Data Visualization with D3.js Cookbook

Over 70 recipes to create dynamic data-driven visualization with D3.js

Nick Qi Zhu
Data Visualization with D3.js Cookbook

Over 70 recipes to create dynamic data-driven visualization with D3.js

Nick Qi Zhu

PACKT Publishing
community experience distilled

BIRMINGHAM - MUMBAI
Data Visualization with D3.js Cookbook

Copyright © 2013 Packt Publishing

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

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

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

First published: October 2013

Production Reference: 1171013

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

Cover Image by Martin Bell (martinb@packtpub.com)
Credits

Author
Nick Qi Zhu

Reviewers
Andrew Berls
Kevin Coughlin
Ismini Lourentzou
Pablo Navarro

Acquisition Editor
Martin Bell

Lead Technical Editor
Sweny M. Sukumaran

Technical Editors
Akashdeep Kundu
Proshonjit Mitra
Sonali S. Vernekar

Project Coordinator
Kranti Berde

Proofreader
Mario Cecere

Indexer
Tejal Soni

Graphics
Yuvraj Mannari

Production Coordinator
Aditi Gajjar

Cover Work
Aditi Gajjar
Nick Qi Zhu is a professional programmer and visualization enthusiast with more than a decade of experience in software development. He is the author of dc.js—a popular multidimensional charting library built on D3. Currently he is having fun and learning as a lead consultant at ThoughtWorks.

I would like to thank the folks at Packt Publishing for supporting me through my journey, especially my editors Martin Bell and Sweny Sukumaran for polishing up my prose making this book much easier to read. And many thanks to my technical reviewers who had really made this book a much better one through their constructive criticism.

Finally to my wife Sherry for being supportive and incredibly patient with me through the last several months; without her support this book would not be possible.
About the Reviewers

Andrew Berls is a Ruby and JavaScript developer and lives in Santa Barbara, CA. He's been building websites ever since he learned what an HTML tag was, and has since fallen in love with full-stack application development. He was recently an intern at Causes.com, where he developed data dashboards using D3.js for visualizing social networks. Andrew is completing his degree in Computer Science at the University of California, Santa Barbara, and when he's not programming you can find him learning to cook (badly) or hiking up a mountain somewhere.

Kevin Coughlin holds both Computer Science and Economics degrees from The College of New Jersey. He is a JavaScript developer with over two years of industry experience. At work and at home, Kevin combines HTML5 standards with cutting-edge client- and server-side technologies such as Angular.js, Backbone.js, and Node.js to produce effective modern solutions for the open web.

Kevin regularly posts tutorials on emerging web technologies on his website http://kevintcoughlin.com.
Ismini Lourentzou has a Business Administration B.Sc. and a long-standing career in the banking sector, at National Bank of Greece. Learning programming in Java in her spare time and her continuous urge for novelty, drove her to pursue a second degree in Computer Science from Athens University of Economics and Business (AUEB). During her undergraduate studies, she has participated in the Knowledge Discovery and Data Mining Cup 2012, as a member of the Data and Web Mining Group of AUEB, headed by Professor Michalis Vazirgiannis, and worked on "Automated Snippet Generation of Online Advertising", which led to a publication at CIKM 2013. Meanwhile, she also participated at ImageClef 2013 as a member of the Information Retrieval Group of AUEB, headed by Professor Theodore Kalambokis. Their participation was placed second in the Textual Ad-hoc image-based retrieval and fifth in Visual Ad-hoc image-based retrieval. Due to her love for research and programming, there was no doubt about changing her career orientation; she is currently a PhD student at University of Illinois at Urbana – Champaign, combining Machine Learning and Information Retrieval in developing intelligent information systems that will improve a user's productivity by decreasing the amount of manual involvement in searching, organizing, and understanding information from mainly textual sources. After completing her PhD, she hopes to continue working in research, and to be able to learn more and more each day.

I would like to thank my family for their support and help, for always being there to motivate me, my mother for taking care of me while my free time was nonexistent, my sister that is always protective of me, my father to being present during difficult situations. Moreover, I am thankful for my boyfriend for his everlasting patience and love and my friends for their advices and help during this process.

Pablo Navarro is a data visualization consultant from Chile. He earned his Master's degree in Applied Mathematics from École des Mines de Saint-Etienne, France. After working for some years in operations research and data analysis, he decided to specialize in data visualization for web platforms, in which he currently works. In his free time, he enjoys doing watercolor illustrations, running and reading about human evolution. His most recent works can be seen at http://pnavarre.github.io.
# Table of Contents

**Preface** 1

**Chapter 1: Getting Started with D3.js** 7

- Introduction 7
- Setting up a simple D3 development environment 8
- Setting up an NPM-based development environment 11
- Understanding D3-style JavaScript 15

**Chapter 2: Be Selective** 23

- Introduction 23
- Selecting a single element 25
- Selecting multiple elements 28
- Iterating through a selection 29
- Performing subselection 31
- Function chaining 34
- Manipulating the raw selection 36

**Chapter 3: Dealing with Data** 39

- Introduction 39
- Binding an array as data 43
- Binding object literals as data 47
- Binding functions as data 51
- Working with arrays 54
- Filtering with data 58
- Sorting with data 61
- Loading data from a server 64

**Chapter 4: Tipping the Scales** 71

- Introduction 71
- Using quantitative scales 73
- Using the time scale 78
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the ordinal scale</td>
<td>81</td>
</tr>
<tr>
<td>Interpolating a string</td>
<td>84</td>
</tr>
<tr>
<td>Interpolating colors</td>
<td>88</td>
</tr>
<tr>
<td>Interpolating compound objects</td>
<td>91</td>
</tr>
<tr>
<td>Implementing a custom interpolator</td>
<td>94</td>
</tr>
<tr>
<td><strong>Chapter 5: Playing with Axes</strong></td>
<td>101</td>
</tr>
<tr>
<td>Introduction</td>
<td>101</td>
</tr>
<tr>
<td>Working with basic axes</td>
<td>101</td>
</tr>
<tr>
<td>Customizing ticks</td>
<td>107</td>
</tr>
<tr>
<td>Drawing grid lines</td>
<td>109</td>
</tr>
<tr>
<td>Dynamic rescaling of axes</td>
<td>113</td>
</tr>
<tr>
<td><strong>Chapter 6: Transition with Style</strong></td>
<td>119</td>
</tr>
<tr>
<td>Introduction</td>
<td>119</td>
</tr>
<tr>
<td>Animating a single element</td>
<td>120</td>
</tr>
<tr>
<td>Animating multiple elements</td>
<td>123</td>
</tr>
<tr>
<td>Using ease</td>
<td>128</td>
</tr>
<tr>
<td>Using tweening</td>
<td>132</td>
</tr>
<tr>
<td>Using transition chaining</td>
<td>136</td>
</tr>
<tr>
<td>Using transition filter</td>
<td>138</td>
</tr>
<tr>
<td>Listening to transitional events</td>
<td>140</td>
</tr>
<tr>
<td>Implementing a custom interpolator</td>
<td>142</td>
</tr>
<tr>
<td>Working with timer</td>
<td>144</td>
</tr>
<tr>
<td><strong>Chapter 7: Getting into Shape</strong></td>
<td>147</td>
</tr>
<tr>
<td>Introduction</td>
<td>147</td>
</tr>
<tr>
<td>Creating simple shapes</td>
<td>149</td>
</tr>
<tr>
<td>Using a line generator</td>
<td>152</td>
</tr>
<tr>
<td>Using line interpolation</td>
<td>156</td>
</tr>
<tr>
<td>Changing line tension</td>
<td>159</td>
</tr>
<tr>
<td>Using an area generator</td>
<td>163</td>
</tr>
<tr>
<td>Using area interpolation</td>
<td>166</td>
</tr>
<tr>
<td>Using an arc generator</td>
<td>169</td>
</tr>
<tr>
<td>Implementing arc transition</td>
<td>173</td>
</tr>
<tr>
<td><strong>Chapter 8: Chart Them Up</strong></td>
<td>179</td>
</tr>
<tr>
<td>Introduction</td>
<td>179</td>
</tr>
<tr>
<td>Creating a line chart</td>
<td>181</td>
</tr>
<tr>
<td>Creating an area chart</td>
<td>188</td>
</tr>
<tr>
<td>Creating a scatter plot chart</td>
<td>192</td>
</tr>
<tr>
<td>Creating a bubble chart</td>
<td>196</td>
</tr>
<tr>
<td>Creating a bar chart</td>
<td>199</td>
</tr>
</tbody>
</table>
## Table of Contents

**Chapter 9: Lay Them Out**  
Introduction  
Building a pie chart  
Building a stacked area chart  
Building a treemap  
Building a tree  
Building an enclosure diagram  

**Chapter 10: Interacting with your Visualization**  
Introduction  
Interacting with mouse events  
Interacting with a multi-touch device  
Implementing zoom and pan behavior  
Implementing drag behavior  

**Chapter 11: Using Force**  
Introduction  
Using gravity and charge  
Generating momentum  
Setting the link constraint  
Using force to assist visualization  
Manipulating force  
Building a force-directed graph  

**Chapter 12: Know your Map**  
Introduction  
Projecting the US map  
Projecting the world map  
Building a choropleth map  

**Chapter 13: Test Drive your Visualization**  
Introduction  
Getting Jasmine and setting up the test environment  
Test driving your visualization – chart creation  
Test driving your visualization – SVG rendering  
Test driving your visualization – pixel-perfect bar rendering  

**Appendix: Building Interactive Analytics in Minutes**  
Introduction  
The crossfilter.js library  
Dimensional charting – dc.js  

Index
D3.js is a JavaScript library designed to display digital data in a dynamic graphical form. It helps you to bring data to life using HTML, SVG, and CSS. D3 allows great control over the final visual result, and it is the hottest and most powerful web-based data visualization technology on the market today.

This book is packed with practical recipes to help you learn every aspect of data visualization with D3. It is designed to provide you with all the guidance you need to get to grips with data visualization with D3. With this book, you will create breathtaking data visualization with professional efficiency and precision with the help of practical recipes, illustrations, and code samples.

This cookbook starts off by touching upon data visualization and D3 basics before gradually taking you through a number of practical recipes covering a wide range of topics you need to know about D3.

You will learn the fundamental concepts of data visualization, functional JavaScript, and D3 fundamentals including element selection, data binding, animation, and SVG generation. You will also learn how to leverage more advanced techniques such as custom interpolators, custom tweening, timers, the layout manager, force manipulation, and so on. This book also provides a number of prebuilt chart recipes with ready-to-go sample code to help you bootstrap quickly.

**What this book covers**

*Chapter 1, Getting Started with D3.js*, is designed to get you up and running with D3.js. It covers the fundamental aspects such as what D3.js is and how to set up a typical D3.js data visualization environment.

*Chapter 2, Be Selective*, teaches you one of the most fundamental tasks you need to perform with any data visualization project using D3—selection. Selection helps you target certain visual elements on the page.
Chapter 3, *Dealing with Data*, explores the most essential question in any data visualization project—how data can be represented both in programming constructs, and its visual metaphor.

Chapter 4, *Tipping the Scales*, deals with a very important subdomain of data visualization. As a data visualization developer, one key task that you need to perform over and over again is to map values in your data domain to visual domain, which is the focus of this chapter.

Chapter 5, *Playing with Axes*, explores the usage of axes' component and some related techniques commonly used in Cartesian coordinate system based visualization.

Chapter 6, *Transition with Style*, deals with transitions. "A picture is worth a thousand words," this age-old wisdom is arguably one of the most important cornerstones of data visualization. This chapter covers transition and animation support provided by D3 library.

Chapter 7, *Getting into Shape*, deals with Scalable Vector Graphics (SVG), which is a mature World Wide Web Consortium (W3C) standard widely used in visualization projects.

Chapter 8, *Chart Them Up*, explores one of the oldest and well trusted companions in data visualization—charts. Charts are well defined and well understood graphical representations of data.

Chapter 9, *Lay Them Out*, focuses on D3 Layout. D3 layouts are algorithms that calculate and generate placement information for a group of elements capable of generating some of the most complex and interesting visualization.

Chapter 10, *Interacting with your Visualization*, focuses on D3 human visualization interaction support, or in other words how to add computational steering capability to your visualization.

Chapter 11, *Using Force*, covers one of the most fascinating aspects of D3—Force. Force simulation is one of the most awe-inspiring techniques that you can add to your visualization.

Chapter 12, *Know your Map*, introduces basic D3 cartographic visualization techniques and how to implement a fully functional geographic visualization in D3.

Chapter 13, *Test Drive your Visualization*, teaches you to implement your visualization like a pro with Test Driven Development (TDD).

Appendix, *Building Interactive Analytics in Minutes* serves as an introduction to Crossfilter.js and dc.js on interactive dimensional charting.
What you need for this book

- A text editor: To edit and create HTML, CSS, and JavaScript files
- A web browser: A modern web browser (Firefox 3, IE 9, Chrome, Safari 3.2 and above)
- A local HTTP server: You need a local HTTP server to host data file for some of the more advanced recipes in this book. We will cover how to set up a Node or Python based simple HTTP server in the first chapter.
- Git client (Optional): If you would like to check out the recipe source code directly from our Git repository, you need a Git client installed on your computer.

Who this book is for
If you are a developer or an analyst familiar with HTML, CSS, and JavaScript, and you wish to get the most out of D3, then this book is for you. This book can also serve as a desktop quick-reference guide for experienced data visualization developers.

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

Code words in text are shown as follows: "We can select HTML elements through the use of the d3.select function."

A block of code is set as follows:

```javascript
instance.description = function (d) {
  if (!arguments.length) d;
  description = d;
  return instance;
};
```

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

```javascript
instance.description = function (d) {
  if (!arguments.length) d;
  description = d;
  return instance;
};
```

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

```bash
> npm install http-server -g
```
New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the Next button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

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

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via 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 on www.packtpub.com/authors.

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

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

Piracy
Piracy of copyright 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
You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.
1

Getting Started with D3.js

In this chapter we will cover:

- Setting up a simple D3 development environment
- Setting up an NPM-based development environment
- Understanding D3-style JavaScript

Introduction

This chapter is designed to get you up and running with D3.js, covering fundamental aspects, such as what D3.js is, and how to set up a typical D3.js data visualization environment. One particular section is also devoted in covering some lesser known areas of JavaScript that D3.js relies heavily on.

What is D3? D3 refers to Data-Driven Documents, and according to the official D3 Wiki:

D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

D3 Wiki (2013, August)
In a sense, D3 is a specialized JavaScript library that allows you to create amazing data visualizations using a simpler (data driven) approach by leveraging existing web standards. D3.js was created by Mike Bostock (http://bost.ocks.org/mike/) and superseded his previous work on a different JavaScript data visualization library called Protovis. For more information on how D3 was created and on the theory that influenced both Protovis and D3.js, please check out links in the following information box. Here in this book we will focus more on how to use D3.js to power your visualization. Initially, some aspects of D3 may be a bit confusing due to its different approach to data visualization using JavaScript. I hope that over the course of this book, a large number of topics, both basic and advanced, will make you comfortable and effective with D3. Once properly understood, D3 can improve your productivity and expressiveness with data visualizations by orders of magnitude.

For more formal introduction to the idea behind D3 see the *Declarative Language Design for Interactive Visualization* paper published by Mike Bostock on IEEE InfoVis 2010 http://vis.stanford.edu/papers/protovis-design.

If you are interested to know how D3 came about, I recommend you to check out the *D3: Data-Driven Document* paper published by Mike Bostock on IEEE InfoVis 2011 at http://vis.stanford.edu/papers/d3.

Protovis, the predecessor of D3.js, also created by Mike Bostock and Jeff Heer of the Stanford Visualization Group can be found at http://mbostock.github.io/protovis/.

---

### Setting up a simple D3 development environment

First thing you need when starting a D3 powered data visualization project is a working development environment. In this recipe, we will show you how a simple D3 development environment can be set up within minutes.

#### Getting Ready

Before we start, make sure you have your favorite text editor installed and ready on your computer.
We'll start by downloading D3.js:

1. Download the latest stable version of D3.js from http://d3js.org/. You can download the archived, older releases from https://github.com/mbostock/d3/tags. Additionally, if you are interested in trying out the bleeding edge D3 build on master branch, then you can fork https://github.com/mbostock/d3.

2. Once downloaded and unzipped, you will find three files `d3.v3.js`, `d3.v3.min.js`, and its license in the extracted folder. For development it is recommended to use `d3.v3.js`, the "non-uglified" (minimized) version, since it can help you trace and debug JavaScript inside D3 library. Once extracted place the `d3.v3.js` file in the same folder with an `index.html` file containing the following HTML:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Simple D3 Dev Env</title>
  <script type="text/javascript" src="d3.v3.js"></script>
</head>
<body>
</body>
</html>
```

If you download D3 from a source or a tagged version, the JavaScript file name will be slightly different. Instead of `d3.v3.js`, it will simply be called `d3.js`.

This is all you need to create, in its simplest form, a D3-powered data visualization development environment. With this setup you can essentially open the HTML file using your favorite text editor to start your development and also view your visualization by opening the file in your browser.

The source code for this recipe can be found at https://github.com/NickQizhu/d3-cookbook/tree/master/src/chapter1/simple-dev-env.
Getting Started with D3.js

How it works...

D3 JavaScript library is very self-sufficient. It has no dependency on any JavaScript library than other what your browser already provides. In fact, it can even be used in a non-browser environment such as Node.js with some minimum setup (I will cover this in more detail in later chapters).

If your visualization's target browser environment includes Internet Explorer 9, it is recommended to use the compatibility library Aight, which can be found at https://github.com/shawnbot/aight, and Sizzle selector engine at http://sizzlejs.com/.

Having the following character encoding instruction in header section is critical:

```html
<meta charset="utf-8">
```

The character encoding instructs browsers and validators what set of characters to use when rendering web pages. Otherwise your browser will not be able to load D3 JavaScript library since D3 uses utf-8 character for certain symbols such as π.

D3 is completely open source, and it is open sourced under a custom license agreement created by its author Michael Bostock. This license is pretty similar to the popular MIT license with only one exception where it explicitly states that Michael Bostock's name cannot be used to endorse or promote products derived from this software without permission.

There's more...

Throughout this cookbook numerous recipe code examples will be provided. All example source code are provided and hosted on GitHub (https://github.com/) a popular open source social coding repository platform.

How to get source code

The easiest way to get all the recipe source code that you need is to clone the Git repository (https://github.com/NickQiZhu/d3-cookbook) for this book. If you are not planning to set up a development environment for the recipes then you can safely skip this section.
If you are not familiar with Git, clone is similar to the check-out concept in other versions of control software. However cloning does a lot more than simply checking out the files. It also copies all branches and histories to your local machine effectively cloning the entire repository to your local machine so you can work completely offline with this cloned repository in your own environment.


Another popular way to get Git and GitHub working is to install the GitHub client, which gives you a richer set of features than simply Git. However, at the time of writing, GitHub only offered client software for Windows and Mac OS.

GitHub for Mac: http://mac.github.com/.

Once the Git client is installed, simply issuing the following command will download all recipe source code to your computer:

> git clone git://github.com/NickQiZhu/d3-cookbook.git

Or if you choose to use GitHub client, then simply click the Fork button on the repository page https://github.com/NickQiZhu/d3-cookbook. This will make this repository appear in your GitHub client.

### Setting up an NPM-based development environment

When you are working on a more complex data visualization project that requires the use of a number of JavaScript libraries, the simple solution we discussed before might become a bit clumsy and unwieldy. In this section, we will demonstrate an improved setup using Node Packaged Modules (NPM)—a de facto JavaScript library repository management system. If you are as impatient as me and want to get to the meaty part of the book—the recipes—you can safely skip this section and come back when you need to set up a more production-ready environment for your project.
Getting Started with D3.js

Getting Ready

Before we start please make sure you have NPM properly installed. NPM comes as part of the Node.js installation. You can download Node.js from http://nodejs.org/download/. Select the correct Node.js binary build for your OS. Once installed the npm command will become available in your terminal console.

> npm -v
1.2.14

The preceding command prints out the version number of your NPM client indicating the installation is successful.

How to do it...

With NPM installed, now we can create a package descriptor file to automate some of the manual setup steps.

1. First, under your project folder, create a file named package.json containing the following code:

```json
{
    "name": "d3-project-template",
    "version": "0.1.0",
    "description": "Ready to go d3 data visualization project template",
    "keywords": ["data visualization", "d3"],
    "homepage": "<project home page>",
    "author": {
        "name": "<your name>",
        "url": "<your url>"
    },
    "repository": {
        "type": "git",
        "url": "<source repo url>"
    },
    "dependencies": {
        "d3": "3.x"
    },
    "devDependencies": {
        "uglify-js": "2.x"
    }
}
```
2. Once the package.json file is defined, you can simply run:

> npm install

**How it works...**

Most of the fields in the package.json file are for informational purpose only, such as the name, description, homepage, author, and the repository. The name and version field will be used if you decide to publish your library into an NPM repository in the future. What we really care about, at this point, is the dependencies and devDependencies fields.

- The dependencies field describes the runtime library dependencies that your project has, meaning the libraries your project needs to run properly in a browser. In this simple example we only have one dependency on d3. d3 is the name that D3 library is published under in the NPM repository. The version number 3.x signifies that this project is compatible with any version 3 releases, and NPM should retrieve the latest stable version 3 build to satisfy this dependency.

D3 is a self-sufficient library with zero external runtime dependency. However, this does not mean that it cannot work with other popular JavaScript libraries. I regularly use D3 with other libraries to make my job easier, for example, JQuery, Zepto.js, Underscore.js, and Backbone.js.

- The devDependencies field describes development time (compile time) library dependencies. What this means is that, libraries specified under this category are only required in order to build this project, and not required for running your JavaScript project.

Detailed NPM package JSON file documentation can be found at [https://npmjs.org/doc/json.html](https://npmjs.org/doc/json.html).

Executing the npm install command will automatically trigger NPM to download all dependencies that your project requires including your dependencies' dependencies recursively. All dependency libraries will be downloaded into node_modules folder under your project root folder. When this is done you can just simply create your HTML file as it has been shown in the previous recipe, and load your D3 JavaScript library directly from node_modules/d3/d3.js.

The source code for this recipe with an automated build script can be found at [https://github.com/NickQiZhu/d3-cookbook/tree/master/src/chapter1/npm-dev-env](https://github.com/NickQiZhu/d3-cookbook/tree/master/src/chapter1/npm-dev-env).
Relying on NPM is a simple and yet effective way to save you from all the trouble of downloading JavaScript libraries manually and the constant need of keeping them up-to-date. However, an astute reader might have already noticed that with this power we can easily push our environment setup to the next level. Imagine if you are building a large visualization project where thousands of lines of JavaScript code will be created, obviously our simple setup described here is no longer sufficient. However modular JavaScript development by itself can fill an entire book; therefore we are not going to try to cover this topic since our focus is on data visualization and D3. If you are interested please refer the source code for this recipe where it is demonstrated how a more modular approach can be implemented on top of what we described here with a simple automated build script. In later chapters, when unit test related recipes are discussed, we will expand the coverage on this topic to show how our setup can be enhanced to run automated unit tests.

**There's more...**

Though in previous sections, it was mentioned that you can just open the HTML page that you have created using your browser to view your visualization result directly, this approach does have its limitations. This simple approach stops working once we need to load data from separate data file (this is what we will do in later chapters and it is also the most likely case in your daily working environment) due to the browser's built-in security policy. To get around this security constraint it is highly recommended that you set up a local HTTP server so your HTML page and the data file can be served from this server instead of loaded from file system directly.

**Setup a local HTTP server**

There are probably a dozen ways to set up an HTTP server on your computer based on which operating system you use and which software package you decide to use to act as an HTTP server. Here I will attempt to cover some of the most popular setups.

**Python Simple HTTP Server**

This is my favorite for development and fast prototyping. If you have Python installed on your OS, which is usually the case with any Unix/Linux/Mac OS distro, then you can simply type this command in your terminal:

```
> python -m SimpleHTTPServer 8888
```

Or with newer Python distribution:

```
> python -m http.server
```

This little python program will launch an HTTP server and start serving any file right from the folder where this program is launched. This is by far the easiest way to get an HTTP server running on any OS.
If you don’t have python installed on your computer yet, you can get it from http://www.python.org/getit/. It works on all modern OS including Windows, Linux and Mac.

**Node.js HTTP Server**

If you have Node.js installed, perhaps as part of the development environment setup exercise we did in the previous section, then you can simply install the http-server module. Similar to Python Simple HTTP Server, this module will allow you to launch a lightweight HTTP server from any folder and starting serving pages right away.

First install the http-server module:

```
> npm install http-server -g
```

The `-g` option in this command will install http-server module globally so it will become available in your command line terminal automatically. Once this is done, then you can launch the server from any folder you are in by simply issuing the following command:

```
> http-server .
```

This command will launch a Node.js powered HTTP server on the default port 8080 or if you want you can use the `-p` option to provide a custom port number for it.

If you are running the `npm install` command on Linux/Unix/Mac OS, you will need to run the command in `sudo` mode or as root in order to use the `-g` global installation option.

**Understanding D3-style JavaScript**

D3 is designed and built using functional style JavaScript which might come as to seem unfamiliar or even alien to someone who is more comfortable with the procedural or object-oriented JavaScript styles. This recipe is designed to cover some of the most fundamental concepts in functional JavaScript required to make sense of D3, and furthermore enable you to write your visualization code in D3 style.

**Getting ready**

Open your local copy of the following file in your web browser: https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter1/functional-js.html
Let's dig a little deeper into the good part of JavaScript—the more functional side. Take a look at the following code snippet:

```javascript
function SimpleWidget(spec) {
    var instance = {}; // <-- A

    var headline, description; // <-- B

    instance.render = function () { // <-- C
        var div = d3.select('body').append('div');

        div.append('h3').text(headline); // <-- D
        div.attr('class', 'box').attr('style', 'color:' + spec.color).append('p').text(description); // <-- E

        return instance; // <-- F
    }

    instance.headline = function (h) { // <-- G
        if (!arguments.length) h;
        headline = h;
        return instance; // <-- H
    }

    instance.description = function (d) { // <-- I
        if (!arguments.length) d;
        description = d;
        return instance;
    }

    return instance; // <-- F
}

var widget = SimpleWidget({color: "#6495ed"})
    .headline("Simple Widget")
    .description("This is a simple widget demonstrating functional javascript.");
widget.render();
```
This code snippet generates the following simple widget on your web page:

![Simple Widget]

A Simple Widget with functional JavaScript

**How it works...**

Despite its simplicity, the interface of this widget has this undeniable similarity to D3 style of JavaScript. This is not by coincidence but rather by leveraging a JavaScript programming paradigm called functional objects. Like many interesting topics, this is another topic that can fill an entire book by itself; nevertheless I will try to cover the most important and useful aspects of this particular paradigm in this section so you the reader cannot only understand D3's syntax but will also be able to create a library in this fashion. As stated on D3's project Wiki this functional programming style gives D3 much of its flexibility:

> D3's functional style allows code reuse through a diverse collection of components and plugins.

*D3 Wiki (2013, August)*

**Functions are objects**

Functions in JavaScript are objects. Like any other object, function is just a collection of name and value pair. The only difference between a function object and a regular object is that function can be invoked and additionally associated with two hidden properties: function context and function code. This might come as a surprise and unnatural, especially if you are coming from a more procedural programming background. Nevertheless this is the critical insight most of us need, to make sense of some of the strange ways that D3 uses function.

> JavaScript in its current form is generally considered not very object oriented, however, function object is probably one aspect where it outshines some of the other more object-oriented cousins.
Now with this insight in mind, let's take a look at the code snippet again:

```javascript
var instance = {}; // <-- A

var headline, description; // <-- B

instance.render = function () {
    var div = d3.select('body').append("div");

    div.append("h3").text(headline); // <-- C

    div.attr("class", "box")
        .attr("style", "color:" + spec.color) // <-- D
        .append("p")
        .text(description); // <-- E

    return instance; // <-- F
};
```

At line marked as A, B, and C we can clearly see that instance, headline, and description are all internal private variables belonging to the SimpleWidget function object. While the render function is a function associated with the instance object which itself is defined as an object literal. Since functions are just an object it can also be stored in an object/function, other variables, arrays, and being passed as function arguments. The result of the execution of function SimpleWidget is the returning of object instance at line I.

```javascript
function SimpleWidget(spec) {
    ...
    return instance; // <-- I
}
```

The render function uses some of the D3 functions that we have not covered yet, but let's not pay too much attention to them for now since we will cover each of them in depth in the next couple of chapters. Also they basically just render the visual representation of this widget, not having much to do with our topic on hand.

**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.
**Static variable scoping**

Curious readers are probably asking by now how the variable scoping is resolved in this example since the render function has seemingly strange access to not only the instance, headline, and description variables but also the spec variable that is passed into the base SimpleWidget function. This seemingly strange variable scoping is actually determined by a simple static scoping rule. This rule can be thought as the following: whenever searching for a variable reference, variable search will be first performed locally. When variable declaration is not found (as in the case of headline on line C) then the search continues to the parent object (in this case SimpleWidget function is its static parent and headline variable declaration is found at line B). If still not found, then this process will continue recursively to the next static parent so on and so forth till it reaches global variable definition, if still not found then a reference error will be generated for this variable. This scoping behavior is very different from variable resolution rules in some of the most popular languages such as Java and C#; it might take some time to get used to, however don't worry too much about it if you still find it confusing. With more practice and keeping static scoping rule in mind you will be comfortable with this kind of scoping in no time.

One word of caution here—again for folks from Java and C# backgrounds—is that JavaScript does not implement block scoping. The static scoping rule we described only applies to function/object but not at the block level.

```
for(var i = 0; i < 10; i++){
    for(var i = 0; i < 2; i++){
        console.log(i);
    }
}
```

You might be inclined to think this code should produce 20 numbers. However in JavaScript this code creates an infinite loop. This is because JavaScript does not implement block scoping so the i in the inner loop is the same i used by the the outer loop. Therefore it gets reset by the inner loop thus and can never end the outer loop.

This pattern is usually referred as functional when compared with the more popular prototype-based **Pseudo-classical pattern**. The advantage of the functional pattern is that it provides a much better mechanism for information hiding and encapsulation since the private variables—in our case the headline and description variables—are only accessible by nested functions via the static scoping rule therefore the object returned by the SimpleWidget function is flexible yet more tamper-proof and durable.

*If we create an object in the functional style, and if all of the methods of the object make no use of this, then the object is durable. A durable object is simply a collection of functions that act as capabilities.*

*(Crockfort D. 2008)*
Variable-parameter function

Something strange happens on line G:

```javascript
instance.headline = function (h) {
  if (!arguments.length) h; // <-- G
  headline = h;
  return instance; // <-- H
};
```

You might be asking where this `arguments` variable on line G came from. It was never defined anywhere in this example. The `arguments` variable is a built-in hidden parameter that is available to functions when they are invoked. The `arguments` variable contains all arguments for a function invocation in an array.

In fact, `arguments` is not really a JavaScript array object. It has length and can be accessed using an index, however it does not have many of the methods associated with a typical JavaScript array object such as `slice` or `concat`. When you need to use a standard JavaScript array method on `arguments`, you need to use the apply invocation pattern:

```javascript
var newArgs = Array.prototype.slice.apply(arguments);
```

This hidden parameter when combined with the ability to omit function argument in JavaScript allows you to write a function like `instance.headline` with unspecified number of parameters. In this case, we can either have one argument `h` or none. Because `arguments.length` returns 0 when no parameter is passed; therefore the `headline` function returns `h` if no parameter is passed, otherwise it turns into a setter if parameter `h` is provided. To clarify this explanation let's take a look at the following code snippet:

```javascript
var widget = SimpleWidget({color: "#6495ed"}).headline("Simple Widget"); // set headline
console.log(widget.headline()); // prints "Simple Widget"
```

Here you can see how `headline` function can be used as both setter and getter with different parameters.

Function chaining

The next interesting aspect of this particular example is the capability of chaining functions to each other. This is also the predominant function invocation pattern that the D3 library deploys since most of the D3 functions are designed to be chainable to provide a more concise and contextual programming interface. This is actually quite simple once you understand the variable-parameter function concept. Since a variable-parameter function—such as the `headline` function—can serve as setter and getter at the same time, then returning the `instance` object when it is acting as a setter allows you to immediately invoke another function on the invocation result; hence the chaining.
Let's take a look at the following code:

```javascript
var widget = SimpleWidget({color: "#6495ed"})
       .headline("Simple Widget")
       .description("This is ...")
       .render();
```

In this example, the `SimpleWidget` function returns the *instance object* (as on line I). Then, the `headline` function is invoked as a setter, which also returns the *instance object* (as on line H). The `description` function can then be invoked directly on return which again returns the *instance object*. Then finally the `render` function can be called.

Now with the knowledge of functional JavaScript and a working ready-to-go D3 data visualization development environment, we are ready to dive into the rich concepts and techniques that D3 has to offer. However before we take off, I would like to cover a few more important areas—how to find and share code and how to get help when you are stuck.

**There's more...**

Let's take a look at some additional helpful resources.

**Finding and sharing code**

One of the great things about D3 when compared with other visualization options is that it offers a wealth of examples and tutorials that you can draw your inspiration from. During the course of creating my own open source visualization charting library and the creation of this book, I had drawn heavily on these resources. I am going to list some of the most popular options available in this aspect. This list is by no means a comprehensive directory but rather a starting place for you to explore:

- The D3 gallery (https://github.com/mbostock/d3/wiki/Gallery) contains some of the most interesting examples that you can find online regarding D3 usage. It contains examples on different visualization charts, specific techniques, and some interesting visualization implementations in the wild, among others.
- BioVisualize (http://biovisualize.github.io/d3visualization) is another D3 gallery with categorization, to help you find your desired visualization example online quickly.
- The D3 tutorials page (https://github.com/mbostock/d3/wiki/Tutorials) contains a collection of tutorials, talks and slides created by various contributors over time, to demonstrate in detail how to use a lot of D3 concepts and techniques.
- D3 plugins (https://github.com/d3/d3-plugins). Maybe some features are missing in D3 for your visualization needs? Before you decide to implement your own, make sure to check out D3 plugin repository. It contains a wide variety of plugins that provide some of the common and, sometimes, uncommon features in the visualization world.
Getting Started with D3.js

- The D3 API (https://github.com/mbostock/d3/wiki/API-Reference) is very well documented. This is where you can find detailed explanations for every function and property that the D3 library has to offer.

- Mike Bostok’s Blocks (http://bl.ocks.org/mbostock) is a D3 example site, where some of the more intriguing visualization example can be found and which is maintained by its author Mike Bostock.

- JS Bin (http://jsbin.com/ugacud/1/edit) is a pre-built D3 test and experiment environment completely hosted online. You can easily prototype a simple script using this tool or share your creation with other members in the community.

- JS Fiddle (http://jsfiddle.net/qAHC2/) is similar to JS Bin; it also is a hosted-online JavaScript code prototyping and sharing platform.

How to get help

Even with all the examples, tutorial, and cookbook like this, you might still run into challenges when creating your visualization. Good news here is that D3 has a broad and active support community. Simply "googling" your question can most often yield a satisfying answer. Even if it does not, don't worry; D3 has a robust community-based support:

- D3.js on Stack Overflow (http://stackoverflow.com/questions/tagged/d3.js): Stack Overflow is the most popular community-based free Q&A site for technologists. D3 is a specific category on the Stack Overflow site to help you reach the experts and get an answer to your question quickly.

- The D3 Google group (https://groups.google.com/forum/?fromgroups#!forum/d3-js): This is the official user group for not just D3 but also other related libraries in its ecosystem.
In this chapter we will cover:

- Selecting a single element
- Selecting multiple elements
- Iterating through a selection
- Performing subselection
- Function chaining
- Manipulating raw selection

Introduction

One of the most fundamental tasks that you need to perform with any data visualization project using D3 is selection. Selection helps you target certain visual elements on the page. If you are already familiar with the W3C standardized CSS selector or other similar selector APIs provided by popular JavaScript libraries, such as jQuery and Zepto.js, then you will find yourself right at home with D3's selection API. Don't worry if you haven't used selector API before, this chapter is designed to cover this topic in steps with the help of some very visual recipes; it will cover pretty much all common use cases for your data visualization needs.

Introducing selection: Selector support has been standardized by W3C so all modern web browsers have built-in support for the selector API. However the basic W3C selector API has its limitations when it comes to web development, especially in the data visualization realm. The standard W3C selector API only provides selector but not selection. What this means is that the selector API helps you to select element(s) in your document, however, to manipulate the selected element(s) you still need to loop through each element, in order to manipulate the selected element(s). Consider the following code snippet using the standard selector API:

```javascript
var i = document.querySelectorAll("p").iterator();
```
var e;
while(e = i.next()){
    // do something with each element selected
    console.log(e);
}

The preceding code essentially selects all `<p>` elements in the document and then iterates through each element to perform some task. This can obviously get tedious quickly, especially when you have to manipulate many different elements on the page constantly, which is what we usually do in data visualization projects. This is why D3 introduced its own selection API, making development less of a chore. For the rest of this chapter we will cover how D3's selection API works as well as some of its powerful features.

**CSS3 selector basics:** Before we dive into D3's selection API, some basic introduction on the W3C level-3 selector API is required. If you are already comfortable with CSS3 selectors, feel free to skip this section. D3's selection API is built based on the level-3 selector or more commonly known as the CSS3 selector support. In this section, we plan to go through some of the most common CSS3 selector syntax that are required to understand D3 selection API.

- `#foo`: select element with `foo` as the value of `id`
  `<div id="foo">`
- `foo`: select element `foo`
  `<foo>`
- `.*foo`: select elements with `foo` as the value of `class`
  `<div class="foo">`
- `[foo=goo]`: select elements with the `foo` attribute value and set it to `goo`
  `<div foo="goo">`
- `foo goo`: select the `goo` element inside the `foo` element
  `<foo><goo></foo>`
- `foo#goo`: select the `foo` element set `goo` as the value of `id`
  `<foo id="goo">`
- `foo.goo`: select the `foo` element with `goo` as the value of `class`
  `<foo class="goo">`
- `foo:first-child`: select the first child of the `foo` elements
  `<foo> // <-- this one
  `<foo>
  `<foo>`
CSS3 selector is a pretty complex topic. Here we have only listed some of the most common selectors that you will need to understand and to be effective when working with D3. For more information on this topic please visit the W3C level-3 selector API document http://www.w3.org/TR/css3-selectors/.

If you are targeting an older browser that does not support selector natively, you can include Sizzle before D3 for backwards-compatibility. You can find Sizzle at http://sizzlejs.com/.

Currently the next generation selector API level-4 is in draft stage with W3C. You can have a peek at what it has to offer and its current draft here at http://dev.w3.org/csswg/selectors4/

Major browser vendors have already started implementing some of the level-4 selectors if you are interested to find out the level of support in your browser, try out this handy website http://css4-selectors.com/browser-selector-test/.

Selecting a single element

It is very common that sometimes you need to select a single element on a page to perform some visual manipulation. This recipe will show you how to perform a targeted single element selection in D3 using CSS selector.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

Let's select something (a paragraph element perhaps) and produce the classic "hello world" on screen.

```html
<p id="target"></p> <!-- A -->
<script type="text/javascript">
</script>
```
Be Selective

```javascript
d3.select("p#target") // <-- B
.text("Hello world!"); // <-- C
</script>
```

This recipe simply produces a **Hello world!** on your screen.

**How it works...**

The `d3.select` command is used to perform a single element selection in D3. This method accepts a string representing a valid CSS3 selector or an element object if you already have a reference to the element you want to select. The `d3.select` command returns a D3 selection object on which you can chain modifier functions to manipulate the attribute, content or inner HTML of this element.

More than one element can be selected using the selector provided only the first element is returned in the selection.

In this example, we simply select the paragraph element with `target` as the value of `id` at line B, and then set its textual content to **Hello world!** on line C. All D3 selections support a set of standard modifier functions. The `text` function we have shown here is one of them. The following are some of the most common modifier functions you will encounter throughout this book:

- **The `selection.attr` function:** This function allows you to retrieve or modify a given attribute on the selected element(s).
  ```javascript
  // set foo attribute to goo on p element
d3.select("p").attr("foo", "goo");
  // get foo attribute on p element
  d3.select("p").attr("foo");
  ```

- **The `selection.classed` function:** This function allows you to add or remove CSS classes on the selected element(s).
  ```javascript
  // test to see if p element has CSS class goo
d3.select("p").classed("goo");
  // add CSS class goo to p element
  d3.select("p").classed("goo", true);
  // remove CSS class goo from p element. classed function
  // also accepts a function as the value so the decision
  // of adding and removing can be made dynamically
  d3.select("p").classed("goo", function(){return false;});
  ```
The `selection.style` function: This function lets you set the CSS style with a specific name to the specific value on the selected element(s).

```javascript
// get p element's style for font-size
d3.select("p").style("font-size");
// set font-size style for p to 10px
d3.select("p").style("font-size", "10px");
// set font-size style for p to the result of some calculation. style function also accepts a function as the value can be produced dynamically
d3.select("p").style("font-size", function(){
    return normalFontSize + 10;
});
```

The `selection.text` function: This function allows you access and set the text content of the selected element(s).

```javascript
// get p element's text content
d3.select("p").text();
// set p element's text content to "Hello"
d3.select("p").text("Hello");
// text function also accepts a function as the value, thus allowing setting text content to some dynamically produced message
d3.select("p").text(function(){
    var model = retrieveModel();
    return model.message;
});
```

The `selection.html` function: This function lets you modify the element's inner HTML content.

```javascript
// get p element's inner html content
d3.select("p").html();
// set p element's inner html content to "<b>Hello</b>"
d3.select("p").text("<b>Hello</b>");
// html function also accepts a function as the value, thus allowing setting html content to some dynamically produced message
d3.select("p").text(function(){
    var template = compileTemplate();
    return template();
});
```

These modifier functions work on both single-element and multi-element selection results. When applied to multi-element selections, these modifications will be applied to each and every selected element. We will see them in action in other, more complex recipes covered in the rest of this chapter.
When a function is used as a value in these modifier functions, there are actually some built-in parameters being passed to these functions to enable data-driven calculation. This data-driven approach is what gives D3 its power and its name (Data-Driven Document) and will be discussed in detail in the next chapter.

Selecting multiple elements

Often selecting a single element is not good enough, but rather you want to apply certain change to a set of elements on the page simultaneously. In this recipe, we will play with D3 multi-element selector and its selection API.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

This is what the d3.selectAll function is designed for. In this code snippet, we will select three different div elements and enhance them with some CSS classes.

```html
    <div></div>
    <div></div>
    <div></div>

    <script type="text/javascript">
        d3.selectAll("div") // <-- A
            .attr("class", "red box"); // <-- B
    </script>
```

This code snippet produces the following visual:

Multi-element selection
How it works...

First thing you probably would notice in this example is how similar the usage of D3 selection API is when compared to the single-element version. This is one of the powerful design choices of the D3 selection API. No matter how many elements you are targeting, whether one or many, the modifier functions are exactly the same. All modifier functions we mentioned in the previous section can be applied directly to multi-element selection, in other words D3 selection is set-based.

Now with that being said, let's take a closer look at the code example shown in this section, though it is generally pretty simple and self-descriptive. At line A, the `d3.selectAll` function is used to select all the `div` elements on the page. The return of this function call is a D3 selection object that contains all three `div` elements. Immediately after that, on line B, the `attr` function was called on this selection to set the `class` attribute to `red box` for all three `div` elements. As shown in this example, the selection and manipulation code is very generic, and will not change if now we have more than three `div` elements on the page. This seems to be an insignificant convenience for now, but in later chapters we will show how this convenience can make your visualization code simpler and easier to maintain.

Iterating through a selection

Sometimes it is handy to be able to iterate through each element within a selection and modify each element differently according to their position. In this recipe, we will show you how this can be achieved using D3 selection iteration API.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

D3 selection object provides simple iterator interface to perform iteration in a similar fashion as how you will iterate through a JavaScript array. In this example we will iterate through three selected `div` elements we worked with in the previous recipe and annotate them with an index number.

```html
<script type="text/javascript">

<div></div>
<div></div>
<div></div>
</script>
```
Selections are essentially arrays albeit with some enhancement. We will explore raw selection in its array form and how to work with it in later sections.

The preceding code snippet produces the following visual:

```
0 1 2
```

Selection iteration

**How it works...**

This example is built on top of what we have already seen in the previous section. Additional to selecting all the `div` elements on the page at line A and setting their class attributes at line B, in this example we call the `each` function on the selection to demonstrate how you can iterate through a multi-element selection and process each element respectively.

```
This form of calling a function on another function's return is called Function Chaining. If you are unfamiliar with this kind of invocation pattern, please review Chapter 1, Getting Started with D3.js, where the topic was explained.
```

**The selection.each(function) function:** The `each` function takes an iterator function as its parameter. The given iterator function can receive two optional parameters `d` and `i` with one more hidden parameter passed in as the `this` reference which points to the current DOM element object. The first parameter `d` represents the datum bound to this particular element (if this sounds confusing to you, don't worry we will cover data binding in depth in the next chapter). The second parameter `i` is the index number for the current element object being iterated through. This index is zero-based meaning it starts from zero and increments each time a new element is encountered.
The selection.append(name) function: Another new function introduced in this example is the append function. This function creates a new element with the given name and appends it as the last child of each element in the current selection. It returns a new selection containing the newly appended element. Now with this knowledge, let's take a closer look at the code example in this recipe.

```
d3.selectAll("div") // <-- A
  .attr("class", "red box") // <-- B
  .each(function (d, i) { // <-- C
    d3.select(this).append("h1").text(i); // <-- D
  });
```

The iterator function is defined on line C with both d and i parameters. Line D is a little bit more interesting. At the beginning of line D, the this reference is wrapped by the d3.select function. This wrapping essentially produces a single element selection containing the current DOM Element. Once wrapped, the standard D3 selection manipulation API can then be used on d3.select(this). After that the append("h1") function is called on the current element selection which appends a newly created h1 element to the current element. Then it simply sets the textual content of this newly created h1 element to the index number of the current element. This produces the visual of numbered boxes as shown in this recipe. Again you should notice that the index starts from 0 and increments 1 for each element.

The DOM element object itself has a very rich interface. If you are interested to know more about what it can do in an iterator function, please refer to the DOM element API at https://developer.mozilla.org/en-US/docs/DOM/element.

---

Performing subselection

It is quite common that you will need to perform scoped selection when working on visualization. For example, selecting all div elements within a particular section element is one use case of such scoped selection. In this recipe, we will demonstrate how this can be achieved with different approaches and their advantages and disadvantages.

Getting ready

Open your local copy of the following file in your web browser:

https://github.com/NickQ1Zhu/d3-cookbook/blob/master/src/chapter2/sub-selection.html
Be Selective

How to do it...

The following code example selects two different div elements using two different styles of subselection supported by D3.

```html
<section id="section1">
  <div>
    <p>blue box</p>
  </div>
</section>
<section id="section2">
  <div>
    <p>red box</p>
  </div>
</section>

<script type="text/javascript">
  d3.select("#section1 > div") // <-- A
     .attr("class", "blue box");

  d3.select("#section2") // <-- B
     .select("div") // <-- C
     .attr("class", "red box");
</script>

This code generates the following visual output:

![Visual output](image)

How it works...

Though producing the same visual effect, this example demonstrates two very different subselection techniques. We will discuss them separately here so you can understand their pros and cons as well as when to use one versus the other.

Selector level-3 combinators: On line A, d3.select is used with a special looking string which consists of one tag name connected with another one using a greater-than sign (U+003E, >). This syntax is called combinators (the greater-than sign here indicates it is a child combinator). Level-3 selector supports a few different kinds of structural combinators. Here we are going to give a quick introduction to the most common ones.
The descendant combinator: This combinator has the syntax like selector selector.

The descendant combinator, as suggested by its name, is used to describe a loose parent-child relationship between two selections. The reason why it is called loose parent-child relationship is that the descendant combinatory does not care if the second selection is a child or a grandchild or a great-grandchild of the parent selection. Let's look at some examples to illustrate this loose relationship concept.

```
<div>
  <span>
    The quick <em>red</em> fox jumps over the lazy brown dog
  </span>
</div>
```

Using the following selector:

```
div em
```

It will select the em element since div is the ancestor of the em element and em is a descendent of the div element.

Child combinator: This combinator has the syntax like selector > selector.

The child combinator offers a more restrictive way to describe a parent-child relationship between two elements. A child combinator is defined using a greater-than sign (U+003E, >) character separating two selectors.

The following selector:

```
span > em
```

It will select the em element since em is a direct child of the span element in our example. While the selector div > em will not produce any valid selection since em is not a direct child of the div element.

The level-3 selector also supports sibling combinators however since it is less common we are not going to cover it here; interested readers can refer to W3C level-3 selector documentation http://www.w3.org/TR/css3-selectors/#sibling-combinators

The W3C level-4 selector offers some interesting additional combinators, that is, following-sibling and reference combinators that can yield some very powerful target selection capability; see http://dev.w3.org/csswg/selectors4/#combinators for more details.
Be Selective

**The D3 subselection**: On line B and C, a different kind of subselection technique was used. In this case a simple D3 selection was made first on line B selecting section #section2 element. Immediately afterwards another select was chained to select a div element on line C. This kind of chained selection defines a scoped selection. In plain English, this basically means to select a div element that is nested under #section2. In semantics, this is essentially similar to using a descendant combinator #section2 div. However, the advantage of this form of subselection is that since the parent element is separately selected therefore it allows you to handle the parent element before selecting the child element. To demonstrate this, let's take a look at the following code snippet:

```javascript
  d3.select("#section2") // <-- B  
  .style("font-size", "2em") // <-- B-1  
  .select("div") // <-- C  
  .attr("class", "red box");
```

As shown in the preceding code snippet, now you can see before we select the div element, we can apply a modifier function to #section2 on line B-1. This flexibility will be further explored in the next section.

**Function chaining**

As we have seen so far, the D3 API is completely designed around the idea of function chaining. Therefore it almost forms a Domain Specific Language (DSL) for building HTML/SVG elements dynamically. In this code example, we will take a look at how the entire body structure of the previous example can be constructed using D3 alone.

If DSL is a new concept for you, I highly recommend checking out this excellent explanation on DSL by Martin Fowler in the form of an excerpt from his book *Domain-Specific Languages*. The excerpt can be found at http://www.informit.com/articles/article.aspx?p=1592379.

**Getting ready**

Open your local copy of the following file in your web browser:

How to do it...

Let's see how function chain can be used to produce concise and readable code that produces dynamic visual content.

```javascript
<script type="text/javascript">
    var body = d3.select("body"); // <-- A

    body.append("section") // <-- B
        .attr("id", "section1") // <-- C
        .append("div") // <-- D
        .attr("class", "blue box") // <-- E
        .append("p") // <-- F
        .text("dynamic blue box"); // <-- G

    body.append("section")
        .attr("id", "section2")
        .append("div")
        .attr("class", "red box")
        .append("p")
        .text("dynamic red box");
</script>
```

This code generates the following visual output (similar to what we saw in the previous chapter):

![Dynamic Blue Box and Dynamic Red Box](image)

*Function chain*

How it works...

Despite the visual similarity to the previous example, the construction process of the DOM elements is significantly different in this example. As demonstrated by the code example there is no static HTML element on the page contrary to the previous recipe where both the `section` and `div` elements existed.
Be Selective

Let's examine closely how these elements were dynamically created. On line A, a general selection was made to the top level body element. The body selection result was cached using a local variable called body. Then at line B, a new element section was appended to the body. Remember that the append function returns a new selection that contains the newly appended element therefore on line C the id attribute can then be set on a newly created section element to section1. Afterwards on line D a new div element was created and appended to #section1 with its CSS class set to blue box on line E. Next step, similarly on line F a paragraph element was appended to the div element with its textual content set to dynamic blue box on line G.

As illustrated by this example, this chaining process can continue to create any structure of arbitrary complexity. As a matter of fact, this is how typically D3 based data visualization structure was created. Many visualization projects simply contain only a HTML skeleton while relying on D3 to create the rest. Getting comfortable with this way of function chaining is critical if you want to become efficient with the D3 library.

Some of D3's modifier functions return a new selection, such as the select, append, and insert functions. It is a good practice to use different levels of indentation to differentiate which selection your function chain is being applied on.

Manipulating the raw selection

Sometimes, though not very often, having access to D3 raw selection array might be beneficial in development whether it's for debugging purposes or for integrating with other JavaScript libraries which require access to raw DOM elements; in this recipe, we will show you ways to do that. We will also see some internal structure of a D3 selection object.

Getting ready

Open your local copy of the following file in your web browser:

Of course you can achieve this by using the `nth-child` selector or selection iterator function via `each`, but there are cases where these options are just too cumbersome and inconvenient. This is when you might find dealing with raw selection array as a more convenient approach. In this example, we will see how raw selection array can be accessed and leveraged.

```html
<table class="table">
  <thead>
    <tr>
      <th>Time</th>
      <th>Type</th>
      <th>Amount</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>10:22</td>
      <td>Purchase</td>
      <td>$10.00</td>
    </tr>
    <tr>
      <td>12:12</td>
      <td>Purchase</td>
      <td>$12.50</td>
    </tr>
    <tr>
      <td>14:11</td>
      <td>Expense</td>
      <td>$9.70</td>
    </tr>
  </tbody>
</table>

<script type="text/javascript">
  var rows = d3.selectAll("tr");  // A
  var headerElement = rows[0][0];  // B
  d3.select(headerElement).attr("class", "table-header");  // C
  d3.select(rows[0][1]).attr("class", "table-row-odd");  // D
  d3.select(rows[0][2]).attr("class", "table-row-even");  // E
  d3.select(rows[0][3]).attr("class", "table-row-odd");  // F
</script>
```
Be Selective

This recipe generates the following visual output:

<table>
<thead>
<tr>
<th>Time</th>
<th>Type</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:22</td>
<td>Purchase</td>
<td>$10.00</td>
</tr>
<tr>
<td>12:12</td>
<td>Purchase</td>
<td>$12.50</td>
</tr>
<tr>
<td>14:11</td>
<td>Expense</td>
<td>$9.70</td>
</tr>
</tbody>
</table>

**How it works...**

In this example, we went through an existing HTML table to color the table. This is not intended to be a good example of how you would color odd versus even rows in a table using D3. Instead, this example is designed to show how raw selection array can be accessed.

A much better way to color odd and even rows in a table would be using the each function and then relying on the index parameter \( i \) to do the job.

On line A, we select all rows and store the selection in the `rows` variable. D3 selection is stored in a two-dimensional JavaScript array. The selected elements are stored in an array then wrapped in a single element array. Thus in order to access the first selected element, you need to use `rows[0][0]` and the second element can be accessed with `rows[0][1]`. As we can see on line B, the table header element can be accessed using `rows[0][0]` and this will return a DOM element object. Again as we have demonstrated in previous sections, any DOM element can then be selected directly using `d3.select` as shown on line C. Line D, E, and F demonstrate how each element in selection can be directly indexed and accessed.

Raw selection access could be handy in some cases; however since it relies on direct access to D3 selection array it creates a structural dependency in your code. In other words, if in future releases of D3 this structure ever changes, it will break your code that relies on it. Hence, it is advised to avoid raw selection manipulation unless absolutely necessary.

This approach is usually not necessary however it might become handy under certain circumstances such as in your unit-test cases when knowing the absolute index for each element quickly and gaining a reference to them could be convenient. We will cover unit-tests in a later chapter in more details.
In this chapter we will cover:

- Binding array as data
- Binding object literals as data
- Binding functions as data
- Working with arrays
- Filtering with data
- Sorting with data
- Loading data from server

### Introduction

In this chapter, we are going to explore the most essential question in any data visualization project, how data can be represented both in programming constructs, and its visual metaphor. Before we start on this topic, some discussion on what data visualization is is necessary. In order to understand what data visualization is, first we need to understand the difference between data and information.

*Data are raw facts. The word raw indicates that the facts have not yet been processed to reveal their meaning...Information is the result of processing raw data to reveal its meaning.*

*(Rob P., S. Morris, and Coronel C. 2009)*

This is how data and information are traditionally defined in the digital information world. However, data visualization provides a much richer interpretation of this definition since information is no longer the mere result of processed raw facts but rather a visual metaphor of the facts. As suggested by Manuel Lima in his *Information Visualization Manifesto* that design in the material world, where form is regarded to follow function.
Dealing with Data

The same data set can generate any number of visualizations which may lay equal claim in terms of its validity. In a sense, visualization is more about communicating the creator's insight into data than anything else. On a more provocative note, Card, McKinlay, and Shneiderman suggested that the practice of information visualization can be described as:

*The use of computer-supported, interactive, visual representations of abstract data to amplify cognition.*

(Card S. & Mackinly J. and Shneiderman B. 1999)

In the following sections, we will explore various techniques D3 provides to bridge the data with the visual domain. It is the very first step we need to take before we can create a cognition amplifier with our data.

**The enter-update-exit pattern**

The task of matching each datum with its visual representation, for example, drawing a single bar for every data point you have in your data set, updating the bars when the data points change, and then eventually removing the bars when certain data points no longer exist, seems to be a complicated and tedious task. This is precisely why D3 was designed to provide an ingenious way of simplifying the implementation of this connection. This way of defining the connection between data and its visual representation is usually referred to as the **enter-update-exit** pattern in D3. This pattern is profoundly different from the typical imperative method most developers are familiar with. However, the understanding of this pattern is crucial to your effectiveness with D3 library, and therefore, in this section, we will focus on explaining the concept behind this pattern. First, let's take a look at the following conceptual illustration of the two domains:

![Data and Visual Set](image)

In the previous illustration, the two circles represent two joined sets. Set A depicts your data set while set B represents the visual elements. This is essentially how D3 sees the connection between your data and visual elements. You might be asking how elementary set theory is going to help your data visualization effort here. Let me explain.
Firstly, let us consider the question, how can I find all visual elements that currently represent its corresponding data point? The answer is $A \cap B$; this denotes the intersection of sets $A$ and $B$, the elements that exist in both Data and Visual domains.

![Diagram of Update Mode]

The shaded area represents the intersection between the two sets—$A$ and $B$. In D3, the `selection.data` function can be used to select this intersection—$A \cap B$.

The `selection.data` function, on a selection, sets up the connection between the data domain and visual domain as we discussed above. The initial selection forms the visual set $B$ while the data provided in the `data` function forms the data set $A$ respectively. The return result of this function is a new selection (a data-bound selection) of all elements existing in this intersection. Now, you can invoke the modifier function on this new selection to update all the existing elements. This mode of selection is usually referred to as the Update mode.

The second question we need to answer here is how can I locate data that has not yet been visualized. The answer is the set difference of $A$ and $B$, denoted as $A \setminus B$, or visually, the following illustration:

![Diagram of Enter Mode]

The shaded area in set $A$ represents the data points that have not yet been visualized. In order to gain access to this $A \setminus B$ subset, the following functions need to be performed on a data-bound D3 selection (a selection returned by the `data` function).
Dealing with Data

The `selection.data(data).enter()` function returns a new selection representing the \( A \setminus B \) subset, which contains all the data that has not yet been represented in the visual domain. The regular modifier function can then be chained to this new selection method to create new visual elements representing the given data elements. This mode of selection is simply referred to as the **Enter** mode.

The last case in our discussion covers the visual elements that exist in our data set but no longer have any corresponding data element associated with them. You might ask how this kind of visual element can exist in the first place. This is usually caused by removing the elements from the data set. If you initially visualized all data elements within your data set, after that you have removed some data elements. Now, you have certain visual elements that are no longer representing any valid data point in your data set. This subset can be discovered by using an inverse of the **Update** difference, denoted as \( B \setminus A \).

![Diagram of Enter and Exit modes]

The shaded area in the previous illustration represents the difference we discussed here. The subset can be selected using the `selection.exit` function on a data-bound selection.

The `selection.data(data).exit` function, when invoked on a data-bound D3 selection, computes a new selection which contains all visual elements that are no longer associated with any valid data element. As a valid D3 selection object, the modifier function can then be chained to this selection to update and remove these visual elements that are no longer needed as part of our visualization. This mode of selection is called the **Exit** mode.

Together, the three different selection modes cover all possible cases of interaction between the data and visual domain. The **enter-update-exit** pattern is the cornerstone of any D3-driven visualization. In the following recipes of this chapter, we will cover the topics on how these selection methods can be utilized to generate data-driven visual elements efficiently and easily.
Binding an array as data

One of the most common and popular ways to define data in D3 visualization is through the use of JavaScript arrays. For example, say you have multiple data elements stored in an array, and you want to generate corresponding visual elements to represent each and every one of them. Additionally, when the data array gets updated, you want your visualization to reflect such changes immediately. In this recipe, we will accomplish this common approach.

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

The first and most natural solution that might come to mind is iterating through the data array elements and generating their corresponding visual elements on the page. This is definitely a valid solution and it will work with D3, however, the enter-update-exit pattern we discussed in the introduction provides a much easier and more efficient way to generate visual elements. Let's have a look at how we do that:

```javascript
var data = [10, 15, 30, 50, 80, 65, 55, 30, 20, 10, 8]; // <- A

function render(data) { // <- B
    // Enter
    d3.select("body").selectAll("div.h-bar") // <- C
        .data(data) // <- D
        .enter() // <- E
        .append("div") // <- F
        .attr("class", "h-bar")
        .append("span"); // <- G

    // Update
    d3.select("body").selectAll("div.h-bar")
        .data(data)
        .style("width", function (d) { // <- H
            return (d * 3) + "px";
        })
        .select("span") // <- I
        .text(function (d) {
            return d;
        });

    // Exit

```
Dealing with Data

d3.select("body").selectAll("div.h-bar")
  .data(data)
  .exit() // <- J
  .remove();
}

setInterval(function () { // <- K
  data.shift();
  data.push(Math.round(Math.random() * 100));
  render(data);
}, 1500);
render(data);

This recipe generates the following visual output:

```
Data as Array

  10
  15
  30
  50
  80
  65
  55
  30
  20
  10
   8
```

How it works...

In this example, data (a list of integers in this case) is stored in a simple JavaScript array as shown on the line marked as A with an arrow left of it. The render function is defined on the line marked as B so that it can be repeatedly invoked to update our visualization. The Enter selection implementation starts on the line marked as C, which selects all div elements on the web page with h-bar CSS class. You are probably wondering why we are selecting these div elements since they don’t even exist on the web page yet. This is in fact true; however, the selection at this point is used to define the visual set we discussed in the introduction. By issuing this selection that we made in the previous line we are essentially declaring that there should be a set of div.h-bar elements on the web page to form our visual set. On the line marked as D, we invoke the data function on this initial selection to bind the array as our data set to the to-be-created visual elements. Once the two sets are defined, the enter() function can be used to select all data elements that are not yet visualized. When the render function is invoked for the very first time, it returns all elements in the data array, as shown in the following code snippet:
On line F, a new div element is created and appended to the body element of each data element selected in the enter function; this essentially creates one div element for each data element. Finally, on line G, an element called span is created and appended to the div element and we set its CSS class to h-bar. At this point, we have basically created the skeleton of our visualization including empty div and span elements. Next step is to change the visual attributes of our elements based on the given data.

D3 injects a property to the DOM element named __data__ to make data sticky with visual elements so when selections are made using a modified data set, D3 can compute the difference and intersection correctly. You can see this property easily if you inspect the DOM element either visually using a debugger or programmatically.

As illustrated by the preceding screenshot, this is a very useful fact to know when you are debugging your visualization implementation.

In the Update section of array-as-data.html, the first two lines are identical to what we have done in the Enter section, and this essentially defines our data set and visual set respectively. The major difference here is on line H. Instead of calling the enter function, as we did in the code mentioned under Enter in the previous paragraphs, in the Update mode we directly apply modifier functions to the selection made by the data function. In the Update mode, data function returns the intersection between the data set and visual set (A∩B). On line H, we apply a dynamic style attribute width to be three times the integer value associated with each visual element shown in the following code snippet:

```javascript
d3.select("body").selectAll("div.h-bar") // <- C
  .data(data) // <- D
  .enter() // <- E
  .append("div") // <- F
    .attr("class", "h-bar")
    .append("span"); // <- G

  // <- H
  .style("width", function (d) {
    return (d * 3) + "px";
  })

  // <- I
  .select("span")
    .text(function (d) {
      return d;
    });
```

```
Dealing with Data

All D3 modifier functions accept this type of dynamic function to compute its value on the fly. This is precisely what it means to "data drive" your visualization. Hence, it is crucial to understand what this function is designed to achieve in our example. This function receives a parameter \( d \), which is the datum associated with the current element. In our example, the first div bar has the value 10 associated as its datum, while the second bar has 15, and so on. Therefore, this function essentially computes a numeric value that is three times the datum for each bar and returns it as the width in pixels.

Another interesting point worth mentioning here is on line I, where we mention the span attribute. The child \( \text{span} \) element can also use dynamic modifier functions and has access to the same datum propagated from its parent element. This is the default behavior of D3 data binding. Any element you append to a data-bound element automatically inherits the parent's datum.

The **dynamic modifier function** actually accepts two parameters \( d \) and \( i \). The first parameter \( d \) is the associated datum we have discussed here and \( i \) is a zero-based index number for the current element. Some recipes in the previous chapter have relied on this index, and in the rest of this chapter, we will see other recipes that utilize this index in different ways.

This is the raw HTML code resulted from this update process:

```html
<div class="h-bar" style="width: 30px;">
  <span>10</span>
</div>
<div class="h-bar" style="width: 45px;">
  <span>15</span>
</div>
....
<div class="h-bar" style="width: 24px;">
  <span>8</span>
</div>
```

Elements created and appended in the enter mode, that is, on line F and G, are automatically added to the update set. So, there is no need to repeat visual attributes modification logic in both enter and update section of code.
The last section—Exit section—is fairly simple as shown here:

```javascript
let d3 = d3.select("body").selectAll("div.h-bar")
  .data(data)
  .exit() // <- J
  .remove();
```

The selection returned by the `exit()` function is just like any other selection. Therefore, although `remove` is the most common action used against the `exit` selection, you can also apply other modifiers or transitions to this selection. We will explore some of these options in later chapters.

On line J, the `exit()` function is called to compute the set difference of all visual elements that are no longer associated with any data. Finally, the `remove()` function is called on this selection to remove all the elements selected by the `exit()` function. This way, as long as we call the `render()` function after we change our data, we can always ensure that our visual representation and data are kept synchronized.

Now, the last block of code is as follows:

```javascript
setInterval(function () { // <- K
  data.shift();
  data.push(Math.round(Math.random() * 100));
  render(data);
}, 1500);
```

On line K, a simple function called `function()` was created to remove the top element in the data array using the `shift()` function, while appending a random integer to the data array using the `push()` function every 1.5 seconds. Once the data array is updated, the `render()` function is called again to update our visualization keeping it synchronized with the new data set. This is what gives our example its animated bar chart look.

### Binding object literals as data

With a more complex visualization, each element we have in a data array might not be a primitive integer value or a string, but a JavaScript object themselves. In this recipe, we will discuss how this more complex data structure can be leveraged to drive your visualization using D3.
Dealing with Data

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

JavaScript object literal is probably the most common data structure you will encounter when loading data sources on the Web. In this recipe, we will look at how these JavaScript objects can be leveraged to generate rich visualization. Here is how to do it in code:

```javascript
var data = [
    {width: 10, color: 23},
    {width: 15, color: 33},
    {width: 30, color: 40},
    {width: 50, color: 60},
    {width: 80, color: 22},
    {width: 65, color: 10},
    {width: 55, color: 5},
    {width: 30, color: 30},
    {width: 20, color: 60},
    {width: 10, color: 90},
    {width: 8, color: 10}
];

var colorScale = d3.scale.linear()
    .domain([0, 100])
    .range(["#add8e6", "blue"]); // <- B

function render(data) {
    d3.select("body").selectAll("div.h-bar")
        .data(data)
        .enter().append("div")
        .attr("class", "h-bar")
        .append("span");

    d3.select("body").selectAll("div.h-bar")
        .data(data)
        .exit().remove();

    d3.select("body").selectAll("div.h-bar")
        .data(data)
        .attr("class", "h-bar")
        .style("width", function (d) { // <- C
            return (d.width * 5) + "px"; // <- D
        })
        .style("background-color", function (d) {
            return colorScale(d.color); // <- E
        })
}
```
This recipe generates the following visualization:

```
<table>
<thead>
<tr>
<th>10</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>80</td>
<td>65</td>
</tr>
<tr>
<td>55</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>
```

**Data as Object**

**How it works...**

In this recipe, instead of simple integers as in the previous recipe, now our data array is filled with objects (see the line marked as A with an arrow left to it). Each data object contains two attributes—width and color—that are both integers in this case.

This recipe is built on top of the previous recipe so if you are not familiar with the fundamental enter-update-exit selection pattern, please review the previous recipe first.
Dealing with Data

```javascript
var data = [ // <- A
    {width: 10, color: 23},
    {width: 15, color: 33},
    ...
    {width: 8, color: 10}
];
```

On line B, we have a complicated-looking color scale defined. Scales, including color scale, will be discussed in depth in the next chapter, so for now let us just assume this is a scale function we can use to produce CSS-compatible color code given some integer input value. This is sufficient for the purpose of this recipe.

The major difference between this recipe and the previous one is how data is handled as shown on line C:

```javascript
function (d) { // <- C
    return (d.width * 5) + "px"; // <- D
}
```

As we can see in the preceding code snippet, in this recipe the datum associated with each visual element is actually an object, not an integer. Therefore, we can access the `d.width` attribute on line D.

If your object has functions of its own, you can also access them here in a dynamic modifier function. This is a convenient way to add some data-specific helper functions in your data source. However, beware that since dynamic functions are usually invoked numerous times during visualization, the function you rely on should be implemented as efficiently as possible. If this is not possible, then it is best to preprocess your data before binding them to your visualization process.

Similarly on line E, the `background-color` style can be computed using the `d.color` attribute with the color scale we defined earlier:

```javascript
.style("background-color", function(d){
    return colorScale(d.color); // <- E
})
.select("span")
.text(function (d) {
    return d.width; // <- F
});
```
The child element \texttt{span} again inherits its parent's associated datum, and hence, it also has access to the same datum object in its dynamic modifier function on line \texttt{F} setting the textual content to \texttt{d.width} attribute.

This recipe demonstrates how JavaScript objects can easily be bound to visual elements using exactly the same method discussed in the previous recipe. This is one of the most powerful capabilities of the D3 library; it allows you to re-use the same pattern and method to handle different types of data, simple or complex. We will see more examples on this topic in the next recipe.

### Binding functions as data

One of the benefits of D3's excellent support for functional style JavaScript programming is that it allows functions to be treated as data as well. This particular feature can offer some very powerful capabilities under certain circumstances. This is a more advanced recipe. Don't worry about it if you are new to D3 and having some difficulty understanding it at first. Over time, this kind of usage will become natural to you.

### Getting Ready

Open your local copy of the following file in your web browser:


### How to do it...

In this recipe, we will explore the possibility of binding functions themselves as data to your visual elements. This capability is extremely powerful and flexible if used correctly:

```html
<div id="container"></div>

<script type="text/javascript">
var data = []; // <- A

var next = function (x) { // <- B
    return 15 + x * x;
}

var newData = function () { // <- C
    data.push(next);
    return data;
}

function render(){
```
Dealing with Data

```javascript
var selection = d3.select("#container")
  .selectAll("div")
  .data(newData); // <- D

selection.enter().append("div").append("span");

selection.exit().remove();

selection.attr("class", "v-bar")
  .style("height", function (d, i) {
    return d(i) + "px"; // <- E
  })
  .select("span")
  .text(function(d, i) {
    return d(i); } // <- F
  );

setInterval(function () {
  render();
}, 1500);

render();
</script>

This preceding code produces the following bar chart:

![Bar Chart](image-url)
In this recipe, we chose to visualize the output of formula $15 + x \times x$ using a series of vertical bars, each of them annotated with its representing integral value. This visualization adds a new bar to the right of the previous one every one and a half seconds. We can of course implement this visualization using the techniques we have discussed in the previous two recipes. So we generated an array of integers using the formula, then just appended a new integer from $n$ to $n+1$ every 1.5 seconds before re-rendering the visualization. However, in this recipe, we decided to take a more functional approach.

This time we started with an empty data array on line A. On line B, a simple function is defined to calculate the result of this formula $15+x^2$. Then on line C, another function is created to generate the current data set which contains $n+1$ references to the `next` function. Here is the code for functional data definition:

```javascript
var data = []; // <- A
var next = function (x) { // <- B
    return 15 + x * x;
};
var newData = function () { // <- C
    data.push(next);
    return data;
};
```

This seems to be a strange setup to achieve our visualization goal. Let's see how we can leverage all these functions in our visualization code. On line D, we bind our data to a selection of div elements just as we did in previous recipes. However, this time the data is not an array but rather the `newData` function:

```javascript
var selection = d3.select("#container")
    .selectAll("div")
    .data(newData); // <- D
```

D3 is pretty flexible when it comes to data. If you provide a function to the `data` function, D3 will simply invoke the given function and use the returned value of this function as a parameter of the `data` function. In this case, the data being returned by the `newData` function is an array of function references. As the result of this, now in our dynamic modifier function, on line E and F, the datum $d$ that is being passed into these functions are actually references to the `next` function, as shown in the following code:

```javascript
selection.attr("class", "v-bar")
    .style("height", function (d, i) {
        return d(i)+"px"; // <- E
    })
    .select("span")
```
Dealing with Data

In JavaScript, functions are special objects, so semantically this is exactly the same as binding objects as data. Another note on this topic is that data can also be considered as functions. Constant values such as integers can be thought of as a static function that simply returns what it receives with no modification made.

This technique might not be the most commonly-used technique in visualization, but when used properly, it is extremely flexible and powerful, especially when you have a fluid data set.

Datum function typically needs to be idempotent to make sense. Idempotence is the property of being able to apply the same function with the same inputs multiple times without changing the result beyond the initial application. For more detail on idempotence visit: http://en.wikipedia.org/wiki/Idempotence

Working with arrays

Most of our data is stored in arrays, and we spend a lot of our effort working with arrays to format and restructure data. This is why D3 provides a rich set of array-oriented utilities functions, making this task a lot easier. In this recipe, we will explore some of the most common and helpful utilities in this aspect.

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

The following code example shows some of the most common and helpful array utility functions offered by the D3 library and their effects:

```html
<script type="text/javascript">
   // Static html code were omitted due to space constraint
</script>
```
var array = [3, 2, 11, 7, 6, 4, 10, 8, 15];

d3.select("#min").text(d3.min(array));
d3.select("#max").text(d3.max(array));
d3.select("#extent").text(d3.extent(array));
d3.select("#sum").text(d3.sum(array));
d3.select("#median").text(d3.median(array));
d3.select("#mean").text(d3.mean(array));
d3.select("#asc").text(array.sort(d3.ascending));
d3.select("#desc").text(array.sort(d3.descending));
d3.select("#quantile").text(
    d3.quantile(array.sort(d3.ascending), 0.25)
);
d3.select("#bisect").text(
    d3.bisect(array.sort(d3.ascending), 6)
);

var records = [
    {quantity: 2, total: 190, tip: 100, type: "tab"},
    {quantity: 2, total: 190, tip: 100, type: "tab"},
    {quantity: 1, total: 300, tip: 200, type: "visa"},
    {quantity: 2, total: 90, tip: 0, type: "tab"},
    {quantity: 2, total: 90, tip: 0, type: "tab"},
    {quantity: 2, total: 90, tip: 0, type: "tab"},
    {quantity: 1, total: 100, tip: 0, type: "cash"},
    {quantity: 2, total: 90, tip: 0, type: "tab"},
    {quantity: 2, total: 90, tip: 0, type: "tab"},
    {quantity: 2, total: 200, tip: 0, type: "cash"},
    {quantity: 1, total: 200, tip: 100, type: "visa"}
];

var nest = d3.nest()
    .key(function (d) { // <- A
        return d.type;
    })
    .key(function (d) { // <- B
        return d.tip;
    })
    .entries(records); // <- C

d3.select("#nest").html(printNest(nest, ""));
Dealing with Data

function printNest(nest, out, i) {
    if (i === undefined) i = 0;

    var tab = "";
    for (var j = 0; j < i; ++j)
        tab += " ";

    nest.forEach(function (e) {
        if (e.key)
            out += tab + e.key + "<br>";
        else
            out += tab + printObject(e) + "<br>";

        if (e.values)
            out = printNest(e.values, out, ++i);
        else
            return out;
    });

    return out;
}

function printObject(obj) {
    var s = "{"
    for (var f in obj) {
        s += f + " : " + obj[f] + ", ";
    }
    s += "}";
    return s;
}

</script>

The preceding code produces the following output:

d3.min => 2

d3.max => 15

d3.extent => 2,15

d3.sum => 66

d3.median => 7

d3.mean => 7.333333333333333
array.sort(d3.ascending) => 2,3,4,6,7,8,10,11,15
array.sort(d3.descending) => 15,11,10,8,7,6,4,3,2

d3.quantile(array.sort(d3.ascending), 0.25) => 4

d3.bisect(array.sort(d3.ascending), 6) => 4
How it works...

D3 provides a variety of utility functions to help perform operations on JavaScript arrays. Most of them are pretty intuitive and straightforward, however, there are a few intrinsic ones. We will discuss them briefly in this section.

Given our array as [3, 2, 11, 7, 6, 4, 10, 8, 15]:

- `d3.min`: This function retrieves the smallest element, that is, 2
- `d3.max`: This function retrieve the largest element, that is, 15
- `d3.extent`: This function retrieves both the smallest and the largest element, that is, [2, 15]
- `d3.sum`: This function retrieves the addition of all elements in the array, that is, 66
- `d3.medium`: This function finds the medium, that is, 7
- `d3.mean`: This function calculates the mean value, that is, 7.33
- `d3.ascending` / `d3.descending`: The d3 object comes with a built-in comparator function that you can use to sort the JavaScript array
  ```javascript
  d3.ascending = function(a, b) { return a < b ? -1 : a > b ? 1 : 0; }
  d3.descending = function(a, b) { return b < a ? -1 : b > a ? 1 : 0; }
  ```

```javascript
// tab
100
{quantity: 2, total: 190, tip: 100, type: tab, }
{quantity: 2, total: 190, tip: 100, type: tab, }
0
{quantity: 2, total: 90, tip: 0, type: tab, }
{quantity: 2, total: 90, tip: 0, type: tab, }
{quantity: 2, total: 90, tip: 0, type: tab, }
{quantity: 2, total: 90, tip: 0, type: tab, }
{quantity: 2, total: 90, tip: 0, type: tab, }

// visa
200
{quantity: 1, total: 300, tip: 200, type: visa, }
100
{quantity: 1, total: 200, tip: 100, type: visa, }

// cash
0
{quantity: 1, total: 100, tip: 0, type: cash, }
{quantity: 2, total: 200, tip: 0, type: cash, }
```
Dealing with Data

- `d3.quantile`: This function calculates the quantile on an already sorted array in ascending order, that is, quantile of 0.25 is 4.
- `d3.bisect`: This function finds an insertion point that comes after (to the right of) any existing element of an already sorted array, that is, bisect(array, 6) produce 4.
- `d3.nest`: D3's `nest` function can be used to build an algorithm that transforms a flat array-based data structure into a hierarchical nested structure, that is, particularly suitable for some types of visualization. D3's `nest` function can be configured using the `key` function chained to `nest`, as seen on lines A and B:

  ```javascript
  var nest = d3.nest()
    .key(function (d) { // <- A
      return d.type;
    })
    .key(function (d) { // <- B
      return d.tip;
    })
    .entries(records); // <- C
  ```

  Multiple `key` functions can be provided to generate multiple levels of nesting. In our case the nesting consists of two levels, first by the `type` amount and then by the `tip` amount, as demonstrated in the output below:

  ```javascript
  tab
    100
  
  {quantity: 2, total: 190, tip: 100, type: tab, }
  {quantity: 2, total: 190, tip: 100, type: tab, }
  ```

  Finally, the `entries()` function is used to supply the flat array-based data set as shown on line C.

Filtering with data

Imagine you need to filter D3 selection based on the associated data elements so that you can hide/show different sub-datasets based on the user's input. D3 selection provides a filter function to perform this kind of data-driven filtering. In this recipe, we will show you how this can be leveraged to filter visual elements in a data-driven fashion.

Getting Ready

Open your local copy of the following file in your web browser:

Chapter 3

How to do it...

The following example code shows how data-based filtering can be leveraged to highlight different visual elements based on its categorization:

```javascript
var data = [
  {expense: 10, category: "Retail"},
  {expense: 15, category: "Gas"},
  {expense: 30, category: "Retail"},
  {expense: 50, category: "Dining"},
  {expense: 80, category: "Gas"},
  {expense: 65, category: "Retail"},
  {expense: 55, category: "Gas"},
  {expense: 30, category: "Dining"},
  {expense: 20, category: "Retail"},
  {expense: 10, category: "Dining"},
  {expense: 8, category: "Gas"}
];

function render(data, category) {
  d3.select("body").selectAll("div.h-bar")
    .data(data)
    .enter()
    .append("div")
    .attr("class", "h-bar")
    .append("span");

  d3.select("body").selectAll("div.h-bar")
    .data(data)
    .exit().remove();

  d3.select("body").selectAll("div.h-bar")
    .data(data)
    .attr("class", "h-bar")
    .style("width", function (d) {
      return (d.expense * 5) + "px";
    })
    .select("span")
    .text(function (d) {
      return d.category;
    });

  d3.select("body").selectAll("div.h-bar")
    .filter(function (d, i) {
      return i >= 3;  // <--E
    });
}
```

```html
<script type="text/javascript">...
</script>
```
The preceding code generates the following visual output once the Dining button is clicked:
How it works...

In this recipe, we have a data set consisting of a list of personal expense records with expense and category as attributes, which is shown on the block of code marked as A. On line B, C, and D, a set of horizontal bars (HTML div) were created using the standard enter-update-exit pattern to represent the expense records. So far, this recipe is similar to the Binding object literals as data recipe. Now let’s take a look at line E:

```javascript
filter(function (d, i) { // <-E
   return d.category == category;
});
```

D3 selection.filter function takes a function as its parameter. It applies the function against every element in the existing selection. The given function for filter takes two parameters with a hidden reference:

- d: It is the datum associated with the current element
- i: It is a zero-based index for the current element
- this: This has the hidden reference points to the current DOM element

D3 selection.filter function expects the given function to return a Boolean value. If the returned value is true, the corresponding element will be included into the new selection being returned by the filter function. In our example, the filter function essentially selects all bars that match the user-selected category and applies a CSS class selected to each one of them. This method provides you a powerful way to filter and generate data-driven sub-selection, which you can further manipulate or dissect to generate focused visualization.

D3 selection.filter function treats the returned value using JavaScript truthy and falsy tests, thus not exactly expecting a strict Boolean value. What this means is that false, null, 0, "", undefined, and NaN (not a number) are all treated as false while other things are considered true.

Sorting with data

In many cases, it is desirable to sort your visual elements according to the data they represent so that you can highlight the significance of different elements visually. In this recipe, we will explore how this can be achieved in D3.

Getting Ready

Open your local copy of the following file in your web browser:

Let's see how data-driven sorting and further manipulation can be performed using D3. In this example, we will sort the bar chart we created in the previous recipe based on either expense (width) or category using user's input:

```javascript
var data = [  // <-A
    {expense: 10, category: "Retail"},
    {expense: 15, category: "Gas"},
    {expense: 30, category: "Retail"},
    {expense: 50, category: "Dining"},
    {expense: 80, category: "Gas"},
    {expense: 65, category: "Retail"},
    {expense: 55, category: "Gas"},
    {expense: 30, category: "Dining"},
    {expense: 20, category: "Retail"},
    {expense: 10, category: "Dining"},
    {expense: 8, category: "Gas"}
];

function render(data, comparator) {
    d3.select("body").selectAll("div.h-bar") // <-B
        .data(data)
        .enter().append("div")
        .attr("class", "h-bar")
        .append("span");

d3.select("body").selectAll("div.h-bar") // <-C
    .data(data)
    .exit().remove();

    d3.select("body").selectAll("div.h-bar") // <-D
    .data(data)
    .attr("class", "h-bar")
    .style("width", function (d) {
       return (d.expense * 5) + "px";
    })
    .select("span")
    .text(function (d) {
       return d.category;
    });

    if(comparator)
    d3.select("body")
        .selectAll("div.h-bar")
        .filter(comparator);
```

```javascript
</script>
```
Chapter 3

```javascript
var compareByExpense = function (a, b) {  // <-F
  return a.expense < b.expense?-1:1;
};
var compareByCategory = function (a, b) {  // <-G
  return a.category < b.category?-1:1;
};
render(data);
function sort(comparator) {
  render(data, comparator);
}
</script>
<div class="control-group">
  <button onclick="sort(compareByExpense)">Sort by Width</button>
  <button onclick="sort(compareByCategory)">Sort by Category</button>
  <button onclick="sort()">Clear</button>
</div>

This preceding code generates sorted horizontal bars as shown in the following screenshot:
Dealing with Data

How it works...

In this recipe, we set up a simple row-based visualization (in line B, C, and D) of some simulated personal expense records containing two attributes: expense and category that are defined on line A. This is exactly the same as the previous recipe and quite similar to what we have done in the Binding object literals as data recipe. Once the basics are done, we then select all existing bars on line E and perform sorting using D3 selection.sort function:

```javascript
    d3.select("body")
        .selectAll("div.h-bar")
        .sort(comparator); // <-E
```

The selection.sort function accepts a comparator function:

```javascript
    var compareByExpense = function (a, b) {  // <-F
        return a.expense < b.expense?-1:1;
    };
    var compareByCategory = function (a, b) {  // <-G
        return a.category < b.category?-1:1;
    };
```

The comparator function receives two data elements a and b to compare, returning either a negative, positive, or zero value. If the value is negative, a will be placed before b; if positive, a will be placed after b; otherwise, a and b are considered equal and the order is arbitrary. The sort() function returns a new selection with all elements sorted in an order which is determined by the specified comparator function. The newly-returned selection can then be manipulated further to generate the desired visualization.

Because a and b are placed arbitrarily when they are equal, D3 selection.sort is not guaranteed to be stable, however, it is guaranteed to be consistent with your browser’s built-in sort method on arrays.

Loading data from a server

It is probably very rare that you will only be visualizing static local data. The power of data visualization usually lays on the ability to visualize dynamic data typically generated by a server-side program. Since this is a common use case, D3 comes with some handy helper functions to make this task as easy as possible. In this recipe, we will see how a remote data set can be loaded dynamically, and we will update an existing visualization once loaded.
Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

In the code example of the asyn-data-load.html file, we will load data dynamically from the server on user's request, and once the data is loaded, we also update our visualization to reflect the new expanded data set. Here is the code where we do that:

```html
<script type="text/javascript">
  var data = [ // <-A
    {expense: 10, category: "Retail"},
    {expense: 15, category: "Gas"},
    {expense: 30, category: "Retail"},
    {expense: 50, category: "Dining"},
    {expense: 80, category: "Gas"},
    {expense: 65, category: "Retail"},
    {expense: 55, category: "Gas"},
    {expense: 30, category: "Dining"},
    {expense: 20, category: "Retail"},
    {expense: 10, category: "Dining"},
    {expense: 8, category: "Gas"}
  ];

  function render(data) {
    d3.select("#chart").selectAll("div.h-bar") // <-B
      .data(data)
      .enter().append("div")
      .attr("class", "h-bar")
      .append("span");

    d3.select("#chart").selectAll("div.h-bar") // <-C
      .data(data)
      .exit().remove();

    d3.select("#chart").selectAll("div.h-bar") // <-D
      .data(data)
      .attr("class", "h-bar")
  }
</script>
```
Dealing with Data

```javascript
.DEALING WITH DATA

.style("width", function (d) {
    return (d.expense * 5) + "px";
})
.select("span")
.text(function (d) {
    return d.category;
});

render(data);

function load(){ // <-E
    d3.json("data.json", function(error, json){ // <-F
        data = data.concat(json);
        render(data);
    });
}
</script>

Here is what our data.json file looks like:

[  
    {"expense": 15,  "category": "Retail"},
    {"expense": 18,  "category": "Gas"},
    ...
    {"expense": 15,  "category": "Gas"}
]
This recipe generates the following visual output after clicking the **Load Data from JSON feed** button once:

![Visual Output](image)

**How it works...**

In this recipe, we initially have a local data set defined on the line marked as A, and a row-based visualization generated by lines B, C, and D. The load function is defined on line E that responds to the user's click on the **Load Data from JSON feed** button, which loads the data from a separate file (`data.json`) served by the server. This is achieved by using the `d3.json` function as shown on line F:

```javascript
function load() {
  d3.json("data.json", function(error, json) {
    data = data.concat(json);
    render(data);
  });
}
```
Since loading a remote data set from a JSON file could take some time, it is performed asynchronously. Once loaded, the data set will be passed to the given handler function, which is specified on line \( F \). In this function, we simply concatenate the newly loaded data with our existing data set, then re-render the visualization to update the display.

Similar functions are also provided by D3 to make the loading of CSV, TSV, TXT, HTML, and XML data a simple task.

If a more customized and specific control is required, the `d3.xhr` function can be used to further customize the MIME type and request headers. Behind the scenes, `d3.json` and `d3.csv` are both using `d3.xhr` to generate the actual request.

Of course this is by no means the only way to load remote data from the server. D3 does not dictate how data should be loaded from the remote server. You are free to use your favorite JavaScript libraries, for example, jQuery or Zepto.js to issue an Ajax request and load a remote data set.
In this chapter we will cover:

- Using quantitative scales
- Using time scale
- Using ordinal scale
- Interpolating string
- Interpolating colors
- Interpolating compound object
- Implementing custom interpolator

**Introduction**

As a data visualization developer, one key task that you need to perform over and over is to map values from your data domain to visual domain, for example, mapping your most recent purchase of a fancy tablet of $453.00 to a 653px-long bar, and your last night's pub bill of $23.59 to a 34px-long bar, respectively. In a sense, this is what data visualization is all about—mapping data elements to their visual metaphor in an efficient and accurate manner. Because this is an absolutely essential task in data visualization and animation (animation will be discussed in Chapter 6, *Transition with Style*, in detail), D3 provides rich and robust support on this topic, which is the focus of this chapter.
What are scales?

D3 provides various constructs called *scales* to help you perform this kind of mapping. Proper understanding of these constructs conceptually is crucial to become an effective visualization developer. This is because scales are not only used to perform the mapping we have mentioned previously, but also to serve as fundamental building blocks for many other D3 constructs, such as transition and axes.

What are these scales anyway?

In short, scales can be thought of as mathematical *functions*. Mathematical functions differ from functions defined in imperative programming languages, such as JavaScript functions. In mathematics, a function is defined as mapping between two sets:

> Let $A$ and $B$ be nonempty sets. A function $f$ from $A$ to $B$ is an assignment of exactly one element of $B$ to each element of $A$. We write $f(a) = b$ if $b$ is the unique element of $B$ assigned by the function $f$ to the element $a$ of $A$.

(Rosen K. H. 2007)

Despite the dryness of this definition, we still could not help but notice how nicely it fits the task we need to perform—mapping elements from the data domain to visual domain.

Another fundamentally important concept we need to illustrate here is the *domain* and *range* of a given function.

> If $f$ is a function from $A$ to $B$, we say that $A$ is the **domain** of $f$ and $B$ is the **codomain** of $f$. If $f(a) = b$, we say that $b$ is the **image** of $a$ and $a$ is a **preimage** of $b$. The **range**, or **image**, of $f$ is the set of all images of elements of $A$. Also, if $f$ is a function from $A$ to $B$, we say that $f$ maps $A$ to $B$.

(Rosen K. H. 2007)

To help us understand this concept, let's take a look at the following illustration:
We can clearly see now, in the preceding illustration for function \( f \), the domain is set \( A \) and the range is set \( B \). Imagine if set \( A \) represents our data domain and \( B \) represents the visual domain, then a function \( f \) defined here is essentially a scale in D3 that maps elements from set \( A \) to set \( B \).

For the mathematically inclined readers, scale function in data visualization are usually **one-to-one** but not **onto** functions. This is a useful insight to know but not critical to the purpose of this book. Therefore, we will not discuss it further here.

Now, we have discussed the conceptual definition of scale functions in D3, so let's take a look at how it can be used to help us develop our visualization project.

## Using quantitative scales

In this recipe, we will examine the most commonly-used scales provided by D3—the quantitative scales including linear, power, and logarithmic scales.

### Getting Ready

Open your local copy of the following file in your web browser:


### How to do it...

Let's take a look at the following code example:

```html
<div id="linear" class="clear"><span>n</span></div>
<div id="linear-capped" class="clear">
  <span>1 &lt;= a*n + b &lt;= 20</span>
</div>
<div id="pow" class="clear"><span>n^2</span></div>
<div id="pow-capped" class="clear">
  <span>1 &lt;= a*n^2 + b &lt;= 10</span>
</div>
<div id="log" class="clear"><span>log(n)</span></div>
<div id="log-capped" class="clear">
  <span>1 &lt;= a*log(n) + b &lt;= 10</span>
</div>
```
Tipping the Scales

```javascript
<script type="text/javascript">
  var max = 11, data = [];
  for (var i = 1; i < max; ++i) data.push(i);

  var linear = d3.scale.linear() // <-A
    .domain([1, 10]) // <-B
    .range([1, 10]); // <-C
  var linearCapped = d3.scale.linear()
    .domain([1, 10])
    .range([1, 20]); // <-D
  var pow = d3.scale.pow().exponent(2); // <-E
  var powCapped = d3.scale.pow() // <-F
    .exponent(2)
    .domain([1, 10])
    .rangeRound([1, 10]); // <-G
  var log = d3.scale.log(); // <-H
  var logCapped = d3.scale.log() // <-I
    .domain([1, 10])
    .rangeRound([1, 10]);

  function render(data, scale, selector) {
    d3.select(selector).selectAll("div.cell")
      .data(data)
      .enter().append("div").classed("cell", true);

    d3.select(selector).selectAll("div.cell")
      .data(data)
      .exit().remove();

    d3.select(selector).selectAll("div.cell")
      .data(data)
      .style("display", "inline-block")
      .text(function (d) {
        return d3.round(scale(d), 2);
      });
  }

  render(data, linear, "#linear");
  render(data, linearCapped, "#linear-capped");
  render(data, pow, "#pow");
</script>
```
The preceding code generates the following output in your browser:

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1</td>
<td>3.11</td>
<td>5.22</td>
<td>7.33</td>
<td>9.44</td>
<td>11.56</td>
<td>13.67</td>
<td>15.78</td>
<td>17.89</td>
<td>20</td>
</tr>
<tr>
<td>n^2</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>81</td>
<td>100</td>
</tr>
<tr>
<td>log(n)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.3</td>
<td>0.48</td>
<td>0.6</td>
<td>0.7</td>
<td>0.78</td>
<td>0.85</td>
<td>0.9</td>
<td>0.95</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Quantitative scale output

How it works...

In this recipe, we have demonstrated some of the most common scales provided by D3.

**Linear Scale**

In the preceding code example, we have our data array filled with integers from 0 to 10—as shown on the line marked as $A$—we created a linear scale by calling the `d3.scale.linear()` function. This returns a linear quantitative scale with the default domain set to $[0, 1]$ and the default range set to $[0, 1]$. Thus the default scale is essentially the identity function for numbers. Therefore, this default function is not that useful to us, but typically needs to be further customized by using its `domain` and `range` functions on line $B$ and $C$. In this case, we set them both to $[1, 10]$. This scale basically defines the function $f(n) = n$.

```javascript
var linear = d3.scale.linear() // <-A
  .domain([1, 10]) // <-B
  .range([1, 10]); // <-C
```

Identity scale
The second linear scale is a little bit more interesting and it illustrates the mapping between two sets better. On line D, we set the range as \([1, 20]\), which is different from its domain. Hence, now this function is essentially representing the following equations:

- \(f(n) = a \cdot n + b\)
- \(1 \leq f(n) \leq 20\)

This is by far the most common case when using D3 scales because your data set will be an identical match of your visual set.

```javascript
var linearCapped = d3.scale.linear()
    .domain([1, 10])
    .range([1, 20]); // <-D
```

In this second scale, D3 will automatically calculate and assign the value of constants \(a\) and \(b\) to satisfy the equation.

Some basic algebraic calculation will tell you that \(a\) is approximately 2.11 and \(b\) is -1.11, as in the previous example.

### Pow Scale

The second scale we have created is a **power scale**. On line E, we defined a power scale with exponent of 2. The `d3.scale.pow()` function returns a default power scale function with its exponent set as 1. This scale effectively defines the function \(f(n) = n^2\).

```javascript
var pow = d3.scale.pow().exponent(2); // <-E
```

On line F, a second power scale was defined, this time with a different range set on line G with rounding; the `rangeRound()` function works pretty much the same as the `range()` function, which sets the range for a scale. However, the `rangeRound` function rounds the output number so that there are no decimal fractions. This is very handy since scales are commonly used to map elements from the data domain to visual domain. So, the output of a scale is very likely to be a number describing some visual characteristics, for example, the number of pixels. Avoiding sub-pixel numbers is a useful technique that prevents anti-alias in rendering.
The second power scale defines the following function $f(n) = a*n^2 + b$, $1 <= f(n) <= 10$.

```javascript
var powCapped = d3.scale.pow() // <-F
  .exponent(2)
  .domain([1, 10])
  .rangeRound([1, 10]); // <-G
```

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;= a*n^2 + b &lt;= 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Power scale

Similar to the linear scale, D3 will automatically find the suitable constants $a$ and $b$ to satisfy the constraints defined by domain and range on a power scale.

**Log Scale**

On line H, a third kind of quantitative scale was created using the `d3.scale.log()` function. The default log scale has a base of 10. Line H essentially defines the following mathematical function $f(n) = \log(n)$.

```javascript
var log = d3.scale.log(); // <-H
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>0.3</th>
<th>0.48</th>
<th>0.6</th>
<th>0.7</th>
<th>0.78</th>
<th>0.85</th>
<th>0.9</th>
<th>0.95</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>log(n)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Simple log scale

On line I, we customized the log scale to have a domain of $[1, 10]$ and a rounded range of $[1, 10]$, which defines the following constrained mathematical function $f(n) = a*\log(n) + b$, $1 <= f(n) <= 10$.

```javascript
var logCapped = d3.scale.log() // <-I
  .domain([1, 10])
  .rangeRound([1, 10]);
```

```
<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;= a*\log(n) + b &lt;= 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Log scale

There's more...

D3 also provides other additional quantitative scales including quantize, threshold, quantile, and identity scales. Due to limited scope in this book and their relatively less common usage, they are not discussed here, however, the basic understanding of scales discussed and explained here will definitely help your understanding of other additional quantitative scales provided by the D3 library. For more information on other types of quantitative scales please visit [https://github.com/mbostock/d3/wiki/Quantitative-Scales#wiki-quantitative](https://github.com/mbostock/d3/wiki/Quantitative-Scales#wiki-quantitative)
Using the time scale

Often, we perform analysis on a data set that is time- and date-sensitive, therefore, D3 provides a built-in time scale to help perform this type of mapping. In this recipe, we will learn how to use D3 time scale.

**Getting Ready**

Open your local copy of the following file in your web browser:


**How to do it...**

First, let's take a look at the following code example:

```javascript
var start = new Date(2013, 0, 1), // <-A
    end = new Date(2013, 11, 31),
    range = [0, 1200],
    time = d3.time.scale().domain([start, end]) // <-B
        .rangeRound(range), // <-C
    max = 12,
    data = [];

for (var i = 0; i < max; ++i){ // <-D
    var date = new Date(start.getTime());
    date.setMonth(start.getMonth() + i);
    data.push(date);
}

function render(data, scale, selector) { // <-E
    d3.select(selector).selectAll("div.fixed-cell")
        .data(data)
            .enter()
                .append("div").classed("fixed-cell", true);

    d3.select(selector).selectAll("div.fixed-cell")
```
This recipe generates the following visual output:

**How it works...**

In this recipe, we have a `Date` range defined on line A between January 1, 2013 and December 31, 2013.

```javascript
var start = new Date(2013, 0, 1), // <-A
    end = new Date(2013, 11, 31),
    range = [0, 900],
    time = d3.time.scale().domain([start, end]) // <-B
    .rangeRound(range), // <-C
```

The JavaScript `Date` object starts its month from 0 and day from 1. Therefore, `new Date(2013, 0, 1)` gives you January 1, 2013 while `new Date(2013, 0, 0)` actually gives you December 31, 2012.

This range was then used to create a D3 `time scale` on line B using the `d3.time.scale` function. Similar to quantitative scales, time scale also supports separate `domain` and `range` definition, which is used to map date- and time-based data points to visual range. In this example, we set the range of the scale to `[0, 900]`. This effectively defines a mapping from any date-and-time value in time range between January 1, 2013 and December 31, 2013 to a number between 0 and 900.
Tipping the Scales

With the time scale defined, we can now map any given `Date` object by calling the `scale` function, for example, `time(new Date(2013, 4, 1))` will return 395 and `time(new Date(2013, 11, 15))` will return 1147, and so on.

On line `D`, we create our data array consisting 12 months from January to December in 2013:

```javascript
for (var i = 0; i < max; ++i) { // <-D
  var date = new Date(start.getTime());
  date.setMonth(start.getMonth() + i);
  data.push(date);
}
```

Then, on line `E`, we created 12 cells representing each month in a year using the `render` function.

To spread the cells horizontally, line `F` performs a mapping from the month to the `margin-left` CSS style using the time scale we defined:

```javascript
.style("margin-left", function(d){ // <-F
  return scale(d) + "px";
})
```

Line `G` generates the label to demonstrating what the scale-based mapping produces in this example:

```javascript
.html(function (d) { // <-G
  var format = d3.time.format("%x"); // <-H
  return format(d) + ", " + scale(d) + "px";
});
```

To generate human-readable strings from a JavaScript `Date` object, we used a D3 time formatter on line `H`. D3 ships with a powerful and flexible time-formatting library, which is extremely useful when dealing with the `Date` object.

There's more...

Here are some of the most useful `d3.time.format` patterns:

- `%a`: This is the abbreviated weekday name
- `%A`: This is the full weekday name
- `%b`: This is the abbreviated month name
- `%B`: This is the full month name
- `%d`: This is the zero-padded day of the month as a decimal number [01,31]
- `%e`: This is the space-padded day of the month as a decimal number [ 1,31]
- `%H`: This is the hour (24-hour clock) as a decimal number [00,23]
- `%I`: This is the hour (12-hour clock) as a decimal number [01,12]
- %j: This is the day of the year as a decimal number [001,366]
- %m: This is the month as a decimal number [01,12]
- %M: This is the minute as a decimal number [00,59]
- %L: This is the milliseconds as a decimal number [000, 999]
- %p: This is the either AM or PM
- %s: This is the second as a decimal number [00,61]
- %x: This is the date, as "%m/%d/%Y"
- %X: This is the time, as "%H:%M:%S"
- %y: This is the year without century as a decimal number [00,99]
- %Y: This is the year with century as a decimal number

See also

- For the complete reference on D3 time format pattern visit the following link:

Using the ordinal scale

In some cases, we might need to map our data to some ordinal values, for example, ["a", "b", "c"] or ["#1f77b4", 
"#ff7f0e", 
"#2ca02c"]). So, how can we perform this kind of mapping using D3 scales? This recipe is dedicated to address this kind of question.

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

This kind of ordinal mapping is quite common in data visualization. For example, you might want to map certain data points through categorization into some textual value or perhaps into RGB color code, which in turn can be used in CSS styling. D3 offers a specialized scale implementation to handle this kind of mapping. We will explore its usage here. Here is the code of the ordinal.scale.html file:

```html
<div id="alphabet" class="clear">
  <span>Ordinal Scale with Alphabet</span>
  <span>Mapping [1..10] to ["a".."j"]</span>
```

```html
```
Tipping the Scales

```javascript
var max = 10, data = [];

for (var i = 0; i < max; ++i) data.push(i); // <-A

var alphabet = d3.scale.ordinal() // <-B
  .domain(data)
  .range(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']);

function render(data, scale, selector) { // <-C
  d3.select(selector).selectAll('div.cell')
    .data(data)
    .enter().append('div').classed('cell', true);

d3.select(selector).selectAll('div.cell')
  .data(data)
  .exit().remove();

d3.select(selector).selectAll('div.cell')
  .data(data)
  .style('display', 'inline-block')
  .style('background-color', function(d) { // <-D
    return scale(d).indexOf('#')>=0?scale(d):'white';
  })
  .text(function (d) { // <-E
```

---
Chapter 4

```
return scale(d);
```

```html
render(data, alphabet, "#alphabet"); // <-F
render(data, d3.scale.category10(), "#category10");
render(data, d3.scale.category20(), "#category20");
render(data, d3.scale.category20b(), "#category20b");
render(data, d3.scale.category20c(), "#category20c"); // <-G
</script>
```

The preceding code outputs the following in your browser:

<table>
<thead>
<tr>
<th>Ordinal Scale with Alphabet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping [1..10] to [&quot;a&quot;...&quot;&quot;]</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>16777b4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ordinal Color Scale Category 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping [1..10] to category 10 colors</td>
</tr>
<tr>
<td>#2ca02c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ordinal Color Scale Category 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping [1..10] to category 20 colors</td>
</tr>
<tr>
<td>#2ca02c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ordinal Color Scale Category 20b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping [1..10] to category 20b colors</td>
</tr>
<tr>
<td>#2ca02c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ordinal Color Scale Category 20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping [1..10] to category 20c colors</td>
</tr>
<tr>
<td>#2ca02c</td>
</tr>
</tbody>
</table>

How it works...

In the above code example, a simple data array containing integers from 0 to 9 is defined on line A:

```javascript
for (var i = 0; i < max; ++i) data.push(i); // <-A
```

```javascript
var alphabet = d3.scale.ordinal() // <-B
  .domain(data)
  .range(["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"]);
```

Then an ordinal scale was created using the d3.scale.ordinal function on line B. The domain of this scale was set to our integer array data while range is set to a list of alphabets from a to j.

With this scale defined, we can perform the mapping by simply invoking the scale function, for example, alphabet(0) will return a, alphabet(4) will return e, and so on.
On line C, the render function was defined to generate a number of div elements on the page to represent the 10 elements in a data array. Each div has its background-color set to scale function's output or white if the output is not an RGB color string:

```
.style("background-color", function(d){  // <-D
    return scale(d).indexOf("#")>=0 ? scale(d) : "white";
})
```

On line E, we also set the text of each cell to display scale function's output:

```
.text(function (d) { // <-E
    return scale(d);
});
```

Now, with all the structures in place, from line F to G, the render function was repetitively called with different ordinal scales to produce different visual outputs. On line F, calling render with the alphabet ordinal scale produces the following output:

```
a b c d e f g h i j
```

Alphabetic ordinal scale

While on line G, calling the render function with the built-in d3.scale.category20c ordinal color scale produces the following output:

```
#3182bd #6baed6 #9ecae1 #c6dbef #74c476 #e6550d #fd8d3c #fdae6b #fdd0a2 #31a354 #6baed6
```

Color ordinal scale

Because assigning different colors to different elements in visualization is a common task, for example, assigning different colors in Pie and Bubble charts, D3 provides a number of different built-in ordinal color scales as we have seen in this recipe.

It is quite easy to build your own simple custom ordinal color scale. Just create an ordinal scale with the range set to the colors you want to use, for example:

```
d3.scale.ordinal()
    .range(["#1f77b4", "#ff7f0e", "#2ca02c"]);
```

**Interpolating a string**

In some cases, you might need to interpolate numbers embedded in a string; perhaps a CSS style for font.
In this recipe, we will examine how you can do that using D3 scale and interpolation. However, before we jump right into string interpolation, a bit of background research on interpolator is due and the following section will cover what interpolation is and how D3 implements interpolator functions.

**Interpolator**

In the first three recipes, we have gone over three different D3 scale implementations, now it is time to delve a little deeper into D3 scales. You are probably already asking the question, "How different scale knows what value to use for different inputs?" In fact this question can be generalized to:

> We are given the values of a function \( f(x) \) at different points \( x_0, x_1, \ldots, x_n \). We want to find approximate values of the function \( f(x) \) for "new" \( x \)'s that lie between these points. This process is called **interpolation**.

*Kreyszig E & Kreyszig H & Norminton E. J. (2010)*

Interpolation is not only important in scale implementation but also essential to many other core D3 capabilities, for example, animation and layout management. It is because of this essential role, D3 has designed a separate and re-usable construct called **interpolator** so that this common cross-functional concern can be addressed in a centralized and consistent fashion. Let's take a simple interpolator as an example:

```javascript
var interpolate = d3.interpolateNumber(0, 100);
interpolate(0.1); // => 10
interpolate(0.99); //=> 99
```

In this simple example, we created a D3 number interpolator with a range of \([0, 100]\). The `d3.interpolateNumber` function returns an `interpolate` function which we can use to perform number-based interpolations. The `interpolate` function is an equivalent to the following code:

```javascript
function interpolate(t) {
    return a * (1 - t) + b * t;
}
```

In this function, \(a\) represents the start of the range and \(b\) represents the end of the range. The parameter \(t\) passed into the `interpolate()` function, is a float-point number ranging from 0 to 1, and it signifies how far the return value is from \(a\).

D3 provides a number of built-in interpolators. Due to limited scope in this book, we will focus on some of the more interesting interpolators for the next few recipes; we are ending our discussion on simple number interpolation here. Nevertheless, the fundamental approach and mechanism remains the same whether it is a number or an RGB color code interpolator.
Tipping the Scales

For more details on number and round interpolation, please refer to the D3 reference documents at https://github.com/mbostock/d3/wiki/Transitions#wiki-d3_interpolateNumber

Now with general interpolation concepts behind, let's take a look at how string interpolator works in D3.

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

String interpolator finds the numbers embedded in the string, then performs interpolation using D3 number interpolator:

```javascript
var sizeScale = d3.scale.linear() // <-A
 .domain([0, max])
 .range([ // <-B
   "italic bold 12px/30px Georgia, serif",
   "italic bold 120px/180px Georgia, serif"
 ]);

for (var i = 0; i < max; ++i){ data.push(i); }

function render(data, scale, selector) { // <-C
  d3.select(selector).selectAll("div.cell")
    .data(data)
    .enter().append("div").classed("cell", true)
    .append("span");

  d3.select(selector).selectAll("div.cell")
```
The preceding code produces the following output:

```
Font Interpolation

0 1 2 3 4 5 6 7 8 9 10

String interpolation
```

**How it works...**

In this example, we created a linear scale on line A with range specified between two strings representing start and end font styles:

```javascript
var sizeScale = d3.scale.linear() // <-A
  .domain([0, max])
  .range([ // <-B
    "italic bold 12px/30px Georgia, serif",
    "italic bold 120px/180px Georgia, serif"
  ]);  
```

As you can see in the code of the string-interpolation.html file, the font style strings contain font-size numbers 12px/30px and 120px/180px, which we want to interpolate in this recipe.
On line C, the `render()` function simply creates 10 cells containing each one's index numbers (line E) styled using interpolated font style string calculated on line D.

```
.style("font", function(d,i){
    return scale(d); // <-D
})
.text(function(d,i){return i;}); // <-E
```

**There's more...**

Though we demonstrated string interpolation in D3 using a CSS font style as an example, D3 string interpolator is not only limited handling CSS styles. It can basically handle any string, and interpolates the embedded number as long as the number matches the following **Regex pattern**:

```
/[-+]?(?:\d+\.?\d*|\.?\d+)(?:[eE][-+]?)\d+/g
```

When generating a string using interpolation, very small values, when stringified, may get converted to scientific notation, for example, `1e-7`. To avoid this particular conversion, you need to keep your value larger than `1e-6`.

**Interpolating colors**

It is sometimes necessary to interpolate colors when you are interpolating values that do not contain numbers but rather RGB or HSL color code. This recipe addresses the question **how can you define scales for color codes and perform interpolation on them?**

**Getting Ready**

Open your local copy of the following file in your web browser:


**How to do it...**

Color interpolation is such a common operation in visualization that D3 actually provides four different kinds of interpolators dedicated for color supporting—RGB, HSL, L*a*b*, and HCL color space. In this recipe, we will demonstrate how color interpolation can be performed in RGB color space. However, all other color interpolators work in the same way.
D3 color interpolate function always returns interpolated color in RGB space no matter what the original color space it is since not all browsers support HSL or L*a*b* color spaces.

```javascript
var max = 21, data = [];

var colorScale = d3.scale.linear() // <-A
 .domain([0, max])
 .range(['white', '#4169e1']);

function divergingScale(pivot) { // <-B
 var divergingColorScale = d3.scale.linear()
 .domain([0, pivot, max]) // <-C
 .range(['white', '#4169e1', 'white']);
 return divergingColorScale;
 }

for (var i = 0; i < max; ++i) data.push(i);

function render(data, scale, selector) { // <-D
 d3.select(selector).selectAll("div.cell")
 .data(data)
 .enter()
 .append("div")
 .classed("cell", true)
 .append("span");

d3.select(selector).selectAll("div.cell")
 .data(data)
 .exit().remove();

d3.select(selector).selectAll("div.cell")
 .data(data)
 .style("display", "inline-block")
 .style("background-color", function(d){
```
The preceding code produces the following visual output:

The first step in this recipe is defining a linear color scale on line A with its range set as ["white", "+#4169e1"].

```
var colorScale = d3.scale.linear() // <-A
 .domain([0, max])
 .range(["white", "+#4169e1"]);
```

As we have demonstrated earlier, D3 color interpolator is pretty smart when it comes to color space. Similar to your browser, it understands both color keywords and hexadecimal values.

One new technique used in this recipe, that we haven't encountered yet, is the **poly-linear scale**, which is defined in the `divergingScale` function on line B.

```
function divergingScale(pivot) { // <-B
 var divergingColorScale = d3.scale.linear()
 .domain([0, pivot, max]) // <-C
 .range(["white", "+#4169e1", "white"]);
 return divergingColorScale;
}
```
A poly-linear scale is a scale with non-uniformed linear progression. It is achieved by providing a poly-linear domain on a linear scale as we can see on line C. You can think of a poly-linear scale as stitching two linear scales with different domains together. So this poly-linear color scale is effectively the two following linear scales combined together.

```javascript
    d3.scale.linear()
        .domain([0, pivot]).range(["white", "#4169e1"]);
    d3.scale.linear()
        .domain([pivot, max]).range(["#4169e1", "white "]);
```

No surprise in the rest of the recipe. The `render()` function defined on line D generates 20 cells that are numbered by its index and colored using the output of two color scales we defined earlier. Clicking the buttons on the web page (such as **Pivot at 5**) will show you the effect of pivoting at different positions in a poly-linear color scale.

**See also**

- For a complete list of supported color keywords in CSS3, please refer to W3C official reference [http://www.w3.org/TR/css3-color/#html4](http://www.w3.org/TR/css3-color/#html4)

**Interpolating compound objects**

There will be cases when what you need to interpolate in your visualization is not a simple value but rather an object consisting of multiple and different values, for example, a rectangular object with width, height, and color attributes. Fortunately, D3 has a built-in support for this type of compound object interpolation.

**Getting Ready**

Open your local copy of the following file in your web browser:


**How to do it...**

In this recipe, we will examine how compound object interpolation is performed in D3. The code for the `compound-interpolation.html` file is as follows:

```html
    <div id="compound" class="clear">
        <span>Compound Interpolation<br></span>
    </div>
```
Tipping the Scales

```javascript
<script type="text/javascript">
    var max = 21, data = [];

    var compoundScale = d3.scale.pow()
        .exponent(2)
        .domain([0, max])
        .range([{
            color: "#add8e6", height: "15px"}, // <-A
            {color: "#4169e1", height: "150px"} // <-B
        ]); // <-A

    for (var i = 0; i < max; ++i) data.push(i);

    function render(data, scale, selector) { // <-C
        d3.select(selector).selectAll("div.v-bar")
            .data(data)
            .enter().append("div").classed("v-bar", true)
            .append("span");

        d3.select(selector).selectAll("div.v-bar")
            .data(data)
            .exit().remove();

        d3.select(selector).selectAll("div.v-bar")
            .data(data)
            .classed("v-bar", true)
            .style("height", function(d){ // <-D
                return scale(d).height;
            })
            .style("background-color", function(d){ // <-E
                return scale(d).color;
            })
            .select("span")
            .text(function(d,i){return i;});
    }

    render(data, compoundScale, ";#compound");
</script>
```
The preceding code generates the following visual output:

![Compound object interpolation](image)

**How it works...**

This recipe is different from the previous recipes of this chapter by the fact that the scale we use in this recipe has a range defined using two objects rather than simple primitive data types:

```javascript
var compoundScale = d3.scale.pow()
    .exponent(2)
    .domain([0, max])
    .range([{
        color: "#add8e6", height: "15px"}, // <-A
        {color: "#4169e1", height: "150px"} // <-B
    ]);
```

We can see on line A and B that the start and end of the scale range are two objects which contain two different kinds of values; one for RGB color and the other one for CSS height style. When you interpolate this kind of a scale containing compound range, D3 will iterate through each of the fields inside an object and recursively apply the simple interpolation rule on each one of them. Thus, in other words, for this example, D3 will interpolate the color field using color interpolation from #add8e6 to #4169e1 while using string interpolation on height field from 15px to 150px.

The recursive nature of this algorithm allows D3 to interpolate on even nested objects. Therefore you can interpolate on an object like this:

```javascript
{
    color: "#add8e6",
    size:
        {
            height: "15px",
            width: "25px"
        }
}
```
A compound scale function, when invoked, returns a compound object that matches the given range definition:

```
.style("height", function(d){
  return scale(d).height; // <-D
})
.style("background-color", function(d){
  return scale(d).color; // <-E
})
```

As we can see on line D and E, the returned value is a compound object, and this is why we can access its attribute to retrieve the interpolated values.

Though it is not a common case, if the start and end of your compound scale range do not have identical attributes, D3 won't complain but rather it will just treat the missing attribute as a constant. The following scale will render the height to be 15px for all the div elements:

```
var compoundScale = d3.scale.pow()
  .exponent(2)
  .domain([0, max])
  .range([{
    color:"#add8e6", height:"15px"}, // <-A
    {color:"#4169e1"} // <-B
  });
```

**Implementing a custom interpolator**

In some rare cases, you might find that the built-in D3 interpolators are not enough to handle your visualization requirement. In such situations, you can choose to implement your own interpolator with specific logic to handle your needs. In this recipe, we will examine this approach and demonstrate some interesting use cases.

**Getting Ready**

Open your local copy of the following file in your web browser:

Let's take a look at two different examples of custom interpolator implementation. In the first example, we will implement a custom function capable of interpolating price in dollars, while in the second one we will implement custom interpolator for alphabets. Here is the code to this implementation:

```javascript
<script type="text/javascript">
  d3.interpolators.push(function(a, b) { // <-A
    var re = /^\$\([0-9,.]+\)$/, // <-B
        ma, mb, f = d3.format(",.02f");
    if ((ma = re.exec(a)) && (mb = re.exec(b))) { // <-C
      a = parseFloat(ma[1]);
      b = parseFloat(mb[1]) - a; // <-D
      return function(t) { // <-E
        return "$" + f(a + b * t); // <-F
      };
    }
  });

  d3.interpolators.push(function(a, b) { // <-G
    var re = /^([a-z])$/, ma, mb; // <-H
    if ((ma = re.exec(a)) && (mb = re.exec(b))) { // <-I
      a = a.charCodeAt(0);
      var delta = a - b.charCodeAt(0); // <-J
      return function(t) { // <-K
        return String.fromCharCode(Math.ceil(a - delta * t));
      };
    }
  });

  var dollarScale = d3.scale.linear()
  .domain([0, 11])
  .range(['"$0", "$300"]); // <-L

  var alphabetScale = d3.scale.linear()
  .domain([0, 27])
  .range(['"a", "z"]); // <-M

  function render(scale, selector) {
    var data = [];
  }
</script>
```
var max = scale.domain()[1];

for (var i = 0; i < max; ++i) data.push(i);

d3.select(selector).selectAll("div.cell")
  .data(data)
  .enter()
    .append("div")
      .classed("cell", true)
      .append("span");

d3.select(selector).selectAll("div.cell")
  .data(data)
  .exit().remove();

d3.select(selector).selectAll("div.cell")
  .data(data)
  .style("display", "inline-block")
  .select("span")
    .text(function(d, i){return scale(d);}); // <-N

render(dollarScale, "#dollar");
render(alphabetScale, "#alphabet");

The preceding code generates the following visual output:

<table>
<thead>
<tr>
<th>Custom Dollar Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.00</td>
</tr>
<tr>
<td>$115.38</td>
</tr>
<tr>
<td>$207.69</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Custom Alphabet Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>k</td>
</tr>
<tr>
<td>o</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

Custom interpolation
Chapter 4

How it works...
The first custom interpolator we encounter in this recipe is defined on line A. The custom
interpolator function is a bit more involved, so, let's take a closer look at how it works:
d3.interpolators.push(function(a, b) { // <-A
var re = /^\$([0-9,.]+)$/, // <-B
ma, mb, f = d3.format(",.02f");
if ((ma = re.exec(a)) && (mb = re.exec(b))) { // <-C
a = parseFloat(ma[1]);
b = parseFloat(mb[1]) - a; // <-D
return function(t) { // <-E
return "$" + f(a + b * t); // <-F
};
}
});

This custom interpolator in the following link was directly extracted from
D3 Wiki:
Transitions#wiki-d3_interpolators

On the line A, we push an interpolator function into d3.interpolators. This is a global
interpolator registry array that contains all known registered interpolators. By default, this
registry contains the following interpolators:
ff

Number interpolator

ff

String interpolator

ff

Color interpolator

ff

Object interpolator

ff

Array interpolator

Any new custom interpolator implementation can be pushed to the tail of the interpolators
array which then becomes globally available. An interpolator function is expected to be a
factory function that takes the start of the range (a) and the end of the range (b) as its input
parameters while returning an implementation of the interpolate function as seen on line E.
You might be wondering how D3 knows which interpolator to use when a certain string value
is presented. The key to this lies on line B. Typically we use a variable called re defined as a
regex pattern of /^\$([0-9,.]+)$/, which is then used to match both parameter a and b
for any number with a leading dollar sign. If both parameters match the given pattern then
the matching interpolate function is constructed and returned; otherwise D3 will continue
iterating through d3.interpolators array to find a suitable interpolator.

95


Instead of an array, `d3.interpolators` is actually better considered as a FILO stack (though not exactly implemented as a stack), where new interpolators can be pushed to the top of the stack. When selecting an interpolator, D3 will pop and check each suitable interpolator from the top. Therefore, in this case, the interpolator pushed later in the stack takes precedence.

The anonymous `interpolate()` function created on line E takes a single parameter `t` with a value ranging from 0 to 1 indicating how far off the interpolated value is from base `a`.

```javascript
return function(t) { // <-E
  return "$" + f(a + b * t); // <-F
};
```

You can think of it as a percentage of how far the desired value has traveled from `a` to `b`. With that in mind, it becomes clear that in line F it performs the interpolation and calculates the desired value based on the offset `t`, which effectively interpolates a price string.

One thing to watch out for here is that the `b` parameter's value has been changed on line D from the end of the range to the difference between `a` and `b`.

```javascript
b = parseFloat(mb[1]) - a; // <-D
```

This is generally considered a bad practice for readability. So, in your own implementations you should avoid modifying input parameters' value in a function.

On the line G, a second custom interpolator was registered to handle single-character lowercase alphabets from `a` to `z`:

```javascript
d3.interpolators.push(function(a, b) { // <-G
  var re = /^([a-z])$/, ma, mb; // <-H
  if ((ma = re.exec(a)) && (mb = re.exec(b))) { // <-I
    a = a.charCodeAt(0);
    var delta = a - b.charCodeAt(0); // <-J
    return function(t) { // <-K
      return String.fromCharCode(Math.ceil(a - delta * t));
    };
  }
});
```

We quickly noticed that this interpolator function follows a very similar pattern with the previous one. Firstly, it has a regex pattern defined on line H that matches single lowercase alphabets. After the matching is conducted on line I, the start and end of the range a and b were both converted from character values into integer values. A difference between `a` and `b` was calculated on line J. The interpolate function again follows exactly the same formula as the first interpolator as shown on line K.
Once these custom interpolators are registered with D3, we can define scales with corresponding ranges without doing any additional work and we will be able to interpolate their values:

```javascript
var dollarScale = d3.scale.linear()
    .domain([0, 11])
    .range([$0", "$300"]); // <-L

var alphabetScale = d3.scale.linear()
    .domain([0, 27])
    .range(["a", "z"]); // <-M
```

As expected, the `dollarScale` function will automatically use the price interpolator, while the `alphabetScale` function will use our alphabet interpolator, respectively. No additional work is required when invoking the scale function to obtain the value we need, as demonstrated on line N:

```javascript
.text(function(d,i){
    return scale(d);} // <-N
);
```

In isolation, custom interpolator does not appear to be a very important concept; however, later on when exploring other D3 concepts in Chapter 6, Transition with Style, we will explore more powerful techniques when custom interpolator is combined with other D3 constructs to achieve interesting custom effects.

**See also**

- If the regular expression used in this chapter is a new concept or a well-known tool in your toolbox and you need a little bit of dusting, you can find a lot of useful resources at http://www.regular-expressions.info
In this chapter we will cover:

- Working with basic axes
- Customizing ticks
- Drawing grid lines
- Dynamic rescaling of axes

**Introduction**

D3 was initially released without the built-in support of the Axis component. This situation did not last long; since axes are the universal building blocks in many Cartesian coordinate system-based visualization projects, it quickly became clear that D3 needs to provide built-in support for axes. Therefore, it was introduced quite early on and is continuously being enhanced ever since it was released. In this chapter, we will explore the usage of the Axis component and some related techniques.

**Working with basic axes**

In this recipe, we will focus on introducing the basic concepts and supports of the Axis component in D3 while covering different types and features of Axis as well as their SVG structures.

**Getting Ready**

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter5/basic-axes.html
Playing with Axes

**How to do it...**

Let's first take a look at the following code sample:

```html
<div class="control-group">
  <button onclick="renderAll('bottom')">horizontal bottom</button>
  <button onclick="renderAll('top')">horizontal top</button>
  <button onclick="renderAll('left')">vertical left</button>
  <button onclick="renderAll('right')">vertical right</button>
</div>

<script type="text/javascript">
var height = 500,
    width = 500,
    margin = 25,
    offset = 50,
    axisWidth = width - 2 * margin,
    svg;

function createSvg(){ // <-A
    svg = d3.select("body").append("svg") // <-B
        .attr("class", "axis") // <-C
        .attr("width", width)
        .attr("height", height);
}

function renderAxis(scale, i, orient){
    var axis = d3.svg.axis() // <-D
        .scale(scale) // <-E
        .orient(orient) // <-F
        .ticks(5); // <-G

    svg.append("g")
        .attr("transform", function(){ // <-H
            if("["top", "bottom"]" .indexOf(orient) >= 0)
                return "translate(+margin+," +i*offset+")");
```
The preceding code produces a visual output with only the four buttons shown in the following screenshot. Once you click on **horizontal bottom**, it shows the following:
How it works...

The first step in this recipe is to create the `svg` element which will be used to render our axes. This is done by the `createSvg` function, which is defined on line A, and using D3 `append` and `attr` modifier functions shown on line B and C.

Let's look at how we created the SVG canvas in the following code:

```javascript
var height = 500,
    width = 500,
    margin = 25,
    offset = 50,
    axisWidth = width - 2 * margin,
    svg;

function createSvg(){ // <-A
    svg = d3.select("body").append("svg") // <-B
        .attr("class", "axis") // <-C
        .attr("width", width)
        .attr("height", height);
}
```
Now we are ready to render the axes on this svg canvas. The renderAxis function is designed to do exactly just that. On line D, we first create an Axis component using the d3.svg.axis function:

```javascript
var axis = d3.svg.axis() // <-D
  .scale(scale) // <-E
  .orient(orient) // <-F
  .ticks(5); // <-G
```

D3 Axis is designed to work out of the box with D3 quantitative, time, and ordinal scales. Axis scale is provided using the scale() function (see line E). In this example, we render three different axes with the following scales:

```javascript
d3.scale.linear().domain([0, 1000]).range([0, axisWidth])
d3.scale.pow().exponent(2).domain([0, 1000]).range([0, axisWidth])
d3.time.scale()
  .domain([new Date(2012, 0, 1), new Date()])
  .range([0, axisWidth])
```

The second customization we have done with the axis object is its orient. Orient tells D3 how this axis will be placed (orientation), therefore, how it should be rendered, whether horizontally or vertically. D3 supports four different orient configurations for an axis:

- top: A horizontal axis with labels placed on top of the axis
- bottom: A horizontal axis with labels placed at the bottom of the axis
- left: A vertical axis with labels placed on the left hand side of the axis
- right: A vertical axis with labels placed on the right hand side of the axis

On line G, we set the number of ticks to 5. This tells D3, ideally how many ticks we want to render for this axis, however, D3 might choose to render slightly more or less ticks based on the available space and its own calculation. We will explore Axis ticks configuration in detail in the next recipe.

Once the axis is defined, the final step in this creation process is to create a svg:g container element, which will then be used to host all SVG structures that are required to render an axis:

```javascript
svg.append("g")
  .attr("transform", function(){ // <-H
    if(["top", "bottom"].indexOf(orient) >= 0)
      return "translate(" + margin + "," + i * offset + ")";
    else
      return "translate(" + i * offset + ", " + margin + ")";
  })
  .call(axis); // <-I
```
Having a `g` element to contain all SVG elements related to an axis is not only a good practice but also a requirement of D3 axis component.

Most of the logic in this code snippet is related to the calculation of where to draw the axis on `svg` canvas using the `transform` attribute (see line 11). In the preceding code example, to shift the axis using offsets we used the `translate` SVG transformation, which allows us to shift an element using a distance parameter that is defined with the coordinates in `x` and `y`.

```
# SVG transformation will be discussed in detail in Chapter 7, Getting into Shape, or you can refer to the following URL for more information on this topic:
http://www.w3.org/TR/SVG/coords.html#TranslationDefined
```

The more relevant part of this code is on line 12, where the `d3.selection.call` function is used with the `axis` object being passed in as the parameter. The `d3.selection.call` function invokes the given function (in our case the `axis` object) with the current selection passed in as an argument. In other words, the function being passed into the `d3.selection.call` function should have the following form:

```
function foo(selection) {
  
  
}
```

The `d3.selection.call` function also allows you to pass in additional arguments to the invoking function. For more information visit the following link:

```
https://github.com/mbostock/d3/wiki/Selections#wiki-call
```

Once the D3 Axis component is called, it will take care of the rest and automatically create all necessary SVG elements for an axis (see line 13). For example, the horizontal-bottom time axis in the example shown in this recipe shown in this recipe has the following complicated SVG structure automatically generated, which we don't really need to know much about:
Customizing ticks

We already saw how to use the `ticks` function in the previous recipe. This is the simplest ticks-related customization you can do on a D3 axis. In this recipe, we will cover some of the most common and useful ticks-related customizations with D3 axis.

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

In the following code example, we will customize the sub-ticks, padding, and formatting of its label. Let's take a look at the code snippet first:

```javascript
var height = 500,
    width = 500,
    margin = 25,
    axisWidth = width - 2 * margin;

var svg = d3.select("body").append("svg")
    .attr("class", "axis")
    .attr("width", width)
```
Playing with Axes

```
    .attr("height", height);

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

    var axis = d3.svg.axis()
        .scale(scale)
        .ticks(5)
        .tickSubdivide(5) // <-A
        .tickPadding(10) // <-B
        .tickFormat(function(v){ // <-C
            return v + "%";
        });

    svg.append("g")
        .attr("transform", function(){
            return "translate(" + margin + "," + margin + ");
        })
        .call(axis);
```

The preceding code generates the following visual output:

![Customized Axis ticks](image)

**How it works...**

The focus of this recipe is the highlighted lines after the `ticks` function. As we have mentioned before, the `ticks` function provides D3 a hint on how many ticks an axis should contain. After setting the number of ticks, in this recipe, we continue to customize the ticks through further function calls. On line A, the `tickSubdivide` function is used to similarly provide a hint to D3 on the number of subdivides an axis should render between each tick. Then on line B, the `tickPadding` function was used to specify the amount of space (in pixels) between tick labels and the axis. Finally, a custom function called `tickFormat` was provided on line C to append a percentage sign to the value.
For more information on the aforementioned functions and other ticks-related customizations visit the D3 Wiki at the following URL:

Drawing grid lines

Quite often, we need horizontal and vertical grid lines to be drawn in consistency with ticks on both x and y axes. As we have shown in the previous recipe, typically we don’t have or don’t want to have precise control of how ticks are rendered on D3 axes. Therefore, we might not know how many ticks are present and their values, before they are rendered. This is especially true if you are building a re-usable visualization library where it is impossible to know the tick configuration ahead of time. In this recipe, we will explore some useful techniques of drawing consistent grid lines on Axis without actually needing to know the tick values.

Getting Ready

Open your local copy of the following file in your web browser:

How to do it...

First, let’s take a look at how we draw grid lines in code:

```javascript
<script type="text/javascript">

  var height = 500,
  width = 500,
  margin = 25;

  var svg = d3.select("body").append("svg")
    .attr("class", "axis")
    .attr("width", width)
    .attr("height", height);

  function renderXAxis(){
    var axisLength = width - 2 * margin;

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

Playing with Axes

```
.range([0, axisLength]);

var xAxis = d3.svg.axis()
  .scale(scale)
  .orient("bottom");

svg.append("g")
  .attr("class", "x-axis")
  .attr("transform", function(){ // <-A
    return "translate(" + margin + "," + (height - margin)
  + ")";
  })
  .call(xAxis);

d3.selectAll("g.x-axis g.tick") // <-B
  .append("line") // <-C
    .classed("grid-line", true)
    .attr("x1", 0) // <-D
    .attr("y1", 0)
    .attr("x2", 0)
    .attr("y2", - (height - 2 * margin)); // <-E
}

function renderYAxis(){
  var axisLength = height - 2 * margin;

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

  var yAxis = d3.svg.axis()
    .scale(scale)
    .orient("left");

  svg.append("g")
    .attr("class", "y-axis")
    .attr("transform", function(){
      return "translate(" + margin + "," + margin + ")";
    })
    .call(yAxis);

d3.selectAll("g.y-axis g.tick")
  .append("line")
    .classed("grid-line", true)
```
Axes and grid lines

The previous code generates the following visual output:
Playing with Axes

How it works...

In this recipe, two axes x and y were created in the renderXAxis and renderYAxis functions, respectively. Let’s take a look at how the x axis was rendered.

Once we understand how to render x axis and its grid lines, the logic used to render y axis can be easily understood since they are almost identical. The x axis and its scale were defined with no complications, as we have already demonstrated a number of times throughout this chapter. An svg:g element was created to contain the x axis structures. This svg:g element was placed at the bottom of the chart using a translate transformation, as shown on line A:

```
.attr("transform", function(){ // <-A
    return "translate(" + margin + "," + (height - margin) + ")";
})
```

It is important to remember that the translate transformation changes the frame of reference for coordinates when it comes to any of its sub-elements. For example, within this svg:g element, if we create a point with its coordinates set as (0, 0), then when we draw this point on the SVG canvas, it will be actually placed as (margin, height - margin). This is because all subelements within the svg:g element are automatically transformed to this base coordinate, hence, the shift of the frame of reference. Equipped with this understanding, let’s see how dynamic grid lines can be generated after the axis is rendered:

```
d3.selectAll("g.x-axis g.tick") // <-B
    .append("line") // <-C
        .classed("grid-line", true)
        .attr("x1", 0) // <-D
        .attr("y1", 0)
        .attr("x2", 0)
        .attr("y2", - (height - 2 * margin));  // <-E
```

Once the axis has been rendered, we can select all the ticks elements on an axis by selecting the g.tick, since each of them is grouped by its own svg:g element (see line B). Then on line C, we append a new svg:line element to each svg:g tick element. SVG line element is the simplest shape provided by the SVG standard. It has four main attributes:

- **x1** and **y1** attributes define the point of origin of this line
- **x2** and **y2** attributes define the point of destination

In our case, here we simply set x1, y1, and x2 to 0, since each g.tick element has already been translated to its position on the axis, therefore, we only need to change the y2 attribute in order to draw a vertical grid line. The y2 attribute is set to -(height - 2 * margin). The reason why the coordinate is negative was because the entire g.x-axis has been shifted down to (height - margin), as mentioned in the previous code. Therefore, in absolute coordinate terms y2 = (height - margin) - (height - 2 * margin) = margin, which is the top of the vertical grid line we want to draw from the x axis.
In SVG coordinates, \((0, 0)\) denotes the top-left corner of the SVG canvas.

This is what the \(x\) axis in SVG structure with associated grid lines looks like:

```
<g class="x-axis" transform="translate(25,475)">
  <line y2="5" x2="0"></line>
  <text y="0" x="0" dy=".71em" style="text-anchor: middle;">0</text>
  <line class="grid-line" x1="0" y1="0" x2="0" y2="7.455"></line>
</g>

<g class="tick major" style="opacity: 1; transform=translate(45,0)"
  class="grid-line", true>
  <line y2="6" x2="0"></line>
  <text y="0" x="0" dy=".71em" style="text-anchor: middle;">0</text>
  <line class="grid-line" x1="0" y1="0" x2="0" y2="7.455"></line>
</g>
```

As we can see in the preceding screenshot, an `svg:line` element representing the grid line was added into the "g.tick" `svg:g` container element as discussed earlier in this section.

The \(y\) axis grid lines are generated using the `identical technique`; the only difference is that instead of setting the \(y2\) attribute on the grid lines, as we did for the \(x\) axis, we change the \(x2\) attribute, since the lines are horizontal (see line \(F\)):

```
d3.selectAll("g.y-axis g.tick")
  .append("line")
  .classed("grid-line", true)
  .attr("x1", 0)
  .attr("y1", 0)
  .attr("x2", axisLength);  // <- F
```

### Dynamic rescaling of axes

In some cases, the scale used by axes might change when triggered by user interaction or changes from data feeds. For example, a user might change the time range for the visualization. This kind of change also needs to be reflected by rescaling the axes. In this recipe, we will explore how this can be achieved dynamically while also redrawing the grid lines associated with each tick.
Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

Here is the code showing how to perform dynamic rescaling:

```
<script type="text/javascript">
var height = 500,
    width = 500,
    margin = 25,
    xAxis, yAxis, xAxisLength, yAxisLength;

var svg = d3.select("body").append("svg")
    .attr("class", "axis")
    .attr("width", width)
    .attr("height", height);

function renderXAxis(){
    xAxisLength = width - 2 * margin;

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

    xAxis = d3.svg.axis()
        .scale(scale)
        .tickSubdivide(1)
        .orient("bottom");

    svg.append("g")
        .attr("class", "x-axis")
        .attr("transform", function(){
            return "translate(" + margin + ","
                + (height - margin) + ");";
        })
        .call(xAxis);
}
```

function rescale() {  // <-A
    var max = Math.round(Math.random() * 100);  
    xAxis.scale().domain([0, max]); // <-B
    svg.select("g.x-axis")
        .transition()
        .call(xAxis); // <-C

    renderXGridlines();
}

function renderXGridlines() {
    var lines = d3.selectAll("g.x-axis g.tick")
        .select("line.grid-line")
        .remove(); // <-D
    lines = d3.selectAll("g.x-axis g.tick")
        .append("line")
        .classed("grid-line", true)
    lines.attr("x1", 0)
        .attr("y1", 0)
        .attr("x2", 0)
        .attr("y2", -yAxisLength);
}

renderXAxis(); renderXGridlines();
</script>
The preceding code generates the following effects:

Due to limited scope in this book, the y-axis-related code has been omitted from the code example in this recipe. See the code example available online for a complete reference.

**How it works...**

As soon as you click the ReScale button on the screen, you will notice both the axes rescale while all the ticks as well as grid lines get redrawn accompanied with a smooth transition effect. In this section, we will focus on how rescaling works and leave the transition effect for the next chapter Transition with Style. Most of the heavy lifting in this recipe is done by the rescale function defined on line A.

```javascript
function rescale(){ // <-A
  var max = Math.round(Math.random() * 100);

  xAxis.scale().domain([0, max]); // <-B
  svg.select("g.x-axis")
    .transition()
    .call(xAxis); // <-C

  renderXGridlines();
}
```
To rescale an axis, we simply change its domain (see line 2). If you recall, the scale domain represents the data domain, while its range corresponds to visual domain. Therefore, visual range should remain constant since we are not resizing the SVG canvas. Once updated, we call the `xAxis` again by passing in the `svg:g` element for the x axis (see line 3); this simple call will take care of the axis updating, hence, our job is done with the axis. In the next step, we also need to update and redraw all the grid lines since the domain change will also change all the ticks:

```javascript
function renderXGridlines(){
  var lines = d3.selectAll("g.x-axis g.tick")
    .select("line.grid-line")
    .remove(); // <D

  lines = d3.selectAll("g.x-axis g.tick")
    .append("line")
    .classed("grid-line", true)

  lines.attr("x1", 0)
    .attr("y1", 0)
    .attr("x2", 0)
    .attr("y2", -yAxisLength);
}
```

This is achieved by removing every grid line by calling the `remove()` function, as shown on line D, and then recreating the grid lines for all the new ticks on rescaled axes. This approach effectively keeps all grid lines consistent with the ticks during rescaling.
6 Transition with Style

In this chapter we will cover:

- Animating a single element
- Animating multiple elements
- Using ease
- Using tweening
- Using transition chaining
- Using transition filter
- Listening to transitional events
- Implementing custom interpolator
- Working with timer

Introduction

"A picture is worth a thousand words."

This age-old wisdom is arguably one of the most important cornerstones of data visualization. Animation on the other hand is generated using a series of still images in quick succession. Human eye-and-brain complex, through positive afterimage, phi phenomenon, and beta movement is able to create an illusion of continuous imagery. As Rick Parent put it perfectly in his brilliant work *Computer Animation Algorithms and Techniques*:

*Images can quickly convey a large amount of information because the human visual system is a sophisticated information processor. It follows, then, that moving images have the potential to convey even more information in a short time. Indeed, the human visual system has evolved to provide for survival in an ever-changing world; it is designed to notice and interpret movement.*

-Parent R. 2012
This is indeed the main goal of animation used in data visualization projects. In this chapter, we will focus on the mechanics of D3 transition, covering topics from the basics to more advanced ones, such as custom interpolation and timer-based transition. Mastering transition is not only going to add many bells and whistles to your otherwise dry visualization, but will also provide a powerful toolset to your visualization and otherwise hard-to-visualize attributes, such as trending and differences.

**What is Transition?**

D3 transition offers the ability to create computer animation with HTML and SVG elements on a web page. D3 transition implements an animation called **Interpolation-based Animation**. Computer's are especially well equipped for value interpolation, and therefore, most of the computer animations are interpolation-based. As its name suggests, the foundation for such animation capability is value interpolation.

If you recall, we have already covered D3 interpolators and interpolation functions in detail in *Chapter 4, Tipping the Scales*. Transition is built on top of interpolation and scales to provide the ability to change values over time, which produces animation. Each transition can be defined using a start and end value (also called **key frames** in animation), while different algorithms and interpolators will fill in the intermediate values frame-by-frame (also called "in-betweening" or simply "tweening"). At the first glance, if you are not already familiar with animation algorithms and techniques, this seems to be a somewhat less controlled way of creating an animation. However, it is quite the opposite in reality; interpolation-based transitions can provide direct and specific expectations about the motion produced down to each and every frame, thus offering tremendous control to the animator with simplicity. In fact, D3 transition API is so well-designed that, in most cases, only a couple of lines of code are enough to implement animations you need in a data visualization project. Now, let's get our hands dirty and try out some transitions to further improve our understanding on this topic.

**Animating a single element**

In this recipe, we will first take a look at the simplest case of transition—interpolating attributes on a single element over time to produce a simple animation.

**Getting Ready**

Open your local copy of the following file in your web browser:

How to do it...

The code necessary to perform this simple transition is extremely short; good news for any animator:

```javascript
<script type="text/javascript">
var body = d3.select("body"),
duration = 5000;

body.append("div") // <-A
  .classed("box", true)
  .style("background-color", "#e9967a") // <-B
  .transition() // <-C
  .duration(duration) // <-D
  .style("background-color", "#add8e6") // <-E
  .style("margin-left", "600px") // <-F
  .style("width", "100px")
  .style("height", "100px");
</script>
```

This code produces a moving, shrinking, and color-changing square, as shown in the following screenshot:

![Single element transition](image)

How it works...

You might be surprised to see that the extra code we have added to enable this animation is only on line C and D:

```javascript
body.append("div") // <-A
  .classed("box", true)
  .style("background-color", "#e9967a") // <-B
  .transition() // <-C
  .duration(duration) // <-D
```
Transition with Style

First on line C, we call the d3.selection.transition function to define a transition. Then, the transition function returns a transition-bound selection that still represents the same element(s) in the current selection. But now, it is equipped with additional functions and allows further customization of the transitional behavior. Line C returns a transition-bound selection of the div element we created on line A.

On line D, we set the duration of the transition to 5000 milliseconds using the duration() function. This function also returns the current transition-bound selection, thus allowing function chaining. As we have mentioned at the start of this chapter, interpolation-based animations usually only require specifying the start and end values while letting interpolators and algorithms fill the intermediate values over time. D3 transition treats all values set before calling the transition function as start values, with values set after the transition function call as end values. Hence in our example:

```javascript
  .style("background-color", "#e9967a") // <-B
```

The background-color style defined on line B is treated as the start value for transition. All styles set in the following lines are treated as end values:

```javascript
  .style("background-color", "#add8e6") // <-E
  .style("margin-left", "600px") // <-F
  .style("width", "100px")
  .style("height", "100px");
```

At this point, you might be asking, why these start and end values are not symmetric? D3 transition does not require every interpolated value to have explicit start and end values. If the start value is missing, then it will try to use the computed style, and if the end value is missing then the value will be treated as a constant. Once the transition starts, D3 will automatically pick the most suitable registered interpolator for each value. In our example, an RGB color interpolator will be used in line E, while a string interpolator—which internally uses number interpolators to interpolate embedded numbers—will be used for the rest of the style values.

Here we will list the interpolated style values with their start and end values:

- **background-color**: The start value #e9967a is greater than end value #add8e6
- **margin-left**: The start value is a computed style and is greater than end value 600px
- **width**: The start value is a computed style and it's greater than the end value 100px
- **height**: The start value is a computed style and greater than the end value 100px
Animating multiple elements

A more elaborate data visualization requires animating multiple elements instead of a single element, as demonstrated in the previous recipe. More importantly, these transitions often need to be driven by data and coordinated with other elements within the same visualization. In this recipe, we will see how a data-driven multielement transition can be created to generate a moving bar chart. New bars are added over time while the chart shifts from right to left with a smooth transition.

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

As expected, this recipe is slightly larger than the previous one, however, not by that much. Let's take a look at the code:

```javascript
var id = 0,
data = [],duration = 500,
chartHeight = 100,
chartWidth = 680;

for (var i = 0; i < 20; i++) {
push(data);
}

function render(data) {
  var selection = d3.select("body")
    .selectAll("div.v-bar")
    .data(data, function(d){return d.id;}); // <-A

  // enter
  selection.enter()
    .append("div")
      .attr("class", "v-bar")
      .style("position", "fixed")
```


Transition with Style

```javascript
.style("top", chartHeight + "px")
.style("left", function(d, i){
    return barLeft(i+1) + "px"; // <-B
})
.style("height", "0px") // <-C
.append("span");

// update
selection
  .transition().duration(duration) // <-D
  .style("top", function (d) {
    return chartHeight - barHeight(d) + "px";
  })
  .style("left", function(d, i){
    return barLeft(i) + "px";
  })
  .style("height", function (d) {
    return barHeight(d) + "px";
  })
  .select("span")
    .text(function (d) {return d.value;});

// exit
selection.exit()
  .transition().duration(duration) // <-E
  .style("left", function(d, i){
    return barLeft(-1) + "px"; //<-F
  })
  .remove(); // <-G
}

function push(data) {
    data.push({
        id: ++id,
        value: Math.round(Math.random() * chartHeight)
    });
}

function barLeft(i) {
    // start bar position is i * (barWidth + gap)
    return i * (30 + 2);
}

function barHeight(d) {
```
The preceding code generates a sliding bar chart in your web browser, as shown in the following screenshots:

How it works...

On the surface, this example seems to be quite complex with complicated effects. Every second a new bar needs to be created and animated while the rest of the bars need to slide over precisely. The beauty of D3 set-oriented functional API is that it works exactly the same way no matter how many elements you are manipulating; therefore, once you understand the mechanics, you will realize this recipe is not so much different than the previous one.
First step, we created a data-bound selection for a number of vertical bars (on line A), which can then be used in a classic enter-update-exit D3 pattern:

```javascript
var selection = d3.select("body")
    .selectAll("div.v-bar")
    .data(data, function(d){return d.id;}); // <-A
```

One thing we have not touched so far is the second parameter in the `d3.selection.data` function. Here, we know that this function is called an **object-identity function**. The purpose of using this function is to provide object constancy. What it means in simple terms is that we want the binding between data and visual element to be stable. In order to achieve object constancy, each datum needs to have a unique identifier. Once the ID is provided, D3 will ensure if a `div` element is bound to `{id: 3, value: 45}`. Then, the next time when the update selection is computed, the same `div` element will be used for the datum with the same `id`, though this time the value might get changed, for example, `{id: 3, value: 12}`. Object constancy is crucial in this recipe; without object constancy, the sliding effect will not work.

If you are interested to know more about object constancy, please check this excellent writing by Mike Bostock, the creator of D3 at the following link:

http://bost.ocks.org/mike/constancy/

The second step is to create these vertical bars with the `d3.selection.enter` function and compute the **left** position for each bar based on the index number (see line B):

```javascript
// enter
selection.enter()
    .append("div")
    .attr("class", "v-bar")
    .style("position", "fixed")
    .style("top", chartHeight + "px")
    .style("left", function(d, i){
        return barLeft(i+1) + "px"; // <-B
    })
    .style("height", "0px") // <-C
    .append("span");
```
Another point worth mentioning here is that in the **enter** section, we have not called **transition** yet, which means any value we specify here will be used as the start value in a transition. If you notice on line C, `bar height` is set to `0px`. This enables the animation of bars growing from zero height to the target height. At the same time, the same logic is applied to the **left** position of the bar (see line B) and was set to `barLeft(i+1)`, thus enabling the sliding transition we desired.

```javascript
// update
selection
  .transition().duration(duration) // <-D
  .style("top", function (d) {
    return chartHeight - barHeight(d) + "px";
  })
  .style("left", function(d, i){
    return barLeft(i) + "px";
  })
  .style("height", function (d) {
    return barHeight(d) + "px";
  })
  .select("span")
  .text(function (d) {return d.value;});
```

After completing the **enter** section, now we can take care of the **update** section, where the transition is defined. First of all, we want to introduce transition for all updates, therefore, we invoke the `transition` function before any style change is applied (see line D). Once the **transition-bound selection** is created, we applied the following style transitions:

- **top**: `chartHeight + px` > `chartHeight - barHeight(d)+"px"
- **left**: `barLeft(i+1) + px` > `barLeft(i) + px`
- **height**: `0px` > `barHeight(d) + px`

The aforementioned three style transitions are all you need to do to handle new bars as well as every existing bar and their sliding effect. Finally, the last case we need to handle here is the **exit** case, when a bar is no longer needed. So, we want to keep the number of bars constant on the page. This is handled in the **exit** section:

```javascript
// exit
selection.exit()
  .transition().duration(duration) // <-E
  .style("left", function(d, i){
    return barLeft(-1) + "px"; // <-F
  })
  .remove(); // <-G
```
So far in this book, prior to this chapter, we have always called the remove() function immediately after the d3.selection.exit function. This immediately removes the elements that are no longer needed. In fact, the exit() function also returns a selection, and therefore, can be animated before calling the remove() function. This is exactly what we did here, starting a transition on line E using the exit selection; then we animated the left value with the following transition change:

- left: `barLeft(i) + "px" > barLeft(i-1) + "px"

Since we are always removing the left-most bar, this transition moves the bar left and out of the SVG canvas, then removes it.

The exit transition is not necessarily limited to simple transitions, such as the one we have shown in this recipe. In some visualization, it could be as elaborate as the update transition.

Once the render function is in place with the defined transition, all that is left is to simply update the data and re-render our bar chart every second using the setInterval function. Now this completes our example.

**Using ease**

Transition can be thought of as a function of time. It is a function that maps time progression into numeric value progression, which then results in object motion (if the numeric value is used for positioning) or morphing (if the value is used to describe other visual attributes). Time always travels at a constant pace; in other words time progression is uniform (unless you are doing visualization near a black hole of course), however, the resulting value progression does not need to be uniform. Easing is a standard technique to provide flexibility and control to this kind of mapping. When a transition generates a uniform value progression, it is called linear easing. D3 provides support for different types of easing capabilities, and in this recipe, we will explore different built-in D3 easing functions, as well as how to implement custom easing functions with D3 transition.

**Getting Ready**

Open your local copy of the following file in your web browser:

In the following code example, we will demonstrate how transition easing can be customized on an element-by-element basis:

```javascript
var data = [ // <-A
  "linear", "cubic", "cubic-in-out",
  "sin", "sin-out", "exp", "circle", "back",
  "bounce",
  function(t){ // <-B
    return t * t;
  }
],
colors = d3.scale.category10();

d3.select("body").selectAll("div")
  .data(data) // <-C
  .enter()
  .append("div")
    .attr("class", "fixed-cell")
    .style("top", function (d, i) {
      return i * 40 + "px";
    })
    .style("background-color", function (d, i) {
      return colors(i);
    })
    .style("color", "white")
    .style("left", "500px")
    .text(function (d) {
      if(typeof d === 'function') return "custom";
      return d;
    });

d3.selectAll("div").each(function(d){
  d3.select(this)
    .transition().ease(d) // <-D
    .duration(1500)
    .style("left", "10px");
});
</script>
```
The preceding code produces a set of moving boxes with different easing effects. The following screenshot is captured at the time the easing effect takes place:

**How it works...**

In this recipe, we have shown a number of different built-in D3 ease functions and their effects on transition. Let’s take a look at how it is done. First, we have created an array to store different ease modes we want to demonstrate:

```javascript
var data = [ // <-A
    "linear",
    "cubic",
    "cubic-in-out",
    "sin",
    "exp",
    "circle",
    "back",
    "bounce",
    function(t){ // <-B
        return t * t;
    }
];
```
While all the built-in ease functions are defined simply using their name, the last element of this array is a custom easing function (**quadric easing**). Then afterwards, a set of div elements created using this data array and a transition with different easing functions was created for each of the div element, respectively, moving them from ("left", "500px") to ("left", "10px").

```javascript
  d3.selectAll("div").each(function(d){
    d3.select(this)
      .transition().ease(d) // <-D
      .duration(1500)
      .style("left", "10px");
  });
```

At this point, you might be asking, *Why did we not just specify easing using a function as we normally would have done for any other D3 attributes?*

```javascript
  .transition().ease(function(d){return d;}) // does not work
  .duration(1500)
  .style("left", "10px");
```

The reason is that it does not work on the ease() function. What we have shown on line D is a workaround of this limitation, though in real-world projects it is fairly rare that you will need to customize easing behavior on a per-element basis.

*Note that it is not possible to customize the easing function per-element or per-attribute;*

*D3 Wiki (2013, August)*

Another way to get around this limitation is by using custom tweening, which we will cover in the next recipe.

As seen on line D, specifying different ease function for D3 transition is very straight forward; all you need to do is call the ease() function on a transition-bound selection. If the pass-in parameter is a string, then D3 will try to find the matching function using the name; if not found it will default to **linear**. On top of named built-in ease functions, D3 also provides ease mode modifiers that you can combine with any ease function to achieve additional effects, for example, **sin-out** or **quad-out-in**. Available ease mode modifiers:

- in: default
- out: reversed
- in-out: reflected
- out-in: reversed and reflected
The default ease effect used by D3 is **cubic-in-out**. For the list of supported D3 ease functions please refer to the following link:

https://github.com/mbostock/d3/wiki/Transitions#wiki-d3_ease

When a custom ease function is used, the function is expected to take the current parametric time value as its parameter in the range of \([0, 1]\).

```javascript
function(t){ // <-B
    return t * t;
}
```

In our example, we have implemented a simple quadric easing function, which is actually available as a built-in D3 ease function, and is named as **quad**.

For more information on easing and Penner's equations (most of the modern JavaScript framework implementations including D3 and jQuery) check out the following link:

http://www.robertpenner.com/easing/

**Using tweening**

*Tween* comes from the word "inbetween", which is a common practice performed in traditional animation where after key frames were created by the master animator, less experienced animators were used to generate frames in between the key frames. This phrase is borrowed in modern computer-generated animation and it refers to the technique or algorithm controlling how the "inbetween" frames are generated. In this recipe, we will examine how the D3 transition supports tweening.

**Getting Ready**

Open your local copy of the following file in your web browser:

In the following code example, we will create a custom tweening function to animate a button label through nine discrete integral numbers:

```javascript
<script type="text/javascript">
  var body = d3.select("body"), duration = 5000;

  body.append("div").append("input")
    .attr("type", "button")
    .attr("class", "countdown")
    .attr("value", "0")
    .style("width", "150px")
    .transition().duration(duration).ease("linear")
    .style("width", "400px")
    .attr("value", "9");

  body.append("div").append("input")
    .attr("type", "button")
    .attr("class", "countdown")
    .attr("value", "0")
    .transition().duration(duration).ease("linear")
    .styleTween("width", widthTween) // <- A
    .attrTween("value", valueTween); // <- B

  function widthTween(a){
    var interpolate = d3.scale.quantize()
      .domain([0, 1])
      .range([150, 200, 250, 350, 400]);

    return function(t){
      return interpolate(t) + "px";
    }
  }

  function valueTween(){
    var interpolate = d3.scale.quantize() // <- C
      .domain([0, 1])
      .range([1, 2, 3, 4, 5, 6, 7, 8, 9]);

    return function(t){ // <- D
      return interpolate(t);
    }
  }
</script>
```
The preceding code generates two buttons morphing at a very different rate, and the following screenshot is taken while this process is going on:

![Transition with Style](image)

### How it works...

In this recipe, the first button was created using simple transition with linear easing:

```javascript
body.append("div").append("input")
  .attr("type", "button")
  .attr("class", "countdown")
  .attr("value", "0")
  .style("width", "150px")
  .transition().duration(duration).ease("linear")
  .style("width", "400px")
  .attr("value", "9");
```

The transition changes the button’s width from "150px" to "400px", while changing its value from "0" to "9". As expected, this transition simply relies on continuous linear interpolation of these values using D3 string interpolator. In comparison, the second button has the effect of changing these values in chunks. Moving from 1 to 2, then to 3, and so on up to 9. This is achieved using D3 tweening support with `attrTween` and `styleTween` functions. Let's first take a look at how the button value tweening works:

```javascript
transition().duration(duration).ease("linear")
.styleTween("width", widthTween) // <- A
.attrTween("value", valueTween); // <- B
```

In the preceding code snippet, we can see that instead of setting the end value for the value attribute as we have done in the case of the first button, we use `attrTween` function and offered a tweening function `valueTween`, which is implemented as the following:

```javascript
function valueTween(){
  var interpolate = d3.scale.quantize() // <-C
    .domain([0, 1])
    .range([1, 2, 3, 4, 5, 6, 7, 8, 9]);

  return function(t){ // <-D
    return interpolate(t);
  };
}
```
In D3, a tween function is expected to be a factory function, which constructs the actual function that will be used to perform the tweening. In this case, we have defined a quantize scale that maps the domain \([0, 1]\) to a discrete integral range of \([1, 9]\), on line C. The actual tweening function defined on line D simply interpolates the parametric time value using the quantize scale which generates the jumping integer effect.

Quantize scales are a variant of linear scale with a discrete range rather than continuous. For more information on quantize scales, please visit the following link:


**There's more...**

At this point we have touched upon all three concepts related to transition: ease, tween, and interpolation. Typically, D3 transition is defined and driven through all the three levels shown in the following sequence diagram:

As we have shown through multiple recipes, D3 transition supports customization in all three levels. This gives us tremendous flexibility to customize the transition behavior exactly the way we want.

Though custom tween is usually implemented using interpolation, there is no limit to what you can do in your own tween function. It is entirely possible to generate custom tween without using D3 interpolator at all.
We used linear easing in this recipe to highlight the effect of tweening, however, D3 fully supports eased tweening, meaning that you can combine any of the ease functions we have demonstrated in the previous recipe with your custom tween to generate even more complex transition effects.

**Using transition chaining**

The first four recipes in this chapter are focused on single transition controls in D3, including custom easing and tweening functions. However, sometimes regardless of how much easing or tweening you do, a single transition is just not enough, for instance, you want to simulate teleporting a `div` element by first squeezing the `div` element into a beam, then passing the beam to a different position on the web page, and finally restoring the `div` to its original size. In this recipe, we will see exactly how this type of transition can be achieved using transition chaining.

**Getting Ready**

Open your local copy of the following file in your web browser:


**How to do it...**

Our simple teleportation transition code is surprisingly short:

```javascript
<script type="text/javascript">
    var body = d3.select("body");

    function teleport(s){
        s.transition().duration(300) // <-A
            .style("width", "200px")
            .style("height", "1px")
        .transition().duration(100) // <-B
            .style("left", "600px")
        .transition().duration(300) // <-C
            .style("left", "800px")
            .style("height", "80px")
            .style("width", "80px");
    }

    body.append("div")
        .style("position", "fixed")
        .style("background-color", "steelblue")
</script>
```
The preceding code performs a div teleportation:

```javascript
function teleport(s){
    s.transition().duration(300) // <-A
        .style("width", "200px")
        .style("height", "1px")
    .transition().duration(100) // <-B
        .style("left", "600px")
    .transition().duration(300) // <-C
        .style("left", "800px")
        .style("height", "80px")
        .style("width", "80px");
}
```

How it works...

This simple teleportation effect was achieved by chaining a few transitions together. In D3, when transitions are chained, they are guaranteed to be executed only after the previous transition reaches its completion state. Now, let's see how this is done in the code:

```javascript
function teleport(s){
    s.transition().duration(300) // <-A
        .style("width", "200px")
        .style("height", "1px")
    .transition().duration(100) // <-B
        .style("left", "600px")
    .transition().duration(300) // <-C
        .style("left", "800px")
        .style("height", "80px")
        .style("width", "80px");
}
```

The first transition was defined and initiated on line A (compression), then on line B a second transition (beaming) was created, and finally the third transition is chained on line C (restoration). Transition chaining is a powerful yet simple technique to orchestrate a complex transition effect by stitching simple transitions together. Finally in this recipe, we have also shown a basic example on re-usable composite transition effect by wrapping the teleportation transition in a function and then applying it on a selection using the `d3.selection.call` function (see line D). Re-usable transition effect is essential to following the DRY (Don't Repeat Yourself) principle, especially when the animation in your visualization becomes more elaborate.
Using transition filter

Under some circumstances, you might find it necessary to selectively apply transition to a subset of a certain selection. In this recipe, we will explore this effect using data-driven transition filtering techniques.

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

In this recipe, we will move a set of div elements (or boxes) across the web page from right to left. After moving all the boxes to the left, we selectively move only the boxes that are marked with Cat back, so they won’t fight each other. Let’s see the following code:

```html
<script type="text/javascript">
    var data = ["Cat", "Dog", "Cat", "Dog", "Cat", "Dog", "Cat", "Dog"],
    duration = 1500;

    d3.select("body").selectAll("div")
        .data(data)
        .enter()
        .append("div")
        .attr("class", "fixed-cell")
        .style("top", function (d, i) {
            return i * 40 + "px";
        })
        .style("background-color", "steelblue")
        .style("color", "white")
        .style("left", "500px")
        .text(function (d) {
            return d;
        })
        .transition() // <- A
        .duration(duration)
        .style("left", "10px")
        .filter(function(d){return d == "Cat";}) // <- B
        .transition() // <- C
        .duration(duration)
        .style("left", "500px");
</script>
```
Here is what the page looks like after the transition:

![Transition filtering](image)

**How it works...**

The initial setup of this recipe is quite simple, since we want to keep the plumbing as minimal as possible which will help you focus on the core of the technique. We have a data array containing interlaced strings of "Cat" and "Dog". Then a set of div boxes are created for the data and a transition was created (see line A) to move all the boxes across the web page to the left-hand side. So far, it is a simple example of a multi-element transition with no surprises yet:

```
  .transition() // <- A
  .duration(duration)
    .style("left", "10px")
  .filter(function(d){return d == "Cat";}) // <- B
  .transition() // <- C
  .duration(duration)
    .style("left", "500px");
```

Then on line B, d3.selection.filter function is used to generate a subselection containing only the "cat" boxes. Remember, D3 transition is still a selection (transition-bound selection), therefore, the d3.selection.filter function works exactly the same way as on a regular selection. Once the subselection is generated by the filter function, we can apply a secondary transition (see line C) to this subselection alone. The filter function returns a transition-bound subselection; therefore, the second transition created on line C is actually generating a transition chain. It will only be triggered after the first transition reaches its completion. By using combinations of transition chaining and filtering we can generate some really interesting data-driven animations; it is a useful tool to have in any data visualizer's toolset.
Transition with Style

See also

- For recipes on D3 data-driven selection filtering, please see the Filtering with data recipe Chapter 3, Dealing with Data
- Read about API doc for the selection.filter function at https://github.com/mbostock/d3/wiki/Selections#wiki-filter

Listening to transitional events

Transition chaining gives you the ability to trigger secondary transitions after the initial transition reaches its completion state; however, sometimes you might need to trigger certain action other than a transition, or maybe do something else during the transition. This is what transition event listeners are designed for, they are the topic of this recipe.

Getting Ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter6/events.html

How to do it...

In this recipe, we will demonstrate how to display different captions on an animated div element based on its transition state. Obviously, this example can easily be extended to perform more meaningful tasks using the same technique:

```javascript
<script type="text/javascript">
var body = d3.select("body"), duration = 3000;

var div = body.append("div")
 .classed("box", true)
 .style("background-color", "steelblue")
 .style("color", "white")
 .text("waiting") // <-A
 .transition().duration(duration) // <-B
 .delay(1000) // <-C
 .each("start", function(){ // <-D
     console.log(arguments);
     d3.select(this).text(function (d, i) {
         return "transitioning";
     });
 });
</script>
```
Chapter 6

The preceding code produces the following visual output where a box appears with waiting label; it moves to the right with the label changed to transitioning and when it's done, it stops moving and changes its label to done:

How it works...

In this recipe, we constructed a single div element with a simple horizontal-movement transition, which, when initiated, also changes the label based on what transition state it is in. Let's first take a look at how we manage to display the waiting label:

```javascript
var div = body.append("div")
  .classed("box", true)
  .style("background-color", "steelblue")
  .style("color", "white")
  .text("waiting") // <-A
  .transition().duration(duration) // <-B
  .delay(1000) // <-C
```

The waiting label is set on line A before the transition is defined on line B, however, we also specified a delay for the transition thus showing the waiting label before the transition is initiated. Next, let's find out how we were able to display the transitioning label during the transition:

```javascript
.each("start", function(){ // <-D
  d3.select(this).text(function (d, i) {
    return "transitioning";
  });
});
```

Transition event handling
Transition with Style

This is achieved by calling the `each()` function and selecting its first parameter set as "start" event name with an event listener function passed in as the second parameter. The `this` reference of the event listener function points to the current selected element, hence, can be wrapped by D3 and further manipulated. The transition "end" event is handled in an identical manner:

```javascript
.each("end", function(){ // <-E
    d3.select(this).text(function (d, i) {
        return "done";
    });
});
```

The only difference here is that the event name is passed into the `each()` function.

Implementing a custom interpolator

In Chapter 4, *Tipping the Scales*, we explored how custom interpolators can be implemented in D3. In this recipe, we will demonstrate how this technique can be combined with D3 transition to generate special transition effects by leveraging custom interpolation.

Getting Ready

Open your local copy of the following file in your web browser:


This recipe builds on top of what we have discussed in the *Implementing a custom interpolator* recipe in Chapter 4, *Tipping the Scales*. If you are not familiar with the concept of custom interpolation, please review the related recipe before proceeding with this one.

How to do it...

Let's look at the code of the `custom-interpolator-transition.html` file and see how it works:

```javascript
<script type="text/javascript">
    d3.interpolators.push(function(a, b) { // <-A
        var re = /^([a-z])$/i, ma, mb;
        if ((ma = re.exec(a)) && (mb = re.exec(b))) {
            a = a.charCodeAt(0);
            var delta = a - b.charCodeAt(0);
            return function(t) {
                return String.fromCharCode(Math.ceil(a - delta * t));
            };
        }
    });
</script>
```
The preceding code generates one ticking box that starts from a and finishes at z:

How it works...

First thing we did in this recipe is register a custom interpolator that is identical to the alphabet interpolator we discussed in Chapter 4, Tipping the Scales:

```javascript
var body = d3.select("body");

var countdown = body.append("div").append("input");

countdown.attr("type", "button")
 .attr("class", "countdown")
 .attr("value", "a") // <-B
 .transition().ease("linear") // <-C
 .duration(4000).delay(300)
 .attr("value", "z"); // <-D

</script>

// -A
var re = /^([a-z])$/; ma, mb;
if (ma = re.exec(a) && mb = re.exec(b)) {
    a = a.charCodeAt(0);
    var delta = a - b.charCodeAt(0);
    return function(t) {
        return String.fromCharCode(Math.ceil(a - delta * t));
    };
}
```
Once the custom interpolator is registered, the transition part has pretty much no custom logic at all. Since it's based on the value that needs to be interpolated and transitioned upon, D3 will automatically pick the correct interpolator to perform the task:

```javascript
countdown.attr("type", "button")
  .attr("class", "countdown")
  .attr("value", "a") // <-B
  .transition().ease("linear") // <-C
  .duration(4000).delay(300)
  .attr("value", "z"); // <-D
```

As we can see in the preceding code snippet, the start value is "a", defined on line B. Afterwards, a standard D3 transition is created on line C and finally all we had to do is set the end value to "z" on line D, then D3 and our custom interpolator takes care of the rest.

**Working with timer**

So far in this chapter we have discussed various topics on D3 transition. At this point you might be asking the question, *What is powering D3 transition that is generating the animated frames?*

In this recipe, we will explore a low-level D3 timer function that you can leverage to create your own custom animation from scratch.

**Getting Ready**

Open your local copy of the following file in your web browser:


**How to do it...**

In this recipe, we will create a custom animation that does not rely on D3 transition or interpolation at all; essentially a custom animation created from scratch. Let's look at the following code:

```javascript
<script type="text/javascript">
  var body = d3.select("body");

  var countdown = body.append("div").append("input");

  countdown.attr("type", "button")
    .attr("class", "countdown")
    .attr("value", "0");
</script>
```
The preceding code generates a box where a timer is set to 0, and by clicking on **Start** the
timer increases until it reaches **100** and stops, as shown in the following:

![Custom timer-based animation](image)

**How it works...**

In this example, we have constructed a custom animation that moves integer from 0 to 100.
For such a simple animation, of course we could have accomplished it using D3 transition and
tweening. However, a simple example like this avoids any distraction from the technique itself.
Additionally, even in this simple example, the timer-based solution is arguably simpler and
more flexible than a typical transition-based solution. The power house of this animation lies
in the `countup` function (see line A):

```javascript
function countup(target){ // <-A
    d3.timer(function(){ // <-B
        var value = countdown.attr("value");
        if(value == target) return true;  // <-C
        countdown.attr("value", ++value); // <-D
    });
}
```

```javascript
function reset(){
    countdown.attr("value", 0);
}
</script>

```html
<div class="control-group">
    <button onclick="countup(100)">
        Start
    </button>
    <button onclick="reset()">
        Clear
    </button>
</div>
```
var value = countdown.attr("value");
if(value == target) return true; // <-C
    countdown.attr("value", ++value); // <-D
});
}

As we have shown in this example, the key to understanding this recipe lies in the
d3.timer function.

This d3.timer(function, [delay], [mark]) starts a custom timer function and
invokes the given function repeatedly, until the function returns true. There is no way to
stop the timer once it is started, so the programmer must make sure the function eventually
returns true. Optionally, you can also specify a delay as well as a mark. The delay starts
from the mark and when the mark is not specified, Date.now will be used as the mark. The
following illustration shows the temporal relationship we discussed here:

In our implementation, the custom timer function increases button caption by one, every
time it is called (see line D) and returns true when the value reaches 100, and therefore the
timer is terminated (see line C).

Internally, D3 transition uses the same timer function to generate its animation. At this point,
you might be asking what is the difference between using d3.timer and using animation
frame directly. The answer is that the d3.timer actually uses animation frame if the browser
supports it, otherwise, it is smart enough to fall back to use the setTimeout function, thus
freeing you from worrying about browser’s support.

See also

- For more information on d3.timer, please visit its API at the following link:
  https://github.com/mbostock/d3/wiki/Transitions#wiki-d3_timer
Getting into Shape

In this chapter we will cover:

- Creating simple shapes
- Using a line generator
- Using line interpolation
- Changing line tension
- Using an area generator
- Using area interpolation
- Using an arc generator
- Implementing arc transition

Introduction

Scalable Vector Graphics (SVG) is a mature World Wide Web Consortium (W3C) standard designed for user interactive graphics on the Web and Mobile platform. Similar to HTML, SVG can coexist happily with other technologies like CSS and JavaScript in modern browsers forming the backbone of many web applications. In today's Web, you can see use cases of SVG everywhere from digital map to data visualization. So far in this book we have covered most of the recipes using HTML elements alone, however, in real-world projects, SVG is the de facto standard for data visualization; it is also where D3's strength really shines. In this chapter, we will cover the basic concept of SVG as well as D3's support for SVG shape generation. SVG is a very rich topic. Volumes of books can be and have been devoted to this topic alone, hence, we are not planning or even going to try to cover all SVG-related topics, rather we'll focus on D3 and data visualization related techniques and features.
What is SVG?

As its name suggests, SVG is about graphics. It is a way to describe graphical image with scalable vectors. Let’s see two of the main SVG advantages:

Vector

SVG image is based on vectors instead of pixels. With the pixel-based approach, an image is composed of a bitmap with x and y as its coordinates filled with color pigmentations. While with the vector-based approach, each image consists of a set of geometric shapes described using simple and relative formulae filled with certain texture. As you can imagine, this later approach fits naturally with data visualization requirement. It is much simpler to visualize your data with lines, bar, and circles in SVG versus trying to manipulate color pigmentations in a bitmap.

Scalability

The second signature capability of SVG is scalability. Since SVG graphic is a group of geometric shapes described using relative formulas, it can be rendered and re-rendered with different sizes and zoom levels without losing precision. On the other hand, when bitmap-based images are resized to a large resolution, they suffer the effect of pixelation, which occurs when the individual pixels become visible, while SVG does not have this drawback. See the following figure to get a better picture of what we just read:

As a data visualizer, using SVG gives you the benefit of being able to display your visualization on any resolution without losing the crispiness of your eye-catching creation. On top of that, SVG offers you some additional advantages such as:

- **Readability**: SVG is based on XML, a human-readable markup language
- **Open standard**: SVG was created by W3C and is not a proprietary vendor standard
- **Adoption**: All modern browsers support SVG standard, even on mobile platform
- **Interoperability**: SVG works well with other web technologies, such as CSS and JavaScript; D3 itself is a perfect demonstration of this capability
- **Lightweight**: Compared to bitmap-based images, SVG is a lot lighter, taking up much less space
Because of all these capabilities we have mentioned so far, SVG has become the de facto standard for data visualization on the Web. From this chapter onwards, all recipes in this book will be illustrated using SVG as its most important part, with which the true power of D3 can be professed.

Some older browsers do not support SVG natively. If your target users are using legacy browsers, please check SVG compatibility before deciding whether SVG is the right choice for your visualization project. Here is a link you can visit to check you browser’s compatibility:

http://caniuse.com/svg

Creating simple shapes

In this recipe, we will explore a few simple built-in SVG shape formulae and their attributes. These simple shapes are quite easy to generate and are usually created manually using D3 when necessary. Though these simple shapes are not the most useful shape generator to know when working with D3, occasionally they could be handy when drawing peripheral shapes in your visualization project.

Getting Ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter7/simple-shapes.html

How to do it...

In this recipe, we will draw four different shapes in four different colors using native SVG shape elements:

```javascript
<script type="text/javascript">
  var width = 600,
      height = 500;

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

  svg.attr("height", height)
      .attr("width", width);

  svg.append("line") // <--A
    .attr("x1", 0)
    .attr("y1", 200)
```
The preceding code generates the following visual output:

Simple SVG shapes

How it works...

We have drawn four different shapes: a line, a circle, a rectangle, and a triangle in this example using SVG built-in shape elements.

A little refresher on SVG coordinate system

SVG x and y coordinate system originates from the top-left corner (0, 0) of the canvas and ends on the lower-right corner (<width>, <height>).

- line: A line element creates a simple straight line with coordinate attributes x1 and y1 as its start point and x2, y2 as its end point (see line A).
- circle: The append() function draws a circle with coordinate attributes cx and cy defining the center of the circle while the attribute r defines the radius of the circle (see line B).
rect: The `append()` function draws a rectangle with coordinate attributes `x` and `y` defining the top-left corner of the rectangular (see line C), attributes `width` and `height` for controlling the size of the rectangle, and the attributes `rx` and `ry` can be used to introduce rounded corners. The attributes `rx` and `ry` control the x- and y-axis radius of the ellipse used to round off the corners of the rectangle (see line E).

polygon: To draw a polygon, a set of points that makes up the polygon need to be defined using a `points` attribute (see line F). The `points` attribute accepts a list of point coordinates separated by space:

```javascript
svg.append("polygon")
  .attr("points", "450,200 500,100 550,200"); // <-F
```

All SVG shapes can be styled using style attributes directly or through CSS similar to HTML elements. Furthermore, they can be transformed and filtered using SVG transformation and filter support, however, due to limited scope in this book, we will not cover these topics in detail. In the rest of this chapter, we will focus on D3-specific supports on SVG shape generation instead.

**There's more...**

SVG also supports `ellipse` and `polyline` elements, however, due to their similarity to `circle` and `polygon` we will not cover them in detail in this book. For more information on SVG shape elements, please visit [http://www.w3.org/TR/SVG/shapes.html](http://www.w3.org/TR/SVG/shapes.html).

**D3 SVG shape generators**

The "swiss army knife" among SVG shape elements is `svg:path`. A path defines the outline of any shape which can then be filled, stroked, or clipped. Every shape we have discussed so far can be mathematically defined using `svg:path` alone. SVG `path` is a very powerful construct and has its own mini-language and grammar. The `svg:path` mini-language is used to set the "d" attribute on an `svg:path` element, which consists of the following commands:

- **moveto**: Command M(absolute)/m(relative) moveto (x y)+
- **closepath**: Z(absolute)/z(relative) closepath
- **lineto**: L(absolute)/l(relative) lineto (x y)+, H(absolute)/h(relative) horizontal lineto x+, V(absolute)/v(relative) vertical lineto y+
- **Cubic Bézier**: C(absolute)/c(relative) curve to (x1 y1 x2 y2 x y)+, S(absolute)/s(relative) shorthand curve to (x2 y2 x y)+
- **Quadratic Bézier curve**: Q(absolute)/q(relative) quadratic Bézier curve to (x1 y1 x y)+, T(absolute)/t(relative) shorthand quadratic Bézier curve to (x y)+
- **Elliptical curve**: A(absolute)/a(relative) elliptical arc (rx ry x-axis-rotation large-arc-flag sweep-flag x y)+
As directly using paths is not a very pleasant method due to its cryptic language, therefore, in most cases, some kind of software, for example, Adobe Illustrator or Inkscape is required to assist us in creating the SVG path element visually. Similarly, D3 ships with a set of SVG shape generator functions that can be used to generate data-driven path formulae; this is how D3 truly revolutionizes the field of data visualization by combining the power of SVG with intuitive data-driven approach. This is also going to be the focus for the rest of this chapter.

See also

- Please refer to [http://www.w3.org/TR/SVG/Overview.html](http://www.w3.org/TR/SVG/Overview.html) for more information on SVG-related topics
- For a complete reference on SVG path formula language and its grammar please visit [http://www.w3.org/TR/SVG/paths.html](http://www.w3.org/TR/SVG/paths.html)

Using a line generator

D3 line generator is probably one of the most versatile generators. Though it is called a "line" generator, it has little to do with the `svg:line` element. In contrast, it is implemented using the `svg:path` element. Like `svg:path`, D3 line generator is so flexible that you can effectively draw any shape using `line` alone, however, to make your life easier, D3 also provides other more specialized shape generators, which will be covered in later recipes in this chapter. In this recipe, we will draw multiple data-driven lines using the `d3.svg.line` generator.

Getting Ready

Open your local copy of the following file in your web browser:


How to do it...

Now, let's see the line generator in action:

```javascript
<script type="text/javascript">
    var width = 500,
        height = 500,
        margin = 50,
        x = d3.scale.linear() // <-A
          .domain([0, 10])
```
151

```javascript
// Chapter 7

.range([margin, width - margin]),
.y = d3.scale.linear() // <-B
  .domain([0, 10])
  .range([height - margin, margin]);

var data = [ // <-C
    [{x: 0, y: 5}, {x: 1, y: 9}, {x: 2, y: 7},
     {x: 3, y: 5}, {x: 4, y: 3}, {x: 6, y: 4},
     {x: 7, y: 2}, {x: 8, y: 3}, {x: 9, y: 2}
    ],
    d3.range(10).map(function(i){
        return {x: i, y: Math.sin(i) + 5};
    })
];

var line = d3.svg.line() // <-D
  .x(function(d){return x(d.x);})
  .y(function(d){return y(d.y);});

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

svg.attr("height", height)
  .attr("width", width);

svg.selectAll("path.line")
  .data(data)
  .enter()
    .append("path") // <-E
      .attr("class", "line")
      .attr("d", function(d){return line(d);}); // <-F

// Axes related code omitted
...
</script>

```
Getting into Shape...  

The preceding code draws multiple lines along with the x and y axes:

![D3 line generator](image)

**How it works...**

In this recipe, the data we used to draw the lines are defined in a two-dimensional array:

```javascript
var data = [ // <-C
    [{x: 0, y: 5},{x: 1, y: 9},{x: 2, y: 7},
     {x: 3, y: 5},{x: 4, y: 3},{x: 6, y: 4},
     {x: 7, y: 2},{x: 8, y: 3},{x: 9, y: 2}]
];

d3.range(10).map(function(i){
    return {x: i, y: Math.sin(i) + 5};
})
];
```

The first data series is defined manually and explicitly, while the second series is generated using a mathematical formula. Both of these cases are quite common in data visualization projects. Once the data is defined, then in order to map data points to its visual representation, two scales were created for the x and y coordinates:

```javascript
x = d3.scale.linear() // <-A
    .domain([0, 10])
    .range([margin, width - margin]),
```
y = d3.scale.linear() // <-B
  .domain([0, 10])
  .range([height - margin, margin]);

Notice that the domains for these scales were set to be large enough to include all data points in both the series, while the range were set to represent the canvas area without including the margins. The y-axis range is inverted since we want our point of origin at the lower-left corner of the canvas instead of the SVG-standard upper-left corner. Once both data and scales are set, all we need to do is generate the lines to define our generator using the `d3.svg.line` function:

```javascript
var line = d3.svg.line() // <-D
  .x(function(d){return x(d.x);})
  .y(function(d){return y(d.y);});
```

The `d3.svg.line` function returns a D3 line generator function which you can further customize. In our example, we simply stated for this particular line generator the x coordinate, which will be calculated using the x scale mapping, while the y coordinate will be mapped by the y scale. Using D3 scales, to map coordinates, is not only convenient but also a widely accepted best practice (separation of concerns). Though, technically you can implement these functions using any approach you prefer. Now the only thing left to do is actually create the `svg:path` elements.

```javascript
svg.selectAll("path.line")
  .data(data)
  .enter()
    .append("path") // <-E
    .attr("class", "line")
    .attr("d", function(d){return line(d);}); // <-F
```

Path creation process was very straightforward. Two `svg:path` elements are created using the data array we defined (on line E). Then the `d` attribute for each path element was set using the line generator we created previously by passing in the data `d` as its input parameter. The following screenshot shows what the generated `svg:path` elements look like:

![Generated SVG path elements](image)

Finally two axes are created using the same x and y scales we defined earlier. Due to limited scope in this book, we have omitted the axes-related code in this recipe and in the rest of this chapter, since they don't really change and also are not the focus of this chapter.
Using line interpolation

By default, the D3 line generator uses linear interpolation mode, however, D3 supports a number of different line interpolation modes. Line interpolation determines how data points will be connected, for example, by a straight line (linear interpolation) or a curved line (cubic interpolation). In this recipe, we will show you how these interpolation modes can be set along with their effects.

Getting Ready

Open your local copy of the following file in your web browser:

https://github.com/NickQizhu/d3-cookbook/blob/master/src/chapter7/line-interpolation.html

This recipe is built on top of what we have done in the previous recipe, so, if you are not yet familiar with basic line generator functions, please review the previous recipe first before proceeding.

How to do it...

Now, let’s see how different line interpolation modes can be used:

```javascript
var width = 500,
    height = 500,
    margin = 30,
    x = d3.scale.linear()
        .domain([0, 10])
        .range([margin, width - margin]),
    y = d3.scale.linear()
        .domain([0, 10])
        .range([height - margin, margin]);

var data = [
    [{x: 0, y: 5},{x: 1, y: 9},{x: 2, y: 7},
     {x: 3, y: 5},{x: 4, y: 3},{x: 6, y: 4},
     {x: 7, y: 2},{x: 8, y: 3},{x: 9, y: 2}
    ],
    d3.range(10).map(function(i){
```
    return {x: i, y: Math.sin(i) + 5};
    });
};

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

svg.attr("height", height)
    .attr("width", width);

renderAxes(svg);

render("linear");

renderDots(svg);

function render(mode){
    var line = d3.svg.line()
        .interpolate(mode) // <-A
        .x(function(d){return x(d.x);})
        .y(function(d){return y(d.y);});

    svg.selectAll("path.line")
        .data(data)
        .enter()
        .append("path")
        .attr("class", "line");

    svg.selectAll("path.line")
        .data(data)
        .attr("d", function(d){return line(d);});
}

function renderDots(svg){ // <-B
    data.forEach(function(set){
        svg.append("g").selectAll("circle")
            .data(set)
            .enter().append("circle") // <-C
            .attr("class", "dot")
            .attr("cx", function(d) { return x(d.x); })
            .attr("cy", function(d) { return y(d.y); })
            .attr("r", 4.5);
    });
}

// Axes related code omitted
The preceding code generates the following line chart in your browser with configurable interpolation modes:

```javascript
var line = d3.svg.line()
  .interpolate(mode) // <-A
  .x(function(d){return x(d.x);})
  .y(function(d){return y(d.y);});
```

Overall, this recipe is similar to the previous one. Two lines are generated using pre-defined data set. However, in this recipe, we allow the user to select a specific line interpolation mode, which is then set using the `interpolate` function on line generator (see line A).
The following interpolation modes are supported by D3:

- **linear**: Linear segments, that is, polyline
- **linear-closed**: Closed linear segments, that is, polygon
- **step-before**: Alternated between the vertical and horizontal segments, as in a step function
- **step-after**: Alternated between the horizontal and vertical segments, as in a step function
- **basis**: A B-spline, with control point duplication on the ends
- **basis-open**: An open B-spline; may not intersect the start or end
- **basis-closed**: A closed B-spline, as in a loop
- **bundle**: Equivalent to basis, except the tension parameter is used to straighten the spline
- **cardinal**: A Cardinal spline, with control point duplication on the ends.
- **cardinal-open**: An open Cardinal spline; may not intersect the start or end, but will intersect other control points
- **cardinal-closed**: A closed Cardinal spline, as in a loop
- **monotone**: Cubic interpolation that preserves monotonicity in $y$

Additionally, in the `renderDots` function (see line B) we have also created a small circle for each data point to serve as reference points. These dots are created using `svg:circle` elements, as shown on line C:

```javascript
function renderDots(svg) { // <-B
  data.forEach(function(set){
    svg.append("g").selectAll("circle")
      .data(set)
      .enter().append("circle") // <-C
      .attr("class", "dot")
      .attr("cx", function(d) { return x(d.x); })
      .attr("cy", function(d) { return y(d.y); })
      .attr("r", 4.5);
  });
}
```

### Changing line tension

If Cardinal interpolation mode (cardinal, cardinal-open, cardinal-closed) is used, then line can be further modified using `tension` settings. In this recipe, we will see how tension can be modified and its effect on line interpolation.
Getting into Shape

Getting Ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter7/line-tension.html

How to do it...

Now, let's see how line tension can be changed and what effect it has on line generation:

```javascript
<script type="text/javascript">
var width = 500,
    height = 500,
    margin = 30,
    duration = 500,
    x = d3.scale.linear()
        .domain([0, 10])
        .range([margin, width - margin]),
    y = d3.scale.linear()
        .domain([0, 1])
        .range([height - margin, margin]);

var data = d3.range(10).map(function(i){
    return {x: i, y: (Math.sin(i * 3) + 1) / 2};
});

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

svg.attr("height", height)
    .attr("width", width);

renderAxes(svg);

render([1]);

function render(tension){
    var line = d3.svg.line()
        .interpolate("cardinal")
```
Chapter 7

```javascript
.x(function(d){return x(d.x);})
.y(function(d){return y(d.y);});

svg.selectAll("path.line")
  .data(tension)
  .enter()
  .append("path")
  .attr("class", "line");

svg.selectAll("path.line")
  .data(tension) // <-A
  .transition().duration(duration).ease("linear") // <-B
  .attr("d", function(d){
    return line.tension(d)(data); // <-C
  });

svg.selectAll("circle")
  .data(data)
  .enter().append("circle")
  .attr("class", "dot")
  .attr("cx", function(d) { return x(d.x); })
  .attr("cy", function(d) { return y(d.y); })
  .attr("r", 4.5);

// Axes related code omitted
...
</script>
<h4>Line Tension:</h4>
<div class="control-group">
  <button onclick="render([0])">0</button>
  <button onclick="render([0.2])">0.2</button>
  <button onclick="render([0.4])">0.4</button>
  <button onclick="render([0.6])">0.6</button>
  <button onclick="render([0.8])">0.8</button>
  <button onclick="render([1])">1</button>
</div>
```
The preceding code generates a Cardinal line chart with configurable tension:

```javascript
svg.selectAll("path.line")
  .data(tension) // <-A
  .transition().duration(duration).ease("linear") // <-B
  .attr("d", function(d)
    return line.tension(d)(data);) // <-C
);```

**How it works...**

Tension sets the Cardinal spline interpolation tension to a specific number in the range of \([0, 1]\). Tension can be set using the `tension` function on line generator (see line C):

```javascript
svg.selectAll("path.line")
  .data(tension) // <-A
  .transition().duration(duration).ease("linear") // <-B
  .attr("d", function(d)
    return line.tension(d)(data);) // <-C
);```
Additionally, we also initiated a transition on line 2 to highlight the tension effect on line interpolation. If the tension is not set explicitly, Cardinal interpolation sets tension to 0.7 by default.

## Using an area generator

Using D3 line generator, we can technically generate an outline of any shape, however, even with different interpolation-support, directly drawing an area using line (as in an area chart) is not an easy task. This is why D3 also provides a separate shape generator function specifically designed for drawing area.

### Getting Ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter7/area.html

### How to do it...

In this recipe, we will add a filled area to a pseudo line chart effectively turning it into an area chart:

```javascript
<script type="text/javascript">
  var width = 500,
      height = 500,
      margin = 30,
      duration = 500,
      x = d3.scale.linear() // <-A
        .domain([0, 10])
        .range([margin, width - margin]),
      y = d3.scale.linear()
        .domain([0, 10])
        .range([height - margin, margin]);

  var data = d3.range(11).map(function(i){ // <-B
    return {x: i, y: Math.sin(i)*3 + 5};
  });

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

  svg.attr("height", height)
</script>
Getting into Shape

```javascript
.renderAxes(svg);
.render("linear");
.renderDots(svg);

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

    svg.selectAll("path.line")
        .data([data])
        .enter()
        .append("path")
        .attr("class", "line");

    svg.selectAll("path.line")
        .data([data])
        .attr("d", function(d){return line(d);});

    var area = d3.svg.area() // <-C
        .x(function(d) { return x(d.x); }) // <-D
        .y0(y(0)) // <-E
        .y1(function(d) { return y(d.y); }); // <-F

    svg.selectAll("path.area") // <-G
        .data([data])
        .enter()
        .append("path")
        .attr("class", "area")
        .attr("d", function(d){return area(d);}); // <-H
}

// Dots rendering code omitted

// Axes related code omitted
...
</script>
```
The preceding code generates the following visual output:

```
x = d3.scale.linear() // <-A
    .domain([0, 10])
    .range([margin, width - margin]),
y = d3.scale.linear()
    .domain([0, 10])
    .range([height - margin, margin]);

var data = d3.range(11).map(function(i){ // <-B
    return {x: i, y: Math.sin(i)*3 + 5};
});
```

How it works...

Similar to the Using a line generator recipe earlier in this chapter, we have two scales defined to map data to visual domain on x and y coordinates (see line A), in this recipe:
On line B, data is generated by a mathematical formula. Area generator is then created using the `d3.svg.area` function (see line C):

```javascript
var area = d3.svg.area() // <-C
    .x(function(d) { return x(d.x); }) // <-D
    .y0(y(0)) // <-E
    .y1(function(d) { return y(d.y); }); // <-F
```

As you can see, D3 area generator is—similar to the line generator—designed to work in a 2D homogenous coordinate system. With the `x` function defining an accessor function for `x` coordinate (see line D), which simply maps data to the visual coordinate using the `x` scale we defined earlier. For the `y` coordinate, we provided the area generator two different accessors; one for the lower bound (`y0`) and the other for the higher bound (`y1`) coordinates. This is the crucial difference between area and line generator. D3 area generator supports higher and lower bound on both `x` and `y` axes (`x0`, `x1`, `y0`, `y1`), and the shorthand accessors (`x` and `y`) if the higher and lower bounds are the same. Once the area generator is defined, the method of creating an area is almost identical to the line generator.

```javascript
svg.selectAll("path.area") // <-G
    .data([data])
    .enter()
    .append("path")
    .attr("class", "area")
    .attr("d", function(d){return area(d);}); // <-H
```

Area is also implemented using the `svg:path` element (see line G). D3 area generator is used to generate the "d" formula for the `svg:path` element on line H with data "d" as its input parameter.

### Using area interpolation

Similar to the D3 line generator, area generator also supports identical interpolation mode, hence, it can be used in combination with the line generator in every mode.

### Getting Ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter7/area-interpolation.html
In this recipe, we will show how interpolation mode can be configured on an area generator. This way matching interpolated area can then be created with corresponding line:

```javascript
var width = 500,
    height = 500,
    margin = 30,
    x = d3.scale.linear()
      .domain([0, 10])
      .range([margin, width - margin]),
    y = d3.scale.linear()
      .domain([0, 10])
      .range([height - margin, margin]);

var data = d3.range(11).map(function(i){
    return {x: i, y: Math.sin(i)*3 + 5};
});

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

svg.attr("height", height)
  .attr("width", width);

renderAxes(svg);

render("linear");

renderDots(svg);

function render(mode){
    var line = d3.svg.line()
      .interpolate(mode) // <-A
      .x(function(d){return x(d.x);})
      .y(function(d){return y(d.y);});

    svg.selectAll("path.line")
      .data([data])
      .enter()
      .append("path")
      .attr("class", "line");

    svg.selectAll("path.line")
      .data([data])
      .attr("d", function(d){return line(d);});

    var area = d3.svg.area()
```
The preceding code generates a pseudo area chart with configurable interpolation mode:
How it works...

This recipe is similar to the previous one except that in this recipe the interpolation mode is passed in based on the user's selection:

```javascript
var line = d3.svg.line()
  .interpolate(mode) // <-A
  .x(function(d){return x(d.x);})
  .y(function(d){return y(d.y);});

var area = d3.svg.area()
  .interpolate(mode) // <-B
  .x(function(d) { return x(d.x); })
  .y0(y(0))
  .y1(function(d) { return y(d.y); });
```

As you can see, the interpolation mode is configured on both lines along with the area generator through the `interpolate` function (see lines A and B). Since D3 line and area generator supports the same set of interpolation mode, they can always be counted on to generate matching line and area as seen in this recipe.

There's more...

D3 area generator also supports the same tension configuration when interpolated using Cardinal mode, however, since it is identical to line generator's tension support, and due to limited scope in this book we will not cover area tension here.

See also

- Please refer to [https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-area](https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-area) for more information on area generator functions

Using an arc generator

Among the most common shape generators—besides the line and area generator—D3 also provides the arc generator. At this point, you might be wondering, *Didn't SVG standard already include circle element? Isn't that enough?*

The simple answer to this is "no". The D3 arc generator is a lot more versatile than the simple `svg:circle` element. The D3 arc generator is capable of creating not only circles but also annulus (donut-like), circular sector, and annulus sector, all of which we will learn in this recipe. More importantly, an arc generator is designed to generate, as its name suggests, an arc (in others words, not a full circle or even a sector but rather arcs of arbitrary angle).
Getting into Shape

Getting Ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter7/arc.html

How to do it...

In this recipe we will use arc generator to generate multi-slice circle, annulus (donut), circular sectors, and annulus sectors.

```javascript
var width = 400,
height = 400,
// angles are in radians
fullAngle = 2 * Math.PI, // <-A
colors = d3.scale.category20c();

var svg = d3.select("body").append("svg")
  .attr("class", "pie")
  .attr("height", height)
  .attr("width", width);

function render(innerRadius, endAngle){
  if(!endAngle) endAngle = fullAngle;

  var data = [ // <-B
    {startAngle: 0, endAngle: 0.1 * endAngle},
    {startAngle: 0.1 * endAngle, endAngle: 0.2 * endAngle},
    {startAngle: 0.2 * endAngle, endAngle: 0.4 * endAngle},
    {startAngle: 0.4 * endAngle, endAngle: 0.6 * endAngle},
    {startAngle: 0.6 * endAngle, endAngle: 0.7 * endAngle},
    {startAngle: 0.7 * endAngle, endAngle: 0.9 * endAngle},
    {startAngle: 0.9 * endAngle, endAngle: endAngle}
  ];

  var arc = d3.svg.arc().outerRadius(200) // <-C
    .innerRadius(innerRadius);

  svg.select("g").remove();

  svg.append("g")
    .attr("transform", "translate(200,200)")
```
Chapter 7

```javascript
.selectAll("path.arc")
.data(data)
.enter()
.append("path")
.attr("class", "arc")
.attr("fill", function(d, i){return colors(i);})
.attr("d", function(d, i){
  return arc(d, i); // <-D
});
render(0);
</script>

<div class="control-group">
  <button onclick="render(0)">Circle</button>
  <button onclick="render(100)">Annulus(Donut)</button>
  <button onclick="render(0, Math.PI)">Circular Sector</button>
  <button onclick="render(100, Math.PI)">Annulus Sector</button>
</div>

The preceding code produces the following circle, which you can change into an arc, a sector, or an arc sector by clicking on the buttons, for example, **Annulus(Donut)** generates the second shape:

![Arc generator](image)

**How it works...**

The most important part of understanding the D3 arc generator is its data structure. D3 arc generator has very specific requirements when it comes to its data, as shown on line B:

```javascript
var data = [ // <-B
  {startAngle: 0, endAngle: 0.1 * endAngle},
  {startAngle: 0.1 * endAngle, endAngle: 0.2 * endAngle},
```
Getting into Shape

{startAngle: 0.2 * endAngle, endAngle: 0.4 * endAngle},
{startAngle: 0.4 * endAngle, endAngle: 0.6 * endAngle},
{startAngle: 0.6 * endAngle, endAngle: 0.7 * endAngle},
{startAngle: 0.7 * endAngle, endAngle: 0.9 * endAngle},
{startAngle: 0.9 * endAngle, endAngle: endAngle}
];

Each row of the arc data has to contain two mandatory fields, startAngle and endAngle. The angles have to be in the range \[0, 2 \times \text{Math.PI}\] (see line A). D3 arc generator will use these angles to generate corresponding slices, as shown earlier in this recipe.

Along with the start and end angles, arc data set can contain any number of additional fields, which can then be accessed in D3 functions to drive other visual representation.

If you are thinking that calculating these angles based on the data you have is going to be a big hassle, you are absolutely correct. This is why D3 provides specific layout manager to help you calculate these angles, and which we will cover in the next chapter. For now, let's focus on understanding the basic mechanism behind the scenes so that when it is time to introduce the layout manager or if you ever need to set the angles manually, you will be well-equipped to do so. D3 arc generator is created using the \texttt{d3.svg.arc} function:

\texttt{var arc = d3.svg.arc().outerRadius(200) // <-C}
\texttt{.innerRadius(innerRadius);}\

The \texttt{d3.svg.arc} function optionally has \texttt{outerRadius} and \texttt{innerRadius} settings. When \texttt{innerRadius} is set, the arc generator will produce an image of annulus (donut) instead of a circle. Finally, the D3 arc is also implemented using the \texttt{svg:path} element, and thus similar to the line and area generator, \texttt{d3.svg.arc} generator function can be invoked (see line D) to generate the \texttt{d} formula for the \texttt{svg:path} element:

\texttt{svg.append("g")}
\texttt{.attr("transform", "translate(200,200)")}
\texttt{.selectAll("path.arc")}
\texttt{.data(data)
.enter()
.append("path")
 .attr("class", "arc")
 .attr("fill", function(d, i){return colors(i);})
 .attr("d", function(d, i){
     return arc(d, i); // <-D
 };
});
One additional element worth mentioning here is the `svg:g` element. This element does not define any shape itself, but serves rather as a container element used to group other elements, in this case, the `path.arc` elements. Transformation applied to the `g` element is applied to all the child elements while its attributes are also inherited by its child elements.

## Implementing arc transition

One area where arc differs significantly from other shapes, such as line and area, is its transition. For most of the shapes we covered so far, including simple SVG built-in shapes, you can rely on D3 transition and interpolation to handle their animation. However, this is not the case when dealing with arc. We will explore the arc transition technique in this recipe.

### Getting Ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter7/arc-transition.html

### How to do it...

In this recipe, we will animate a multi-slice annulus transitioning each slice starting from angle 0 to its final desired angle and eventually reaching a full annulus:

```javascript
var width = 400,
    height = 400,
    endAngle = 2 * Math.PI,
    colors = d3.scale.category20c();

var svg = d3.select("body").append("svg")
  .attr("class", "pie")
  .attr("height", height)
  .attr("width", width);

function render(innerRadius) {
  var data = [
    {startAngle: 0, endAngle: 0.1 * endAngle},
    {startAngle: 0.1 * endAngle, endAngle: 0.2 * endAngle},
    {startAngle: 0.2 * endAngle, endAngle: 0.4 * endAngle},
    {startAngle: 0.4 * endAngle, endAngle: 0.6 * endAngle},
    {startAngle: 0.6 * endAngle, endAngle: 0.7 * endAngle},
    {startAngle: 0.7 * endAngle, endAngle: 0.9 * endAngle},
    ];
```
The preceding code generates an arc which starts rotating and eventually becomes a complete annulus:

Arc transition with tweening
How it works...

When confronted with the requirement of such transition, your first thought might be using the vanilla D3 transition while relying on built-in interpolations to generate the animation. Here is the code snippet which will do just that:

```javascript
svg.append("g")
  .attr("transform", "translate(200,200)"

  .selectAll("path.arc")
    .data(data)
    .enter()
      .append("path")
        .attr("class", "arc")
        .attr("fill", function (d, i) {
            return colors(i);
        })
        .attr("d", function(d){
            return arc({startAngle: 0, endAngle: 0});
        })
        .transition().duration(1000).ease("linear")
        .attr("d", function(d){return arc(d);});
```

As shown with highlighted lines in the preceding code snippet, with this approach we initially created slice path with both `startAngle` and `endAngle` set to zero. Then, through transition we interpolated the path "d" attribute to its final angle using the arc generator function `arc(d)`. This approach seems to make sense, however, what it generates is the transition shown in the following:

![Arc transition without tweening](image)

This is obviously not the animation we want. The reason for this strange transition is that by directly creating a transition on the `svg:path` attribute "d", we are instructing D3 to interpolate this string:

```
d="M1.2246063538223773e-14,-200A200,200 0 0,1 1.2246063538223773e-14,-200L6.123031769111886e-15,-100A100,100 0 0,0 6.123031769111886e-15,-100Z"
```
Getting into Shape

To this string linearly:

\[
d = "M1.2246063538223773e-14,-200A200,200 0 0,1 117.55705045849463,-161.80339887498948L58.778525229247315,-80.90169943749474A100,100 0 0,6.123031769111886e-15,-100Z"
\]

Hence, this particular transition effect.

![Though this transition effect is not what we desire in this example, this is still a good showcase of how flexible and powerful built-in D3 transition is.](image)

In order to achieve the transition effect we want, we need to leverage the D3 attribute tweening (for detailed description on tweening, see the Using tweening recipe of Chapter 6, Transition with Style):

```javascript
svg.append("g")
  .attr("transform", "translate(200,200)")
  .selectAll("path.arc")
  .data(data)
  .enter()
  .append("path")
  .attr("class", "arc")
  .attr("fill", function (d, i) {
    return colors(i);
  })
  .transition().duration(1000)
  .attrTween("d", function (d) { // <-A
    var start = {startAngle: 0, endAngle: 0}; // <-B
    var interpolate = d3.interpolate(start, d); // <-C
    return function (t) {
      return arc(interpolate(t)); // <-D
    };}
  });
```

Here, instead of transitioning the `svg:path` attribute "d" directly, we created a tweening function on line A. As you can recall, D3 `attrTween` expects a factory function for a tween function. In this case, we start our tweening from angle zero (see line B). Then we create a compound object interpolator on line C, which will interpolate both start and end angles for each slice. Finally on line D, the arc generator is used to generate a proper `svg:path` formula using already interpolated angles. This is how a smooth transition of properly-angled arcs can be created through custom attribute tweening.
There's more...

D3 also provides support for other shape generators, for example, symbol, chord, and diagonal. However, due to their simplicity and the limited scope in this book we will not cover them individually here, although we will cover them as parts of other more complex visual constructs in the following chapters. More importantly, with well-grounded understanding of these shape generators that we introduced in this chapter, you should be able to pick up other D3 shape generators without much trouble.

See also

- For more information on transition and tweening, refer to the Using tweening recipe in Chapter 6, Transition with Style
In this chapter we will cover:

- Creating a line chart
- Creating an area chart
- Creating a scatter plot chart
- Creating a bubble chart
- Creating a bar chart

**Introduction**

In this chapter, we will turn our attention to one of the oldest and well trusted companions in data visualization—charts. Charts are a well defined and well understood graphical representation of data; the following definition just confirms it:

*(In charts) the data is represented by symbols, such as bars in a bar chart, lines in a line chart, or slices in a pie chart.*

*Jensen C. & Anderson L. (1991)*

When charts are used in data visualization, their well understood graphical semantics and syntax relieve the audience of your visualization from the burden of learning the meaning of the graphical metaphor. Hence they can focus on the data itself and the information generated through visualization. The goal of this chapter is not only to introduce some of the commonly used chart types but also demonstrate how the various topics and techniques we have learned so far can be combined and leveraged in producing sleek interactive charts using D3.
Recipes in this chapter are much longer than the recipes we have encountered so far since they are designed to implement fully functional reusable charts. I have tried to break it into different segments and with consistent chart structures to ease your reading experience. However, it is still highly recommended to open the companion code examples in your browser and your text editor while going through this chapter to minimize potential confusion and maximize the benefit.

**D3 chart convention:** Before we dive into creating our first reusable chart in D3, we need to cover some charting conventions commonly accepted in the D3 community otherwise we might risk creating charting libraries that confuse our user instead of helping them.

As you would have imagined, D3 charts are most commonly implemented using SVG instead of HTML; however, the convention we discuss here would also apply to HTML-based charts albeit the implementation detail will be somewhat different.

Let's first take a look at the following diagram:

As shown in this diagram the point of origin (0, 0) in an SVG image is at its top-leftmost corner as expected, however, the most important aspect of this convention pertains to how chart margins are defined and furthermore where the axes are placed.
Margins: First of all, let us see the most important aspect of this convention—the margins. As we can see for each chart there are four different margin settings: left, right, top, and bottom margins. A flexible chart implementation should allow its user to set different values for each of these margins and we will see in later recipes how this can be achieved.

Coordinate translation: Secondly, this convention also suggests that the coordinate reference of the chart body (grey area) should be defined using a SVG translate transformation \( \text{translate}(\text{margin.left, margin.top}) \). This translation effectively moves the chart body area to the desired point, and one additional benefit of this approach is, by shifting the frame of reference for chart body coordinates, it simplifies the job of creating sub-elements inside the chart body since the margin size becomes irrelevant. For any sub-element inside the chart body, its point of origin \((0, 0)\) is now the top-leftmost corner of the chart body area.

Axes: Lastly, the final aspect of this convention is regarding how and where chart axes are placed. As shown by this diagram chart axes are placed inside chart margins instead of being part of the chart body. This approach has the advantage of treating axes as peripheral elements in a chart, hence not convoluting the chart body implementation and additionally making axes rendering logic chart-independent and easily reusable.

Now let's create our first reusable D3 chart with all the knowledge and techniques we learned so far.

## Creating a line chart

Line chart is a common, basic chart type that is widely used in many fields. This chart consists of a series of data points connected by straight line segments. A line chart is also typically bordered by two perpendicular axes: the x axis and y axis. In this recipe, we will see how this basic chart can be implemented using D3 as a reusable JavaScript object that can be configured to display multiple data series on a different scale. Along with that we will also show the technique of implementing a dynamic multi-data-series update with animation.

### Getting ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter8/line-chart.html

It is highly recommended to have the companion code example open while reading this recipe.
How to do it...

Let's take a look at the code that implements this chart type. Due the length of the recipe we will only show the outline of the code here while diving into the details in the following How it works... section.

```javascript
// First we define the chart object using a functional object
function lineChart() { // <-1A
...

    // main render function
    _chart.render = function () { // <-2A
        ...
    }

    // axes rendering function
    function renderAxes(svg) {
        ...
    }

    // function to render chart body
    function renderBody(svg) { // <-2D
        ...
    }

    // function to render lines
    function renderLines() {
        ...
    }

    // function to render data points
    function renderDots() {
    }

    return _chart; // <-1E
}
```

This recipe generates the following chart:
How it works...

As we can see, this recipe is significantly more involved than anything we have encountered so far, so now I will break it into multiple detailed sections with different focuses.

**Chart object and attributes:** First, we will take a look at how this chart object is created and how associated attributes can be retrieved and set on the chart object.

```javascript
function lineChart() { // <-1A
    var _chart = {};

    var _width = 600, _height = 300, // <-1B
        _margins = {top: 30, left: 30, right: 30, bottom: 30},
        _x, _y,
        _data = [],
        _colors = d3.scale.category10(),
        _svg,
        _bodyG,
        _line;

    // <-1C
    _chart.height = function (h) {
        if (!arguments.length) return _height;
        _height = h;
        return _chart;
    };
```
As we can see, the chart object is defined using a function called `lineChart` on line 1A following the functional object pattern we have discussed in the Understanding D3-Style JavaScript recipe in Chapter 1, Getting Started with D3.js. Leveraging the greater flexibility with information hiding offered by the functional object pattern, we have defined a series of internal attributes all named starting with an underscore (line 1B). Some of these attributes are made public by offering accessor function (line 1C). Publically accessible attributes are:

- `width`: Chart SVG total width in pixels
- `height`: Chart SVG total height in pixels
- `margins`: Chart margins
- `colors`: Chart ordinal color scale used to differentiate different data series
- `x`: x axis scale
- `y`: y axis scale
The accessor functions are implemented using the technique we introduced in Chapter 1, Getting Started with D3.js, effectively combining both getter and setter functions in one function, which behaves as a getter when no argument is given and a setter when an argument is present (line 1C). Additionally, both lineChart function and its accessors, return a chart instance thus allowing function chaining. Finally, the chart object also offers an addSeries function which simply pushes a data array (series) into its internal data storage array (_data), see line 1D.

**Chart body frame rendering:** After covering the basic chart object and its attributes, the next aspect of this reusable chart implementation is the chart body svg:g element rendering and its clip path generation.

```javascript
_chart.render = function () { // <-2A
  if (!_svg) {
    _svg = d3.select("body").append("svg") // <-2B
      .attr("height", _height)
      .attr("width", _width);

    renderAxes(_svg);

    defineBodyClip(_svg);
  }

  renderBody(_svg);
};
...
function defineBodyClip(svg) { // <-2C
  var padding = 5;

  svg.append("defs")
    .append("clipPath")
    .attr("id", "body-clip")
    .append("rect")
    .attr("x", 0 - padding)
    .attr("y", 0)
    .attr("width", quadrantWidth() + 2 * padding)
    .attr("height", quadrantHeight());
}

function renderBody(svg) { // <-2D
  if (!_bodyG)
    _bodyG = svg.append("g")
      .attr("class", "body")
      .attr("transform", "translate("
The `render` function defined on line 2A is responsible for creating the `svg:svg` element and setting its `width` and `height` (line 2B). After that, it creates an `svg:clipPath` element that covers the entire chart body area. The `svg:clipPath` element is used to restrict the region where paint can be applied. In our case we use it to restrict where the line and dots can be painted (only within the chart body area). This code generates the following SVG element structure that defines the chart body:

```xml
<svg height="360" width="568">
  <g class="axes"></g>
  <defs>
    <clipPath id="body-clip"></clipPath>
  </defs>
  <g class="body" transform="translate(30,30)" clip-path="url(#body-clip)"></g>
</svg>
```

For more information on clipping and masking please visit [http://www.w3.org/TR/SVG/masking.html](http://www.w3.org/TR/SVG/masking.html)

The `renderBody` function defined on line 2D generates the `svg:g` element which wraps all the chart body content with a translation set according to the chart margin convention we have discussed in the previous section (line 2E).

**Render axes**: Axes are rendered in the function `renderAxes` (line 3A).

```javascript
function renderAxes(svg) { // <-3A
  var axesG = svg.append("g")
    .attr("class", "axes");

  renderXAxis(axesG);

  renderYAxis(axesG);
}
```

As discussed in the previous chapter, both x and y axes are rendered inside the chart margin area. We are not going into details about axes rendering since we have discussed this topic in much detail in Chapter 5, *Playing with Axes*. 
Render data series: Everything we discussed so far in this recipe is not unique to this chart type alone but rather it is a shared framework among other Cartesian coordinates based chart types. Finally, now we will discuss how the line segments and dots are created for multiple data series. Let's take a look at the following code fragments that are responsible for data series rendering.

```javascript
function renderLines() {
    _line = d3.svg.line() // <-4A
        .x(function (d) { return _x(d.x); })
        .y(function (d) { return _y(d.y); });

    _bodyG.selectAll("path.line")
        .data(_data)
        .enter() // <-4B
        .append("path")
        .style("stroke", function (d, i) { return _colors(i); // <-4C
            return _colors(i); // <-4C
        })
        .attr("class", "line");

    _bodyG.selectAll("path.line")
        .data(_data)
        .transition() // <-4D
        .attr("d", function (d) { return _line(d); });
}

function renderDots() {
    _data.forEach(function (list, i) {
        _bodyG.selectAll("circle._" + i) // <-4E
            .data(list)
            .enter()
            .append("circle")
            .attr("class", "dot _" + i);

        _bodyG.selectAll("circle._" + i)
            .data(list)
            .style("stroke", function (d, i) { return _colors(i); // <-4F
                return _colors(i); // <-4F
            })
            .transition() // <-4G
            .attr("cx", function (d) { return _x(d.x); })
            .attr("cy", function (d) { return _y(d.y); })
            .attr("r", 4.5);
    });
}
```
The line segments and dots are generated using techniques we introduced in Chapter 7, *Getting into Shape*. The `d3.svg.line` generator was created on line 4A to create `svg:path` that maps the data series. The Enter-and-Update pattern is used to create the data line (line 4B). Line 4C sets a different color for each data line based on its index. Lastly, line 4E sets the transition in the update mode to move the data line smoothly on each update. The `renderDots` function performs a similar rendering logic that generates a set of `svg:circle` elements representing each data point (line 4E), coordinating its color based on the data series index (line 4F), and finally also initiates a transition on line 4G, so the dots can move with the line whenever the data is updated.

As illustrated by this recipe, creating a reusable chart component involves actually quite a bit of work. However, more than two-thirds of the code is required in creating peripheral graphical elements and accessors methods. Therefore in a real-world project you can extract this logic and reuse a large part of this implementation for other charts; though we did not do this in our recipes in order to reduce the complexity, so you can quickly grasp all aspects of chart rendering. Due to limited scope in this book, in later recipes we will omit all peripheral rendering logic while only focusing on the core logic related to each chart type.

## Creating an area chart

An area chart or an area graph is very similar to a line chart and largely implemented based on the line chart. The main difference between an area chart and a line chart is that in the area chart, the area between the axis and the line are filled with colors or textures. In this recipe we will explore techniques of implementing a type of area chart known as **Layered Area Chart**.

### Getting ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter8/area-chart.html

### How to do it...

Since an area chart implementation is largely based on the line chart implementation and it shares a lot of common graphical elements such as the axes and the clip path, therefore in this recipe we will only show the code concerning the area chart implementation specifics:

```javascript
function renderBody(svg) {
  if (!_bodyG)
    _bodyG = svg.append("g")
      .attr("class", "body")
```
function renderLines() {
  _line = d3.svg.line()
    .x(function (d) { return _x(d.x); })
    .y(function (d) { return _y(d.y); });

  _bodyG.selectAll("path.line")
    .data(_data)
    .enter()
    .append("path")
    .style("stroke", function (d, i) {
      return _colors(i);
    })
    .attr("class", "line");

  _bodyG.selectAll("path.line")
    .data(_data)
    .transition()
    .attr("d", function (d) { return _line(d); });
}

function renderDots() {
  _data.forEach(function (list, i) {
    _bodyG.selectAll("circle._" + i)
      .data(list)
      .enter().append("circle")
      .attr("class", "dot_" + i);

    _bodyG.selectAll("circle._" + i)
      .data(list)
      .style("stroke", function (d, i) {
        return _colors(i);
      })
  })
}
This recipe generates the following layered area chart:
Chapter 8

How it works...

As we mentioned before, since the area chart implementation is based on our line chart implementation, a large part of the implementation is identical to the line chart. In fact the area chart needs to render the exact line and dots implemented in the line chart. The crucial difference lies in the renderAreas function. In this recipe we rely on the area generation technique discussed in Chapter 7, Getting into Shape. The d3.svg.area generator was created on line A with its upper line created to match the line while its lower line (y0) fixed on x-axis.

```javascript
var area = d3.svg.area() // <-A
  .x(function(d) { return _x(d.x); })
  .y0(yStart())
  .y1(function(d) { return _y(d.y); });
```

Once the area generator is defined, a classic Enter-and-Update pattern is employed to create and update the areas. In the Enter case (line B), an svg:path element was created for each data series and colored using its series index so it will have matching color with our line and dots (line C).

```javascript
_bodyG.selectAll("path.area")
  .data(_data)
  .enter() // <-B
    .append("path")
    .style("fill", function (d, i) {
      return _colors(i); // <-C
    })
    .attr("class", "area");
```

Whenever the data is updated, as well as for newly created areas, we start a transition (line D) to update the area svg:path elements' d attribute to the desired shape (line E).

```javascript
_bodyG.selectAll("path.area")
  .data(_data)
  .transition() // <-D
    .attr("d", function (d) {
      return area(d); // <-E
    });
```

Since we know that the line chart implementation animates both line and dots when updated, therefore our area update transition here effectively allows the areas to be animated and moved in accordance with both lines and dots in our chart.
Finally, we also add the CSS style for `path.area` to decrease its opacity so areas become see-through; hence allowing the layered effect we desire.

```css
.area {
    stroke: none;
    fill-opacity: .2;
}
```

### Creating a scatter plot chart

A scatter plot or scatter graph is another common type of diagram used to display data points on Cartesian coordinates with two different variables. Scatter plot is especially useful when exploring the problem of clustering and classification. In this recipe, we will learn how to implement a multi-series scatter plot chart in D3.

### Getting ready

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter8/scatterplot-chart.html

### How to do it...

A scatter plot is another chart which uses Cartesian coordinates. Thus, a large part of its implementation is very similar to the charts we have introduced so far, therefore the code concerning peripheral graphical elements are again omitted to save space in this book. Please review the companion code for the complete implementation.

```javascript
_symbolTypes = d3.scale.ordinal() // <-A
 .range(["circle",
    "cross",
    "diamond",
    "square",
    "triangle-down",
    "triangle-up"]);
```

...
function renderBody(svg) {
  if (!_bodyG)
    _bodyG = svg.append("g")
      .attr("class", "body")
      .attr("transform", "translate(
        + xStart() + ",",
        + yEnd() + ")")
      .attr("clip-path", "url(#body-clip)");

    renderSymbols();
  }

function renderSymbols() { // <-B
  _data.forEach(function (list, i) {
    _bodyG.selectAll("path._" + i)
      .data(list)
      .enter()
      .append("path")
      .attr("class", "symbol _" + i);

    _bodyG.selectAll("path._" + i)
      .data(list)
      .classed(_symbolTypes(i), true)
      .transition()
      .attr("transform", function(d){
        return "translate("
          + _x(d.x)
          + ","
          + _y(d.y)
          + ")";
      })
      .attr("d",
      d3.svg.symbol().type(_symbolTypes(i)));
  });
  ...
}
This recipe generates a scatter plot chart:

![Scatter plot chart](image)

**How it works...**

The content of the scatter plot chart is mainly rendered by the `renderSymbols` function on line B. You probably have already noticed that the `renderSymbols` function implementation is very similar to the `renderDots` function we discussed in the *Creating a line chart* recipe. This is not by accident since both are trying to plot data points on Cartesian coordinates with two variables (x and y). In the case of plotting dots, we were creating `svg:circle` elements, while in scatter plot we need to create `d3.svg.symbol` elements. D3 provides a list of predefined symbols that can be generated easily and rendered using an `svg:path` element. On line A we defined an ordinal scale to allow mapping of data series index to different symbol types:

```javascript
_symbolTypes = d3.scale.ordinal() // <-A
  .range(["circle",
         "cross",
         "diamond",
```
"square",
"triangle-down",
"triangle-up");

Plotting the data points with symbols is quite straightforward. First we loop through the data series array and for each data series we create a set of svg:path elements representing each data point in the series.

```javascript
_data.forEach(function (list, i) {
    _bodyG.selectAll("path._" + i)
    .data(list)
    .enter()
    .append("path")
    .attr("class", "symbol _" + i);
    ...
});
```

Whenever data series are updated, as well as for newly created symbols, we apply the update with transition (line C) placing them on the right coordinates with an SVG translation transformation (line D).

```javascript
_bodyG.selectAll("path._" + i)
    .data(list)
    .classed(_symbolTypes(i), true)
    .transition() // <-C
    .attr("transform", function(d){
      return "translate(" + _x(d.x) + ","
      + _y(d.y) + ")";
    })
    .attr("d",
      d3.svg.symbol() // <-E
      .type(_symbolTypes(i))
    );
```

Finally, the d attribute of each svg:path element is generated using the d3.svg.symbol generator function as shown on line E.
Creating a bubble chart

A bubble chart is a typical visualization capable of displaying three data dimensions. Every data entity with its three data points is visualized as a bubble (or disk) on Cartesian coordinates, with two different variables represented using x axis and y axis, similar to the scatter plot chart. While the third dimension is represented using the radius of the bubble (size of the disk). Bubble chart is particularly useful when used to facilitate understanding of relationships between data entities.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

In this recipe we will explore techniques and ways of implementing a typical bubble chart using D3. The following code example shows the important implementation aspects of a bubble chart with accessors and peripheral graphic implementation details omitted.

```javascript
var _width = 600, _height = 300,
    _margins = {top: 30, left: 30, right: 30, bottom: 30},
    _x, _y, _r, // <-A
    _data = [],
    _colors = d3.scale.category10(),
    _svg,
    _bodyG;

_chart.render = function () {
    if (!_svg) {
        _svg = d3.select("body").append("svg")
            .attr("height", _height)
            .attr("width", _width);

        renderAxes(_svg);

        defineBodyClip(_svg);
    }

    renderBody(_svg);
};
```
function renderBody(svg) {
    if (!_bodyG)
        _bodyG = svg.append("g")
            .attr("class", "body")
            .attr("transform", "translate(" + xStart() + "," + yEnd() + ")")
            .attr("clip-path", "url(#body-clip)");
    renderBubbles();
}

function renderBubbles() {
    _r.range([0, 50]); // <-B
    _data.forEach(function (list, i) {
        _bodyG.selectAll("circle._" + i)
            .data(list)
            .enter()
            .append("circle") // <-C
            .attr("class", "bubble _" + i);
        _bodyG.selectAll("circle._" + i)
            .data(list)
            .style("stroke", function (d, j) {
                return _colors(j);
            })
            .style("fill", function (d, j) {
                return _colors(j);
            })
            .transition()
            .attr("cx", function (d) {
                return _x(d.x); // <-D
            })
            .attr("cy", function (d) {
                return _y(d.y); // <-E
            })
            .attr("r", function (d) {
                return _r(d.r); // <-F
            });
    });
}...
This recipe generates the following visualization:

![Bubble chart](image)

**How it works...**

Overall, bubble chart implementation follows the same pattern as other chart implementations introduced in this chapter so far. However, since in bubble chart we want to visualize three different dimensions (x, y, and radius) instead of two, therefore a new scale \_r was added in this implementation (line A).

```javascript
var _width = 600, _height = 300,
    _margins = {top: 30, left: 30, right: 30, bottom: 30},
    _x, _y, _r, // <-A
    _data = [],
    _colors = d3.scale.category10(),
    _svg,
    _bodyG;
```

Most of the bubble chart related implementation details are handled by the renderBubbles function. It starts with setting the range on the radius scale (line B). Of course we can also make the radius range configurable in our chart implementation; however, for simplicity we chose to set it explicitly here:

```javascript
function renderBubbles() {
    _r.range([0, 50]); // <-B

    _data.forEach(function (list, i) {
        _bodyG.selectAll("circle._" + i)
            .data(list)
            .enter()
            .append("circle") // <-C
            .attr("class", "bubble _" + i);
```
Once the range is set, then we iterated through our data series and for each series we created a set of svg:circle elements (line C). Finally we handled the newly created bubble as well as its update in the last section, where svg:circle elements are colored and placed to the correct coordinates using its cx and cy attributes (line D and E). In the end, the bubble size is controlled using its radius attribute r mapped using the _r scale we defined earlier (line F).

In some bubble chart implementations, the implementer also leverages the color of each bubble to visualize a fourth data dimension, though some believe this kind of visual representation is hard to grasp and superfluous.

### Creating a bar chart

A bar chart is a visualization that uses either horizontal (row charts) or vertical (column charts) rectangular bars with length proportional to the values that they represent. In this recipe we will implement a column chart using D3. A column chart is capable of visually representing two variables at the same time with its y axis; in other words, the bar height, and its x axis. The x axis values can be either discrete or continuous (for example, a histogram). In our example we choose to visualize continuous values on the x axis and hence effectively implementing a histogram. However, the same techniques can be applied when working with discrete values.
Getting ready

Open your local copy of the following file in your web browser:


How to do it...

The following code example shows the important implementation aspects of a histogram with accessors and peripheral graphic implementation details omitted.

```javascript
var _width = 600, _height = 250,
     _margins = {top: 30, left: 30, right: 30, bottom: 30},
     _x, _y,
     _data = [],
     _colors = d3.scale.category10(),
     _svg,
     _bodyG;

_chart.render = function () {
    if (!_svg) {
        _svg = d3.select("body").append("svg")
            .attr("height", _height)
            .attr("width", _width);

        renderAxes(_svg);

        defineBodyClip(_svg);
    }

    renderBody(_svg);
};

...
function renderBody(svg) {
    if (!_bodyG)
        _bodyG = svg.append("g")
            .attr("class", "body")
            .attr("transform", "translate(" + xStart() + "," + yEnd() + ")")
            .attr("clip-path", "url(#body-clip)");

    renderBars();
}

function renderBars() {
    var padding = 2; // <-A

    _bodyG.selectAll("rect.bar")
        .data(_data)
        .enter()
        .append("rect") // <-B
        .attr("class", "bar");

    _bodyG.selectAll("rect.bar")
        .data(_data)
        .transition()
        .attr("x", function (d) { // <-C
            return _x(d.x);
        })
        .attr("y", function (d) { // <-D
            return _y(d.y);
        })
        .attr("height", function (d) { // <-E
            return yStart() - _y(d.y);
        })
        .attr("width", function (d){ // <-E
            return Math.floor(quadrantWidth() / _data.length) - padding;
        });
}

...
This recipe generates the following visualization:

![Bar chart (histogram)](image)

**How it works...**

One major difference here is that the bar chart implementation does not support multiple data series. Therefore instead of using a 2-dimensional array storing multiple data series as we did with other charts so far, in this implementation, the `_data` array simply stores a single set of data points directly. Main bar chart related visualization logic resides in the `renderBars` function.

```javascript
function renderBars() {
  var padding = 2; // <-A
  ...
}
```

In the first step, we defined the padding between bars (line A), so later on we can automatically calculate the width of each bar. Afterwards we generate an `svg:rect` element (the bars) for each data point (line B).

```javascript
    _bodyG.selectAll("rect.bar")
    .data(_data)
    .enter()
    .append("rect") // <-B
    .attr("class", "bar");
```

Then in the update section we place each bar at the correct coordinates using its `x` and `y` attributes (line C and D) and extend each bar all the way down to touch the x axis with an adaptive height calculated on line E.

```javascript
    _bodyG.selectAll("rect.bar")
    .data(_data)
    .transition()
```
Finally we calculate the optimal width for each bar using the number of bars as well as the padding value we have defined earlier.

```
.attr("width", function(d){
    return Math.floor(quadrantWidth() / _data.length) - padding;
});
```

Of course in a more flexible implementation, we can make the padding configurable instead of being fixed to 2 pixels.

**See also**

Before planning to implement your own reusable chart for your next visualization project, make sure you also check out the following open source reusable chart projects based on D3:

In this chapter we will cover:

- Building a pie chart
- Building a stacked area chart
- Building a treemap
- Building a tree
- Building an enclosure diagram

Introduction

The D3 layout is the focus of this chapter—a concept we have not encountered before. As expected, D3 layouts are algorithms that calculate and generate placement information for a group of elements. However there are a few critical properties worth mentioning before we dive deeper into the specifics:

- **Layouts are data**: Layouts are purely data centric and data driven, they do not generate any graphical or display related output directly. This allows them to be used and reused with SVG or canvas or even when there is no graphical output.

- **Abstract and reusable**: Layouts are abstract, allowing a high degree of flexibility and reusability. You can combine and reuse layouts in various different interesting ways.

- **Layouts are different**: Each layout is different. Every layout provided by D3 focuses on a very special graphical requirement and data structure.

- **Stateless**: Layouts are mostly stateless by design to simplify their usage. What statelessness means here is that generally layouts are like functions, they can be called multiple times with different input data and generate different layout output.

Layouts are interesting and powerful concepts in D3. In this chapter we will explore some of the most commonly used layouts in D3 by creating fully functional visualization leveraging these layouts.
Building a pie chart

A pie chart or a circle graph is a circular diagram divided into sectors (slices). Pie charts are popular in many fields and widely used to demonstrate relationships between different entities though not without criticism. Let's take a look at how a pie chart is implemented using d3.pie.layout first.

```javascript
function pieChart() {
    var _chart = {};

    var _width = 500, _height = 500,
    _data = [],
    _colors = d3.scale.category20(),
    _svg,
    _bodyG,
    _pieG,
```

This essentially requires the calculation of the angle partition for each slice out of an entire circle of 2 * Math.PI. Obviously this process can be automated by an algorithm which is exactly what d3.layout.pie is designed for. In this recipe, we will see how pie layout can be used to implement a fully functional pie chart.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

```javascript
var data = [
    {startAngle: 0, endAngle: 0.6283185307179586},
    {startAngle: 0.6283185307179586, endAngle: 1.2566370614359172},
    ...
    {startAngle: 5.654866776461628, endAngle: 6.283185307179586}
];
```
```javascript
_radius = 200,
_innerRadius = 100;

_chart.render = function () {
  if (!_svg) {
    _svg = d3.select("body").append("svg")
      .attr("height", _height)
      .attr("width", _width);
  }

  renderBody(_svg);
};

function renderBody(svg) {
  if (!_bodyG)
    _bodyG = svg.append("g")
      .attr("class", "body");

  renderPie();
}

function renderPie() {
  var pie = d3.layout.pie()
    .sort(function (d) {
      return d.id;
    })
    .value(function (d) {
      return d.value;
    });

  var arc = d3.svg.arc()
    .outerRadius(_radius)
    .innerRadius(_innerRadius);

  if (!_pieG)
    _pieG = _bodyG.append("g")
      .attr("class", "pie")
      .attr("transform", "translate(" + _radius + "," + _radius + ")");

  renderSlices(pie, arc);

  renderLabels(pie, arc);
```

This recipe generates the following pie chart:
This recipe is built over what we have learned in the Chapter 7, Getting into Shape. One major difference is that we rely on `d3.layout.pie` to transform the raw data into arcs data for us. The pie layout was created on line A with both sort and value accessors specified.

```javascript
var pie = d3.layout.pie() // <-A
  .sort(function (d) {
    return d.id;
  })
  .value(function (d) {
    return d.value;
  });
```

The `sort` function tells the pie layout to sort slices by its ID field, so that we can maintain stable order amongst slices. Without the sorting, by default the pie layout will order the slices by value resulting in the swapping of slices whenever we update the pie chart. The `value` function is used to provide value accessor which in our case returns the `value` field. When rendering slices, now with the pie layout, we directly set the pie layout as data (remember layouts are data) to generate the arc `svg:path` elements (line B).

```javascript
function renderSlices(pie, arc) {
  var slices = _pieG.selectAll("path.arc")
    .data(pie(_data)); // <-B
  slices.enter()
    .append("path")
    .attr("class", "arc")
    .attr("fill", function (d, i) {
      return _colors(i);
    });
  slices.transition()
    .attrTween("d", function (d) {
      var currentArc = this.__current__;//<-C
      if (!currentArc)
        currentArc = {startAngle: 0,
                      endAngle: 0};
      var interpolate = d3.interpolate(
                          currentArc, d);
      this.__current__ = interpolate(1);//<-D
      return function (t) {
```
The rest of the rendering logic is pretty much the same as what we have learned in Chapter 7, Getting into Shape, with one exception on line C. On line C we retrieve the current arc value from the element so the transition can start from the current angle instead of zero. Then on line D we reset the current arc value to the latest one so the next time when we update the pie chart data we can repeat the stateful transition.

**Technique – stateful visualization**

Technique of value injection on a DOM element is a common approach to introduce statefulness to your visualization. In other words, if you need your visualizations to remember what their previous states are, you can save them in DOM elements.

Finally we also need to render labels on each slice so our user can understand what each slice is representing. This is done by the `renderLabels` function.

```javascript
function renderLabels(pie, arc) {
  var labels = _pieG.selectAll("text.label")
    .data(pie(_data)); // <-E

  labels.enter()
    .append("text")
    .attr("class", "label");

  labels.transition()
    .attr("transform", function (d) {
      return "translate(" + arc.centroid(d) + ")"; // <-F
    })
    .attr("dy", ".35em")
    .attr("text-anchor", "middle")
    .text(function (d) {
      return d.data.id;
    });
}
```

Once again we use the pie layout as data to generate the `svg:text` elements. The placement of the labels is calculated using `arc.centroid` (line F). Additionally, the label placement is animated through the transition so they can be moved with arcs in unison.
There's more...

Pie charts are very widely used in many different domains. However, they have also been widely criticized due to the fact that they are difficult for human eyes to compare different sections of a given pie chart as well as their low information density. Therefore, it is highly recommended to limit the number of sections to less than 3, with 2 to be ideal. Otherwise, you can always use a bar chart or a small table to replace a pie chart in places with better precision and communicative power.

See also

- The Using arc generators recipe in Chapter 7, Getting into Shape
- The Implementing arc transition recipe in Chapter 7, Getting into Shape

Building a stacked area chart

In the Creating an area chart recipe in Chapter 8, Chart Them Up, we have explored how a basic layered area chart can be implemented using D3. In this recipe, we will build over what we have learned in the area chart recipe to implement a stacked area chart. Stacked area chart is a variation of the standard area chart in which different areas are stacked on top of each other giving your audience not only the ability to compare different data series individually but also their relationship to the total in proportion.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

This recipe is built over what we have implemented in Chapter 8, Chart Them Up, therefore in the following code example only the parts that are particularly relevant to the stacked area chart are included:

```javascript
function stackedAreaChart() {
    var _chart = {};

    var _width = 900, _height = 450,
    _margins = {top: 30, left: 30, right: 30, bottom: 30},
```
_x, _y,
_data = [],
_colors = d3.scale.category10(),
_svg,
_bodyG,
_line;

_chart.render = function () {
  if (!_svg) {
    _svg = d3.select("body").append("svg")
      .attr("height", _height)
      .attr("width", _width);

    renderAxes(_svg);

    defineBodyClip(_svg);
  }

  renderBody(_svg);
};

function renderBody(svg) {
  if (!_bodyG)
    _bodyG = svg.append("g")
      .attr("class", "body")
      .attr("transform", "translate("
          + xStart() + ","
          + yEnd() + ")")
      .attr("clip-path", "url(#body-clip)"),

  var stack = d3.layout.stack() //<-A
    .offset('zero');
  stack(_data); //<-B

  renderLines(_data);

  renderAreas(_data);
}

function renderLines(stackedData) {
  // explained in details in the 'how it works...' section
function renderAreas(stackedData) {
    // explained in details in the 'how it works...' section
    ...
} ...

This recipe generates the following visualization:

![Stacked area chart](image)

**How it works...**

The main difference between this recipe and standard area chart as well as the focus on this recipe is the stacking. The stacking effect as illustrated in this recipe was achieved through `d3.layout.stack` created on line A.

```javascript
var stack = d3.layout.stack() //<-A
  .offset('zero');
stack(_data); //<-B
```
The only customization we have done on stack layout is setting its offset to zero. D3 stack layout supports a few different offset modes which determine what stacking algorithm to use; this is something that we will explore in this and the next recipe. In this case we use the zero offset stacking which generates a zero base-lined stacking algorithm, which is exactly what we want in this recipe. Next, on line B, we invoked the stack layout on the given data array which generates the following layout data:

As shown, the stack layout automatically calculates a baseline \( y_0 \) for each datum in our three different data series. Now with this stacked dataset in hand, we can easily generate stacked lines.

```
function renderLines(stackedData) {
    _line = d3.svg.line()
        .x(function (d) {
            return _x(d.x); //<-C
        })
        .y(function (d) {
            return _y(d.y + d.y0); //<-D
        });
    _bodyG.selectAll("path.line")
        .data(stackedData)
        .enter()
        .append("path")
        .style("stroke", function (d, i) {
            return _colors(i);
        })
        .attr("class", "line");

    _bodyG.selectAll("path.line")
        .data(stackedData)
        .transition()
        .attr("d", function (d) {
            // Code continues here...
        });
```
A D3 line generator function was created with its x value directly mapped to the x (line C) and its y value mapped to $y + y_0$ (line D). This is all you need to do for line stacking. The rest of the \texttt{renderLines} function is essentially the same as in the basic area chart implementation. The area stacking logic is slightly different:

\begin{verbatim}
function renderAreas(stackedData) {
    var area = d3.svg.area()
        .x(function (d) {
            return _x(d.x); //<-E
        })
        .y0(function(d){return _y(d.y0);}) //<-F
        .y1(function (d) {
            return _y(d.y + d.y0); //<-G
        });

    _bodyG.selectAll("path.area")
        .data(stackedData)
        .enter()
        .append("path")
        .style("fill", function (d, i) {
            return _colors(i);
        })
        .attr("class", "area");

    _bodyG.selectAll("path.area")
        .data(_data)
        .transition()
        .attr("d", function (d) {
            return area(d);
        });
}
\end{verbatim}

Similar to the line rendering logic when rendering area, the only place we need to change is in the \texttt{d3.svg.area} generator setting. For areas the x value is still directly mapped to x (line E) with its y0 directly mapped with y0 and finally again y1 is the sum of y and y0 (line G).

As we have seen so far, D3 stack layout is nicely designed to be compatible with different D3 SVG generator functions. Hence, using it to generate the stacking effect is quite straightforward and convenient.
There's more...

Let's take a look at a couple of variations of the stacked area chart.

**Expanded area chart**

We have mentioned that `d3.layout.stack` supports different offset modes. In addition to the zero offset we have seen so far, another very useful offset mode for area chart is called `expand`. With the `expand` mode, stack layout will normalize different layers to fill the range of [0, 1]. If we change the offset mode in this recipe and the y axis domain to [0, 1], we will get the expanded (normalized) area chart shown below.

For the complete companion code example please visit: https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter9/expanded-area-chart.html.

**Streamgraph**

Another interesting variation of stacked area chart is called streamgraph. Streamgraph is a stacked area chart displayed around a central axis creating a flowing and organic shape. Streamgraph was initially developed by Lee Byron and popularized by its use in a New York Times article on movie box office revenues in 2008. The D3 stack layout has built-in support for this kind of stacking algorithm therefore changing a zero based stacked area chart to streamgraph is trivial. The key difference is that streamgraph uses `wiggle` as its layout offset mode.
For the complete companion code example please visit: https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter9/streamgraph.html.

See also

- d3.layout.stack offers several additional functions to customize its behavior; for more information on stack layout visit https://github.com/mbostock/d3/wiki/Stack-Layout
- The Creating an area chart recipe in Chapter 8, Chart Them Up

Building a treemap

Treemaps were introduced by Ben Shneiderman in 1991. A treemap displays hierarchical tree-structured data as a set of recursively subdivided rectangles. In other words, it displays each branch of the tree as a large rectangle which is then tiled with smaller rectangles representing sub-branches. This process continuously repeats itself till it reaches the leaves of the tree.

For more information on treemaps, see this paper by Ben Shneiderman at http://www.cs.umd.edu/hcil/treemap-history/

Before we dive into the code example, let’s first define what we mean by hierarchical data.
So far we have learned many types of visualizations capable of representing flat data structure usually stored in one or two dimensional arrays. In the rest of this chapter, we will switch our focus onto another common type of data structure in data visualization—the hierarchical data structure. Instead of using arrays, as in the case of flat data structures, hierarchical data are usually structured as a rooted tree. The following JSON file shows a typical hierarchical data you would expect in a data visualization project:

```json
{
  "name": "flare",
  "children": [
    {
      "name": "analytics",
      "children": [
        {
          "name": "cluster",
          "children": [
            {"name": "AgglomerativeCluster", "size": 3938},
            {"name": "CommunityStructure", "size": 3812},
            {"name": "MergeEdge", "size": 743}
          ]
        },
        {
          "name": "graph",
          "children": [
            {"name": "BetweennessCentrality", "size": 3534},
            {"name": "LinkDistance", "size": 5731}
          ]
        },
        {
          "name": "optimization",
          "children": [
            {"name": "AspectRatioBanker", "size": 7074}
          ]
        }
      ]
    }
  ]
}
```

This is a shortened version of a popular hierarchical dataset used in the D3 community for demonstration purposes. This data is extracted from a popular flash based data visualization library—Flare, created by the UC Berkeley Visualization Lab. It shows the size and hierarchical relationship amongst different packages within the library.
As we can see quite easily this particular JSON feed is structured as a typical singly-linked rooted tree with each node having a single parent and multiple child nodes stored in the `children` array. This is the most natural way to organize your hierarchical data in order to be consumed by the D3 hierarchical layouts. For the rest of this chapter, we will use this particular dataset for exploring different hierarchical data visualization techniques D3 has to offer.

**Getting ready**

Open your local copy of the following file in your web browser:


**How to do it...**

Now let’s see how we can use the D3 treemap layout to visually represent this kind of hierarchical data.

```javascript
function treemapChart() {
  var _chart = {};

  var _width = 1600, _height = 800,
      _colors = d3.scale.category20c(),
      _svg,
      _nodes,
      _x = d3.scale.linear().range([0, _width]),
      _y = d3.scale.linear().range([0, _height]),
      _valueAccessor = function (d) {
        return 1;
    },
      _bodyG;

  _chart.render = function () {
    if (!_svg) {
      _svg = d3.select("body").append("svg")
        .attr("height", _height)
        .attr("width", _width);
    }

    renderBody(_svg);

    d3.json("http://flare.prefuse.org/flare.json", function (error, data) {
      _nodes = data;
    });

    d3.selectAll(_nodes).on("load", function () {
      var x = _x, y = _y,
          i = 0,
          n = _nodes.length;

      while (i < n) {
        var n = _nodes[i++];

        _bodyG.append("rect")
          .attr("x", x(_valueAccessor(n)))
          .attr("y", y(_valueAccessor(n)))
          .attr("width", x(1) - x(_valueAccessor(n)))
          .attr("height", y(1) - y(_valueAccessor(n)))
          .style("fill", _colors(i % 20));
      }
    });
  }
}
```

See the official Flare site for more information on the project: [http://flare.prefuse.org/](http://flare.prefuse.org/).
Lay Them Out

```javascript
};

function renderBody(svg) {
  // explained in details in the 'how it works...' section
  ...

  renderCells(cells);
}

function renderCells(cells){
  // explained in details in the 'how it works...' section
  ...
}

// accessors omitted
...

return _chart;
}

d3.json("flare.json", function (nodes) {
  var chart = treemapChart();
  chart.nodes(nodes).render();
});
```

This recipe generates the following treemap visualization:
At this point you might be surprised how little code is needed to implement a complex data visualization like this. This is because most of the heavy lifting is done by `d3.layout.treemap`.

```javascript
function renderBody(svg) {
  if (!_bodyG) {
    _bodyG = svg.append("g")
      .attr("class", "body");

    _treemap = d3.layout.treemap() //<-A
      .round(false)
      .size([_width, _height])
      .sticky(true); 
  }

  _treemap.value(_valueAccessor); //<-B

  var nodes = _treemap.nodes(_nodes) //<-C
    .filter(function (d) {
      return !d.children; //<-D
    });

  var cells = svg.selectAll("g") //<-E
    .data(nodes);

  renderCells(cells);
}
```

The treemap layout is defined on line A with some basic custom settings:

- `round(false)`: If rounding is on, the treemap layout will round to exact pixel boundaries. This is great when you want to avoid antialiasing artifacts in SVG.
- `size([_width, _height])`: It sets the layout boundary to the size of this SVG.
- `sticky(true)`: In sticky mode, the treemap layout will try to preserve the relative arrangement of nodes (rectangles in our case) across the transition.
- `value(_valueAccessor)`: One feature this recipe offers is the ability to switch the treemap value accessor on the fly. Value accessor is used by a treemap to access value field on each node. In our case, it can be either one of the following functions:
  ```javascript
  function(d){ return d.size; } // visualize package size  
  function(d){ return 1; }  // visualize package count  
  ```
To apply a treemap layout on Flare JSON datafeed, we simply set the nodes on the treemap layout to the root node in our JSON tree (line C). Treemap nodes are then further filtered to remove parent nodes (nodes that have children) on line D since we only want to visualize the leaf nodes while using coloring to highlight the package grouping in this treemap implementation. The layout data generated by treemap layout contains the following structure:

```
Treemap node object
```

As shown, the treemap layout has annotated and calculated quite a few attributes for each node using its algorithm. Many of these attributes can be useful when visualizing and in this recipe we mostly care about the following attributes:

- **x**: Cell x coordinate
- **y**: Cell y coordinate
- **dx**: Cell width
- **dy**: Cell height

On line E, a set of `svg:g` elements were created for the given nodes. The function `renderCells` is then responsible for creating rectangles and its labels:

```javascript
function renderCells(cells) {
    var cellEnter = cells.enter().append("g")
        .attr("class", "cell");

    cellEnter.append("rect")
    cellEnter.append("text");

    cells.transition().attr("transform", function (d) {
        return "translate(-d.x,-d.y)"; //<-F
    })
    .select("rect")
    .attr("width", function (d) {return d.dx - 1;})
```
Each rectangle is placed at its location \((x, y)\) determined by the layout on line F, and then its width and height are set to \(dx\) and \(dy\). On line G, we colored every cell using its parent's names therefore making sure all children belonging to the same parent are colored the same way. From line H onward we created the label (\(\text{svg:text}\)) element for each rectangle and setting its text to the node name. One aspect worth mentioning here is that in order to avoid displaying label for the cells that are smaller than the label itself, the opacity of label is set to 0 if the label is larger than the cell width (line I).

### Technique – auto-hiding label

What we have seen here on line I is a useful technique in visualization to implement auto-hiding labels. This technique can be considered generally in the following form:

```javascript
.style("opacity", function (d) {
  d.w = this.getComputedTextLength();
  return d.dx > d.w ? 1 : 0;
});
```

### See also

- This recipe is inspired by Mike Bostock's treemap layout example, which you can find at [http://mbostock.github.io/d3/talk/20111018/treemap.html](http://mbostock.github.io/d3/talk/20111018/treemap.html)
Building a tree

When working with hierarchical data structures, a tree (tree graph) is probably one of the most natural and common visualizations typically leveraged to demonstrate structural dependencies between different data elements. Tree is an undirected graph in which any two nodes (vertices) are connected by one and only one simple path. In this recipe, we will learn how to implement a tree visualization using tree layout.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

Now let's see d3.layout.tree in action:

```javascript
function tree() {
  var _chart = {};

  var _width = 1600, _height = 800,
      _margins = {top:30, left:120, right:30, bottom:30},
      _svg,
      _nodes,
      _i = 0,
      _tree,
      _diagonal,
      _bodyG;

  _chart.render = function () {
    if (!_svg) {
      _svg = d3.select("body").append("svg")
        .attr("height", _height)
        .attr("width", _width);

    }

    renderBody(_svg);
  }

  function renderBody(svg) {
    if (!_bodyG) {
      _bodyG = svg.append("g")
        .attr("class", "body")
```
.attr("transform", function (d) {
    return "translate(" + _margins.left
    + "," + _margins.top + ")";
});

_tree = d3.layout.tree()
  .size([(_height - _margins.top - _margins.bottom),
           (_width - _margins.left - _margins.right)]);

_diagonal = d3.svg.diagonal()
  .projection(function (d) {
    return [d.y, d.x];
  });

_nodes.x0 = (_height - _margins.top - _margins.bottom) / 2;
_nodes.y0 = 0;

render(_nodes);
}

function render(source) {
  var nodes = _tree.nodes(_nodes);

  renderNodes(nodes, source);
  renderLinks(nodes, source);
}

function renderNodes(nodes, source) {
    // will be explained in the 'how it works...' section
    ...
}

function renderLinks(nodes, source) {
    // will be explained in the 'how it works...' section
    ...
}

// accessors omitted
...

return _chart;
}
This recipe generates the following tree visualization:

```
How it works...
As we have mentioned before, this recipe is built over the D3 tree layout. d3.layout.tree is specifically designed to convert a hierarchical data structure into a visual layout data suitable for generating tree graph. Our tree layout instance is defined as the following:

```js
_tree = d3.layout.tree()
  .size([(_height - _margins.top - _margins.bottom),
         (_width - _margins.left - _margins.right)]);
```
The only setting we provided here is the size of our visualization, which is the size of our SVG image minus the margins. `d3.layout.tree` will then take care of the rest and calculate every node's position accordingly. To use the tree layout, you simply invoke its `nodes` function.

```javascript
var nodes = _tree.nodes(_nodes);
```

If you peek into the `nodes` layout data, it contains node data looking like this:

```
0: Object
  children: Array[2]
  depth: 0
  id: 1
  name: "flars"
  x: 346.875
  y: 0
  y0: 0
  __proto__: Object
1: Object
  children: null
  depth: 1
  id: 2
  name: "analytics"
  parent: Object
    x: 154.16666666666669
    y: 180
  y0: 180
  __proto__: Object
2: Object
```

Tree layout data

One new D3 SVG shape generator we need for this recipe that is worth mentioning is `d3.svg.diagonal`. The diagonal generator is designed to create `svg:path` that connects two points. In this recipe, we use diagonal generator with tree layout `links` function to generate a path connecting every node in the tree.

```javascript
_diagonal = d3.svg.diagonal()
    .projection(function (d) {
      return [d.y, d.x];
    });
```

In this case we configure our diagonal generator to project using Cartesian orientation and simply reply on the x and y coordinates calculated by the tree layout for positioning. The actual rendering was performed by the following functions. First let's take a look at the `renderNodes` function:

```javascript
function renderNodes(nodes, source) {
  nodes.forEach(function (d) {
    d.y = d.depth * 180;
  });
```
Lay Them Out

Here we loop through all the nodes and artificially assign a 180-pixel spacing between them. You probably are wondering why we are using the y coordinate instead of x. The reason is that in this recipe we want to render a horizontal tree instead of a vertical one; therefore we have to reverse the x and y coordinates here.

```javascript
var node = _bodyG.selectAll("g.node")
  .data(nodes, function (d) {
    return d.id || (d.id = ++_i);
  });
```

Now we bind the nodes that were generated by the tree layout as data to generate the tree node element. At this point, we also assign an ID to each node using an index to obtain object constancy.

```javascript
var nodeEnter = node.enter().append("svg:g")
  .attr("class", "node")
  .attr("transform", function (d) {
    return "translate(" + source.y0 + "," + source.x0 + ")";
  });
```

At this point, we create the nodes and move them to the same point of origin as set in the renderBody function.

```javascript
nodeEnter.append("svg:circle")
  .attr("r", 1e-6);
```

```javascript
var nodeUpdate = node.transition()
  .attr("transform", function (d) {
    return "translate(" + d.y + "," + d.x + ")";
  });
nodeUpdate.select("circle")
  .attr("r", 4.5);
```

Now we start a transition in the update section to move the nodes to their proper position.

```javascript
var nodeExit = node.exit().transition()
  .attr("transform", function (d) {
    return "translate(" + source.y + "," + source.x + ")";
  })
  .remove();
nodeExit.select("circle")
  .attr("r", 1e-6);
renderLabels(nodeEnter, nodeUpdate, nodeExit);
```
At last, we handle the exit case and remove the nodes after a brief animation of the collapsing effect. The `renderLabels` function is quite simple so we will not cover it in detail here. Please see the complete online code companion for details.

Now let's take a look at the more interesting `renderLinks` function.

```javascript
function renderLinks(nodes, source) {
    var link = _bodyG.selectAll("path.link")
        .data(_tree.links(nodes), function (d) {
            return d.target.id;
        });

    link.enter().insert("svg:path", "g")
        .attr("class", "link")
        .attr("d", function (d) {
            var o = {x: source.x0, y: source.y0};
            return _diagonal({source: o, target: o});
        });

    link.transition()
        .attr("d", _diagonal);

    link.exit().transition()
        .attr("d", function (d) {
            var o = {x: source.x, y: source.y};
            return _diagonal({source: o, target: o});
        })
        .remove();
}
```

In the `enter` section, the `svg:path` elements were created to visually represent the links between source and target nodes. To generate the `d` attribute for the path element we rely on the `d3.svg.diagonal` generator we defined earlier. During creation we temporarily set the links to zero length paths by setting both source and target to the same point of origin. So when later we transition the link to its proper length, it will generate the expanding effect.

```javascript
    link.transition()
        .attr("d", _diagonal);
```

Now we transition the links to its proper length and position using the links data generated by the tree layout.

```javascript
    link.exit().transition()
        .attr("d", function (d) {
            var o = {x: source.x, y: source.y};
            return _diagonal({source: o, target: o});
        })
        .remove();
```

When we remove the nodes again, we rely on the same trick of setting the link to its parent's position with zero length in order to simulate the collapsing effect.
Building an enclosure diagram

An enclosure diagram is an interesting visualization of hierarchical data structures that uses the recursive circle packing algorithm. It uses containment (nesting) to represent hierarchy. Circles are created for each leaf node in a data tree while its size is proportional to a particular quantitative dimension of each data element. In this recipe, we will learn how to implement this kind of visualization using D3 pack layout.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

In this recipe, let’s see how we can implement an enclosure diagram using `d3.layout.pack`.

```javascript
function pack() {
  var _chart = {};

  var _width = 1280, _height = 800,
  _svg,
  _r = 720,
  _x = d3.scale.linear().range([0, _r]),
  _y = d3.scale.linear().range([0, _r]),
  _nodes,
```
function renderBody(svg) {
  if (!_bodyG) {
    _bodyG = svg.append("g")
      .attr("class", "body")
      .attr("transform", function (d) {
        return "translate("
          + (_width - _r) / 2 + ",";
        + (_height - _r) / 2
        + ")";
      });
  }

  var pack = d3.layout.pack()
    .size([_r, _r])
    .value(function (d) {
      return d.size;
    });

  var nodes = pack.nodes(_nodes);

  renderCircles(nodes);

  renderLabels(nodes);
}

function renderCircles(nodes) {
  // will be explained in the 'how it works...' section
  ...
}

function renderLabels(nodes) {
  // omitted
This recipe generates the following visualization:
How it works...

First thing we need to take care of in this recipe is to define our layout; in this case we need to use the `d3.layout.pack` layout.

```javascript
var pack = d3.layout.pack()
  .size([_r, _r])
  .value(function (d) {
    return d.size;
  });

var nodes = pack.nodes(_nodes);
```

Now we set the size of the layout using the outer circle's radius and set the value to use the Flare package size, which in turn will determine each circle's size; hence, effectively making each circle's size proportional to the package size in our data feed. Once layout is created, we feed our data elements through its `nodes` function generating the layout data with the following structure:

```
Pack layout data

0: Object
  ▶ children: Array[2]
    ▶ depth: 0
      name: "flare"
    r: 360
    value: 124856
    x: 360
    y: 360
  ▶ __proto__: Object

1: Object
  ▶ children: Array[3]
    ▶ depth: 1
      name: "analytics"
    ▶ parent: Object
      r: 124.81356646107181
      value: 24832
      x: 124.81356646107184
      y: 360
  ▶ __proto__: Object
```

Circle rendering is done in the `renderCircle` function:

```javascript
function renderCircles(nodes) {
  var circles = _bodyG.selectAll("circle")
    .data(nodes);

  circles.enter().append("svg:circle");
```
Then we simply bind the layout data and create the `svg:circle` elements for each node.

```javascript
circles.transition()
    .attr("class", function (d) {
        return d.children ? "parent" : "child";
    })
    .attr("cx", function (d) {return d.x; })
    .attr("cy", function (d) {return d.y; })
    .attr("r", function (d) {return d.r; });
```

For update, we set `cx`, `cy`, and `radius` to the value that the pack layout has calculated for us for each circle.

```javascript
circles.exit().transition()
    .attr("r", 0)
    .remove();
```

Finally when removing the circle, we reduce the size of the circle down to zero first, before removing them to generate a more smooth transition. Label rendering in this recipe is pretty straightforward with some help from the auto-hiding technique we introduced in this chapter, so we will not cover the function in detail here.

**See also**

- The *Building a treemap* recipe for auto label hiding technique.
- This recipe is inspired by Mike Bostock's pack layout example, which you can find at [http://mbostock.github.io/d3/talk/20111018/pack.html](http://mbostock.github.io/d3/talk/20111018/pack.html).
In this chapter we will cover:

- Interacting with the mouse
- Interacting with a multi-touch device
- Implementing zoom and pan behavior
- Implementing the drag behavior

Introduction

The ultimate goal of visualization design is to optimize applications so that they help us perform cognitive work more efficiently.

Ware C. (2012)

The goal of data visualization is to help the audience gain information from a large quantity of raw data quickly and efficiently through metaphor, mental model alignment, and cognitive magnification. So far in this book we have introduced various techniques to leverage D3 library implementing many types of visualization. However, we haven’t touched a crucial aspect of visualization: human interaction. Various researches have concluded the unique value of human interaction in information visualization.

Visualization combined with computational steering allows faster analyses of more sophisticated scenarios...This case study adequately demonstrate that the interaction of a complex model with steering and interactive visualization can extend the applicability of the modelling beyond research

Barrass I. & Leng J (2011)
In this chapter we will focus on D3 human visualization interaction support, or as mentioned earlier learn how to add computational steering capability to your visualization.

**Interacting with mouse events**

The mouse is the most common and popular human-computer interaction control found on most desktop and laptop computers. Even today, with multi-touch devices rising to dominance, touch events are typically still emulated into mouse events; therefore making application designed to interact via mouse usable through touches. In this recipe we will learn how to handle standard mouse events in D3.

**Getting ready**

Open your local copy of the following file in your web browser:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter10/mouse.html

**How to do it...**

In the following code example we will explore techniques of registering and handling mouse events in D3. Although, in this particular example we are only handling `click` and `mousemove`, the techniques utilized here can be applied easily to all other standard mouse events supported by modern browsers:

```javascript
<script type="text/javascript">
    var r = 400;

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

    var positionLabel = svg.append("text")
        .attr("x", 10)
        .attr("y", 30);

    svg.on("mousemove", function () { //<-A
        printPosition();
    });

    function printPosition() { //<-B
        var position = d3.mouse(svg.node()); //<-C
        positionLabel.text(position);
    }
</script>
```
svg.on("click", function () {
    for (var i = 1; i < 5; ++i) {
        var position = d3.mouse(svg.node());

        var circle = svg.append("circle")
            .attr("cx", position[0])
            .attr("cy", position[1])
            .attr("r", 0)
            .style("stroke-width", 5 / (i))
            .transition()
                .delay(Math.pow(i, 2.5) * 50)
                .duration(2000)
                .ease('quad-in')
            .attr("r", r)
            .style("stroke-opacity", 0)
            .each("end", function () {
                d3.select(this).remove();
            });
    }
});
</script>

This recipe generates the following interactive visualization:
How it works...

In D3, to register an event listener, we need to invoke the `on` function on a particular selection. The given event listener will be attached to all selected elements for the specified event (line A). The following code in this recipe attaches a `mousemove` event listener which displays the current mouse position (line B):

```javascript
svg.on("mousemove", function () { //<-A
    printPosition();
});

function printPosition() { //<-B
    var position = d3.mouse(svg.node()); //<-C
    positionLabel.text(position);
}
```

On line C we used `d3.mouse` function to obtain the current mouse position relative to the given container element. This function returns a two-element array `[x, y]`. After this we also registered an event listener for mouse `click` event on line D using the same `on` function:

```javascript
svg.on("click", function () { //<-D
    for (var i = 1; i < 5; ++i) {
        var position = d3.mouse(svg.node());
        var circle = svg.append("circle")
            .attr("cx", position[0])
            .attr("cy", position[1])
            .attr("r", 0)
            .style("stroke-width", 5 / (i)) // <-E
            .transition()
            .delay(Math.pow(i, 2.5) * 50) // <-F
            .duration(2000)
            .ease('quad-in')
            .attr("r", r)
            .style("stroke-opacity", 0)
            .each("end", function () {
                d3.select(this).remove(); // <-G
            });
    }
});
```

Once again, we retrieved the current mouse position using `d3.mouse` function and then generated five concentric expanding circles to simulate the ripple effect. The ripple effect was simulated using geometrically increasing delay (line F) with decreasing `stroke-width` (line E). Finally when the transition effect is over, the circles were removed using transition `end` listener (line G). If you are not familiar with this type of transition control please review Chapter 6, Transition with Style, for more details.
There's more...

Although, we have only demonstrated listening on the click and mousemove events in this recipe, you can listen on any event that your browser supports through the on function. The following is a list of mouse events that are useful to know when building your interactive visualization:

- **click**: Dispatched when user clicks a mouse button
- **dbclick**: Dispatched when a mouse button is clicked twice
- **mousedown**: Dispatched when a mouse button is pressed
- **mouseenter**: Dispatched when mouse is moved onto the boundaries of an element or one of its descendent elements
- **mouseleave**: Dispatched when mouse is moved off of the boundaries of an element and all of its descendent elements
- **mousemove**: Dispatched when mouse is moved over an element
- **mouseout**: Dispatched when mouse is moved off of the boundaries of an element
- **mouseover**: Dispatched when mouse is moved onto the boundaries of an element
- **mouseup**: Dispatched when a mouse button is released over an element

See also

- *Chapter 6, Transition with Style*, for more details on the ripple effect technique used in this recipe
- W3C DOM Level 3 Events specification for a complete list of event types: [http://www.w3.org/TR/DOM-Level-3-Events/](http://www.w3.org/TR/DOM-Level-3-Events/)

Interacting with a multi-touch device

Today, with the proliferation of multi-touch devices, any visualization targeting mass consumption needs to worry about its interactability not only through the traditional pointing device, but through multi-touches and gestures as well. In this recipe we will explore touch support offered by D3 to see how it can be leveraged to generate some pretty interesting interaction with multi-touch capable devices.
Getting ready

Open your local copy of the following file in your web browser:


How to do it...

In this recipe we will generate a progress-circle around the user’s touch and once the progress is completed then a subsequent ripple effect will be triggered around the circle. However, if the user prematurely ends his/her touch, then we shall stop the progress-circle without generating the ripples:

```javascript
<script type="text/javascript">
  var initR = 100,
    r = 400,
    thickness = 20;

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

  d3.select("body")
    .on("touchstart", touch)
    .on("touchend", touch);

  function touch() {
    d3.event.preventDefault();

    var arc = d3.svg.arc()
      .outerRadius(initR)
      .innerRadius(initR - thickness);

    var g = svg.selectAll("g.touch")
      .data(d3.touches(svg.node()), function (d) {
        return d.identifier;
      });

    g.enter()
      .append("g")
        .attr("class", "touch")
        .attr("transform", function (d) {
          return "translate(" + d[0] + "," + d[1] + ")";
        })
      .append("path")
        .attr("class", "arc")
        .attr("d", arc);

    g.exit()
      .transition()
        .duration(1000)
        .attr("fill", "none")
        .remove();

    var ripple = d3.select("body")
      .selectAll("circle")
      .data(d3.touches(svg.node()), function (d) {
        return d.identifier;
      });

    ripple.enter()
      .append("circle")
        .attr("class", "ripple")
        .attr("fill", "#409EFF")
        .attr("r", 5)
        .attr("transform", function (d) {
          return "translate(" + d[0] + "," + d[1] + ")";
        });

    ripple.exit()
      .transition()
        .duration(1000)
        .attr("opacity", 0)
        .remove();

  }
</script>
```
Chapter 10

```javascript
.chapter().duration(2000)
.attrTween("d", function (d) {
  var interpolate = d3.interpolate(
    {startAngle: 0, endAngle: 0},
    {startAngle: 0, endAngle: 2 * Math.PI}
  );
  return function (t) {
    return arc(interpolate(t));
  };
})
.each("end", function (d) {
  if (complete(g))
    ripples(d);
  g.remove();
});

g.exit().remove().each(function () {
  this.__stopped__ = true;
});
}

function complete(g) {
  return g.node().__stopped__ != true;
}

function ripples(position) {
  for (var i = 1; i < 5; ++i) {
    var circle = svg.append("circle")
      .attr("cx", position[0])
      .attr("cy", position[1])
      .attr("r", initR - (thickness / 2))
      .style("stroke-width", thickness / (i))
      .transition().delay(Math.pow(i, 2.5) * 50).
      duration(2000).ease('quad-in')
        .attr("r", r)
        .style("stroke-opacity", 0)
        .each("end", function () {
          d3.select(this).remove();
        });
  }
}

</script>
```
Interacting with your Visualization

This recipe generates the following interactive visualization on a touch enabled device:

![Touch Interaction](image)

**How it works...**

Event listener for touch events are registered through D3 selection’s `on` function similar to what we have done with mouse events in the previous recipe:

```javascript
    d3.select("body")
        .on("touchstart", touch)
        .on("touchend", touch);
```

One crucial difference here is that we have registered our touch event listener on the `body` element instead of the `svg` element since with many OS and browsers there are default touch behaviors defined and we would like to override it with our custom implementation. This is done through the following function call:

```javascript
    d3.event.preventDefault();
```

Once the touch event is triggered we retrieve multiple touch point data using the `d3.touches` function as illustrated by the following code snippet:

```javascript
    var g = svg.selectAll("g.touch")
        .data(d3.touches(svg.node()), function (d) {
            return d.identifier;
        });
```

Instead of returning a two-element array as what `d3.mouse` function does, `d3.touches` returns an array of two-element arrays since there could be multiple touch points for each touch event. Each touch position array has data structure that looks like the following:

```javascript
[0: Array[2]
  0: 669
  1: 232
  identifier: 0
  length: 2
  _proto_: Array[0]
```
Other than the \([x, y]\) position of the touch point each position array also carries an identifier to help you differentiate each touch point. We used this identifier here in this recipe to establish object constancy. Once the touch data is bound to the selection the progress circle was generated for each touch around the user's finger:

```javascript
  g.enter()
    .append("g")
    .attr("class", "touch")
    .attr("transform", function (d) {
      return "translate(" + d[0] + "," + d[1] + ")";
    })
    .append("path")
    .attr("class", "arc")
    .transition().duration(2000).ease('linear')
    .attrTween("d", function (d) { // <-A
      var interpolate = d3.interpolate(
        {startAngle: 0, endAngle: 0},
        {startAngle: 0, endAngle: 2 * Math.PI}
      );
      return function (t) {
        return arc(interpolate(t));
      };
    })
    .each("end", function (d) { // <-B
      if (complete(g))
        ripples(d);
      g.remove();
    });
```

This is done through a standard arc transition with attribute tweening (line A) as explained in Chapter 7, *Getting into Shape*. Once the transition is over if the progress-circle has not yet been canceled by the user then a ripple effect similar to what we have done in the previous recipe was generated on line B. Since we have registered the same event listener touch function on both touchstart and touchend events, we can use the following lines to remove progress-circle and also set a flag to indicate that this progress circle has been stopped prematurely:

```javascript
  g.exit().remove().each(function () {
    this.__stopped__ = true;
  });
```

We need to set this stateful flag since there is no way to cancel a transition once it is started; hence, even after removing the progress-circle element from the DOM tree the transition will still complete and trigger line B.
Interacting with your Visualization

There's more...

We have demonstrated touch interaction through the touchstart and touchend events; however, you can use the same pattern to handle any other touch events supported by your browser. The following list contains the proposed touch event types recommended by W3C:

- **touchstart**: Dispatched when the user places a touch point on the touch surface
- **touchend**: Dispatched when the user removes a touch point from the touch surface
- **touchmove**: Dispatched when the user moves a touch point along the touch surface
- **touchcancel**: Dispatched when a touch point has been disrupted in an implementation-specific manner

See also

- Chapter 6, Transition with Style, for more details on object constancy and the ripple effect technique used in this recipe
- Chapter 7, Getting into Shape, for more details on the progress-circle attribute tween transition technique used in this recipe
- W3C Touch Events proposed recommendation for a complete list of touch event types: [http://www.w3.org/TR/touch-events/](http://www.w3.org/TR/touch-events/)

Implementing zoom and pan behavior

Zooming and panning are common and useful techniques in data visualization, which work particularly well with SVG based visualization since vector graphic does not suffer from pixelation as its bitmap counterpart would. Zooming is especially useful when dealing with large data set when it is impractical or impossible to visualize the entire data set, thus a zoom and drill-down approach needs to be employed. In this recipe we will explore D3’s built-in support for both zooming and panning.

Getting ready

Open your local copy of the following file in your web browser:

In this recipe we will implement geometric zooming and panning using D3 zoom support. Let's see how this is done in code:

```javascript
<script type="text/javascript">
var width = 960, height = 500, r = 50;

var data = [
    [width / 2 - r, height / 2 - r],
    [width / 2 - r, height / 2 + r],
    [width / 2 + r, height / 2 - r],
    [width / 2 + r, height / 2 + r]
];

var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height)
    .call(
        d3.behavior.zoom()
        .scaleExtent([1, 10])
        .on("zoom", zoom)
    ).append("g");

svg.selectAll("circle")
    .data(data)
    .enter().append("circle")
    .attr("r", r)
    .attr("transform", function (d) {
        return "translate(" + d + ")";
    });

function zoom() {
    svg.attr("transform", "translate(" + d3.event.translate + ")scale(" + d3.event.scale + ")")
}
</script>
"Interacting with your Visualization"

This recipe generates the following zooming and panning effect:
At this point you might be surprised to see how little code is necessary to implement this fully-functional zoom and pan effect with D3. If you have this recipe open in your browser, you will also notice zooming and panning reacts perfectly well to both mouse wheel and multi-touch gesture. Most of the heavy lifting is done by D3 library. What we have to do here is to simply define what zoom behavior is. Let's see how this is done in the code. Firstly, we need to define zoom behavior on a SVG container:

```javascript
var svg = d3.select("body").append("svg")
    .attr("style", "1px solid black")
    .attr("width", width)
    .attr("height", height)
    .call( // <-A
        d3.behavior.zoom() // <-B
        .scaleExtent([1, 10]) // <-C
        .on("zoom", zoom) // <-D
    )
    .append("g");
```

As we can see on line A, a `d3.behavior.zoom` function was created (line B) and invoked on the `svg` container. `d3.behavior.zoom` will automatically create event listeners to handle the low-level zooming and panning gesture on the associated SVG container (in our case the `svg` element itself). The low-level zoom gesture will then be translated to a high-level D3 zoom event. The default event listeners support both mouse and touch events. On line C we define `scaleExtent` with a 2-element array `[1, 10]` (a range). The scale extent defines how much zoom should be allowed (in our case we allow 10X zoom). Finally, on line D we register a custom zoom event handler to handle D3 zoom events. Now, let's take a look at what job this zoom event handler performs:

```javascript
function zoom() {
    svg.attr("transform", "translate("
        + d3.event.translate
        + ")scale(" + d3.event.scale + ")");
}
```

In the `zoom` function we simply delegate the actual zooming and panning to SVG transformation. To further simplify this task D3 zoom event has also calculated necessary translate and scale. So all we need to do is embed them into SVG transform attribute. Here are the properties contained in a zoom event:

- **scale**: A number representing the current scale
- **translate**: A two-element array representing the current translation vector
At this point you might be asking what is the point of having this zoom function. Why can't D3 take care of this step for us? The reason is that D3 zoom behavior is not designed specifically for SVG, but rather designed as a general zoom behavior support mechanism. Therefore, this zoom function implements the translation of general zoom and pan events into SVG specific transformation.

There's more...

The zoom function is also capable of performing additional tasks other than simple coordinate system transformation. For example, a common technique is to load additional data when the user issues a zoom gesture, hence implementing the drill-down capability in zoom function. A well-known example is a digital map; as you increase zoom level on a map, more data and details then can be loaded and illustrated.

See also

- *Chapter 2, Be Selective*, for more details on `d3.selection.call` function and selection manipulation
- W3C SVG Coordinate system transformations specification for more information on how zoom and pan effect was achieved in SVG: [http://www.w3.org/TR/SVG/coords.html#EstablishingANewUserSpace](http://www.w3.org/TR/SVG/coords.html#EstablishingANewUserSpace)

Implementing drag behavior

Another common behavior in interactive visualization that we will cover in this chapter is **drag**. Drag is useful to provide capabilities in visualization allowing graphical rearrangement or even user input through force, which we will discuss in the next chapter. In this recipe we will explore how drag behavior is supported in D3.

Getting ready

Open your local copy of the following file in your web browser:

How to do it...

Here, we will produce four circles that can be dragged using D3 drag behavior support and additionally with SVG boundary detection while dragging. Now, let’s see how to implement this in code:

```javascript
<script type="text/javascript">
  var width = 960, height = 500, r = 50;

  var data = [
    [width / 2 - r, height / 2 - r],
    [width / 2 - r, height / 2 + r],
    [width / 2 + r, height / 2 - r],
    [width / 2 + r, height / 2 + r]
  ];

  var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g");

  var drag = d3.behavior.drag()
    .on("drag", move);

  svg.selectAll("circle")
    .data(data)
    .enter().append("circle")
    .attr("r", r)
    .attr("transform", function (d) {
      return "translate(" + d + ")";
    })
    .call(drag);

  function move(d) {
    var x = d3.event.x,
        y = d3.event.y;

    if(inBoundaries(x, y))
      d3.select(this)
        .attr("transform", function (d) {
          return "translate(" + x + ", " + y + ")";
        });

  }

  function inBoundaries(x, y){
    return (x >= (0 + r) && x <= (width - r))
      && (y >= (0 + r) && y <= (height - r));
  }

</script>
```
This recipe generates drag behavior on the following four circles:

![Original](image1)

![Dragged](image2)

**How it works...**

As we can see, similar to D3 zoom support, drag support follows a similar pattern. The main drag capability is provided by `d3.behavior.drag` function (line A). D3 drag behavior automatically creates appropriate low-level event listeners to handle drag gestures on the given element then translates low-level events to high-level D3 drag events. Both mouse and touch events are supported:

```javascript
var drag = d3.behavior.drag() // <-A
.on("drag", move);
```

In this recipe we are interested in the `drag` event and it is handled by our `move` function. Similar to the zoom behavior, D3 drag behavior support is event driven, therefore, allowing maximum flexibility in implementation, supporting not only SVG but also the HTML5 canvas. Once defined, the behavior can be attached to any element by calling it on a given selection:

```javascript
svg.selectAll("circle")
.data(data)
.enter().append("circle")
.attr("r", r)
.attr("transform", function (d) {
    return "translate(" + d + ")";
```

---

248
Next, in the `move` function we simply use SVG transformation to move the dragged element to proper location (line D) based on the information conveyed by the drag event (line C):

```javascript
function move(d) {
    var x = d3.event.x, // <-C
        y = d3.event.y;

    if(inBoundaries(x, y))
        d3.select(this)
            .attr("transform", function (d) { // <-D
                return "translate(" + x + ", " + y + ")";
            });
}
```

One additional condition we check here is to calculate the SVG boundaries constraint so the user cannot drag an element outside of the SVG. This is achieved by the following check:

```javascript
function inBoundaries(x, y){
    return (x >= (0 + r) && x <= (width - r))
        && (y >= (0 + r) && y <= (height - r));
}
```

### There's more...

Other than the drag event, D3 drag behavior also supports two other event types. The following list shows all supported drag event types and their attributes:

- **dragstart**: Triggered when a drag gesture starts.
- **drag**: Fired when the element is dragged. `d3.event` will contain `x` and `y` properties representing the current absolute drag coordinates of the element. It will also contain `dx` and `dy` properties representing the element's coordinates relative to its position at the beginning of the gesture.
- **dragend**: Triggered when a drag gesture has finished.

### See also

- *Chapter 2, Be Selective*, for more details on `d3.selection.call` function and selection manipulation
In this chapter we will cover:

- Using gravity and charge
- Generating momentum
- Setting the link constraint
- Using force to assist visualization
- Manipulating force
- Building a force-directed graph

Introduction

Use the force, Luke!

A master's words of wisdom to his apprentice

In this chapter we are going to cover one of the most fascinating aspects of D3: force. Force simulation is one of the most awe-inspiring techniques that you can add to your visualization. Through a number of highly interactive and fully-functional examples, we will help you explore not only the typical application of D3 force (for example, the force-directed graph), but also other essential aspects of force manipulation.

D3 force simulation support was created not as a separate capability, but rather as an additional D3 layout. As we have mentioned in Chapter 9, Lay Them Out, D3 layouts are non-visual data oriented layout management programs designed to be used with different visualization. Force layout was originally created for the purpose of implementing a specific visualization type called force-directed graph. Its implementation uses standard verlet integration based particle motion simulation with support for simple constraints.
Using Force

In other words, D3 implements a numeric method that is capable of loosely simulating Newton's equation of motion on particle level and with simple constraints simulated as links between particles. This kind of layout, of course, was ideal in implementing a force-directed graph; however, we will also discover through recipes in this chapter that force layout is capable of generating many other interesting visualization effects due to its flexibility in custom force manipulation. The application of the techniques introduced in this chapter go even beyond the data visualization realm and has practical applications in many other domains, for example, user interface design. Of course, we will also cover the classical application of force layout: the force-directed graph in this chapter.

Using gravity and charge

In this recipe we will introduce you to the first two fundamental forces: gravity and charge. As we have mentioned before, one objective of force layout's design is to loosely simulate Newton's equation of motion with particles, and one major feature of this simulation is the force of charge. Additionally, force layout also implements pseudo gravity or more accurately a weak geometric constraint typically centered on the SVG that can be leveraged to keep your visualization from escaping the SVG canvas. In the following example we will learn how these two fundamental, and sometimes opposing forces, can be leveraged to generate various effects with a particle system.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

In the following example we will experiment with the force layout gravity and charge settings so you can better understand different opposing forces involved and their interaction:

```javascript
var w = 1280, h = 800,
    force = d3.layout.force()
       .size([w, h])
       .gravity(0)
       .charge(0)
       .friction(0.7);

var svg = d3.select("body")
```
Chapter 11

```javascript
章 11

console.log("svg")
.attr("width", w)
.attr("height", h);

force.on("tick", function () {
  svg.selectAll("circle")
    .attr("cx", function (d) {return d.x;})
    .attr("cy", function (d) {return d.y;});
});

svg.on("mousemove", function () {
  var point = d3.mouse(this),
      node = {x: point[0], y: point[1]}; // <-A

  svg.append("circle")
    .data([node])
    .attr("class", "node")
    .attr("cx", function (d) {return d.x;})
    .attr("cy", function (d) {return d.y;})
    .attr("r", 1e-6)
    .transition()
    .attr("r", 4.5)
    .transition()
    .delay(7000)
    .attr("r", 1e-6)
    .each("end", function () {
      force.nodes().shift(); // <-B
      remove();

      force.nodes().push(node); // <-C
    force.start(); // <-D
  });

  function changeForce(charge, gravity) {
    force.charge(charge).gravity(gravity);
  }
</script>

<div class="control-group">
  <button onclick="changeForce(0, 0)">
    No Force
  </button>
  <button onclick="changeForce(-60, 0)">
    No Force
  </button>
</div>
```
Using Force

```
Mutual Repulsion
</button>
<button onclick="changeForce(60, 0)"
    Mutual Attraction
</button>
<button onclick="changeForce(0, 0.02)"
    Gravity
</button>
<button onclick="changeForce(-30, 0.1)"
    Gravity with Repulsion
</button>
</div>

This recipe generates a force-enabled particle system that is capable of operating in the
modes shown in the following diagram:

<table>
<thead>
<tr>
<th>No Force</th>
<th>Mutual Repulsion</th>
<th>Mutual Attraction</th>
<th>Gravity</th>
<th>Gravity with Repulsion</th>
</tr>
</thead>
</table>

**How it works...**

Before we get our hands dirty with the preceding code example, let's first dig a little bit deeper into the concept of gravity, charge, and friction so we can have an easier time understanding all the magic number settings we will use in this recipe.

**Charge**
Charge is specified to simulate mutual n-body forces among the particles. A negative value results in a mutual node repulsion while a positive value results in a mutual node attraction. The default value for charge is -30. Charge value can also be a function that will be evaluated for each node whenever the force simulation starts.

**Gravity**
Gravity simulation in force layout is not designed to simulate physical gravity, which can be simulated using positive charge. Instead, it is implemented as a weak geometric constraint similar to a virtual spring connecting to each node from the center of the layout. The default gravitational strength is set to 0.1. As the nodes get further away from the center the gravitational strength gets stronger in linear proportion to the distance while near the center of the layout the gravitational strength is almost zero. Hence, gravity will always overcome repulsive charge at some point, therefore, preventing nodes from escaping the layout.
Friction

Friction in D3 force layout does not represent a standard physical coefficient of friction, but it is rather implemented as a velocity decay. At each tick of the simulation particle, velocity is scaled down by a specified friction. Thus a value of 1 corresponds to a frictionless environment while a value of 0 freezes all particles in place since they lose their velocity immediately. Values outside the range of [0, 1] are not recommended since they might destabilize the layout.

Alright, now with the dry definition behind us, let's take a look at how these forces can be leveraged to generate interesting visual effects.

Setting up zero force layout

First, we simply set up force layout with neither gravity nor charge. The force layout can be created using the `d3.layout.force` function:

```javascript
var w = 1280, h = 800,
force = d3.layout.force()
  .size([w,h])
  .gravity(0)
  .charge(0)
  .friction(0.7);
```

Here, we set the size of the layout to the size of our SVG graphic, which is a common approach though not mandatory. In some use cases you might find it useful to have a layout larger or smaller than your SVG. At the same time, we disable both gravity and charge while setting the friction to 0.7. With this setting in place, we then create additional nodes represented as `svg:circle` on SVG whenever the user moves the mouse:

```javascript
svg.on("mousemove", function () {
  var point = d3.mouse(this),
      node = {x: point[0], y: point[1]}; // <-A
  
  svg.append("circle")
  .data([node])
  .attr("class", "node")
  .attr("cx", function (d) {return d.x;})
  .attr("cy", function (d) {return d.y;})
  .attr("r", 1e-6)
  .transition()
  .attr("r", 4.5)
  .transition()
  .delay(7000)
  .attr("r", 1e-6)
  .each("end", function () {
    force.nodes().shift(); // <-B
  })
})
```

```javascript
```

---

255
Node object was created initially on line A with its coordinates set to the current mouse location. Like all other D3 layouts, force layout is not aware and has no visual elements. Therefore, every node we create needs to be added to the layout's nodes array on line C and removed when visual representation of these nodes was removed on line B. On line D we call the `start` function to start force simulation. With zero gravity and charge the layout essentially lets us place a string of nodes with our mouse movement as shown in the following screenshot:

![No Gravity or Charge](image)

**Setting up mutual repulsion**

In the next mode, we will set the charge to a negative value while still keeping gravity to zero in order to generate a mutual repulsive force field:

```javascript
function changeForce(charge, gravity) {
    force.charge(charge).gravity(gravity);
}

changeForce(-60, 0);
```

These lines tell force layout to apply -60 charge on each node and update the node's \(x, y\) coordinate accordingly, based on the simulation result on each tick. However, only doing this is still not enough to move the particles on SVG since the layout has no knowledge of the visual elements. Next, we need to write some code to connect the data that are being manipulated by force layout to our graphical elements. Following is the code to do that:

```javascript
force.on("tick", function () {
    svg.selectAll("circle")
        .remove();

    force.nodes().push(node); // <-C
    force.start(); // <-D
});
```
Here, we register a tick event listener function that updates all circle elements to its new position based on the force layout's calculation. Tick listener is triggered on each tick of the simulation. At each tick we set the \( cx \) and \( cy \) attribute to be the \( x \) and \( y \) values on \( d \). This is because we have already bound the node object as datum to these circle elements, therefore, they already contain the new coordinates calculated by force layout. This effectively establishes force layout's control over all the particles.

Other than tick, force layout also supports some other events:

- **start**: Triggered when simulation starts
- **tick**: Triggered on each tick of the simulation
- **end**: Triggered when simulation ends

This force setting generates the following visual effect:

![Mutual Repulsion](image)

**Setting up mutual attraction**

When we change the charge to a positive value, it generates mutual attraction among the particles:

```javascript
function changeForce(charge, gravity) {
  force.charge(charge).gravity(gravity);
}
changeForce(60, 0);
```
**Using Force**

This generates the following visual effect:

![Mutual Attraction](image1.png)

**Setting up gravity**

When we turn on gravity and turn off charge then it generates a similar effect as the mutual attraction; however, you can notice the linear scaling of gravitational pull as the mouse moves away from the center:

```javascript
function changeForce(charge, gravity) {
    force.charge(charge).gravity(gravity);
}
changeForce(0, 0.02);
```

With gravity alone this recipe generates the following effect:

![Gravity](image2.png)
Using gravity with repulsion

Finally, we can turn on both gravity and mutual repulsion. The result is an equilibrium of forces that keeps all particles somewhat stable neither escaping the layout nor colliding with each other:

```javascript
function changeForce(charge, gravity) {
  force.charge(charge).gravity(gravity);
}
changeForce(-30, 0.1);
```

Here is what this force equilibrium looks like:

![Gravity with Repulsion](image)

See also

- Physical simulation: [http://www.gamasutra.com/resource_guide/20030121/jacobson_pfV.htm](http://www.gamasutra.com/resource_guide/20030121/jacobson_pfV.htm)
- The content of this chapter is inspired by Mike Bostock's brilliant talk on D3 Force: [http://mbostock.github.io/d3/talk/20110921/](http://mbostock.github.io/d3/talk/20110921/)
- *Chapter 10, Interacting with your Visualization*, for more details on how to interact with the mouse in D3
Generating momentum

In our previous recipe we have touched upon force layout node object and its \( \{ x, y \} \) attributes, which determine where a node locates on the layout. In this recipe we will discuss another interesting aspect of physical motion simulation: momentum. D3 force layout has built-in support for momentum simulation which relies on the \( \{ px, py \} \) attributes on the node object. Let’s see how this can be done in the example described in this recipe.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

In this recipe we will modify the previous recipe by first disabling both gravity and charge then giving newly added node some initial velocity. As a result now the faster you move the mouse higher the initial velocity and momentum will be for each node. Here is the code to do that:

```javascript
<script type="text/javascript">
var force = d3.layout.force()
    .gravity(0)
    .charge(0)
    .friction(0.95);

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

force.on("tick", function () {
    // omitted, same as previous recipe
    ...
});

var previousPoint;

svg.on("mousemove", function () {
    var point = d3.mouse(this),
        node = {
            x: point[0],
            y: point[1],
            px: previousPoint ? previousPoint[0] : point[0],
            py: previousPoint ? previousPoint[1] : point[1],
            vx: 0,
            vy: 0
        },
        dx = point[0] - previousPoint[0],
        dy = point[1] - previousPoint[1],
        distance = Math.sqrt(dx ** 2 + dy ** 2),
        angle = Math.atan2(dy, dx),
        inertia = 1.0;

    node.momentum = inertia * node.momentum + [dx * inertia, dy * inertia];
    node.vx = node.momentum[0] / distance;
    node.vy = node.momentum[1] / distance;
    node.momentum = inertia * node.momentum;

    previousPoint = point;
};
```
This recipe generates a particle system with initial directional velocity proportional to the user's mouse movement as shown in the following screenshot:

![](image)

**How it works...**

The overall structure of this recipe is very similar to the previous one. It also generates particles as the user moves the mouse around. Moreover, once the force simulation starts, the particle position is fully controlled by force layout in its `tick` event listener function. However, in this recipe we have turned off both gravity and charge so that we can focus more clearly on momentum alone. We left some friction so the velocity decay making simulation look more realistic. Here is our force layout configuration:

```javascript
var force = d3.layout.force()
  .gravity(0)
  .charge(0)
  .friction(0.95);
```
The major difference in this recipe is that we keep track of not only the current mouse position, but also the previous mouse position. Additionally, whenever the user moves the mouse we generate a node object containing the current location \( \{x, y\} \) as well as the previous location \( \{px, py\} \):

```javascript
var previousPoint;

svg.on("mousemove", function () {
    var point = d3.mouse(this),
        node = {
            x: point[0],
            y: point[1],
            px: previousPoint ? previousPoint[0] : point[0],
        };

    previousPoint = point;
    ...
}
```

Since user mouse location is sampled on fixed interval, the faster the user moves the mouse the further apart these two positions will be. This property plus the directional information gained from these two positions are nicely translated automatically by force layout into initial momentum for each particle we create as we have demonstrated in this recipe.

Besides the \( \{x, y, px, py\} \) attributes we have discussed so far, force layout node object also supports some other useful attributes that we will list here for your reference:

- **index**: Zero-based index of the node within the nodes array.
- **x**: The x-coordinate of the current node position.
- **y**: The y-coordinate of the current node position.
- **px**: The x-coordinate of the previous node position.
- **py**: The y-coordinate of the previous node position.
- **fixed**: A Boolean indicating if the node position is locked.
- **weight**: The node weight; the number of associated links. Links are used to connect nodes in a force layout, which we will cover in depth in the next recipe.

### See also

- The *Interacting with mouse events* recipe in *Chapter 10, Interacting with your Visualization*, for more details on how to interact with the mouse in D3
- D3 Force Layout Nodes API for more details on force layout node attributes
  
Setting the link constraint

So far we have covered some important aspects of the force layout such as gravity, charge, friction, and momentum. In this recipe we will discuss another critical functionality: links. As we have mentioned in the introduction section, D3 force layout implements a scalable simple graph constraint, and in this recipe we will demonstrate how link constraint can be leveraged in conjunction with other forces.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

In this recipe, whenever the user clicks their mouse we will generate a force-directed ring of particles constrained by links between nodes. Here is how it is implemented:

```javascript
<script type="text/javascript">
    var force = d3.layout.force()
        .gravity(0.1)
        .charge(-30)
        .friction(0.95)
        .linkDistance(20)
        .linkStrength(1);

    var duration = 60000; // in milliseconds

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

    force.size([1100, 600])
        .on("tick", function () {
            // omitted, will be discussed in details later
        ...});

    function offset() {
        return Math.random() * 100;
    }

    function createNodes(point) {
        var numberOfNodes = Math.round(Math.random() * 10);
        var nodes = [];
```
for (var i = 0; i < numberOfNodes; ++i) {
    nodes.push({
        x: point[0] + offset(),
        y: point[1] + offset()
    });
}

return nodes;
}

function createLinks(nodes) {
    // omitted, will be discussed in details later ...
}

svg.on("click", function () {
    var point = d3.mouse(this),
        nodes = createNodes(point),
        links = createLinks(nodes);

    nodes.forEach(function (node) {
        svg.append("circle")
            .data([node])
            .attr("class", "node")
            .attr("cx", function (d) {return d.x;})
            .attr("cy", function (d) {return d.y;})
            .attr("r", 1e-6)
            .call(force.drag)
            .transition()
            .attr("r", 7)
            .transition()
            .delay(duration)
            .attr("r", 1e-6)
            .each("end", function () {force.nodes().shift();})
            .remove();
    });

    links.forEach(function (link) {
        // omitted, will be discussed in details later ...
    });

    nodes.forEach(function (n) {force.nodes().push(n);});
    links.forEach(function (l) {force.links().push(l);});

    force.start();
});
This recipe generates force-directed particle rings on a mouse click as shown in the following screenshot:

![Force-Directed Particle Rings](image)

**How it works...**

Link constraint adds another useful dimension to force assisted visualization. In this recipe we set up our force layout with the following parameters:

```javascript
var force = d3.layout.force()
  .gravity(0.1)
  .charge(-30)
  .friction(0.95)
  .linkDistance(20)
  .linkStrength(1);
```

Besides gravity, charge, and friction, this time we have two additional parameters: link distance and link strength. Both parameters are exclusively link related:

- **linkDistance**: Could be a constant or a function; defaults to 20 pixels. Link distances are evaluated when the layout starts, and it is implemented as weak geometric constraints. For each tick of the layout, the distance between each pair of linked nodes is computed and compared to the target distance; the links are then moved towards each other or away from each other.

- **linkStrength**: Could be a constant or a function; defaults to 1. Link strength sets the strength (rigidity) of links with value in the range of $[0, 1]$. Link strength is also evaluated on layout start.
When the user clicks their mouse, a random number of nodes are being created and put under force layout’s control similar to what we have done in the previous recipes. The major addition in this recipe is the link creation and its control logic is shown in the following code snippet:

```javascript
function createLinks(nodes) {
  var links = [];
  for (var i = 0; i < nodes.length; ++i) { // <-A
    if(i == nodes.length - 1)
      links.push({
        source: nodes[i],
        target: nodes[0]
      });
    else
      links.push({
        source: nodes[i],
        target: nodes[i + 1]
      });
  }
  return links;
}

svg.on("click", function () {
  var point = d3.mouse(this),
    nodes = createNodes(point),
    links = createLinks(nodes);
  ...
  links.forEach(function (link) {
    svg.append("line") // <-B
      .data([link])
      .attr("class", "line")
      .attr("x1", function (d) {
        return d.source.x;
      })
      .attr("y1", function (d) {
        return d.source.y;
      })
      .attr("x2", function (d) {
        return d.target.x;
      })
      .attr("y2", function (d) {
        return d.target.y;
      })
  })
...
In the createLinks function, n-1 link objects were created connecting a set of nodes into a ring (for loop on line A). Each link object must have two attributes specified as source and target, telling force layout which pair of nodes are connected by this link object. Once created, we decided to visualize the links in this recipe using svg:line element (line B). We will see in the next recipe that this does not have to always be the case. As a matter of fact, you can use pretty much anything; you can imagine to visualize (including hiding them, but retain the links for layout computation) the links as long as it makes sense for the audience of your visualization. After that we also need to add link objects to force layout's links array (on line C) so they can be put under force layout's control. Finally, we need to translate the positioning data generated by force layout to SVG implementation in the tick function for each link similar to what we did for the nodes:

```javascript
force.size([1100, 600])
.on("tick", function () {
    svg.selectAll("circle")
        .attr("cx", function (d) {return d.x;})
        .attr("cy", function (d) {return d.y;});

    svg.selectAll("line")
        .attr("x1", function (d) {return d.source.x;})
        .attr("y1", function (d) {return d.source.y;})
        .attr("x2", function (d) {return d.target.x;})
        .attr("y2", function (d) {return d.target.y;});
});
```
Using Force

As we can see here, D3 force layout has again done most of the heavy lifting, therefore, all we need to do is simply set \( \{x_1, \ y_1\} \) and \( \{x_2, \ y_2\} \) on the `svg:line` elements in the `tick` function. For reference, the following screenshot is what a link object looks like after it has been manipulated by force layout:

![Link Object](image)

One last additional technique worth mentioning in this recipe is force-enabled dragging. All nodes generated by this recipe are "draggable" and force layout automatically re-computes all forces and constraints as user drags the rings around as shown in the following screenshot:

![Dragging with Force Layout](image)
D3 force layout has dragging built-in, hence, this fancy effect is quite easily achieved by simply calling `force.drag` on the `svg:circle` selection (line D):

```javascript
nodes.forEach(function (node) {
  svg.append("circle")
  .data([node])
  .attr("class", "node")
  ...
  .call(force.drag) // <-D
  .transition()
  ...
  .each("end", function () { force.nodes().shift();})
  .remove();
});
```

See also

- `force.linkDistance()`: https://github.com/mbostock/d3/wiki/Force-Layout#wiki-linkDistance
- `force.linkStrength()`: https://github.com/mbostock/d3/wiki/Force-Layout#wiki-linkStrength

Using force to assist visualization

So far we have learned to use force layout visualizing particles and links similar to how you would use force layout in its classic application, the forced-directed graph. This kind of visualization is what force layout was designed for in the first place. However, this is by no means the only way to utilize force in your visualization. In this recipe we will explore techniques that I call force-assisted visualization. With this technique you can add some randomness and arbitrariness into your visualization by leveraging force.

Getting ready

Open your local copy of the following file in your web browser:

How to do it...

In this recipe we will generate bubbles on user mouse click. The bubbles are made of \texttt{svg:path} elements filled with gradient color. The \texttt{svg:path} elements are not strictly controlled by force layout though they are influenced by force, therefore, giving them the randomness required to simulate a bubble in real-life:

\begin{verbatim}
<svg>
  <defs>
    <radialGradient id="gradient" cx="50%" cy="50%" r="100%"
      fx="50%" fy="50%">
      <stop offset="0%" style="stop-color:blue;stop-opacity:0"/>
      <stop offset="100%" style="stop-color:rgb(255,255,255);stop-opacity:1"/>
    </radialGradient>
  </defs>

  <script type="text/javascript">
    var force = d3.layout.force()
      .gravity(0.1)
      .charge(-30)
      .friction(0.95)
      .linkDistance(20)
      .linkStrength(0.5);
    var duration = 10000;
    var svg = d3.select("svg");
    var line = d3.svg.line()
      .interpolate("basis-closed")
      .x(function(d){return d.x;})
      .y(function(d){return d.y;});
    force.size([svg.node().clientWidth, svg.node().clientHeight])
      .on("tick", function () {
        // omitted, will be discussed in details later
        ...
      });
    function offset() {
      return Math.random() * 100;
    }
  </script>
</svg>
\end{verbatim}
function createNodes(point) {
  // omitted, same as previous recipe
  ...
}

function createLinks(nodes) {
  // omitted, same as previous recipe
  ...
}

svg.on("click", function () {
  // omitted, will be discussed in details later
  ...
});
</script>

This recipe generates force assisted bubbles on user mouse click as shown in the following screenshot:
Using Force

How it works...

This recipe is built on top of what we have done in the previous recipe, therefore, its overall approach is quite similar to the last recipe in which we created force controlled particle rings on user mouse click. The major difference between this recipe and the last one is in this one we decided to use d3.svg.line generator to create the svg:path element that outlines our bubbles instead of using svg:circle and svg:line:

```javascript
var line = d3.svg.line() // <-A
   .interpolate("basis-closed")
   .x(function(d){return d.x;})
   .y(function(d){return d.y;});
...
```

On line A we created a line generator with basis-closed interpolation mode since this gives us the smoothest outline for our bubble. Whenever user clicks the mouse a svg:path element was created connecting all nodes (line C). Additionally, we also fill the bubble with our pre-defined gradient to give it a nice glow (line B). Finally, we also need to implement the force based positioning in the tick function:

```javascript
force.size([svg.node().clientWidth, svg.node().clientHeight])
   .on("tick", function () {
     svg.selectAll("path")
       .attr("d", line);
   });
```
In the `tick` function we simply re-invoke the line generator function to update the `d` attribute for each path thus animating the bubbles using force layout computation.

See also

- SVG Gradients and Patterns: http://www.w3.org/TR/SVG/pservers.html
- The Using line generator recipe in Chapter 7, Getting into Shape, for more information on D3 line generator

**Manipulating force**

So far we have explored many interesting aspects and applications of D3 force layout; however, in all of these prior recipes we simply apply force layout's computation (gravity, charge, friction, and momentum) directly to our visualization. In this recipe we will go one step further to implement custom force manipulation, hence creating our own type of force.

In this recipe we will first generate five sets of colored particles then we assign corresponding colors and categorical force pull to user's touch, hence pulling only the particles that match the color. Since this recipe is a bit complex, I will give an example here: if I touch the visualization with my first finger it will generate a blue circle and pull all blue particles to that circle, while my second touch will generate an orange circle and only pull the orange particles. This type of force manipulation is commonly referred to as categorical multi-foci.

**Getting ready**

Open your local copy of the following file in your web browser:


**How to do it...**

Here is how you can achieve this in code:

```javascript
<script type="text/javascript">
var svg = d3.select("body").append("svg:svg"),
    colors = d3.scale.category10(),
    w = 900,
    h = 600;

svg.attr("width", w).attr("height", h);

var force = d3.layout.force()
    .gravity(0.1)
```

```
Using Force

```javascript
var nodes = force.nodes(),
    centers = [];

for (var i = 0; i < 5; ++i) {
    for (var j = 0; j < 50; ++j) {
        nodes.push({
            x: w / 2 + offset(),
            y: h / 2 + offset(),
            color: colors(i), // <-A
            type: i}); // <-B
    }
}

function offset() {
    return Math.random() * 100;
}

svg.selectAll("circle")
    .data(nodes).enter()
    .append("circle")
    .attr("class", "node")
    .attr("cx", function (d) {return d.x;})
    .attr("cy", function (d) {return d.y;})
    .attr("fill", function (d){return d.color;})
    .attr("r", 1e-6)
    .transition()
    .attr("r", 4.5);

force.on("tick", function(e) {
    // omitted, will discuss in detail
    ...});

d3.select("body")
    .on("touchstart", touch)
    .on("touchend", touch);

function touch() {
    // omitted, will discuss in detail
    ...
}

force.start();
</script>
```
This recipe generates multi-categorical foci on touch as shown in the following screenshot:

![Multi-Categorical Foci on Touch](image)

**How it works...**

The first step of this recipe is to create colored particles and standard force equilibrium between gravity and repulsion. All node objects contain separate color and type ID attributes (line A and B) so they can be easily identified later. Next, we need to create a `<svg:circle>` element on user touch to represent the touch point:

```javascript
function touch() {
  d3.event.preventDefault();

  centers = d3.touches(svg.node());

  var g = svg.selectAll("g.touch")
    .data(centers, function (d) {
      return d.identifier;
    });

  g.enter()
    .append("g")
    .attr("class", "touch")
    .attr("transform", function (d) {
      return "translate(" + d[0] + "," + d[1] + ")";
    });
```

275
Once the touch point is identified, all custom force magic is implemented in the tick function. Now, let's take a look at the tick function:

```javascript
force.on("tick", function(e) {
    var k = e.alpha * .2;
    nodes.forEach(function(node) {
        var center = centers[node.type];
        if(center) {
            node.x += (center[0] - node.x) * k; // <-C
            node.y += (center[1] - node.y) * k; // <-D
        }
    });

    svg.selectAll("circle")
        .attr("cx", function(d) { return d.x; })
        .attr("cy", function(d) { return d.y; });
});
```

The first new concept we encounter here is the alpha parameter. Alpha is an internal cooling parameter used by force layout. Alpha starts with 0.1 and moves towards 0 as layout ticks. In simpler terms the higher the alpha value the more chaotic the forces are and as alpha approaches 0 the layout becomes more stable. In this implementation we leverage the alpha value to make our custom force implementation cool down in synchronous with other built-in forces, since the movements of the particles are calculated with k coefficient (a derivative of alpha) on line C and D moving them closer to the matching touch point.

**See also**

- The [Interacting with a multi-touch device recipe in Chapter 10, Interacting with your Visualization](#), for more information on D3 multi-touch support
Building a force-directed graph

At last, we will show how to implement a force-directed graph, the classic application of D3 force layout. However, we believe with all the techniques and knowledge you have gained so far from this chapter implementing force-directed graph should feel quite straightforward.

Getting ready

Open your local copy of the following file in your web browser:


How to do it...

In this recipe we will visualize the flare data set as a force-directed tree (tree is a special type of graph):

```javascript
<script type="text/javascript">
var w = 1280,
h = 800,
z = d3.scale.category20c();

var force = d3.layout.force()
    .size([w, h]);

var svg = d3.select("body").append("svg")
    .attr("width", w)
    .attr("height", h);

d3.json("/data/flare.json", function(root) {
    var nodes = flatten(root),
        links = d3.layout.tree().links(nodes); // <-B

    force
        .nodes(nodes)
        .links(links)
        .start();

    var link = svg.selectAll("line")
        .data(links)
        .enter().insert("line")

</script>
```
Using Force

```javascript
var node = svg.selectAll("circle.node")
  .data(nodes)
  .enter().append("circle")
  .attr("r", 4.5)
  .style("fill", function(d) {
    return z(d.parent && d.parent.name);
  });

force.on("tick", function(e) {
  link.attr("x1", function(d) { return d.source.x; })
    .attr("y1", function(d) { return d.source.y; })
    .attr("x2", function(d) { return d.target.x; })
    .attr("y2", function(d) { return d.target.y; });

  node.attr("cx", function(d) { return d.x; })
    .attr("cy", function(d) { return d.y; });
});

function flatten(root) {
  var nodes = [];
  function traverse(node, depth) {
    if (node.children) {
      node.children.forEach(function(child) {
        child.parent = node;
        traverse(child, depth + 1);
      });
    }
    node.depth = depth;
    nodes.push(node);
  }
  traverse(root, 1);
  return nodes;
}
</script>
```
This recipe visualizes hierarchical flare data set as a force-directed tree:

![Force-Directed Graph (Tree)](image)

**How it works...**

As we can already see, this recipe is pretty short and a quarter of the code was actually devoted to data processing. This is due to the fact that force-directed graph is what force layout was designed for in the first place. Thus there is really not much to do other than simply apply the layout with correct data structure. First, we flatten the hierarchical data set in flatten function (line A) since this is what force layout expects. Second, we leverage the `d3.layout.tree.links` function to generate proper linkage between tree nodes. The `d3.layout.tree.links` function returns an array of link objects representing links from parent to child for each given node object, in other words, builds the tree structure. Once the data is properly formatted the rest of this recipe applies standard force layout usage with hardly any customization at all.

**See also**

- The *Building a tree* recipe in Chapter 9, *Lay Them Out*, for more information on D3 tree layout
- For more information on force-directed graphs, visit the site: [http://en.wikipedia.org/wiki/Force-directed_graph_drawing](http://en.wikipedia.org/wiki/Force-directed_graph_drawing)
In this chapter we will cover:

- Projecting the US map
- Projecting the world map
- Building a choropleth map

**Introduction**

The ability to project and correlate data points to geographic regions is crucial in many types of visualizations. Geographic visualization is a complex topic with many competing standards emerging and maturing for today's web technology. D3 provides few different ways to visualize geographic and cartographic data. In this chapter we will introduce basic D3 cartographic visualization techniques and how to implement a fully-functional choropleth map (a special purpose colored map) in D3.

**Projecting the US map**

In this recipe we are going to start with projecting the US map using D3 geo API, while also getting familiar with a few different JSON data formats for describing geographic data. Let's first take a look at how geographic data are typically presented and consumed in JavaScript.

**GeoJSON**

The first standard JavaScript geographic data format we are going to touch upon is called GeoJSON. GeoJSON format differs from other GIS standards in that it was written and is maintained by an Internet working group of developers.
GeoJSON is a format for encoding a variety of geographic data structure. A GeoJSON object may represent geometry, a feature, or a collection of features. GeoJSON supports the following geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. Features in GeoJSON contain a geometry object and additional properties, and a feature collection represents a list of features.

Source: http://www.geojson.org/

GeoJSON format is a very popular standard for encoding GIS information and is supported by numerous open source as well as commercial softwares. GeoJSON format uses latitude and longitude points as its coordinates, therefore, it requires any software, including D3, to find the proper projection, scale and translation method in order to visualize its data. The following GeoJSON data describes the state of Alabama in feature coordinates:

```json
{
    "type": "FeatureCollection",
    "features": [{
        "type": "Feature",
        "id": "01",
        "properties": {"name": "AL"},
        "geometry": {
            "type": "Polygon",
            "coordinates": [[
                [-87.359296, 35.00118],
                [-85.606675, 34.984749],
                [-85.431413, 34.124869],
                [-85.184951, 32.859696],
                ...
                [-88.202745, 34.995703],
                [-87.359296, 35.00118]
            ]
        }
    }
}]
```

GeoJSON is currently the de facto GIS information standard for JavaScript project and is well supported by D3; however, before we jump right into D3 geographic visualization using this data format, we want to also introduce you to another emerging technology closely related to GeoJSON.
TopoJSON

TopoJSON is an extension of GeoJSON that encodes topology. Rather than representing geometries discretely, geometries in TopoJSON files are stitched together from shared line segments called arcs. TopoJSON eliminates redundancy, offering much more compact representations of geometry than with GeoJSON; typical TopoJSON files are 80% smaller than their GeoJSON equivalents. In addition, TopoJSON facilitates applications that use topology, such as topology-preserving shape simplification, automatic map coloring, and cartograms.

TopoJSON Wiki https://github.com/mbostock/topojson

TopoJSON was created by D3's author Mike Bostock and designed to overcome some of the drawbacks in GeoJSON while providing a similar feature set when describing geographic information. In most cases concerning cartographic visualization TopoJSON can be a drop-in replacement for GeoJSON with much smaller footprint and better performance. Therefore, in this chapter we will use TopoJSON instead of GeoJSON. Nevertheless, all techniques discussed in this chapter will work perfectly fine with GeoJSON as well. We will not list TopoJSON example here since its arcs based format is not very human readable. However, you can easily convert your shapefiles (popular open source geographic vector format file) into TopoJSON using ogr2ogr command line tool provided by GDAL (http://www.gdal.org/ogr2ogr.html).

Now equipped with this background information let's see how we can make a map in D3.

Getting ready

Open your local copy of the following file in your web browser hosted on your local HTTP server:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter12/usa.html

How to do it...

In this recipe we will load US TopoJSON data and render them using D3 Geo API. Here is the code sample:

```html
<script type="text/javascript">
    var width = 960, height = 500;

    // use default USA Albers projection
    var path = d3.geo.path();

    var svg = d3.select("body").append("svg")
</script>"
This recipe projects US map with Albers USA mode:

US map projected with Albers USA mode
How it works...

As you can see, the code required to project a US map using TopoJSON and D3 is quite short, especially the part concerning map projection. This is because both D3 geographic API and TopoJSON library are built explicitly to make this kind of job as easy as possible for developers.

To make a map, first you need to load the TopoJSON data file (line A). The following screenshot shows what the topology data looks like once loaded:

```
{ "type": "Topology", "transform": Object, "objects": Object, "arcs": Array[10890] }
```

Once the topology data is loaded, all we have to do is to use the TopoJSON library `topojson.feature` function to convert topology arcs into coordinates similar to what GeoJSON format provides as shown in the following screenshot:

```
{ "type": "FeatureCollection", "features": Array[53] }
```

Then `d3.geo.path` will automatically recognize and use the coordinates to generate `svg:path` highlighted in the following code snippet:

```javascript
var path = d3.geo.path();
...
g.selectAll("path")
  .data(topojson.feature(topology, topology.objects.states).features)
  .enter().append("path")
  .attr("d", path);
```
That's it! This is all you need to do to project a map in D3 using TopoJSON. Additionally, we have also attached a zoom handler to the parent `svg:g` element:

```javascript
var g = svg.append('g')
    .call(d3.behavior.zoom()
        .scaleExtents([1, 10])
        .on("zoom", zoom));
```

This allows the user to perform simple geometric zoom on our map.

### See also

- GeoJSON v1.0 specification: [http://www.geojson.org/geojson-spec.html](http://www.geojson.org/geojson-spec.html)
- TopoJSON Wiki: [https://github.com/mbostock/topojson/wiki](https://github.com/mbostock/topojson/wiki)
- More on making map from shapefiles to TopoJSON: [http://bost.ocks.org/mike/map/](http://bost.ocks.org/mike/map/)
- Chapter 3, **Dealing with Data**, for more information on asynchronous data loading
- Chapter 10, **Interacting with your Visualization**, for more information on how to implement zooming
- Mike Bostock’s post on Albers USA projection on which this recipe is based [http://bl.ocks.org/mbostock/4090848](http://bl.ocks.org/mbostock/4090848)

### Projecting the world map

What if our visualization project is not just about US, but rather concerns the whole world? No worries, D3 comes with various built-in projection modes that work well with the world map that we will explore in this recipe.

### Getting ready

Open your local copy of the following file in your web browser hosted on your local HTTP server:


### How to do it...

In this recipe we will project the world map using various different D3 built-in projection modes. Here is the code sample:

```javascript
<script type="text/javascript">
var width = 300, height = 300,
```
translate = [width / 2, height / 2];

var projections = [
  // <-A
  {name: 'azimuthalEqualArea', fn:
    d3.geo.azimuthalEqualArea()
    .scale(50)
    .translate(translate)},
  {name: 'conicEquidistant', fn: d3.geo.conicEquidistant()
    .scale(35)
    .translate(translate)},
  {name: 'equirectangular', fn: d3.geo.equirectangular()
    .scale(50)
    .translate(translate)},
  {name: 'mercator', fn: d3.geo.mercator()
    .scale(50)
    .translate(translate)},
  {name: 'orthographic', fn: d3.geo.orthographic()
    .scale(90)
    .translate(translate)},
  {name: 'stereographic', fn: d3.geo.stereographic()
    .scale(35)
    .translate(translate)}
];

d3.json("/data/world-50m.json", function (error, world) {//<-B
  projections.forEach(function (projection) {
    var path = d3.geo.path() // <-C
      .projection(projection.fn);
    var div = d3.select("body")
      .append("div")
      .attr("class", "map");
    var svg = div
      .append("svg")
      .attr("width", width)
      .attr("height", height);
    svg.append("path") // <-D
      .datum(topojson.feature(world,
        world.objects.land))
      .attr("class", "land")
      .attr("d", path);
This recipe generates world maps with different projection modes as shown in the following screenshot:
Chapter 12

How it works...

In this recipe we first define an array containing six different D3 projection modes on line A. A world topology data was loaded on line B. Similar to the previous recipe we have a d3.geo.path generator defined on line C; however, in this recipe we customized the projection mode for geo path generator calling its projection function. The rest of the recipe is almost identical to what we have done in the previous recipe. The topojson.feature function was used to convert topology data into geographic coordinates so d3.geo.path can generate svg:path required for map rendering (line D and E).

See also

- D3 wiki Geo Projection page (https://github.com/mbostock/d3/wiki/Geo-Projections) for more information on different projection modes as well as on how raw custom projection can be implemented

Building a choropleth map

Choropleth map is a thematic map, in other words, a specially designed map not a general purpose one, which is designed to show measurement of statistical variable on the map using different color shades or patterns; or sometimes referred as geographic heat-map in simpler terms. We have already seen in the previous two recipes that geographic projection in D3 consists of a group of svg:path elements, therefore, they can be manipulated as any other svg elements including coloring. We will explore this feature in geo-projection and implement a Choropleth map in this recipe.

Getting ready

Open your local copy of the following file in your web browser hosted on your local HTTP server:

https://github.com/NickQiZhu/d3-cookbook/blob/master/src/chapter12/choropleth.html

How to do it...

In a choropleth map different geographic regions are colored according to their corresponding variables, in this case based on 2008 unemployment rate in US by county. Now, let's see how to do it in code:

```
<script type="text/javascript">
    var width = 960, height = 500;

    var color = d3.scale.threshold() // <-A
```
Know your Map

```javascript
.domain([.02, .04, .06, .08, .10])
.range(["#f2f0f7", "#dadaeb", "#bcbddc",
"#9e9ac8", "#756bb1", "#54278f"])

var path = d3.geo.path();

var svg = d3.select("body").append("svg")
  .attr("width", width)
  .attr("height", height);

var g = svg.append("g")
...  
d3.json("/data/us.json", function (error, us) {
  d3.tsv("/data/unemployment.tsv", function (error,
    unemployment) {
    var rateById = {};
    unemployment.forEach(function (d) { // <-B
      rateById[d.id] = +d.rate;
    });

    g.append("g")
      .attr("class", "counties")
      .selectAll("path")
      .data(topojson.feature(us,
        us.objects.counties).features)
      .enter().append("path")
      .attr("d", path)
      .style("fill", function (d) {
        return color(rateById[d.id]); // <-C
      });

    g.append("path")
      .datum(topojson.mesh(us, us.objects.states,
        function(a, b) { return a !== b; }))
      .attr("class", "states")
      .attr("d", path);
  });
...  
</script>
```
This recipe generates the following choropleth map:

![Choropleth Map of 2008 Unemployment Rate](image)

**How it works...**

In this recipe we loaded two different data sets: one for the US topology and the other containing unemployment rate by county in 2008. This technique is generally considered as layering and is not necessarily limited to only two layers. The unemployment data are stitched to counties by their ID (line B and C). Region coloring is achieved by using a threshold scale (line A). One last point worth mentioning is the `topojson.mesh` function used to render state borders. `topojson.mesh` is useful for rendering strokes in complicated objects efficiently since it only renders shared edge by multiple features once.

**See also**

- TopoJSON Wiki for more information on mesh function: [https://github.com/mbostock/topojson/wiki/API-Reference#wiki-mesh](https://github.com/mbostock/topojson/wiki/API-Reference#wiki-mesh)
- Mike Bostock's post on choropleth map which this recipe is based on: [http://bl.ocks.org/mbostock/4090848](http://bl.ocks.org/mbostock/4090848)
In this chapter we will cover:

- Getting Jasmine and setting up the test environment
- Test driving your visualization – chart creation
- Test driving your visualization – SVG rendering
- Test driving your visualization – pixel-perfect bar rendering

**Introduction**

Whenever we program as a professional programmer it is always important to test the program we write in order to make sure it functions as designed and produces the expected outcome. D3 data visualization mainly consists of JavaScript programs hence just like any other program we write, data visualization needs to be tested to make sure it represents the underlying data accurately. Obviously, we can perform our validation through visual examination and manual testing, which is always a critical part of the process of building data visualization since visual observation gives us a chance to verify not only the correctness, but also the aesthetics, usability, and many other useful aspects. However, manual visual inspection can be quite subjective, therefore, in this chapter we will focus our effort on automated unit testing. Visualization well covered by unit tests can free the creator from the manual labor of verifying correctness by hand additionally, allowing him/her to focus more on the aesthetics, usability, and other important aspects where it is hard to automate with machine.
Introduction to unit testing

Unit testing is a method in which a smallest unit of the program is tested and verified by another program called the test case. The logic behind unit testing is that at unit level the program is typically simpler and more testable. If we can verify if every unit in the program is correct then putting these correct units together will give us a higher confidence that the integrated program is also correct. Furthermore, since unit tests are typically cheap and fast to execute, a group of unit test cases can be quickly and frequently executed to provide feedback whether our program is performing correctly or not.

Software testing is a complex topic and so far we have only scratched the surface; however, due to limited scope in this chapter, we will have to stop our introduction now and dive into developing unit tests.

For more information on testing please check out the following links:
- Unit test: http://en.wikipedia.org/wiki/Unit_testing
- Test driven development: http://en.wikipedia.org/wiki/Test-driven_development

Getting Jasmine and setting up the test environment

Before we start writing our unit test cases we need to set up an environment where our test cases can be executed to verify our implementation. In this chapter, we will show how this environment and necessary libraries can be set up for a visualization project.

Getting ready

Jasmine (http://pivotal.github.io/jasmine/) is a behavior-driven development (BDD) framework for testing JavaScript code.

BDD is a software development technique that combines Test Driven Development (TDD) with domain driven design.

We chose Jasmine as our testing framework because of its popularity in JavaScript community as well as its nice BDD syntax. You can download the Jasmine library from:

https://github.com/pivotal/jasmine/downloads
Once downloaded you need to unzip it into the `lib` folder. Besides the `lib` folder we also need to create the `src` and `spec` folders for storing source files as well as test cases (in BDD terminology test cases are called specification). See the following screenshot for the folder structure:

![Testing Directory Structure](image)

**How to do it...**

Now, we have Jasmine in our library, next thing to do is to set up an HTML page that will include Jasmine library as well as our source code plus test cases so they can be executed to verify our program. This file is called `SpecRunner.html` in our setup which includes the following code:

```html
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner</title>

  <link rel="stylesheet" type="text/css" href="lib/jasmine-1.3.1/jasmine.css">
  <script type="text/javascript" src="lib/jasmine-1.3.1/jasmine.js"></script>
  <script type="text/javascript" src="lib/jasmine-1.3.1/jasmine-html.js"></script>
  <script type="text/javascript" src="../../lib/d3.js"></script>

  <!-- include source files here... -->
  <script type="text/javascript" src="src/bar_chart.js"></script>
  <!-- include spec files here... -->
  <script type="text/javascript" src="spec/spec_helper.js"></script>
  <script type="text/javascript" src="spec/bar_chart_spec.js"></script>
</head>
```
Test Drive your Visualization

<script type="text/javascript">
(function () {
    var jasmineEnv = jasmine.getEnv();
    jasmineEnv.updateInterval = 1000;
    var htmlReporter = new jasmine.HtmlReporter();
    jasmineEnv.addReporter(htmlReporter);
    jasmineEnv.specFilter = function (spec) {
        return htmlReporter.specFilter(spec);
    };
    var currentWindowOnload = window.onload;
    window.onload = function () {
        if (currentWindowOnload) {
            currentWindowOnload();
        }
        execJasmine();
    };
    function execJasmine() {
        jasmineEnv.execute();
    }
})();
</script>

</head>

How it works...

This code follows standard Jasmine spec runner structure and generates execution report directly into our HTML page. Now, you have a fully functional test environment set up for your visualization development. If you open the SpecRunner.html file with your browser you will see a blank page at this point; however, if you check out our code sample you will see the following report:
See also


### Test driving your visualization – chart creation

With test environment ready, we can move on and develop a simple bar chart very similar to what we have done in the Creating a bar chart recipe in Chapter 8, Chart Them Up, though this time in a test-driven fashion. You can see how the bar chart looks if you open the tdd-bar-chart.html page:
By now we all know very well how to implement a bar chart using D3; however, building a bar chart is not the focus of this recipe. Instead, we want to show how we can build test cases every step of the way and verify automatically that our bar chart implementation is doing what it is supposed to do. The source code of this recipe was built using test driven development method; however, we will not show you every step in the TDD process due to limited scope in this book. Instead, we have grouped multiple steps into three larger sections with different focuses in this chapter and this recipe is the first step we take.

How to do it...

First step we need to take is to make sure our bar chart implementation exists and can receive the data. The starting point of our development could be arbitrary and we decide to drive from this simplest function to set up the skeleton for our object. Here is what the test case looks like:

```javascript
describe('BarChart', function () {
  var div,
      chart,
      data = [
        {x: 0, y: 0},
        {x: 1, y: 3},
        {x: 2, y: 6}
      ];

  beforeEach(function () {
    div = d3.select('body').append('div');
    chart = BarChart(div);
  });

  afterEach(function () {
    div.remove();
  });

  describe('.data', function () {
    it('should allow setting and retrieve chart data', function () {
      expect(chart.data(data).data()).toBe(data);
    });
  });
});
```
How it works...

In this first test case we used a few Jasmine constructs:

- **describe**: This function defines a suite of test cases; within describe a sub-suite can be nested and test cases can be defined.
- **it**: This function defines a test case.
- **beforeEach**: This function defines a pre-execution hook which will execute the given function before the execution of each test case.
- **afterEach**: This function defines a post-execution hook which will execute the given function after the execution of each test case.
- **expect**: This function defines an expectation in your test case which can then be chained with matchers (for example, **toBe** and **toBeEmpty**) to perform assertion in your test case.

In our example we use the **beforeEach** hook to set up a div container for each test case and then remove div after execution in **afterEach** hook to improve the isolation between different test cases. The test case itself is almost trivial; it checks if the bar chart can take data and also return data attribute correctly. At this point if we run our SpecRunner, it will display a red message complaining there is no BarChart object, so let’s create our object and function:

```javascript
function BarChart(p) {
    var that = {};
    var _parent = p, data;
    that.data = function (d) {
        if (!arguments.length) return _data;
        _data = d;
        return that;
    };

    return that;
}
```

Now, if you run SpecRunner.html again it will give you a happy green message showing our only test case is passing.

Test driving your visualization – SVG rendering

Now we have the basic skeleton of our bar chart object created, and we feel that we are ready to try to render something, so in this second iteration we will try to generate the svg:svg element.
Test Drive your Visualization

How to do it...

Rendering the `svg:svg` element should not only simply add the `svg:svg` element to the HTML body, but also translate the width and height setting on our chart object to proper SVG attributes. Here is how we express our expectation in our test cases:

```javascript
describe('.render', function () {
  describe('svg', function () {
    it('should generate svg', function () {
      chart.render();
      expect(svg()).not.toBeEmpty();
    });
    it('should set default svg height and width', function () {
      chart.render();
      expect(svg().attr('width')).toBe('500');
      expect(svg().attr('height')).toBe('350');
    });
    it('should allow changing svg height and width', function () {
      chart.width(200).height(150).render();
      expect(svg().attr('width')).toBe('200');
      expect(svg().attr('height')).toBe('150');
    });
  });
  function svg() {
    return div.select('svg');
  }
});
```

How it works...

At this point, all of these tests will fail since we don't even have the render function; however, it clearly articulates that we expect the render function to generate the `svg:svg` element and setting the `width` and `height` attributes correctly. The second test case also makes sure that if the user does not provide the `height` and `width` attributes we will supply a set of default values. Here is how we will implement the render method to satisfy these expectations:

```javascript
... var _parent = p, _width = 500, _height = 350
_data;
```
that.render = function () {
    var svg = _parent
        .append("svg")
        .attr("height", _height)
        .attr("width", _width);
};

that.width = function (w) {
    if (!arguments.length) return _width;
    _width = w;
    return that;
};

that.height = function (h) {
    if (!arguments.length) return _height;
    _height = h;
    return that;
};
...

At this point our SpecRunner.html is once again all green and happy. However, it's still not doing much since all it does is generate an empty svg element on the page and not even use the data at all.

**Test driving your visualization – pixel-perfect bar rendering**

In this iteration we will finally generate the bars using the data we have. Through our test cases we will make sure all bars are not only rendered but rendered with pixel-perfect precision.

**How to do it...**

Let's see how we test it:

```javascript
describe('chart body', function () {
    it('should create body g', function () {
        chart.render();
        expect(chartBody()).not.toBeEmpty();
    });

    it('should translate to (left, top)', function () {
        chart.render();
```
describe('bars', function () {
  beforeEach(function () {
    chart.data(data).width(100).height(100)
      .x(d3.scale.linear().domain([0, 3]))
      .y(d3.scale.linear().domain([0, 6]))
    .render();
  });

  it('should create 3 svg:rect elements', function () {
    expect(bars().size()).toBe(3);
  });

  it('should calculate bar width automatically', function () {
    bars().each(function () {
      expect(d3.select(this).attr('width')).toBe('18');
    });
  });

  it('should map bar x using x-scale', function () {
    expect(d3.select(bars()[0][0]).attr('x')).toBe('0');
    expect(d3.select(bars()[0][1]).attr('x')).toBe('20');
    expect(d3.select(bars()[0][2]).attr('x')).toBe('40');
  });

  it('should map bar y using y-scale', function () {
    expect(d3.select(bars()[0][0]).attr('y')).toBe('60');
    expect(d3.select(bars()[0][1]).attr('y')).toBe('30');
    expect(d3.select(bars()[0][2]).attr('y')).toBe('0');
  });

  it('should calculate bar height based on y', function () {
    expect(d3.select(bars()[0][0]).
      attr('y')).toBe(60);
  });
});
In the preceding test suite we describe our expectations of having the chart body svg:g element correctly transform and correct number of bars with appropriate attributes (width, x, y, height) set. The implementation is actually going to be shorter than our test case which is quite common in well tested implementation:

...  
var _parent = p, _width = 500, _height = 350, _margins = {top: 10, left: 30, right: 10, bottom: 30}, _data, _x = d3.scale.linear(), _y = d3.scale.linear(); 

that.render = function () {
  var svg = _parent
    .append("svg")
    .attr("height", _height)
    .attr("width", _width);

  var body = svg.append("g")
    .attr("class", 'body')
    .attr("transform", "translate( + _margins.left + ","
      + _margins.top + ")")

  if (_data) {
    _x.range([0, quadrantWidth()]);
    _y.range([quadrantHeight(), 0]);
body.selectAll('rect.bar')
  .data(_data).enter()
  .append('rect')
  .attr("class", 'bar')
  .attr("width", function () {
    return quadrantWidth() / _data.length - BAR_PADDING;
  })
  .attr("x", function (d) {return _x(d.x); })
  .attr("y", function (d) {return _y(d.y); })
  .attr("height", function (d) {
    return _height - _margins.bottom - _y(d.y);
  });

I think you are getting the picture and now you can repeat this cycle over and over to drive your implementation. D3 visualization is built on HTML and SVG and both are simple mark-up languages that can be verified easily. Well thought-out test suite can make sure your visualization is pixel-perfect even sub-pixel perfect.

See also

- Test driven development: http://en.wikipedia.org/wiki/Test-driven_development
Building Interactive Analytics in Minutes

In this appendix we will cover:

- The crossfilter.js library
- Dimensional charting - dc.js

Introduction

Congratulations! You have finished an entire book on data visualization with D3. Together we have explored various topics and techniques. At this point you will probably agree that building interactive, accurate, and aesthetically appealing data visualization is not a trivial matter even with the help of a powerful library like D3. It typically takes days or even weeks to finish a professional data visualization project even without counting the effort usually required on the backend. What if you need to build an interactive analytics quickly, or a proof-of-concept before a full-fledged visualization project can be commenced, and you need to do that not in weeks or days, but minutes. In this appendix we will introduce you to two JavaScript libraries that allow you to do that: building quick in-browser interactive multidimensional data analytics in minutes.
The crossfilter.js library

Crossfilter is also a library created by D3's author Mike Bostock, initially used to power analytics for Square Register.

Crossfilter is a JavaScript library for exploring large multivariate datasets in browser. Crossfilter supports extremely fast (<30ms) interaction with coordinated views, even with datasets containing a million or more records.

-Crossfilter Wiki (August 2013)

In other words, Crossfilter is a library that you can use to generate data dimensions on large and typically flat multivariate datasets. So what is a data dimension? A data dimension can be considered as a type of data grouping or categorization while each dimensional data element is a categorical variable. Since this is still a pretty abstract concept, let's take a look at the following JSON dataset and see how it can be transformed into dimensional dataset using Crossfilter. Assume that we have the following flat dataset in JSON describing payment transactions in a bar:

```javascript
[{
  "date": "2011-11-14T01:17:54Z", "quantity": 2, "total": 190,
  "tip": 100, "type": "tab"},
{
  "date": "2011-11-14T02:20:19Z", "quantity": 2, "total": 190,
  "tip": 100, "type": "tab"},
{
  "date": "2011-11-14T02:28:54Z", "quantity": 1, "total": 300,
  "tip": 200, "type": "visa"},
...
]
```

Sample dataset borrowed from Crossfilter Wiki: https://github.com/square/crossfilter/wiki/API-Reference.

How many dimensions do we see here in this sample dataset? The answer is: it has as many dimensions as the number of different ways that you can categorize the data. For example, since this data is about customer payment, which is observation on time series, obviously the "date" is a dimension. Secondly, the payment type is naturally a way to categorize data; therefore, "type" is also a dimension. The next dimension is bit tricky since technically we can model any of the field in the dataset as dimension or its derivatives; however, we don't want to make anything as a dimension which does not help us slice the data more efficiently or provide more insight into what the data is trying to say. The total and tip fields have very high cardinality, which usually is an indicator for poor dimension (though tip/total, that is, tip in percentage could be an interesting dimension); however, the "quantity" field is likely to have a relatively small cardinality assuming people don't buy thousands of drinks in this bar, therefore, we choose to use quantity as our third dimension. Now, here is what the dimensional logical model looks like:  

```
```
These dimensions allow us to look at the data from a different angle, and if combined will allow us to ask some pretty interesting questions, for example:

- Are customers who pay by tab more likely to buy in larger quantity?
- Are customers more likely to buy larger quantity on Friday night?
- Are customers more likely to tip when using tab versus cash?

Now, you can see why dimensional dataset is such a powerful idea. Essentially, each dimension gives you a different lens to view your data, and when combined, they can quickly turn raw data into knowledge. A good analyst can quickly use this kind of tool to formulate a hypothesis, hence gaining knowledge from data.

**How to do it...**

Now, we understand why we would want to establish dimensions with our dataset; let's see how this can be done using Crossfilter:

```javascript
var timeFormat = d3.time.format.iso;
var data = crossfilter(json); // <-A

var hours = data.dimension(function(d){
    return d3.time.hour(timeFormat.parse(d.date)); // <-B
});
```
Building Interactive Analytics in Minutes

```javascript
var totalByHour = hours.group().reduceSum(function(d){
  return d.total;
});

var types = data.dimension(function(d){return d.type;});
var transactionByType = types.group().reduceCount();

var quantities = data.dimension(function(d){return d.quantity;});
var salesByQuantity = quantities.group().reduceCount();
```

**How it works...**

As shown in the preceding section, creating dimensions and groups are quite straight-forward in Crossfilter. First step before we can create anything is to feed our JSON dataset, loaded using D3, through Crossfilter by calling the `crossfilter` function (line A). Once that's done, you can create your dimension by calling the `dimension` function and pass in an accessor function that will retrieve the data element that can be used to define the dimension. In the case for `type` we will simply pass in `function(d){return d.type;}`. You can also perform data formatting or other task in dimension function (for example, date formatting on line B). After creating the dimensions, we can perform the categorization or grouping by using the dimension, so `totalByHour` is a grouping that sums up total amount of the sale for each hour, while `salesByQuantity` is a grouping of counting the number of transactions by quantity. To better understand how group works, we will take a look at what the group object looks like. If you invoke the `all` function on the `transactionsByType` group you will get the following objects back:

```javascript
[ ▼ Object 3 ▼ , ▼ Object 3 ▼ , ▼ Object 3 ▼ ]
  key: "cash"  key: "tab"  key: "visa"
  value: 4     value: 16   value: 5
  ▼ proto__ : Object  ▼ proto__ : Object  ▼ proto__ : Object
```

We can clearly see that `transactionByType` group is essentially a grouping of the data element by its type while counting the total number of data elements within each group since we had called `reduceCount` function when creating the group.

The following are the description for functions we used in this example:

- `crossfilter`: Creates a new crossfilter with given records if specified. Records can be any array of objects or primitives.
- `dimension`: Creates a new dimension using the given value accessor function. The function must return naturally-ordered values, that is, values that behave correctly with respect to JavaScript's `<`, `<=`, `>=`, and `>` operators. This typically means primitives: Booleans, numbers, or strings.
Appendix

- **dimension.group**: Creates a new grouping for the given dimension, based on the given `groupValue` function, which takes a dimension value as input and returns the corresponding rounded value.

- **group.all**: Returns all groups, in ascending natural order by key.

- **group.reduceCount**: A shortcut function to count the records; returns this group.

- **group.reduceSum**: A shortcut function to sum records using the specified value accessor function.

At this point we have everything we want to analyze. Now, let's see how this can be done in minutes instead of hours or days.

### There's more...

We have only touched a very limited number of Crossfilter functions. Crossfilter provides a lot more capability when it comes to how dimension and group can be created; for more information please check out its API reference: [https://github.com/square/crossfilter/wiki/API-Reference](https://github.com/square/crossfilter/wiki/API-Reference).

### See also


### Dimensional charting – dc.js

Visualizing Crossfilter dimensions and groups is precisely the reason why dc.js was created. This handy JavaScript library was created by your humble author and is designed to allow you to visualize Crossfilter dimensional dataset easily and quickly.

### Getting ready

Open your local copy of the following file as reference:

How to do it...

In this example we will create three charts:

- A line chart for visualizing total amount of transaction on time series
- A pie chart to visualize number of transactions by payment type
- A bar chart showing number of sales by purchase quantity

Here is what the code looks like:

```html
<div id="area-chart"></div>
<div id="donut-chart"></div>
<div id="bar-chart"></div>

... 

dc.lineChart("#area-chart")
  .width(500)
  .height(250)
  .dimension(hours)
  .group(totalByHour)
  .x(d3.time.scale().domain([timeFormat.parse("2011-11-14T01:17:54Z"), timeFormat.parse("2011-11-14T18:09:52Z")])
    .elasticY(true)
    .xUnits(d3.time.hours)
    .renderArea(true)
    .xAxis().ticks(5);

dc.pieChart("#donut-chart")
  .width(250)
  .height(250)
  .radius(125)
  .innerRadius(50)
  .dimension(types)
  .group(transactionByType);

dc.barChart("#bar-chart")
  .width(500)
  .height(250)
  .dimension(quantities)
  .group(salesByQuantity)
  .x(d3.scale.linear().domain([0, 7]))
```
This generates a group of coordinated interactive charts:

![Interactive dc.js charts](image1)

When you click or drag your mouse across these charts you will see the underlying Crossfilter dimensions being filtered accordingly on all charts:

![Filtered dc.js charts](image2)

**How it works...**

As we have seen through this example, dc.js is designed to generate standard chart-based visualization on top of Crossfilter. Each dc.js chart is designed to be interactive so user can apply dimensional filter by simply interacting with the chart. dc.js is built entirely on D3, therefore, its API is very D3-like and I am sure with the knowledge you have gained from this book you will feel quite at home when using dc.js. Charts are usually created in the following steps.

1. First step creates a chart object by calling one of the chart creation functions while passing in a D3 selection for its anchor element, which in our example is the `div` element to host the chart:
   ```javascript
   <div id="area-chart"></div>
   ...
   dc.lineChart("#area-chart")
   ```
2. Then we set the width, height, dimension, and group for each chart:

```javascript
cart.width(500)
    .height(250)
    .dimension(hours)
    .group(totalByHour)
```

For coordinate charts rendered on a Cartesian plane you also need to set the x and y scale:

```javascript
cart.x(d3.time.scale().domain([timeFormat.parse("2011-11-14T01:17:54Z"), timeFormat.parse("2011-11-14T18:09:52Z")])).elasticY(true)
```

In this first case, we explicitly set the x axis scale while letting the chart automatically calculate the y-scale for us. While in the next case we set both x and y scale explicitly.

```javascript
cart.x(d3.scale.linear().domain([0, 7]))
    .y(d3.scale.linear().domain([0, 12]))
```

There's more...

Different charts have different functions for customizing their look-and-feel and you can see the complete API reference at https://github.com/NickQiZhu/dc.js/wiki/API.

Leveraging crossfilter.js and dc.js allows you to build sophisticated data analytics dashboard fairly quickly. The following is a demo dashboard for analyzing the NASDAQ 100 Index for the last 20 years http://nickqizhu.github.io/dc.js/:
At the time of writing this book, dc.js supports the following chart types:

- Bar chart (stackable)
- Line chart (stackable)
- Area chart (stackable)
- Pie chart
- Bubble chart
- Composite chart
- Choropleth map
- Bubble overlay chart

For more information on the dc.js library please check out our Wiki page at https://github.com/NickQiZhu/dc.js/wiki.
See also

The following are some other useful D3 based reusable charting libraries. Although, unlike dc.js they are not designed to work with Crossfilter natively nevertheless they tend to be richer and more flexible when tackling general visualization challenges:

- NVD3: http://nvd3.org/
- Rickshaw: http://code.shutterstock.com/rickshaw/
A
advantages, SVG
   adoption 148
   interoperability 148
   lightweight 148
   open standard 148
   readability 148
   scalability 148
   vector 148
Aight
   about 10
   URL 10
animation 119
arbitrary 64
arc generator
   about 169
   using 170-173
arc transition
   about 173
   implementing 173-176
area chart
   about 188
   creating 188-192
area generator
   about 163
   using 163-166
area interpolation
   using 167-169
array
   binding, as data 43-47
arrays
   working with 54-58
axes
   about 181
   rescaling, dynamically 113-117

Axis component 101

B
Backbone.js 13
bar chart
   about 199
   creating 200, 202, 299-301
basic axes
   working with 101-106
behavior-driven development (BDD) 296
BioVisualize
   about 21
   URL 21
bubble chart
   about 196
   creating 196-199

C
cartographic visualization techniques 283
charge
   about 254, 256
   using 254, 256
chart attributes 183
chart body frame rendering 185
chart object 183
charts
   about 179
   area chart 188
   bar chart 199
   bubble chart 196
   creating 312, 313
   line chart 181
   scatter plot 192
child combinator 33
choropleth map
about 291
building 291-293
click event 236, 239
clipping
URL, for info 186
closepath command 151
cognition amplifier 40
cognitive magnification 235
colors
interpolating 88-91
combinators 32
comparator function 64
compound objects
interpolating 91-94
coordinate translation 181
countup function
listening to 145
createSvg function 104
Crossfilter
about 308
working 310
crossfilter.js library 308
CSS 147
CSS3 selector
basics 24, 25
CSS selector
used, for selecting single element 25-27
Cubic Bézier 151
custom interpolator
implementing 94-99, 142, 143

D
D3
about 7
code, searching 21
code, sharing 21
help, obtaining 22
URLs 8
D3 API
about 22
URL 22
D3 chart convention
about 180
axes 181
coordinate translation 181
margins 181
URL 180
D3 development environment
setting up 8, 9
source code, obtaining 10, 11
D3 gallery
about 21
URL 21
D3 Google group
about 22
URL 22
D3.js
about 7
URL, for downloading 9
D3.js, on Stack Overflow
URL 22
D3 layout
about 205
properties 205
d3.layout.tree 230
D3 plugins
about 21
URL 21
d3.selectAll function 28
d3.select command 26
D3-style JavaScript
about 15
function chaining 20
functions, are objects 17
static variable scoping 19
variable-parameter function 20
working 17
D3 subselection 34
d3.svg.diagonal generator 230
d3.svg.line function 155
D3 SVG shape generators 151
d3.time.format patterns 80
D3 transition 120
D3 tutorials page
about 21
URL 21
d3.v3.js file 9
d3.v3.min.js file 9
data
about 39
array, binding as 43-47
functions, binding as 51-53
loading, from server 64-67
object literals, binding as 47-51
Data-Driven Documents. See D3
data-driven filtering 58, 61
data-driven sorting 61-64
data() function 53
data visualization 8, 39, 235
dbclick event 239
delay 146
dependencies field 13
descendant combinator 33
devDependencies field 13
dimensional charting 311
divergingScale function 90
domain 72
Domain Specific Language (DSL) 34
drag 248
drag behavior
  implementing 248-251
dragend event 251
drag event 251
drag event types
  drag 251
dragend 251
dragstart 251
dragstart event 251
DRY principle 137
duration() function 122
dynamic modifier function 46
E
  ease
    using 128-132
eased tweening 136
ease() function 131
ease mode modifiers 131
easing 128
Elliptical curve 151
enclosure diagram
  about 230
  building 230-234
enter() function 44
enter-update-exit pattern 40-42
exit() function 47, 128
expanded area chart 216
F
  falsy tests 61
  Flare site
    URL 219
force
  about 253
  manipulating 275-278
  used, for assisting visualization 271-275
force-directed graph
  building 279, 281
force-directed graph 253
force layout 253
force simulation 253
friction 257
function chaining 20, 30, 34, 35, 36
functions
  about 72
  binding, as data 51-53
G
gеographic visualization 283
GeoJSON 283, 284
Git 11
GitHub
  about 10
  URL 10
GitHub, for Mac
  URL 11
GitHub, for Windows
  URL 11
gravity
  about 254, 256
  setting up 260
  using 254, 256
  using, with replusion 261
grid lines
  drawing 109-113
H
  hierarchical data 217
HTML 147
http-server module
  about 15
  installing 15
human interaction 235
I
idempotent 54
identity function 75
images 119
imperative method 40
information 39
interpolate() function 85, 98, 169
Interpolation-based animation 120
interpolation modes
basis 159
basis-closed 159
basis-open 159
bundle 159
cardinal 159
cardinal-closed 159
cardinal-open 159
linear 159
linear-closed 159
monotone 159
step-after 159
step-before 159
interpolator 85

J
Jasmine
about 296
obtaining 296
Jasmine constructs
afterEach function 301
beforeEach function 301
describe function 301
expect function 301
it function 301
Jasmine Reference Document
URL 299
JavaScript 147
jQuery 13, 23
JS Bin
about 22
URL 22
JS Fiddle
about 22
URL 22
K
key frames 120
L
Layered Area Chart 188
linear easing 128
linear interpolation mode 156
linear scale 75, 76
line chart
about 181
creating 182-188
line generator
about 152
using 152-155
line interpolation
using 156-159
line tension
modifying 159-163
lineto command 151
link constraint
setting 265-271
linkDistance parameter 267
linkStrength parameter 267
local HTTP server
setting up 14
log scale 77
low-level D3 timer function
working with 144-146
M
margins 181
mark 146
masking
URL, for info 186
mathematical functions 72
mental model alignment 235
metaphor 235
momentum
generating 262-264
mouse 236
mousedown event 239
mouseenter event 239
mouse events
   click 239
   dbclick 239
   interacting with 236-238
   mousedown 239
   mouseenter 239
   mouseleave 239
  mousemove 239
  mouseout 239
   mouseover 239
   mouseup 239
   mouseleave event 239
   mousemove event 236, 238, 239
   mouseout event 239
   mouseover event 239
   mouseup event 239
   moveto command 151
   multiple elements
      animating 123-128
      selecting 28, 29
   multi-touch device
      interacting with 240-243
   mutual attraction
      setting up 259
   mutual repulsion
      setting up 258, 259

N

Node.js
   about 10
   URL, for downloading 12
   Node.js HTTP Server 15
   Node Packaged Modules (NPM) 11
   NPM-based development environment
      local HTTP server, setting up 14
      Node.js HTTP Server 15
      Python Simple HTTP Server 14
      setting up 12-14
   npm command 12
   npm install command 13, 15
   NPM package JSON file documentation
      URL 13
   NVD3
      URL 203

O

   object-identity function 126
   object literals
      binding, as data 47-51
   one-to-one functions 73
   onto functions 73
   ordinal scale
      using 81-84

P

   package.json file 13
   panning
      about 244
      implementing 245-248
   physical simulation
      URL 261
   picture 119
   pie chart
      about 206
      building 206-210
   pie layout 206
   pixelation 148
   pixel-perfect bar rendering 303-306
   poly-linear scale 90
   power scale 76
   Protovis 8
   pseudo-classical pattern 19
   push() function 47
   Python Simple HTTP Server 14

Q

   Quadratic Bézier curve 151
   quadric easing 131
   quantitative scales
      linear scale 75, 76
      log scale 77
      power scale 76
      using 73-77

R

   range 72
   rangeRound() function 76
raw selection
manipulating 36-38

Regex pattern 88
remove() function 47, 117, 128
render axes 186
renderAxis function 105
renderBars function 202
renderBody function 186, 228
renderBubbles function 198
renderCircle function 233
render data series 187
renderDots function 159, 194
render() function 18, 47
renderLabels function 229
renderLinks function 229
renderNodes function 227
replusion
gravity, using with 261

Rickshaw
URL 203

S
Scalable Vector Graphics. See SVG
scale() function 105
scales 72
scatter plot chart
about 192
creating 192, 194
selection
about 23
iterating through 29-31
selection.append(name) function 31
selection.attr function 26
selection.classed function 26
selection.data(data).enter() function 42
selection.data(data).exit function 42
selection.data(data) function 41
selection.each(function) function 30
selection.exit function 42
selection.filter function 61
selection.html function 27
selection.sort function 64
selection.style function 27
selection.text function 27
selector 23

server
data, loading from 64-67
simple shapes
creating 149, 150
single element
animating 121, 122
selecting, CSS selector used 25-27

Sizzle
URL 25
Sizzle selector engine
about 10
URL 10
software testing 296
sort() function 64, 209
stacked area chart
about 211
building 211-215
Stack Overflow 22
stateful visualization 210
static variable scoping 19
streamgraph 216
string
interpolating 84-88
subselection
performing 31-34
SVG
about 147
advantages 148, 149
SVG coordinate system 150
SVG rendering 301, 302
SVG structures 101

T
Test Driven Development (TDD) 296
test environment
setting up 297, 298
tickFormat function 108
tick function 275
tickPadding function 108
ticks
customizing 107, 108
ticks function 107, 108
ticksSubdivide function 108
time scale
using 78-80
TopoJSON 285

touchcancel event 244
touchend event 244
touch events 236
touch event types
  about 244
touchcancel 244
touchend 244
touchmove 244
touchstart 244
touchmove event 244
touchstart event 244
transitional events
  listening to 140, 141
transition chaining
  about 136
  using 136, 137
transition filter
  using 138, 139
tree
  about 224
  building 224-229
treemap
  about 217
  building 218-223
  URL 217
truthy tests 61
Tween 132
tweening
  using 133-135

U

Underscore.js 13
unit testing 296
US map
  projecting 283-288

V

variable-parameter function 20
vector 148
verlet integration 253
Verlet integration
  URL 261
visualization
  assisting, force used 271-275

W

W3C 23
W3C level-3 selector API
  URL, for documentation 25
W3C selector API
  limitations 23
world map
  projecting 288-291
World Wide Web Consortium (W3C) 147

Z

Zepto.js 13, 23
zero force layout
  setting up 257, 258
zooming
  about 244
  implementing 245-248
Thank you for buying
Data Visualization with D3.js Cookbook

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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

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, 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.
Data Visualization: a successful design process
A structured design approach to equip you with the knowledge of how to successfully accomplish any data visualization challenge efficiently and effectively

1. A portable, versatile, and flexible data visualization design approach that will help you navigate the complex path towards success
2. Explains the many different reasons for creating visualizations and identifies the key parameters which lead to very different design options
3. Thorough explanation of the many visual variables and visualization taxonomy to provide you with a menu of creative options

Tableau Data Visualization Cookbook
Over 70 recipes for creating visual stories with your data using Tableau

1. Quickly create impressive and effective graphics which would usually take hours in other tools
2. Lots of illustrations to keep you on track
3. Includes examples that apply to a general audience

Please check www.PacktPub.com for information on our titles
**HTML5 Graphing and Data Visualization Cookbook**

Learn how to create interactive HTML5 charts and graphs with canvas, JavaScript, and open source tools

1. Build interactive visualizations of data from scratch with integrated animations and events
2. Draw with canvas and other html5 elements that improve your ability to draw directly in the browser
3. Work and improve existing 3rd party charting solutions such as Google Maps

---

**Google Visualization API Essentials**

Make sense of your data: make it visual with the Google Visualization API

1. Wrangle all sorts of data into a visual format, without being an expert programmer
2. Visualize new or existing spreadsheet data through charts, graphs, and maps
3. Full of diagrams, core concept explanations, best practice tips, and links to working book examples

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles