByDonor(d, name));
    let donationsValue = donatedTo.reduce((last, curr) => {
      let value = Number(curr.Value.replace(/[^\\d\\.]*/g, '')) ;
      return value ? last + value : last;
    }, 0);

    return {
      name: name,
      donated: donationsValue,
      children: donatedTo.map((d) => {
        return {
          name: name2(d),
          count: 0,
          children: []
        };
      })
    };
  });

  return tree;
}
```

Wow, there's a lot going on here! We avoided recursion because we know our data will never nest more than two levels deep.

The tree holds an empty root node at first. We use `helpers.uniques` to get a list of names. Then we map through the array and define the children of the root node by counting everyone's donations and using `helpers.binPerName` to get an array of children.

The code is wibbly-wobbly because we use `filterByDonor`, `name1`, and `name2` for data accessors, but making this function flexible makes it useful in all hierarchical examples.

Drawing a tree

The tree layout displays data in a tree using the Reingold-Tilford tidy algorithm. We'll use it to display our dataset in a large circular tree, with every node connected to its parent by a curvy line.

We begin the method by fixating colors, turning data into a tree, and defining a way to draw curvy lines:

```
tree(filterString = ' MP') {
  let filtered = this.data.filter(
    (d) => d.EntityName.match(filterString) );
  helpers.fixateColors(filtered);

  let tree = helpers.makeTree(filtered,
    (d, name) => d.DonorName === name,
    (d) => d.EntityName,
    (d) => d.EntityName || '');

  tree.children = tree.children.filter(
    (d) => d.children.length > 1)

  let diagonal = d3.svg.diagonal.radial()
    .projection((d) => [d.y, d.x / 180 * Math.PI]);
}
```

You know `fixateColors` from before. We defined `makeTree` about one page ago, and we talked about the `diagonal` generator in *Chapter 2, A Primer on DOM, SVG, and CSS*:

```
let layout = d3.layout.tree()
  .size([360, this.width / 5]);
let nodes = layout.nodes(tree);
let links = layout.links(nodes);
```

We create a new tree layout by calling `d3.layout.tree()`. Defining its size with `.size()` and executing it with `.nodes().size()` tells the layout how much room it's got—in this case, we're using `x` as an angle (360 degrees) and `y` as a radius, though the layout itself doesn't really care about that.

To avoid worrying about centering later on, we put a grouping element center stage:

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

First, we are going to draw the links, and then the nodes and their labels:

```
let link = chart.selectAll('.link')
  .data(links)
  .enter()
  .append('path')
  .attr('class', 'link')
  .attr('d', diagonal);
```

You should be familiar with this by now; go through the data and append new paths shaped with the `diagonal` generator:

```
let node = chart.selectAll('.node')
  .data(nodes)
  .enter().append('g')
  .attr('class', 'node')
  .attr('transform',
    (d) => `rotate(${d.x - 90})translate(${d.y})`);
```

For every node in the data, we create a new grouping element and move it into place using `rotate` for angles and `translate` for radius positions.

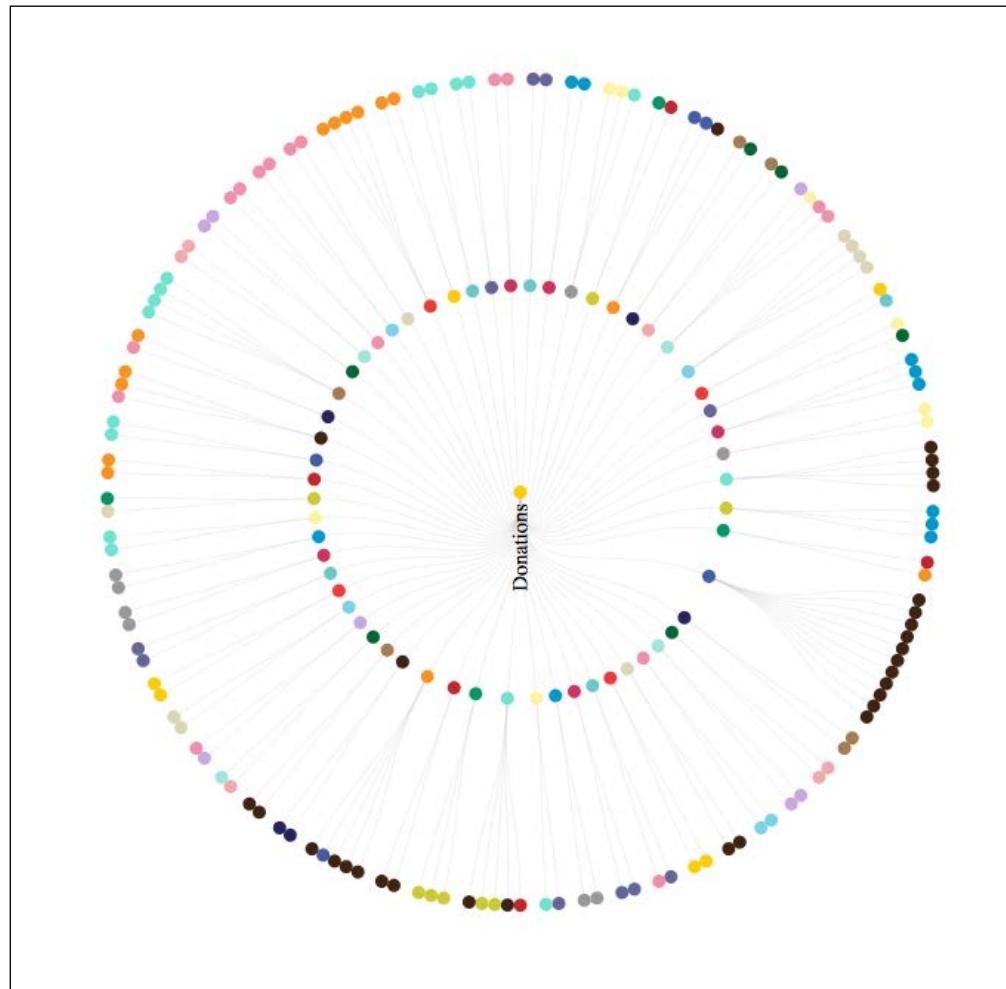
Now it's just a matter of adding a circle and a label:

```
node.append('circle')
  .attr('r', 4.5)
  .attr('fill', (d) => helpers.color(d.name))
  .on('mouseover', function(d) {
    d3.select(this.nextSibling).style('visibility', 'visible');
  })
  .on('mouseout', function(d) {
    d3.select(this.nextSibling).style('visibility', 'hidden');
  });
node.append('text')
  .attr('dy', '.31em')
  .attr('text-anchor', (d) => d.x < 180 ? 'start' : 'end')
```

```
.attr('transform', (d) => d.x < 180 ? 'translate(8)' :
'rotate(180)translate(-8)')
.text((d) => d.depth > 1 ? d.name : d.name.substr(0, 15) +
(d.name.length > 15 ? '...' : ''))
.style({
  'font-size': (d) => d.depth > 1 ? '0.6em' : '0.9em',
  'visibility': (d) => d.depth > 0 ? 'hidden' : 'visible'
});
```

Every node is colored with the user's native color, and the text is transformed similarly to the earlier pie and chord examples. Finally, we've made leaf nodes' text smaller to avoid overlap.

Our tree looks something like this:



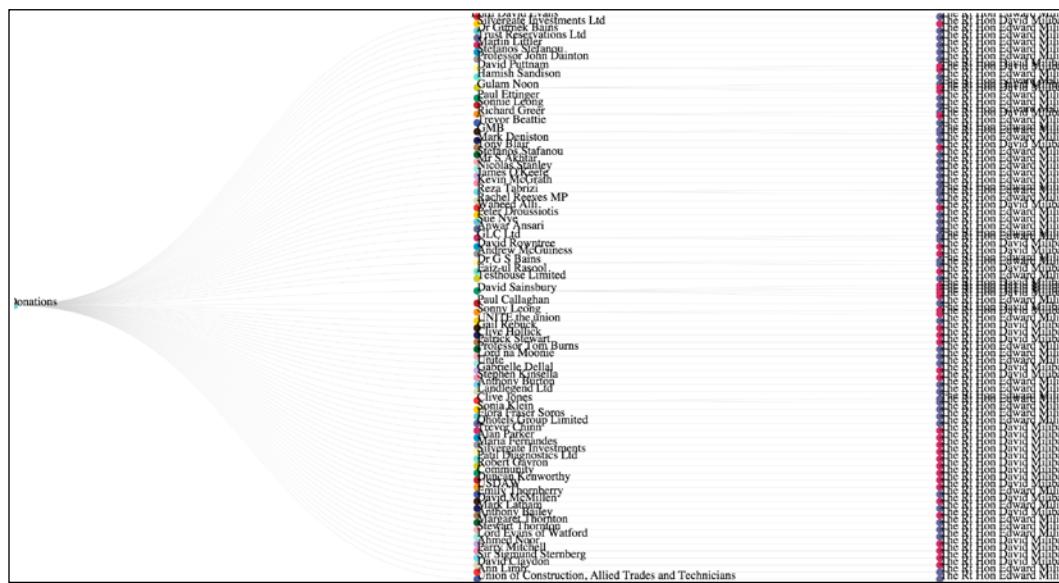
It's rather big, so you should try it out in the browser. Just remember that the inner ring represents users, people, or groups giving karma donations to the politicians in the outer ring.

Showing clusters

The cluster layout is the same as the tree layout, except that the leaf nodes line up.

Code-wise, this example is the same as the last, so we won't go through it again. Really, the only difference is that we don't have to flip labels at certain angles. You can look at the code in `chapter5.js` of the book's repo.

We end up with a very tall graph that looks something like this:



Partitioning a pie

Now we're getting somewhere! The next three layouts fit our data perfectly—we're taking three looks at how our core politicians' donations are structured.

The partition layout creates adjacency diagrams, where you don't draw nodes with links between them but next to each other so that it looks as if the children partition the parent.

We are going to draw a two-layer donut chart. Users will go on the first layer and the layer on top will show us where the donations are coming from.

We begin by munging the dataset and fixating colors; it's the same as before:

```
partition(filterString = ' MP') {
  let filtered = this.data.filter(
    (d) => d.EntityName.match(filterString) );
  let tree = helpers.makeTree(filtered,
    (d, name) => d.DonorName === name,
    (d) => d.EntityName,
    (d) => d.EntityName || '');

  helpers.fixateColors(filtered);
}
```

Then we use the `partition` layout:

```
let partition = d3.layout.partition()
  .value((d) => d.parent.donated)
  .sort((a, b) =>
    d3.descending(a.parent.donated, b.parent.donated))
  .size([2 * Math.PI, 300]);

let nodes = partition.nodes(tree);
```

We used `.value()` to tell the layout that we care about the `.donated` values, and we'll get a better picture if we `.sort()` the output. Similar to the `tree` layout, `x` will represent angles — this time in radians — and `y` will be radii.

We need an `arc` generator as well, as shown in the following code:

```
let arc = d3.svg.arc()
  .innerRadius((d) => d.y)
  .outerRadius((d) => d.depth ? d.y + d.dy / d.depth : 0);
```

The generator will use each node's `.y` property for the inner radius and add `.dy` for the outer radius. Fiddling shows that the outer layer should be thinner. Hence, we are dividing it by the tree depth.

Notice that there's no accessor for `.startAngle` and `.endAngle`, which are stored as `.x` and `.dx`. It's easier to just fix the data:

```
nodes = nodes.map((d) => {
  d.startAngle = d.x;
  d.endAngle = d.x + d.dx;
  return d;
});
nodes = nodes.filter((d) => d.depth);
```

It is as simple as mapping the data, defining angle properties, and then filtering the data to make sure that the root isn't drawn.

We use the familiar grouping trick to center our diagram:

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

The preparation work is done. It's drawing time!

```
let node = chart.selectAll('g')
  .data(nodes)
  .enter()
  .append('g');

node.append('path')
  .classed('partition', true)
  .attr({
    d: arc,
    fill: (d) => helpers.color(d.name)
  });
```

An arc is drawn for every node. The color is chosen as usual. Lastly, we add tooltips because adding labels is a bit too messy on this one:

```
node.call(helpers.tooltip(function (d) { return d.nick; }));
```

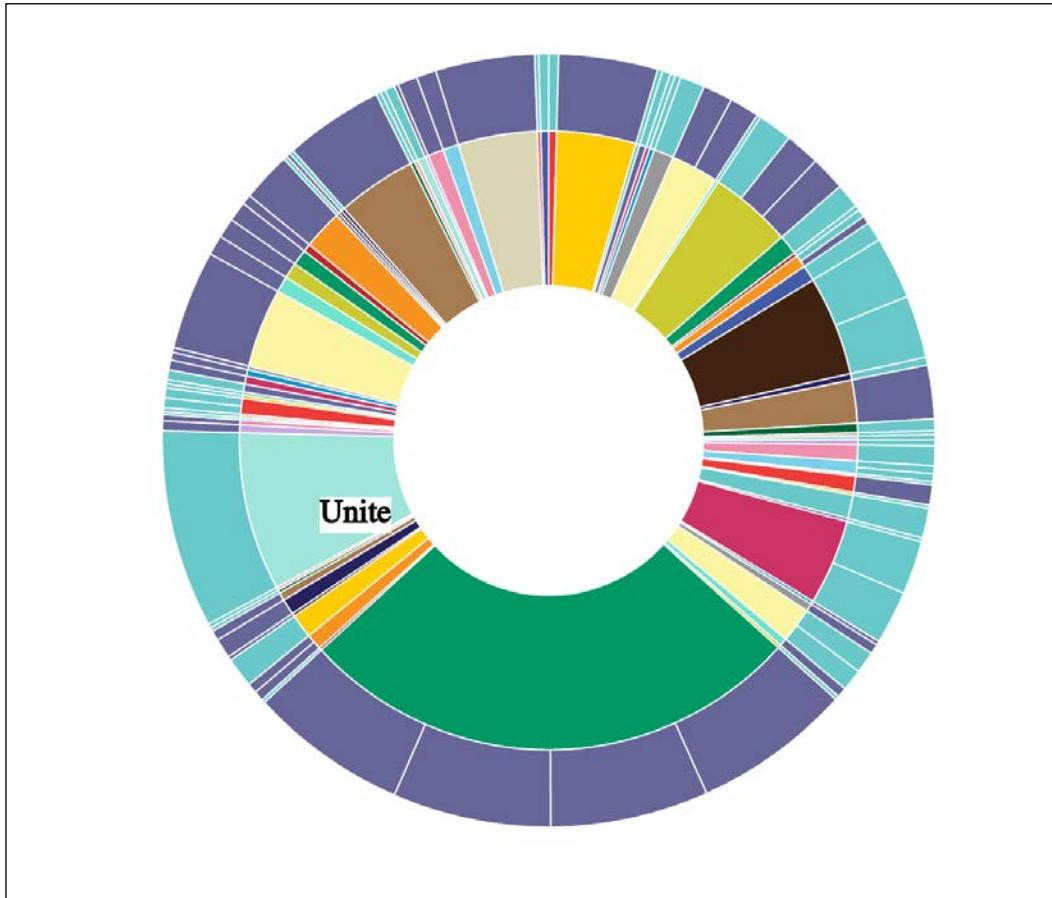
Add some more CSS to chapter5.css:

```
path.partition {
  stroke: white;
  stroke-width: 1;
}
```

Finally, instantiate in `index.js`. We're going to continue filtering by the Miliband brothers:

```
new PoliticalDonorChart('partition', 'Miliband');
```

The adjacency diagram looks like this:



The outer segments represent either Ed or David Miliband (depending on the color),
and the inner segments represent the donors

Packing it in

The pack layout uses packing to visually represent hierarchies. It stuffs children nodes into their parents, trying to conserve space and sizing each node so that it's the cumulative size of its children.

Layouts – D3's Black Magic

Conceptually, it's very similar to the `treemap` layout, so I'm going to skip all of the code and just show you the image. To see the code that generated it, check out the `pack` method in `chapter5.js`:



It looks nice, but the pack layout probably isn't the best way to visualize this particular dataset

The code is rather familiar – generate a tree, fixate the colors, create the layout, tweak a few parameters, get computed nodes, draw the nodes, and add tooltips. Simple!

It looks very pretty, but it's not too informative. Adding labels wouldn't help much either because most of the nodes are too small.

Subdividing with treemap

The treemap layout subdivides nodes with horizontal and vertical slices, essentially packing children into their parents, just like the pack layout, but using rectangles. As a result, node sizes on every level can be compared directly, making this one of the best layouts for analyzing cumulative effects of subdivisions.

We are going to have some fun with this example. Tooltips will name the parent—parents are almost completely obscured by the children—and moving the mouse arrow over a node will make unrelated nodes become lighter, making the graph less confusing (at least in theory).

It's also a cool effect and a great way to end this chapter on layouts. But we begin with the boring stuff—prepare data and fixate the colors:

```
treemap(filterString = ' MP') {
  let filtered = this.data.filter(
    (d) => d.EntityName.match(filterString) );
  let tree = helpers.makeTree(filtered,
    (d, name) => d.DonorName === name,
    (d) => d.EntityName,
    (d) => d.EntityName || '');

  helpers.fixateColors(filtered);
}
```

Creating the treemap layout follows familiar patterns:

```
let treemap = d3.layout.treemap()
  .size([this.width, this.height])
  .padding(3)
  .value((d) => d.parent.donated)
  .sort(d3.ascending);

let nodes = treemap.nodes(tree)
  .filter((d) => d.depth);
```

We added some padding with `.padding()` to give the nodes room to breathe. Every node will become a group element holding a rectangle. The leaves will also hold a label:

```
let node = this.chart.selectAll('g')
  .data(nodes)
  .enter()
  .append('g')
  .classed('node', true)
```

```
.attr('transform', (d) => `translate(${d.x},${d.y})`);  
  
node.append('rect')  
.attr({  
    width: (d) => d.dx,  
    height: (d) => d.dy,  
    fill: (d) => helpers.color(d.name)  
});
```

Now, for the first fun bit. Let's fit labels into as many nodes as they can possibly go:

```
let leaves = node.filter((d) => d.depth > 1);  
  
leaves.append('text')  
.text((d) => {  
    let name = d.name.match(/(^[\s]+[\s]+|^[\s]+)/ MP$/)  
    .shift().split(' ');\br/>    return `${name[0].substr(0, 1)}. ${name[1]}`;  
})  
.attr('text-anchor', 'middle')  
.attr('transform', function (d) {  
    let box = this.getBBox();  
    let transform = `translate(${d.dx / 2},  
    ${d.dy / 2 + box.height / 2})`;  
  
    if (d.dx < box.width &&  
        d.dx > box.height && d.dy > box.width) {  
        transform += 'rotate(-90)';  
    } else if (d.dx < box.width || d.dy < box.height) {  
        d3.select(this).remove();  
    }  
  
    return transform;  
});
```

Finally! That was some interesting code!

We found all the leaves and started adding text. To fit labels into nodes, we get their size with `this.getBBox()`. Then we move them to the middle of the node, and check for fit. We also do a bit of regex and array manipulation to the text so that it removes everything except the last name and the initial before that. This is not the most sure-fire way to sanitize those strings—it's much better to use something like OpenRefine to clean up your data beforehand and put it in the format you ultimately want to present it as—but it works for our purpose right now.

If the label is too wide but fits vertically, we rotate it; otherwise, we remove the label after verifying again that it doesn't fit. Checking the height is important because some nodes are very thin.

We add tooltips with `helpers.tooltip`:

```
leaves.call(helpers.tooltip((d) => d.parent.name, this.chart));
```

Another fun bit—partially hiding nodes from different parents:

```
leaves.on('mouseover', (d) => {
  let belongsTo = d.parent.name;
  this.chart.selectAll('.node')
    .transition()
    .style('opacity', (d) => {
      if (d.depth > 1 && d.parent.name !== belongsTo) {
        return 0.3;
      }

      if (d.depth == 1 && d.name !== belongsTo) {
        return 0.3;
      }

      return 1;
    });
})

.on('mouseout', () => {
  d3.selectAll('.node')
    .transition()
    .style('opacity', 1);
})

.on('click', (d) => alert(d.name));
```

We used two mouse event listeners: one creates the effect, another removes it. The `mouseover` listener goes through all the nodes and lightens those with a different parent or those that aren't parents (that is, `d.parent.name` and `d.name` are different). This listener removes all changes. We also have it do an alert popup with the full label text if we click on any of the rect elements.

The `alert()` is without a doubt the most disgusting way of presenting information to the user. Even after ignoring the indescribably obnoxious ways in which this browser "feature" was used by Internet marketers in the 90s, it should still never be used, given their `alert()` method's visual similarity to operating system alerts, which can confuse the recipient. Furthermore, you can't style them, and they steal focus from whatever else is going on in the browser. You shouldn't even use them for debugging; `console.log` and `console.dir` are far more descriptive for dumping variables.

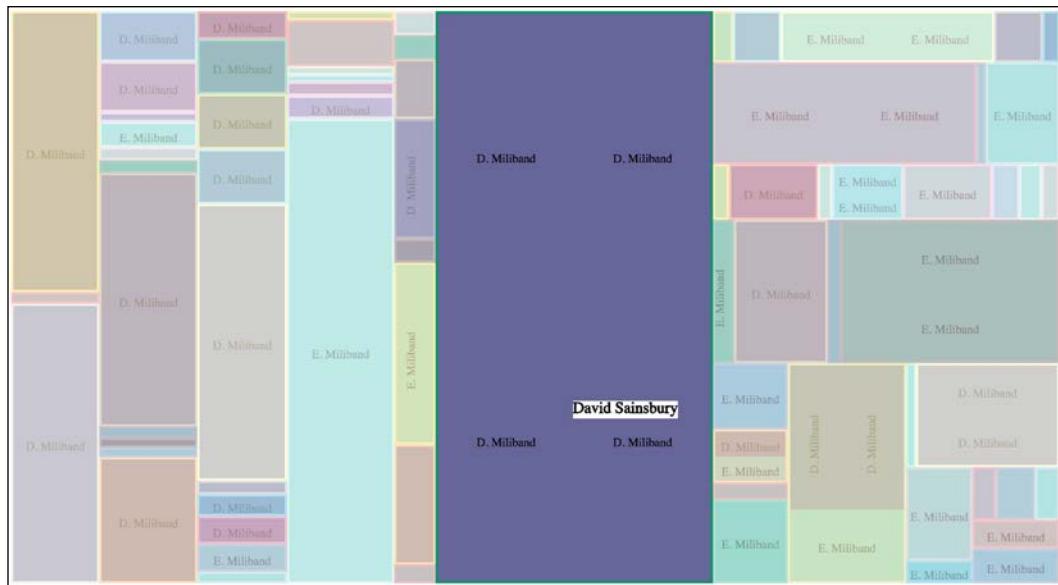
I used them in this example purely for demonstrative purposes; in all actuality, a much better way to handle click events would be to update a text element beneath the treemap. I'll leave this as an exercise for the reader.



After this, add a few more lines of CSS to chapter5.css:

```
.name text {  
    font-size: 1.5em;  
}  
  
.name rect {  
    fill: white;  
}
```

The end result looks like an abstract painting:



Each grouping is a single donor, with the size of its contents reflecting the size of the donation.

Summary

Despite the near-mythical power of D3 layouts, they turn out to be nothing more than helpers that turn your data into a collection of coordinates.

After going all-out with these examples, we used almost every trick we've explained so far. We even wrote so much code that we had to make a separate library! With a bit of generalization, some of those functions could be layouts of their own. There's a whole world of community-developed layouts for various types of charts. The `d3-plugins` repository on GitHub (<https://github.com/d3/d3-plugins>) is a good place to start exploring.

You now understand what all the default layouts are up to, and I hope you're already thinking about using them for purposes beyond the original developers' wildest dreams.

In the next chapter, you'll learn how to use D3 *outside* of the browser. That's right folks, we're headed to Server Town! In doing so, we'll strip D3 right down to its bare bones and use it to render things that aren't even SVG!

6

D3 on the Server with Node.js

Here's where we start to get really funky with D3. Not only can it render beautiful charts on the frontend, but we can also use D3 to generate things before they even get to the user's browser. This is really the cutting edge of D3, so realize that the skills you've learned in the preceding chapters will serve you in 95 percent of situations and don't sweat it if you want to stick to the frontend for now. This chapter will still be here for that rainy afternoon when you want to try to figure out how **Heroku** works.

We've added this chapter to the second edition because it uses D3 in a really abstract sense and we can start to tie up a lot of the stray concepts we've started discussing in the book

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)** webhosting providers, and you can manage an entire fleet of servers using a few simple tools. Instead of fighting with a huge monolithic Apache or Nginx configuration, you can deploy a new instance for every app you create, which sandboxes them and allows for much more compact infrastructure. We'll discuss how to deploy to Heroku later in the chapter; for now, we're just going to test everything locally.



What is Heroku and do you have to use it? Heroku is a way of deploying applications that use "12-factor app" principles (for the specifics, visit <http://12factor.net>). Without going too deep into the 12-factor app philosophy, the idea is that you try to create *stateless* applications that use web services in place of a large, monolithic piece of server infrastructure (for instance, a Linux-based virtual server running both the webserver and database processes). I use Heroku in this instance because it's simple to deploy to and free to use in limited capacities, but you can also deploy the code we'll write in this chapter on any server infrastructure that has Node.js installed.

You may not know it, but practically everything you need to write a server application resides in our project directory. If you've never done Node.js development before, it's very similar to what we've done in the preceding chapters with the browser, but with its own APIs and concepts. The JavaScript engine running on both Google Chrome and Node.js is called "V8", so you don't have to learn a totally different set of languages or skills in order to start immediately building server applications.

We're going to install a few more dependencies via npm:

```
$ npm install express body-parser --save
```

express is the leading Node.js web server library; it nicely abstracts Node.js's ability to open ports and serve content into an easy-to-use API that is very light and fast. However, because Node.js uses the CommonJS module loading standard, we need to add a new set of build instructions to our `webpack.config.js`. We need to make the object being exported an array (so both configurations run), so make it look as follows (the first config has been truncated for space reasons):

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

module.exports = [
  { ... }, // This is the first config; leave it be!
  {
    name: 'server',
    entry: './src/chapter6.js',
    target: 'node',
    output: {
      path: path.resolve(__dirname, 'build'),
      filename: 'server.js'
    },
    externals: {
      canvas: 'commonjs canvas'
```

```
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: 'node_modules',
        loader: 'babel'
      },
      {
        test: /\.json$/,
        loader: 'json-loader'
      }
    ]
  }
];

```

It's pretty much the same; we've just changed the target to node. The only thing that's different is we've told Webpack to treat `node-canvas` (which we'll use later in the chapter) as an external library instead of trying to bundle it.

Note that this will still emit all your other code during a build; if it's going slowly, feel free to temporarily disable the first `config` by commenting it out.

All aboard the Express train to Server Town!

Okay, let's get into the nitty-gritty right away and I'll explain what's going on. Add all of this into a new file called `chapter6.js`:

```
import express from 'express';
import bodyParser from 'body-parser';
import d3 from 'd3';
import {readFile} from 'fs';

import {nearestVoronoi} from './helpers';

let app = express();

app.use(bodyParser.urlencoded());
```

Here we're just importing all of our libraries; nothing to see here... Oh wait – didn't I say we have to use CommonJS because we're in Nodeville? No – because we're using Babel and Webpack to transpile all of our code, we can still use the lovely ES2015 module loading syntax without issue – even for Node.js modules that aren't themselves in ES6!

Anyway, we essentially instantiate Express, assign it to `app` and tell it to use the `urlencoded` form data body parser for `POST` requests (we installed this at the same time as `express`).

Add the following code:

```
app.get('/', (req, res) => { res.send('Hi there!'); });

app.listen(process.env.PORT || 8081, () => {
  console.log("We're up and running on " +
  http://localhost:(process.env.PORT || 8081);)
```

This sets up a route for `GET` (that is, a normal visit to a page, as opposed to a form submission, for instance) on the application's root path. Then it starts listening for requests on either port 8081 or *whatever that \$PORT environment variable is set to*. I've emphasized that last line because it's a key tenant in building web applications like this – keep all configuration in environment variables, so you never have to worry about storing plaintext passwords in your source code.

What are these "environment variables" I keep mentioning?

Your shell keeps a number of variables persistent that it uses to do things – you might be familiar with `$PATH`, which is a list of directories the shell looks in for executable code. Environment variables can also be used by web servers to hold configuration details, which can then be consumed by web applications (such as the one we're making!).



One example of where this is useful is database connection details – you generally don't want to version control sensitive details such as database passwords, so instead you provide them to your web server as environment variables, which your application then uses to connect to the database. There are a ton of other benefits to this, but suffice it to say, making your applications configurable through environment variables is a key aspect of writing good JavaScript server applications.

Now go back to the command line and type the following:

```
$ npm run server
```

Then visit <http://localhost:8081> and you should be greeted with the following text:



Congrats, you are now a bonafide web applications developer!

Well, not quite, but, *rapidly getting there!*

One thing worth noting at this point is that all instructions in a closure execute simultaneously, just like in the browser. In this way, JavaScript is said to be "asynchronous" – by executing everything simultaneously, you can't rely on something to be blocking like you can in PHP, for instance. Although it's a bit hard to wrap your head around this when starting in Node.js, this has really exciting ramifications for server code, making it very easy and very quick to scale horizontally.

Let's do something way less lame, and let's break out a brand new D3 feature while we're at it.

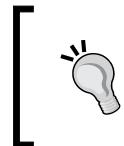
Proximity detection and the Voronoi geom

A **Voronoi geom** chops a geographic shape into discrete regions around points, such that no section overlaps and the entirety of the area is covered. Anything within a particular point's section is closer to that point than to any other point. We're going to use this to figure out what the closest major airport is to your current location, which we'll supply via the HTML5 Geolocation API.

Replace your call to `app.get` in `chapter6.js` with the following:

```
app.get('/', (req, res) => {
  res.send(`<!doctype html>
<html>
```

```
<head>
  <title>Find your nearest airport!</title>
</head>
<body>
  <form method="POST" action="/">
    <h1>Enter your latitude and longitude, or allow your
    browser to check.</h1>
    <input type="text" name="location" /> <br />
    <input type="submit" value="Check" />
  </form>
  <script type="text/javascript">
    navigator.geolocation.getCurrentPosition(function(position)
    {
      document.getElementById('latlon').value =
      position.coords.latitude + ',' + position.coords.longitude;
    });
  </script>
</body>
</html>`);
}) ;
```



You'll notice we put view code inside of our web server logic, which isn't great, but it works for our purposes. You'll want to use Express's views system for anything more elaborate than what we're doing.



If you haven't recently, restart the server by press *Ctrl + C* and then running the following:

```
$ npm run server
```

It's worth noting that, unlike in the frontend, we need to restart the server every time there's a change. If something isn't working as expected, try restarting it.

This sends the web browser a basic HTML document that asks them to fill in latitude and longitude as a comma-separated value. Or, if they accept the browser's request to use the HTML5 Geolocation API, it will auto-populate.

Next we need to create a new function for the Voronoi calculations. Inside of `helpers.js`, add the following:

```
export function nearestVoronoi(location, points) {
  let nearest = {};
  let projection = d3.geo.equirectangular();
```

```

location = location.split(/,\s?/);

let voronoi = d3.geom.voronoi(
  points.map((point) => {
    let projected = projection([point.longitude,
      point.latitude]);
    return [projected[0], projected[1], point];
  })
  .filter((d) => d);

voronoi.forEach((region) => {
  if (isInside(projection([location[1], location[0]]), region))
  {
    nearest = {
      point: region.point[2],
      region: region
    };
  }
});
}

if (nearest === {}) throw new Error('Nearest not findable');
else return nearest;
}

```

Lots and lots going on here; let's unpick it a bit.

We have two function arguments, `location` and `points`: `location` is a comma-separated latitude/longitude pair, `points` will be an array with all of our airports when we supply it from our server code in `chapter6.js`.

We then start things off by splitting our latitude/longitude string into an array, and setting up an equirectangular projection like we did with our first map all the way back in *Chapter 3, Making Data Useful*.

Finally, we get to configuring our Voronoi geom. Despite how complex this might look, it's pretty simple underneath – you ultimately just supply it with a bunch of points and it calculates the max size of each region. We do this via an `Array.prototype.map`, where we first project the longitude and latitude, then return these as an array. We also include the original airport object in the array so we know which point corresponds to which airport.



You can include as many additional array elements as you want and they'll all show up in each Voronoi region's `.point` object. The Voronoi geom only reads the first two elements of the array when constructing the Voronoi regions.

We also filter by the datum's identity, as follows:

```
.filter((d) => d);
```

This is because the Voronoi geom will return a point as `undefined` if there are any duplicates (such as there are in this particular dataset), which will mess up other stuff (particularly if you want to then use a path generator to draw your Voronoi regions). This strips out all the `undefined` items.

Our Voronoi regions in hand, we then do a `.forEach` loop to check whether our user's location is inside of each region. Once we find which Voronoi region our point is interior to, we return that. If none are found, we throw an error.

The last thing we need to do is write the `isInside` function, which we'll also put in `helpers.js`. This is taken from `substack/point-in-polygon` on GitHub, which you can install via `npm` if you want to save yourself some typing:

```
export function isInside(point, polygon) {
  let x = Number(point[0]), y = Number(point[1]);

  let inside = false;
  for (let i = 0, j = polygon.length - 1; i < polygon.length; j =
i++) {
    let xi = polygon[i][0], yi = polygon[i][1];
    let xj = polygon[j][0], yj = polygon[j][1];

    let intersect = ((yi > y) != (yj > y))
      && (x < (xj - xi) * (y - yi) / (yj - yi) + xi);
    if (intersect) inside = !inside;
  }

  return inside;
}
```

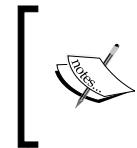
Without getting too deep into the preceding code, it ultimately draws lines out from a test point and looks at how many boundaries it crosses. It's based on the code from the following URL, which goes into much greater detail: https://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html.

Now we just need to wire this all up back in `chapter6.js`. Instead of a `GET` request, this time we're going to handle a `POST` request (which will contain our current location) to the same address.

```
app.post('/', (req, res) => {
  let location = req.body.location;
```

```
let airportPromise = new Promise((res, rej) =>
  readFile('src/data/airports.dat', 'utf8',
    (err, data) => err ? rej (err) : res(data)));
);
```

Pretty straightforward — when Express receives a POST request (such as when you submit a form), body-parser will intercept the form data and assign it to `req.body`. Next, we create a new Promise, where we use `fs.readFile` to load in our airport database.



We use `fs.readFile` because Node doesn't know how to use `d3.csv`, which uses `XMLHttpRequest`, which is only in the browser. Because the file we want is on the same computer as our server application, we can just load it in from the disk instead.

Let's resolve that promise and feed our helper functions some data:

```
airportPromise.then((airportData) => {
  let points = d3.csv.parseRows(airportData)
    .filter((airport) => !airport[5].match(/\N/))
    && airport[4] !== '')
    .map((airport) => {
      return {
        name: airport[1],
        location: airport[2],
        country: airport[3],
        code: airport[4],
        latitude: airport[6],
        longitude: airport[7],
        timezone: airport[11]
      }
    });
}

let airport = nearestVoronoi(location, points);
})
.catch((err) => console.log(err));
```

This is pretty straightforward; we're just reformatting our data so it's a bit nicer to work with. Note that we filter out airports without both the airport codes, which means we'll only get large international airports.



It's worth noting the `airports.dat` data set not only includes airports ranging in size from tiny Moose Jaw Municipal Airport to the gargantuan Beijing International, but also includes some international rail stations — for instance, London St. Pancras International. Which, *I guess* kind of makes sense? Trains are *sort* of like ground planes, right?

After that, we put both our location and point arrays into our `nearestVoronoi` function, which we assign to `airport`. We also log any errors to the console via `Promise.catch`, because otherwise we won't have any idea what's going on if it fails.

The only thing left to do now is return a HTML document to the web browser if we're successful. Add this after `let airport`:

```
res.send(`<!doctype html>
<html>
<head>
  <title>Your nearest airport is: ${airport.point.name}</title>
</head>
<body style="text-align: center;">
  <h1>
    The airport closest to your location is: ${airport.point.name}
  </h1>
  <table style="margin: 0 auto;">
    <tr>
      ${Object.keys(airport.point).map((v) =>
        `<th>${v}</th>`).join('')}
    </tr>
    <tr>
      ${Object.keys(airport.point).map((v) =>
        `<td>${airport.point[v]}</td>`).join('')}
    </tr>
  </body>
</html>`);
```

It's not going to win any prizes for beauty, but it does summarize the data with a minimal amount of code.

Okay, go back to your terminal. Hit `Ctrl + C` to kill off Node.js if it was still running from earlier, and then run the following from the root of your project directory:

```
$ npm run server
```

It'll take a second for Webpack to churn through everything, but once it's done, visit <http://localhost:8081> to see a screen like the following:

Enter your latitude and longitude, or allow your browser to check.

51.5975025,-0.0783584
Check

Either enter in a latitude/longitude pair, or let your web browser take your current position instead. Click `Submit` to see the results:

The airport closest to your location is: City

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! And we've figured something out using D3 that didn't even involve drawing anything!

By now you should be seeing D3 less as a bunch of magical tools that turn data into pretty visual things, and more just as a collection of functions that output mathematics a particular way. While D3 is certainly the most interesting when it's drawing things in a web browser, it's such a powerful library that you can use it for things far removed from its original use case of rendering SVG in the DOM.

Rendering in Canvas on the server

How about we do another one of those things right now? As mentioned before, the output from our little server app is pretty dull. Let's render a map using Canvas!

For this to work on the server, we're going to need to install `node-canvas`, which uses Cairo as a dependency. Assuming you're in Mac OS X and have Homebrew installed, run the following:

```
$ brew install pkg-config cairo libpng jpeg giflib
```

If you're not an OS X user with Homebrew installed, I suggest visiting <https://github.com/Automattic/node-canvas> and following the instructions there.



Alternatively, you can skip this entirely if you don't care to test locally, as we'll be deploying this all to Heroku later on in the chapter.



Next, add node-canvas to our app's dependencies:

```
$ npm install canvas earth-topojson --save
```

It's worth noting that this is a super weird way to use Canvas compared to how we can in the browser; normally we'd just rely on the browser's built-in Canvas renderer, but we don't have that luxury on the server. Note however, that the Canvas code we're writing for node-canvas will work the same way in the browser, in case you want to use it there.

We also install earth-topojson, which is a quick and dirty way of getting some basic political boundaries into a project.

Create a new function in chapter6.js:

```
function drawCanvasMap(location, airports) {
  let Canvas = require('canvas');
  let topojson = require('topojson');

  let canvas = new Canvas(960, 500);
  let ctx = canvas.getContext('2d');
  let projection = d3.geo.mercator()
    .center([location.split(/,\s?/) [1],
              location.split(/,\s?/) [0]])
    .scale(500);
}
```

This sets up Canvas and creates a new Mercator projection centered on the user's location. ctx is your canvas context; it's where and how you do your drawing.

Next, add the following to drawCanvasMap:

```
let boundaries = require('earth-topojson/110m.json');
let airport = nearestVoronoi(location, airports);
let airportProjected = projection([airport.point.longitude,
                                    airport.point.latitude]);

let path = d3.geo.path()
  .projection(projection)
```

```
.context(ctx);

ctx.beginPath();
path(topojson.feature(boundaries,
boundaries.objects.countries));
ctx.stroke();

ctx.fillStyle = '#f00';
ctx.fillRect(airportProjected[0] - 5, airportProjected[1] - 5,
10, 10);

return canvas.toDataURL();
```

The first three lines are what you'd expect — first we load in our boundaries, then we use our Voronoi function from earlier to calculate the nearest airport. From that we get a point, which we project into our coordinate system.

Then it's time to draw: we create a geo path generator, and supply `ctx` to it using the `.context` method. This is really pretty cool, because D3 does all the hard work in drawing paths in Canvas for us.

With our path generator doing its thing, we tell `ctx` to start drawing a path. We then run the path generator on our geometry, and add a stroke to it. Unlike what we normally do in D3, we don't chain these methods — we run them one right after the other, or *procedurally*.

We then draw a tiny little square on our closest airport, because that's how we do!

Finally, we use one of Canvas' super awesome features, which is outputting directly to a base64-encoded PNG data string. We return this data URL, because we'll be updating our `POST` method to output it.

Inside the call to `app.post`, add the following line above the call to `res.send`:

```
let canvasOutput = drawCanvasMap(location, points);
```

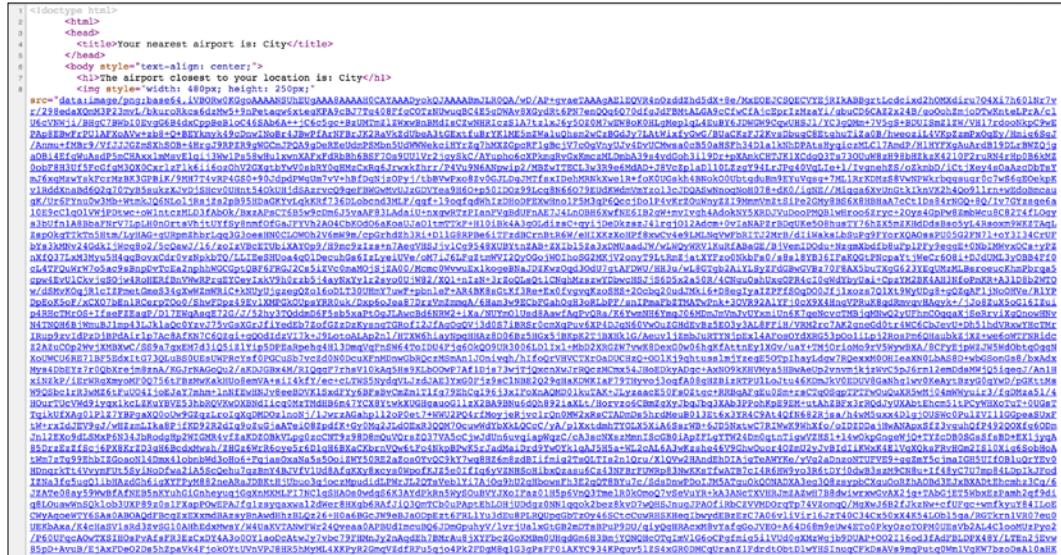
Then replace the call to `res.send` with the following:

```
res.send(`<!doctype html>
<html>
<head>
<title>Your nearest airport is:
${airport.point.name}</title>
</head>
<body style="text-align: center;">
<h1>The airport closest to your location is:
${airport.point.name}</h1>
```

D3 on the Server with Node.js

```
  
<table style="margin: 0 auto;">  
<tr>  
    ${Object.keys(airport.point).map((v) =>  
`<th>${v}</th>`).join('')}  
</tr>  
<tr>  
    ${Object.keys(airport.point).map((v) =>  
`<td>${airport.point[v]}</td>`).join('')}  
</tr>  
</tbody>  
</table>
```

This simply adds an image tag, with the `src` attribute set to our data string. If you look at it by right-clicking and selecting **Inspect Element**, you can see what our function is actually outputting, as follows:

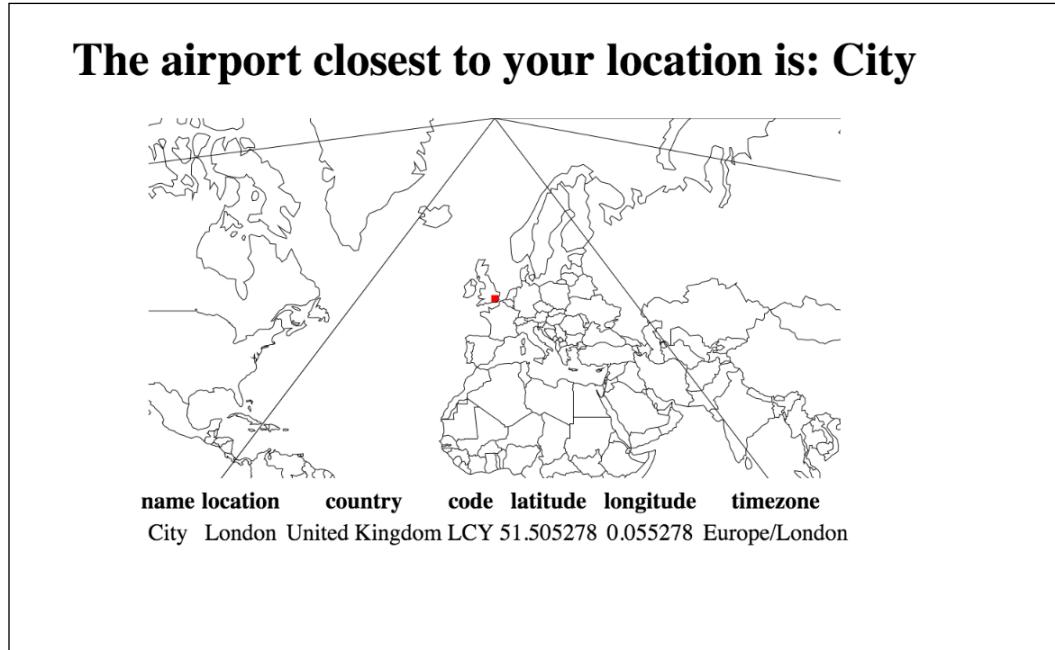
```
<!doctype html>  
<html>  
<head>  
<title>Your nearest airports in City</title>  
</head>  
<body style="text-align: center;">  
<div style="font-size: 1em; margin: 0 auto; width: fit-content; margin: 20px auto;">  
    <img alt="A map showing various airports in a city area." data-bbox="172 411 822 674"/>  
      
</div>  
</body>
```

This is what an image rendered to a base64 string looks like

Hit **Ctrl + C** if you still have Node.js running and then restart it by typing the following:

```
$ npm run server
```

Open up `localhost:8081`, enter a lat/long pair, and now your results page has a handy little static map!



Pretty cool, huh? Canvas, despite being a bit weird to use in D3, is a super powerful technology, particularly when rendering huge amounts of data (any DOM-based display language tends to get really slow after about 1000 elements; Canvas is effectively just a 2D drawing plane, so never has that problem).

Deploying to Heroku

A server app isn't very useful without a server!

Luckily, **Heroku** provides free plans for limited use and is super easy to deploy to. At the moment, they allow 18 hours of uptime per day on the free plan, with your machine downcycling when it isn't active (in effect, this means that your server generally won't ever run out of uptime hours provided it isn't being hit with traffic constantly).

Start by creating an account at <http://www.heroku.com> and install the Heroku Toolbelt from <http://toolbelt.heroku.com>. Once you've done so, go to the root of your project folder and type the following:

```
$ heroku create
```

This will create a new Git remote and set up your app at a random URL, like <https://calm-dusk-16214.herokuapp.com/>.

Next, create a new file named `Procfile`. Heroku looks at this when you deploy to know how to run your app. Add the following contents:

```
web: node build/server.js
```

Save, then make sure you have the latest bundle built, as follows:

```
$ npm run server
```

Hit `Ctrl + C` after it builds; we don't need to run it any more.

Because the servers you get from Heroku are really basic by default, we need to find a way of installing Cairo and any other dependencies `node-canvas` might have. Run the following:

```
$ heroku config:add BUILDPACK_URL=https://github.com/mojodna/heroku-buildpack-multi.git#build-env
```

This tells Heroku to use a custom buildpack when deploying. A buildpack is effectively just a recipe for configuring a server a certain way. We define our buildpacks in a new file called `.buildpacks` (notice the dot at the beginning). Add the following to it:

```
https://github.com/mojodna/heroku-buildpack-cairo.git  
https://github.com/heroku/heroku-buildpack-nodejsNode.js.git
```

Finally, add everything to a commit:

```
$ git add . && git commit -am "Time for Heroku"
```

We're now going to deploy! Assuming you're working from the master branch, type the following:

```
$ git push heroku master
```



Heroku always deploys from the master branch. If you've checked out the `chapter6` branch in Git, type the following instead:

```
$ git push heroku chapter6:master
```



Visit the URL provided by `heroku create` — and you should see your app in all its glory, online and accessible to anyone on the Internet! Congratulations, you've just written a pretty awesome webapp!

Didn't think you'd get a crash course on writing backend code in a book about data visualization, did you?!

Summary

In this chapter, first we set up Webpack to produce a separate bundle for the server, then we wrote a simple webapp using Express and D3's Voronoi geom to find the nearest airport to a user. We then upgraded our server app to draw a map using D3 and Canvas, which we then outputted to the user as a PNG.

Wasn't that all really pretty weird but also kind of fun? Writing server-side code is like that, but it can also be really nicely cathartic after spending a bunch of time doing frontend development, which tends to be really finicky due to having to support so many devices. If nothing else, hopefully you've begun to see how processing on the backend can improve the performance of your projects by offloading some of the processing work from the user's machine.

There's clearly quite a lot more you could learn about this topic that I just simply don't have room to cover here. We didn't go into scalability at all (which is super important when building things for large audiences, such as in the newsroom), nor into how to properly architect a non-trivial application — for that I'd recommend installing a few Express-based Yeoman generators and seeing how they scaffold projects (I'm particularly a fan of `generator-angular-fullstack`), or checking out Alexandru Vlăduțu's *Mastering Web Application Development with Express*, Packt Publishing, 2014.

You now have a pretty full toolbox for confronting a very wide array of data visualization challenges. In the next chapter, we'll add two more tools to it — unit testing and strong typing — in order to help you have more confidence in the work you produce.

7

Designing Good Data Visualizations

Data visualization is a tool that can be used in many ways. As you've seen while building examples throughout the book, data visualization is sometimes used to communicate information in a novel or interesting way; sometimes data visualization provides clarity, other times it's just used to make cool things.

Regardless of whether you're a journalist wanting to highlight a change in GDP, a scientist needing to communicate the results of an experiment, or a software engineer looking to integrate visualization into a product, chances are you'll want data visualization that is clear, concise, and does not mislead. Although the examples in this chapter will mainly be from a news media context, many of the points we'll discuss apply in a similar way to data visualization in general.

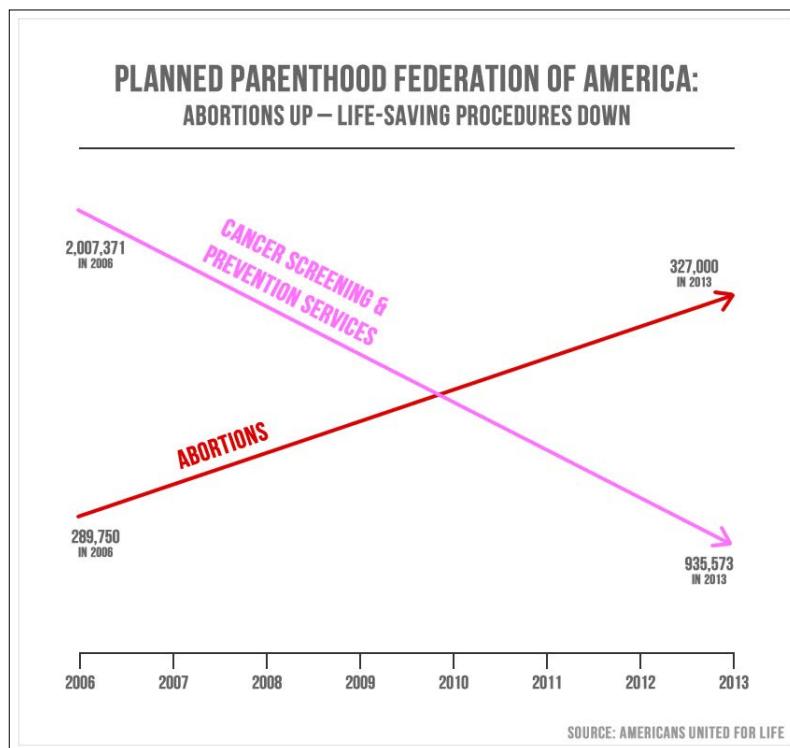
In this chapter, we'll look at a few general principles to keep in mind while building data visualizations, and I'll give some examples of good data visualization as well. Note that I'm in no way a data visualization expert, *per se* — I'm a developer and a journalist with a degree of learned design experience, and my thoughts on what constitute "good data visualization" are very much influenced by my background in building explanatory data-driven graphics for titles like *The Times*, *The Economist*, and *The Guardian*. These are very fast-paced newsrooms, and the goal when visualizing data is generally to communicate the important bits of a dataset to an audience instead of letting them explore the data. Although you might not have the same demands in your use of D3 as that of a newsroom developer, much of what I'll be discussing also applies if you're using D3 in academia, publishing, or elsewhere — the skill of being able to quickly and succinctly communicate information is incredibly valuable no matter your profession.

With that caveat out of the way, let's get on with discussing what exactly comprises good data visualization.

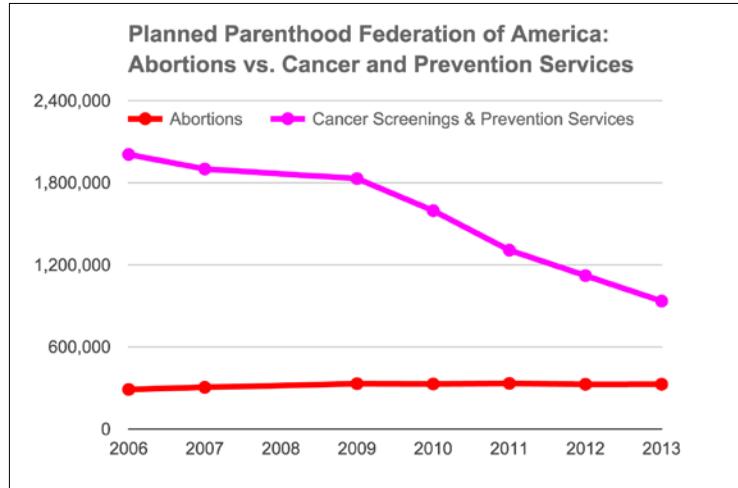
Clarity, honesty, and sense of purpose

There are two big schools of thinking in terms of data visualization at the moment: there's the ultra-minimalist philosophy espoused by Alberto Cairo and Edward Tufte, where the primary goal of data visualization is to reduce confusion, and then there are those who use data to create beautiful things that uphold design over communication. If you couldn't tell by the title of this section, I generally believe the former is far more appropriate in most cases. As somebody wishing to visually communicate data, the absolute worst thing you can do is mislead an audience, whether intentionally or not — not only do you lose credibility with your audience once they discover how they've been misled, but you also increase public skepticism over the ability of data to communicate the truth.

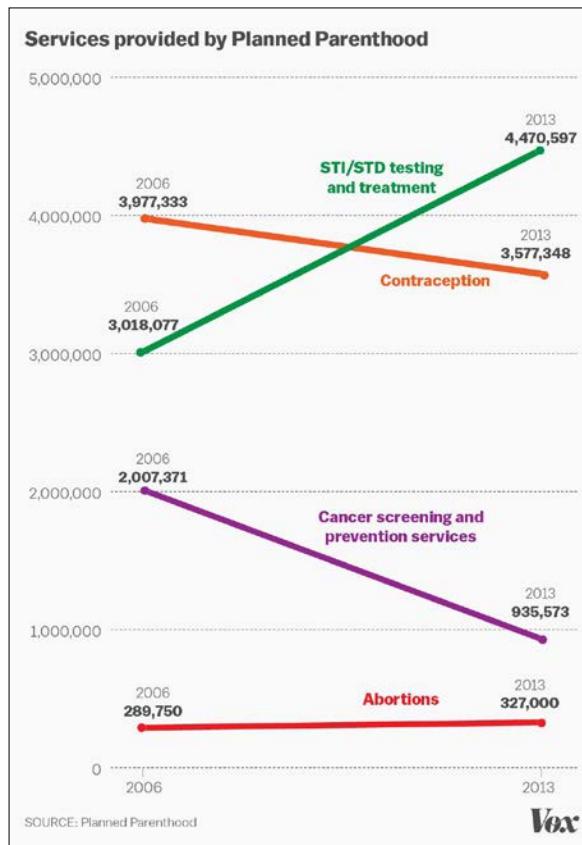
Here's a contemporary example. In September 2015, the U.S. Congress held a hearing on Planned Parenthood, the American reproductive and women's health group. During the hearing, Republican congressman Jason Chaffetz showed the following chart, created by the anti-abortion group Americans United for Life:



There are many problems with this chart, not least the complete lack of *y* axes. PolitiFact redrew the chart with corrected axes and it came out like this:



Vox took it a step further and drew the rest of Planned Parenthood's services:



As you can see, while there has definitely been a decrease in cancer screenings and prevention services (as well as contraceptives, for that matter), and a slight rise in the number of abortions, there has also been a dramatic increase in spending for STI/STD treatment and prevention. As Alberto Cairo commented on the original chart:

"That graphic is a damn lie ... Regardless of whatever people think of this issue, this distortion is ethically wrong."

The public backlash about this one misleading chart led it to being named "2015's Most Misleading Chart" by Quartz. Regardless of what the creators of the chart originally intended to demonstrate with it, any hope of achieving that goal was obliterated once viewers felt they were being misled.



For a more thorough discussion of everything wrong with Chaffetz's chart, I highly recommend the commentary by both PolitiFact and Vox, at <http://www.politifact.com/truth-o-meter/statements/2015/oct/01/jason-chaffetz/chart-shown-planned-parenthood-hearing-misleading-/> and <http://www.vox.com/2015/9/29/9417845/planned-parenthood-terrible-chart> respectively.



In the preceding quote, Cairo makes an interesting point in that communicating data carries with it certain fundamental ethical requirements. This is how *data visualization* differs from *data art* — in the latter, what's ethically required of the *artist* is to purposefully and honestly communicate their emotions, beliefs, fears, and so on. This is a long way off from the ethical requirement of the *visualizer*, which is to communicate specific qualities of the data through visual methods. Taking this a step further, the ethics of data journalism compel the journalist to tell the audience what the data really means and how it relates to that audience.

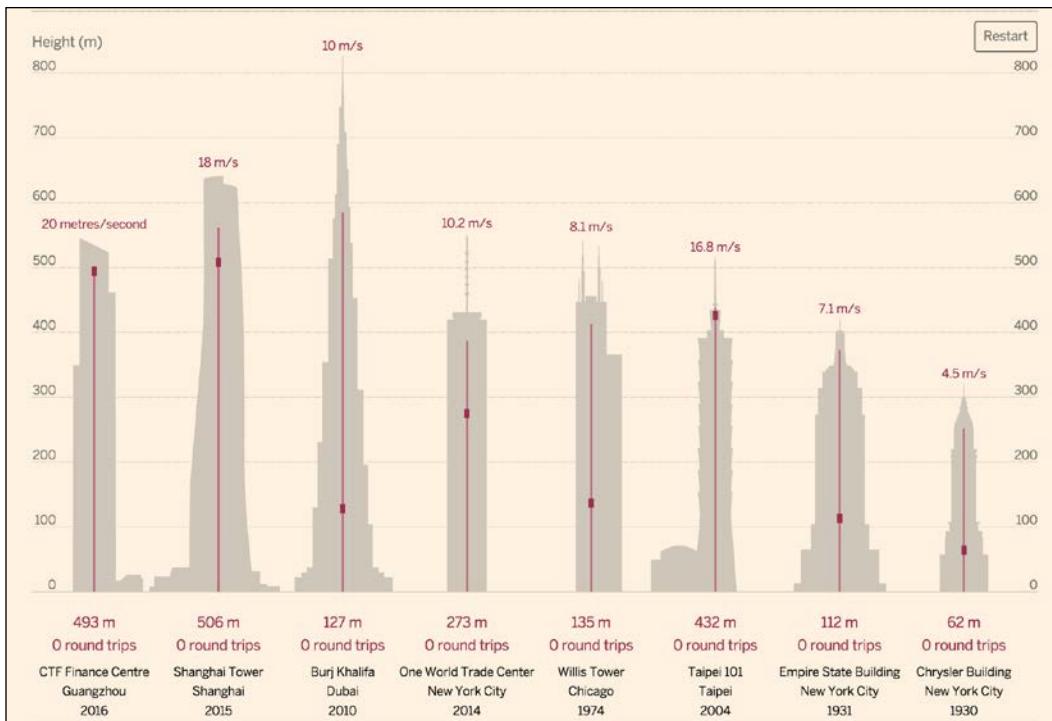
In your projects, decide where your scope lies. Are you acting in the role of a data journalist, with a desire to walk the reader through a specific bunch of numbers and figures? Are you acting as a data visualizer, perhaps creating a dashboard designed to quickly and effectively summarize a very large, multivariate dataset? Or do you want to build something fun and entertaining that leverages data merely as a method by which to achieve that aim? All three of these roles are perfectly acceptable, and there is room for work ranging from incisively explained line-charts all the way through to *objets d'art* that give us a better understanding of our size and place in the universe. But whatever you do, be clear with your intentions and never mislead.

Helping your audience understand scale

A big part of visualizing data is conveying scale and differences in magnitude. The following few examples do this particularly well.

To start with, please view John Burn-Murdoch's graphic on high-speed elevators for the at <http://www.ft.com/cms/s/2/1392ab72-64e2-11e4-ab2d-00144feabdc0.html>.

The following screengrab doesn't really do it justice:



If the above were the live visualization, you would see the elevators in each building endlessly rise and fall, with a counter beneath tracking how many times the elevator has gone up and down while you were looking at the page. A nice bit of easing at the top and bottom makes you feel like the little magenta square traveling along the line is a real elevator, subject to physics in the same way a big metal cage rapidly moving up and down the world's tallest buildings would be. Although this printed version only communicates one dimension — the relative heights of each elevator and building — the interactive version is able to use animation to convey a second quality; that is, the speed of the elevator. In this, one can really see the power of using the digital medium to communicate data in ways a static print version will never be able to. Scale is demonstrated by tying visual content to time; in this case, merely saying "the elevator can do a return trip in 2 minutes" would not be nearly as effective as demonstrating what "2 minutes" feels like to the reader.

Another good use of animation to explain scale is Hans Rosling's Gapminder project (<http://www.gapminder.org/world/>), which allows viewers to see how the world has changed over time. In his excellent TED talk about understanding the world from a data-driven perspective, Rosling discusses how, if looking at measures such as life expectancy and GDP per capita, the world is getting substantially better as time goes on, and that our perceptions of other countries are often rooted in a degree of ignorance as to how similar we generally are.



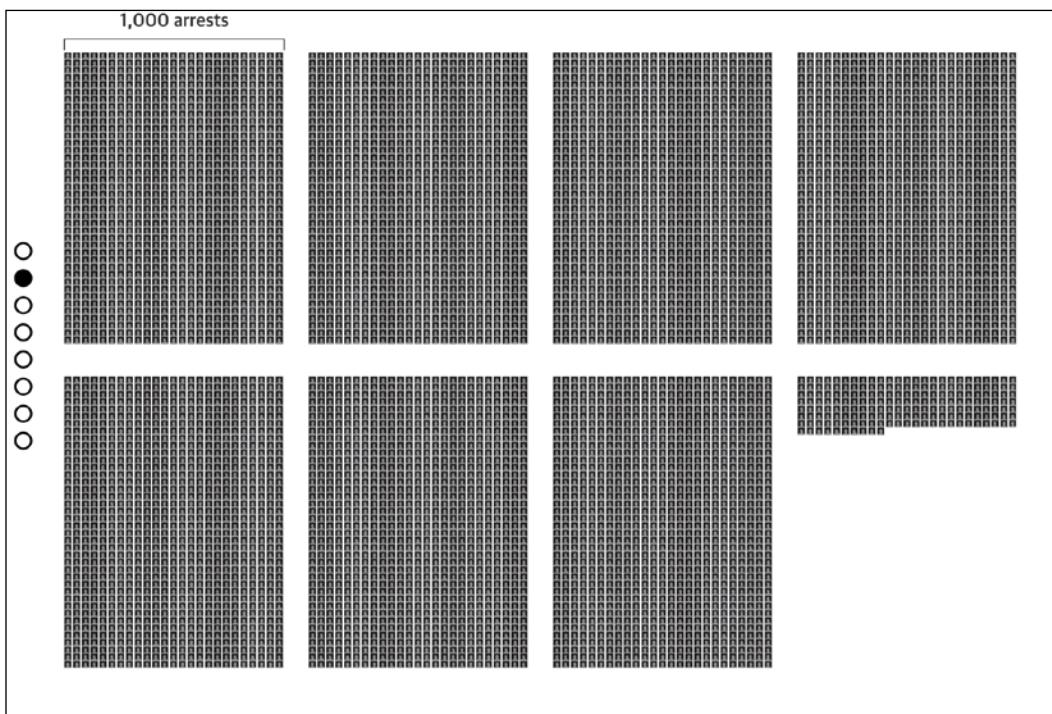
As time goes on, you can watch as the bubbles in the chart migrate from the bottom-left quadrant to the upper-right. It's worth comparing the fairly-dry exploratory visualization to the TED talk (https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen). Although depicting similar data, the latter is far more enjoyable simply due to Rosling's narration, with excitement evident in his voice as he explains how the world is improving and changing as societies develop. Good explanatory data visualization doesn't have to be limited to text output.



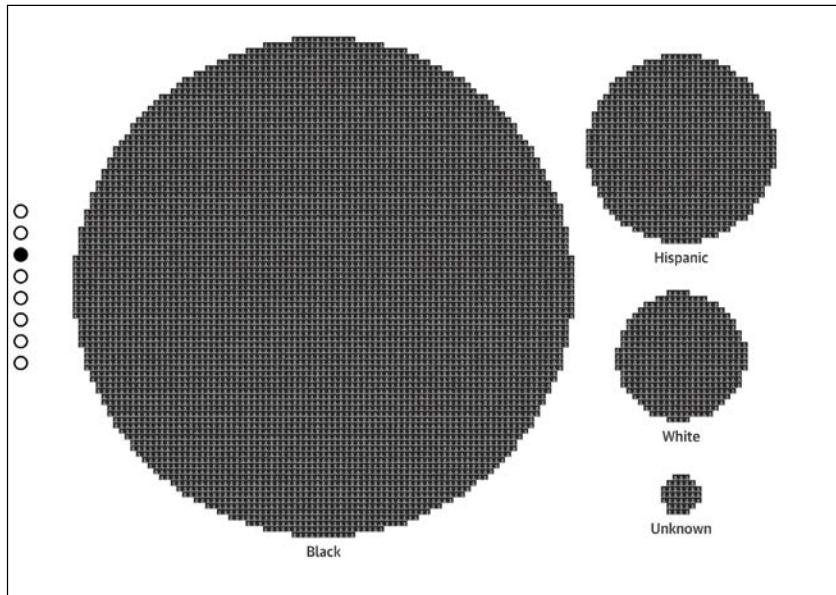
You can see an example of the preceding chart recreated using D3 by visiting Mike Bostock's implementation at <http://bost.ocks.org/mike/nations/>.

A much more elaborate example of embracing the digital medium to help convey scale is The Guardian's *Homan Square: A Portrait of Chicago's Detainees* interactive by Spencer Ackerman, Zach Stafford and the Guardian US interactive team. If you haven't seen it, please visit and prepare to have your mind utterly blown: <http://www.theguardian.com/us-news/ng-interactive/2015/oct/19/homan-square-chicago-police-detainees>.

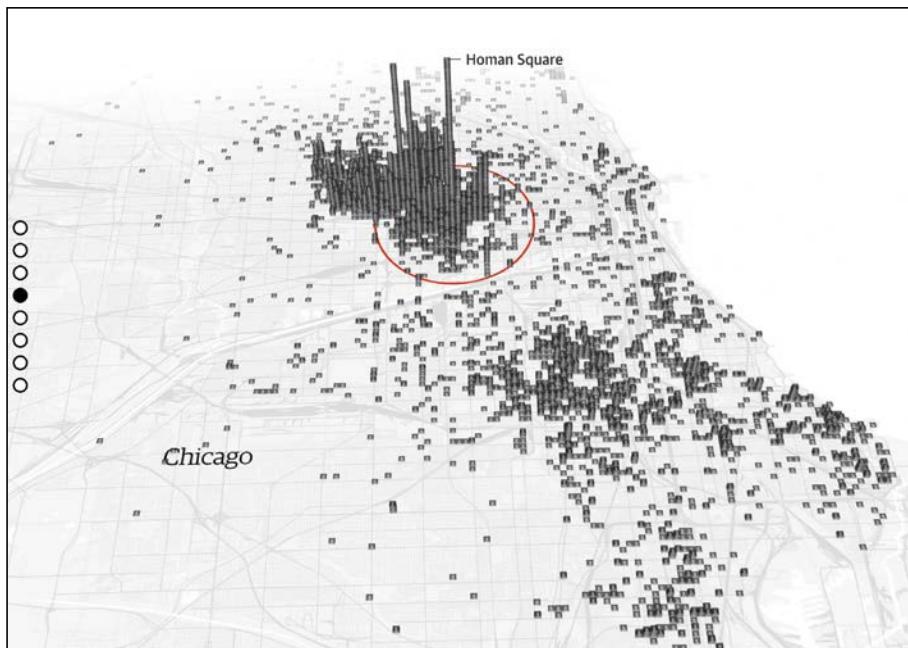
It starts with a big collection of faceless grey silhouettes, each representing a single person in custody at a secretive police warehouse in Chicago:

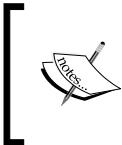


As you scroll, the faces fly around the screen to make up different configurations, such as this bubble chart:



And even this isomorphic map:



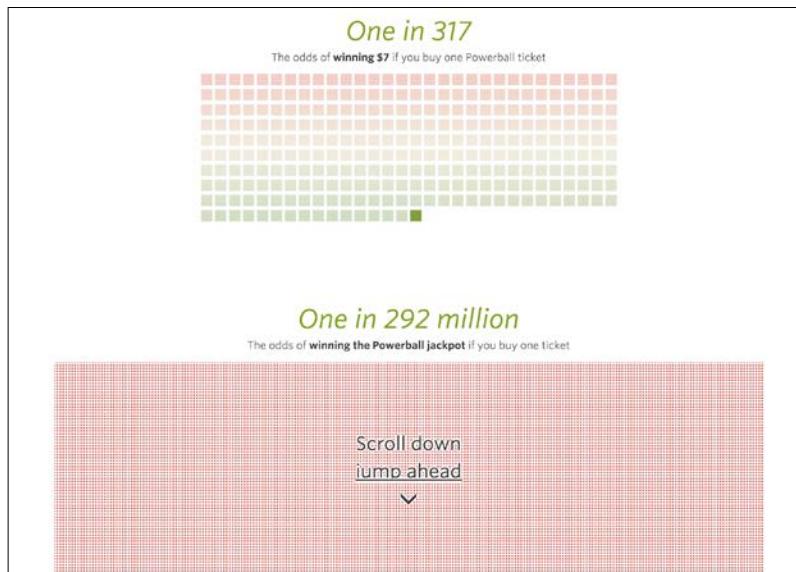


The Guardian's U.S. interactive team did a fantastic Q&A about the process behind this visualization that is well worth reading – <https://source.opennews.org/en-US/articles/how-we-made-homan-square-portrait/>.

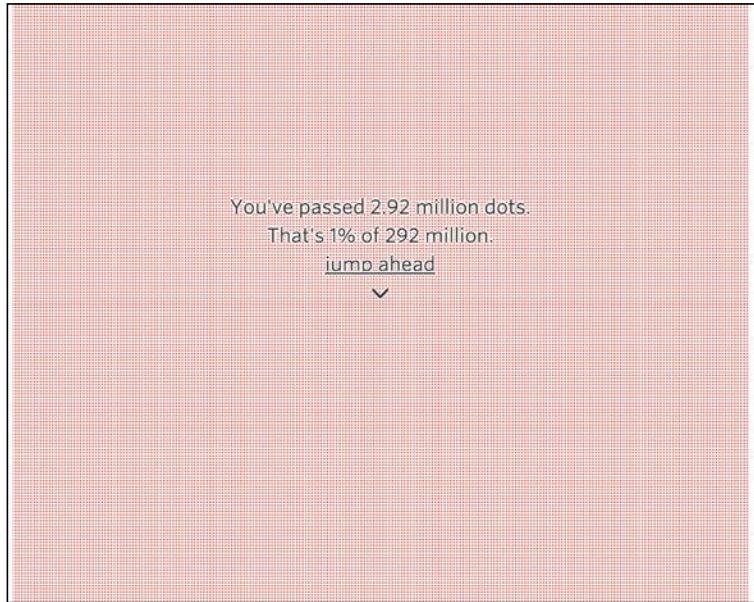
Although the data morphs in many different ways, you still feel an attachment to each point, remembering how they were originally displayed — they're not merely pixels on a screen; each portrait represents another person whose life has been impacted (a feeling reinforced when they single out individual portraits as case studies later on in the piece).

These examples are phenomenal for communicating scale to the reader. One of the biggest reasons data visualization is such a powerful tool is that it helps answer the questions "how big is 'big'?" and "how small is 'small'?" As reporting becomes increasingly reliant on data, it becomes very easy to mislead the reader by over- or under-emphasizing scale (indeed, this is a big reason why the chart in the preceding section was considered so dishonest). This can often be mitigated by designing visual output in such a way that the viewer feels some attachment to the visual stimuli onscreen.

As a final example, please visit this graphic by Ana Becker of *The Wall Street Journal* that attempts to visualize what your chances are of winning the Powerball lottery jackpot — it's another one where the following screenshots really won't do it justice. Also, try not to hit the **jump ahead** button for at least a little while, <http://graphics.wsj.com/lottery-odds/>:



After a while of continuous scrolling, the screen starts to look like this:



At this point, you realize there's no chance that you're going to ever physically scroll to the bottom, and so either click the **jump ahead** link or grab the scrollbar. If at that point you're still convinced you can plan your retirement based on your lottery earnings, probably nothing will change your mind.

In a sense, scale is emphasized through comparing the initial "coin-toss" probability to the probability of picking the winning Powerball numbers. Comparing the coin-toss's 1 in 2 with the Powerball's 1 in 292,201,337 is impossible to do in one screen (and especially so in print), because no matter what sort of scale you use, the latter completely dwarfs the former. Making use of the browser's practically unlimited vertical screen real estate is a very effective way of tying visual elements to a physical property (much like the elevator speeds interactive earlier), insomuch that the physical effort involved in scrolling down to just 1 percent of the page (much less, count how many dots that is) very effectively demonstrates how mind-bogglingly big a number like 292m is in the context of comparing probabilities.

Using color effectively

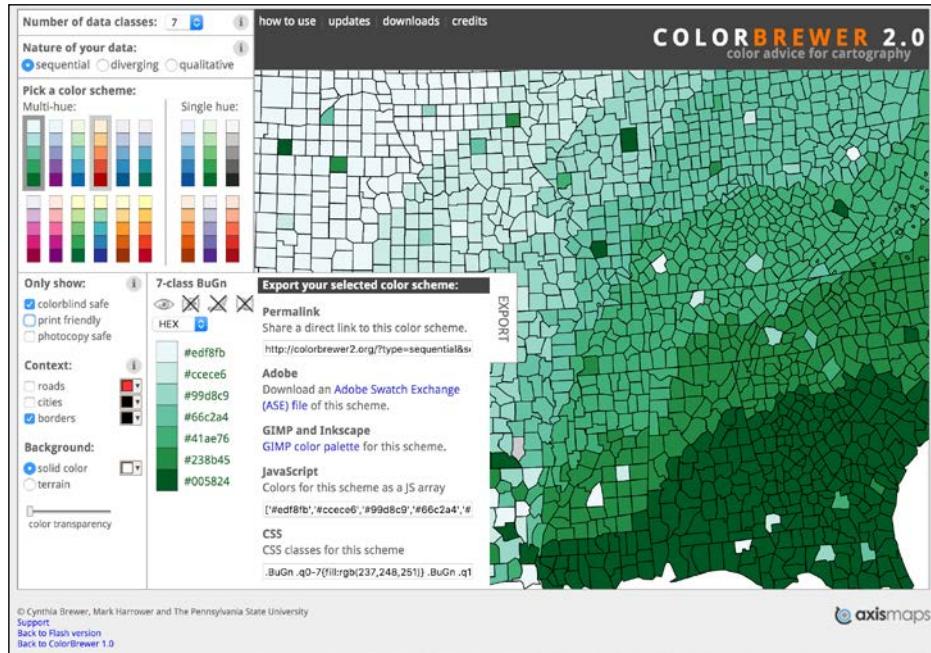
One of the biggest choices you'll make when building data visualizations is choosing what colors to use to represent what. While it's certainly possible to convey a lot of information through purely monochromatic charts, using the wide range of color representable through a digital display can be a very effective way of depicting another property dimension of the data you're visualizing.

If you plan to use color to communicate information, there are a few things to consider. The first is whether a user will be able to discern the pattern necessary to understand what the colors mean. If you have a legend explaining what color corresponds to what, try turning it off and thinking about whether you're still able to understand what the color combination means. Is it intending to show increasing intensity? Diverging values? Or just that it has a particular quality (in which case, do you really need a legend)?

Secondly, pay some attention to people who are colorblind. The most common color combination to use for choropleth maps is generally the "stoplight" color scheme: green for low values, yellow for medium values, and red for high values (or the inverse, depending on whether a high value is a good thing or not). There's a problem with this, though — for people who are red-green colorblind, the highest and lowest values look nearly identical.

Also, while still on the topic of stoplight colors, when discussing sensitive topics like immigration, a fair degree of care should be taken when color-coding anything green or red — if red means "bad" or "severe", is it actually that way? Or is it displaying red merely because the color scale has been constrained to the values in the data set? Colors are much more emotive, and it's easy to unintentionally present an opinion when using them. In general, using a sequential color scheme (scaled to the average of what one would expect that value to be) is a much safer bet.

A very good way of finding a color scheme for a map (and quite a lot else) is using ColorBrewer (<http://colorbrewer2.org>), a tool built by Cynthia Brewer and Mark Harrower at Penn State. It provides a bunch of different pre-built color schemes for representing data, and you can choose the type of relationship you want to depict. Additionally, it helpfully allows filtering out color schemes that aren't colorblind-friendly, or even printer- and photocopier-friendly.



You can use ColorBrewer schemes directly in D3! There's a file in the D3 repo at `lib/colorbrewer/colorbrewer.js`, but at the time of writing it has been removed in releases and isn't really easy for us to use with modular JavaScript. Instead, install it separately via:

`$ npm install colorbrewer --save`

And then require it as normal. For the above **7-class BuGn** scale (look next to the **EXPORT** pane for the scheme's name), you'd write:

```
let BuGn = require('colorbrewer').BuGn[7];
```

Lastly, listen to natural cues from your data: if you're doing a visualization of political parties, it makes sense to color-code the data at least somewhat similarly to the party colors (as boring as it is to always color-code political data to party colors, the novelty of not doing it this way is easily outweighed by the cognitive dissonance it causes).

Understanding your audience (or "trying not to forget about mobile")

Your audience is one of the most critical things to consider when beginning a new data visualization project. This has two parts: the first is from an editorial perspective (what is the audience's background knowledge of this topic? What types of charts will the audience be able to recognize and properly read? How do these charts work within the broader contexts of this story and other work published?), while the second is technological (what platforms and devices will be used to consume this content?).

It's really important to tentatively sketch out any bespoke data visualization before you start writing code, and this can take many forms. On one hand, it never hurts to figure out the rough shape of your data before committing to a particular visualization style — frequently I get asked for pie charts with a few small outlier values highlighted, which doesn't work (the rest of the chart dwarfs the outliers). You don't necessarily need to get a pencil and paper out for this — pasting your data into Excel and playing with its default charts often helps before committing the data idea to code.

The second way you should sketch out your visualization is on a component or an interaction level. This is where understanding which devices your audience use is important. If you have previous work out there, look at its analytics. To see what percentage of readers use a specific browser in *Google Analytics*, look under **Audience | Technology**. Pay attention also to **Audience | Mobile | Overview**, which will tell you what proportion of your audience is comprised of mobile and desktop. If you have no analytic data to work with (for instance, if you're launching a new project), it's generally a smart bet to assume that half of your audience will be on mobile, and so you should design your project accordingly.

Developing for an array of screen resolutions is called **responsive design**, and there are two major workflows: mobile-first and desktop-first. Traditionally, designs begin at desktop size and are then scaled down for mobile, but this often means that mobile support is an afterthought and it's often problematic to shrink down large elements later on. Mobile-first design starts at the smallest possible size and scales up, which is often easier as you get the more difficult versions of the site designed first and out of the way, and you can let things grow as screen resolution increases. Whether you need to do either depends on your audience — it's quite possible it predominantly uses one platform or the other, reducing the need to accommodate both; again, check your analytics. That said, while ensuring things work well cross-device and cross-browser takes a lot of additional work, it should be a standard of excellence that you strive towards as you create things for broader consumption.

If creating a mobile-first design, take out some paper and a black marker and draw a bunch of phone-sized shapes. Draw squares where you want each user-interface element (buttons, dropdown boxes, radio buttons, and so on) or chart to go — how you distinguish each element from the others is entirely up to you, just make sure that whatever vocabulary you use is shared by those you work with. You don't need to be super artistic with it, just be detailed enough to express how you think each user interaction (clicking/tapping, swiping, dragging, and so on) should *feel* within the project. Sketch out roughly how you think each element should *flow* on the different devices. Then do the same for a larger desktop screen size.



A rough drawing versus the finished page. Prototypes are meant to be discussion pieces you can use to solicit feedback from colleagues and refine throughout the development process. Don't worry if the end result looks nothing like your original sketches, or if (like me) you can't draw a straight line to save your life.

If possible, run your pen-and-paper prototypes past a few people to see whether your user interactions feel natural.

Some principles for designing for mobile and desktop

Mobile and desktop computing differ in some key ways, and understanding these is crucial for building data visualizations that are effective on both platforms.

Mobile:

- Has multiple screen orientations.
- Has uniformly-small screen dimensions.
- Does not have a pointer; interaction is derived from touch gestures. There is no "hover" state.
- Relies on often inconsistent data availability.

- Has significantly less computing power than comparable laptops.
- Keyboard is visible on-screen when in use; it is not used for navigation.

Desktop:

- Generally has only one screen orientation.
- Has variably-large screen dimensions (and is often connected to huge screens).
- Uses a pointer (and keyboard) to interact. The "hover" state is usable.
- Generally has reliable data availability
- Has significantly more computational power than comparable phones and tablets.
- Keyboard can be used without impacting what's visible on the screen; additionally, can be used for navigation.

I really don't have the space to go into a full course about how to do responsive design right here, but a few basic principles to keep in mind will get you started.

Columns are for desktops, rows are for mobile

On some level, it's fairly safe to assume that most people will view your work in portrait mode on mobile. This means that you effectively have one column, and every element in your one column is in a full row extending perpendicularly to the column's direction. Paragraphs should always flow vertically in a single column (instead of in multiple columns positioned left to right), and it's not an awful idea to rotate bar charts 90° so that your bars aren't squished together in one narrow horizontal row. Much like the *WSJ* lottery probabilities graphic above, make sure use of the infinite vertical space you have to scroll.

On desktop, however, a single paragraph spanning the entire horizontal width of the display is very hard to read, as it requires the eye to move back and forth quite a lot. Using columns is often preferable, as it not only makes better use of the horizontal space, but it also improves readability. Form elements in particular often look much better when grouped into columns.

The good news is that Flexbox makes it really easy to switch the orientation of groups of elements, provided you don't have to support older versions of Internet Explorer. If you aren't using flexbox already, take the effort to do so — it will make your life so much easier (at least, once you figure out how to use it).



Want to learn flexbox the fun way? Try out Flexbox Froggy, a game that teaches you how to position elements using Flexbox and Frogs. <http://flexboxfroggy.com/>.

Still find flexbox hideously frustrating? You're in good company. Try out Flexbox Grid, which uses Twitter Bootstrap-like classes to create responsive grids using flexbox. <http://flexboxgrid.com/>.

Be sparing with animations on mobile

Animation on mobile is really tricky, not only because you have reduced graphics processing power, but also because scroll events have traditionally messed with JavaScript execution timing. If you don't totally disable animation on mobile, try to only use CSS transitions, as these are more performant than iterating through properties via JavaScript. When in doubt, disable animation on mobile.

Realize similar UI elements react differently between platforms

Things like radio buttons, sliders, and checkboxes are available both on mobile and desktop, but some are easier to use on mobile than others. In general, web browsers draw all of these elements slightly too small for comfort on most mobile devices. Where possible (for instance, select dropdowns), make individual form elements stretch the entire device width on mobile, or in the case of things such as checkboxes and radio buttons, use the `for` property of the `<label>` element to make labels tapable and scale these horizontally.

Avoid "mystery meat" navigation

Pointer-based devices have the benefit of being able to use the cursor's "hover" state to reveal information (for instance, labels on buttons). While this can allow for more minimalist-looking interfaces on desktop, it's a really terrible anti-pattern referred to as "mystery meat navigation" when on mobile. When you hover over a button, you're not committing to clicking it. However, because mobile devices lack a hover state, users must commit to a user interaction to understand what a button actually does. Given how slow the mobile reading experience can be, users tend to be a lot more cautious before committing to any action that might cause the page to reload.

The solution is simple: unless using incredibly clear iconography, label your buttons on mobile.



For some good mobile icons, try Font Awesome (<https://fontawesome.github.io/Font-Awesome/>) and Google's Material Icons (<https://design.google.com/icons>).



Be wary of the scroll

A common user interaction idiom on desktop (popularized by the *New York Times'* "Snowfall" long-form piece) is to tie animations to the browser's scroll event. This is frequently fraught with peril on mobile, because scrolling triggers a memory-intensive redraw that often blocks JavaScript execution. While newer mobile operating systems handle this better than before (I'm looking mainly at you, iOS), it's often unsafe to assume that a scroll-dependent animation will play properly and not feel non-performant and "janky" on mobile. Again, this has improved quite a lot with newer devices, but it's still never a bad idea to tie animation states to tap events (possibly delivered via a button) on mobile.

Summary

Hopefully by now, you feel like you have a secure understanding of the work you want to do using D3 to visualize data. We went through some examples and laid some good ground rules for building high-quality data visualizations that not only inform an audience, but also look pretty spectacular while doing so. We also discussed how to make sure your work functions well on mobile.

In the next chapter, we'll tie things up by helping you feel more confident with the visualizations you produce by using type checking and automated testing to ensure that nothing goes wonky at the worst possible time. Stay tuned...

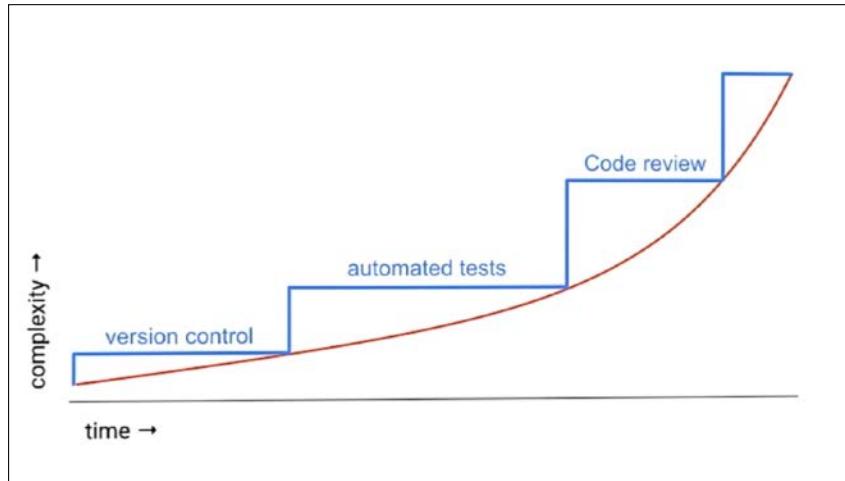
8

Having Confidence in Your Visualizations

When you're building things for as big an audience as the Web provides, a very real fear is that a software glitch prevents data from displaying correctly. When developing projects on a tight timeline, testing is one aspect that often gets neglected as the deadline gets closer and closer, and often things need to be viewed by other people even earlier (editors, managers, and other people further on down the line – possibly even lawyers) in turn emphasizing output over process. Let me be clear – if you want to ensure that your visualizations are of a high quality, you need to take steps to make sure that they are well tested and functioning properly. On some level, doing this is an exercise in managing complexity.

The next chart depicts project complexity over time. As you can see, complexity increases somewhat exponentially as time passes by. Adding more team members, more lines of code, and/or more dependencies increases the project's complexity dramatically. Meanwhile, the step chart depicts how tooling processes improve in response to complexity. Although it's possible to implement all of these at the beginning of every project, it's often better to do so incrementally in response to the project's demands. For instance, generally everyone will start with version control because it helps in collaboration, can revert mistakes, and provides a project with history (plus it's really easy to set up). Then, say you add a few more team members, or make a project open source. Now you have code flying at you from all directions. Having a way to automatically test whether that code will break anything or not starts to become incredibly useful.

A bit later, imagine a lot of bad code still getting past the automated testing; having manual code reviews might help this further. Each of these improvements to the process take time to both implement and use, and whether they'll benefit your project or not is hugely dependent on how big your team is and how well you trust every member of it (or, if working alone, how well you trust yourself not to introduce errors).



As time passes by, complexity increases somewhat exponentially, but tooling increases in a step pattern; source: Martin Probst and Alex Eagle, <https://youtu.be/yy4c0hzNXKw?t=245>

The preceding chart is from Martin Probst and Alex Eagle's talk on TypeScript at AngularConnect 2015. They touch upon a lot of the same topics as this chapter and it's worth watching (<https://www.youtube.com/watch?v=yy4c0hzNXKw>):

- In this chapter, we'll focus on a few technologies that help manage project complexity. We start by talking about linting. **Linting** is when you run your code against preset rules to ensure that it conforms to a set of standards.
- Then we will move on to **static** type checking. This is a tool for ensuring that the data flowing through your application doesn't act unexpectedly.
- We will end this chapter with **automated** testing. This is what it says on the tin: you write tests for your code that must all pass. Failing tests can be used to diagnose bugs or other issues with the code.

You honestly don't need to do any of these things to make stuff with D3, but getting into the habit of using these tools when necessary will both improve the quality of your code and make you a better developer. Let's dive in!

Linting all the things

A **linter** is a piece of software that runs source code past a set of rules and then causes a stink if your code breaks any of those rules. Now, I know what you're thinking: "my boss/manager/editor/significant other is *already* giving me more feedback than I'd care for. Why do I need *yet another* thing to do that?" Glad you asked!

Linting rules are often based on industry best practices, and most open source projects have a customized rule set corresponding to their community guidelines. This simultaneously ensures that code looks consistent even when delivered by a multitude of people and lets contributors know when they're doing something that is a little confusing or error prone in their code. Note, however, that all of these rules are just opinions—you don't *have* to write your code following them, but it tends to help everyone else out if you do.

If you've been following along with the GitHub repository for this book, then, perhaps, you've noticed a hidden file called `.eslintrc`, or noticed `eslint` in package `.json`. ESLint is currently the best linter for ES62016 code, and it works very similarly to its predecessors, JSHint and JSCS. It is configured via the `.eslintrc` file, which contains both the specific rules to be checked against (or a set of defaults to extend, such as what we've used) and information about the code's environment (for instance, NodeJS has different global variables than the browser).

To run ESLint against the current project, type this line:

```
$ npm run lint
```

Although there won't be any linting errors in the repository by the time this book goes to print, here's an example of what the output looked like at the writing stage of the book:

```
> learning-d3@1.0.0 lint /Users/aendrew/Sites/learning-d3
> ./node_modules/eslint/bin/eslint.js src/*.js

/Users/aendrew/Sites/learning-d3/src/chapter2.js
  44:12  error  "d" is defined but never used  no-unused-vars

/Users/aendrew/Sites/learning-d3/src/chapter3.js
  139:11  error  "lines" is defined but never used  no-unused-vars
  379:9   error  Unexpected constant condition  no-constant-condition
  387:9   error  Unexpected constant condition  no-constant-condition

/Users/aendrew/Sites/learning-d3/src/chapter4.js
  80:9   error  "timer" is defined but never used  no-unused-vars
  178:30  error  "rej" is defined but never used  no-unused-vars
  185:30  error  "rej" is defined but never used  no-unused-vars

/Users/aendrew/Sites/learning-d3/src/chapter5.js
  227:53  error  Irregular whitespace not allowed  no-irregular-whitespace
  338:9   error  "link" is defined but never used  no-unused-vars
  354:33  error  "d" is defined but never used  no-unused-vars
  357:32  error  "d" is defined but never used  no-unused-vars

/Users/aendrew/Sites/learning-d3/src/voronoi-airports.js
  3:25  error  "location" is defined but never used  no-unused-vars
  13:7   error  "path" is defined but never used  no-unused-vars

✖ 13 problems (13 errors, 0 warnings)
```

Tsk-tsk! What a lot of errors we have here!

We use npm to run ESLint in this instance, which is effectively an alias for the following line:

```
$ node ./node_modules/eslint/bin/eslint.js src/*.js
```

You can also install ESLint globally and then just run it anywhere.

Although this is helpful, linting is way more useful when you see it all the time while developing. Let's make ESLint scream at us while we're using webpack-dev-server. First, install eslint-loader:

```
$ npm install eslint-loader --save-dev
```

Next, we add `eslint-loader` as a preloader to our Webpack config, so it runs on our code *before* Babel does its thing with it. In `webpack.config.js`, add the following code in the module section of each build item:

```
preLoaders: [
  {test: /\.js$/, loader: "eslint-loader",
   exclude: /node_modules/}
]
```

Then run `webpack-dev-server`:

```
$ npm start
```

Ta-da! Now, you can get instant feedback about how messy all of your code is every time you hit save! You're utterly thrilled by this. I can feel it!

Linting is a very light way of managing complexity. Generally, a linting failure won't necessarily cause anything other than a message to appear on the developer's screen. It's up to you and your team to maintain a sense of discipline in terms of not committing code until it passes linting.

Static type checking with TypeScript and Flow

Static type checking is where you have a process that looks at how variables are being used, and then throws a wobbly if you do something weird. By this, I mean that it looks at the *type* of each variable and uses *type annotations* (bits of text defining what type a variable is when the variable itself is defined) to ensure that functions don't mutate a variable in an unexpected way. This is called **static typing**, and it is a feature built into many robust languages, such as C++ and Java. While JavaScript's dynamic typing (also shared by lots of other web languages, such as PHP and Ruby) is helpful in some ways and enables a certain style of programming, it can also be incredibly frustrating due to its ability to introduce silent errors. Because we're using a transpiler to transform our JavaScript anyway (throughout the book, this has been Babel, though that doesn't necessarily have to be the case), we can introduce static type checking to JavaScript at the same time if we so want.

There are two big players in statically typed JavaScript right now: Facebook's Flow project and Microsoft's TypeScript. Which you use is somewhat dependent upon when and how you plan to employ type checking, though my personal favorite is TypeScript for reasons that I'll get into later.

Although it's a level of tooling complexity above both version control and linting, in all honesty, to get the most from any of those three tools, it really helps to use them from the outset and be really disciplined with their use (particularly if you have two or more developers on your project). Flow is nice in that it is much easier to work into a Babel-based workflow than TypeScript, but TypeScript has a ton of benefits that extend beyond just ensuring that types don't change, and is particularly well-suited for working with D3.

Warning!

Lots of boring details about configuring stuff ahead! Wait, though! I have a solution!

The following sections will describe how to get started with either TypeScript or Flow, and give a bit of light config info, so you can get a sense of how it all fits together. Alas! Configuring things is really dull and should be avoided. If you want to dive right into playing with this stuff, you have two options. You can simply check out the chapter8 branch of the book repository by doing the following:

`$ git stash save && git checkout origin/chapter8`

Alternatively, you can install my handy `strong-d3` Yeoman generator, which will quickly scaffold out a brand new D3 project using either Flow or TypeScript. To install it via npm, run this line:

`$ npm install --global generator-strong-d3 yo`

Create a new directory for your project, switch into it, and run the generator, replacing `myProject` with whatever you want to call what you're working on:

`$ mkdir myProject && cd myProject && yo strong-d3 myProject`

Answer the questions it asks and you're on your way! For full usage instructions, visit [github . com/aendrew/generator-strong-d3](https://github.com/aendrew/generator-strong-d3).



The new kid on the block – Facebook Flow

If you have an existing ES2016 project and you want to introduce type checking into it retroactively, Flow might be the ticket. Unlike TypeScript, which totally replaces Babel, Flow lets you selectively add type checking to your projects. In many cases, this is all you'll need.

We're not actually going to use Facebook's normal Flow implementation (which starts up a whole server and is way too heavy for our uses); instead, we're going to use a Babel plugin that uses Flow-style syntax, `babel-plugin-typecheck`. Install that first:

```
$ npm install --save-dev babel-plugin-typecheck@2
```

Next, change your `.babelrc` file to the following:

```
{  
  plugins: ["typecheck"]  
}
```

Let's test this out. Start up `webpack-dev-server` with this command:

```
$ npm start
```

Then open `index.js`. Clear it out and add the following code:

```
function giveMeANumber(base: number): number {  
  return base * Math.random();  
}  
alert(giveMeANumber('hi!'));
```

What we do here is create a function that multiplies a random number by whatever number the function is supplied. As you can see, the function declaration looks a bit different:

```
function giveMeANumber(base: number): number {
```

What we're doing here is saying that the argument `base` should be a number and the function should always return a number. We then try to completely disregard what we've just said it should be and try to supply the function with a string:

```
  alert(giveMeANumber('hi!'));
```

If you open this up in your web browser, your console should let you know you've messed up:

```
✖ ▶ Uncaught TypeError: Value of argument 'base' violates contract, expected number got string index.js:1
```

Although this is a truly trivial example, imagine you had a massive project with tons of code, and this was your first day as a new developer on the project. Instead of possibly introducing a bug while learning the ins and outs of the code, Flow will smack you down straightaway, letting you know that you're using the code wrong. Plus, because the type definitions are right in the code, you can go straight to that function, look at the type annotation, and realize that you need to give it a number instead of a string. Combining it with a modern editor that lets you do this in a keystroke results in a very nice development experience.

TypeScript – the current heavyweight champion

That development experience is made even better with TypeScript, which is somewhat more stable and more widely used than Flow (at least at the time of writing this book). There are many upsides to using TypeScript, which I'll get into in just a moment. But before that, be forewarned that there's slightly more effort involved with getting TypeScript set up. This is because it's a transpiler in its own right (instead of just a type checker), which means we can't use Babel anymore.

The good news, however, is that TypeScript compiles modern JavaScript very similarly to how Babel does, and we just need to tweak a few things before we can begin using it. In practice, unless you're using a lot of Babel plugins, you probably won't notice the difference between the two. Further, we're actually going to use both simultaneously, importing TypeScript modules into Babel and vice versa like it ain't no *thang*.

First, let's install some more stuff. Modern JavaScript development is like 50 percent coding, 20 percent being confused about what libraries to use, and 30 percent installing those libraries. So let's get to it:

```
$ npm install typescript@1.8 ts-loader --save-dev
```

This will install both the TypeScript transpiler and the Webpack loader. Next, we're going to install a utility called `typings`, which helps us download prewritten type definitions for libraries such as D3:

```
$ npm install --g typings
```

Next, generate a `typings.json` file:

```
$ typings init
```

Then install the D3 TypeScript definition:

```
$ typings install d3 --save
```

Next, let's update our Webpack config. Under `loaders`, add the following:

```
{
  test: /\.ts$/,
  loader: 'ts-loader'
}
```

This will use TypeScript to load all files ending in `.ts`. You can go back through all your old files and rename them to `.ts`, or you can just do as we are doing here and use *both* Babel and TypeScript. Once you get used to TypeScript, you'll probably want to start with it from the beginning, but this time around, we're going to mash up both ES6 and TypeScript for great awesomeness.

First, let's add some lines to `index.js` to import our forthcoming TypeScript class and instantiate it with some data:

```
import {TypeScriptChart} from './chapter8.ts';
let data = require('./data/chapter1.json');
new TypeScriptChart(data);
```

Importing a TypeScript file into Babel is now as difficult as specifying the `.ts` extension.

Next, create a new file in `src/` called `chapter8.ts` and add this code at the top:

```
/// <reference path="../typings/main.d.ts" />
import * as d3 from 'd3';
```

This both includes your TypeScript definitions and imports D3 from `node_modules`. From here, we can start using TypeScript as we would normally.

 One worthwhile thing to do at this point is to create a `tsconfig.json` file, which is a TypeScript config file. With this, you can prevent a number of annoying behaviors from the TypeScript transpiler, and some TypeScript IDE integrations make extensive use of it (particularly `atom-typescript`). For an example `tsconfig.json` file, look at the `chapter8` branch of the book's repository.

Let's create a brand new chart, using our old friend `BasicChart` as a base. "But wait a minute, that's not in TypeScript?" you ask. Eh, give it a shot anyway!

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

Run Webpack:

```
$ webpack
```

And you'll get the following error:

```
ERROR in ./src/chapter8.ts
(2,26): error TS2307: Cannot find module './basic-chart'.
```

Eh, was worth a try! How do we solve this?

Create a new file in `src/` named `basic-chart.d.ts`. We are going to write a really simple, ambient type definition for our `BasicChart` class, which will allow us to add type checking to our existing class code without changing it or converting it to TypeScript. This is similar to how we consume D3 with TypeScript—we use the `Typings` utility to install D3's TypeScript definition, which gives us all the advantages of TypeScript with D3, all without any extra effort on Mr. Bostock's behalf (the definition is maintained separately by the fantastic folks at a project called `DefinitelyTyped`).

Here it is in one go:

```
import * as d3 from 'd3';
export declare class BasicChart {
    constructor(data?: any);
    data: Array<any>|Object;
    svg: d3.Selection<SVGELEMENT>;
    chart: d3.Selection<SVGElement>;
    width: number;
    height: number;
    margin: {
        left: number;
        top: number;
        right: number;
        bottom: number;
    }
}
```

This probably looks a bit different from how we've been doing things, so bear with me.

First off, we import everything from the `d3` module and call it `d3`. This is how you import D3 3.5.x as an ES2016 module, by the way. Although it didn't matter whether we used the shorter `require()` when our code was transpiled with Babel earlier in the book, in TypeScript you really want to use the ES6 module syntax.

Next, we declare the `BasicChart` class and export it. All we do then is list its properties (and methods if it had any), noting what types each property should take. All the numerical properties at the end of it are pretty self-explanatory. Let's take a quick look at the first four:

```
constructor(data?: any);
data: any;
```

We put our constructor and the arguments it takes. It has one optional argument (specified by the question mark), and we're not really prescriptive in terms of what it should be, so we use the `any` type. Because we then directly assign the argument to the class' `data` property, I've set that as `any`.



If you set `compilerOptions.noImplicitAny` in your `tsconfig.json` file to `false`, any variable without a type definition will be given the `any` type. If you're having trouble getting your code to work with types initially and just want to add types to existing code on an ad hoc basis, setting `noImplicitAny` to `false` might be something worth trying.

If we did some transformation on the data first—changing it to, say, an array of numbers—we can write something like this:

```
data: Array<number>
```

We supply the `number` argument to the `Array` interface, which is what we're doing with these weirdo angled brackets.

Similarly, we define the `svg` and `chart` properties using the D3 selection interface, containing `SVGELEMENTS`:

```
svg: d3.Selection<SVGELEMENT>;
chart: d3.Selection<SVGELEMENT>;
```



Where on earth did I get `d3.Select<SVGELEMENT>` from? Although we could have just specified these as the `any` type as an easy way out, TypeScript is more powerful when you give variables as specific a type annotation as you possibly can. For instance, the D3 definition declares several interfaces corresponding to the various types of variables created by D3, `Selection` being only one.

Throughout this book, I've attempted to be very neutral on the topic of editing environments and IDEs, not caring whether you use **Atom**, **Sublime Text**, **vi**, or something awful like Notepad. However, you *really really* should use an IDE if you're going to work with TypeScript. I use Atom (with the `atom-typescript` plugin). My environment tells me what interfaces the D3 module has while I'm typing, and I can just browse through them. Not only that, the autocomplete will tell me what arguments to specify for the functions I'm using, because a TypeScript definition effectively acts as machine-readable API documentation for the editor. It is super helpful!

Save and run Webpack again. It should compile without issues. Go back to `chapter8.js`, and we'll start filling out our new chart class. Let's kick it old school and redo the first chart that we did in this book as TypeScript.

```
import * as d3 from 'd3';
import {BasicChart} from './basic-chart';
export class TypeScriptChart extends BasicChart{
    constructor(data: Array<ITypeScriptChartData>) {
        super(data);
    }
}
```

Okay, this looks familiar. But what's that `ITypeScriptChartData` thing? Things that start with a capital `I` in TypeScript are generally interfaces, or reusable sets of typings. As you can see, our constructor takes an array of these. Let's create that interface now. Put this code at the bottom of the file, outside of your class:

```
interface ITypeScriptChartData {
    population: Array<IPopulation>;
    name: string;
}
```

If you remember, our data is comprised of an array of objects containing a name string and an array of population measurements. We don't need to create a new interface for the population array items, but it makes things a bit more readable if we do. Add the following code afterwards:

```
interface IPopulation {
    module_name: Array<any>;
    module_type: string;
    value: string;
    demography: {
        "04M": string;
        "04F": string;
        "511M": string;
        "511F": string;
        "1217M": string;
        "1217F": string;
        "1859M": string;
        "1859F": string;
        "60M": string;
        "60F": string;
    }
}
```

We go back to our constructor, and let's start fleshing out our class a bit more. Let's define our class' data property as an array of objects that are strings and numbers. Below the constructor, add this line:

```
data: Array<{name: string; population: number}>
```

It's good to do this because otherwise TypeScript will see the any type we used in our parent class and not know how the data is structured (as a result, it will need a bunch of type annotations in things such as D3 data accessors, which gets annoying).

Let's try to directly assign our constructor argument to `this.data` inside our constructor, as follows:

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

If you run the TypeScript compiler now, you'll get the following error:

```
ERROR in ./src/chapter8.ts
(15,7): error TS2322: Type 'ITypeScriptChartDatum[]' is not
assignable to type '{ name: string; population: number; }[]'.
  Type 'ITypeScriptChartDatum' is not assignable to type '{'
    name: string; population: number; }'.
    Types of property 'population' are incompatible.
      Type 'IPopulation[]' is not assignable to type 'number'.
```

Note that TypeScript, even if it's screaming at you for whatever reason, will still compile the JS files, which is why sometimes it works even if the compiler throws errors.

But not today! Let's munge that data like it's going out of style! Replace that last line with:

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

No more errors! Alright! Let's set some happy little scales inside the constructor:

```
this.x = d3.scale.ordinal().rangeRoundBands(
  [this.margin.left, this.width - this.margin.right], 0.1);
this.y = d3.scale.linear().range(
  [this.height, this.margin.bottom]);
this.x.domain(this.data.map((d) => d.name));
this.y.domain([0, d3.max(this.data, (d) => d.population)]);
```

Nothing new here, except that now we're throwing a TypeScript error, saying that the `x` and `y` properties do not exist. What?!

What we need to do here is define the types of these class properties. Just as with `data`, we need to tell TypeScript what form these should take. Add the following two lines beneath the constructor but still inside the class:

```
x: d3.scale.Ordinal<string, number>;
y: d3.scale.Linear<number, number>;
```

What we do here is use the `Ordinal` and `Linear` interfaces for D3 to let TypeScript know what kinds of scales these are supposed to be. Because our `x` axis is ultimately going to be a bunch of place names mapped to a location on the screen, we provide `string` as the domain argument and `number` as the range argument (the argument signature for the `Linear` interface helpfully provided by my IDE's autocompletion). Meanwhile, the `y` axis is a linear scale that maps numbers to numbers. Cool, no more errors!

Let's add the axes:

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

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

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

Nothing surprising here at all! Finally, let's draw the bars:

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

this.bars.transition()
```

```
.delay((d, i) => i*200)
.duration(800)
.attr('y', (d) => this.y(d.population))
.attr('height', (d) => this.height -
this.y(d.population));
```

Because we're saving the reference to our bars selection as a class property, we need to define what it is. Beneath the two scale type annotations, add the following code:

```
bars: d3.Selection<{name: string; population: number}>;
```

Woo! We've built our first strictly typed D3 chart using TypeScript!

This was a ludicrously shallow overview of TypeScript that barely scratched the surface, but hopefully, you're using a good editor and are already feeling how powerful it can be. To be able to use TypeScript effectively, you should know at least how to typecast each variable as well as create things such as interfaces. If TypeScript interests you as a technology, I highly recommend the handbook at <http://www.typescriptlang.org/Handbook>.

Behavior-driven development with Karma and Mocha Chai

All of these will take you pretty far towards having more confidence in your visualizations, but another step you can take to be even more of a rock star is adding automated testing to your projects.

There are many reasons to write automated tests. If you have a product that needs to render charts reliably and the chart rendering is merely a part of a much larger application, you will likely want to use automated testing to ensure that changes to the application don't break your charts. Likewise, if you've created an open source project that receives a lot of pull requests from various people who use your library, you might want tests to ensure that none of this outside code causes regressive bugs. Beyond this, automated tests are great if you want to be able to show your editor proof that your chart is working and is accurate, or if you merely want to gain more confidence in your data visualization work.

There are fundamentally two ways by which you can approach testing—you can build your project and add testing after the fact, possibly needing to refactor parts so that it's more easily testable; or you can write your tests at the very beginning of your project, before any code, and *then* build your project, ensuring that it passes the tests you've created at each step of the way.

The latter approach is called **test-driven development (TDD)**, and it should be seen as the hallmark of having reached some degree of skill with JavaScript. An extension of it is called **behavior-driven development (BDD)**, which tends to be more focused on user interface interactions. BDD tests tend to be less brittle, as they focus more on how a feature is functioning than how it works.

I personally find the syntax of BDD-style testing frameworks much easier to read and write, which is what I'll use here, via the Mocha Chai library.

There are many types of automated tests you can do, but we're going to mainly focus on unit and functional testing.

Unit testing is when you test each of your project's functions in an isolated fashion, which requires you to write your code in such a way that side effects are minimized. If you remember from *Chapter 3, Making Data Useful*, one aim of functional programming is to not have your functions produce side effects, and unit tests are a way of both ensuring and verifying that this is in fact the case. TDD focuses on writing unit tests for each part of your application as part of the initial development process—you're simultaneously quality-assuring your code as you write it. One upshot of this is you can be less reliant on ad hoc testing methods; that is, instead of switching between the web browser and your code editor every time you hit *save* to see whether something worked or not, you can switch to your command line's test runner, which will explicitly tell you whether that thing worked or not. This can often be much faster, which helps offset the time spent on writing the tests before anything else.

Functional testing, on the other hand, is more comparable to looking at how the application behaves in a consumer context. Imagine you buy a new phone. The phone has had each of its components tested at a very high level at the factory, and that whole process is opaque to you; you assume that it passed all tests because it made it out of the factory. However, you still test it in your own ways to make sure you like it: how does it feel in your hand? Is it light and flimsy-feeling or do the materials used make it feel like a premium product? Is the touch interface responsive? How do the buttons feel? Is the screen bright enough? How long does the battery last?

BDD is more geared towards testing the latter, and it's particularly helpful with data visualizations. In some ways, D3 does some of the unit testing work for you. Because your project is using a release of D3 that passes all of its tests (that is, "the phone has left the factory"), you don't need to worry so much about D3 doing anything wrong. Rather, your bigger concerns are preventing silent errors due to dynamic typing (that is, concatenating the numbers "1" and "1" as strings and getting "11" instead of adding them together to get "2") and ensuring that user manipulation of data doesn't cause errors. This is where BDD comes in.

Setting up your project with Mocha and Karma

We're going to use two things to write our unit tests: Karma and Mocha. Karma is the process that makes all our tests run inside of a browser, and Mocha is the test framework we're going to use. Chai is an add-on to Mocha that allows us to write BDD-style test assertions.

Install all of them in your project first, along with a few other goodies:

```
$ npm install mocha chai karma karma-webpack karma-chrome-launcher karma-nyan-reporter karma-sourcemap-loader karma-mocha --save-dev
```



Why do we keep using `--save-dev` instead of just `--save` in this chapter? We use `--save-dev` because these are developer tools that only run in the developer environment. They shouldn't be part of the distributed code.

We need to create a config file for Karma. You can either run through the wizard by running this:

```
$ ./node_modules/karma/bin/karma init
```

And then spend the next four hours reading documentation about how to set up Karma, or you can just grab the prebuilt `karma.conf.js` file in the `chapter8` of the book repo (which has lots of comments explaining what I've done). Alternatively, you can use `generator-strong-d3` to scaffold things out really easily—generally, saving time by using known good build environments is a sound strategy. Configuring build tools is a massive time sink that can usually be avoided by using good boilerplate code.

Let's get a feel of BDD-style development with Mocha and Chai by updating our TypeScript class from earlier with a few new behaviors.

Testing behaviors first – BDD with Mocha

We really aren't going to write a full test suite because we have very little space left in the book and writing about automated testing to any significant degree could fill way more pages than we really have left together. But what we'll do is update our `TypeScriptChart` class to sort the bar chart either by population or alphabetically. However, we will do so in a BDD fashion.

To start, create a new folder called `test/` and create a file called `chapter8.spec.js`. The convention is to name your test files the same as the file it's testing, but with `.spec` before the file extension. Before we write any code, it's often helpful to write out our goals in plain English:

- The chart should sort data alphabetically ascending by region name
- The chart should sort data alphabetically descending by region name
- The chart should sort data ascending by population
- The chart should sort data descending by population

As you'll see, our assertions in Mocha will resemble these statements quite closely. Open up `chapter8.spec.js` and add the following code:

```
import {TypeScriptChart} from '../src/chapter8.ts';
let data = require('../src/data/chapter1.json');
let chart = new TypeScriptChart(data);
describe('ordering a TypeScriptChart', () => {
  describe('alphabetically', () => {
    it('should sort data ascending', () => {});
    it('should sort data descending', () => {});
  });

  describe('by population', () => {
    it('should sort data ascending', () => {});
    it('should sort data descending', () => {});
  });
});
```

First, we import our chart library and our data and set up our parent `describe` block. The `describe` block is used to organize your tests. Having one parent `describe` block per `.spec` file that then contains more logical groupings is a good convention to follow. We then scaffold our assertions using `it`. Similar to `describe`, `it` groups a number of tests together (the assertions that we'll soon write in each `it` statement's callback function). Think of each `it` statement as a test and the assertions contained therein as the parts of the test needed for it to verify what the test says.

In our tests here, we're using the actual data that the chart will display. Because these are behavior tests, we're just testing to ensure that the behavior that we expect actually occurs. Often, instead of loading in a dataset, you'll create a mock data sample, which you'll use to test each function or method. Using mock data is often much faster and helps ensure that the tests are functionally correct; in other words, we ensure that they're not passing merely because of an edge-case bug resulting from, for instance, a dirty dataset. Which testing method you use is entirely up to you and how detailed you want to be with your testing.

Before we add assertions, we need to make our code do the behavior we're testing. Update your test to resemble this:

```
describe('ordering a TypeScriptChart', () => {
  describe('alphabetically', () => {
    it('should sort data ascending', () => {
      chart.order('alphabetical', 'asc');
    });
    it('should sort data descending', () => {
      chart.order('alphabetical', 'desc');
    });
  });

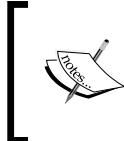
  describe(' by population', () => {
    it('should sort data ascending', () => {
      chart.order('population', 'asc');
    });
    it('should sort data descending', () => {
      chart.order('population', 'desc');
    });
  });
});
```

As you can see, our chart now seems to have an `order` method, which takes two arguments: ordering type and direction. Our humble `TypeScriptChart` class doesn't have one of these yet, but we'll get to that shortly. First, we need to write our assertions. Update the first nested `describe` block to resemble the following:

```
describe('alphabetically', () => {
  it('should sort data ascending', () => {
    chart.order('alphabetical', 'asc');
    chart.x.domain().should.have.length(8);
    chart.x.domain()[0].should.equal('burundi');
    chart.x.domain()[7].should.equal('yemen');
  });
  it('should sort data descending', () => {
    chart.order('alphabetical', 'desc');
    chart.x.domain().should.have.length(8);
    chart.x.domain()[0].should.equal('yemen');
    chart.x.domain()[7].should.equal('burundi');
  });
});
```

We look at the `x` scale's domain to see whether the sort has worked or not because that's what we use to update things such as the `x` axis. From querying our dataset, we know that the first item alphabetically is Burundi and the last is Yemen. In each test, we first verify that the domain has the right number of items, and then verify that the first item is what it should be and the last is what it should be. If we wanted, we could have checked the entire array, but that would be more work and make the test slightly more brittle. Let's do the second block now:

```
describe('by population', () => {
  it('should sort data ascending', () => {
    chart.order('population', 'asc');
    chart.x.domain().should.have.length(8);
    chart.x.domain()[0].should.equal('liberia');
    chart.x.domain()[7].should.equal('syria');
  });
  it('should sort data descending', () => {
    chart.order('population', 'desc');
    chart.x.domain().should.have.length(8);
    chart.x.domain()[0].should.equal('syria');
    chart.x.domain()[7].should.equal('liberia');
  });
});
```



It suffices to say that these really aren't the best tests. Because you're not using mock data, it's not entirely obvious where strings such as `syria` and `liberia` are coming from. Our dataset is small enough that we can get away with it, though.



From the preceding lines, you can get a taste for what Mocha Chai BDD syntax is like. There are also a few other types of syntax you can use, depending on your preference and testing style, but I find the Chai BDD style easiest to read. As well, I use the "Should" variant; the other variant, "Expect", would look like this:

```
expect(chart.x.domain()).to.have.length(8);
```

Cool! Now that we've written our assertions, it's time to write some *actual* code! Open up `chapter8.ts` and add the following function to your `TypeScriptChart` class:

```
order(type, direction) {
  this.data = this.data.sort((a, b) => {
    switch(type) {
      case 'population':
        return direction.indexOf('asc') ?
          b.population - a.population : a.population -
          b.population;
```

```

        case 'alphabetical':
            return direction.indexOf('asc') ?
                (a.name < b.name ? 1 : a.name > b.name ? -1 : 0) :
                (a.name > b.name ? 1 : a.name < b.name ? -1 : 0);
        }
    });
    this.redraw();
}

```

We still need to write the `redraw` function to update all the scales; put this below the preceding code:

```

redraw() {
    this.bars.data(this.data)
        .enter()
        .append('rect')
        .attr('class', 'bar')
        .attr('x', (d) => this.x(d.name))
        .attr('width', this.x.rangeBand())
        .attr('y', () => this.y(this.margin.bottom))
        .attr('height', 0);

    this.bars.transition()
        .delay((d, i) => i*200)
        .duration(800)
        .attr('y', (d) => this.y(d.population))
        .attr('height', (d) => this.height - this.y(d.population));

    this.x.domain(this.data.map((d) => d.name));
    let xAxis = d3.svg.axis().scale(this.x).orient('bottom');
    this.chart.select('.x.axis').call(xAxis);
}

```

We are finally done! Jump back to your terminal and run Karma:

```
$ ./node_modules/karma/bin/karma start
```

Look at that! 4/4 passed! Nyan Cat loves us!

```

01 03 2016 00:28:22.958:WARN [karma]: No captured browser, open http://localhost:9876/
01 03 2016 00:28:22.969:INFO [karma]: Karma v0.13.21 server started at http://localhost:9876/
01 03 2016 00:28:22.975:INFO [launcher]: Starting browser Chrome
01 03 2016 00:28:24.188:INFO [Chrome 48.0.2564 (Mac OS X 10.11.3)]: Connected on socket /#N17wa4xXuBXISqFDAAAA with id 87436581
4
4
4
4
0
0
0
0
4 total 4 passed 0 failed 0 skipped

```

It doesn't always look like that though. If we had run Karma before we wrote our `redraw` function, the output would have looked like this:

```
4  ✓ ordering a TypeScriptChart alphabetically
   ✓ should sort data ascending
     Chrome 48.0.2564 (Mac OS X 10.11.3)
       1) TypeError: this.redraw is not a function
          at TypeScriptChart.order (/Users/andrew/Sites/learning-d3/test/chapter8.spec.js:132:15 <-- webpack:///src/chapter8.ts:62:8)
          at Context.<anonymous> (/Users/andrew/Sites/learning-d3/test/chapter8.spec.js:58:14 <-- webpack:///test/chapter8.spec.js:9:18)

   ✓ should sort data descending
     Chrome 48.0.2564 (Mac OS X 10.11.3)
       2) TypeError: this.redraw is not a function
          at TypeScriptChart.order (/Users/andrew/Sites/learning-d3/test/chapter8.spec.js:132:15 <-- webpack:///src/chapter8.ts:62:8)
          at Context.<anonymous> (/Users/andrew/Sites/learning-d3/test/chapter8.spec.js:74:14 <-- webpack:///test/chapter8.spec.js:24:18)

  by population
    ✓ should sort data ascending
      Chrome 48.0.2564 (Mac OS X 10.11.3)
        3) TypeError: this.redraw is not a function
          at TypeScriptChart.order (/Users/andrew/Sites/learning-d3/test/chapter8.spec.js:132:15 <-- webpack:///src/chapter8.ts:62:8)
          at Context.<anonymous> (/Users/andrew/Sites/learning-d3/test/chapter8.spec.js:73:14 <-- webpack:///test/chapter8.spec.js:24:18)

    ✓ should sort data descending
      Chrome 48.0.2564 (Mac OS X 10.11.3)
        4) TypeError: this.redraw is not a function
          at TypeScriptChart.order (/Users/andrew/Sites/learning-d3/test/chapter8.spec.js:132:15 <-- webpack:///src/chapter8.ts:62:8)
          at Context.<anonymous> (/Users/andrew/Sites/learning-d3/test/chapter8.spec.js:79:14 <-- webpack:///test/chapter8.spec.js:38:18)

  4 total  0 passed  4 failed  0 skipped
```

Yikes, Nyan Cat isn't looking so good now.

In some ways, the latter is far more useful. It tells us where our tests are failing, which we can use to quickly debug problems with our code. This is the whole point of tests — to be able to pinpoint exactly where something's broken when something goes wrong during development.

Once again, there's absolutely no way I can go into a topic this big in the space of one chapter, but hopefully, you get a sense of how you can start building testing into your projects. If this is something that interests you, I highly recommend doing quite a bit more reading about the philosophy behind automated testing and BDD. Although writing tests is fairly easy, writing *good* tests takes quite a lot of skill to master.

Even though automated testing is a good tool for giving yourself confidence in your data visualizations, don't trust it absolutely. It's possible to write a whole lot of tests that don't really test anything, and things like confirmation bias have a tendency to creep into test writing. That aside, it can be a tremendously helpful tool for producing world-class code.



Next time you're feelin' like you're an awesome automated test writer, do this fun 2-minute interactive quiz from NYT:

<http://www.nytimes.com/interactive/2015/07/03/upshot/a-quick-puzzle-to-test-your-problem-solving.html>.

Then, if you get it wrong, write more negative test assertions.

Summary

As much as it pains me, I believe it is nearly time to bid each other adieu. I truly hope you've learned some things and enjoyed the preceding eight chapters – writing a textbook is a fine balance of getting to the quick 'n' dirty learning bits while also having some fun along the way. Regardless, if you have somehow read this thing from beginning to end, my hats off to you, as we have covered an absolutely mind-boggling array of technologies and approaches to software development.

We started off with some super basic stuff, talking about the DOM and CSS. Then we deep-dived into SVG and you learned how to build super pretty web vector graphics. After that, we started doing some really neat stuff with D3 – remember that chapter on layouts when we made, like, a gazillion charts?!

We've done some cool stuff with Node.js and Canvas, we've made a boatload of cool maps, and we even put some stuff on Heroku because we're web development ninjas! Hee-yah! So much stuff!

Lastly, we talked about making truly great projects with all of this technology and being confident in our work. Whether it's looking to others for inspiration or wiring up a solid test suite to make sure that our work is accurate, data visualization is a craft. It is one craft that will only continue to grow in importance as our world becomes all the more "data rich," but one that requires a degree of precision. Don't worry if you get some things wrong, but do try your hardest to get everything right.

We are in a truly amazing era of the Web, where the restrictions that were once preventing developers from building brilliant things are slowly being eliminated, by smarter approaches, bolder decision-making, and better tooling. It is all moving *frighteningly* fast, but don't let that hold you back from trying new things and playing with code that is on the cutting edge. There is no other field in computing where you can build things that work so *universally* and so *instantaneously*. Although web development can be really difficult (hello, responsive design!), never has any one set of technologies been so utterly crucial to the way the world consumes information. Having finished this book, you now have in possession a massive war chest of tools you can try to use. Some of them (particularly the ones I sprinted through, that is, Node, Canvas, TypeScript, and Mocha) you clearly need to learn a lot more about to use effectively, but hopefully, I've given you a few things to try out in your journey to understand and make use of all this stuff. At the very least, you should have everything you need to start visualizing your world with D3 and be able to share it with the world through *the magic of the Internet*.

Finally, if you get frustrated and need either a hand or a place to express your annoyance that this *thing* you've been working on is still broken two days later, come join us on the D3 Slack channel—there's usually somebody or the other around who's been there and can help.

-Æ.

Module 2

D3.js By Example

*Create attractive web-based data visualizations using the amazing
JavaScript library D3.js*

1

Getting Started with D3.js

D3.js is an open source JavaScript library that provides the facility for manipulating HTML documents based upon data, using JavaScript as the language for implementing the mapping of data to the documents. Hence, the name **D3** (Data Driven Documents). Many consider D3.js as a data visualization library. This may be correct, but D3.js provides much more to its user than just visualization, such as:

- Efficient selection of items in the HTML DOM.
- Binding of data to visual elements.
- Specifications on handling the addition and removal of data items.
- The ability to style DOM elements dynamically.
- Definition of an interaction model for the user with the data.
- The ability to specify transitions between data visualizations based upon dynamic changes in data.
- D3.js helps you bring data to life using **HTML**, **SVG**, and **CSS**. It focuses on the data, the way it is presented to the user, the changes in visualization with changes in data, and the way the user interacts with data through the visualization.

We are about to start on a fabulous journey of discovery with creating rich data visualizations with D3.js, and focusing on project-based learning of D3.js through practical examples. We will start out with the basic concepts, and then move through various examples of creating living data visualizations with D3.js.

In this first chapter, we will start with a brief overview of several of the concepts in D3.js, create a minimal D3.js application, and examine several of the tools that you can use to build D3.js applications.

Specifically, in this chapter, we will cover the following topics:

- A brief overview of D3.js
- The key design features of D3.js, including selection, data management, interaction, animation, and modules
- An introduction to development tools to get you going quickly with D3.js
- A simple Hello World program using D3.js
- Examining the DOM generated by D3.js with the Google Chrome Developer tools

A brief overview of D3.js

D3.js is a JavaScript library for manipulating DOM objects based upon data. By using D3.js and modern browsers, specifically those which can display and manipulate SVG, you can create rich visualizations of data. These visualizations not only visualize the data, but can also include descriptions to change what is shown to the user based upon the changes in the data, and the way in which the user can interact with the visuals which represent the data.

[ You can get D3.js at <http://d3js.org>.]

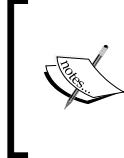


D3.js differs from other data visualization frameworks such as **Processing** (<https://processing.org/>) in that it provides a domain-specific language for transforming the DOM based upon data, whereas tools like Processing provide a lower level direct rendering model. D3.js lets you describe the means of visualizing the data instead of coding all of the specific details to draw the pixels of the visualizations. This facilitates easy creation of visualizations by allowing D3.js to worry about the details on rendering the data, based on the standards of SVG and CSS.

A fundamental concept in D3.js is the ability to easily manipulate the DOM in a web document. This is often a complicated problem, and many frameworks (such as **jQuery**) have been created to perform this task. D3.js provides capabilities similar to jQuery, and for those familiar with jQuery, much of D3.js will feel familiar.

But D3.js takes what libraries like jQuery provide and extends them to provide a more declarative nature of modifying the DOM to create visuals based on the structure of the data instead of simply being a framework for low level DOM manipulation.

This is important, as data visualization requires more than an ability to simply modify the DOM; it should also describe how the DOM should be changed when data is modified, including the way it changes when the user interacts with the visual elements representing the data.



We will not cover jQuery in this book. Our focus will purely be on how we can manipulate the DOM using the facilities provided by D3.js. We will use D3.js constructs to apply styles instead of depending on CSS. All of this is to exemplify how to use the facilities of D3.js instead of hiding any of it with other tools.

We will examine many concepts in D3.js in detail, but let's start with a few high-level ideas in D3.js that are worth mentioning first.

Selections

The core operation in D3.js is **selection**, which is a filtered set of DOM elements queried from the document. As the data changes (that is, it is either loaded or modified), the result of the selection filter is changed by D3.js based on how the data was changed. Hence, the visual representation also changes.

D3.js uses the W3C selectors API (<http://www.w3.org/TR/selectors-api/>) for identifying the items in the DOM. This is a mini-language consisting of predicates that can filter the elements in the DOM by tag, class, id, attribute, containment, adjacency, and several other facets of the DOM. Predicates can also be intersected or unioned, resulting in a rich and concise selection of elements.

Selections are implemented by D3.js through the global namespace `d3`, which provides the `d3.select()` and `d3.selectAll()` functions. These functions utilize the mini-language and return, respectively, the first or all items matching the specification. Using the result of these selections, D3.js provides additional abilities for modifying those elements based upon your data using a process known as **data binding**.

Data and data binding

The data in D3.js is **bound** to the DOM elements. Through **binding**, D3.js tracks a collection of objects along with their properties, and based upon rules that you specify, it modifies the DOM of the document based upon that data. This binding is performed through various operators provided by D3.js, which can easily be used to describe the mapping of the visual representation of the data. At this point, we'll introduce the three stages of data binding, and dive into more details on the process in *Chapter 2, Selections and Data Binding*.

The process of binding in D3.js consists of three stages: **Enter**, **Update**, and **Exit**. When performing a selection for the first time with D3.js, you can specify the data that is to be bound and needs to be entered. You can also specify the code to be executed for each of these stages.

When data is first joined into a selection, new visuals will need to be created in the DOM for each data item. This is performed using the `enter()` process which is started by calling the `.enter()` function. Code that you specify after the `.enter()` function will be used to specify each and every piece of data that is represented visually, and D3.js will use this code to automatically generate the DOM that is required instead of you needing to code it all in detail.

As the application modifies this bound data, we will execute the selection repeatedly. D3.js will make a note of the existing visuals and the data they are bound to, and allow us to make modifications to the visuals based upon the way the data changed.

If data items are removed, we can use the D3.js `.exit()` function in a selection to inform D3.js to remove the visuals from the display. Normally, this is done by telling D3.js to remove the associated DOM elements, but we can also execute animations to make the removal demonstrate to the user how the visual is changing instead of a jarring change of display.

If we create a selection without an explicit reference to `.enter()` or `.exit()`, we are informing D3.js that we want to potentially make modifications to the visuals that are already bound to the data. This gives us the chance to examine the properties of each data item and instruct D3.js on changing the bound visuals appropriately.

This separation of enter, update, and exit processes allows for very precise control of the visual element lifecycle. These states allow you to update visuals as the data changes either internally or through user interaction. It also gives you the ability to provide well-defined transitions or animations for each of the three states, which are essential for dynamic data visualizations that demonstrate data not simply statically but also how it changes through motion.

Interaction and animation

D3.js provides facilities for animating the visual elements based upon the changes in data or upon events created by the user such as mouse events. These are performed by integrating with the events in the DOM using the `.on()` function as part of a selection.

D3.js event handlers are similar to those provided by jQuery. However, instead of just calling a function, they also expose the bound data item to the function, and if you want, the index of the data item in a collection. This saves us from having to write code that looks up the data item based upon things like mouse positions, and therefore, greatly simplifies our code.

Additionally, through integration with the enter, update, and exit selection processes, we can declaratively code scene transitions in each of these scenarios. These transitions expose the `style` and `attr` operators of the selections. Any changes that we make to those properties are noticed by D3.js, which will then apply an **interpolator** to transition the property values from the previous to the new values over a given period of time.

By using interpolation, we can avoid coding the repeated changes in the values of the visual properties (such as location and color) at each step of the animation. D3.js does all this for us automatically!

Additionally, D3.js automatically manages the scheduling of animations and transitions. This removes the need for you to manage complicated concurrency issues and guarantees exclusive access to the resources for each element along with highly optimized animation through a shared timer managed by D3.js.

Modules

D3.js provides a number of **modules** of prebuilt functionality for helping us code many of the things that we need to do for creating rich and interactive data visualizations. These modules in D3.js are grouped into a number of generated categories based upon the capabilities provided to the programmer.

- **Shapes:** The shapes module gives us numerous prebuilt visuals including, and not limited to, lines, arcs, areas, and scatterplot symbols. By using D3.js shapes, we can simply add the geometric renderings to the visualization, and not worry about drawing each in detail, pixel by pixel.
- **Scales:** This module gives us a means of converting data values into coordinates within the browser. These save us from coding repetitive, complex, and often error-prone translations by providing them out of the box. They also provide the basis for generating the visuals for axes, again saving us much effort in rendering visuals that would otherwise be complicated.
- **Layouts:** The layouts module gives us the tools to easily (if not automatically) calculate the visual relationships between the elements in our visualizations. This is often the most complicated part of data visualizations, and D3.js provides us with many prebuilt hierarchical and physical layouts that make our lives as programmers much simpler.
- **Behaviors:** This module provides implementations of the common user interaction patterns. An example would be a selection behavior that implements listening to the mouse events on visual elements, and changes the presentation of the item to represent that the user has selected it.
- **Data-processing modules:** D3.js also includes various data-processing utilities such as nest and cross operators, and parsers for data in formats, such as CSV, JSON, TSV, and for data, in date and number formats.

We will discuss these modules in detail in their dedicated chapters.

Tools for creating and sharing D3.js visualizations

D3.js applications can be built using many, if not any, web development tools. The choice of tool is often dependent upon the individual coder, as each platform (.Net, Node.JS, Ruby on Rails, and so on) provides their own (and many third-party) tools.

This book will not be prescriptive and specify editors. Instead, it will generally refer you to the online and functional examples of all the code, and leave it to the readers to reproduce them in their own development environment.

The examples in this book will be delivered using a combination of **Js Bin** (<http://jsbin.com/>) and **bl.ocks.org** (<http://bl.ocks.org/>), and we will use the Google Chrome Developer tools for examining the DOM in our examples. A brief introduction to each is therefore worthwhile, as each example in this book will be linked to an example on bl.ocks.org, which itself will contain a link to the code in Js Bin for you to play with dynamically.

Js Bin

Js Bin (<http://jsbin.com/>) is a website that functions as a development tool for facilitating the quick creation and sharing of simple JavaScript applications that run within the browser. It provides many features, including saving and sharing of HTML and JavaScript, real-time update of the UI while you are editing, and a very cool ability to push your code and data to GitHub.



GitHub is a free code sharing and source code management tool. If you are not familiar with it, check it out at <http://www.github.com>.

I think that Js Bin provides one of the least-friction means of getting up and coding with D3.js. You can simply go to the website, start editing in HTML, CSS, or JavaScript, and see the results as you type in the browser pane. No need for installing any development tools or web servers!

Getting Started with D3.js

As an example of Js Bin, the following link will take you to the first of our examples, the canonical Hello World application written purely in HTML. <http://jsbin.com/zimeqe/edit?html,output>.



The screenshot shows the Js Bin interface. At the top, there's a navigation bar with 'File', 'Add library', 'Share', and tabs for 'HTML', 'CSS', 'JavaScript', 'Console', and 'Output'. On the right, there are links for 'Account', 'Blog' (with a red notification dot), and 'Help'. Below the navigation, the 'HTML' tab is selected, displaying the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="description" content="D3byEX 1.1">
  </head>
  <body>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <script>
      d3.select('body')
        .append('h1')
        .text('Hello World!');
    </script>
  </body>
</html>
```

To the right of the code, under the 'Output' tab, the text 'Hello World!' is displayed in a large, bold, black font. Above this text are two buttons: 'Run with JS' and 'Auto-run JS' (which is checked). Below the output text is a small box containing 'd3byex' and '12 Nov'.

[ Don't worry right now about the code embedded in HTML in this demonstration. We will again look at this example along with more complicated ones later in this chapter.]

The preceding screenshot displays a single bin, a combination of HTML, CSS, and JavaScript that is stored within Js Bin's servers. The Js Bin user interface provides multiple tabs/panes for the HTML, CSS, JavaScript, Console, and HTML output from the code in the bin. With **Auto-run JS** selected, the output will be regenerated on every interactive change to any of the code.

This makes Js Bin excellent for interactively demonstrating and creating D3.js visualizations.

bl.ocks.org

bl.ocks.org (<http://bl.ocks.org>) is a service for D3.js code examples that you place on GitHub, a free source code and sharing repository, in an entity known as a gist. A gist is simply one or more reusable and sharable piece of code that are managed by GitHub. They are an excellent means of remembering and sharing small code examples.

bl.ocks.org was created by Mike Bostock, the original creator of D3.js. It is able to create great D3.js visualizations using gists, provided that the gist itself is a piece of D3.js code. Many, if not most, D3.js examples on the Web are presented as examples on bl.ocks.org, and this book will follow this model.

For a demonstration, open <http://bl.ocks.org/d3byex/ed79b9fee311091333d6>, which takes you to a bl.ock.org version of the **Hello World** example. Opening the link will present you with the following content.

The screenshot shows a screenshot of a bl.ock.org gist. At the top, it says "d3byex's block #ed79b9fee311091333d6 November 12, 2015". Below that is the title "D3byEX 1.1". Under the title is the heading "Hello World!". Below the heading is a note "Live code available at [JSBIN.COM](#)". To the right of the note is a link "Open in a new window.". The main content area contains the following code:

```
#index.html

<!DOCTYPE html>
<html>
  <head>
    <meta name="description" content="D3byEX 1.1 Hello World">
  </head>
  <body>
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <script>
      d3.select('body')
        .append('h1')
        .text('Hello World!');
    </script>
  </body>
</html>
```

This bl.ock follows a pattern that will be used throughout the book. Each example will be in its own bl.ocks.org and consist of a title, the D3.js code in operation, a link to live code on Js Bin, and then the HTML and any data that is in use in the example.

Getting Started with D3.js

At the very top of the page, there is a link that you can click which will also take you to the gist on GitHub.



The link displayed in the preceding screenshot takes you to the following page at <https://gist.github.com/d3byex/ed79b9fee311091333d6>

A screenshot of the GitHub Gist page for "d3byex / README.md". The page has a header with "GitHub Gist" and a search bar. It features a main area for sharing code, notes, and snippets. The gist title is "d3byex / README.md" and it was created 28 days ago. It includes tabs for "Code" and "Revisions 1". There are buttons for "Embed", "Download ZIP", and "blocks". The "Code" tab shows two files: "README.md" and "index.html". The "README.md" file contains a note about live code available at JSBIN.COM. The "index.html" file contains the following code:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta name="description" content="D3byEX 1.1 Hello World">
5    </head>
6    <body>
7      <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
8      <script>
9        d3.select('body')
10          .append('h1')
11          .text('Hello World!');
12      </script>
13    </body>
14  </html>
```

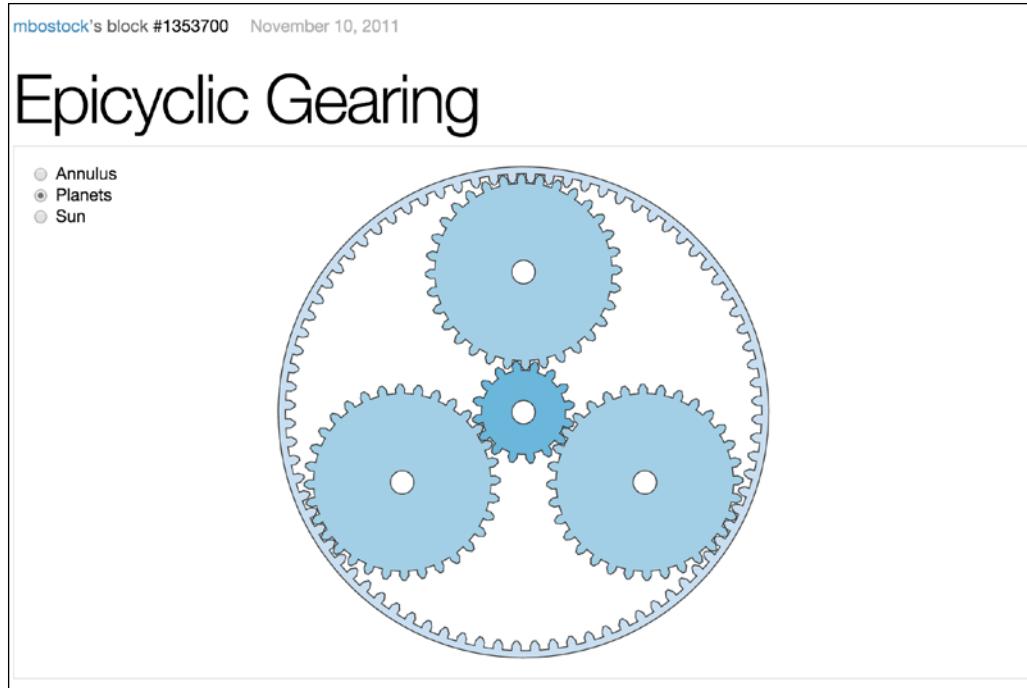
This code is not dynamic like the one on Js Bin, but you can click on the **Download Zip** button, and all the files in the gist get downloaded to your system as a ZIP file.

Google Chrome and Developer tools

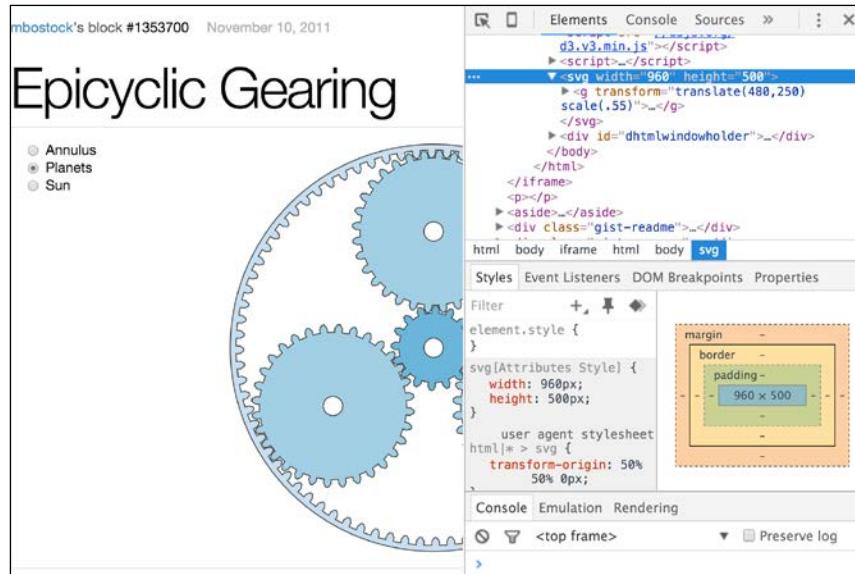
D3.js applications can be developed in any number of tools. In this book, we will use Google Chrome as a browser, and use its embedded development tools. You can also use Firefox or Internet Explorer and their respective development plugins. Theoretically, all the examples will run identically in all three browsers, but have only been tested in Google Chrome.

You can access the developer tools from the Chrome settings button, or by using the key combination of *option + command + I* (on Mac) or *Ctrl + Shift + I* (on Windows). Pressing the *F12* button also takes you to the Chrome Developer tools on a Windows platform.

The following screenshot demonstrates the Google Chrome Developer tools open on the **Epicyclic Gearing** block at <http://bl.ocks.org/mbostock/1353700>



On opening the developer tools, you will be presented with a panel that opens in the browser which displays the details of the content on the page.



In this case the pane opens on the right (you can configure the location where it opens), and displays the HTML for the page with the main SVG element of the page highlighted. While selecting nodes in HTML, the tools will highlight that element in the web page, and display the selected details for the element, in this case, the styles. We will use these tools in *Chapter 2, Selections and Data Binding*, to demonstrate how D3.js binds data to the DOM elements, and in later chapters to understand how.

Hello World – D3.js style

Now let's apply what we have learned in this chapter by stepping through an example, and see how we use D3.js to modify the DOM. The example will be the same as the one we just saw in the previous section; we'll walk through it to see how it functions.

The following is the entire HTML for the application:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="description" content="D3byEX 1.1">
  </head>
  <body>
```

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



block (1.1): <http://goo.gl/7KkIuC>



The code appends a level one header using an `h1` tag to the `body` tag of the document. The `h1` tag then has its content set to the text `Hello World`. And as we saw earlier, the output in the browser looks like the following screenshot:

There are two primary parts to this application, both of which we will see in almost every example. The first part includes a reference to the D3.js script, which is performed with the following code placed just inside the `<body>` tag:

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

This references the minified D3.js file directly from the D3.js (<http://d3js.org/>) website. You can also copy this file and place it locally on your web server or in a Web project. Since all the examples in this book are online, we will always use this URL.

Note that we also have to specify `charset="utf-8"`. This is normally not required for most JavaScript libraries, but D3.js is UTF-8 encoded and not including this can cause issues. So, make sure you don't forget this attribute.

The actual D3.js code in this example consists of the following three functions placed within another `<script>` tag within the body of the document.

```
d3.select('body')
  .append('h1')
  .text('Hello World!');
```

Let's examine how this puts the text in the web page.

```
d3.select('body')
```

All D3.js statements will start with the use of the `d3` namespace. This is the root of where we start accessing all the D3.js functions. In this line, we call the `.select()` function, passing its `body`. This is telling D3.js to find the first body element in the document and return it to us for performing other operations upon it.

The `.select()` function returns a D3.js object representing the body DOM object. We can immediately call `.append('h1')` to add the header element inside the body of the document.

The `.append()` function returns another D3.js object, but this one represents the new `h1` DOM element. So all we need to do is to make a **chained** call: `.text('Hello World!')`, and our code is complete.

This process of calling functions in this manner is referred to in D3.js parlance as **chaining**, and in general, it is referred to as a **fluent** API in programming languages. This chaining is the aforementioned mini-language. Each chained D3.js function call further specifies the operation, allowing you to very easily describe how you want to modify the DOM through chained method calls.

This sometimes feels strange to those who have not had experience in using a fluent syntax, but once you get used to it, I guarantee that you will see the reason behind using this type of syntax. As we will see through the examples that we cover, this provides us with a very concise means of declaratively instructing D3.js on what we want in our visualization.

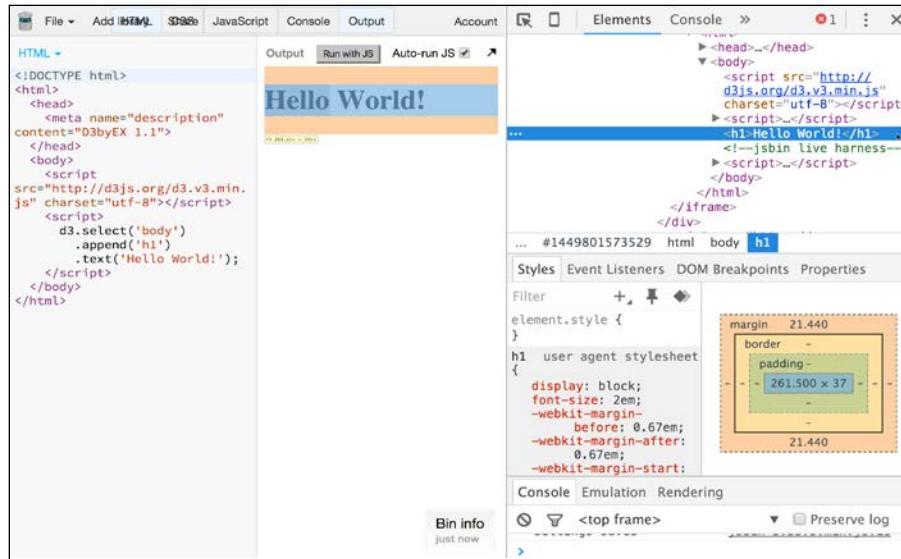
For those familiar with jQuery, this syntax will look familiar. An equivalent piece of code could be written in JQuery as `$('body').append('h1').text('Hello World');`

But as we will see in more complex examples, the features provided by D3.js will give us much more power to create data visualizations than can be done with jQuery.



Examining the DOM generated by D3.js

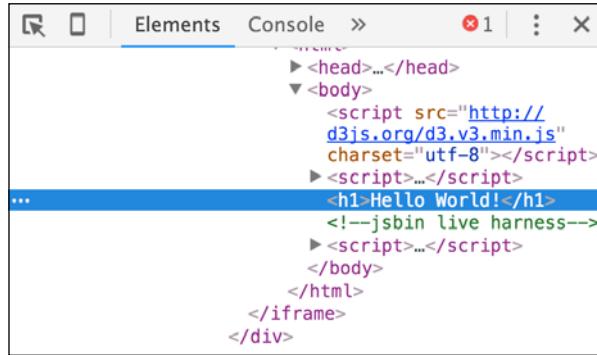
Now let's take a quick look at the DOM that was created by this code using the Chrome Developer Tools. Open the developer tools using the instructions given earlier in the chapter. I prefer mine to be displayed to the right of the page, and the book will follow this convention.



Since this example (and all the examples in this book) is hosted within Js Bin, there is a bunch of HTML content generated automatically and injected into our page by the backend of Js Bin. To find the element corresponding to the text generated by our code, you can drill through the DOM in the explorer. Otherwise, you can right-click on the element in the output pane of the browser, and select **Inspect Element** as seen in the following screenshot:



Then you can move directly to the element in the developer tools.



In the preceding screenshot, we can visually verify that the `<body>` tag had a new `<h1>` tag added with the text as we desired.

Summary

In this chapter, we looked at several high level concepts in D3.js: selections, data, interactions and animation, and modules. Then we briefly covered several of the tools that can be used to build D3.js applications, and which will be used in examples that follow in the remainder the book: Js Bin, bl.ocks.org, Google Chrome, and Google Chrome Developer tools. We closed the chapter with a very simple example that demonstrates how to include D3.js in your application and performing a simple selection that inserts content into the web page.

In the next chapter, we will expand upon the concept of selection and use it to bind data to visual elements in the DOM. We will expand our use of D3.js to create and modify DIV elements. Further on in *Chapter 3, Creating Visuals with SVG*, we will get into the real power of D3.js by using it to manipulate SVG.

2

Selections and Data Binding

In this chapter, you will learn how to use D3.js to select and manipulate the DOM of an HTML page based upon data. Rendering of visuals in D3.js takes a declarative approach, where you inform D3.js of how to visualize a piece of data instead of imperatively programming exactly how to draw the visual and iterate across the data. This process is referred to as **selection** and **data binding** in the D3.js nomenclature.

To demonstrate how D3.js can be used to create DOM elements driven by data, we will progress through a number of examples that demonstrate creating DIV elements to display various arrays of integer values. We will first examine how selection can be used to extract the existing DOM elements, and how D3.js is used to associate the data to each DOM element. Then we will examine the ways to instruct D3.js to create new DOM elements from the data. That will be followed by discussing the procedure for updating the existing elements, and for removing visual elements when particular data items are removed.

We will focus purely upon the HTML DOM elements, and will progress to using SVG in later chapters. Specifically, we will progress through the following topics in this chapter:

- Using D3.js selections to modify DOM elements
- Modifying the style of DOM elements using D3.js selectors
- Binding data to the DOM using `.data()`
- Using `.enter()` for creating DOM elements from new data items
- Updating the existing DOM elements based upon the changes in data
- Using `.exit()` to remove DOM elements when the associated data is no longer to be visualized
- A laundry list of tips on performing data binding with D3.js

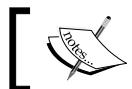
D3.js selections

At its core, D3.js is about selection, which is a process of finding and creating DOM elements that visualize data. At a simple level, a selection can just be a means of finding and manipulating elements in the DOM that already exist. However, D3.js selections can also be used for explicitly creating new elements in the DOM as well as for implicitly creating and removing DOM elements based upon the changes in an underlying data model.

In *Chapter 1, Getting Started with D3.js*, we saw a simple example of selection in which we used selection to make a D3.js version of the canonical Hello World application. Now we will dive deeper into the power of selections. We will look at two examples of selecting a DOM element and changing its style.

Changing the style of a DOM element

In this first example, we will create a page with four `div` elements, each with a unique ID. We will then use D3.js to find the first `div` tag, and change its background color.



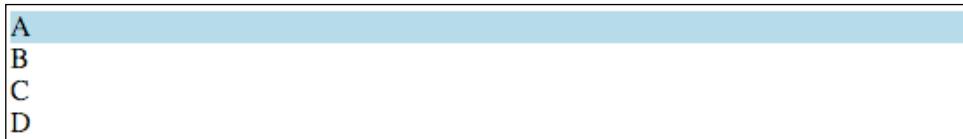
block (2.1): <http://goo.gl/EnAQBC>



The `body` tag of the document contains the following code:

```
<div id='div1'>A</div>
<div id='div2'>B</div>
<div id='div3'>C</div>
<div id='div4'>D</div>
<script>
  d3.select('div').style('background-color', 'lightblue')
</script>
```

The result of this preceding code is as follows:



This example uses the `d3.select()` function, which returns the first element in the DOM that matches the given tag—in this case, '`DIV`'. The result of `d3.select()` is a D3.js object representing the DOM element that was identified and the data that D3.js has associated with that element.

This concept in D3.js is referred to as a **selector**. The function `d3.select()` always represents a single DOM element or a null value if the element is not found.

A selector has methods such as `.style()`, which can be used to change the CSS style properties of the underlying element, attributes using `.attr()`, and the text property using the `.text()` function.

In this case, we use the `.style()` function to set the `background-color` property of the `DIV` elements style to `lightblue`.

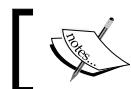
Changing the style of multiple items

To select multiple items in the DOM, we can use the `d3.selectAll()` function. The result is a selector which can represent multiple DOM elements that match the criteria.

To demonstrate, we will change the single line of the D3.js code in our previous example to the following:

```
d3.selectAll('div').style('background-color', 'lightblue')
```

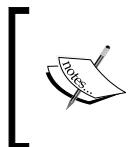
As a result of this, the call to `.selectAll()` will represent each of the four `div` elements in the document. The call to `.style()` will be applied to each of the DOM element represented, which results in the following output:



block (2.2): <http://goo.gl/61p8Nv>



This demonstrates one of the advantages of using D3.js for selection. Chained function calls will be applied to all DOM elements resulting from a D3.js selection. Therefore, we do not need to explicitly iterate through all the items. This saves us from excessive coding, and helps in reducing the potential of errors.



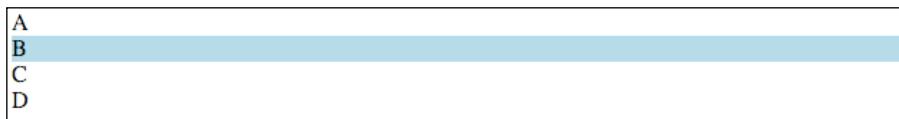
Note that, by default, the items in a selector are fixed at the time of the creation of the D3.js selector. If we were to add another `div` after the selection, then the elements in the existing selector will not have the new `div` tag added.



The parameter passed to the functions `d3.selectAll()` and `d3.select()` can also include various CSS rules as part of the query. As an example, to select all the elements with a specific ID, prepend the parameter with `#`. The following example selects only those DOM elements whose id is `div2`:

```
d3.selectAll('#div2').style('background-color', 'lightblue')
```

This results in the following output:



bl.ock (2.3): <http://goo.gl/TC4Yox>



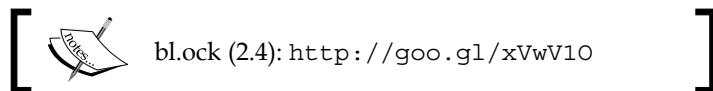
Note that this selection will return all the DOM elements which have the ID `div2`, be they `div` or other types of DOM elements. This example only has `div` tags, so that is all that we will retrieve. Moreover, it is bad practice to have identical ID values on a page. But the way in which the query functions is viable.

If we want to ensure that this query returns only `div` elements, then we can use the following query, which places the type of the element before the hash symbol:

```
d3.selectAll("div#div3").style('background-color', 'lightblue')
```

The preceding query has the following result:

A
B
C
D



Now let's examine the scenario where we would like to apply a different style to each DOM element in the selector. To do this, we can pass an accessor function to the `.style()` instead of a value. For example, the following code will alternate the color of the background of the `div` tags between `lightblue` and `lightgray`.

```
d3.selectAll("div")
  .style('background-color', function (d, i) {
    return (i % 2 === 0) ? "lightblue" : "lightgray";
});
```

The preceding code results in the following output:

A
B
C
D



Accessor functions are commonly used through D3.js. An accessor function has two parameters, the first of which represents the datum that has been associated by D3.js to the DOM element (we'll come back to this later in the chapter). The second parameter represents the 0-based array position of the DOM element in the result of the selection.



The second parameter of an accessor function is optional.

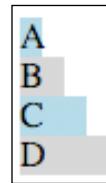


The return value of the selector function is another selector (or the same selector) in many cases. This allows us to chain the method calls together. We can do this to conveniently set multiple styles on all the DOM elements represented by the selector.

As an example, the following code first sets the background color, and then sets the width of each DIV to an increasing value:

```
d3.selectAll("div")
  .style('width', function(d, i) {
    return (10 + 10 * i) + "px";
  })
  .style('background-color', function (d, i) {
    return (i % 2 === 0) ? 'lightblue' : 'lightgray';
 });
```

The output for the preceding code will be as follows:

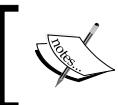


bl.ock (2.6): <http://goo.gl/ukFFYL>



Multiple style properties can also be set in a single call to `.style()` by passing a hash of property names and values. The following has the same result as the previous example:

```
d3.selectAll("div").style({
  width: function (d, i) { return (10 + 10 * i) + "px" },
  'background-color': function (d, i) {
    return (i % 2 === 0) ? 'lightblue' : 'lightgray';
  }
});
```



bl.ock (2.7): <http://goo.gl/17FVJs>. The image of the output is omitted as it is a duplicate of the previous bl.ock.



D3.js and data binding

The example in the previous section relied upon the elements that already exist in the DOM. Normally, in D3.js we would start with a set of data, and then build visualizations based on this data. We would also want to change the visualization as the data changes as a result of either adding more data items, removing some or all of them, or changing the properties of the existing objects.

This process of managing mapping of data to visual elements is often referred to as **binding of data**, and in terms of the D3.js nomenclature, it is referred to as a **data join** (do not confuse this with an SQL join). Binding in D3.js is performed by using the `.data()` function of a selector.

Let's dive in, and examine a few examples of binding data in some detail.

Data binding

Binding of data can be one of the hardest things for someone new to D3.js to get used to. Even for somebody who uses other languages and frameworks that provide data binding, the way in which D3.js binds data is a little different, and getting to know how it does so will save a lot of time down the road. Therefore, we will take the time to examine it in detail as it is essential for creating effective D3.js visualizations.

In D3.js, we drive the visualization of data through binding using the following functions of a selector.

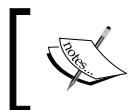
Function	Purpose
<code>.data()</code>	Specifies the data to be used to drive the visualization
<code>.enter()</code>	Returns a selector representing the new items to be displayed
<code>.exit()</code>	Returns a selector representing the items that are no longer to be displayed

This pattern in which test functions are used is so ingrained in the D3.js code that it is often referred to as the **enter/update/exit** pattern or **general update** pattern. It provides a powerful means of declaratively telling D3.js how you want the dynamic data to be displayed, and to let D3.js handle the rendering.

We will come back to these details of the specifics of enter/update/exit in a little bit. For now, let's start by examining our selection example from earlier in the chapter, where we selected all the `div` objects in the document. This will help us understand the basis of how a selector facilitates the rendering process.

We will use a slight variant on the `d3.selectAll()` function from the previous example. Here, we will assign the result to a variable named `selector`:

```
<div id='div1'>A</div>
<div id='div2'>B</div>
<div id='div3'>C</div>
<div id='div4'>D</div>
<script>
  var selector = d3.select('body')
    .selectAll('div');
</script>
```



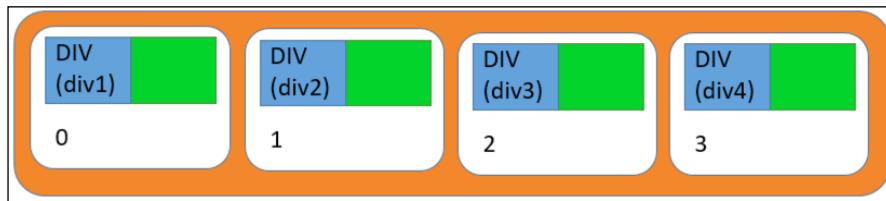
bl.ock (2.8): <http://goo.gl/etDgJV>. The output is not shown as the code does give results visually different from the previous examples.



There are two other subtle differences in this preceding statement from the previous examples. The first is that we select the body DOM element, and the second is that we chain a call to `.selectAll()` for the `div` tags.

Using this pattern of a function chain, we are instructing D3.js to select all the `div` tags that are a child of the `body` tag. This chaining of select function calls allows us to navigate through the HTML document to look for tags in specific places, and as we will see shortly, specify where to put the new visual elements.

To help conceptualize a selector, I believe that a selector can be thought of as a collection of mappings between the DOM elements and the data that D3.js has associated with those element(s). I find it useful to mentally picture a selector with diagrams such as the following:



The orange part in the preceding diagram represents the overall selector that results from our selection. This selector contains four items represented by white, rounded rectangles, one for each `div`, and which we can think of as being numbered from 0 through 3.



Do not confuse a selector with an array – the individual elements in this diagram cannot be accessed using `[]`.



The ordering is important as we will see when we update the data. By default, the ordering depends on how the identified DOM elements are ordered in the DOM at the point of selection (in this case, children of the `body` tag).

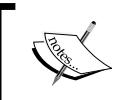
Each item in a selector can then be thought of as consisting of two other objects. The first is the actual DOM element that was identified by the selection, represented by a blue square in the preceding diagram. Inside that square in the image is the DOM element type (`div`), and the value of its `id` property.

The second is the datum that D3.js has associated with that DOM element, represented by the green square. In this case, there is no data that is bound at this point by D3.js, so the data for each is `null` (or empty in the diagram). This is because these DOM elements were created in HTML and not with D3.js, and hence there is no associated datum.

Let's change that and bind some data to these `div` tags. We do this by chaining a call to `.data()` immediately following the selection functions. This function is passed a collection of values or objects, and it informs D3.js that you want to associate each datum with a specific visual representation created by the function calls that follow.

To demonstrate this, let's modify the code to the following, binding the array of integers to the `div` tags:

```
var selector = d3.select('body')
    .selectAll('div')
    .data([10, 20, 30, 40]);
```

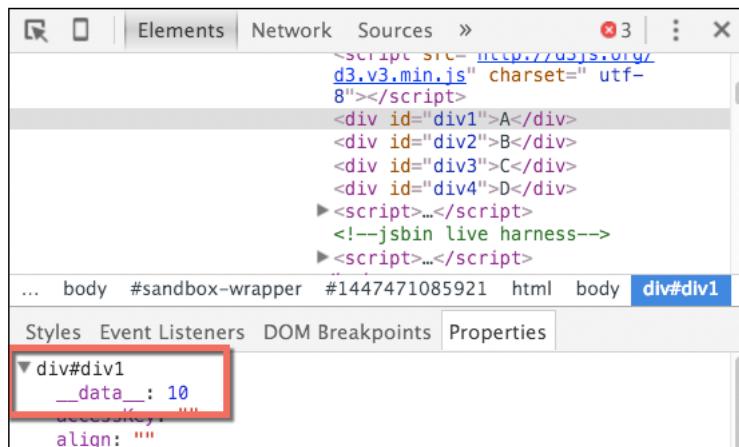


block (2.9): <http://goo.gl/h1O1wX>. The output is omitted from the book as it is not visually different from the previous example.



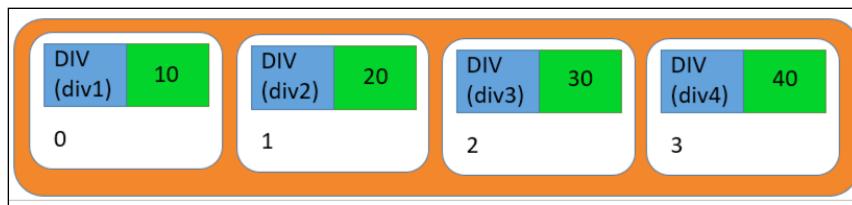
The result of chaining the call to `.data()` tells D3.js that for each item identified in the selector, the datum at the same index in the data should be assigned. In this example, this does not change the visual. It simply assigns a datum to each `div` element.

To check this, let's examine the result using the developer tools. If you right-click on A in the browser, and select inspect item, the tools will open. Next, open the properties panel, as shown in the following screenshot:



The highlighted red rectangle in the preceding screenshot shows that the `div` tag now has a `__data__` property, and its value is 10. This is how D3.js binds data to the visuals, by creating this property on the DOM element and assigning the datum. If you examine the three other `div` tags, you will see that they all have this property and the associated value.

Using the visual for our selector, we get the following values:



Now you might ask what happens if the count of items in the call to `.data()` does not equal the amount of items in the selector? Let's take a look at those scenarios, starting with the case of fewer data items than the selected DOM elements:

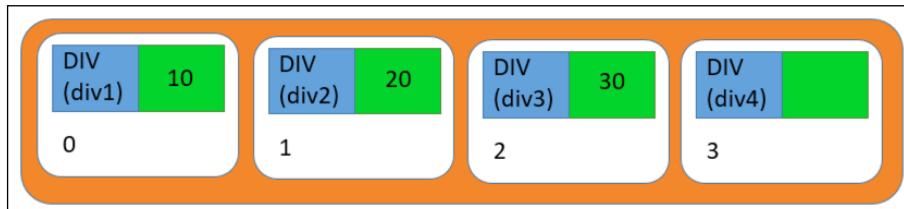
```
var selector = d3.select('body')
    .selectAll('div')
    .data([10, 20, 30]);
```



block (2.10): <http://goo.gl/89NReN>. The output has been omitted again, since the visual did not change.

If you open the Developer tools after running this example, and examine the properties for each of our `div` tags, you will notice that the first three have a `_data_` property with the values assigned. The fourth tag does not have the property added. This is because D3.js iterates through the items in the data, assigning them one by one, and any extra DOM elements in the selector are ignored.

Conceptually, the selector then looks like following:



Now let's change the code to have more data items than the DOM elements:

```
var selector = d3.select('body')
    .selectAll('div')
    .data([10, 20, 30, 40, 50]);
```



block (2.11): <http://goo.gl/CvuxNJ>. The output has been omitted again since the visual did not change.

Examining the resulting DOM in the Developer tools, you can see that there are still only four `div` elements, with 10 through 40 assigned respectively. There is no new visual created for the extra data item.

```
▼<body>
  <script src="http://d3js.org/d3.v3.min.js" charset=" utf-8"></script>
  <div id="div1">A</div>
  <div id="div2">B</div>
  <div id="div3">C</div>
  <div id="div4">D</div>
```

Why is a visual not created in this case? It is because the call to `.data()` assigns data only to the existing visual elements in the selector. Since `.data()` iterates the items passed to it, it stops at the last item, and the extra DOM elements are ignored.

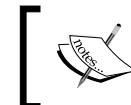


We will examine how we add visuals for these stray data items in the next section.



There is one more case that I think is worth examining. The examples so far for `.data()` have had pre-existing `div` tags in the document. Let's now try binding some data items when there are no existing `div` tags. The body of code for this is as follows:

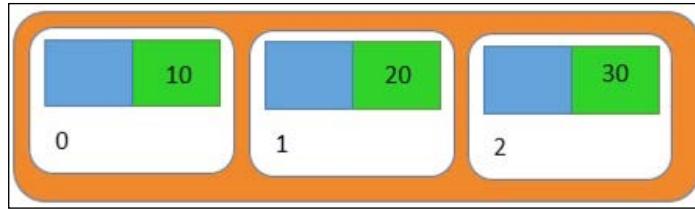
```
var selector = d3.select('body')
  .selectAll('div')
  .data([10, 20, 30]);
```



block (2.12): <http://goo.gl/5gsEGe>. The output has been omitted as there are no visuals.



This does not create any DOM elements, since we do not chain any functions to create them after `.data()`. However, the variable `selector` is a valid selector with three items. In our visual, it would look like the following diagram, where the blue squares are empty:



If you take a look at the output created on the console, you will see that this selector indeed has an array of three items:

```
[ [undefined, undefined, undefined] ]
```

The output does not necessarily show the data, but it does demonstrate that the selector consists of three items. Our conceptual model shows more, but it is only a conceptual model after all, and intended for understanding and not for representing the underlying data structures.

Now let's see how we instruct D3.js to create some visuals for the data items to fill in those blue squares, and put something on the screen.

Specifying the entering elements with `.enter()`

To create visuals with D3.js, we need to call the `.enter()` method of the selector after the call to `.data()`. Then we chain the other method calls to append one or more DOM elements, and normally, also call various functions for setting the properties of those DOM elements.

To exemplify the use of `.enter()`, let's take a look at the last example from the previous section, where we started without any `div` tags in the body and used D3.js to bind three integers:

```
var selector = d3.select('body')
    .selectAll('div')
    .data([10, 20, 30]);
```

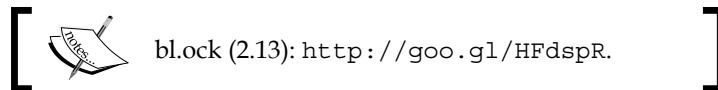
Now using the selector variable, we call the `.enter()` function and assign it to a variable named `entering`:

```
var entering = selector.enter();
```

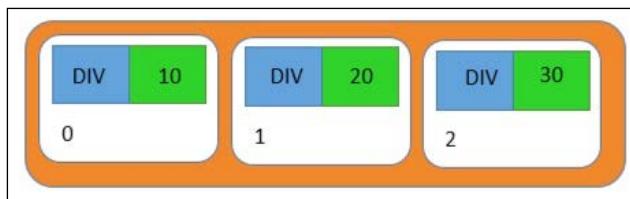
The value of `entering` will represent the new items in the selector that need to be created. `selector` did not have any `div` tags selected, and since we bound to three items, this variable represents the three new items in the selector that need to be created.

We can then use the `entering.append('div')` and `.text(function(d) { return d; })` functions to specify how to render the visuals for each item:

```
entering.append('div')
    .text(function(d) { return d; });
```



After execution, the value of `selector` contains three items, with both values assigned and the DOM elements created:



The resulting output on the page will be as follows:

```
10
20
30
```

Examining the resulting DOM, we see that three `div` tags have been created:

```
▼<body>
  <script src="http://d3js.org/
  d3.v3.min.js" charset=" utf-8"></script>
  ▶<script>...</script>
    <div>10</div>
    <div>20</div>
    <div>30</div>
```



I will leave it as an exercise for you to examine the properties of these elements for verifying the creation of the `__data__` property and assignment of the values.

Adding new items using .enter()

Now that we have created DOM elements from data without any existing visuals, let's change the code to update the data by adding a new datum upon the press of a button.

In D3.js, data which need new visuals created are said to be in a state referred to as *entering*. After calling `.data()`, we can call the `.enter()` method on that same resulting selector. This method identifies the items in the selector that are entering, and hence require visuals to be created. We then simply chain methods on the result of `.enter()` to tell D3.js how each data item should be visualized.

Let's change our code a little bit to demonstrate this in action.



This code makes a few modifications to the previous example. First we add a button that can be pressed. This button will call a function named `render()` and pass an array of four values to it, the first three of which are identical in value. There also exists a new datum at the end:

```
<button onclick='render([10, 20, 30, 40])'>Take action!</button>
```

The `render` function itself does the selection and creation of the new visual elements, but it uses the values passed to the function instead of a hard-coded array of values.

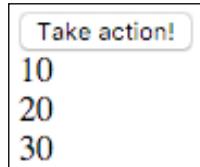
```
function render(dataToRender) {
  var selector = d3.select('body')
    .selectAll('div')
    .data(dataToRender);

  var entering = selector.enter();
  entering.append('div')
    .text(function(d) { return d; });
}
```

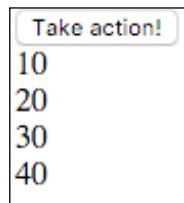
When the page is first loaded, we call `render`, telling it to create elements in a different array.

```
render([10, 20, 30]);
```

The initial page that is loaded will contain the following content:



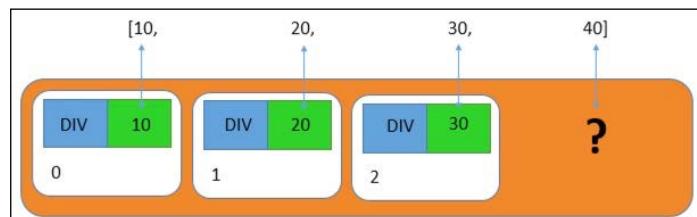
When we press the button we call render again, but pass it four values. This results in the content on the page changing as follows:



This may appear as if the previously existing div tags were replaced with four new ones, but what happens is actually more subtle. The second time that render() is called, the call to .selectAll('div') creates a selector that has three items, each of which has DOM elements and their bound data:



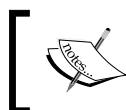
Then, .data([10, 20, 30, 40]) is executed. D3.js iterates this array, and it compares the value of each datum to the item in the selector at the same index. In this case, the items at positions 0, 1, and 2 have the values 10, 20, and 30, which are each equal to the values at the same position in the data. Therefore, D3.js does not do anything to these items. But the fourth value, 40, does not have an associated item in the selector.



Therefore, D3.js will create a new item in the selector for the datum 40, and then apply the functions for creating the visuals, resulting in the following:



D3.js has left the first three items (and their DOM elements) untouched, and added new DOM elements for just the 40 datum.



One thing to point out in this example is that I did not set the ID property, and hence the conceptual selector does not show the property.



Updating values

Now let's look at an example where we change the value of several of the items in our data. In this case, we do not want to remove and insert a new visual in the DOM, but to simply update the properties in the DOM to represent a change in the underlying values.



An example of an update like this could be the price of a stock that needs to be updated.



To demonstrate this, let's make a quick change to the previous example, where when we click the button, we will now execute the following:

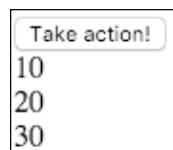
```
<button onclick='render([20, 30, 50])'>Take action!</button>
```



block (2.15): <http://goo.gl/nyUrRL>



On pressing the button, we get the following result:

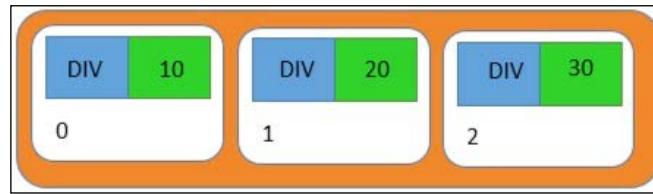


Nothing has changed on the page! Shouldn't the page be displaying 20, 30, and 50?

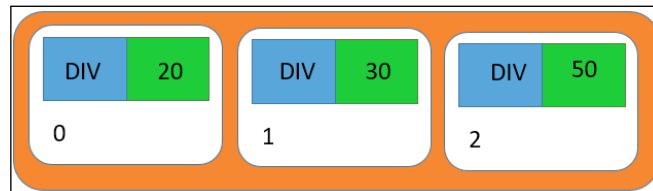
This gets into some of the subtleties of D3.js data binding. Let's step through this to explain this result:

```
var selector = d3.select('body')
  .selectAll('div')
  .data(dataToRender);
```

The call to `.selectAll('div')` identifies the three `div` tags when the page was loaded:



Following that, the call to `.data()` binds new values to each item in the selector:



D3.js has changed the bound values, but all the items were reused, and hence, are not tagged as entering. Therefore, the following statement results in an empty set of entering items.

```
var entering = selector.enter();
```

As a result, the chained methods are not executed, and the DOM elements are not updated.

How do we fix this? It's actually quite simple: we need to handle both, the case of entering elements and the case of the already existing ones. To do this, change the render function to the following:

```
function render(dataToRender) {
  var selector = d3.select('body')
    .selectAll('div')
    .data(dataToRender);

  var entering = selector.enter();
```

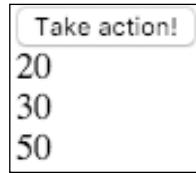
```
entering.append('div')
    .text(function(d) { return d; });

selector.text(function(d) { return d; });
}
```

The only difference is that we have added the following line:

```
selector.text(function(d) { return d; });
```

When we chain methods to the original selector, the chained functions will be applied to all the items in the selector that are neither entering nor exiting (we cover exiting in the next section). And the result is what we expected:



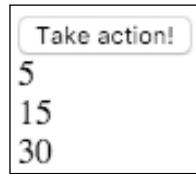
Removing items with `.exit()`

Now let's discuss how visuals change when items are removed from the collection of bound data. To handle exit, we simply need to use the `.exit()` function on the result of `.data()`. The return value of `.exit()` is a collection of the selector items which D3.js has determined need removal from the visualization based upon the change in data.

To demonstrate the removal of items, we will make a couple of simple modifications to the previous example. First, let's change the button code to render the following array upon clicking:

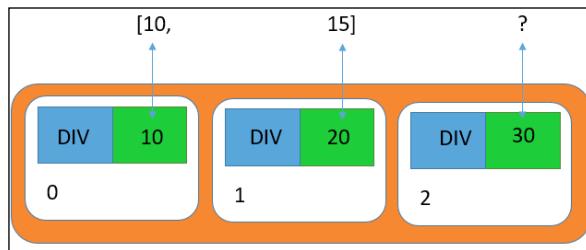
```
<button onclick='render([5, 15])'>Take action!</button>
```

When we execute the page with this change, we get the following result:

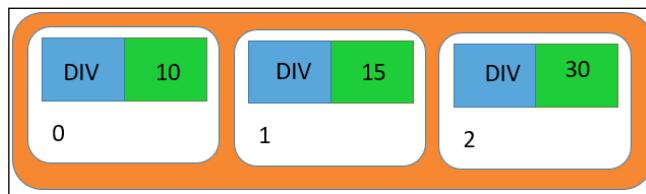


Conceptually, we would have expected the resulted as a page with just 5 and 15, and not 5, 15, and 30.

The reason for this result is again because of the way that D3.js handles data binding. When we call `.data()` with the updated data, D3.js attempts to reconcile the following:



Since all that `.data()` does is update the bound value in each item of the selector, and since there are fewer values than the selector items, we get the following selector as a result:



We then call our code to handle the enter and update states. In this case, there are no entering items, whereas items at positions 0 and 1 are scheduled for update. Hence, the first two div tags get new text values, and the third div is left unchanged in the DOM.

All that we have to do to fix this is make a call to `.exit()`, and use the results of this call to remove those items from the DOM. We can modify `render()` to the following, which gives us our desired result:



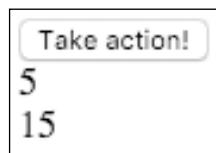
```
function render(dataToRender) {  
    var selector = d3.select('body')  
        .selectAll('div')  
        .data(dataToRender);  
  
    var entering = selector.enter();
```

```
entering.append('div')
    .text(function(d) { return d; });

selector.text(function(d) { return d; });

var exiting = selector.exit();
exiting.remove();
}
```

The only change is the addition of the last two lines. Now when we press the button, we get the desired result:



A few notes for the wise about the general update pattern

To close this chapter, I'd like to emphasize several points about managing visuals based upon data using D3.js. I believe these will definitely help you avoid problems in learning D3.js. Having come from other development platforms where data binding works in a different manner, I definitely struggled with these issues, and I want to pass along the insights that have I have learned to save you a lot of stress. It's kind of a long list, but I believe it to be very valuable.

- A visualization is almost always based upon data, and is not just coded explicitly.
- Normally, a D3.js application, on page load, will perform a `.selectAll()` on the document for the DOM elements that would represent data. Often, the result of this selection does not have any elements, as the page was just loaded.
- A call is then made to `.data()` to bind data to the selector that results from the selection.
- `.data()` iterates across the datum that are passed to it, and ensures that there are items in the selector to correlate the datum to the visuals. The value of the datum is copied into this item. DOM elements are not created by the call to `.data()`.

- Data in many apps changes dynamically over time without reloading the page, either by user interaction or through code that updates the data based upon other events. You would want to update the visualization when this happens. Therefore, you will need to call `.data()` multiple times.
- If the number of items in the data is more than the number of items in the selector it is applied to, then more selector items will be created at the end of the selector. These will be marked as in a state referred to as entering. These will be accessible using the `.enter()` function on the selector. You then chain the function calls to create DOM elements for each new item in the selector.
- If the number of items in the data is less than the number of items in the selector, then selector items will be removed from the end of the selector. These will be marked as exiting. These selector items will be available through a call to the `.exit()` function. These DOM elements will not be removed from the DOM automatically, and you will need to make a call to `.remove()` to make this so.
- To optimize this process, D3.js really only concerns itself with ensuring the number of items in the selector matches the number of datum that you specify with `.data()`.
- The data associated with a selector item is by value and not reference. Hence, `.data()` copies data into the `__data__` property on the DOM element. On subsequent calls to `.data()`, there is no comparison performed between the datum and the value of the `__data__` property.
- To update data, you write code to chain methods for generating DOM on the result of a selection, in addition to code that chains on the `.enter()` and `.exit()` functions.
- If a new datum has the same value as is already associated to a selector item, D3.js does not care. Even though the values have not changed, you will be rendering it again, but reusing the DOM elements. You will need to provide your own facilities to manage not setting the properties again if the data is the same, so as to optimize the browser re-rendering the elements.
- If you have 1,000,000 data items, and then change just one and call `.data()` again, D3.js will inherently force you to loop through all the 1,000,000 items. There will likely be visual updates to just one set of visuals, but your application will make the effort to iterate through everything every time. However, if you have 1,000,000 data items, you probably should be looking at another means of summarizing your data before visualizing it.

- D3.js optimizes around the reuse of visual elements. The assumption is that a visualization will only be periodically making updates to the existing items, and that addition or removal of items will be relatively infrequent. Hence, the general update pattern would consist of exit, update, and exit, and not comparing data.
- Normally, the rule of thumb is that one or two thousand data items and the associated visuals are handled pretty effectively by D3.js.

Well, that's quite a long list. But as we progress through this book, all the examples will follow these guidelines. By the end, these will be second nature.

Summary

In this chapter, we covered many examples to demonstrate how you can create data-driven visualizations using D3.js. We started with examples of the D3.js concept of selectors, using them to select elements from within the DOM, and discussed how selectors are used to map data items to the visuals that D3.js creates. We then examined several scenarios of binding new data, updating data, and removing data from a D3.js visualization.

Throughout this chapter, the visuals that we created with D3.js were pure HTML objects, primarily `div` tags. Although we changed the size of these `div` tags, the background color, and included text within them, the examples are a very basic form of graphical representation.

In the next chapter, we will start to get significantly more graphical by changing the focus of the examples towards working with SVG, creating real graphics (not just HTML `div` tags), and setting a framework for the rich visualizations that we will create later in the book.

3

Creating Visuals with SVG

In this chapter, we will learn about **Scalable Vector Graphics**, commonly referred to as **SVG**. SVG is a web standard for creating vector-based graphics within the browser. We will begin the chapter with several basic examples of directly coding SVG within the browser, and in the end, examine how to use D3 to create SVG elements based on data.

In this chapter, we will cover the following topics:

- A brief introduction to SVG, coordinates, and attributes
- A simple example of SVG that draws circles
- Working with fundamental shapes: ellipses, rectangles, lines, and paths
- The relationship of CSS with SVG and D3.js
- Using strokes, line caps, and dashes
- Fundamental transformations: rotate, translate, and scale
- Grouping SVG elements and uniformly applying transforms
- Transparency and layering of SVG elements

Introducing SVG

Up to this point, we have used D3 to create new DIV elements in the DOM. While many great visualizations can be created using D3 and DIVs, the true expressive power of D3 lies in using it to create and manipulate SVG elements.

SVG is an XML markup language that has been designed to express very rich 2D visualizations. SVG can take advantage of the computer's graphics processor to accelerate rendering, and is also optimized for user interaction and animation.

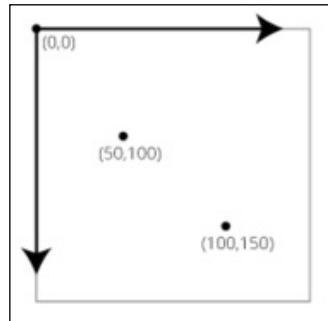
Instead of directly manipulating the pixels on the screen, SVG uses vectors for building a model of the presentation, and then transforms this representation into pixels on your behalf. This makes coding of visualizations much simpler as compared to other web technologies such as HTML5 Canvas.

Since the image is stored as a vector-based representation, the visualization of the model can be scalable. This is because all the visual elements can be easily scaled (both larger and smaller) without resulting in visual artifacts as a result of the scaling.

SVG has a convenience in that its language can be used directly within HTML on browsers that support SVG. D3 provides direct support and manipulation of SVG with D3, which feels exactly like manipulating the DOM with D3.

The SVG coordinate system

The coordinate system of SVG has an origin in the upper-left corner of the SVG element, which is $(0,0)$; the value of x increases towards the right, while those of y increase towards the bottom. This is common in computer graphics systems, but can occasionally be confusing for those used to mathematical graphs where the origin is in the lower-left or dead center.



SVG attributes

SVG, while being able to seamlessly integrate with HTML, is not HTML. Specifically, properties and styles may operate differently. An example of this is that where most HTML elements have width and height elements, but not all SVG elements use these properties.

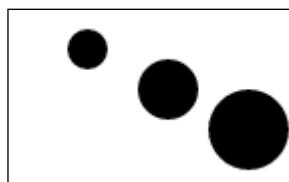
A second important point about SVG is that the position of an element is set through attributes. Due to this, it is not possible to set the position of the SVG elements using a style. Additionally, to change the position of an SVG element, such as within an animation, it is necessary to have code which sets the properties for positioning the element.

Drawing circles with SVG

We work with SVG within HTML by using an SVG tag, and placing the SVG elements within that tag. A very simple example is the following, which creates three circles:

```
<svg width="720" height="120">
  <circle cx="40" cy="20" r="10"></circle>
  <circle cx="80" cy="40" r="15"></circle>
  <circle cx="120" cy="60" r="20"></circle>
</svg>
```

This results in the following image within the browser:



bl.ock (3.1): <http://goo.gl/UMCLt1>



The SVG element itself is not visible on the page, and only provides a holder for the child tags. In this book, we will always explicitly set the width and heights of the SVG tag. In this example, it is set to be 720 pixels wide by 120 pixels tall.

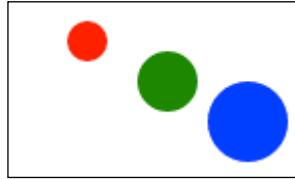
The positioning of a circle within an SVG element is performed by specifying the center *x* and *y* values of the circle. This location is relative to the upper-left corner of the SVG element, with positive *x* values moving to the right from the origin, and positive *y* values moving downwards. The size of the circle is specified by the *r* attribute, which indicates the radius of the circle.

The example did not specify a color for these circles, so the default color of the circles is black. Most SVG elements specify the color by using the CSS style attribute, and then by setting the *fill* attribute of the style.

For example, the following code gives different colors (red, green and blue) to the three circles:

```
<svg width="720" height="120">
  <circle cx="40" cy="20" r="10" style="fill:red"></circle>
  <circle cx="80" cy="40" r="15" style="fill:green"></circle>
  <circle cx="120" cy="60" r="20" style="fill:blue"></circle>
</svg>
```

This results in the following output:



bl.ock (3.2): <http://goo.gl/2k1ZIm>



D3 selections work identically with SVG elements as they do with the DOM elements. As a quick example, the following selects all the circles within the selected `svg` tag, and sets their colors to a uniform `teal` color.

```
<svg width="720" height="120">
  <circle cx="40" cy="20" r="10" style="fill:red"></circle>
  <circle cx="80" cy="40" r="15" style="fill:green"></circle>
  <circle cx="120" cy="60" r="20" style="fill:blue"></circle>
</svg>
<script>
  d3.selectAll('circle').style('fill', 'teal');
</script>
```

This results in the following output within the browser:



bl.ock (3.3): <http://goo.gl/bszmEf>



The basic shapes provided by SVG

Having some preliminaries out of the way, let's now examine the various SVG shapes that we will commonly use through the book. We have already seen how to create a circle; now let's look at some other shapes.

Ellipse

A circle is a special case of an ellipse that has an identical *x* and *y* radii. Ellipses can and often have different size radii. An ellipse is specified in SVG using the `<ellipse>` tag. We still use `cx` and `cy` attributes to position the ellipse, but instead of using `r` for radius, we use two attributes `rx` and `ry` to specify the radius in the *x* and *y* directions:

```
<ellipse cx="50" cy="30" rx="40" ry="20" />
```



block (3.4): <http://goo.gl/05QCnG>



Rectangle

Rectangles are specified using the `<rect>` tag. The upper-left corner is specified using the `x` and `y` attributes. The `width` and `height` attributes specify those respective sizes for the rectangle:

```
<rect x="10" y="10" width="150" height="100"></rect>
```



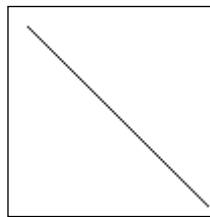
block (3.5): <http://goo.gl/b3w1Rq>



Lines

It is possible to draw lines with SVG using the `<line>` tag. A line requires at least four attributes to be specified, and normally uses five. The first two, `x1` and `y1`, specify the starting position of the line. Two more attributes, `x2` and `y2`, specify the end point for the line. The last property, albeit not required, is `stroke`, which specifies the color to be used to draw the line. Usually, we must specify the stroke to actually see the line. Here we set it to `black`.

```
<line x1="10" y1="10" x2="100" y2="100" stroke="black" />
```



block (3.6): <http://goo.gl/4qZejC>



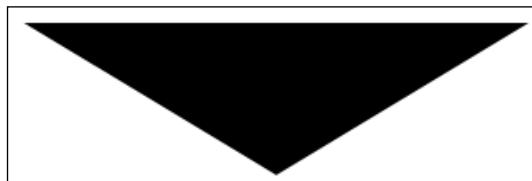
Paths

Paths are one of the most powerful drawing constructs in SVG. They provide a symbolic notion that can be used to create many geometries. A path can be shapes such as circles and rectangles. Paths also provide the user the ability to create curves using control points.

The drawing of the path is controlled by specifying one attribute, `d`, which is passed a string that specifies drawing commands that will be executed.

The basic concept of a path is that you can draw a series of either straight or curved lines, and then have the option of filling the space inside if the shape is closed. For example, the following command creates a triangle filled with black color:

```
<path d="M 10 10 L 310 20 L 160 110 Z"/>
```





bl.ock (3.7): <http://goo.gl/kCTbv7>



A path usually starts with an `M` command, which starts drawing at that specific location, in this case `(10, 10)`. The next command, `L 310 10`, draws a line from the previous point to `(310, 10)`. The next command, `L 160 10`, then draws a line from `(310, 10)` to `(160, 10)`. The final command is `Z`, which tells SVG that the shape is closed. Essentially, this informs SVG that there is an implicit line to the first position in the string of commands, in this case `(10, 10)`.



Note that we did not specify a fill or stroke color. These default to black in a path.



The mini-language for paths is quite robust, and therefore, also complex. The following table lists several other common path commands:

Command	Purpose
<code>M</code>	Move-to
<code>L</code>	Line-to
<code>H</code>	Horizontal line-to
<code>V</code>	Vertical line-to
<code>C</code>	Curve-to
<code>Q</code>	Quadratic Bezier curve-to
<code>T</code>	Smooth quadratic Bezier curve-to
<code>A</code>	Elliptical arc
<code>Z</code>	Close path

D3 provides a number of tools to facilitate the use of paths that make them much simpler to use compared to manually specifying them with string literals. We will examine a few of those later in the chapter.

Text

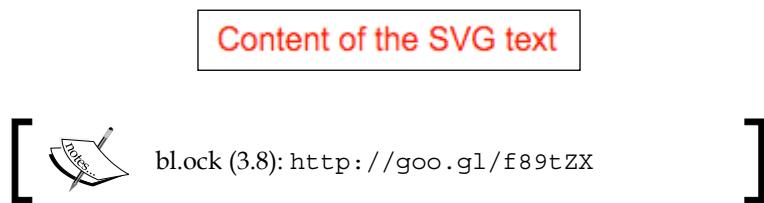
The `<text>` SVG tag allows us to place text within the SVG element. The placing of text within an SVG is different than the way it is done in HTML. The SVG text items are drawn with vector graphics instead of being rasterized. Hence, text rendered in SVG is more flexible than the rasterized text rendered with HTML. Curves in letters rendered with SVG remain smooth instead of becoming pixelated when zoom levels are applied to the entire graphic.

Text is positioned using the `x` and `y` attributes, which specify the **baseline** of the text be located at `y` and the text to be left-justified to `x`, the bottom baseline of the text (the portion at the bottom of the main part of the letters, excluding the descenders) left of the text being the anchor of the positioning.

This is demonstrated by the following, which also sets the font family, size and fill color. The actual text to be displayed is set with the inner text content of the tag:

```
<text x="10" y="20"  
      fill="Red" font-family="arial" font-size="16">  
  Content of the SVG text  
</text>
```

Which renders the following in the upper-left side of the SVG element:



Applying CSS styles to SVG elements

SVG elements can be styled identically to the way HTML elements are styled. The same CSS with ID and class attributes can be used to direct styles to the SVG elements, or you can just use the `style` attribute directly and specify CSS as its content. However, many of the actual styles in HTML differ in SVG. For example, SVG uses `fill` for a rectangle, whereas HTML would use `background` for a `div` tag that represents a rectangle.

In this book, we will generally try to avoid using CSS, and explicitly code the `style` attributes using the functions provided by D3.js. But many examples on the web do use CSS combined with SVG, so it is worth a quick mention.

The following example demonstrates styling SVG with CSS. The example uses two styles to set the fills of several rectangles. The first style will make all the rectangles red by default. The second one defines a style that makes all the rectangles with ID `willBeGreen` filled with green color. The example then creates three rectangles: the first two using the CSS styles, and the third using CSS within the `style` attribute set as `fill` to blue.



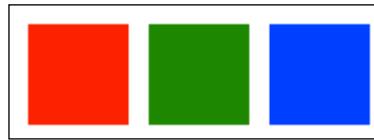
The styles defined in the sample are the following:

```
<style>
  svg rect { fill: red; }
  svg rect#willBeGreen { fill: green; }
</style>
```

And the rectangles are created as follows:

```
<rect x="10" y="10" width="50" height="50" />
<rect x="70" y="10" width="50" height="50" id="willBeGreen" />
<rect x="130" y="10" width="50" height="50" style="fill:blue" />
```

The resulting output will be as shown in the following image:



Strokes, caps, and dashes

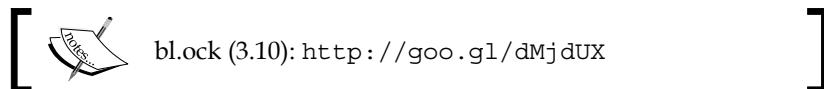
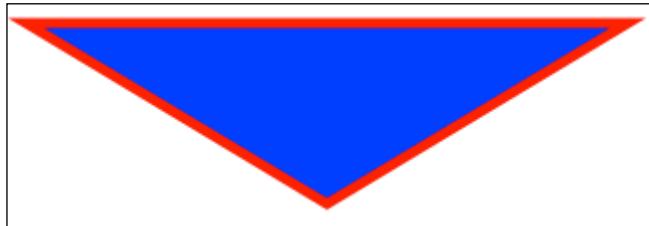
SVG shapes have an attribute known as `stroke`. The attribute `stroke` specifies the color of a line that outlines an SVG shape. We saw the use of `stroke` with a line, but it can be used with most of the SVG elements.

Whenever we specify `stroke`, we usually also specify a stroke width using the `stroke-width` attribute. This informs SVG about the thickness (in pixels) of the outline that will be rendered.

To demonstrate `stroke` and `stroke-width` attributes, the following example recreates the path from the path example, and sets a stroke to be 10 pixels thick, using red as its color. Additionally, we set the `fill` of the path to blue. We set all the attributes using the `style` property of `stroke`:

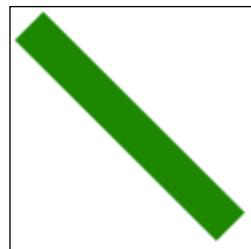
```
<path d="M 10 10 L 210 10 L 110 120 z"
      style="fill:blue;stroke:red;stroke-width:5" />
```

The preceding example results in the following rendering:



As we saw earlier, we can set `stroke` on a line. It can also have its `stroke-width` set. Let's examine this by changing our line example to set the thickness of the line to 20 and the color to green:

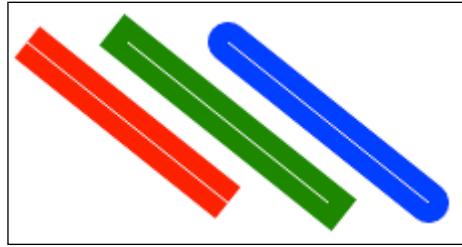
```
<line x1="10" y1="10" x2="110" y2="110"  
      stroke="green" stroke-width="20" />
```



Notice how this line actually looks like a rectangle. This is because lines have an attribute named `stroke-linecap` which describes the shape of the end of the line, known as the line cap.

The default for this value is `butt`, which gives us the 90 degree sharp corners. The other values that can be used are `square` or `round`. The following example demonstrates the same line with all these different `stroke-linecap` values:

```
<line x1="10" y1="20" x2="110" y2="100"
      stroke="red" stroke-width="20" stroke-linecap="butt" />
<line x1="60" y1="20" x2="160" y2="100"
      stroke="green" stroke-width="20" stroke-linecap="square" />
<line x1="110" y1="20" x2="210" y2="100"
      stroke="blue" stroke-width="20" stroke-linecap="round" />
<path d="M 10 20 L 110 100 M 60 20 L 160 100 M 110 20 L 210 100"
      stroke="white" />
```



bl.ock (3.12): <http://goo.gl/Xcaz41>

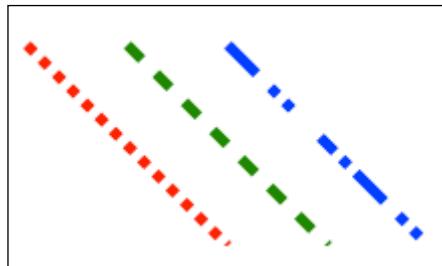


Note that for each of the three lines, we drew a line with a `stroke-width` of 20, and then within each line, we drew a white line using a single path with three move and line commands. The white line helps in distinguishing the effect of the end caps on the lines.

First examine the red line. The ends are flush with the end of the white line. Contrast it with the green line. In this line, the line-cap, still square, extends past the white line by the width of the stroke. The blue line, with a round line-cap, is drawn using a half circle of radius of one half of the `stroke-width`.

By default, the SVG lines are solid, but they can also be created with dashes, specified by using the `stroke-dasharray` attribute. This attribute is given a list of integer values which specify a repeating pattern of line segment widths, the first starting with the `stroke` color and alternating with empty space:

```
<line x1="10" y1="20" x2="110" y2="120"
      stroke="red" stroke-width="5"
      stroke-dasharray="5,5" />
<line x1="60" y1="20" x2="160" y2="120"
      stroke="green" stroke-width="5"
      stroke-dasharray="10,10" />
<line x1="110" y1="20" x2="210" y2="120"
      stroke="blue" stroke-width="5"
      stroke-dasharray="20,10,5,5,5" />
```



bl.ock (3.13): <http://goo.gl/VyBBwy>



Applying SVG transforms

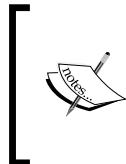
The **S** in SVG stands for **Scalable**, while **V** stands for **Vector**. These are the two important parts of the name. This allows us to be able to apply a variety of transforms prior to the rendering of SVG shapes.

Each SVG shape is represented by one or more vectors, where a vector in SVG is a tuple (x, y) distance from an origin in the coordinate system. As an example, a rectangle will be represented by four 2D vectors, one for each corner of the rectangle.

When creating graphical visualizations, this modeling of data with vectors has several benefits. One of those is that we can define a shape around a coordinate system for just that shape. Modeling this way allows us to make copies of the shape, but position them in different places in a larger image, rotate them, scale them, and perform many other operations beyond the scope of this text.

Secondly, these transformations are applied on the model before being rendered into pixels on the screen. Because of this, SVG can ensure that irrespective of the level of scale applied to the image, it does not get pixelated.

Another important concept in transformations is that they can be applied in a chain and in any sequence. This is an extremely powerful concept in linear algebra for creating composite models of visuals.



There are many ramifications of the transformations and their sequencing that can take effect on the result of the rendering in SVG. Unfortunately, an explanation of these is beyond the scope of the book, but when they have an effect on examples, we will examine them in light of that particular example.

In this section, and in other examples in this book, we will use three general types of transformations provided by SVG: `translate`, `rotate`, and `scale`. Transformations can be applied to an SVG element by using the `transform` attribute.

To demonstrate transforms, we will look at several examples that apply each transform to a rectangle to see how they affect the resulting rendering of the rectangle.

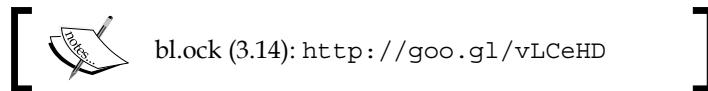
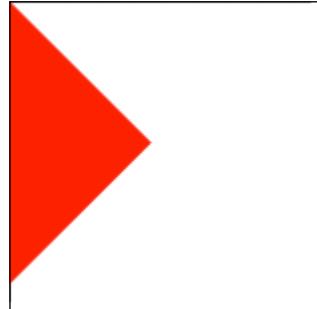
Rotate

The first transformation that we will examine is rotation. We can rotate an SVG object by a specified number of degrees using `.rotate(x)`, where `x` specifies the number of degrees to rotate the element.

To demonstrate this, the following example rotates our rectangle by 45 degrees. A simple axis with two lines is rendered to give a frame of reference for the translation. This will be included in this code snippet, but excluded in the rest for brevity:

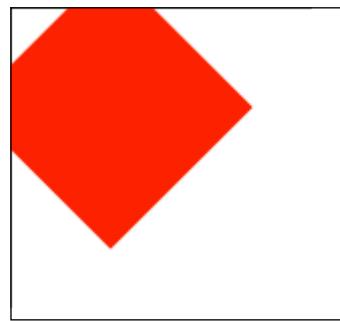
```
<line x1="0" y1="150" x2="0" y2="0" stroke="black" />
<line x1="0" y1="0" x2="150" y2="0" stroke="black" />
<rect x="0" y="0" width="100" height="100" fill="red"
      transform="rotate(45)" />
```

The preceding snippet gives us the following result:



This is not quite the effect that we may have wanted. This is because the rotation of a rectangle is performed around its upper-left corner. To make this appear to have rotated around its center, we need to use an alternate form of `rotate()` which takes three parameters: the angle to rotate followed by an offset from the upper-left corner of the rectangle to a point that represents the center of the rectangle:

```
<rect x="0" y="0" width="100" height="100" fill="red"  
      transform="rotate(45,50,50)" />
```



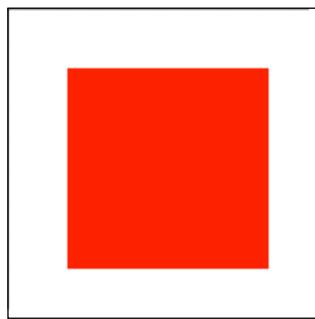
The rectangle has now been rotated about its center, but there is an issue of several of the corners being clipped outside the bounds of the containing SVG element. We will fix this when we look at translations in the next section.

Translate

An SVG element can be repositioned within its containing element by using a **transformation**. A transform is performed using the `translate()` function. `translate()` takes two values: the distance in `x` and `y` and the distance to reposition the element within its parent.

The following example will draw our rectangle, and translate it 30 pixels to the right and 30 pixels down:

```
<rect x="0" y="0" width="100" height="100" fill="red"  
      transform="translate(30,30)" />
```

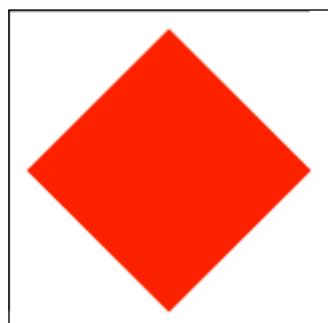


block (3.16): <http://goo.gl/jANiXU>



Now let's look back at the last rotation example, where two of the corners of the rectangle were clipped. We can fix this to see those corners by specifying a translation on the rectangle to move it right and down 30 pixels prior to the rotation:

```
<rect x="0" y="0" width="100" height="100" fill="red"  
      transform="translate(30,30) rotate(45,50,50)" />
```





bl.ock (3.17): <http://goo.gl/W6MeSc>



This also demonstrates applying multiple transformations within a single string supplied to transform. You can sequentially apply many transforms in this manner to handle complex modeling scenarios.

A common question about translate transform is why not just change the *x* and *y* attributes to position the elements instead of using the transform?

The answer to this can be very complicated and has many reasons. The first is that not all SVG elements are positioned with *x* and *y* attributes, for example, a circle, which is positioned using its *cx* and *cy* attributes. Hence, there is no consistent set of attributes for positioning. Using a translate transform therefore allows us to have a uniform means of positioning the SVG elements no matter what type they are.

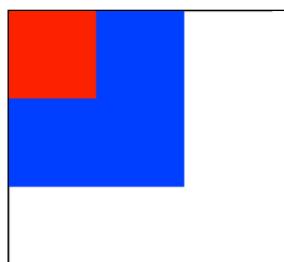
Another reason is that when applying multiple transforms, it is not easy (or possible) to access the *x* and *y* attributes. Moreover, through various transforms, the actual location of an SVG element may not match directly with the pixels or points specified in another coordinate system which using *x* and *y* attributes be included.

Scale

Scaling an object changes its apparent visual size by a given percentage along both the *x* and *y* axes. Scaling is performed using the `scale()` function. It can be uniformly applied to each axis, or you can also specify a different scale value for each.

The following example demonstrates scaling. We will draw two rectangles, one atop the other. The rectangle at the bottom will be blue, and the one on top, red. The red will then be scaled to 50 percent of its size:

```
<rect x="0" y="0" width="100" height="100" fill="blue"/>
<rect x="0" y="0" width="100" height="100" fill="red"
      transform="scale(0.5)" />
```



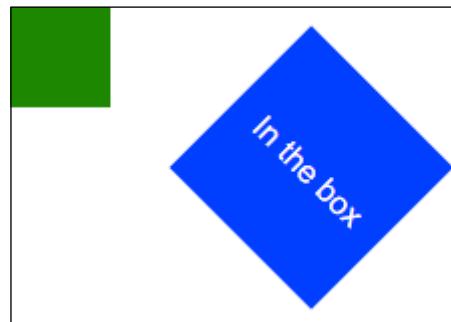
bl.ock (3.18): <http://goo.gl/fCAhg7>

Groups

SVG elements can be grouped together using the `<g>` tag. Any transformations applied to the group will be applied to each of the elements in the group. This is convenient for applying an overall transform to a particular group of items only.

The following example demonstrates both the translation of a group of items (the blue rectangle with text) and the way the transform on the group affects both the items. Note that the green rectangle is not affected because it is not part of the transform:

```
<g transform="translate(100,30) rotate(45 50 50)">
  <rect x="0" y="0" width="100" height="100" style="fill:blue" />
  <text x="15" y="58" fill="White" font-family="arial"
        font-size="16">
    In the box
  </text>
</g>
```

bl.ock (3.19): <http://goo.gl/FY6q4D>

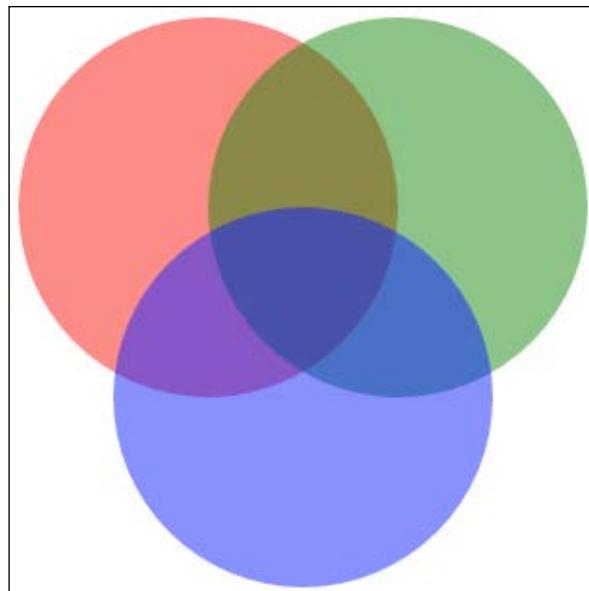
Also notice that the placement of the text on top of the rectangle is relative to the top-left corner of the group, not the SVG element. This is important for ensuring that the text rotates properly relative to the blue rectangle.

Transparency

SVG supports drawing of transparent elements. This can be done by either setting the `opacity` attribute or by using the `rgba` (red-green-blue-alpha) value when specifying a color.

The following example renders three circles of different colors, all of which are 50 percent transparent. The first two use the `opacity` attribute, and the third uses a transparent color specification for the fill.

```
<circle cx="150" cy="150" r="100"
        style="fill:red" opacity="0.5" />
<circle cx="250" cy="150" r="100"
        style="fill:green" opacity="0.5" />
<circle cx="200" cy="250" r="100"
        style="fill:rgba(0, 0, 255, 0.5)" />
```



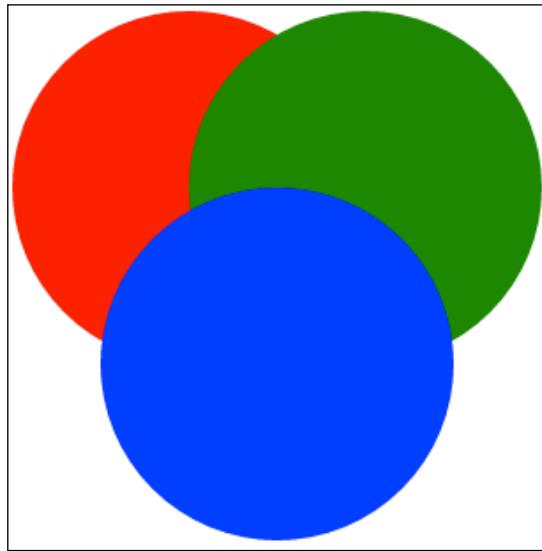
block (3.20): <http://goo.gl/xRzArg>



Layers

You may have noticed that the SVG elements overlay each other in a particular order, with certain elements appearing to be closer and obscuring those that are behind. Let's examine this using an example that overlays three circles on top of each other:

```
<circle cx="150" cy="150" r="100" style="fill:red" />
<circle cx="250" cy="150" r="100" style="fill:green" />
<circle cx="200" cy="250" r="100" style="fill:blue" />
```



[ bl.ock (3.21): <http://goo.gl/h04xmc>]

The blue circle is drawn in front of the green circle, which is drawn in front of the red circle. This order is defined by the sequence that is specified in the SVG markup, with each successive element being rendered atop the previous elements.

[ If you have used other graphics packages or UI tools, you would know that they often provide a concept known as a Z-order, with Z being a pseudo-dimension where the drawing order of the elements is from the lowest to the highest Z-order. SVG does not offer this ability, but we will see in later chapters that we can address this by sorting the selections before laying them out.]

Summary

In this chapter, you learned how to use SVG to create various shapes, how to lay out the SVG elements using the SVG coordinates, and how layers affect the rendering. You also learned to perform transformation on SVG elements, which will be used frequently in examples throughout this book and form an essential part of creating visuals using D3.

In the next chapter, we move back to a focus on D3.js, and in particular we will take what we have learned in this chapter with SVG and use D3 to create a data-driven bar graph using D3.js selections and SVG elements.

4

Creating a Bar Graph

Now that we have examined binding data and generating SVG visuals with D3, we will turn our attention to creating a bar graph using SVG in this chapter. The examples in this chapter will utilize an array static of integers, and use that data to calculate the height of bars, their positions, add labels to the bars, and add margins and axes to the graph to assist the user in understanding the relationships in the data.

In this chapter, we will cover the following topics:

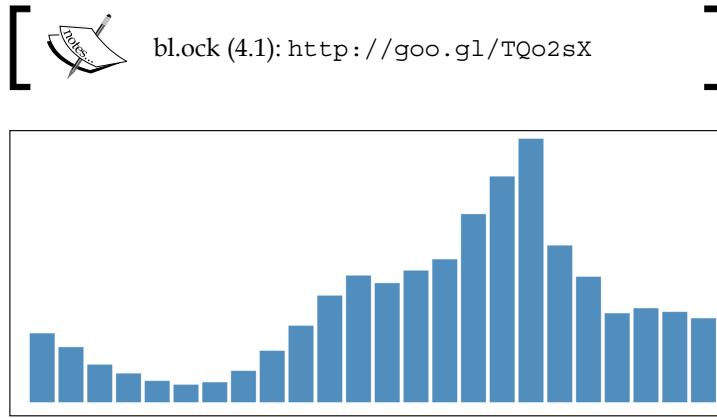
- Creating a series of bars that are bound to data
- Calculating the position and height of the bars
- Using a group to uniformly position multiple elements representing a bar
- Adding margins to the graph
- Creating and manipulating the style and labels in an axis
- Adding an axis to the graph

The basic bar graph

We have explored everything that we need to draw a series of bars based upon data in the first three chapters. The first example in this chapter will leverage using SVG rectangles for drawing the bars. What we need to do now is calculate the size and position of the bars based upon the data.

Creating a Bar Graph

The code for our bar graph is available at the following location. Open this link in your browser, and we will walk through how the code creates the visual that follows.



The code starts with a declaration of the data that is to be represented as a graph. This example uses a hard-coded array of integers. We will look at more complex data types later in the book; for now, we simply start with this to get used to the process of binding and that of creating the bars:

```
var data = [55, 44, 30, 23, 17, 14, 16, 25, 41, 61, 85,  
           101, 95, 105, 114, 150, 180, 210, 125, 100, 71,  
           75, 72, 67];
```

Now we define two variables that define the width of each bar and the amount of spacing between each bar:

```
var barWidth = 15, barPadding = 3;
```

We need to scale the height of each bar relative to the maximum value in the data. This is determined by using the `d3.max()` function.

```
var maxValue = d3.max(data);
```

Now we create the main SVG element by placing it inside the body of the document, and assign a width and height that we know will hold our visual. Finally, and as a matter of practice that will be used throughout this book, we will append a top-level group element in the SVG tag. We will then place our bars within this group instead of placing them directly in the SVG element:

```
var graphGroup = d3.select('body')  
  .append('svg')  
  .attr({ width: 1000, height: 250 })  
  .append('g');
```



I find this practice of using a top-level group useful as it facilitates placing multiple complex visuals in the same SVG, such as in the case of creating a dashboard.

In this example, we are not going to scale the data, and use an assumption that the container is the proper size to hold the graph. We will look at better ways of doing this, as also for calculating the positions of the bars, in *Chapter 5, Using Data and Scales*. We simply strive to keep it simple for the moment.

We need to perform two pieces of math to be able to calculate the *x* and *y* location of the bars. We are positioning these bars at pixel locations starting at the bottom and the left of `graphGroup`. We need two functions to calculate these. The first one calculates the *x* location of the left side of the bar:

```
function xloc(d, i) { return i * (barWidth + barPadding); }
```

During binding, this will be passed the current datum and its position within the `data` array. We do not actually need the value for this calculation. We simply calculate a multiple of the sum of the width and padding for the bar based upon the array position.

Since SVG uses an upper-left origin, we need to calculate the distance from the top of the graph as the location from where we start drawing the bar down towards the bottom of the visual:

```
function yloc(d) { return maxValue - d; }
```

When we position each bar, we will use a `translate` transform that takes advantage of each of these functions. We can facilitate this by declaring a function which, given the current data item and its array position, returns the calculated string for the `transform` property based upon this data and the functions:

```
function translator(d, i) {
  return "translate(" + xloc(d, i) + "," + yloc(d) + ")";
}
```

All we need to do now is generate the SVG visuals from the data:

```
barGroup.selectAll("rect")
  .data(data)
  .enter()
  .append('rect')
  .attr({
    fill: 'steelblue',
    transform: translator,
    width: barWidth,
    height: function (d) { return d; }
});
```

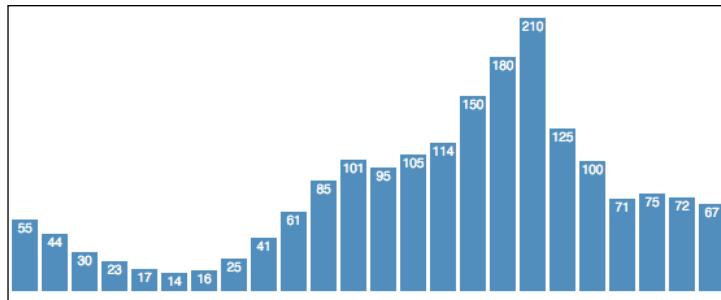
Pretty good for just a few lines of code. But all we can tell from the graph is the relative sizes of the data. We need more information than this to get an effective data visualization.

Adding labels to the bars

Now we will add a label holding the value of the datum right at the top of each bar. The code for this example is available at the following link:



The following image demonstrates the resulting visual:



To accomplish this, we will modify our SVG generation such that:

1. Each bar is represented by an SVG group instead of a `rect`.
2. Inside each group that represents a bar, we add an SVG and a text element.
3. The group is then positioned, hence positioning the child elements as well.
4. The size of the `rect` is set as before, causing the containing group to expand to the same size.
5. The text is positioned relative to the upper-left corner of its containing group.

By grouping these elements in this manner, we can reuse the previous code for positioning, and utilize the benefit of the group for locating all the child visuals for a bar. Moreover, we only need to size and position those child elements relative to their own group, making that math very simple. The code for this is identical to the previous example through the declaration of the positioning functions.

The first change is in the creation of the selector that represents the bars:

```
var barGroups = g.selectAll('g')
  .data(data)
  .enter()
  .append('g')
  .attr('transform', translator);
```

Instead of creating a `rect`, the code now creates a group element. The group is initially empty, and it is assigned the transform that moves it into the appropriate position.

Using the selector referred to by `barGroups`, the code now appends a `rect` into each group while also setting the appropriate attributes.

```
barGroups.append('rect')
  .attr({
    fill: 'steelblue',
    width: barWidth,
    height: function(d) { return d; }
  });

```

The next step is to add a `text` element to show the value of the datum. We are going to position this text such that it is right at the top of the bar, and centered in the bar.

To accomplish this, we need a translate transform that represents an offset halfway into the bar and at the top. This is common for each bar, so we can define a variable that is reused for each:

```
var textTranslator = "translate(" + barWidth / 2 + ",0)";
```

Next, we append a text element into each group, setting its text (the string value of the datum), the appropriate attributes for the text, and finally, the style for the font.

```
barGroups.append('text')
  .text(function(d) { return d; })
  .attr({
    fill: 'white',
    'dominant-baseline': 'text-before-edge',
    'text-anchor': 'middle',
    transform: textTranslator
  })
  .style('font', '10px sans-serif');
```

That was pretty easy, and it is nice to have the labels on the bars, but our graph could still really use axes. We will look at adding those next.

Margins and axes

Adding axes to the graph will give the reader a much better understanding of the scope of the graph and the relationship between the values in the data. D3.js has very powerful constructs built in for allowing us to create axes.

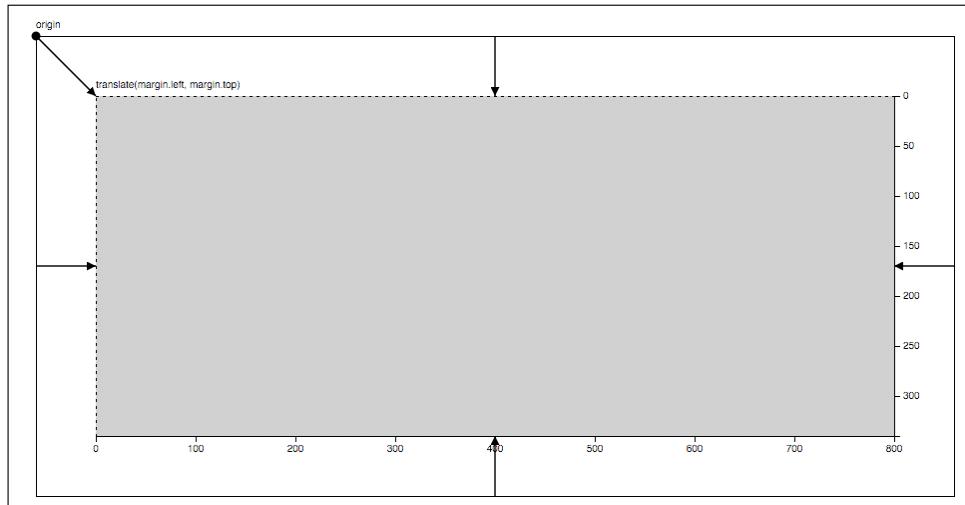
Axes in D3.js are based upon another concept known as scales. While scales are by themselves very useful (we will cover scales in more detail in *Chapter 5, Using Data and Scales*), for the remainder of this chapter, we will examine using them to create basic axes in our bar chart.

However, before we get to axes, we will first take a short but important diversion into the concept of margins, and that of adding a margin to our bar chart to make room for the axes.

Creating margins in the bar graph

Margins have several practical uses in a graph. They can be used to provide spacing between the graph and other content on the page, giving the reader clean sightlines between their visualization and the other content. However, the real practical use of margins is to provide space on one or more sides of the visualization for providing axes.

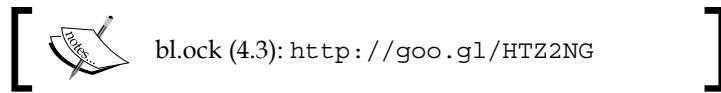
The following image demonstrates what we want to accomplish with margins:



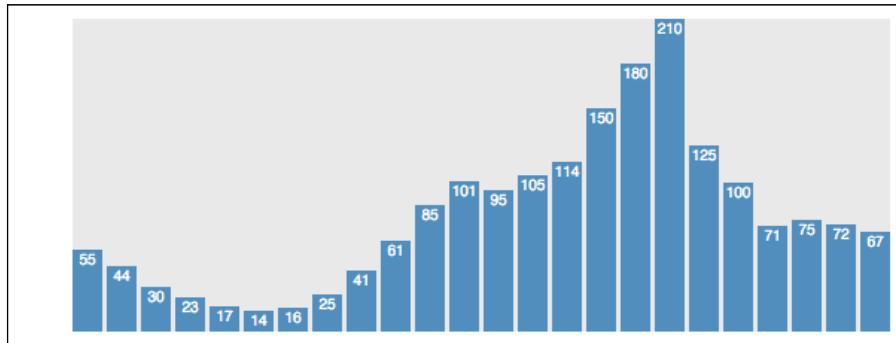
The grey portion is where we will place our existing graph. Then, depending upon the axes that we decide to use (left, top, right, bottom), we need to provide space in our visualization for rendering those axes just outside the graph. Note that a single graph could use any or all of the margins for different axes, so it is a good practice to build code that plans for all of them.

In D3.js applications, this is generally performed using a concept referred to as **margin convention**. We will step through an example of using this concept to add margins to our graph. Additionally, instead of using a static size for our layout, we will compute the height and width of the graph based upon the number of data points in this example.

To get started, load the example from the following information box.



The resulting visualization for the code is seen in the following graph:



Besides the margins, this example adds a grey background to the area behind the chart. This highlights the area used for the chart, and emphasizes it relative to the margins that are added. It also puts a rectangle around the main SVG element to highlight its boundaries, as it helps us see the extent of the margins added to the graph.

Let's step through this example and examine how it differs from the previous example. We start with calculating the actual width of the area of the bars:

```
var graphWidth = data.length * (barWidth + barPadding)
    - barPadding;
```

Now we declare a JavaScript object that will represent the size of our margins:

```
var margin = { top: 10, right: 10, bottom: 10, left: 50 };
```

Using these values, we can calculate the total size of the entire visualization:

```
var totalWidth = graphWidth + margin.left + margin.right;
var totalHeight = maxValue + margin.top + margin.bottom;
```

Now we can create the main SVG element, and set it to the exact size that it needs to be:

```
var svg = d3.select('body')
.append('svg')
.attr({ width: totalWidth, height: totalHeight });
```

For the visual effect, the following adds a rectangle that shows us the boundaries of the main SVG element:

```
svg.append('rect').attr({
  width: totalWidth,
  height: totalHeight,
  fill: 'white',
  stroke: 'black',
  'stroke-width': 1
});
```

Now we add a group to hold the main part of the graph:

```
var graphGroup = svg
.append('g')
.attr('transform', 'translate(' + margin.left + ',' +
margin.top + ")");
```

To emphasize the area of the actual graph, a grey rect is added to the group:

```
graphGroup.append('rect').attr({
  fill: 'rgba(0,0,0,0.1)',
  width: totalWidth - (margin.left + margin.right),
  height: totalHeight - (margin.bottom + margin.top)
});
```

The remainder of the code remains the same.

At this point, we have added margins around the graph, and made room on the left side for an axis to be drawn. Before we put that in the visualization, let's first take a look at an example of creating an axis to learn some of the concepts involved.

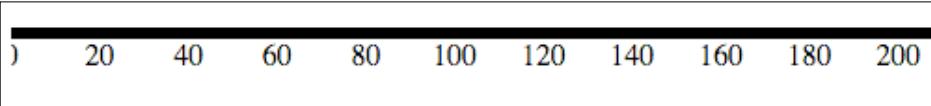
Creating an axis

To demonstrate the creation of an axis, we will start with creating an axis appropriate for placement at the bottom of the graph, referred to as a **bottom** axis. This is the default type of axis created with D3.js. So we will start with it, and then examine changing the orientation after looking at a few concepts related to axes.

The following is the code for the example that we will walk through, and which results in the generation of the subsequent axis:



bl.ock (4.4): <http://goo.gl/TyDAH6>



In our example, we create the scale and axis with the following lines of code:

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

var axis = d3.svg.axis().scale(scale);
svg.call(axis);
```

To create an axis, we first need to create a **scale** object using `d3.scale()`. A scale informs the axis about the range of values it will represent (known as the **domain**), and the overall size for which the axis should be rendered in the visual (referred to as the **range**). In this example, we are using a **linear** scale. A linear scale informs the axis that the values will be linearly interpolated from the lower to the higher value, in this case, 0 to 210.



D3.js scales have uses for things other than axes. We will examine these uses in *Chapter 5, Using Data and Scales*.



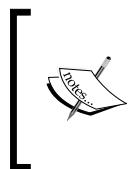
Creating a Bar Graph

The axis is then created using the `d3.svg.axis()` function, and by passing it the scale by chaining a call to `.scale()`.

The axis scale then needs to be associated with a selection, which is performed using the `.call()` function. This informs D3.js that when it renders the visual that it should call the axis function to render itself.

This feels a little different than the way we have created visual elements so far. This technique is used by D3.js because an axis is a complex set of SVG elements that need to be generated. The use of `.call()` allows us to separate complex rendering logic into a function call during the rendering pipeline, and the design of D3.js was made to render axes in this manner.

The labels on the axis are automatically generated by D3.js, and are based upon the values of the domain. The visualized size of the axis is specified by the range. In this case, since this is a bottom axis, the labelling starts at the minimum value of 0, and D3.js uses intervals of 20 for the labels. The last label that fits is 200, so D3.js does not actually create a label for the maximum value of 210.



In the output, the label 0 is clipped. This is because the axis is positioned flush to the left of the SVG element. This orientation is such that the line in the axis is flush. Since the text for the first label is center-justified on the tick, its left half gets clipped. This can be fixed easily with a translation, which we will examine when we place the axis next to our graph.

Inspecting the rendered axis using Developer tools, you will see the effort that D3.js has made to generate the axis:

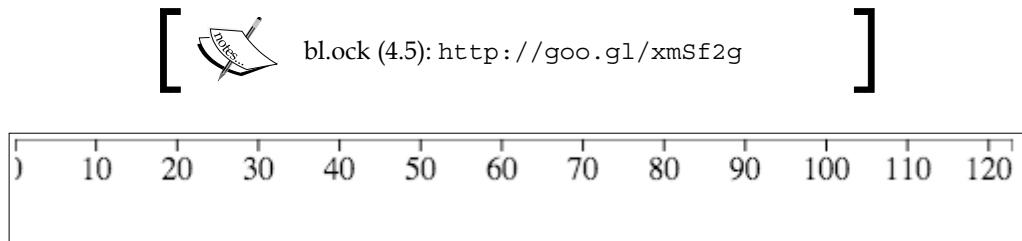
```
<svg width="500" height="500">
  <g>
    <g class="tick" transform="translate(0,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(47.61904761904761,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(95.23809523809523,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(142.85714285714286,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(190.47619047619045,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(238.09523809523807,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(285.7142857142857,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(333.333333333333,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(380.9523809523809,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(428.57142857142856,0)" style="opacity: 1;"></g>
    <g class="tick" transform="translate(476.19047619047615,0)" style="opacity: 1;"></g>
    <path class="domain" d="M0,6V0H500V6"></path>
  </g>
</svg>
```

What D3.js has done is generate a group for each tick on the axis and a single path that renders the line of the axis. Each tick group itself consists of a line that represents the tick on the axis and the label on the tick.

Examining the output, you will notice that we do not actually see any ticks along the axis. This makes it difficult to realize the actual point on the axis that is associated with the label. This is due to the default styling that is used. We will make this axis look better in the next section.

The reason we could not see the ticks on our axis is due to the default thickness of the path representing the axis. We can change this by simply modifying the style of the path representing the axis as well as the style of the ticks.

Open the following example in your browser to learn how to accomplish this:



This code makes a few small modifications to be able to change the style as shown in the following section of code:

```
var axisGroup = svg.append('g');
var axis = d3.svg.axis().scale(scale);

var axis = d3.svg.axis().scale(scale);
var axisNodes = axisGroup.call(axis);
var domain = axisNodes.selectAll('.domain');
domain.attr({
    fill: 'none',
    'stroke-width': 1,
    stroke: 'black'
});
var ticks = axisNodes.selectAll('.tick line');
ticks.attr({
    fill: 'none',
    'stroke-width': 1,
    stroke: 'black'
});
```

The first change is that we create a group, represented by the variable `axisGroup`, to hold the axis that is generated. This will be used so that we can select the SVG elements in the axis representing the ticks and the axis line, and change their styles.



It's good practice to always put the axes in a group. This facilitates style changes like the one we are performing in this example. Moreover, it is almost always the case that the axis needs to be translated into a specific position in the visualization. An axis itself cannot be translated, so placing it in a group element and then transforming the group accomplishes this task.

Secondly, the code captures the nodes that result from the generation of the axis in the `axisNodes` variable. Using `axisNode`, we can then perform two more selections to find specific elements in the axis: one for the element with the `domain` class, and the other for the line elements with the `line` class. Using the results of each of these two selections, the code then sets the `fill`, `stroke`, and `stroke-width` properties to make them all one pixel thick black lines.

Changing the axis orientation

D3.js axes can be rendered into four different orientations using the `.orient()` function on the axis, passing it the name of the orientation that is desired. The following table shows the orientation names that can be used:

'top'	Horizontal axis, with ticks and labels above the axis line.
'bottom'	Horizontal axis, with ticks and labels below the axis line (default)
'left'	Vertical axis, with ticks and labels to the left of the axis line
'right'	Vertical axis, with ticks and labels to the right of the axis line

Essentially, each of these relate to one of the four sides of a graph, such as the margins that were covered earlier. There is no effect of this function on the location of the axis in the visual (we have to do that ourselves). Instead, it decides whether the axis line is horizontal or vertical, and also if the labels are on the top or bottom of a horizontal axis or to the left or right of a vertical axis.

To demonstrate this, we will quickly inspect the top, right, and left orientations. Open the following link for an example of a top axis:



bl.ock (4.6): <http://bl.ocks.org/d3byex/8791783ee37ab76a8517>

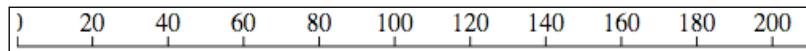
There are two small modifications from the previous example. The primary change is that when we create the axis, we make a call to `.orient('top')`:

```
var axis = d3.svg.axis()
    .orient('top')
    .scale(scale);
```

The second change is that we need to translate the axis down the Y axis. We do this using the following statement:

```
axisGroup.attr('transform', 'translate(0,50)');
```

The result of the preceding example is as follows:



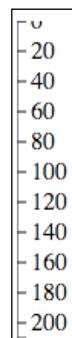
The orientation has moved the label and ticks to the top of the axis line instead of below.

The need for a transform is perhaps a little more subtle. If the axis was not transformed, all that we would see in the result is a single black line at the top of the rendering. This is because the positioning of an axis is relative to the path rendering the axis line. In this case, the line would be at $y = 0$, and the ticks and text would be clipped as they are above the line and not visible.

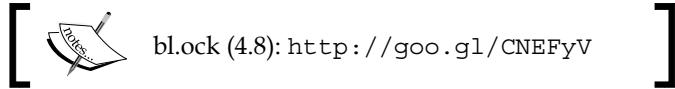
Now open the code for the following example, which renders a right-oriented axis. We will not examine the code, as it is a single simple change of calling `.orient('right')`.



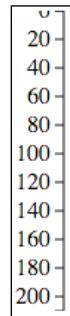
The result of the preceding code is the following:



The following example demonstrates a left-oriented axis. This is again a simple change of the parameter to `.orient()`. Additionally, the code also translates the axis to the right a bit, as the ticks and labels would be clipped off the left.



And the results are as follows:



Inverting the labels on an axis

We want to place a left axis on our bar chart, essentially the output of example 4.8. But if you examine the axis, you will notice that the labels are increasing from the top to the bottom. Our graph represents 0 at the bottom with values increasing upwards. This axis will not be appropriate for our graph.

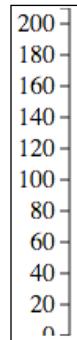
This inversion of labels is a very simple change to the code. Open the following example:



The code is identical to example 4.8 except for one change.

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

We change the order of the values passed to the domain, which will essentially reverse the order of the labels. This gives us the following result:



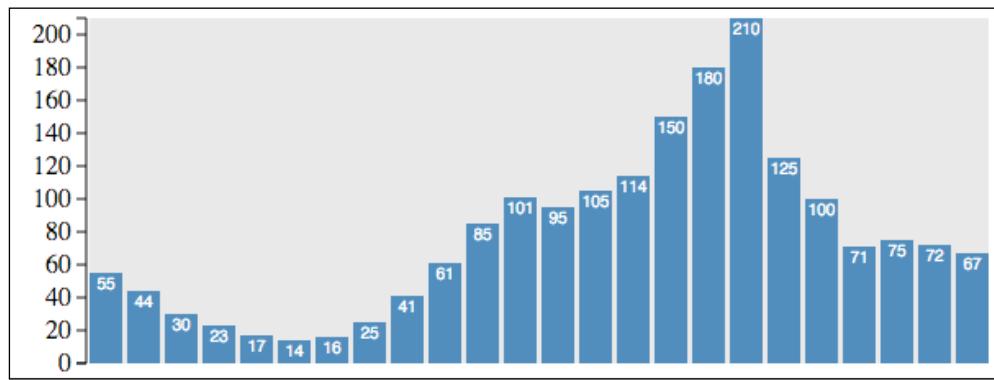
The labels have been reversed into the order that we desire. Note now that the label 0 is clipped at the bottom. We will fix this in our next example when we combine the axis with the bar graph.

Adding the axis to the graph

We now have everything that we need to create a bar graph with an axis. Essentially, we only need to combine the code from example 4.3 with the axis code from example 4.9. The following example does exactly this:



And the result is the following graph, which is exactly what we wanted:



The code in this example is identical to the one in example 4.3 up to the point where we create the group to contain the axis:

```
var leftAxisGroup = svg.append('g');
var axisPadding = 3;
leftAxisGroup.attr({
  transform: 'translate(' + (margin.left - axisPadding) + ','
  + margin.top + ')' });
```

The change here is the translation of the axis along the X axis by the width of the left margin, and down the Y axis by the size of the top margin. For aesthetics, the code simply renders the axis with three pixels of padding.



Also note that since we have margin space at the bottom, the 0 label is no longer truncated.



Summary

In this chapter, you extended your knowledge of using D3 to create a bar graph from a collection of integers. You learned how to position and size each element of the graph according to its data, and how to position groups of data that contain multiple visuals representing a single bar—specifically, how to add a label that represents the value of the underlying datum at the top of a bar.

We then examined the facilities in D3.js for creating axes. We introduced the concept of a scale, which is an important facet of implementing axes. We further examined the different orientations available for an axis, and how to invert the order of the labels on an axis. Finally, we combined the axis and the bar graph together into an effective visualization of the data.

As great as our bar chart looks in this example, we will still have several issues. The overall size of the graph was related to the actual values of the data. This was convenient for demonstrating the construction of a bar graph visualization, but what if the values are not integers, or if the values are extremely small or large? We might not see the bars at all, or the bars may be so large as to exceed the size of the main SVG element.

In the next chapter, we will address these issues by learning more about scales. Scales will provide an exceptionally easy means of mapping data into the physical dimensions of a visualization. You will also learn about loading data from external sources, and about working with data that is more complex in structure than simple integers.

5

Using Data and Scales

In *Chapter 4, Creating a Bar Graph*, you learned how to create a bar graph that was based upon a sequence of integers that were statically coded within the application. Although the resulting graph looks quite nice, there are several issues with the way the data is provided and rendered.

One of the issues is that the data is hard-coded within the application. Almost invariably, we are going to load the data from an external source. D3.js provides a rich set of functionalities for loading data from sources over the web, and which is represented in different formats. In this chapter, you will learn to use D3.js for loading data from the web in JSON, CSV, and TSV formats.

A second issue with the data in the example given in the previous chapter was that it was simply an array of integers. Data will often be represented as collections of objects with multiple properties, many of which we do not need for our visualization. They are also often represented as strings instead of numeric values. In this chapter, you will learn how to select just the data that you want and to convert it to the desired data type.

Yet another issue in our previous bar graph was that we assumed that the values represented in the data had a direct mapping to the pixels in the visualization. This is normally not the case, and we need to scale the data into the size of our rendering in the browser. This can be easily accomplished using scales, which we already examined relative to axes, and now we will apply them to data.

One last issue in the previous example was that our code for calculating the size and positions of the bars was performed manually. Bar graphs are common enough in D3.js applications, and there are built-in functions that can do this for us automatically. We will examine using these to simplify our code.

So let's jump in. In this chapter, we will specifically cover the following topics:

- Loading data in JSON, TSV, or CSV formats from the Web
- Extracting fields from objects using the `.map()` function
- Converting string values into their representative numeric data types
- Using linear scales for transforming continuous values
- Using ordinal scales for mapping discrete data
- Using bands for calculating the size and position of our bars
- Applying what we've learned to date for creating a rich bar graph using real data

Data

Data is the core of creating a data visualization. Almost every visual item created in D3 will need to be bound to a piece of data. This data can come from a number of sources. It can be explicitly coded in the visualization, loaded from an external source, or result from manipulation or calculation from other data.

Most data used to create a D3.js visualization is either obtained from a file or a web service or URL. This data is often in one of many formats such as **JSON**, **XML**, **CSV (Comma Separated Values)**, and **TSV (Tab Separated Values)**. We will need to convert the data in these formats into JavaScript objects, and D3.js provides us with convenient functions for doing this.

Loading data with D3.js

D3.js provides a number of helper functions to load data from outside the browser as well as to simultaneously convert it into JavaScript objects. Probably, the most common data formats that you may come across and which we will cover are:

- JSON
- TSV
- CSV



You may have noticed that I have omitted XML from the list in our examples. D3.js does have functions to load XML, but unlike with JSON, TSV and CSV, the results of the load are not converted automatically into JavaScript objects, and require additional manipulation using the JavaScript XML/DOM facilities. XML will be considered out of scope for this text as most of the scenarios you will currently come across will be handled with these three formats, if not solely by JSON, which has become almost the ubiquitous data format for the Web.

To demonstrate working with all these formats of data, we will examine a dataset that I have put together and placed in a GitHub that represents the viewership of the episodes of Season 5 of AMC's *The Walking Dead*.



This GitHub was built manually using data on [https://en.wikipedia.org/wiki/The_Walking_Dead_\(season_5\)](https://en.wikipedia.org/wiki/The_Walking_Dead_(season_5)).

Loading JSON data

Data in the **JavaScript Object Notation (JSON)** format is convenient for conversion into JavaScript objects. It is a very flexible format which supports named properties as well as hierarchical data.

The JSON data for this example is stored in GitHub and is available at https://gist.github.com/d3byex/e5ce6526ba2208014379/raw/8fefb14cc18f0440dc00248f23cbf6aec80dcc13/walking_dead_s5.json.



The URL is a little unwieldy. You can go directly to the gist with all three versions of this data at <https://goo.gl/OfD1hc>.

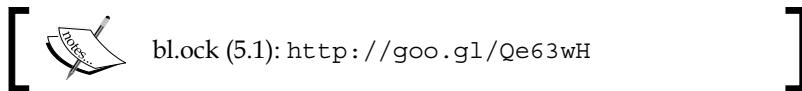
Clicking on the link will display the data in the browser. This file contains an array of JavaScript objects, each of which has six properties and represents an individual episode of the program. The first two objects are the following:

```
[
{
  "Season": 5,
  "Episode": 1,
  "SeriesNumber": 52,
  "Title": "No Sanctuary",
  "FirstAirDate": "10-12-2014",
  "USViewers": 17290000
},
```

```
{  
  "Season": 5,  
  "Episode": 2,  
  "SeriesNumber": 53,  
  "Title": "Strangers",  
  "FirstAirDate": "10-19-2014",  
  "USViewers": 15140000  
},  
...  
]
```

This data can be loaded into our D3.js application using the `d3.json()` function. This function, like many others in D3.js, performs asynchronously. It takes two parameters: the URL of the data to load, and a callback function that is called when the data has been loaded.

The following example demonstrates loading this data and displaying the first item in the array.



The main portion of the code that loads the data is as follows:

```
var url = "https://gist.githubusercontent.com/d3byex/  
e5ce6526ba2208014379/raw/8fefb14cc18f0440dc00248f23cbf6aec80dcc13/  
walking_dead_s5.json";  
d3.json(url, function (error, data) {  
  console.log(data[0]);  
});  
console.log("Data in D3.js is loaded asynchronously");
```

There is no visible output from this example, but the output is written to the JavaScript console:

```
"Data in D3.js is loaded asynchronously"  
[object Object] {  
  Episode: 1,  
  FirstAirDate: "10-12-2014",  
  Season: 5,  
  SeriesNumber: 52,  
  Title: "No Sanctuary",  
  USViewers: 17290000  
}
```

Note that the loading of data in D3.js is performed asynchronously. The output from the `console.log()` call shows that the data is loaded asynchronously and is executed first. Later, when the data is loaded, we see the output from the second call to `console.log()`.

The callback function itself has two parameters. The first is a reference to an object representing an error if one occurs. In such a case, this variable will be non-null and contain details. Non-null means the data was loaded, and is represented by the `data` variable.

Loading TSV data

TSV is a type of data that you will come across if you do enough D3.js programming. In a TSV file, the values are separated by tab characters. Generally, the first line of the file is a tab-separated sequence of names for each of the values.

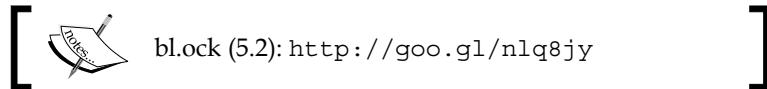
TSV files have the benefit of being less verbose than JSON files, and are often generated automatically by many systems that are not JavaScript based.

The episode data in the TSV format is available at https://gist.github.com/d3byex/e5ce6526ba2208014379/raw/8fefb14cc18f0440dc00248f23cbf6aec80dcc13/walking_dead_s5.tsv.

Clicking on the link, you will see the following in your browser:

```
Season Episode SeriesNumber Title FirstAirDate USViewers
5 1 52 No Sanctuary 10-12-2014 17290000
5 2 53 Strangers 10-19-2014 15140000
5 3 54 Four Walls and a Roof 10-26-2014 13800000
5 4 55 Slabtown 11-02-2014 14520000
5 5 56 Self Help 11-09-2014 13530000
5 6 57 Consumed 11-16-2014 14070000
5 7 58 Crossed 11-23-2014 13330000
5 8 59 Coda 11-30-2014 14810000
5 9 60 What Happened and What's Going On 02-08-2015 15640000
5 10 61 Them 02-15-2015 12270000
5 11 62 The Distance 02-22-2015 13440000
5 12 63 Remember 03-01-2015 14430000
5 13 64 Forget 03-08-2015 14530000
5 14 65 Spend 03-15-2015 13780000
5 15 66 Try 03-22-2015 13760000
5 16 67 Conquer 03-29-2015 15780000
```

We can load the data from this file using `d3.tsv()`. The following contains the code for the example:



The code is identical to the JSON example except for the URL and the call to `d3.json()`. The output in the console is, however, different.

```
[object Object] {  
  Episode: "1",  
  FirstAirDate: "10-12-2014",  
  Season: "5",  
  SeriesNumber: "52",  
  Title: "No Sanctuary",  
  USViewers: "17290000"  
}
```

Notice that the properties `Episode`, `Season`, `SeriesNumber`, and `USViewers` are now of type string instead of integer. TSV files do not have a means of implying the type like JSON does, so everything defaults to string. These will often need to be converted to another type, and we will examine that in the next section on mapping and data conversion.

Loading CSV data

CSV is a format similar to TSV except that instead of tab characters delimiting the fields, a comma is used. CSV is a fairly common format, common as output from spreadsheet applications, which is used for creating data to be consumed by other applications in many organizations.

The CSV version of the data is available at https://gist.github.com/d3byex/e5ce6526ba2208014379/raw/8fefb14cc18f0440dc00248f23cbf6aec80dcc13/walking_dead_s5.csv.

Opening the link, you will see the following in your browser:

```
Season,Episode,SeriesNumber,Title,FirstAirDate,USViewers  
5,1,52,No Sanctuary,10-12-2014,17290000  
5,2,53,Strangers,10-19-2014,15140000  
5,3,54,Four Walls and a Roof,10-26-2014,13800000  
5,4,55,Slabtown,11-02-2014,14520000  
5,5,56,Self Help,11-09-2014,13530000
```

```
5,6,57,Consumed,11-16-2014,14070000
5,7,58,Crossed,11-23-2014,13330000
5,8,59,Coda,11-30-2014,14810000
5,9,60,What Happened and What's Going On,02-08-2015,15640000
5,10,61,Them,02-15-2015,12270000
5,11,62,The Distance,02-22-2015,13440000
5,12,63,Remember,03-01-2015,14430000
5,13,64,Forget,03-08-2015,14530000
5,14,65,Spend,03-15-2015,13780000
5,15,66,Try,03-22-2015,13760000
5,16,67,Conquer,03-29-2015,15780000
```

The example for demonstrating the loading of the preceding data using `d3.csv()` is available at the following link:



The result is identical to that of the TSV example in that all the fields are loaded as strings.

Mapping fields and converting strings to numbers

We are going to use this data (in its CSV source) to render a bar graph that shows us the comparison of the viewership levels for each episode. If we are to use these fields as-is for creating the bar graph, those values will be interpreted incorrectly as their types are strings instead of numbers, and our resulting graph will be incorrect.

Additionally, for the purpose of creating a bar chart showing viewership, we don't need the properties and can omit the `Season`, `SeriesNumber`, and `FirstAirDate` fields. It's not a real issue with this dataset, but sometimes, the data can have hundreds of columns and billions of rows, so it will be more efficient to extract only the necessary properties to help save memory.

These can be accomplished in a naive manner using a `for` loop, copying the desired fields into a new JavaScript object, and using one of the parse functions to convert the data. D3.js gives us a better way, a functional way, to perform this task.

D3.js provides us with the a `.map()` function that can be used on an array, which will apply a function to each of the array's items. This function returns a JavaScript object, and D3.js collects all these objects and returns them in an array. This gives us a simple way of selecting just the properties that we want and to convert the data, all in a single statement.

To demonstrate this in action, open the example given at the following link:



The important portion of the code is the call to `data.map()`:

```
var mappedAndConverted = data.map(function(d) {
    return {
        Episode: +d.Episode,
        USViewers: +d.USViewers,
        Title: d.Title
    };
});
console.log(mappedAndConverted);
```

The function that is passed to the `.map()` returns a new JavaScript object for each item in the array `data`. This new object consists of only the three specified properties. These objects are all collected by `.map()` and stored in the `mappedAndConverted` variable.

The following code shows the first two objects in the new array:

```
[{"object Object": {
    Episode: 1,
    Title: "No Sanctuary",
    USViewers: 17290000
}, {"object Object": {
    Episode: 2,
    Title: "Strangers",
    USViewers: 15140000
}},
```

Note that `Episode` and `USViewers` are now numeric values. This is accomplished by applying the unary `+` operator, which will convert a string to its appropriate numeric type.

Scales

Scales are functions provided by D3.js that map a set of values to another set of values. The input set of values is referred to as the domain, and the output is the range. The basic reason for the existence of scales is to prevent us from coding loops, and doing a lot of math to make these conversions happen. This is a very useful thing.

There are three general categories of scales: quantitative, ordinal, and time-scale. Within each category of scale, D3.js provides a number of concrete implementations that exist for accomplishing a specific type of mapping data useful for data visualization.

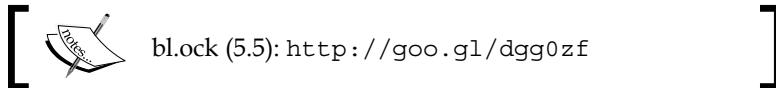
Covering examples of every type of scale would consume more space than is available in this book, and at the same time become tedious to read. We will examine several common scales that are used – kind of the 80/20 rule, where the few we cover here will be used most of the time you use scales.

Linear scales

Linear scales are a type of quantitative scale that are arguably the most commonly used ones. The mapping performed is linear in that the output range is calculated using a linear function of the input domain.

A good example of using a linear scale is the scenario with our *The Walking Dead* viewership data. We need to draw bars from this data; but if we use the code that we used earlier in the book, our bars will be extremely tall since that code has a one to one mapping between the value and the pixels.

Let's assume that our area for the bars on the graph has a height of 400 pixels. We would like to map the lowest viewership value to a bar that is 100 pixels tall, and map the largest viewership value to 400. The following example performs this task:



The code starts, as with the CSV example, by loading that data and mapping/converting it. The next task is to determine the minimum and maximum viewership values:

```
var viewership = mappedAndConverted.map(function (d) {
    return d.USViewers;
});
var minViewership = d3.min(viewership);
var maxViewership = d3.max(viewership);
```

Next, we define several variables representing the minimum and maximum height that we would like for the bars:

```
var minBarHeight = 100, maxBarHeight = 400;
```

The scale is then created as follows:

```
var yScale = d3.scale
    .linear()
    .domain([minViewership, maxViewership])
    .range([minBarHeight, maxBarHeight]);
```

We can now use the `yScale` object as though it is a function. The following will log the results of scaling the minimum and maximum viewership values:

```
console.log(minViewership + " -> " + yScale(minViewership));
console.log(maxViewership + " -> " + yScale(maxViewership));
```

Examining the console output, we can see that the scaling resulted in the expected values:

```
"12270000 -> 100"
"17290000 -> 400"
```

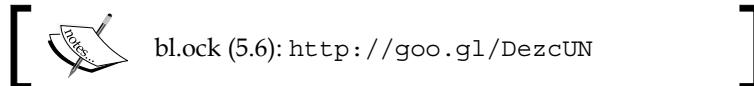
Ordinal scales

Ordinal scales are, in a way, similar to dictionary objects. The values in the domain and range are discrete. There must be an entry in the range for every unique input value, and that value must have a mapping to a single value in the range.

There are several common uses for ordinal scales, and we will examine four common uses that we will use throughout the remainder of this book.

Mapping color strings to codes

Open the following link for an example of an ordinal scale. This example does not use the data from *The Walking Dead*, and simply demonstrates the mapping of string literals representing primary colors into the corresponding color codes.



bl.ock (5.6): <http://goo.gl/DezcUN>

The scale is created as follows:

```
var colorScale = d3.scale.ordinal()  
  .domain(['red', 'green', 'blue'])  
  .range(['#ff0000', '#00ff00', '#0000ff']);
```

We can now pass any of the range values to the `colorScale`, as demonstrated with the following:

```
console.log(colorScale('red'),  
           colorScale('green'),  
           colorScale('blue'));
```

Examining the console output, we can see the results of this mapping as follows:

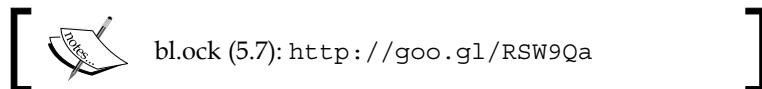
```
"#ff0000"  
"#00ff00"  
"#0000ff"
```

Mapping integers to color scales

D3.js comes with several special built-in scales that are referred to as **categorical** scales. It sounds like a fancy term, but they are simply mappings of a set of integers to unique colors (unique within that scale).

These are useful when you have a set of sequential 0-based integer keys in your data, and you want to use a unique color for each, but you do not want to manually create all the mappings (like we did for the three strings in the previous example).

Open the following link for an example of using a 10 color categorical scale:



The preceding example renders 10 adjacent rectangles, each with a unique color from a `category10()` color scale. You will see this in your browser when executing this example.



The example starts by creating an array of 10 integers from 0 to 9.

```
var data = d3.range(0, 9);
```

The scale is created next:

```
var colorScale = d3.scale.category10();
```

Now we can bind the integers to the rectangles, and set the fill for each by passing the value to the `colorScale` function:

```
var svg = d3.select('body')
    .append('svg')
    .attr({width: 200, height: 20});

svg.selectAll('rect')
    .data(data)
    .enter()
    .append('rect')
    .attr({
        fill: function(d) { return colorScale(d); },
        x: function(d, i) { return i * 20 },
        width: 20,
        height: 20
    });
}
```

D3.js provides four sets of categorical color scales that can be used depending upon your scenario. You can take a look at them on the D3.js documentation page at <https://github.com/mbostock/d3/wiki/Ordinal-Scales>.

The ordinal scale using rangeBands

In *Chapter 4, Creating a Bar Graph*, when we drew the graph we calculated the positions of the bars based upon a fixed bar size and padding. This is actually a very inflexible means of accomplishing this task. D3.js gives us a special scale that we can use, given the domain values and essentially a width, that will tell us the start and end values for each bar such that all the bars fit perfectly within the range!

Let's take a look using this special scale with the following example:



This example creates a simple ordinal scale specifying the range using the `.rangeBands()` function instead of `.range()`. The entire code of the example is as follows:

```
var bands = d3.scale.ordinal()
  .domain([0, 1, 2])
  .rangeBands([0, 100]);
console.log(bands.range());
console.log(bands.rangeBand());
```

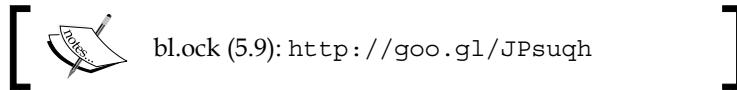
The `.range()` function will return an array with values representing the extents of an equal number of evenly-spaced divisions of the range specified to `.rangeBands()`. In this case, the width of the range is 100, and there are three items specified in the domain; hence, the result is the following:

```
[0, 33.33333333333336, 66.66666666666667]
```

Technically, this result is the values that represent the start of each band. The width of each band can be found using the `.rangeBand()` function, in this case returning the following:

```
33.33333333333336
```

This width may seem simplistic. Why have this function if we can just calculate the difference between two adjacent values in the result of `.range()`? To demonstrate, let's look at a slight modification of this example, available at the following link.



This makes one modification to the call to `.rangeBands()`, adding an additional parameter that specifies the padding that should exist between the bars:

```
var bands = d3.scale.ordinal()
  .domain([0, 1, 2])
  .rangeRoundBands([0, 100], 0.1);
```

The output differs slightly due to the addition of padding between the bands:

```
[3.2258064516129035, 35.483870967741936, 67.74193548387096]
29.032258064516128
```

The width of each band is now 29.03, with a padding of 3.23 between bands (including on the outside of the two outer bands).

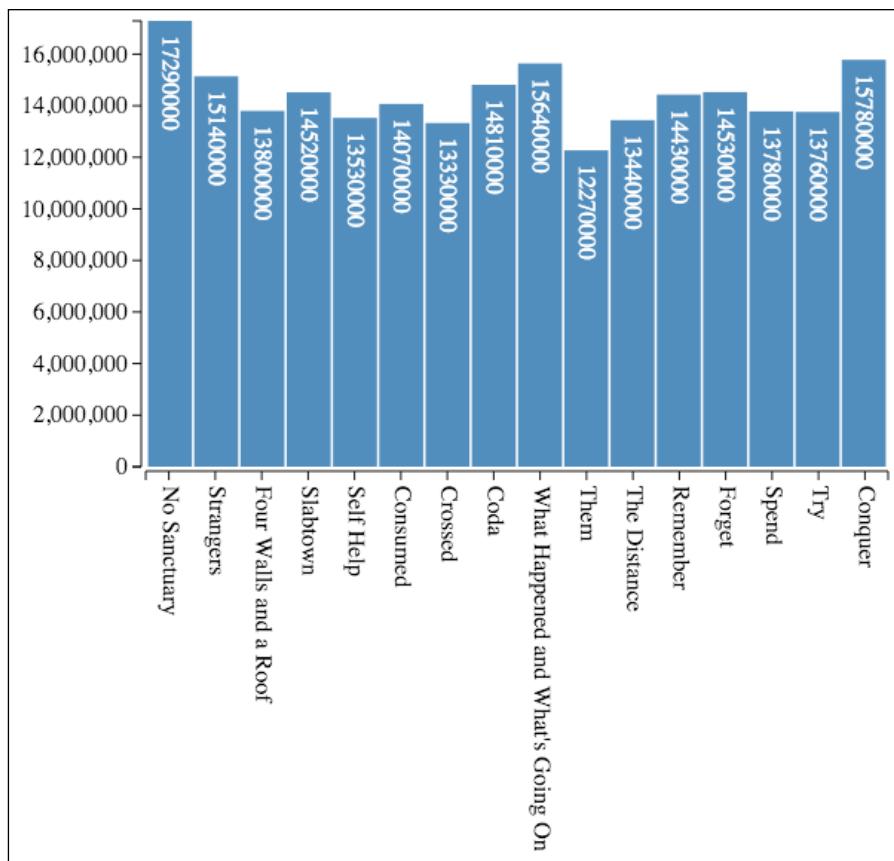
The value for padding is a value between 0.0 (the default, and which results in a padding of 0) and 1.0, resulting in bands of width 0.0. A value of 0.5 makes the padding the same width as each band.

Visualizing The Walking Dead viewership

Now we pull everything from the chapter together to render a bar graph of the viewership across all the episodes of *The Walking Dead*:



The output of the preceding example is as follows:



Now let's step through how this is created. After loading the data from the JSON file, the first thing that is performed is the extraction of the `usviewership` values and the determining of the maximum value:

```
var viewership = data.map(function (d) {  
    return d.USViewers;  
});  
  
var maxViewers = d3.max(viewership);
```

Then various variables, which represent various metrics for the graph, and the main SVG element are created:

```
var margin = { top: 10, right: 10, bottom: 260, left: 85 };  
  
var graphWidth = 500, graphHeight = 300;  
  
var totalWidth = graphWidth + margin.left + margin.right;  
var totalHeight = graphHeight + margin.top + margin.bottom;  
  
var axisPadding = 3;  
  
var svg = d3.select('body')  
    .append('svg')  
    .attr({ width: totalWidth, height: totalHeight });
```

The container for holding the bars is created next:

```
var mainGroup = svg  
    .append('g')  
    .attr('transform', 'translate(' + margin.left + ',' +  
          margin.top + ')');
```

Now we create an ordinal scale for the bars using `.rangeBands()`. We will use this to calculate the bar position and padding:

```
var bands = d3.scale.ordinal()  
    .domain(viewership)  
    .rangeBands([0, graphWidth], 0.05);
```

We also require a scale to calculate the height of each bar:

```
var yScale = d3.scale  
    .linear()  
    .domain([0, maxViewers])  
    .range([0, graphHeight]);
```

The following function is used by the selection that creates the bars to position each of them:

```
function translator(d, i) {
    return "translate(" + bands.range() [i] + "," +
           (graphHeight - yScale(d)) + ")";
}
```

Now we create the groups for the content of each bar:

```
var barGroup = mainGroup.selectAll('g')
    .data(viewership)
    .enter()
    .append('g')
    .attr('transform', translator);
```

Next we append the rectangle for the bar:

```
barGroup.append('rect')
    .attr({
        fill: 'steelblue',
        width: bands.rangeBand(),
        height: function(d) { return yScale(d); }
    });
});
```

And then add a label to the bar to show the exact viewership value:

```
barGroup.append('text')
    .text(function(d) { return d; })
    .style('text-anchor', 'start')
    .attr({
        dx: 10,
        dy: -10,
        transform: 'rotate(90)',
        fill: 'white'
    });
});
```

The bars are now complete, so we move on to creating both the axes. We start with the left axis:

```
var leftAxisGroup = svg.append('g');
leftAxisGroup.attr({
    transform: 'translate(' + (margin.left - axisPadding) + ',' +
               margin.top + ')'
});

var yAxisScale = d3.scale
```

```
.linear()
.domain([maxViewers, 0])
.range([0, graphHeight]);  
  
var leftAxis = d3.svg.axis()
.orient('left')
.scale(yAxisScale);
var leftAxisNodes = leftAxisGroup.call(leftAxis);
styleAxisNodes(leftAxisNodes);
```

And now create a bottom axis which displays the titles:

```
var titles = data.map(function(d) { return d.Title; });
var bottomAxisScale = d3.scale.ordinal()
.domain(titles)
.rangeBands([axisPadding, graphWidth + axisPadding]);  
  
var bottomAxis = d3.svg
.axis()
.scale(bottomAxisScale)
.orient("bottom");  
  
var bottomAxisX = margin.left - axisPadding;
var bottomAxisY = totalHeight - margin.bottom + axisPadding;  
  
var bottomAxisGroup = svg.append("g")
.attr({ transform: 'translate(' + bottomAxisX + ',' + bottomAxisY
+ ')' });  
  
var bottomAxisNodes = bottomAxisGroup.call(bottomAxis);
styleAxisNodes(bottomAxisNodes);  
  
bottomAxisNodes.selectAll("text")
.style('text-anchor', 'start')
.attr({
dx: 10,
dy: -5,
transform: 'rotate(90)'
});
```

The following function is reusable code for styling the axes:

```
function styleAxisNodes(axisNodes) {
axisNodes.selectAll('.domain')
.attr({
fill: 'none',
```

```
        'stroke-width': 1,
        stroke: 'black'
    });
axisNodes.selectAll('.tick line')
    .attr({
        fill: 'none',
        'stroke-width': 1,
        stroke: 'black'
    });
}
```

Summary

In this chapter, you learned how to load data from the web and use it as the basis for a bar graph. We started with loading data in the JSON, CSV, and TSV formats. You learned how to use the `.map()` function to extract just the values that you desire from this data, and examined the issues and solutions needed for converting string values into numeric values.

Next we covered scales in some more detail, and looked at several examples of the ways to use scales for mapping data from one range of values to another as well as to map discrete values such as color names to color codes. We covered categorical scales, a means of mapping integer values into predefined color maps, and a concept that we will use frequently in our examples. Our examination of scales ended with a demonstration of using `.rangeBands()`, and how it can help us size and place bars within a predefined area.

We closed the chapter by combining all of these concepts together into, what is up to this point, our best example of generating a bar chart. This demonstrated loading the data, using multiple scales for both data and axes, and using `.rangeBands()` to determine the placement of the bars, as well as using not only a vertical but also a horizontal axis.

In the next chapter, we will branch out of bar graphs into another type of data visualization—scatter (and bubble) plots.

6

Creating Scatter and Bubble Plots

In this chapter, we extend our examples of using D3.js for plotting data to explain how to create scatter and bubble plots. Scatter and bubble plots visualize multivariate data, as compared to univariate data that is visualized by bar charts. Multivariate data consists of two or more variables, and scatter plots allow us to visualize two variable, and bubble plots extend this to three or four variables.

We will begin by first creating a simple scatter plot with fixed symbols, based upon stock correlation data. We start with using solid circles for symbols, and will progress through several enhancements including the use of color, outlines, and opacity. We will wrap up scatter plots with an example of multiple, overlying sets of data, each using different symbols and colors.

When we've finished examining the of creation of a bubble plot, we will extend that example to change the size of the points based upon the data, and then to color the points based upon categorical information. This last example will demonstrate how we can visualize four different variables within a single visualization, and how the use of visuals can help us derive meaning from the underlying information.

Specifically, in this chapter we will cover the following topics:

- Creating a basic scatter plot using fixed-sized and solid points
- Using outlines instead of solid fills to make the plot more legible
- Adding gridlines to help determine the location of points
- Extending the scatter plot code to create bubble plots

Creating scatter plots

Scatter plots consist of two axes, one for each variable. Each axis can be based on continuous or categorical variables. For each measurement (a **measurement** being the paired combination of X and Y values), a symbol is placed on the plot at the specified location. The end result is a plot that allows the person viewing it to determine how much one variable is affected by the other.

Underpinning our first few examples will be a data set that represents the correlation between the AAPL and MSFT stocks on a daily basis for the year 2014. For purposes of creating a scatter plot, the meaning of this data is not important—it is just that it represents two dimensional data, where the value for each stock represents a location on the respective axis.

The data for this example is available at <https://goo.gl/BZkC8B>.

Opening this link in the browser, you will see the following as the first few lines of the data:

```
Date,AAPL,MSFT
2014-01-02,-0.01406,-0.00668
2014-01-03,-0.02197,-0.00673
2014-01-06,0.00545,-0.02113
2014-01-07,-0.00715,0.00775
2014-01-08,0.00633,-0.01785
2014-01-09,-0.01277,-0.00643
2014-01-10,-0.00667,0.01435
2014-01-13,0.00524,-0.02941
2014-01-14,0.0199,0.02287
2014-01-15,0.02008,0.02739
```

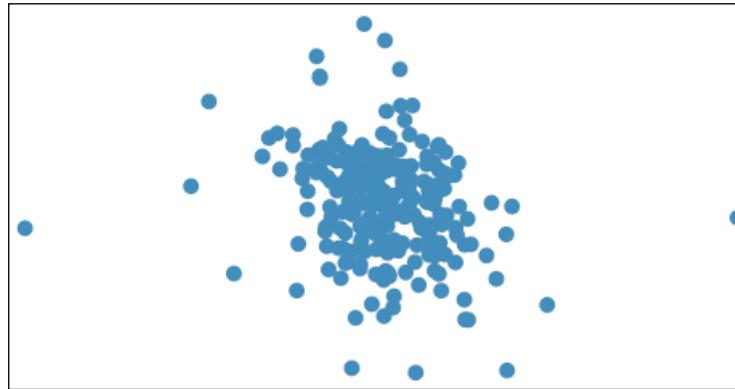
Plotting points

Our first example will demonstrate the process of drawing points in a scatter plot. To keep it simple, it forgoes the axes and other stylistic elements (these will be added in the next example).

The example is available at the following location:



The resulting plot is seen in the following image:



Now let's examine how this is created. The example starts by loading the data:

```
var url = "https://gist.githubusercontent.com/
d3byex/520e6dc30e673c149cc/raw/432623f00f6740021bdc13141612ac0b619
6b022/corr_aapl_msft.csv";
d3.csv(url, function (error, rawData) {
```



The entire URL has to be specified, as apparently, the data load functions do not follow redirects.



We need to convert the properties AAPL and MSFT from strings to numbers. We do this by creating a new array of objects with x and y properties, with AAPL mapped into x and MSFT into y, which also converts the data type:

```
var data = rawData.map(function(d) {
    return { X: +d.AAPL, Y: +d.MSFT }
});
```

To effectively scale a scatter plot, we need to know the extents of the data in both the x and y series:

```
var xExtents = d3.extent(data, function(d) { return d.X; });
var yExtents = d3.extent(data, function(d) { return d.Y; });
```

These values will help us create the scales that are required for both dimensions of the graph. This plot will actually use the maximum absolute value of these four extents. We can determine this value with the following:

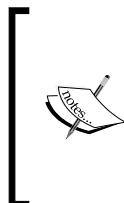
```
var maxExtent = d3.max(  
    xExtents.concat(yExtents),  
    function(d) { return Math.abs(d);  
});
```

We are now ready to create the properties of the graph, including its width, height, and the size of the radius for the circles that will represent the points:

```
var graphWidth = 400, graphHeight = 400;  
var radius = 5;
```

Now we have all of the information required to create the scale needed to map the data into the locations in the rendering:

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



This example (and those in the remainder of this chapter) scale the data such that the domain is the negative and positive of the absolute values of the extents. In simple terms, this scale ensures that when rendering into a square canvas, all points are visible and any specific distance along the X dimension represents an identical change in the value of the data as the distance along the Y dimension.



The rendering begins with the creation of the main SVG element:

```
var svg = d3.select('body')  
    .append('svg')  
    .attr('width', graphWidth)  
    .attr('height', graphHeight);
```

Finally, we create a circle of the specified radius to represent each point:

```
svg.selectAll('circle')  
    .data(data)  
    .enter()  
    .append('circle')  
    .attr({  
        cx: function(d) { return xScale(dAAPL); },  
        cy: function(d) { return yScale(dMSFT); },  
        r: radius,
```

```

        fill: 'steelblue'
    });
}); // closing the call to d3.csv

```

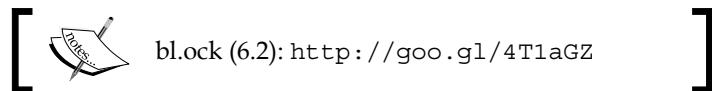
Congratulations, you have created your first scatter plot!

Sprucing up the scatter plot

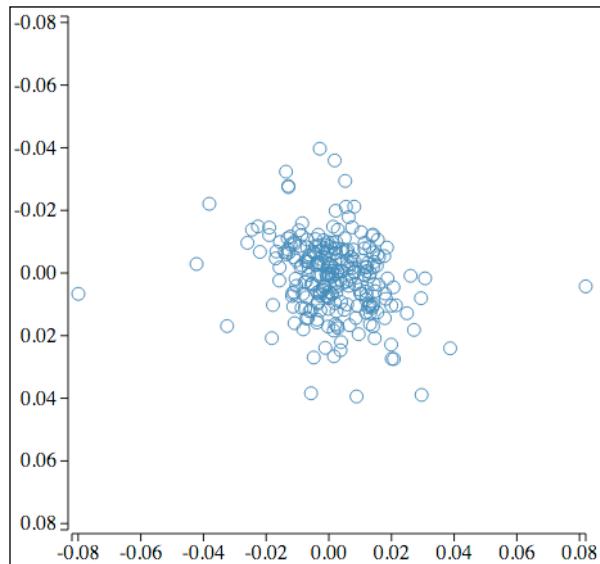
There are several issues with the plot in the previous example. First, notice that there is a circle that is clipped towards the right boundary. With the code as it is, one point, the one at the maximum extent, will have half of its area clipped. This can easily be resolved by including margins that are at least half of the radius of the circles.

Another issue is that there are a lot of circles that overlap, confusing the visual understanding of the data. A common means of addressing this issue in scatter plots is not to use a solid fill in the circles, and simply use an outline instead.

A final issue, which is really just a decision made to keep the previous example simple, is not to have any axes. The example at the following link addresses each of these concerns:



The preceding example has the following output:



This result is a much more effective scatter plot. We can make sense of the previously obscured points, and the axes give us a sense of the values at each point.

The changes to the code are relatively minor. Besides adding axes by using the code that we have seen in other examples (including grouping groups for the various main elements), and sizing the main SVG element to account for those axes, the only change is in the way the circles are created:

```
graphGroup.selectAll('circle')
  .data(data)
  .enter()
  .append('circle')
  .attr({
    cx: function(d) { return scale(d.X); },
    cy: function(d) { return scale(d.Y); },
    r: radius,
    fill: 'none',
    stroke: 'steelblue'
  });
}
```

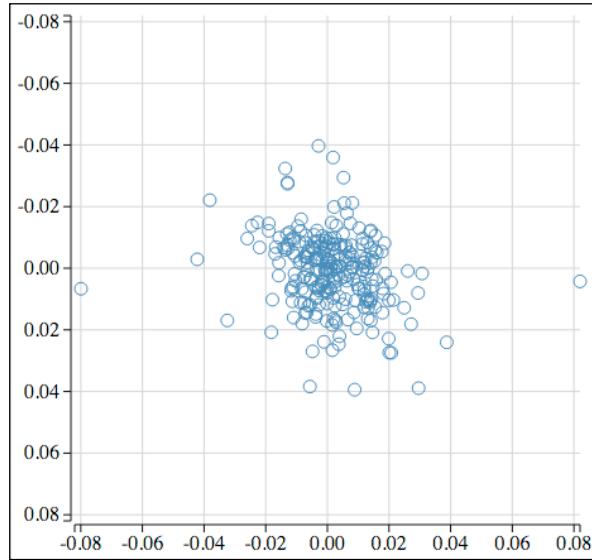
Adding gridlines

Our scatter plot would be more effective if it had gridlines. The way in which we add gridlines to a chart in D3.js is actually a little trick: gridlines are actually the ticks of an axis, the ticks being the width or height of the graphic with both the labels and main line of the axis hidden.

To add gridlines to our plot, we will create two additional axes. The horizontal gridlines will be rendered by creating a left-oriented axis positioned in the right margin. We will set the labels on this axis to be empty and the line of the axis to be hidden. The ticks are then sized to extend all the way back to the other axis in the left margin. We will perform a similar process to create the vertical gridlines except with a bottom axis placed in the top margin.



And the resulting graph is shown in the following image:



The only difference from the previous examples are the several lines for creating these new axes and a function to style them:

```
var yGridlinesAxis = d3.svg.axis().scale(scale).orient("left");
var yGridlineNodes = svg.append('g')
    .attr('transform', 'translate(' + (margins.left + graphWidth)
        + ',' + margins.top + ')')
    .call(yGridlinesAxis
        .tickSize(graphWidth + axisPadding, 0, 0)
        .tickFormat ""));
styleGridlineNodes(yGridlineNodes);
```

The code begins with creating a left-oriented axis, and then renders it in a group which is translated to the right margin.

Instead of simply passing the axis object to `.call()`, we first call two of its functions. The first, `.tickSize()`, sets the size of the ticks to stretch across the entire area where the points will be rendered. Calling `.tickFormat("")` informs the axis that the labels should be empty.

Now we just need to perform a little styling on the axis. This is performed by the `styleGridLineNodes()` function:

```
function styleGridlineNodes(axisNodes) {
    axisNodes.selectAll('.domain')
        .attr({
            fill: 'none',
            stroke: 'none'
        });
    axisNodes.selectAll('.tick line')
        .attr({
            fill: 'none',
            'stroke-width': 1,
            stroke: 'lightgray'
        });
}
```

This sets the fill and stroke of the main line of the axis so that it is not visible. It then makes the actual ticks light gray.

The vertical gridlines are then created by a similar process:

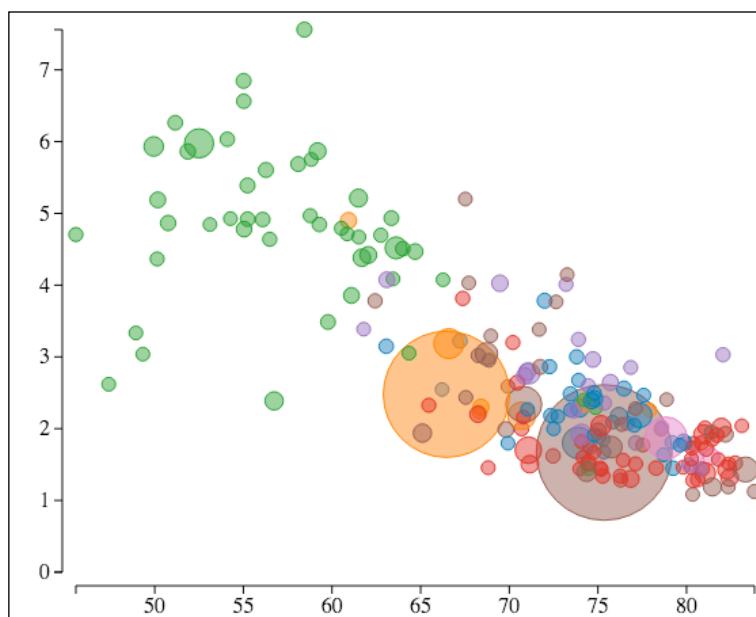
```
var xGridlinesAxis = d3.svg.axis().scale(scale).orient("bottom");
var xGridlineNodes = svg.append('g')
    .attr('transform', 'translate(' + margins.left + ', ' +
           (totalHeight - margins.bottom + axisPadding) + ')')
    .call(xGridlinesAxis
        .tickSize(-graphWidth - axisPadding, 0, 0)
        .tickFormat ""));
styleGridlineNodes(xGridlineNodes);
```

A final point about this process is the sequence of the renderings: gridlines, then axes, then the points. This ensures that each of these appear on top of the others. It is most important for the points to be on top of the gridlines and axes, but the gridlines also being behind the visible axes is good practice. It gives you a little wiggle room to be a few pixels long on the gridlines.

Creating a bubble plot

Bubble plots help us to visualize three or four dimensions of data. Each datum in a bubble plot consists not only of two values used to plot against the X and Y axes, but also one or two additional values which are commonly represented by different size symbols and/or colors.

To demonstrate a bubble plot, the following image shows the result of our example:



The data behind this chart is a data set that was pulled together from three different datasets from the World Bank. This data correlates life expectancy at birth relative to the fertility rate for all the countries in the World Bank data for the year 2013.

This chart plots age along the X axis and the birth rate along the Y axis. The relative population of a country is represented by the size of the circle, and the color of the circle represents the economic region of the country as categorized by the World Bank.

We won't dive deeply into this data. It is available at <https://goo.gl/K3yuuy>.

The first few lines of the data are the following:

```
CountryCode,CountryName,LifeExp,FertRate,Population,Region  
ABW,Aruba,75.33217073,1.673,102911,Latin America & Caribbean  
AFG,Afghanistan,60.93141463,4.9,30551674,South Asia
```

If you want to check out the original data, you can use the following links:

- Life expectancy (in years) at birth: <http://data.worldbank.org/indicator/SP.DYN.LE00.IN>
- Fertility rate: <http://data.worldbank.org/indicator/SP.DYN.TFRT.IN>
- Total population: <http://data.worldbank.org/indicator/SP.POP.TOTL>

The code for the example is available at the following link:

block (6.4): <http://goo.gl/KQJceE>

The example starts with the loading of data and converting the data types:

```
var url = "https://gist.githubusercontent.com/  
d3byex/30231953acaa9433a46f/raw/6c7eb1c562de92bdf8d0cd99c6912048161c18  
7e/fert_pop_exp.csv";  
var data = rawData.map(function(d) {  
    return {  
        CountryCode: d.CountryCode,  
        CountryName: d.CountryName,  
        LifeExp: +d.LifeExp,  
        FertRate: +d.FertRate,  
        Population: +d.Population,  
        Region: d.Region  
    }  
});
```

Now we define several variables for defining the minimum and maximum bubble size and the margins, which we will set to be half of the radius of the largest bubble:

```
var minBubbleSize = 5, maxBubbleSize = 50;  
var margin = { left: maxBubbleSize/2, top: maxBubbleSize/2,  
              bottom: maxBubbleSize/2, right: maxBubbleSize/2  
};
```

This particular plot requires three linear scales and one ordinal scale based upon the following four series of data:

```
var lifeExpectancy = data.map(function(d) { return d.LifeExp; });
var fertilityRate = data.map(function(d) { return d.FertRate; });
var population = data.map(function(d) { return d.Population; });
var regions = data.map(function(d) { return d.Region; });
```

The scale for the X axis will vary from the minimum to the maximum life expectancy:

```
var xScale = d3.scale.linear()
    .domain([d3.min(lifeExpectancy), d3.max(lifeExpectancy)])
    .range([0, graphWidth]);
```

The Y axis will range from the maximum fertility rate at the top to 0 at the bottom:

```
var yScale = d3.scale.linear()
    .domain([d3.max(fertilityRate), 0])
    .range([0, graphHeight]);
```

The size of each bubble represents the population, and will range in radius from the minimum to the maximum values configured earlier:

```
var popScale = d3.scale.linear()
    .domain(d3.extent(population))
    .range([minBubbleSize, maxBubbleSize]);
```

Each bubble will be colored based upon the value of the region. To do this, we set up a mapping between each of the unique names of the regions and a 10-color categorical scale:

```
var uniqueRegions = d3.set(regions).values();
var regionColorMap = d3.scale.ordinal()
    .domain(uniqueRegions)
    .range(d3.scale.category10().range());
```

Now we can start rendering the visuals, starting with the axes:

```
var yAxis = d3.svg.axis().scale(yScale).orient('left');
var yAxisNodes = svg.append('g')
    .attr('transform', 'translate(' +
        (margin.left - axisPadding) + ',' + margin.top + ')')
    .call(yAxis);
styleAxisNodes(yAxisNodes);
```

```
var xAxis = d3.svg.axis().scale(xScale).orient('bottom');
var xAxisNodes = svg.append('g')
    .attr('transform', 'translate(' + margin.left + ',' +
        (totalHeight - margin.bottom + axisPadding) + ')')
    .call(xAxis);
styleAxisNodes(xAxisNodes);
```

The final task is to render the bubbles:

```
svg.append('g')
    .attr('transform', 'translate(' + margin.left + ',' +
        margin.top + ')')
    .selectAll('circle')
    .data(data)
    .enter()
    .append('circle')
    .each(function(d) {
        d3.select(this).attr({
            cx: xScale(d.LifeExp),
            cy: yScale(d.FertRate),
            r: popScale(d.Population),
            fill: regionColorMap(d.Region),
            stroke: regionColorMap(d.Region),
            'fill-opacity': 0.5
        });
    });
});
```

Summary

In this chapter, we put together several examples of creating scatter and bubble plots. You learned a number of techniques for organizing data that represents between two and four distinct dimensions, using axes for two of the dimensions, and then using color and size-of-points as two more.

In the next chapter, we will begin with animation. We will start with the fundamentals of animation, and by the end of the chapter, we will extend this chapter's final example and use animation to represent an extra dimension, a fifth dimension—time.

7

Creating Animated Visuals

We are now going to look at using D3.js transitions to represent changes in the information underlying a visual. We will start with examples for examining several concepts involved in using D3.js to animate the properties of visual elements from one state to another.

By the end of this chapter, we will extend the bubble visualization from *Chapter 6, Creating Scatter and Bubble Plots*, to demonstrate how we can animate our bubbles as we move through multiple years of data. This will demonstrate the construction of a relatively complex animation through which a user can easily deduce trends in the information.

In this chapter, we will cover the following topics through examples:

- Animating using transitions
- Animating the fill color of a rectangle
- Animating multiple properties simultaneously
- Delaying an animation
- Creating chained transitions
- Handling the start and end events of transitions
- Changing the content and size of text using tweening
- Using timers to schedule the steps of an animation
- Adding a fifth dimension to a bubble chart through animation: time

Introduction to animation

D3.js provides extensive capabilities for animating your visualizations. Through the use of animation, we can provide the viewer with a means to understanding how data changes over time.

Animation in D3.js is all about changing the properties of the visual objects over time. When these properties are changed, the DOM is updated and the visual is modified to represent the new state.

To animate properties, D3.js provides the following capabilities that we will examine:

- Transitions
- Interpolators and tweenings
- Easings
- Timers

Animating using transitions

D3.js animations are implemented via the concept of **transitions**. Transitions provide instructions and information to D3.js for changing one or more visual attribute values over a specific duration of time.

When D3.js starts a transition on a visual, it calculates the initial style and ending style for the element that is being transitioned. These are often referred to as the start and end **keyframes**. Each keyframe is a set of styles and other properties that you can specify as part of the animation. D3.js will then animate those properties from the start values to the end values.

Animating the fill color of a rectangle

To demonstrate a transition in action, we will start with an example and animate the color of a rectangle from one color to another. The code for this example is available at the following link:



In this example, we start by creating the following SVG rectangle and setting its initial `fill` to `red`, followed by transitioning the fill color to `blue` over a period of five seconds.

When running the example, you will see a single rectangle that changes from red to blue over a period of five seconds. During that time, it smoothly animates through intermediate colors such as purple, as seen in the following image:



The primary part of this code that does the animation is the following; it starts by creating the rectangle and setting its initial color to red:

```
svg.append('rect')
  .attr({
    x: '10px',
    y: '10px',
    width: 80,
    height: 80,
    fill: 'red'
  })
  .transition()
  .duration(5000)
  .attr({ fill: 'blue' });
```

The call to `.transition()` informs D3.js that we want to transition one or more properties of the `rect` element that are made to the attributes of the `rect` element using calls to `.style()` or `.attr()`.

The call to `.transition()` instructs D3.js to track any changes that are made to the attributes of the SVG element using calls to `.style()` or `.attr()`.

In this case, we specify that the `fill` of the rectangle should be `blue` at the end of the transition. D3.js uses this to calculate the starting and ending keyframes, which tracks the `fill` on the rectangle should change from red to blue in this case.

When the rendering of these elements begins, D3.js also starts the animation and smoothly changes the `fill` property over the specified period.

Animating multiple properties simultaneously

Multiple properties can be animated on an object during a transition. To accomplish this, all that is required is to set multiple attributes after the call to `.transition()`.

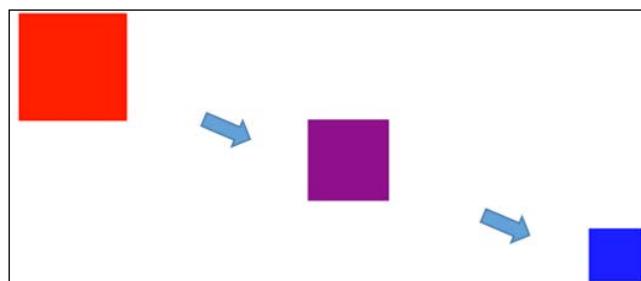
As an example, the following code animates the position of the rectangle and its size over the five-second period:



The code extends the previous example by animating not only the fill, but also by changing the position to move the rectangle along a diagonal, and modifying the size to make the rectangle half the width and height at the end of the transition:

```
svg.append('rect')
  .attr({
    x: 10,
    y: 10,
    width: 80,
    height: 80,
    fill: 'red'
  })
  .transition()
  .duration(5000)
  .attr({
    x: 460,
    y: 150,
    width: 40,
    height: 40,
    fill: 'blue'
  });
}
```

The resulting animation looks like the following image, where the rectangle moves along the path of the arrows, while changing both, color and size:



Delaying a transition

If you do not want an animation to start instantaneously, you can use a delay. A delay defers the start of the transition for the specified period of time.

The following example defers the start of the transition for one second, then runs the transition for four seconds, completing the transition in an overall time of five seconds.

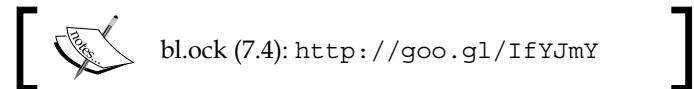


The code for the preceding example is the same as the previous one except for the following lines:

```
.transition()
.delay(1000)
.duration(4000)
```

Creating chained transitions

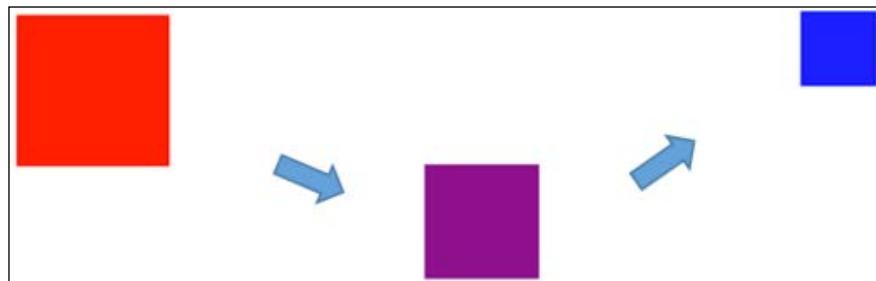
A single transition changes the properties between only one set of keyframes. However, it is possible to chain transitions together for providing multiple sequences of animations. The following example demonstrates the chaining of two transitions (and also a delay at the start).



The first transition is executed for two seconds and animates the size, color, and position of the rectangle to the middle of the SVG area. The second transition then moves the rectangle to the upper-right corner for another two seconds while still continuing to change its color and size. The total execution time remains five seconds:

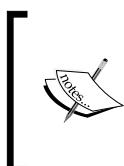
```
svg.append('rect')
  .attr({
    x: 10,
    'y': 10,
    width: 80,
    height: 80,
    fill: 'red'
  })
```

```
.transition()  
.delay(1000)  
.duration(2000)  
.attr({  
    x: 240,  
    y: 80,  
    width: 60,  
    height: 60,  
    fill: 'purple'  
})  
.transition()  
.duration(2000)  
.attr({  
    width: 40,  
    height: 40,  
    x: 460,  
    y: 10,  
    fill: 'blue'  
});
```



Handling the start and end events of transitions

It is possible to handle the start and end events of a transition using the `.each()` function. This is useful for ensuring that the starting or ending style is exactly what you desire at the start or end of the transition. This can be an issue when interpolators (covered in the next section) are at an exact expected value, where the start values are not known until the animation is running, or there are browser-specific issues that need to be addressed.



An example of a browser issue is that of transparent colors being represented by `rgba(0, 0, 0, 0)`. This is black but completely transparent. However, an animation using this will always start with fully opaque black. The start event can be used to patch up the color at the start of the animation.

The following example demonstrates hooking on to the start event of the first transition and the end event of the second transition by modifying the previous example:



block (7.5): <http://goo.gl/746hLo>

There are two fundamental changes in this example. The hooking on to the start event of the first transition changes the color of the rectangle to green. This causes the rectangle to flash from red to green just after the delay finishes:

```
.each('start', function() {
  d3.select(this).attr({ fill: 'green' });
})
```

The following code shows the second change, which reforms the rectangle to yellow at the end of the second animation:

```
.each('end', function() {
  d3.select(this).attr({ fill: 'yellow' });
});
```

Note that when using the `.each()` function, the function that is called loses the context of the selection, and does not know the current item. We can get that back using the call to `d3.select(this)`, which will return the current datum that the functions are being applied to.

In my experience, I have found that the setting of attributes before and after transitions must use a consistent notation. If you use `.style()` prior to the transition, and then `.attr()` later, even on the same attribute, the transition will not work for that attribute. So, if you use `.style()` before `.transition()`, make sure to use `.style()` after (and vice versa for `.attr()`).

Changing the content and size of text using tweening

Tweening provides a means of telling D3.js the way to calculate property values during transitions without D3.js tracking the keyframes. Keyframes can be a performance issue when animating a large quantity of items, so tweening can help out in such situations.

Tweening gives us the opportunity to connect in our own **interpolator** for providing values at each step during an animation. An interpolator is a function that is passed a single value between 0.0 and 1.0, which represents the current percentage of the transition completed. The implementation of the interpolator then uses this value to calculate the value at that point in time.

We will look at two examples of tweening. The first example, available at the following link, animates the value of a text item from 0 to 10 over a period of ten seconds:



This is actually something that cannot be done using attribute animation. We must call the `.text()` function of the DOM element to set the text, so we cannot use that technique to animate the change in content. We have to use tweening. The following snippet from the example creates the tween that sets the text content during the animation:

```
svg.append('text')
    .attr({ x: 10, y: 50 })
    .transition()
    .duration(10000)
    .tween("mytween", function () {
        return function(t) {
            this.textContent = d3.interpolateRound(0, 10)(t);
        }
    });
});
```

The first parameter to `.tween()` is simply a name for this tween. The second parameter is a factory function which returns another function to D3.js that will be called at each step during the transition, passing it the current percentage of transition completed.

The factory function is called once for each datum at the start of the animation. The function it returns is called repeatedly, and uses the `d3.interpolateRound()` function to return rounded numbers between 0 and 10 based upon the value of `t`.

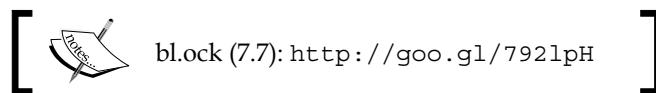
There are a number of interpolation functions provided by D3.js, such as:

- `d3.interpolateNumber`
- `d3.interpolateRound`
- `d3.interpolateString`
- `d3.interpolateRgb`
- `d3.interpolateHsl`
- `d3.interpolateLab`
- `d3.interpolateHcl`
- `d3.interpolateArray`
- `d3.interpolateObject`
- `d3.interpolateTransform`
- `d3.interpolateZoom`

D3.js also has a function `d3.interpolate(a, b)`, which returns the appropriate interpolation function from the previous list based upon the type of the end value `b`, using the following algorithm:

- If `b` is a color, `interpolateRgb` is used
- If `b` is a string, `interpolateString` is used
- If `b` is an array, `interpolateArray` is used
- If `b` is an object and not coercible to a number, `interpolateObject` is used
- Otherwise, `interpolateNumber` is used

For a demonstration of `d3.interpolate()` and some of the underlying smarts, open the following example:



This example uses the `.styleTween()` function to change the font property of the style for the piece of text, increasing the size of the font from 12 px to 36 px over five seconds.

Watch my size change → **Watch my size change**

```
svg.append("text")
    .attr({ x: 10, y: 50 })
    .text('Watch my size change')
    .transition()
    .duration(5000)
    .styleTween('font', function() {
        return d3.interpolate('12px Helvetica', '36px Helvetica');
    });
}
```

The `.styleTween()` function operates in a way similar to `.tween()` except that the first parameter specifies the name of the property that will be set to the value which is returned by the interpolation function provided by the factory method. There is also a `.attrTween()` function that does the same but on an attribute instead of a style.

The function `d3.interpolate()` is smart enough to determine that it should use `d3.interpolateString()`, and to identify that the two strings represent a font size and name besides performing the appropriate interpolation.

Timers

D3.js manages transitions using timers that internally schedule the code to be run at a specific time. These timers are also exposed for your use.

A timer can be created using `d3.timer(yourFunction, [delay], [mark])`, which takes a function to be called, a delay, and a starting time. This starting time is referred to as the **mark**, and it has a default value of `Date.now`.

D3.js timers are not executed at regular intervals – they are not periodic timers. Timers start execution at the time specified by `mark + delay`. The function will then be called as frequently as possible by D3.js, until the function it calls returns `true`.

The use of `mark` and `delay` can allow very specific declaration of time for starting execution. As an example, the following command schedules an event four hours prior to September 1, 2015:

```
d3.timer(notify, -4 * 1000 * 60 * 60, +new Date(2015, 09, 01));
```

To implement a one-shot timer, simply return `true` from the first call of the function.

As a final note on timers, if you want to use a timer to alert you on a regular basis, it is often better to use the JavaScript built-in function `setInterval()`. We will examine using a timer on a periodic basis in the following section.

Adding a fifth dimension to a bubble plot – time

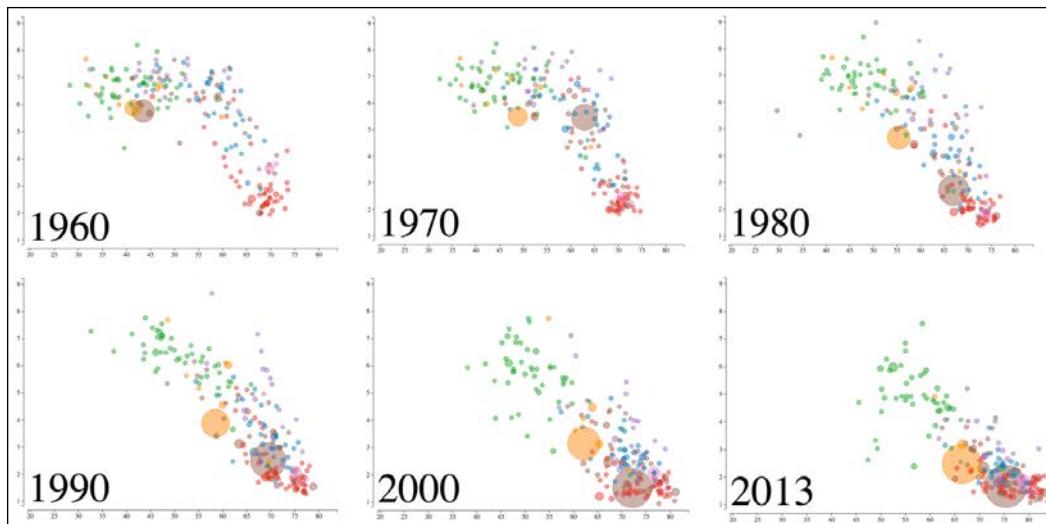
Now let's apply everything we have learned about animation to some real data. We are going to revisit our bubble plot visualization from *Chapter 6, Creating Scatter and Bubble Plots*, expanding the set of data from a single year (2013) to all the available years (1960 through 2013). We will modify the rendering of the visual to periodically update and animate the bubbles into new position and sizes based upon the change in the values of the data.

The expanded data set is available at <https://goo.gl/rC5WS0>. The fundamental difference is the inclusion of a year column, and data covering 54 years.

The code for and the demo of the example is available at the following link:



When you run this, you will see a smooth animation of the data over the years. It is obviously impossible to show this effectively in a static medium such as a book. But for demonstration, I have provided screenshots of the visualization at the start of each decade, except for 2010, which is substituted with the year 2013:



As the years advance, there is a strong tendency for all the countries towards an increased lifespan as well as decrease in fertility. This happens at a different pace for different countries. But it gives you a really good sense that something is going on that is causing this effect. Deciphering the plot is further made easier due to the addition of this extra dimension of time to the bubble plot.

Now let's examine how this is implemented in the example. A good portion of the code is identical to that of the example from *Chapter 6, Creating Scatter and Bubble Plots*, which it is based upon. The loading and cleansing of the data is slightly different due to a different URL and the need to process the Year column in the data:

```
var url = "https://gist.githubusercontent.com/
d3byex/8fcf43e446b1e4dd0146/raw/7a11679cb4a810061dee660be0d30b6a9fe6
9f26/lfp_all.csv";
d3.csv(url, function (error, rawData) {
  var data = rawData.map(function (d) {
    return {
      CountryCode: d.CountryCode,
      CountryName: d.CountryName,
```

```

    LifeExp: +d.LifeExp,
    FertRate: +d.FertRate,
    Population: +d.Population,
    Region: d.Region,
    Year: d.Year
  } ;
}) ;

```

We will be rendering each year of data one at a time. As a part of this, we will need to extract only the data for each specific year. There are a number of ways that we can go about this. D3.js provides a very powerful function to do this for us: `d3.nest()`. This function pivots the `Year` column into the index of an associative array:

```

var nested = d3.nest()
  .key(function (d) { return d.Year; })
  .sortKeys(d3.ascending)
  .map(data);

```

We can then access all the data for a particular year using array semantics such as `nested[1975]`, which will give us the data (only the rows) for just 1975.



For more info on `d3.nest()`, see <https://github.com/mbostock/d3/wiki/Arrays#nest>.

The code is then identical through the creation of the axes. The next new piece of code is to add a text label on the graph to show the year that the data represents. This is positioned in the lower-left corner of the area where the bubbles will be rendered:

```

var yearLabel = svg.append('g')
  .append('text')
  .attr('transform', 'translate(40, 450)')
  .attr('font-size', '75');

```

Then a group is created to contain the bubbles. The rendering function will select this group each time it is called:

```
var bubblesHolder = svg.append('g');
```

This marks the beginning of the code that renders and animates the bubbles. It starts by declaring the interval for which each year should be drawn (10 times per second):

```
var interval = 100;
```

Since the bubbles must be repeatedly rendered, we create a function that can be called to render the bubbles for just a specified year:

```
function render(year) {
    var dataForYear = nested[year];

    var bubbles = bubblesHolder
        .selectAll("circle")
        .data(dataForYear, function (datum) {
            return datum.CountryCode;
        });

    bubbles.enter()
        .append("circle")
        .each(setItemAttributes);

    bubbles
        .transition()
        .duration(interval)
        .each(setItemAttributes);

    bubbles.exit().remove();

    yearLabel.text(year);
}
```

This function first extracts the rows for the specific year, and then binds the data to the circles in the bubblesHolder group. The call to `.data()` also specifies that `CountryCode` will be used as the key. This is very important, because as we move from year to year, D3.js will use this to map the existing bubbles to the new data, making decisions based on this the key on which to enter-update-exit the circles.

The next statement executes the enter function creating new circles and calling a function to set the various attributes of the circles:

```
function setItemAttributes(d) {
    d3.select(this).attr({
        cx: xScale(d.LifeExp),
        cy: yScale(d.FertRate),
        r: popScale(d.Population),
        style: "fill:" + regionColorMap(d.Region) + ";" +
               "fill-opacity:0.5;" +
               "stroke:" + regionColorMap(d.Region) + ";"
    });
}
```

We use a function, as this is also used by the code to update. Finally, there is a case where occasionally a country disappears from the data, so we will remove any bubbles in the scenario.

The final thing we need to do is perform the time animation. This is done by iterating through each year at the specified interval. To do this, we need to know the start and ending year, which we can obtain with the following:

```
var minYear = d3.min(data, function (d) { return d.Year; });
var maxYear = d3.max(data, function (d) { return d.Year; });
```

This follows with setting a variable for the current year and rendering that year:

```
var currentYear = minYear;
render(currentYear);
```

Now we create a function to be called by a timer. This function returns another function which increments the year, and if the year is less than the max year, calls render again, and then schedules another timer instance to run at an interval of milliseconds. This pattern effectively uses a series of D3.js timers for implementing the periodic timer:

```
var callback = function () {
    return function () {
        currentYear++;
        console.log(currentYear);
        if (currentYear <= maxYear) {
            render(currentYear);
            d3.timer(callback(), interval);
        }
        return true;
    }
}
```



Note that this code returns `true` every time it is called. This makes it a one-shot timer. But before returning `true`, if we need to render another year, we start another timer.

The last thing to be done is to start the timer for the first time:

```
d3.timer(callback(), interval);
```

Summary

In this chapter, you learned the fundamentals of animation in D3.js, and by the end of the chapter, applied these simple concepts to make what appears to be a very complex data visualization.

We started with examples of transitions, using them to animate attributes from one state to another across an interval of time, and chaining animations together. Next we looked at handling animation without keyframes using tweening. We also took a quick look at interpolation.

We finished by examining timers, and then applied all the concepts of the chapter to progressively render a large set of data, giving the viewer of the visualization a sense of how data changes by animating time.

In the next chapter, we will examine changing the visuals when the user interacts with the application, learning concepts such as the dragging and filtering of data based upon interactive events.

8

Adding User Interactivity

Great visualizations provide more than a pretty picture and animations; they allow the user to interact with the data, giving them the ability to play with the data to discover the meaning in the data that may not be obvious through a given static presentation.

Exceptional interactions allow the users to steer their way through large amounts of information. It allows them to pan through data too large for a single display, to dive into summary information, and also zoom out to get a higher level view—in essence, it allows users to see the forest from the trees.

Also of great value is the capability to allow the user to easily select, reorder, and reposition visual elements. Through these actions, the user is able to see details of a datum simply by mouseover or touch, to rearrange items for exposing other insights, and to also see how data moves around when reordered. This provides the user a sense of constancy and shows how the data changes when asked to reshuffle on demand.

In this chapter, we will examine a number of techniques for adding interactivity to your D3.js visualizations. We will examine concepts involved in using the mouse to highlight information and provide contextual information, to pan and zoom your visualizations, and to use brushing to select and zoom the view of information in and out.

Specifically, in this chapter we will learn the following topics:

- Hooking into mouse events on D3.js visuals
- Clicking and responding to mouse events
- Building several models of visual animation to provide feedback on interaction
- Handling mouse hovers to provide detailed information on specific visuals
- Creating fluid animations that respond to mouse events
- Brushing and its use in selecting data
- Implementing a context-focus pattern of interaction for viewing stock data

Handling mouse events

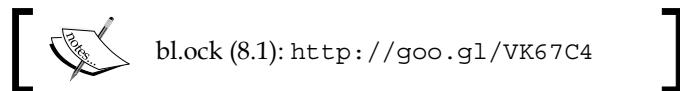
The mouse is the most common device available to users for interacting with D3.js visualizations. Touch is commonly used in case of tablets, and in many cases, touch events can be mapped to mouse events. In this chapter, we will focus exclusively on the mouse. But most of everything we cover also applies to touch. Touch concepts such as pinching can also be easily supported on touch devices with D3.js.

To work with mouse events in D3.js, we attach event listeners to the SVG elements for which we desire to handle the events. The handlers are added using the `.on()` function, which takes as parameters the name of the event and a function to call when the mouse event happens.

We will examine the handling of four mouse events: `mousemove`, `mouseenter`, `mouseout`, and `click`.

Tracking the mouse position using `mousemove`

The movement of the mouse on an SVG visual is reported to your code by listening for the `mousemove` event. This following example demonstrates tracking and reporting the mouse position:



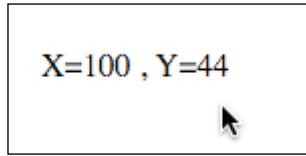
```

var svg = d3.select('body')
.append('svg')
.attr({
    width: 450,
    height: 450
});
var label = svg.append('text')
.attr('x', 10)
.attr('y', 30);

svg.on('mousemove', function () {
    var position = d3.mouse(svg.node());
    label.text('X=' + position[0] + ' , Y=' + position[1]);
});

```

We listen to `mousemove` events using `.on()`, passing it when the event fires, and the example updates the content of the text in the SVG text element:



The position of the mouse is not passed to the function as parameters. To get the actual mouse position, we need to call the `d3.mouse()` function, passing it to the return value of `svg.node()`. This function then calculates the X and Y mouse position relative to the SVG element for which the mouse is moving over.

Capturing the mouse entering and exiting an SVG element

The mouse entering and exiting a particular SVG element is captured using the respective `mouseenter` and `mouseout` events. The following example shows this by creating several circles and then changing their color while the mouse is within the area of the circle (also known as **hovering**).



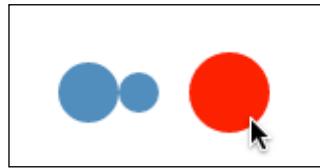
This code creates three circles of varying size (30, 20, and 40 pixel radius):

```
var data = [30, 20, 40],
```

Tracking the enter and exit of the mouse is performed by hooking into those two events:

```
.on('mouseenter', function() {
  d3.select(this).attr('fill', 'red');
})
.on('mouseout', function() {
  d3.select(this).attr('fill', 'steelblue');
});
```

When you run this example, you will be presented by three steelblue circles of slightly varying size, and when you hover the mouse over any of them you will see it change to red:



Note that the SVG element which the mouse is currently entering or exiting is not passed to the functions, so we need to retrieve them using `d3.select(this)`.

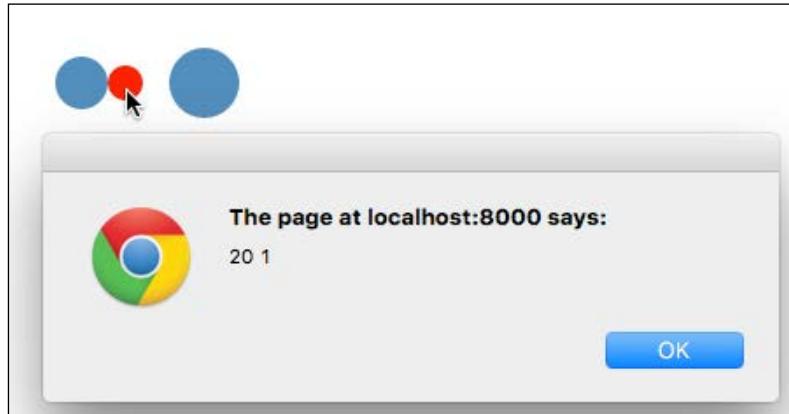
Letting the user know they have clicked the mouse

When the user clicks a button on the mouse, the mouse can track the mouse being clicked by using the `mouseclick` event. The code at the following link demonstrates handling the click event handler:

[ bl.ock (8.3): <http://goo.gl/91rt4S>]

This code adds an event handler to the code in example 8.2 to capture the click event and pop up an alert box that shows the value of the datum and its position in the collection:

```
.on('click', function(d, i) {
  alert(d + ' ' + i);
});
```



This is pretty neat as you are given the data underlying the visual that you clicked. There is no need to retain a map of the visuals to the data to just look this up.

Using behaviors to drag, pan, and zoom

Mouse events often need to be combined to create more complex interactions such as drag, pan, and zoom. Normally, this requires a good quantity of code to track sequences of the `mouseenter`, `mousemove`, and `mouseleave` events.

D3.js provides us with a better way of implementing these interactions through the use of **behaviors**. These behaviors are a complex set of DOM/SVG interactions through D3.js itself handling the mouse events. In a sense, behaviors function similarly to gesture recognizers on mobile platforms.

D3.js currently provides two built-in behaviors:

- **Drag:** This tracks mouse or multi-touch movements relative to an origin
- **Zoom:** This emits zoom and pan events in response to dragging or pinching

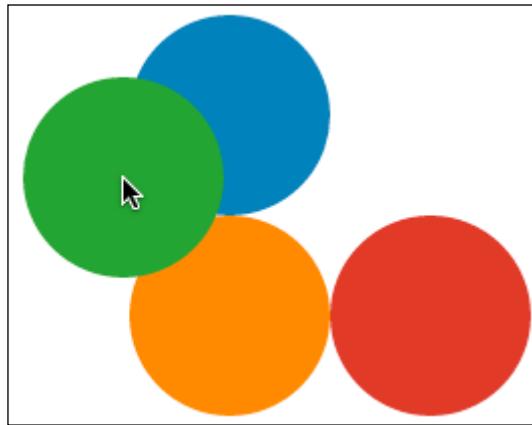
Let's examine an example of implementing drag and another that also adds pan and zoom capabilities.

Drag

Drag is a common behavior in interactive visualization that allows the movement of visual elements by the user via the mouse or touch. The following example demonstrates using the drag behavior:



The preceding example renders four circles and lets you move them around the SVG area using the mouse, but also constrains the movement so that the circles remain completely within the SVG element's visual area:



The code begins by calculating the positions for the circles and rendering them using a selection. The drag behavior is then implemented with the following code:

```
var dragBehavior = d3.behavior.drag()  
    .on('drag', onDrag);  
circles.call(dragBehavior);  
  
function onDrag(d) {  
  var x = d3.event.x,  
      y = d3.event.y;  
  if ((x >= radius) && (x <= width - radius) &&  
      (y >= radius) && (y <= height - radius)) {  
    d3.select(this)
```

```

        .attr('transform', function () {
            return 'translate(' + x + ', ' + y + ')';
        });
    }
}

```

The behavior is created using `d3.behavior.drag()`. This object then requires us to tell it that we are interested in listening to drag events. You can also specify handlers for `dragstart` and `dragended` events to identify the start and completion of a drag behavior.

Next, we need to inform D3.js to hook up the behavior to SVG elements. This is done by using the `.call()` function on the selection. As we saw when rendering axes, the function we specified will be called by D3.js during the rendering of each selected item. In this case, this will be our drag behavior, and hence, the implementation of this function can perform all the event processing needed for dragging an SVG element on our behalf.

Our event handler for the drag behavior is then called whenever the user drags an associated SVG element. This function first retrieves the new `x` and `y` position for the item being dragged from the `d3.event` object. These values are computed and set by D3.js prior to this function being called.

All that is required at this point is to set a new transform for the respective SVG element to move it into the new position. This example also checks that the circle is still completely within the SVG element and only sets the new position if that is `true`.

Pan and zoom

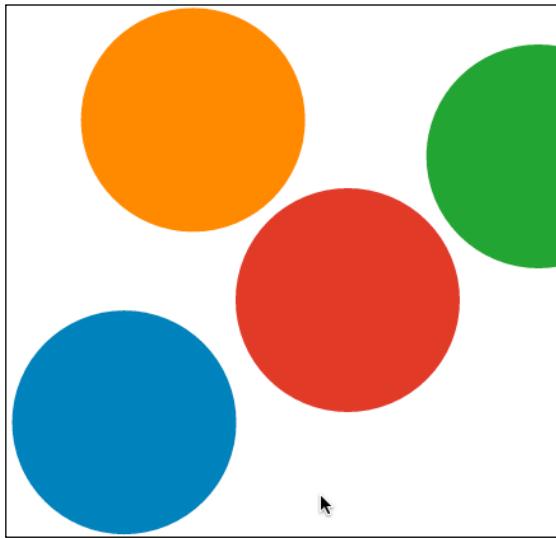
Panning and zooming are two common techniques in data visualization. Panning allows the user to drag the entire visual around the screen. This exposes visuals that would otherwise be rendered outside of the bounds of the visual area. A common scenario for panning is to move a map around to expose areas previously out of view.

Zooming allows you to scale up or down the perceived distance of the user from the visual. This can be used to make small items bigger or to zoom out to see items that were too big or out of the extent of the visual display.

Both panning and zooming are implemented by the same D3.js behavior, `d3.behavior.zoom()`. The following example demonstrates its use:



When running this example, you can not only drag the circles, but you can drag the background to move all the circles at once (the pan) and use your mouse wheel to zoom in and out:



There are a few small changes to the previous example for adding these additional features. These start with the declaration of the zoom behavior:

```
var zoomBehavior = d3.behavior.zoom()  
    .scaleExtent([0.1, 10])  
    .on('zoom', onZoom);
```

The initial zoom level is 1.0. The call to `.scaleExtent()` informs the behavior that it should zoom down to 0.1, one-tenth of the original size, and up to 10, or 10x of the original. Moreover, the behavior should call the `onZoom()` function when zoom events occur.

Now we create the main SVG element and attach the zoom behavior to it using `.call()`:

```
var svg = d3.select('body')  
    .append('svg')  
    .attr({  
        width: width,  
        height: height  
    })  
    .call(zoomBehavior)  
    .append('g');
```

The code also appends a group element to the SVG element and the `svg` variable then refers to this group. The pan and zoom events are routed by the top level SVG element to our handler, which then sets the translate and scale factor on this group, therefore creating the effects on the circles.

Now we just need to implement the `zoomIt` function:

```
function onZoom() {
    svg.attr('transform', 'translate(' + d3.event.translate +
        ')' + 'scale(' + d3.event.scale + ')');
}
```

Just before the behavior calls this function, it sets the `d3.event.translate` variable to represent the extent of translation that should occur on the entire visual.

The `d3.event.scale` variable is also set by D3.js to represent the appropriate level of zoom. In this example, this ranges from 0.1 to 10.

Another small change is in the way the drag behavior is declared.

```
var dragBehavior = d3.behavior.drag()
    .on("drag", onDrag)
    .on("dragstart", function() {
        d3.event.sourceEvent.stopPropagation();
    });

```

This is done because there will be an issue with the example if this is not modified in the preceding manner. If left as-is, the pan and zoom behavior and the drag behavior will conflict with each other. When dragging a circle, the `svg` element will also pan when it should stay in place.

By handling the `dragstart` event and calling `d3.event.sourceEvent.stopPropagation()`, we prevent this mouse event on a circle from **bubbling up** to the `svg` element and starting a pan. Problem solved!

Enhancing a bar graph with interactivity

Now let's apply what we have learned about mouse event handling to create an interactive bar graph. Mouse events on a bar chart can provide useful contextual information to the person interacting with the graph.

Adding User Interactivity

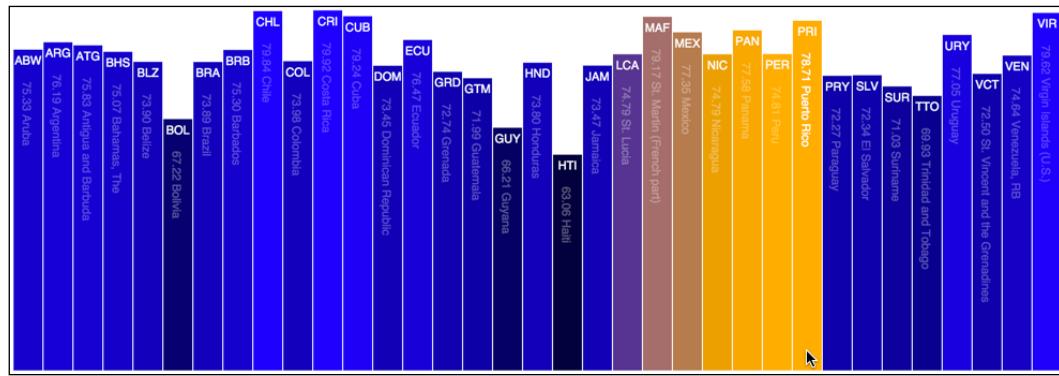
The data for the examples will use a stripped-down version of the life expectancy vs fertility dataset that was used in earlier chapters. This dataset will use the data for the Latin American and Caribbean economic regions only, which contain roughly 35 countries, for the year 2013.

The bars in the examples will represent the longevity, will be annotated at the top with the country code, and have vertically oriented text representing the actual longevity value and the full country name. The examples will omit axes and margins to keep things simple.

The code and live example for this example is available at the following location:



This interaction pattern can be used to visually accentuate a particular bar in a bar chart when the mouse is moved over it. We have seen this when using `mouseenter` and `mouseout` events earlier as applied to circles. Here, we will use it to highlight the bar:



The rectangles representing the bars are created with the following code:

```
svg.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr({
    width: barWidth,
    height: 0,
    y: height
  })
```

After the creation of the bars, the code hooks up the `mouseover` and `mouseout` events. The `mouseover` event makes the vertical text completely opaque and sets the bar color to orange:

```
.on('mouseover', function (d) {
  d3.select('text.vert#' + d.CountryCode)
    .style('opacity', maxOpacity);
  d3.select(this).attr('fill', 'orange');
})
```

The `mouseout` event animates and sets the text opacity back to the original value and starts an animation to set the color back to its original shade. This animation gives the appearance of mouse trails when moving across the bars:

```
.on('mouseout', function (d) {
  d3.select('text.vert#' + d.CountryCode)
    .style('opacity', minOpacity);
  d3.select(this)
    .transition()
    .duration(returnToColorDuration)
    .attr('fill', 'rgb(0, 0, ' +
      Math.floor(colorScale(d.LifeExp)) + ')');
```

The last portion of the selection creating the bars performs an animation to make the bars grow and transition from black to their eventual colors which the graph loads:

```
.transition()
.duration(barGrowDuration)
.attr({
  height: function (d) { return yScale(d.LifeExp); },
  x: function (d, i) { return xScale(i); },
  y: function (d) {
    return height - yScale(d.LifeExp);
  },
  fill: function (d) {
    return 'rgb(0, 0, ' +
      Math.floor(colorScale(d.LifeExp)) + ')';
  }
});
```

To also enhance the presentation of the underlying information, we will place two pieces of data on each bar: the country code as horizontal text at the top and a piece of vertical text which shows the actual value of the datum and the full name of the country. The following code creates the horizontal text:

```
svg.selectAll('text')
  .data(data)
  .enter()
  .append('text')
  .text(function (d) { return d.CountryCode; })
  .attr({
    x: function (d, i) { return xScale(i) + barWidth / 2; },
    y: height,
    fill: 'white',
    'text-anchor': 'middle',
    'font-family': 'sans-serif',
    'font-size': '11px'
  })
  .transition()
  .duration(barGrowDuration)
  .attr('y', function (d) {
    return height - yScale(d.LifeExp) +
      horzTextOffsetY; });

```

The vertical text is created by the following code:

```
svg.selectAll('text.vert')
  .data(data)
  .enter()
  .append('text')
  .text(function (d) { return d.LifeExp.toFixed(2) + ' ' +
    d.CountryName; })
  .attr({
    id: function (d) { return d.CountryCode; },
    opacity: minOpacity,
    transform: function (d, i) {
      var x = xScale(i) + halfBarWidth -
        verticalTextOffsetX;
      var y = height - yScale(d.LifeExp) +
        verticalTextOffsetY;
      return 'translate(' + x + ',' + y + ')rotate(90)';
    },
    'class': 'vert',
  });

```

```
'font-family': 'sans-serif',
'font-size': 11,
'fill': 'white'
});
```

Highlighting selected items using brushes

A **brush** in D3.js provides the ability for the user to interact with your visualization by allowing the selection of one or more visual elements (and the underlying data items) using the mouse.

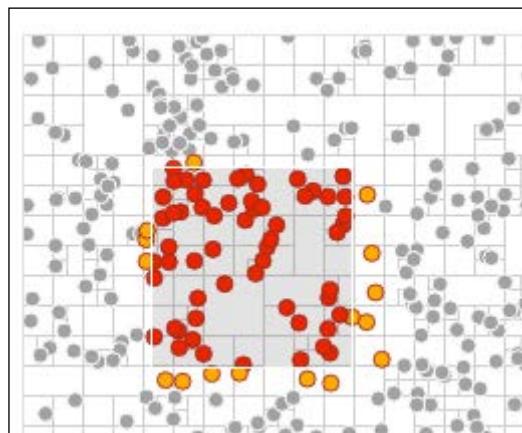
This is a very important concept in exploratory data analysis and visualization, as it allows users to easily drill in and out of data or select specific data items for further analysis.

Brushing in D3.js is very flexible, and how you implement it depends upon the type of visualization you are presenting to the user. We will look at several examples or brushes and then implement a real example that lets us use a brush to examine stock data.

Online examples of brushes

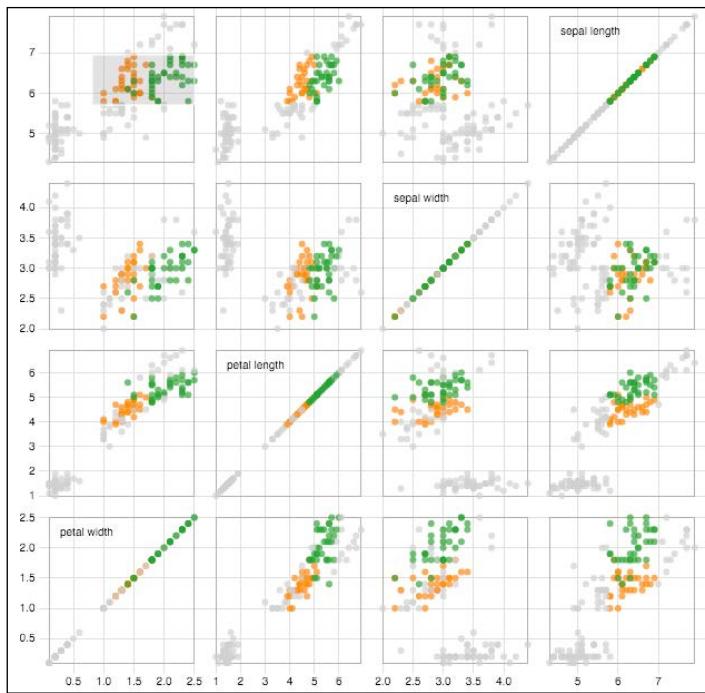
To understand brushes, let's first take a look at several brush examples on the Internet. These are all examples available on the web that you can go and play with.

The following brush shows the use of rectangular selection for selecting data that is within the brush (<http://bl.ocks.org/mbostock/4343214>):

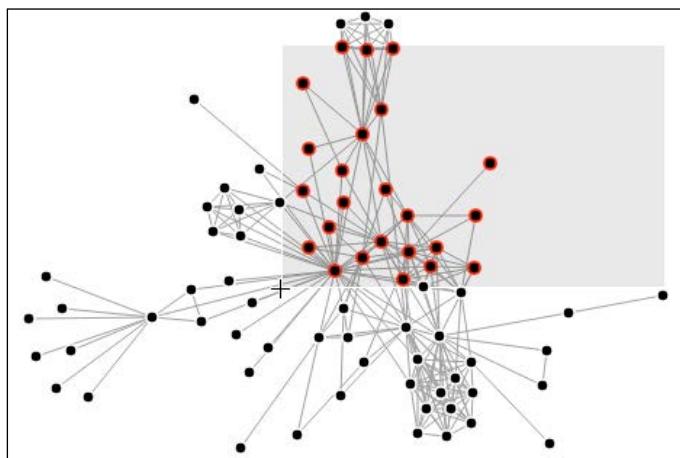


Adding User Interactivity

Another example of this brushing is the scatterplot matrix brush at <http://bl.ocks.org/mbostock/4063663>. This example is notable for the way in which you can select points on any one of the scatter plots. The app then selects the points on all the other plots so that the data is highlighted on those too:



The following example demonstrates using a brush to select a point within a force-directed network visualization (<http://bl.ocks.org/mbostock/4565798>):



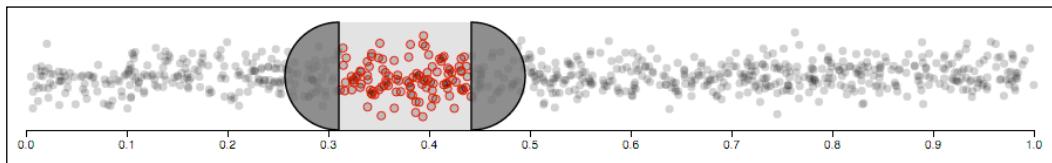


We will learn about force-directed network visualizations in greater detail in *Chapter 11, Visualizing Information Networks*.

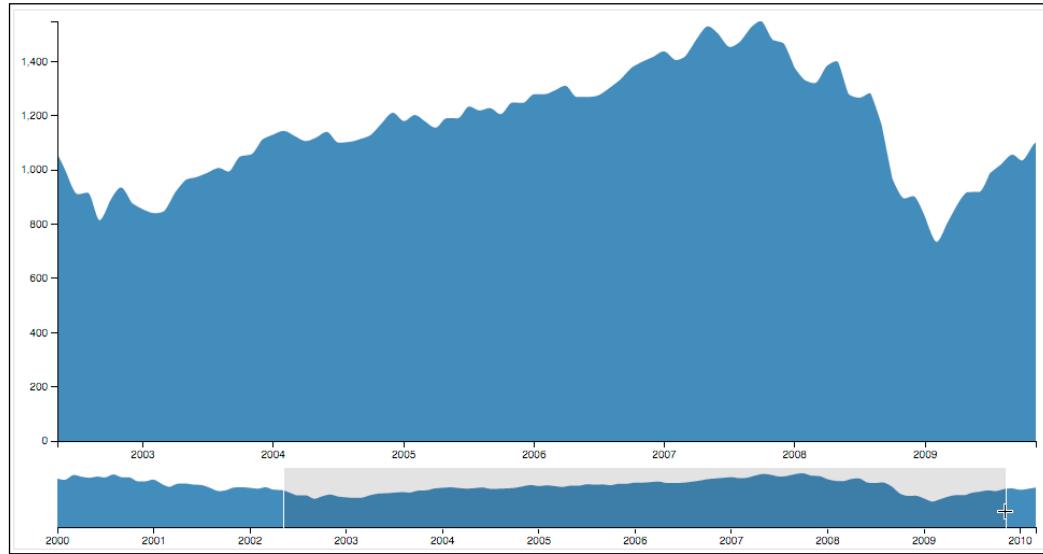
The creation of custom brush handles is a common scenario you will see when using brushes. Handles provide you a means of providing a custom rendering of the edges of the brush to provide a visual cue to the user.

As an example of a custom brush, the following creates semicircles as the handles:
<http://bl.ocks.org/mbostock/4349545>.

You can resize the brush by dragging either handle and reposition it by dragging the area between the handles:



The last example of a brush (before we create our own) is the following, which demonstrates a concept referred to as `focus + context`:



In this example, the brush is drawn atop the smaller graph (the context). The context graph is static in nature, showing a summary of the entire range of data. As the brush is changed upon the context, the larger graph (the focus) animates in real-time while the brush is changed.

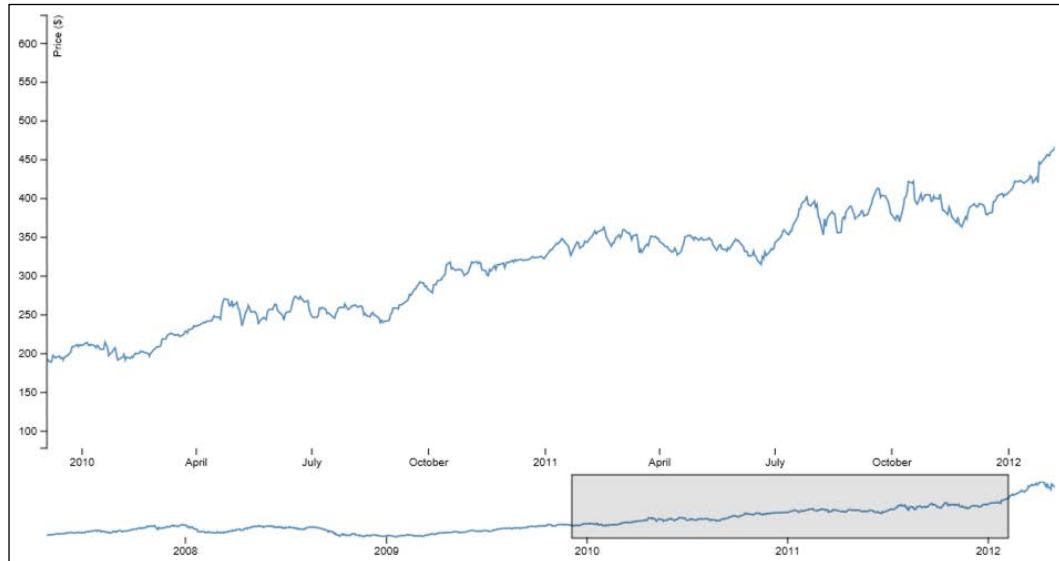
In the next section, we will examine creating a similar version of this graph which utilizes financial data, a common domain for this type of interactive visualization.

Implementing focus + context

Now let's examine how to implement **focus + context**. The following example that we will use will apply this concept to a series of stock data:



The resulting graph will look like the following graph:



The top graph is the focus of the chart and represents the detail of the stock data that we are examining. The bottom graph is the context and is always a plot of the full series of data. In this example, we focus on data from just before the start of 2010 until just after the start of 2012.

The context area supports brushing. You can create a new brush by clicking on the context graph and dragging the mouse to select the extents of the brush. The brush can then be slid back and forth by dragging it, and it can be resized on the left and right by dragging either boundary. The focus area will always display the details of the area selected by the context.

To create this visualization, we will be drawing two different graphs, and hence, we need to layout the vertical areas for each and create the main SVG element with a size enough to hold both:

```
var width = 960, height = 600;

var margins = { top: 10, left: 50, right: 50,
               bottom: 50, between: 50 };

var bottomGraphHeight = 50;
var topGraphHeight = height - (margins.top + margins.bottom +
                                margins.between + bottomGraphHeight);
var graphWidths = width - margins.left - margins.right;
```

This example will also require the creation of a clipping area. As the line drawn in the focus area scales, it may be drawn overlapping on the left with the *y* axis. The clipping area prevents the line from flowing off to the left over the axis:

```
svg.append('defs')
    .append('clipPath')
    .attr('id', 'clip')
    .append('rect')
    .attr('width', width)
    .attr('height', height);
```

This clipping rectangle is referred to in the styling for the lines. When the lines are drawn, they will be clipped to this boundary. We will see how this is specified when examining the function to style the lines.

Now we add two groups that will hold the renderings for both the focus and context graphs:

```
var focus = svg
    .append('g')
    .attr('transform', 'translate(' + margins.left + ',' +
                           margins.top + ')');
```

Adding User Interactivity

```
var context = svg.append('g')
  .attr('class', 'context')
  .attr('transform', 'translate(' + margins.left + ',' +
    (margins.top + topGraphHeight + margins.between) + ')');
```

This visual requires one *y* axis, two *x* axes, and the appropriate scales for each. These are created with the following code:

```
var xScaleTop = d3.time.scale().range([0, graphWidths]),
  xScaleBottom = d3.time.scale().range([0, graphWidths]),
  yScaleTop = d3.scale.linear().range([topGraphHeight, 0]),
  yScaleBottom = d3.scale.linear()
    .range([bottomGraphHeight, 0]);

var xAxisTop = d3.svg.axis().scale(xScaleTop)
  .orient('bottom'),
xAxisBottom = d3.svg.axis().scale(xScaleBottom)
  .orient('bottom');
var yAxisTop = d3.svg.axis().scale(yScaleTop)
  .orient('left');
```

We will be drawing two lines, so we create the two line generators, one for each of the lines:

```
var lineTop = d3.svg.line()
  .x(function (d) { return xScaleTop(d.date); })
  .y(function (d) { return yScaleTop(d.close); });

var lineBottom = d3.svg.line()
  .x(function (d) { return xScaleBottom(d.date); })
  .y(function (d) { return yScaleBottom(d.close); });
```

The last thing we need to do before loading the data and actually rendering it is to create our brush using `d3.svg.brush()`:

```
var brush = d3.svg.brush()
  .x(xScaleBottom)
  .on('brush', function brushed() {
    xScaleTop.domain(brush.empty() ? xScaleBottom.domain() :
      brush.extent());
    focus.select('.x.axis').call(xAxisTop);
  });

```

This preceding snippet informs the brush that we want to brush along the *x* values using the scale defined in `xScaleBottom`. Brushes are event-driven and will handle the `brush` event, which is raised every time the brush is moved or resized.

And finally, the last major thing the code does is load the data and establish the initial visuals. You've seen this code before, so we won't explain it step by step. In short, it consists of loading the data, setting the domains on the scales, and adding and drawing the axes and lines:

```
d3.tsv('https://gist.githubusercontent.com/d3byex/b6b753b6ef178fdb06a2/raw/0c13e82b6b59c3ba195d7f47c33e3fe00cc3f56f/aapl.tsv', function (error, data) {
  data.forEach(function (d) {
    d.date = d3.time.format('%d-%b-%y').parse(d.date);
    d.close = +d.close;
  });

  xScaleTop.domain(d3.extent(data, function (d) {
    return d.date;
 )));
  yScaleTop.domain(d3.extent(data, function (d) {
    return d.close;
 )));
  xScaleBottom.domain(d3.extent(data, function (d) {
    return d.date;
 )));
  yScaleBottom.domain(d3.extent(data, function (d) {
    return d.close;
 )));

  var topXAxisNodes = focus.append('g')
    .attr('class', 'x axis')
    .attr('transform', 'translate(' + 0 + ', ' +
      (margins.top + topGraphHeight) + ')')
    .call(xAxisTop);
  styleAxisNodes(topXAxisNodes, 0);

  focus.append('path')
    .datum(data)
    .attr('class', 'line')
    .attr('d', lineTop);

  var topYAxisNodes = focus.append('g')
    .call(yAxisTop);
  styleAxisNodes(topYAxisNodes);

  context.append('path')
    .datum(data)
    .attr('class', 'line')
    .attr('d', lineBottom);
```

```
var bottomXAxisNodes = context.append('g')
    .attr('transform', 'translate(0,' +
           bottomGraphHeight + ')')
    .call(xAxisBottom);
styleAxisNodes(bottomXAxisNodes, 0);

context.append('g')
    .attr('class', 'x brush')
    .call(brush)
    .selectAll('rect')
    .attr('y', -6)
    .attr('height', bottomGraphHeight + 7);

context.selectAll('.extent')
    .attr({
        stroke: '#000',
        'fill-opacity': 0.125,
        'shape-rendering': 'crispEdges'
    });

styleLines(svg);
});
```

Congratulations! You have stepped through creating a fairly complicated interactive display of stock data. But the beauty is that through the underlying capabilities of D3.js, it was comprised of a relatively small set of simple steps that result in **Beautiful Data**.

Summary

In this chapter, you learned how to use mouse events provided by D3.js to create interactive visualization. We started by explaining how to hook up mouse events and respond to them, changing the visualization as the events occurred. Then we examined behaviors and how we can use them to implement drag, pan, and zoom, which allow the user to move around data, take a closer look, as well as zoom in and out. Finally, we covered brushing and how it can be used to select multiple visuals/ data items, ending with a slick example of applying **focus + context** to visualize financial data.

In the next chapter on layouts, we will move a little higher up the visual stack of D3.js to examine layouts, which are essentially generators for complex data visualizations.

9

Complex Shapes Using Paths

In *Chapter 3, Creating Visuals with SVG*, we briefly examined the concept of paths. We saw that we could use paths and their associated mini-language to create multi-segment renderings by creating a sequence of commands. These paths, although very powerful, can be cumbersome to create manually.

But don't fret, as D3.js provides a number of objects to create complex paths using just a few JavaScript statements. These **path generators** take much of the pain out of creating complex paths manually, as they do the heavy lifting of assembling the sequence of commands automatically.

Additionally, an important type of graph we have not looked at in this book is a line graph. This has been purposefully pushed off until now, as it is the most commonly used to create lines using path generators. After the examples in this chapter, the power of the path to create lines will be evident.

In this chapter, we will cover the following topics:

- An overview of path data generators
- Lines and area generators
- Arcs and pie generators
- Symbols generators
- Diagonals and radial generators
- Line interpolators

An overview of path data generators

D3.js goes to great lengths to make using SVG easy, particularly when creating complex paths. To do this, D3 provides a number of helper functions referred to as path generators that have been created to handle the gory details of path generation from a set of data.

The generators we examine will follow a common pattern of usage, so once you learn to use one, the use of the others will come naturally. These steps include:

1. Creating the generator object.
2. Specifying accessor functions that can be used to find the X and Y values.
3. Calling any additional methods to specify various rendering instructions.
4. Adding a path to the visual.
5. Specifying the data using `.datum()` on that path.
6. And finally, setting the `d` attribute of the path to the generator, which tells the path where to find the generator object for the path.

Once these are completed and D3.js renders the visual, it uses the generator that is attached to a `d` attribute and creates path commands based on your data. This is also why we use `.datum()` instead of `.data()`, as datum assigns the data to just that one element and does not force the execution of an enter-update-exit loop on that data.

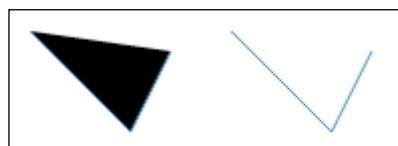
Now let's examine doing this with various common generators – this will be fun!

Creating a sequence of lines

The line generator creates the necessary commands to draw a sequence of lines that are connected to each other:



The preceding example creates a single line path generator and renders it twice, resulting in the following graphics:



The path generator is created with the following data and using a `d3.svg.line()` object. On that object, we call two functions, `x()` and `y()`, which we give a function that tells the generator how to locate the `X` and `Y` values for each datum:

```
var data = [
  { X: 10, Y: 10 },
  { X: 60, Y: 60 },
  { X: 80, Y: 20 }
];
var generator = d3.svg.line()
  .x(function(d) { return d.X; })
  .y(function(d) { return d.Y; });
```

The next step is to add a path, call its `.datum()` function passing the data, and setting the `d` attribute at a minimum to specify which generator to use. The example creates two paths that use the same data and generator but apply a different fill:

```
svg.append('path')
  .datum(data)
  .attr({
    d: generator,
    fill: 'none',
    stroke: 'steelblue'
  });
svg.append('path')
  .datum(data)
  .attr({
    transform: 'translate(100,0)',
    d: generator,
    fill: 'none',
    stroke: 'steelblue'
  });
```

This path specifies two lines. The default operation for a line path is to connect the last point with the first and fill in the internals. In the case of the first path, this is a black fill and results in the black triangle (if you zoom in, you will see the `steelblue` outline on two of the sides). The latter path sets the fill to empty, so the result is just the two lines.

This example also demonstrates using a single path generator but applying a transform and a different style to the actual path.

Examining the generated SVG, we see that D3.js has created the two paths and automatically generated the path data that is assigned to the `d` property of the path:

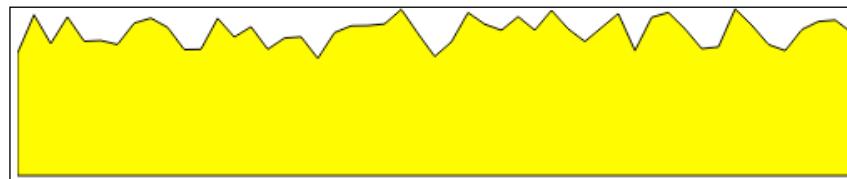
```
▼<svg width="500" height="250">
  <path d="M10,10L60,60L80,20" stroke="steelblue"></path>
  <path transform="translate(100,0)" d="M10,10L60,60L80,20"
    fill="none" stroke="steelblue"></path>
</svg>
```

Areas

An area path generator allows us to make line plots where the area below the line plot is filled in with a particular color. A practical use of these is for the creation of area graphs. The following example demonstrates the creation of an area graph:



This preceding example results in something that looks like the following image. The data is random, so it will be different each time it runs:



The data is generated by generating 100 random numbers between 0 and 30, and defining `y` as the random value and `x` increasing in increments of 10:

```
var data = d3.range(100)
  .map(function(i) {
    return Math.random() * 30;
  })
  .map(function(d, i) {
    return { x: i * 10, y: d }
  });

```

The path is generated using a `d3.svg.area()` object:

```
var generator = d3.svg.area()
  .y0(100)
  .x(function(d) { return d.x; })
  .y1(function (d) { return d.y; });
```

An area path generator requires providing three accessor functions:

- `x()`: This specifies where to get the X values
- `y0()`: This gets the position of the baseline of the area
- `y1()`: This retrieves the height at the given `x()` value

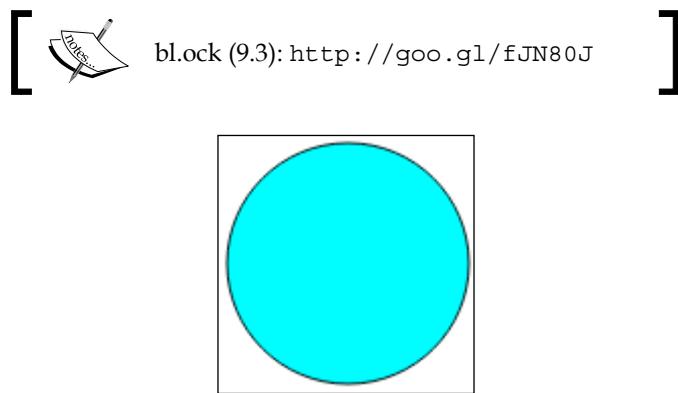
The actual SVG path is then created and styled similar to the previous example.

Creating arcs, donuts, wedges, and segments

An arc is a slice of a circle that has a portion of it swept through two specific angles. An arc swept through a full 360 degrees will actually result in a circle. A sweep of less than 360 degrees gives you a wedge of that circle and is often called a pie **wedge**.

An arc is created using the `d3.svg.arc()` function. This generator takes four parameters, describing the mathematics of the arc. The size of the wedge is defined by using the `.outerRadius()` function and an inner radius that is specified using `.innerRadius()`.

The following example uses an arc to draw a circle:

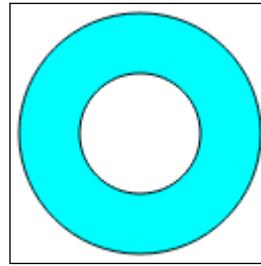
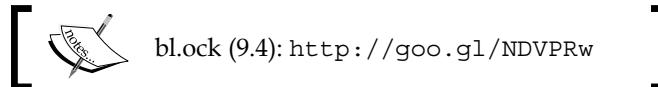


The code to create the generator is the following:

```
var generator = d3.svg.arc()  
    .innerRadius(0)  
    .outerRadius(60)  
    .startAngle(0)  
    .endAngle(Math.PI * 2);
```

The generator specifies an inner radius of 0 and outer radius of 60. The start and end angles are in radians and sweep out an entire circle.

The following example increases the size of the inner radius to create a donut:



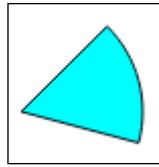
The only difference in the code is the call to `.innerRadius()`:

```
var generator = d3.svg.arc()  
    .innerRadius(30)  
    .outerRadius(60)  
    .startAngle(0)  
    .endAngle(Math.PI * 2);
```

We have now created a donut! Now how about an example of creating a pie wedge? We can create a pie wedge by specifying an inner radius of 0 and setting the start angle and end angle to not sweep out a full 360 degrees, as shown in the following example.

To demonstrate, the following example creates a pie wedge sweeping between 45 and 105 degrees:

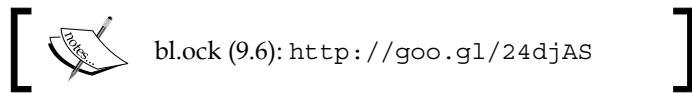




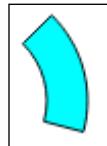
The generator for the preceding pie wedge is as follows:

```
var generator = d3.svg.arc()  
    .innerRadius(0)  
    .outerRadius(60)  
    .startAngle(45 * Math.PI * 2 / 360)  
    .endAngle(105 * Math.PI * 2 / 360);
```

A final example for arcs is to create a segment by increasing the inner radius of the previous example to be greater than 0:



```
var generator = d3.svg.arc()  
    .innerRadius(40)  
    .outerRadius(60)  
    .startAngle(45 * Math.PI * 2/360)  
    .endAngle(105 * Math.PI * 2/360);
```



Creating a pie chart

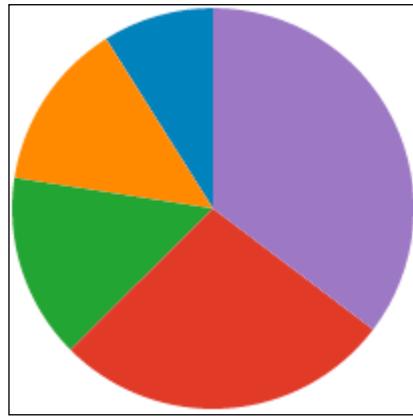
One of the most common type of charts is the pie chart (it is also one of the most reviled). A pie chart could be created by using multiple arc generators and placing them manually.

To make this simpler, D3.js provides us with a tool to help us generate pies and the associated arcs with a generator for pies, `d3.layout.pie()`. From an array of data, this function will generate an array of arc specifications that we can then use to generate all the pie segments automatically.

So, let's examine the creation of a pie:



The preceding code results in the following pie chart:



The example starts by declaring values that represent each piece of the pie and then we pass that to the `d3.layout.pie()` function:

```
var data = [21, 32, 35, 64, 83];
var pieSegments = d3.layout.pie()(data);
```

If you examine the contents of `pieSegments`, you will see a series of objects similar to the following:

```
[{"data": 21,
  "endAngle": 6.283185307179587,
  "padAngle": 0,
  "startAngle": 5.721709173346517,
  "value": 21},
 ...]
```

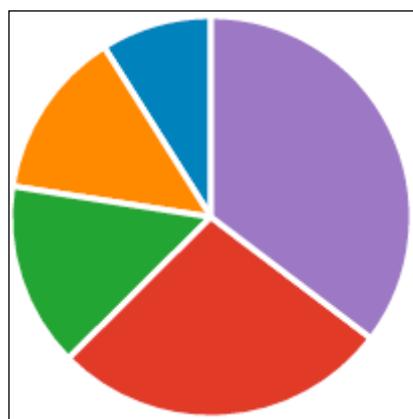
We can use this data via an arc generator and generate the pie:

```
var arcGenerator = d3.svg.arc()
  .innerRadius(0)
  .outerRadius(100)
  .startAngle(function(d) {
    return d.startAngle;
  })
  .endAngle(function(d) {
    return d.endAngle;
  });
var colors = d3.scale.category10();
group.selectAll('path')
  .data(pieSegments)
  .enter()
  .append('path')
  .attr('d', arcGenerator)
  .style('fill', function(d, i) {
    return colors(i);
});
```

Exploding the pie

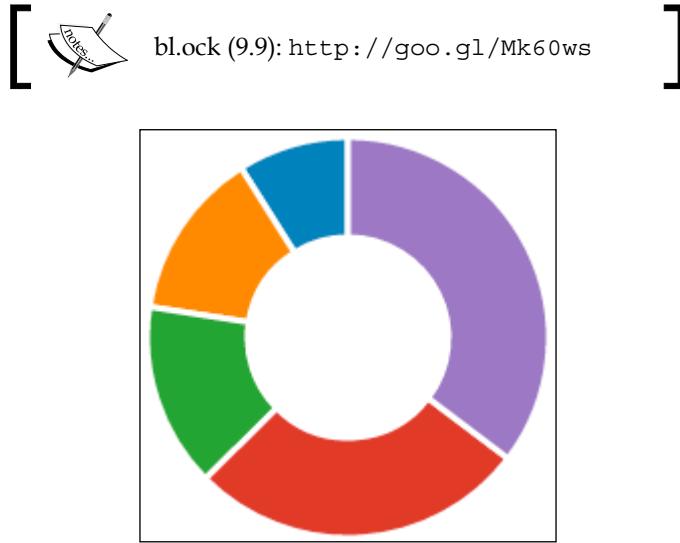
We can make an exploded pie by setting the width of the border of the pie wedges. The following example demonstrates this in action. We'll skip the walkthrough of the code, as it's just adding a `stroke` and `stroke-width` to the previous example:

[ bl.ock (9.8): <http://goo.gl/fhQEau>]



Creating a ring graph

We can also easily make this into a ring chart by increasing the inner radius, as shown in the following example (with a brief modification to the previous example):



Creating symbols

Symbols are little shapes that can be used on a chart, much like how we used small circles and squares in the chapter on scatter plots. D3.js comes with a generator that creates six symbols: circle, cross, diamond, square, triangle-down, and triangle-up.

These symbols are named, and the `d3.svg.symbolTypes` contains an array of the names of the available symbol types. A symbol is then created by passing the symbol name to the `d3.svg.symbol().type()` function, which returns a path generator for the specified symbol.

An example of rendering the available symbols is available at the following link:



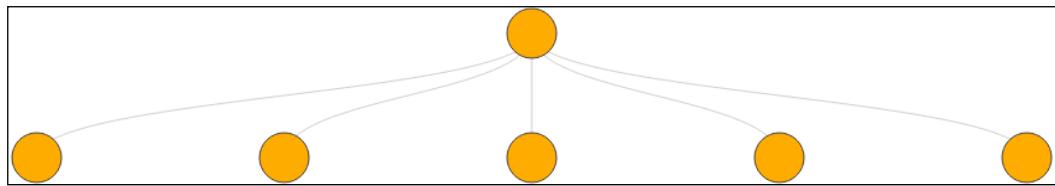
The preceding code renders the following symbols as the result:



Perhaps they're not the most exciting things in the world, but they are useful for representing different data items on scatter plots or as point markers on line diagrams.

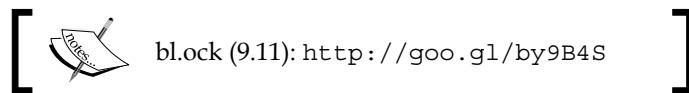
Using diagonals to create curved lines

The diagonal is one of my personal favorites, and it can be used in many complex visualizations. This is a concept which I believe is best understood by seeing an example:



The diagonal generator generates the curved lines between one point and a set of other points, generating the appropriate curves based on the position of the target points.

The following example creates the previous image:



This example starts with defining the source and target positions using JavaScript objects, and then, by creating from those an array of objects representing every combination of source and target:

```
var source = { x: 500, y: 50 };
var targets = [
  { x: 100, y: 150 },
  { x: 300, y: 150 },
  { x: 500, y: 150 },
  { x: 700, y: 150 },
  { x: 900, y: 150 }
];
var links = targets.map(function (target) {
  return { source: source, target: target };
});
```

We can then generate the curved lines using the following selection, which uses a `d3.svg.diagonal()` object as the generator of the path data:

```
svg.selectAll('path')
  .data(links)
  .enter()
  .append('path')
  .attr({
    d: d3.svg.diagonal(),
    fill: 'none',
    stroke: '#ccc'
 });
```

The circles are not rendered by the diagonal generator. The code renders and positions them based on the positions of the source and target points.

Drawing line graphs using interpolators

Now let's examine creating line graphs using a number of the built-in line generators. The capability for rendering lines in D3.js is very robust, and can be used to generate lines with straight segments, or to fit curves through a series of points using a number of different algorithms.

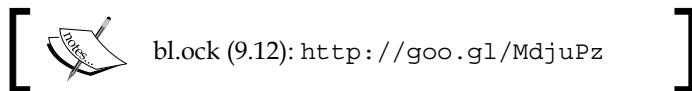
When rendering a line using a line generator, D3.js applies an interpolator across your data to determine how to create the path segments that connect your data points. The following tables lists the available line interpolators that are available:

Interpolator	Operation
<code>linear</code>	Straight lines between points
<code>linear-closed</code>	Closes the line segment, last point to first, making a polygon
<code>step-before</code>	Step-wise drawing vertically then horizontally
<code>step-after</code>	Step-wise drawing horizontally then vertically
<code>basis</code>	Renders a b-spline curve with control points at the ends
<code>basis-open</code>	Renders a b-spline curve with control points at the ends, not closing the loop
<code>basic-closed</code>	Renders a b-spline curve with control points at the ends, closing the loop
<code>bundle</code>	Equivalent to basis, but with a tension parameter
<code>cardinal</code>	A cardinal spline, with control points at the ends

Interpolator	Operation
cardinal-open	A cardinal spline, with control points at the ends; the line may not intersect the end points, but will pass through the internal points
cardinal-closed	Closes the cardinal spline into a loop
monotone	A cubic interpolation that preserves monotonicity in y

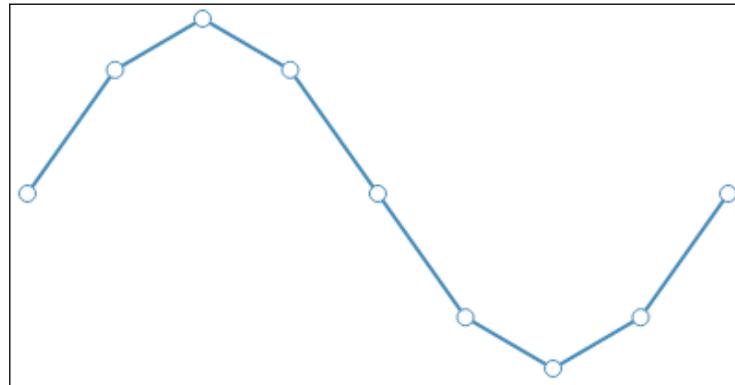
The default is to use a linear interpolator, which essentially draws a straight line between each pair of adjacent points. We will take a look at each of these, as I think they are worth demonstrating (and are fun!).

The example is available at the following link:



The application presents the user with two options. One is to select the type of interpolation and the other is to select a tension value, which is only used when the selected interpolation is bundle.

The example then generates a cycle of a sine wave that is represented by 8 points. As an example, the result is the following when linear interpolation is selected:



The application starts by creating the dropdown boxes and the main SVG element in HTML. It then sets up the scales for the sine wave to map the points into the SVG. The first time the page is loaded, and upon every change of a selection for interpolation or tension, the `redraw()` function is called and the graph is generated.

The `redraw()` function retrieves the current value from the interpolation dropdown and uses it to create the line path generator using the selected value:

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

If the selected interpolation is `bundle`, it also retrieves the selected value for the tension and applies that value to the line path generator:

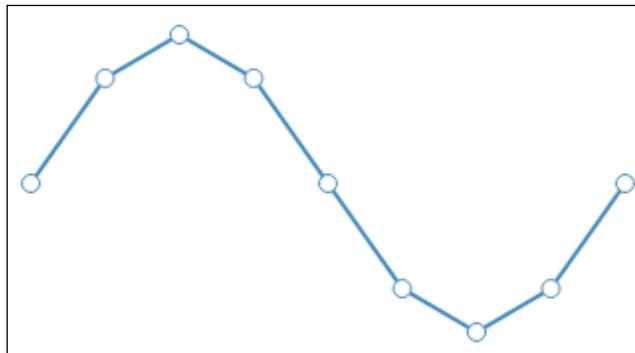
```
if (interpolation === "bundle") {
  var tensionsSel = document.getElementById('tensions');
  var tension = tensionsSel.options[
    tensionsSel.selectedIndex].value;
  line.tension(tension);
}
```

The lines are then generated using a path and the associated generator and then circles are added at the location of each point.

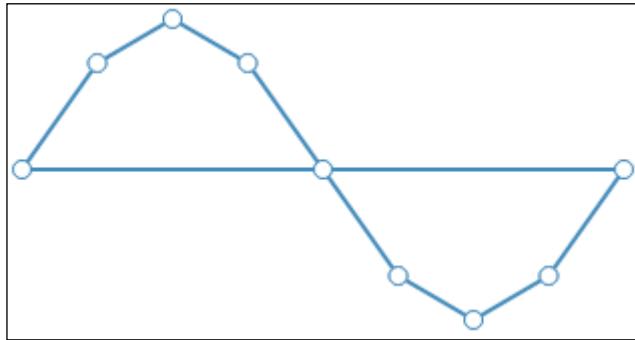
Now let's examine what each of these interpolations does to render our sine wave.

Linear and linear-closed interpolators

The linear interpolator draws straight lines between the points:



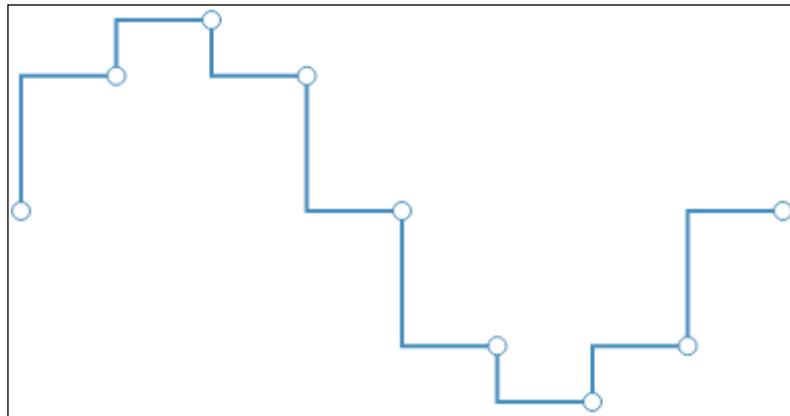
Linear closed is a slight variant that also connects the last point to the first:



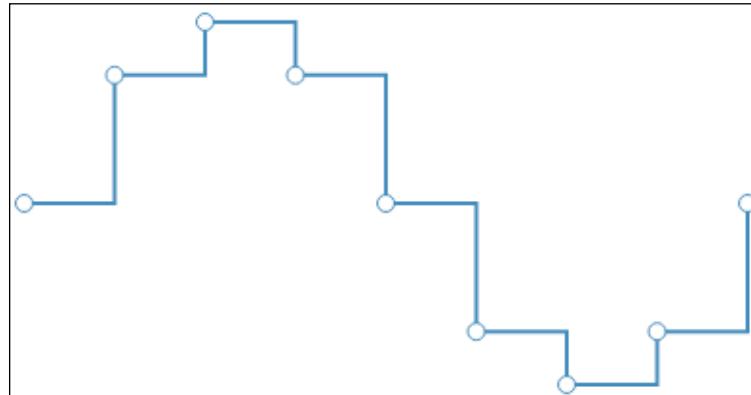
Step-before and step-after interpolations

The best way to demonstrate a step-before and step-after is just by giving examples. But essentially, each pair of points are connected by two lines, one horizontal and the other vertical.

With step-before, the vertical line come first:



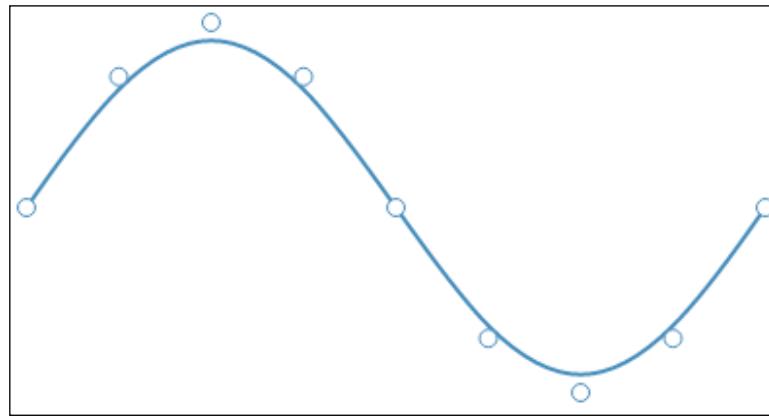
A step-after renders the horizontal line first:



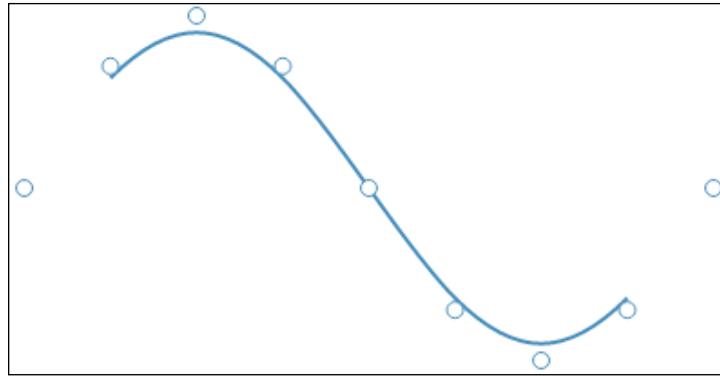
Creating curved lines using the basis interpolation

A **basis** curve will pass through the end points but may not pass through internal points. The internal points influence the curve of the line, but it is not necessary that the line runs through any of the internal points.

The following is an example of the basis interpolation:

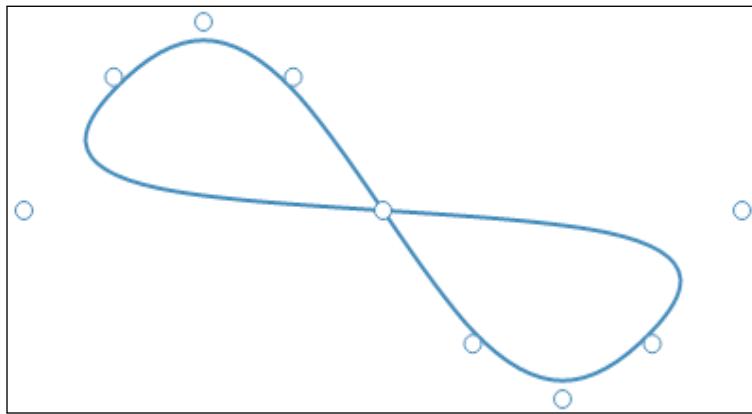


A **basis-open** interpolation does not pass through the end points. It looks similar to basis, but with the line not being drawn between the first and second points and between the next to last and last points:



Why would we want this? This would be in the case where the first and last are control points and can be changed in the X and Y values to influence how the curve moves through the inner points. Examining this is beyond the scope of this book, but I challenge you to take the concepts that you learned in *Chapter 8, Adding User Interactivity*, allowing the user to drag the control points around, and see how that changes the flow of the line.

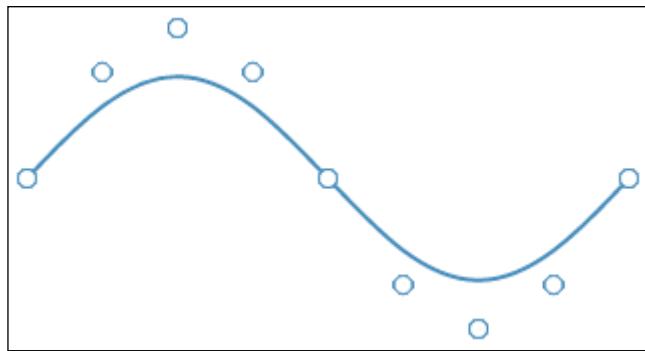
basis-closed tells the generator to close the loop and ensure the loop is smooth across all points (the small change in code is omitted again). The result is the following:



Pretty awesome! As you can see, you can use these interpolators to create really complex curved shapes. Imagine doing this by creating the path commands all by yourself. I dare you to examine the path commands for this line – there are a lot of them.

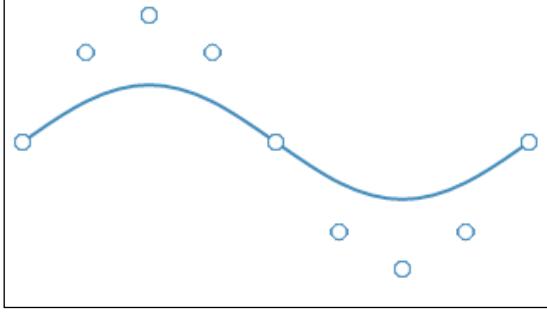
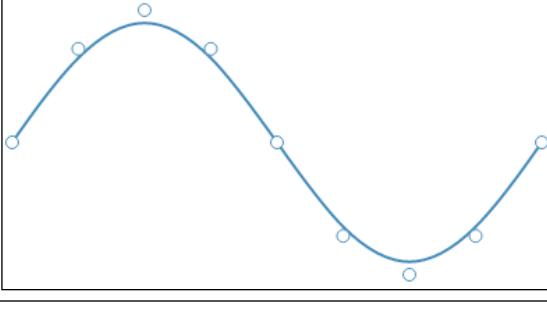
Creating curved lines using the bundle interpolation

`bundle` is similar to `basis`, except you can specify a parameter for the amount of tension in the line that is generated. Tension allows you to control how tightly the line will conform to the given points. To specify tension, chain the `.tension()` function with a parameter value between `0.0` and `1.0` (the default is `0.7`). The following shows a selected tension of `0.75`:



You can see how the generated line (well, curve) is now influenced a lot less closely by the points. If you set the value to `0.0`, this would actually be a straight line. To demonstrate efficiently other values for tension, the following table demonstrates the change in shape at various points of tension from `0.0` to `1.0`:

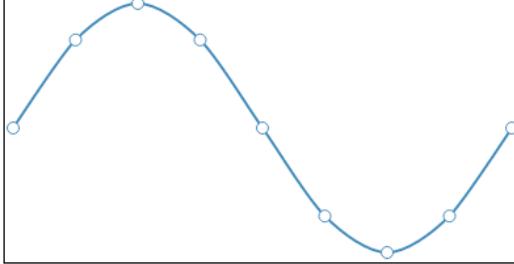
Tension	Result
<code>0.0</code>	

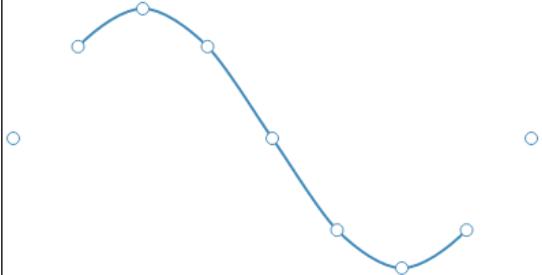
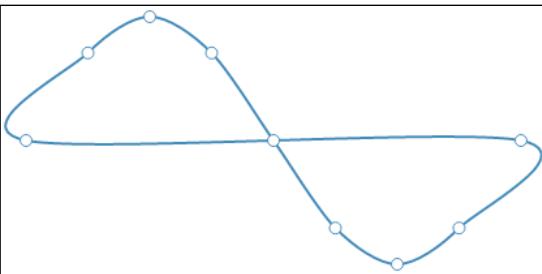
Tension	Result
0.5	
1.0	

If you compare a tension of 1.0 to the basis interpolation, you will notice they are identical.

Creating curved lines using the cardinal interpolation

cardinal curves are like basis curves, except that the lines are forced to run through all points. The following graph demonstrates the normal, open, and closed forms:

Interpolation	Result
cardinal	

Interpolation	Result
cardinal-open	
cardinal-closed	

Summary

In this chapter, we examined several techniques of creating complex shapes using D3.js path data generators. We started with examples of common generators, including line, area, circles, donuts, arcs, and diagonals. These are extremely powerful tools and enhance your ability to create complex visualizations easily.

We finished the chapter with an examination of line interpolators, a means of informing the line path generator of how to fit lines between data points. These interpolations, including the default linear interpolation, are for the basis of efficiently creating complex line graphs and curved shapes that fit data.

In the next chapter on layouts, we will move a little higher up the visual stack of D3.js to examine layouts, which are essentially generators for complex data visualizations.

10

Using Layouts to Visualize Series and Hierarchical Data

We are now going to get into what some refer to as the most powerful features of D3.js —layouts. Layouts encapsulate algorithms that examine your data and calculate the positions for visual elements for specific type of graphs such as bars, areas, bubbles, chords, trees, and many others.

We will dive into several useful layouts. These will be categorized into several main categories based upon the structure of the data and type of visualization such as stacked, hierarchical, chords, and flow-based diagrams. For each of the categories, we will go over creating a number of examples, complete with data and code.

Specifically, we will examine creating the following types of graphs and layouts:

- Stacked layouts to create bar and area graphs
- Hierarchical diagrams including trees, cluster dendrograms, and enclosures
- Relationships between items using chord diagrams
- Flowing data using streamgraphs and Sankey diagrams

Using stacked layouts

Stacks are a class of layouts that take multiple series of data, where each measurement from each series is rendered atop each other. These are suited for demonstrating the comparative size of the measurements from each series at each measurement. They are also great at demonstrating how the multiple streams of data change over the entire set of measurements.

Stacked diagrams basically come down to two different representations: stacked bar graphs and stacked area graphs. We will examine both of these and explain how to create these with D3.js.

Creating a stacked bar graph

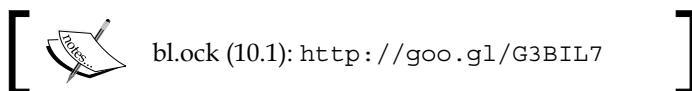
The implementation of a stacked bar graph is similar to that of a bar graph, except that we need to take into account the fact that the height of each bar consists of the sum of each measurement. Normally, each bar is subdivided, with each division sized relative to the sum, and is given a different color to differentiate it.

Let's jump into creating our own stacked bar graph. The data that will be used can be found at <https://goo.gl/6fJrxE>.

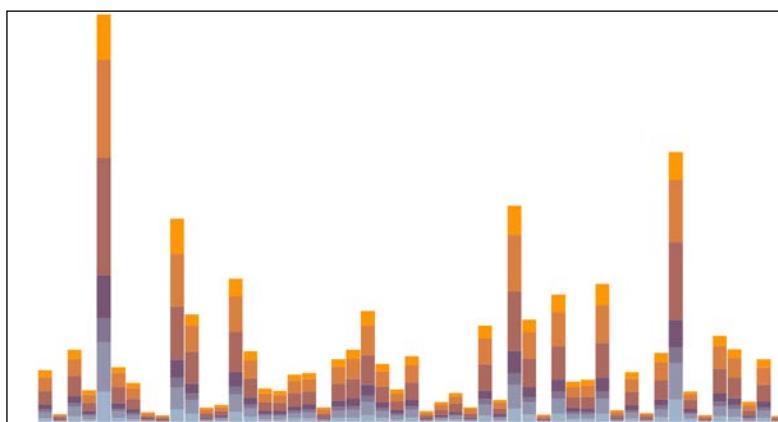
The following are the first few lines of the file. This data represents seven series of data, each series a specific range of age, broken down categorically by state. Each value represents the population for the given state in that age group.

```
State,Under 5 Years,5 to 13 Years,14 to 17 Years,18 to 24 Years,25 to  
44 Years,45 to 64 Years,65 Years and Over  
AL,310504,552339,259034,450818,1231572,1215966,641667  
AK,52083,85640,42153,74257,198724,183159,50277  
AZ,515910,828669,362642,601943,1804762,1523681,862573
```

The online example is available at the following link:



The resulting bar graph is the following:



The data is loaded using `d3.csv()`:

```
var url = 'https://gist.githubusercontent.com/d3byex/25129228aa50c30ef01f/raw/17838a0a03d94328a529de1dd768e956ce217af1/stacked_bars.csv';
d3.csv(url, function (error, data) {
```

Examining the first object in the resulting data, we see the following structure:

```
▼ 0: Object
  5 to 13 Years: "552339"
  14 to 17 Years: "259034"
  18 to 24 Years: "450818"
  25 to 44 Years: "1231572"
  45 to 64 Years: "1215966"
  65 Years and Over: "641667"
  State: "AL"
  Under 5 Years: "310504"
```

This array has 51 elements, one for each state of the US and Washington D.C. This data needs to be converted into a structure that gives us information for rendering each bar and the rectangle for each series within each of the bars. To do this, we need to go through three steps, the last one culminating with the use of `d3.layout.stack()`.

First, the code extracts the unique keys for each series of data, which is the age groups. This can be retrieved by filtering out all properties of each object in the array where the property name is not equal to `State`.

```
var keys = d3.keys(data[0])
  .filter(function (key) {
    return key !== "State";
  });
["Under 5 Years", "5 to 13 Years", "14 to 17 Years", "18 to 24 Years", "25 to 44 Years",
 "45 to 64 Years", "65 Years and Over"]
```

Using these keys, we can reorganize the data so that we have an array representing the values for each age group:

```
var statesAndAges = keys.map(function (ageRange) {
  return data.map(function (d) {
```

```
        return {
          x: d.State,
          y: +d[ageRange]
        };
      });
    );
  );
}
```

The `statesAndAges` variable now is a seven-element array, with each element being an array of objects representing the `x` and `y` values for each series:

```
▼ [Array[51], Array[51], Array[51], Array[51], Array[51], Array[51], Array[51]]
  ▼ 0: Array[51]
    ▼ 0: Object
      x: "AL"
      y: 310504
      ► __proto__: Object
    ▼ 1: Object
      x: "AK"
      y: 52083
      ► __proto__: Object
```

Now, using these keys, we create a `d3.layout.stack()` function and have it process this data.

```
var stackedData = d3.layout.stack()(statesAndAges);
```

The result of the stacking of this data is that the `stack` function will add an additional property, `y0`, to each object in each series. The value of `y0` will be the value of the sum of the `y` values in the previously lower-numbered series. To demonstrate, the following are the values of the objects in the first object of each array:

```
Object {x: "AL", y: 310504, y0: 0}
Object {x: "AL", y: 552339, y0: 310504}
Object {x: "AL", y: 259034, y0: 862843}
Object {x: "AL", y: 450818, y0: 1121877}
Object {x: "AL", y: 1231572, y0: 1572695}
Object {x: "AL", y: 1215966, y0: 2804267}
Object {x: "AL", y: 641667, y0: 4020233}
```

The value of y_0 in the first object is 0. The value of y_0 in the second is 310504, which is equal to $y_0 + y$ of the first object. The value of y_0 in the third object is $y_0 + y$ of the second, or 862843. This function has stacked the y values, with each y value being the value of y for the individual segment of the bar that will be rendered.

The data is now organized to render the bar graph. The next step is to create the main SVG element:

```
var width = 960, height = 500;
var svg = d3.select('body')
    .append("svg")
    .attr({
        width: width,
        height: height
   });
```

The code next calculates the x and y scales to map the bars into the specified number of pixels. The y scale will have a domain that ranges from 0 to the maximum sum of y_0 and y within all the series:

```
var yScale = d3.scale.linear()
    .domain([0,
        d3.max(stackedData, function (d) {
            return d3.max(d, function (d) {
                return d.y0 + d.y;
            });
        })
    ])
    .range([0, height]);
```

The x scale is set up as an ordinal `rangeRoundBands`, one for each state:

```
var xScale = d3.scale.ordinal()
    .domain(d3.range(stackedData[0].length))
    .rangeRoundBands([0, width], 0.05);
```

The code then creates a group for each of the series, assigning to each the color that the rectangles within will be filled with:

```
var colors = d3.scale.ordinal()
    .range(["#98abc5", "#8a89a6", "#7b6888",
        "#6b486b", "#a05d56", "#d0743c", "#ff8c00"]);
```

```
var groups = svg.selectAll("g")
  .data(stackedData)
  .enter()
  .append("g")
  .style("fill", function (d, i) {
    return colors(i);
});
```

The last step is to render all the rectangles. The following performs this by creating 51 rectangles within each group:

```
groups.selectAll("rect")
  .data(function (d) { return d; })
  .enter()
  .append("rect")
  .attr("x", function (d, i) {
    return xScale(i);
})
  .attr("y", function (d, i) {
    return height - yScale(d.y) - yScale(d.y0);
})
  .attr("height", function (d) {
    return yScale(d.y);
})
  .attr("width", xScale.rangeBand());
});
```

That's it! You have drawn this graph using this data.

Modifying the stacked bar into a stacked area graph

Stacked area graphs give a different view of the data than a stacked bar does. To create a stacked area graph, we change the rendering of each series of data as a path. The path is defined using an area generator, which has the y values on the lower end and the sum of $y_0 + y$ on the upper end.

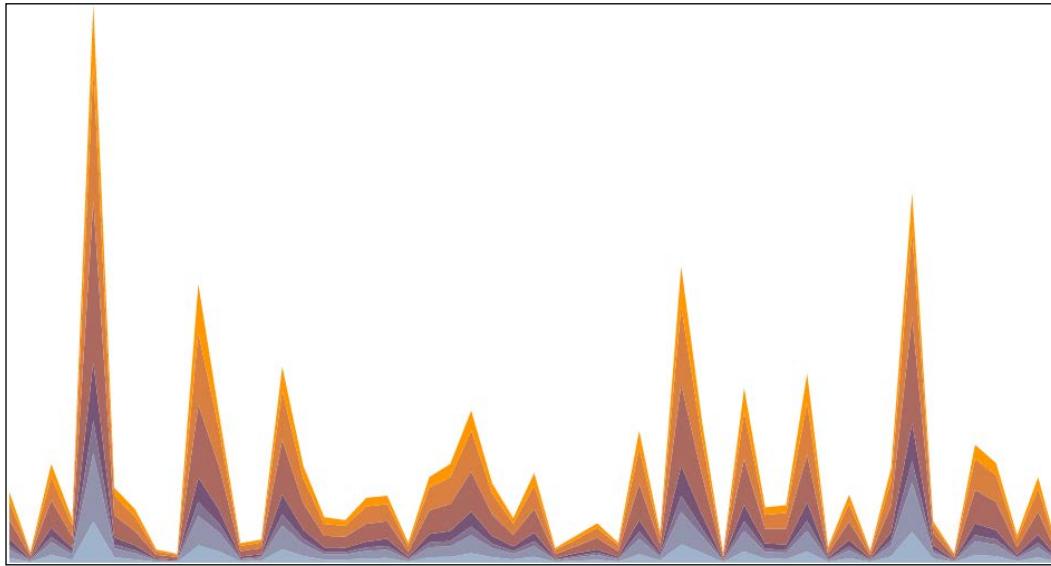
The code for the stacked area graph is available online at the following link:



bl.ock (10.2): <http://goo.gl/PRw8wv>



The resulting output from this example is the following:



The change from the previous example is relatively small. The data is loaded and organized exactly the same. The scales and colors are also created the same way.

The difference comes in the rendering of the visuals. Instead of groups of rectangles, we render a filled path for each series. The following creates these paths and assigns the color for each:

```
svg.selectAll("path")
  .data(stackedData)
  .enter()
  .append("path")
  .style("fill", function (d, i) {
    return colors(i);
});
```

This has generated the path elements, but has not assigned the path's `d` property yet to create the actual path data. That's our next step, but we first need to create an area generator to convert our data to that which is needed for the path. This area generator needs to have three values specified, the `x` value, `y0` (which represents the bottom of the area), and `y1` (which is at the top of the area):

```
var area = d3.svg.area()
  .x(function (d, i) {
    return xScale(i);
  })
  .y0(function (d) {
    return height - yScale(d.y0);
  })
  .y1(function (d) {
    return height - yScale(d.y + d.y0);
 });
```

And finally, we select the paths we just created and bind to each the appropriate series, setting the `d` attribute of the corresponding path to the result of calling the area generator. Note that this calls the area generator for each series:

```
svg.selectAll("path")
  .data(stackedData)
  .transition()
  .attr("d", function (d) {
    return area(d);
 });
```

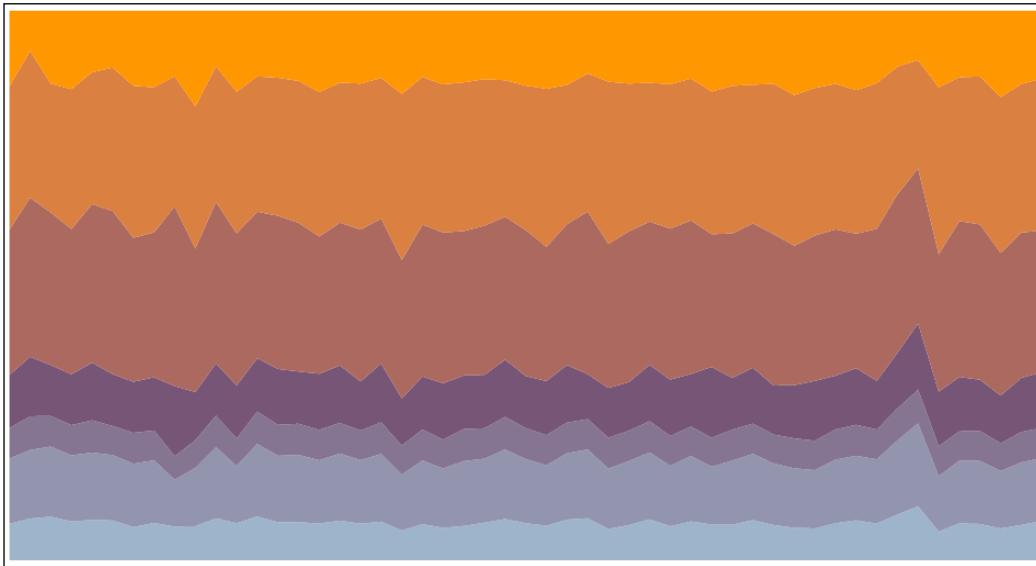
Converting the area graph to an expanded area graph

There is a variant of a stacked area graph known as an expanded area graph. An expanded area graph fills the entire area of the graph completely and can be used to easily visualize the relative percentage that each series represents at each point.

This type of graph is created from a stacked area graph by normalizing the data at each point across all series to 1.0. The following example demonstrates how this is performed:



The resulting graph is the following:



This visually gives us a good feel of how the relative size of each age group changes during the period. For the most part, the age groups have stayed at the same proportion, except for perhaps one state near the end of the data.

It's a really easy thing to convert the stacked area graph to an expanded area graph. To accomplish this, we need to do two things. The first of these is to change how we stack the data. We change the stack operation to the following:

```
var stackedData = d3.layout.stack()  
    .offset('expand')(statesAndAges);
```

The change here is to add a call to `.offset("expand")`. This informs D3.js to normalize the results to `[0, 1]` for each point. The default offset is "zero", which, as we have seen, starts Y values at 0 and does a running sum.

The data is now ready, and the second change is to change the y scale to the account for the domain as `[0, 1]`:

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

You now have your expanded area graph.

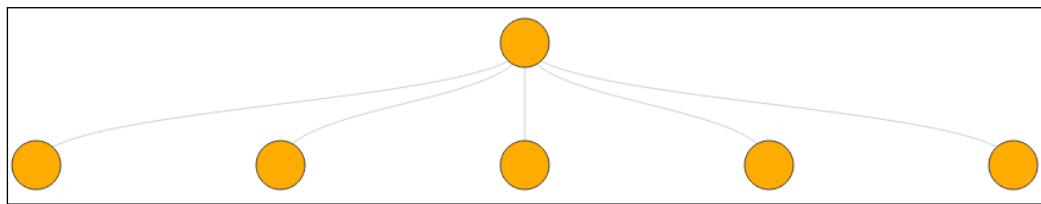
Visualizing hierarchical data

Hierachal layouts display information that is hierarchical in nature. That is perhaps a slightly recursive definition, but the basic idea is that certain data items break down into zero or more data items at a lower level, and then perhaps to another level, and so on, for as many levels as is required.

Hierarchical layouts are all created from the `d3.layout.hierarchy()` function, but there are specializations of this function that create various layouts which fall into common visual patterns such as trees, clusters, and enclosures and packs. We will take a look at an example of each of these types of layouts.

Tree diagrams

Tree diagrams are essentially node-link diagrams. In *Chapter 9, Complex Shapes using Paths*, we saw the use of a path generator known as a diagonal. This generator was able to create curved line segments that can connect a node to one or more nodes. To refresh you, we had an example that generated the following:



This is a basic node-link diagram. Tree diagrams utilize diagonals and apply them to many levels of hierarchy. The diagram can be structured as a tree or in other more complex layouts such as a radial cluster (which we will examine). The layouts will calculate the positions of the nodes and then we need to render the nodes and the attached diagonals.

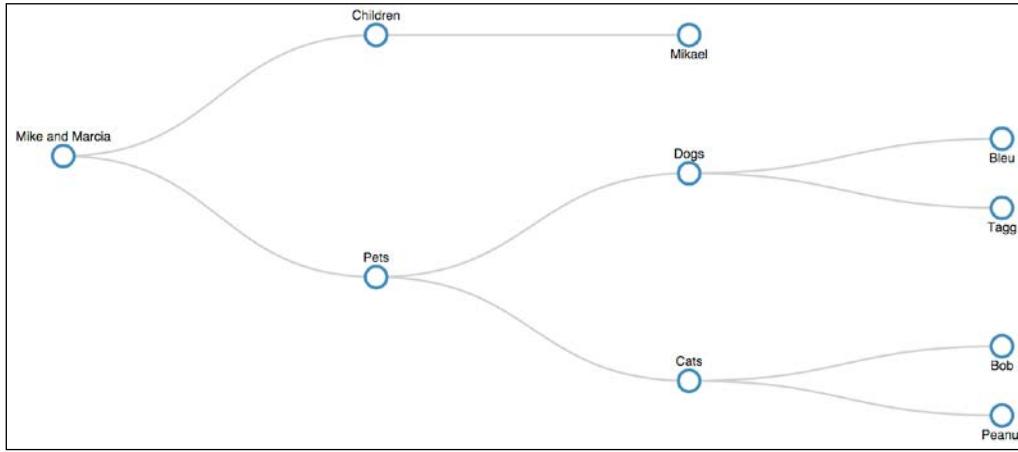
We will start by creating a simple tree diagram. The data is available at <https://goo.gl/mcdT9r>. The contents of the data are the following:

```
{
  "name": "Mike and Marcia",
  "children": [
    {
      "name": "Children",
      "children": [
        { "name": "Mikael" }
      ]
    },
    {
      "name": "Pets",
      "children": [
        {
          "name": "Dogs",
          "children": [
            { "name": "Bleu" },
            { "name": "Tagg" }
          ]
        },
        {
          "name": "Cats",
          "children": [
            { "name": "Bob" },
            { "name": "Peanut" }
          ]
        }
      ]
    }
  ]
}
```

The example is available at the following location:



The result of the rendering is the following tree diagram:



Our example begins with loading the data, establishing metrics for the diagram, creating the main SVG element, and establishing a main group and margins:

```
var url = 'https://gist.github.com/d3byex/25129228aa50c30ef01f/raw/c1c3ad9fa745c42c5410fba29cefccac47cd0ec7/familytree.json';
d3.json(url, function (error, data) {
    var width = 960, height = 500,
        nodeRadius = 10,
        margin = {
            left: 50, top: 10,
            bottom: 10, right: 40
        };

    var svg = d3.select('body')
        .append("svg")
        .attr({
            width: width,
            height: height
        });
    var mainGroup = svg.append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ')');
```

To convert the data into a visual representation of a tree, we will create a tree layout using the `d3.layout.tree()` function.

```
var tree = d3.layout.tree()
  .size([
    height - (margin.bottom + margin.top),
    width - (margin.left + margin.right),
  ]);
```

This informs D3.js that we want to create a tree that will map its data into a rectangle specified by `height` and `width`. Notice that the `height` is specified before `width`.

There are two visual components to the graph: the nodes, represented by circles, and the edges, which are diagonals. To calculate the nodes, we use the `.nodes()` function of the layout and pass it our data.

```
var nodes = tree.nodes(data);
```

The tree function looks for a top-level node with a `children` property. It will traverse all the nodes in the hierarchy and determine its depth, which, in this case, has four levels. It will then add `x` and `y` properties to each node, where these represent the calculated position of the nodes based upon the layout and the specific `width` and `height`.

Examining the contents of the `nodes` variable, we can see that D3.js has given us the positions for each node (the following shows the first two nodes):

```
▼ [Object, Object, Object, Object, Object, Object, Object, Object, Object]
  ▼ 0: Object
    ► children: Array[2]
      depth: 0
      name: "Mike and Marcia"
      x: 176
      y: 0
      ► __proto__: Object
  ▼ 1: Object
    ► children: Array[1]
      depth: 1
      name: "Children"
      ► parent: Object
        x: 64
        y: 290
```

To get the links in the tree, we call `tree.links(nodes)`:

```
var links = tree.links(nodes);
```

The following shows the link that results in this example:

```
▼ [Object, Object, Object, Object, Object, Object, Object, Object, Object]
  ▼ 0: Object
    ▼ source: Object
      ► children: Array[2]
        depth: 0
        name: "Mike and Marcia"
        x: 176
        y: 0
      ► __proto__: Object
    ▼ target: Object
      ► children: Array[1]
        depth: 1
        name: "Children"
      ► parent: Object
        x: 64
        y: 290
      ► __proto__: Object
    ► __proto__: Object
```

The newly created data structure consists of an element for each link, of which each object contains a `source` and `target` property that points to the node that is on each end of the link.

We now have our data ready for creating visuals. Next is the statement for creating the generator for the diagonals. We use the `.projection()` function, since we need to tell the generator how to find the `x` and `y` value from each datum:

```
var diagonal = d3.svg.diagonal()
  .projection(function(d) {
    return [d.y, d.x];
});
```

Now we can create the diagonals, reusing the generator for each. The diagonals are created before the nodes, because we want the nodes to be in front:

```
mainGroup.selectAll('path')
  .data(links)
  .enter()
  .append('path', 'g')
  .attr({
    d: diagonal,
    fill: 'none',
    stroke: '#ccc',
    'stroke-width': 2
  });

```

Now the code creates the circles and the labels. We will represent each node with a group containing a circle and a piece of text. The following creates these groups and places them at the calculated locations:

```
var circleGroups = mainGroup.selectAll('g')
  .data(nodes)
  .enter()
  .append('g')
  .attr('transform', function (d) {
    return 'translate(' + d.y + ', ' + d.x + ')';
  });

```

Next, we add the circles as a child of each node's group element:

```
circleGroups.append('circle')
  .attr({
    r: nodeRadius,
    fill: '#fff',
    stroke: 'steelblue',
    'stroke-width': 3,
  });

```

And then we add the text for the node label to the group:

```
circleGroups.append('text')
  .text(function (d) {
    return d.name;
  })
  .attr('y', function (d) {
    return d.children || d._children ?
      -nodeRadius * 2 : nodeRadius * 2;
  })
  .attr({
    dy: '.35em',
    'text-anchor': 'middle',
    'fill-opacity': 1
  })
  .style('font', '12px sans-serif');
```

The function, when assigning the `y` attribute, offsets the position of the text to be above the circle if the node is not a leaf and underneath the node if it is a leaf node.

Creating a cluster dendrogram

A hierarchy can also be visualized as a variant of a tree known as a **cluster dendrogram**. A cluster dendrogram differs from a tree graph in that we use a cluster layout. This layout places the root of the tree at the center. The depth of the data is calculated, and that number of levels of concentric circles are fit into the diagram. The nodes for each level of depth are then placed around the edge of the circle for their respective depth.

To demonstrate this, we will utilize the data available at <https://goo.gl/t3M7n1>. This data represents three levels of data, with one root node and four nodes on the second level; each of those nodes has nine children.

The following is a sample of the data:

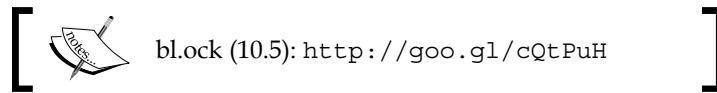
```
{
  "name": "1",
  "children": [
    {
      "name": "1-1",
      "children": [
        { "name": "1-1-1" },
        { "name": "1-1-2" },
        { "name": "1-1-3" },
```

```

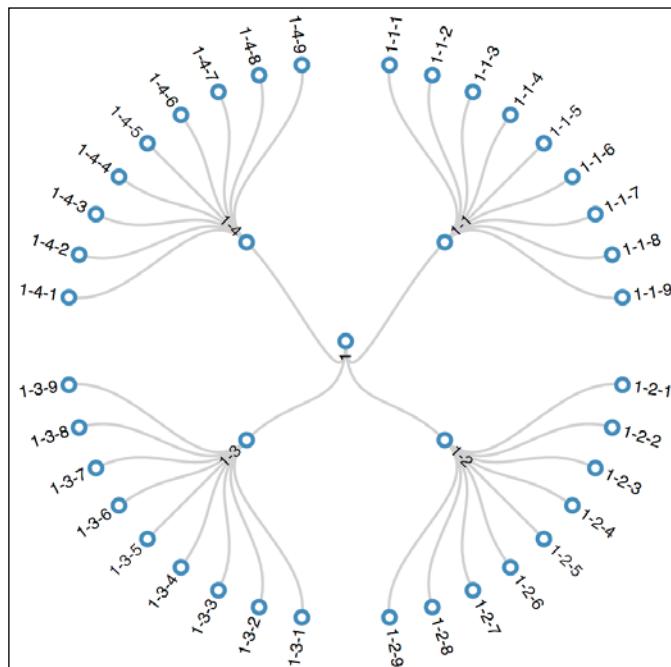
    {
      "name": "1-1-4" },
      {
        "name": "1-1-5" },
        {
          "name": "1-1-6" },
            {
              "name": "1-1-7" },
                {
                  "name": "1-1-8" },
                    {
                      "name": "1-1-9" }
        ]
    },
    {
      "name": "1-2",
      "children": [
        {
          "name": "1-2-1" },
          {
            "name": "1-2-2" },
              {
                "name": "1-2-3" },
        ...
      ]
    }
  }

```

The example is available at the following location:



The resulting graph is the following:



Let's step through how this is created. The code is similar to the tree example, but with some differences. After the data is loaded, the main SVG element is created, and then a group is placed within the element:

```
var center = width / 2;  
var mainGroup = svg.append('g')  
    .attr("transform", "translate(" + center + "," +  
          center + ")");
```

The layout algorithm will calculate the points around a center at **(0, 0)**, so we center the group to center the graph.

The layout is then created using `d3.layout.cluster()`:

```
var cluster = d3.layout.cluster()  
    .size([  
        360,  
        center - 50  
    ]);
```

The size specifies two things; the first parameter is the number of degrees that the points will sweep through on the outer circle. This specifies 360 degrees so that we completely fill the outer circle. The second parameter is the tree depth, or what is essentially the radius of the outermost circle.

Next, we use the layout to calculate the position for the nodes and links:

```
var nodes = cluster.nodes(data);  
var links = cluster.links(nodes);
```

It is worth examining the first few nodes that result from these calculations:

```
[Object, Object,  
Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object,  
Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object]  
▼ 0: Object  
  ▷ children: Array[4]  
    depth: 0  
    name: "1"  
    x: 180  
    y: 0  
  ▷ __proto__: Object  
▼ 1: Object  
  ▷ children: Array[9]  
    depth: 1  
    name: "1-1"  
  ▷ parent: Object  
    x: 45  
    y: 100  
  ▷ __proto__: Object  
▼ 2: Object  
  depth: 2  
  name: "1-1-1"  
  parent: Object  
  x: 9  
  y: 200  
  ▷ __proto__: Object
```

The `x` and `y` properties specify a direction and distance at which the node (and edges) is to be placed. The `x` property specifies the angle from vertical, and the value of the `y` property specifies the distance.

The diagonals are calculated using a radial diagonal, which needs to convert the `x` values into radians:

```
var diagonal = d3.svg.diagonal().radial()
    .projection(function(d) {
        return [
            d.y,
            d.x / 180 * Math.PI
        ];
    });
});
```

Now we can use this radial generator diagonal that connects the nodes:

```
mainGroup.selectAll('path')
    .data(links)
    .enter()
    .append('path')
    .attr({
        'd': diagonal,
        fill: 'none',
        stroke: '#ccc',
        'stroke-width': 2
    });
});
```

Next, we create a group to hold the node and the text. The trick to this is that we need to translate and rotate the group into the correct position:

```
var nodeGroups = mainGroup.selectAll("g")
    .data(nodes)
    .enter()
    .append("g")
    .attr("transform", function(d) {
        return "rotate(" + (d.x - 90) + ")translate(" + d.y + ")";
    });
});
```

We rotate the group by 90 degrees from the calculated angle. This changes the orientation of the the text to flow out from the circle, along the diagonals. Note that rotate works in degrees, not radians, as was required for the radial generator. The translate uses just the y value, which moves the group out that distance along the specified angle. Now we add the circles to the group:

```
nodeGroups.append("circle")
  .attr({
    r: nodeRadius,
    fill: '#fff',
    stroke: 'steelblue',
    'stroke-width': 3
  });
}
```

And finally, we add the text. Note the small calculation around the text being at an angle greater or less than 180 degrees. This essentially says that nodes on the left-half of the diagram are positioned with the end of the text against the node and on the right side, start at the beginning of the text. The text is also transformed by twice the circle radius to prevent it from overlapping the circle:

```
nodeGroups.append('text')
  .attr({
    dy: '.31em',
    'text-anchor': function(d) {
      return d.x < 180 ? 'start' : 'end';
    },
    'transform': function(d) {
      return d.x < 180 ?
        'translate(' + (nodeRadius*2) + ') ' +
        'rotate(180)' +
        'translate(' + (-nodeRadius*2) + ')';
    }
  })
  .style('font', '12px sans-serif')
  .text(function(d) { return d.name; });
}
```

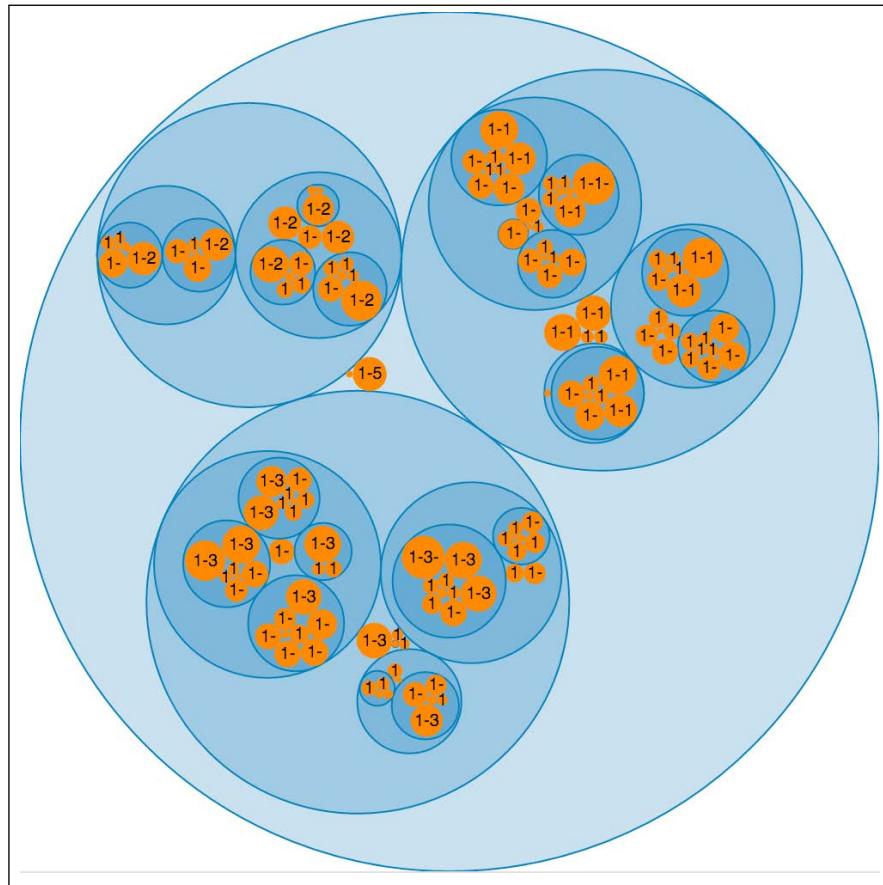
Representing hierarchy with an enclosure diagram

Enclosure diagrams use nesting of visuals to represent the hierarchy. The size of each leaf node's circle reveals a quantitative dimension of each data point. The enclosing circles show the approximate cumulative size of each subtree, but note that because of wasted space, there is some distortion between levels. Therefore, only the leaf nodes can be compared accurately.

The following is the location of the online example:



The following image is the resulting visual:



The data used by the example is available at <https://goo.gl/Rzv1v3>. It is similar in structure to the data in the previous example, except that a `value` property is added to each node. The values of the leaf nodes are summed in their parents, repeating all the way to the top.

Essentially, this data is a rollup of the values, much like what would be performed when rolling up sales numbers from sales persons, to offices, to divisions, to the corporate level. The diagram then allows us to see relative sizes of the numbers in the leaf nodes, which are colored orange, and then get an idea of the total at each level up the tree.

Now let's examine how this is created. The example begins with loading of the data and then creating an SVG element of a specified diameter. Then, a pack layout is created that is also used the diameter. The hierarchical bubbles that are created will be measured to fit within the specified diameter:

```
var pack = d3.layout.pack()  
    .size([diameter, diameter])  
    .value(function (d) { return d.value; });
```

Now we render the circles. For each node, we append a group that is translated to the appropriate position and then a circle is appended with its radius set to the calculated radius (`d.r`), the `fill`, `fill-opacity`, and the `stroke` to different values depending on whether the node is a leaf or not:

```
var nodes = svg.datum(data)  
    .selectAll('g')  
    .data(pack.nodes)  
    .enter()  
    .append('g')  
    .attr('transform', function (d) {  
        return 'translate(' + d.x + ',' + d.y + ')';  
    });  
  
nodes.append('circle')  
    .each(function (d) {  
        d3.select(this)  
            .attr({
```

```
r: d.r,  
fill: d.children ? 'rgb(31, 119, 180)' :  
    '#ff7f0e',  
'fill-opacity': d.children ? 0.25 : 1.0,  
stroke: d.children ? 'rgb(31, 119, 180)' : 'none'  
});  
});
```

The last step is to add the text to the leaf circles (the ones without children, as specified using the filter):

```
nodes.filter(function(d) {  
    return !d.children;  
})  
.append('text')  
.attr('dy', '.3em')  
.style({  
    'text-anchor': 'middle',  
    'font': '10px sans-serif'  
})  
.text(function(d) {  
    return d.name.substring(0, d.r / 3);  
});
```

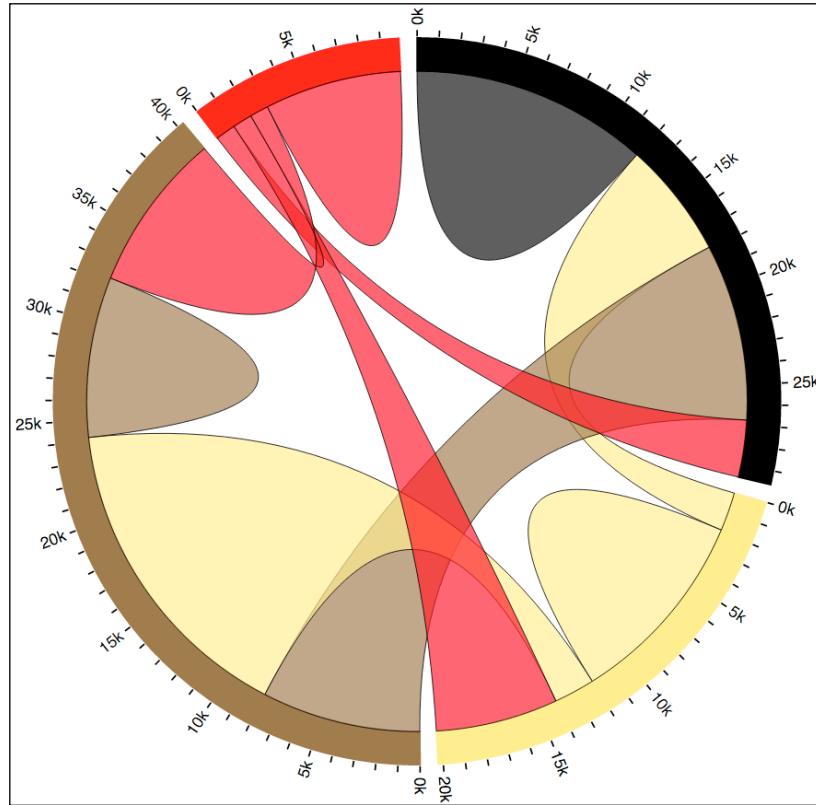
Representing relationships with chord diagrams

Chord diagrams demonstrate the relationships among a group of entities. To demonstrate, we will use the example available at the following link:



bl.ock (10.7): <http://goo.gl/8mRDSg>

The resulting diagram is the following:



The data in this example is a square matrix of data, with rows and columns representing hair color (black, blonde, brown, and red). The data represents a total sample of **100000** measurements, where each row demonstrates the total count of the other hair colors which a person of a given hair color prefers:

	Prefers				
Has	Black	Blonde	Brown	Red	Total
Black	11975	5871	8916	2868	29630
Blonde	1951	10048	2060	6171	20230
Brown	8010	16145	8090	8045	40290
Red	1013	990	940	6907	9850
Total	22949	30354	20006	23991	100000

To explain the diagram, each outer ring segment represents the number of people that have a given hair color. The size of these ring segments is relative to the percentage of people of a given hair color. Each arc from a given color ring segment to another ring segment (or itself) represents the number of people of that hair color that prefer the hair color on the other side of that arc and vice versa. The ticks on the outside of each ring segment gives a feel for the total number of the people represented.

Now let's step through creating this graph. First, we create our top-level SVG elements. The main group is translated to the center, as the positions will be centered around **(0, 0)**:

```
var width = 960, height = 500;
var svg = d3.select('body')
    .append('svg')
    .attr({
        width: width,
        height: height
    });
var mainGroup = svg.append('g')
    .attr('transform', 'translate(' + width / 2 + ',' +
           height / 2 + ')');
```

Now let's declare the data. We will use a hard-coded array instead of reading from a file. These values represent the values from the previous table, exclusive of the totals:

```
var matrix = [
    [11975, 5871, 8916, 2868],
    [1951, 10048, 2060, 6171],
    [8010, 16145, 8090, 8045],
    [1013, 990, 940, 6907]
];
```

We then use the `d3.layout.chord()` function to create the layout object for this graph.

```
var layout = d3.layout.chord()
    .padding(.05)
    .matrix(matrix);
```

`.padding(0.05)` states that there will be 0.05 radians of space between the sections on the outside of the diagram, and the call to `.matrix()` specifies the data to use.

The following line of code creates the colors that will be utilized (black, blondish, brownish, and reddish):

```
var fill = d3.scale.ordinal().domain(d3.range(4))
    .range(['#000000', '#FFEE89', '#957244', '#FF0023']);
```

Then, the ring segments are rendered. The inner and outer radius of the ring segments is calculated as percentages of the smallest dimension of the visual. The data that is bound is the group's property of the layout object. For each of these, we render a path using an arc generator:

```
var innerRadius = Math.min(width, height) * 0.41,
    outerRadius = innerRadius * 1.1;
mainGroup.append('g')
    .selectAll('path')
    .data(layout.groups)
    .enter()
    .append('path')
    .style('fill', function(d) { return fill(d.index); })
    .style('stroke', function(d) { return fill(d.index); })
    .attr('d', d3.svg.arc()
        .innerRadius(innerRadius)
        .outerRadius(outerRadius));
```

Next, the chords are rendered. A `d3.svg.chord()` function will be applied to each datum, and a path of the size `innerRadius` is generated:

```
mainGroup.append('g')
    .selectAll('path')
    .data(layout.chords)
    .enter()
    .append('path')
    .attr('d', d3.svg.chord()
        .radius(innerRadius))
    .style('fill', function(d) { return fill(d.target.index); })
    .style({
        opacity: 1,
        stroke: '#000',
        'fill-opacity': 0.67,
        'stroke-width': '0.5px'
    });
});
```

At this point, we have created the entire chord graph sans ticks and labels. We will omit covering those in the book, but feel free to check out the sample code with the text to see how this is performed.

Techniques to demonstrate the flow of information

The last two layouts and corresponding visualizations that we will examine help the viewer to understand how data changes as it flows over time or through intermediate points.

Using streamgraphs to show changes in values

A **streamgraph** demonstrates the change in values in a multiple series of data as a flowing stream of data. The height of each stream represents the value of that stream at that moment in time.

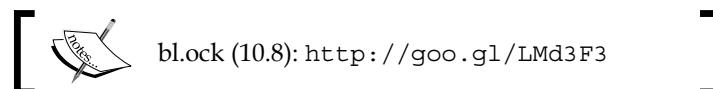
They are useful for demonstrating where certain categories start or stop at different points along the graph. Common examples are data such as box-office receipts or the number of listeners for various artists on streaming media as they change over time.

To demonstrate a streamgraph, we will use the data available at <https://goo.gl/HTL4HG>.

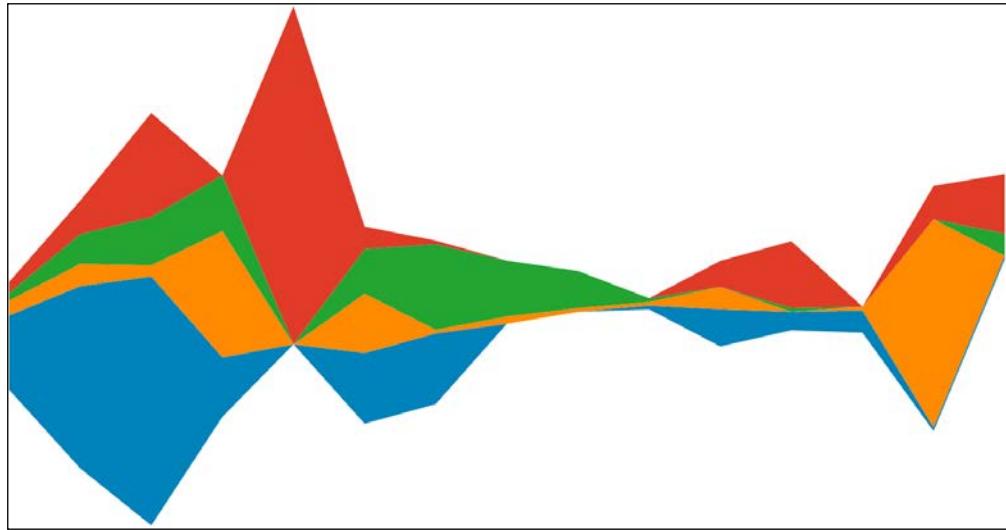
This data consists of four series of data:

```
[  
  [ 20, 49, 67, 16, 0, 19, 19, 0, 0, 1, 10, 5, 6, 1, 1 ],  
  [ 4, 6, 3, 34, 0, 16, 1, 2, 1, 1, 6, 0, 1, 56, 0 ],  
  [ 2, 8, 13, 15, 0, 12, 23, 15, 10, 1, 0, 1, 0, 0, 6 ],  
  [ 3, 9, 28, 0, 91, 6, 1, 0, 0, 0, 7, 18, 0, 9, 16 ]  
]
```

The online example is available at the following location:



The following is the resulting streamgraph:



This graph allows us to see how each individual series of data is related to each other at each point of measurement. It is, in a way, like a stacked area chart, but instead of each being fixed at a common baseline, the bottom of the graph is also allowed to vary in location.

The example begins by loading the data and setting up the main SVG element:

```
var url = 'https://gist.githubusercontent.com/  
d3byex/25129228aa50c30ef01f/raw/4393a0e579cbfd9bb20a431ce93c72fb1  
ea23537/streamgraph.json';  
d3.json(url, function (error, rawData) {  
    var width = 960, height = 500;  
    var svg = d3.select('body')  
        .append('svg')  
        .attr({  
            'width': width,  
            'height': height  
        });
```

We need to massage the data a little bit, as the call to the layout function will expect it in the same format as an area graph, which is an array of arrays of objects with `x` and `y` properties. The following code creates this, using the position of the value in each array as the `x` value:

```
var data = Array();
d3.map(rawData, function (d, i) {
    data[i] = d.map(function (i, j) {
        return { x: j, y: i };
    });
});
```

Next, the code creates the axes, with the X axis being a linear axis representing the number of points in each series:

```
var numPointsPerLayer = data[0].length;

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

The layout is the familiar stack layout that was used in the area graph example, but we chain a call to `.offset('wiggle')`:

```
var layers = d3.layout.stack()
    .offset('wiggle')(data);
```

The remainder of the code continues just as an area graph, using an area path generator and similarly scaled Y axis.

Representing flows through multiple nodes

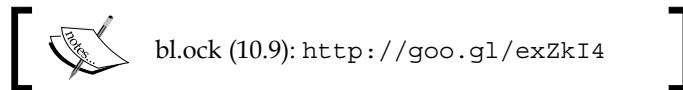
Instead of showing a continuous flow like a streamgraph, a [Sankey](#) diagram emphasizes how the flow quantity changes proportionally. This is somewhat like a chord diagram, but a Sankey has the ability to visualize more complex flows than just between two items.

In a Sankey diagram, the width of the lines between the nodes represents the volume of the flow between two nodes. Normally, flows start at one or more nodes on the left, flow through intermediates, and then terminate at nodes on the right.

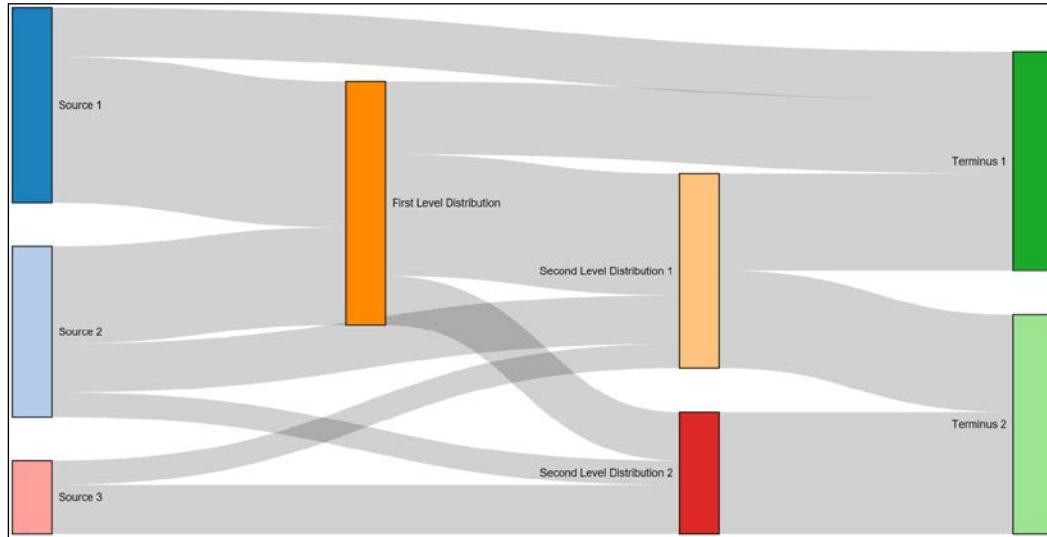
The example diagram uses the data available at <https://goo.gl/lgQZBz>. This data consists of declarations for eight nodes and then the links between the nodes along with the amount of the flow between the nodes:

```
{  
  "nodes": [  
    {"node":0, "name":"Source 1"},  
    {"node":1, "name":"Source 2"},  
    {"node":2, "name":"First Level Distribution"},  
    {"node":3, "name":"Second Level Distribution 1"},  
    {"node":4, "name":"Terminus 1"},  
    {"node":5, "name":"Terminus 2"},  
    {"node":6, "name":"Second Level Distribution 2"},  
    {"node":7, "name":"Source 3"}  
  "links": [  
    {"source":0, "target":2, "value":6},  
    {"source":0, "target":4, "value":2},  
  
    {"source":1, "target":2, "value":4},  
    {"source":1, "target":3, "value":2},  
    {"source":1, "target":6, "value":1},  
  
    {"source":2, "target":3, "value":5},  
    {"source":2, "target":4, "value":3},  
    {"source":2, "target":6, "value":2},  
  
    {"source":3, "target":4, "value":4},  
    {"source":3, "target":5, "value":4},  
  
    {"source":6, "target":5, "value":5},  
    {"source":7, "target":6, "value":2},  
    {"source":7, "target":3, "value":1}  
}
```

The online example is available at the following location:



The resulting Sankey diagram from this data will be the following:



The Sankey layout is considered a plugin to D3.js. It is not in the base library, so you need to retrieve the code, and make sure to reference it in your app. This code is available at <https://github.com/d3/d3-plugins/tree/master/sankey>, or you can grab it from the book's example.

The example begins by loading the data and creating the main SVG elements:

```
var url = 'https://gist.githubusercontent.com/d3byex/25129228aa50c30ef01f/raw/e6ea7c4728e45fb8d0464b21686eec806687e117/sankey.json';
d3.json(url, function(error, graph) {
    var width = 950, height = 500;
    var svg = d3.select('body')
        .append('svg')
        .attr({
            width: width,
            height: height
        });
    var mainGroup = svg.append('g');
```

We create the layout using the plugin as follows. There are lot of parameters here to specify the size of the nodes, the padding, overall size of the diagram, where to get the links and nodes in your data, and layout specifying the number of iterations to be processed for positioning the nodes:

```
var sankey = d3.sankey()
  .nodeWidth(36)
  .nodePadding(40)
  .size([width, height])
  .nodes(graph.nodes)
  .links(graph.links)
  .layout(10);
```

The flow paths (links) are rendered by creating paths representing flows. The structure of the path is provided by referencing the `sankey.link()`, which is a function that creates the path data for the flow:

```
mainGroup.append('g')
  .selectAll('g.link')
  .data(data.links)
  .enter()
  .append('path')
  .attr({
    d: sankey.link(),
    fill: 'none',
    stroke: '#000',
    'stroke-opacity': 0.2,
    'stroke-width': function(d) { return Math.max(1, d.dy) }
  })
  .sort(function(a, b) { return b.dy - a.dy; });
```

Now we create a group to hold the nodes and place them into position based on the `x` and `y` properties provided by the layout. The `.node` style is used simply to differentiate the selection of these groups from those of the paths (which used `.link`):

```
var nodes = mainGroup.append('g')
  .selectAll('g.node')
  .data(data.nodes)
  .enter()
  .append('g')
  .attr('transform', function(d) {
    return 'translate(' + d.x + ', ' + d.y + ')';
});
```

Then, we insert a colored rectangle into the groups:

```
var color = d3.scale.category20();
nodes.append('rect')
  .attr({
    height: function(d) { return d.dy; },
    width: sankey.nodeWidth(),
    fill: function(d, i) {
      return d.color = color(i);
    },
    stroke: 'black'
  });

```

We also include text to describe the node, with some logic to position the label:

```
nodes.append('text')
  .attr({
    x: -6,
    y: function(d) { return d.dy / 2; },
    dy: '.35em',
    'text-anchor': 'end'
  })
  .style('font', '10px sans-serif')
  .text(function(d) { return d.name; })
  .filter(function(d) { return d.x < width / 2; })
  .attr({
    x: 6 + sankey.nodeWidth(),
    'text-anchor': 'start'
  });

```

Summary

We have covered a lot in this chapter. The overall focus was on creating complex graphs that utilize D3.js layout objects. These included a multitude of graphs in different categories including stacked, packed, clustered, flow-based, hierarchical, and radial.

One of the beauties of D3.js is the ease at which it allows you to create these complex visuals. They are pattern-oriented such that the code for each is often very similar, with just a slight change of layout objects.

In the next chapter, we will look at a specific type of graph in detail: the network diagram. These extend upon several concepts we have seen in this chapter, such as flow and hierarchy, to allow us to visualize very complex network data such as those found in social networks.

11

Visualizing Information Networks

In this chapter, we will examine a specific type of layout known as a **force-directed graph**. These are a type of visualization that are generally utilized to display network information: interconnected nodes.

A particularly common type of network visualization is of the relationships within a social network. A visualization of a social network can help you understand how different people have formed various relationships. These include links between others as well as the way groups of people form clusters or cliques of friends and how those groups interrelate.

D3.js provides extensive capabilities for creating very complex network visualizations using force-directed networks. We will overview a number of representative examples of these graphs, cover a little bit of the theory of how they operate, and dive into a few examples to demonstrate their creation and usage.

Specifically, in this chapter we will cover the following topics:

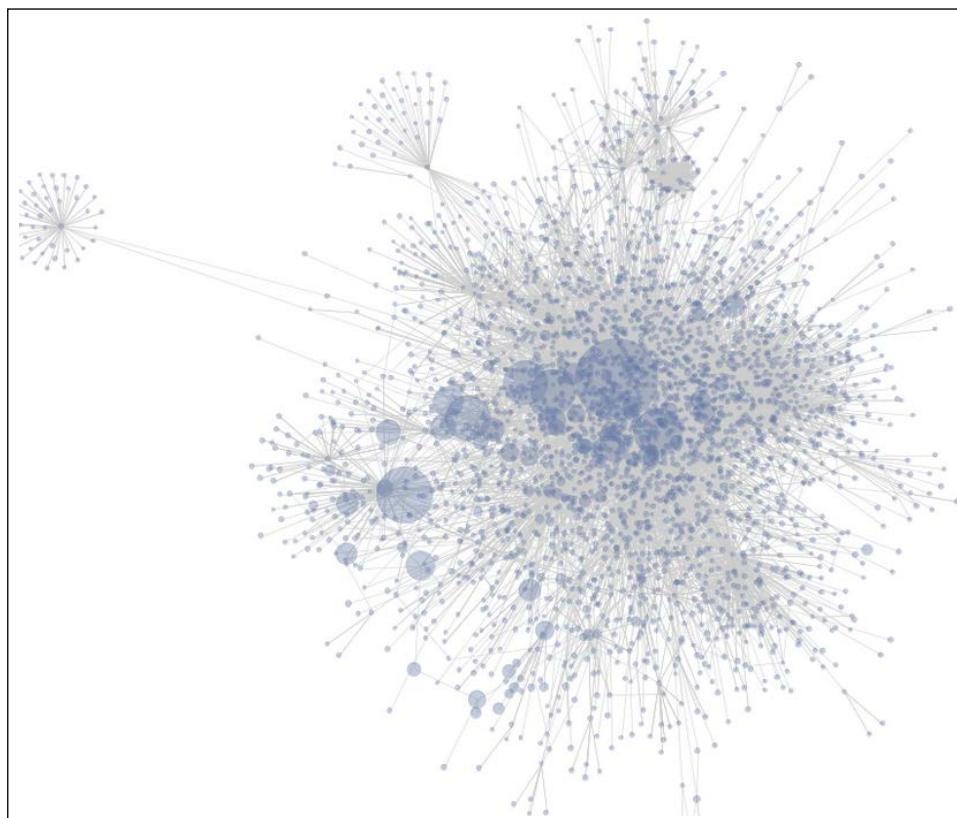
- A brief overview of force-directed graphs
- Creating a basic force-directed graph
- Modifying the length of the links
- Forcing nodes to move away from each other
- Labeling the nodes
- Forcing nodes to stay in place
- Expressing directionality and type with link visuals

An overview of force-directed graphs

There are a number of means of rendering network data. A particularly common one, which we will examine in this chapter, is to use a class of algorithms known as force-directed layouts.

These algorithms position the nodes in the graph in a two or three dimensional space. The positioning is performed by assigning forces along edges and nodes, and then these forces are used to simulate moving the nodes into a position where the amount of energy in the entire system is minimized.

The following is a representative picture of a force-directed graph from a Wiki. Nodes are pages, and the lines between the nodes represent the links between the pages. Node size varies based on the number of links in/out of a particular node:



The fundamental components of a force-directed graph are the nodes in the graph and the relations between those nodes. The graph is iteratively laid out, usually animated during the process, and can take quite a few iterations to **stabilize**.

The force layout algorithm in D3.js takes into account a number of factors. A few of the important parameters of the algorithm and how they influence the simulation are the following:

- **Size (width and height):** This represents an overall size of the diagram, and a center of gravity, normally the center of the diagram. Nodes in the diagram will tend to move towards this point. If nodes do not have an initial x and y position, then they will be placed randomly in a position between 0 and width in the x direction and height in the y direction.
- **Charge:** This describes how much a node attracts other nodes. Negative values push away other nodes, and positive numbers attract. The larger the value in either direction, the stronger is the force in that direction.
- **Charge distance:** This specifies the maximum distance over which charge has effect (it defaults to infinity). Smaller values assist in performance of the layout, and result in a more localized layout of nodes in clusters.
- **Friction:** Represents an amount of velocity delay. This value should be in the range of [0, 1]. At each tick of the layout, the velocity of every node is multiplied by this value. Using a value of 0 therefore, freezes all nodes in place, and 1 is a frictionless environment. Values in between eventually slow the nodes to a point where overall motion is small enough, and the simulation can be considered complete as the total amount of movement falls below the layout threshold at which point the graph is referred to as stable.
- **Link distance:** This specifies a desired distance between nodes at the end of the simulation. At each tick of the simulation, the distance between linked nodes is compared to this value, and nodes move towards or away from each other to try to reach the desired distance.
- **Link strength:** This is a value in the range of [0, 1], specifying how stretchable the link distance is during the simulation. A value of 0 is rigid and 1 is completely flexible.
- **Gravity:** This specifies an attraction of each node to the center of the layout. This is a weak geometric constraint. That is, the higher the overall gravity, the further away it is from the center of the rendering. This value is useful for keeping layouts relatively centered in the diagram and in keeping disconnected nodes from flying out to infinity.

We will go over enough of these parameters to get a good feel for making useful visualizations.



More detail on all the layout parameters is available at <https://github.com/mbostock/d3/wiki/Force-Layout>.



In addition to the parameters that facilitate the actual layout of the nodes, it is also possible to use other visual in a force-directed graph to convey various values in the underlying information:

- The color of a node can be used to distinguish nodes of particular types, such as people versus employers, or by their relation, such as all persons who work at a particular employer, or how many degrees of separation the node is from another node.
- The size of a node, which generally represents the magnitude of importance of the node. Often the number of links influence the size of a node.
- The thickness of the rendering of a link can be used to demonstrate that certain links have more influence than others or that the links are of particular types, that is, highways versus railways.
- The directionality of link, showing that the link has either no directionality or is one or bi-directional.

A simple force-directed graph

Our first example will demonstrate how to construct a force-directed graph. The online example is available at the following link:



bl.ock (11.1): <http://goo.gl/ZyxCej>



All our force-directed graphs will start by loading data that represents a network. This example uses the data at https://gist.githubusercontent.com/d3byex/5a8267f90a0d215fcb3e/raw/ba3b2e3065ca8eafb375f01155dc99c569fae66b/uni_network.json.

The following are the contents of the file at the preceding link:

```
{  
  "nodes": [  
    { "name": "Mike" },  
    { "name": "Marcia" },  
    { "name": "Chrissy" },  
    { "name": "Selena" },  
    { "name": "William" },  
    { "name": "Mikael" },  
    { "name": "Bleu" },  
    { "name": "Tagg" },  
    { "name": "Bob" },  
    { "name": "Mona" }  
  ],  
  "edges": [  
    { "source": 0, "target": 1 },  
    { "source": 0, "target": 4 },  
    { "source": 0, "target": 5 },  
    { "source": 0, "target": 6 },  
    { "source": 0, "target": 7 },  
    { "source": 1, "target": 2 },  
    { "source": 1, "target": 3 },  
    { "source": 1, "target": 5 },  
    { "source": 1, "target": 8 },  
    { "source": 1, "target": 9 },  
  ]  
}
```

The force-directed layout algorithms in D3.js require the data to be in this format. This needs to be an object with a `nodes` and an `edges` property. The `nodes` property can be an array of any other objects you like to use. These are typically your data items.

The `edges` array must consist of objects with both `source` and `target` properties, and the value for each is the index into the `nodes` array of the source and target nodes. You can add other properties, but we need to supply at least these two.

To start rendering the graph, we load this data and get the main SVG element created:

```
var url = 'https://gist.githubusercontent.com/d3byex/5a8267f90a0d215fcb3e/raw/ba3b2e3065ca8eafb375f01155dc99c569fae66b/uni_network.json';
d3.json(url, function(error, data) {
    var width = 960, height = 500;
    var svg = d3.select('body').append('svg')
        .attr({
            width: width,
            height: height
        });
});
```

The next step is to create the layout for the graph using `d3.layout.force()`. There are many options, several of which we will explore over the course of our examples, but we start with the following:

```
var force = d3.layout.force()
    .nodes(data.nodes)
    .links(data.edges)
    .size([width, height])
    .start();
```

This informs the layout about the location of the nodes and links using the `.node()` and `.link()` functions respectively. The call to `.size()` informs the layout about the area to constrain the layout within and has two effects on the graph: the gravitational center and the initial random position.

The call to `.start()` begins the simulation, and must be called after the layout is created and the nodes and links are assigned. If the nodes and links change later, it can be called again to restart the simulation. Note that the simulation starts after this function returns, not immediately. So, you can still make other changes to the visual.

Now we can render the links and nodes:

```
var edges = svg.selectAll('line')
    .data(data.edges)
    .enter()
    .append('line')
    .style('stroke', '#ccc')
    .style('stroke-width', 1);
```

```
var colors = d3.scale.category20();
var nodes = svg
  .selectAll('circle')
  .data(data.nodes)
  .enter()
  .append('circle')
  .attr('r', 10)
  .attr('fill', function(d, i) {
    return colors(i);
})
.call(force.drag);
```

Note that we also chained the `.call()` function passing it a reference to the `force.drag` function of our layout. This function is provided by the layout object to easily allow us a means of dragging the nodes in the network.

There is one more step required. A force layout is a simulation and consists of a sequence of **ticks** that we must handle. Each tick represents that the layout algorithm has passed over the nodes and recalculated their positions, and this gives us the opportunity to reposition the visuals.

To hook into the ticks, we can use the `force.on()` function, telling it that we want to listen to `tick` events, and on each event, call a function to allow us to reposition our visuals. The following is our function for this activity:

```
force.on('tick', function() {
  edges.attr({
    x1: function(d) { return d.source.x; },
    y1: function(d) { return d.source.y; },
    x2: function(d) { return d.target.x; },
    y2: function(d) { return d.target.y; }
  });

  nodes.attr('cx', function(d) { return d.x; })
    .attr('cy', function(d) { return d.y; });
});
```

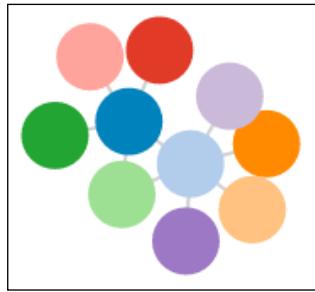
On each tick, we need to reposition each node and edge appropriately. Notice how we are doing this. D3.js has added to our data `x` and a `y` properties, which are the calculated position. It also has added a `px` and `py` property to each data node, which represents the previous `x` and `y` position.



You can also use `start` and `end` as parameters of the `on()` method to trap when the simulation begins and completes.



On running this, the output will be similar to the following:



Every time this example is executed, the nodes will finish in a different position. This is due to the algorithm specifying a random start position for each node.

The nodes are very close in this example, to the point where the links are almost not visible. But it is possible to drag the nodes with the mouse, which will expose the links. Also notice that the layout is executed while you drag and the nodes snap back to the middle when the dragged node is released.

Using link distance to spread out the nodes

These nodes in the previous example are a little too close together and we have a hard time seeing the edges. To add more distance between the nodes, we can specify a link distance. This is demonstrated by the following example:



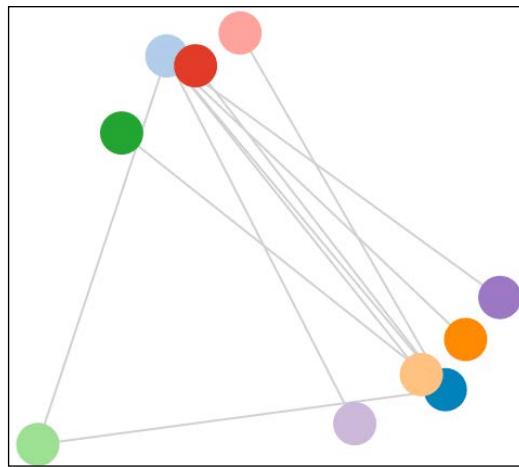
bl.ock (11.2): <http://goo.gl/dd1T3O>



The one modification this example makes to the previous one is that it increases the link distance to 200 (the default is 20):

```
var force = d3.layout.force()
  .nodes(data.nodes)
  .links(data.edges)
  .size([width, height])
  .linkDistance(200)
  .start();
```

This modification results in some better spacing of the nodes at the end of the simulation:



Drag the nodes around. It will demonstrate some of the physics in play:

- No matter where you move any node(s), the graph returns to the center of the visualization. This is the effect of gravity on the layout and of it being placed in the center.
- The nodes always come together, but are always at least the link distance apart. The gravity attracts them to the center and the default charge, which is -30, makes the nodes push away from each other, but not enough to stretch the links much or make the nodes escape the center of gravity.
- The preceding point has an important ramification in the result of the visualization. The links between nodes will generally cross each other. In many network visualizations, it is desirable to try and make the links not cross each other, as it simplifies the ability to follow the links, and hence, the relationships. We will examine how to fix this in the next example.

Adding repulsion to nodes for preventing crossed links

The means by which we attempt to prevent crossing links is to apply an amount of repulsion to each of the nodes. When the amount of repulsion exceeds the pull of the center of gravity, the nodes can move away from this point. They will also move away from the other nodes, tending to expand the result graph out to a maximum size, with the effect of causing the links to not cross.

The following example demonstrates node repulsion:

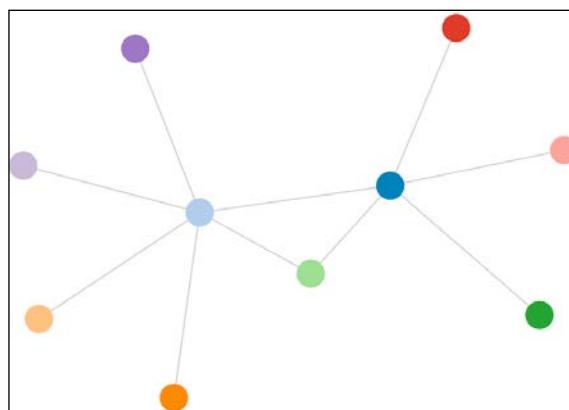


This example makes two modifications to the previous example:

```
var force = d3.layout.force()  
  .nodes(data.nodes)  
  .links(data.edges)  
  .size([width, height])  
  .linkDistance(1)  
  .charge(-5000)  
  .start();
```

This creates a charge with a value of `-5000`, meaning that the nodes actually repulse each other. There is also a smaller link distance, as the repulsion will push the nodes apart quite a bit, therefore stretching the links. Leaving the links at `200` would make the links very long.

When this simulation completes, you will have a graph that looks like the following:

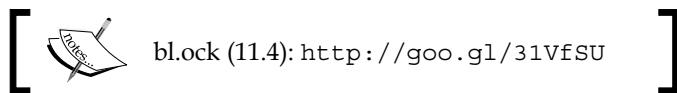


Notice how the nodes now tried to get as far away from each other as possible! The links were stretched quite a bit too, even though the link distance is set to 1. Links are, by default, elastic and will be stretched or compressed based on the charges and gravity in the system.

Rerun this simulation again and again. You will notice that it almost always converges to this same shape with the nodes in the same relative places in the graph (the group itself will likely be rotated a different amount each time). In a really rare case, there may still be a crossed edge, but the repulsion is set high enough to prevent this for most executions.

Labelling the nodes

Something that has been missing in our force-directed graphs is labelling of the nodes so that we can tell what data the nodes represent. The following example demonstrates how to add labels to the nodes:



The difference in this preceding example is that instead of representing a node by a single circle SVG element, we represent it by a group which contains both a circle and a text element:

```
var nodes = svg.selectAll('g')
  .data(data.nodes)
  .enter()
  .append('g')
  .call(force.drag);

var colors = d3.scale.category20();
nodes.append('circle')
  .attr('r', 10)
  .attr('fill', function (d, i) {
    return colors(i);
})
  .call(force.drag);

nodes.append('text')
  .attr({
    dx: 12,
    dy: '.35em',
```

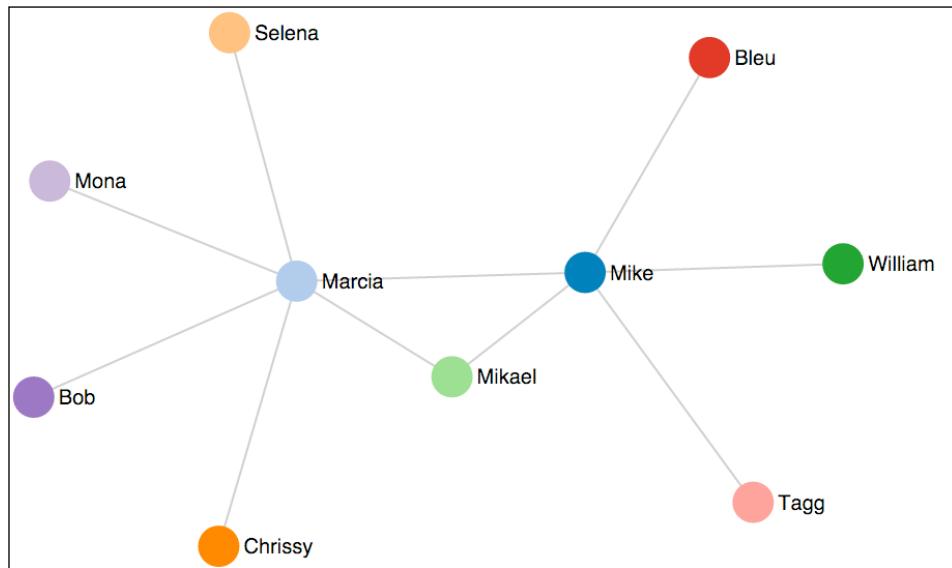
```
        'pointer-events': 'none'
    })
.style('font', '10px sans-serif')
.text(function (d) { return d.name });
```

Then we need one more change during the processing of the tick event. Since we now need to position an SVG group instead of a circle, this code needs to translate the group into position instead of using the x and y properties:

```
force.on('tick', function () {
    edges.each(function (d) {
        d3.select(this).attr({
            x1: d.source.x,
            y1: d.source.y,
            x2: d.target.x,
            y2: d.target.y
        });
    });

    nodes.attr('transform', function (d) {
        return 'translate(' + d.x + ',' + d.y + ')';
    });
});
```

The result of this example now looks like the following:



Making nodes stick in place

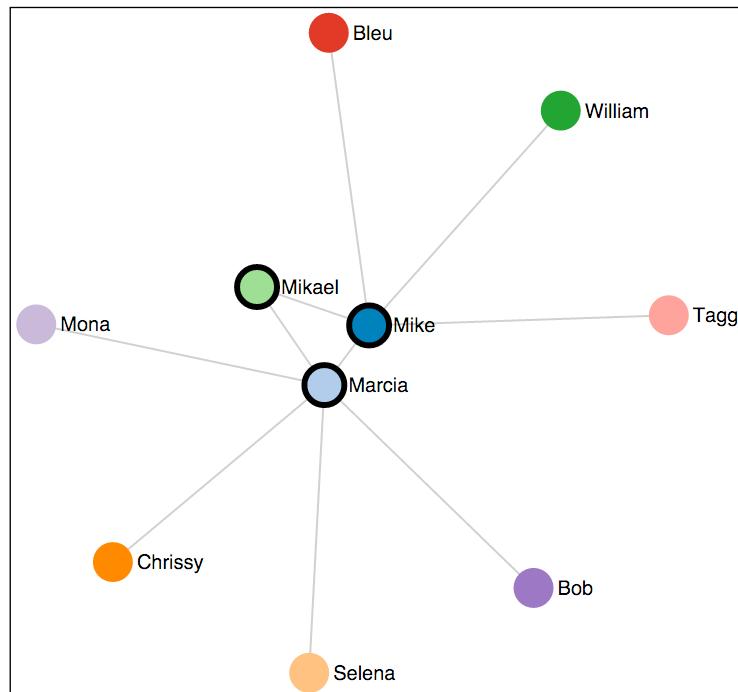
A common—and frustrating—issue when examining nodes in a force network is that when you move one node of a clump of other nodes to see it better and then let it go, it goes back to where it was. I'll bet you've experienced this madness already just while using these examples.

This can be remedied by using a concept known as making the nodes sticky. The following example demonstrates this in operation:



Now, when you drag a node, it will stay where you leave it. Nodes that are fixed in place will change to have a thick black border. To release a node, double click it and it will be put back into the force layout.

The following image shows this with three nodes fixed in place:



Now let's examine the modifications needed to make this work. This works by adding a few function chains to our code to create the circles:

```
nodes.append('circle')
    .attr('r', 10)
    .attr({
        r: 10,
        fill: function(d, i) {
            return colors(i);
        },
        stroke: 'black',
        'stroke-width': 0
    })
    .call(force.drag()
        .on("dragstart", function(d) {
            d.fixed = true;
            d3.select(this).attr('stroke-width', 3);
        })
        .on('dblclick', function(d) {
            d.fixed = false;
            d3.select(this).attr('stroke-width', 0);
        }));
}
```

When the circle is first created, in addition to having its fill color specified, it will also have a stroke color of black but of width 0.

Then, instead of using `.call(force.drag)`, we replace that with a custom drag implementation. At the start of the drag, the code sets a property, `fixed`, on the data object to `true`. If the force layout object sees that the object has this property, and its value is `true`, then it will not attempt to reposition the item. And then, the border is set to be three pixels in width.

The last modification is to handle the `dblclick` mouse event, which will set the `fixed` property to `false`, releasing the node to be part of the layout and then hiding the thick border.

Adding directionality markers and style to the links

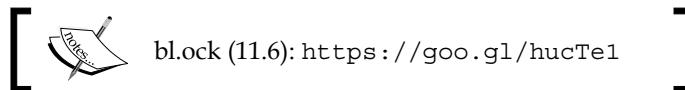
Relationships between a node can be one-way or bi-directional. The code we have written so far assumed one-way, or perhaps, non-directional. Let's now look at how we can express the direction in the relationship by adding arrow heads to the lines.

The example we will create will assume that each entry in the edges collection of the data represents a one-way link from the source to the target. If there is a bi-directional link, there will be an additional entry in edges with the source and target reversed.

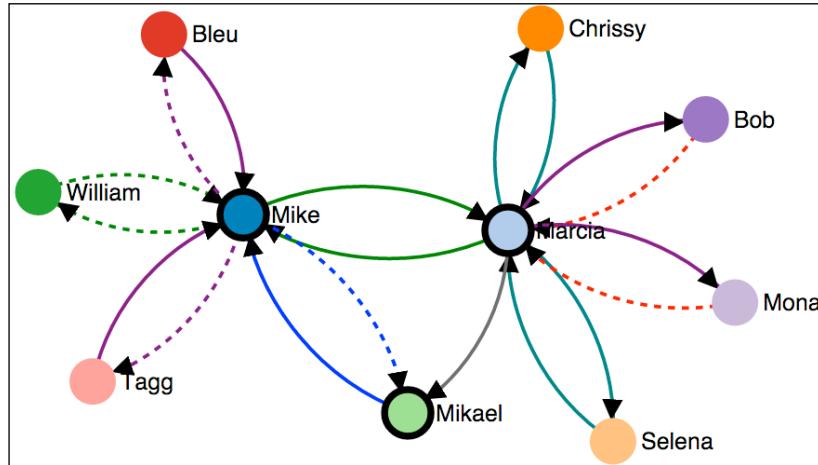
The example will use the data from https://gist.githubusercontent.com/d3byex/5a8267f90a0d215fcb3e/raw/8469d2a7da14c1c8180ebb2ea8ddf1e2944f990c/multi_network.html, which has several bi-directional links added as well as a type property to specify the type of the relationship.

The edges collection in this data is the following. The nodes have not changed:

```
"edges": [
  { "source": 0, "target": 1, "type": "spouse" },
  { "source": 1, "target": 0, "type": "spouse" },
  { "source": 0, "target": 4, "type": "coworker" },
  { "source": 4, "target": 0, "type": "coworker" },
  { "source": 0, "target": 5, "type": "father" },
  { "source": 5, "target": 0, "type": "son" },
  { "source": 0, "target": 6, "type": "master" },
  { "source": 6, "target": 0, "type": "pet" },
  { "source": 0, "target": 7, "type": "master" },
  { "source": 1, "target": 2, "type": "spouse" },
  { "source": 1, "target": 3, "type": "friend" },
  { "source": 1, "target": 5, "type": "mother" },
  { "source": 1, "target": 8, "type": "pet" },
  { "source": 8, "target": 1, "type": "master" },
  { "source": 1, "target": 9, "type": "pet" },
  { "source": 5, "target": 10, "type": "pet" }
]
```



The following image depicts the result of this example:



Let's see how the code goes about creating this visualization.

The first thing that is changed in this example is that it uses styles to color the different types of links:

```
.link {  
    fill: none;  
    stroke: #666;  
    stroke-width: 1.5px;  
}  
  
.link.spouse {  
    stroke: green;  
}  
  
.link.son {  
    stroke: blue;  
}  
  
.link.father {  
    stroke: blue;  
    stroke-dasharray: 0, 2, 1;  
}  
  
.link.friend {
```

```
        stroke: teal;
    }

.link.pet {
    stroke: purple;
}

.link.master {
    stroke: purple;
    stroke-dasharray: 0, 2, 1;
}

.link.ruler {
    stroke: red;
    stroke-dasharray: 0, 2, 1;
}

.link.coworker {
    stroke: green;
    stroke-dasharray: 0, 2, 1;
}
```

The code to load the data and to set up the SVG element and the force layout is the same as the last example. The other difference is that the code needs to determine the specific link types as they will be used for markers and styles:

```
var linkTypes = d3.set(data.edges.map(function (d) {
    return d.type;
})).values();
```

Next, there are markers created for each of the link types. These will render a curved path with an arrow head on each end, created by the last chained function to set the `d` attribute:

```
svg.append("defs")
    .selectAll("marker")
    .data(linkTypes)
    .enter()
    .append("marker")
    .attr({
        id: function (d) { return d; },
        viewBox: "0 -5 10 10",
        refX: 15,
```

```
    refY: -1.5,
    markerWidth: 6,
    markerHeight: 6,
    orient: "auto"
})
.append("path")
.attr("d", "M0,-5L10,0L0,5");
```

The next step is to create the edges:

```
var edges = svg.append("g")
.selectAll("path")
.data(force.links())
.enter()
.append("path")
.attr("class", function (d) {
    return "link " + d.type;
})
.attr("marker-end", function(d) {
    return "url(#" + d.type + ")";
});
```

Instead of using a line, the code now uses a path. The `d` property of the path is not specified at this time. It will be set at every tick of the simulation. This path references one of the styles by using the type as part of the class name, and the `marker-end` attributes specifies which marker definition to use for this segment.

The circles are created in the same manner as the previous example, and so is the text. The last change is that the tick handler is modified to not only reposition the nodes, but to also regenerate paths based on arcs:

```
force.on("tick", function () {
    edges.attr("d", function (d) {
        var dx = d.target.x - d.source.x,
            dy = d.target.y - d.source.y,
            dr = Math.sqrt(dx * dx + dy * dy);
        return "M" + d.source.x + "," + d.source.y + "A" +
            dr + "," + dr + " 0 0,1 " +
            d.target.x + "," + d.target.y;
    });
    nodes.attr("transform", function (d) {
        return "translate(" + d.x + "," + d.y + ")";
    });
});
```

Summary

In this chapter, we explained how to use D3.js for generating force-directed graphs. These types of graphs are some of the most interesting types of graphs and can be used to visualize large sets of interconnected data such as social networks.

The chapter started by going over the basic concepts of creating a graph, stepping through an example that progressively refined the graph, while making the effort to demonstrate how several of the parameters effect the result of the graph.

We then covered several techniques for enhancing and making the graphs more usable. These included labeling nodes with text, replacing nodes with images, and styling links to show direction and type.

In the next chapter, we will cover using D3.js for creating maps. We will also learn quite a bit about GeoJSON and TopoJSON, both of which, when combined with D3.js, allow us to create complex visuals based on geographic data.

12

Creating Maps with GeoJSON and TopoJSON

D3.js provides extensive capabilities for creating maps and to facilitate you in presenting data as part of the map or as an overlay. The functions for mapping within D3.js leverage a data format known as GeoJSON, a form of JSON that encodes geographic information.

Another common type of data for maps in D3.js is TopoJSON. TopoJSON is a more compressed form of GeoJSON. Both these formats are used to represent the cartographic information required to create a map, and D3.js processes this data and performs its usual magic of converting this information into SVG paths that visualize the map.

This chapter will start with a brief overview of GeoJSON and TopoJSON. This will give you the foundation to understand how maps are represented and rendered with D3.js. We will then jump into many examples using both data formats for rendering maps of various types, coloring the geometries within the map based upon data, and for overlaying information at specific locations on those maps.

The specific topics that we will cover in this chapter include:

- A brief overview of TopoJSON and GeoJSON
- Drawing a map of the United States with GeoJSON
- Using TopoJSON to draw the countries of the world
- Styling the geometries that comprise a map
- Panning and zooming of a map

- Interaction with a globe
- Highlighting the boundaries of geometries on mouseover events
- Adding symbols to a map at specific locations
- Rendering maps of regions based upon data (using a choropleth)

Introducing TopoJSON and GeoJSON

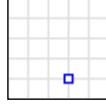
Almost every map example in D3.js will use either **GeoJSON** or **TopoJSON**.

GeoJSON is an open, standard, JSON-based format for representing basic geographical features as well as the non-spatial properties for those features (such as the name of a city or a landmark).

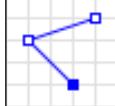
The core geometries in GeoJSON are points, line strings, and polygons. The basic description of a GeoJSON entity uses the following syntax:

```
{  
  "type": name of the type of geometry (point, line string, ...)  
  "coordinates": one or more tuple of latitude / longitude  
}
```

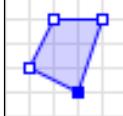
Let's take a look at the four basic types of geometry types available in GeoJSON. A **point** represents a position in two-dimensional space, and consists of a pair of one latitude and longitude. A point is normally used to specify the location of a feature on a map (such as a building):

Example	Representative GeoJSON
	{ "type": "Point", "coordinates": [30, 10] }

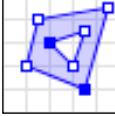
LineString describes a sequence of points which have a line drawn between them, starting at the first, through all intermediate points, and ending at the last coordinate. The name conjures up visions of stretching a string caught between all the points. These shapes are normally used to represent items such as, roads or rivers:

Example	Representative GeoJSON
	{ "type": "LineString", "coordinates": [[30, 10], [10, 30], [40, 40]]}

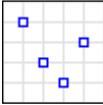
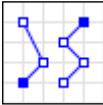
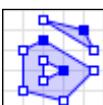
A **polygon** is a closed shape normally consisting of three or more points, where the last point is the same as the first and forms a closed shape. The JSON representation is shown as follows; note that the coordinates are an array of arrays of tuples:

Example	Representative GeoJSON
	{ "type": "Polygon", "coordinates": [[[30, 10], [40, 40], [20, 40], [10, 20], [30, 10]]]}

The purpose of an array of arrays of tuples is to allow multiple polygons to be defined, which exclude each other, thereby allowing the exclusions of one or more polygonal regions within one another:

Example	Representative GeoJSON
	{ "type": "Polygon", "coordinates": [[[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]], [[20, 30], [35, 35], [30, 20], [20, 30]]]}

It is possible to define multi-part geometries where a particular geometry type is reused, and where the coordinates describe multiple instances of the type of geometry. These types are the previous types prefaced with *Multi*—`MultiPoint`, `MultiLineString`, and `MultiPolygon`. Each is demonstrated as follows:

Type	Example	Representative GeoJSON
<code>MultiPoint</code>		{ "type": "MultiPoint", "coordinates": [[10, 40], [40, 30], [20, 20], [30, 10]] }
<code>MultiLineString</code>		{ "type": "MultiLineString", "coordinates": [[[10, 10], [20, 20], [10, 40]], [[40, 40], [30, 30], [40, 20], [30, 10]]] }
<code>MultiPolygon</code>		{ "type": "MultiPolygon", "coordinates": [[[[40, 40], [20, 45], [45, 30], [40, 40]]], [[[20, 35], [10, 30], [10, 10], [30, 5], [45, 20], [20, 35]], [[30, 20], [20, 15], [20, 25], [30, 20]]]] }

These basic geometries can be wrapped within a **feature**. A feature contains a geometry and also a set of properties. As an example, the following defines a feature which consists of a point geometry, and which has a single property, name, which can be used to describe a name for that feature:

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [46.862633, -114.011593]
  },
  "properties": {
    "name": "Missoula"
  }
}
```

We can go up one more level in the hierarchy, and define what is known as a **feature collection**:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {"type": "Point",
      "coordinates": [102.0, 0.5]},
      "properties": {"prop0": "value0"} },
    {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]]
        ],
      "properties": { "prop0": "value0", "prop1": 0.0 }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
            [100.0, 1.0], [100.0, 0.0] ] ]
        ],
      "properties": {
        "prop0": "value0", "prop1": {"this": "that"} }
    }
  ]
}
```

By combining geometries, features, and feature collections, it is possible to describe very complex shapes such as maps.

But one of the problems with GeoJSON is that it is very verbose, and particular geometries and features cannot be reused. If the same geometry is required in multiple locations, it must be completely specified a second time.

To help fix this situation, TopoJSON was created. TopoJSON provides additional constructs for the encoding of topology and reuse. Instead of discretely describing each geometry, TopoJSON allows you to define geometries, and then stitch them together using concepts known as **arcs**.

Arcs allows TopoJSON to eliminate redundancy, and to provide a much more compact representation as compared to GeoJSON. It is stated that TopoJSON can commonly provide 80 percent compression over GeoJSON. With every millisecond of the download time of a web page being important, this can be significant for user experience when using large sets of geometry.

A full explanation of TopoJSON is a bit beyond the scope of this book, but to briefly demonstrate it, we can look at the following and briefly examine its content:

```
{  
  "type": "Topology",  
  "objects": {  
    "example": {  
      "type": "GeometryCollection",  
      "geometries": [  
        { "type": "Point",  
          "properties": {  
            "prop0": "value0" }  
          "coordinates": [102, 0.5]  
        },  
        { "type": "LineString",  
          "properties": {  
            "prop0": "value0",  
            "prop1": 0 },  
          "arcs": [0]  
        },  
        { "type": "Polygon",  
          "properties": {  
            "prop0": "value0",  
            "prop1": {  
              "this": "that"  
            }  
          },  
        }  
      ]  
    }  
  }  
}
```

```

        "arcs": [[-2]]
    }
]
}
},
"arcs": [
  [[102, 0], [103, 1], [104, 0], [105, 1]],
  [[100, 0], [101, 0], [101, 1], [100, 1], [100, 0]]
]
}

```

This TopoJSON object has three properties: `type`, `objects`, and `arcs`. The value of `type` is always "topology". The `objects` property consists of a geometry collection similar to those in GeoJSON, with the difference that instead of specifying coordinates, the object can, instead, specify one or more arcs.

Arcs are the big difference in TopoJSON versus GeoJSON, and represent the means of reuse. The `arcs` property provides an array of arrays of positions, where a position is essentially a coordinate.

These arcs are referenced by geometries of 0-based array semantics. Hence, the `LineString` geometry in the preceding code is referencing the first arc in the `topology` object by specifying `arcs[0]`.

The polygon object is referencing an arc with value -2. A negative arc value specifies that the one's complement of the arc that should be utilized. This essentially infers that the positions in the arc should be reversed. Therefore, -2 instructs to get the reversed position of the second arc. This is one of the strategies that TopoJSON uses to reuse and compress data.

There are other options, such as transforms and bounding boxes, and other rules. For a more detailed specification, please see <https://github.com/mbostock/topojson-specification>.



An important thing to note about TopoJSON is that D3.js itself only uses GeoJSON data. To use data in the TopoJSON format, you will need to use the TopoJSON plugin available at <https://github.com/mbostock/topojson>. This plugin will convert TopoJSON into GeoJSON that can be used by D3.js functions, thereby affording the capabilities of TopoJSON to your D3.js application.

Creating a map of the United States

Our first examples will examine creating a map of the United States. We will start with an example that loads the data and gets the map rendered, and then we will examine styling the map to make it more visible, followed by examples of modifying the projection used to render the content more effectively.

Creating our first map of the United States with GeoJSON

Our first map will render the United States. We will use a GeoJSON data file, `us-states.json`, available at <https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb56e36/us-states.json>. The following are the first few lines of this file, and demonstrate how the shapes of the states are organized within the file:

```
{"type": "FeatureCollection", "features": [
  { "type": "Feature",
    "id": "01",
    "properties": { "name": "Alabama" },
    "geometry": {
      "type": "Polygon",
      "coordinates": [ [
        [ -87.359296, 35.00118 ], [ -85.606675, 34.984749 ],
        [ -85.431413, 34.124869 ], [ -85.184951, 32.859696 ],
        [ -85.069935, 32.580372 ], [ -84.960397, 32.421541 ],
        [ -85.004212, 32.322956 ], [ -84.889196, 32.262709 ],
        ...
      ]
    }
  }
]}
```

`FeatureCollection` at the top level consists of an array of features, each element of which is a state (or territory) as well as Washington D.C. Each state is a feature, has a single property `Name`, and a polygon geometry representing the outline of the state expressed in latitude and longitude tuples.

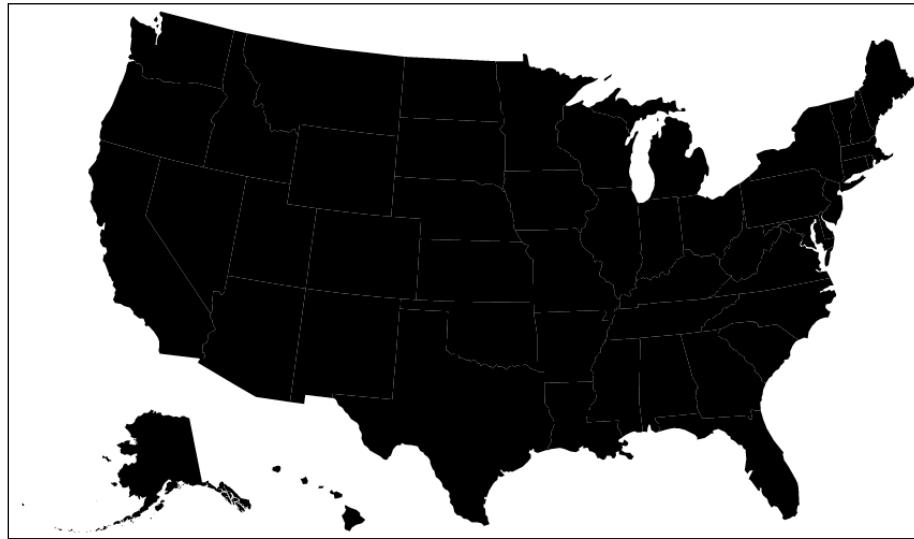
The code for the example is available at the following link:



bl.ock (12.1): <http://goo.gl/dzKsVd>



On opening the URL, you will see the following map:



The code required to take this data and render a map is sublimely simple (by design). It begins by creating the main SVG element:

```
var width = 950, height = 500;
var svg = d3.select('body')
    .append('svg')
    .attr({
        width: width,
        height: height
   });
```

GeoJSON is simply JSON and can be loaded with `d3.json()`:

```
var url = 'https://gist.githubusercontent.com/
d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb5
6e36/us-states.json';
d3.json(url, function (error, data) {
    var path = d3.geo.path();
    svg.selectAll('path')
        .data(data.features)
        .enter()
        .append('path')
        .attr('d', path);
});
d3.json("/data/us-states.json", function (error, data) {
```

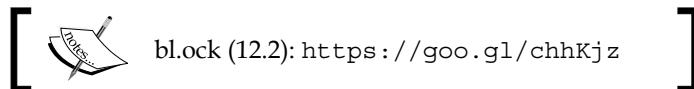
Once we have the data, we can then create a `d3.geo.path()`. This object has the smarts for taking the features in the GeoJSON and converting them into an SVG path. The code then adds a path to the main SVG element, binds the data, and sets the `d` property of the path to our `d3.geo.path()` object.

Wow, with just a few lines of code, we have drawn a map of the United States!

Styling the map of the United States

Overall, this image is dark, and the borders between the states are not particularly visible. We can change this by providing a style for the fill and stroke values used to render the map.

The code for this example is available at the following link:



When opening this URL, you will see the following map:



The only change to the previous example is to set the fill to transparent, and the borders to black:

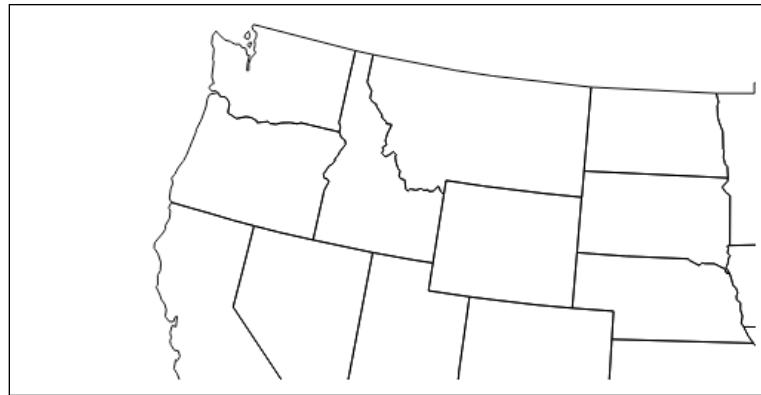
```
svg.selectAll('path')
  .data(data.features)
  .enter()
  .append('path')
  .attr('d', path)
  .style({ fill: 'none', stroke: 'black' });
```

Using the albersUsa projection

You may have a few questions about the map in the previous two examples. First, how is the map scaled to the size of the SVG element? Second, can I change this scale? And why are Alaska and Hawaii drawn down where Mexico would normally be?

These are related to some underlying assumptions about a **projection**. A projection is a way of taking geographic data, which is 2D data (latitude and longitude), but which is really on a three dimensional sphere (the earth), and rendering it onto a 2D surface with specific dimensions (your computer screen or viewport in the browser).

In this example, D3.js made some implicit assumptions on these factors. To help exemplify these assumptions, suppose we change the SVG element to be of size 500 x 250. When running this, we get the following output:



The code that creates this is available at the following location. The only change from the previous example is that the height and width of the SVG element have each been halved:



block (12.3): <http://goo.gl/41wyCY>



The result is that the actual rendering is the same size, and we have clipped the lower and rightmost three-quarters of the map due to the smaller container.

Why is this? It is because, by default, D3.js uses a projection known as an **albersUsa** projection, which has a number of assumptions that come with it:

- The dimensions of the resulting map are 1024 x 728
- The map is centered at half of the width and height (512, 364)
- The projection also places Alaska and Hawaii in the lower-left side of the map (aha!)

To change these assumptions, we can create our own **albersUsa** projection using a `d3.geo.albersUsa()` projection object. This object can be used to specify both a translation and scaling of the rendering of the results.

The following example creates an **albersUsa** projection and centers the map:



block (12.4): <http://goo.gl/1e4DGp>



With the following result:



The code creates a `d3.geo.albersUsa` projection, and tells it to center the map of the United States at `[width/2, height/2]`:

```
var projection = d3.geo.albersUsa()  
    .translate([width / 2, height / 2]);
```

The projection object then needs to be assigned to the `d3.geo.path()` object using its `.projection()` function:

```
var path = d3.geo.path()  
    .projection(projection);
```

We have translated the center of the map, but the scale is still the same size. To change the scale, we use the projection's `.scale()` function. The following example sets the scale to the width, telling D3.js that the width of the map should not be 1024, but the value of `width` and `height`:



The preceding example results in a properly scaled map:



The only difference in the code is the call to `.scale()` on the projection:

```
var projection = d3.geo.albersUsa()  
    .translate([width / 2, height / 2])  
    .scale([width]);
```

Note that we only pass a single value to `scale`. The projection scales along the width, and then automatically and proportionately along the height.

Creating a flat map of the world

The `albersUsa` projection is one of many D3.js supplied projection objects. You can see the full list of these projections at <https://github.com/mbostock/d3/wiki/Geo-Projections>.

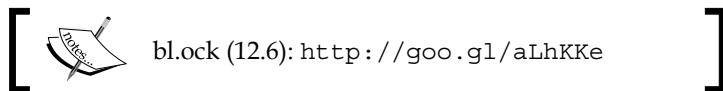
We don't have space to demonstrate all of these in this book, but a few are worth the effort to demonstrate a couple of TopoJSON concepts. Specifically, we will demonstrate the rendering of a map of the countries of the world, sourced from TopoJSON, and projected onto both flat and spherical surfaces.

For data in these examples, will use the `world-110m.json` data file provided with the TopoJSON data library source code available at <https://gist.github.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb56e36/world-110m.json>.

This data represents country data with features, specified at a 110-meter resolution.

Loading and rendering with TopoJSON

Now let's examine loading and rendering of TopoJSON. The following example demonstrates the process:



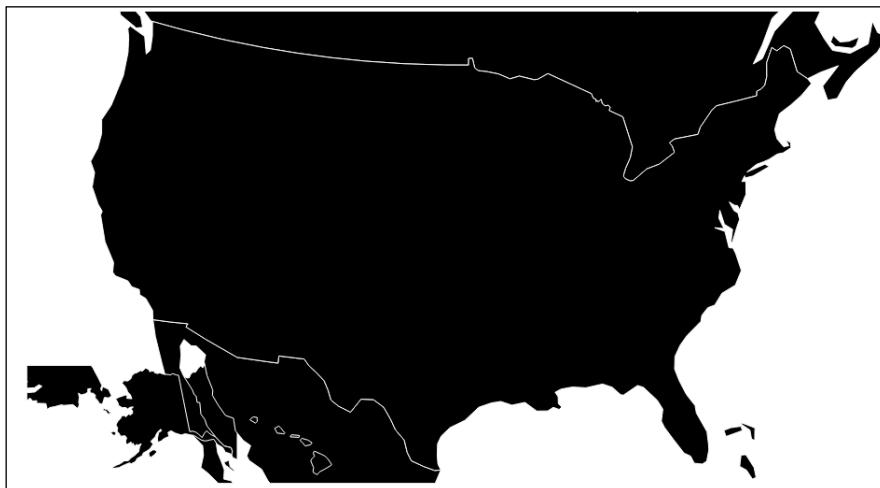
The code does not vary much from the previous example. The change comes after the data is loaded:

```
var path = d3.geo.path();
var countries = topojson.feature(world,
    world.objects.countries).features;
svg.selectAll('path')
    .data(countries)
    .enter()
    .append('path')
    .attr('d', path)
    .style({
        fill: 'black',
        stroke: 'white'
    });
}
```

The example still uses a `d3.geo.path()` object, but this object cannot directly be given the TopoJSON. What needs to be done is to first extract the portion of this data that represents the countries, which is done by calling the `topojson.feature()` function.

The `topojson` variable is globally declared in the `topojson.js` file. Its `.feature()` function, when given a TopoJSON object (in this case, `world`), and a `GeometryCollection` (in this case, `world.objects.countries`), returns a GeoJSON feature that can be used by a path.

The selection to render the map then binds to this result, giving us the following map:

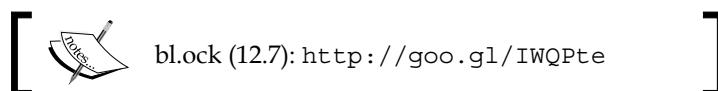


Whoops! That's not what we expected (but as we will see, it is exactly what we coded). Why is everything globed together? It is because we are still using the default projection, a `d3.geo.albersUsa()` projection.

Creating a map of the world using a Mercator projection

To fix this, we simply need to create a Mercator projection object, and apply it to the path. This is a well known projection that renders the map of the globe in a rectangular area.

The process is demonstrated in the following example:



bl.ock (12.7): <http://goo.gl/IWQPte>

The only difference in this code is the setup of the path to use a Mercator projection object:

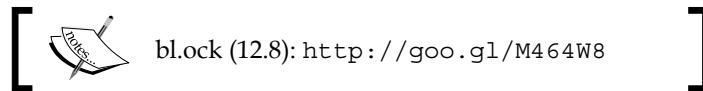
```
var projection = d3.geo.mercator()  
    .scale((width + 1) / 2 / Math.PI)  
    .translate([width / 2, height / 2]);  
var path = d3.geo.path().projection(projection);
```

We need to give the projection object a little information about the width and height of our rendering, and the resulting map is now the following, which looks a lot more like the familiar world map:



Creating spherical maps with orthographic projection

Now let's change our projection to an **orthographic** projection. This projection maps data on to a simulated sphere. This is demonstrated by the following example:



This example simply changes the previous one by using a `d3.geo.orthographic()` projection object:

```
var projection = d3.geo.orthographic();  
var path = d3.geo.path().projection(projection);
```

The preceding example code gives us this beautiful rendering of the planet:



If you examine this closely, you will notice that it is not quite perfect. Notice that Australia seems to be colliding with Africa and Madagascar, and New Zealand is seen in the South Atlantic ocean.

This is because this projection renders through all 360 degrees of the globe, and we are essentially seeing through a clear globe to the backside of the land masses on the far side.

To fix this, we can use the `.clipAngle()` function of the Mercator projection. The parameter is the number of degrees around the center point to which the landmasses should be rendered.

The following example demonstrates this in action:



This changes one line of code:

```
var projection = d3.geo.orthographic()  
    .clipAngle(90);
```

And gives us the following result:



It may not be apparent in the image provided in the book, but this image of the globe on the web page is fairly small. We can change the scaling of the rendering using the `.scale()` function of the projection. The default value for scale is 150, and the corresponding values will make the rendering larger or smaller.

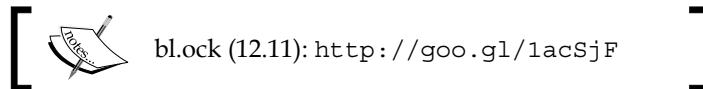
The following example makes the globe twice as large along with setting the center of the globe to not be clipped by the SVG container:



```
var projection = d3.geo.orthographic()
  .scale(300)
  .clipAngle(90)
  .translate([width / 2, height / 2]);
```

This orthographic projection, by default, centers the view on the globe at latitude and longitude (0,0). If we want to center on another location, we need to `.rotate()` the projection by a number of degrees of latitude and longitude.

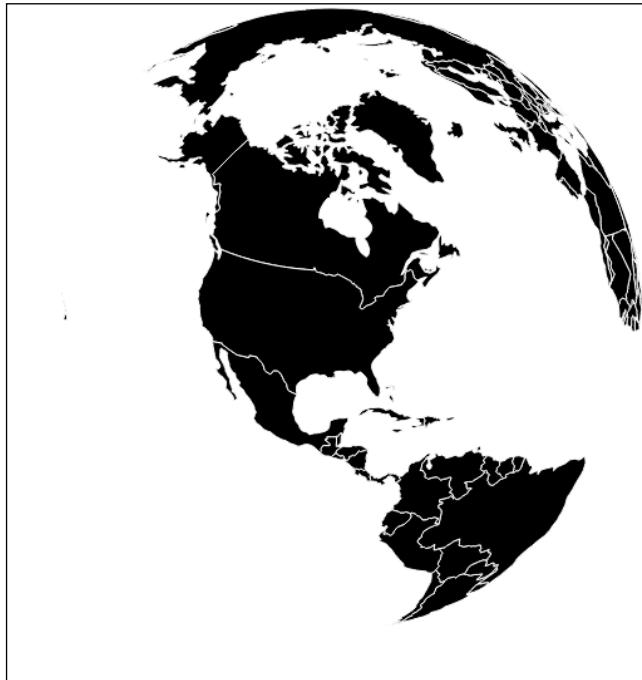
The following example rotates the globe to show the United States prominently:



The one change to the projection is the following:

```
var projection = d3.geo.orthographic()
  .scale(300)
  .clipAngle(90)
  .translate([width / 2, height / 2])
  .rotate([90, -40]);
```

This change in the projection gives us the following result:



Spicing up a globe

Although this globe is quite impressive for the amount of code used to create it, it feels a little dull. Let's differentiate the countries a little more, and also add the lines of latitude and longitude.

Coloring the countries on a globe

We can color the countries on the globe using a `d3.scale.category20()` color scale. But we can't simply rotate through the colors, as there will be cases where adjacent countries will be filled with the same color.

To avoid this, we will take advantage of another function of TopoJSON, `topojson.neighbors()`. This function will return, given a set of geometries (like the countries), a data structure that identifies which geometries are adjacent to each other. We can then utilize this data to prevent the potential problem with colors.

The process is demonstrated in the following example:



The projection in this example remains the same. The remainder of the code is changed.

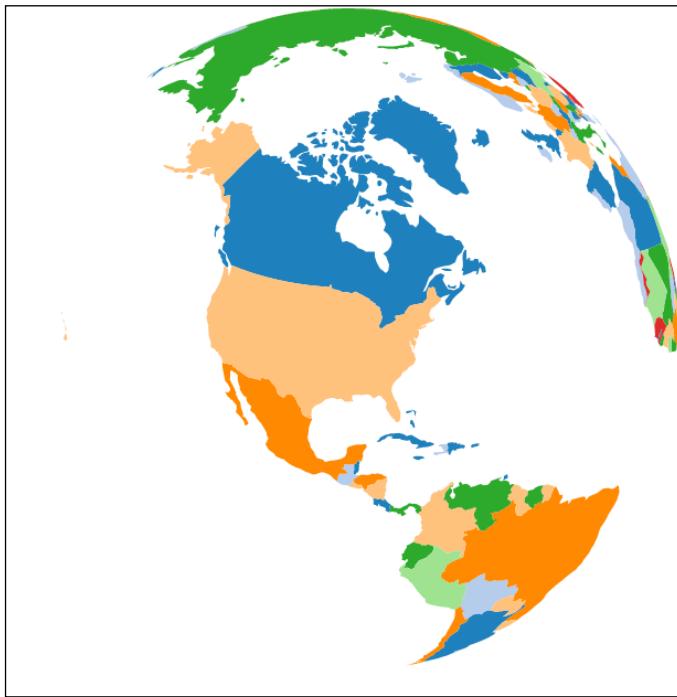
We start by using the same projection as the last example so that code is not repeated here. The following creates the data structure of the colors, the countries, and the neighbors:

```
var color = d3.scale.category20();
var countries = topojson.feature(world,
                                 world.objects.countries).features;
var neighbors = topojson.neighbors(
  world.objects.countries.geometries);
```

The creation of the globe then uses the following statement:

```
var color = d3.scale.category20();
svg.selectAll('.country')
  .data(countries)
  .enter()
  .append('path')
  .attr('d', path)
  .style('fill', function (d, i) {
    return color(d.color = d3.max(neighbors[i]),
      function (n) {
        return countries[n].color;
      })
      + 1 | 0);
});
```

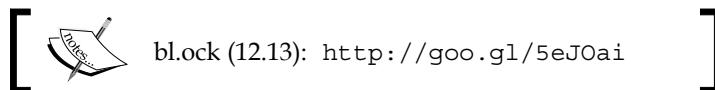
Our resulting globe is the following:



Pretty nice! But it's still lacking in the lines of longitude or latitude, and you can't really tell what the extents of the globe are. Let's fix that now by adding the lines of latitude and longitude.

You'll be really surprised at how easy it is to add the latitudes and longitudes. In D3.js, these are referred to as **graticules**. We create them by instantiating a `d3.geo.graticules()` object, and then by appending a separate path prior to the path for the countries.

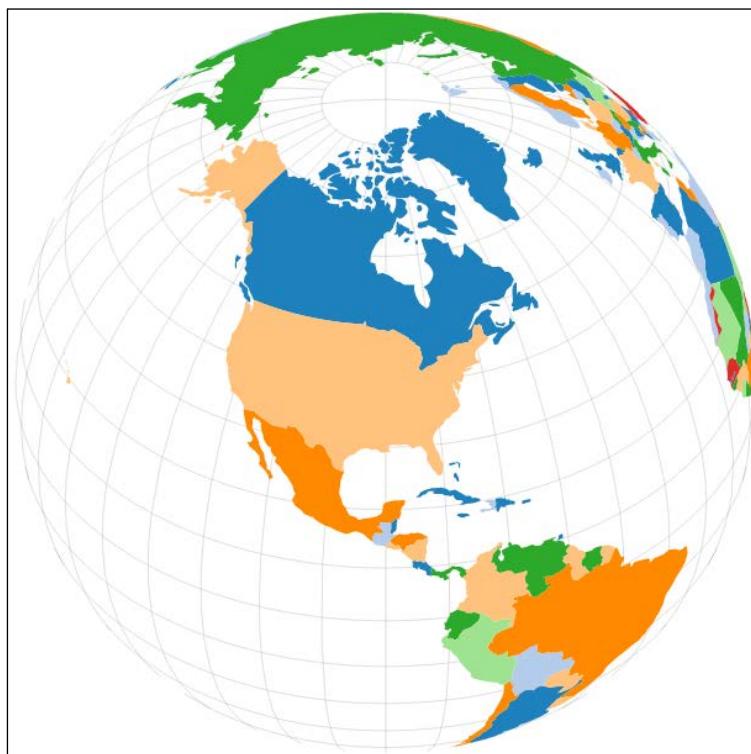
This is demonstrated in the following example:



The only code added to the previous example is the following:

```
var graticule = d3.geo.graticule();
svg.append('path')
  .datum(graticule)
  .attr('d', path)
  .style({
    fill: 'none',
    stroke: '#777',
    'stroke-width': '.5px',
    'stroke-opacity': 0.5
});
```

The change in code results in the following:



Voila! And as they say, easy-peasy!

Adding interactivity to maps

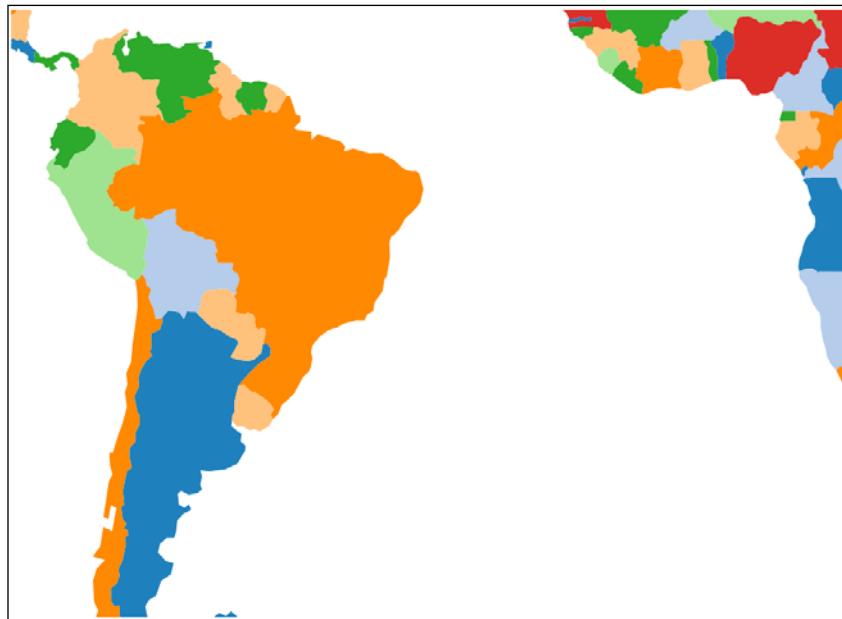
What good is a map if the user is not able to pan and zoom around the map to change the focus, and take a closer look at things? Fortunately, because of D3.js, this becomes very simple to implement. We will look at three different examples of interactivity and maps:

- Panning and zooming a world map
- Highlighting country borders on mouseover
- Rotating a globe with the mouse

Panning and zooming a world map

To demonstrate panning and zooming of a world map, we will make a few modifications to our world Mercator projection example. These modifications will be for using the mouse wheel to zoom in and out, and to be able to drag the map to move it to another center.

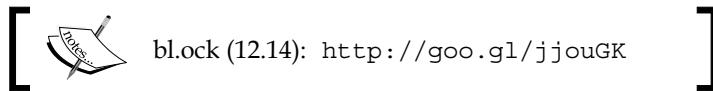
A possible image with this version of the map code could look like the following, which is centered just east of Brazil, and brought up several factors of zoom:



There are a couple of considerations that we should take into account when panning and zooming a map:

- We can only zoom in and out between two extents so that we do not zoom out too far as to lose sight of the map, or too close as to get lost in a single country
- We can only drag the map to a certain extent to ensure that it is constrained and not dragged off some edge

The example is available at the following location:



Much of the code is reused from the Mercator projection example, and also adds the code to uniquely color the countries.

The creation of the main SVG element differs to allow for drag and zoom. This starts with creating a zoom behavior, and assigning it to the main SVG element. Additionally, since we need to zoom the client elements, we add a group to facilitate this action:

```
var zoom = d3.behavior.zoom()
    .scaleExtent([1, 5])
    .on('zoom', moveAndZoom);

var svg = d3.select('body')
    .append('svg')
    .attr({
        width: width,
        height: height
    })
    .call(zoom);
var mainGroup = svg.append('g');
```

The rest of the main part of the code loads the data and renders the map, and is identical to the previous examples.

The `moveAndZoom` function, which will be called on any drag and zoom events, is given as follows:

```
function moveAndZoom() {  
    var t = d3.event.translate;  
    var s = d3.event.scale;  
  
    var x = Math.min(  
        (width / height) * (s - 1),  
        Math.max(width * (1 - s), t[0]));  
  
    var h = height / 4;  
    var y = Math.min(  
        h * (s - 1) + h * s,  
        Math.max(height * (1 - s) - h * s, t[1]));  
  
    mainGroup.attr('transform', 'translate(' + x + ',' + y +  
        ')scale(' + s + ')');  
}
```

From these values, we need to adjust the SVG translate on the map based upon the current mouse position, while taking into account the scale level. We also do not want this to translate the map in any direction such that there is padding between the map and the boundaries; this is handled by combined calls to `Math.min` and `Math.max`.

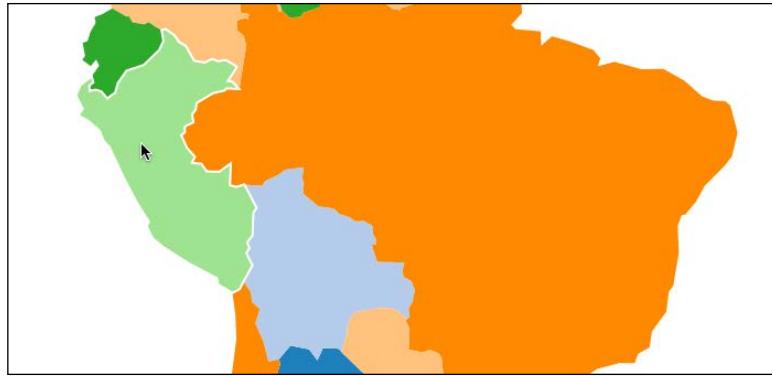
Congratulations, you now have a fully pan and scan map!



Note that as you zoom in, the boundaries on the countries are fairly ragged. This is due to the 110-meter resolution of the data. To have more accurate graphics, use the files with the finer details. Even better, dynamically change to higher resolution data depending upon the zoom level.

Highlighting country borders on mouse hover

Now let's add another interactivity effect to our map: highlighting the border of a country which has the mouse currently over its geometry. This will help us accentuate the country the user is currently examining. A quick demonstration of this is the following, where Peru has a thin white border:



The example is available at the following location:



This is implemented with a few modifications to the previous example. The modifications start with the creation of the top-level group element:

```
mainGroup.style({
  stroke: 'white',
  'stroke-width': 2,
  'stroke-opacity': 0.0
});
```

This code informs D3.js that all SVG elements contained within the group will have a 2-pixel white border, which is initially transparent. When we hover the mouse, we will make this visible on the appropriate geometry.

Now we need to hook up mouse event handlers on each of the path elements that represent countries. On the `mouseover` event, we make the `stroke-opacity` opaque, and set it back to transparent when the mouse exits:

```
mainGroup.selectAll('path')
.on('mouseover', function () {
  d3.select(this).style('stroke-opacity', 1.0);
});
mainGroup.selectAll('path')
.on('mouseout', function () {
  d3.select(this).style('stroke-opacity', 0.0);
});
```

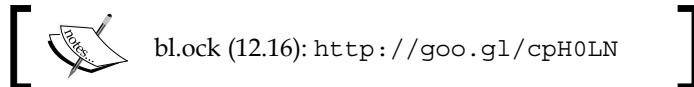
There is one more small change that we will want to make whenever the zoom level changes. As the zoom level goes up, the country borders get disproportionately thick. To prevent this, we can add the following statement to the end of the `moveAndZoom` function:

```
g.style("stroke-width", ((1 / s) * 2) + "px");
```

This is stating that the border of a country should always stay at what is visually 2px thick, no matter what the zoom level.

Rotating a globe using the mouse

Interactivity can also be applied to other projections. We will examine rotating an orthographic globe using the mouse. The example is available at the following location:



To save a little space, we won't show an image here, as it looks the same as the earlier example in the chapter, except that it rotates following the mouse. That, and the rotation effect is lost in a print medium.

But the way this works is very simple. The technique involves creating two scales, one for longitude and the other for latitude. Longitude is calculated as mapping the mouse position from 0 to the width of the graphic to -180 and 180 degrees of longitude. The latitude is a mapping of the vertical mouse position to 90 and -90 degrees:

```
var scaleLongitude = d3.scale.linear()
  .domain([0, width])
  .range([-180, 180]);

var scaleLatitude = d3.scale.linear()
  .domain([0, height])
  .range([90, -90]);
```

When the mouse is moved over the SVG element, we capture it and scale the mouse position into a corresponding latitude and longitude; we then set the rotation of the projection:

```
svg.on('mousemove', function() {
  var p = d3.mouse(this);
  projection.rotate([scaleLongitude(p[0]),
    scaleLatitude(p[1])]);
  svg.selectAll('path').attr('d', path);
}) ;
```

It's a pretty cool little trick of mathematics and scales that allows us to be able to see every position on the entire globe.

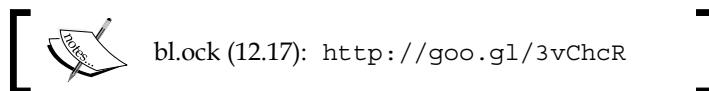
Annotating a map

Our final examples of working with maps will demonstrate making annotations to a map. The first two will demonstrate placing labels and markers on a map, and the third will demonstrate the use of gradient colors to color regions all the way down to a state level.

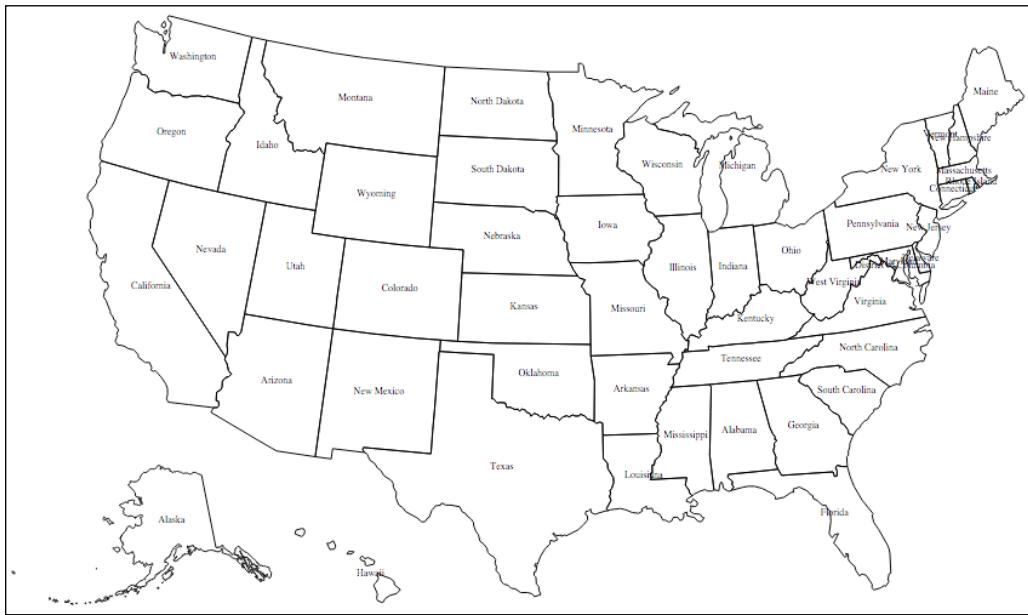
All of these techniques would normally involve some fairly complex math if we had to do it on our own, but thankfully, D3.js again comes to help us solve this with just a few statements.

Labelling states using centroids

The maps of the United States we've created up to this point feel a little lacking in content, as they have not had the names of the states placed over their geometries. It would be very helpful to many reading a map to have the names visible. The example is available at the following location:



The result of the example is the following:



This is actually fairly easy to implement, with only the addition of one statement to our United States Mercator projection example. The following code is placed immediately after the `.selectAll()` statement that creates the boundaries for all the states:

```
svg.selectAll('text')
  .data(data.features)
  .enter()
  .append('text')
  .text(function(d) { return d.properties.name; })
  .attr({
    x: function(d) { return path.centroid(d)[0]; },
    y: function(d) { return path.centroid(d)[1]; },
    'text-anchor': 'middle',
    'font-size': '6pt'
  });

```

This statement creates a text element for each geometric feature in the data file, and sets the text to be the value of the `name` property of the geometry object.

The position of the text uses a function of the path that calculates the **centroid** of the geometry. The centroid is the mathematical center of the geometry, and can be calculated using the `.centroid()` function of a path.

For most states, especially rectangular ones, this works well. For others with irregular shapes, take Michigan for example, the placement is perhaps not optimal for aesthetics. There are various ways to fix this, but those are beyond the scope of this book (a hint: it involves adding additional data to represent location offsets for each geometry).

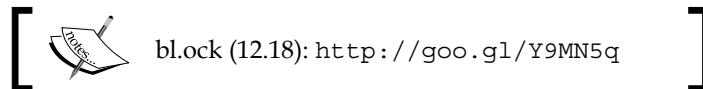
Placing symbols at specific geographic locations

The last example with maps that we will look at will be to place SVG elements on the map at specific coordinates. Specifically, we will place circles at the position of the 50 most populous cities, and size the circle relative to the population.

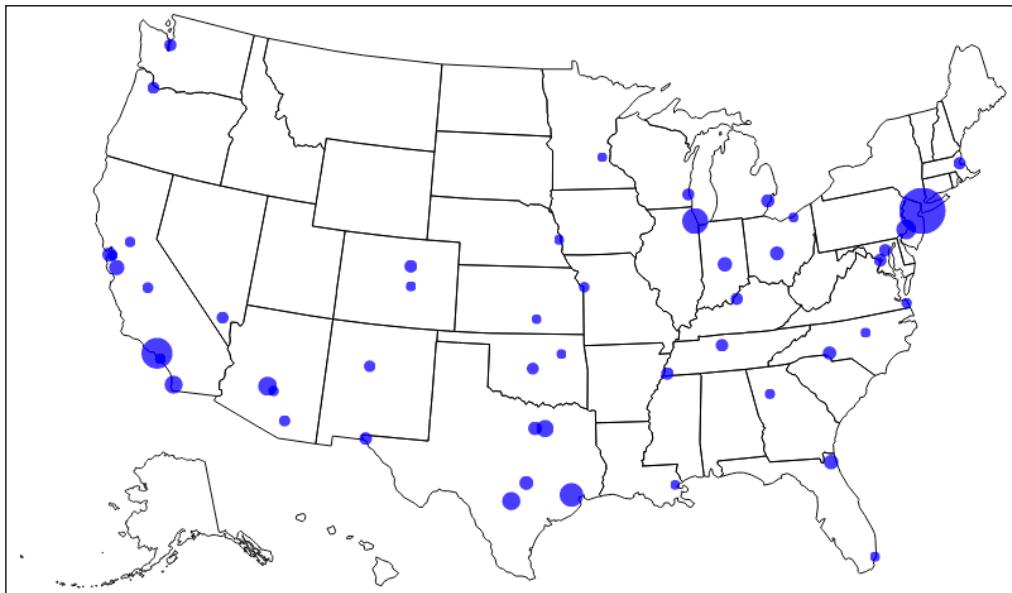
The data we will use is in `us-cities.csv`, which is available at <https://gist.github.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb56e36/us-cities.csv>. The data is straightforward; the following are the first few lines:

```
name,population,latitude,longitude
New York,8491079,40.6643,-73.9385
Los Angeles,3792621,34.0194,-118.4108
Chicago,2695598,41.8376,-87.6818
```

The example is available at the following location:



The resulting visualization is the following:



The preceding example leverages the United States Mercator examples code. This example does, however, need to load two data files. To facilitate this, we will use a library called **queue** created by Mike Bostock to load these files asynchronously, and when both are complete, execute the `ready()` function. You can get this library and documentation at <https://github.com/mbostock/queue>:

```
queue()
  .defer(d3.json, usDataUrl)
  .defer(d3.csv, citiesDataUrl)
  .await(function (error, states, cities) {
```

The map is then rendered as in the earlier examples. Then we need to place the circles. To do this, we will need to convert the latitude and longitude values to X and Y pixel locations. We can do this in D3.js using the projection object:

```
svg.selectAll('circle')
  .data(cities)
  .enter()
  .append('circle')
  .each(function(d) {
    var location = projection([d.longitude, d.latitude]);
    d3.select(this).attr({
      cx: location[0],
      cy: location[1],
      r: Math.sqrt(+d.population * 0.00004)
    });
  })
  .style({
    fill: 'blue',
    opacity: 0.75
});
```

For each circle that is created, this code calls the projection function passing it the latitude and longitude for each city. The return value is the x and y location of the pixel representing that location. So we just set the center of the circle to this result, and assign the circle a radius that is a scale value of the population.

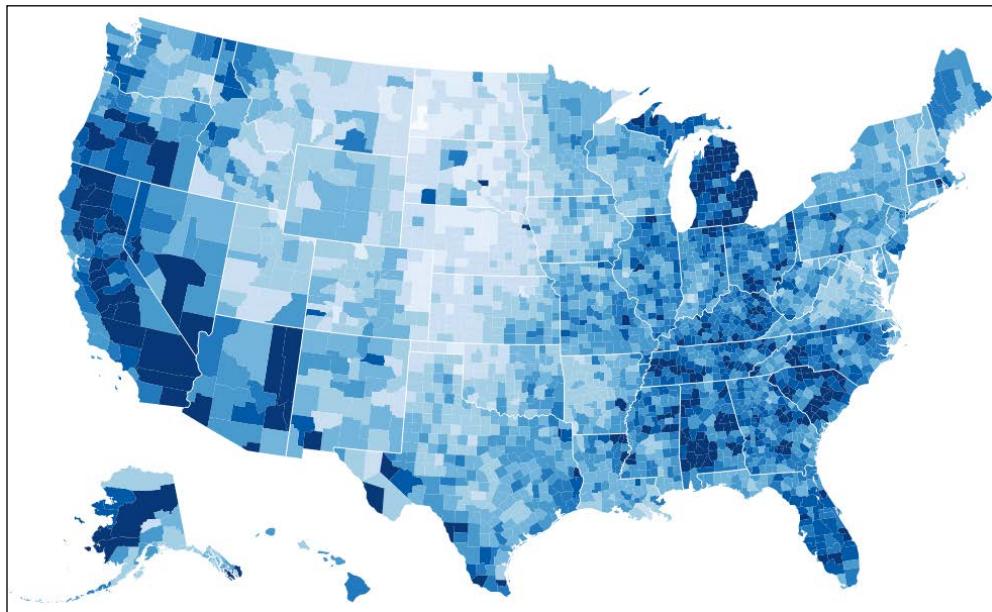
Creating a choropleth

Our last map example is for creating a **choropleth**. A choropleth is a map with areas filled in with different colors to reflect the underlying data values—not just differing colors to represent different geographic boundaries. These are quite common types of visuals, and they commonly show a difference in opinion amongst the populations in adjacent regions, or how economic factors differ along neighbors.

The example is available at the following location:



The resulting visualization is the following:



This choropleth represents the unemployment rate in the US counties for the year 2008. The shade of blue varies from darker, representing lower unemployment, to lighter and higher unemployment.

The data for unemployment is available at <https://gist.github.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb56e36/unemployment.tsv>. The first few lines are the following:

id	rate
1001	.097
1003	.091
1005	.134
1007	.121
1009	.099
1011	.164

The data consists of a pair of a county identifier and the respective unemployment rate. The county ID will be matched to county IDs in the `us.json` file available at <https://gist.github.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb56e36/us.json>.

This file consists of TopoJSON describing the shape of all of the counties in the US, each with the same county ID in the unemployment file. A snippet of this file is the following, which shows for country 1001 the arcs that should be used to render it:

```
{  
  "type": "Polygon",  
  "id": 1001,  
  "arcs": [ [ -8063, 8094, 8095, -8084, -7911 ] ]  
},
```

Our goal is to quantize the unemployment rates, and then fill each geometry with a color mapped to that quantile. It's actually easier to do than it may seem.

In this example, we will map our unemployment rates into ten quantiles. The color used for each will be specified using a style with a specific name. These are declared as follows:

```
<style>  
  .q0-9 { fill:rgb(247,251,255); }  
  .q1-9 { fill:rgb(222,235,247); }  
  .q2-9 { fill:rgb(198,219,239); }  
  .q3-9 { fill:rgb(158,202,225); }  
  .q4-9 { fill:rgb(107,174,214); }  
  .q5-9 { fill:rgb(66,146,198); }  
  .q6-9 { fill:rgb(33,113,181); }  
  .q7-9 { fill:rgb(8,81,156); }  
  .q8-9 { fill:rgb(8,48,107); }  
</style>
```

The data is loaded using the `queue()` function:

```
queue()  
  .defer(d3.json, usDataUrl)  
  .defer(d3.tsv, unempDataUrl, function(d) {  
    rateById.set(d.id, +d.rate);  
  })  
  .await(function(error, us) {
```

This code uses an alternate form of `.defer()` for the unemployment data, which calls a function for each data item that is loaded (another cool thing about queue). This builds a `d3.map()` object (like a dictionary object) that maps the county ID to its unemployment rate, and we use this map later during rendering.

The county data is rendered first. To do this, we need to create a quantile scale which maps the domain from 0 to 0.15. This will be used to map the unemployment levels to one of the styles. The range is then configured to generate the names of the nine styles:

```
var quantize = d3.scale.quantize()
  .domain([0, .15])
  .range(d3.range(9).map(function(i) {
    return 'q' + i + '-9';
}));
```

Next, the code creates the `albersUsa` projection and an associated path:

```
var projection = d3.geo.albersUsa()
  .scale(1280)
  .translate([width / 2, height / 2]);

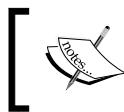
var path = d3.geo.path()
  .projection(projection);
```

The next step is to create a group to hold the shaded counties. Then, to this group, we will add a path for each county by binding it to the `counties` features:

```
svg.append('g')
  .attr('class', "counties")
  .selectAll("path")
  .data(topojson.feature(us, us.objects.counties).features)
  .enter()
  .append("path")
  .attr("class", function(d) {
    return quantize(rateById.get(d.id));
  })
  .attr("d", path);
```

Finally, we overlay the outlines of the states using a white stroke for the borders to help us differentiate the state borders:

```
svg.append('path')
  .datum(topojson.mesh(us, us.objects.states))
  .attr({
    'class': 'states',
    fill: 'none',
    stroke: '#fff',
    'stroke-linejoin': 'round',
    'd': path
});
```



This particular piece of code also uses the `topojson.mesh` function to extract the **Multipolygon** (GeoJSON) data for all of the states from the TopoJSON object.



And that's all! We've created a choropleth, and used a coding pattern that can be reused easily with other types of data.

Summary

We started this chapter by looking briefly at GeoJSON and TopoJSON. If you do anything with maps in D3.js, you will be using one or both of these. We covered it just enough to give an understanding of its structure, and how it is used to define data that can be rendered as a map.

From there, we dove into creating several maps and covered many of the concepts that you will use in their creation. These included loading the data, creating projections, and rendering the geometries within the data.

We examined two projections, Mercator and orthographic, to give an idea of how these present data. Along the way, we also looked at how to style elements on the map, filling geometries with color, and highlighting geometries on `mouseover`.

Then we examined how to annotate our maps with labels as well as color elements based upon data (choropleths), and to place symbols on the map at specific geographic positions, with a size that is based upon the data.

At this point in the book, we have been pretty thorough in covering much of the core of D3.js, at least enough to make you very dangerous with it. But we have also only ever created stand-alone visualizations, ones that do not interact with other visualizations.

In the next chapter, the final one of this book, we will look at combining multiple D3.js visualizations using AngularJS, and where those visuals also react to the user manipulating other content on their page.

13

Combining D3.js and AngularJS

The final topic in this book will demonstrate using multiple D3.js visuals on a single web page. These examples will also demonstrate constructing D3.js visuals in a modular manner, which allows their reuse through simple HTML tags, and at the same time abstracting the data from the code that renders the visual. This will enable the creation of more generic D3.js visuals, which can be placed on a page using a single HTML tag and are also loosely coupled with the source of the data.

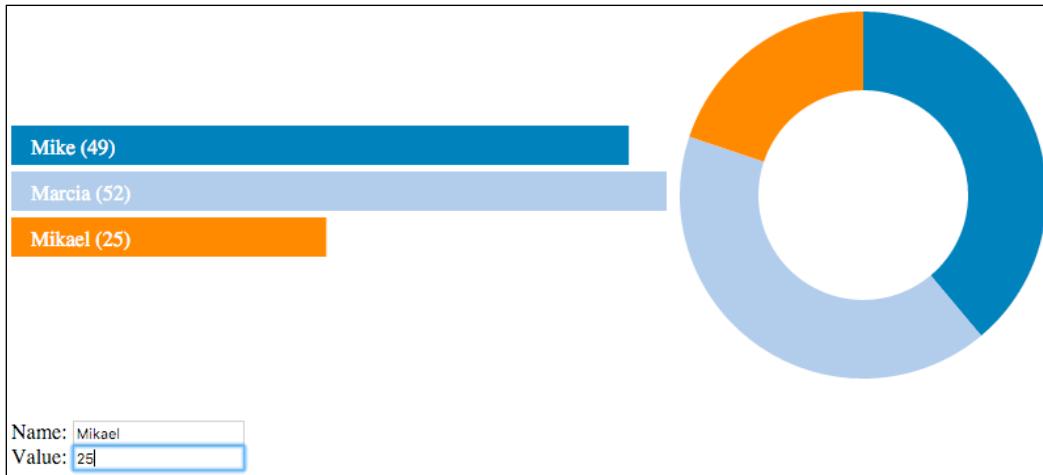
To implement these features, we will utilize **AngularJS**, a JavaScript framework used to create dynamic and modular web applications. The examples will demonstrate how to integrate both AngularJS (v1.4) and D3.js to make reusable and interoperable visualizations. An introductory knowledge of AngularJS is expected for this chapter, but the focus will be on how to use the features of AngularJS to create reusable and extensible D3.js controls; therefore, even someone new to AngularJS will be able to follow along.

In this chapter, we will accomplish this by going through the following topics:

- An overview of composite visualization
- Creating a bar chart using an AngularJS application, controller, and directive
- Adding a second directive to add a donut graph to the page
- Adding a detail view and interactivity between the visuals
- Updating the graphs upon modification of details in the data

An overview of composite visualization

Before jumping into the examples, let's start by examining the end result to help conceptualize several of the goals that we will attempt to accomplish using AngularJS combined with D3.js. The following figure represents a static image of the resulting interactive and composite graphs:



Each component of the page – the bar graph, the donut graph, and the input form – will initially be built independently and will be able to function on its own. To do this, the examples will use features from AngularJS to facilitate the following features:

- Each visual should be expressed in HTML as a simple HTML tag instead of copying the code for each onto the page. This is performed using AngularJS directives.
- Instead of loading the data once within the code for each visual, we will leverage a common application-level data model shared across each element. In AngularJS, this is done by creating a JavaScript data model and injecting it into the controllers for each directive.
- The bar graph will provide a means of exposing notifications of updates to a currently selected item, upon which the detail model can update its data. This will be implemented through a `selectedItem` property in the model that the details directive can monitor for updates using AngularJS template bindings.
- Also, when the application model is updated in the details directive, the bar and donut graphs will be notified by AngularJS to be updated to represent the modifications.



A note of difference in this chapter from the previous is that the code is not available online on bl.ock.org or JSBIN.COM and must be retrieved from the Packt website. This is because the examples utilize AngularJS, which doesn't play as well with bl.ock.org and JSBIN.COM. The code must therefore be run locally from a web server. You can simply unzip the code and place it in the root of a web server or start a web server of your choice in the root folder of the content. Each example is implemented as a different HTML file in the root of the folder, and each of these refers to multiple other files in various subdirectories.

Creating a bar graph using AngularJS

The first example will create a reusable bar chart component to demonstrate creating an AngularJS directive with an underlying controller. This is implemented within an HTML file, `01_just_bars.html`, which consists of the following components:

- **The AngularJS application object:** This functions as an entry point for AngularJS code in the page (that is, in `app.js`)
- An AngularJS controller (in `controllers/basic_dashboard.js`): This creates the data and sends it to the directive that renders the HTML code for the graph
- The directive: This renders the D3.js bar chart in `directives/bars.js`.

The web page and application

The AngularJS application is presented to the user via a web page, which begins by loading the AngularJS and D3.js libraries (this is common in all the examples in this chapter). Take a look at the following code:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.10/
angular.min.js"></script>
<script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
```

The page then loads the implementations of the AngularJS application object, directive, and controller. Now, execute the following code:

```
<script src="app.js"></script>
<script src="directives/bars.js"></script>
<script src="controllers/basic_dashboard.js"></script>
```

The details of these will be examined in a moment. Before we look at these, the remainder of the HTML code in this file creates the AngularJS application and the controller for our directive using a `<div>` tag with the `ng-app` and `ng-controller` properties. Add the following code:

```
<div ng-app="dashboardApp" ng-controller="dashboardController">
    <bars-view width="500" height="105"></bars-view>
</div>
```

The use of the `ng-app` attribute tells AngularJS where to find the implementation, which is a module (that is, a piece of AngularJS JavaScript referable) named `dashboardApp`.

In this example, this module is declared in `app.js` (this is the same for each example):

```
angular.module('dashboardApp', []);
```

This example does not actually declare any code for the application module and is simply a place for the HTML markup to reach into AngularJS and start locating various objects. In a more elaborate application, this would be a good place to inject other dependent modules and do some application-level initialization.

The tag within this `<div>` tag defines a construct known as an AngularJS directive. This renders the data represented in the controller. Before we get to the implementation of the directive, let's take a look at the controller that provides the data to the directive.

The controller

The `ng-controller` attribute on the `<div>` tag specifies a name of a controller that is used to provide data to the AngularJS directives that are specified as the child elements of this `<div>` tag. AngularJS searches for a controller with the specified name within one of the modules specified by `ng-app`. In this example, this controller is declared in `controllers/basic_dashboard.js`, as follows:

```
angular.module('dashboardApp')
.controller('dashboardController',
 ['$scope', function ($scope) {
     $scope.items = [
         { Name: 'Mike', Value: 49 },
         { Name: 'Marcia', Value: 52 },
         { Name: 'Mikael', Value: 18 }
     ];
}]);
```

This creates an AngularJS controller using `.controller()` with the name `dashboardController`, which is a part of the application's `dashboardApp` module. Take a look at the following script:

```
angular.module('dashboardApp')
    .controller('dashboardController',
        ['$scope', function ($scope) {
```

The second parameter of `.controller()` is an array that specifies the variables to be injected into the method implementing the controller and then the function that implements the controller.

Now, this informs AngularJS that we would like the AngularJS variable `$scope`, which represents the data of the controller and will be injected into the directives of the control to be passed into this function that is to be initialized.

The last statement in the following command declares the data that is to be provided to the view by adding an item's property to the scope:

```
$scope.items = [
    { Name: 'Mike', Value: 49 },
    { Name: 'Marcia', Value: 52 },
    { Name: 'Mikael', Value: 18 }
];
```

The directive for a bar graph

An angular directive is a custom HTML tag that instructs AngularJS on how to create HTML based on the data provided by the controller. In the HTML code of the example is a tag declared that is named `<bars-view>`. When loading the page, AngularJS examines all the tags in HTML, and if a tag is not recognized as a standard HTML tag, AngularJS searches for a directive that you declared as part of the application to provide an implementation for this tag.

In this case, it converts the hyphenated name of the tag, `<bars-view>`, to a camel case version, `barsView`, and looks for a directive within a module that was declared with this name. If found, AngularJS executes the code that is provided for the directive to generate the HTML code.

In this example, AngularJS finds the `<bars-view>` tag implemented in the `directives/bars.js` file. This file starts by informing AngularJS that we want to declare a directive named `barsView` in the `dashboardApp` module:

```
angular.module('dashboardApp')
  .directive('barsView', function () {
    return {
      restrict: 'E',
      scope: { data: '=' },
      link: renderView
    };
  });

```

The second parameter to `.directive()` is a function that informs AngularJS how to apply and construct the view. In this example, there are three instructions specified:

- `restrict: 'E'`: This informs AngularJS that this directive applies to HTML elements only and not to their attributes or CSS class names.
- `scope: { data: "=" }`: This tells AngularJS that we want to have **two-way binding** between the data in the scope and the elements in the view. If data changes in the controller, AngularJS will update the view and vice versa.
- `link: renderView`: This property informs AngularJS which function will be called when the view is created. This function will then generate DOM constructs to represent the view. This is where we will put our D3.js code.

The `renderView` function is declared as follows:

```
function renderView($scope, $elements, $attrs) {
```

When AngularJS calls this function to render a tag for a directive, it passes the scope object represented by the related controller as the `$scope` parameter. The second parameter, `$elements`, is passed an AngularJS object that can be used to identify the top-level DOM element where the directive should append new elements. The last parameter, `$attrs`, is passed any custom attribute defined in the root DOM element in the prior parameter.

The code to implement the bar graph is not significantly different from our earlier bar graph examples. The first thing it does that is different because of AngularJS gets the data from the scope that was passed into the function, as follows:

```
var data = $scope.$parent.items;
```

The `<bars-view>` directive is assigned a scope object by AngularJS. The data from the controller is actually a property of the `parent` scope property of this object. This object has the `items` property that we defined in the controller and its associated data as the `items` property.

The width and height of the element, as specified in the HTML code, can be retrieved using the `width` and `height` attributes of the `$attrs` parameter. Take a look at the following command:

```
var width = $attrs.width, height = $attrs.height;
```

After obtaining the width and height, we can create the main SVG element of the graph. This will be appended to `$element[0]`, which represents the root DOM element for this directive (The `$element` object is actually an AngularJS one wrapping the root element, which is access using the `[0]` indexer), as follows:

```
var svg = d3.select($element[0])
.append("svg");
```

The remainder of the code is similar to the examples covered in previous chapters to create a bar graph with overlaid text. It begins by setting the size of the SVG element and setting up various variables required to calculate the size and positions of the bars, as shown in the following code:

```
svg.attr({
  width: width,
  height: height
});

var max = d3.max(data, function(d) {
  return d.Value;
});

var colors = d3.scale.category20();

var barHeight = 30;
var leftMargin = 15;
var barTextOffsetY = 22;
```

The bars are then created and set to animate to their maximum respective sizes. Take a look at the following:

```
svg.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr({
    height: barHeight,
    width: 0,
    x: 0,
    y: function(d, i) {
```

```
        return i * barHeight;
    },
    stroke: 'white'
})
.style('fill', function(d, i) {
    return colors(i);
})
.transition()
.duration(1000)
.attr('width', function(d) {
    return d.Value / (max / width);
});
```

Now, all the existing D3.js elements are selected in an update scenario, which transitions the size of any existing bar to the new size. Take a look at this code:

```
svg.selectAll("rect")
  .data(data)
  .transition()
  .duration(1000)
  .attr("width", function(d, i) {
    return d.Value / (max / width);
});
```

Then, the cases to create both the entering labels on the bars and change the text on the bars if the data values change are implemented, as follows:

```
svg.selectAll('text')
  .data(data)
  .enter()
  .append('text')
  .attr({
    fill: '#fff',
    x: leftMargin,
    y: function(d, i) {
      return i * barHeight + barTextOffsetY;
    }
  })
  .text(function(d) {
    return d.Name + ' (' + d.Value + ')';
  });

svg.selectAll('text')
  .data(data)
  .attr({
```

```
        fill: '#fff',
        x: leftMargin,
        y: function(d, i) {
            return i * barHeight + barTextOffsetY;
        }
    })
    .text(function(d) {
        return d.Name + ' (' + d.Value + ')';
    });
}
}
```

When opening this page in the browser, the following graph is presented:



Adding a second directive for a donut

The next example adds a second D3.js visualization to represent a donut graph of the values in the data. This implementation requires creating a new directive and adding this directive to the web page. It reuses the implementation of the controller and also the data that it creates.

The web page

The web page for this example is available in `02_bars_and_donut.html`. The web page is slightly different from the previous one in that it includes one additional view for the donut. Take a look at the following:

```
<script src="app.js"></script>
<script src="views/bars.js"></script>
<script src="views/donut.js"></script>
<script src="controllers/basic_dashboard.js"></script>
```

The declaration of the content for the page now becomes the following:

```
<div ng-app="dashboardApp" ng-controller="BasicBarsController">
  <bars-view width="500" height="105"
    style="display: table-cell; vertical-align: middle">
  </bars-view>
  <donut-view width="300" height="300"
    style="display: table-cell">
  </donut-view>
</div>
```

This adds an additional directive for `donut-view`. There is also a style added to the directives to make them float next to each other.

The directive for the donut graph

The implementation of the donut directive begins by declaring that that this directive will be added to the `dashboardApp` module and that its name will be `donutView` (hence we use `<donut-view>` in the HTML code). As with the bar graph directive, it also instructs AngularJS that this code should be applied only to DOM elements, have two-way data binding, and be implemented by a function named `renderView`; take a look at the following code:

```
angular.module('dashboardApp')
.directive('donutView', function () {
  return {
    restrict: 'E',
    scope: { data: '=' },
    link: renderView
  };
});
```

This version of `renderView` follows a similar pattern to the implementation for `bars-view`. It begins by getting the data from the scope, including the width and height of the visual, and also calculates the radius for the donut. The following code is executed:

```
function renderView($scope, $elements, $attrs) {
  var data = $scope.$parent.items;

  var width = $attrs.width,
  height = $attrs.height,
  radius = Math.min(width, height) / 2;
```

The rendering of the donut is then started using a pie layout, as follows:

```
var pie = d3.layout.pie()  
    .value(function (d) { return d.Value; })  
    .sort(null);
```

The arcs fill between 10 and 70 pixels from the outside of the boundary of the SVG element, which is based on the calculated radius. Take a look at the following code:

```
var arc = d3.svg.arc()  
    .innerRadius(radius - 70)  
    .outerRadius(radius - 10);
```

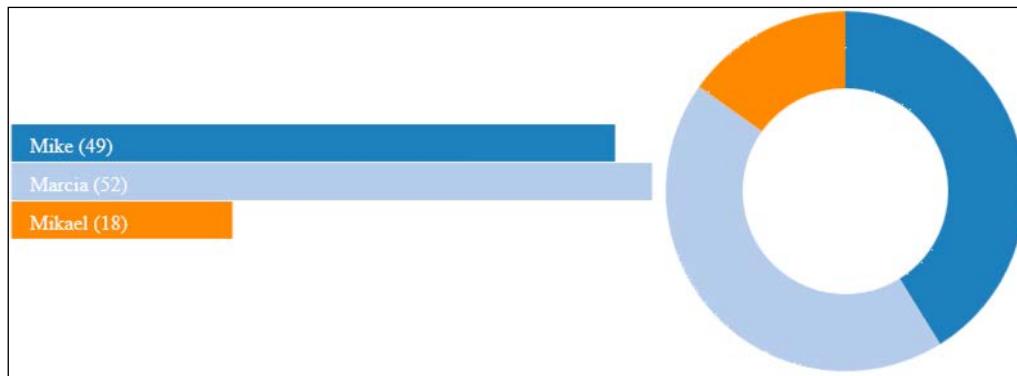
Then, the visual is started to be constructed by appending the main SVG element to `$elements[0]`, as follows:

```
var svg = d3.select($elements[0])  
    .append('svg')  
    .attr({  
        width: width,  
        height: height  
});
```

Finally, the visual elements for the donut graph are constructed using a color scale and path generator for each entering datum, as follows:

```
var colors = d3.scale.category20();  
graphGroup  
    .datum(data)  
    .selectAll('path')  
    .data(pie)  
    .enter()  
    .append('path')  
    .attr('fill', function(d, i) {  
        return colors(i);  
    })  
    .attr('d', arc)  
    .each(function(d) {  
        this._current = d;  
    });
```

Upon loading this page in the browser, it presents the following visual, which now has two D3.js visuals on a single web page:



Adding a detail view and interactivity

The next example adds a details directive to the page and also interactivity such that when a bar is clicked, the details directive will display the appropriate data for the selected bar.

To achieve this interactivity, the bar graph directive is modified so that it produces an action that can be monitored by other parts of the AngularJS application. This action will be to set a `selectedItem` property on the model, which other controllers or directives can watch for changes and then take action.

The web page

The web page for this example is contained in `03_with_detail.html`. The content included differs slightly, in that we will include a new implementation of our `<bars-view>` directive in `directives/bars_with_click.js` and the controller in `controllers/enhanced_controller.js` and a reference to a new directive representing the detail view in `directives/detail.js`. Take a look at the following:

```
<script src="app.js"></script>
<script src="directives/bars_with_click.js"></script>
<script src="directives/donut.js"></script>
<script src="directives/detail.js"></script>
<script src="controllers/enhanced_controller.js">
</script>
```

The declaration of the main `<div>` tag changes slightly to the following by adding a directive for `details-view`:

```
<div ng-app="dashboardApp" ng-controller="dashboardController">
    <bars-view width="500" height="105"
        style="display: table-cell; vertical-align: middle">
    </bars-view>
    <donut-view width="300" height="300"
        style="display: table-cell"></donut-view>
    <details-view data="selectedItem" width="300">
    </details-view>
</div>
```

Note that this new directive uses an attribute named `data` and sets its value to `selectedItem`. This is a special AngularJS attribute/binding that specifies that the model data for this directive will be located in the `selectedItem` property of the nearest scope object upward in the DOM hierarchy. In this case, it is the scope defined on the `div` tag, and whenever this property on the scope is changed, this directive will update its data and visualization automatically.

Specifying an initial `selectedItem` in the controller

The details view controller expects to have access to a `selectedItem` property of the model to use as its data, and it will, therefore, need to set an initial value to this property. The following adds a single line to accomplish this task:

```
angular.module('dashboardApp')
    .controller('dashboardController',
        ['$scope', function ($scope) {
            $scope.items = [
                { Name: 'Mike', Value: 49 },
                { Name: 'Marcia', Value: 52 },
                { Name: 'Mikael', Value: 18 }
            ];
            $scope.selectedItem = $scope.items[0];
        }]);

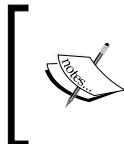
```

The modified bars view directive

The `<bars-view>` directive then adds a click handler to set the value of the selected item whenever a bar is clicked, as follows:

```
.on('click', function (d, i) {
  $timeout(function () {
    parent.selectedItem = d;
  });
})
```

This click handler performs one action: it updates the value of the selected item in the parent scope to the value of the data item underlying the clicked visual. It does not send messages to other components, nor should it. Other directives, if interested in this update, will be able to take this action by looking for changes in the model.



This is wrapped in a call to the AngularJS `$timeout` function, which will have the browser update the UI, based on the change of this property. If this is not performed, any interested element will not be notified by AngularJS.



Implementing the details view directive

The details view is a fairly simple piece of code that starts with a directive declaration. Take a look at the following:

```
angular.module('dashboardApp')
.directive('detailsView', function () {
  return {
    restrict: 'E',
    scope: { data: "=" },
    templateUrl: 'templates/static_item.html'
  };
});
```

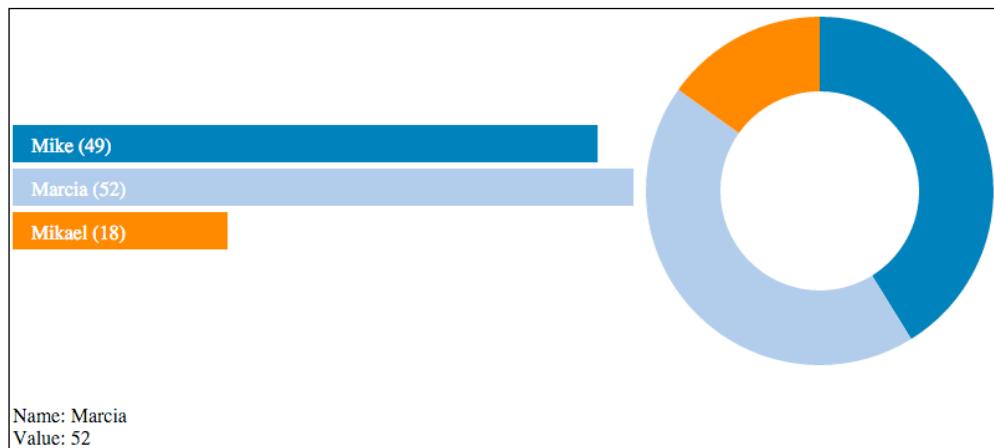
A difference in this declaration from our other directives is that the code does not specify a `link` property but a `templateUrl` property and an associated value. This tells AngularJS that this directive will not be implemented by a call to a JavaScript function but should use content from the `templates/static_item.html` file. The contents of this file are the following:

```
Name: {{data.Name}}
<br/>
Value: {{data.Value}}
```

This HTML code will be injected into DOM by AngularJS. The HTML contains embedded **handlebars** syntax that AngularJS will notice and substitute the content of. In this case, the values of the Name and Value properties of the object specified by the data attribute of the directive will be used, where data is the bound value of `selectedItem` from the model, which is the currently selected bar. Whenever this property is updated, AngularJS will automatically update DOM correctly on our behalf without any additional coding.

The resulting interactive page

The following image is an example of a possible display rendered by this page:



In this image, the second bar was clicked on, and so the details view displays the data for this bar. As you click on the different bars, the values in the details change to match.

Updating graphs upon the modification of details data

The final example will make the update of the data bidirectional between the details view and bar and donut graphs. The previous example only updates the detail view upon clicking on a bar. The content of the details view is static text, and hence, the user cannot modify the data. This is changed by modifying the template to utilize text input fields. There is no change to the controller, so it will not be discussed.

The web page

The web page for this example, `04_dynamic.html`, contains several small changes from the previous example to reference new implementations for the bars, donut, and details directives. The `<div>` tag remains the same. Take a look at the following code:

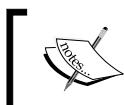
```
<script src="app.js"></script>
<script src="directives/bars_with_click_and_updates.js"></script>
<script src="directives/donut_with_updates.js"></script>
<script src="directives/dynamic_detail.js"></script>
<script src="controllers/enhanced_controller.js">
</script>
```

The revised bar-view directive

The new `<bar-view>` directive has one behavioral change along with a small structural change. This behavioral change is to watch for changes to the `selectedItem` property of the scope that is supplied to it. To do this, the following statement is added near the top of the code for `renderView()`:

```
parent.$watch("selectedItem", render, true);
```

This informs AngularJS that we want it to watch for changes in the bound scope object's `selectedItem` property. When this property or any property of this object changes (as specified by `true` as the third parameter), AngularJS will call the `render()` function.



Note that this watch process does not have to be performed in the details view controller as the use of a template and handlebars sets this up automatically.



The structural change to the code is made after the call to select the `svg` element and the setting of its size. The code to create the visual is now wrapped in the new `render()` function, which is called the first time the directive is loaded and then each time the value of `selectedItem` is changed. When the latter happens, the bar graph is updated, it animates the bars to new sizes, and it also modifies the text labels.

The revised donut-view directive

Similarly to the updates to the bar-view directive, this directive is changed by adding a call to watch the `selectedItem` property of the scope as well as wrapping the rendering code in an `updatePath()` function, which can be called when the value of this property changes, as follows:

```
parent.$watch('selectedItem', updatePath, true);
```

The `updatePath()` function only needs to regenerate the path for each of the arc segments, as shown in the following code:

```
function updatePath() {
    path = path.data(pie);
    path.transition()
        .duration(750)
        .attrTween('d',
            function() {
                var i = d3.interpolate(this._current, a);
                this._current = i(0);
                return function(t) {
                    return arc(i(t));
                };
            });
}
```

The detail-view directive

The new `<detail-view>` directive has one modification, which is to use a different template. Take a look at the following code:

```
angular.module('dashboardApp')
.directive('detailsView', function () {
    return {
        restrict: 'E',
        scope: { data: "=" },
        templateUrl: 'templates/dynamic_item.html'
    };
});
```

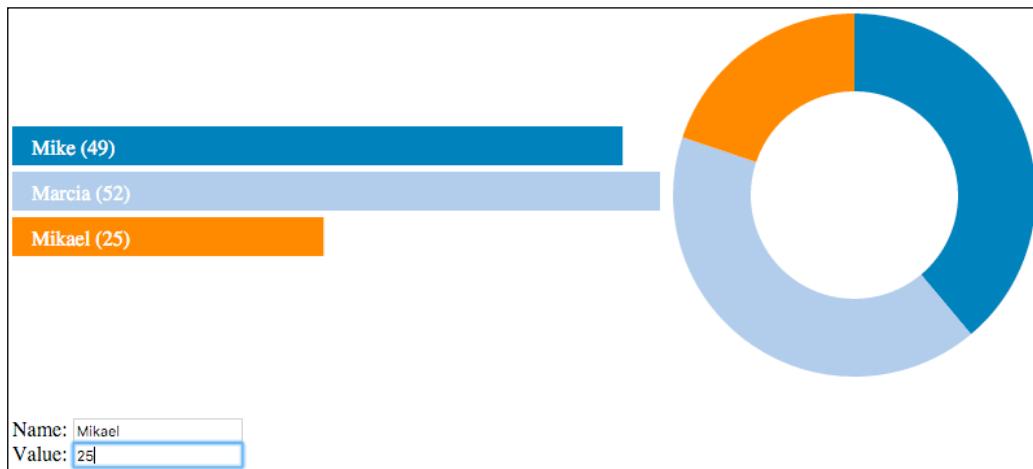
The contents of this template specify input boxes instead of text fields, as follows:

```
Name: <input type="text" ng-model="data.Name"/>  
<br/>  
Value: <input type="text" ng-model="data.Value"/>
```

Note that for input fields to update handlebars, notations cannot be utilized. For this to work, you need to use the AngularJS `ng-model` attribute and point it to the bound data object and respective property.

The results

The following screenshot shows this example in action:



In this demonstration, the third bar was clicked on, and `details-view` now provides edit controls to allow us to change the values. The value for **Mikael** was then changed to **25**, and the bar and donut graphs were animated to represent the change in values.

One of the really nice things going on here is that literally, key stroke by key stroke on both of these input fields, AngularJS will update these properties and both the bar and donut charts will be updated on each key stroke!

Summary

The examples in this chapter demonstrated how to use AngularJS to make modular and composite D3.js visualizations. They started by showing how to place data within an AngularJS controller and share it with multiple D3.js visuals. Next, we demonstrated how to share data from a single controller to multiple directives. The final two examples demonstrated how to use a shared property for two-way communication and implement a details view to allow the editing of data.

This wraps up this book on using D3.js through examples. The book started with the basic concepts of D3.js and using its constructs to bind data and generate SVG from it. From this foundation, we progressed through adding features to the examples, each of which demonstrated progressive extensions of the previous examples within the same chapter as well as with incrementally complex constructs from chapter to chapter. In the end, the examples covered many of the concepts in D3.js that can take you from a novice to being able to construct rich, interactive, and composite visualizations, all through examples.

Module 3

Mastering D3.js

Bring your data to life by creating and deploying complex data visualizations with D3.js

1

Data Visualization

Humans began to record things long before writing systems were created. When the number and diversity of things to remember outgrew the capacity of human memory, we began to use external devices to register quantitative information. Clay tokens were used as early as 8000-7500 BC to represent commodities like measures of wheat, livestock, and even units of man labor. These objects were handy to perform operations that would have been difficult to do with the real-life counterparts of the tokens; distribution and allocation of goods became easier to perform. With time, the tokens became increasingly complex, and soon, the limitations of the complex token system were identified and the system began to be replaced with simpler yet more abstract representations of quantities, thereby originating the earlier systems of writing.

Keeping records has always had a strong economic and practical drive. Having precise accounts of grains and pastures for the livestock allowed people to plan rations for the winter, and knowing about seasons and climate cycles allowed people to determine when to plant and when to harvest. As we became better at counting and registering quantitative information, trading with other nations and managing larger administrative units became possible, thereby providing us with access to goods and knowledge from other latitudes. We keep records because we think it's useful. Knowing what we have allows us to better distribute our assets, and knowing the past allows us to prepare for the future.

Today, we register and store more data than ever. Imagine that you want to go out for a morning cup of coffee. If you pay in cash, the date, price of the coffee, and the kind of coffee will be recorded before your coffee was actually prepared. These records will feed the accounting and stock systems of the store, being aggregated and transformed to financial statements, staff performance reports, and taxes to be paid by the store. Paying with credit card will generate a cascade of records in the accounting system of your bank. We measure things hoping that having the information will help us to make better decisions and to improve in the future.

History demonstrates that gathering and understanding data can help to solve relevant problems. An example of this is the famous report of John Snow about the Broad Street cholera outbreak. On August 31, 1854, a major outbreak of cholera was declared in the Soho district of London. Three days later, 127 people died from the disease. At the time, the mechanism of transmission of the cholera was not understood. The germ theory was yet to exist, and the mainstream theory was that the disease spread by a form of bad air. The physician, John Snow, began to investigate the case, collecting and classifying facts, recording deaths and their circumstances as well as a great number of testimonials. Refer to the following screenshot:



Details of the original map made for Snow, displaying the deaths by cholera in the Soho district

He gave special attention to the exceptions in the map and noticed that neither the workhouse inmates nor the brewery workers had been affected. The exceptions became further proof as he discovered that about 70 employees who worked in the brewery drank only beer made with water from a pump inside the walls of the brewery. In the workhouse, which also had its own water pump, only 5 out of 500 died, and further investigation revealed that the deceased were admitted when the outbreak had already begun. Although the map is convincing enough, Snow's original report contains more than 150 pages filled with tables and testimonials that support or raise questions about his theory. The local council decided to disable the pump by removing its handle, when the outbreak had already begun to decline.

The report from John Snow is a great triumph of detective work and data visualization. He gathered information about the deaths and their circumstances and displayed them as data points in their geographic context, which made the pattern behind the causalities visible. He didn't stop at studying the data points; he also investigated the absence of the disease in certain places, faced the exceptions instead of quietly dismissing them, and eventually formed stronger evidence to support his case.

In this chapter, we will discuss what makes visual information so effective and discuss what data visualization is. We will comment about the different kinds of data visualization works, which gives a list of references to learn more about it. We will also discuss D3 and its differences with other tools to create visualizations.

Defining data visualization

Our brains are specially adapted to gather and analyze visual information. Images are easier to understand and recall. We tend to analyze and detect patterns in what we see even when we are not paying attention. The relation between visual perception and cognition can be used to our advantage if we can provide information that we want to communicate in a visual form.

Data visualization is the discipline that studies how to use visual perception to communicate and analyze data. Being a relatively young discipline, there are several working definitions of data visualization. One of the most accepted definitions states:

"Data visualization is the representation and presentation of data that exploits our visual perception in order to amplify cognition."

The preceding quote is taken from *Data Visualization: A successful design process*, Andy Kirk, Packt Publishing.

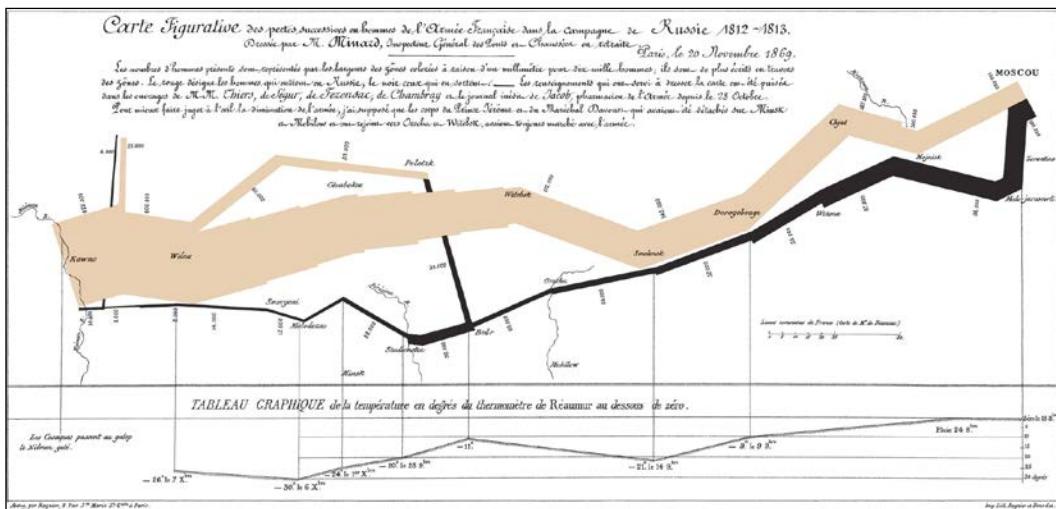
There are several variants for this definition, but the essence remains the same – data visualization is a visual representation of data that aims to help us better understand the data and its relevant context. The capacity for visual processing of our brains can also play against us. Data visualization made without proper care can misrepresent the underlying data and fail to communicate the truth, or worse, succeed in communicating lies.

The kind of works that fall under this definition are also diverse; infographics, exploratory tools, and dashboards are data visualization subsets. In the next section, we will describe them and give some notable examples of each one.

Some kinds of data visualizations

There are countless ways to say things, and there are even more ways to communicate using visual means. We can create visualizations for the screen or for printed media, display the data in traditional charts, or try something new. The choice of colors alone can be overwhelming. When creating a project, a great number of decisions have to be made, and the emphasis given by the author to the different aspects of the visualization will have a great impact on the visual output.

Among this diversity, there are some forms that are recognizable. Infographics are usually suited with a great deal of contextual information. Projects more inclined to exploratory data analysis will tend to be more interactive and provide less guidance. Of course, this classification is only to provide reference points; the data visualization landscape is a continuum between infographics, exploratory tools, charts, and data art. Charles Minard's chart, which shows the number of men in Napoleon's 1812 Russian campaign, is shown in the following screenshot:

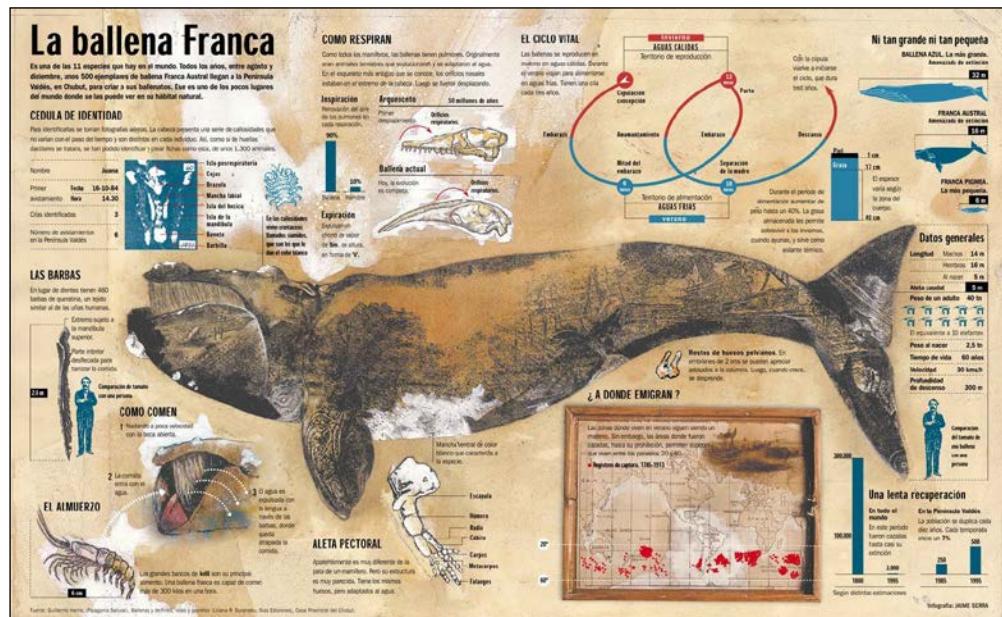


Charles Minard's flow map of Napoleon's march

It would be difficult to classify Charles Minard's figure as an infographic or as a flow chart because it allows for both. The information displayed is primarily quantitative, but it's shown in a map with contextual information that allows us to better understand the decline in the Napoleonic forces. There are several dimensions being displayed at once such as the number of soldiers, the geographic location of the soldiers during the march, and the temperature at each place. The figure does amazing work by showing how diminished the forces were when they arrived at Moscow and how the main enemy was the cold winter.

Infographics

Infographics is a form of data visualization that is focused on communicating and explaining one or more particular views of a subject. It usually contains images, charts, and annotations, which provides context and enhances the reader's capacity to understand the main display of information. The award-winning infographic about the right whale (*La ballena Franca* in original Spanish), created by Jaime Serra and published in the Argentinian newspaper, *Clarín*, in 1995 is a great example of how infographics can be a powerful tool to enlighten and communicate a particular subject. This can be found at http://3.bp.blogspot.com/_LCqDL30ndZQ/TBPKvZIQAiI/AAAAAAAik/OrjA6TShNsk/s1600/INFO-BALLENA.jpg. A huge painting of the right whale covers most of the infographic area. A small map shows where this species can be found during their migratory cycles. There are outlines of the right whale alongside other kinds of whales, comparing their sizes. The image of the whale is surrounded by annotations about their anatomy that explain how they swim and breathe. Bar charts display the dramatic decline in their population and how they are recovering at least in some corners of the globe. All these elements are integrated in a tasteful and beautiful display that accomplishes its purpose, which is to display data to inform the reader. The Right Whale, Jaime Serra, 1995, can be seen in the following image:



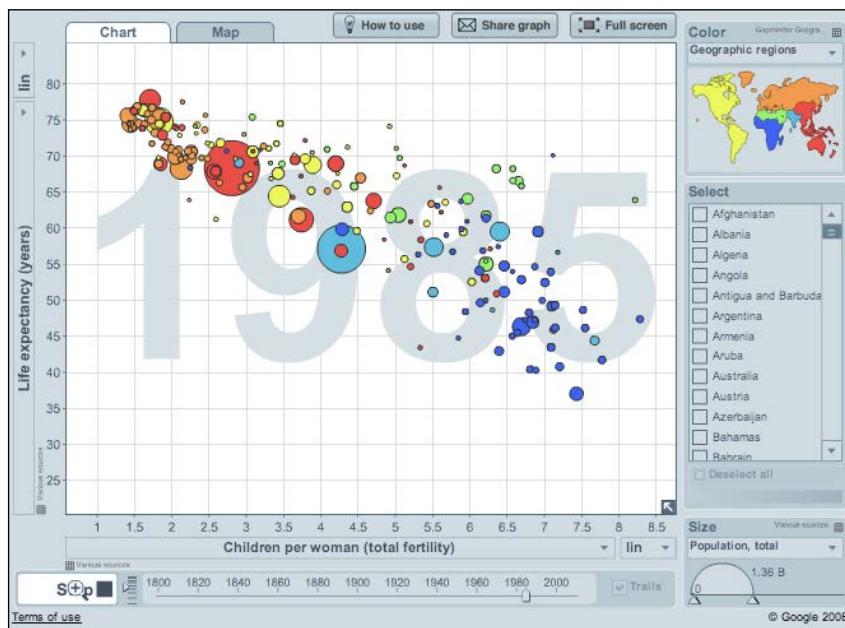
The Right Whale by Jaime Sierra

There are people who don't consider infographics as proper data visualization because they are designed to guide the reader through a story with the main facts already highlighted, as opposed to a chart-based data visualization where the story and the important facts are to be discovered by the reader.

Exploratory visualizations

This branch of data visualization is more focused on providing tools to explore and interpret datasets. These visualizations can be static or interactive. The exploration can be either looking at the charts carefully or to interact with the visualization to discover interesting things. In interactive projects, the user is allowed to filter and interact with the visualizations to discover interesting patterns and facts with little or no guidance. This kind of project is usually regarded as being more objective and data centered than other forms.

A great example is *The Wealth and Health of Nations*, from the Gapminder project (<http://www.gapminder.org/world>). The Gapminder World tool helps us explore the evolution of life in different parts of the world in the last two centuries. The visualization is mainly composed of a configurable bubble chart. The user can select indicators such as life expectancy, fertility rates, and even consumption of sugar per capita and see how different countries have evolved in regard to these indicators. One of the most interesting setups is to select life expectancy in the *y* axis, income per person in the *x* axis, and the size of the bubbles as the size of the population of each country. The bubbles will begin to animate as the years pass, bouncing and making loops as the life expectancy in each country changes. If you explore your own country, you will soon realize that some of the backward movements are related to economic crisis or political problems and how some countries that were formerly similar in their trends in these dimensions diverge. A visualization from Gapminder World, powered by Trendalyzer from www.gapminder.org, is shown in the following screenshot:



The time series for dozens of variables allow the user to explore this dataset, uncover stories, and learn very quickly about how countries that are similar in some regards can be very different in other aspects. The aim of the Gapminder project is to help users and policy makers to have a fact-based view of the world, and the visualization certainly succeeds in providing the means to better understand the world.

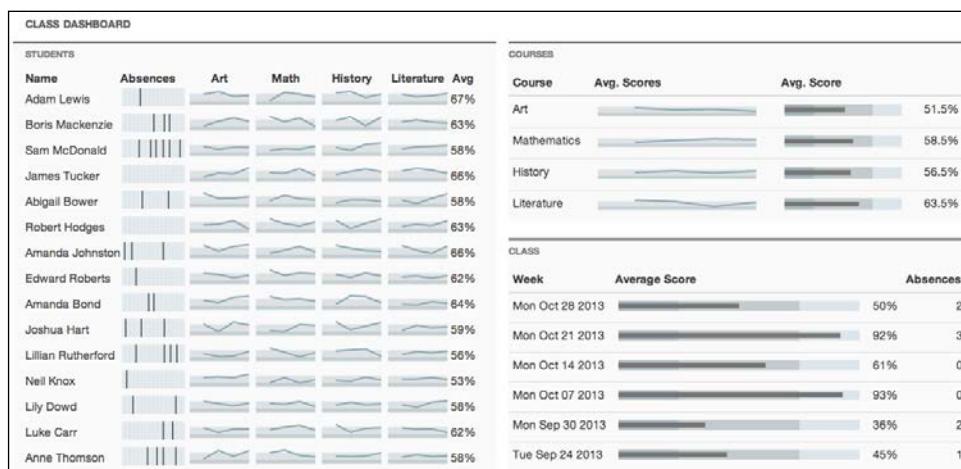
Dashboards

Dashboards are dense displays of charts that help us to understand the key metrics of an issue as quickly and effectively as possible. Business intelligence dashboards and website users' behavior are usually displayed as dashboards. Stephen Few defines an information dashboard as follows:

"A visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance."

The preceding quote can be found in *Information Dashboard Design: The Effective Visual Communication of Data*, Stephen Few, O'Reilly Media.

As the information has to be delivered quickly, there is no time to read long annotations or to click controls; the information should be visible, ready to be consumed. Dashboards are usually bundled with complementary information systems to further investigate issues if they are detected. The distribution of the space in a dashboard is the main challenge when designing them. Compact charts will be preferred in this kind of project, as long as they still allow for speedy decoding of the information. We will learn about designing dashboards in *Chapter 9, Creating a Dashboard*. An example dashboard from *Chapter 9, Creating a Dashboard*, showing the performance of students in a class can be seen in the following screenshot:



This classification mentions only some of the forms of data visualization projects; most parts of data visualizations won't fit exactly under these labels. There is plenty of room to experiment with new formats and borrow elements of infographics, dashboards, and traditional charts to communicate more effectively.

Learning about data visualization

Despite being a young discipline, there are great books on data visualization and information design. A successful data visualization practitioner should also know about design, statistics, cognition, and visual perception, but reading data visualization books is a good start.

Edward Tufte is an expert in information design and his works are a must-read in this field. They are filled with good and bad examples of information design and comments about how to better communicate quantitative information. They contain collections of images from ancient charts and visualizations, which explain their historic context and the impact they had. The discussion is not restricted to how to communicate quantitative information; there are examples ranging from natural history to architecture:

- *Visual Explanations: Images and Quantities, Evidence and Narrative*, Edward R. Tufte, Graphics Press
- *The Visual Display of Quantitative Information*, Edward R. Tufte, Graphics Press
- *Beautiful Evidence*, Edward R. Tufte, Graphics Press
- *Envisioning Information*, Edward R. Tufte, Graphics Press

Stephen Few is a data visualization consultant who specializes in how to display and communicate quantitative information, especially in business environments. His books focus on dashboard and quantitative information and provide actionable guidelines on how to effectively communicate data:

- *Information Dashboard Design: The Effective Visual Communication of Data*, Stephen Few, O'Reilly Series
- *Now You See It: Simple Visualization Techniques for Quantitative Analysis*, Stephen Few, Analytics Press

Alberto Cairo teaches visualization at the University of Miami. He has extensive experience in data journalism and infographics. His most recent book focuses on data visualization and how good infographics are made. He also has a strong presence on social media; be sure to follow him at <http://twitter.com/albertocairo> to be informed about infographics and data visualization:

- *The Functional Art: An introduction to information graphics and visualization*, Alberto Cairo, New Riders

Andy Kirk is a data visualization consultant and author. He recently published a book sharing his experiences in creating data visualizations. He gives guidelines to plan and make the creation of visualizations more systematic. The book is filled with actionable advice about how to design and plan our visualization projects. Andy's blog (<http://www.visualisingdata.com>) is a great source to be informed about the latest developments in the field:

- *Data Visualization: A Successful Design Process*, Andy Kirk, Packt Publishing

There isn't a universal recipe to create good data visualizations, but the experience and guidelines from experts in the field can help us to avoid mistakes and create better visualizations. It will take time to have the necessary skills to create great data visualizations, but learning from experienced people will help us make a safer journey. As with many other things in life, the key to learning is to practice, get feedback, and improve over time.

Introducing the D3 library

In 2011, I was working in a hedge fund, and most of my work consisted of processing and analyzing market data. It mostly consisted of time series, each row containing a timestamp and two prices: the bid and asking prices for stock options. I had to assess the quality of two years of market data and find whether there were errors or gaps between millions of records. The time series were not uniform; there can be hundreds of records in a couple of seconds or just a few records in an hour. I decided to create a bar chart that shows how many records there were in each hour for the two years of data. I created a Python script using the excellent packages NumPy and Matplotlib. The result was a folder with thousands of useless bar charts. Of course, the software was not to blame.

In my second attempt, I tried to create a heat map, where the columns represented hours in a week and the rows represented the weeks of a year. The color of each cell was proportional to the number of quotes in that hour. After tweaking the colors and the size of the cells, my first visualization emerged. Success! The pattern emerged. My coworkers began to gather around, recognizing and explaining the variations on market activity. The black columns at the end of the chart corresponded to weekends, when the market was closed. Mondays were brighter and had more activity than other days. Holidays were easy to spot after a quick consult to the holidays calendar for the year. More interesting patterns were also discernible; there was frantic activity at the beginning of the working day and a slight but noticeable decline at lunch. It was fun and interesting to recognize what we already knew.

However, besides the gaps explained by common sense, there were small gaps that couldn't be explained with holidays or hungry stock traders. There were hours with little or no activity; in the context of a year of market activity, we could see that it was something unusual. A simple heat map allowed us to find the gaps and begin to investigate the anomalies.

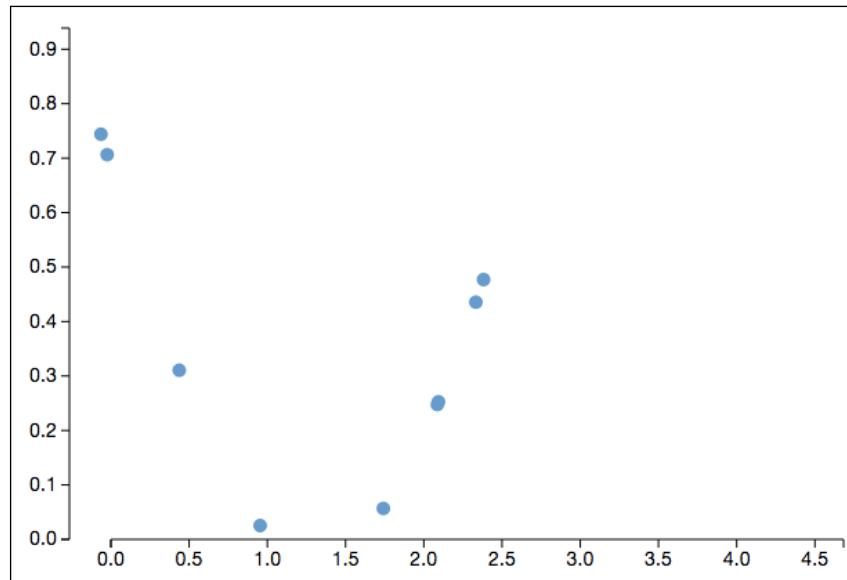
Of course, this first heat map required a better version, one that could allow the exploring of the dataset more easily. We needed an interactive version to know the exact date and time of the gaps and how many records there were in each hourly block. It should also highlight the weekends and holidays. This required better tools, something that allows for more interaction and that doesn't require Python's virtual environments and numerous packages to generate the graphics. This search led me to D3, and I began to learn.

There are several charting packages for web platforms, but D3 excels among them by its flexibility and strong features. A quick visit to the D3 home page (<http://www.d3js.org>) will amaze us with hundreds of examples of what can be done, from the humble bar chart to beautifully crafted interactive maps. Newcomers will soon realize that D3 is not a charting package, but is a tool to bind data items with DOM elements and associate data attributes with visual properties of the DOM elements. This could sound abstract, but this is all we need to create almost any chart.

A chart is a visual representation of a dataset. To create a chart, we must associate attributes of the data items with properties of graphic objects. Let's consider the following dataset:

x	y
2.358820	0.70524774
2.351551	0.71038206
...	...
3.581900	-0.426217726

This series of numbers doesn't have an intrinsic visual representation; we should encode the attributes of each record and assign them corresponding visual attributes. Using the most traditional representation for this kind of data, we can represent the rows as dots on a surface. The position of the dots will be determined by their x and y attributes. Their horizontal position will be proportional to the x attribute and their vertical position will be proportional to the y attribute. This will generate the following scatter plot:



Scatter plot, a visual representation of two-dimensional quantitative data

To help the viewer trace back from position to data attributes, we can add axes, which are essentially annotations for the visual representation of the data. All charts work on the same principle, which is associate visual attributes to data attributes.

With D3, we can manipulate attributes of DOM elements based on attributes of the data items. This is the essence of creating charts. SVG stands for Scalable Vector Graphics, and in most browsers, SVG images can be included in the page and thereby become a part of the DOM. In most cases, we will use svg elements to create charts and other graphic elements. SVG allows us to create basic shapes as rectangles, circles, and lines as well as more complex elements as polygons and text. We can color the elements by assigning them classes and adding CSS styles to the page, or we can use the fill attribute of svg objects. D3 and SVG form a powerful combination, which we will use to create interactive charts and maps.

Of course, there is a price to pay to effectively use these powerful tools. We must learn and understand how browsers work and know our way with JavaScript, CSS, and HTML. One of the fundamentals of D3 is that it manipulates DOM elements, knowing little or nothing about the visual representation of the elements. If we want to create a circle, D3 doesn't provide a `createCircle(x, y, radius)` function, but rather we should append a circle `svg` element in a DOM node (the element with the container ID) and set their attributes:

```
// Appending a circle element to a DOM node
d3.select('#container').append('circle')
  .attr('cx', 10)
  .attr('cy', 10)
  .attr('r', 10);
```

Downloading the example code

You can download the example code files for all Packt books you have purchased 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.

As D3 doesn't know anything else other than the fact that we are appending a DOM element, it is up to us to check whether the parent element is an `svg` element and that `cx`, `cy`, and `r` are valid attributes for a circle.

As we mentioned before, D3 doesn't have ready-to-use charts, but has several tools to make creating visualizations and charts easy. Binding data to DOM elements allows us to create from bar charts to interactive maps by following similar patterns. We will learn how to create reusable charts so that we don't have to code them each time we want to add a chart to a page. For big projects, we will need to integrate our D3-based charts with third-party libraries that support our need, which is out of the D3 scope. We will also learn about how to use D3 in conjunction with external libraries.

Fortunately, D3 has a great community of developers. Mike Bostock, the creator of D3, has created a nice collection of in-depth tutorials about the trickiest parts of D3 and examples demonstrating almost every feature. Users of the library have also contributed with examples covering a wide range of applications.

Summary

In this chapter, we gave a working definition of data visualization, one of the main fields of application of the D3 library.

This book is about D3 and how to create interactive data visualizations in real-life settings. We will learn about the inner working of D3 and create well-structured charts to be used and shared across projects. We will learn how to create complete applications using D3 and third-party libraries and services as well as how to prepare our development environment to have maintainable and comfortable workflows.

Learning D3 may take some time, but it's certainly rewarding. The following chapters are focused on providing the tools to learn how to use D3 and other tools to create beautiful charts that will add life to your data.

2

Reusable Charts

In this chapter, we will learn how to create configurable charts and layout algorithms and how to use these components. One important characteristic of configurable charts is that we can use them in different contexts without having to change the code. In this chapter, we will cover the following topics:

- Learn how to create reusable charts
- Create a reusable barcode chart
- Create a reusable layout algorithm
- Use the layout and the barcode chart

We will begin by defining what we understand by reusable charts and construct a reusable chart from scratch.

Creating reusable charts

A chart is a visual representation of a dataset. As datasets grow in complexity and number of elements, we might want to encapsulate the chart in a reusable piece of code to share it and use it in other projects. The success of the reusability of the chart depends on the choices made during the design process.

We will follow the proposal given in the article *Towards Reusable Charts* by Mike Bostock with some modifications. Charts created with this model will have the following attributes:

- **Configurable:** The chart appearance and behavior can be modified without having to change the code of the chart. The chart should expose an API that allows customization.

- **Repeatable:** A chart can be rendered in a selection that contains several elements, displaying independent visualizations in each node. The update process and the variables associated to the data items must be independent as well.
- **Composable:** A consequence of the previous attributes is that a chart can have inner charts as long as the inner charts follow the same conventions.

We must evaluate which aspects of the chart must be configurable and which aspects are fixed, because adding more flexibility increases the complexity of the chart.

There are other ways to implement reusability. For instance, the D3.Chart package by the Miso Project proposes a model that allows extension of the charts in terms of existing charts, add event-listening capabilities to the charts, and includes the mini framework D3.layer that allows us to configure life cycle events without changing the rendering logic of the chart.

We will begin by reviewing the creation and data-binding processes in D3 in order to construct a reusable chart step by step.

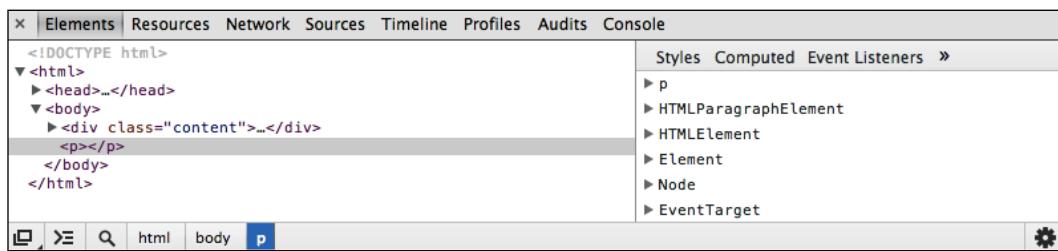
Creating elements with D3

In this section, we will review the mechanism of creation and manipulation of data-bound DOM elements using D3. To follow the examples in this section, open the chapter02/01-creating-dom-elements.html file and open the browser's Developer Tools.

We can use D3 to create and modify the elements in a web page. In the Developer Tools Console, we can create a new paragraph at the end of the body with a single command as follows:

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

Inspecting the document structure, we can see that an empty paragraph element was appended at the end of the body. The appended method returns a selection that contains the new paragraph element. Refer to the following screenshot:



As we stored a reference to the paragraph selection in the `p` variable, we can use this reference to modify the content of the paragraph as follows:

```
> p.html('New paragraph.');
```

The `html` method will also return the paragraph selection, allowing us to use method chaining to modify the paragraph font color as follows:

```
> p.html('blue paragraph').style('color', 'blue');
```

The `style` method will once more return the paragraph selection.

Binding data

Before creating a chart, we will create a set of divs bound to a simple dataset and improve the example step-by-step in order to understand the process of creating a chart. We will begin with a data array that contains three strings and use D3 to create three corresponding div elements, each one with a paragraph that contains the strings. In the example file, we have a div element classed `chart-example` and with ID `chart`. This div element will be used as container for the divs to be created:

```
<div class="chart-example" id="chart"></div>
```

In the script, we define our dataset as an array with three strings:

```
var data = ['a', 'b', 'c'];
```

For each element, we want to append an inner div, and in each one of them, we append a paragraph. The container div can be selected using its ID:

```
var divChart = d3.select('#chart');
```

Next, we create a selection that contains the divs classed `data-item` and bind the data array to this selection:

```
var divItems = divChart.selectAll('div.data-item')
  .data(data);
```

Before invoking the `data` method, keep in mind that the inner divs don't exist yet, and the selection will be empty. We can check this by creating an empty selection:

```
// This will return true
> divChart.selectAll('p.test-empty-selection').empty()
```

The `data()` method joins the data array with the current selection. At this point, the data elements in the `divItems` selection don't have corresponding DOM nodes. The `divItems.enter()` selection stores placeholder DOM nodes for the elements to be created. We can instantiate new divs using the `append` method:

```
divItems.enter()  
  .append('div')  
  .attr('class', 'data-item');
```

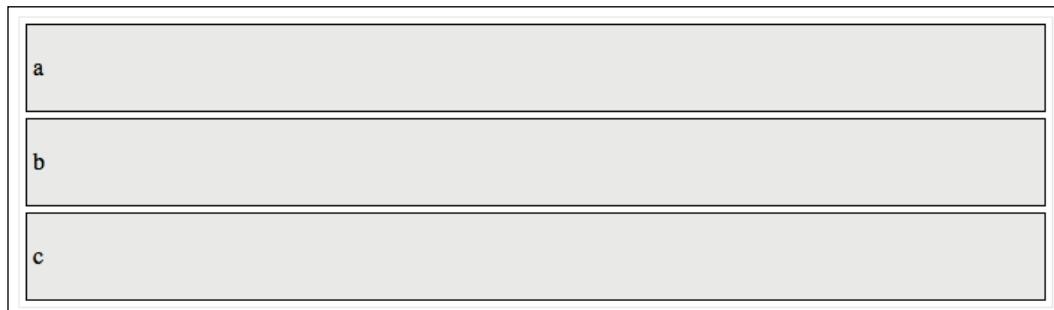
As seen in the preceding example, the `append` method returns the created divs, and we can set its class directly. The `divItems` selection now contains three div elements, each one bounded to a data item. We can append a paragraph to each div and set its content:

```
var pItems = divItems.append('p')  
  .html(function(d) { return d; });
```

The method-chaining pattern allows us to write the same sequence of operations in one single, compact statement:

```
d3.select('#chart').selectAll('div.data-item')  
  .data(data)  
  .enter()  
  .append('div')  
  .attr('class', 'data-item')  
  .append('p')  
  .html(function(d) { return d; });
```

The div elements and their corresponding paragraphs are created inside the container div. Refer to the following screenshot:



The creation of DOM elements by joining a data array with a selection of div elements

Encapsulating the creation of elements

We could add more elements on top of the previous elements, but the code would become confusing and monolithic. The `selection.call` method allows us to encapsulate the creation of the div contents as follows:

```
d3.select('#chart').selectAll('div.data-item')
  .data(data)
  .enter()
  .append('div')
  .attr('class', 'data-item')
  .call(function(selection) {
    selection.each(function(d) {
      d3.select(this).append('p').html(d);
    });
  });
}
```

The `call` method invokes its argument function, passing the current selection as the argument and setting the `this` context to the current selection.

The `selection.each` method invokes its own argument function, passing the bound data item to the function. The `this` context in the function is the current DOM element; in our case, it's the recently created div.

The argument of the `call` method can be defined elsewhere. We will define the `initDiv` function and use it to create the content of the div:

```
function initDiv(selection) {
  selection.each(function(data) {
    d3.select(this).append('p')
      .html(data);
  });
}
```

The `initDiv` function encapsulates the creation and configuration of the div contents; the code that creates the elements is more compact, shown as follows:

```
d3.select('#chart').selectAll('div.data-item')
  .data(data)
  .enter()
  .append('div')
  .attr('class', 'data-item')
  .call(initDiv);
```

Creating the svg element

We can use the same structure to create an `svg` element inside the inner `div` instead of the paragraph. We will need to define the width and height of `svg` as follows:

```
var width = 400,  
    height = 40;
```

We will change the name of the `initDiv` function to the more appropriate name, `chart`, and replace the creation of the paragraph with the creation of `svg` and a background rectangle. Note that we are binding the `svg` selection to an array with a single element (the `[data]` array) and appending the `svg` and the rectangle only to the `enter()` selection as follows:

```
function chart(selection) {  
  selection.each(function(data) {  
    // Select and bind the svg element.  
    var div = d3.select(this).attr('class', 'data-item'),  
        svg = div.selectAll('svg').data([data]),  
        svgEnter = svg.enter();  
  
    // Append the svg and the rectangle on enter.  
    svgEnter.append('svg')  
      .attr('width', width)  
      .attr('height', height)  
      .append('rect')  
      .attr('width', width)  
      .attr('height', height)  
      .attr('fill', 'white');  
  });  
}
```

The code to create the `svg` element in each `div` remains the same except for the name of the `chart` function, shown as follows:

```
d3.select('#chart').selectAll('div.data-item')  
  .data(data)  
  .enter()  
  .append('div')  
  .attr('class', 'data-item')  
  .call(chart);
```

The barcode chart

A barcode chart displays a series of discrete events in a given time interval, showing the occurrence of each event with a small vertical bar. It uses the position of the bars as the principal visual variable, giving the reader a clear idea of the distribution of events in time. It might have a time axis or it might represent a fixed time interval. As it is a very compact display, a barcode chart can be integrated along with the text in a paragraph or in a table, giving context and allowing parallel comparison. Refer to the following screenshot:

Name	Today Mentions	mentions/hour
AAPL		34.3
MSFT		11.1
GOOG		19.2
NFLX		6.7

An example of barcode charts in a table

In the previous section, we created an `svg` element using a charting function. In that implementation, the width and height are global variables, and as we all know, global variables are evil. A chart can have dozens of configurable values, and we can't depend on the user to define each one of them properly. To provide default values and encapsulate the chart-related variables, we will create a closure (the `chart` function) and define the variables in its private scope, as follows:

```
// Closure to create a private scope for the charting function.
var barcodeChart = function() {

    // Definition of the chart variables.
    var width = 600,
        height = 30;

    // Charting function.
    function chart(selection) {
        selection.each(function(data) {
            // Bind the dataset to the svg selection.
            var div = d3.select(this),
```

```
svg = div.selectAll('svg').data([data]);  
  
    // Create the svg element on enter, and append a  
    // background rectangle to it.  
    svg.enter()  
        .append('svg')  
        .attr('width', width)  
        .attr('height', height)  
        .append('rect')  
        .attr('width', width)  
        .attr('height', height)  
        .attr('fill', 'white');  
    );  
}  
  
return chart;  
};
```

Note that the `barcodeChart` function returns an instance of the `chart` function, which will be used to create the chart later. Refer to the following code:

```
// The Dataset  
var data = ['a', 'b', 'c'];  
  
// Get the charting function.  
var barcode = barcodeChart();  
  
// Bind the data array with the data-item div selection, and call  
// the barcode function on each div.  
d3.select('#chart').selectAll('div.data-item')  
    .data(data)  
    .enter()  
    .append('div')  
    .attr('class', 'data-item')  
    .call(barcode);
```

Accessor methods

The `width` and `height` are attributes of the `chart` function, and the `barcode` function has access to these variables. To allow the user to configure the chart attributes, we will add accessor methods to the chart. We will also add a `margin` attribute as follows:

```
var barcodeChart = function() {  
  
    // Chart Variables.Attributes
```

```
var width = 600,
    height = 30,
    margin = {top: 5, right: 5, bottom: 5, left: 5};

function chart(selection) {
    // Chart creation ...
}

// Accessor function for the width
chart.width = function(value) {
    if (!arguments.length) { return width; }
    width = value;
    // Returns the chart to allow method chaining.
    return chart;
};

// Accessor functions for the height and the margin ...

return chart;
};
```

Note that when invoked without arguments, the accessor method will return the variable value. When setting the value, the accessor method will set the value and return the chart. This allows us to call other accessors using method chaining, as follows:

```
// Create and configure the chart.
var barcode = barcodeChart()
    .width(600)
    .height(25);
```

Chart initialization

In the chart function, we can use the `call` method to encapsulate the initialization code. We will add the `chart.svgInit` method, which will be in charge of setting the size of the `svg` element, create a container group for the chart, and add the background rectangle as follows:

```
// Initialize the SVG Element

function svgInit(svg) {
    // Set the SVG size
    svg
        .attr('width', width)
```

```
.attr('height', height);

// Create and translate the container group
var g = svg.append('g')
    .attr('class', 'chart-content')
    .attr('transform', 'translate(' + [margin.top,
margin.left] + ')');

// Add a background rectangle
g.append('rect')
    .attr('width', width - margin.left - margin.right)
    .attr('height', height - margin.top - margin.bottom)
    .attr('fill', 'white');
};
```

In the chart function, we call the `svgInit` function that passes the appended `svg` element. The `chart` function is more compact. Refer to the following code:

```
function chart(selection) {
  selection.each(function(data) {
    // Bind the dataset to the svg selection.
    var div = d3.select(this),
        svg = div.selectAll('svg').data([data]);

    // Call thesvgInit function on enter.
    svg.enter()
      .append('svg')
      .call(svgInit);
  });
}
```

With the chart structure ready, we can proceed to draw the bars.

Adding data

We will generate a data array by repeatedly adding a random number of seconds to a date. To compute these random number of seconds, we will generate a random variable of exponential distribution. The details about how to generate the random variable are not important; just remember that the `randomInterval` function returns a random number of seconds as follows:

```
// Compute a random interval using an Exponential Distribution
function randomInterval(avgSeconds) {
  return Math.floor(-Math.log(Math.random()) * 1000 * avgSeconds);
};
```

We will create a function that returns an array with objects that have increasing random dates:

```
// Create or extend an array of increasing dates by adding
// a number of random seconds.
function addData(data, numItems, avgSeconds) {
    // Compute the most recent time in the data array.
    var n = data.length,
        t = (n > 0) ? data[n - 1].date : new Date();

    // Append items with increasing times in the data array.
    for (var k = 0; k < numItems - 1; k += 1) {
        t = new Date(t.getTime() + randomInterval(avgSeconds));
        data.push({date: t});
    }

    return data;
}
```

Invoking the function with an empty array as the first argument will generate the initial data with 150 elements, with an average of 300 seconds between each date:

```
var data = addData([], 150, 300);
```

The structure of the data array will be something like the following code:

```
data = [
    {date: Tue Jan 01 2013 09:48:52 GMT-0600 (PDT)},
    {date: Tue Jan 01 2013 09:49:14 GMT-0600 (PDT)},
    ...
    {date: Tue Jan 01 2013 21:57:31 GMT-0600 (PDT)}
]
```

With the dataset ready, we can modify the chart function to draw the bars. First, we compute the horizontal scale, select the container group, and create a selection for the bars. Refer to the following code:

```
function chart(selection) {
    selection.each(function(data) {
        // Creation of the SVG element ...

        // Compute the horizontal scale.
        var xScale = d3.time.scale()
            .domain(d3.extent(data, function(d) { return d.date;
        }))
```

```
.range([0, width - margin.left - margin.right]);  
  
    // Select the containing group  
    var g = svg.select('g.chart-content');  
  
    // Bind the data to the lines selection.  
    var bars = g.selectAll('line')  
        .data(data, function(d) { return d.date; });  
  
        // Append the bars on the enter selection ...  
    });  
}
```

Each bar should be associated with a date, so we configure the key function to return the date. We append the line elements on enter and set the initial position and stroke of the bars as follows:

```
// Append the bars on the enter selection  
bars.enter().append('line')  
    .attr('x1', function(d) { return xScale(d.date); })  
    .attr('x2', function(d) { return xScale(d.date); })  
    .attr('y1', 0)  
    .attr('y2', height - margin.top - margin.bottom)  
    .attr('stroke', '#000')  
    .attr('stroke-opacity', 0.5);
```

We set the `stroke-opacity` attribute to 0.5, so we can see the overlapping lines. Finally, the barcode chart has some bars, as shown in the following screenshot:



The first version of the barcode chart

Adding the date accessor function

The current implementation of the chart assumes that the dataset contains objects with the `date` attribute. This is an inconvenience, because the user could have a data array with the date information in an attribute named `time`, or the user might need to process other attributes to compute a valid date. We will add a configurable accessor for the date as follows:

```
var barcodeChart = function() {  
  
    // Set the default date accessor function
```

```
var value = function(d) { return d.date; };

// chart function ...

// Accessor for the value function
chart.value = function(accessorFunction) {
    if (!arguments.length) { return value; }
    value = accessorFunction;
    return chart;
};

return chart;
};
```

We need to replace the references to `d.date` with invocations to the `value` method in the `chart` function as follows:

```
function chart(selection) {
    selection.each(function(data) {

        // Creation of the SVG element ...

        // Compute the horizontal scale using the date accessor.
        var xScale = d3.time.scale()
            .domain(d3.extent(data, value))
            .range([0, width - margin.left - margin.right]);
        // ...

        // Bind the data to the bars selection.
        var bars = g.selectAll('line').data(data, value);

        // Create the bars on enter and set their attributes, using
        // the date accessor function.
        bars.enter().append('line')
            .attr('x1', function(d) { return xScale(value(d)); })
            .attr('x2', function(d) { return xScale(value(d)); })
            // set more attributes ...
            .attr('stroke-opacity', 0.5);
    });
}
```

A user who has the date information in the `time` attribute can use the chart by setting the `value` accessor without modifying the chart code or the data array:

```
// This will work if the array of objects with the time attribute.
var barcode = barcodeChart()
    .value(function(d) { return d.time; });
```

A barcode chart must represent a fixed time interval, but right now, the chart shows all the bars. We would like to remove the bars that are older than a certain time interval. We will then add the `timeInterval` variable:

```
// Default time interval.  
var timeInterval = d3.time.day;
```

Add the corresponding accessor method:

```
// Time Interval Accessor  
chart.timeInterval = function(value) {  
    if (!arguments.length) { return timeInterval; }  
    timeInterval = value;  
    return chart;  
};
```

Then, update the horizontal scale in the `chart` function:

```
// Compute the first and last dates of the time interval  
var lastDate = d3.max(data, value),  
    firstDate = timeInterval.offset(lastDate, -1);  
  
// Compute the horizontal scale with the correct domain  
var xScale = d3.time.scale()  
    .domain([firstDate, lastDate])  
    .range([0, width - margin.left - margin.right]);
```

The chart width represents the default time interval, and the user can set the time interval by using the `timeInterval` accessor method:

```
var barcode = barcodeChart()  
    .timeInterval(d3.time.day);
```

The barcode chart length represents 24 hours. The dataset contains events covering about 11 hours as shown in the following barcode:



Updating the dataset

In most parts of the applications, the dataset is not static. The application might poll the server every couple of minutes, receive a stream of data items, or update the data on user request. In the case of a barcode chart, the new items will probably have more recent dates than the existing data items. The barcode chart is supposed to display the most recent item at the right-hand side of the chart, moving the old bars to the left-hand side. We can do this by updating the position of the bars as follows:

```
// Create the bars on enter ...

// Update the position of the bars.
bars.transition()
.duration(300)
.attr('x1', function(d) { return xScale(value(d)); })
.attr('x2', function(d) { return xScale(value(d)); });
```

The transitions aren't just to make the chart look pretty; they allow the user to follow the objects as they change. This is called **object constancy**. If we just move the bars instantly, the user might have difficulty understanding what happened with the old bars or realizing whether the chart changed at all. We will add a button to add items to the dataset in the page as follows:

```
<button id="btn-update">Add data</button>
<div class="chart-example" id="chart"></div>
```

We can use D3 to configure the callback for the click event of the button. The callback function will add 30 new items to the dataset (with 180 seconds between them on an average) and rebind the selection to the updated dataset as follows:

```
d3.select('#btn-update')
.on('click', function() {
    // Add more random data to the dataset.
    data = addData(data, 30, 3 * 60);
    // Rebind the data-item selection with the updated dataset.
    d3.select('#chart').selectAll('div.data-item')
        .data([data])
        .call(barcode);
});
```

Fixing the enter and exit transitions

If we click on the button a couple of times, we will see the new bars appear suddenly, and then, the existing bars shifting to the left-hand side. We would expect the new bars to enter by the right-hand side, moving all the bars together to the left-hand side. This can be achieved by adding the new bars using `xScale` as it was before adding the new elements and then updating the position of all the bars at the same time.

This strategy will work, except that we didn't store the previous state of `xScale`.

We do, however, have access to the data before appending the new elements.

We can access the data bounded to the selection of lines as follows:

```
// Select the chart group and the lines in that group
var g = svg.select('g.chart-content'),
    lines = g.selectAll('line');
```

The first time we use the chart, the `lines` selection will be empty; in this case, we need to use the most recent item of the data array to compute the last date. If the selection isn't empty, we can use the previous most recent date. We can use the `selection.empty` method to check whether or not the chart contains bars as follows:

```
// Compute the most recent date from the dataset.
var lastDate = d3.max(data, value);

// Replace the lastDate with the most recent date of the
// dataset before the update, if the selection is not empty.
lastDate = lines.empty() ? lastDate : d3.max(lines.data(), value);

// Compute the date of the lastDate minus the time interval.
var firstDate = timeInterval.offset(lastDate, -1);

// Compute the horizontal scale with the new extent.
var xScale = d3.time.scale()
    .domain([firstDate, lastDate])
    .range([0, width - margin.left - margin.right]);
```

We can bind the data now, and we can create the new bars and set their position with the old scale:

```
// Select the lines and bind the new dataset to it.
var bars = g.selectAll('line').data(data, value);

// Create the bars on enter
bars.enter().append('line')
    .attr('x1', function(d) { return xScale(value(d)); })
    // set more attributes ...
    .attr('stroke-opacity', 0.5);
```

Once the new bars were appended, we can update the `xScale` domain to include the most recent items and update the position of all the bars:

```
// Update the scale with the new dataset.
lastDate = d3.max(data, value);
firstDate = timeInterval.offset(lastDate, -1);
xScale.domain([firstDate, lastDate]);

// Update the position of the bars, with the updated scale.
bars.transition()
.duration(300)
.attr('x1', function(d) { return xScale(value(d)); })
.attr('x2', function(d) { return xScale(value(d)); });
```

The last thing to do is to remove the bars that don't have corresponding data items, fading them by changing their stroke opacity to zero:

```
// Remove the bars that don't have corresponding data items.
bars.exit().transition()
.duration(300)
.attr('stroke-opacity', 0)
.remove();
```

A basic version of the barcode chart is now ready. There are some additional attributes of the chart that we might want to configure; a user might want to change the color of the bars, their opacity, the duration of the transitions, or the color of the background rectangle.

Using the barcode chart

In this section, we will use the barcode chart with a more complex dataset and learn how to use the chart that is integrated within a table. Imagine that we have an application that monitors the mention of stocks on Twitter. One element of this fictional application might be a table that displays the aggregated information about the stock's mentions and the barcode chart with the mentions of the last day. We will assume that the data is already loaded on the page. Each data item will have the name of the stock, an array with mentions, and the average of mentions by hour. Refer to the following code:

```
var data = [
  {name: 'AAPL', mentions: [...], byHour: 34.3},
  {name: 'MSFT', mentions: [...], byHour: 11.1},
  {name: 'GOOG', mentions: [...], byHour: 19.2},
  {name: 'NFLX', mentions: [...], byHour: 6.7}
];
```

The `mentions` array will have objects with the `date` attribute. These items can have other attributes as well. We will create the table structure with D3, binding the rows of the table body to the data array. We create the table by binding the `table` element with a single element array as follows:

```
// Create a table element.  
var table = d3.select('#chart').selectAll('table')  
    .data([data])  
    .enter()  
    .append('table')  
    .attr('class', 'table table-condensed');
```

We append the table head and body:

```
// Append the table head and body to the table.  
var tableHead = table.append('thead'),  
    tableBody = table.append('tbody');
```

We add three cells in the row header to display the column headers:

```
// Add the table header content.  
tableHead.append('tr').selectAll('th')  
    .data(['Name', 'Today Mentions', 'mentions/hour'])  
    .enter()  
    .append('th')  
    .text(function(d) { return d; });
```

We append one row to the table body for each element in the data array:

```
// Add the table body rows.  
var rows = tableBody.selectAll('tr')  
    .data(data)  
    .enter()  
    .append('tr');
```

For each row, we need to add three cells, one with the stock name, one with the barcode chart, and the last one with the hourly average of mentions. To add the name, we simply add a cell and set the text:

```
// Add the stock name cell.  
rows.append('td')  
    .text(function(d) { return d.name; });
```

Now, we add a cell with the chart. The data item bound to the row is not an array with dates, so we can't call the barcode function directly. Using the `datum` method, we can bind the data item to the `td` element. Note that this method does not perform a join, and thus, it doesn't have the `enter` and `exit` selections:

```
// Add the barcode chart.
rows.append('td')
  .datum(function(d) { return d.mentions; })
  .call(barcode);
```

The `datum` method receives a data item directly; it doesn't require an array like the `data` method. Finally, we add the last cell with the hourly average of mentions. The content of this cell is a number, so it must be aligned to the right-hand side:

```
// Add the number of mentions by hour, aligned to the right.
rows.append('td').append('p')
  .attr('class', 'pull-right')
  .html(function(d) { return d.byHour; });
```

The barcode charts are integrated in the table, along with other information about the stock mentions as shown in the following screenshot:

Name	Today Mentions	mentions/hour
AAPL		34.3
MSFT		11.1
GOOG		19.2
NFLX		6.7

The barcode chart, integrated within a table, displaying fictional Twitter mentions of stocks

We used D3 to create a data-bound table with a chart in each row. We could have created the structure and header of the table in the HTML document and bound the data array to the rows in the table body, but we created the entire table with D3, instead.

If the table will be used in more than one page, we can also think of creating the table as a reusable chart, using the structure presented in the previous section. We could even add an attribute and an accessor to set the charting function and use the table chart with a different chart without having to change the code of the table chart.

Creating a layout algorithm

Every chart makes assumptions about the kind and structure of the data that they can display. A scatter plot needs pairs of quantitative values, a bar chart requires categories with a quantitative dimension, and a tree map needs nested objects. To use a chart, the user will need to group, split, or nest the original dataset to fit the chart requirements. Functions that perform these transformations are called **layout algorithms**. D3 already provides a good set of layouts, from the simple pie layout to the more complex force layout. In this section, we will learn how to implement a layout algorithm, and we will use it to create a simple visualization using the barcode dataset.

The radial layout

The array with dates used in the barcode example can be visualized in several ways. The barcode chart represents every data item as a small bar in a time interval. Another useful way to display a series of events is to group them in intervals. The most common among these kinds of visualizations is a bar chart, with one bar for each time interval and the height of each bar representing the number of events that occurred in the corresponding time interval.

A **radial chart** is a circular arrangement of arc segments, each one representing a category. In this chart, each arc has the same angle, and the area of each arc is proportional to the number of items in the category. We will create a radial layout that groups and counts the events in hourly segments and compute the start and end angles for each arc.

The purpose of a layout algorithm is to allow the user to easily transform its dataset to the format required by a chart. The layout usually allows a certain amount of customization. We will implement the layout function as a closure with accessors to configure the layout behavior as follows:

```
var radialLayout = function() {  
  
    // Layout algorithm.  
    function layout(data) {  
        var grouped = [];  
        // Transform and returns the grouped data ...  
        return grouped;  
    }  
    return layout;  
};
```

The usage of a layout is similar to the usage of the barcode chart. First, we invoke `RadialLayout` to get the layout function and then call the layout with the dataset as the argument in order to obtain the output dataset. We will generate an array of random dates using the `addData` function from the previous section:

```
// Generate a dataset with random dates.  
var data = addData([], 300, 20 * 60);  
  
// Get the layout function.  
var layout = radialLayout();  
  
// Compute the ouput data.  
var output = layout(data);
```

We need the layout to group and count the data items for each hour and to compute the start and end angles for each arc. To make the counting process easier, we will use a map to temporarily store the output items. D3 includes `d3.map`, a dictionary-like structure that provides key-value storage:

```
function layout(data) {  
    // Create a map to store the data for each hour.  
    var hours = d3.range(0, 24),  
        gmap = d3.map(),  
        groups = [];  
  
    // Append a data item for each hour.  
    hours.forEach(function(h) {  
        gmap.set(h, {hour: h, startAngle: 0, endAngle: 0, count: 0});  
    });  
  
    // ...  
  
    // Copy the values of the map and sort the output data array.  
    groups = gmap.values();  
    groups.sort(function(a, b) { return a.hour > b.hour ? 1 : -1; });  
    return groups;  
}
```

As the layout must return an array, we will need to transfer the map values to the groups array and sort it to return it as the output. The output items don't have any useful information yet:

```
[  
  {hour: 0, startAngle: 0, endAngle: 0, count: 0},  
  ...  
  {hour: 23, startAngle: 0, endAngle: 0, count: 0}  
]
```

The next thing to do is to count the items that belong to each hour. To do this, we iterate through the input data and compute the hour of the date attribute:

```
// Count the items belonging to each hour  
data.forEach(function(d) {  
  // Get the hour from the date attribute of each data item.  
  var hour = d.date.getHours();  
  
  // Get the output data item corresponding to the item  
  // hour.  
  var val = gmap.get(hour);  
  
  // We increment the count and set the value in the map.  
  val.count += 1;  
  gmap.set(hour, val);  
});
```

At this point, the output contains the count attribute with the correct value. As we did in the barcode chart, we will add a configurable accessor function to retrieve the date attribute:

```
var radialLayout = function() {  
  // Default date accessor  
  var value = function(d) { return d.date; }  
  
  function layout(data) {  
    // Content of the layout function ...  
  }  
  
  // Date Accessor Function  
  layout.value = function(accessorFunction) {  
    if (!arguments.length) { return value; }  
    value = accessorFunction;  
    return layout;  
  };
```

In the layout function, we replace the references to `d.date` with invocations to the date accessor method, `value(d)`. The user now can configure the date accessor function with the same syntax as that in the barcode chart:

```
// Create and configure an instance of the layout function.
var layout = radialLayout()
    .value(function(d) { return d.date; });
```

Computing the angles

With the `count` attribute ready, we can proceed to compute the start and end angles for each output item. The angle for each arc will be the same, so we can compute `itemAngle` and then iterate through the `hours` array as follows:

```
// Compute equal angles for each hour item.
var itemAngle = 2 * Math.PI / 24;

// Adds a data item for each hour.
hours.forEach(function(h) {
  gmap.set(h, {
    hour: h,
    startAngle: h * itemAngle,
    endAngle: (h + 1) * itemAngle,
    count: 0
  });
});
```

The output dataset now has the start and end angles set. Note that each data item has a value that is 1/24th of the circumference:

```
[
  {hour: 0, startAngle: 0, endAngle: 0.2618, count: 7},
  {hour: 1, startAngle: 0.2618, endAngle: 0.5236, count: 14},
  ...
  {hour: 23, startAngle: 6.0214, endAngle: 6.2832, count: 17}
]
```

Here, we used the entire circumference, but a user might want to use a semicircle or want to start in a different angle. We will add the `startAngle` and `endAngle` attributes and the `angleExtent` accessor method in order to allow the user to set the angle extent of the chart:

```
var radialLayout = function() {

  // Default values for the angle extent.
  var startAngle = 0,
```

```
endAngle = 2 * Math.PI;

// Layout function ...

// Angle Extent
layout.angleExtent = function(value) {
    if (!arguments.length) { return value; }
    startAngle = value[0];
    endAngle = value[1];
    return layout;
};

};
```

We need to change the `itemAngle` variable in order to use the new angle range. Also, we add the layout start angle to the start and end angles for each output item:

```
// Angle for each hour item.
var itemAngle = (endAngle - startAngle) / 24;

// Append a data item for each hour.
hours.forEach(function(h) {
    gmap.set(h, {
        hour: h,
        startAngle: startAngle + h * itemAngle,
        endAngle: startAngle + (h + 1) * itemAngle,
        count: 0
    });
});
```

We can configure the start and end angles of the layout to use a fraction of the circumference:

```
// Create and configure the layout function.
var layout = radialLayout()
    .angleExtent([Math.PI / 3, 2 * Math.PI / 3]);
```

In this section, we implemented a simple layout algorithm that counts and groups an array of events in hours and computes the start and end angles to display the returned value as a radial chart. As we did in the barcode chart example, we implemented the layout as a closure with getter and setter methods.

Using the layout

In this section, we will use the radial layout to create a radial chart. To keep the code simple, we will create the visualization without creating a chart function. We begin by creating a container for the radial chart:

```
<div class="chart-example" id="radial-chart"></div>
```

We define the visualization variables and append the `svg` element to the container. We append a group and translate it to the center of the `svg` element:

```
// Visualization Variables
var width = 400,
    height = 200,
    innerRadius = 30,
    outerRadius = 100;

// Append a svg element to the div and set its size.
var svg = d3.select('#radial-chart').append('svg')
    .attr('width', width)
    .attr('height', height);

// Create the group and translate it to the center.
var g = svg.append('g')
    .attr('transform', 'translate(' + [width / 2, height / 2] + ')');
```

We represent each hour as an arc. To compute the arcs, we need to create a radius scale:

```
// Compute the radius scale.
var rScale = d3.scale.sqrt()
    .domain([0, d3.max(output, function(d) { return d.count; })])
    .range([2, outerRadius - innerRadius]);
```

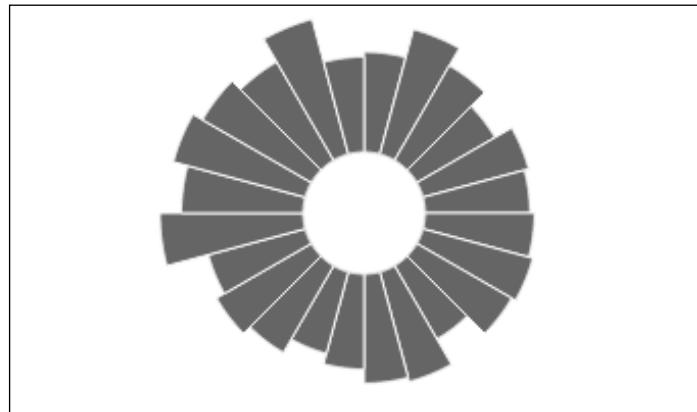
As we have the angles and the radius, we can configure the `d3.svg.arc` generator to create the arc paths for us. The arc generator will use the `startAngle` and `endAngle` attributes to create the arc path:

```
// Create an arc generator.
var arc = d3.svg.arc()
    .innerRadius(innerRadius)
    .outerRadius(function(d) {
        return innerRadius + rScale(d.count);
});
```

The arc function receives objects with `startAngle`, `endAngle`, and `count` attributes and returns the path string that represents the arc. Finally, we select the path objects in the container group, bind the data, and append the paths:

```
// Append the paths to the group.  
g.selectAll('path')  
  .data(output)  
  .enter()  
  .append('path')  
    .attr('d', function(d) { return arc(d); })  
    .attr('fill', 'grey')  
    .attr('stroke', 'white')  
    .attr('stroke-width', 1);
```

The radial chart represents the number of items in each hour as radial arcs. Refer to the following screenshot:



We have shown you how to use the radial layout to create a simple visualization. As we mentioned previously, the layout can be used to create other charts as well. For instance, if we ignore the start and end angles, we can use the radial layout to create a bar chart or even use the output data to create a table with the data aggregated by hour.

Summary

In this chapter, we learned how to create a reusable chart and how to add configuration methods to it so that the chart can be used in several projects without having to change its code in order to use it. We created a barcode chart and used it with data with a different format. We also learned how to create a reusable layout algorithm and how to use it to transform the data source to the format expected by a chart.

In the next chapter, we will learn how to create data visualizations in D3 for browsers without SVG support using canvas and div elements.

3

Creating Visualizations without SVG

Most of the visualizations created with D3 use the SVG element. SVG graphics are resolution independent, which means that they can be scaled without pixelation, and they are relatively easy to create with D3. The SVG elements are also part of the DOM tree and allow us to select and manipulate individual elements of the figures and to change their attributes, triggering an automatic update by the browser.

There are a significant number of users who don't use a browser with SVG support, and sometimes, we can't just forget them. In this chapter, we will examine alternatives that provide visualizations without using SVG. We will create a visualization using only the div elements, discuss libraries that provide SVG support for older browsers, and show an example of integrating D3 and the canvas element.

SVG support in the browser market

The global browser market has a good support for SVG, both in mobile and desktop browsers. There is, however, a significant portion of users who don't enjoy SVG support; the most notable examples are the users of Internet Explorer 8.0 and older as well as users of the stock browser of Android under 3.0.

According to <http://caniuse.com/>, about 86 percent of the global browser market has basic SVG support (as of May 2014). Most of the applications can't afford to leave 14 percent of their users behind. With these users in mind, we will learn how to create data visualizations without using SVG and how to add SVG support to the browser using polyfilling. You can check a more up-to-date version of this and other tables on *Can I use...* (<http://caniuse.com/#feat=svg>).

Visualizations without SVG

In this section, we will create a visualization without using SVG. We will create a bubble chart to show the usage of browser versions in the browser market. A circle will represent each browser version; the area of the circle will be proportional to the global browser usage. Each browser will be assigned a different color. We will use the force layout to group the bubbles on the screen. To follow the examples, open the chapter03/01-bubble-chart.html file.

Loading and sorting the data

To make the processing easier, the browser market data was arranged in a JSON file. The main object will have a name that describes the dataset and an array that will contain the data for each browser version. Refer to the following code:

```
{  
  "name": "Browser Market",  
  "values": [  
    {  
      "name": "Internet Explorer",  
      "version": 8,  
      "platform": "desktop",  
      "usage": 8.31,  
      "current": "false"  
    },  
    // more items ...  
  ]  
}
```

We will use the `d3.json` method to load the JSON data. The `d3.json` method creates an asynchronous request to the specified URL and invokes the callback argument when the file is loaded or when the request fails. The callback function receives an error (if any) and the parsed data. There are similar methods to load text, CSV, TSV, XML, and HTML files. Refer to the following code:

```
<script>  
  // Load the data asynchronously.  
  d3.json('/chapter03/browsers.json', function(error, data) {  
  
    // Handle errors getting or parsing the JSON data.  
    if (error) {  
      console.error('Error accessing or parsing the JSON file.');
```

```
    return error;
}

// visualization code ...

});
</script>
```

Note that the callback function will be invoked only when the data is loaded. This means that the code after the d3.json invocation will be executed while the request is being made, and the data won't be available at this point. The visualization code should go either inside the callback or somewhere else and should use events to notify the charting code that the data is ready. For now, we will add the rendering code inside the callback.

We will create the circles with the div elements. To avoid the smaller elements being hidden by the bigger elements, we will sort the items and create the elements in decreasing usage order, as follows:

```
// Access the data items.
var items = data.values;

// Sort the items by decreasing usage.
items.sort(function(a, b) { return b.usage-a.usage; });
```

The Array.prototype.sort instance method sorts the array in place using a comparator function. The comparator should receive two array items: a and b. If a must go before b in the sorted array, the comparator function must return a negative number; if b should go first, the comparator function must return a positive value.

The force layout method

The force layout is a method to distribute elements in a given area, which avoids overlap between the elements and keeps them in the drawing area. The position of the elements is computed based on simple constraints, such as adding a repulsive force between the elements and an attractive force that pulls the elements towards the center of the figure. The force layout is specially useful to create bubble and network charts.

Although the layout doesn't enforce any visual representation, it's commonly used to create network charts, displaying the nodes as circles and the links as lines between the nodes. We will use the force layout to compute the position of the circles, without lines between them, as follows:

```
// Size of the visualization container.  
var width = 700,  
    height = 200;  
  
// Configure the force layout.  
var force = d3.layout.force()  
    .nodes(items)  
    .links([])  
    .size([width, height]);
```

As we don't intend to represent the relationships between the browser versions, we will set the `links` attribute to an empty array. To start the force computation, we invoke the `start` method as follows:

```
// Start the force simulation.  
force.start();
```

The force layout will append additional properties to our data items. Of these new attributes, we will use only the `x` and `y` attributes, which contain the computed position for each node. Note that the original data items shouldn't have names that conflict with these new attributes:

```
{  
  "name": "Android Browser",  
  "version": 3,  
  "platform": "mobile",  
  "usage": 0.01,  
  "current": "false",  
  "index": 0,  
  "weight": 0,  
  "x": 522.7463498711586,  
  "y": 65.54744869936258,  
  "px": 522.7463498711586,  
  "py": 65.54744869936258  
}
```

Having computed the position of the circles, we can proceed to draw them. As we promised not to use SVG, we will need to use other elements to represent our circles. One option is to use the `div` elements. There are several ways to specify the position of divs, but we will use **absolute positioning**.

A block element styled with absolute positioning can be positioned by setting its `top` and `left` offset properties (the bottom and right offsets can be specified as well). The offsets will be relative to their closest positioned ancestors in the DOM tree. If none of its ancestors are positioned, the offset will be relative to the viewport (or the `body` element, depending on the browser). We will use a positioned container div and then set the position of the divs to absolute. The container element will be the div with the `#chart` ID. We will select this to modify its style to use the relative position and set its width and height to appropriate values, as follows:

```
<!-- Container div -->
<div id="chart"></div>
```

We will also set padding to 0 so that we don't have to account for it in the computation of the inner element positions. Note that in order to specify the style attributes that represent length, we need to specify the units, except when the length is zero, as follows:

```
// Select the container div and configure its attributes
var containerDiv = d3.select('#chart')
    .style('position', 'relative')
    .style('width', width + 'px')
    .style('height', height + 'px')
    .style('padding', 0)
    .style('background-color', '#eeeeec');
```

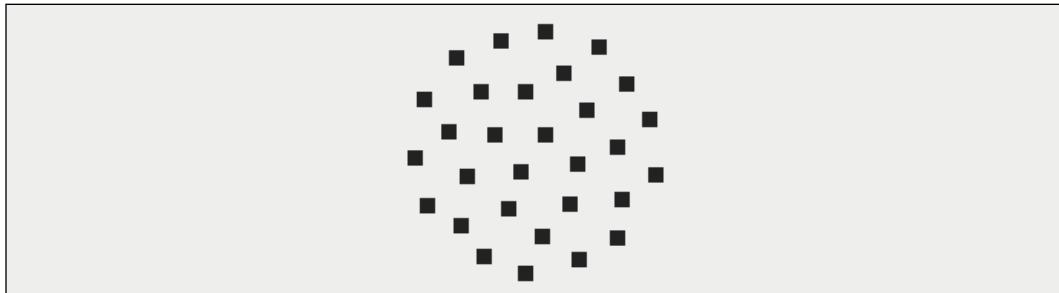
We can now create the inner divs. As usual, we will select the elements to be created, bind the data array to the selection, and append the new elements on enter. We will also set the style attributes to use absolute positioning, and set their offsets and their width and height to 10px as follows:

```
// Create a selection for the bubble divs, bind the data
// array and set its attributes.
var bubbleDivs = containerDiv.selectAll('div.bubble')
    .data(items)
    .enter()
    .append('div')
    .attr('class', 'bubble')
    .style('position', 'absolute')
    .style('width', '10px')
    .style('height', '10px')
    .style('background-color', '#222');
```

The force layout will compute the position of the nodes in a series of steps or ticks. We can register a **listener** function to be invoked on each tick event and update the position of the nodes in the listener as follows:

```
// Register a listener function for the force tick event, and
// update the position of each div on tick.
force.on('tick', function() {
    bubbleDiv
        .style('top', function(d) { return (d.y - 5) + 'px'; })
        .style('left', function(d) { return (d.x - 5) + 'px'; });
});
```

The divs will move nicely to their positions. Note that we subtract half of the div width and height when setting the offset. The divs will be centered in the position computed by the force layout as shown in the following screenshot:



The nodes are nicely positioned, but they all have the same size and color

Setting the color and size

Now that we have our nodes positioned, we can set the color and size of the div elements. To create a color scale for the nodes, we need to get a list with unique browser names. In our dataset, the items are browser versions; therefore, most of the browser names are repeated. We will use the `d3.set` function to create a set and use it to discard duplicated names, as follows:

```
// Compute unique browser names.
var browserList = items.map(function(d) { return d.name; }),
    browserNames = d3.set(browserList).values();
```

With the browser list ready, we can create a categorical color scale. Categorical scales are used to represent values that are different in kind; in our case, each browser will have a corresponding color:

```
// Create a categorical color scale with 10 levels.
var cScale = d3.scale.category10()
    .domain(browserNames);
```

The default range of `d3.scale.category10` is a set of 10 colors with similar lightness but different hue, specifically designed to represent categorical data. We could use a different set of colors, but the default range is a good starting point. If we had more than 10 browsers, we would need to use a color scale with more colors. We will also set the `border-radius` style attribute to half the height (and width) of the div in order to give the divs a circular appearance. Note that the `border-radius` attribute is not supported in all the browsers but has better support than SVG. In browsers that don't support this attribute, the divs will be shown as squares:

```
// Create a selection for the bubble divs, bind the data
// array and set its attributes.
var bubbleDivs = containerDiv.selectAll('div.bubble')
    .data(items)
    .enter()
    .append('div')
    // set other attributes ...
    .style('border-radius', '5px')
    .style('background-color', function(d) {
        return cScale(d.name);
    });
});
```

We can now compute the size of the circles. To provide an accurate visual representation, the area of the circles should be proportional to the quantitative dimensions that they represent, in our case, the market share of each version. As the area of a circle is proportional to the square of the radius, the radius of the circles must be proportional to the square root of the market share. We set the minimum and maximum radius values and use this extent to create the scale:

```
// Minimum and maximum radius
var radiusExtent = [10, 50];

// create the layout ...

// Create the radius scale
var rScale = d3.scale.sqrt()
    .domain(d3.extent(items, function(d) { return d.usage; }))
    .range(radiusExtent);
```

We will use the radius to compute the width, height, and border radius of each circle. To avoid calling the `scale` function several times (and to have cleaner code), we will add the radius as a new attribute of our data items:

```
// Add the radius to each item, to compute it only once.  
items.forEach(function(d) {  
    d.r = rScale(d.usage);  
});
```

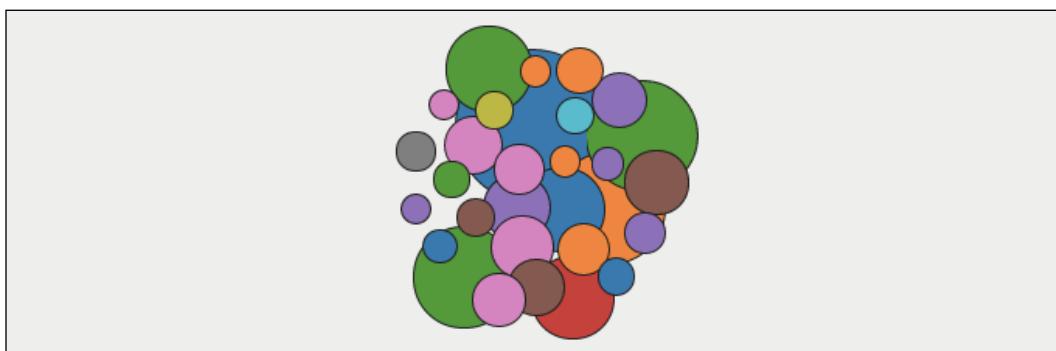
We can modify the width, height, and border radius of the divs to use the new attribute, as follows:

```
// Create the bubble divs.  
var bubbleDivs = containerDiv.selectAll('div.bubble')  
    .data(items)  
    .enter()  
    .append('div')  
    // set other attributes ...  
    .style('border-radius', function(d) { return d.r + 'px'; })  
    .style('width', function(d) { return (2 * d.r) + 'px'; })  
    .style('height', function(d) { return (2 * d.r) + 'px'; });
```

We need to update the position of the div elements to account for the new radius:

```
// Update the div position on each tick.  
force.on('tick', function() {  
    bubbleDiv  
        .style('top', function(d) { return (d.y - d.r) + 'px'; })  
        .style('left', function(d) { return (d.x - d.r) + 'px'; });  
});
```

The first draft of the visualization is shown in the following screenshot:

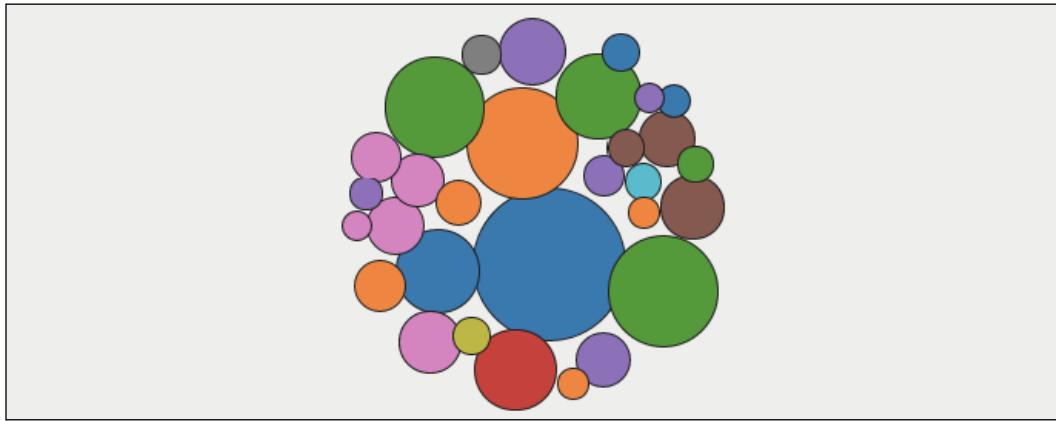


At this point, we have the first draft of our visualization, but there are still some things that need to be improved. The space around each div is the same, regardless of the size of each circle. We expect to have more space around bigger circles and less space around smaller ones. To achieve this, we will modify the `charge` property of the force layout, which controls the strength of repulsion between the nodes.

The `charge` method allows us to set the charge strength of each node. The default value is `-30`; we will use a function to set greater charges for bigger circles. In physical systems, the charge is proportional to the volume of the body; so, we will set the charge to be proportional to the area of each circle as follows:

```
// Configure the force layout.
var force = d3.layout.force()
  .nodes(items)
  .links([])
  .size([width, height])
  .charge(function(d) { return -0.12 * d.r * d.r; })
  .start();
```

We don't know in advance which proportionality constant will give a good layout; we need to tweak this value until we are satisfied with the visual result. Bubbles created with chart with the divs and force layout is shown in the following screenshot:



Now that we have a good first version of our visualization, we will adapt the code to use a reusable chart pattern. As you will surely remember, a reusable chart is implemented as a closure with the setter and getter methods as follows:

```
function bubbleChart() {  
  
    // Chart attributes ...  
  
    function chart(selection) {  
        selection.each(function(data) {  
  
            // Select the container div and configure its  
            // attributes.  
            var containerDiv = d3.select(this);  
  
            // create the circles ...  
        });  
    }  
  
    // Accessor methods ...  
  
    return chart;  
};
```

The code in the chart function is basically the same code that we have written until now. We also added accessor methods for the color scale, width, height, and radius extent, as well as accessor functions for the value, name, and charge function to allow users to adapt to the repulsion force when using the chart with other datasets. We can create and invoke the charting function in the callback of `d3.json` as follows:

```
d3.json('../data/browsers.json', function(error, data) {  
  
    // Handle errors getting or parsing the JSON data.  
    if (error) { return error; }  
  
    // Create the chart instance.  
    var chart = bubbleChart()  
        .width(500);  
  
    // Bind the chart div to the data array.  
    d3.select('#chart')  
        .data([data.values])  
        .call(chart);  
});
```

The visualization is incomplete without a legend, so we will create a legend now. This time, we will create the legend as a reusable chart from the beginning.

Creating a legend

The legend should display which color represents which browser. It can also have additional information such as the aggregated market share of each browser. We must use the same color code as that used in the visualization:

```
function legendChart() {  
  
    // Chart Properties ...  
  
    // Charting function.  
    function chart(selection) {  
        selection.each(function(data) {  
  
            });  
    }  
  
    // Accessor methods ...  
  
    return chart;  
};
```

We will implement the legend as a div element that contains paragraphs; each paragraph will have the browser name and a small square painted with the corresponding color. In this case, the data will be a list of browser names. We will add a configurable color scale to make sure that the legend uses the same colors that are used in the bubble chart:

```
function legendChart() {  
  
    // Color Scale  
    var cScale = d3.scale.category20();  
  
    // Charting function.  
    function chart(selection) {  
        // chart content ...  
    }  
  
    // Color Scale Accessor  
    chart.colorScale = function(value) {  
        if (!arguments.length) { return cScale; }  
    }  
};
```

```
    cScale = value;
    return chart;
};

return chart;
};
```

We can create a div for the legend and put it alongside the chart div as follows:

```
d3.json('/chapter03/browsers.json', function(error, data) {
  // Create an instance of the legend chart.
  var legend = legendChart()
    .colorScale(chart.colorScale());

  // Select the container and invoke the legend.
  var legendDiv = d3.select('#legend')
    .data([chart.colorScale().domain()])
    .call(legend);
});
```

We used the domain of the chart's color scale as the dataset for the legend and set the color scale of the legend with the color scale of the chart. This will ensure that you have the same items and colors in the legend that are in the chart. We also added a width attribute and its corresponding accessor. In the legend chart function, we can create the title and the legend items using the data:

```
// Select the container element and set its attributes.
var containerDiv = d3.select(this)
  .style('width', width + 'px');

// Add the label 'Legend' on enter
containerDiv.selectAll('p.legend-title')
  .data([data])
  .enter().append('p')
  .attr('class', 'legend-title')
  .text('Legend');

// Add a div for each data item
var itemDiv = containerDiv.selectAll('div.item')
  .data(data)
  .enter().append('div')
  .attr('class', 'item');
```

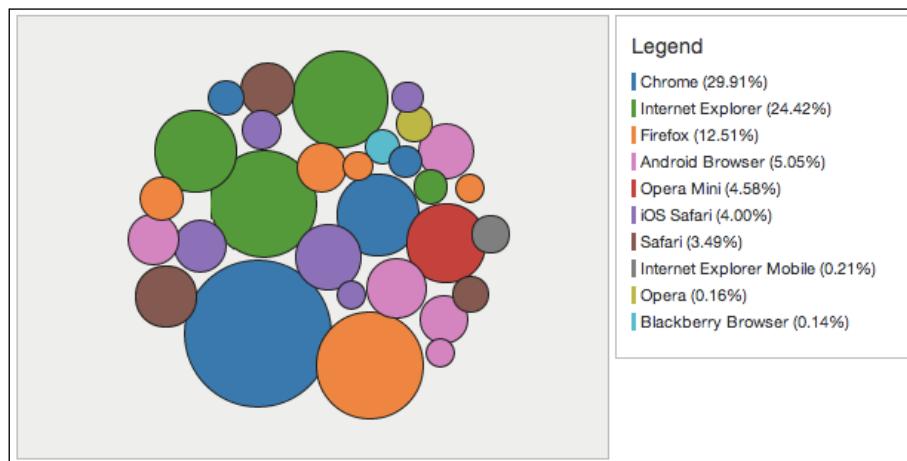
We have labels that show up in the legend, but we need to add a marker with the corresponding color. To keep things simple, we will add two points and set them with the same background and text color:

```
itemP.append('span').text('...')  
    .style('color', cScale)  
    .style('background', cScale);
```

To finish the legend, we will compute the market share of each browser. We will create a map to store each browser name and its aggregated usage, as follows:

```
// Create a map to aggregate the browser usage  
var browsers = d3.map();  
  
// Adds up the usage for each browser.  
data.values.forEach(function(d) {  
    var item = browsers.get(d.name);  
    if (item) {  
        browsers.set(d.name, {  
            name: d.name,  
            usage: d.usage + item.usage  
        });  
    } else {  
        browsers.set(d.name, {  
            name: d.name,  
            usage: d.usage  
        });  
    }  
});
```

The final version of the bubble chart is shown in the following screenshot:



In this example, we created a simple visualization using the div elements and displayed them as circles with the help of the border-radius attribute. Using divs with rounded corners is not the only alternative to create this visualization without using SVG; we could have used raster images of circles instead of div elements, using absolute positioning and changing the image width and height.

One great example of a sophisticated visualization made without SVG is the Electoral Map by the New York Times graphic department (<http://elections.nytimes.com/2012/ratings/electoral-map>). In this visualization, the user can create their own scenarios for the presidential elections of 2012 in the United States.

Polyfilling

A **polyfill** is a JavaScript library that replicates an API feature for the browsers that don't have it natively. Usually, a polyfill doesn't add its own API or additional features; it just adds the missing feature.

Polyfills are available for almost every HTML5 and CSS3 feature, but this doesn't mean that we can start adding libraries to provide all the modern features in the web browser. Also, the modern features can conflict with each other, so polyfills must be included carefully. To support SVG in those browsers, the following two polyfills can be used:

- **svgweb**: This provides partial SVG support, falling back to Flash if the browser doesn't support SVG (<https://code.google.com/p/svgweb/>).
- **canvg**: This is an SVG parser written in JavaScript. It parses the SVG element and renders it using a canvas (<https://code.google.com/p/canvg/>).

The first step to use a polyfill is to detect whether a feature is available in the browser or not.

Feature detection

There are several ways to find out whether the browser supports a particular feature. One of them is to get the user agent attribute of the navigator global object, but this is highly unreliable because the user can configure the user agent property. Another option is try to use the feature to check whether it has the methods and properties that we expect. However, this method is error prone and depends on the particular feature that we are looking for.

The most reliable way is to use the **Modernizr** library. Despite its name, it doesn't add any modern features to old browsers; it only detects the availability of the HTML5 and CSS3 features. However, it does interact well with the libraries that implement the missing features, providing a script loader to include the libraries in order to fill the gaps. The library can be customized to include the detection of only the features that we need.

The library performs a suite of tests to detect which features are available and which are not, and sets the results of these tests in the Boolean attributes of the global `Modernizr` object, add classes to the `HTML` object that explains which features are present. The library should be loaded in the header, because the features that we want to add must be available before the `<body>` element:

```
<!-- Include the feature detection library -->
<script src="/assets/js/lib/modernizr-latest.js"></script>
```

To detect the support of SVG in the browser, we can use the `Modernizr.svg` property. We can also use it to properly handle the lack of support:

```
<script>
    // Handle the availability of SVG.
    if (Modernizr.svg) {
        // Create a visualization with SVG.
    } else {
        // Fallback visualization.
    }
</script>
```

The canvg example

We will begin our example by creating an SVG image with D3. In this example, we will create an array of circles in SVG and then display them with canvas using the `canvg` library. We begin by including the libraries in the header as follows:

```
<!-- Canvg Libraries -->
<script src="/assets/js/lib/rgbcolor.js"></script>
<script src="/assets/js/lib/StackBlur.js"></script>
<script src="/assets/js/lib/canvg.js"></script>
```

For now, we will begin as usual by selecting the container div, appending the SVG element, and setting its width and height:

```
// Set the width and height of the figure.  
var width = 600,  
    height = 300;  
  
// Select the container div and append the SVG element.  
var containerDiv = d3.select('#canvg-demo'),  
    svg = containerDiv.append('svg')  
        .attr('width', width)  
        .attr('height', height);
```

We will generate a data array with one item per circle. We want to have one circle for each 10 pixels. The position of each circle will be given by the x and y attributes. The z attribute will contain a number proportional to both x and y; this number will be used to compute the radius and color scales, as follows:

```
// Generate data for the position and size of the rectangles.  
var data = [];  
for (var k = 0; k < 60; k += 1) {  
    for (var j = 0; j < 30; j += 1) {  
        data.push({  
            x: 5 + 10 * k,  
            y: 5 + 10 * j,  
            z: (k - 50) + (20 - j)  
        });  
    }  
}
```

We can create the radius and color scales. Both the scales will use the extent of the z property to set their domains:

```
// Create a radius scale using the z attribute.  
var rScale = d3.scale.sqrt()  
    .domain(d3.extent(data, function(d) { return d.z; }))  
    .range([3, 5]);  
  
// Create a linear color scale using the z attribute.  
var cScale = d3.scale.linear()  
    .domain(d3.extent(data, function(d) { return d.z; }))  
    .range(['#204a87', '#cc0000']);
```

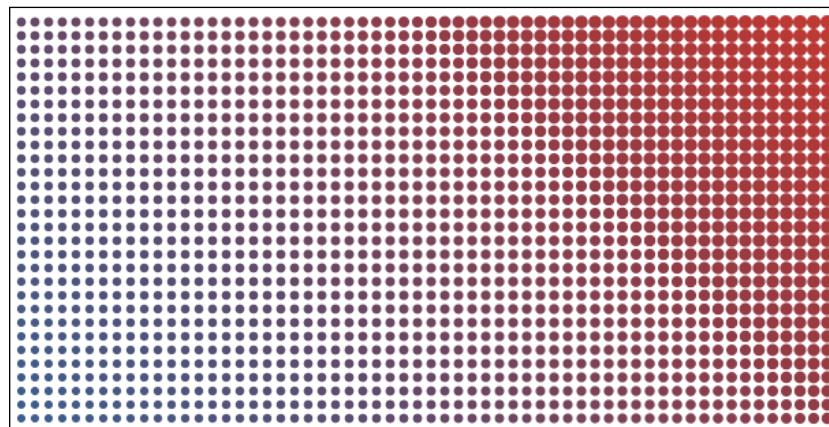
We can now create the circles in the SVG element. We create a selection for the circles to be created, bind the data array, append the circles on enter, and set their attributes:

```
// Select the circle elements, bind the dataset and append
// the circles on enter.
svg.selectAll('circle')
  .data(data)
  .enter()
  .append('circle')
  .attr('cx', function(d) { return d.x; })
  .attr('cy', function(d) { return d.y; })
  .attr('r', function(d) { return rScale(d.z); })
  .attr('fill', function(d) { return cScale(d.z); })
  .attr('fill-opacity', 0.9);
```

Until now, this is a standard D3. In a browser without SVG support, the elements will be created but not rendered. The `canvg` function interprets the SVG content and draws it with canvas instead. The function receives the canvas target (where we want SVG to be drawn), the SVG string, and an object with options. If it is called without arguments, the function will convert all the SVG elements present on the page. We will use this option as follows:

```
// Replace all the SVG elements by canvas drawings.
canvg();
```

If you inspect the page, you will see that the SVG element is gone, and in its place, there is a canvas element of the same size as the original SVG. The visual result is the same as that without using the `canvg` polyfill. Note that the event handlers bound to the original SVG elements won't work. For instance, if we added a callback for the `click` event on the circles, the callback for the event won't be invoked in the canvas version. Using `canvg` to render an SVG element is shown in the following screenshot:



Using canvas and D3

Until now, we have used D3 to create visualizations by manipulating SVG elements and divs. In some cases, it can be more convenient to render the visualizations using the canvas elements, for performance reasons or if we need to transform and render raster images. In this section, we will learn how to create figures with the HTML5 canvas element and how to use D3 to render figures with the canvas element.

Creating figures with canvas

The HTML canvas element allows you to create raster graphics using JavaScript. It was first introduced in HTML5. It enjoys more widespread support than SVG and can be used as a fallback option. Before diving deeper into integrating canvas and D3, we will construct a small example with canvas.

The canvas element should have the `width` and `height` attributes. This alone will create an invisible figure of the specified size:

```
<!-- Canvas Element -->
<canvas id="canvas-demo" width="650px" height="60px"></canvas>
```

If the browser supports the canvas element, it will ignore any element inside the canvas tags. On the other hand, if the browser doesn't support the canvas, it will ignore the canvas tags, but it will interpret the content of the element. This behavior provides a quick way to handle the lack of canvas support:

```
<!-- Canvas Element -->
<canvas id="canvas-demo" width="650px" height="60px">
<!-- Fallback image -->
</img>
</canvas>
```

If the browser doesn't support canvas, the fallback image will be displayed. Note that unlike the `` element, the canvas closing tag (`</canvas>`) is mandatory. To create figures with canvas, we don't need special libraries; we can create the shapes using the canvas API:

```
<script>
  // Graphic Variables
  var barw = 65,
      barh = 60;

  // Append a canvas element, set its size and get the node.
```

```

var canvas = document.getElementById('canvas-demo');

// Get the rendering context.
var context = canvas.getContext('2d');

// Array with colors, to have one rectangle of each color.
var color = ['#5c3566', '#6c475b', '#7c584f', '#8c6a44',
    '#9c7c39',
    '#ad8d2d', '#bd9f22', '#cdb117', '#ddc20b', '#edd400'];

// Set the fill color and render ten rectangles.
for (var k = 0; k < 10; k += 1) {
    // Set the fill color.
    context.fillStyle = color[k];
    // Create a rectangle in incremental positions.
    context.fillRect(k * barw, 0, barw, barh);
}
</script>

```

We use the DOM API to access the canvas element with the `canvas-demo` ID and to get the **rendering context**. Then, we set the color using the `fillStyle` method and use the `fillRect` canvas method to create a small rectangle. Note that we need to change `fillStyle` every time or all the following shapes will have the same color. The script will render a series of rectangles, each filled with a different color, shown as follows:



A graphic created with canvas

Canvas uses the same coordinate system as SVG, with the origin in the top-left corner, the horizontal axis augmenting to the right, and the vertical axis augmenting to the bottom. Instead of using the DOM API to get the canvas node, we could have used D3 to create the node, set its attributes, and created scales for the color and position of the shapes. Note that the shapes drawn with canvas don't exist in the DOM tree; so, we can't use the usual D3 pattern of creating a selection, binding the data items, and appending the elements if we are using canvas.

Creating shapes

Canvas has fewer primitives than SVG. In fact, almost all the shapes must be drawn with paths, and more steps are needed to create a path. To create a shape, we need to open the path, move the cursor to the desired location, create the shape, and close the path. Then, we can draw the path by filling the shape or rendering the outline. For instance, to draw a red semicircle centered in (325, 30) and with a radius of 20, write the following code:

```
// Create a red semicircle.  
context.beginPath();  
context.fillStyle = '#ff0000';  
context.moveTo(325, 30);  
context.arc(325, 30, 20, Math.PI / 2, 3 * Math.PI / 2);  
context.fill();
```

The `moveTo` method is a bit redundant here, because the `arc` method moves the cursor implicitly. The arguments of the `arc` method are the x and y coordinates of the arc center, the radius, and the starting and ending angle of the arc. There is also an optional Boolean argument to indicate whether the arc should be drawn counterclockwise. A basic shape created with the canvas API is shown in the following screenshot:



Integrating canvas and D3

We will create a small network chart using the force layout of D3 and canvas instead of SVG. To make the graph look more interesting, we will randomly generate the data. We will generate 250 nodes that are sparsely connected. The nodes and links will be stored as the attributes of the `data` object:

```
// Number of Nodes  
var nNodes = 250,  
    createLink = false;  
  
// Dataset Structure  
var data = {nodes: [], links: []};
```

We will append nodes and links to our dataset. We will create nodes with a `radius` attribute and randomly assign it a value of either 2 or 4 as follows:

```
// Iterate in the nodes
for (var k = 0; k < nNodes; k += 1) {
    // Create a node with a random radius.
    data.nodes.push({radius: (Math.random() > 0.3) ? 2 : 4});

    // Create random links between the nodes.
}
```

We will create a link with a probability of 0.1 only if the difference between the source and target indexes are less than 8. The idea behind this way to create links is to have only a few connections between the nodes:

```
// Create random links between the nodes.
for (var j = k + 1; j < nNodes; j += 1) {

    // Create a link with probability 0.1
    createLink = (Math.random() < 0.1) && (Math.abs(k - j)
        < 8);

    if (createLink) {
        // Append a link with variable distance between
        // the nodes
        data.links.push({
            source: k,
            target: j,
            dist: 2 * Math.abs(k - j) + 10
        });
    }
}
```

We will use the `radius` attribute to set the size of the nodes. The links will contain the distance between the nodes and the indexes of the source and target nodes. We will create variables to set the `width` and `height` of the figure:

```
// Figure width and height
var width = 650,
    height = 300;
```

We can now create and configure the force layout. As we did in the previous section, we will set the charge strength to be proportional to the area of each node. This time we will also set the distance between the links using the `linkDistance` method of the layout:

```
// Create and configure the force layout
var force = d3.layout.force()
    .size([width, height])
    .nodes(data.nodes)
    .links(data.links)
    .charge(function(d) { return -1.2 * d.radius * d.radius; })
    .linkDistance(function(d) { return d.dist; })
    .start();
```

We can create a canvas element now. Note that we should use the `node` method to get the canvas element, because the `append` and `attr` methods will both return a selection, which doesn't have the canvas API methods:

```
// Create a canvas element and set its size.
var canvas = d3.select('div#canvas-force').append('canvas')
    .attr('width', width + 'px')
    .attr('height', height + 'px')
    .node();
```

We get the rendering context. Each canvas element has its own rendering context. We will use the '2d' context to draw two-dimensional figures. At the time of writing this, there are some browsers that support the `webgl` context; more details are available at https://developer.mozilla.org/en-US/docs/Web/WebGL/Getting_started_with_WebGL. Refer to the following '2d' context:

```
// Get the canvas context.
var context = canvas.getContext('2d');
```

We register an event listener for the force layout's tick event. As canvas doesn't remember previously created shapes, we need to clear the figure and redraw all the elements on each tick:

```
force.on('tick', function() {
    // Clear the complete figure.
    context.clearRect(0, 0, width, height);

    // Draw the links ...
    // Draw the nodes ...
});
```

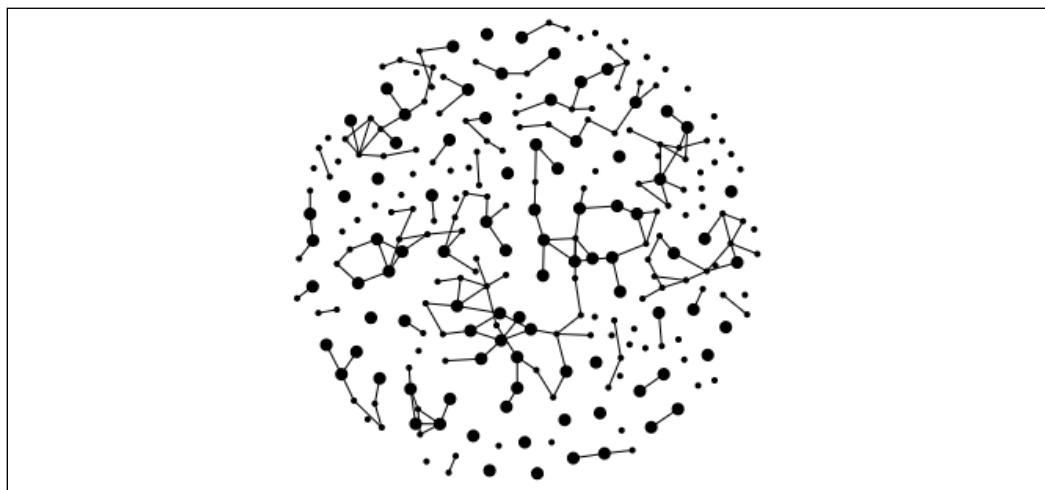
The `clearRect` method cleans the figure under the specified rectangle. In this case, we clean the entire canvas. We can draw the links using the `lineTo` method. We iterate through the links by beginning a new path for each link, by moving the cursor to the position of the source node, and by creating a line towards the target node. We draw the line with the `stroke` method:

```
// Draw the links
data.links.forEach(function(d) {
    // Draw a line from source to target.
    context.beginPath();
    context.moveTo(d.source.x, d.source.y);
    context.lineTo(d.target.x, d.target.y);
    context.stroke();
});
```

We iterate through the nodes and draw each one. We use the `arc` method to represent each node with a black circle:

```
// Draw the nodes
data.nodes.forEach(function(d, i) {
    // Draws a complete arc for each node.
    context.beginPath();
    context.arc(d.x, d.y, d.radius, 0, 2 * Math.PI, true);
    context.fill();
});
```

We obtain a constellation of disconnected network graphs slowly gravitating towards the center of the figure. Using the force layout and canvas to create a network chart is shown in the following screenshot:



We can think that to erase all the shapes and redraw each shape again and again could have a negative impact on the performance. In fact, sometimes it's faster to draw the figures using canvas, because this way, the browser doesn't have to manage the DOM tree of the SVG elements (but we still have to redraw them if the SVG elements are changed).

Summary

In this chapter, we learned how to handle the lack of SVG support in older browsers. We learned how to create visualizations using only the div elements and how to detect the availability of SVG support. We also discussed how to use polyfills to provide the missing functionality. In particular, we created an example of rendering SVG with canvas using the canvg library.

In the next chapter, we will create a color picker based on the Lab color model. We will learn how to use the drag behavior and use it to create a reusable slider element. Also, we will use the slider to create the color picker.

4

Creating a Color Picker with D3

In this chapter, we will implement a slider and a color picker using D3. We will use the reusable chart pattern to create the slider and the color picker. We will also learn how to compose reusable charts in order to create more complex components.

Creating a slider control

A slider is a control that allows a user to select a value within a given interval without having to type it. It has a handle that can be displaced over a base line; the position of the handle determines the selected value. The value is then used to update other components of the page. In this section, we will create a slider with D3 using the reusable chart pattern. We will include an API to change its visual attributes and modify other elements when the slider value changes. Note that in HTML5, we can create an `input` element of type `range`, which will be displayed as a slider with configurable minimum and maximum steps and values. The type `color` is also available, which allows us to use the native color picker. Native controls include accessibility features and using the keyboard to control the slider. More details on the `input` element can be found in <https://developer.mozilla.org/en/docs/Web/HTML/Element/Input>. To follow the examples of this section, open the `chapter04/01-slider.html` file.



The final slider component

The drag behavior

We will review how to use the drag behavior with a simple example. We will begin by creating the `svg` element and put a gray circle in the center:

```
// Width and height of the figure.  
var width = 600, height = 150;  
  
// Create the svg element.  
var svg = d3.select('#chart').append('svg')  
    .attr('width', width)  
    .attr('height', height);  
  
// Append a grey circle in the middle.  
var circle = svg.append('circle')  
    .attr('cx', width / 2)  
    .attr('cy', height / 2)  
    .attr('r', 30)  
    .attr('fill', '#555');
```

This will create the `svg` and `circle` elements, but the circle can't be moved yet. D3 allows us to detect gestures on an element by using behaviors, which are functions that create event listeners for gesture events on a container element. To detect the drag gesture, we can use the `drag` behavior. The `drag` behavior detects dragging events of three types, `dragstart`, `drag`, and `dragend`. We can create and configure the `drag` behavior by adding event listeners for one or more drag events; in our case, we will add a listener for the `drag` event as follows:

```
// Create and configure a drag behavior.  
var drag = d3.behavior.drag().on('drag', dragListener);
```

For more details on the D3 drag behavior, consult the D3 wiki page on this subject at <https://github.com/mbostock/d3/wiki/Drag-Behavior>. To add the `drag` behavior to the `circle`, we can call the `drag` function, passing the `circle` selection to it or using the `call` method as follows:

```
// Add dragging handling to the circle.  
circle.call(drag);
```

When the user drags the circle, the `dragListener` function will be invoked. The `dragListener` function receives the data item bound to the circle (if any), with the `this` context set to the container element, in our case, the `circle` element. In the `dragListener` function, we will update the position of the circle. Note that the `cx` and `cy` attributes are returned as strings, and prepending a plus sign will cast these values to numbers. Refer to the following code:

```
// Moves the circle on drag.  
function dragListener(d) {
```

```
// Get the current position of the circle
var cx = +d3.select(this).attr('cx'),
    cy = +d3.select(this).attr('cy');

// Set the new position of the circle.
d3.select(this)
    .attr('cx', cx + d3.event.dx)
    .attr('cy', cy + d3.event.dy);
}
```

The `dragListener` function updates the `cx` and `cy` attributes of the circle, but it can change other attributes as well. It can even change properties of other elements. In the next section, we will use the drag behavior to create a simple SVG slider.

Creating the slider

The slider component will have a configurable width, domain, and a listener function to be called on the slide. To use the slider, attach it to an `svg` group. As the group can be translated, rotated, and scaled, the same slider can be displayed horizontally, vertically, or even diagonally anywhere inside the `SVG` element. We will implement the slider using the reusable chart pattern:

```
function sliderControl() {
    // Slider Attributes...

    // Charting function.
    function chart(selection) {
        selection.each(function(data) {
            // Create the slider elements...
        });
    }

    // Accessor Methods...

    return chart;
}
```

We will add attributes for `width` and `domain`, as well as their corresponding accessor methods, `chart.width` and `chart.domain`. Remember that the accessor methods should return the current value if they are invoked without arguments and return the `chart` function if a value is passed as an argument:

```
function chart(selection) {
    selection.each(function(data) {
        // Select the container group.
```

```
var group = d3.select(this);

    // Create the slider content...
}) ;
}
```

We will assume that the slider is created within an `svg` group, but we could have detected the type of the container element and handle each case. If it were a `div`, for instance, we could append an `svg` element and then append a group to it. We will work with the group to keep things simple. We will create the base line using the `svg` line element:

```
// Add a line covering the complete width.
group.selectAll('line')
  .data([data])
  .enter().append('line')
  .call(chart.initLine);
```

We encapsulate the creation of the line in the `chart.initLine` method. This function will receive a selection that contains the created line and sets its position and other attributes:

```
// Set the initial attributes of the line.
chart.initLine = function(selection) {
  selection
    .attr('x1', 2)
    .attr('x2', width - 4)
    .attr('stroke', '#777')
    .attr('stroke-width', 4)
    .attr('stroke-linecap', 'round');
};
```

We set the `x1` and `x2` coordinates of the line. The default value for the coordinates is zero, so we don't need to define the `y1` and `y2` coordinates. The `stroke-linecap` attribute will make the ends of the line rounded, but we will need to adjust the `x1` and `x2` attributes to show the rounded corners. With a stroke width of 4 pixels, the radius of the corner will be 2 pixels, which will be added in each edge of the line. We will create a circle in the group in the same way:

```
// Append a circle as handler.
var handle = group.selectAll('circle')
  .data([data])
  .enter().append('circle')
  .call(chart.initHandle);
```

The `initHandle` method will set the radius, fill color, stroke, and position of the circle. The complete code of the function is available in the example file. We will create a scale to map the value of the slider to the position of the circle:

```
// Set the position scale.  
var posScale = d3.scale.linear()  
    .domain(domain)  
    .range([0, width]);
```

We correct the position of the circle, so its position represents the initial value of the slider:

```
// Set the position of the circle.  
handle  
    .attr('cx', function(d) { return posScale(d); });
```

We have created the slider base line and handler, but the handle can't be moved yet. We need to add the drag behavior to the circle:

```
// Create and configure the drag behavior.  
var drag = d3.behavior.drag().on('drag', moveHandle);  
  
// Adds the drag behavior to the handler.  
handler.call(drag);
```

The `moveHandle` listener will update only the horizontal position of the circle, keeping the circle within the slider limits. We need to bind the value that we are selecting to the handle (the circle), but the `cx` attribute will give us the position of the handle in pixels. We will use the `invert` method to compute the selected value and rebind this value to the circle so that it's available in the caller function:

```
function moveHandle(d) {  
    // Compute the future position of the handler  
    var cx = +d3.select(this).attr('cx') + d3.event.dx;  
  
    // Update the position if it's within its valid range.  
    if ((0 < cx) && (cx < width)) {  
        // Compute the new value and rebind the data  
        d3.select(this).data([posScale.invert(cx)])  
            .attr('cx', cx);  
    }  
}
```

Creating a Color Picker with D3

To use the slider, we will append an SVG figure to the container div and set its width and height:

```
// Figure properties.  
var width = 600, height = 60, margin = 20;  
  
// Create the svg element and set its dimensions.  
var svg = d3.select('#chart').append('svg')  
    .attr('width', width + 2 * margin)  
    .attr('height', height + 2 * margin);
```

We can now create the slider function, setting its width and domain:

```
// Valid range and initial value.  
var value = 70, domain = [0, 100];  
  
// Create and configure the slider control.  
var slider = sliderControl().width(width).domain(domain);
```

We create a selection for the container group, bind the data array that contains the initial value, and append the group on enter. We also translate the group to the location where we want the slider and invoke the `slider` function using the `call` method:

```
var gSlider = svg.selectAll('g')  
    .data([value])  
    .enter().append('g')  
    .attr('transform', 'translate(' + [margin, height / 2] + ')')  
    .call(slider);
```

We have translated the container group to have a margin, and we have centered it vertically. The slider is now functional, but it doesn't update other components or communicate changes in its value. Refer to the following screenshot:



The slider appended to an SVG group

We will add a user-configurable function that will be invoked when the user moves the handler, along with its corresponding accessor function, so that the user can define what should happen when the slider is changed:

```
function sliderControl() {
    // Slider attributes...

    // Default slider callback.
    var onSlide = function(selection) { };

    // Charting function...
    function chart() {...}

    // Accessor Methods
    // Slide callback function
    chart.onSlide = function(onSlideFunction) {
        if (!arguments.length) { return onSlide; }
        onSlide = onSlideFunction;
        return chart;
    };

    return chart;
}
```

The `onSlide` function will be called on the drag listener function, passing the handler selection as an argument. This way, the value of the slider will be passed to the `onSlide` function as the bound data item of the selection argument:

```
function moveHandle(d) {

    // Compute the new position of the handler
    var cx = +d3.select(this).attr('cx') + d3.event.dx;

    // Update the position within its valid range.
    if ((0 < cx) && (cx < width)) {
        // Compute the new value and rebind the data
        d3.select(this).data([posScale.invert(cx)])
            .attr('cx', cx)
            .call(onSlide);
    }
}
```

Remember that the `onSlide` function should receive a selection, and through the selection, it should receive the value of the slider. We will use the `onSlide` function to change the color of a rectangle.

Using the slider

We use the slider to change the color of a rectangle. We begin by creating the `svg` element, setting its `width`, `height`, and `margin`:

```
// Create the svg element
var svg = d3.select('#chart').append('svg')
    .attr('width', width + 2 * margin)
    .attr('height', height + 3 * margin);
```

We create a linear color scale; its range will be the colors yellow and red. The domain of the scale will be the same as that in the slider:

```
// Create a color scale with the same domain of the slider
var cScale = d3.scale.linear()
    .domain(domain)
    .range(['#edd400', '#a40000']);
```

We add a rectangle in the `svg`, reserving some space in its upper side to put the slider on top. We also set its `width`, `height`, and `fill` color:

```
// Add a background to the svg element.
var rectangle = svg.append('rect')
    .attr('x', margin)
    .attr('y', 2 * margin)
    .attr('width', width)
    .attr('height', height)
    .attr('fill', cScale(value));
```

We create the slider control and configure its attributes. The `onSlide` function will change the rectangle fill color using the previously defined scale:

```
// Create and configure the slider control.
var slider = sliderControl()
    .domain(domain)
    .width(width)
    .onSlide(function(selection) {
        selection.each(function(d) {
            rectangle.attr('fill', cScale(d));
        });
});
```

Finally, we append a group to contain the slider and translate it to put it above the rectangle. We invoke the `slider` function using the `call` method:

```
// Create a group to hold the slider and add the slider to it.
var gSlider = svg.selectAll('g').data([value])
  .enter().append('g')
  .attr('transform', 'translate(' + [margin, margin] + ')')
  .call(slider);
```

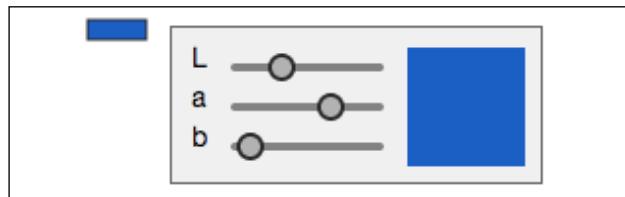
Refer to the following screenshot:



The fill color of the rectangle is controlled by the slider

Creating a color picker

We will implement a color picker using the slider from the previous section. A color picker is a UI control that allows the user to select a color. Usually, the color picker is shown as a small, colored rectangle, and when the user clicks on it, a window is opened with controls to change the color. To follow the code snippets, open the `chapter04/02-color-picker.html` file. Refer to the following screenshot:



The color picker selector and window that uses the sliders from the previous section

We will implement the color picker using the reusable chart pattern. This will allow us to modularize the components in two parts, the color picker selector and the color picker window. We begin by creating the color picker selector.

The color picker selector

The control element will be shown as a small rectangle. When the user clicks on the rectangle, the color picker will appear, and clicking on the control again will set the value. We will create a color picker using the CIELAB 1976 color model, which is informally known as Lab. In this model, L is for lightness, and the a and b parameters represent colors. This color model aims to be more perceptually uniform than other models, which means that changes in the color value are perceived as changes of about the same visual importance. We create the structure of the color picker using the reusable chart pattern as follows:

```
function labColorPicker() {  
  
    // Selector Attributes  
  
    // Selector shape  
    var width = 30,  
        height = 10;  
  
    // Default Color  
    var color = d3.lab(100, 0, 0);  
  
    // Charting function  
    function chart(selection) {  
        selection.each(function() {  
            // Creation of the color picker...  
        });  
    }  
  
    // Width and height accessor methods...  
  
    // Color Accessor  
    chart.color = function(value) {  
        if (!arguments.length) { return color; }  
        color = d3.lab(value);  
        return chart;  
    };  
  
    return chart;  
}
```

The chart `.color` method receives a color in any format that can be converted by `d3.lab` and returns the picker color as a `d3.lab` object with the current color of the picker. We can also add accessors for the width and height (not shown for brevity). To use the color picker, we need to create a container group for it and use the `call` method to create the color selector:

```
// Create the svg figure...

// Create the color picker and set the initial color
var picker = labColorPicker().color('#a40000');

// Create a group for the color picker and translate it.
var grp = svg.append('g')
    .attr('transform', 'translate(' + [offset, offset] + ')')
    .call(picker);
```

We will translate the group to add a margin between the color picker selector and the surrounding elements. In the charting function, we will create a selection with a rectangle, bind the current color to that selection, and create the rectangle on enter:

```
function chart(selection) {
    selection.each(function() {
        // Create the container group and rectangle
        var group = d3.select(this),
            rect = group.selectAll('rect');

        // Bind the rectangle to the color item and set its
        // initial attributes.
        rect.data([chart.color()])
            .enter().append('rect')
            .attr('width', width)
            .attr('height', height)
            .attr('fill', function(d) { return d; })
            .attr('stroke', '#222')
            .attr('stroke-width', 1);
    });
}
```

This will create the picker rectangle with the initial color.

Adding the color picker window

The color picker window should show up if the color selector is clicked on and hide it if the selector is clicked on a second time. We will use a div element to create the color picker window:

```
// Bind the rectangle to the data
rect.data([chart.color()])
  .enter().append('rect')
  // set more attributes ...
  .on('click', chart.onClick);
```

The openPicker function receives the data item bound to the rectangle. When the user clicks on the rectangle, the openPicker function will be invoked. This method will create the color picker window (a div) or remove it if already exists:

```
var openPicker = function(d) {
  // Select the color picker div and bind the data.
  var div = d3.select('body').selectAll('div.color-picker')
    .data([d]);

  if (div.empty()) {
    // Create the container div, if it doesn't exist.
    div.enter().append('div')
      .attr('class', 'color-picker');
  } else {
    // Remove the color picker div, if it exists.
    d3.select('body').selectAll('div.color-picker')
      .remove();
  }
};
```

Here, we detect whether the element exists using the empty method. If the selection is empty, we create the div and set its attributes. If the selection is not empty, we remove the color picker window. We want the color picker window to appear near the rectangle; we will use the position of the pointer to locate the window to the right-hand side of the selector. To position the color picker window, we need to set its position to absolute, and set its top and left offsets to appropriate values. We also set a provisional width, height, and background color for the div:

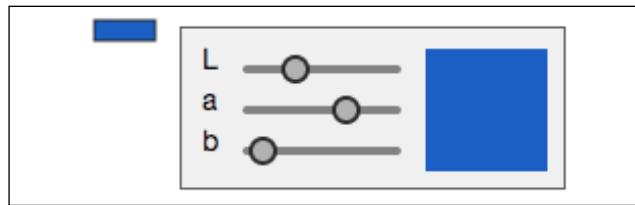
```
// Create the container div, if it doesn't exist.
div.enter().append('div')
  .attr('class', 'color-picker')
  .style('position', 'absolute')
  .style('left', (d3.event.pageX + width) + 'px')
  .style('top', d3.event.pageY + 'px')
```

```
.style('width', '200px')
.style('height', '100px')
.style('background-color', '#eee');
```

We now have a div element that is displayed when the user clicks on it and hidden when the user clicks again. Most importantly, the window div is bound to the same data item as the rectangle, and the `this` context in the `onClick` method is the rectangle node. We can now create the color picker window as a reusable chart and bind it to the color picker selector.

The color picker window

Let's review what we have done so far. The color picker has two parts: the color picker selector and the color picker window. The color picker window is a div that appears when the user clicks on the selector and disappears if the user clicks again. We will use the reusable chart pattern to create the color picker window. Refer to the following screenshot:



Components of the color picker; we will use the slider from the previous section

We can create the color picker content as an independent chart. For simplicity, this time, we won't add the `width`, `height`, and `margins` as configurable attributes. In this case, there are many elements that need to be created and positioned in the figure; we will only show you the most important ones:

```
function labColorPickerWindow() {

    // Chart Attributes...
    var margin = 10,
        // set more attributes...
        width = 3 * margin + labelWidth + sliderWidth + squareSize,
        height = 2 * margin + squareSize;

    function chart(selection) {
        selection.each(function(data) {
            // Select the container div and set its style
            var divContent = d3.select(this);
```

Creating a Color Picker with D3

```
// set the divContent size and position...

// Create the SVG Element
var svg = divContent.selectAll('svg')
    .data([data])
    .enter().append('svg');

// set the svg width and height...
// add more elements...
});

}

return chart;
};
```

Again, we have used the basic structure of a reusable chart. Remember that in the `selection.each` function, the data is the color of the selector, and the context is the container div. We begin by adding the square that will show you the selected color:

```
// Add the color square
var colorSquare = svg.append('rect')
    .attr('x', 2 * margin + sliderWidth + labelWidth)
    .attr('y', margin)
    .attr('width', squareSize)
    .attr('height', squareSize)
    .attr('fill', data);
```

This will put the square to the right-hand side of the window. Next, we will create a scale to position each slider vertically. The `rangePoints` method of the scale allows us to evenly distribute the sliders in the vertical space:

```
// Scale to distribute the sliders vertically
var vScale = d3.scale.ordinal()
    .domain([0, 1, 2])
    .rangePoints([0, squareSize], 1);
```

We will use this scale to set the position of the groups that will contain the slider elements for each color component. We create a slider for the `l` component of the color:

```
var sliderL = sliderControl()
    .domain([0, 100])
    .width(sliderWidth);
.onSlide(function(selection) {
    selection.each(function(d) {
        data.l = d;
        updateColor(data);
    });
});
```

The `l` component of the color is updated on the slider, and then the `updateColor` function is invoked, passing the color as the argument. We add a group to display the slider and translate it to the appropriate location. Remember that the data bound to the group is the value that is changed when the user moves the handler:

```
var gSliderL = svg.selectAll('g.slider-l')
  .data([data.l])
  .enter().append('g')
  .attr('transform', function() {
    var dx = margin + labelWidth,
        dy = margin + vScale(0);
    return 'translate(' + [dx, dy] + ')';
})
.call(sliderL);
```

This will create the first slider. In a similar way, we add a slider for the `a` and `b` color components with its corresponding groups. When the user moves the slider, the square and the rectangle in the selector should get updated. This chart is independent of the color picker selector to which we don't have access to the selector rectangle in this scope. We will add a configurable method that will be invoked on color change, so the user of `labColorWindow` can update other components. The `updateColor` function will update the color of the square and invoke the `onColorChange` function:

```
// Update the color square and invoke onColorChange
function updateColor(color) {
  colorSquare.attr('fill', color);
  divContent.data([color]).call(onColorChange);
}
```

Note that the color is bound to the color picker window `div`, and `onColorChange` receives the selection that contains the window. We need to add a default function and an accessor to configure this function. The default function will be just an empty function. We can now update the color picker selector, more precisely, the `onClick` method, to create the color picker window as follows:

```
chart.onClick = function(d) {
  // Select the picker rectangle
  var rect = d3.select(this);

  // Select the color picker div and bind the data...
  if (div.empty()) {
    // Create the Color Picker Content
    var content = labColorPickerWindow()
      .onColorChange(function(selection) {
```

Creating a Color Picker with D3

```
selection.each(function(d) {
    rect.data([d]).attr('fill', d);
});

// Create the container div, if it doesn't exist.
div.enter().append('div')
    .attr('class', 'color-picker')
    // set more attributes....
    .call(content);

// Bind the data to the rectangle again.
rect.data([div.datum()]);
} else {
    // Update the color of the rectangle
    rect.data([div.datum()])
        .attr('fill', function(d) { return d; });

    // Remove the color picker window.
    d3.select('body')
        .selectAll('div.color-picker').remove();
}
};


```

We can now select a color using the color picker. There is only one thing missing; the user will want to do something with the color once it has been changed. We will add an `onChangeColor` function and its corresponding accessor to the color picker window and invoke it at the end of the `chart.onclick` method. With this function, the user will be able to use the color to change other components:

```
chart.onClick = function(d) {
    // ...

    // Invoke the user callback.
    onColorChange(color);
};


```

To use the color picker, we need to attach it to a selection that contains a group and configure the `onColorChange` function:

```
// Create the color picker
var picker = labColorPicker()
    .color('#fff')
    .onColorChange(function(d) {


```

```
// Change the background color of the page
d3.select('body').style('background-color', d);
});

// Create a group for the color picker and translate it.
var grp = svg.append('g')
    .attr('transform', 'translate(30, 30)')
    .call(picker);
```

This will change the background color of the example page when the user selects a color. In this section, we have used the slider components to create a color picker. The color picker has two independent components: the selector and the color picker window. The color picker selector creates an instance of the color picker window when the user clicks on it and removes the window container when the user clicks on it again.

Summary

In this chapter, we used the drag behavior and the reusable chart pattern to create a slider control. This control can be used to allow users to select values within a range. We used the slider component to create a color picker for the Lab color space. In the implementation of the color picker, we didn't need to know about the internals of the slider; we only used the slider's public interface. The composition between reusable components allows us to create rich components without having to handle the details of their internal elements.

In the next chapter, we will learn how to create tooltips for our charts and how to implement more advanced user interface components. We will also create an area chart that allows us to measure variations between two points in the chart by using brushing.

5

Creating User Interface Elements

In the previous chapter, we learned how to use the drag behavior and SVG elements to create reusable controls and user interface elements. In this chapter, we will learn how to create additional elements to complement our projects. When designing data visualizations, screen real estate is one of the scarcest resources; we need to get the most out of our pixels without cluttering the screen. One of the strategies to solve this problem is to add contextual user interface elements, allowing the user to request any additional information in a quick and nonintrusive way. A **tooltip** does just that: it displays additional information about an item without cluttering the entire visualization.

If the page has a large number of elements, the user can lose track of the important parts of the visualization or have difficulties in tracking individual elements. One solution is to highlight the important elements, so we can guide the users' attention to the most relevant elements in the page.

In this chapter, we will learn how to highlight elements and create reusable tooltips. We will also create a chart with a brushing control, which will allow us to select an interval and display additional information about that interval.

Highlighting chart elements

We will create a simple chart depicting a series of circles that represent fruits and the number of calories we can get from 100 grams of each fruit. To make things easier, we have created a JSON file with information about the fruits. The file structure is as follows:

```
{  
  "name": "Fruits",  
  ...}
```

```
"data": [
  {
    "name": "Apple",
    "description": "The apple is the pomaceous fruit...",
    "amount_grams": 100,
    "calories": 52,
    "color": "#FF5149"
  },
  ...
]
```

We will represent each fruit with a circle and arrange them horizontally. We will map the area of the circle to the calories by serving, coloring them with the color indicated in the data item. As usual, we will use the reusable chart pattern, creating a closure function with the chart attributes and a charting function that contains the rendering logic, as shown in the following code:

```
function fruitChart() {

  // Chart Attributes
  var width = 600,
      height = 120;

  // Radius Extent
  var radiusExtent = [0, 40];

  // Charting Function
  function chart(selection) {
    selection.each(function(data) {
      // charting function content ...
    });
  }

  // Accessor Methods...

  return chart;
}
```

Although it is unlikely that we will reuse the fruit chart, the reusable chart structure is still useful because it encapsulates the chart variables and we get cleaner code. In the `charting` function, we select the `div` container, create the `svg` element, and set its width and height, as follows:

```
// Charting Function
function chart(selection) {
    selection.each(function(data) {

        // Select the container div and create the svg selection
        var div = d3.select(this),
            svg = div.selectAll('svg').data([data]);

        // Append the svg element on enter
        svg.enter().append('svg');

        // Update the width and height of the SVG element
        svg.attr('width', width).attr('height', height);

        // add more elements...
    });
}
```

We want to have the circles evenly distributed in the horizontal dimension. To achieve this, we will use an ordinal scale to compute the position of each circle, as shown in the following code:

```
// Create a scale for the horizontal position
var xScale = d3.scale.ordinal()
    .domain(d3.range(data.length))
    .rangePoints([0, width], 1);
```

The `rangePoints` method will configure the scale, dividing the `[0, width]` range into the number of elements in the domain. The second argument allows you to add padding, expressed as a multiple of the distance between two items. We will also add a scale for the radius, mapping the number of calories to the area of the circle. As discussed earlier, the area of the circle should be proportional to the quantitative dimensions that we are representing:

```
// Maximum number of calories
var maxCal = d3.max(data, function(d) {
    return d.calories;
});

// Create the radius scale
```

```
var rScale = d3.scale.sqrt()  
  .domain([0, maxCal])  
  .rangeRound(radiusExtent);
```

We will create groups and translate them to the location where we want the circles and labels. We will append the circles and labels to the groups, as follows:

```
// Create a container group for each circle  
var gItems = svg.selectAll('g.fruit-item').data(data)  
  .enter()  
  .append('g')  
  .attr('class', 'fruit-item')  
  .attr('transform', function(d, i) {  
    return 'translate(' + [xScale(i), height / 2] + ')';  
});
```

We can now append the circle, the label that displays the fruit name, and another label that shows the number of calories per serving. We will set the style of the labels to align them to the center and set the font size for the labels:

```
// Add a circle to the item group  
var circles = gItems.append('circle')  
  .attr('r', function(d) { return rScale(d.calories); })  
  .attr('fill', function(d) { return d.color; });

// Add the fruit name  
var labelName = gItems.append('text')  
  .attr('text-anchor', 'middle')  
  .attr('font-size', '12px')  
  .text(function(d) { return d.name; });

// Add the calories label  
var labelKCal = gItems.append('text')  
  .attr('text-anchor', 'middle')  
  .attr('font-size', '10px')  
  .attr('y', 12)  
  .text(function(d) { return d.calories + ' kcal'; });
```

We can use the chart at this point. We load the JSON file, create and configure the fruit chart, select the div container, and call the chart, passing the selection as an argument, as shown in the following code:

```
// Load and parse the json data  
d3.json('/chapter05/fruits.json', function(error, root) {  
  
  // Display the error message
```

```

if (error) {
  console.error('Error getting or parsing the data.');
  throw error;
}

// Create and configure the chart
var fruits = fruitChart();

d3.select('div#chart')
  .data([root.data])
  .call(fruits);
});

```

We obtained a series of circles where each one represents a fruit, but without any highlighting yet.



The first draft of the chart, without any highlighting

We will highlight the circles when the pointer moves over the circles by changing the background color to a brighter color of the same hue and by adding a small border, returning the circles to their original state when the mouse leaves the element.

The DOM API allows you to bind listeners for events to individual elements, but the `selection.on` method allows you to apply a listener to all the elements in a selection at the same time. As in almost every D3 operator, the `this` context is set to the selected element in the listener function. The `d3.rgb` function constructs a RGB color from the hexadecimal string. The `brighter()` method returns a brighter version of the color. The method receives an optional parameter, `k`, which can be used to specify the increment of brightness. The default value is 1, and the brightness is increased by multiplying each channel by 0.7^{-k} . We will use this method to highlight the circles on a `mouseover` event:

```

// We add listeners to the mouseover and mouseout events
circles
  .on('mouseover', function(d) {
    d3.select(this)
      .attr('stroke-width', 3)
      .attr('fill', d3.rgb(d.color).brighter())
      .attr('stroke', d.color);
  })

```

```
.on('mouseout', function(d) {
  d3.select(this)
    .attr('stroke-width', 0)
    .attr('fill', d.color);
});
```

The elements are now highlighted when the user moves the pointer over the circles.



The highlighted element is brighter and has a small border

This is a very simple example, but the method to highlight elements is the same in bigger charts. There are other strategies to highlight elements; for instance, a highlight class can be added or removed when the user moves the cursor over the elements. This strategy is particularly useful when we don't want to hardcode the styles.

Creating tooltips

A **tooltip** is a small element that provides contextual information when the user locates the pointer over an element. This allows you to provide details without cluttering the visualization. In this section, we will create the tooltip as a reusable chart but with a different structure than that in the previous examples. In the previous charts, we bound the data to a selection of containers for the charts; while in this case, the tooltip chart will be bound to the element on which the tooltip should appear. This implies that the selection argument in the charting function contains the elements on which the tooltip will appear. In the case of the fruit chart, we will want the tooltip to appear when the user moves the pointer over the circles, follow the pointer as it moves over the circle, and disappear when the pointer leaves the circle. We will create a tooltip as a reusable chart, but instead of invoking the tooltip on a selection of containers, we will invoke the tooltip passing it a selection containing the circle under the cursor. We begin by creating the tooltip chart, bearing in mind these considerations:

```
function tooltipChart() {

  // Tooltip Attributes...

  // Charting function
  function chart(selection) {
```

```

selection.each(function(d) {
    // Bind the mouse events to the container element
    d3.select(this)
        .on('mouseover', create)
        .on('mousemove', move)
        .on('mouseout', remove);
}) ;
}

// Accessor methods...

return chart;
}

```

Here, we added listeners for the `mouseover`, `mousemove`, and `mouseout` events on the `selection` argument. The data bound to each element will be passed on to the `create`, `move`, and `remove` listeners. These functions will create, move, and remove the tooltip, respectively. To create the tooltip, we will create a `div` container under the `body` element and set its left and top offsets to the pointer position, plus we add a small offset, as shown in the following code:

```

// Create the tooltip chart
var create = function(data) {

    // Create the tooltip container div
    var tooltipContainer = d3.select('body').append('div')
        .datum(data)
        .attr('class', 'tooltip-container')
        .call(init);

    // Move the tooltip to its initial position
    tooltipContainer
        .style('left', (d3.event.pageX + offset.x) + 'px')
        .style('top', (d3.event.pageY + offset.y) + 'px');
};

```

To locate the tooltip near the pointer, we need to set its position to `absolute`. The `pointer-events` style must be set to `none` so that the tooltip doesn't capture the mouse events. We set the position and other style attributes in an inline style element. We also set the style for the tooltip's title and content, as shown in the following code:

```

<style>
.tooltip-container {
    position: absolute;
    pointer-events: none;

```

```
padding: 2px 4px 2px 6px;
background-color: #eee;
border: solid 1px #aaa;
}

.tooltip-title {
    text-align: center;
    font-size: 12px;
    font-weight: bold;
    line-height: 1em;
}

.tooltip-content {
    font-size: 11px;
}
</style>
```

In the initialization function, we will create the `div` container for the tooltip and add paragraphs for the title and the content. We also added the `title` and `content` methods with their corresponding accessors so that the user can configure the title and content based on the bound data:

```
// Initialize the tooltip
var init = function(selection) {
    selection.each(function(data) {
        // Create and configure the tooltip container
        d3.select(this)
            .attr('class', 'tooltip-container')
            .style('width', width + 'px');

        // Tooltip Title
        d3.select(this).append('p')
            .attr('class', 'tooltip-title')
            .text(title(data));

        // Tooltip Content
        d3.select(this).append('p')
            .attr('class', 'tooltip-content')
            .text(content(data));
    });
};
```

The `chart.move` method will update the position of the tooltip as the pointer moves, changing its left and top offsets. The `chart.remove` method will just remove the tooltip from the document:

```
// Move the tooltip to follow the pointer
var move = function() {
    // Select the tooltip and move it following the pointer
    d3.select('body').select('div.tooltip-container')
        .style('left', (d3.event.pageX + offset.x) + 'px')
        .style('top', (d3.event.pageY + offset.y) + 'px');
};

// Remove the tooltip
var remove = function() {
    d3.select('div.tooltip-container').remove();
};
```

Using the tooltip

We can use the tooltip in the fruit chart, and add tooltips when the user moves the pointer over the circles. We will create and configure the `tooltip` function in the fruit chart closure, as follows:

```
function fruitChart() {

    // Create and configure the tooltip
    var tooltip = tooltipChart()
        .title(function(d) { return d.name; })
        .content(function(d) { return d.description; });

    // Attributes, charting function and accessors...

    return chart;
}
```

In the `charting` function, we can invoke the `tooltip` function by passing the selection of the circles as an argument, as follows:

```
function fruitChart() {

    // Chart attributes...

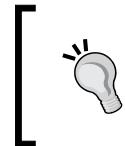
    // Charting Function
    function chart(selection) {
        selection.each(function(data) {
```

```
// Charting function content...

// The event listeners of the tooltip should be
// namespaced to avoid overwriting the listeners of the
// circles.
circles
  .on('mouseover', function(d) { ... })
  .on('mouseout', function(d) { ... })
  .call(tooltip);
})
}

// Accessor methods....
```

return chart;
}

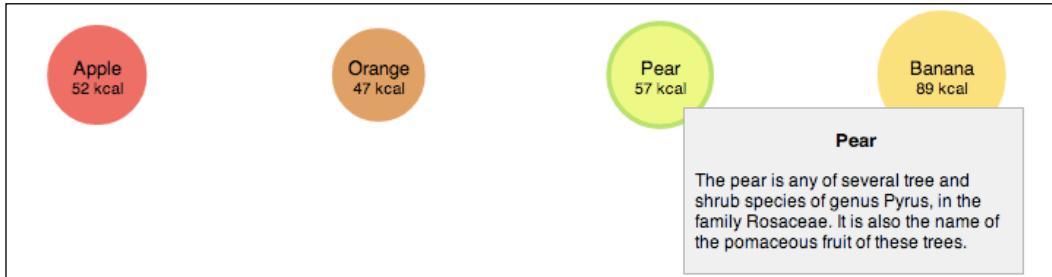


Remember that the circles already have listeners for the mouseover and mouseout events. If we add the tooltip as it is, the first listener will be removed before the new listener is added, disabling the highlighting.

To register multiple listeners for the same event type, we can add an optional namespace to the tooltip-related events, as follows:

```
// Tooltip charting function
function chart(selection) {
  selection.each(function(d) {
    // Bind the mouse events to the container element
    d3.select(this)
      .on('mouseover.tooltip', create)
      .on('mousemove.tooltip', move)
      .on('mouseout.tooltip', remove);
  });
}
```

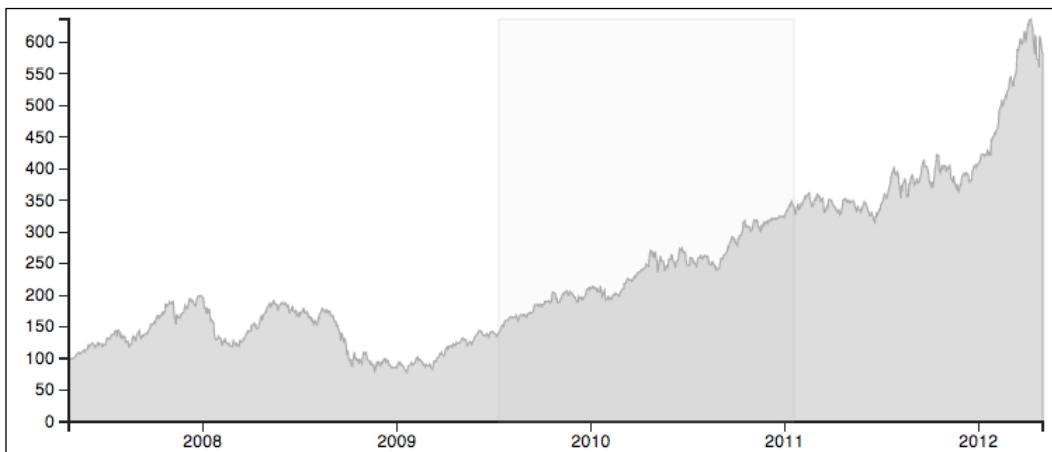
Now, we have the tooltips and highlighting enabled in the fruit chart.



The tooltip and the highlighting listeners are enabled for the fruit chart

Selecting a range with brushing

In this section, we will create an area chart to display stock prices and use brushing to allow the user to select an interval and get additional information about that time interval.



Selecting a time interval with brushing

We will use the time series of the prices of the AAPL stock, available as a TSV file in the D3 examples gallery. The file contains the date and closing price for the date, covering almost 5 years of activity, as shown in the following code:

```
date      close
1-May-12  582.13
30-Apr-12 583.98
27-Apr-12 603.00
26-Apr-12 607.70
...
...
```

Creating the area chart

We begin by creating the structure of a reusable chart; we will add the width, height, and margin as the chart attributes, and add their corresponding accessors. The complete code is available in the `chapter05/02-brushing.html` file. In the charting function, we initialize and set the size of the `svg` element as follows:

```
// Chart Creation
function chart(selection) {
    selection.each(function(data) {

        // Select the container element and create the svg selection
        var div = d3.select(this),
            svg = div.selectAll('svg').data([data]);

        // Initialize the svg element
        svg.enter().append('svg')
            .call(svgInit);

        // Initialize the svg element
        svg.attr('width', width).attr('height', height);

        // Creation of the inner elements...
    });
}
```

The `svgInit` function will be called only on enter, and it will create groups for the axis and the chart content. We will parse the input data, so we don't have to transform each item later. To parse the date, we will use the `d3.time.format` D3 method:

```
// Configure the time parser
var parseDate = d3.time.format(timeFormat).parse;

// Parse the data
```

```
data.forEach(function(d) {  
    d.date = parseDate(d.date);  
    d.close = +d.close;  
});
```

The `timeFormat` variable is defined as a chart attribute; we also added an accessor function so that the user can use the chart for other datasets. In this case, the input date format is `%d-%b-%y`, that is, the day, abbreviated month name, and year.

We can now create the *x* and *y* axis:

```
// Create the scales and axis  
var xScale = d3.time.scale()  
    .domain(d3.extent(data, function(d) { return d.date; }))  
    .range([0, width - margin.left - margin.right]);  
  
// Create the x axis  
var xAxis = d3.svg.axis()  
    .scale(xScale)  
    .orient('bottom');  
  
// Invoke the xAxis function on the corresponding group  
svg.select('g.xaxis').call(xAxis);
```

We do the same with the *y* axis, but we will use a linear scale instead of the time scale and orient the axis to the left side. We can now create the chart content; we create and configure an area generator that will compute the path and then append the path to the chart group, as follows:

```
// Create and configure the area generator  
var area = d3.svg.area()  
    .x(function(d) { return xScale(d.date); })  
    .y0(height - margin.top - margin.bottom)  
    .y1(function(d) { return yScale(d.close); });  
  
// Create the area path  
svg.select('g.chart').append("path")  
    .datum(data)  
    .attr("class", "area")  
    .attr("d", area);
```

We will modify the styles for the classes of the axis groups and the area to have a better-looking chart, as follows:

```
<style>
.axis path, line{
  fill: none;
  stroke: #222;
  shape-rendering: crispEdges;
}
.axis text {
  font-size: 11px;
}

.area {
  fill: #ddd;
}
</style>
```

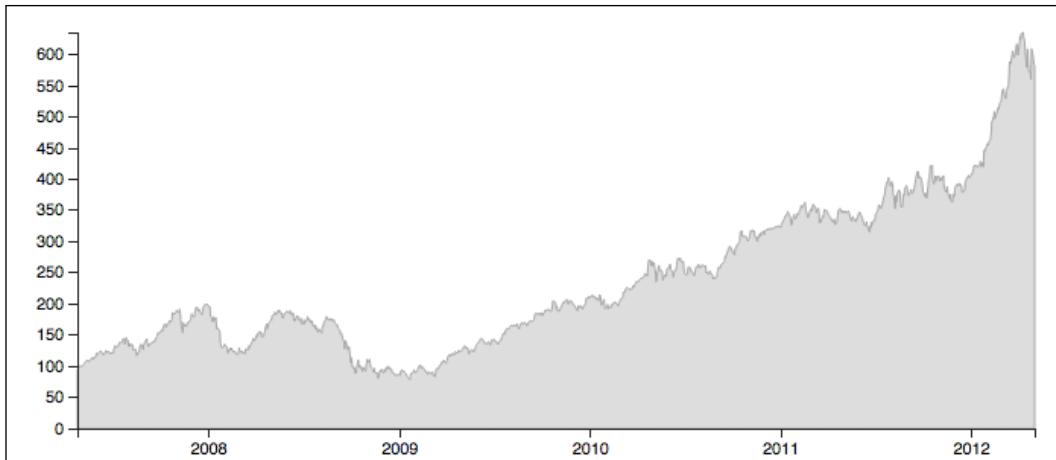
We load the dataset using the `d3.tsv` function, which retrieves and parses tabular delimited data. Next, we will configure the chart, select the container element, and bind the dataset to the selection as follows:

```
// Load the TSV Stock Data
d3.tsv('/chapter05/aapl.tsv', function(error, data) {

  // Handle errors getting or parsing the data
  if (error) {
    console.error(error);
    throw error;
  }

  // Create and configure the area chart
  var chart = areaChart();

  // Bind the chart to the container div
  d3.select('div#chart')
    .datum(data)
    .call(chart);
});
```



The first version of the area chart

Adding brushing

A brush is a control that allows you to select a range in a chart. D3 provides built-in support for brushing. We will use brushing to select time intervals in our area chart, and use it to show the price and date of the edges of the selected interval. We will also add a label that shows the relative price variation in the interval. We will create an SVG group to contain the brush elements. We will add this group in the `svgInit` method, as follows:

```
// Create and translate the brush container group
svg.append('g')
    .attr('class', 'brush')
    .attr('transform', function() {
        var dx = margin.left, dy = margin.top;
        return 'translate(' + [dx, dy] + ')';
});
```

The group should be added at the end of the `svg` element to avoid getting it hidden by the other elements. With the group created, we can add the brush control in the charting function as follows:

```
function chart(selection) {
    selection.each(function(data) {
        // Charting function contents...

        // Create and configure the brush
```

Creating User Interface Elements

```
var brush = d3.svg.brush()  
    .x(xScale)  
    .on('brush', brushListener);  
});  
}
```

We set the scale of the brush in the horizontal axis, and add a listener for the brush event. The brush can be configured to select a vertical interval by setting the *y* attribute with an appropriate scale and even be used to select areas by setting both the *x* and *y* attributes.

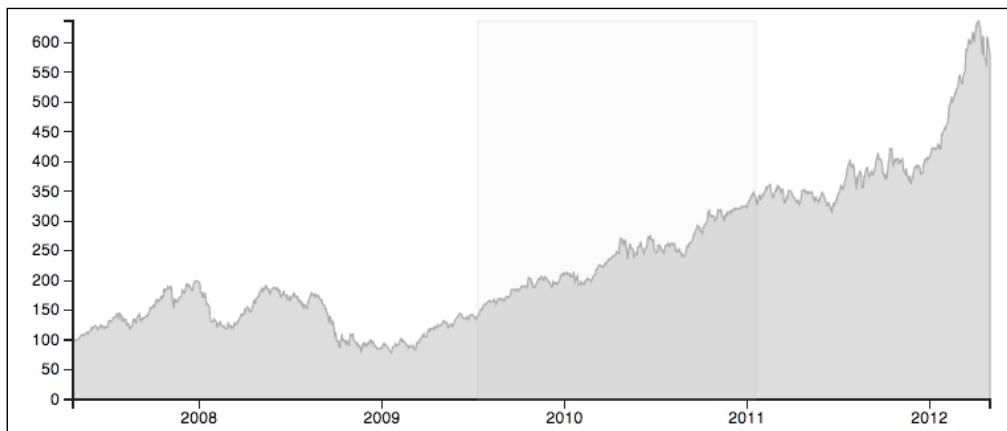
The `brushListener` function will be invoked if the brush extent changes. The `brushstart` and `brushend` events are also available, but we don't need to use them at the moment. In the following code, we apply the `brush` function to the brush group using the `call` method of the selection:

```
var gBrush = svg.select('g.brush').call(brush);
```

When we invoke the `brush` function in a group, a series of elements are created. A background rectangle will capture the brush events. There will also be a rectangle of the `extent` class, which will resize as the user changes the brush area. Also, there are two invisible vertical rectangles at the brush edges; so, it's easier for the user to select the brush boundary. The rectangles will initially have zero height; we will set the height to cover the chart area:

```
// Change the height of the brushing rectangle  
gBrush.selectAll('rect')  
    .attr('height', height - margin.top - margin.bottom);
```

We will modify the `extent` class, so the selected region is visible. We will set its color to gray and set the fill opacity to 0.05.



Adding brushing to the chart

The brush listener

We will add lines to mark the prices at the beginning and end of the selected period, and add a label to display the price variation in the interval. We begin by adding the elements in the `chart.svgInit` function and set some of its attributes. We will create groups for the line markers and for the text elements that will display the price and date. We also add a text element for the price variation. We create the `brushListener` function in the charting function scope, as shown in the following code:

```
// Brush Listener function
function brushListener() {
    var s = d3.event.target.extent();

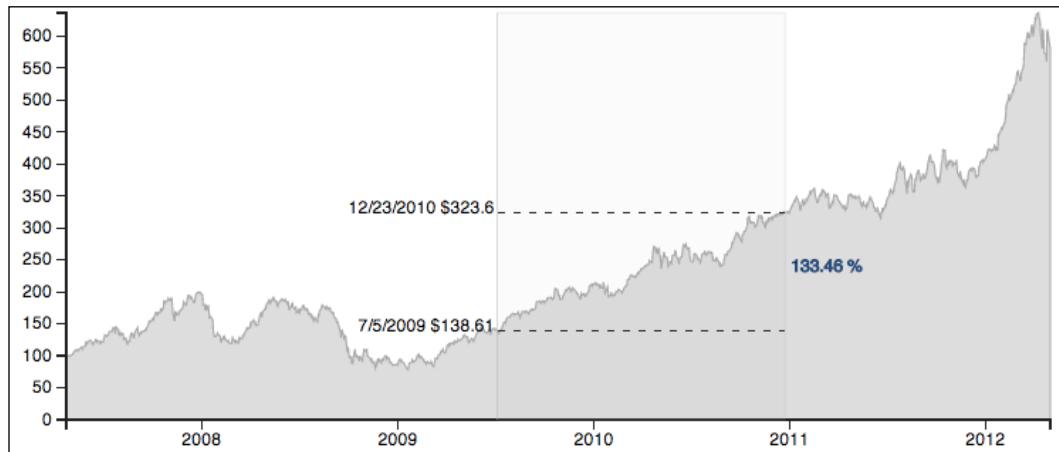
    // Filter the items within the brush extent
    var items = data.filter(function(d) {
        return (s[0] <= d.date) && (d.date<= s[1]);
    });
}
```

When the brush event is triggered, the brush listener will have access to the event attributes through the `d3.event` object. Here, we get the brush extent and use it to filter the dates that lie within the selected interval. Note that the selection is an approximation, because there are a limited number of pixels in the screen. At the beginning of the brush event, the time interval might be too small to contain data items. We will compute the prices only when at least two items have been selected. We then select the first and last elements of the item array, as follows:

```
// Compute the percentual variation of the period
if (items.length > 2) {
    // Get the prices in the period
    priceB = items[0].close;
    priceA = Math.max(items[n - 1].close, 1e-8);

    // Set the lines and text position...
}
```

Having the first and last elements of the selected period, we can compute the relative price variation and set the position of the marker lines and labels. We will also set the color of the variation label to `blue` if the variation is positive and to `red` if it's negative. As the configuration of the positions and labels is rather large, we won't include the code here. However, the code is available in the `chapter05/02-brushing.html` file for reference.



The area chart with brushing and annotations

Summary

In this chapter, we learned how to highlight elements when a user moves the pointer over them, making it easier for the user to spot the elements under the cursor and give hints of which elements can provide additional interactions. We created a reusable tooltip component that can be configured and used in other charts.

We also learned how to use the built-in brush component to create a control in order to select intervals, and used this control to allow the user to select a time interval in an area chart. We used the brush area to further annotate the chart with useful information about variations and the edges of the selected time interval.

In the next chapter, we will learn how to add interaction between chart components and how to integrate D3 and Backbone to create complex applications.

6

Interaction between Charts

Visualization projects are usually implemented as single page applications. Single page applications usually load their code when the browser loads the page and make requests to retrieve additional data when the user interacts with the page, avoiding full page reloads. The application can be used while the request is fulfilled, thereby improving the user experience.

Single page applications generally have a single payload that retrieves the scripts, styles, and markup required to create the interface. When the user interacts with UI components and additional data is required, the client-side code makes asynchronous requests to the server in the background and updates the corresponding elements when the data is ready, allowing the user to continue using the application during the request.

This increase in the complexity of client-side applications has led frontend developers to improve the architecture of client-side components. One of the most successful designs to face these challenges is the **MVC pattern** and its many variations.

In this chapter, we will cover the basics of the Backbone library and use D3 with Backbone to create a stock explorer with several components interacting between them, maintaining a consistent visualization state. We will also learn how to update the application URL to reflect a particular state in the application, allowing users to create bookmarks, navigate, and share the application state.

Learning the basics of Backbone

Backbone is a JavaScript library that helps us structure applications by implementing a version of the **MV* pattern**, which helps us separate different application concerns. The main Backbone components include models, collections, views, and routers; all these components communicate among themselves by triggering and listening to events.

The only hard dependency of Backbone is Underscore, which is a small utility library that provides functional programming support for collections, arrays, functions, and objects. It also provides additional utilities, such as a small template engine, which we will use later. To use the Backbone router and manipulate the DOM, jQuery or Zepto must also be included.

Events

The Events module can be used to extend an object, giving it the ability to listen and trigger custom events, such as listening for key presses, or sending an event when a variable changes. Backbone models, views, collections, and routers have event support. When we include Backbone in a page, the Backbone object will be available, and it can be used to listen or trigger events.

Models

In Backbone, a model is a data container. Model instances can be created, validated, and persisted to a server endpoint. When an attribute of the model is set, the model triggers a change event. The granularity of the change events allows observers to listen for a change in a particular attribute or any attribute in the model.

Besides being data containers, models can validate or convert data from its original format. Models are ignorant of views or external objects observing its changes; a model only communicates when its attributes are changed.

Collections

Collections are ordered sets of models, which are useful in order to manage model instances as a set. If a model instance in the collection is modified (added or removed), the collection will trigger a change event (triggering the add and remove events in each case).

Collections also have a series of enumerable methods that are useful to iterate, find, group, and compute aggregate functions on the collection elements. Of course, collections can be also extended to add new methods. Collections can be synced to the server in order to retrieve records and create new model instances from them and to push model instances created within the application.

Views

The views are components that render one or more attributes of a model in the page. Each view has one DOM element bound to it and a corresponding model (or collection) instance. Usually, the views listen for changes in one or more attributes of a model. Views can be updated from each other independently when a model (or some attributes of a model) changes, updating the view without redrawing the entire page. Note that when using Backbone models and views, the DOM elements in a view don't have references to the corresponding data elements; the view stores references to the DOM elements and the model. In D3, the DOM element contains a reference to the data bound to it.

Views also listen for DOM elements inside the container element. We can bind DOM events in a child element to a view method. For instance, if a view contains a button, we can bind the `click` event of the button with the custom `toggleClicked` method of the view. This is commonly used to update a model attribute.

In most Backbone applications, the views are rendered using templates. In most of the views, we will use D3 to render them instead of templates, but we will also include an example of a view that has been rendered using Underscore templates.

Routers

The Backbone router allows you to connect URLs to the application, allowing the application states to have URLs. This allows the user to navigate between visited application states using the browser's back and forth buttons, save a bookmark, and share specific application states.

Backbone is a subject on its own and we can't cover all its features in one chapter. It's a good idea to invest some time learning Backbone or one of its alternatives.

There are a great number of resources available to help you learn Backbone. The most complete references are the following:



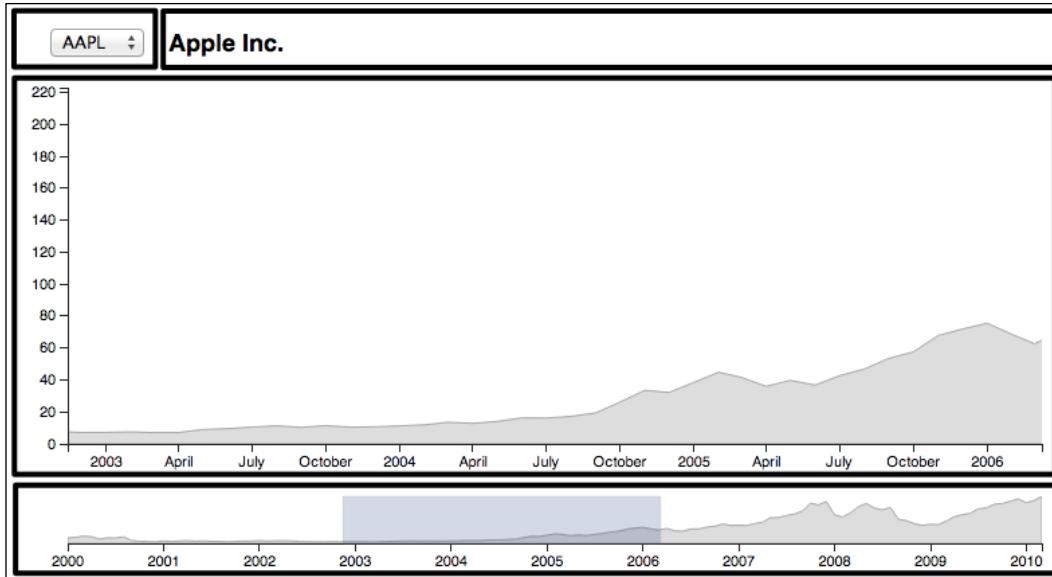
- **Backbone Fundamentals** (<http://addyosmani.github.io/backbone-fundamentals>): This book, written by Addy Osmani, describes the Backbone components in depth and has two complete examples of Backbone-based applications.
- **Backbone** (<http://backbonejs.org>): The official website contains the documentation of the library. The source code of Backbone is also extensively commented on.
- **TodoMVC** (<http://todomvc.com>): As the number of JavaScript MV* frameworks and libraries that allow you to structure an application can be overwhelming, the TodoMVC project contains the same Todo application that was implemented in the most popular MV* JavaScript frameworks available.

The stock explorer application

In this section, we will use D3 and Backbone to create a single page application to display a time series of stock prices. The user will be able to select different stocks and a period of time in order to get a detail view. The page will have several components:

- **Context chart:** This chart will display the complete series of prices that are available for the stock. It will have a brush component that selects a time interval.
- **Detail chart:** This will be a bigger area chart that will show you the stock prices for the time interval selected in the context chart.
- **Control view:** This view will show you a control that selects the stock.
- **Stock title:** This will display the name of the company.

We will display the control and title view on top of the page, a big area with the detail view, and a context area chart at the bottom of the page.



A diagram of the application components

As you can see, there are several components that should be in sync. If we change the stock, the area chart should be updated. If we change the time interval in the context chart, the detail chart must be updated to show you only the selected period. We will use Backbone to structure this application.

The state of our application can be described by the stock and the time interval that we want to examine. We will create a model to store the application state and one view for each component.

[ The general strategy is that each view will contain an instance of a D3-based chart, which is created following the reusable chart pattern. In the `initialize` method, we will tell the view to listen for changes in the model and invoke the `render` method when one of the model attributes is changed. In the `render` method of the view, we will create a selection for the container element of the view, bind the data corresponding to the selected stock, and invoke the charting function using the `selection.call` method. We will begin by creating reusable charts with D3.]

Creating the stock charts

In this section, we will implement the charts that will be used by the application. This time, the code of the charts will be in a separated JavaScript file. To follow the examples in this section, open the `chapter06/stocks/js/lib/stockcharts.js` and `chapter06/01-charts.html` files.

We will begin by creating the stock title chart and then implement the stock area chart, which we will be using in the context and detail views. Note that the title chart is not really necessary, but it will be helpful to introduce the pattern that integrates reusable charts and Backbone.

The stock title chart

The stock title chart is a reusable chart that creates a paragraph with the title of the stock. As we mentioned previously, it's probably not a good idea to create a chart just to write a string, but it shows you how to integrate a reusable chart that doesn't involve SVG with Backbone. It has a configurable title accessor function, so the user can define the content of the paragraph using the data that is bound to the container selection. The chart is structured using the reusable chart pattern, as shown in the following code:

```
function stockTitleChart() {
    'use strict';

    // Default title accessor
    var title = function(d) { return d.title; };

    // Charting function
    function chart(selection) {
        selection.each(function(data) {
            // Creation and update of the paragraph...
        });
    }

    // Title function accessor
    chart.title = function(titleAccessor) {
        if (!arguments.length) { return title; }
        title = titleAccessor;
        return chart;
    };

    return chart;
}
```

In the charting function, we select the `div` element and create a selection for the paragraph. We add the `stock-title` class to the paragraph in order to allow the user to modify its style, as shown in the following code:

```
// Charting function
function chart(selection) {
    selection.each(function(data) {

        // Create the selection for the title
        var div = d3.select(this),
            par = div.selectAll('p.stock-title').data([data]);

        // Create the paragraph element on enter
        par.enter().append('p')
            .attr('class', 'stock-title');

        // Update the paragraph content
        par.text(title);
    });
}
```

As usual, we can use the chart by creating and configuring a chart instance, selecting the container element, binding the data, and invoking the chart using the `selection.call` method, as shown in the following code:

```
// Create and configure the title chart
var titleChart = stockTitleChart()
    .title(function(d) { return d.name; });

// Select the container element, bind the data and invoke
// the charting function on the selection
d3.select('div#chart')
    .data([{name: 'Apple Inc.'}])
    .call(titleChart);
```

The stock area chart

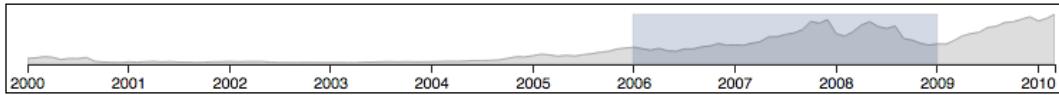
The stock area chart will display the time series for the stock price as an area chart. In *Chapter 5, Creating User Interface Elements*, we implemented an area chart that uses the brush behavior to select a time interval and annotate the chart with additional information about the price variation in the period. We will create an improved version of this chart and use it in the stock explorer application.

Besides having the usual width, height, and margin attributes and accessors methods, this chart will have an optional axis, brush behavior, and a configurable brush listener function so that the user can define actions to be performed on the brush. The time extent can be also be configured, allowing the user to show only part of the chart.

We have added all these methods so that we can use two chart instances for different purposes: one to allow the user to select a time interval and another to display the selected time interval in more detail. In the `chapter06/01-charts.html` file, we created one instance in order to select the time interval:

```
var contextAreaChart = stockAreaChart()
  .height(60)
  .value(function(d) { return d.price; })
  .yaxis(false)
  .onBrushListener(function(extent) {
    console.log(extent);
  });
});
```

We will use the chart accessor methods to set the height, disable the *y* axis, and set the value accessor and the brush listener functions. In this case, the brush listener function will display the time extent in the browser console on brush.



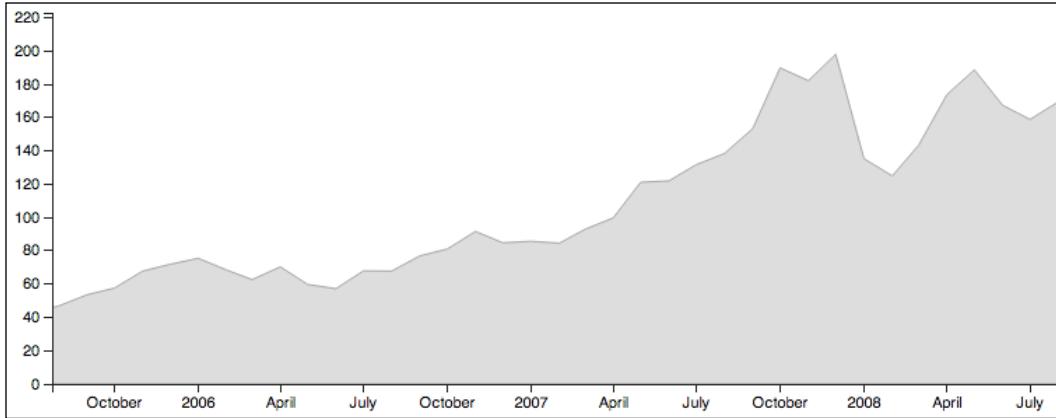
A stock area chart with the brush behavior enabled

We will use a second instance of the same chart to display a specific time interval. In this instance, we will disable the brush control and set the initial time extent, the value, and the date accessors. This chart will display the stock prices between the `from` and `to` dates:

```
// Set the time extent
var from = new Date('2002/01/01'),
  to = new Date('2004/12/31');

// Create and configure the detail area chart
var detailAreaChart = stockAreaChart()
  .value(function(d) { return d.price; })
  .date(function(d) { return new Date(d.date); })
  .timeExtent([from, to])
  .brush(false);
```

As you have probably guessed, the first instance is intended to control the time extent of the second chart instance. We will get to that soon; in the meantime, we will discuss some implications of controlling the time extent of the chart.



A stock area chart with the y-axis enabled and the brush behavior disabled

If we change the time extent of the chart, we will want the chart to reflect its new state. If the brush is dragged left in the first chart, we will want the area of the second chart to move to the right-hand side until it matches the interval selected in the first chart, and if we shorten the time interval in the first chart, we will want the area of the second chart to compress itself to display the selected interval in the same horizontal space.

The stock area chart will be implemented as a reusable chart. As most of the chart structure is similar to the chart presented in the previous section, we will skip some parts for brevity:

```
function stockAreaChart() {
    'use strict';

    // Chart Attributes
    var width = 700,
        height = 300,
        margin = {top: 20, right: 20, bottom: 20, left: 20};

    // Time Extent
    var timeExtent;

    // The axis and brush are enabled by default
    var yaxis = true,
        xaxis = true,
```

```
brush = true;

// Default accessor functions
var date = function(d) { return new Date(d.date); };
var value = function(d) { return +d.value; };

// Default brush listener
var onBrush = function(extent) {};

function chart(selection) {
    selection.each(function(data) {
        // Charting function contents...
    });
}

var svgInit = function(selection) { ... };

// Accessor Methods...

return chart;
}
```

In order to have the detail chart moving in sync with the context chart, we will need to draw the complete series in the detail chart but only displaying the interval selected with the brush in the context chart. To prevent the area chart from being visible outside the charting area, we will define a clip path, and only the content inside the clipping path will be visible:

```
var svgInit = function(selection) {
    // Define the clipping path
    selection.append('defs')
        .append('clipPath')
        .attr('id', 'clip')
        .append('rect')
        .attr('width', width - margin.left - margin.right)
        .attr('height', height - margin.top - margin.bottom);

    // Create the chart and axis groups...
};
```

The element that will be clipped should reference the clipping path using the `clip-path` attribute. In the `charting` function, we select the container element and create the SVG element on enter. We also set the SVG element's width and height and translate the axis, chart, and brush groups. We create the scales and axis (if they are enabled) and create and configure the area generator to draw the chart area path:

```
// Charting function...
// Add the axes
if (xaxis) { svg.select('g.xaxis').call(xAxis); }
if (yaxis) { svg.select('g.yaxis').call(yAxis); }

// Area Generator
var area = d3.svg.area()
  .x(function(d) { return xScale(date(d)); })
  .y0(yScale(0))
  .y1(function(d) { return yScale(value(d)); });
```

We create a selection for the path and bind the time series array to the selection. We append the path on enter and set its class to `stock-area`. We set the path data using the area generator and set the `clip-path` attribute using the `clipPath` variable defined previously:

```
// Create the path selection
var path = svg.select('g.chart').selectAll('path')
  .data([data]);

// Append the path element on enter
path.enter().append('path')
  .attr('class', 'stock-area');

// Set the path data string and clip the area
path.attr('d', area)
  .attr('clip-path', 'url(#clip)');
```

We create an envelope brush listener function. In this function, we retrieve the brush extent and invoke the user-configurable `onBrush` function, passing the extent as an argument. We initialize the brush behavior and bind the `brushListener` function to the brush event:

```
// Brush Listener Function
function brushListener() {
  timeExtent = d3.event.target.extent();
  onBrush(timeExtent);
}

// Brush Behavior
```

```
var brushBehavior = d3.svg.brush()  
  .x(xScale)  
  .on('brush', brushListener);
```

The initial time extent of the chart can be configured. If that's the case, we update the brush behavior extent, so the brush overlay fits the configured time extent:

```
// Set the brush extent to the time extent  
if (timeExtent) {  
  brushBehavior.extent(timeExtent);  
}
```

We call the brush behavior using the `selection.call` method on the brush group and set the overlay height:

```
if (brush) {  
  svg.select('g.brush').call(brushBehavior);  
  
  // Change the height of the brushing rectangle  
  svg.select('g.brush').selectAll('rect')  
    .attr('height', h);  
}
```

The preceding charts are implemented following the reusable chart pattern and were created with D3 only. We will use the charts in a Backbone application, but they can be used in other applications as standalone charts.

Preparing the application structure

In Backbone projects, it is a common practice to create directories for the models, views, collections, and routers. In the `Chapter06` directory, we created the `stocks` directory to hold the files for this application:

```
stocks/  
  css/  
  js/  
    views/  
    models/  
    collections/  
    routers/  
    lib/  
    app.js  
  data/  
  index.html
```

The `models`, `views`, `collections`, and `routers` folders contain JavaScript files that contain the Backbone models, views, collections, and routers. We add the D3 charts to the `js/lib` directory; additional JavaScript libraries would be there too. There is also a `data` folder with JSON files for the stock data. The `index.html` file contains the application markup.

The index page

In the header of the page, we include style sheets and JavaScript libraries that we need for our application. To create the page more quickly, we will use a CSS library that will add styles to enable uniform fonts, sizes, and default colors among browsers and define the grid system. A grid system is a set of styles that allows us to define rows and columns of standard column sizes without having to define the styles for each size ourselves. We will use Yahoo's Pure CSS modules to use the grid system, which is a pretty minimal set of CSS modules. These modules are used only in this page; if you are more comfortable with Bootstrap or other libraries, you are free to replace the `div` classes or define the sizes and behaviors of each container yourself.

We will create a container for the application and add the `pure-g-r` class, which is a container with responsive behavior enabled. If the viewport is wide, the columns will be shown side by side; if the user has a small screen, the columns will be shown stacked. We will also create two child containers, one for the stock control and title and a second container for the stock area chart, both classed `pure-u-1`, that is, containers with full width. The `pure` container uses fractional sizes to define the `div` width; in order to have a `div` that covers 80 percent of the parent container width, we can set its class to `pure-u-4-5`:

```
<div class="pure-g-r" id="stock-app">
    <!-- Stock Selector and Title -->
    <div class="pure-u-1">
        <div id="stock-control"></div>
        <div id="stock-title"></div>
    </div>
    <div class="pure-u-1 charts">
        <div id="stock-detail"></div>
        <div id="stock-context"></div>
    </div>
</div>
```

We include the application files at the end of the page so that the markup is rendered while the remaining assets are loaded:

```
<!-- Application Components -->
<script src="/chapter06/stocks/js/models/app.js"></script>
<script src="/chapter06/stocks/js/models/stock.js"></script>
```

```
<script src="/chapter06/stocks/js/collections/stocks.js"></script>
<script src="/chapter06/stocks/js/views/stocks.js"></script>
<script src="/chapter06/stocks/js/views/app.js"></script>
<script src="/chapter06/stocks/js/routers/router.js"></script>
<script src="/chapter06/stocks/js/app.js"></script>
```

Creating the models and collections

Models contain application data and the logic related to this data. For our application, we will need a model to represent the stock information and the application model, which will store the visualization state. We will also create a collection that holds the available stock instances. To avoid polluting the global namespace, we will encapsulate the application components in the app variable:

```
var app = app || {};
```

Adding this line to all the files in the application will allow us to extend the object with models, collections, and views.

The stock model

The stock model will contain basic information about each stock. It will contain the stock name (Apple Inc.), the symbol (AAPL), and the URL where the time series of prices can be retrieved (`aapl.json`). Models are created by extending Backbone.Model:

```
// Stock Information Model
app.Stock = Backbone.Model.extend({
```



```
    // Default stock symbol, name and url
    defaults: {symbol: null, name: null, url: null},
```



```
    // The stock symbol is unique, it can be used as ID
    idAttribute: 'symbol'
```

```
});
```

Here, we defined the default values for the model to `null`. This is not really necessary, but it might be useful to know which properties are expected. We will use the stock symbol as `ID`. Besides this, we don't need any further initialization code. As stock symbols are unique, we will use the symbol as the ID for easier retrieval later. We can create stock instances by using the constructor and setting the attributes that pass an object:

```
var appl = new app.Stock({
    symbol: 'AAPL',
```

```
        name: 'Apple',
        url: 'aapl.json'
    }) ;
```

We can set or get its attributes using the accessor methods:

```
aapl.set('name', 'Apple Inc.');
aapl.get('name'); // Apple Inc.
```

In this application, we will create and access stock instances using a collection rather than creating individual instances.

The stock collection

To define a collection, we need to specify the model. When defining the collection, we can set the URL of an endpoint where the collection records can be retrieved, which is usually the URL of a REST endpoint. In our case, the URL points towards a static JSON file that contains the stocks records:

```
// Stock Collection
app.StockList = Backbone.Collection.extend({
    model: app.Stock,
    url: '/chapter06/stocks/data/stocks.json'
}) ;
```

Individual stocks can be added to the collection one by one, or they can be fetched from the server using the collection URL. We can also specify the URL when creating the collection instance:

```
// Create a StockList instance
var stockList = new app.StockList({});

// Add one element to the collection
stockList.add({
    symbol: 'AAPL',
    name: 'Apple Inc.',
    url: 'aapl.json'
});
stockList.length; // 1
```

As we defined the stock symbol as `idAttribute`, individual stock instances can be retrieved using the stock's ID. In this case, the stock symbol is the ID of the stock model, so we can retrieve stock instances using the symbol:

```
var aapl = stockList.get('AAPL');
```

Models use the URL of the collection to construct their own URL. The default URL will have the form `collectionUrl/modelId`. If the server provides a RESTful API, this URL can be used to create, update, and delete records.

The application model

We will create an application model to store and manage the application state.

To define the application model, we extend the `Backbone.Model` object, adding the corresponding default values. The `stock` attribute will contain the current stock symbol (AAPL), and the `data` will contain the time series for the current stock:

```
// Application Model
app.StockAppModel = Backbone.Model.extend({  
  
    // Model default values  
    defaults: {  
        stock: null,  
        from: null,  
        to: null,  
        data: []  
    },  
  
    initialize: function() {  
        this.on('change:stock', this.fetchData);  
        this.listenTo(app.Stocks, 'reset', this.fetchData);  
    },  
  
    // Additional methods...  
    getStock: function() {...},  
    fetchData: function() {...}  
});
```

We will also set a template for the stock collection data. In this case, the base URL is `chapter06/stocks/data/`. As we mentioned previously, there is a JSON file in the `data` directory with the data of the available stocks:

```
// Compiled template for the stock data url
urlTemplate: _.template('/chapter06/stocks/data/<%= url %>'),
```

We have also added a `fetchData` method in order to retrieve the time series for the corresponding stock. We invoke the template that passes the current stock data and use the parsed URL to retrieve the stock time series. We use the `d3.json` method to get the stock data and set the model data attribute to notify the views that the data is ready:

```
fetchData: function() {
    // Fetch the current stock data
    var that = this,
        stock = this.getStock(),
        url = this.urlTpl(stock.toJSON());

    d3.json(url, function(error, data) {
        if (error) { return error; }
        that.set('data', data.values);
    });
}
```

Implementing the views

To integrate the D3-based charts with Backbone Views, we will use the following strategy:

1. We will create and configure a chart instance as an attribute of the view.
2. In the initialization method, we tell the view to listen for changes on the model application and render the view on model updates.

The views for the page components are in the `chapter06/stocks/js/views/stocks.js` file, and the application view code is in the `chapter06/stocks/js/views/app.js` file.

The title view

This view will simply display the stock symbol and name. It's intended to be used as a title of the visualization. We create and configure an instance of the underlying chart and store a reference to the chart in the `chart` attribute. In the `initialize` method, we tell the view to invoke the `render` method when the model's `stock` attribute is updated.

In the `render` method, we create a selection that will hold the container element of the view, bind this element to a dataset that contains the current stock, and invoke the chart using `selection.chart`:

```
app.StockTitleView = Backbone.View.extend({  
  
  chart: stockTitleChart()  
    .title(function(d) {  
      return _.template('<%= symbol %><%= name %>', d);  
    }),  
  
  initialize: function() {  
    this.listenTo(this.model, 'change:stock', this.render);  
    this.render();  
  },  
  
  render: function() {  
    d3.select(this.el)  
      .data([this.model.getStock().toJSON()])  
      .call(this.chart);  
  
    return this;  
  }  
});
```

Changes to the `stock` attribute of the application model will trigger the `change:stock` event, causing the view to invoke its `render` method, updating the D3 chart. In this particular view, using a reusable chart is overkill; for a real-life problem, we could have used a small Backbone View with a template. We did this to have a minimal example of reusable charts working with Backbone Views.



Rendered stock title view

The stock selector view

To add some diversity, we will create the selector without using D3. This view will show you the available stocks as a selection menu, updating the application model's `stock` attribute when the user selects a value. In this view, we will use a template. To create a template, we create a `script` element of type `text/template` in the `index.html` file and assign it an ID, in our case, `stock-selector-tpl`:

```
<script type="text/template" id="stock-selector-tpl">
```

```
<select id="stock-selector">
  ...
</select>
</script>
```

Underscore templates can render variables using `<%= name %>` and execute JavaScript code using `<% var a = 1; %>`. Here, for instance, we evaluate the callback function on each element of the `stocks` array:

```
<!-- Create the stocks selector and add its options -->
<% _.each(stocks, function(s) { %>
  <option value="<%= s.symbol %>"><%= s.symbol %></option>
<% }) ; %>
```

For each one of the elements of the `stocks` array, we add an option with the stock symbol attribute as the value and content of the option element. After rendering the template with the application data, the HTML markup will be as follows:

```
<select id="stock-selector">
  <option value="AAPL">AAPL</option>
  <option value="MSFT">MSFT</option>
  <option value="IBM">IBM</option>
  <option value="AMZN">AMZN</option>
</select>
```

In the Backbone View, we select the content of the script with the `stock-selector-tpl` ID, compile the template to use it later, and store a reference to the compiled template in the `template` attribute:

```
// Stock Selector View
app.StockSelectorView = Backbone.View.extend({
  // Compiles the view template
  template: _.template($('#stock-selector-tpl').html()),

  // DOM Event Listeners
  events: {
    'change #stock-selector': 'stockSelected'
  },

  // Initialization and render methods...
});
```

We set the `events` attribute, with maps' DOM events of the inner elements of the view, to methods of the view. In this case, we bind changes to the `stock-selector` select element (the user changing the stock) with the `stockSelected` method.

In the `initialize` method, we tell the view to render when the `app.Stocks` collection emits the `reset` event. This event is triggered when new data is fetched (with the `{reset: true}` option) and when an explicit reset is triggered. If a new set of stocks is retrieved, we will want to update the available options. We also listen for changes to the application model's `stock` attribute. The current stock should always be the selected option in the select element:

```
initialize: function() {
  // Listen for changes to the collection and the model
  this.listenTo(app.Stocks, 'reset', this.render);
  this.listenTo(this.model, 'change:stock', this.render);
  this.render();
}
```

In the `render` method, we select the container element and set the element content to the rendered template, writing the necessary markup to display the drop-down control. We pass a JavaScript object with the `stock` attribute set to an array that contains the `app.Stocks` model's data. Finally, we iterate through the options in order to mark the option that matches the current stock symbol as selected:

```
render: function() {
  // Stores a reference to the 'this' context
  var self = this;

  // Render the select element
  this.$el.html(this.template({stocks: app.Stocks.toJSON()}));

  // Update the selected option
  $('#stock-selector option').each(function() {
    this.selected = (this.value === self.model.get('stock'));
  });
}
```

Backbone models and collection instances have the `JSON` method, which transforms the model or collection attributes to a JavaScript object. This method can be overloaded if we need to add computed properties besides the existing attributes. Note that in the `each` callback, the `this` context is set to the current DOM element, that is, the option element. We store a reference to the `this` context in the `render` function (the `self` variable) in order to reference it later.



Stock selector view allows selecting a stock by symbol

The stock context view

The context view contains a small area chart that allows the user to select a time interval that can be displayed in the detail view:

We will use the same strategy as the one used in the previous views to create and configure an instance of `stockAreaChart`, and store a reference to it in the `chart` attribute of the view:

```
app.StockContextView = Backbone.View.extend({
    // Initialize the stock area chart
    chart: stockAreaChart()
        .height(60)
        .margin({top: 5, right: 5, bottom: 20, left: 30})
        .date(function(d) { return new Date(d.date); })
        .value(function(d) { return +d.price; })
        .yaxis(false),

    // Render the view on model changes
    initialize: function() { ... },

    render: function(e) { ... }
});
```

In the `initialize` method, we tell the view to listen for changes to the application model and set the chart brush listener to update the `from` and `to` attributes of the model:

```
initialize: function() {
    // Get the width of the container element
    var width = parseInt(d3.select(this.el).style('width'), 10);

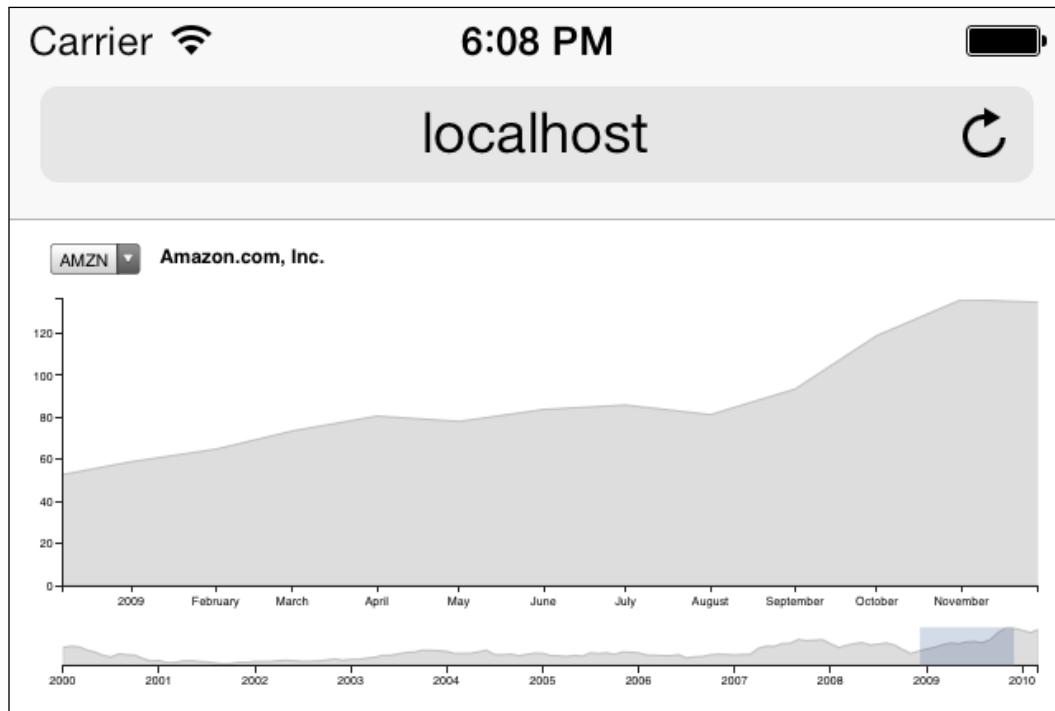
    // Bind the brush listener function. The listener will update
    // the model time interval
    var self = this;

    this.chart
        .width(width)
        .brushListener(function(extent) {
            self.model.set({from: extent[0], to: extent[1]});
        });

    // The view will render on changes to the model
    this.listenTo(this.model, 'change', this.render);
},
```

Interaction between Charts

We get the width of the `this.el` container element using D3 and set the chart width. This will make the chart use the full width of the user's viewport.



The chart fills the container width in Safari Mobile (iPhone Simulator)

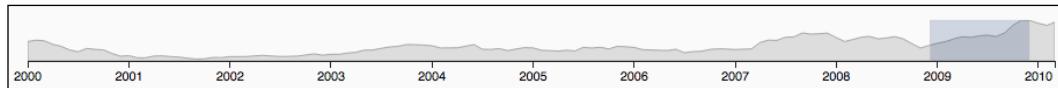
The render method will update the chart's time extent, so it reflects the current state of the model, creates a selection that holds the container element of the view, binds the stock data, and invokes the chart using `selection.call`:

```
render: function() {
    // Update the time extent
    this.chart
        .timeExtent([
            this.model.get('from'),
            this.model.get('to')
        ]);

    // Select the container element and call the chart
    d3.select(this.el)
        .data([this.model.get('data')])
        .call(this.chart);

    return this;
}
```

This view is the only component that can change the `from` and `to` attributes of the model.



The stock context view uses the brush behavior to set the time interval

The stock detail view

The stock detail view will contain a stock area chart, showing only a given time interval. It's designed to follow the time interval selected in the stock context view.

We create and configure a `stockAreaChart` instance, setting the margin, value, and date accessors and disabling the brushing behavior:

```
// Stock Detail Chart
app.StockDetailView = Backbone.View.extend({  
  
    // Initialize the stock area chart
    chart: stockAreaChart()  
        .margin({top: 5, right: 5, bottom: 30, left: 30})  
        .value(function(d) { return +d.price; })  
        .date(function(d) { return new Date(d.date); })  
        .brush(false),  
  
    // Render the view on model changes
    initialize: function() { ... },  
    render: function() { ... }  
});
```

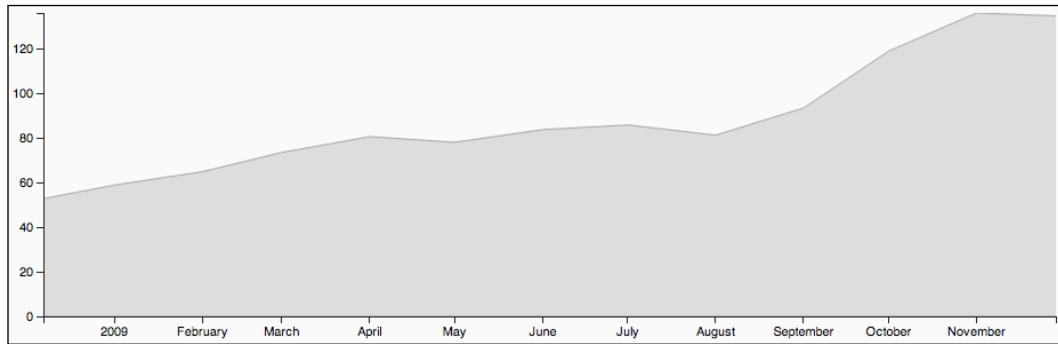
As we did in the context view, we tell the view to invoke the `render` method on model changes in the `initialize` method:

```
initialize: function() {  
  
    // Get the width of the container element
    var width = parseInt(d3.select(this.el).style('width'), 10);  
  
    // Set the chart width to fill the container
    this.chart.width(width);  
  
    // The view will listen the application model for changes
    this.listenTo(this.model, 'change', this.render);  
},
```

Interaction between Charts

In the `render` method, we update the chart time extent, so the visible section of the area chart matches the time interval specified by the application model's `from` and `to` attribute:

```
render: function() {  
  
    // Update the chart time extent  
    var from = this.model.get('from'),  
        to = this.model.get('to');  
  
    this.chart.timeExtent([from, to]);  
  
    // Select the container element and create the chart  
    d3.select(this.el)  
        .data([this.model.get('data')])  
        .call(this.chart);  
}
```



The detail view shows you the stock prices for the selected time interval

Note that when using `object.listenTo(other, 'event', callback)`, the `this` context in the callback function will be the object that listens for the events (`object`).

The application view

The application view will be in charge of creating instances of the views for each component of the application.

The `initialize` method binds the `reset` event of the `app.Stocks` collection and then invokes the collection's `fetch` method, passing the `{reset: true}` option. The collection will request the data to the server using its `url` attribute. When the data is completely loaded, it will trigger the `reset` event, and the application view will invoke its `render` method:

```
// Application View
app.StockAppView = Backbone.View.extend({

    // Listen to the collection reset event
    initialize: function() {
        this.listenTo(app.Stocks, 'reset', this.render);
        app.Stocks.fetch({reset: true});
    },

    render: function() { ... }
});
```

In the `render` method, we create instances of the views that we just created for each component. At this point, the current symbol of the application can be undefined, so we get the first stock in the `app.Stocks` collection and set the `stock` attribute of the model if it is not already set.

We proceed to initialize the views for the title, the selector, the context chart, and the detail chart, passing along a reference to the model instance and the DOM element where the views will be rendered:

```
render: function() {

    // Get the first stock in the collection
    var first = app.Stocks.first();

    // Set the stock to the first item in the collection
    if (!this.model.get('stock')) {
        this.model.set('stock', first.get('symbol'));
    }

    // Create and initialize the title view
    var titleView = new app.StockTitleView({
        model: this.model,
        el: 'div#stock-title'
    });

    // Create and initialize the selector view
```

Interaction between Charts

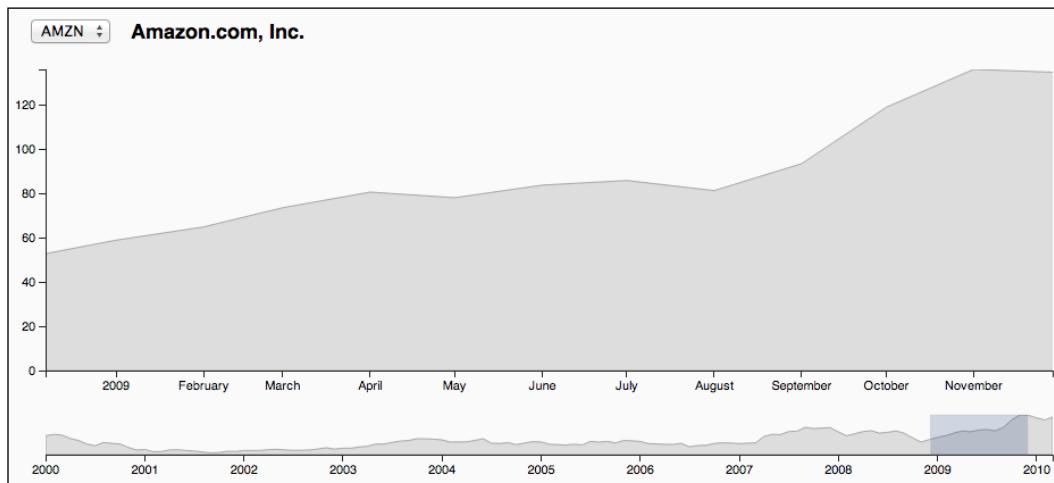
```
var controlView = new app.StockSelectorView({
    model: this.model,
    el: 'div#stock-control'
}) ;

// Create and initialize the context view
var contextView = new app.StockContextView({
    model: this.model,
    el: 'div#stock-context'
}) ;

// Create and initialize the detail view
var detailView = new app.StockDetailView({
    model: this.model,
    el: 'div#stock-detail'
}) ;

// Fetch the stock data.
this.model.fetchData() ;
return this;
}
```

Finally, we tell the model to fetch the stock data to allow the context and detail chart to be rendered. Remember that when the data is ready, the model will set its data attribute, notifying the charts to update its contents.



Components of the rendered application

Defining the routes

In our application, the state of the visualization can be described by the stock symbol and the time interval selected in the context chart. In this section, we will connect the URL with the application state, allowing the user to navigate (using the back button of the browser) the bookmark and share a particular state of the application.

We define the routes for our application by assigning callbacks for each hash URL (Backbone provides support for real URLs too). Here, we define two routes, one to set the stock and one to set the complete state of the application. If the user types the #stock/AAPL hash fragment, the `setStock` method will be invoked, passing the 'AAPL' string as the argument. The second route allows you to navigate to a specific state of the application using a URL fragment of the #stock/AAPL/from/Mon Dec 01 2003/to/Tue Mar 02 2010 form; this will invoke the `setState` method of the router:

```
app.StockRouter = Backbone.Router.extend({  
  
    // Define the application routes  
    routes: {  
        'stock/:stock': 'setStock',  
        'stock/:stock/from/:from/to/:to': 'setState'  
    },  
  
    // Initialize and route callbacks...  
});
```

The router also has an `initialize` method, which will be in charge of synchronizing changes in the application URL with changes in the application model. We will set the model for the router and configure the router to listen for change events of the model. At the beginning, the data might not have been loaded yet (and the `from` and `to` attributes might be undefined at this point); in this case, we set the stock symbol only. When the data finishes the loading, the `from` and `to` attributes will change and the router will invoke its `setState` method:

```
// Listen to model changes to update the url route  
initialize: function(attributes) {  
    this.model = attributes.model;  
    this.listenTo(this.model, 'change', function(m) {  
        if (m.get('from') && m.get('to')) {  
            this.setState(m.get('stock'), m.get('from'),  
m.get('to'));  
        } else {  
            this.setStock(m.get('stock'));  
        }  
    });  
},
```

The `setStock` method updates the `symbol` attribute of the model. The `navigate` method updates the browser URL to reflect the change of stock. Note that we are using the time interval as a variable of the application state. If we select an interval, the back button of the browser will get us to the previously selected time intervals. This might not be desirable in some cases. The choice of which variables should be included in the URL will depend on the application and the behavior that most users will expect. In this case for instance, an alternative approach could be to update the application state on drag start and drag end, not on every change in the interval:

```
// Set the application stock and updates the url
setStock: function(symbol) {
    var urlTpl = _.template('stock/<%= stock %>');
    this.model.set({stock: symbol});
    this.navigate(urlTpl({stock: symbol}), {trigger: true});
},
```

The `setState` method parses the `from` and `to` parameters from the URL as dates and sets the model's `stock`, `from`, and `to` attributes. As we cast the strings to the date, we can use any format recognizable by the date constructor (YYYY-MM-DD, for instance), but this will imply that we format the `from` and `to` attributes to this format when the model changes in order to update the URL. We will use the `toDateString` method to keep things simple. After setting the model state, we construct the URL and invoke the `navigate` method to update the browser URL:

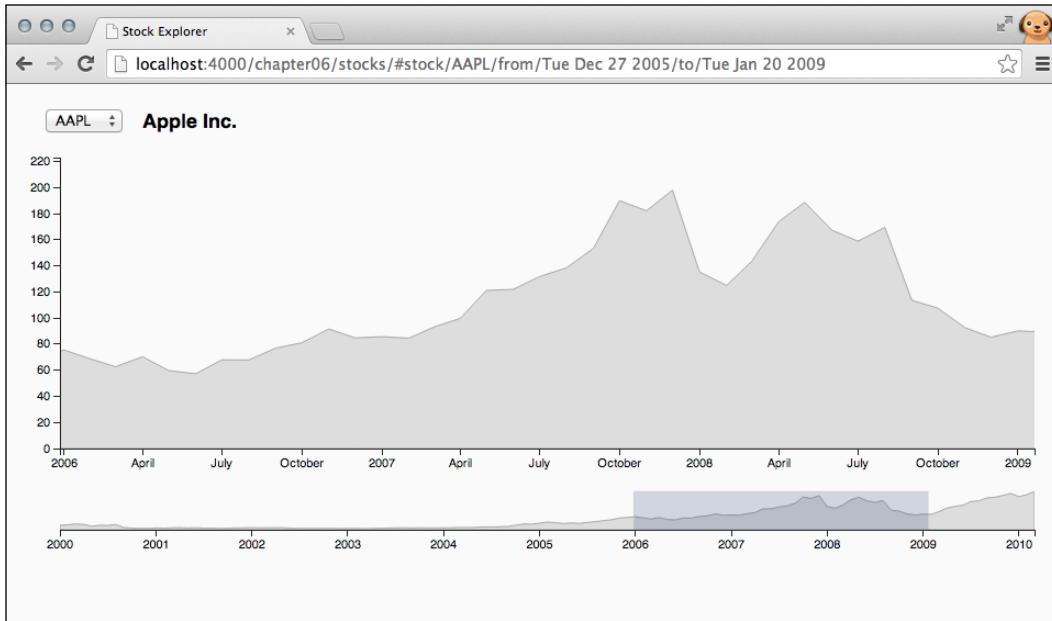
```
// Set the application state and updates the url
setState: function(symbol, from, to) {
    from = new Date(from),
    to = new Date(to);

    this.model.set({stock: symbol, from: from, to: to});

    var urlTpl = _.template('stock/<%= stock %>/from/<%= from
%>/to/<%= to %>'),
        fromString = from.toDateString(),
        toString = to.toDateString();

    this.navigate(urlTpl({stock: symbol, from: fromString,
        to: toString}), {trigger: true});
}
```

The simple addition of a router can make an application way more useful, allowing users to bookmark and share a particular state of the page and navigate back to previous states of the application.



The application state is displayed in the browser URL

Initializing the application

Once we have created the application models, collections, views, and router, we can create the instances for the application model and view. The application initialization code is in the `chapter06/stocks/js/app.js` file.

We begin by creating an instance of the `app.StockList` collection:

```
// Create an instance of the stocks collection
app.Stocks = new app.StockList();
```

The collection instances will be retrieved later.

We create an instance of the application model and an instance of the application view. We initialize the application model by indicating the model of the view and the container element ID:

```
// Create the application model instance
app.appModel = new app.StockAppModel();

// Create the application view
app.appView = new app.StockAppView({
  model: app.appModel,
  el: 'div#stock-app'
});
```

Finally, we initialize the router, passing the application model as the first argument, and then we tell Backbone to begin monitoring changes to hashchange events:

```
// Initializes the router
var router = new app.StockRouter({model: app.appModel});
Backbone.history.start();
```

Summary

In this chapter, you learned how to create a single page application that integrates D3 and Backbone. We used the reusable chart pattern and embedded the charts in Backbone Views, allowing us to enjoy the structure of Backbone and keep all the visualization components synchronized.

We created the stock explorer application. This application allows the user to choose and explore the time series of stock prices, allowing the user to select the stock and a time interval in order to have a detail view of the price variations in that period. We used Backbone to store the visualization state and the views in sync.

We used a router to connect the visualization state with the URL, allowing us to share, bookmark, and navigate through visited application states.

In the next chapter, you will learn how to create a charting package, which will contain a layout and a reusable chart. We will also learn how to configure it to make it easier to distribute, install, upgrade, and manage its dependencies on other packages.

7

Creating a Charting Package

Writing good quality software involves several tasks in addition to writing the code. Maintaining a code repository, testing, and writing consistent documentation are some of the tasks that need to be done when working with other people. This is also the case when we write charts and visualizations with D3. If we create a charting package, we would like to make it easy for others to use it and integrate it in their projects. In this chapter, we will describe a workflow and the tools that will help us to create a charting package. In this chapter, we will cover the following tasks:

- **Creating a repository:** A version control system should be used. In some cases, this involves configuring a central repository.
- **Designing the API:** Decide how to organize the code in logic units and how the package functionality will be exposed.
- **Writing the code:** Implement the package components and features.
- **Testing:** The package components should be tested to minimize the risk of introducing unexpected behavior and breaking the existing functionality.
- **Building:** The source code isn't shipped as is to build a package. This implies that at least source files need to be concatenated and a minified version of the distributable files should be created.
- **Hosting the package:** The package should be accessible for others to use, even if it's intended for internal usage.

Also, there are a number of conventions and norms about how these tasks should be performed. There may be protocols for the following:

- **Committing changes:** This is a workflow to know how to merge hot fixes or new features for the next release or the development version of the project. This may include tasks to be done before you push for changes, such as testing the code or having code review sessions.

- **Creating a release:** This is a procedure to create, tag, and release new versions of the software, including how to follow a system to tag releases with version numbers.
- **Writing code:** This is a set of coding practices and conventions agreed upon by the teams.

Most of these tasks and protocols depend on the team and the type of project, but it is almost certain that a number of them will (or should) be in place.

In this chapter, we will create a D3-based charting package. We will use tools to check code conventions, test, and build distributable files for our charting package. Even if the tools that we will use have proven to be successful and are widely used in frontend projects, you may prefer to use different tools for some of the tasks. Feel free to explore and discover a toolset more appropriate for your workflow.

The development workflow

In this section, we will provide an overview of the workflow to create and distribute our charting package. We will also discuss some conventions regarding version numbers and the process of creating a release, introduce tools that will help us to manage the dependences with other projects, run the tests, and automate the package building process.

Writing the code

We begin by creating the project directory and the initial package content. During the development of our charting package, we need to perform the following actions:

- Implement new features or modify the existing code
- Check whether our code follows the coding guidelines
- Implement tests for the new functionality or create additional tests for the existing features
- Run tests to ensure that the modifications don't introduce unexpected behaviors or break the public API
- Concatenate the source files to generate a single JavaScript file that contains the charting package
- Generate a minified file

When implementing new features or fixing bugs, we will modify the code, check for errors, and run the tests. We will repeat the modify-check-test cycle several times until we finish implementing the feature or fix the bug. At this point, we will check and test the code again, build the package, and commit our changes.

As mentioned before, there are many tools that will help us to automate these tasks. There are a great number of tools to make the frontend workflow easier, and every developer has his or her preferences and opinions in this regard. In this chapter, we will use Node.js modules to orchestrate our development tasks. We will use the following tools:

- **Vows**: This is an asynchronous, behavior-driven JavaScript testing framework for Node.js. We will use Vows to test our charting package.
- **Grunt**: This is a task runner for Node.js. We will use Grunt and some plugins to check the source files, concatenate, minify, and test our package.
- **Bower**: This is a frontend package management system. We will configure our package such that users can install our package and its dependencies (D3) easily.

Creating a release

Depending on our development workflow, we may want to create a **release**. A release is a state of our package that we distribute for it to be used. It's usually identified with a version number that indicates how much it has changed from the previous versions.

Semantic Versioning

The version number is especially important in systems with many dependencies. Depending too much on the functionality provided by a specific version of a package can lead to a version lock, that is, the inability to update the package without having to release new versions of our own package. On the other hand, if we update the package assuming that it's compatible with our software, we will eventually find that it is not the case and that the package has made changes to the API that are not compatible with our software.

Semantic Versioning is a useful convention that helps you to know if it's safe to update a package, as long as it follows the Semantic Versioning convention. The complete specification of Semantic Versioning 2.0.0 (yes, the specification itself is versioned) is available at <http://semver.org/>. The key points of the Semantic Versioning convention are as follows.

Each release should be assigned a version number of the form **MAJOR.MINOR.PATCH**, with optional identifiers after a dash (1.0.0-beta, for instance). The version numbers are integers (without leading zeros) and should be incremented when we create a new release by the following rules:

- **MAJOR**: This version is used when you make backward-incompatible changes to the API
- **MINOR**: This version is used when a new functionality is added without breaking the API
- **PATCH**: This version is used for improvements and bug fixes that don't change the public API

When we increment the MAJOR version, MINOR and PATCH are set to zero; increments in MINOR will reset the PATCH number to zero. The content of a release must not be modified; any modification should be released as a new version.

If we follow this convention, users will know that upgrading from 2.1.34 to 2.1.38 is safe and upgrading from 2.1.38 to 2.2.0 is also safe (and it may provide additional backward-compatible features), while upgrading from 2.2.0 to 3.0.1 will require you to check whether the changes in the new version are still compatible with the existing code. In the next sections, we will create the initial content of the package and configure the tools to test and build our package.

Creating the package contents

We will create a small package containing a heat map chart and a layout function. We begin by choosing a name for our project, creating an empty directory, and initializing the repository. The name of our package will be `Windmill`. Once we have created the directory, we can create the initial content. We will organize the code in components and implement the chart and the helper functions in separate files. The source code will be organized in folders, one for each component. Later, we will concatenate the files in the correct order to generate the `windmill.js` file, containing all the components of the package:

```
src/
  chart/
    chart.js
    heatmap.js
  layout/
    layout.js
    matrix.js
  svg/
    svg.js
```

```
transform.js
start.js
end.js
```

We add the `version` attribute and indicate that we will follow the Semantic Version specification:

```
!function() {
  var windmill = {version: '0.1.0'}; // semver
  // Charts
  windmill.chart = {};
  windmill.chart.heatmap = function() {...};
  // Other Components...
}();
```

We can load this file in the browser or using Node.js. In either case, the anonymous function will be invoked, but nothing more will happen. In order to expose the package functionality in Node, we need to load D3 as a Node package in the context of the module and export the contents of the `windmill` object. This will allow other Node modules (our tests, for instance) to load our package as a Node module. If the file is loaded in the browser, we assume that D3 is available and add the `windmill` attribute to the global object and set its value to the package contents. Note that when we run the anonymous function in the global scope, the `this` context is set to the global object:

```
!function() {
  var windmill = {version: '0.1.0'}; // semver

  // Charts
  windmill.chart = {};
  windmill.chart.heatMap = function() {...};

  // Other Components...

  // Expose the package components
  if (typeof module === 'object' && module.exports) {
    // The package is loaded as a node module
    this.d3 = require('d3');
    module.exports = windmill;
  } else {
    // The file is loaded in the browser.
    this.windmill = windmill;
  }
}();
```

To generate this file, we need to concatenate the source files in order. The `src/start.js` file will contain the beginning of the function and the package version, which must be updated in each release:

```
!function() {  
    var windmill = {version: '0.1.0'}; // semver
```

Each component will add an attribute to the `windmill` variable. For instance, the `src/chart/chart.js` file will contain the following attribute:

```
// Charts  
windmill.chart = {};
```

The `src/chart/heatmap.js` file will add a function to the `chart` attribute:

```
windmill.chart.heatMap = function() {...};
```

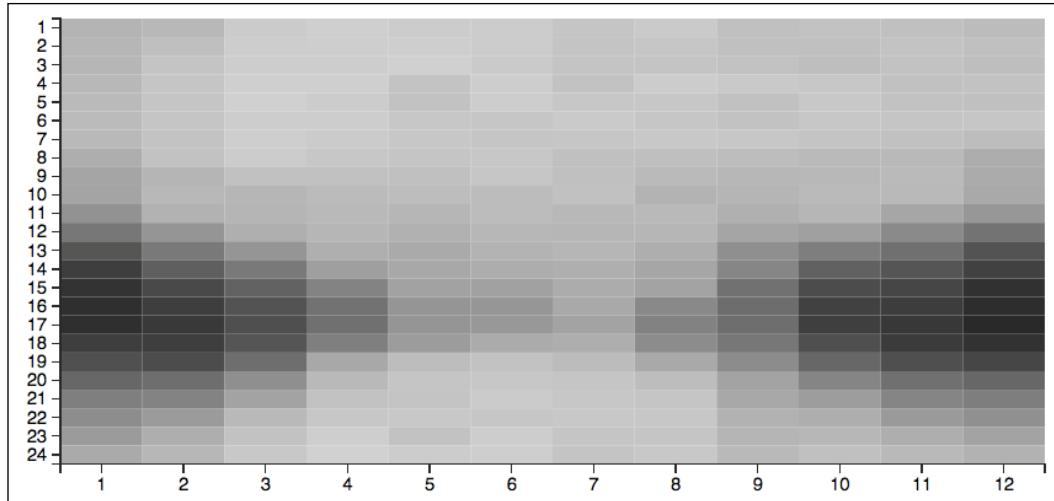
The `matrix` layout should be included in the same way. The `src/end.js` file will contain the last part of the consolidated file. If the file is loaded as a Node.js module, the `module.exports` variable will be defined; we import the D3 library and export the `windmill` package. If the file is loaded in the browser, we assign the `windmill` object to the `window` object, making it available as a global variable:

```
// Expose the package components  
if (typeof module === 'object' && module.exports) {  
    // The package is loaded as a node module  
    this.d3 = require('d3');  
    module.exports = windmill;  
} else {  
    // The file is loaded in the browser.  
    window.windmill = windmill;  
}  
}();
```

The heat map chart

A heat map is a chart that aims to represent the dependency of one quantitative variable as a function of two ordinal variables.

A heat map is a chart that allows you to visualize the dependency between a variable and two other variables. It resembles a matrix where each cell's color is proportional to the value of the main variable, and the rows and columns represent the other variables. Refer to the following screenshot:



A heat map showing the average wind speeds by hour and month

Heat maps are useful to detect patterns and the dependency between the target variable in the function of the variables represented by the rows and columns. We will put the code for the heat map chart in the `src/chart/heatmap.js` file. We will implement the heat map chart as a reusable chart, but this time, we will generate the accessor methods automatically to avoid writing similar code for each chart attribute. We begin by adding the `heatmap` function as an attribute of the `windmill.chart` object:

```
// HeatMap Chart
windmill.chart.heatmap = function() {
    'use strict';

    function chart(selection) {
        // ...
    }

    return chart;
};
```

Heatmap function usually receives a matrix as data input while heat maps usually receive a matrix as input data. We can represent a matrix as a nested array or use a columnar representation of a matrix, where each cell is represented as an item in a list. Each item has a row, column, and value, indicating the content of each cell in the matrix (this representation is especially useful for sparse matrices). The default input data will be an array of objects with at least three attributes: rows, columns, and values:

```
var data = [
  {row: 1, column: 1, value: 5.5},
  {row: 1, column: 2, value: 2.5},
  // more items...
  {row: 6, column: 4, value: 7.5}
];
```

We will add configurable accessor functions so that the user can configure which attributes will be the rows, columns, and values. As we will generate the accessor methods automatically, we will hold the chart properties in the `attributes` object. This element will be used to generate the accessor methods:

```
// HeatMap Chart
windmill.chart.heatmap = function() {
  'use strict';

  // Default Attribute Container
  var attributes = {
    width: 600,
    height: 300,
    margin: {top: 20, right: 20, bottom: 40, left: 40},
    colorExtent: ['#000', '#aaa'],
    value: function(d) { return d.value; },
    row: function(d) { return d.row; },
    column: function(d) { return d.column; }
  };

  // Charting function...

  return chart;
}
```

We set the default values for the width, height, margin, and color extent. We also define default accessor functions for the rows, columns, and values.

We will generate accessor methods for each attribute. We want the generated accessors to have the same behavior as that of the accessors written explicitly. Also, we want to be able to overwrite accessors if we need to include more logic than to simply get or set a value. Until now, we have written the accessor method for the `width` attribute as follows:

```
chart.width = function(w) {
    if (!arguments.length) { return width; }
    width = w;
    return chart;
};
```

If no arguments are passed, the `chart.width` method will return the current value of the `width` variable. If we pass a value, the `width` value is updated and we return the chart to allow method chaining. Since our `attributes` object holds the chart properties, we should update this method:

```
chart['width'] = function(val) {
    if (!arguments.length) { return attributes['width']; }
    attributes['width'] = val;
    return chart;
};
```

Here, we avoided hardcoding the `width` attribute. This will work, but we still have to write a function that receives a value and modifies the `width` attribute. We will create a function that returns an accessor function for a specific attribute:

```
// Create an accessor function for the given attribute
function createAccessor(attr) {
    // Accessor function
    function accessor(value) {
        if (!arguments.length) { return attributes[attr]; }
        attributes[attr] = value;
        return chart;
    }
    return accessor;
}
```

We can now assign an accessor function for the `width` attribute using the following code:

```
// Set the accessor function for the width
chart['width'] = createAccessor('width');
```

This is still not good enough; we should iterate the properties of the `attributes` object and create one accessor method for each property. First, we should check whether the accessor already exists to avoid overwriting it, and we should verify that the properties are of the `attributes` object, and not from the higher accessor in the prototype chain:

```
// Create accessors for each element in attributes
for (var attr in attributes) {
    if (!chart[attr] && (attributes.hasOwnProperty(attr))) {
        chart[attr] = createAccessor(attr);
    }
}
```

This will generate an accessor for each property in the `attributes` object. Note that the accessors will just get and set the attributes; there are no validations or logic besides assigning or returning the values. If we need a more complex accessor for an attribute, we can add it and it won't be overwritten.

The charting function will select the `div` container and create an `svg` element to contain the chart. As we have done before, we will encapsulate the initialization of the `svg` element in the `chart.svgInit` method:

```
// Charting function
function chart(selection) {
    selection.each(function(data) {
        // Initialize the SVG element on enter
        var div = d3.select(this),
            svg = div.selectAll('svg').data([data])
                .enter().append('svg')
                .call(chart.svgInit);
    });
}
```

In the `chart.svgInit` method, we set the dimensions of the `svg` element and create groups for the chart and the horizontal and vertical axes:

```
// Initialize the SVG Element
chart.svgInit = function(svg) {

    // Compute the width and height of the charting area
    var margin = chart.margin(),
        width = chart.width() - margin.left - margin.right,
        height = chart.height() - margin.top - margin.bottom,
        translate = windmill.svg.translate;

    // Set the size of the svg element
```

```
svg
    .attr('width', chart.width())
    .attr('height', chart.height());

    // Chart Container
    svg.append('g')
        .attr('class', 'chart')
        .attr('transform', translate(margin.left, margin.top));

    // X Axis Container
    svg.append('g')
        .attr('class', 'axis xaxis')
        .attr('transform', translate(margin.left, margin.top +
height));

    // Y Axis Container
    svg.append('g')
        .attr('class', 'axis yaxis')
        .attr('transform', translate(margin.left, margin.top));
};
```

Here, we used the accessor method generated previously to access the width, height, and margin. We can also access these through the attributes object in the charting code, but the attribute won't be accessible for code using the chart object. In the charting function, we compute the width and height of the charting area and create shortcuts for the row, column, and value accessors. Without these shortcuts, we would need to invoke the row function either as attributes.row(d) or chart.row()(d):

```
// Compute the width and height of the chart area
var margin = chart.margin(),
    width = chart.width() - margin.left - margin.right,
    height = chart.height() - margin.top - margin.bottom;

// Retrieve the accessor functions
var row = chart.row(),
    col = chart.column(),
    val = chart.value();
```

We can create the scales for the position and color of the rectangles. We will use ordinal scales and use the `rangeBands` range. This option allows you to divide an interval into n evenly spaced bands, where n is the number of unique elements in the domain:

```
// Horizontal Position
var xScale = d3.scale.ordinal()
    .domain(data.map(col))
    .rangeBands([0, width]); 

// Vertical Position
var yScale = d3.scale.ordinal()
    .domain(data.map(row))
    .rangeBands([0, height]); 

// Color Scale
var cScale = d3.scale.linear()
    .domain(d3.extent(data, val))
    .range(chart.colorExtent());
```

We can create the rectangles on enter as follows:

```
// Create the heatmap rectangles on enter
var rect = gchart.selectAll('rect').data(data)
    .enter().append('rect');
```

We can set the width, height, and position of the rectangles, and set the fill color using the aforementioned scales and accessor functions for the rows, columns, and values. The width and height are set using the width of a band, which is computed by the scale:

```
// Set the attributes of the rectangles
rect.attr('width', xScale.rangeBand())
    .attr('height', yScale.rangeBand())
    .attr('x', function(d) { return xScale(col(d)); })
    .attr('y', function(d) { return yScale(row(d)); })
    .attr('fill', function(d) { return cScale(val(d)); });
```

Finally, we add the axes for the horizontal and vertical axes:

```
// Create the Horizontal Axis
var xAxis = d3.svg.axis()
    .scale(xScale)
    .orient('bottom');
```

```
svg.select('g.xaxis').call(xAxis);

// Create the Vertical Axis
var yAxis = d3.svg.axis()
    .scale(yScale)
    .orient('left');
svg.select('g.yaxis').call(yAxis);
```

We will create an example file for the heat map. In the `examples/heatmap.html` file, we create a container div with the `chart01` ID:

```
<div id="chart01"></div>
```

We will generate a sample data array and create a chart instance, configuring the width, height, and color extent:

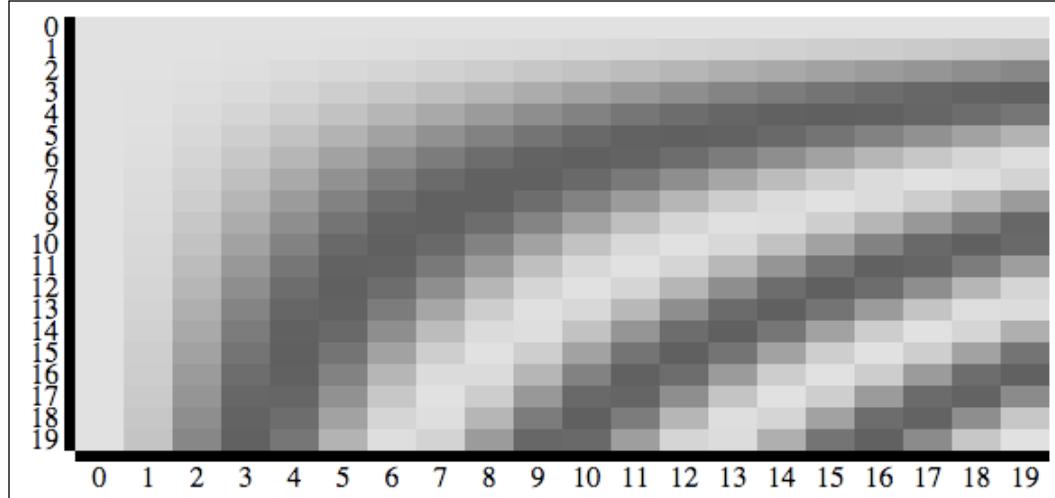
```
// Generate a sample data array
var data = [];
for (var k = 0; k < 20; k += 1) {
    for (var j = 0; j < 20; j += 1) {
        data.push({
            row: k,
            column: j,
            value: Math.cos(Math.PI * k * j / 60)
        });
    }
}

// Create and configure the heatmap chart
var heatmap = windmill.chart.heatmap()
    .width(600)
    .height(300)
    .colorExtent(['#555', '#ddd']);
```

Here, the lower values will be **dark gray**, and the higher values will be **blue**. The matrix data contains values for two rows and four columns. We have given a value for each cell; if there were missing items, the cells would just not be drawn. We select the container element, bind the data array, and invoke the heat map using the `selection.call` method:

```
// Create the heatmap chart in the container selection
d3.select('div#chart01').data([data])
    .call(heatmap);
```

The generated heat map will have 400 cells; the color of each cell represents the value's magnitude corresponding to each row and column. Refer to the following screenshot:

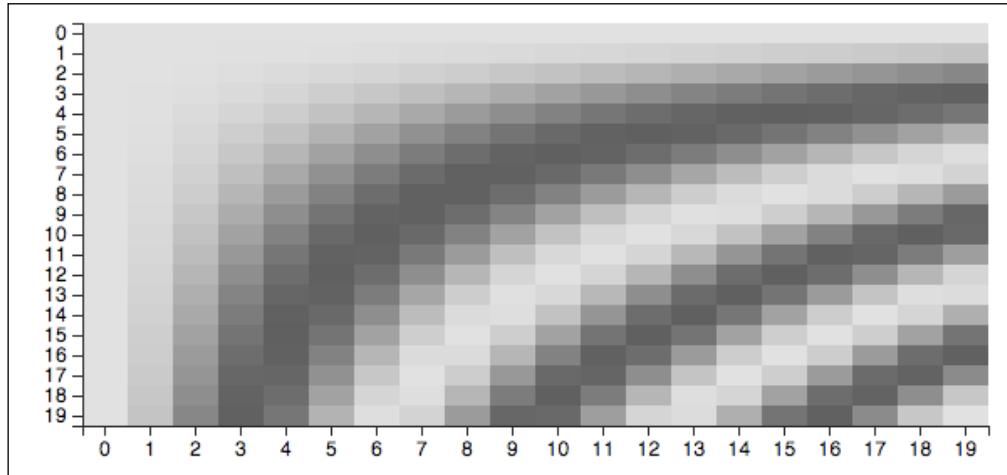


A heat map with the default styles

We will create and include a style sheet file to get a better-looking axis. As the users will need the CSS files to use the chart, we should make the styles available to the users as well. A user could also modify the styles to accommodate the appearance of the chart to the client application. We will add the `windmill.css` file to the `css` directory and add the following content:

```
/* Axis lines */
.axis path, line {
    fill: none;
    stroke: #222222;
    shape-rendering: crispEdges;
}
/* Style for the xaxis */
.xaxis {
    font-size: 12px;
    font-family: sans-serif;
}
/* Style for the yaxis */
.yaxis {
    font-size: 12px;
    font-family: sans-serif
}
```

With these styles, the heat map looks better, which can be seen as follows:



The heat map with improved styles

The matrix layout

In the heat map chart, we assumed that we have unique combinations of the rows and columns in our dataset; having more than one element with the same row and column would cause two overlapping rectangles.

We will create a layout to aggregate the values with the same rows and columns using a configurable aggregation function. For instance, we could have the following dataset:

```
var data = [
  {row: 1, column: 2, value: 5},
  {row: 1, column: 2, value: 4},
  {row: 1, column: 2, value: 9},
  // ...
];
```

The matrix layout will allow us to group the values and calculate a single aggregated value, which is usually the sum, count, average, minimum, or maximum value. We will create the matrix layout in the `src/layout/matrix.js` file. As in the heat map chart, we begin by adding the `matrix` property to the `layout` object. We will use the reusable chart pattern, except that the charting function is replaced with the `layout` function, which receives the input data and returns the aggregated array:

```
windmill.layout.matrix = function() {
  'use strict';

  function layout(data) {
```

```
//...
    return groupedData;
}
return layout;
};
```

We will use the technique presented in the heat map chart to automatically generate accessor methods. We add the default accessor methods as follows:

```
// Default Accessors
var attributes = {
  row: function(d) { return d.row; },
  column: function(d) { return d.column; },
  value: function(d) { return d.value; },
  aggregate: function(values) {
    var sum = 0;
    values.forEach(function(d) { sum += d; });
    return sum;
  }
},
```

The `row`, `column`, and `value` accessor functions compute the row, column, and value for each input data element. The `aggregate` function receives an array of values and computes a single aggregated value. The default `aggregate` function will return the sum of the elements in the input array:

```
// Layout function
function layout(data) {
  // Output data array
  var groupedData = [];

  // Group and aggregate the input values...

  return groupedData;
}
```

We begin by grouping the values for each combination of row and column. We iterate through the input array and compute the item's row, column, and value. The `groupedData` array will contain elements with the `row`, `col`, and `values` attributes. If the `groupedData` array contains an element with the same row and column as the current item, we append the value of the item to the `values` array; if not, we append a new element to the `groupedData` array:

```
// Group by row and column
data.forEach(function(d) {

  // Compute the row, column, and value
  row = attributes.row(d);
```

```
col = attributes.column(d);
val = attributes.value(d);

// Search corresponding items in groupedData
found = false;

groupedData.forEach(function(item, idx) {
    if ((item.row === row) && (item.col === col)) {
        groupedData[idx].values.push(val);
        item.values.push(val);
        found = true;
    }
});

// Append the item, if not found
if (!found) {
    groupedData.push({
        row: row,
        col: col,
        values: [val]
    });
}
});
```

We can now aggregate the values using the `aggregate` function. This function receives an array and returns a single value. Finally, we remove the `values` attribute:

```
// Aggregate the values
groupedData.forEach(function(d) {
    // Compute the aggregated value
    d.value = attributes.aggregate(d.values);
    delete d.values;
});
```

We generate accessor methods for the layout automatically, with code that is similar to the code in the heat map chart:

```
// Create accessor functions
function createAccessor(attr) {
    function accessor(value) {
        if (!arguments.length) { return attributes[attr]; }
        attributes[attr] = value;
        return layout;
    }
    return accessor;
```

```
}

// Generate automatic accessors for each attribute
for (var attr in attributes) {
    if (!layout[attr] && (attributes.hasOwnProperty(attr))) {
        layout[attr] = createAccessor(attr);
    }
}
```

We will create an example for the `matrix` layout. In the `examples/layout.html` file, we declare a data array with sample data as follows:

```
// Sample data array
var data = [
    {a: 1, b: 1, c: 10},
    {a: 1, b: 1, c: 5},
    // ...
    {a: 2, b: 2, c: 5}
];
```

Note that for each combination of row and column, we have more than one element in the array. We define the average function, computing the sum of all the elements in the input array and dividing the sum by the array's length:

```
// Define the aggregation function (average of values)
var average = function(values) {
    var sum = 0;
    values.forEach(function(d) { sum += d; });
    return sum / values.length;
}
```

We create and configure a `matrix` layout instance, setting the `row` accessor to be a function that returns the `a` attribute of each element, the `column` function will return the `b` attribute, and the value will be the `c` property. We set the `aggregate` function to use our `average` function:

```
// Create and configure a matrix layout instance
var matrix = windmill.layout.matrix()
    .row(function(d) { return d.a; })
    .column(function(d) { return d.b; })
    .value(function(d) { return d.c; })
    .aggregate(average);
```

We invoke the layout using the sample data array, obtaining an array that groups the values by row and column:

```
var grouped = matrix(data);
```

The value of each element will be the average value for the items with the same combination of row and column:

```
// Output values
grouped = [
    {col: 1, row: 1, value: 7.5},
    // ...
    {col: 2, row: 2, value: 10}
];
```

The `matrix` layout allows us to group and aggregate values, making it easier to format data for the heat map chart. Now that we have the initial content, we will configure the tools to build and distribute the package.

The project setup

In this section, we will install and configure the tools that we will use to build our package. We will assume that you know how to use the command line and you have Node installed on your system. You can install Node by either following the instructions from Node.js's website (<http://nodejs.org/download/>) or using a package manager in Unix-like systems.

Installing the Node modules

The **Node Package Manager (npm)** is a program that helps us to manage dependencies between Node projects. As our project could be a dependency of other projects, we need to provide information about our package to **npm**. The package `.json` file is a JSON file that should contain at least the project name, version, and dependencies for use and development. For now, we will add just the name and version:

```
{
  "name": "windmill",
  "version": "0.1.0",
  "dependencies": {},
  "devDependencies": {}
}
```

We will install Grunt, Vows, Bower, and D3 using npm. When installing a package, we can pass an option to save the package that we are installing as a dependency. With the `--save-dev` option, we can specify the development dependencies:

```
$ npm install --save-dev grunt vows bower
```

D3 will be a dependency for our package. If someone needs to use our package, D3 will be needed; the previous packages will be necessary only for development. To install the project dependencies, we can use the `--save` option:

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

A directory named `node_modules` will be created at the topmost level of the project. This directory will contain the installed modules:

```
node_modules/
  bower/
  d3/
  grunt/
  vows/
```

The package `.json` file will be updated with the dependencies as well:

```
{
  "name": "windmill",
  "version": "0.1.0",
  "dependencies": {
    "d3": "~3.4.1"
  },
  "devDependencies": {
    "grunt": "~0.4.2",
    "vows": "~0.7.0"
  }
}
```

Note that the dependencies specify the version of each package. Node.js packages should follow the Semantic Versioning specification. We will include additional modules to perform the building tasks, but we will cover that later.

Building with Grunt

Grunt is a task runner for Node.js. It allows you to define tasks and execute them easily. To use Grunt, we need to have a `package.json` file with the project information and the `Gruntfile.js` file, where we will define and configure our tasks. The `Gruntfile.js` file should have the following structure; all the Grunt tasks and configurations should be in the exported function:

```
module.exports = function(grunt) {
  // Grunt initialization and tasks
};
```

The Grunt tasks may need configuration data, which is usually passed to the `grunt.initConfig` method. Here, we import the package configuration from the `package.json` file. This allows you to use the package configuration values in order to generate banners in target files or to display information in the console when we run tasks:

```
module.exports = function(grunt) {
    // Initialize the Grunt configuration
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json')
    });
};
```

There are hundreds of Grunt plugins to automate every development task with minimal effort. A complete list of the Grunt plugins is available at <http://gruntjs.com/plugins>.

Concatenating our source files

The `grunt-contrib-concat` plugin will concatenate our source files for us. We can install the plugin as any other Node.js module:

```
$ npm install --save-dev grunt-contrib-concat
```

To use the plugin, we should enable it and add its configuration as follows:

```
module.exports = function(grunt) {
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),

        concat: {
            // grunt-contrib-concat configuration...
        }
    });

    // Enable the Grunt plugins
    grunt.loadNpmTasks('grunt-contrib-concat');
};
```

We add the `grunt-contrib-concat` configuration. The `concat` object can contain one or more targets, each containing an array of sources and a destination path for the concatenated file. The files in `src` will be concatenated in order:

```
// Initialize the Grunt configuration
grunt.initConfig({

    // Import the package configuration
```

```
pkg: grunt.file.readJSON('package.json'),  
  
    // Configure the concat task  
    concat: {  
        js: {  
            src: [  
                'src/start.js',  
                'src/svg/svg.js',  
                'src/svg/transform.js',  
                'src/chart/chart.js',  
                'src/chart/heatmap.js',  
                'src/layout/layout.js',  
                'src/layout/matrix.js',  
                'src/end.js'  
            ],  
            dest: 'windmill.js'  
        }  
    },  
});
```

There are options to add a banner too, which can be useful to add a comment indicating the package name and version. We can run the concat task from the command line as follows:

```
$ grunt concat  
Running "concat:js" (concat) task  
File "windmill.js" created.  
Done, without errors.
```

If we have several targets, we can build them individually by passing the target after the concat option:

```
$ grunt concat:js  
Running "concat:js" (concat) task  
File "windmill.js" created.  
Done, without errors.
```

The `windmill.js` file contains the sources of our package concatenated in order, preserving the original spaces and comments.

Minifying the library

It's common practice to distribute two versions of the library: one version with the original format and comments for debugging and another minified version for production. To create the minified version, we will need the `grunt-contrib-uglify` plugin. As we did with the `grunt-contrib-concat` package, we need to install this and enable it in the `Gruntfile.js` file:

```
module.exports = function(grunt) {
    // ...

    // Enable the Grunt plugins
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
};
```

We need to add the `uglify` configuration to the `grunt.initConfig` method as well. In `uglify`, we can have more than one target. The `options` attribute allows us to define the behavior of `uglify`. In this case, we set `mangle` to `false` in order to keep the names of our variables as they are in the original code. If the `mangle` option is set to `true`, the variable names will be replaced with shorter names, as follows:

```
// Uglify Configuration
uglify: {
    options: {
        mangle: false
    },
    js: {
        files: {
            'windmill.min.js': ['windmill.js']
        }
    }
}
```

We can run the minification task in the command line using the same syntax as in the concatenation task:

```
$ grunt uglify
Running "uglify:js" (uglify) task
File windmill.min.js created.
Done, without errors.
```

This will generate the `windmill.min.js` file, which is about half the size of the original version.

Checking our code with JSHint

In JavaScript, it is very easy to write code that doesn't behave as we expect. It could be a missing semicolon or forgetting to declare a variable in a certain scope. A **linter** is a program that helps us to detect potential errors and dangerous constructions by enforcing a series of coding conventions. Of course, a static code analysis tool can't detect these if your program is correct. **JSHint** is a tool that helps us to detect these potential problems by checking the JavaScript code against code conventions. The behavior of JSHint can be configured to match our coding conventions.



JSHint is a fork of **JSLint**, a tool created by Douglas Crockford to check code against a particular set of coding standards. His choices on coding style are explained in the book *JavaScript: The Good Parts*.

In JSHint, the code conventions can be set by writing a `jshintrc` file, a JSON file containing a series of flags that will define the JSHint behavior. For instance, a configuration file might contain the following flags:

```
{  
  "curly": true,  
  "eqeqeq": true,  
  "undef": true,  
  // ...  
}
```

The `curly` option will enforce the use of curly braces (`{` and `}`) around conditionals and loops, even if we have only one statement. The `eqeqeq` option enforces the use of `==` and `!=` to compare objects, instead of `==` and `!=`. If you don't have a set of coding conventions already, I would recommend that you read the list of JSHint options available at <http://www.jshint.com/docs/options/> and create a new `.jshintrc` file. Here, the options are listed and explained, so you can make an informed decision about which flags to enable.

Many editors have support for live linting, but even if the text editor checks the code as you write, it is a good practice to check the code before committing your changes. We will enable the `grunt-contrib-jshint` module and configure it so that we can check our code easily. We will enable the plugin as follows:

```
// Enable the Grunt plugins  
grunt.loadNpmTasks('grunt-contrib-concat');  
grunt.loadNpmTasks('grunt-contrib-uglify');  
grunt.loadNpmTasks('grunt-contrib-jshint');
```

Next, we configure the plugin. We will check the `Gruntfile.js` file, our tests, and the code of our chart:

```
jshint: {
  all: [
    'Gruntfile.js',
    'src/svg/*.js',
    'src/chart/*.js',
    'src/layout/*.js',
    'test/*.js',
    'test/**.js'
  ]
}
```

We can check our code using the following command lines:

```
$ grunt jshint
Running "jshint:all" (jshint) task
>> 11 files lint free.
Done, without errors.
```

Testing our package

Distributing a software package is a great responsibility. The users of our charting package rely on our code and assume that everything works as expected. Despite our best intentions, we may break the existing functionality when implementing a new feature or fixing a bug. The only way to minimize these errors is to extensively test our code.

The tests should be easy to write and run, allowing us to write the tests as we code new features and to run the tests before committing changes. There are several test suites for JavaScript code. In this section, we will use Vows, an asynchronous behavior-driven test suite for Node.js.

Writing a simple test

In Vows, the largest test unit is a **suite**. We will begin by creating a simple test using JavaScript without any libraries. In the `test` directory, we create the `universe-test.js` file.

We will load the `vows` and `assert` modules and assign them to the local variables:

```
// Load the modules
var vows = require('vows'),
  assert = require('assert');
```

We can create a suite now. The convention is to have one suite per file and to match the suite description with the filename. We create a suite by invoking the `vows.describe` method:

```
// Create the suite
var suite = vows.describe('Universe');
```

Tests are added to the suite in batches. A suite can have zero or more batches, which will be executed sequentially. The batches are added using the `suite.addBatch` method. Batches allow you to perform tests in a given order:

```
suite.addBatch({
  //...
});
```

A batch, in turn, contains zero or more **contexts**, which describe the behaviors or states that we want to test. Contexts are run in parallel, and they are asynchronous; the order in which they will be completed can't be predicted. We will add a context to our batch, as follows:

```
suite.addBatch({
  'the answer': {
    //...
  }
});
```

A context contains a topic. The topic is a value or function that returns an element to be tested. The vows are the actual tests. The vows are the functions that make assertions about the topic. We will add a topic to our context as follows:

```
suite.addBatch({
  'the answer': {
    topic: 42,
    //...
  }
});
```

In this case, all our vows in the context `the answer` will receive the value 42 as the argument. We will add some vows to assert whether the topic is `undefined`, `null`, or a number, and finally, whether the topic is equal to 42. Refer to the following code:

```
suite.addBatch({
  'the answer': {
    topic: 42,
    "shouldn't be undefined": function(topic) {
      assert.notEqual(topic, undefined);
    },
    "is 42": function(topic) {
      assert.equal(topic, 42);
    }
});
```

```

        "shouldn't be null": function(topic) {
            assert.notEqual(topic, null);
        },
        "should be a number": function(topic) {
            assert.isNumber(topic);
        },
        "should be 42": function(topic) {
            assert.equal(topic, 42);
        }
    }
});

```

To execute all the tests in the `test` directory as a single entity (instead of having to run each one separately), we need to export the suite:

```
suite.export(module);
```

We can run these tests individually by passing the `test` path as the argument:

```
$ vows test/universe-test.js --spec
◊ Universe

    the answer to the Universe
✓ shouldn't be undefined
✓ shouldn't be null
✓ should be a number
✓ should be 42

✓ OK » 4 honored (0.007s)
```

We will temporarily modify our topic to introduce an error as follows:

```
suite.addBatch({
    'the answer': {
        topic: 43,
        //...
    }
});
```

The output of the test will show which vows were honored and which failed, displaying additional details for the broken vows. In this case, three vows where honored and one was broken.

```
◊ Universe

    the answer
✓ shouldn't be undefined
```

```
✓ shouldn't be null
✓ should be a number
✖ should be 42
  » expected 42,
    got    43 (==) // universe-test.js:27

✖ Broken » 3 honored • 1 broken (0.564s)
```

This simple example shows you how to create a suite, context, topics, and vows to test a simple feature. We will use the same structure to test our heat map chart.

Testing the heat map chart

The tests for the heat map chart will be more involved than the test from the previous example; for one thing, we need to load D3 and the `windmill` library as Node modules.

D3 is a library that can be used to modify DOM elements based on data. In node applications, we don't have a browser and the DOM doesn't exist. To have a document with a DOM tree, we can use the JSDOM module. When we load D3 as a module, it creates the document and includes JSDOM for us; we don't need to load JSDOM (or create the `document` and `window` objects).

To create a test for the heat map chart, we create the `test/chart/heatmap-test.js` file and load the `vows`, `assert`, and `d3` modules. We also load our charting library as a local file:

```
// Import the required modules
var vows = require("vows"),
    assert = require("assert"),
    d3 = require("d3"),
    windmill = require("../..//windmill");
```

We will also add a data array and use it later to create the charts. This array will be accessible for the `vows` and contexts in the module, but it won't be exported.

```
// Sample Data Array
var data = [
  {row: 1, column: 1, value: 5.5},
  {row: 1, column: 2, value: 2.5},
  // ...
  {row: 2, column: 4, value: 7.5}
];
```

The suite will contain tests for the heat map chart. We will describe the suite. It is not necessary to describe the suite with the path to the method being tested, but it's a good practice and helps you to locate errors when the tests don't pass.

```
// Create a Test Suite for the heatmap chart
var suite = vows.describe("windmill.chart.heatmap");
```

We will add a batch that contains the contexts to be tested. In the first context topic, we will create a div element, create a chart with the default options, bind the data array with the div element, and create a chart in the first context topic:

```
// Append the Batches
suite.addBatch({
  "the default chart svg": {
    topic: function() {

      // Create the chart instance and a sample data array
      var chart = windmill.chart.heatmap();

      // Invoke the chart passing the container div
      d3.select("body").append("div")
        .attr("id", "default")
        .data([data])
        .call(chart);

      // Return the svg element for testing
      return d3.select("div#default").select("svg");
    },
    // Vows...
  }
});
```

We will create vows to assert whether the svg element exists, its width and height match the default values, it contains groups for the chart and axis, and the number of rectangles match the number of elements in the data array:

```
// Append the Batches
suite.addBatch({
  "the default chart svg": {
    topic: function() {...},
    "exists": function(svg) {
      assert.equal(svg.empty(), false);
    },
    "is 600px wide": function(svg) {
```

```
        assert.equal(svg.attr('width'), '600') ;
    },
"is 300px high": function(svg) {
    assert.equal(svg.attr('height'), '300') ;
},
"has a group for the chart": function(svg) {
    assert.equal(svg.select("g.chart").empty(), false) ;
},
"has a group for the xaxis": function(svg) {
    assert.equal(svg.select("g.xaxis").empty(), false) ;
},
"has a group for the yaxis": function(svg) {
    assert.equal(svg.select("g.yaxis").empty(), false) ;
},
"the group has one rectangle for each data item":
function(svg) {
    var rect = svg.select('g').selectAll("rect") ;
    assert.equal(rect[0].length, data.length) ;
}
});
});
```

We can run the test with `vows` and check whether the default attributes of the chart are correctly set and the structure of the inner elements is organized as it should be:

```
$ vows test/chart/heatmap-test.js --spec
◊ windmill.chart.heatmap

    the default chart svg
✓ exists
✓ is 600px wide
✓ is 300px high
✓ has a group for the chart
✓ has a group for the xaxis
✓ has a group for the yaxis
✓ the group has one rectangle for each data item

✓ OK » 7 honored (0.075s)
```

In a real-world application, we would have to add tests for many more configurations.

Testing the matrix layout

The matrix layout is simpler to test, because we don't need the DOM or even D3. We begin by importing the required modules and creating a suite, as follows:

```
// Create the test suite
var suite = vows.describe("windmill.layout.matrix");
```

We add a small data array to test the layout:

```
// Create a sample data array
var data = [
  {a: 1, b: 1, c: 10},
  // ...
  {a: 2, b: 2, c: 5}
];
```

We define an average function, as we did in the example file:

```
var average = function(values) {
  var sum = 0;
  values.forEach(function(d) { sum += d; });
  return sum / values.length;
};
```

We add a batch and a context to check the default layout attributes and generate the layout in the context's topic:

```
// Add a batch to test the default layout
suite.addBatch({
  "default layout": {
    topic: function() {
      return windmill.layout.matrix();
    }
  }
});
```

We add vows to test whether the layout is a function and has the `row`, `column`, and `value` methods:

```
"is a function": function(topic) {
  assertisFunction(topic);
},
"has a row method": function(topic) {
  assertisFunction(topic.row);
},
"has a column method": function(topic) {
  assertisFunction(topic.column);
},
```

```
        "has a value method": function(topic) {
            assertisFunction(topic.value);
        }
    );
});
```

We can run the tests using `vows test/layout/matrix-test.js --spec`, but we will automate the task of running the tests with Grunt.

Running the tests with Grunt

We will add a test task to the `Gruntfile.js` file in order to automate the execution of tests. We will need to install the `grunt-vows` module:

```
$ npm install --save-dev grunt-vows
```

As usual, we need to enable the `grunt-vows` plugin in the `Gruntfile.js` file:

```
// Enable the Grunt plugins
grunt.loadNpmTasks('grunt-contrib-concat');
// ...
grunt.loadNpmTasks("grunt-vows");
```

We will configure the task to run all the tests in the `test` directory. As we have done when running the tests, we will add the `spec` option to obtain detailed reporting. Removing this option will use the default value, displaying each test as a point in the console:

```
vows: {
    all: {
        options: {reporter: 'spec'},
        src: ['test/*.js', 'test/**/*.js']
    }
},
```

We could create additional targets to test the components individually as we modify them. We can now run the task from the command line:

```
$ grunt vows
Running "vows:all" (vows) task
(additional output not shown)
Done, without errors.
```

Testing the code doesn't guarantee that you will have bug-free code, but it will certainly help you to detect unexpected behaviors. A mature software usually has thousands of tests. At the time of writing, for instance, D3 has about 2,500 tests.

Registering the sequences of tasks

We have created and configured the essential tasks for our project, but we can automate the process further. For instance, while modifying the code, we need to check and test the code, but we won't need a minified version until we are ready to push the changes to the repository. We will register two groups of tasks, `test` and `build`. The `test` task will check the code, concatenate the source files, and run the tests. To register a task, we give the task a name and add a list of the subtasks to be executed:

```
// Test Task
grunt.registerTask('test', ['jshint', 'concat', 'vows']);
```

We can execute the `test` task in the command line, triggering the `jshint`, `concat`, and `vows` tasks in a sequence:

```
$ grunt test
```

We register the `build` task in a similar way; this task will run `jshint`, `concat`, `vows`, and `uglify` in order, generating the files that we want to distribute:

```
// Generate distributable files
grunt.registerTask('build', ['jshint', 'vows', 'concat',
'uglify']);
```

A default task can be added too. We will add a default task that just runs the `build` task:

```
// Default task
grunt.registerTask('default', ['build']);
```

To run the default task, we invoke Grunt without arguments. There are hundreds of Grunt plugins available; they can be used to automate almost everything. There are plugins to optimize images, copy files, compile the LESS or SASS files to CSS, minify the CSS files, monitor files for changes, and run tasks automatically, among many others. We could automate additional tasks, such as updating the version number in the source files or running tasks automatically when we modify source files.

The objective of automating tasks is not only to save time, but it also makes it easier to actually do the tasks and establish a uniform workflow among peers. It also allows developers to focus on writing code and makes the development process more enjoyable.

Managing the frontend dependencies

If our package is to be used in web applications, we should declare that it depends on D3 and also specify the version of D3 that we need. For many years, projects just declared their dependencies on their web page, leaving the task of downloading and installing the dependencies to the user. Bower is a package manager for web applications (<http://bower.io/>). It makes the process of installing and updating packages easier. Bower is a Node module; it can be installed either locally using npm, as we did earlier in the chapter, or globally using `npm install -g bower`:

```
$ npm install --save bower
```

This will install Bower in the `node_modules` directory. We need to create a `bower.json` file containing the package metadata and dependencies. Bower can create this file for us:

```
$ bower init
```

This command will prompt us with questions about our package; we need to define the name, version, main file, and keywords, among other fields. The generated file will contain essential package information, as follows:

```
{
  "name": "windmill",
  "version": "0.1.0",
  "authors": [
    "Pablo Navarro"
  ],
  "description": "Heatmap Charts",
  "main": "windmill.js",
  "keywords": ["chart", "heatmap", "d3"],
  "ignore": [
    "**/.*", "**/.**",
    "node_modules",
    "bower_components",
    "app/_bower_components",
    "test",
    "tests"
  ],
  "dependencies": {}
}
```

Bower has a registry of frontend packages; we can use Bower to search the registry and install our dependencies. For instance, we can search for the D3 package as follows:

```
$ bower search d3
Search results:
d3 git://github.com/mbostock/d3.git
nvd3 git://github.com/novus/nvd3
d3-plugins git://github.com/d3/d3-plugins.git
...
```

The results are displayed, showing the package name and its Git endpoint. We can use either the name or the endpoint to install D3:

```
$ bower install --save d3
```

This will create the `bower_components` directory (depending on your global configuration) and update the `bower.json` file, including the D3 library in its most recent release. Note that we included D3 both in our Node dependencies and in the Bower dependencies. We included D3 in the Node dependencies to be able to test our charts (which depend on D3); here, we include D3 as a frontend dependency, so the other packages that use our charting package can download and install D3 using Bower:

```
{
  "name": "windmill",
  "version": "0.1.0",
  ...
  "dependencies": {
    "d3": "~3.4.1"
  }
}
```

We can specify which version of the package we want to include. For instance, we could have installed the release 3.4.0:

```
$ bower install d3#3.4.0
```

We don't need to register the packages to install them with Bower; we can use Bower to install the unregistered packages using their Git endpoint:

```
$ bower install https://github.com/mbostock/d3.git
```

The Git endpoint could also be a local repository, or even a ZIP or TAR file:

```
$ bower install /path/to/package.zip
```

Bower will extract and copy each dependency in the `bower_components` directory. To use the packages, we can use a reference to the `bower_components` directory, or write a Grunt task to copy the files to another location. We will use the `bower_components` directory to create example pages for our charts.

Using the package in other projects

In this section, we will create a minimal web page that uses the windmill package. We begin by creating an empty directory, initializing the repository, and creating a `README.md` file. We create the `bower.json` file using `bower init`. We will install bootstrap to use it in our web page:

```
$ bower install --save bootstrap
```

This will download bootstrap and its dependencies to the `bower_components` directory. We will install the windmill library using the Git endpoint of the repository:

```
$ bower install --save https://github.com/pnavarrc/windmill.git
```

This will download the current version of windmill and the version of D3 on which windmill depends. The contents of the `bower_components` directory are as follows:

```
bower_components/
  bootstrap/
  d3/
  jquery/
  windmill/
```

In the index page, we will display the average wind speed in a certain city during 2013. We will store the data in the `wind.csv` file located in the `data` directory. The CSV file has three columns that display the data (MM/DD/YY), the hour of the measurement, and the average speed in meters per second:

```
Date,Hour,Speed
1/1/13,1,0.2554
1/1/13,2,0.1683
...
...
```

In the header of the index file, we include D3 and the windmill CSS and JavaScript files:

```
<link href="/bower_components/windmill/css/windmill.css"
rel="stylesheet">

<script src="/bower_components/d3/d3.min.js" charset="utf-8"></script>
<script src="/bower_components/windmill/windmill.min.js"></script>
```

In the body of the page, we add the title and a container div:

```
<div class="container">
  <h1>Wind Speed</h1>
  <div id="chart01"></div>
</div>
```

We create and configure the chart and layout. We want to display the average wind speed by month and hour of the day. We set the rows to be the hours, the columns to return the month number, and the value to return the speed. We will use the average function to aggregate values with the same hour and month:

```
// Aggregation function (average)
function average(values) {
  var sum = 0;
  values.forEach(function(d) { sum += d; });
  return sum / values.length;
}

// Matrix Layout
var matrix = windmill.layout.matrix()
  .row(function(d) { return +d.Hour; })
  .column(function(d) { return +d.Date.getMonth(); })
  .value(function(d) { return +d.Speed; })
  .aggregate(average);
```

We create and initialize the heat map chart, setting the width, height, and color extent of the chart:

```
// Create and configure the heatmap chart
var heatmap = windmill.chart.heatmap()
  .column(function(d) { return d.col; })
  .width(700)
  .height(350)
  .colorExtent(['#ccc', '#222']);
```

We load the data using `d3.csv` and parse the dates. We select the container div and bind the grouped data, as follows:

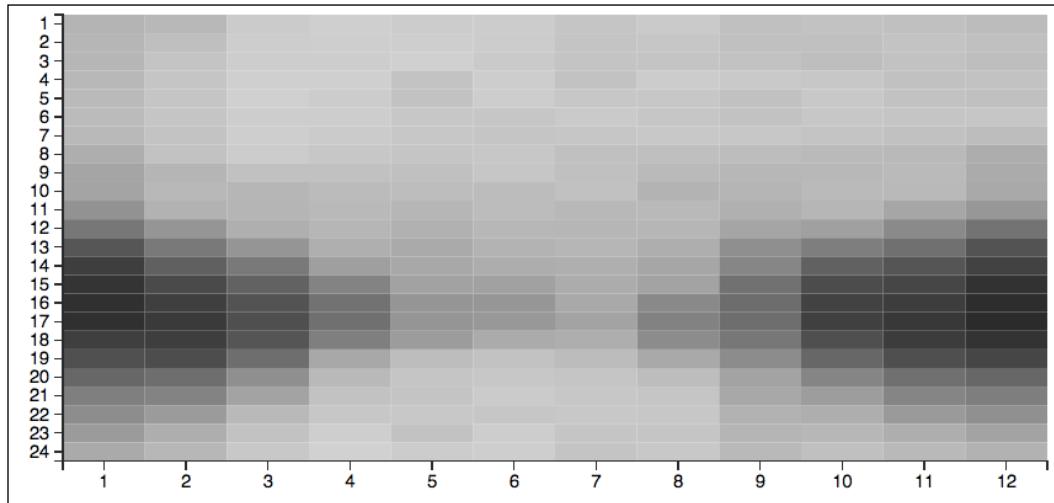
```
// Load the CSV data
d3.csv('/data/wind.csv', function(error, data) {

    // Handle errors getting or parsing the data
    if (error) { return error; }
    // Parse the dates
    data.forEach(function(d) {
        d.Date = new Date(d.Date);
    });

    // Create the heatmap chart in the container selection
    d3.select('div#chart01')
        .data([matrix(data)])
        .call(heatmap);
});

});
```

The resulting chart will display the variations of wind speed as a function of the hour of the day and the month. We can see that the wind is stronger between 2 pm and 8 pm and that it is weaker between March and September.



A heat map of the average wind speed by hour and month for 2013

Summary

In this chapter, we created a simple charting package with two components, a layout and a heat map chart. We also discussed the workflow and tasks related to the creation and distribution of a frontend package. We used Grunt with some plugins to concatenate, check the code for errors, test, and minify the assets. We used Vows to create test suites, and Bower to make our package easily installable in third-party frontend projects.

We created a small project with a single web page, which includes the charting package as an external dependency, and used it to visualize the average wind speed by hour and month as a heat map.

In the next chapter, we will learn how to create a data-driven application using third-party data and how to host the application using GitHub pages and Jekyll.

8

Data-driven Applications

In this chapter, we will create a data-driven application using data from the **Human Development Data API** from the United Nations website. We will use D3 to create a reusable chart component, and use Backbone to structure and maintain the application state. We will learn how to use **Jekyll** to create web applications using templates and a simplified markup language. We will also learn how to host our static site both on Amazon Simple Storage Service (S3) and GitHub Pages.

Creating the application

In this section, we will create a data visualization to explore the evolution of the **Human Development Index (HDI)** for different countries, and show the life expectancy, education, and income components of the index. We will create the visualization using D3 and Backbone.

The HDI is a composite statistic of life expectancy, education, and income created to compare and measure the quality of life in different countries. This indicator is used by the United Nations Development Program to measure and report the progress of the ranked countries in these areas.

In this visualization, we want to display how a particular country compares to other ranked countries in the evolution of the index. We will use the Human Development Data API to access the time series of the HDI for the ranked countries and to retrieve information about their main components.

The chart will show the evolution of the HDI for all the ranked countries, highlighting the selected country. In the right-hand side pane, we will display the main components of the HDI: life expectancy at birth, mean and expected years of schooling, and gross national income per capita (GNI). As there are almost two hundred ranked countries, we will add a search form with autocomplete to search among the countries. Selecting a country in the search input field will update the chart and the right-hand side pane.



A screenshot of the visualization elements

We will implement our application using D3 and Backbone, following the same pattern as presented in *Chapter 6, Interaction between Charts*. We will also leverage other libraries to provide the design elements and the functionality that we need.

The project setup

When creating software, we are continuously modifying our own work and the work developed for others. The ability to control how those changes are integrated in the project codebase and how to recover the previous versions is the heart of version control systems. As with many other tools, there are plenty of tools available, each with its own characteristics.

Git is a popular version control system. It's distributed, which means that it's not necessary to have a centralized location as the reference point; each working copy of the repository can be used as a reference. In this section, we will use Git as a version control system. We can't include an introduction to Git in this book; you can learn about Git from its website (<http://git-scm.com>).

Some content in this chapter is specific to Git and GitHub, a code-hosting repository based on Git, specifically the sections on how to use GitHub Pages to host static pages and the setup of the project. The example application can be implemented even without a version control system or hosting service.

We will begin our application by creating a repository for it in GitHub. To create a repository, go to <http://www.github.com>, sign in with your account (or create an account), and select **+New repository**. To create a repository, we will need to add a name and description, and optionally, select a license and add a README file. After doing this, the repository URL will be generated; we will use this URL to clone the project and modify it. In our case, the URL of the repository is <https://github.com/pnavarrc/hdi-explorer.git>. Cloning the repository will create a new directory, containing the initial content set on GitHub, if any:

```
$ git clone https://github.com/pnavarrc/hdi-explorer.git
```

Alternatively, we could have created an empty directory and initialized a Git repository in it to begin working on it right away.

```
$ mkdir hdi-explorer
$ cd hdi-explorer
$ git init
```

If we decide to use GitHub, we can add a remote repository to push our code. Remote repositories are locations on the Internet or in a network to make our code accessible for others. It's customary to set the `origin` remote to the primary repository for the project. We can set the `origin` remote and push the initial version of our project by executing the following commands in the console:

```
$ git remote add origin https://github.com/pnavarrc/hdi-explorer.git
$ git push -u origin
```

In either case, we will have a configured repository that is ready to be worked on.

As you may remember, we learned how to use Bower to manage the frontend dependencies in *Chapter 7, Creating a Charting Package*. As we will be using several libraries, we will begin by creating the `bower.json` file using `bower init`. We will set the name, version, author, description, and home page of our project. Our `bower.json` file will contain the basic information of our project as follows:

```
{  
  "name": "hdi-explorer"  
  "version": "0.1.0"  
  "authors": [  
    "Pablo Navarro <pnavarrc@gmail.com>"  
  ],  
  "description": "Human Development Index Explorer"  
  "main": "index.html"  
  "homepage": "http://pnavarrc.github.io/hdi-explorer"  
  "private": true,  
  "dependencies": {}  
}
```

As mentioned before, we will use D3 and Backbone to create the charts and structure our application. We will install the dependencies in our project:

```
$ bower install --save-dev d3 backbone underscore
```

This will create the `bower_components` directory and download the packages for us. We will also need *Bootstrap* and *Font Awesome* to include the HDI component icons. We will install these packages as well. As jQuery is a dependency of Bootstrap, Bower will install it automatically:

```
$ bower install --save-dev bootstrap font-awesome typeahead.js
```

We will use the Typeahead library from Twitter to add autocompletion to our search form. Installing packages with the `--save-dev` option will update our `bower.json` file, adding the packages to the development dependencies. These libraries would normally be regular dependencies; we are including them as development dependencies because we will later create a file with all the dependencies in that file.

```
{  
  "name": "hdi-explorer"  
  // ...  
  "devDependencies": {  
    "d3": "~3.4.1"  
    "bootstrap": "~3.1.0"  
    "backbone": "~1.1.0"  
    "underscore": "~1.5.2"
```

```
    "font-awesome": "~4.0.3"
    "typeahead.js": "~0.10.0"
  }
}
```

Using Bower will help us to manage our dependencies and update the packages without breaking our application. Bower requires its packages to adhere to semantic version numbering, and it's smart enough to update only the nonbackwards-incompatible releases.

Generating a static site with Jekyll

In the previous chapters, we covered how to create data-driven applications and how to use tools to make this task easier. In this section, we will cover how to generate websites using Jekyll and host web applications using Jekyll and GitHub Pages.

Jekyll is a simple, blog-aware, static site generator written in Ruby. This means that we can create a website or blog without having to install and configure web servers or databases to create our content. To create pages, we can write them in a simple markup language and compile them in HTML. Jekyll also supports the use of templates and partial HTML code to create the pages.

In Linux and OS X, we can install Jekyll from the command line. Remember that in order to use Jekyll, Ruby and RubyGems should be installed. To install Jekyll, run the following command in the console:

```
$ gem install jekyll
```

For other platforms and more installation options, refer to the documentation available at <http://jekyllrb.com/docs/installation/>. Jekyll includes a generator that configures and creates the project boilerplate with sample content and templates (see `jekyll new --help`), but this time, we will create the templates, content, and configuration from scratch. We will begin by creating the subdirectories and files needed for the Jekyll components:

```
hdi-explorer/
  _includes/
    navbar.html
  _layouts/
    main.html
  _data/
  _drafts/
  _posts/
  index.md
  _config.yml
```

The `_config.yml` file is a configuration file written in YAML, a serialization standard similar to JSON but with additional types, support for comments, and which uses indentation instead of brackets to indicate nesting (for more information, see <http://www.yaml.org/>). The content in this file defines how Jekyll will generate the content and defines some site-wide variables. For our project, the `_config.yml` file defines some Jekyll options and the name, base URL, and repository of our site:

```
# Jekyll Configuration
safe: true
markdown: rdiscount
permalink: pretty

# Site
name: Human Development Index Explorer
baseurl: http://pnavarrc.github.io/hdi-explorer
github: http://github.com/pnavarrc/hdi-explorer.git
```

The `safe` option disables all the Jekyll plugins, the `markdown` option sets the `markdown` language that we will use, and the `permalink` option defines which kind of URL we want to generate. There are additional options to set the time zone and excluded files, among others.

The `_layouts` directory will contain page templates with placeholders to be replaced by content. On each page, we can declare which layout we will use in a special area called the YAML front matter, which we will describe later. The templates can have variables, such as the `{{ content }}` tag, and include the `{% include navbar.html %}` tag. The content of the variables is defined either in the front matter or in the `_config.yml` file. The `include` tags replace the block with the content of the corresponding file in the `_includes` directory. For instance, our `main.html` template contains the following base page structure:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>{{ page.title }}</title>
  <link href="{{ site.baseurl }}/hdi.css" rel="stylesheet">
</head>
<body>
  <!-- Navigation Bar -->
  {% include navbar.html %}

  <!-- Content -->
```

```
<div class="container-fluid">
  {{ content }}
</div>
</body>
</html>
```

The value of the `{{ site.baseurl }}` variable is set in the `_config.yml` file, the `{{ page.title }}` variable will use the value of the font matter of the pages using the template, and the `{{ content }}` variable will be replaced with the content of the files that use the template. In the `_config.yml` file, we defined the `baseurl` variable to `http://pnavarrc.github.io/hdi-explorer`. When Jekyll uses this template to generate content, it will replace the `{{ site.baseurl }}` variable with `http://pnavarrc.github.io/hdi-explorer`, and the generated page will have the complete URL for the CSS style, `http://pnavarrc.github.io/hdi-explorer/hdi.css`. A complete reference to the Liquid Templating language is available at <http://liquidmarkup.org/>.

The `_includes` directory contains the HTML fragments to be included in other pages. This is useful to modularize some parts of our page, such as to separate the footer, header, or navigation bar. Here, we created the `navbar.html` file with the content of our navigation bar:

```
<!-- Navigation Bar -->
<nav class="navbar navbar-default" role="navigation">
  <div class="container-fluid">
    <!-- ... more elements -->
    <a class="navbar-brand" href="#">{{ site.name }}</a>
    <!-- ... -->
  </div>
</nav>
```

The content of the `navbar.html` file will replace the `{% include navbar.html %}` liquid tag in the templates. The files in the `_includes` directory can also contain liquid tags, which will be replaced properly.

The `_posts` directory contains blog posts. Each blog post should be a file with a name in the `YEAR-MONTH-DAY-title.MARKDOWN` form. Jekyll will use the filename to compute the date and URL of each post. The `_data` directory contains additional site-wide variables, and the `_draft` directory contains posts that we will want to publish later. In this project, we won't create posts or drafts.

The `index.md` file contains content to be rendered using some of the layouts in the project. The beginning of the file contains the YAML front matter, that is, the lines between three dashes. This fragment of the file is interpreted as YAML code and is used to render the templates. For instance, in the main template, we had the `{ { page.title } }` placeholder. When rendering the page, Jekyll will replace this with the `title` variable in the page's front matter. The content after the front matter will replace the `{ { content } }` tag in the template.

```
---
layout: main
title: HDI Explorer
---

<!-- Content -->
Hello World
```

We can add any number of variables to the front matter, but the `layout` variable is mandatory; so, Jekyll knows which layout should be used to render the page. In this case, we will use the `main.html` layout and set the title of the page to `HDI Explorer`. The content of the page can be written in HTML or in text-to-HTML languages, such as *Markdown* or *Textile*. Once we have created the layouts and initial content, we can use Jekyll to generate the site.

```
$ jekyll build
```

This will create the `_site` directory, which contains all the content of the directory except the Jekyll-related directories such as `_layouts` and `_includes`. In our case, it will generate a directory containing an `index.html` file. This file is the result of injecting the contents of the `index.md`, `navbar.html`, and `_config.yml` files in the `main.html` layout. The generated file will look as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>HDI Explorer</title>
  <link href="http://pnavarrc.github.io/hdi-explorer/hdi.css"
rel="stylesheet">
</head>
<body>
  <!-- Navigation Bar -->
  <nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
      <!-- ... more elements -->
      <a class="navbar-brand" href="#">
```

```
Human Development Index Explorer
</a>
<!-- ... -->
</div>
</nav>

<!-- Content -->
<div class="content">
    <!-- Content -->
    <p>Hello World</p>
</div>
</body>
</html>
```

We can see that the generated file is pure HTML. Using Jekyll allows us to modularize the page, separate its components, and allows us to focus on writing the actual content of each page.

Jekyll also allows us to serve the generated pages locally, watching for changes in the project files. To do this, we need to overwrite the `baseurl` variable in order to use the localhost address instead of the value defined in the configuration file. As the addresses will be relative to the project directory, we can set the base URL to an empty string:

```
$ jekyll serve --watch --baseurl=
```

We can now access our site by pointing the browser to `http://localhost:4000` and use our site. In the next section, we will create the contents of our application using D3 and Backbone, integrating the JavaScript files, styles, and markup with the Jekyll templates and pages created in this section.

Creating the application components

We will separate the Backbone application components from the chart; we will put the models, collections, views, and setup in the `js/app` directory:

```
js/
  app/
    models/
      app.js
      country.js
    collections/
      countries.js
```

```
views/
  country.js
  countries.js
app.js
setup.js
```

In the `app.js` file, we just define a variable that will have the components of our application:

```
// Application container
var app = {};
```

The `setup.js` file contains the creation of the model, collection, and view instances; the binding of the events; and the callbacks of different components. We will review the models, collections, and views in detail later.

Creating the models and collections

The application model will reflect the application state. In our application, the selected country defines the state of the application; we will use three-letter country codes as the only attribute of the application model:

```
// Application Model
app.ApplicationModel = Backbone.Model.extend({
  // Code of the Selected Country
  defaults: {
    code: ''
  }
});
```

We will have two additional models: the `CountryInformation` model will represent information about the current HDI value and its main components, and the `CountryTrend` model will contain information about the country and time series of HDI measurements.

The data source for these models will be an endpoint of the Human Development Data API. The API allows us to retrieve data about poverty, education, health, social integration, and migrations, among many others. A complete list of the API endpoints and some examples of queries are available at the Human Development Data API website (<http://hdr.undp.org/en/data/api>). The API exposes the data in several formats and receives parameters to filter the data.

The CountryInformation model will retrieve information about the **Human Development Index and its Components** endpoint. For instance, we can access this endpoint by passing name=Germany as the parameter of the request. The request to <https://data.undp.org/resource/myer-egms.json?country=Germany> will return a JSON file with the main components of the HDI:

```
[ {
  "hdi_rank" : "6",
  "_2012_expected_years_of_schooling" : "16.3",
  "_2012_2013_change_in_rank" : "0",
  "_2012_human_development_index_hdi_value" : "0.911",
  "_2013_life_expectancy_at_birth_years" : "80.7",
  "_2012_mean_years_of_schooling" : "12.9",
  "_2013_hdi_value" : "0.911",
  "country" : "Germany",
  "_2013_gross_national_income_gni_per_capita_2011_ppp" : "43049"
} ]
```

We will define the model such that it has the name and code attributes and some of the information provided by the JSON file. We will add the url, baseurl, and urltpl attributes to construct the URL for each country, as follows:

```
// Country Information Model
app.CountryInformation = Backbone.Model.extend({


  // Default attributes, the name and code of the country
  defaults: {
    code: '',
    name: ''
  },

  // URL to fetch the model data
  url: '',

  // Base URL
  baseurl: 'http://data.undp.org/resource/wxub-qc5k.json',

  // URL Template
  urltpl: _.template('<%= baseurl %>?Abbreviation=<%= code %>')
});
```

Each country has the `abbreviation` field; this field contains the code for the country. We will use this code as the ID of the country. The names of the attributes of the JSON object contain data; for instance, the `_2012_life_expectancy_at_birth` attribute contains the year of the measurement. If we create an instance of the model and invoke its `fetch` method, it will retrieve the data from the JSON endpoint and add attributes to each attribute of the retrieved object. This will be a problem because the endpoint returns an array and not all the countries have up-to-date measurements. In the case of Germany, the only object in the array has the `_2012_life_expectancy_at_birth` attribute, but in the case of other countries, the most recent measurements could be from 2010.

To have a uniform representation of the data, we can strip the year out of the attribute names before we set the attributes for the model. We can do this by setting the `parse` method, which is invoked when the data is fetched from the server. In this method, we will get the first element of the retrieved array and strip the first part of the attributes beginning with `_` to only have attributes of the form `life_expectancy_at_birth`:

```
// Parse the response and set the model contents
parse: function(response) {

    // Get the first item of the response
    var item = response.pop(),
        data = {
            code: item.abbreviation,
            name: item.name
        };

    // Parse each attribute
    for (var attr in item) {
        if (attr[0] === '_') {
            // Extract the attribute name after the year
            data[attr.slice(6)] = item[attr];
        }
    }

    // Return the parsed data
    return data;
}
```

We will also add a method to update the model with the selected country in the application. The `setState` method will receive the application model and use its `code` attribute to construct the URL for the selected country and to fetch the new information, as follows:

```
setState: function(state) {
    // Construct the URL and fetch the data
    this.url = this.urltpl({
        baseurl: this.baseurl,
        code: state.get('code')
    });
    this.fetch({reset: true});
}
```

We will create the `CountryTrend` model to store the trends of the HDI for each country, and the `Countries` collection to store the `CountryTrend` instances. The `CountryTrend` model will hold the country code and name, a flag to indicate whether the country has been selected or not, and a series of HDI measurements for different years. We will use the code of the country as an ID attribute:

```
// Country Trend Model
app.CountryTrend = Backbone.Model.extend({

    // Default values for the Country Trend Model
    defaults: {
        name: '',
        code: '',
        selected: false,
        hdiSeries: []
    },

    // The country code identifies uniquely the model
    idAttribute: 'code'
});
```

The Countries collection will contain a set of the CountryTrend instances. The collection will have an endpoint to retrieve the information for the model instances. We will need to define the parse method in the CountryTrend model, which will be invoked automatically before new CountryTrend instances are generated. In the parse method, we construct an object with the attributes of the new model instance. The data retrieved for the collection will have the following structure:

```
{  
  _1990_hdi: "0.852"  
  _1980_hdi: "0.804"  
  _2000_hdi: "0.922"  
  // ...  
  _2012_hdi: "0.955"  
}
```

Here, the year of the HDI measurement is contained in the key of the object. There are other attributes also, which we will ignore. We will split the attribute name using the `_` character to extract the year:

```
// Parse the country fields before instantiating the model  
parse: function(response) {  
  
  var data = {  
    code: response.country_code,  
    name: response.country_name,  
    selected: false,  
    hdiSeries: []  
  };  
  
  // Compute the HDI Series  
  for (var attr in response) {  
    var part = attr.split('_'),  
      series = [];  
  
    if ((part.length === 3) && (part[2] === 'hdi')) {  
      data.hdiSeries.push({  
        year: parseInt(part[1], 10),  
        hdi: parseFloat(response[attr])  
      });  
    }  
  }  
  
  // Sort the data items  
  data.hdiSeries.sort(function(a, b) {  
    return b.year - a.year;  
  });  
  
  return data;  
}
```

The data object will contain the country's code, name, the selected flag, and the array of hdiSeries, which will contain objects that have the year and HDI value. The Countries collection will contain the CountryTrend instances for each country. The data will be retrieved from the **Human Development Index Trends** endpoint. We set the collection model, the endpoint URL, and a parse method, which will filter the items that have a country code (there are items for groups of countries). We will also add a method to set a selected item in order to ensure that there is only one selected item:

```
// Countries Collection
app.Countries = Backbone.Collection.extend({


    // Model
    model: app.CountryTrend,


    // JSON Endpoint URL
    url: 'http://data.undp.org/resource/efc4-gjvq.json',


    // Remove non-country items
    parse: function(response) {
        return response.filter(function(d) {
            return d.country_code;
        });
    },


    // Set the selected country
    setSelected: function(code) {

        var selected = this.findWhere({selected: true});

        if (selected) {
            selected.set('selected', false);
        }

        // Set the new selected item
        selected = this.findWhere({code: code});
        if (selected) {
            selected.set('selected', true);
        }
    }
});
```

We will proceed to create the views for the models.

Creating the views

The application will have three views: the chart with the trends of HDI values for the ranked countries, the information view at the right-hand side, and the search form.

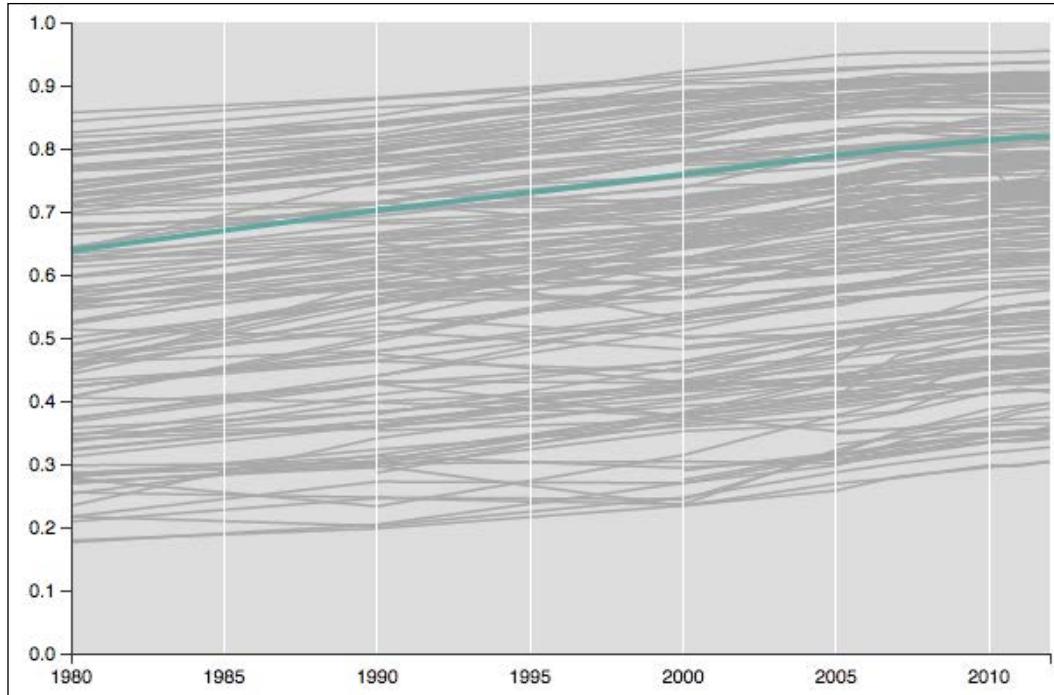
`CountriesTrendView` is a view of the `Countries` collection, which displays the evolution of the HDI for the ranked countries. As we did in *Chapter 6, Interaction between Charts*, we will create a Backbone View that will contain an instance of a D3-based chart:

```
// Countries Trend View
app.CountriesTrendView = Backbone.View.extend({  
  
    // Initialization and render  
    initialize: function() {  
        this.listenTo(this.collection, 'reset', this.render);  
        this.listenTo(this.collection, 'change:selected', this.  
render);  
    }  
});
```

In the `initialize` method, we start listening for the `reset` event of the collection and the `change:selected` event, which are triggered when an element of the collection is selected. In both cases, we will render the view. We will add and configure an instance of the D3-based chart, `hdi.chart.trends`:

```
app.CountriesTrendView = Backbone.View.extend({  
  
    // Initialization and render...  
  
    // Initialize the trend chart  
    chart: hdi.chart.trend()  
        .series(function(d) { return d.hdiSeries; })  
        .x(function(d) { return d.year; })  
        .y(function(d) { return d.hdi; }),  
  
    // Initialize and render methods...  
});
```

We will skip the description of the `hdi.chart.trend` chart for brevity, but as usual, the chart is implemented using the reusable chart pattern and has accessor methods to configure its behavior. The chart displays the time series of HDI measurements for all the ranked countries, highlighting the line bound to a data item with the `selected` attribute set to `true`.



The HDI trend chart

In the `render` method, we get the width of the container element of the view and update the width of the chart using this value. We select the container element, bind the collection data to the selection, and invoke the chart:

```
// Update the chart width and bind the updated data
render: function() {
    // Update the width of the chart
    this.chart.width(this.$el.width());

    // Rebind and render the chart
    d3.select(this.el)
        .data([this.collection.toJSON()])
        .call(this.chart);
},
```

We will also add the `setState` method to change the selected country of the underlying collection. This method will help us to update the selected item of the collection when the application model changes the selected country. We will do this later in the application setup:

```
// Update the state of the application model
setState: function(state) {
    this.collection.setSelected(state.get('code'));
}
```

The search form will allow the user to search among the ranked countries to select one of them. We will use the **Typeahead jQuery plugin** from Twitter (<http://twitter.github.io/typeahead.js/>) to provide autocompletion, and we will populate the suggestion list with the items in the `Countries` collection. The Typeahead plugin contains two components: Bloodhound, the autocompletion engine, and Typeahead, the plugin that adds autocompletion capabilities to an input field.

In the `initialize` method, we bind the `reset` event of the collection to the `render` method in order to update the view when the list of country trends is retrieved:

```
// Search Form View
app.CountriesSearchView = Backbone.View.extend({

    // Initialize
    initialize: function() {
        this.listenTo(this.collection, 'reset', this.render);
    },

    // Events and render methods...
});
```

The DOM element associated with this view will be a div containing the search form, which is located in the navigation bar. We assign an ID to the div and to the input field:

```
<div class="form-group" id="search-country">
    <input type="text" class="form-control typeahead"
        placeholder="Search country" id="search-country-input">
</div>
```

To provide autocompletion, we need to initialize the autocompletion engine and add the autocompletion features to the search input item:

```
// Render the component
render: function() {
    // Initialize the autocompletion engine
    // Add autocompletion to the input field
},
```

We initialize the Typeahead autocomplete engine; setting the `datumTokenizer` option is a function that, given a data element, returns a list of strings that should be associated with the element. In our case, we want to match the country names; so, we use the whitespace tokenizer and return the country name split by whitespace characters. The input of the search field will be split using whitespace characters too. We add the list of elements among which we want to search, which in our case are the elements in the collection:

```
// Render the component
render: function() {
    // Initialize the autocomplete engine
    this.engine = new Bloodhound({
        datumTokenizer: function(d) {
            return Bloodhound.tokenizers.whitespace(d.name);
        },
        queryTokenizer: Bloodhound.tokenizers.whitespace,
        local: this.collection.toJSON()
    });
    this.engine.initialize();

    // Add autocomplete to the input field...
},

```

To add the autocomplete features to the search form, we select the input element of the view and configure the typeahead options. In our case, we just want to show the name of the country and use the engine dataset as the source for the autocomplete:

```
// Render the element
this.$el.children('#search-country-input')
.typeahead(null, {
    displayKey: 'name',
    source: this.engine.ttAdapter()
}),

```

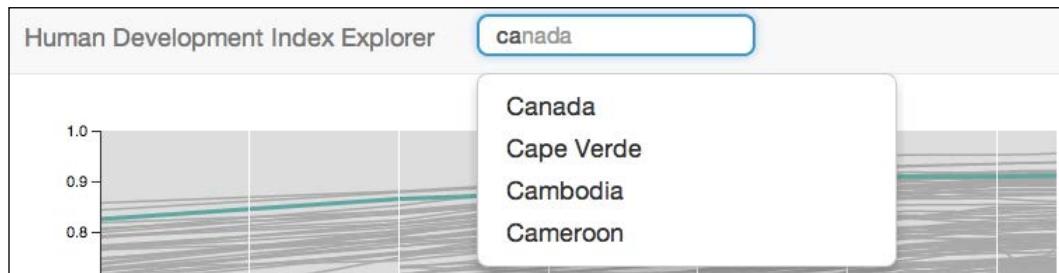
When the user begins to type in the input field, the options that match the input will be displayed. When the user selects an option, the `typeahead:selected` event will be triggered by the input element. We will add this to the event's hash of the view, binding the event to the `setSelected` callback:

```
events: {
    'typeahead:selected input[type=text]': 'setSelected'
},

```

Note that the `typeahead:selected` event is a jQuery event. The callback will receive the event and the data item selected by the user, and it will update the selected item in the collection, as follows:

```
// Update the selected item in the collection
setSelected: function(event, datum) {
    this.collection.setSelected(datum.code);
}
```



The Typeahead autocomplete in action

The last view will be `CountryInformationView`. This view is a visual representation of the `CountryInformation` model. For this view, we will add the `_includes/country-information.html` file with the contents of the template and include it in the `index.md` file:

```
---
layout: main
title: HDI Explorer
---

{% include country-information.html %}

<!-- More content... -->
```

The template will contain several internal div elements; we will show only a part of the template here:

```
<!-- Country Information Template -->
<script type="text/template" id="country-summary-template">

<!-- Country Name and Rank -->
<div class="row country-summary-title">
    <div class="col-xs-8"><%= name %></div>
```

```

<div class="col-xs-4 text-right">#<%= hdi_rank %></div>
</div>

<!-- HDI Value and Rank of the Country -->
<div class="row country-summary-box">
    <!-- Header -->
    <div class="col-xs-12 country-summary-box-header">
        <i class="fa fa-bar-chart-o fa-fw"></i>
        human development index
    </div>
    <!-- HDI Index -->
    <div class="col-xs-12">
        <div class="col-xs-9">human development index</div>
        <div class="col-xs-3 text-right"><%= hdi_value %></div>
    </div>
    <!-- Country Rank -->
    <div class="col-xs-12">
        <div class="col-xs-9">hdi rank</div>
        <div class="col-xs-3 text-right"><%= hdi_rank %></div>
    </div>
</div>

<!-- More divs with additional information... -->
</script>

```

Here, we create the structure of the bar on the right-hand side, which will contain the Human Development Index, rank, life expectancy, education statistics, and income for the selected country. We will use the Underscore templates to render this view. The view structure is simpler in this case; we just compile the template, listen to the changes of country name in the model, and render the template with the model data:

```

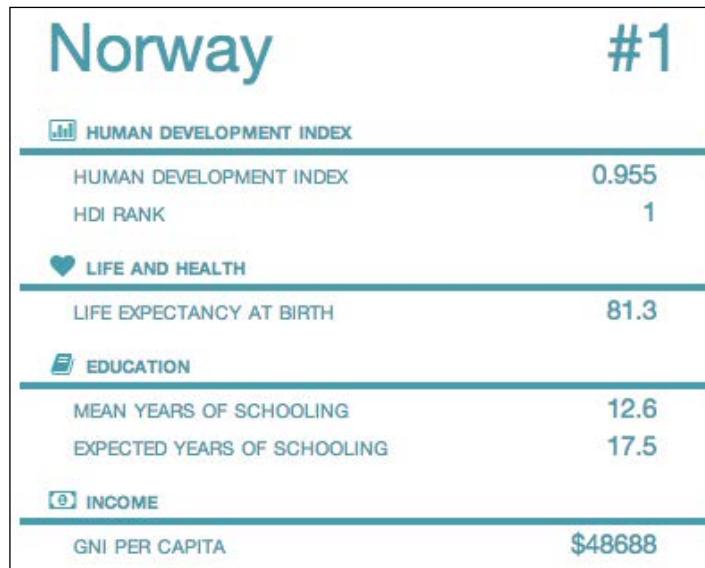
// Country Information View
app.CountryInformationView = Backbone.View.extend({
    // View template
    template: _.template($('#country-summary-template').html()),

    initialize: function() {
        // Update the view on name changes
        this.listenTo(this.model, 'change:name', this.render);
    },

    render: function() {

```

```
// Render the template
this.$el.html(this.template(this.model.toJSON()));
}
});
```



The rendered view will display the current values for the HDI components

The application setup

With the models, collections, and views created, we can create the respective instances and bind events to callbacks in order to keep the views in sync. We begin by creating an instance of the application model and the collection of country HDI trends. In the `js/app/setup.js` file, we create and configure the model, collection, and view instances:

```
// Application Model
app.state = new app.ApplicationModel();

// HDI Country Trends Collection
app.countries = new app.Countries();
```

After the application's state changes, we will have to update the selected item in the Countries collection. We bind the `change:code` event of the application model to the callback that will update the selected item in the collection:

```
// Update the selected item in the countries collection
app.countries.listenTo(app.state, 'change:code', function(state) {
    this.setSelected(state.get('code'));
});
```

We need to update the application state when the Countries collection is populated for the first time. We will set the application state's `code` attribute to the code of the first element in the collection of countries. We also bind the `change:selected` event of the collection to update the application model:

```
app.countries.on({
    'reset': function() {
        app.state.set('code', this.first().get('code'));
    },
    'change:selected': function() {
        var selected = this.findWhere({selected: true});
        if (selected) {
            app.state.set('code', selected.get('code'));
        }
    }
});
```

Note that when we are selecting an item, we are also deselecting another item. Both the items will trigger the `change:selected` event, but the application should change its state only when an item is selected. We can now fetch the countries data, passing the `reset` flag to ensure that any existing data is overwritten:

```
app.countries.fetch({reset: true});
```

We create an instance of the `CountryInformation` model and bind the changes to the `code` attribute of the application to the changes of the state in the model. The `setState` method will fetch the information for the code given by the application state:

```
// HDI Information
app.country = new app.CountryInformation();
app.country.listenTo(app.state, 'change:code', app.country.setState);
```

We can now create instances of the views. We will create an instance of the CountriesTrendView. This view will be rendered in the div element with the #chart ID:

```
// Countries Trend View
app.trendView = new app.CountriesTrendView({
  el: $('#chart'),
  collection: app.countries
});
```

We create an instance and configure the CountriesSearchView. This view will be rendered in the navigation bar:

```
app.searchView = new app.CountriesSearchView({
  el: $('#search-country'),
  collection: app.countries
});
```

We also create a CountryInformationView instance, which will be rendered in the right-hand side of the page:

```
app.infoView = new app.CountryInformationView({
  el: $('#table'),
  model: app.country
});
```

In the `index.md` file, we create the elements where the views will be rendered and include the application files. In the `main.html` layout, we include the CSS styles of the application, Bootstrap and Font Awesome:

```
---
layout: main
title: HDI Explorer
---

{%- include country-information.html %}

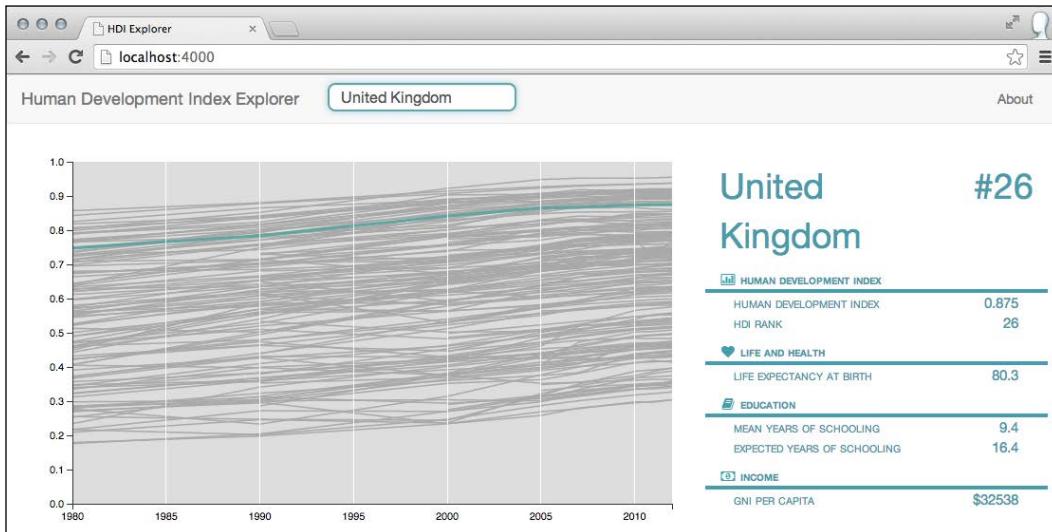
<div class="container-fluid">
  <div class="row">
    <div class="col-md-8" id="chart"></div>
    <div class="col-md-4 country-summary" id="table"></div>
```

```

</div>
</div>

<script src="{{ site.baseurl }}/dependencies.min.js"></script>
<script src="{{ site.baseurl }}/hdi.min.js"></script>

```



The application served by Jekyll on a localhost

Here, we consolidated jQuery, Bootstrap, Underscore, Backbone, Typeahead, and D3 in the `dependencies.min.js` file and the application models, collections, views, and chart in the `hdi.min.js` file. To create these consolidated files, we created a Gruntfile and configured concatenation and minification tasks as we did in *Chapter 7, Creating a Charting Package*. As the configuration of the tasks is similar to the configuration presented in the previous chapter, we will skip its description.

It is also worth mentioning that in general, it is not a good practice to include complete libraries. For instance, we included the complete Bootstrap styles and JavaScript components, but in the application, we used only a small part of the features.

Bootstrap allows you to include the components individually, reducing the payload of the page and improving the performance. We also included the Font Awesome fonts and styles only to include four icons. In a project where performance is crucial, we will probably include only the components that we really need.

Hosting the visualization with GitHub Pages

In the previous section, we created a web application using Jekyll, Backbone, and D3. With Jekyll, we created a template for the main page and included the minified JavaScript libraries and styles. With Jekyll, we can compile the markup files to generate a static website or serve the site without generating a static version using `jekyll serve`. In this section, we will publish our site using GitHub Pages, a hosting service for personal and project sites.

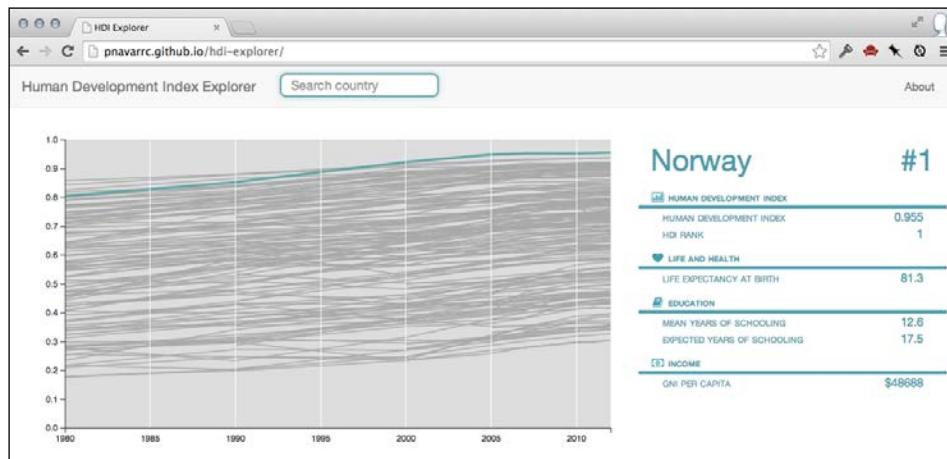
GitHub Pages is a service from GitHub that provides hosting for static websites created in Jekyll or HTML. To publish our Jekyll site, we need to create a branch named `gh-pages` and push the branch to GitHub. If this branch is a Jekyll project or contains an `index.html` file, GitHub will serve the content of this branch as a static site. We can create the branch from the master branch:

```
$ git checkout -b gh-pages
```

Next, push the branch to our origin, the GitHub endpoint:

```
$ git push -u origin gh-pages
```

This will push the `gh-pages` branch to GitHub, and GitHub Pages will generate the site in a few minutes. The application will be published and will be accessible through an URL of the form `http://user.github.io/project-name`, in our case, `http://pnavarrc.github.io/hdi-explorer`. It's important to remember that the base URL for the project will be `http://user.github.io/project-name`. Set the `baseurl` variable in the `_config.yml` file correctly to avoid path problems with the styles and JavaScript files.



The published version of the HDI Explorer application

We can create personal pages as well, but in this case, we would need to create a repository with a name of the form `user.github.io`, and the site will be a server from the `http://user.github.io` URL. GitHub Pages also allows you to use custom domains and plain HTML instead of Jekyll. More information about GitHub Pages can be found in the project site at `http://pages.github.com/`.

Hosting the visualization in Amazon S3

As an alternative to publishing our pages using GitHub Pages, we can also serve static sites using Amazon S3. Amazon S3 is a data storage service provided by Amazon. It can be used to store files of any kind, and in particular, to store and serve static websites. Amazon S3 stores data with 99.99 percent availability and scales well in terms of storage capacity, number of requests, and users. It also provides fine-grained control access, allowing you to store sensible data.

The pricing depends on how much data you store and how many access requests are made, but it starts at less than 0.1 USD per GB per month, and there is a free tier available to use the platform at no charge to store up to 5 GB (and 20,000 requests per month). In this section, we will assume that you have an Amazon Web Services account. If you don't, you can sign up at `http://aws.amazon.com/` and create an account.

In Amazon S3, the files are stored in **buckets**. A bucket is a container for objects. Buckets are stored in one of several regions; the region is usually chosen to optimize the latency or to minimize costs. The name of the bucket needs to be unique among the buckets in Amazon.

To host our site, we will create a bucket. To create the bucket, we need to go to the Amazon S3 console at `https://console.aws.amazon.com` and select **Create Bucket**. Here, we need to name our bucket and assign it a region. We will name it `hdi-explorer` and select the default region. When the bucket is created, we select the bucket and go to **Properties**. In the **Static Website Hosting** section, we can enable the hosting and retrieve the URL of our bucket. We will use this URL as the base URL of the site.

Configuring Jekyll to deploy files to S3

To deploy static content to Amazon S3, we need to generate a version of the site with the `baseurl` variable set to the Amazon S3 endpoint. We will create an alternate Jekyll configuration file and use it to generate the S3 version of the site. In this case, we only need to update the base URL of the site, but this new file can have different configuration values. We will create the `_s3.yml` file with the following options:

```
# Jekyll Configuration
safe: true
markdown: rdiscount
permalink: pretty
destination: _s3
exclude:
  - bower_components
  - node_modules
  - Gruntfile.js
  - bower.json
  - package.json
  - README.md
# Site
name: Human Development Index Explorer
baseurl: http://hdi-explorer.s3-website-us-east-1.amazonaws.com
```

We set the destination folder to `_s3`. This will generate the files that we need to deploy to Amazon in the `_s3` directory. We have also excluded the files that are not needed to serve the page. We can now use this configuration to build the D3 version of the site:

```
$ jekyll build --config _s3.yml
```

We can check whether the links in the generated files point towards the S3 endpoint.

Uploading the site to the S3 bucket

We can upload the files using the web interface in the Amazon AWS console; however, the interface doesn't allow you to upload complete directories. Instead, we will use `s3cmd`, a command-line tool that helps to upload, download, and sync directories with S3 Buckets. To download and install `s3cmd`, follow the instructions available on the project website (<http://s3tools.org/s3cmd>).

Before uploading the files, we need to configure `s3cmd` to provide the Amazon security credentials. To generate new access keys, go to your account, then go to security credentials, and then select **Access Keys**. You can generate a new pair of access key ID and secret. These strings allow you to authenticate applications to access your S3 Buckets.

With the access key and secret, we can configure `s3cmd` to use it to upload our files:

```
$ s3cmd --configure
```

This command will request for our access key ID and secret. We can now upload the files. As we just want to upload our files, we can simply use the following command:

```
$ s3cmd sync _s3/ s3://bucket-name
```

The first time, this will upload the files to your bucket. Once you have the content in S3, it will keep the bucket synchronized with the `_s3` directory and upload only the files that have changed.

Finally, we need to make our files public so that everyone with the URL can access the application. To do this, go to the bucket page in the browser, select all the files, and select **Make Public** in the **Actions** menu. The site will now be available at the bucket endpoint URL.

Summary

In this chapter, we learned how to create static sites with Jekyll and how to integrate third-party data sources using API endpoints. We used the Human Development Data API from the United Nations to visualize the evolution of the HDI of the ranked countries, displaying the main components of this indicator as a table.

To create the application, we used several JavaScript and CSS libraries, and we used Grunt to concatenate and minify the project assets before publishing the site. We also learned how to publish sites created with Jekyll using GitHub Pages for projects or personal pages, and how to configure and use Amazon S3 to host static websites.

In the next chapter, we will learn how to create data visualization dashboards and how to make our visualizations responsive.

9

Creating a Dashboard

Dashboards are a special kind of data visualization. They are widely used to monitor website analytics, business intelligence metrics, and brand presence in social media, among many other things. Dashboards are especially difficult to design, because a great amount of information should be displayed in a limited amount of space.

In this chapter, we will define what a dashboard is and discuss some of the strategies and design patterns that can help us design an effective dashboard. We will design and create a dashboard to monitor the performance of students in a class.

Defining a dashboard

Before we create our dashboard, it will be useful to clarify what a dashboard is. A quick search on Google Images will reveal that there isn't any consistent definition of what a dashboard is. Most of the results are a collection of charts, tables, gauges, and indicators that seem intended to monitor business performance, website analytics, and presence of a brand in social media, among many other things. Stephen Few, a specialist in business intelligence and information design, provides a great definition of dashboards:

A dashboard is a visual display of the most important information needed to achieve one or more objectives, which is consolidated and arranged on a single screen so the information can be monitored at a glance.

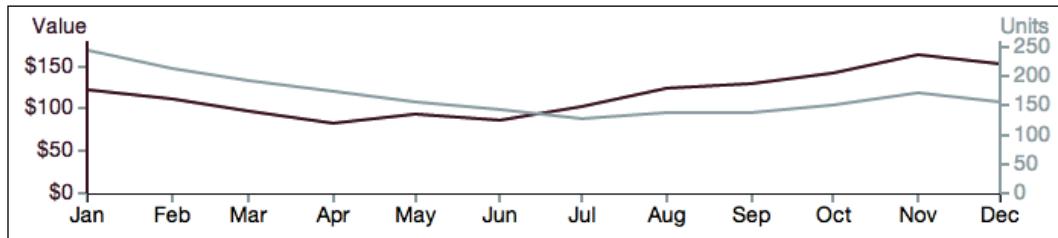
Creating a Dashboard

This definition has several implications. First, dashboards are visual displays of information. A dashboard can contain text elements, but it's mainly a visual display. Well-designed graphics and charts are a highly effective medium to communicate quantitative information. Suppose we have a list with the monthly sales of a person in value and units sold. We can display this information in a table, as follows:

Name		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
John Doe	Value	\$123	\$112	\$98	\$82	\$93	\$87	\$103	\$125	\$129	\$143	\$163	\$153
	Units	244	214	193	174	155	144	127	138	139	152	171	155

Series of sales shown as a table

We can also create a line chart with the same information. The following image displays the same data in a visual form. We have two lines and their corresponding axes, one for the units sold and another for the value of the monthly sales.



Series of sales, this time as a line chart

In the line chart, we can easily spot seasonal changes in the sales, find the minimum or maximum value, and identify patterns and changes in sales. Of course, if we need precision, we will need the table, but if we need to detect changes quickly, the chart is a better choice.

Dashboards should be designed with a purpose in mind. If our dashboard shows general information, it won't be useful. We should know what kind of decisions will be made, which problems need to be detected, and what information will help the dashboard users make decisions and take action.

The dashboard should display the most important information that is needed to achieve its purpose. This means that it should gather all the relevant data, perhaps from different sources, to help decision makers detect problems.



The previous definition of dashboards states that the information in a dashboard should be arranged in a single screen, or more generally, the dashboard should fit in the eye span of the user. This is important in order to provide an overall view. The information should be visible at all times, and the user shouldn't have to scroll the page to view a chart or click on something to have a modal window with additional information.

In the next section, we will design and create a dashboard to monitor the performance of students in a class. We will state the purpose of our dashboard, list the relevant information that we need, organize this information in sections, and create charts for each section in order to finally organize our information to help the user detect problems and take actions to solve them.

Good practices in dashboard design

When designing a dashboard, we need to get the most out of each section of the screen so that we can design our graphics to maximize the absorption of information. Some visual attributes are more easily perceived than others, and some of them can communicate effectively without having to pay full attention to them. These attributes are called **preattentive attributes of visual perception**. We will discuss some of them in the context of dashboard design:

- **Color:** There are several ways to describe color. One of the color models is called the **HSL model**, which is better for humans to understand. In this model, the color is described by three attributes, hue (what we usually call color), saturation (intensity of a color), and lightness or brightness. The perception of color depends on the context. A light color will draw attention if it's surrounded by dark colors; a highly saturated blue will be flashy if the background is a pale color. The dashboard should guide the viewer's attention to issues that require action. To achieve this, we need to choose the colors wisely, reserving colors with high contrast to elements and areas that deserve special attention.

- **Form:** Length, width, and size can encode quantitative dimensions effectively, with different degrees of precision. We can quickly determine whether a line is twice the length of another, but it can be more difficult to determine this with the width or the size of circles. Items of different shapes are perceived as belonging to different categories or kinds of elements.
- **Position:** The position of items plays an important role in the communication of information. Scatter plots encode pairs of values with two-dimensional positions; we are inclined to think that items that are close are related. Position can also encode hierarchy. Higher items are considered better or more important than items below them, and items located on the left-hand side will be seen first if the viewer's language is written from left to right.

We can use all these elements to design the components of our dashboard so the viewer's attention is directed to the issues that require action. As dashboards usually contain a huge amount of information, every inch of the screen counts. In desktop environments, the horizontal space is usually enough, but the vertical space is scarce. In mobile environments, there is usually more vertical space, but the overall available space is more challenging.

To use the available space efficiently, we need to select the information that really counts, and prefer compact charts and graphics. It is essential to use charts and graphics that are clear and direct, reducing explicit decoding to the absolute minimum.

The dashboard should be well organized in order to allow the user to quickly locate each piece of information. We will use the aforementioned visual attributes to establish a clear hierarchy between elements and clearly define sections dedicated to displaying information about the students, courses, and the entire class.

These are just a few guidelines that should be considered when creating a dashboard; for more in-depth treatment, please refer to *Information Dashboard Design* by Stephen Few (see the reference in *Chapter 1, Data Visualization*).

Making a dashboard

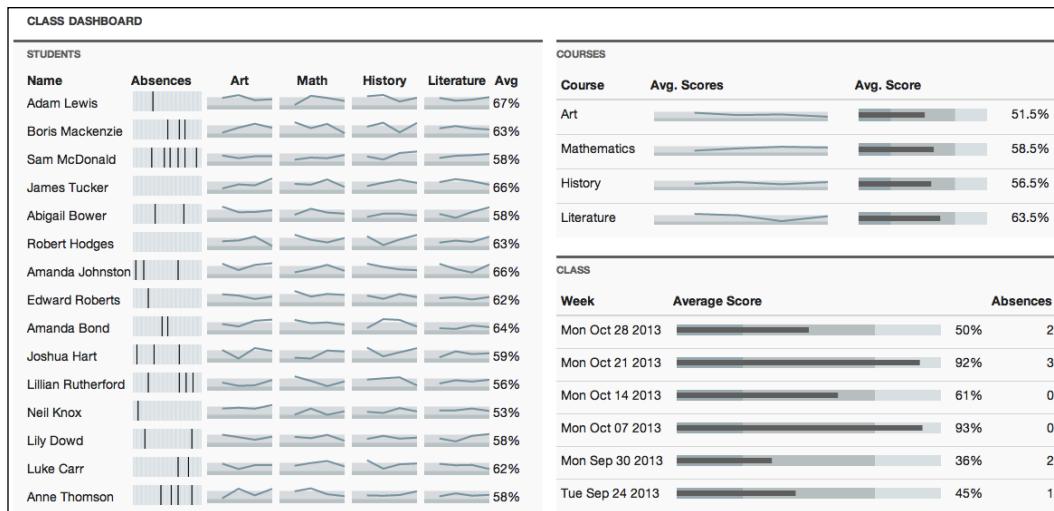
As we mentioned previously, designing an effective dashboard is a challenging task. The first step to create a useful dashboard is to determine which questions need to be answered by the dashboard, which problems need to be detected on time, and why a dashboard is required.

Once we have determined the purpose of the dashboard, we can begin to gather all the data that can help us answer the questions and understand the issues stated in the dashboard purpose. The data can be originated from several sources.

We will then need to organize the information in meaningful sections in order to help the user easily navigate the dashboard and find the required information.

We will also need to choose the visual displays for each piece of data that we want to put in the dashboard. We need to choose compact displays that are familiar to the user in order to minimize the amount of effort taken in decoding the information.

In this section, we will design and implement a dashboard to monitor the performance of students in a class. We will define the purpose of the dashboard, gather the necessary datasets, choose the charts and graphics that we will use, and organize the information in sections dedicated to the students, courses, and the entire class.



The dashboard that monitors the performance of students in a class

As the main topic in this section is the design of dashboards and we have created several charts previously, we won't include the code for the examples. The code for the charts, each section, and the complete dashboard is available in the chapter09 directory of the code bundle.

Defining the purpose of the dashboard

The dashboard should be an overall view of the performance of the students in a class. If the scores of all the students in a given course are declining, there might be a problem with the methodology chosen by the teacher; if the scores of just one of the students are dropping, that student might be having personal issues that are interfering with his/her learning. The aim of the dashboard is to display all the information in order to easily detect problems and make decisions to help the students or teachers improve the learning process.

A teacher will want to detect drops in performance at three levels: an individual student, the students in a course, or the students in all the courses at the same time. Besides detecting learning problems, it would be useful to have information that helps identify possible causes of bad performance. The specific objectives of our dashboard will be as follows:

- Assess the performance of each student in a course. The most obvious way to do this is to display the scores of the students in each class. We might want to identify possible causes of bad performance, such as repeated absences.
- Monitor the aggregated performance of the students in each course. This will allow us to take action if a great number of students are having issues with a particular subject.
- Get an overall measure of the performance of the complete class. This will help us detect problems that could affect the class as a group.

We will need to gather the students' data and decide which information is relevant in order to achieve these objectives.

Obtaining the data

As we mentioned earlier, we need to monitor the performance of the students, courses, and the entire class. For this example, the data will be generated with a script. We will need the absences and scores of the students for each course. We will assume that we have a JSON endpoint that provides us with the students' data in the following format:

```
[  
  {  
    id: 369  
    name: 'Adam Lewis',  
    absences: [ ... ],  
    courses: [ ... ],  
    avgScore: 58.84  
  },  
  {  
    id: 372  
    name: 'Abigail Bower',  
    absences: [ ... ],  
    courses: [ ... ],  
    avgScore: 67.78  
  },  
  ...  
]
```

For each student, we will have the `name` and `id` attributes. We will also have an `absences` attribute, which will contain just a list of dates on which the student didn't show up to class:

```
{
  name: 'Adam Lewis',
  absences: [
    '2013-09-06',
    '2013-10-04',
    ...
  ],
  ...
}
```

We will also need the scores of the students in their courses. The `courses` field will contain a list of the students' courses, and each course will have the course's name and a list of the scores obtained by the student in the assignments or assessments. For convenience, we will also add the average score of the students for the current period:

```
{
  name: 'Adam Lewis',
  absences: [ ... ],
  courses: [
    {
      name: 'Mathematics',
      scores: [
        {date: '2013-09-23', score: 78},
        {date: '2013-10-04', score: 54},
        ...
      ]
    },
    {
      name: 'Art',
      scores: [...]
    }
  ],
  avgScore: 58.84
}
```

We will also need information about the courses of the students. We will assume that there is a JSON endpoint that provides us with information about the courses:

```
[
  {
    name: 'Mathematics',
    avgScores: [

```

```
{date: '2013-09-18', score: 72.34},  
 {date: '2013-10-07', score: 64.45},  
 ...  
 ],  
 avgScore: 63.21  
 },  
 {  
 name: 'Arts',  
 avgScores: [  
 {date: '2013-09-16', score: 76.62},  
 {date: '2013-10-01', score: 58.53},  
 ...  
 ],  
 avgScore: 63.21  
 }  
 ]
```

The `avgScores` attribute will contain a list with the dates of assessment and the average score obtained by the students. The `avgScore` attribute will contain the average of the scores for all the assessments in the current period.

Organizing the information

The next step is to organize the information in logical units. Each section of the dashboard should help us detect the issues that require attention.

In our example, the organization of the information is fairly direct; there will be sections for the students, courses, and the complete class.

The students section will help us detect the performance drop of individual students. We will consider that absences can be an explanatory factor in individual changes in the assessment scores.

The course section will help us monitor scores, thereby aggregating the scores of all the students at the same time. Drops in the scores of all the students at the same time could mean that the cause is a particular subject in the course or the teaching methodology.

The class section will allow us to monitor the average scores of the entire class, thereby averaging scores in each course for all the students. Here, the number of absences will be considered an important factor as well.

Creating the dashboard sections

In this section, we will discuss each dashboard section separately, explaining what information will be present in each section. We will also choose charts to represent these pieces of information. Later, we will decide how to organize the sections in the dashboard to make good use of the space and reflect the hierarchy of the information presented in each area.

The students section

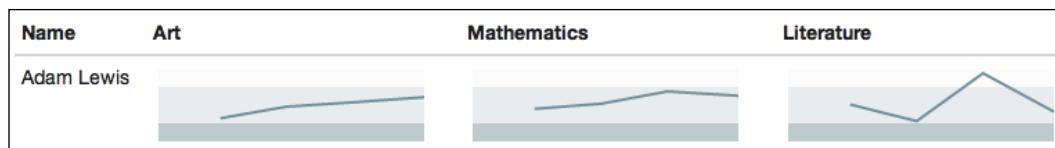
The students section of the dashboard will display the students in a table, displaying information about each student in rows. The most relevant information will be the scores, but we will also include the absences from class, because this could be a possible explanation of changes in performance.

The absences will be displayed using a barcode chart that is similar to the one presented in *Chapter 2, Reusable Charts*. This version will be smaller and have a background. This chart will help teachers know how many students didn't show up in class and when, whether the absences are concentrated in a certain period, or whether they are evenly distributed.

Name	Absences
Adam Lewis	

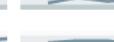
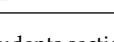
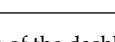
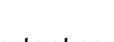
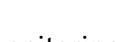
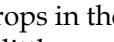
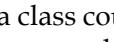
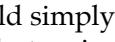
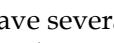
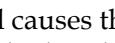
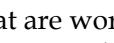
Absences for given students, displayed as a barcode chart

We will add a column for each course and display the scores of the students in the course assignments as a line chart. The scales of the chart will be implicit; they will always cover from 0 to 100 percent in the y axis and the current period in the x axis. The background of the chart will highlight score ranges that are considered important; the area below 25 percent has a different background, and the area between 25 percent and 75 percent has a different background.



Scores of a student in different classes. The background of the chart highlights areas of poor and high performance.

Finally, we will include the average score of the students in the current period. The complete students section will gather all these elements.

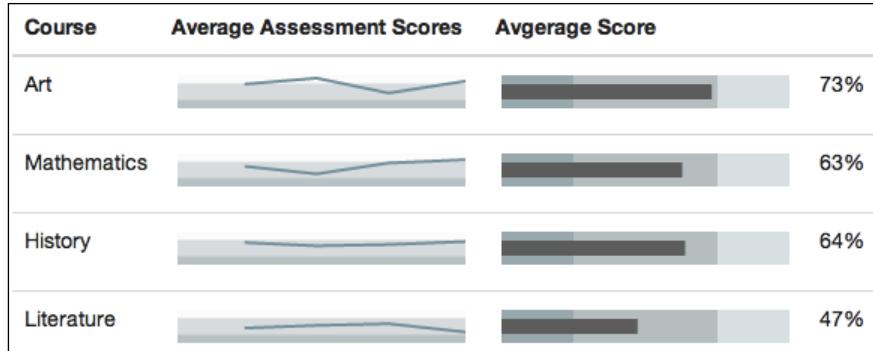
Name	Absences	Art	Math	History	Literature	Avg
Adam Lewis						60%
Boris Mackenzie						62%
Sam McDonald						61%
James Tucker						60%
Abigail Bower						61%
Robert Hodges						54%
Amanda Johnston						55%
Edward Roberts						55%
Amanda Bond						60%
Joshua Hart						55%
Lillian Rutherford						54%
Neil Knox						59%
Lily Dowd						59%
Luke Carr						61%
Anne Thomson						60%

Students section of the dashboard

The courses section

Monitoring the performance of the courses is as important as monitoring the scores of individual students. Small drops in the scores for a class could simply mean that the concepts being taught are a little more difficult than usual, but an important drop in the average score can have several causes that are worth investigating. For instance, it could be interesting to know whether there are other courses with similar behavior, as they could be interfering with each other due to difficult assignments on the same dates, for instance.

The courses section of the dashboard will show the user the evolution of the scores of each course and the average score of each course. To display the average score, we will use bullet charts, which are a compact display that shows us how actual measures compare with target values. Bullet charts are used to show us the value of an indicator, adding backgrounds to give context to the indicator's value. In this case, we will define regions of poor, regular, and good performance, and use the bullet chart to know the average score of each class based on these regions. The following figure illustrates a bullet chart:



Average scores of the students in each course

The class section

In the class context, we need to know whether there are relevant changes in average scores of the entire class. We will monitor weekly performance metrics so the teachers can detect problems before they become too difficult to solve.

We will display the average score of the students in all the courses. The absences for each week will also be included in the dashboard, so the user can quickly work out whether scores and absences are related for a particular student. We will list the date of the Monday of each week, displaying the average score for that week as a bullet chart. We will also include the average score as a number and the number of absences in the week as well.

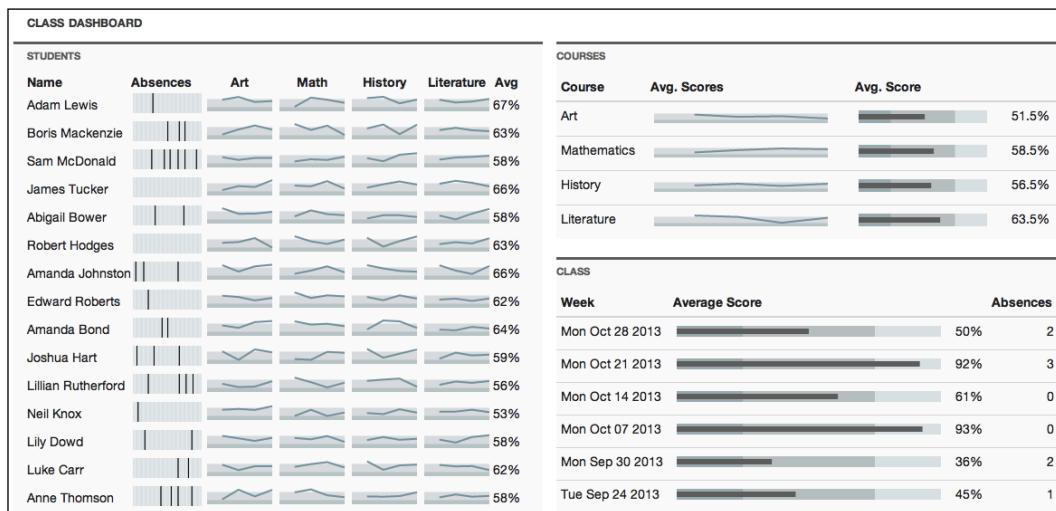
Week	Average Score	Absences
Mon Sep 02 2013		67% 1
Mon Sep 09 2013		65% 3
Mon Sep 16 2013		36% 3
Tue Sep 24 2013		45% 2
Mon Sep 30 2013		87% 3
Mon Oct 07 2013		60% 0
Mon Oct 14 2013		34% 3
Mon Oct 21 2013		56% 2
Mon Oct 28 2013		56% 2

Weekly average scores for all the courses and students in the class

Gathering the dashboard sections

The last step is to gather all the sections in one screen. The final layout of the dashboard will depend on the relative importance of the sections. In this case, we will organize them from the most granular to the more general, giving more space to the students section. The rationale behind this is that most of the time, the performance problems will be at an individual level and less frequently at the level of a course or class.

We will render each section inside a `div` element and assign it the `section` class. We will add styles to help us differentiate the sections and make logical groups more evident. We will add a light grey background and add a small border on top of each section. We also added a small title to give it an additional context.



The completed dashboard. The sections are delimited with a light grey background.

In this example, the title of the dashboard is neither useful nor informative. In a real-world dashboard, the area of the title would be a good place to add navigation menus and links to other dashboard sections.

Summary

In this chapter, we described the characteristics of good dashboards and discussed the process behind the design and implementation of dashboards. We learned good practices to create effective dashboards. We also created an example dashboard to monitor the performance of the students in a class, including sections to assess the scores of individual students, courses, and classes.

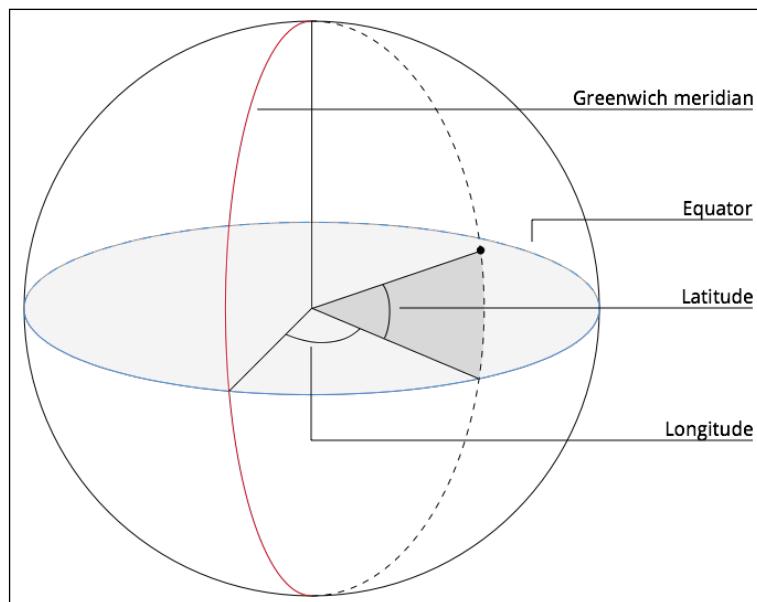
In the next chapter, we will learn how to use GeoJSON and TopoJSON files to create maps with D3. We will learn about projections, how to use maps to display data, and how to integrate D3 with Mapbox.

10

Creating Maps

Maps are a 2D representation of the relevant features of places. Which features are relevant will depend on the purpose of the map; a map for a zoo will show the entrances, thematic areas, gift store, and where each animal is. In this case, there is no need for the sizes and distances to be precise. In a geologic map, we will need accurate distances and representations of the rock units and geologic strata.

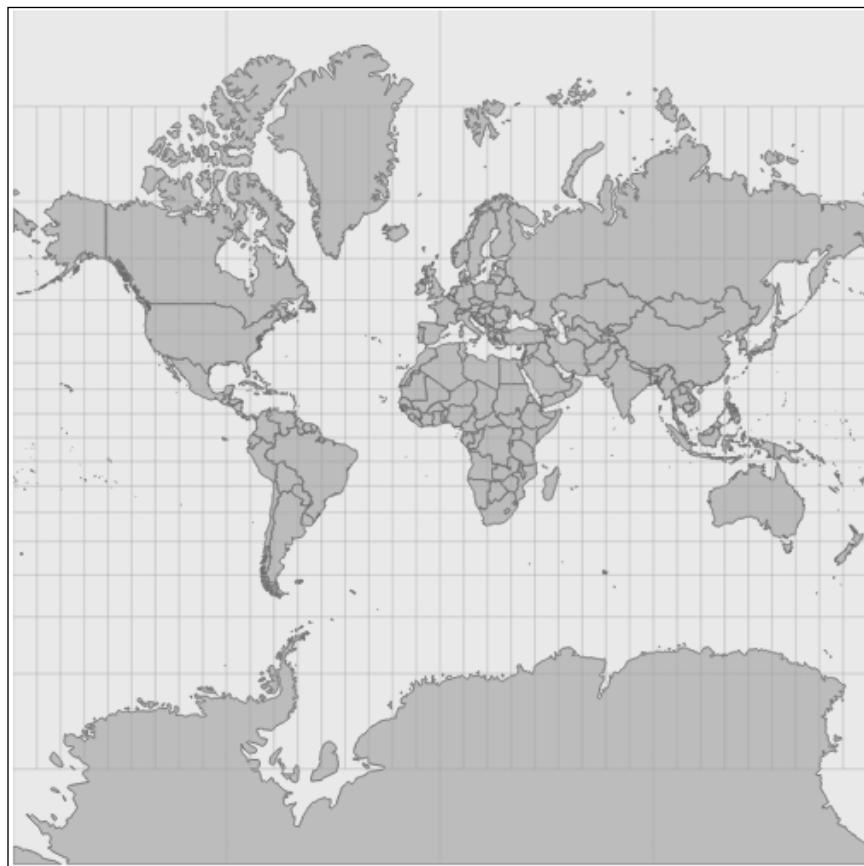
Positions on the surface of the earth are described by two coordinates, longitude and latitude. The longitude of a point is the angle between the point and the Greenwich meridian, and the latitude is the angle between the point and the equator. The latitude and longitude are the angles measured with respect to the equator and the Greenwich meridian, as shown in the following diagram:



The longitude can take values between -180 and 180, while the latitude can have values between 90 (the North Pole) and -90 (the South Pole).

Shapes on the earth can be described by listing the coordinates of the points in their boundary in order. We can, for instance, describe an island by listing the coordinates of points on the coastline separated by one kilometer in a clockwise order. This representation won't be perfectly accurate, because it will not represent the irregularities in the coastline that are smaller than one kilometer in size, but this will be useful if we only need to have an idea of the shape of the island.

To create a map of a feature, we need to translate the coordinates that describe the feature to points in a 2D surface. The functions that perform this translation are called **projections**. As projections intend to represent a 3D surface in a 2D medium, distortions will be introduced as follows:



The Mercator projection severely distorts the areas near the poles

Each projection has been created in order to minimize distortions of some kind. Some projections will represent relative directions accurately but not the area of certain regions; others will do the exact opposite. When using maps, it's important to know what kinds of distortions are acceptable in each case.

In this chapter, we will learn how to create map-based charts with D3. We will learn how to obtain, transform, and use geographic data in GeoJSON and TopoJSON formats to create SVG-based maps. We will create a map to visualize the distortions introduced by the Mercator projection, coloring each country by its area. Maps in which regions are colored by a characteristic of the regions (population, income, and so on) are called **choropleths**. In this section, we will implement a choropleth map using D3 and GeoJSON. We will also learn how to use TopoJSON to create maps with more compact geographic data files, and how to use TopoJSON to display topologic information, such as the connection between features and boundaries. Lastly, we will learn how to integrate D3 with Mapbox, an excellent map provider.

Obtaining geographic data

To create a map, we will need files that describe the coordinates of the features that we intend to include in our map. One of the most reliable sources for medium-scale geographic data is Natural Earth (<http://www.naturalearthdata.com>), a collaborative effort to curate and organize geographic datasets.

The geographic datasets available at Natural Earth are in the public domain and are available at 1:10,000, 1:50,000, and 1:110,0000 scales. There are vector and raster datasets, and the map files are classified in three categories:

- **Cultural:** This contains countries, administrative divisions, states and provinces, populated places, roads, urban areas, and parks
- **Physical:** This describes coastlines, land, islands, oceans, rivers, lakes, and glaciated areas among others
- **Raster:** This contains images, depicting the relief as shades and with colors based on climate

The files are in the **ESRI shapefile** format, the de facto standard for geographic data. ESRIshape files represent the geometry of features as sets of points, lines, and polygons. The files might also contain additional attributes about the features, such as the name of the place, population in the last census, or the average income of the population living in that area. A shapefile is a set of several files, which must include the following three files:

- .shp: Shape format, the geometry of the feature
- .shx: An index to locate the features in the .shp file
- .dbf: The feature attributes in the dBase IV format

The shapefile set can also contain optional files, such as the .prj file, which contains information of the coordinate system and projection in which the vector data is stored.

Shapefiles are widely used in geographic information systems; however, the format is not suited for its use in web platforms. We will transform the shapefiles to JSON-based formats and use D3 to create maps from them.

Understanding the GeoJSON and TopoJSON formats

The most widely used formats to create maps with D3 are the GeoJSON and TopoJSON formats. In this section, we will describe both the formats briefly and learn how to transform ESRIshapefiles to GeoJSON or TopoJSON.

The **GeoJSON** format encodes geometries, features, or collections of features using the JSON format. We will describe the central aspects of the GeoJSON format; the complete specification is available at <http://geojson.org/>.

A GeoJSON file always contains one top-level object. The GeoJSON object must have a type attribute; for geometries, the type can be Point, MultiPoint, LineString, MultiLineString, Polygon, or MultiPolygon. For a collection of geometries, features, and collections of features, the type will be GeometryCollection, Feature, or FeatureCollection, respectively. One GeoJSON file can contain a small island or a collection of countries nested in the top-level object.

The GeoJSON objects that represent geometries must have a coordinates attribute, whose contents will depend on the type of geometry. For a Point attribute, the coordinates attribute will be an array of two elements, representing a position:

```
{"type": "Point", "coordinates": [10.0, 10.0]}
```

A `LineString` object represents a line, and its coordinates array contains a pair of locations (the longitude and latitude of a place):

```
{
  "type": "LineString",
  "coordinates": [[10.0, 10.0], [10.0, 0.0]]
}
```

A `Polygon` object is more complex. It's intended to represent a polygon that can have holes on it. Polygons are represented by first describing the exterior boundary and then the boundaries of the holes on it. A `Polygon` object with one hole could be described with the following GeoJSON object:

```
{
  "type": "Polygon",
  "coordinates": [
    [[0, 0], [0, 10], [10, 10], [0, 10], [10, 0]],
    [[2, 2], [8, 2], [8, 8], [8, 2], [2, 2]]
  ]
}
```

Here, the exterior ring describes a square of size 10, and the hole on it is a centered square of size 6. Also, there are geometries to describe the collections of the mentioned geometries; a `MultiPoint` object will contain an array of points, a `MultiLineString` object will contain an array with pairs of points, and a `MultiPolygon` object will contain an array of polygons.

A `Feature` object must have a `geometry` attribute that will contain a geometry object, usually `Polygon` or `MultiPolygon`. It may have a `properties` member that can be used to store nongeographic data about the features, such as the name of the place, population, or average income. For instance, the following feature object describes the country of Aruba; it contains about a dozen properties, and the polygon contains 26 points:

```
{
  "type": "Feature",
  "properties": {
    ...
    "type": "Country",
    "admin": "Aruba",
    "adm0_a3": "ABW",
    ...
  },
  "geometry": {
    "type": "Polygon",
```

```
"coordinates": [
  [
    [
      [-69.899121, 12.452001],
      [-69.895703, 12.422998],
      ...
    ]
  ]
}
```

A `FeatureCollection` object will contain a `features` array, which will contain feature objects.

As mentioned earlier, GeoJSON files describe features in terms of their geometry, and in some cases, they are highly redundant. If two features share a boundary, the common boundary coordinates will appear twice, once for each feature. Moreover, the description of the boundary might not match exactly, generating artificial gaps between the shapes. These shortcomings are addressed by the TopoJSON format created by Mike Bostock, the creator of D3.

TopoJSON objects encode topologies instead of geometries, that is, describing the relationship between points, lines, and shapes. In a TopoJSON object, shapes are described as sequences of `arcs`, which are essentially boundary segments. Each arc is defined once, but it can be referenced several times by the shapes that share that arc. This removes redundancy, making TopoJSON lighter than their GeoJSON counterparts. The coordinates in TopoJSON files are encoded in a more efficient way, making TopoJSON even more compact.

The following TopoJSON object represents the country of Aruba. The object contains an array of `arcs`, a `transform` attribute with information on how to decode the coordinates of the arcs to longitude and latitude, and the `objects` member, which contains the description of the features:

```
{
  "type": "Topology",
  "objects": {
    "aruba": {
      "type": "GeometryCollection",
      "geometries": [
        {

```

```
        "type": "Polygon",
        "arcs": [[ [0] ]
      }
    ]
  }
},
"arcs": [
  [
    [9798,    1517],
    [ 201, -1517],
    [-2728,    812],
    ...
  ]
],
"transform": {
  "scale": [
    0.000017042719896989698,
    0.00001911323944894461
  ],
  "translate": [
    -70.06611328125,
    12.422998046874994
  ]
}
}
```

The `objects` member can contain one or more objects describing geometry, but instead of listing the coordinates of the geometry object, the `arcs` array contains a list of references to the arcs defined at the top level of the TopoJSON object. If several features share a boundary, they will reference the same arc.

Besides the TopoJSON format, Mike Bostock created utilities to manipulate TopoJSON and GeoJSON files. The TopoJSON program has two components: the command-line program and the client-side library. The command-line program allows you to convert shapefiles, CSV or GeoJSON formats, to TopoJSON. It has options to simplify the features, combine several files in one output, and add or remove properties from the original features. The JavaScript library allows you to parse the TopoJSON files and construct Feature objects. The complete specification of the format and the programs is available at <https://github.com/mbostock/topojson>.

Transforming and manipulating the files

We might need to manipulate the geographic data files in several ways. We might want to reduce the level of detail of our features to have smaller files and simpler features, include additional metadata not present in the original files, or even filter some features.

To convert the files from one format to another, we will need to install the **Geospatial Data Abstraction Library (GDAL)**. GDAL provides command-line tools to manipulate and convert geographic data between different formats, shapefiles and GeoJSON among them. There are binaries available for Windows, Mac, and Linux systems on their site, <http://www.gdal.org/>.

Depending on how we want our files, our workflow will include several steps that we might need to redo later. It's a good idea to automate this process in a way that allows us to understand why and how we transformed the files. One way to do this is to use the **make** program or a similar system. We will use *make* to download and transform the geographic datasets that we need for this chapter.

For the first chart, we will need to download the cultural vectors from Natural Earth. In a world map, we don't need the most detailed level; we will download medium-scale data. Then, we will uncompress the shapefiles and use the **ogr2ogr** program to transform the shapefiles to GeoJSON.

We will implement these transformations using *make*. The **Makefile** is in the `chapter10/data` directory of the code bundle. Each step can be done individually in a terminal provided that it is performed in the correct order. In a **Makefile**, we describe each step in the transformation process as a target, which can have zero or more **dependencies**. To generate the target file from the dependencies, we need to perform one or more commands. For instance, to generate the `ne_50m_admin_0_countries.shp` target file, we need to uncompress the `ne_50m_admin_0_countries.zip` file. This file is a dependency of the target file, because the target can't be generated if the ZIP file doesn't exist. The command to generate the shapefile is `unzip ne_50m_admin_0_countries.zip`. The following **Makefile** will generate a GeoJSON file that contains all the countries:

```
# Variables
ADMIN0_URL = http://.../ne_50m_admin_0_countries.zip

# Targets

# Download the Compressed Shapefiles
ne_50m_admin_0_countries.zip:
```

```
curl -LO $(ADMIN0_URL)

# Uncompress the Shapefiles
ne_50m_admin_0_countries.shp: ne_50m_admin_0_countries.zip
  unzip ne_50m_admin_0_countries.zip
  touch ne_50m_admin_0_countries.shp

# Convert the shapefiles to GeoJSON
countries.geojson: ne_50m_admin_0_countries.shp
  ogr2ogr -f GeoJSON countries.geojson ne_50m_admin_0_countries.shp
```

The Makefiles manage the dependencies between targets; building the `countries.geojson` file will check the dependency chain and run the target commands in the correct order. To generate the GeoJSON file in one step, run the following command:

```
$ make countries.geojson
```

It will download, uncompress, and transform the shapefiles to GeoJSON. It will only perform the commands to generate the files that are not present in the current directory; it won't download the ZIP file again if the file is already present.

We can use the GeoJSON file to create a TopoJSON version of the same file. By default, `topojson` will strip all the properties of the original file. We can preserve the properties by using the `-p` name option:

```
$ topojson -o countries.topojson -p admin -p continent
  countries.geojson
```

Note that the file size of the GeoJSON file is about 4.4 M; the TopoJSON file is only 580 K. We will begin by using the `countries.geojson` file to create our first maps, and then learn how to use the TopoJSON files to create maps. We will also include this command in the Makefile to be able to replicate this conversion easily.

Creating maps with D3

In this section, we will create map-charts based on SVG. We will use the GeoJSON file with the countries to create a choropleth map that shows the distortions introduced by the Mercator projection.

We will also create maps using the more compact format, TopoJSON, and use topologic information contained in the file to find the neighbors and specific frontiers between countries.

Creating a choropleth map

In this section, we will create a choropleth map to compare the areas of different countries. We will paint each country according to its area; countries with greater areas will be colored with darker colors. In general, the Mercator projection is not suitable to create choropleth maps showing large areas, as this projection shows the regions near the poles bigger than they really are. For instance, Antarctica is smaller than Russia, but using the Mercator projection, it seems bigger. Brazil has a greater area than Greenland, but with this projection, it looks smaller.

In this example, we will use the Mercator projection to show this effect. Our choropleth map will allow us to compare the size of the countries. We will use the GeoJSON file, `chapter10/data/countries.geojson`, available in the code bundle. The `chapter10/01-countries-geojson.html` file displays the contents of the file for a more convenient inspection of the features and their attributes.

We will begin by reading the contents of the GeoJSON file and creating the SVG element to display the map. GeoJSON is encoded in the JSON format, so we can use the `d3.json` method to retrieve and parse the content from GeoJSON:

```
d3.json(geoJsonUrl, function(error, data) {  
  
    // Handle errors getting or parsing the GeoJSON file  
    if (error) { return error; }  
  
    // Create the SVG container selection  
    var div = d3.select('#map01'),  
        svg = div.selectAll('svg').data([data]);  
  
    // Create the SVG element on enter  
    svg.enter().append('svg')  
        .attr('width', width)  
        .attr('height', height);  
});
```

The data variable contains the GeoJSON object. In this case, the GeoJSON object contains a `FeatureCollection` object, and the `features` array contains `Feature` objects, one for each country.

To map the feature coordinates, we will need a projection function. D3 includes about a dozen of the most used projections, and there are even more available as plugins (see <https://github.com/d3/d3-geo-projection> for the complete list). We will create an instance of the Mercator projection and translate it so that the point with the coordinates [0, 0] lies in the center of the SVG figure:

```
// Create an instance of the Mercator projection
var projection = d3.geo.mercator()
    .translate([width / 2, height / 2]);
```

With this function, we can compute the SVG coordinates of any point on earth. For instance, we can compute the SVG coordinates of a point in the coast of Aruba by invoking the projection function with the [longitude, latitude] array as an argument:

```
projection([-69.899121, 12.452001])
// [17.004529145013294, 167.1410458329102]
```

We have the geometric description of each feature; the geometries contain arrays of coordinates. We could use these arrays to compute the projection of each point and draw the shapes using the `d3.svg.path` generator, but this will involve interpreting the geometries of the features. Fortunately, D3 includes a geographic path generator that does the work for us:

```
// Create the path generator and configure its projection
var pathGenerator = d3.geo.path()
    .projection(projection);
```

The `d3.geo.path` generator needs the projection to compute the paths. Now, we can create the path objects that will represent our features:

```
// Create a selection for the countries
var features = svg.selectAll('path.feature')
    .data(data.features);

// Append the paths on enter
features.enter().append('path')
    .attr('class', 'feature');

// Set the path of the countries
features.attr('d', pathGenerator);
```

We have added the class feature to each path in order to configure its style using CSS. After including the `chapter10/map.css` style sheet file, our map will look similar what is shown in to the following figure:



A map of the world countries, using the Mercator projection with the default scale

We can see that the features are correctly drawn, but we would like to add colors for the ocean and scale the map to show all the countries. Projections have a `scale` method that allows you to set the projection's scale. Note that different projections interpret the scale in different ways. In the case of the Mercator projection, we can map the entire world by setting the scale as the ratio between the figure width and the complete angle, in radians:

```
// The width will cover the complete circumference
var scale = width / (2 * Math.PI);

// Create and center the projection
var projection = d3.geo.mercator()
  .scale(scale)
  .translate([width / 2, height / 2]);
```

We will also want to add a background to represent the oceans. We could simply add a SVG rectangle before inserting the features; instead, we will create a feature object that spans the globe and uses the path generator to create an SVG path. This is a better approach because if we change the projection later, the background will still cover the complete globe. Note that we need to close the polygon by adding the first point in the last position:

```
var globeFeature = {
  type: 'Feature',
  geometry: {
    type: 'Polygon',
    coordinates: [
      [
        [
          [
            [-179.999, 89.999],
            [179.999, 89.999],
            [179.999, -89.999],
            [-179.999, -89.999],
            [-179.999, 89.999]
          ]
        ]
      ]
    ]
  }
};
```

To avoid overlapping, the rectangle defined by the coordinates doesn't completely cover the globe. We can now create the path for the globe and add a style to it, as we did for the rest of the features:

```
// Create a selection for the globe
var globe = svg.selectAll('path.globe')
  .data([globeFeature]);

// Append the graticule paths on enter
globe.enter().append('path')
  .attr('class', 'globe');

// Set the path data using the path generator
globe.attr('d', pathGenerator);
```

We will also add reference lines for the meridians and parallels. These lines are called as graticules, and D3 includes a generator that returns a `MultiLineString` object with the description of the lines:

```
// Create the graticule feature generator
var graticule = d3.geo.graticule();

// Create a selection for the graticule
```

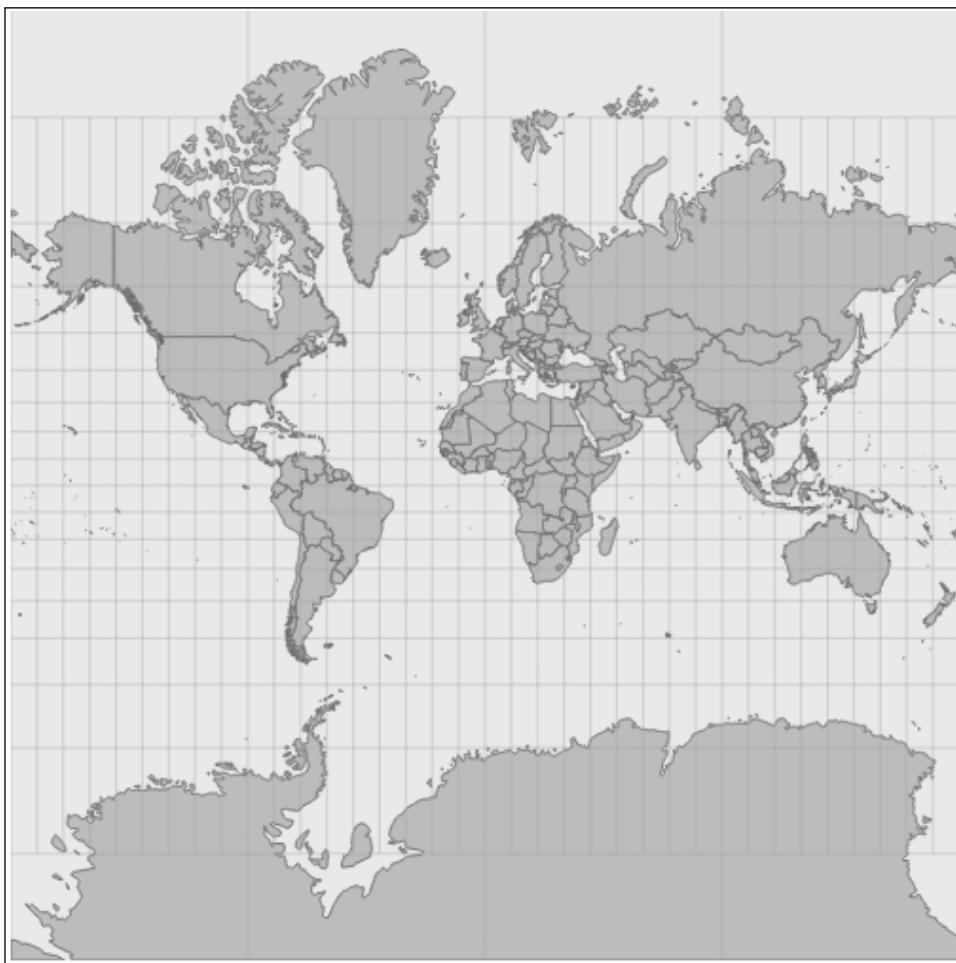
Creating Maps

```
var grid = svg.selectAll('path.graticule')
  .data([graticule()])

// Append the graticule paths on enter
grid.enter().append('path')
  .attr('class', 'graticule');

// Set the path attribute for the graticule
grid.attr('d', pathGenerator);
```

We have also added a class to the graticule lines to apply styles using CSS. The map now looks better:



The map with the oceans and the graticule

When creating a choropleth map, one of the most important choices to be made is the color scale to be used. Color Brewer (<http://colorbrewer2.org/>) is an online tool that helps developers and designers to choose good color scales for their maps. There are color scales for qualitative dimensions as well as for sequential and diverging quantitative dimensions. The colors of each palette have been carefully chosen so that there is a good contrast between the colors and they look good on the screen.

For a map that shows a qualitative dimension, the color scale should be composed of colors that differ primarily in hue but have a similar brightness and saturation. An example of this could be a map showing which languages are spoken in each country, as shown in the following figure:



A qualitative color scale doesn't suggest order between the items

For quantitative variables that are sequential, that is, ranging from less to more in one dimension, a color scale with increasing darkness will be a good choice. For instance, to display differences in income or housing costs, a sequential scale would be a good fit, as shown in the following figure:



A sequential color scale is a good choice for ordinal variables

For a quantitative variable that covers two extremes, the color palette should emphasize the extremes with dark colors of different hue and show the critical mid-range values with lighter colors. An example of this could be a map that shows the average winter temperatures, showing the countries with temperatures below zero in blue, temperatures above zero in red, and the zero value in white, as shown in the following figure:



A diverging color scale is useful to show variables that cover extremes

In our case, we will use a sequential scale because our quantitative variable is the area of each country. Countries with a bigger surface will be shown in a darker color. We will use Color Brewer to generate a sequential palette:

```
var colorRange = [
  '#f7fcfd',
  '#e0ecf4',
  '#bfd3e6',
  '#9ebcda',
  '#8c96c6',
  '#8c6bb1',
  '#88419d',
  '#6e016b'];
```

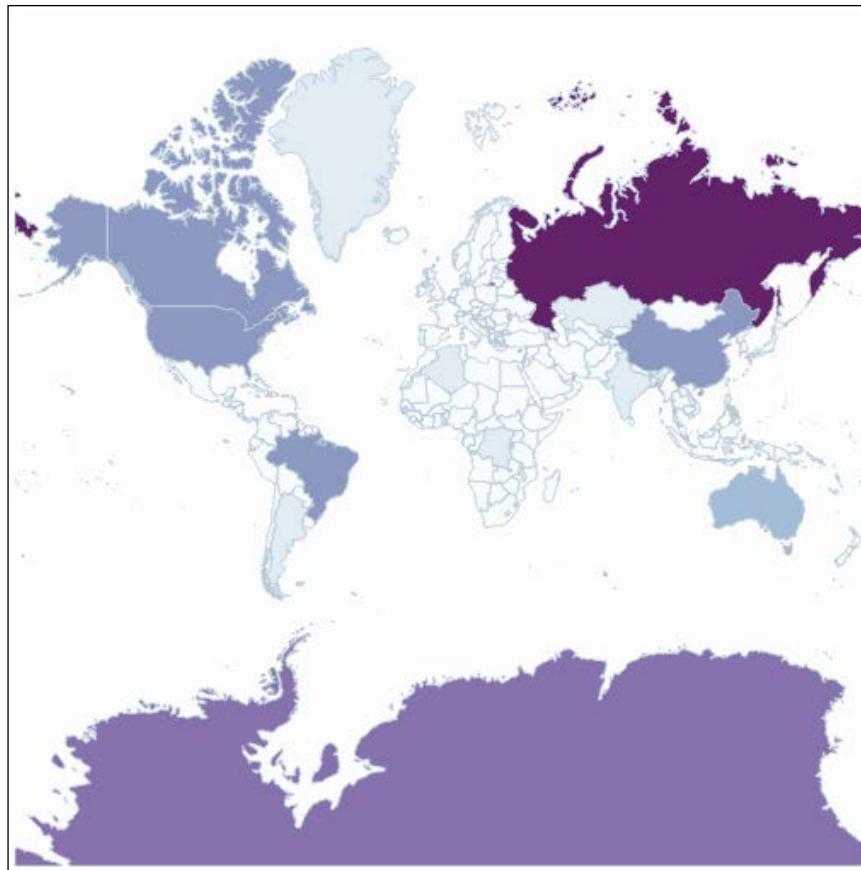
This color palette has eight levels, ranging from almost white to dark purple. The `d3.geo.area` method computes the area of a feature in steradians, which is the measurement unit for solid angles. We will use the `d3.scale.quantize` scale to assign the area of each feature to one of the colors of our palette:

```
// Create the color scale for the area of the features
var colorScale = d3.scale.quantize()
  .domain(d3.extent(data.features, d3.geo.area))
  .range(colorRange);
```

We use the `d3.geo.area` method to compute the area of each feature and to compute the fill color using the color scale:

```
// Set the path of the countries
features.attr('d', pathGenerator)
  .attr('fill', function(d) {
    return colorScale(d3.geo.area(d));
});
```

We obtain a map with each country colored according to its area, as shown in the following figure:



The choropleth that shows the area of each country

When we take a look at the choropleth, we can see several inconsistencies between the size and the colors. Greenland, for instance, looks twice as big as Brazil, but it's actually smaller than Brazil, Australia, and the United States.

Mapping topology

As mentioned in the previous section, TopoJSON files are more compact than their GeoJSON counterparts, but the real power of TopoJSON is that it encodes topology, that is, information about connectedness and the boundaries of the geometries it describes. Topology gives us access to more information than just the shapes of each feature; we can identify the neighbors and boundaries between geometries.

In this section, we will use the `countries.topojson` file to create a world map, replacing GeoJSON from the previous section. We will also identify and map the neighboring countries of Bolivia and identify a particular frontier using the TopoJSON library. As we did in the previous section, we have created the `chapter10/03-countries-topojson.html` file to display the contents of the TopoJSON file for easier inspection.

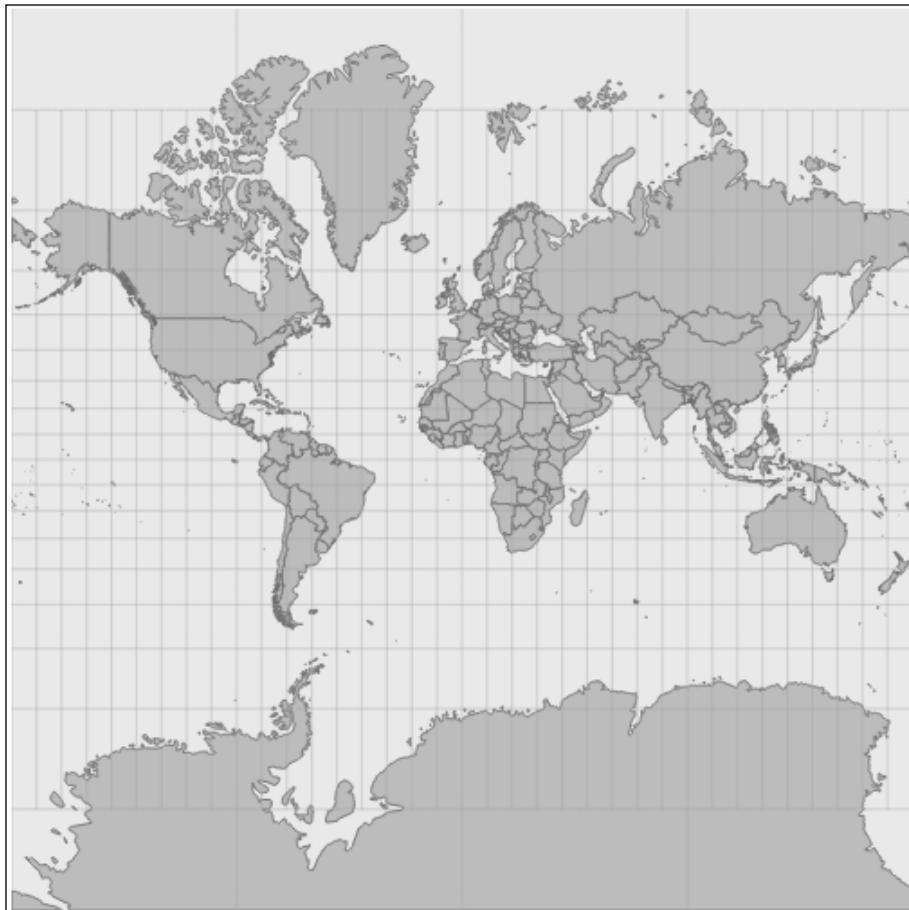
To create a world map with TopoJSON, we need to create the SVG container and the projection in the same way that we did with the GeoJSON file. The geographic path generator expects GeoJSON objects, but at this point, we have a TopoJSON object that encodes the geometry of our features. The `topojson.feature` object computes the GeoJSON Feature or FeatureCollection object, corresponding to the object given as the second argument. Remember that the `object` attribute contains the TopoJSON geometry objects, which have an array of references to the arcs defined at the top level of the TopoJSON object. In this case, the `geodata` variable stores a FeatureCollection object, which we can use to generate the shapes using the same code as that used earlier:

```
d3.json(url, function(error, data) {  
  
    // Create the SVG container...  
  
    // Construct the Feature Collection  
    var geodata = topojson.feature(data, data.objects.countries);  
  
    // Render the features  
});
```

The `geodata` object is a FeatureCollection object, and we can use the same projection and path generator that we used in the last section to generate the world map:

```
// Create a selection for the countries and bind the feature data  
var features = svg.selectAll('path.feature')  
    .data(geodata.features)  
    .enter()  
    .append('path')  
    .attr('class', 'feature')  
    .attr('d', pathGenerator);
```

This allows us to replace the GeoJSON file that we used in the previous section with the TopoJSON file, which is about one-eighth the size of the `countries.geojson` file and obtains the same result, as shown in the following figure:



The world map using TopoJSON

As TopoJSON files describe geometries by listing references to the arcs, we can verify that two geometries are neighbors by checking whether they have arcs in common. The `topojson.neighbors` method does exactly this, given an array of geometries; it returns an array with the indices of the neighbors of each geometry object.

To illustrate this point, we will create a map that highlights the countries that share a boundary with Bolivia. We will begin by scaling and centering the map to display South America. We begin by filtering the countries that belong to South America and creating a `FeatureCollection` object for them:

```
// Construct the FeatureCollection object
var geodata = topojson.feature(data, data.objects.countries);

// Filter the countries in South America
var southAmerica = geodata.features.filter(function(d) {
    return (d.properties.continent === 'South America');
});

// Create a feature collection for South America
var southAmericaFeature = {
    type: 'FeatureCollection',
    features: southAmerica
};
```

We would like to adapt the scale and translation of the projection to just display South America. D3 provides tools to compute the bounding box and centroid of a feature:

```
// Compute the bounds, centroid, and extent of South America
// to configure the projection
var bounds = d3.geo.bounds(southAmericaFeature),
    center = d3.geo.centroid(southAmericaFeature);
```

The `d3.geo.bounds` method returns the bottom-left and top-right corners of the feature's bounding box in geographic coordinates. The `d3.geo.centroid` method returns an array with the longitude and latitude of the centroid of the feature.

To compute a scale factor that displays our feature object properly, we need to know the angular distance between the two corners of our bounding box. We could also use another characteristic distance of the feature, such as the distance from the centroid to one of the bounding box corners:

```
// Compute the angular distance between bound corners
var distance = d3.geo.distance(bounds[0], bounds[1]);
```

The `d3.geo.distance` method takes two locations and returns the angular distance between the points (in radians). For the Mercator projection, we can compute the scale as the ratio between the desired screen size and the angular span of our feature. We can recompute the scale with the angular distance between the corners of the bounding box:

```
// The width will cover the complete circumference  
var scale = width / distance;
```

We can now center and scale our projection. This will show our feature centered in the screen and at a better scale:

```
// Create and scale the projection  
var projection = d3.geo.mercator()  
  .scale(scale)  
  .translate([width / 2, height / 2])  
  .center(center);
```

Note that this way of computing the scale will only work with the Mercator projection. The scales are not consistent among projections, as shown in the following figure:



Centering and scaling a map around a feature

Having centered and scaled the South American continent properly, we can proceed to compute the neighbors of Bolivia. We begin by obtaining the neighbors of each country in our dataset:

```
// Compute the neighbors of each geometry object.  
var neighbors =  
    topojson.neighbors(data.objects.countries.geometries);
```

This will return an array with as many elements as the input array. Each element will be an array with the indices of the neighbor geometries of each object. The geometry of Bolivia is described by the thirtieth element in the `data.objects.countries.geometries` array. The contents of `neighbors[30]` is the `[8, 31, 39, 169, 177]` array; each element is the index of the geometry element of each neighboring country. To find Bolivia's neighbors, we need to know the index of Bolivia, so, we will search for it in the `geometries` array:

```
// Find the index of Bolivia in the geometries array  
var countryIndex = 0;  
data.objects.countries.geometries.forEach(function(d, i) {  
    if (d.properties.admin === 'Bolivia') {  
        countryIndex = i;  
    }  
});
```

The `neighbors[countryIndex]` array will contain the indices of the neighbors of Bolivia. To display the neighbor's features in the map, we need to compute a feature that contains their geometries. We can create a `GeometryCollection` object in order to use the `topojson.feature` method to construct the `FeatureCollection` object with the neighbors:

```
// Construct a Geometry Collection with the neighbors  
var geomCollection = {  
    type: 'GeometryCollection',  
    geometries: []  
};  
  
// Add the neighbor's geometry object to the geometry collection  
neighbors[countryIndex].forEach(function(i) {  
    var geom = data.objects.countries.geometries[i];  
    geomCollection.geometries.push(geom);  
});
```

The geometry collection object we just created contains the geometries of the countries that share boundaries with Bolivia. We will create a feature collection object for these geometries in order to add them to the map:

```
// Construct a Feature object for the neighbors
var neighborFeature = topojson.feature(data, geomCollection);
```

We can now create the path for the feature containing the neighbors, and add a class to the path to set its style with CSS:

```
// Add paths for the neighbor countries
var neighborPaths = svg.selectAll('path.neighbor')
  .data([neighborFeature])
  .enter()
  .append('path')
  .attr('class', 'neighbor')
  .attr('d', pathGenerator);
```

The countries that share a boundary with Bolivia are highlighted in the following figure:



The TopoJSON file gives us more information. Let's say that we need to show the frontier between Bolivia and Brazil. We know that this frontier is identifiable, because we could inspect the geometries for both countries and select the arcs that are common to both geometries, but there is an easier way. The `topojson.mesh` method returns a GeoJSON `MultiLineString` geometry that represents the mesh for a given object and geometry. It has an optional argument to filter the arcs that meet a condition. We will use this method to generate a `MultiLineString` object, representing the boundary between Brazil and Bolivia:

```
// Compute the mesh of the boundary between Brazil and Bolivia
var frontier = topojson.mesh(data, data.objects.countries, function(a,
b) {
    return ((a.properties.admin === 'Brazil') && (b.properties.admin
=== 'Bolivia')) ||
    ((a.properties.admin === 'Bolivia') && (b.properties.admin
=== 'Brazil'));
});
```

Note that the optional filter receives two TopoJSON geometries, not GeoJSON features. To obtain the frontier between Brazil and Bolivia, we need to select the arcs that are shared by both the countries. We can now create a path for the frontier and add it to our map:

```
// Add the frontier to the SVG element
var frontierPath = svg.selectAll('path.frontier')
    .data([frontier])
    .enter()
    .append('path')
    .attr('class', 'frontier')
    .attr('d', pathGenerator);
```

The boundary between the two countries is highlighted in the following updated map:



Note that the `topojson.mesh` method can be used to identify frontiers of any kind. In other datasets, this can be useful to show or hide internal frontiers or frontiers with countries that meet certain conditions.

In this section, we learned how to use GeoJSON to create SVG-based maps. We have also learned how to use TopoJSON files to reconstruct GeoJSON objects and create a map. We've also learned how to create maps that highlight topologic relations between places, such as highlighting countries that are connected to each other and show specific boundaries between features.

Using Mapbox and D3

SVG-based maps are great for data visualization projects, but sometimes, we will need more advanced features in our maps, such as a feature that allows us to search for an address or location, get information at the street level, or show satellite images. The most convenient way to provide these features is to integrate our visualization with map providers such as Google Maps, Yahoo! Maps, or Mapbox. In this section, we will learn how to integrate D3 with Mapbox, an excellent map provider.

Mapbox is an online platform used to create custom-designed maps for web and mobile applications. It provides street maps and terrain and satellite view tiles. The street maps from Mapbox use data from OpenStreetMap, a community-powered open data repository with frequent updates and accurate information.

A distinctive feature of Mapbox is that it allows customization of the map views. Users can customize the visual aspects of every feature in their maps. The web platform has tools to customize the maps, and the desktop tool, TileMill, makes this customization even easier.

To follow the examples in this section, you will need a Mapbox account. The free plan allows you to create maps, add markers and features, and get up to 3,000 views per month. To create a Mapbox account, visit <https://www.mapbox.com>.



Mapbox counts the views to your map, and each plan has a limit on the number of views per month. If you create a visualization using Mapbox and it becomes popular (I hope so!), you might want to upgrade your account. In any case, you will be notified if you are about to spend your monthly quota.

The Mapbox JavaScript API is implemented as a Leaflet plugin and includes a release of Leaflet, an open source library to create interactive map applications. Leaflet provides an API that allows you to create layers, interactive elements (such as zooming and panning), add custom markers, and many others. We will use the Mapbox and Leaflet APIs to create maps and integrate the maps with D3.

Creating a Mapbox project

We will begin by pointing the browser to <https://www.mapbox.com/projects/> and create a new project. In Mapbox, a project is a map that we can customize according to our needs. By clicking on **Create a Project**, we can access the map editor, where we can customize the colors of land, buildings, and other features; select the base layer (street, terrain, or satellite); and select the primary language for the features and locations in the map. We can also add markers and polygons and export them to KML or GeoJSON in order to use them in other projects. You can also set the map as private or public, remove geocoding, or remove sharing buttons.

Once we save the map, it will be given a map ID. This ID is necessary to load the map using the JavaScript API, view the tiles, or embed the map in a page. The ID is composed of the username and a map handle. To use the JavaScript API, we need to include the `Mapbox.js` library and styles. These files can also be downloaded and installed locally with Bower (`bower install --save-devmapbox.js`):

```
<script src='https://api.tiles.mapbox.com/mapbox.js/
v1.6.2/mapbox.js'></script>
<link href='https://api.tiles.mapbox.com/mapbox.js/
v1.6.2/mapbox.css' rel='stylesheet' />
```

To include the map in our page, we need to create a container div and set its position to absolute. We will also create a container for this div element in order to give the map container a positioning context:

```
<div class="map-container">
  <div id="map"></div>
</div>
```

The top and left offsets of the map container will be governed by the parent element. For this example, we will add the styles at the top of our page:

```
<style>
.map-container {
  position: relative;
  width: 600px;
  height: 400px;
}

#map {
  position: absolute;
  top: 0;
  bottom: 0;
  width: 100%;
}
</style>
```

Creating Maps

The next step is to create an instance of the map. We can set the view's center and zoom level by chaining the `setView` method:

```
<script>
var mapID = 'usernamexxxxxxxx', // replace with your map ID
    center = [12.526, -69.997],
    zoomLevel = 11;

var map = L.mapbox.map('map', mapID)
    .setView(center, zoomLevel);
</script>
```

The map of Aruba created by Mapbox is shown in the following figure:



The map will be rendered in the container div. Note that the maps support all the interactions that we expect; we can zoom in or out and drag the map to explore the surrounding areas.

Integrating Mapbox and D3

In this example, we will create a bubble plot map to show the population of the main cities in Aruba. We have created a JSON file with the necessary information. The JSON file has the following structure:

```
{
  "country": "Aruba",
  "description": "Population of the main cities in Aruba",
  "cities": [
    {
      "name": "Oranjestad",
      "population": 28294,
      "coordinates": [12.519, -70.037]
    },
    ...
  ]
}
```

To create the bubbles, we will create a **layer**. Layers are objects attached to a particular location, such as markers or tiles. Most Leaflet objects are created by extending the `L.Class` object, which implements simple classical inheritance and several utility methods. We will create a D3 layer class by extending the `L.Class` object. Layers must at least have an `initialize` and the `onAdd` methods. We will also include the `onRemove` method to remove the bubbles if the layer is removed. A basic layer will have the following structure:

```
var D3Layer = L.Class.extend({
  initialize: function(arguments...) {
    // Initialization code
  },
  onAdd: function(map) {
    // Create and update the bubbles
  },
  onRemove: function(map) {
    // Clean the map
  }
});
```

The `initialize` method will be invoked when the layer instance is created. The `onAdd` method is invoked when the layer is added to the map. It receives the map object as an argument, which gives us access to the panes, zoom level, and other properties of the map. The `onRemove` method is invoked when the layer is removed from the map, and it also receives the map as an argument.

Whenever the user interacts with the map by dragging or zooming, the map triggers the `viewreset` event. This method notifies that the layers need to be repositioned. The `latLngToLayerPoint` method can be used to set the new position of the objects, given their geographic coordinates.

In our case, the `initialize` method will receive a single `data` argument, which will contain the array with the cities of Aruba. We will store the data array as an attribute of the layer:

```
initialize: function(data) {  
    this._data = data;  
},
```

The data array will be accessible from each method of the layer. In the `onAdd` method, we will select a pane from the map and append the bubbles to it. The container div for our bubbles will be the overlay pane of the map, which is a pane designed to contain custom objects. The overlay pane gets repositioned automatically each time the user pans the map.

One strategy is to create a single svg container to hold our bubbles and to resize it each time the user zooms in or out. Another strategy we will use is to create one svg element for each bubble and position it absolutely using the projection of the coordinates of each feature. We will create a selection for the svg elements and bind the data to the selection:

```
onAdd: function(map) {  
  
    // Create SVG elements under the overlay pane  
    var div = d3.select(map.getPanes().overlayPane),  
        svg = div.selectAll('svg.point').data(this._data);  
  
    // Create the bubbles...  
},
```

To position the SVG elements, we need to project the latitude and longitude of each point and use these coordinates to set the offsets of our svg containers. We will add the `L.LatLng` objects with the coordinates of each city to each data item:

```
// Stores the latitude and longitude of each city  
this._data.forEach(function(d) {
```

```
d.LatLng = new L.LatLng(d.coordinates[0],  
d.coordinates[1]);  
});
```

We will use this attribute in just a few moments. The svg elements will be created to contain just one bubble, so they don't overlap other areas of the map. Before setting the size of the svg elements, we need to compute the radius scale. The area of the bubbles should be proportional to the population of the city:

```
// Create a scale for the population  
var rScale = d3.scale.sqrt()  
    .domain([0, d3.max(this._data, function(d) {  
        return d.population;  
    })])  
    .range([0, 35]);
```

We can now create the svg elements and set their size and position. Note that the svg element's width and height are twice as big as the radius of the bubble:

```
svg.enter().append('svg')  
    .attr('width', function(d) {  
        return 2 * rScale(d.population);  
    })  
    .attr('height', function(d) {  
        return 2 * rScale(d.population);  
    })  
    .attr('class', 'point leaflet-zoom-hide')  
    .style('position', 'absolute');
```

We have added the `leaflet-zoom-hide` class to each svg element so that they are hidden when the map is being zoomed in or out by the user. We also set the position of the svg container to absolute. We can finally add the bubbles as usual, appending a circle to each svg container:

```
svg.append('circle')  
    .attr('cx', function(d) { return rScale(d.population); })  
    .attr('cy', function(d) { return rScale(d.population); })  
    .attr('r', function(d) { return rScale(d.population); })  
    .attr('class', 'city')  
    .on('mouseover', function(d) {  
        d3.select(this).classed('highlight', true);  
    })  
    .on('mouseout', function(d) {  
        d3.select(this).classed('highlight', false);  
    });
```

We added event listeners for the `mouseover` and `mouseout` events. In these events, we add the `highlight` class to the circles, which will increase the opacity of the circles.

When the user drags the map, the overlay pane will be moved as well and the bubbles will be well positioned. When the user zooms in/out of the map, the `viewreset` event will be triggered, and we must reposition the `svg` containers. We will create an `updateBubbles` function to update the position of the bubbles on zoom, and invoke this function on `viewreset`:

```
// Update the position of the bubbles on zoom
map.on('viewreset', updateBubbles);
```

When the callback of the `viewreset` event is invoked, the `map.latLngToLayerPoint` projection method is updated with the new zoom level and position. So, we can use it to set the offsets of the `svg` elements:

```
function updateBubbles() {
  svg
    .style('left', function(d) {
      var dx = map.latLngToLayerPoint(d.LatLng).x;
      return (dx - rScale(d.population)) + 'px';
    })
    .style('top', function(d) {
      var dy = map.latLngToLayerPoint(d.LatLng).y;
      return (dy - rScale(d.population)) + 'px';
    });
}
```

Finally, we invoke the `updateBubbles` method to render the bubbles:

```
// Render the bubbles on add
updateBubbles();
```

The `onRemove` method is simpler; we just select the overlay pane and remove all the `svg` elements from it:

```
onRemove: function(map) {
  var div = d3.select(map.getPanels().overlayPanel);
  div.selectAll('svg.point').remove();
}
```

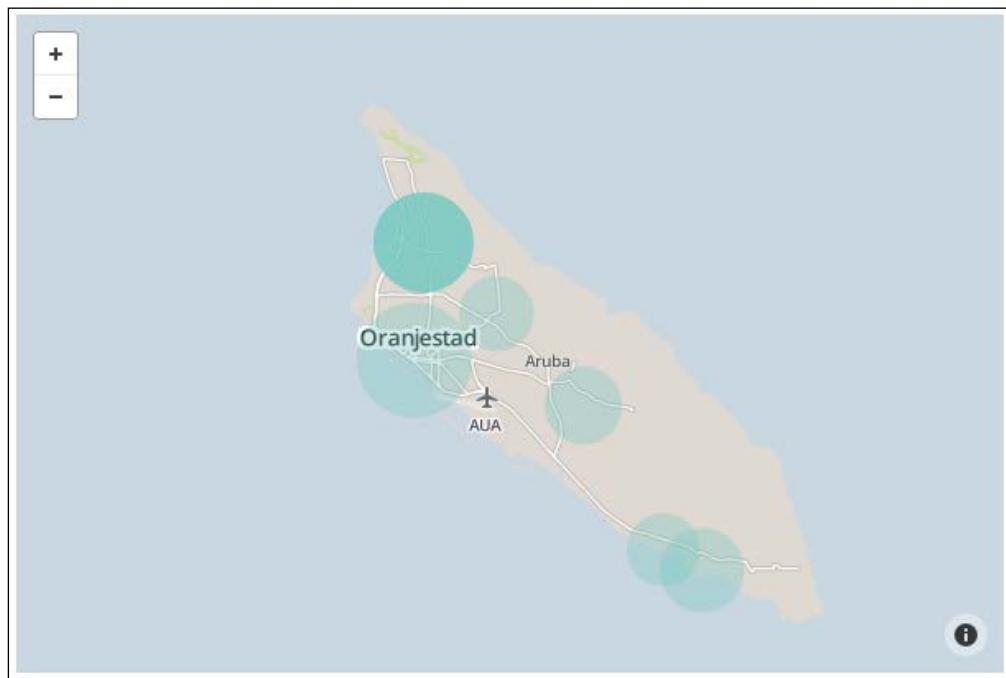
Having created our layer, we can retrieve the JSON file and append it to the map:

```
// Retrieve the dataset of cities of Aruba
d3.json('/chapter10/data/aruba-cities.json', function(error, data) {

    // Handle errors getting or parsing the data
    if (error) { return error; }

    // Create a layer with the cities data
    map.addLayer(new D3Layer(data.cities));
});
```

Our following map shows the bubbles that represent the population of the cities of Aruba:



A bubble plot map with D3 and Mapbox

Summary

In this chapter, we learned how to obtain and transform geographic datasets using open source tools. We learned how to use and interpret two popular formats for mapping information for the Web, GeoJSON, and TopoJSON.

We also learned how to create simple charts based on SVG, rendering the geographic features as svg paths. We created a choropleth map using the Mercator projection and used TopoJSON to obtain information about topology, allowing us to identify neighbors and frontiers between countries.

Finally, we learned how to use Mapbox and D3 to create data visualizations for applications that require street-level detail.

In the next chapter, we will learn how to use other projections to create 3D-like views of our maps and how to project raster images in our maps.

11

Creating Advanced Maps

In the last chapter, we learned how to use the GeoJSON and TopoJSON formats to create map-based charts using SVG. In this chapter, we will explore different cartographic projections and learn how to use the Orthographic and Stereographic projections to create 3D-like renderings of our maps.

We will add interaction to our maps by adding drag and zoom behavior, allowing the user to rotate and zoom the map views. We will use the Orthographic projection to create a rotating view of the Earth, and we will create a star map using the Orthographic projection.

We will also learn how to project raster images of the Earth using canvas and D3 projections in order to have realistic renderings of it.

Using cartographic projections

As we mentioned in the previous chapter, cartographic projections are functions that map positions on the Earth to points on a flat surface. The `d3.geo` module of D3 implements about a dozen of the most used cartographic projections, and there are even more cartographic projections available in the extended geographic projections plugin at <https://github.com/d3/d3-geo-projection/>.

There are a great number of projections because none of them are appropriate for every application. The Mercator projection, for instance, was created as a navigation tool. Straight lines on the Mercator projection are rhumb lines, which are lines of constant compass bearing. This projection is very useful for navigation, but the areas near the poles are extremely distorted. The poles, which are points on the surface of the Earth, are represented as lines that are as long as the equator. The Orthographic projection, on the other hand, is closer to what we would see from space, creating a more accurate mental image of how the Earth really is, but it's probably not very useful to navigate by sea.

In this section, we will learn how to use more projections and discuss some of their properties. The examples in this section are in the `chapter11/01-projections` file in the code bundle. For the examples in this section, we will use a TopoJSON file that contains land features, which are generated from the medium-scale shapefiles from Natural Earth. The `Makefile` in the `chapter11/data` folder will download and transform the necessary files for us.

Using the Equirectangular projection

The **Equirectangular** projection linearly maps longitude to a horizontal position and latitude to a vertical position using the same scale. As there are 180 degrees from pole to pole, and the circumference of the earth covers 360 degrees, the width of a world map created with this projection will be twice its height. Its mathematical simplicity is about its only useful property.

To create a world map, we begin by loading the TopoJSON file and using the `topojson.feature` method to compute the GeoJSON object, representing the shapes described in the `ne_50m_land` object:

```
d3.json('/chapter11/data/land.json', function(error, data) {  
  
    // Notifies about errors getting or parsing the data  
    if (error) { console.error(error); }  
  
    // Construct the GeoJSON features  
    var geojson = topojson.feature(data,  
        data.objects.ne_50m_land);  
  
    // Create the projection and draw the features...  
});
```

As usual, we set the width and height of the `svg` container of the map. We select the `container` div for the map and append the `svg` element, setting its width and height:

```
// Set the width and height of the svg element  
var width = 600,  
    height = 300;  
  
// Append the svg container and set its size  
var div = d3.select('#map-equirectangular'),  
    svg = div.append('svg')  
        .attr('width', width)  
        .attr('height', height);
```

We create an instance of the `d3.geo.equirectangular` projection and configure its scale and translation by chaining the corresponding methods. Note that scales are not consistent among projections. In this case, in order to show the world map, we can set the scale either to `height / Math.PI` or `width / (2 * Math.PI)`. In both cases, the result will be the same:

```
// Create an instance of the Equirectangular projection
var equirectangular = d3.geo.equirectangular()
  .scale(width / (2 * Math.PI))
  .translate([width / 2, height / 2]);
```

Once we have the projection created, we can create an instance of the geographic path generator and set its `projection` attribute:

```
// Create and configure the geographic path generator
var path = d3.geo.path()
  .projection(equirectangular);
```

The path generator receives a GeoJSON feature or feature collection and uses the projection to compute the corresponding svg path string. Finally, we append a path element to the svg container, bind the feature collection to the selection, and set the path data string to be computed with the path generator:

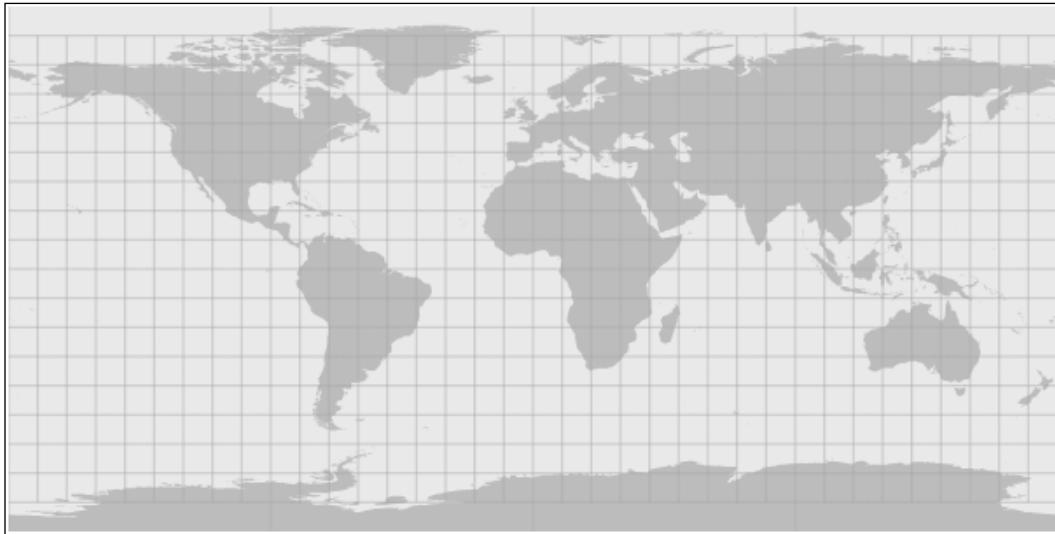
```
// Append the path of the features to the svg container
svg.append('path').datum(geojson)
  .attr('class', 'land')
  .attr('d', path);
```

We will also add parallels and meridians to the figure. D3 has a generator that creates these lines. The `d3.geo.graticule()` method returns a configurable function that creates a feature collection that contains the graticule lines:

```
// Create the graticule lines
var graticule = d3.geo.graticule();

// Add the graticule to the figure
svg.append('path').datum(graticule())
  .attr('class', 'graticule')
  .attr('d', path);
```

A world map created with the Equirectangular projection is shown in the following screenshot:



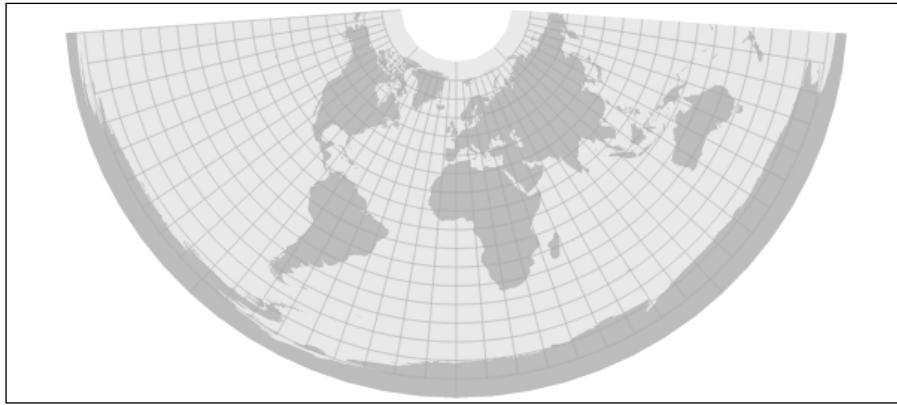
The Conic Equidistant projection

The Conic Equidistant projection maps the sphere into a cone whose axis coincides with the axis of the Earth. The cone can be tangent to the sphere in one parallel or secant in two parallels, which are called standard parallels. In this projection, the poles are represented with arcs, and the local shapes are true among the standard parallels. In the Conic Equidistant projection, the distances among meridians are proportionally correct. They are better suited to represent regions that have a small range of latitude, such as regional maps or small countries.

The process to generate a world map with this projection is the same as the previous process, except that this time, we need to create and configure an instance of the `d3.geo.conicEquidistant` projection. For this projection, computing the exact scale will be more difficult, but it's easy to adjust it until it has the correct size:

```
// Create and configure an instance of the projection
var conic = d3.geo.conicEquidistant()
  .scale(0.75 * width / (2 * Math.PI))
  .translate([width / 2, height / 2]);
```

The world map created with the Conic Equidistant projection is shown in the following screenshot:



As we mentioned earlier, this projection is not appropriate to display the world map, but it can represent small areas accurately. We can rotate the projection to center it around New Zealand and set a bigger scale. We set the standard parallels to 5 degrees north and 15 degrees south. This will minimize the distortion among these parallels:

```
// Create and configure an instance of the projection
var conic = d3.geo.conicEquidistant()
  .scale(0.85 * width / (Math.PI / 3))
  .rotate([-141, 0])
  .translate([width / 2, height / 2])
  .parallels([5, -15]);
```

The map of New Zealand using the Conic Equidistant projection is shown in the following screenshot:



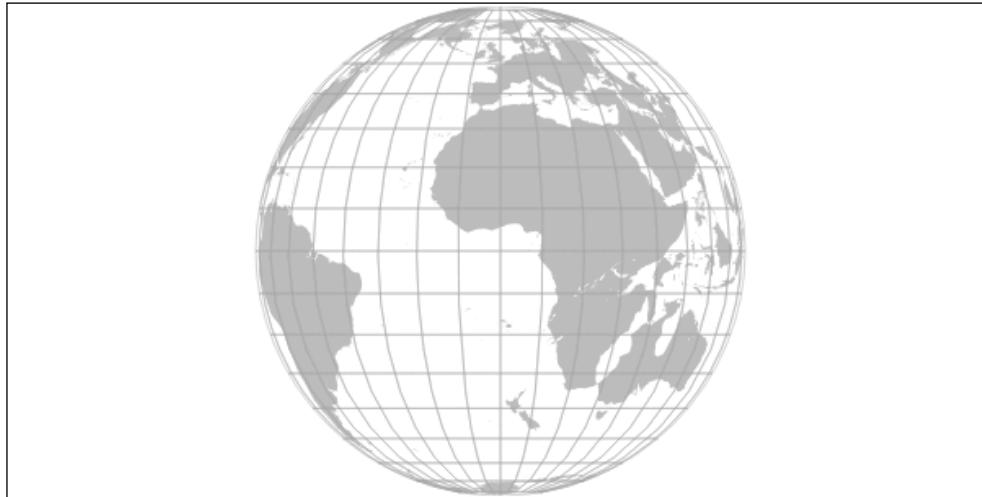
The Orthographic projection

The Orthographic projection is a perspective projection that shows the Earth as seen from space. This gives us the illusion of a three-dimensional view. Only one hemisphere can be seen at a time without overlapping. This has minimal distortion near the center and huge distortion towards the horizon.

To use this projection, we need to set the scale and translation of the projection and use it to configure the path generator:

```
// Create an instance of the Orthographic projection
var orthographic = d3.geo.orthographic()
  .scale(height / 2)
  .translate([width / 2, height / 2]);
```

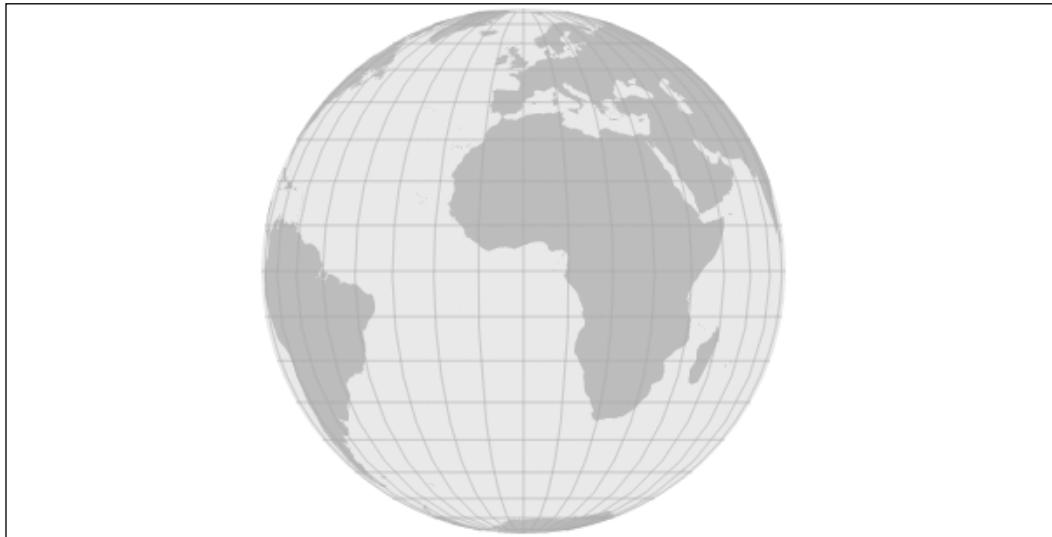
An orthographic view of the Earth is shown in the following screenshot:



To avoid overlapping, we would need to display only the features that are on the same side as the observer. To do this, we need to clip the projection to hide the features that are at the other side of the Earth. The `clipAngle` method allows us to clip the features beyond the clipping angle. Setting this angle to 90 will modify the geometry of the features whose angular distance is greater than 90 from the center of the projection; so, they are not shown in the image:

```
// Create an instance of the Orthographic projection
var orthographic = d3.geo.orthographic()
  .scale(height / 2)
  .translate([width / 2, height / 2])
  .clipAngle(90);
```

The following screenshot shows us Earth from the side of the observer:



An Orthographic view of Earth with clipping

The projections shown in this section are only a small sample of the projections available in the geographic module of D3. As we can see, the pattern of use of different projections is always the same, but the parameters of each projection should be adjusted to get the desired result. In the next section, we will learn how to use the drag behavior to rotate the globe.

Creating a rotating globe

The Orthographic projection displays the Earth like a 3D object, but it only shows us one side at a time, and only the center is shown accurately. In this section, we will use this projection and the zoom behavior to allow the user to explore the features by rotating and zooming in on the globe.

The code of this example is available in the `chapter11/02-rotating` file of the code bundle. We will begin by drawing a globe using the Orthographic projection. As we did in the previous section, we load the TopoJSON data and construct the GeoJSON feature collection that represents the `ne_50m_land` object:

```
d3.json('/chapter11/data/land.json', function(error, data) {  
    // Handle errors getting or parsing the data
```

```
if (error) { console.error(error); }

// Construct the GeoJSON feature collection using TopoJSON
var geojson = topojson.feature(data,
  data.objects.ne_50m_land);

// Create the svg container...
});
```

We set the width and height of the svg element and use these dimensions to create and configure an instance of the Orthographic projection. We also select the container div and append the svg container to the map:

```
// Width and height of the svg element
var width = 600,
  height = 300;

// Create an instance of the Orthographic projection
var orthographic = d3.geo.orthographic()
  .scale(height / 2)
  .translate([width / 2, height / 2])
  .clipAngle(90);

// Append the svg container and set its size
var div = d3.select('#map-orthographic'),
  svg = div.append('svg')
    .attr('width', width)
    .attr('height', height);
```

We create an instance of the geographic path generator and set its projection to be the Orthographic projection instance:

```
// Create and configure the geographic path generator
var path = d3.geo.path()
  .projection(orthographic);
```

We will add features to represent the globe, the land, and lines for the parallels and meridians. We will add a feature to represent the globe in order to have a background for the features. The path generator supports an object of the `Sphere` type. An object of this type doesn't have coordinates, since it represents the complete globe. We will also append the GeoJSON object that contains the land features:

```
// Globe
var globe = svg.append('path').datum({type: 'Sphere'})
  .attr('class', 'globe')
```

```

    .attr('d', path);

    // Features
    var land = svg.append('path').datum(geojson)
        .attr('class', 'land')
        .attr('d', path);

```

We will also add the graticule, which is a set of parallels and meridian lines, in order to give us a more accurate reference for the orientation and rotation of the sphere:

```

    // Create the graticule generator
    var graticule = d3.geo.graticule();

    // Append the parallel and meridian lines
    var lines = svg.append('path').datum(graticule())
        .attr('class', 'graticule')
        .attr('d', path);

```

The preceding code will show us the Earth with the same aspect as the previous section. The strategy to add rotation and zoom to the globe will be to add an invisible overlay over the globe and add listeners for the pan and zoom gestures using the zoom behavior. The callback for the `zoom` event will update the projection rotation and scale and update the paths of the features using the path generator configured earlier. To keep the state of the zoom behavior and the projection in sync, we will store the current rotation angles and the scale of the projection in the `state` variable:

```

    // Store the rotation and scale of the projection
    var state = {x: 0, y: -45, scale: height / 2};

```

We will update the configuration of the projection to use the attributes of this variable, just for consistency:

```

    // Create and configure the Orthographic projection
    var orthographic = d3.geo.orthographic()
        .scale(state.scale)
        .translate([width / 2, height / 2])
        .clipAngle(90)
        .rotate([state.x, state.y]);

```

The zoom and pan should be triggered only when the user performs these gestures over the globe, not outside. We will create an overlay circle of the same size as the globe and set its `fill-opacity` attribute to zero. We will bind the `state` variable to the overlay in order to modify it in the `zoom` callback:

```

    // Append the overlay and set its attributes
    var overlay = svg.append('circle').datum(state)

```

```
.attr('r', height / 2)
.attr('transform', function() {
    return 'translate(' + [width / 2, height / 2] + ')';
})
.attr('fill-opacity', 0);
```

We need to create an instance of the zoom behavior and bind it to the overlay. This will add event listeners for the pan and zoom gestures to the overlay. We will limit the scale extent to between 0.5 and 8:

```
// Create and configure the zoom behavior
var zoomBehavior = d3.behavior.zoom()
    .scaleExtent([0.5, 8])
    .on('zoom', zoom);

// Add event listeners for the zoom gestures to the overlay
overlay.call(zoomBehavior);
```

When the user zooms or pans the overlay, a `zoom` event is triggered. The current event is stored in the `d3.event` variable. Each event type has its own attributes. In the case of the `zoom` event, the zoom translation vector and the scale are accessible through the `d3.event.translate` and `d3.event.scale` attributes. We need to transform the scale and the translation vector to the appropriate projection scale and rotation:

```
function zoom(d) {

    // Compute the projection scale and the constant
    var scale = d3.event.scale,
        dx = d3.event.translate[0],
        dy = d3.event.translate[1];

    // Maps the translation vector to rotation angles...
}
```

The `zoom` event will be triggered several times when the user performs the drag gesture. The `translate` array accumulates the horizontal and vertical translations from the point where the drag gesture originated. If the user drags the globe from the left-hand side to the right-hand side, the globe should be rotated 180 degrees, counterclockwise. We will map the horizontal position of the translation vector to an angle between 0 and 180 degrees:

```
// Maps the translation vector to rotation angles
d.x = 180 / width * dx;      // Horizontal rotation
d.y = -180 / height * dy;   // Vertical rotation
```

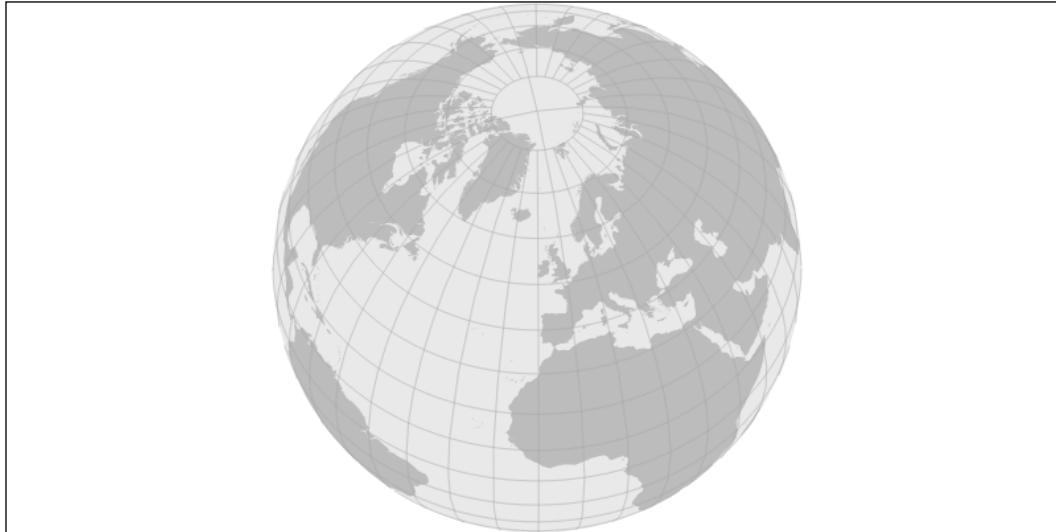
If the user drags the North Pole towards the bottom of the image, we will want the globe to rotate forward. As the latitude is measured from the equator to the poles, we need to rotate the projection by a negative angle in order to have a rotation forward from the point of view of the observer. With the rotation angle computed, we can update the projection's `rotate` and `scale` attributes. The zoom scale is a relative zoom factor, and we need to multiply it by the original size of the map to obtain the updated scale of the projection:

```
// Update the projection with the new rotation and scale
orthographic
.rotate([d.x, d.y])
.scale(scale * d.scale);
```

To update the image, we need to reproject the features and compute the svg paths. The path has a reference to the projection instance, so we need to just update all the paths in the svg with the updated projection:

```
// Recompute the paths and the overlay radius
svg.selectAll('path').attr('d', path);
overlay.attr('r', scale * height / 2);
```

We also updated the overlay radius so that the dragging area coincides with the globe, even if the user changes the zoom level. The globe can be rotated and zoomed as shown in the following screenshot:



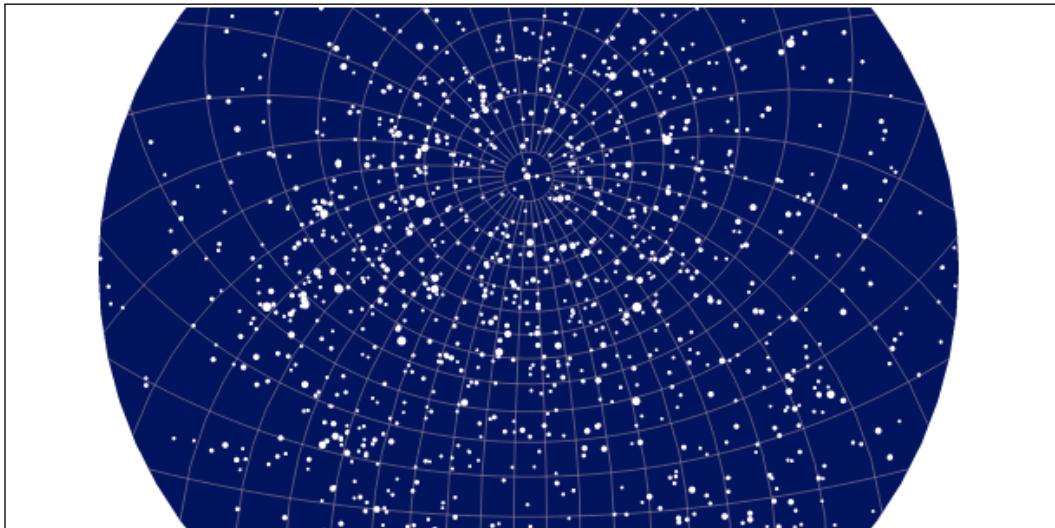
The globe can be rotated and zoomed with mouse or touch gestures, allowing the user to explore every part of the globe in detail. The strategy to add rotation and zoom behaviors can be used with any projection, adapting the mapping between the zoom translation and scale to rotations and scale of the projection. Depending on the level of detail of the features, there can be some performance issues during the rotation. The projection and path generation are being done on each rotation step. This can be avoided by rendering simpler features during the rotation, or even showing just the graticule when rotating and rendering the features when the user releases the mouse.

The rotation of the globe is not perfect, but it's a good approximation of what we would expect. For a better (but more complex) strategy, see the excellent article from Jason Davies at <https://www.jasondavies.com/maps/rotate/>.

In the next section, we will use the Stereographic projection and the zoom behavior to create an interactive star map.

Creating an interactive star map

In this section, we will use the Stereographic projection and a star catalog to create an interactive celestial map. The Stereographic projection displays the sphere as seen from inside. A star map created with the Stereographic projection is shown in the following screenshot:



Celestial coordinate systems describe positions of celestial objects as seen from Earth. As the Earth rotates on its axis and around the Sun, the position of the stars relative to points on the surface of the Earth changes. Besides rotation and translation, a third movement called precession slowly rotates the Earth's axis by one degree every 72 years. The Equatorial coordinate system describes the position of stars by two coordinates, the **declination** and the angle from the projection of the Earth's equator to the poles. This angle is equivalent to the Earth's longitude. The **right ascension** is the angle measured from the intersection of the celestial equator to the ecliptic, measured eastward. The ecliptic is the projection of the Earth's orbit in the celestial sphere. The right ascension is measured in hours instead of degrees, but it is the equivalent of longitude.

Choosing our star catalog

To create our star map, we will use the HYG database, which is a celestial catalog that combines information from the Hipparcos Catalog, the Yale Bright Star Catalog, and the Gliese Catalog of Nearby Stars. This contains about 120,000 stars, most of which are not visible to the naked eye. The most recent version of the HYG database is available at <https://github.com/astronexus/HYG-Database>.

As we did in the previous sections, we will add targets to `Makefile` in order to download and parse the data files that we need. In order to filter and process the stars of the catalog, we wrote a small Python script that filters out the less bright stars and writes a GeoJSON file that translates the declination and right ascension coordinates to equivalent latitudes and longitudes. Note that due to the rotation of the Earth, the equivalent longitude is not related to the Earth's longitude, but it would be useful to create our visualization. We can compute a coordinate equivalent to the right ascension using the `longitude = 360 * RA / 24 - 180` expression. The generated GeoJSON file will have the following structure:

```
{  
  "type": "FeatureCollection",  
  "features": [  
    {  
      "geometry": {  
        "type": "Point",  
        "coordinates": [-179.6006208, -77.06529438]  
      },  
      "type": "Feature",  
      "properties": {
```

```
        "color": 1.254,
        "name": "",
        "mag": 4.78
    }
},
...
]
}
```

In this case, every feature in the GeoJSON file is a point. We will begin by creating a chart using the Equirectangular projection to have a complete view of the sky while we are implementing the map and change the projection to stereographic later. We begin by loading the GeoJSON data and creating the svg container for the map:

```
d3.json('/chapter11/data/hyg.json', function(error, data) {

    // Handle errors getting and parsing the data
    if (error) { console.log(error); }

    // Container width and height
    var width = 600, height = 300;

    // Select the container div and creates the svg container
    var div = d3.select('#equirectangular'),
        svg = div.append('svg')
            .attr('width', width)
            .attr('height', height);

    // Creates an instance of the Equirectangular projection...
});
```

We create and configure an instance of the Equirectangular projection, setting the scale to display the entire sky in the svg container:

```
// Creates an instance of the Equirectangular projection
var projection = d3.geo.equirectangular()
    .scale(width / (2 * Math.PI))
    .translate([width / 2, height / 2]);
```

Drawing the stars

We will represent the stars with small circles, with bigger circles for brighter stars. For this, we need to create a scale for the radius, which will map the apparent magnitude of each star to a radius. Lower magnitude values correspond to brighter stars:

```
// Magnitude extent
var magExtent = d3.extent(data.features, function(d) {
    return d.properties.mag;
});

// Compute the radius for the point features
var rScale = d3.scale.linear()
    .domain(magExtent)
    .range([3, 1]);
```

By default, the path generator will create circles of constant radius for features of the Point type. We can configure the radius by setting the path's pathRadius attribute. As we might use the same path generator to draw point features other than stars, we will return a default value if the feature doesn't have the properties attribute:

```
// Create and configure the geographic path generator
var path = d3.geo.path()
    .projection(projection)
    .pointRadius(function(d) {
        return d.properties ? rScale(d.properties.mag) : 1;
});
```

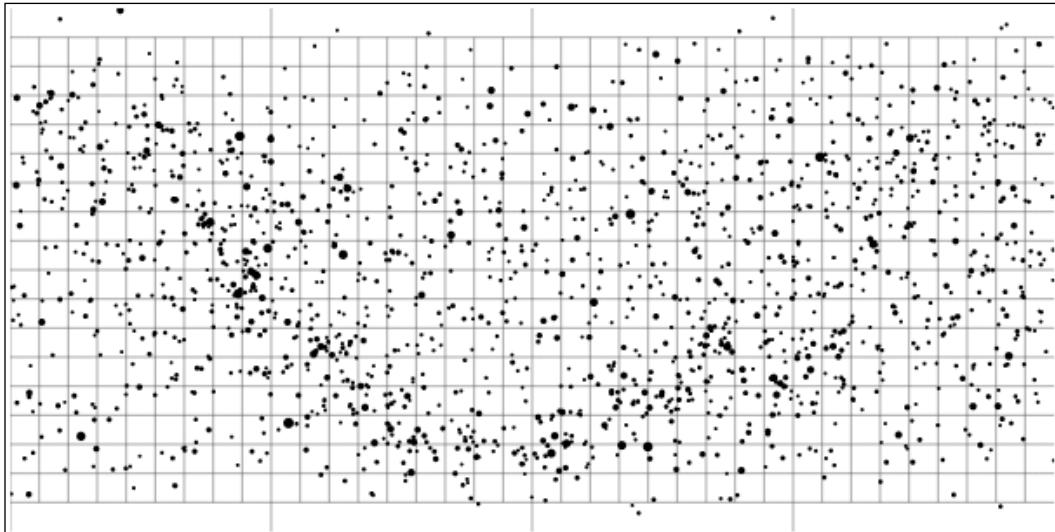
With our path generator configured, we can append the graticule lines and the features for the stars to the svg container:

```
// Add graticule lines
var graticule = d3.geo.graticule();

svg.selectAll('path.graticule-black').data([graticule()])
    .enter().append('path')
    .attr('class', 'graticule-black')
    .attr('d', path);

// Draw the stars in the chart
svg.selectAll('path.star-black').data(data.features)
    .enter().append('path')
    .attr('class', 'star-black')
    .attr('d', path);
```

We obtain the following celestial map, which shows us the graticule and the stars as small black circles:



Celestial map created with the Equirectangular projection

Changing the projection and adding rotation

We will replace the Equirectangular projection with the Stereographic projection and add styles to the elements of the map to make it more attractive. We will use the drag behavior to allow the user to rotate the chart. As we did with the rotating globe, we will create a variable to store the current rotation of the projection:

```
// Store the current rotation of the projection
var rotate = {x: 0, y: 45};
```

We can create and configure an instance of the Stereographic projection. We will choose a suitable scale, translate the projection center to the center of the svg container, and clip the projection to show only a small part of the sphere at a time. We use the rotation variable to set the initial rotation of the projection:

```
// Create an instance of the Stereographic projection
var projection = d3.geo.stereographic()
  .scale(1.5 * height / Math.PI)
  .translate([width / 2, height / 2])
  .clipAngle(120)
  .rotate([rotate.x, -rotate.y]);
```

We won't duplicate the code to create the svg container, graticule, and features because it's very similar to the rotating globe from earlier. The complete code for this example is available in the `chapter11/03-celestial-sphere` file. In this example, we also have an invisible overlay. We create and configure a drag behavior instance and add event listeners for the drag gesture to the invisible overlay:

```
// Create and configure the drag behavior
var dragBehavior = d3.behavior.drag()
    .on('drag', drag);

// Add event listeners for drag gestures to the overlay
overlay.call(dragBehavior);
```

The drag function will be invoked when the user drags the map. For drag events, the `d3.event` object stores the gesture's x and y coordinates. We will transform the coordinates to horizontal and vertical rotations of the projection with the same method as the one used in the previous section:

```
// Callback for drag gestures
function drag(d) {
    // Compute the projection rotation angles
    d.x = 180 * d3.event.x / width;
    d.y = -180 * d3.event.y / height;

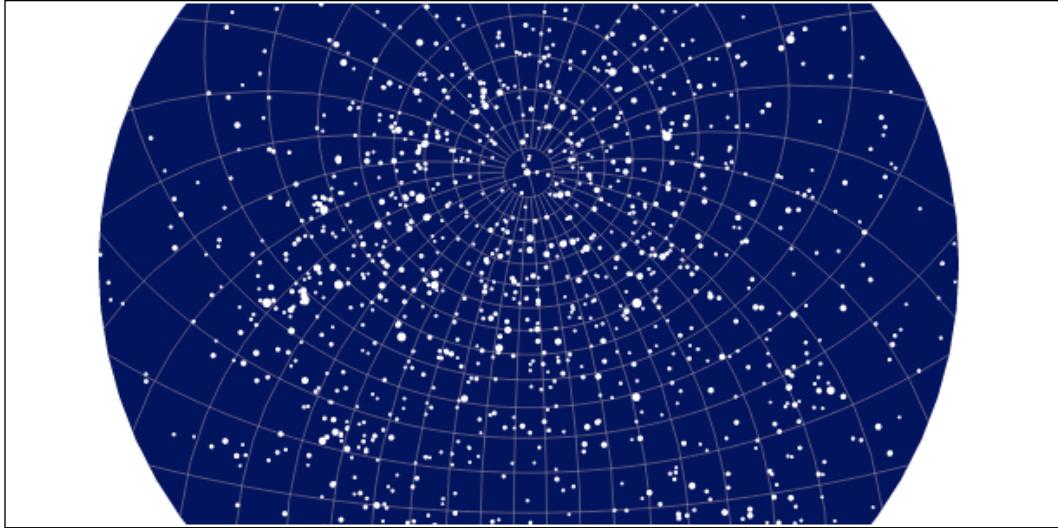
    // Updates the projection rotation...
}
```

We update the projection rotation and the paths of the stars and graticule lines. As we have clipping in this example, the path will be undefined for stars outside the clipping angle. In this case, we return a dummy svg command that just moves the drawing cursor to avoid getting errors:

```
// Updates the projection rotation
projection.rotate([d.x, d.y]);

// Update the paths for the stars and graticule lines
stars.attr('d', function(u) {
    return path(u) ? path(u) : 'M 10 10';
});
lines.attr('d', path);
```

In the style sheet file, `chapter11/maps.css`, we have included styles for this map to display a dark blue background, light graticule lines, and white stars. The result is a rotating star map that displays a coarse approximation of how the stars look from Earth.



Rotating star map created with the Stereographic projection

Adding colors and labels to the stars

We will create a fullscreen version of the star map. The source code for this version of the map is available in the `chapter11/04-fullscreen` file in the code bundle. For this example, we need to set the body element and the container div to cover the complete viewport. We set the width and height of the body, HTML, and the container div to 100 percent and set the padding and margins to zero. To create the `svg` element with the correct size, we need to retrieve the width and height in pixels, which are computed by the browser when it renders the page:

```
// Computes the width and height of the container div
var width = parseInt(div.style('width'), 10),
    height = parseInt(div.style('height'), 10);
```

We create the projection and the path generator as done earlier. In this version, we will add colors to the stars. Each star feature contains the attribute color, which indicates the color index of the star. The color index is a number that characterizes the color of the star. We can't compute a precise scale for the color index, but we will use a color scale that approximates the colors:

```
// Approximation of the colors of the stars
var cScale = d3.scale.linear()
    .domain([-0.3, 0, 0.6, 0.8, 1.42])
    .range(['#6495ed', '#fff', '#fcff6c', '#ffb439',
        '#ff4039']);
```

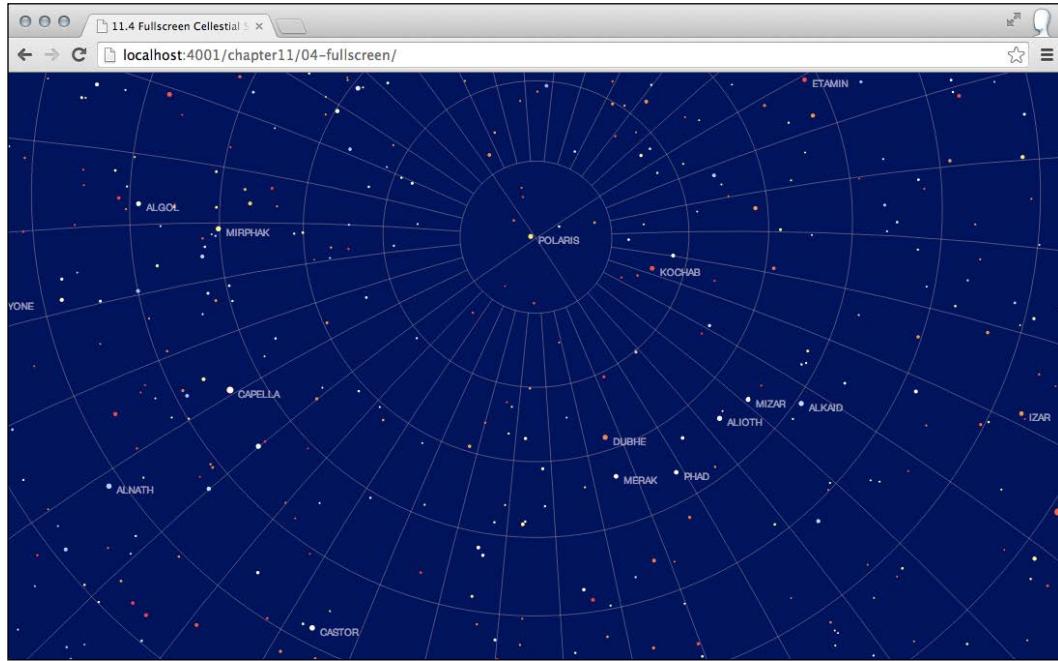
We will set the color to the features using the `fill` attribute of the paths that correspond to the stars:

```
// Add the star features to the svg container
var stars = svg.selectAll('path.star-color')
    .data(data.features)
    .enter().append('path')
    .attr('class', 'star-color')
    .attr('d', path)
    .attr('fill', function(d) {
        return cScale(d.properties.color);
});
```

We will also add labels for each star. Here, we use the projection directly to compute the position where the labels should be, and we also compute a small offset:

```
// Add labels for the stars
var name = svg.selectAll('text').data(data.features)
    .enter().append('text')
    .attr('class', 'star-label')
    .attr('x', function(d) {
        return projection(d.geometry.coordinates)[0] + 8;
    })
    .attr('y', function(d) {
        return projection(d.geometry.coordinates)[1] + 8;
    })
    .text(function(d) { return d.properties.name; })
    .attr('fill', 'white');
```

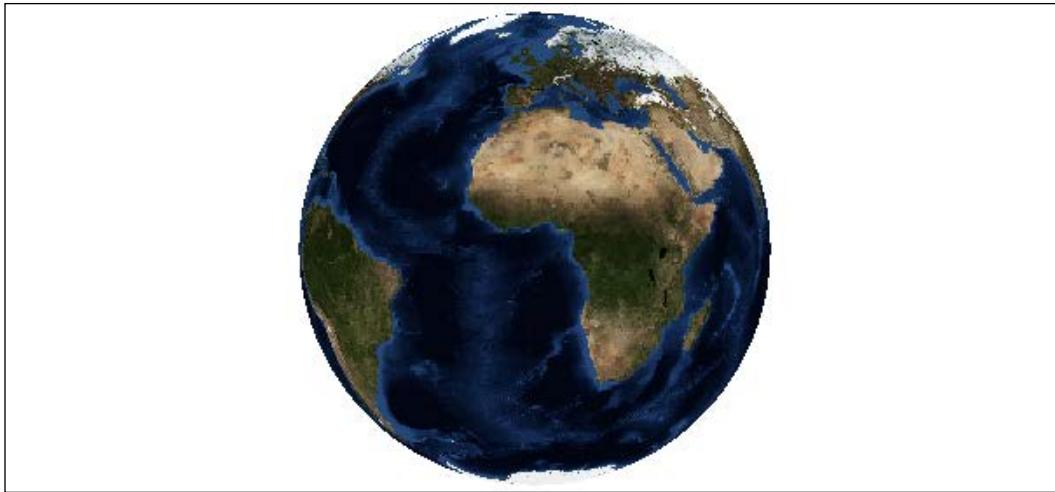
The star map visualization in fullscreen is shown in the following screenshot:



We create the overlay and the drag behavior and configure the callback of the zoom event as earlier, updating the position of the labels in the zoom function. Now we have a fullscreen rotating star map.

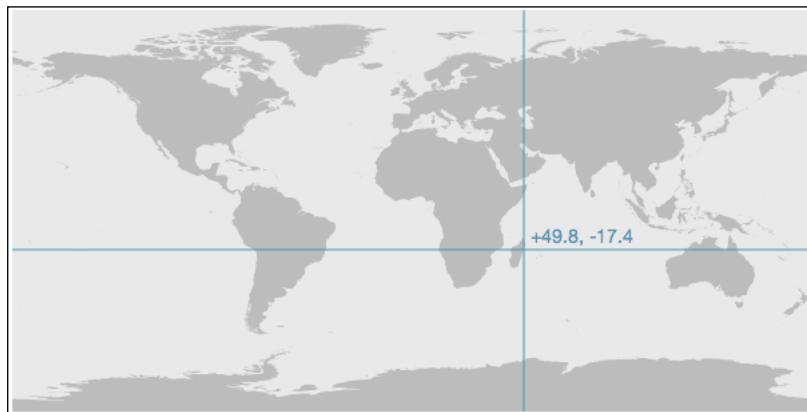
Projecting raster images with D3

Until now, we have used SVG to create maps. In this section, we will learn how to use D3 to project raster images in canvas elements. This will allow us to use JPG or PNG images to generate orthographic views of these images. A raster image reprojected using the Orthographic projection is shown in the following screenshot:



Rendering an image of Earth using the Orthographic projection (or any other projection) involves manipulating two projections. First, for each pixel in the original image, we compute its corresponding geographic coordinates using the image inverse projection. Then, we use the geographic coordinates of each pixel to render them using the Orthographic projection. Before beginning the implementation of these steps, we will discuss the inverse method of a projection.

Projections are functions that map geographic coordinates to points on the screen. The **inverse** projections do the reverse operation; they take coordinates on the two-dimensional surface and return geographic coordinates. In the `chapter11/05-raster` file, there is an interactive example that computes the geographic coordinates of the point under the mouse. Note that not all projections in D3 have an inverse operation. Computing the geographic coordinates of a point under the mouse is shown in the following screenshot:



To obtain the geographic coordinates that correspond to a point under the cursor, we can add a callback to the mouseover event over an element:

```
// Callback of the mouseover event
rect.on('mousemove', function() {
    // Compute the mouse position and the corresponding
    // geographic coordinates.
    var pos = d3.mouse(this),
        coords = equirectangular.invert(pos);
})
```

The **Next Generation Blue Marble** images are satellite images of Earth captured, processed, and shared by NASA. There are monthly images that show Earth with a resolution of 1 pixel every 500 meters. These images are available in several resolutions at the NASA Earth Observatory site at <http://earthobservatory.nasa.gov/Features/BlueMarble/>. We will use a low-resolution version of the Blue Marble image in this section.

Rendering the raster image with canvas

This time, we won't use `svg` to create our visualization, because we need to render individual pixels in the screen. Canvas is more suitable for this task. We will begin by creating a canvas element, loading the Blue Marble image, and drawing the image in the canvas element. We select a container `div` and append a `canvas` element, setting its width and height as follows:

```
// Canvas element width and height
var width = 600,
    height = 300;

// Append the canvas element to the container div
var div = d3.select('#canvas-image'),
    canvas = div.append('canvas')
        .attr('width', width)
        .attr('height', height);
```

The `canvas` element is just a container. To draw shapes, we need to get the 2D context. Remember that only the 2d context exists:

```
// Get the 2D context of the canvas instance
var context = canvas.node().getContext('2d');
```

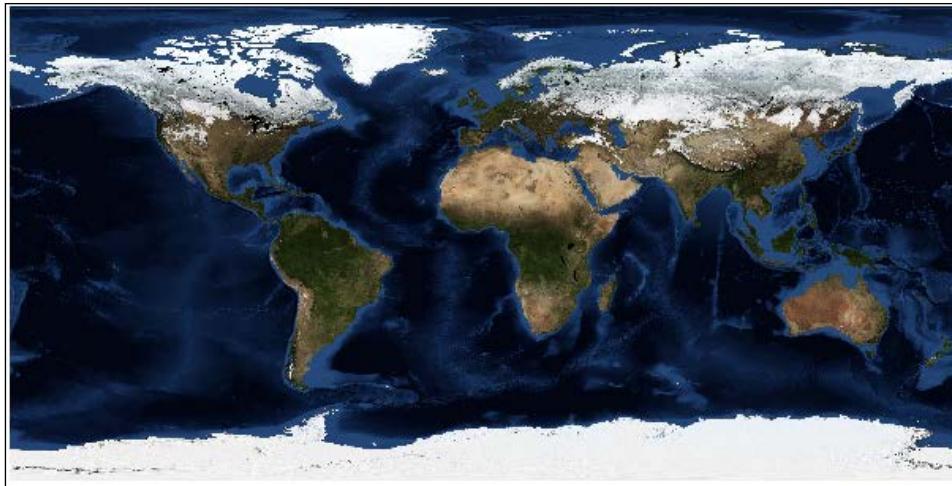
Then, we create an instance of `Image`. We set the image source and set a callback that can be invoked when the image is fully loaded:

```
// Create the image element
var image = new Image();
image.onload = onLoad;
image.src = '/chapter11/data/world.jpg';
```

In the `onLoad` function, we use the canvas context to draw the image. The arguments of the `drawImage` method are the image, the offset, and size of the source image and the offset and size of the target image. In this case, the original image size is 5400 x 2700 pixels; the target image size is just 600 x 300 pixels:

```
// Copy the image to the canvas context
function onLoad() {
    context.drawImage(image, 0, 0, image.width, image.height,
        0, 0, width, height);
}
```

The Blue Marble image rendered in a canvas element is shown in the following screenshot:



As the Blue Marble image was created using the Equirectangular projection, we can create an instance of this projection and use the `invert` method to compute the longitude and latitude that corresponds to each pixel:

```
// Create and configure the Equirectangular projection
var equirectangular = d3.geo.equirectangular()
    .scale(width / (2 * Math.PI))
    .translate([width / 2, height / 2]);
```

Computing the geographic coordinates of each pixel

We can add an event listener for the `mousemove` event on the canvas element. The `d3.mouse` method returns the position of the mouse relative to its argument, in this case, the canvas element:

```
// Add an event listener for the mousemove event
canvas.on('mousemove', function(d) {

    // Retrieve the mouse position relative to the canvas
    var pos = d3.mouse(this);

    // Compute the coordinates of the current position
});
```

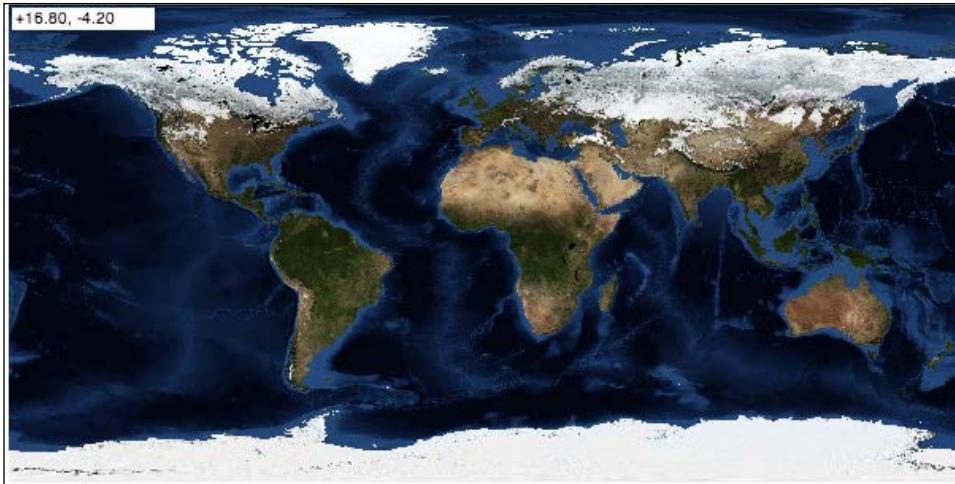
We can use the `invert` method of the projection to compute the geographic coordinates of the point under the cursor. To display the geographic coordinates that correspond to the position of the cursor, we clear a small rectangle of the canvas content and add `fillText` to add the label in the upper-left corner of the image:

```
// Compute the coordinates of the current position
var coords = equirectangular.invert(pos);

// Create a label string, showing the coordinates
var label = [fmt(coords[0]), fmt(coords[1])].join(', ');

// Cleans a small rectangle and append the label
context.clearRect(2, 2, 90, 14);
context.fillText(label, 4, 12);
```

Using the `invert` method to compute geographic coordinates is shown in the following screenshot:



Reprojecting the image using the Orthographic projection

Until now, we have learned how to copy an image element in canvas and how to use the `invert` method of a projection to compute the geographic coordinates that correspond to a pixel in the canvas element. We will use this to **reproject** the raster image using the Orthographic projection instead of the original Equirectangular projection. The strategy to project the image into a different projection is as follows:

- Insert the source image in the canvas element, setting its width and height.
- Create an instance of the source projection (the projection used to generate the image) and configure it in a way that if it were used to project the world, it would fit exactly with the source image.
- Create an empty target image. The size of this image should fit the target projection.
- Create and configure an instance of the target projection. In this case, we will use an instance of the Orthographic projection.
- For each pixel in the target image, use the `invert` method of the target projection to compute the geographic coordinates that correspond to that pixel. Using the source projection, compute the pixel coordinates of that location, and copy the pixel data to the pixel in the target image.

The procedure sounds a little convoluted, but it's basically about copying the pixels from the source image to the target image using the geographic coordinates in order to know where each pixel can be copied to.

We begin by drawing the source image in the canvas element once the image is fully loaded. Images in canvas are represented as arrays. We read the data array of the source data and created an empty target image and got its data:

```
// Copy the image to the canvas context
function onLoad() {

    // Copy the image to the canvas area
    context.drawImage(image, 0, 0, image.width, image.height);

    // Reads the source image data from the canvas context
    var sourceData = context.getImageData(0, 0, image.width,
        image.height).data;

    // Creates an empty target image and gets its data
    var target = context.createImageData(image.width,
        image.height),
    targetData = target.data;

    // ...
}
```

Note that the target image is not shown yet, but we will use it later. In canvas, images are not stored as matrices; they are stored as arrays. Each pixel has four elements in the array, which are its red, green, blue, and alpha components. The rows of the image are stored sequentially in the image array. With this structure, for an image of 200 x 100 pixels, the index of the red component of the pixel 23 x 12 will be $4 * (200 * 11 + 23) = 844$.

To make things easier, we will iterate the image data as if it were a matrix, computing the index of each pixel with the aforementioned expression. We iterate in columns and rows of the target image and compute the corresponding coordinates of the current pixel using the invert method of the target projection:

```
// Iterate in the target image
for (var x = 0, w = image.width; x < w; x += 1) {
    for (var y = 0, h = image.height; y < h; y += 1) {

        // Compute the geographic coordinates of the current pixel
```

```
var coords = orthographic.invert([x, y]);  
  
    // ...  
}  
}  
}
```

The inverse projection could be undefined for a given pixel; we need to check this before we try to use it. We can now use the source projection to compute the pixel coordinates of the current location in the source image. This is the pixel that we need to copy to the current pixel in the target image:

```
// Source and target image indices  
var targetIndex, sourceIndex, pixels;  
  
// Check if the inverse projection is defined  
if ((!isNaN(coords[0])) && (!isNaN(coords[1]))) {  
  
    // Compute the source pixel coordinates  
    pixels = equirectangular(coords);  
  
    // ...  
}
```

Knowing which source pixel we need to copy, we need to compute the corresponding index in the source and target image data arrays. The projection could have returned decimal numbers, so we will need to approximate them to integers. We will also ensure that the indices of the red channel for both images should be exactly divisible by four:

```
// Compute the index of the red channel  
sourceIndex = 4 * (Math.floor(pixels[0]) + w *  
    Math.floor(pixels[1]));  
sourceIndex = sourceIndex - (sourceIndex % 4);  
  
targetIndex = 4 * (x + w * y);  
targetIndex = targetIndex - (targetIndex % 4);
```

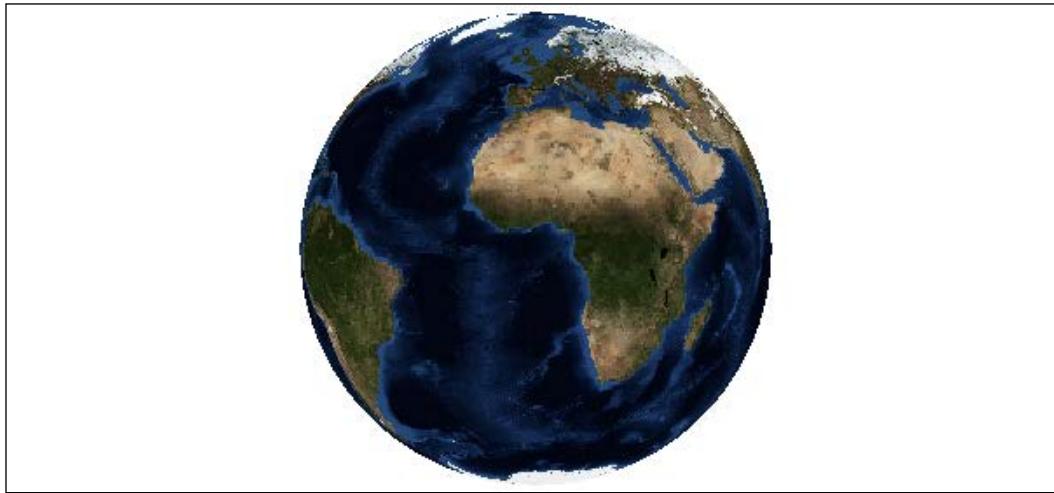
We can copy the color channels using the indices that were just computed:

```
// Copy the red, green, blue and alpha channels  
targetData[targetIndex]      = sourceData[sourceIndex];  
targetData[targetIndex + 1]   = sourceData[sourceIndex + 1];  
targetData[targetIndex + 2]   = sourceData[sourceIndex + 2];  
targetData[targetIndex + 3]   = sourceData[sourceIndex + 3];
```

When we finish iterating, the target image data array should be complete and ready to be drawn in the canvas container. We can clear the canvas area and copy the target image:

```
// Clear the canvas element and copy the target image
context.clearRect(0, 0, image.width, image.height);
context.putImageData(target, 0, 0);
```

We obtain the Blue Marble image that is displayed using the Orthographic projection. The Blue Marble image rendered using the Orthographic projection is shown in the following screenshot:



There is more to know about reprojecting raster images. For instance, Jason Davies has a demo on projecting raster tiles and adding zoom behavior at <http://www.jasondavies.com/maps/raster/>. Also, Nathan Vander Wilt has a well-documented demo on how to use WebGL to reproject raster images using the GPU at http://andyet.iriscouch.com/world/_design/webgl/demo2.html.

Summary

In this chapter, we used several cartographic projections to create interactive maps that can be zoomed and rotated. We created a globe that can be rotated and zoomed using the zoom behavior and the Orthographic projection. We used the Stereographic projection and the HYG combined catalog to create an interactive star map.

We also learned how to use the canvas elements and a combination of projections to project raster images of the Earth using arbitrary projections, creating realistic views of the Earth.

In the next chapter, we will learn how to add social media to our visualization projects and how to have several users interact with our visualizations at the same time.

12

Creating a Real-time Application

In this chapter, we will create a real-time application to explore the distribution of geotagged tweets containing user-defined topics. Creating this visualization will involve implementing a server-side application and a client-side application.

The server-side application will handle connections from the clients, receive topics to be heard on Twitter, and send tweets matching the topics to the corresponding clients.

The client-side application will connect with the streaming server, send it topics as the user enters the streaming server, receive tweets, and update the visualization as the tweets arrive.

We will begin this chapter by learning the basics of real-time interaction in client-side applications. We will use the HDI Explorer application from *Chapter 8, Data-driven Applications*, a service that provides the necessary backend to implement real-time applications.

We will then implement the server-side application using Node, Twit, and **Socket.IO**, a library that provides real-time communication support. Lastly, we will use Backbone, Socket.IO, and D3 to create the client-side application.

Collaborating in real time with Firebase

In some visualization projects, it can be convenient to have the application state shared among users so that they can collaborate and explore the visualization as a group.

Adding this feature would usually imply the installation and configuration of a server and the use of WebSockets or a similar technology to manage the communication between the server and client-side code in the browser. Firebase is a service that provides real-time data storage and synchronization between application instances using the client-side code. If the data changes at one location, Firebase will notify the connected clients so that they can update the state of the application. It has libraries for several platforms, including OS X, iOS, Java, and JavaScript. In this section, we will use Firebase's JavaScript library to add real-time synchronization to the HDI Explorer application.

Adding synchronization to HDI Explorer doesn't make sense in most cases; it would be weird if a user is seeing the evolution of the Human Development Index for one country and it suddenly changes to a different country. On the other hand, if several users were seeing the same visualization on their respective computers and discussing it, this would be useful because changing the selected country would update the visualization for the rest of the users as well, so they would all see the same content. To differentiate between the two scenarios, we will create a new page of the HDI Explorer application with synchronization and leave the index and share pages as they are now. The examples of this section are in the `firebase.md` file of the `hdi-explorer` repository.

Configuring Firebase

To add real-time support to our application, we need to create a Firebase account. Firebase offers a free plan for development, allowing up to 50 connections and 100 MB of data storage, which is more than enough for our application. Once we have created our Firebase account, we can add a new application. The name of our application will be `hdi-explorer`. This name will be used to generate the URL that identifies our application, in our case, `http://hdi-explorer.firebaseio.com`. By accessing this URL, we can see and modify the application data. We will create a single object with the `code` attribute to store the country code of the HDI Explorer application. Once we have our account and initial data configured, we can install the JavaScript library with Bower:

```
$ bower install --save-dev firebase
```

This will install Firebase in the `bower_components` directory. We will also update the `Gruntfile` to concatenate the Firebase library along with the other dependencies of our application in the `dependencies.min.js` file. Firebase data for the HDI Explorer application is shown in the following screenshot:



We can connect to the Firebase application using the Firebase client. We will create a script element at the end of the `firebase.md` page that contains the synchronization code and create an instance of the Firebase reference to our data:

```
<script>
  // Connect to the Firebase application
  var dataref = new Firebase('https://hdi-explorer.firebaseio.
  com/');
  // Application callbacks...
</script>
```

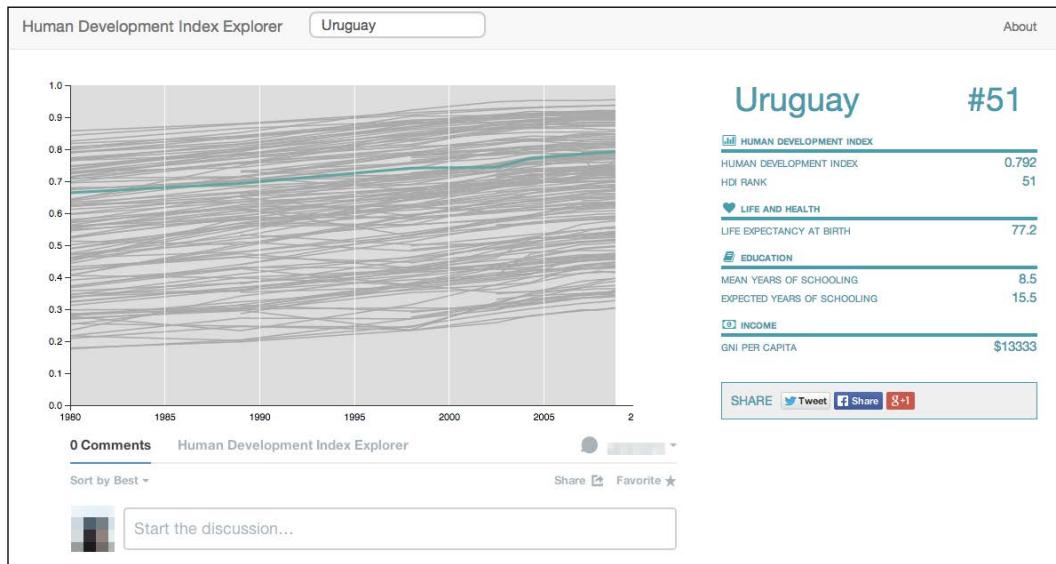
Integrating the application with Firebase

Before integrating Firebase, we will review the structure of the HDI Explorer application. We used Backbone to organize the components of the application and the REST API of the World Bank as the data source for our models.

Our application has three models, that is, the `ApplicationModel`, `CountryTrend`, and `CountryInformation` models. The application model manages the application state, which is defined by the three-letter code of the country selected by the user. The country trend model contains the time series of the aggregated Human Development Index, and the country information model contains information about the main components of the index: education, life expectancy, and average income.

Creating a Real-time Application

The Countries collection contains instances of the CountryTrend model. This collection has two views, the search view and the chart of the HDI trends. The CountryInformation model has one associated view, the table of indicators at the right-hand side of the page. The HDI Explorer application is shown in the following screenshot:



All these elements are initialized in the `app/setup.js` file; instances are created and the callbacks for the `change:code` event in the application model are defined. The views will update themselves when the user selects a country in the search field. To synchronize the application state among the connected users, we need to synchronize the country code.

We can read the data from Firebase through asynchronous callbacks. These callbacks will be invoked in the same way for both the initial state and for the changes in the data. Events will be triggered if any object under the current location is changed, added, removed, or moved. The `value` event will be triggered for any of these modifications. We will use this event to update the application state as follows:

```
// Update the application state
dataref.on('value', function(snapshot) {
  app.state.set('code', snapshot.val().code);
});
```

When something changes under the current location (such as the country code), the `value` event will be triggered, and the callback will be called with a **snapshot** of the current data as the argument. The snapshot will contain the most up-to-date object, representing the state of our application. We can access the object by calling the `val()` method of the snapshot and get the value of the current country code.

If we modify the value of the code in Firebase, users connected to the same URL will have their application instances updated. We also want to synchronize the state from the application to Firebase. We will add an event listener to the application model in order to update the Firebase data:

```
// The model will update the object with the selected country code.  
app.state.on('change:code', function(model) {  
    dataref.set({code: model.get('code')});  
});
```

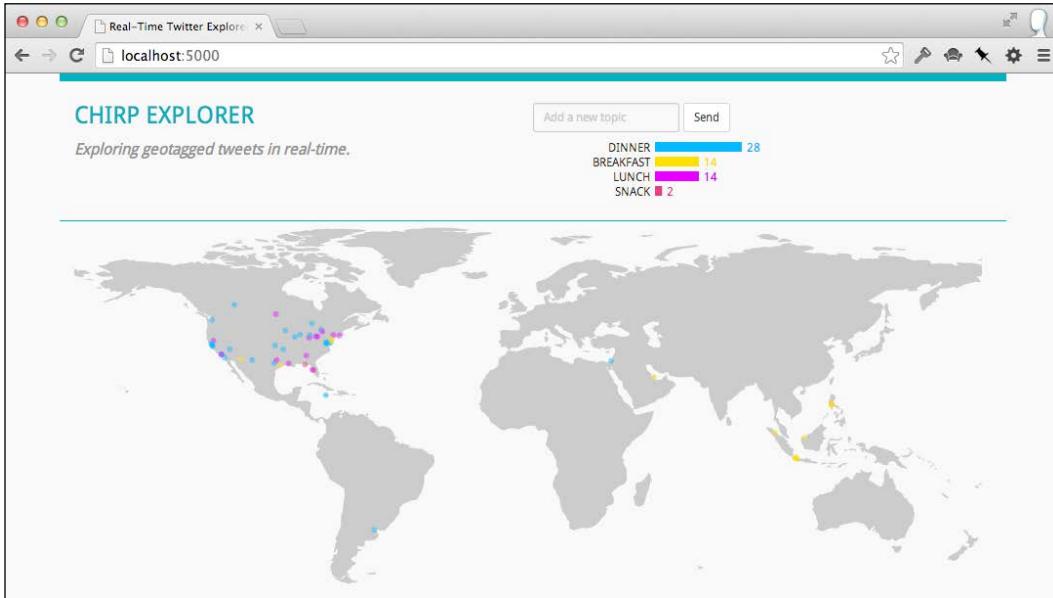
The `set` method will update the contents of the Firebase location, triggering an update of all the views of the local application instance and any other client connected to the `firebase` page.

In this section, we learned how to add real-time interaction to the application created in *Chapter 8, Data-driven Applications*. We learned how to change the state of the client-side application by adding callbacks for the events triggered at the backend.

Creating a Twitter explorer application

In this section, we will create an application to explore the distribution of geotagged tweets at a given time. Users can enter topics in the input box. The topics will be sent to the server-side application, which will begin to send tweets that match the topic to the client-side instance of the corresponding user. This application can be used to track the geographic distribution of a set of topics. For instance, a user might be interested to know which kind of meal is being discussed in different regions of the world, to monitor the Twitter stream for earthquake-related words, or to track the mentions of a particular brand. Our application will support multiple users to be connected at the same time, allowing each user to add up to five topics.

A screenshot of the client application is shown as follows:



Our application will consist of two components, the streaming server and the client-side application. The streaming server will handle the connection with clients, a global list of topics to track, connect to the Twitter-streaming API, and deliver tweets that match the topics to the corresponding client. When a client disconnects, the server will remove the client's topics from the topics list. We will implement the streaming server in Node using Socket.IO to handle the connection with the user and send them tweets, and the **Twit** module to handle the connection to the Twitter-streaming API.

The client-side application will establish the connection with the streaming server, send new topics as they are added, and update the bar chart and the map components. We will use Backbone, Bootstrap, Socket.IO, and D3 to implement the client-side application.

We will begin by learning how to use the Twitter-streaming API and Socket.IO to implement the streaming server and then implement the client application.

Creating the streaming server

In this section, we will use Node to create the streaming server. The server will handle the client connections and the connection to the Twitter-streaming API; it will also manage the list of topics for all the clients who deliver tweets as they arrive.

We will begin by creating Twitter authentication tokens; we will learn how to use the Twit module to manage the connection to the Twitter API, and how to use Socket.IO to handle real-time communication between the server and the client application.

To follow the examples in this section, open the `chirp-server` project directory in the terminal and run the following command to install the project dependencies:

```
$ npm install
```

This will create the `node_modules` directory and download the project dependencies. If you haven't installed Node, download the binaries for your platform from <http://nodejs.org/download/> and follow the instructions on the page.

Using the Twitter-streaming API

Twitter provides a streaming API that allows developers to access Twitter's global stream of tweets through several endpoints. The following endpoints allow access to different streams:

- `statuses/sample`: This allows access to a small random sample of public statuses. All the applications connected to this endpoint will receive the same tweets.
- `statuses/filter`: This returns public statuses that match one or more predicates. We will use this endpoint in the project.
- `statuses/firehose`: This returns all the public statuses. This endpoint requires special access.

Also, there are the `statuses/user` and `statuses/site` endpoints, which allow you to access the public tweets of a particular user or website. When applications establish a connection with the Twitter endpoint, they are delivered a feed of tweets.

To run the examples in this chapter, you need to go to the Twitter Application Management page (<https://apps.twitter.com/>), create a new application, and generate API keys for the application. You will need the consumer key, the consumer secret, the access token, and the token secret.

The `credentials.js` file in the root directory of the project contains placeholders for the authentication tokens of the application. You can either replace the placeholder strings in this file with your own keys or create a new file with the same structure. In either case, make sure that the keys remain a secret. It would be a good idea to add this file to the `.gitignore` file in order to avoid accidentally pushing it to GitHub:

```
// Authentication tokens (replace with your own)
module.exports = {
  "consumer_key": "xxx",
  "consumer_secret": "xxx",
  "access_token": "xxx",
  "access_token_secret": "xxx"
}
```

Using Twit to access the Twitter-streaming API

As mentioned earlier, we will use the Twit Node module (you can access the documentation in the project's repository at <https://github.com/ttezel/twit/>) to connect to the Twitter-streaming API. Twit handles both the REST and streaming APIs, keeping the connection alive and reconnecting automatically if the connection drops. The `01-twitter-sample.js` file contains the code to connect to the `statuses/sample` stream. We begin by importing the `twit` module and loading the configuration module:

```
// Import node modules
var Twit = require('twit'), // Twitter API Client
    config = require('./credentials.js'); // Credentials
```

The `config` object will contain the authentication tokens from the `credentials.js` file. We can now set the Twitter credentials and connect to the `statuses/sample` stream as follows:

```
// Configure the Twit object with the application credentials
var T = new Twit(config);

// Subscribe to the sample stream and begin listening
var stream = T.stream('statuses/sample');
```

The `stream` object is an instance of `EventEmitter`, a built-in Node class that allows you to emit custom events and add listener functions to these events. To begin listening to tweets, we can just attach a listener to the `tweet` event:

```
// The callback will be invoked on each tweet.
stream.on('tweet', function(tweet) {
  // Do something with the tweet
});
```

The `tweet` object will contain information about the tweet, such as the date of creation, tweet ID in the numeric and string formats, tweet text, information about the user who generated the tweet, and language and tweet entities that may be present, such as hashtags and mentions. For a complete reference to the tweet attributes, refer to the Field Guide at <https://dev.twitter.com/docs/platform-objects/tweets>. A typical tweet will have the following structure, along with many other attributes:

```
{  
  created_at: 'Thu May 15 22:27:37 +0000 2014',  
  ...  
  text: 'tweet text...',  
  user: {  
    name: 'Pablo Navarro',  
    screen_name: 'pnavarrc',  
    ...  
  },  
  ...  
  coordinates: {  
    type: 'Point',  
    coordinates: [ -76.786264, -33.234588 ]  
  },  
  ...  
  entities: {  
    hashtags: [],  
    ...  
  },  
  lang: 'en'  
}
```

You may have noticed that the `coordinates` attribute is a GeoJSON object, in particular, a GeoJSON point. This point is an approximation of where the tweet was generated. Less than 10 percent of the tweets in the sample stream contain this information, but the information is still useful and interesting to explore.

The `stream` object will emit other events that we might need to handle. The `connect` event will be emitted when Twit attempts to connect to the Twitter-streaming API:

```
stream.on('connect', function(msg) {  
  console.log('Connection attempt.');  
});
```

If the connection is successful, the `connected` event will be triggered. Note that when the application is running, the connection to the Twitter stream can be interrupted several times. Twit will automatically try to reconnect following the reconnection guidelines from Twitter:

```
// The connection is successful.  
stream.on('connected', function(msg) {  
    console.log('Connection successful.');//  
});
```

If a reconnection is scheduled, the `reconnect` event will be emitted, passing the request, response, and interval within milliseconds of the next reconnection attempt as follows:

```
// Emitted when a reconnection is scheduled.  
stream.on('reconnect', function(req, res, interval) {  
    console.log('Reconnecting in ' + (interval / 1e3) + '  
seconds.');//  
});
```

Twitter creates a queue with the tweets to be delivered to our application. If our program doesn't process the tweets fast enough, the queue will get longer, and Twitter will send a warning message to the application notifying us about the issue. If this happens, Twit will emit the `warning` event, passing the warning message as an argument to the callback function:

```
// The application is not processing the tweets fast enough.  
stream.on('warning', function(msg) {  
    console.warning('warning');//  
});
```

Twitter can disconnect the stream for several reasons. Before actually dropping the connection, Twitter will notify us about the disconnection, and Twit will emit the corresponding event as well. The complete list of events can be consulted in the Twit project repository at <https://github.com/ttezel/twit>.

In the project repository, there are examples of connections to the `statuses/sample` and `statuses/filter` streams. In the `01-twitter-sample.js` file, we configure the Twitter credentials and use the `statuses/sample` endpoint to print the tweets in the terminal screen. To run this example (remember to add your credentials), type the following command in the root directory of the project:

```
$ node 01-twitter-sample.js
```

This will print the tweets on the console as they are received. As mentioned earlier, there are more streaming endpoints available. The statuses/filter stream allows you to track specific topics. The stream object receives a list of comma-separated strings, which should contain the words to be matched. If we pass the topics good morning and breakfast, the stream will contain tweets that match either the word breakfast or both good and morning. Twit allows us to specify the topics that need to be matched as a list of strings.

In the 02-twitter-filter.js file, we have the same setup as in the first example; however, in this case, we define the list of topics to track and configure the stream to connect to the statuses/filter endpoint, passing along the list of topics to track:

```
// List of topics to track
var topics = ['good morning', 'breakfast'];

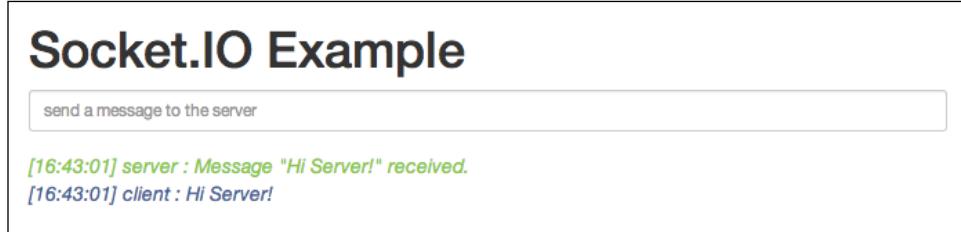
// Subscribe to a specific list of topics
var stream = T.stream('statuses/filter', {track: topics});
```

To determine a match, Twitter will compare the tracking topics with the tweet text, the user name, screen name, and entities, such as hashtags and URLs. Note that the tweets won't include information about which term was matched; as our application will need this information, we will check which terms were matched in the tweet callback.

Using Socket.IO

Socket.IO is a JavaScript library that allows real-time communication between the client and the server. The library has two parts, the client-side library that runs in the browser and the server-side library for Node.

In this example, we will create an application that will allow you to send text messages to the server, which will send a text message back to inform you that the message was received. To follow the code in this example, open the 03-socketio-example.js file for the server-side code and the socketio-example.html file for the client-side code. A screenshot of the client-side application is shown as follows:



Socket.IO Example

send a message to the server

[16:43:01] server : Message "Hi Server!" received.
[16:43:01] client : Hi Server!

We will begin by implementing the server-side code. In the server-side code, we import the `socket.io` module. This will expose the Socket.IO server:

```
// Import the Socket.IO module
var IOServer = require('socket.io');
```

We can now create an instance of the Socket.IO server. We can either use the built-in server or an instance of a different server, such as those provided by the HTTP or express modules. We will use the built-in version, passing along the port number for the server:

```
// Start the server, listening on port 7000
var io = new IOServer(7000);
```

The server is ready to receive connections and messages. At this point, the server won't do anything other than listen; we need to attach a callback function for the connection event. The callback will receive a socket object as the argument. The socket is the client-side endpoint of the connection. Refer to the following code:

```
// Listen for connections from incoming sockets
io.on('connection', function(socket) {

    // Print the socket ID on connection
    console.log('Client ' + socket.id + ' connected.');

    // Attach listeners to socket events...
});
```

This callback will display a log when a client connects to the server, displaying the socket ID:

```
$ node 03-socketio-example.js
Client ID pk0XiCmUNgDVRP6zAAC connected.
```

We can now attach event listeners to the socket events. We will add a callback for the disconnect event, which will display a log message:

```
// Displays a log message if the client disconnects.
socket.on('disconnect', function() {
    console.log('client disconnected.');
});
```

We can attach listeners for custom events as well, and send JavaScript objects that can be serialized as arguments for the callback. Socket.IO also supports the sending of binary data. We will add a callback for the `client-message` custom event:

```
// The server will emit a response message.
socket.on('client-message', function (data) {
    socket.emit('server-message', {
        msg: 'Message "' + data.msg + '" received.'
    });
});
```

The callback for the `client-message` event will just send a message back to the client, indicating that the message was received. If we want to send a message to all the connected clients, we can use `io.emit('event name', parameters)`.

Socket.IO will serve the client-side library, which can be very useful. We can use either this version or download the client-side library separately. We can use the served version by adding `/socket.io/socket.io.js` to the server's URL. To follow the client-side code, open the `socketio-example.html` file from the project directory. In this example, we will use the served version, adding the following line in the header:

```
<script
src="http://localhost:7000/socket.io/socket.io.js"></script>
```

Note that if you want to use the server from an external device, you should replace `localhost` with the URL of the server. We will add an input element in the page, where the user will type the messages to the server. Under the input element, a list will display the messages both from the client and the server. The older messages will be displaced to the bottom as new messages are received. Refer to the following code:

```
<div class="container">

<h1>Socket.IO Example</h1>

<!-- Input element to send messages -->
<form role="form" class="form-horizontal" id="msgForm">
    <div class="form-group">
        <label for="msgToServer" class="col-sm-1">Message</label>
        <div class="col-sm-9">
            <input type="text" class="form-control input-sm"
id="msgToServer" placeholder="Send a message to the server.">
        </div>
```

```
<button type="submit" class="btn btn-default btn-sm">Send</button>
</div>
</form>

<!-- List with messages -->
<ul id='msg-list' class='list-unstyled'></ul>

</div>
```

We will add a script with the code, which will establish the connection with the server and update the messages list. We will include D3 to handle the user input and to update the list of messages. We begin by opening a connection with the socket server:

```
// Open a connection with the Socket.IO server
var socket = io('http://localhost:7000');
```

Note that in this case, the `socket` variable refers to the server's endpoint. The client API is almost identical to the server API; the `socket` object will also trigger the `connect` event once the connection with the server is established:

```
// The callback will be invoked when the connection establishes
socket.on('connect', function() {
  console.log('Successful connection to the server.');
});
```

We will store each message, sender, and message timestamp in an array to display the messages in a list. We will also create a time formatter to display the timestamp of each message in a friendly format:

```
// Declare variables for the message list and the time formatter
var messages = [],
  dateFmt = d3.time.format('[%H:%M:%S]');
```

We will select the input element with the `#message` ID and attach a listener for the `submit` event. The callback for the event will retrieve the content of the input element and send the message to the server. We use `d3.event.preventDefault()` to prevent the form from trying to submit the form values to the server:

```
d3.select('#msg-form').on('submit', function() {

  var inputElement = d3.select('#message').node(),
    message = inputElement.value.trim();

  // Check that the message is not empty
  if (message) {
```

```
// Sends the message to the server...
}

d3.event.preventDefault();
});
```

We retrieve the contents of the input element and verify that the message is not empty before we send the `client-message` signal. We also reset the input element value so that the user can write a new message without having to erase the message already sent:

```
// Check that the message is not empty
if (message) {
    // Sends the message to the server
    socket.emit('client-message', {msg: message});

    // Append the message to the message list
    messages.push({
        from: 'client',
        date: new Date(),
        msg: message
    });
}

// Update the list of messages
updateMessages();

// Resets the form, clearing the input element
this.reset();
}
```

In the `updateMessages` function, we will sort the messages by date, create a selection for the `li` elements, and append them on to the `enter` selection. The `li` elements will contain the time of the message, who sent the message, and the contents of the message. We will also add a class indicating who sent the message in order to set different colors for the server and the client:

```
// Update the message list
function updateMessages() {

    // Sort the messages, most recent first
    messages.sort(function(a, b) { return b.date - a.date; });

    // Create the selection for the list elements
    var li = d3.select('#msg-list')
```

```
.selectAll('li').data(messages) ;  
  
// Append the list elements on enter  
li.enter().append('li') ;  
  
// Update the list class and content.  
li  
  .attr('class', function(d) { return d.from; })  
  .html(function(d) {  
    return [dateFmt(d.date), d.from, ':', d.msg].join(' ')  
  })  
};
```

The message list will be updated when the user sends a message to the server and when the server sends a message to the client. The application allows the sending and receiving of messages to the server, as shown in the following screenshot:

Socket.IO Example

send a message to the server

[16:43:01] server : Message "Hi Server!" received.
[16:43:01] client : Hi Server!

Implementing the streaming server

In the previous sections, we learned how to use the `Twit` module to connect and receive tweets from the Twitter-streaming API's endpoints. We also learned how to implement bidirectional communication between a Node server and the connected clients using the `Socket.IO` module. In this section, we will use both the modules (`Twit` and `Socket.IO`) to create a server that allows multiple clients to track their own topics on Twitter in real time.

When the user accesses the application, a connection with the streaming server is established. `Socket.IO` will manage this connection, reconnecting and sending heartbeats if necessary. The user can then add up to five words to be tracked by the streaming server.

The streaming server will manage connections with the connected clients and one connection with the Twitter-streaming API. When a client adds a new topic, the streaming server will add it to the topics list. When a new tweet arrives, the server will examine its contents to check whether it matches any of the terms in the topic list and send a simplified version of the tweet to the corresponding client.

The code of the streaming server is in the `chirp.js` file in the top level of the project directory. We will begin by importing the Node modules and the `credentials.js` file, which exports the Twitter authentication tokens:

```
// Import the Node modules
var Twit      = require('twit'),
    IOServer = require('socket.io'),
    config   = require('./credentials.js');
```

To keep track of the correspondence between the topics and clients, we will store the topic and a reference to the topic in the topics list. For instance, if the client with `socket` adds the word '`'breakfast'`', we will add the `{word: 'breakfast', client: socket}` object to the topics list:

```
// List of topics to track
var topics = [];
```

As mentioned in the previous sections, we can use either the `statuses/sample` or the `statuses/filter` endpoints to capture tweets. We will use the `statuses/filter` endpoint in our application, but instead of filtering by topic, we will filter by location and language (tweets in English only). We will set the `locations` parameter to `'-180,-90,180,90'`, meaning that we want results from anywhere in the world, and set the `language` parameter to `'en'`. Passing a list of words to the `statuses/filter` endpoint will force us to reset the connection when the user adds a new topic. This is a waste of resources, and Twitter could apply rate limits if we open and close connections frequently. We will use the `statuses/filter` endpoint to listen to anything in the stream and filter the words we want. This will allow you to add or remove words from the topics list without having to reset the connection. We will initialize the `Twit` object, which will read the Twitter credentials and store them to create the streaming requests to Twitter. We will also create the stream to the `statuses/filter` endpoint and store a reference to it in the `twitterStream` variable. We will filter only using the language (English) and location (the world), and match the items by comparing the topic word with the tweet contents:

```
// Configure the Twit object with the application credentials
var T = new Twit(config);

// Filter by location (the world) and tweets in English
var filterOptions = {
```

```
locations: '-180,-90,180,90',
language: 'en'};

// Creates a new stream object, tracking the updated topic list
var twitterStream = T.stream('statuses/filter', filterOptions);
```

We will define functions to handle the most important Twit stream events. Most of these callbacks will just log a message in the console, stating that the event has occurred. We will define a callback for the `connect` event, which will be triggered when a connection is attempted, and a callback for the `connected` event, which will be emitted when the connection to the Twitter stream is established:

```
// Connection attempt ('connect' event)
function twitOnConnect(req) {
    console.log('[Twitter] Connecting...');
}

// Successful connection ('connected' event)
function twitOnConnected(res) {
    console.log('[Twitter] Connection successful.');
}
```

We will display a log message if a reconnection is scheduled, indicating the interval in seconds:

```
// Reconnection scheduled ('reconnect' event).
function twitOnReconnect(req, res, interval) {
    var secs = Math.round(interval / 1e3);
    console.log('[Twitter] Disconnected. Reconnection scheduled in
        ' + secs + ' seconds.');
}
```

We will also add callbacks for the `disconnect` and `limit` events, which will occur when Twitter sends a `disconnect` or `limit` message, respectively. Note that Twit will close the connection if it receives the `disconnect` message, but not if it receives the `limit` message. In the `limit` callback, we will display a message and stop the stream explicitly:

```
// Disconnect message from Twitter ('disconnect' event)
function twitOnDisconnect(disconnectMessage) {
    // Twit will stop itself before emitting the event
    console.log('[Twitter] Disconnected.');
}

// Limit message from Twitter ('limit' event)
function twitOnLimit(limitMessage) {
```

```
// We stop the stream explicitly.  
console.log('[Twitter] Limit message received. Stopping.');
```

```
twitterStream.stop();  
}  
}
```

Adding log messages for these events can help debug issues or help us know what is happening if we don't receive messages for a while. The event that we should certainly listen for is the `tweet` event, which will be emitted when a tweet is delivered by the streaming endpoint. The callback of the event will receive the `tweet` object as the argument.

As mentioned earlier, we will only send geotagged tweets to the connected clients. We will check whether the tweet text matches any of the terms in the topics list. If a term is found in the tweet text, we will send a simplified version of the tweet to the client who added the term:

```
// A tweet is received ('tweet' event)  
function twitOnTweet(tweet) {  
  
    // Exits if the tweet doesn't have geographic coordinates  
    if (!tweet.coordinates) { return; }  
  
    // Convert the tweet text to lowercase to find the topics  
    var tweetText = tweet.text.toLowerCase();  
  
    // Check if any of the topics are contained in the tweet text  
    topics.forEach(function(topic) {  
  
        // Checks if the tweet text contains the topic  
        if (tweetText.indexOf(topic.word) !== -1) {  
  
            // Sends a simplified version of the tweet to the  
            // client  
            topic.socket.emit('tweet', {  
                id: tweet.id,  
                coordinates: tweet.coordinates,  
                word: topic.word  
            });  
        }  
    });  
}
```

As we are not using the tweet text or its creation date, we will send just the tweet ID, the coordinates, and the matched word to the client. We can now attach the listeners to their corresponding events as follows:

```
// Add listeners for the stream events to the stream instance
twitterStream.on('tweet',      twitOnTweet);
twitterStream.on('connect',    twitOnConnect);
twitterStream.on('connected',  twitOnConnected);
twitterStream.on('reconnect',  twitOnReconnect);
twitterStream.on('limit',     twitOnLimit);
twitterStream.on('disconnect', twitOnDisconnect);
```

We have initialized the connection to the Twitter-streaming API, but as we don't have any topics in our list, nothing will happen. We need to create the Socket.IO server to handle connections with clients, which can add topics to the list. We will begin by defining the port where the Socket.IO server will listen and creating an instance of the server. Note that we can create an instance of the Socket.IO server with or without the new keyword:

```
// Create a new instance of the Socket.IO Server
var port = 9720,
    io = new IOServer(port);

// Displays a message at startup
console.log('Listening for incoming connections in port ' + port);
```

The `io` server will begin listening for incoming connections on port 9720. We can use other port numbers too; remember that ports between 0 and 1023 are privileged, as they require a higher level of permission to bind.

When a client connects, the `connection` event will be emitted by the `io` server, passing the `socket` as an argument to the event callback. In this case, we will display a log message in the console, indicating that a new connection was established, and add listeners for the socket events:

```
// A client's established a connection with the server
io.on('connection', function(socket) {

    // Displays a message in the console when a client connects
    console.log('Client ', socket.id, ' connected.');

    // Add listeners for the socket events...
});
```

The socket object is a reference to the client endpoint in the communication. If the client adds a new topic, the add custom event will be emitted, passing the added topic to the event callback. In the callback for this event, we will append the word and a reference to the socket to the topic list and display a log message in the console:

```
// The client adds a new topic
socket.on('add', function(topic) {
    // Adds the new topic to the topic list
    topics.push({
        word: topic.word.toLowerCase(),
        socket: socket
    });

    console.log('Adding the topic "' + topic.word + '"');
});
```

When the client disconnects, we will remove its topics from the list and display a log message in the terminal:

```
// If the client disconnects, we remove its topics from the
list
socket.on('disconnect', function() {
    console.log('Client ' + socket.id + ' disconnected.');
    topics = topics.filter(function(topic) {
        return topic.socket.id !== socket.id;
    });
});
```

At this point, the server is capable of handling multiple clients connected at the same time, each adding their own terms to the topic list. When we implement (and access) the client-side application, we will have the streaming server generate logs such as the following:

```
$ node chirp.js
Listening for incoming connections in port 9720
[Twitter] Connecting...
[Twitter] Connection successful.
Client 4WDFIrqsbxtf_NO_AAAA connected.
Adding the topic "day"
Adding the topic "night"
Client 4WDFIrqsbxtf_NO_AAAA disconnected.
```

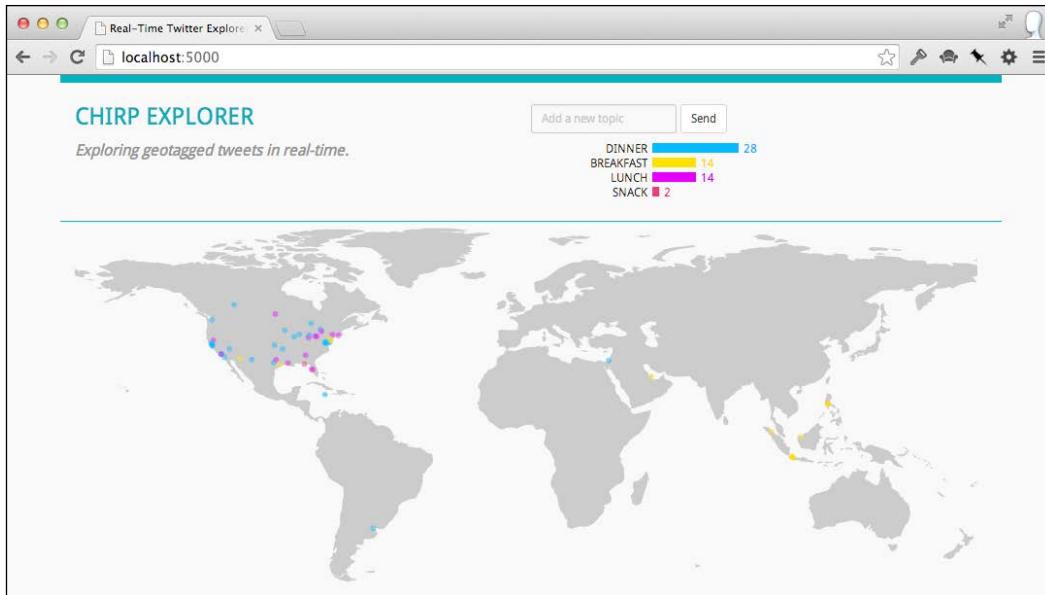
```
Client P8mb97GiLOhc-noLAAAB connected.  
Adding the topic "coffee"  
Adding the topic "tea"  
Adding the topic "milk"  
Adding the topic "beer"  
[Twitter] Disconnected. Reconnection scheduled in 0 seconds.  
[Twitter] Connecting...  
[Twitter] Connection successful.  
Client P8mb97GiLOhc-noLAAAB disconnected.  
Client p3lFgVrxGI0bLPOFAAAC connected.  
...
```

In the next section, we will use the client-side Socket.IO library, D3, and Backbone to create a visualization that shows the geographic distribution of tweets matching the user-defined topics.

Creating the client application

In the previous section, we implemented the streaming server. The server application allows other applications to send words that you can listen to from the `statuses/sample` endpoint from Twitter. When the server receives geotagged tweets containing the words tracked by a client, it will deliver the client a simplified version of the tweet. The server doesn't enforce what the client applications do with the tweets; the client application could just count the tweets, visualize the frequency of each term in time with a heat map, or create a network chart showing the co-occurrence of the terms.

In this section, we will implement a client application. The application will display a map showing the location of the tweets that match each term in a world map, and will display a bar chart that will display the count for each term. As we did in *Chapter 6, Interaction between Charts*, we will use Backbone and D3 to structure our application. A screenshot of the application is shown as follows:



The code of the application is available in the `chapter12/chirp-client` folder. We will begin by describing the project structure, and then implement the project components.

The application structure

As mentioned earlier, we will use Backbone to structure the application's components. As we want to visualize the places where different topics are mentioned in the world at any given time, we will define a topic as the main component of our application. A topic will contain a word (the string that we want to track in Twitter), a color to visualize it, and a list of matching tweets. We will create a `Topic` model containing these attributes.

We will create a collection for the topics. The collection will be in charge of creating the topic instances when the user adds words in the input element and appending the tweets to the corresponding topic instance as they arrive. We will provide our collection with three views: the world map view, the barchart view, and the input element, where the user can add a new topic.

The code for the application components will be in the `src/app` directory in the project folder. We will have separate directories for the models, collections, and views of our application; we will have the `app.js` and `setup.js` files to define the application namespace and to set up and launch our application:

```
app/
  app.js
  models/
    collections/
    views/
  setup.js
```

We will encapsulate the components of our application under the `App` variable, which is defined in the `app/app.js` file. We will add attributes for the collections, models, and views to the `App` object, as follows:

```
// Define the application namespace
var App = {
  Collections: {},
  Models:      {},
  Views:       {}
};
```

To run the application, run the server application and go to the `chirp-client` directory and start a static server. The `Gruntfile` of the project contains a task to run a static server in the development mode. Install the Node modules with the following command:

```
$ npm install
```

After installing the project dependencies, run `grunt serve`. This will serve the files in the directory as static assets and open the browser in the correct port. We will review the application models and views.

Models and collections

The main component of our application will be the `Topic` model. A `Topic` instance will have a `word`, `color`, and an array of simplified tweets matching the topic's word. We will define our model in the `app/models/topic.js` file. The `Topic` model will be created by extending the `Backbone.Model` object:

```
// Topic Model
App.Models.Topic = Backbone.Model.extend({  
  
  // The 'word' attribute will uniquely identify our topic
```

```
idAttribute: 'word',  
  
// Default model values  
defaults: function() {  
    return {  
        word: 'topic',  
        color: '#555',  
        tweets: []  
    };  
},  
  
// addTweet method...  
});
```

The `word` string will uniquely identify our models in a collection. The topics will be given a color to identify them in the bar chart and the map views. Each topic instance will contain an array of tweets that match the `word` of the topic. We will add a method to add tweets to the array; in this method, we will add the topic's color. Array attributes (such as the `tweets` property) are treated as pointers; mutating the array won't trigger the change event. Note that we could also create a model and collection for the tweets, but we will use an array for simplicity.

We will trigger this event explicitly to notify potential observers that the array has changed:

```
// Adds a tweet to the 'tweets' array.  
addTweet: function(tweet) {  
  
    // Adds the color of the topic  
    tweet.color = this.get('color');  
  
    // Append the tweet to the tweets array  
    this.get('tweets').push(tweet);  
  
    // We trigger the event explicitly  
    this.trigger('change:tweets');  
}
```

Note that it is not necessary to define the default values for our model, but we will add them so that we remember the names of the attributes when describing the example.

We will also create a collection for our topics. The Topics collection will manage the creation of the Topic instances and add tweets to the corresponding collection as they arrive from the server. We will make the socket endpoint accessible to the Topics collection, passing it as an option when creating the Topic instance. We will bind the on socket event to a function that will add the tweet to the corresponding topic. We also set a callback for the add event to set the color for a topic when a new instance is created, as shown in the following code snippet:

```
// Topics Collection
App.Collections.Topics = Backbone.Collection.extend({  
  
    // The collection model
    model: App.Models.Topic,  
  
    // Collection Initialization
    initialize: function(models, options) {  
  
        this.socket = options.socket;  
  
        // Store the current 'this' context
        var self = this;  
  
        this.socket.on('tweet', function(tweet) {
            self.addTweet(tweet);
        });  
  
        this.on('add', function(topic) {
            topic.set('color', App.Colors[this.length - 1]);
            this.socket.emit('add', {word: topic.get('word')});
        });
    },  
  
    // Add tweet method...
});
```

The addTweet method of the collection will find the topic that matches the tweet's word attribute and append the tweet using the addTweet method of the corresponding topic instance:

```
addTweet: function(tweet) {  
  
    // Gets the corresponding model instance.
```

```

var topic = this.get(tweet.word);

// Push the tweet object to the tweets array.
if (topic) {
    topic.addTweet(tweet);
}
}

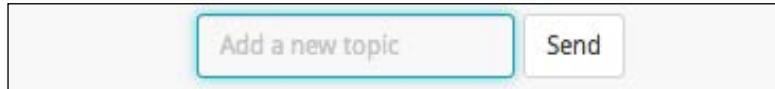
```

Implementing the topics views

In our application, we only need views for the `Topics` collection and for the application itself. We will have a view associated with the bar chart, a view for the map of tweets, and a view for the input element, which will be used to create new topic instances. The code for each of these views is in the `src/app/views` directory in the `topics-barchart.js`, `topics-map.js`, and `topics-input.js` files, respectively.

The input view

The input element will allow the user to add a new topic. The user can write a new topic in the input box and click on the **Send** button to send it to the server and add it to the list of watched topics. The input element is shown in the following screenshot:



To render this view, we will create a template with the markup of the form. We will add the template under the body tag in the `index.html` document:

```

<!-- Input Element Template -->
<script type='text/template' id='topics-template'>
    <form role="form" class="form-horizontal form-inline" id="topic-form">
        <div class="form-group">
            <label for="msgToServer" class="sr-only">Message</label>
            <input type="text" class="form-control input-sm" id="new-topic" placeholder="Add a new topic">
            <button type="submit" class="btn btn-default btn-sm">Send</button>
        </div>
    </form>
</script>

```

We will implement this view by extending the `Backbone.View` object. The `template` attribute will contain the compiled template. In this case, the template doesn't have placeholder text to be replaced when rendering, but we will keep `_.template()` to allow the use of template variables in the future:

```
// Topic Input
App.Views.TopicsInput = Backbone.View.extend({

    // Compile the view template
    template: _.template($('#topics-template').html()),

    // DOM Events
    events: {
        'submit #topic-form': 'addOnSubmit',
    },

    initialize: function (options) {
        // The input element will be disabled if the collection
        // has five or more items
        this.listenTo(this.collection, 'add', this.disableInput);
    },

    render: function () {
        // Renders the input element in the view element
        this.$el.html(this.template(this.collection.toJSON()));
        return this;
    },

    disableInput: function() {
        // Disable the input element if the collection has five or
        // more items
    },

    addOnSubmit: function(e) {
        // adds a topic when the user press the send button
    }
});
```

We will allow up to five topics by a user to avoid having too many similar colors in the map. We have limited space for the bar chart as well, and having an unlimited number of topics per user could have an impact on the performance of the server. When a new topic instance is created in the collection, we will invoke the `disableInput` method, which will disable the input element if our collection contains five or more elements:

```
disableInput: function() {
    // Disable the input element if the collection has five or
    // more items
    if (this.collection.length >= 5) {
        this.$('input').attr('disabled', true);
    }
},
```

To add a new topic, the user can type the terms in the input element as soon as the **Send** button is pressed. When the user clicks on the **Send** button, the submit event is triggered, and the event is passed as an argument to the `addOnSubmit` method. In this method, the default action of the form is prevented as this would cause the page to reload, the topic is added to the topics collection, and the input element is cleared:

```
addOnSubmit: function(e) {

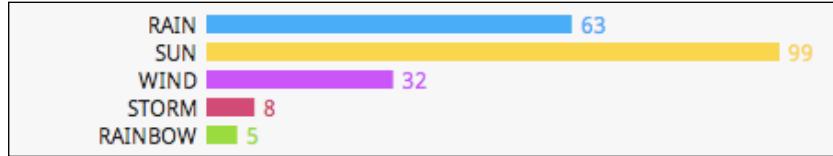
    // Prevents the page from reloading
    e.preventDefault();

    // Content of the input element
    var word = this.$('input').val().trim();

    // Adds the topic to the collection and cleans the input
    if (word) {
        this.collection.add({word: word});
        this.$('input').val('');
    }
}
```

The bar chart view

The bar chart view will encapsulate a reusable bar chart made in D3. We will describe the view and then comment on the implementation of the chart. For now, we will just need to know about the interface of the chart. The code for the bar chart view is available in the `src/app/views/topics-barchart.js` file. A bar chart showing the tweet count for a set of topics is shown in the following screenshot:



We begin by extending and customizing the `Backbone.View` object. We add the `chart` attribute, which will contain a configured instance of the `charts.barChart` reusable chart. The bar chart will receive an array of objects that will be used to create the bars. The `label` attribute allows you to set a function to compute the label for each bar. The `value` attribute will allow you to configure the value that will be mapped to the bar's length, and the `color` attribute will allow you to configure the color of each bar. We set functions for each one of these attributes, assuming that our array will contain elements with the `word`, `count`, and `color` properties:

```
// Bar Chart View
App.Views.TopicsBarchart = Backbone.View.extend({  
  
    // Create and configure the bar chart
    chart: charts.barChart()
        .label(function(d) { return d.word; })
        .value(function(d) { return d.count; })
        .color(function(d) { return d.color; }),  
  
    initialize: function () {
        // Initialize the view
    },  
  
    render: function () {
        // Updates the chart
    }
});
```

In the `initialize` method, we add a callback for the `change:tweets` event in the collection. Note that the `change:tweets` event is triggered by the topic instances; the collection just echoes the events. We will also render the view when a new topic is added to the collection:

```
initialize: function () {
    // Render the view when a tweet arrives and when a new
    // topic is added
    this.listenTo(this.collection, 'change:tweets',
        this.render);
    this.listenTo(this.collection, 'add', this.render);
},
```

The `render` method will construct the data array in the format required by the chart, computing the `count` attribute for each topic; this method will select the container element and update the chart. Note that the `toJSON` collection method returns a JavaScript object, not a string representation of an object in the JSON format:

```
render: function () {

    // Transform the collection to a plain JSON object
    var data = this.collection.toJSON();

    // Compute the tweet count for each topic
    data.forEach(function(item) {
        item.count = item.tweets.length;
    });

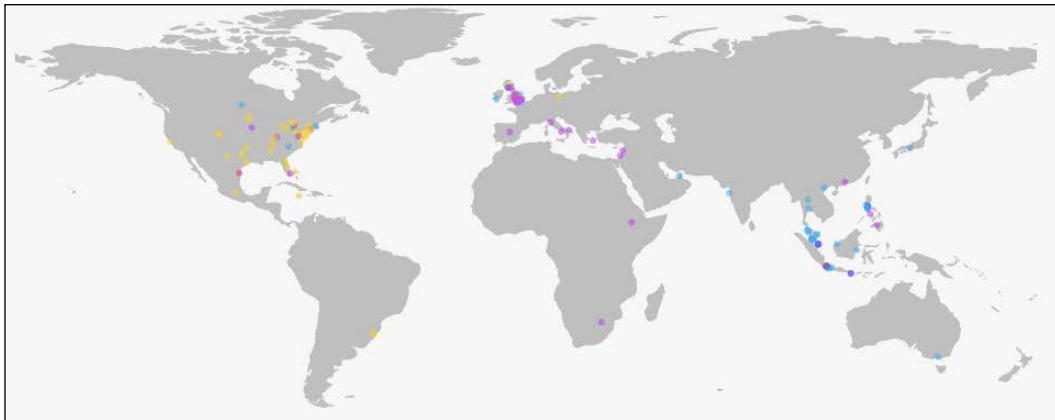
    // Compute the container div width and height
    var div = d3.select(this.el),
        width = parseInt(div.style('width'), 10),
        height = parseInt(div.style('height'), 10);

    // Adjust the chart width and height
    this.chart.width(width).height(height);

    // Select the container element and update the chart
    div.data([data]).call(this.chart);
    return this;
}
```

The topics map view

In the topics map view, the tweets from all the topics will be drawn as points in a map, with each tweet colored as per the corresponding topic. The code of this view is in the `src/app/topics-map.js` file. Tweets for each topic in a world map is shown in the following screenshot:



To create this view, we will use the `charts.map` reusable chart, which will render an array of GeoJSON features and a GeoJSON object base. This chart uses the equirectangular projection; this is important to set the width and height in a ratio of 2:1. In the `initialize` method, we set the base GeoJSON object provided in the options object. In this case, the `options.geojson` object is a feature collection with countries from Natural Earth. We will render the view only when a tweet arrives:

```
// Topics Map View
App.Views.TopicsMap = Backbone.View.extend({  
  
    // Create and configure the map chart
    chart: charts.map()
        .feature(function(d) { return d.coordinates; })
        .color(function(d) { return d.color; }),  
  
    initialize: function (options) {
        // Sets the GeoJSON object with the world map
        this.chart.geojson(options.geojson);  
  
        // Render the view when a new tweet arrives
        this.listenTo(this.collection, 'change:tweets',
        this.render);
```

```
},
render: function () {

    // Gather the tweets for all the topics in one array
    var tweets = _.flatten(_.pluck(this.collection.toJSON(),
        'tweets'));

    // Select the container element
    var div = d3.select(this.el),
        width = parseInt(div.style('width'), 10);

    // Update the chart width, height and scale
    this.chart
        .width(width)
        .height(width / 2)
        .scale(width / (2 * Math.PI));

    // Update the chart
    div.data([tweets]).call(this.chart);
    return this;
}
});
```

In the render method, we gather the tweets for all the topics in one array, select the container element, and update the chart. The geotagged tweets will be drawn as small points on the world map.

Creating the application view

We will create a view for the application itself. This is not really necessary, but we will do it to keep things organized. We will need a template that contains the markup for the application components. In this case, we will have a row for the header, which will contain the page title, lead paragraph, input element, and bar chart. Under the header, we will reserve a space for the map with the tweets:

```
<!-- Application Template -->
<script type='text/template' id='application-template'>

<div class="row header">
    <!-- Title and about -->
    <div class="col-md-6">
```

```
<h1 class="title">chirp explorer</h1>
<p class="lead">Exploring geotagged tweets in real-time.</p>
</div>

<!-- Barchart -->
<div class="col-md-6">
    <div id="topics-form"></div>
    <div id="topics-barchart" class="barchart-block"></div>
</div>
</div>

<div class="row">
    <div class="col-md-12">
        <div id="topics-map"></div>
    </div>
</div>
</script>
```

The application view implementation is in the `src/app/views/application.js` file. In this case, we don't need to compile a template for the view, but we will compile it if we add template variables as the application name or lead text:

```
// Application View
App.Views.Application = Backbone.View.extend({

    // Compile the applicaiton template
    template: _.template($('#application-template').html()),

    // Render the application template in the container
    render: function() {
        this.$el.html(this.template());
        return this;
    }
});
```

In the `render` method, we will just insert the contents of the template in the container element. This will put the markup of the template in the container defined when instantiating the view.

The application setup

With our models, collections, and views ready, we can proceed to create the instances and wire things up. The initialization of the application is in the `src/app/setup.js` file. We begin by creating the `app` variable to hold the instances of collections and views. When the DOM is ready, we will invoke the function that will create the instances of our views and collections:

```
// Container for the application instances
var app = {};

// Invoke the function when the document is ready
$(function() {
    // Create application instances...
});
```

We begin by creating an instance of the application view and setting the container element to the div with the `application-container` ID. The application view doesn't have an associated model or collection; we can render the view immediately as shown in the following code:

```
// Create the application view and renders it
app.applicationView = new App.Views.Application({
    el: '#application-container'
});
app.applicationView.render();
```

We can create an instance of the `Topics` collection. At the beginning, the collection will be empty, waiting for the user to create new topics. We will create a connection to the streaming server and pass a reference to the server endpoint as well as to the collection of topics. Remember that in the initialization method of the `Topics` collection, we add a callback for the `tweet` event of the socket, which adds the tweet to the corresponding collection. Remember to change `localhost` to an accessible URL if you want to use the application on another device:

```
// Creates the topics collection, passing the socket instance
app.topicList = new App.Collections.Topics([], {
    socket: io.connect('http://localhost:9720')
});
```

As we have an instance of the `Topics` collection, we can proceed to create instances for the topic views. We create an instance of the `TopicsInput` view, the `TopicsBarchart` view, and the `TopicsMap` view:

```
// Input View
app.topicsInputView = new App.Views.TopicsInput({
```

```
        el: '#topics-form',
        collection: app.topicList
    });

// Bar Chart View
app.topicsBarchartView = new App.Views.TopicsBarchart({
    el: '#topics-barchart',
    collection: app.topicList
});

// Map View
app.topicsMapView = new App.Views.TopicsMap({
    el: '#topics-map',
    collection: app.topicList
});
```

In the map chart of the TopicsMap view, we need a GeoJSON object with the feature or feature collection to show as the background. We use d3.json to load the TopoJSON file containing the world's countries and convert it to the equivalent GeoJSON object using the TopoJSON library. We use this GeoJSON object to update the map chart's geojson attribute and render the view. This will display the map that shows the world's countries:

```
// Loads the TopoJSON countries file
d3.json('dist/data/countries.json', function(error, geodata) {

    if (error) {
        // Handles errors getting or parsing the file
        console.error('Error getting or parsing the TopoJSON
            file');
        throw error;
    }

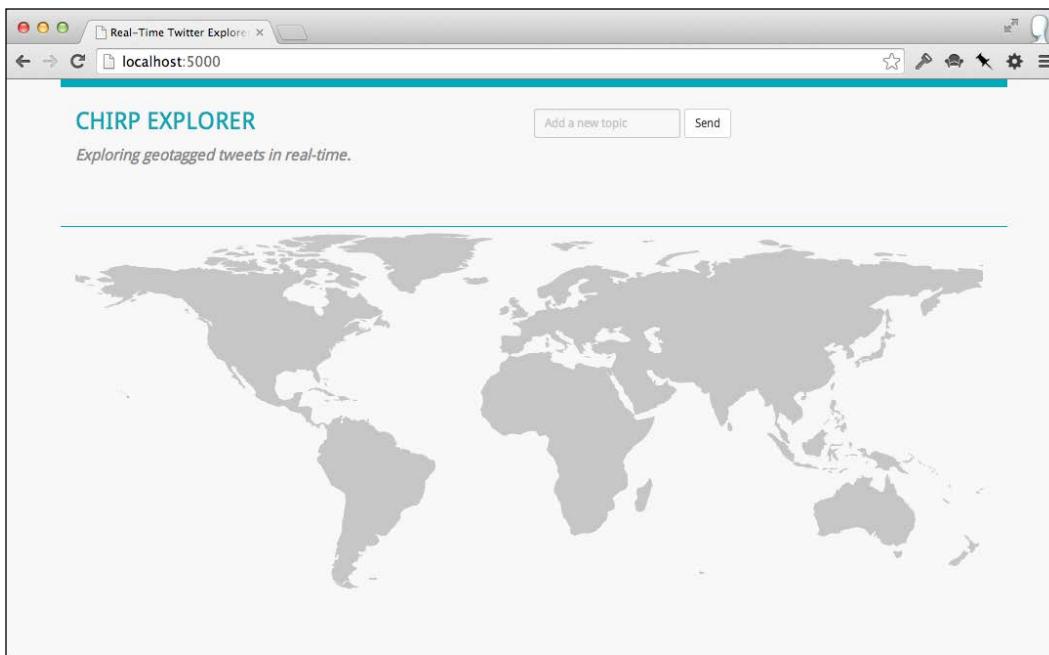
    // Transform from TopoJSON to GeoJSON
    var geojson = topojson.feature(geodata,
        geodata.objects.countries);

    // Update the map chart and render the map view
    app.topicsMapView.chart.geojson(geojson);
    app.topicsMapView.render();
});
```

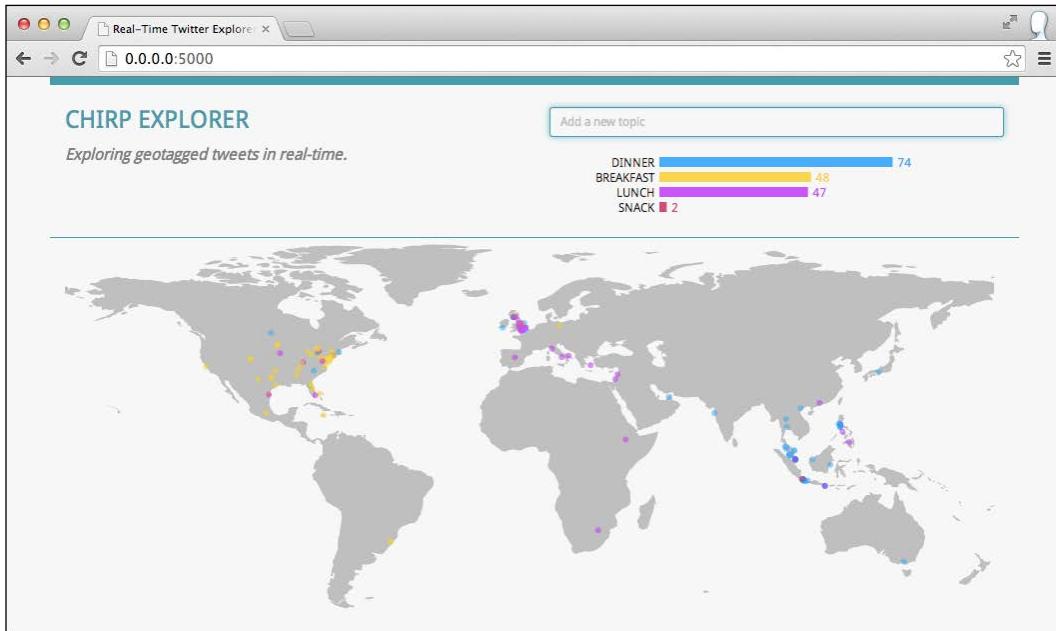
Finally, we render the views for the `topicList` collection:

```
// Render the Topic Views  
app.topicsInputView.render();  
app.topicsBarchartView.render();  
app.topicsMapView.render();
```

At this point, we will have the input element, an empty bar chart, and the world map without tweet points. As soon as the user adds a topic, the server will begin to deliver tweets, which will appear in the views. The application and rendered topic views can be seen in the following screenshot:



Let's recapitulate how the client application works. Once the views are rendered, the user can add topics by typing words in the input box. When the *Enter* key is pressed, the contents of the input element will be added as the word attribute of a new topic instance. The collection will send the word to the streaming server, which will add the topic and a reference to the client in the topics list. The server will be connected to the Twitter-streaming API. Each time the server receives a geotagged tweet, it will compare the tweet text with the topics in the list; if a match is found, a simplified version of the tweet will be sent to the client. In the client, the tweet will be added to the `tweets` array of the topic that matches the tweet text. This will trigger the views to render, updating the bar chart and the map. Here, we can guess where people are having their breakfast and where they are having dinner, as shown in the following screenshot:



The streaming server can be used with any application that can send the `add` event and receive the tweets when the server emits the `tweet` event. We chose to create a client to visualize the geographic distribution of tweets, but we could have implemented a different representation of the same data. If you want to experiment, use the streaming server as a base component to create your own client visualization. Here are some suggestions:

- The time dimension was neglected in this application. It might be interesting to display a heat map that shows how the tweet count varies in time or makes the old tweets fade away.
- Adding zoom and pan to the map chart could be useful to study the geographic distribution of tweets at a more local level.
- The user can't remove topics once they are created; adding a way to remove topics can be useful if the user decides that a topic was not interesting.
- Use brushing to allow the user to select tweets only from a particular region of the world. This would probably involve modifying the server such that it sends the tweets from that location to the client who selected it.
- Use a library of sentiment analysis and add information on whether the topic was mentioned in a positive or negative way.
- Add the ability to show and hide topics so that the overlapping doesn't hide information.

Summary

In the last chapter of the book, we learned how to use D3 and Socket.IO to create a real-time visualization of geotagged tweets. In this chapter, we described two applications: the streaming server and the client application.

We implemented the streaming server in Node. The streaming server keeps a persistent connection to the Twitter-streaming endpoints and supports several connected users at the same time, with each user adding topics to be tracked on the Twitter-streaming API. When the tweets match one of the user topics, they are delivered to the corresponding client.

In the client application, we used Backbone, Socket.IO, and D3 to create a visualization of where the topics are defined by the user in the world. The user can add topics at any given time; the server will add the topic to its list and begin to send tweets that match the topic to the client.

Through this book, we learned how to use D3 to create several kinds of charts, but mostly, we learned how to create visual components that can be reused across several projects. We learned how to integrate D3 and reusable charts with other libraries to structure applications better, how to deploy web applications, and how to add real-time updates to the charts. As we have seen in the examples of this book, D3 is powerful and flexible, making it especially attractive for tinkerers and creative developers.

Bibliography

This Learning Path is a blend of content, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Learning d3.js Data Visualization - Second Edition- Andrew H. Rininsland*
- *D3.js By Example Michael Heydt*
- *Mastering D3.js - Pablo Navarro Castillo*



**Thank you for buying
D3.js: Cutting-edge Data Visualization**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check www.PacktPub.com for information on our titles