Andy Kirk, Simon Timms,
Ændrew Rininsland, Swizec Teller

Data Visualization: Representing Information on Modern Web

Unleash the power of data by creating interactive, engaging, and compelling visualizations for the web

Packt

Data Visualization: Representing Information on Modern Web

Table of Contents

Data Visualization: Representing Information on Modern Web
Data Visualization: Representing Information on Modern Web
Credits
Preface
- What this learning path covers
- What you need for this learning path
- Who this learning path is for
- Reader feedback
- Customer support
  - Downloading the example code
  - Errata
  - Piracy
  - Questions

1. Module 1
  - 1. The Context of Data Visualization
    - Exploiting the digital age
    - Visualization as a discovery tool
    - The bedrock of visualization knowledge
    - Defining data visualization
    - Visualization skills for the masses
    - The data visualization methodology
    - Visualization design objectives
      - Strive for form and function
      - Justifying the selection of everything we do
      - Creating accessibility through intuitive design
      - Never deceive the receiver
    - Summary
  - 2. Setting the Purpose and Identifying Key Factors
    - Clarifying the purpose of your project
      - The reason for existing
      - The intended effect
    - Establishing intent – the visualization's function
      - When the function is to explain
      - When the function is to explore
      - When the function is to exhibit data
    - Establishing intent – the visualization's tone
      - Pragmatic and analytical
      - Emotive and abstract
    - Key factors surrounding a visualization project

The "eight hats" of data visualization design
The initiator
The data scientist
The journalist
The computer scientist
The designer
The cognitive scientist
The communicator
The project manager
Summary
3. Demonstrating Editorial Focus and Learning About Your Data
The importance of editorial focus
Preparing and familiarizing yourself with your data
Refining your editorial focus
Using visual analysis to find stories
An example of finding and telling stories
Summary
4. Conceiving and Reasoning Visualization Design Options
Data visualization design is all about choices
Some helpful tips
The visualization anatomy – data representation
Choosing the correct visualization method
Considering the physical properties of our data
Determining the degree of accuracy in interpretation
Creating an appropriate design metaphor
Choosing the final solution
The visualization anatomy – data presentation
The use of color
To represent data
To bring the data layer to the fore
To conform to design requirements
Creating interactivity
Annotation
Arrangement
Summary
5. Taxonomy of Data Visualization Methods
Data visualization methods
Choosing the appropriate chart type
Comparing categories
Dot plot
Bar chart (or column chart)
Floating bar (or Gantt chart)
Pixelated bar chart

- Histogram
- Slopegraph (or bumps chart or table chart)
- Radial chart
- Glyph chart
- Sankey diagram
- Area size chart
- Small multiples (or trellis chart)
- Word cloud

Assessing hierarchies and part-to-whole relationships

- Pie chart
- Stacked bar chart (or stacked column chart)
- Square pie (or unit chart or waffle chart)
- Tree map
- Circle packing diagram
- Bubble hierarchy
- Tree hierarchy

Showing changes over time

- Line chart
- Sparklines
- Area chart
- Horizon chart
- Stacked area chart
- Stream graph
- Candlestick chart (or box and whiskers plot, OHLC chart)
- Barcode chart
- Flow map

Plotting connections and relationships

- Scatter plot
- Bubble plot
- Scatter plot matrix
- Heatmap (or matrix chart)
- Parallel sets (or parallel coordinates)
- Radial network (or chord diagram)
- Network diagram (or force-directed/node-link network)

Mapping geo-spatial data

- Choropleth map
- Dot plot map
- Bubble plot map
- Isarithmic map (or contour map or topological map)
- Particle flow map
- Cartogram
- Dorling cartogram
- Network connection map

- Summary
- 6. Constructing and Evaluating Your Design Solution
  - For constructing visualizations, technology matters
    - Visualization software, applications, and programs
    - Charting and statistical analysis tools
    - Programming environments
    - Tools for mapping
      - Other specialist tools
  - The construction process
  - Approaching the finishing line
  - Post-launch evaluation
  - Developing your capabilities
    - Practice, practice, practice!
    - Evaluating the work of others
    - Publishing and sharing your output
    - Immerse yourself into learning about the field
  - Summary
2. Module 2
  - 1. Visualizing Data
    - There's a lot of data out there
    - Getting excited about data
      - Data beyond Excel
      - Social media data
        - Why should I care?
    - HTML visualizations
    - Summary
  - 2. JavaScript and HTML5 for Visualizations
    - Canvas
    - Scalable Vector Graphics
      - Which one to use?
    - Summary
  - 3. OAuth
    - Authentication versus authorization
    - The OAuth protocol
      - OAuth versions
    - Summary
  - 4. JavaScript for Visualization
    - Raphaël
    - d3.js
      - Custom color scales
      - Labels and axes
    - Summary
  - 5. Twitter

- Getting access to the APIs
- Setting up a server
- OAuth
- Visualization
  - Server side
  - Client side
- Summary

6. Stack Overflow
   - Authenticating
   - Creating a visualization
   - Filters
   - Summary
7. Facebook
   - Creating an app
   - Using the API
   - Retrieving data
   - Visualizing
   - Summary
8. Google+
   - Creating an app
   - Retrieving data
   - Visualization
   - Summary

3. Module 3
   1. Getting Started with D3, ES2016, and Node.js
      - What is D3.js?
      - What's ES2016?
        - Getting started with Node and Git on the command line
        - A quick Chrome Developer Tools primer
        - The obligatory bar chart example
      - Summary
   2. A Primer on DOM, SVG, and CSS
      - DOM
        - Manipulating the DOM with D3
        - Selections
        - Let's make a table!
      - What exactly did we do here?
        - Selections example
        - Manipulating content
        - Joining data to selections
        - An HTML visualization example
      - Scalable Vector Graphics
        - Drawing with SVG

- Manually adding elements and shapes
  - Text
  - Shapes
  - Transformations
- Using paths
  - Line
  - Area
  - Arc
  - Symbol
  - Chord
  - Diagonal
- Axes
- CSS
- Colors
- Summary

3. Making Data Useful
- Thinking about data functionally
  - Built-in array functions
  - Data functions of D3
- Loading data
  - The core
  - Convenience functions
- Scales
  - Ordinal scales
  - Quantitative scales
  - Continuous range scales
  - Discrete range scales
  - Time
    - Formatting
    - Time arithmetic
- Geography
  - Getting geodata
  - Drawing geographically
  - Using geography as a base
- Summary

4. Defining the User Experience – Animation and Interaction
- Animation
- Animation with transitions
  - Interpolators
  - Easing
  - Timers
  - Animation with CSS transitions
- Interacting with the user

- Basic interaction
- Behaviors
  - Drag
  - Zoom
  - Brushes
- Summary

5. Layouts – D3's Black Magic
- What are layouts and why should you care?
  - Built-in layouts
  - The dataset
- Normal layouts
- Using the histogram layout
- Baking a fresh 'n' delicious pie chart
  - Labeling your pie chart
- Showing popularity through time with stack
  - Adding tooltips to our streamgraph
- Highlighting connections with chord
  - Drawing with force
- Hierarchical layouts
- Drawing a tree
- Showing clusters
  - Partitioning a pie
  - Packing it in
  - Subdividing with treemap
- Summary

6. D3 on the Server with Node.js
- Readying the environment
- All aboard the Express train to Server Town!
- Proximity detection and the Voronoi geom
- Rendering in Canvas on the server
- Deploying to Heroku
- Summary

7. Designing Good Data Visualizations
- Clarity, honesty, and sense of purpose
- Helping your audience understand scale
- Using color effectively
- Understanding your audience (or "trying not to forget about mobile")
  - Some principles for designing for mobile and desktop
    - Columns are for desktops, rows are for mobile
    - Be sparing with animations on mobile
    - Realize similar UI elements react differently between platforms
    - Avoid "mystery meat" navigation
    - Be wary of the scroll

Summary
8. Having Confidence in Your Visualizations
Linting all the things
Static type checking with TypeScript and Flow
The new kid on the block – Facebook Flow
TypeScript – the current heavyweight champion
Behavior-driven development with Karma and Mocha Chai
Setting up your project with Mocha and Karma
Testing behaviors first – BDD with Mocha
Summary
A. Bibliography
Index

Data Visualization: Representing Information on Modern Web

Data Visualization: Representing Information on Modern Web

Unleash the power of data by creating interactive, engaging, and compelling visualizations for the web

A course in three modules

BIRMINGHAM - MUMBAI

Data Visualization: Representing Information on Modern Web

Copyright © 2016 Packt Publishing

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

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

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

Published on: September 2016

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-976-4

www.packtpub.com

Credits

Authors

Andy Kirk

Simon Timms

Ændrew Rininsland

Swizec Teller

Reviewers

Alberto Cairo

Ben Jones

Santiago Ortiz

Jerome Cukier

Jonathan Petitcolas

Saurabh Saxena

Elliot Bentley

Content Development Editor

Priyanka Mehta

Graphics

Disha Haria

Production Coordinator

Aparna Bhagat

Preface

Welcome to the craft of data visualization—a multidisciplinary recipe of art, science, math, technology, and many other interesting ingredients. Not too long ago we might have associated charting or graphing data as a specialist or fringe activity—it was something that scientists, engineers, and statisticians did.

Nowadays, the analysis and presentation of data is a mainstream pursuit. Yet, very few of us have been taught how to do these types of tasks well. Taste and instinct normally prove to be reliable guiding principles, but they aren't sufficient alone to effectively and efficiently navigate through all the different challenges we face and the choices we have to make.

This course offers a handy strategy guide to help you approach your data visualization work with greater know-how and increased confidence. It is a practical course structured around a proven methodology that will equip you with the knowledge, skills, and resources required to make sense of data, to find stories, and to tell stories from your data.

This course will help you understand about moulding data into a form which is more understandable. It is about taking some of the richest data sources of our time—social networks—and turning their vast array of data into an understandable format. To that effect, we make use of the latest in HTML, JavaScript and D3.js.

It will provide you with a comprehensive framework of concerns, presenting step-by-step all the things you have to think about, advising you when to think about them and guiding you through how to decide what to do about them.

Once you have worked through this course, you will be able to tackle any project—big, small, simple, complex, individual, collaborative, one-off, or regular—with an assurance that you have all the tactics and guidance needed to deliver the best results possible.

What this learning path covers

Module 1, *Data Visualization: a successful design process*, explores the unique fusion of art and science that is data visualization; a discipline for which instinct alone is insufficient for you to succeed in enabling audiences to discover key trends, insights and discoveries from your data. This module will equip you with the key techniques required to overcome contemporary data visualization challenges.

Module 2, *Social Data Visualization with HTML5 and JavaScript*, provides you with an introduction to creating an accessible view into the massive amounts of data available in social networks. Developers with some JavaScript experience and a desire to move past creating boring charts and tables will find this module a perfect fit. You will learn how to make use of powerful JavaScript libraries to become not just a programmer, but a data artist.

Module 3, *Learning d3.js Data Visualization*, covers various features of D3.js to build a wide range of visualizations. This module also focus on the entire process of representing data through visualizations so that developers and those interested in data visualization will get the entire process right and will provide a strong foundation in designing compelling web visualizations with D3.js.

What you need for this learning path

As with most skills in life that are worth pursuing, to become a capable data visualization practitioner takes time, patience, and practice.

You don't need to be a gifted polymath to get the most out of this course, but ideally you should have reasonable computer skills (software and programming), have a good basis in mathematics, and statistics in particular, and have a good design instinct.

There are many other facets that will, of course, be advantageous but the most important trait is just having a natural creativity and curiosity to use data as a means of unlocking insights and communicating stories. These will be key to getting the maximum benefit from this text.

There are very few tools needed to make use of the examples and code in this course. You'll need to install node.js (http://nodejs.org/) which is covered in Module 3, Learning d3.js Data Visualization,, Chapter 1: *Getting Started with D3, ES2016, and Node.js* and Module 2, Social Data Visualization with HTML5 and JavaScript, Chapter 5: *Twitter*.

You can run it on pretty much anything, but having a few extra gigabytes of RAM available will probably help while developing. Some of the mapping examples later in the course are kind of CPU-intensive, though most machines produced since 2014 should be able to handle them.

You'll want to download d3.js (http://d3js.org), jQuery (http://jquery.com), and Raphael.js (http://raphaeljs.com/). All the demos can be viewed in any modern web browser. The code has been tested against Chrome but should work on FireFox, Opera, and even Internet Explorer.

You will also need the latest version of your favorite web browser; the code is tested on Chrome, and has been used in the examples, but Firefox also works well. You can try to work in Safari, Internet Explorer/Edge, Opera, or any other browser.

Who this learning path is for

Regardless of whether you are an experienced visualizer or a rookie just starting out, this course should prove useful for anyone who is serious about wanting to optimize his or her design approach.

The intention of this course is to be something for everyone—you might be coming into data visualization as a designer and want to bolster your data skills, you might be strong analytically but want inspiration for the design side of things, you might have a great nose for a story but don't quite possess the means for handling or executing a data-driven design.

Some of you may never actually fulfill the role of a designer and might have other interests in learning about data visualization. You may be commissioning work or coordinating a project team and want to know how to successfully handle and evaluate a design process.

Hopefully, it will inform and inspire all who wish to get involved in data visualization design work regardless of role or background.

Readers should have a working knowledge of both JavaScript and HTML. jQuery is used numerous times throughout the course, so readers would do well to be familiar with the basics of that library but no prior experience with data visualization or D3 is required to follow this course. Some exposure to node.js would be helpful but not necessary.

Reader feedback

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

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the title of the course in the subject of your message.

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

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

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

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

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

The code bundle for the course is also hosted on GitHub at https://github.com/PacktPublishing/Data-Visualization--Representing-Information-on-Modern-Web. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

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

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

Piracy

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

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

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

Questions

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

Part 1. Module 1

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

Chapter 1. The Context of Data Visualization

This opening chapter provides an introduction to the subject of data visualization and the intention behind this book.

We start things off with some context about the subject. This will briefly explain why there is such an appetite for data visualization and why it is so relevant in the modern age against the backdrop of enhanced technology, increasing capture and availability of data, and the desire for innovative forms of communication.

After this introduction, we then look at the theoretical basis of data visualization, specifically the importance of understanding visual perception. To help establish a term of reference for the rest of the book, we'll then consider a proposed definition for this subject.

Next, we introduce the data visualization methodology, a recommended approach that forms the core of this book, and discuss its role in supporting an effective and efficient design process.

Finally, we consider some of the fundamental data visualization design objectives. These provide a useful framework for evaluating the suitability of the choices we make along the journey towards an accomplished design solution.

Exploiting the digital age

The following is a quotation from Hal Varian, Google's chief economist (http://www.mckinseyquarterly.com/Hal_Varian_on_how_the_Web_challenges_managers_2286):

> *The ability to take data—to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it—that's going to be a hugely important skill in the next decades.*

Data visualization is not new; the visual communication of data has been around in various forms for hundreds and arguably thousands of years. Popular methods that still dominate the boardrooms of corporations across the land—the line, bar, and pie charts—originate from the eighteenth century.

What *is* new is the contemporary appetite for and interest in a subject that has emerged from the fringes and into mainstream consciousness over the past decade.

Catalyzed by powerful new technological capabilities as well as a cultural shift towards greater transparency and accessibility of data, the field has experienced a rapid growth in enthusiastic participation.

Where once the practice of this discipline would have been the preserve of specialist statisticians, engineers, and academics, the globalized field that exists today is a very active, informed, inclusive, and innovative community of practitioners pushing the craft forward in fascinating directions. The following image shows a screenshot of the OECD 'Better Life Index', comparing well-being across different countries. This is just one recent example of an extremely successful visual tool emerging from this field.

Image from "OECD Better Life Index" (http://oecdbetterlifeindex.org), created by Moritz Stefaner (htpp://moritz.stefaner.eu) in collaboration with Raureif GmbH (http://raureif.net)

Data visualization is the multi-talented, boundary-spanning trendy kid that has seen many esteemed people over the past few years, such as Hal Varian, forecasting this as one of the next big things.

Anyone considering data visualization as a passing fad or just another vacuous buzzword is short-sighted; the need to make sense of and communicate data to others will surely only increase in relevance. However, as it evolves from the *next* big thing to the *current* big thing, the field is at an important stage of its diffusion and maturity. Expectancy has been heightened and it does have a certain amount to prove; something concrete to deliver beyond just experimentation and constant innovation.

It is an especially important discipline with a strong role to play in this modern age. To help frame this, let's first look at the data side of things.

Take a minute to imagine your data footprint over the past 24 hours; that is, the activities you have been involved in or the actions you have taken that will have resulted in data being created and captured.

You've probably included things such as buying something in a shop, switching on a light, putting some fuel in your car, or watching a TV program: the list can go on and on.

Almost everything we do involves a digital consequence; our lives are constantly being recorded and quantified. That sounds a bit scary and probably a little too close for comfort to Orwell's dystopian vision. Yet, for those of us with an analytical curiosity, the amount of data being recorded creates exciting new opportunities to make and share discoveries about the world we live in.

Thanks to incredible advancements and pervasive access to powerful technologies we are capturing, creating, and mobilizing unbelievable amounts of data at an unbelievable rate. Indeed, such is the exponential growth in digital information, in the last two years alone, humanity has created more data than had ever previously been amassed (http://www.emc.com/leadership/programs/digital-universe.htm).

Data is now rightly seen as an invaluable asset, something that can genuinely help change the world for the better or potentially create a competitive goldmine, depending on your perspective. "Data is the new oil", first voiced in 2006 and attributed to Clive Humby of Dunnhumby, is a term gaining traction today. Corporations, government bodies, and scientists, to name but a few, are realizing the challenges and, moreover, opportunities that exist with effective utilization of the extraordinary volumes, large varieties, and great velocity of data they govern.

However, to unlock the potential contained within these deep wells of ones and zeros requires the application of techniques to explore and convey the key insights.

Flipping to the opposite side of the data experience, we also identify ourselves as consumers of data. As you would expect, given the volume of captured data, never before in our history have we been faced with the prospect of having to process and digest so much.

Through newspapers, magazines, advertising, the Web, text messaging, social media, and e-mail, our eyes and brains are being relentlessly bombarded by information. In a typical day, it is said we can expect to consume about 100,000 words (http://hmi.ucsd.edu/howmuchinfo_research_report_consum.php), which is an astonishing quantity of signals for us to have to make sense of.

Unquestionably, a majority of this visual onslaught flies past us without consequence. We see much of it as noise and we zone out as a way of coping with the overload and saturation of things to think and care about.

What this shows is the necessity to be more effective and efficient in how data is communicated. It needs to be portrayed in ways that help to get our messages across in both an engaging and informative way.

If data is the oil, then data visualization is the engine that facilitates its true value and that is why it is such a relevant discipline for exploiting our digital age.

Visualization as a discovery tool

One of the most compelling arguments for the value of data visualization is expressed in this quote from John W Tukey (*Exploratory Data Analysis*).

The greatest value of a picture is when it forces us to notice what we never expected to see.

Through visualization, we are seeking to portray data in ways that allow us to see it in a new light, to visually observe patterns, exceptions, and the possible stories that sit behind its raw state. This is about considering visualization as a tool for discovery.

A well known demonstration that supports this notion was developed by noted statistician Francis Anscombe (incidentally, brother-in-law to Tukey) in the 1970s. He compiled an experiment involving four sets of data, each exhibiting almost identical statistical properties including mean, variance, and correlation. This was known as "Anscombe's quartet".

x1	y1	x2	y2	x3	y3	x4	y4
10	8.04	10	9.14	10	7.46	8	6.58
8	6.95	8	8.14	8	6.77	8	5.76
13	7.58	13	8.74	13	12.74	8	7.71
9	8.81	9	8.77	9	7.11	8	8.84
11	8.33	11	9.26	11	7.81	8	8.47
14	9.96	14	8.1	14	8.84	8	7.04
6	7.24	6	6.13	6	6.08	8	5.25
4	4.26	4	3.1	4	5.39	19	12.5
12	10.84	12	9.13	12	8.15	8	5.56
7	4.82	7	7.26	7	6.42	8	7.91
5	5.68	5	4.74	5	5.73	8	6.89

Sample data sets recreated from Anscombe, Francis J. (1973) Graphs in statistical analysis. American Statistician, 27, 17–21

Ask yourself, what can you *see* in these sets of data? Do any patterns or trends jump out? Perhaps the sequence of eights in the fourth set? Otherwise there's nothing much of interest evident.

So what if we now visualize this data, what can we see then?

Image published under the terms of "Creative Commons Attribution-Share Alike", source: http://commons.wikimedia.org/wiki/File:Anscombe%27s_quartet_3.svg

Through the previous graphical display, we can immediately see the prominent patterns created by the relationships between the X and Y values across the four sets of data as follows:

- the general tendency about a trend line in **X1**, **Y1**
- the curvature pattern of **X2**, **Y2**
- the strong linear pattern with single outlier in **X3**, **Y3**
- the similarly strong linear pattern with an outlier for **X4**, **Y4**

The intention and value of Anscombe's experiment was to demonstrate the importance of presenting data graphically. Rather than just describing a dataset based on a selection of some of its key statistical properties alone, to make proper sense of data, and avoid forming false conclusions we need to also employ visualization techniques.

It is much easier to discover and confirm the presence (or even absence) of patterns, relationships,

and physical characteristics (such as outliers) through a visual display, reinforcing the essence of Tukey's quote about the value of pictures.

Data visualization is about a discovery process, enabling the reader to move from just looking at data to actually seeing it. This is a subtle but important distinction.

The bedrock of visualization knowledge

Data visualization is not easy. Let's make that clear from the start. It should be genuinely viewed as a craft. It is a unique convergence of many different skills and requires a great deal of practice and experience, which clearly demands time and patience.

Above all, it requires a deep and broad knowledge across several traditionally discrete subjects, including cognitive science, statistics, graphic design, cartography, and computer science.

This multi-disciplinary recipe unquestionably makes it a challenging subject to master but equally provides an exciting proposition for many. This is evidenced by the field's popular participation, drawing people from many diverse backgrounds.

If we look at this subject convergence at a more summary level, data visualization could be described as an intersection of art and science. This combination of creative and scientific perspectives represents a delicate mixture. Achieving an appropriate balance between these contrasting ingredients is one of the fundamental factors that will determine the success or failure of a designer's work.

The *art* side of the field refers to the scope for unleashing design flair and encouraging innovation, where you strive to design communications that appeal on an aesthetic level and then survive in the mind on an emotional one. Some of the modern-day creative output from across the field is extraordinary and we'll see a few examples of this throughout the chapters ahead.

The *science* behind visualization comes in many shapes. I've already mentioned the presence of computer science, mathematics, and statistics, but one of the key foundations of the subject comes through an understanding of cognitive science and in particular the study of visual perception. This concerns how the functions of the eye and the brain work together to process information as visual signals.

One of the other most influential founding studies about visual perception emerged from the Gestalt School of Psychology in the early 1900s, specifically in the shape of the Laws of Perceptual Organization (http://www.interaction-design.org/encyclopedia/data_visualization_for_human_perception.html).

These laws provide an organized understanding about the different ways our eyes and brain inherently and automatically form a global sense of patterns based on the arrangement and physical attributes of individual elements.

Here, we can see two visual examples of Gestalt Laws.

On the left-hand side is a demonstration of the "Law of Similarity". This shows a series of rows with differently shaded circles. When we see this our visual processes instantly determine that the similarly shaded circles are related and part of a group that is separate and different to the non-shaded rows. We don't need to think about this and wait to form such a conclusion; it is a preattentive

reaction.

Images republished from the freely licensed media file repository Wikimedia Commons, source: http://en.wikipedia.org/wiki/File:Gestalt_similarity.svg and http://en.wikipedia.org/wiki/File:Gestalt_proximity.svg

On the right-hand side is a demonstration of the "Law of Proximity". The arrangement of closely packed-together pairs of columns means we assume these to be related and distinct from the other pairings. We don't really view this display as six columns, rather we view them as three clusters or sets.

At the root of visual perception knowledge is the understanding that our visual functions are extremely fast and efficient processes whereas our cognitive processes, the act of thinking, is much slower and less efficient. How we exploit these attributes in visualization has a significant impact on how effectively the design will aid interpretation.

Consider the following examples, both portraying analysis of the placement of penalties taken by soccer players.

When we look at the first image, the clarity of the display allows us to instantly identify the football symbols, their position, and their classifying color. We don't need to think about how to interpret it, we just do. Our thoughts, instead, are focused on the consequence of this information: what do these patterns and insights mean to us? If you're a goalkeeper, you'll be learning that, in general, the penalty taker tends to place their shots to the right of the goal.

Image republished under the terms of "fair use", source: http://www.facebook.com/castrolfootball

By contrast, this second display's attempt to portray the same type of data presentation causes significant visual clutter and confusion. Rather than using a simple and relatively blank image like the previous one, this display includes strong colors and imagery in the background. The result is that our eyes and brain have to work much harder to spot the footballs and their colors because the data layer has to compete for attention with the background imagery. We are therefore unable to rely on the capabilities of our preattentive visual perception (determined by the Law of Similarity) because we cannot easily perceive the shapes and their attributes representing the data. This delays our interpretative processes considerably and undermines the effectiveness and efficiency of the communication exchange.

Image republished under terms of "fair use", source: http://www.mirror.co.uk/sport/football/euro-2012-where-italy-will-place-their-penalties-907506

This is just a single, simple example but it does reveal the significance of understanding and obeying visual perception laws when portraying our data.

When we design a visualization, we need to take advantage of the strengths of the visual function and avoid the disadvantages of the cognitive functions. We need to minimize the amount of thinking or "working out" that goes into reading and interpreting data and simply let the eyes do their efficient and effective job.

Through the pioneering studies and development of theories acquired and refined over many years by the Gestalt School of Psychology as well as influential academics and theorists like Jacques Bertin, Francis Anscombe, John W Tukey, Jock McKinlay, and William Cleveland, we now have a greater understanding of how to achieve effective and efficient visualization design.

There is still a great amount of empirical evidence to gather, studies to conduct, and firm answers to unearth, but the wealth of knowledge available to us is a significant help to remove an undue amount of instinct in our design work.

Defining data visualization

It is important now to consider a definition of data visualization. To do this, we first need to consider the main agents involved in the exchange of information; namely, the messenger, the receiver, and the message. The relationship between these three is clearly very important, as this illustration explains:

On one side we have a messenger looking to impart results, analysis, and stories. This is the designer. On the other side, you have the receiver of the message. These are the readers or the users of your visualization. The message in the middle is the channel of communication. In our case this is the data visualization; a chart, an online interactive, a touch screen installation, or maybe an infographic in a newspaper. This is the form through which we communicate to the receiver.

The task for you as the designer is to put yourself in the shoes of the reader. Try to imagine, anticipate, and determine what they are going to be seeking from your message. What stories are they seeking? Is it just to learn something new or are they looking for persuasion, something with more emotional impact? This type of appreciation is what fundamentally shapes the best practices in visualization design: considering and respecting the needs of the reader.

The important point is this: to ensure that our message is conveyed in the most effective and efficient form, one that will serve the requirements of the receiver, we need to make sure we design (or "encode") our message in a way that actively exploits how the receiver will most effectively interpret (or "decode") the message through their visual perception capabilities.

From this illustration we can form the following definition to clarify, at this early stage, what we mean by data visualization:

The representation and presentation of data that exploits our visual perception abilities in order to amplify cognition.

Let's take a closer look at the key elements of this definition to clarify its meaning; these are as follows:

- The **representation** of data is the way you decide to depict data through a choice of physical forms. Whether it is via a line, a bar, a circle, or any other visual variable, you are taking data as the raw material and creating a representation to best portray its attributes. We will cover this aspect of design much more in Chapter 4, *Conceiving and Reasoning Visualization Design Options* and Chapter 5, *Taxonomy of Data Visualization Methods*.
- The **presentation** of data goes beyond the representation of data and concerns how you integrate your data representation into the overall communicated work, including the choice of colors, annotations, and interactive features. Similarly, this will be covered in depth in Chapter 4, *Conceiving and Reasoning Visualization Design Options*.
- Exploiting our **visual perception abilities** relates to the scientific understanding of how our eyes and brains process information most effectively, as we've just discussed. This is about harnessing our abilities with spatial reasoning, pattern recognition, and big-picture thinking.
- **Amplify cognition** is about maximizing how efficiently and effectively we are able to process the information into thoughts, insights, and knowledge. Ultimately, the objective of data visualization should be to make a reader or users feel like they have become better informed about a subject.

The definition that I've put forward here is not dissimilar to the many others articulated by authors, academics, and designers down the years. It is not intended to offer a paradigm shift in our understanding of what this is all about. Rather, it represents a personal perspective of the discipline influenced by many years of experience teaching, practicing, and constantly studying the subject.

The fact that data visualization is such a dynamic and evolving field, with this unique conjunction of art and science shaping its practice, means that a single, perfect, and universally-agreed definition is always going to be difficult to construct. However, this proposed definition should at least help you develop an appreciation of the boundaries of data visualization and recognize when something evolves into a different form of creative output.

Visualization skills for the masses

The following is a quote from Stephen Few from his book *Show Me the Numbers*:

> *"The skills required for most effectively displaying information are not intuitive and rely largely on principles that must be learned."*

More and more of us are becoming responsible for the analysis, presentation, and interpretation of data. This naturally reflects the explosion in access to data and the value attributed to potential insights that are contained.

As I've already stated, where once this was typically a specialist role, nowadays the responsibility for dealing with data has crept into most professional duties. This has been accelerated by the ubiquitous availability of a range of accessible productivity tools to handle and analyze data.

This means visualization has become both a problem and an opportunity for the masses, which makes the importance and dissemination of effective practice a key imperative.

The quote from Stephen Few will resonate with many of you reading this. If you were to ask yourself "Why do I design visualizations in the way I do?", what would be your answer? Think about any chart or graphic you produce to communicate information to others. How do you design it? What factors do you take into account? Perhaps your response would fall in to one or more of the following:

- You have a certain design style based on personal taste
- You just play around until something emerges that you instinctively like the look of
- You trust software defaults and don't go beyond that in terms of modifying the design
- You have limited software capabilities, so you don't know how to modify a design
- You just do as the boss tells you—"can you do me some fancy charts?"

For many people, the idea of a conscious data visualization design technique isquite new. The absence of any formal coaching, at almost any level of education, in the techniques of visualization means until you become aware of the subject, you have probably never even thought about your visualization design approach.

Before discovering this subject, my own approach to presenting data was certainly not informed by any training or prior knowledge. I'd never even thought about it. Taste and gut-feel were my guiding principles alongside a perceived need to show off technical competencies in tools like Excel. Indeed, I'd like to take this opportunity to apologize for much of my graphical output between 1995 and 2005 where striking gradients and "impressive" 3D were commonplace. The thing is, as I've just said, I didn't realize there was a better way; it simply wasn't on my radar.

In some respects, the reliance on instinct, playing about with solutions that seem to work fine for us, can suffice for most of our needs. However, these days, you often hear the desire being expressed to move beyond devices like the bar chart and find different creative ways to communicate data.

While it is a perfectly understandable desire, just aiming for something different (or even worse, something "cool") is not a good enough motive in itself.

If we want to optimize the way we approach a data visualization design, whether it be a small, simple chart or a complicated interactive graphic, we need to be better equipped with the necessary knowledge and appreciation of the many design and analytical decisions we need to make.

As suggested previously, instinct and taste have got us so far but to move on to a whole new level of effectiveness, we need to understand the key design concepts and learn about the creative process. This is where the importance of a methodology comes in.

The data visualization methodology

The design methodology described in this book is intended to be portable to any visualization challenge. It presents a sequence of important analytical and design tasks and decisions that need to be handled effectively.

As any fellow student of Operational Research (the "Science of Better") will testify, through planning and preparation, and the development and deployment of strategy, complex problems can be overcome with greater efficiency, effectiveness, and elegance. Data visualization is no different.

Adopting this methodology is about recognizing the key stages, considerations, and tactics that will help you navigate smoothly through your visualization project.

Remember, though, design is rarely a neat, linear process and indeed some of the stages may occasionally switch in sequence and require iteration. It is natural that new factors can emerge at any stage and influence alternative solutions, so it is important to be open-minded and flexible. Things might need to be revisited, decisions reversed, and directions changed. What we are trying to do, where possible, is find the best path through the minefield of design choices.

Some may feel uncomfortable at the prospect of following a process to undertake what is fundamentally an iterative, creative design process. But I would argue everyone should find value from working in a more organized and sequenced way especially if it helps to reduce inefficiency and wasted resource.

The design challenges involved in data visualization are predominantly technology related; the creation and execution of a visualization design will typically require the assistance of a variety of applications and programs. However, the focus of this methodology is intended to be technology-neutral, placing an emphasis on the concepting, reasoning, and decision-making.

The variety, evolution, and generally fragmented nature of software in this field (there is no single tool that can do everything) highlights the extra importance of reasoned decision-making, regardless of the richness and power individual solutions can offer.

Another key point to remark on is to emphasize, if it wasn't already clear, that data visualization is not an exact science. There is rarely, if ever, a single right answer or single best solution. It is much more about using heuristic methods to determine the most satisfactory solutions.

On that note, the content of the methodology intentionally avoids any sense of dogmatic instruction, preferring to focus on guidelines over explicit rules; sometimes an ounce of chaos, a certain license to experiment, a leaning on instinct, and a sense of randomness can spark greater creativity and serendipitous discovery.

The methodology is intended to be adopted flexibly, based on your own judgment and discretion, by simply laying out all the important things you need to take into account and proposing some potential

solutions for different scenarios.

Finally, as I stressed with my definition of the subject earlier, I'm not suggesting this is a ground-breaking new take on the creative process. It is merely a personal interpretation based on experience and also exposure to the many brilliant people out there who share their own design narratives. It is, though, consistent with how most established observers of the subject would recommend you undertake this task. Moreover, it is an approach that I fundamentally believe works and it has genuinely helped me improve my own work since I've adopted it more deliberately, allowing me to cut through projects with the efficiency and elegance I've always yearned for.

Visualization design objectives

Before we launch in to the first stages of the methodology in Chapter 2, *Setting the Purpose and Identifying Key Factors*, it is important to acknowledge a handful of key, overriding design objectives that should provide you with a framework to test your progress and the suitability of your design decisions.

Whereas the methodology will introduce a number of key thoughts and decisions at each stage of the process, these objectives transcend any individual step and highlight the intricate issues you have to handle throughout your process.

The key objectives are as follows:

Strive for form and function

The following is a quote from Frank Lloyd Wright:

> *"Form follows function—that has been misunderstood. Form and function should be one, joined in a spiritual union."*

The first objective brings us immediately face-to-face with the age-old debate of form versus function or style over substance. As Frank Lloyd Wright proposed, all the way back in 1908, these are aspects of design that should be combined and brought together in harmony, not at the sacrifice of one or the other. There's room and a need for both.

It is a very difficult balancing act to achieve, as I've already alluded to in the discussion about art and science, but our aim should be to hit that sweet-spot where something is aesthetically inviting and functionally effective.

The designer and author Don Norman (http://www.jnd.org/dn.mss/emotion_design.html) talks about how we're more tolerant about things that are attractive and more likely to want them to perform well. Indeed, there is a school of thought that suggests how we think cannot be separated from how we feel.

Norman goes on to describe how well-executed aesthetics can naturally create favorable emotional and mental responses, but emotional affection can also come from the experience of good usability and the accomplishment of insight. Fundamentally, attractive form enhances function and the function portrays beauty through its effect.

Throughout this book, we will see examples of designs that have succeeded in creating elegance in form and in function. The following image is taken from an animated wind map developed by Fernanda Viégas and Martin Wattenberg. It is a beautiful piece of work, exceptionally well designed and executed but it also serves its purpose as a way of informing users about the wind patterns, strength, and directions occurring across the United States. This is form and function in spiritual union:

Image from "Wind Map" (http://hint.fm/wind/) created by Fernanda Viégas and Martin Wattenberg

The general advice, especially for beginners, is to initially focus on securing the functional aspects of your visualization. First, try to achieve the foundation of something that informs—that functions—before exploring the ways of enhancing its form. The simplest analogy would be build the house before decorating it, but I wouldn't want to create too much separation between the two as they are often intrinsically linked. Over time, you will be much more confident and capable of synthesizing the two demands in harmony. We shall discuss this in more depth in Chapter 4, *Conceiving and Reasoning Visualization Design Options*.

Justifying the selection of everything we do

The following is a quote from Amanda Cox (http://vimeo.com/29391942), who works as a graphics editor at the New York Times:

> *"We're so busy thinking about if we can do things, we forget to consider whether we should."*

In many ways, the central idea behind the methodology is encouraging you to determine that everything you do is thoroughly planned, understood, and reasoned.

This particular objective is about recognizing and responding to the scoping information that you will gather at the start of the methodology, to ensure that everything undertaken thereafter serves the purpose of our work and the needs of the audience.

Here, we should consider the idea of deliberate design, which means that the inclusion, exclusion, and execution of every single mark, characteristic, and design feature is done for a reason.

When we reach the stage of designing, concepting, and construction, you should be prepared to challenge everything; the use of a shape, the selection of a color pallet, the position of a label, or the use of an interaction.

In this next example, when displaying a section of a tree-hierarchy work by data illustrator, Stefanie Posavec, every visible property presented is used to communicate data, whether it be the use of color, the arc lengths of the petals, the position and sequence of stems; nothing is redundant and everything is deliberate.

Image from "Literary Organism" (http://itsbeenreal.co.uk/index.php?/wwwords/literary-

organism/), created by Stefanie Posavec

It is also important to make sure that any visual property that is included, but does not represent data, such as shading, labels, colors, and axes among other properties, should only be included to aid the process of visual perception, not hinder it.

Furthermore, for interactive and animated visualizations, remember Amanda Cox's quote—"just because you can, doesn't mean you should." Don't succumb to the belief (like I did for many years) of thinking a visualization is a platform solely to showcase your technical competence.

Cluttering visualizations with fancy interactive features is a trap that is easy to fall into and leads to projects that look nice or are impressive technically but fail to serve their intended purpose. Instead, they interfere with the efficiency and effectiveness of the information exchange thus demonstrating a failure to synthesize form and function.

Creating accessibility through intuitive design

The following is a quote from Edward Tufte (http://adage.com/article/adagestat/edward-tufte-adagestat-q-a/230884/):

> *"Overload, clutter, and confusion are not attributes of information, they are failures of design."*

When you next happen to be in a town or city center, take a look around you and observe how often people are confused by and struggle with the basic operation of correctly opening and entering doors into a store. Notice how the accessibility and function of a door—the simple act of opening and walking through it—is often impaired through a lack of intuitive design.

The method of opening a door should be straightforward, but often the aesthetics of features such as stylish door handles means we pull when we should push and we push when we should pull. This is a flaw in the intuitiveness and logic of the design, a failure in perceived affordance—it doesn't do what it looks like it should do.

This idea is an important concept to translate into visualization. As we have already outlined, we are trying to exploit the inherent spatial reasoning and pattern recognition functions of visual perception. We don't want people to have to spend unnecessary time thinking about how to use or how to read and interpret something.

When you are creating a visualization, you are integrating visual design with a subject matter's data. The former is the window into the latter, and it is the design and execution of this window that creates the accessibility.

But it is important to create a distinction between accessibility and immediacy. The speed with which you are able to read or interpret a visualization should be determined by the complexity of the subject and the purpose of the project, not by the ineffectiveness of design.

Sometimes subjects are fundamentally simple and the portrayal of the data is straightforward and intuitive. This in turn means the reader's task of interpreting the data should be relatively easy.

On other occasions, a data framework might be more complex. Your challenge will be to respect the complexity and avoid simplifying, diluting, or reducing the essence of this subject. This might mean something is not immediately easy to interpret. Some visualizations will require effort to be put in, forcing the reader to undertake a certain amount of experiential practice in order for the eye and mind to essentially become trained in reading the display.

Think of it being like muscle memory, but for the eye and the brain. We are so used to reading bar charts and line charts that they have become entrenched and programmed into our interpretative toolkit. But when we are faced with something new, something different or seemingly complex, its not always immediately clear how we are supposed to handle it.

In the following example, we see a demonstration of what is quite a complex data framework. This is an image of a legend that was used to explain how to read an innovative visualization to portray three separate indicators of a movie's success. On the left-hand side of the image is the aggregate reviews (the higher the value, the better) and on the right-hand side of the image are both the budget and gross takings (the bigger the gap, the better):

Image from "Spotlight on Profitability" (http://www.szucskrisztina.hu), created by KrisztinaSzucs

It is an unusual representation of data, not something as preprogrammed as the bar or line chart, and so it takes a short while to learn how to read and interpret the resulting shapes formed by the movie data shown across piece. This is absolutely legitimate as an effective approach to visualizing this data so long as the efforts that go into learning how to read it eventually leads the user to understand it.

Take another example, which portrays the key events in a couple of soccer matches showing completed passes (green lines), shots (blue triangles), and goals (red dots) as shown in the following image:

Image from "Umbro World Cup Poster" (http://www.mikemake.com/Umbro-s-World-Cup-Poster), created by Michael Deal

Once the reader has mastered the understanding of what each shape and its position means, these displays provide a powerful and rewarding insight in to the key incidents and the general ebb and flow of each game.

In simple terms, so long as you can avoid all the negative characteristics that Edward Tufte mentions at the top of this section, you should succeed in giving people an accessible route in to the data. Make sure that the efforts needed from the reader or user to understand how to use and interpret a visualization are ultimately rewarded with a worthy amount of insight gained.

Never deceive the receiver

Visualization ethics relates to the potential deception that can be created, intentionally or otherwise, from an ineffective and inappropriate representation of data. Sometimes it can be through a simple lack of understanding of visual perception.

In the following diagram, we see a 2D pie chart and a 3D version. When the eye interprets a graphic like this, what it is actually doing is perceiving the proportion of visible pixels:

Image from "The Curious Incident of Kevins in Zurich…and other stories" (http://www.researchobservatories.org.uk/EasysiteWeb/getresource.axd?AssetID=38334) by Alan Smith.

On the left-hand side of the diagram, we see a blue segment representing 82 percent and an orange segment representing 18 percent. These are the actual values. However, when we introduce a third dimension on the right—incidentally, a dimension which is purely decorative and has no relationship with data values—our eyes are deceived because we are not capable of easily adjusting our interpretation of the values across this isometric projection. With the introduction of the extra dimension and the visible height of the pie itself, we now perceive 91 percent of the visible area as blue and only 9 percent orange. This is clearly a hugely distorted reading of the values.

Another similar example comes from a Wikipedia fundraising campaign from a few years ago and a progress bar depicting the status of their efforts; as shown in the following screenshot:

Image published under the terms of "Creative Commons Attribution-Share Alike", source:

https://donate.wikimedia.org/

As with the pie chart, for a bar chart we perceive the visible pixels as being representative of the values. The label indicates a total of **$0.8M USD** had been raised (10.7 percent towards target) but if you calculate the actual length of the bar displayed, this occupies 24.6 percent of the overall bar length. Once again, a significant distortion of the truth.

This next example is a demonstration of where aesthetics and style completely hijack a visualization. Here, we have a still showing a 3D bar chart that swooshes impressively onto the screens of those watching soccer on TV in the UK:

But what have we here? There is a yellow **Drawn** bar representing the value **1** and this appears to be more than half the length of a red **Aston Villa** bar representing **4**. How can that be?

The designers of this visual have chosen to include the category labels within the bar's length, thus completely distorting the values being represented. Now, this is possibly one of the least interesting statistics you'll come across, and I'm assured the world will not stop turning as a result of this graphical misdemeanor, but it should demonstrate the pitfalls of decoration and overly stylized design.

Obeying visualization ethics is clearly an objective for any project, but really it is just about basic, good practice, respect for your readers, and attention to detail.

Summary

In this chapter, we have learned about the context of the digital era and the role data visualization can play in helping us make greater sense of the huge volumes of captured data we have access to in today's world.

We have discussed how more and more people are getting involved in activities that require visualization techniques, but the skills required to accomplish this effectively go beyond instinct and require careful learning and practice.

The methodology presented in this book will provide a strategy for designers to develop these techniques through good practice. It will help them navigate through the key decisions that are required throughout the creative process.

Finally, to commence the design thinking, we have learned about some important overriding objectives that should provide a useful assessment of the effectiveness of your visual solution throughout its creation.

In the next chapter, we will commence the data visualization methodology by exploring the first stage of any design challenge: establishing the project's purpose and identifying its inherent key influencing factors.

Chapter 2. Setting the Purpose and Identifying Key Factors

Chapter 1, *Context of Data Visualization*, gave us some initial idea about the definition, context, and relevance of data visualization. We learned about the intersection that exists between art and science and outlined the significance of visual perception.

We also discussed the proposed value of a data visualization methodology and, ahead of our design process, learned about some key objectives we need to bear in mind throughout our work.

In this second chapter, we start the design methodology with the vital task of identifying the purpose of your visualization.

Before you undertake any design work, you have to be clear about the motivation behind a project's inception. This involves identifying who it is for and what needs you are trying to fulfill; this has a big influence on the scope of your work.

The second aspect of purpose will take a close look at the intention behind your project and how you define the visualization's function and tone. Once again, we tackle this now so that we can start to shape our decision-making at the earliest possible opportunity.

We then look at identifying and assessing the impact of the additional key factors that will have an effect on your project. This will help you surface all the restrictions, characteristics, and requirements surrounding your project that will determine how you tackle it.

Finally, we will consider in more depth a particularly influential matter: the skills, knowledge, and general capabilities that are necessary to accomplish an effective visualization solution.

Clarifying the purpose of your project

We start at the very beginning. Why are we doing this project? What is its purpose?

These might seem like blunt questions to ask but it is so important to establish this type of clarity before we go much further. You might think: "please can we just get on with it?" and you're probably itching to get on to a computer and start playing with some data, but these initial few stages of this methodology are very deliberately designed to get you in to the habit of this type of reflective or analytical assessment.

If you look at a dictionary definition for "purpose", it will usually say something similar to: reason for existing, intended effect. These two dimensions neatly capture the focus of our definition work at this point in the journey.

The reason for existing

Let's start with the reason for existing. This is about recognizing the trigger behind the project or the origin from where it emerged. This gives us an idea of the scope and context of what we are about to undertake, how much creative control we might have, whether we've been encouraged to follow a particular creative direction and what ideas have already been formed.

A project will typically form in one of following two ways: you've either been asked to do it or you've decided to do something yourself. You might think that's obvious, but these are very different scenarios for working creatively.

If it's the former, a project that's essentially been commissioned, you will have most likely been given the task by a colleague or a manager or from a client you are working with/for. A further source may be something like an invited assignment such as tendering for some work or even entering a design contest.

You will have received or read a brief and possibly had some initial discussions that provided you with an outline of the requirements. You might have some instructions and a general idea of what they are seeking.

From gathering this contextual information, you should have a reasonably clear idea about the background to the project, what you're being asked to do, why you're doing it and who you're doing it for. It may be quite loose and open-ended, in which case you've got a greater range of possibilities, but it's more likely to be quite defined and specific.

You've probably also experienced the pain of hearing some of the initial ideas flying round, as your creative soul dies a slow death in response to requests such as "cool charts" and "make it fancy" or "I want an Edward Tufte style piece".

By contrast, for a project that is self-initiated, things are very different. Maybe, it is a dataset that you've found about a subject that interests you, maybe you've decided to test out some theory or you've been chatting to (or, more truthfully, drinking with) mates and have struck on a particular curiosity that simply must be explored.

This scenario is a completely self-defined, self-determined, and more flexible context than that of a commissioned project. It doesn't involve a client, or a brief, or a set of instructions, or restrictions on scope, timescales, or audience—you've got a blank canvas to follow the scent of what it is that motivated you in the first place.

A very different proposition to a commissioned project and an important distinction that needs to be established.

The intended effect

Whatever the motivation and background for doing the project, you will inevitably start to form a vision in your mind of what you might be about to create, how it might look, and what it might do. This is a natural instinct as you embark on a creative process.

This vision might leap into your mind the minute you start to think around the task, regardless of its origin. You might recall certain influential or inspirational works that you've seen in the past or remember concepts you previously developed that went no further.

It's important to capture these thoughts if they do form. Make sure you keep notes, in your sketchbook, on your tablet, or on a cigarette packet—it doesn't matter where, just do it before you forget. While we don't want to be closed off and commit ourselves to the pursuit of the first thing we think of, these instinctive thoughts could prove valuable later on.

It's from these sparks of creativity that we shift our thoughts to the second dimension of purpose, which is the intended effect of the visualization project. This is a really critical matter, so we're going to take a bit of time to get our heads around it.

Here's why it's so important to be thinking about all this in such depth. Even though it is very early in the process, the decisions, or more accurately, the definitions we form now, will have a strong bearing on the creative direction we pursue. It's not doing anything that can't be reversed or refined further down the line but it is important to establish as much focus and clarity about our intentions now so that we can reduce the complexity of the challenge and the potential variety of the solution.

The choices we make now will also influence the resources (technical and personnel) we might need to deploy, but we shall look more at this towards the end of the chapter.

Remember in Chapter 1, *Context of Data Visualization* (you should do, after all it's only been a few pages) we saw a definition for data visualization that proposed the overall aim as being to "amplify cognition" or, in other words, make someone feel better informed. That was a deliberately equivocal aim because there are many different motivations and reasons for creating a data visualization.

Consider the following sample collection of phrases, which articulate a variety of viable intentions behind creating a visual representation of data:

If we look closely at the verbs and the overall language being used, we can start to recognize quite a range of differing effects that might be sought.

For example, a visualization to assist with the monitoring of signals or facilitating a visual lookup of data will be very different from a design that is intended to grab attention or change behavior. Similarly, presenting arguments and telling a story is a very different setting to conducting analysis or 'playing' with data.

What we have here is evidence of different dimensions of intent. Identifying your intended effect means deciding what you're aiming to achieve and how you're going to achieve it.

At the root of this is an appreciation of your target audience, one of the most important considerations we have to take into account. During this initial definition and scoping work, it is crucial to profile your intended readers/users.

These are the people who we are serving and so we need to recognize what type of engagement they are likely to require, for example:

- Is it a boardroom environment with a small collection of senior colleagues who have existing domain subject knowledge?
- Is it a large range of customers, covering all social demographics but potentially representing a captive audience for the subject matter?
- Is it a completely global, undefined audience with no influencing characteristics—in a sense no specific target, just anyone and everyone?
- Is it a one-to-one exchange with a manager?
- Is it an entirely personal engagement between you and data—a desire to learn about and explore data yourself?

Clarity of what and who your target audience is will help shape your intent and from this we'll be able to define two very important dimensions: the function and tone of your visualization project.

Establishing intent – the visualization's function

The intended function of a data visualization concerns the functional experience you create between your design, the data, and the reader/user.

If we revisit the range of phrases presented earlier, it is possible to form three separate clusters or categories of function. While there is always a chance of slight overlap, there will be a significant difference in your design choices depending on whether the function of your visualization is to:

- Convey an **explanatory** portrayal of data to a reader
- Provide an interface to data in order to facilitate visual **exploration**
- Use data as an **exhibition** of self-expression

When the function is to explain

Explanatory data visualization is about conveying information to a reader in a way that is based around a specific and focused narrative. It requires a designer-driven, editorial approach to synthesize the requirements of your target audience with the key insights and most important analytical dimensions you are wishing to convey.

There are many ways in which you can "explain" data. It could be through an information dashboard in a corporate setting, where you are conveying the latest performance figures and highlighting the key issues requiring attention. It might be a graphic in a newspaper, explaining the complexity and severity of the problems around the economic crisis. It could be an animated design to display patterns of population migration over time. It could also be a physical or ambient visualization designed to draw attention to the sugar content of certain drinks.

The end result is typically a visual experience built around a carefully constructed narrative. Your objective as the designer is to create a graphical display, made accessible through intuitive, visual design that clearly portrays the narrative you are seeking to impart.

Here is an example of an explanatory visualization, based on a chart type called a **Sankey** diagram, which portrays analysis of the top ten freshwater-consuming countries and the breakdown of its usage:

Image from "Top 10 Freshwater Consumers" (http://www.scientificamerican.com/article.cfm?id=water-in-water-out) created by Mark Fischetti and Jen Christiansen, Scientific American, June 2012. Reproduced with permission. Copyright © 2012 Scientific American, Inc. All rights reserved.

Note that explanatory visualizations are not limited to just being static in design. Indeed, some of the most impactive, narrative-driven pieces can be framed within an interactive or animated construction.

When the function is to explore

Exploratory data visualization design is a slightly different matter compared to creating an explanatory piece. Here, we are seeking to facilitate the familiarization and reasoning of data through a range of user-driven experiences. In contrast to explanatory-based functions, exploratory data visualizations lack a specific, single narrative. They are more about visual analysis than just the visual presentation of data.

Exploratory solutions aim to create a tool, providing the user with an interface to visually explore the data. Through this they can seek out personal discoveries, patterns, and relationships, thereby triggering and iterating curiosities. It also opens up the possibility for chance or serendipitous findings caused by forming different combinations of variable displays.

Really, the key feature that differentiates an exploratory piece from an explanatory piece is the amount of work you have to do as a reader to discover insights. For explanatory pieces, the designer should do the hard work and create a clear portrayal of the interesting stories and analysis from a dataset. An exploratory piece will be more about the readers doing the analysis themselves, putting the effort in to discover things that strike them as being significant or interesting.

In the following image, we see a **scatterplot matrix** visualization: a method used to reveal correlations across a multivariate dataset, enabling the eye to efficiently scan the entire matrix to quickly identify variable pairings with strong or weak relationships. This is a perfect example of an exploratory visualization design:

Image from "Scatterplot Matrix" (http://mbostock.github.com/d3/ex/splom.html), created by Mike Bostock.

Exploratory visualizations are not limited to being interactive. Visual analysis can be facilitated through static portrayals of data. The previous example is actually interactive but a static version would still offer a discovery of the relationships and patterns of the dataset.

That said, it is fair to say that in order to create a truly exploratory experience, interactivity does introduce the potential for so much extra functionality to help immerse the user into a dynamic, problem-solving challenge.

Features such as filtering, sorting, brushing (selecting or isolating certain data values), variable adjustment, and view modification are just some of the important ways you can help a user investigate data. We'll discuss more about interactivity in Chapter 4, *Preparing and Familiarizing with Data*.

It is also worth highlighting that while explanatory visualization is primarily created for others, exploratory data and the process of visual analysis can be as much for your own discovery purpose as it is for others. It is clearly a particularly relevant function for scientists, for example, to find patterns and unearth key findings in research work before the publication of results (which would then require the use of explanatory-based visual evidence).

When the function is to exhibit data

The final classification of intended function is in some respects a controversial one, because including the exhibiting of data as an intended function of visualization will not be consistent with many people's definition of data visualization.

We're not talking controversy on the level of a political scandal. Let's put it into some perspective: within the context of this field, this is a big deal and the cause of so much debate.

As with any attempt at classification, there is a spectrum of variety within and so clear boundaries are difficult to establish and very much open to personal interpretation.

We are talking here about designs that use data as the raw material, but where the intention is perhaps somewhat removed from a pure desire to inform. Rather, the objective is closer to a form of exhibition or self-expression through data representation. This genre of work embodies the term "data art".

Data art is characterized by a lack of structured narrative and absence of any visual analysis capability. Instead, the motivation is much more about creating an artifact, an aesthetic representation or perhaps a technical/technique demonstration. At the extreme end, a design may be more guided by the idea of fun or playfulness or maybe the creation of ornamentation.

This particular strand of data visualization is contentious simply because it challenges those seeking to identify the boundaries of this field and its proximity to other disciplines such as graphic design, generative design, or creative art.

In the following example, we see an example of data art (as defined by the creator himself) that visualizes all the adjectives used in Cormac McCarthy's book *The Road*. The adjectives are arranged radially in alphabetical order and each line represents a timeline of the book, beginning at the perimeter:

Image from "Adjectives of The Road" (http://distantshape.com/the_road.html), created by Kemper Smith.

The effect is an interesting artifact to look at and its construction is representative of an impressive technical or algorithmic solution, but its primary intent is not to easily allow us to learn about the language in the book. I would personally position this right on the boundary between an exploratory visualization (static, but allowing readers to look up combinations of adjectives and see patterns) and data art.

We characterized explanatory visualization as a single visual experience and exploratory as a numerous but finite set of experiences. By contrast, the range of reactions to exhibition-type designs has a more open and unlimited potential. It aims for and triggers more of an aesthetic reaction, which means our emotional connection and interpretation can vary significantly from one person to the next.

Establishing intent – the visualization's tone

Earlier, we looked at a collection of different phrases that articulate realistic intentions for creating a data visualization. We've just proposed three classifications for how you might organize these terms according to their function.

There will sometimes be an overlap between these otherwise distinct categories, but you should be able to determine relatively clearly where your work intends to fit on the scale between explanation, exploration, and exhibition.

Setting the function is just one part of the "intent" equation. The clarity of your potential design pathway will be much more apparent as we now consider the second dimension of intent—tone.

Establishing a suitable tone goes beyond function and more towards the style of the design experience. It concerns the type of stimulus or desired emotional response that you are trying to create. It is therefore important for you, as the designer, to be able to reason what sort of design will achieve that tone.

With this type of judgment to make, you will inevitably find yourself juggling creative and scientific perspectives. This dynamic poses a significant challenge for any data visualization designer to reason and resolve. Indeed, the design objectives outlined at the end of the previous chapter highlight the intricate issues you have to handle.

I've alluded to the debates and arguments around data art, well this subject inspires the most prominent dividing lines and debates that exist within the field revealing the occasionally awkward and misunderstood relationship between art and science, the pillars of this field.

While it is possibly a crude generalization, the standpoint of the science side is characterized as being concerned with preserving the efficiency and accuracy of judgments derived from a visualization. Variations in data representation that steer away from this goal are believed to reduce the quality and effectiveness of a visualization.

On the other side, the artistic section of the community can be viewed as being concerned with experimentation, finding creative expressions of data, and new aesthetic connections with an audience. Practitioners in this cohort, typically, will have arrived in the field from a strong, design-led or computer science background.

For the rest of us, somewhere in the middle, we find ourselves either holding hands with both sides, following the direction of the wind from one day to the next, or simply sat on a fence without any strong favor towards either side.

So how can we rationalize the role of these two very different, opposing perspectives and beliefs? In my view, the latter enhance the field by demonstrating what can be achieved through the aesthetic and technological creativity. The former help us understand what we should do through the pursuit of

evidence and observation of rules around human cognition and visual perception. We need visualizations that look appealing and we need visualizations that perform well.

However, sometimes there has to be mutual recognition that for different scenarios there might be good reason for leaning more towards one direction than the other.

Let's look at the language of two potential motives behind creating a data visualization:

- "We need a chart to help monitor…"
- "We need to present this in a way that persuades people…"

Here, we have two situations both aiming to better inform a reader or a user, but the intended effect or outcome from the experience will be different.

The reaction of a user reading, for example, a dashboard full of bar charts and line charts to help monitor monthly performance will be quite analytical and pragmatic in style. It is unlikely to involve or stir much emotion (unless things are suddenly and unexpectedly plummeting). The style of the visualization design will be consistent with the intended nature of this particular type of engagement, probably quite sober and with an emphasis on the precision of perception.

Compare that with the intended impact of a presentation that depicts how many lives could be saved if a charity was able to achieve a certain level of fundraising. The setting and intent will be more about persuasion making it emotionally charged. It will need to attempt to create an experience that is much more personal and more impactive.

Here, we see potentially two ends of a spectrum for judging the right tone. Yet they fundamentally share the overall motivation of wishing to inform people about a subject through the visual representation of data.

One scenario would achieve this in a relatively pragmatic style, influenced by a desire to optimize the efficiency and accuracy of interpretation. The other would be judged as effective if it evoked a suitably positive emotional response to the data story.

We can therefore describe tone as being a continuum from a pragmatic or analytical portrayal through to a more emotive or abstract concept.

Pragmatic and analytical

The following is a quotation from Jock Mackinlay (http://hci.stanford.edu/courses/cs448b/f10/lectures/CS448B-20100923-DataAndImageModels.pdf):

> *"A visualization is more effective than another visualization if the information conveyed by one visualization is more readily perceived than the information in the other."*

This quote perfectly captures the priority and intent behind pragmatic or analytical visualizations. Some might term them simple or boring but that is short-sighted and lacking in appreciation for the setting in which these types of data portrayal is vital.

Designs that fit this classification will often involve data being represented through the use of bar charts, line charts and dot plots, for example. Stylistically, they will be characterized by a rather clinical look-and-feel that is consistent with the next sample image, taken from a project analyzing Olympic results over the years:

Creating a visualization with a pragmatic tone is about recognizing a need for a design that delivers fast, efficient and precise portrayals of data. Typically, you will have a captive audience, a readership who want to or need to interact and learn from the data. This could be a corporate

environment, where people need to simply learn about recent performance of operational activity or undertake visual analysis to discover potentially revealing patterns.

In these cases, there is no value placed on attempting to draw attention to the visualization, or trying to encourage somebody to read or interact with a graphic by employing aesthetic novelty. Furthermore, it's not about trying to inject any emotional or metaphorical connection with the data stories presented.

The purity and impact of its function—the satisfaction that comes from an efficient intake of understanding—fulfils the aesthetics of the charting methods deployed. Therein lies the elegance of pragmatic work.

Emotive and abstract

The following is a quotation from Chris Jordan, *TED2008* (http://www.ted.com/talks/chris_jordan_pictures_some_shocking_stats.html)

> *"I have a fear that we aren't feeling enough, we aren't able to digest these huge numbers."*

At the other end of the spectrum are visualizations where the tonal intent is much more emotive and/or abstract in form.

Sometimes you just want to and need to move beyond bars, straight lines, and right angles and more towards curves, circles, and other bendy things. As we will see in Chapter 4, *Preparing and Familiarizing with Data*, there are consequences to this choice, in respect of the known reduction in the accuracy of value perception this will cause. That is a sacrifice you as the designer need to juggle and justify.

Abstract visualization, in terms of its tone, is more about creating an aesthetic that portrays a general story or sense of pattern. You might not be able to pick out every data point or category, but there is enough visual information to give you a feel for the physicality of the data.

This next image is taken from a project to visualize the global airline transportation network consisting of all commercial flights worldwide. The routes highlighted are those flights in and out of Toronto Pearson airport. The project was designed to assess the threat of infectious diseases.

Image from "Toronto Flight Lines" (http://www.biodiaspora.com/) created by Bio.Diaspora 2012

The design does not intend to offer an analytical summary of air travel statistics. Instead it creates a more immersive experience in to the data, offering a visual interface to establish a greater sense of how interconnected the world is through air travel. It causes us to imagine just how easy it could be for diseases to spread across the globe in a short period of time.

For more emotive visualizations, you might be seeking to generate a different type of emotional connection with the design. This connection comes both at the start of the engagement—creating attraction and appeal—as well as after the engagement—the outcome.

In contrast to pragmatic works, as we described in the introduction, here we might be seeking to achieve impact, to emphasize issues, and perhaps to engender shock. We might also seek to generate a certain amount of visual attention through novelty and innovation in a way that more pragmatic approaches would not be able to achieve.

In the following image, we see a section taken from a newspaper infographic that depicted Iraq's bloody toll. While the chart method is nothing more complex than an upside down bar chart, the tone is very impactive and metaphorically emphatic, creating a strong emotional impact to the story it portrays:

Iraq's bloody toll

The United States officially marked the end of almost nine years of bloody military engagement in Iraq on Thursday. Over 4,800 coalition soldiers and tens of thousands of Iraqis lost their lives in a war that defined a decade.

The biggest killers

How civilians died, according to the data available*

Coalition fatalities by area

Baghdad and the vast desert province of Anbar saw some of the fiercest fighting, the latter being the scene of sectarian tensions and a bloody Sunni insurgency

SCMP graphic: Simon Scarr

*Deaths from unknown causes are not included. Causes of death accounting for less than 0.5 per cent of killings also not included.

Sources: Casualties, Iraq Body Count, New England Journal of Medicine, Global Security, Brookings Institution

Image from "Iraq's Bloody Toll", published in the South China Morning Post on Saturday, December 13, 2011 (http://graphics-info.blogspot.hk/2012/09/malofiej-20-look-at-our-participation.html), created by Simon Scarr.

Of course, it is important not to stretch the functional and tonal responsibility and capability of visualization too far. This is where any hype and disproportionate expectation about the potential impact of data visualization can be misplaced.

Data visualization is a means to an end, not an end in itself. It's merely a bridge connecting the messenger to the receiver and its limitations are framed by our own inherent irrationalities, prejudices, assumptions, and irrational tastes. All these factors can undermine the consistency and reliability of any predicted reaction to a given visualization, but that is something we can't realistically influence.

All we can do is form a best judgment about where on the continuum of design style, from a pragmatic experience through to an emotive one, the purpose of our data visualization will be most suitably defined. The ultimate responsibility for what happens beyond the visualization engagement sits with the reader or the user.

Key factors surrounding a visualization project

The following is a quotation from Edward Tufte's book, *The Visual Display of Quantitative Information*:

> *"Most principles of design should be greeted with some skepticism... we may come to see only through the lenses of word authority rather than with our own eyes."*

While establishing the purpose of the visualization project sets the desired tone of the design and its function, there are inevitably many other factors that will have a significant influence on the shape and direction of our visualization design.

It is especially important to identify and recognize the impact of the contextual conditions, within and around your project that will affect what you can and can't achieve and how you might achieve it.

This list of factors may seem quite obvious and fairly rudimentary, but if we wish to eradicate the likelihood of misjudgments or misunderstandings, and maximize the efficiency and effectiveness of the process, we need to nail them early on.

There is simply no point waiting until it is too late to consider these, because by then you will have already followed a certain path and spent valuable time and resources on your work.

Here are some of the most important factors to consider and to evaluate their potential impact:

- **The aim**: As we have seen already, there are different origins and triggers for a project. We mentioned the self-initiated ones as being almost free of external constraint and essentially framed by our own capabilities and intentions. The important thing worth reinforcing here is the need to take responsibility when a project involves a brief, commissioned by a client or a colleague. You must demonstrate excellent communication skills to ensure you seek and gather as much of an understanding as possible of what it is they are aiming to achieve. Sometimes, you might be provided with a very open brief because a client may not even know what it is they are seeking. In these situations, your responsibility needs to extend to assist them in the scoping and requirements of the work. On other occasions you will be asked to create something that goes against your general practice (for example, the subject matter or requested style) or it might even be simply impossible to deliver (perhaps due to the desired design or available resources). Here again your communication skills are going to be required to manage the expectations. It is easy to be shy and delay asking vital questions but this will only cause you pain later.
- **Time pressures**: Common to just about every commissioned design project will be the pressure of time and deadlines. Most projects have clear timescales, from in-day quick turnaround pieces to longer-term grand projects. The challenge of maintaining objective creativity in the face of diminishing time is something that will severely test designers of all experiences. Whatever your situation, you have to use your time effectively and that's where value will come from following the tactics in this methodology. Plan your work and create a balanced layout of the things you need to accomplish, so that you avoid disproportionately spending time on tasks that are less

important than others. Often you will find yourself undertaking a visualization project in parallel with many other commitments. Not only will your capacity be limited, the momentum and duration of your focus will be impacted. This is where project management skills come to the fore as well as a realistic appreciation of what you should and shouldn't commit to undertaking. It also highlights the importance of keeping notes so that you can move seamlessly between projects and not lose track of your thoughts, ideas, or progress.

- **Costs**: The issue of financial resource will unquestionably emerge, especially for large-scale projects. Costs will significantly influence the time you are able to commit to a project, the scope for bringing in additional collaborators, and the range of tools or technical resources you might be able to utilize. Once again, the planning and preparation stages will be invaluable to surface all potential issues around financial matters.
- **Client pressures**: Aside from time pressures, you need to anticipate and reduce the impact of potential unexpected pressures and interruptions coming from your client or colleagues. This might be changes in requirements, new demands, interference in the design solution, and generally annoying things that get in the way of your progress. A further manifestation of the pressure that can come from clients is the insistence on observing organizational visual or brand identities, layout rules, editorial guidelines, and technical frameworks. All of these will shape the scope of your design choices. You have to be prepared for and capable of managing this relationship, and the mutual expectations, effectively so always be open with your client, keep them regularly updated with progress and, where applicable, involve them in the key decision moments throughout the process.
- **Format**: From a design perspective this will be a significant influencing aspect. Are you creating a static or an interactive design? Maybe it's a multifaceted project and you are looking to create both. If it is an interactive design, what platform do you need to achieve compatibility with? Will it be for the Web, a tablet, and/or smartphones? If it is a static design, will it be a small graphic in a publication, a full-page spread, or a large poster display? Maybe it will be a video animation or an ambient display out in the wild, or a large touch screen installation in a museum. This is a vital consideration that needs to be cleared up at the earliest possible stage. Another factor to take into account will be the likely frequency of the project—is it a one-off piece or will it be something that needs to be replicable and/or scalable? That could hugely affect what you can or can't deliver.
- **Technical capabilities**: Aside from your own technical capabilities, what are the technical resources to which you have access? For example, are you limited to free tools or can you access more premium software? Do you have the most appropriate technical infrastructure, such as server speed and capacity if it is an online project? Depending on your format choices, what frameworks are you going to deploy, what browsers do you need to have it working on, what backend database technologies are you going to require? This is a wide-ranging and very technical set of decisions that will likely require a specialist technician to determine.

The "eight hats" of data visualization design

The final scoping issue to consider at this stage of your visualization design project is an assessment of your personal capabilities and those of any collaborators that you involve in the work. What skills and knowledge do you collectively possess or lack? This is a big issue for many, so we need to spend the remaining pages of this chapter looking at it closely.

The demands on a visualization designer in terms of capability are many, reflecting the truly multidisciplinary nature of the subject. The convergence of different ingredients introduces a wonderful richness and variety of issues to be concerned with, but it can equally present quite a challenge for people looking to master the subject.

For many, the prospect of trying to acquire the necessary array of knowledge and skill across the entire range of capabilities is something that can be intimidating or at least exist as a perceptual barrier. There is a sense that to be successful you need to be some sort of superhero.

Taking an analytical look at the range of required capabilities reveals a role and need for many types of people, which can of course be fulfilled by a number of people or just one.

These are proposed as the "eight hats of data visualization design". Influenced by the concept of Edward de Bono's six thinking hats, which related to the different thinking perspectives we should try to occupy when tackling complex problems, this is an attempt to organize the different attributes required to accomplish success in visualization.

It should help you recognize where you fit it in to the spectrum of duties and responsibilities, helping you identify your strengths and your weaknesses accordingly. You may then choose to address these weaknesses personally or plug the gaps with support from others.

The initiator

The **initiator** is the leader, the person who is seeking a solution to the task as per the brief or self-initiated curiosity. The hat is that of an explorer; they want to explore data and different design avenues to find answers to problems or evidence to serve their researcher mindset. The initiator will be responsible for much of the considerations covered in this chapter. They will establish the functional and tonal direction of the project, as well as identify and profile the target audience. The initiator will also define other parameters such as the intended format/platform of the solution and some of the key technological issues.

The data scientist

The **data scientist** is characterized as the data miner, wearing the miner's hat. They are responsible for sourcing, acquiring, handling, and preparing the data. This means demonstrating the technical skills to work with data sets large and small and of many different types. Once acquired, the data scientist is responsible for examining and preparing the data. In this proposed skill set model, it is the data scientist who will hold the key statistical and mathematical knowledge and they will apply this to undertake exploratory visual analysis to learn about the patterns, relationships, and descriptive properties of the data.

The journalist

The **journalist** is the storyteller, the person who establishes the narrative approach to the visualization's problem context. Working with the data scientist and the initiator, they are able to establish the key stories and angles with which to slice the analysis. They work on formulating the data questions that help keep the project's focus on its intended editorial path. Building on the initiator's initial sparks of ideas, the journalist will develop a deeper researcher mindset to really explore the analytical opportunities.

The computer scientist

The **computer scientist** is the executor, the person who brings the project alive. With their critical technical capability they are ultimately the ones who will construct the solution. They will also bolster the data scientist with their technical know-how to most effectively and efficiently handle the data gathering, manipulation, and pre-production visualization activities. The breadth of software and programming literacy will have a great bearing on the potential direction and sophistication of the data visualization solution, whether this is created within a tool or through programming.

The designer

The **designer** is the creative, the one, who, in harmony with the computer scientist, will deliver the solution. They have the eye for visual detail, a flair for innovation and style and are fully appreciative of the potential possibilities that exist. However, they also have the necessary discipline to follow the message established by the initiator and taken on by the journalist. They respect the capabilities of the computer scientist in terms of what solutions could be feasible, but themselves have the helicopter-like vision to rationalize and reason what things will work and will not work, and why.

Their key responsibility is also to be capable of ensuring the harmony of the solution between its form and its function, ensuring it is aesthetically appealing to draw in the reader while fundamentally delivering the intended, communicated message.

The cognitive scientist

The **cognitive scientist** is the thinker in terms of appreciating the science behind the effectiveness of the technical and designed solutions. They have the visual perception knowledge about how the eye and the brain work most effectively and efficiently. They also have deep knowledge about concepts such as the Gestalt Laws, communication theories, color theories, and human-computer interaction principles. Additionally, they are able to inform the design process in relation to the complexities of how the mind works in terms of memory, attention, decision-making, and behavioral change.

The communicator

The **communicator** is, naturally, concerned with the communication side of the project. With their hard hat on, they act as the negotiator and presenter, operating at the client-customer-designer gateway, helping to inform all those who are involved on progress, requirements, problems, and solutions. The communicator needs to be close to all stages of the process, understanding requirements, appreciating restrictions, recognizing possibilities, and then ultimately launching, publicizing, and showcasing the final work. An ability to articulate and explain matters to different types of people, technical and non-technical, and be capable of managing expectations and relationships is vital.

The project manager

This final role is essentially that of the manager or coordinator, the person who does much to pick up many of the unpopular duties to help bring the whole project together. They manage the project's process and its progress, ensuring it is cohesive, on time, and on message. They understand the brief and identify/manage all the key factors surrounding the project. Ultimately, this role is required to ensure things get finished, so they need to have an eye for detail, the commitment and patience to check everything and they should also be concerned with integrity matters around visualization ethics.

Summary

In this chapter, we have started our journey through the data visualization methodology. The emphasis has been on the importance of planning, preparation, and scoping our project, before we embark on any design work. Without this early work we could undermine the effectiveness and efficiency of our eventual design process: something any designer can ill afford to allow.

We have seen how data visualization is a means of facilitating the discovery of patterns and relationships that exist within data. These are insights that would otherwise be practically impossible to draw from data in its raw state.

The importance of establishing the purpose of our visualization project was the key part of this first stage. Specifically, we highlighted the distinction between functional intent and tonal intent.

Within these characteristics we described the difference between visualizations that are functionally seeking to explain, explore, or exhibit data. Furthermore, we saw the significance in potential design differences between visualization styles that serve a pragmatic tone and those that are more emotive or abstract.

As we will appreciate throughout the remaining chapters, developing the clarity of our purpose at this early stage is paramount to the success of our visualization design process. The choices we make fundamentally influence our design choices and the potential experience of our target audience.

We explored some of the key factors that can have a strong influence on the shape and scope of our visualization project. Whether it is the technical matters, the issue of format, financial resources, or timescales, each factor mentioned can have a huge impact on your creative path and scope.

Finally, we looked in depth at the range of personal capabilities required to successfully deliver a visualization design and drew attention to how you might need to personally address any gaps through development or collaboration.

In the next chapter we'll look at two further important stages of planning and preparation: identifying your intended narrative and getting intimate with your data.

Chapter 3. Demonstrating Editorial Focus and Learning About Your Data

In the previous chapter, we introduced the data visualization methodology, starting off with a look at two important preparatory activities: establishing the project's purpose and identifying the influencing factors surrounding the project.

It is worth acknowledging that the intention of our visualization may evolve, particularly as we journey through the upcoming design phases and as new influencing factors emerge. Any decisions we make across this process can be revisited and refined but the greater clarity we achieve now will ultimately help minimize wasted efforts and lead to a more efficient process.

In this chapter, we move on to the next phase of the methodology where we look at the intertwining issues of the data we're working with and the stories we aim to extract and present. This activity provides a bridge between project inception and design conception and involves the following tasks:

- We will look to develop and refine our editorial focus around the key communication dimensions of our visualization problem: What is the story we are trying to tell? What is the key narrative we are looking to portray? What questions do we wish readers to be able to answer through the visualization?
- One of the biggest challenges, and usually the most time-consuming, is the acquiring and preparing of the data, ensuring it is fit for purpose, and in good shape in advance of the design stage. We'll explore the mechanics of working through this often hidden activity.
- Finally, we'll see an example of how we can use visual analysis techniques to combine the task of familiarizing with our data and discovering key insights. We will show how learning about the physical properties of data helps you develop your editorial focus, specify your data questions, and influence the potential design choices we make later on.

The importance of editorial focus

The following is a quote from Edward Tufte (http://adage.com/article/adagestat/edward-tufte-adagestat-q-a/230884/):

> *"Good content reasoners and presenters are rare, designers are not."*

In Chapter 2, *Setting the Purpose and Identifying Key Factors*, we looked at some of the considerations involved in identifying the purpose behind your visualization project; these are as follows:

- What is the reason for its existence?
- For whom are we creating it and how well defined are the requirements?
- What function is it seeking to fulfill?
- What is the likely tone of the design we're intending to portray?

Over the course of the full design process, it is possible that these initial definitions may need to be modified. As we learn more deeply about the relationship between what we want to do, what we can do and, importantly, what we should do, our creative proposition may be molded into a slightly shape.

That's fine and is to be expected. However, the earlier we can make firm judgments on our creative direction the better. This gives us a solid starting point and helps inform the important decisions we need to make about what it is we are trying to say with the visualization we are developing.

The matter of *how* this is said will be covered in the design stage but, ahead of that work, we first need to determine *what* are the specific messages we are looking to communicate to our audience.

Some of the most influential and esteemed visualization and infographic design work, perhaps unsurprisingly, comes from newspaper and magazine organizations.

The New York Times would probably be at the top of many peoples' list of the most celebrated graphics work, but there are so many other examples of great innovation and excellence from across the industry and right around the world, including The Guardian (UK), National Geographic (US), the Washington Post (US), the Boston Globe (US), La Informacion (Spain), and Época (Brazil), to name but a few.

A key reason behind the success of the work produced by these departments is the demonstration of what Edward Tufte describes at the beginning of this section—editorial focus.

Regardless of the size and inherent complexity of the data challenge you are working on, this is one of the most important capabilities you will need to develop in order to succeed in data visualization and is something that can singularly influence the success, or otherwise, of a design.

An editorial approach to visualization design requires us to take responsibility to filter out the noise from the signals, identifying the most valuable, most striking, or most relevant dimensions of the subject matter in question.

To do this we need to weigh-up the potential appetite of the intended audience—what it is we think they will want to know or will find interesting—and the opportunities that exist within the data—what data stories can you find and might you portray.

Determining what an audience needs is not always straightforward, particularly when you might have a broad range of different types and background of readers engaging and interpreting your work. Nevertheless, you should still have a sufficiently sympathetic view of how your target demographic will most positively and constructively relate to different slices of analysis of your subject matter.

For projects triggered by a client or colleague, there may be specific analytical dimensions that are already established and you have been asked to present and communicate them. The scope for veering away from this existing focus may not exist.

Otherwise, irrespective of whether you are tasked with the work or just pursuing a self-initiated curiosity, in most cases, you will have a certain degree of liberty to undertake the dual role of analyst and storyteller influencing the selection of what you will portray.

The execution of a design is clearly hugely significant to the success of a project, but without the foundation clarity and justification for the message you are trying to communicate, your resulting visualization will fundamentally lack focus.

Rather than just throwing everything available at a reader, good visualization involves showing a degree of editorial care—just because you have some data, doesn't mean to say you have to use it all. Be selective.

This attitude is necessary for all types of visualization projects. You might think the idea of telling stories is only relevant for explanatory pieces. That's not the case. With exploratory designs you still need to demonstrate this editorial focus. The difference is that with these projects you are not so much telling stories rather you are making them accessible and discoverable. You still need to frame the subject matter and define the important dimensions of analysis that will be made available for manipulation and interrogation. You still need that level of care for the audience's interpretive experience.

As we'll see later, some of the most effective data visualization designs manage to create a combination of these functional characteristics, offering a sweet spot of engaging exploratory features framed within defined story dimensions.

Conversely, if you take a look at a gallery of visualization work and find examples that you believe are ineffective, they will likely exhibit a weak narrative, an absence of stories, and a lack of genuine care for the interpretive needs of the audience. This is a really influential dimension of visualization design.

Preparing and familiarizing yourself with your data

The following is a quote from Simon Rogers, *The Guardian*, *Facts Are Sacred: The Power of Data*:

> *"80% perspiration, 10% great idea, 10% output."*

Before we get too far down the line of developing and defining our intended stories and analytical slices, we need to roll up our sleeves and get our hands dirty with the task of accessing and preparing our data.

Whether you get the data first or shape your desired story dimensions first is mainly going to be influenced by the context of your project. It is a somewhat "chicken and egg" situation—which comes first, the data or the focus? You need some focus to determine what data you need, but you don't know what potential insights exist in the data until you have it.

It is best to accept that there will be a certain amount of iteration as you alternate between the mindset of a data scientist and a journalist progressing both issues simultaneously.

Data is our raw material, the principle ingredient in our creative recipe. Irrespective of what we intend or hope to show through our visualization design, the data will ultimately do the talking.

If we don't have the data we want, or the data we do have doesn't tell us what we hoped it would, or the findings we unearth aren't as interesting as we wish them to be there is nothing we can (legitimately) do about it. That is an important factor to remember. No amount of 3D-snazzy-cool-fancy-design dust sprinkled on to a project can change that.

An incomplete, error strewn or just plain dull dataset will simply contaminate your visualization with the same properties. So, the primary duty for us now is to avoid this happening, remove all guessing and hoping, and just get on with the task of acquiring our data and immerse ourselves into it to learn about its condition, its characteristics, and the potential stories it contains.

To achieve this, we must go through the often painful mechanics of data familiarization and preparation; as follows:

Acquisition: First, you need to get hold of your data. As we have discussed, this might already be provided to you from those commissioning the work. You might have independently formed a sense of the specific subject dimensions on which you require data. Alternatively, it may be that you have yet to focus beyond a broad subject level. It all really depends on how well-defined your requirements or intentions already are.

The places where you might acquire your data and the methods to accomplish it will be something you will know best. It could come from origins such as these:

- Obtained from a colleague, client, or other third-party entity
- A download taken from an organizational system
- Manually gathered and recorded
- Extracted from a web-based API
- Scraped from a website
- Extracted from a PDF file (you have my sympathies)

Here, we see an image taken from a visualization project that was created to demonstrate the social expansion of the US using the story of the spread of post offices. In this screenshot of the final piece, we see a representation of the 11,000+ post office locations recorded across the country between 1700 and 1900:

Image from "Posted: Visualizing US Expansion Through Post Offices" (http://blog.dwtkns.com/2011/posted/), created by Derek Watkins

The entire data for this project was scraped from the US Postal Service website. After cross-referencing the dataset with a gazetteer to establish accurate geo-locations, almost 1,500 records (12 percent) had to be discarded, as they weren't readily "mappable".

This just shows the great amount of effort and pain that often goes in to sourcing and preparing your

data. No matter from where you are accessing your data, you will often have to work hard to get it into the shape and form that you need it. Therefore, you need to ensure you have factored in as much time as possible for this vital stage of the process.

Examination: Once we've got the data, a thorough examination will determine your level of confidence in the suitability of what you have acquired. This involves assessing the completeness and fitness of the data to potentially serve your needs. There are many tools out there that can help you work through this stage efficiently. Depending on the size and complexity of your data, and obviously your own capabilities, software like Excel, Tableau, or Google Refine (among plenty of others), will enable you to quickly scan, filter, sort, and search through your dataset to establish its state of quality. As you go through this process, you should be examining the following potential issues:

- **Completeness**: Is it all there or do you need more? Is the size and shape consistent with your expectations? Does it have all the categories you were expecting? Does it cover the time period you wanted? Are all the fields or variables included? Does it contain the expected number of records?
- **Quality**: Are there noticeable errors? Are there any unexplained classifications or coding? Any formatting issues such as unusual dates, ASCII characters? Are there any incomplete or missing items? Any duplicates? Does the accuracy of the data appear fine? Are there any unusual values or obvious outliers?

Data types: Understanding the properties of our raw material is such an important task. We will do some visual exploring later to learn about the physical patterns and relationships but, for now, we need to understand the fundamental structure of our data in terms of the variables types. This will become important when we move into the design discussion in Chapter 4, *Preparing and Familiarizing With Data*. The following table outlines the discrete types of data with associated examples:

Types	Examples
Categorical nominal	Countries, gender, text
Categorical ordinal	Olympic medals, "Likert" scale
Quantitative (interval-scale)	Dates, temperature
Quantitative (ratio-scale)	Prices, age, distance

As well as capturing the types of data we have, it is a useful exercise to also make a note of the range of values or at least a sample of the data held against each field. For illustration, this might be from a dataset about the Olympics:

Data	Types	Range

Event	Quantitative (interval-scale)	27 different years (1896–2012)
Medal	Categorical ordinal	Gold, silver, bronze
Athlete	Categorical nominal	1500+ different athlete names
Result	Quantitative (ratio-scale)	Race results (9.59s > 4:02:59)
Country	Categorical nominal	96 different country names

Transforming for quality: This task is naturally about tidying and cleaning your data in response to the examination stage above. We are looking to resolve any of the errors we discovered in order to transform the condition of the data we're going to be working with for our design. Plugging the gaps caused by missing data, removing duplicates, cleaning up erroneous values, and handling uncommon characters are just some of the treatments we may be required to apply.

Transforming for analysis: In contrast to transforming for quality, we move away from cleaning data and focus more on preparing and refining it in anticipation of its intended use for analysis and presentation. Here, we consider actions such as:

- Parsing (split up) any variables, such as extracting *year* from a date value
- Merging variables to form new ones, such as creating a whole name out of *title*, *forename*, and *surname*
- Converting qualitative data/free-text into coded values or keywords
- Deriving new values out of others, such as *gender* from *title* or a sentiment out of some qualitative data
- Creating calculations for use in analysis, such as percentage proportions
- Removing redundant data for which you have no planned use (be careful though!)

Another important consideration is to determine what level of resolution you might wish to, or indeed need to, present your data. The decision you take about this may require you to aggregate or disaggregate your data to achieve get the right level of detail.

Design agency Periscopic were faced with some intricate resolution decisions in their preparatory work for this near real-time visualization developed about the Yahoo! Mail network. The objective was to show the huge volumes of e-mails being sent and processed around the world at any given point, and the efforts Yahoo! is taking to help reduce and intercept spam e-mails. This is shown in the following screenshot:

Image from "Visualizing Yahoo! Mail" (http://www.periscopic.com/#/work/yahoo-mail/), created by Periscopic

With approximately 5.6 billion e-mails (and a further 20.5 billion spam) sent every day, the sheer amount of data potentially being fed into this project clearly posed a challenge in terms of what level of detail they could reasonably show.

This was not just a matter of how they could handle the velocity and volume of new data on the technical side but also what was the appropriate resolution with which to tell this story

They decided on the following strategy:

- The headline statistics shown in the titles and presented across a range of supplementary graphics across the project would be representative of the full data quantities.
- For the geo-spatial view, a carefully designed algorithm was applied to extract a representative sample of data. This would be more than adequate to capture the nuances of the activity seen with the full dataset and would avoid the technical impracticalities involved in attempting to show 100 percent of the data.
- The geographical data was clustered to a city or regional aggregate, represented by the circle positions and sizes, to help draw out the key signals and patterns.

This is a perfect demonstration of how important it is to handle data resolution issues as early as possible so we know what treatment to apply to our data.

When you are faced with similar decisions, albeit perhaps rarely on the same scale, you will typically have these options available to you:

- **Full resolution**: Plotting all data available as individual data marks.
- **Filtered resolution**: Exclude records based on a certain criteria.
- **Aggregate resolution**: "Roll-up" the data by, for instance, month, year, or specific category.
- **Sample resolution**: Apply certain mathematical selection rules to extract a fraction of your potential data. This is a particularly useful tactic during a design stage if you have very large amounts of data and want to quickly develop mock-ups or test out ideas.
- **Headline resolution**: Just showing the overall statistical totals.

Consolidating: When you originally access your data, you will likely believe, or hope that you have everything you need. However, it may be that after the examination and preparation work, you identify certain gaps in your subject matter.

Additional layers of data may be required to be combined ("mashed-up") with our existing dataset, applied to perform additional calculations, or just to sit alongside this initial resource to help contextualize and enhance the scope of our communication. Always spend a bit of time considering if there is anything else you anticipate needing to supplement your data to help frame the subject or tell the stories you want to communicate.

Seasoned designers will confirm that acquiring, handling, and preparing your data is often the most time-consuming and intensive activity involved in any visualization project.

It is the hidden battle. As Simon Rogers quantifies at the start of this section, if you imagine a visualization design project as an iceberg, the final design would be the bit we see sticking out of the water and the ugly data preparation work would be the vast amount hidden beneath the surface.

There is a good chance that you will have expended most perspiration in the many thankless, uncelebrated duties you have to undertake in this part of the process. However, just know that the value of your efforts and the associated rewards will emerge in due course, so try not to lose enthusiasm or patience.

Refining your editorial focus

The following is a quote from Amanda Cox, New York Times (http://seekingalpha.com/article/66269-an-amazing-graphic-on-box-office-receipts):

> *"Different forms do better jobs at answering different types of questions."*

Now that we have prepared our data, we revisit the matter of editorial focus.

To avoid being prematurely tempted into diving into the construction of a visualization design, we first need to do more work to fine-tune our analysis of what are the important messages.

In the first section, we explained the importance of taking responsibility to make sense of data, to find stories and tell stories to your intended audience. This demonstrates a level of care. It shows that you are not just going through the motions of communicating; you are taking it seriously, seeking to help your audience unlock insights from the subject matter.

The journalistic capability for unearthing the most relevant stories from data is a talent that any designer should aspire to develop.

In the example shown in the following screenshot, we see a recent visualization project that was developed to enlighten people about the matter of education around the world, presenting some striking facts and figures:

Image from "In Numbers: Education Around the World" (http://visualdata.dw.de/specials/bildung/en/index.html), created by Gregor Aisch for Deutsche Welle.

As you might imagine there will be myriad ways of telling data stories about global education matters. In such contexts, a designer is faced with the challenge of rationalizing so many different potential dimensions.

The strength of this particular project comes from the scoping and definition of the chosen narrative and slices of analysis. Rather than bombarding the reader with endless pages of facts and figures, or offering seemingly infinite combinations of interactive variable selections, the subject is framed for us around a small number of interesting angles about education: literacy by region, literacy rates by country/gender, enrollment ratios, and expenditure on education versus military.

As we then navigate through each story panel we are presented with a series of explanatory visualizations. They don't just show data, they present and explain it.

Image from "In Numbers: Education Around the World" (http://visualdata.dw.de/specials/bildung/en/index.html), created by Gregor Aisch for Deutsche Welle

In this example, we see a scatter plot of education spend versus military spend for all countries. But it is more than just a plot. The designer takes responsibility for telling the story, providing effective written (labeling and captions), and visual annotation (reference lines and background shading) to help maximize the potential insights. The inclusion of filtering features to highlight particular countries and regions introduces an exploratory dimension to enable the discovery of further layers of understanding.

This is a strong demonstration of editorial focus and storytelling with data—four key stories, elegantly told.

What we see with this project is a visualization that answers "data questions". Data questions are the lines of interrogation and the dimensions of interpretation users will likely seek to pursue when reading a visualization design.

It's more than just framing a story, it is about the specific insights we are making accessible. It is the most defined and detailed level of editorial focus we should aim to achieve. We want our

visualization to be able to respond to the most likely and relevant questions a user will raise about the data and the subject matter.

At this point, we are starting to consider the relationship between our editorial focus and the potential visualization design options.

As Amanda Cox describes earlier, the way that you choose to represent your data—the form you give it through your selection of chart type—should be influenced by the questions you are trying to answer.

For instance, if you are asking a chart to facilitate a comparison between the values of different categories, you might deploy a bar chart. You wouldn't use a line chart to achieve this, but you would if you wanted to show how a value or values change over time. The scatter plot we just saw was the perfect method of comparing two quantitative values for all those different countries. It was the right form to answer the specific data questions identified.

So we need to know what questions we're trying to answer.

Unless you've already had them specifically outlined to you, an effective approach to tackling this can be drawn from the practice of logical reasoning, specifically induction and deduction. These techniques are common to academic and scientific research.

Deductive reasoning involves confirming or finding evidence to support specific ideas. It is a targeted and quite narrow approach concerned with validating certain hypotheses. A deductive approach to defining your data questions will involve a certain predetermined sense of what stories might be interesting, relevant, and potentially available within your data. You are pursuing a curiosity by interrogating your dataset in order to substantiate your ideas of what may be the key story dimensions.

Inductive reasoning works the opposite way. It is much more open-ended and exploratory. We're not sure precisely what the interesting stories might be. We use analytical and visualization techniques to try and unearth potentially interesting discoveries, forming different and evolving combinations of data questions. We may end up with nothing, we may find plenty—the insights we observe may be serendipitous as we follow our nose for the scent of evidence. Fundamentally, this is about using visual analysis to find stories.

For most visualization projects, if we have the time, ideally we would seek to use both deduction and induction in conjunction in order to learn as much as possible about what stories the dataset can reveal about the given subject matter.

Using visual analysis to find stories

The following is a quote from Ben Schneiderman:

> *"Visualization gives you answers to questions you didn't know you had."*

In the Chapter 2, *Setting the Purpose and Identifying Key Factors*, we discussed the different intentions and motives you might have for developing a data visualization. In most cases we think of it as something we create and provide to others. What we sometimes neglect to consider is the potential of visualization for ourselves, when we are the intended users looking to discover insights about a subject.

This is where we consider the application of visual analysis. Visually analyzing a dataset, and employing both inductive and deductive reasoning, enables us—as the designer—to learn more about our subject by exploring a dataset from all directions.

As Ben Schneiderman articulates above, and as we saw through the demonstration of Francis Anscombe's experiment, rather than just looking at data, we are using visualization to actually see it, to find previously undiscoverable properties of our raw material, to learn about its shape, and the relationships that exists within.

This activity can also be described as data sketching or preproduction visualization. We are using visualization techniques to become more intimate with our raw material and to start to form an understanding of what we might portray to others and how we might accomplish that.

Visual analysis requires a high degree of graphical literacy, the ability to read and interpret data represented visually. This is something we might not really think about too often. In fact, if we're honest, many of us would probably have to admit that we can actually be quite passive in how we engage with a visualization or infographic.

This activity requires a much more committed level of attention to interpretation. As we explore the evolving visual analysis of our data, we need to be prepared to observe the following characteristics that will lead to the identification of our key stories:

Comparisons and proportions:

- **Range and distribution**: Discovering the range of values and the shape of their distribution within each variable and across combinations of variables
- **Ranking**: Learning about the order of data in terms of general magnitude, identifying the big, medium, and small values.
- **Measurements**: Looking beyond just the order of magnitude to learn about the significance of absolute values
- **Context**: Judging values against the context of averages, standard deviations, targets, and forecasts.

Using methods like a bar chart will enable comparison across values and categories to pick out the type of physical qualities just listed, as shown here:

Trends and patterns:

- **Direction**: Are values changing in an upward, downward, or flat motion?
- **Rate of change**: How steep or flat do pattern changes occur? Do we see a consistent, linear pattern, or is it much more exponential in shape?
- **Fluctuation**: Do we see evidence of consistent patterns or is there significant fluctuation? Maybe there is a certain rhythm, such as seasonality, or perhaps patterns are more random
- **Significance**: Can we determine if the patterns we see are meaningful signals or simply represent the noise within the data?
- **Intersections**: Do we observe any important intersections or overlaps between variables, crossover points that indicate a significant change in relationship?

Using a line chart is a perfectly suitable method to observe patterns and trends, as we see below:

Relationships and connections:

- **Exceptions**: Can we identify any significant values that sit outside of the norm, such as outliers that change the dynamics of a given variable's range?
- **Correlations**: Is there evidence of strong or weak correlations between variable combinations?
- **Associations**: Can we identify any important connections between different combinations of variables or values?
- **Clusters and gaps**: Where is there evidence of data being "bunched"? Where are there gaps in values and data points?
- **Hierarchical relationships**: Determining the composition, distribution, and relevance of the data's categories and subcategories.

Using a scatter plot will enable visibility of these types of relationships, as shown below:

Through embarking on such in-depth visual analysis we should achieve the level of data intimacy required to refine our editorial focus. The visual interrogations we perform on the data will unearth evidence of the features listed over the previous couple of pages.. Where we find these, we will find the stories.

The process of visual analysis can potentially go on endlessly, with seemingly infinite combinations of variables to explore, especially with the rich opportunities bigger data sets give us. However, by deploying a disciplined and sensible balance between deductive and inductive enquiry you should be able to efficiently and effectively navigate towards the source of the most compelling stories.

The chart types that we have seen being used previously are illustrative of just a small section of the gallery of options we have to call upon. We will learn much more in Chapter 5, *Taxonomy of Data Visualization Methods*, about the different chart types and their functions to understand which ones are best deployed for different enquiries of our data.

The product of our work here is a more sophisticated understanding of the stories existing in our datasets about the given subject matter. This will help us form the specific data questions that we'll be

asking our visualization designs to answer. We've found our stories, now we need the appropriate methods to tell them and that's what Chapter 4, *Conceiving and Reasoning Visualization Design Options*, will explore.

An example of finding and telling stories

Before we move on, to help embed the understanding of data familiarization, visual analysis and the difference between finding stories and telling stories, let's work through a basic example.

Take the following sample table of data. The subject matter is the Olympic games and specifically the total medals won by the top eight participating nations over five recent events. The selection of the top eight is based on them being the top ranked countries at the Beijing Olympics in 2008.

Suppose you were briefed to unearth some key stories around Olympics medal winning trends in recent years, how would you go about it?

	Total medals won in the Summer Olympics				
Country	**2008**	**2004**	**2000**	**1996**	**1992**
United States of America	110	103	92	101	108
People's Republic of China	100	63	59	50	54
Russian Federation	72	92	88	63	112*
Great Britain	47	30	28	15	20
Australia	46	49	58	41	27
Germany	41	49	56	65	82
France	40	33	38	37	29
Republic of Korea	31	30	28	27	29
ALL	951	929	925	842	815

** When part of former Soviet Union. Data from http://www.databaseolympics.com/index.htm*

Let's start by just scanning the data with our eyes to find anything that stands out.

The main data issue appears to be that the Russian Federation medals total for 1992 was actually when it was known as the Soviet Union. It is noticeably higher than for all the other Olympic events, due to the contributions of additional member states that then made up the Soviet Union but who are now independent countries competing in their own right. As it will be hard to unpick this value to isolate just those athletes who would now be considered part of the Russian Federation, it will be sensible to just ignore this value from our analysis. Otherwise, it will skew our interpretations.

We can see that the event order goes from left to right in reverse chronological order and the vertical sorting is organized by the most successful nations as at 2008. In addition to the medal winning totals for the selected countries, we also have the aggregate of all medals across all countries.

We now continue our examination by noting some of the dataset's descriptive and statistical properties to develop an increased level of familiarity:

- Two variables: Country and event year

- Country is a categorical nominal variable with nine values (each country and the aggregate)
- Event year is a quantitative (interval-scale) variable with five values
- The maximum country medal count value is 110 medals, the minimum is 15
- The maximum aggregate value is **951** and the minimum is 815 (but that includes the Russian Federation contribution)
- Each event year is spaced 4 years apart
- The longest country name is People's Republic of China, the shortest is France

This gives us a sense of the physicality of the data and the potential influencing attributes that might shape our visualization architecture.

What other data preparation tasks might we undertake?

We have no real transformation activities to undertake in terms of addressing data quality aside from already deciding to ignore the Russian Federation total.

For transforming the data for its use in analysis we may decide to create some calculations to show the percentage of medals won out of each event total. You may also decide to abbreviate some of the county values to potentially help accommodate the space required for labeling.

We also need to consider data consolidation. For the purpose of this demonstration, we are going to stick to our original dataset on its own but there could be many different options to enhance and contextualize this subject matter, including the following:

- The details behind the medal totals of how many golds, silvers, and bronzes each country has won
- The full dataset of medal statistics for all the other countries who have competed, not just the recent top eight
- The full dataset of medal statistics for every Olympic games
- The number of competitors who were taking part in the games for each country, in order to understand the percentage of success of each team
- The split of performances between the different sporting events
- Population figures to contextualize the achievements, maybe even sporting participation figures if they were recorded
- Historical milestones of socio-political and geo-political issues to help us appreciate the status of the different countries at these key points in time
- You might look to bolster the ingredients of your visualization design resources with national flags' image files or URL links to national Olympic associations

Whether we could obtain these additional data items is another matter and they may not even help with our stories. But it is always good to let your imagination roam and explore ideas for content that could really enhance your work.

Our data is now in good shape. Next up, we look to develop our editorial focus, specifically considering the following:

- What initial sparks of curiosity crossed our minds when we were given the brief and initially saw the data?
- What dimensions of analysis do we think might be of interest or relevance about this subject matter?
- What data questions will we seek to answer in our visualization design?

To refine our focus we need to commence our visual analysis work to explore our dataset and see what comparisons, trends, patterns, and relationships we can identify. Out of this we will hope to unearth some interesting stories to tell.

Given we have a small dataset with only two variables we shouldn't need to embark on too much varied visual analysis.

The first graphic takes a look at the variation of medal winning across the years, showing the range of totals for each country using a floating bar chart:

Through interpreting this chart in conjunction with the descriptive statistics we just collected, we are able to form some interesting data questions about the subject and start to get a feel about the main stories, such as:

Question	**Answer**
Which countries have experienced a significant change in their medal-winning	We're looking for the widest bars to show the variability, this could be improvement, decline or inconsistency. We would identify the spread of Germany and China as being

performance levels?	particularly interesting.
Which countries have maintained consistency in their performance levels?	Now we're looking for the narrowest bars, the tightest of value ranges. This leads to noticing the USA, France, and especially Republic of Korea.
What have been the most interesting country stories in terms of the transition of their performance and rankings?	Possibly too hard to see with this chart, but there is potentially something going on with the bars that intersect and exceed the lengths of others. At this stage, the story of China seems to stand out as being something to look out for.

Let's now repeat the same chart type but apply it to a transformed version of the data that has been standardized to show the medals won as a percentage of the overall total:

Does this alter the focus of our questioning or change our impressions of the main insights?

If anything it reinforces them, especially our interest in the varied performance levels for Germany and China. It also emphasizes the remarkable consistency of Republic of Korea and France.

At this point, we have definitely established a scent for the story. We have started to articulate the data questions that best interrogate this data and most likely reflect what the readers of a visualization about this subject will wish to learn.

We now need a different visual representation. Using the floating bar we have seen the categorical view of the countries and their performances. Now, we need to switch our perspective to the other main variable, that of event year, to pursue our curiosities about the transition of medal-winning

performances and the transition in ranking of the individual countries across the five Olympic Games.

For this next visual sketch we turn to a line chart. On this single chart we plot the eight countries, differentiated by color, showing the absolute medal wins from left to right across the five Olympic events:

It looks a bit messy doesn't it? Don't worry. Remember, this is an exploratory visualization for ourselves. We are the audience and we just want to see if we can discover some interesting physical properties about the data in this display.

You wouldn't and shouldn't publish an isolated, cluttered, and poorly-annotated chart like this to convey a story to others, but when it is a visualization serving yourself, it is a different matter. You created it and you know what you're looking out for. Quick and dirty is absolutely fine.

The decision to place all countries onto one graphic is to enable visibility of the interesting transitions, the crossovers, the seemingly cluttered parts, and the empty parts. You could separate each country out into its own line chart and assess a matrix of eight small-multiples. However, this would only show you the individual country stories. Our keen interest here is in the relationship between the countries.

The chart shows how Germany's (blue) wide range of results, actually reflects their general decline in

medal winning levels and, by extension, their relative rank.

By contrast, China's wide distribution shows a country on the rise over the past four games at least. The extended fascination of this trend would be whether they will catch up and possibly overtake the US once we have the results and data for the 2012 Games (not available at the time of writing this book!).

Elsewhere, Russia can be seen to have moved up and down over the years and has now been overtaken by China. There is an interesting chunk of white space for the 2008 results either side of the Russian value, leaving them quite comfortably in third position. Interestingly, the UK has seemingly demonstrated a very similar pattern of improvement relative to the Chinese over the past five events.

Sometimes no change is as interesting as some change and, in this respect, the consistency of Republic of Korea is quite stark given the different generation of competitors who will have contributed to those totals.

Otherwise there is nothing else really of significant interest. The charts have served their purpose in discovering and confirming some relevant and interesting stories concerning the contrasting experiences of China, Germany and, potentially, the Republic of Korea.

Of course, sometimes you simply may not find a story. There just might not be anything of substance to convey to others visually, in which case a table of data may prove to be the most appropriate solution.

However, we *have* found our stories, so how do we tell them? As a bridge to the next chapter, where we will be focusing on design matters around presenting our stories, let's attempt a quick solution.

Remember the quote we saw earlier from Amanda Cox: "different forms do better jobs and answering different questions"? Let's reduce the story to a simple contrast between China and Germany. Our main data question will be something like "how have the medal-winning performances of China and Germany compared over the past five events?"

The most suitable method for giving form to and answering this question will still be a line chart. Similar to the one we used for the visual analysis, we are trying to show the relationship between these two countries' respective performance over time.

However, the design execution will be different. This time we're conveying the story to others, so we need to refine the visuals in order to make it an explanatory piece:

- We need to elevate the important features of the main story and relegate any background context and secondary content.
- We need to ensure that there are annotations for labels, values, and captions so the reader is entirely clear about what is being communicated.

Here is a proposed solution for telling this story:

The first thing to point out is that we have used the calculated data for medals won as a percentage of the total. This is more appropriate for this story as it helps standardize and contextualize the performance across all events in a more comparable way.

The aim here is to provide a clear visual hierarchy emphasizing the two main countries in our story and diminishing the contextualizing six nations into the background. We could have removed the other six countries but, through the use of a subtle shade of grey, we can still see them well enough to get a sense of the overall rankings. That is all we need from them—context.

The title neatly frames the story, the subheading describes the chart and the data, and the labels help the reader compare the two countries' relative trajectory.

The use of color attempts to help imply the positive improvement (orange = hot = good) of China and the negative decline (blue = cold = bad) of Germany. Only the bare minimum chart apparatus (the axis line) is included, once again, to allow the main story to come to the fore.

Contrast this design approach for telling a story (explanatory) with the design of the same chart method we used to find the stories (exploratory); here we provide nothing more and nothing less than the reader requires to easily interpret the story. This use of contrasting visual approaches for the same chart types but for different intentions is important to recognize in your design work.

Summary

In this chapter, we have learned about the importance of editorial focus and content reasoning—the ability to recognize the most important and relevant stories in your data and the discipline of taking responsibility to optimize the interpreting experience of your intended audience.

We worked through the mechanics of acquiring, preparing, and familiarizing with your dataset. In particular, we highlighted the importance of our own graphical literacy in the task of conducting visual analysis.

We identified numerous physical characteristics of our data that will help us to discover key stories and help inform the types of data questions we may seek to answer in our designs.

Finally, we worked through a demonstration of using visual analysis to make sense of your data, to find stories yourself and then tell those stories to others. We also saw an example of the contrasting visual design solutions used for exploratory and explanatory visualization.

Now that all our preparatory work has been covered, in the next chapter, we will move the methodology forward onto the design stage. Here we will learn about all the options we need to judge and the decisions we need to take across the five key layers that form the anatomy of any data visualization design.

Chapter 4. Conceiving and Reasoning Visualization Design Options

So far the focus of our attention has been on the uncelebrated but vital preparatory and scoping stages of the data visualization methodology. We have established the purpose of our design and the key factors surrounding the project have been identified and weighed-up. We have also acquired and prepared our data and begun exploring it to identify the key data stories and analytical slices around which we may form our editorial focus.

These contextualizing activities are often neglected because they are understandably deemed not as fun as the design stage. Yet, they will save you time and pain, helping your work to proceed more efficiently by avoiding blind alleys and creative misjudgments.

In this chapter, we will be taking a forensic look at the many design choices involved in the process of establishing an effective visualization solution. We will tackle these choices by working through the anatomy of a visualization design, separating our thoughts into the complementary dimensions of the representation and presentation of data.

For rookie and experienced designers alike, the framework of design considerations outlined here should help you successfully navigate through the creative challenges and opportunities you are facing.

Data visualization design is all about choices

Here is a quote from Jer Thorp (http://blog.blprnt.com/blog/blprnt/138-years-of-popular-science):

> *"My working process is riddled with dead-ends, messy errors, and bad decisions—the final product usually sits on top of a mountain of iterations that rarely see the light of day."*

Over the past two chapters, we have explored some of the key preparatory activities of the visualization design process. In doing so we have built a detailed level of clarity about what we want to achieve in our visual communication and why.

The heading for this section may seem obvious but it needs stating. As we'll see in this chapter, the scope for creativity can be quite overwhelming. How well you rationalize the many decisions you face throughout the process will strongly determine whether you achieve an effective visualization design.

To frame this discussion, do a quick image search in Google for the term "data visualization" and scroll through the first few screens. You will see just a snapshot of evidence of the innumerable variety of ways in which you can represent data. Some are good, some are bad. Some are really bad. Some shouldn't even be connected with the term data visualization.

Additionally, have a glance through the collection of submissions uploaded on to sites that run some of the main data visualization design contests (for example, www.visualizing.org, www.infobeautyawards.com). Choose a particular contest and explore the spectrum of proposed solutions, all typically emerging from the same dataset and responding to the same brief.

What can we learn from this? What does this evidence of the variety of ways in which people interpret visualization design challenges reveal to us?

The first thing to say is that there is never a single path towards a "best" solution. The inherent creativity and individualism of design work ensures that. An idealistic desire for a single and simple set of rules to achieve a guaranteed effective solution is simply unreasonable due to the many different factors that will shape the scope and intention of any given project.

There is, however, an established body of theoretical and practical evidence that guides us to understand which techniques work better for certain situations and less well for others. Importantly, these guides transcend instinct or personal taste and help us frame many of our design options, influencing the choices we make.

Beyond that it is more about managing trade-offs, about trusting your judgment to make sense of the problem context in which you are working, the requirements you are responding to, and keeping in mind the overall objectives of visualization design, as outlined in Chapter 1, *Context of Data Visualization*.

The second key observation is to remark that the very moment we take on a visualization challenge, and start our journey towards a design solution, we are commencing a unique creative route formed by numerous permutations of choices. Nobody else will go through the same experience nor arrive at exactly the same solution.

You won't always get there easily. That's important to recognize too. As Jer Thorp expressed in his quote, even the best make mistakes and end up wasting time following ideas that lead nowhere and having to change course halfway through. However, by following the approach we have outlined in this book, and specifically the framework of considerations for this chapter, we hope to reduce the waste and eliminate inefficiency. This allows us to fail faster and recover more quickly.

A useful way to look at a data visualization challenge is to recognize that we are actually seeking to reduce choices. This is achieved through recognizing influential factors, by considering the desired function and tone of our work, familiarizing with our data and identifying stories. We are building clarity through selection and rejection. We are reducing the problem by enhancing our clarity.

The reasoning involved in eliminating options is just as important a skill as determining those we shall pursue. This lets us control our work, it helps us plan better, and prepare for the creative avenues down which we may proceed.

In many ways you could equate this design process with the responsibilities of being a film director, managing the dramatic, artistic, and technical aspects of a film. A director has to create the film's vision, direct the cast, manage the crew, oversee the script, coordinate the choice of locations, the music, and the post-production effects. All these different perspectives require separate attention and unique treatments until they are brought together into a cohesive single product: a movie. We're trying a similar approach with our visualization design.

As we now move into the creative stage it is helpful to follow a framework that will help you to understand the many different design options about which you will have to make a decision.

An effective way to think about this is to consider the visualization "anatomy". By definition, anatomy refers to "the bodily structure of an organism", so we are appropriating the term to apply it to the structural layers of a data visualization design.

In the first chapter, you will recall the proposed definition of data visualization and how it separated the idea of representation and presentation of data. We see these as separate dimensions of our design task:

- **Data representation**: This is the foremost layer, how we give form to our data through the use of "visual variables" to construct chart or graph types.
- **Data presentation**: This is the delivery format, appearance, and synthesis of the entire design. It concerns the layers of color use, interactivity, annotation, and the arrangement of all elements.

Some helpful tips

Before we commence our design thinking, here are a few useful tactics to help you move smoothly through this process and achieve the best outcome:

- **Sketching**: Drawing out rough ideas on paper is a really good discipline to get into before you go anywhere near a computer. It doesn't matter if it is on the back of a beer mat, on a white board, or in a beautiful portfolio pad, try to sketch out your thoughts and concepts in order to download those ideas floating around in your mind. This is shown in the following screenshot:

This is especially invaluable if you are collaborating with others and the creative process is a shared experience. Doing this could save you a lot of time clarifying and formalizing your collective thoughts. It will also be a safeguard against the risk of forgetting those great initial sparks of creativity that came to mind when the project was first triggered. It will also enable you to quickly refine and change direction, if necessary, without having committed time to any technical development.

- **Note taking**: As you develop your expertise in data visualization, one of the best methods to

improve your creative judgment is to document your decision-making process. Keep a log of all the choices you've made, the reasons why you've decided to do one thing or reject something else, the ideas you've had along the way. You should also record details of any procedures you have established for tasks such as cleaning or transforming data or the cropping treatment applied to a set of images—you may need to repeat these and follow the same exact stages later. Finally, it is vital to make a note of all your data sources and references (including things that have influenced/inspired you). All this material will be particularly beneficial if you have the opportunity to publish a narrative of your design process, which can be extremely educational as much to yourself as it is to others.

- **Time management**: As with any design project, the planning and allocation of your time and available resource is a vital discipline. Utilize time wisely otherwise you can quite easily get sucked into spending too long on the minutiae of one stage (especially on the more attractive creative tasks). Don't be surprised to find that a great deal of your time is spent doing relatively mundane tasks such as data cleaning or preparation. That is to be expected. Also, developing an awareness of your typical design activity duration is very important, especially if you are a freelance designer looking to develop a more sophisticated approach to estimating and pricing for clients.
- **Reinventing the wheel**: As the field continues to mature and innovative new techniques are developed, there is a constantly growing reference library of potential solutions. As you will see soon, we shouldn't always need to feel like we have to constantly invent something new. There are plenty of creative options already available to be influenced or inspired by, providing us with idea templates to develop and tweak for our needs. Of course, you should always attribute influence and reference designers' work when it has been utilized as a significant and apparent source of help.

The visualization anatomy – data representation

The process of identifying the most effective and appropriate solution for representing our data is unquestionably the most important feature of our visualization design. Working on this layer involves making decisions that cut across the artistic and scientific foundations of the field.

Here we find ourselves face-to-face with the demands of achieving that ideal harmony of form and function that was outlined in the objectives section of Chapter 1, *Context of Data Visualization*. We need to achieve the elegance of a design that aesthetically suits our intent and the functional behavior required to fulfill the effective imparting of information.

What we're doing here is determining how we are going to show what it is we want to say. It is a difficult skill to master—something of a dark art—particularly given the set of factors we need to consider and the trade-offs we might need to make. Our task involves considering the following:

- Choosing the correct visualization "method" for the stories we're telling
- Accommodating the physical properties of your data
- Facilitating the desired degree of precision
- Creating an appropriate metaphor to depict our subject stylistically

Choosing the correct visualization method

The first matter is to determine the choice of visualization method. We aren't necessarily committing just yet to a specific chart or graph type, though we might have some in mind. Rather, this is about the general family or collection of chart types as defined by their primary storytelling method.

For example, a bar chart serves the function of comparing categories of values; a line chart, by contrast, enables us to show changes of values over time, geo-spatial data can often be best displayed over a map.

Your choice of visualization method will be mostly driven by the outcome of your work in Chapter 3, *Demonstrating Editorial Focus and Learning About Your Data*. You've developed your editorial thinking about the key data stories, analytical dimensions, and the questions you're trying to answer in your visualization.

Of course, it is often likely that you have determined several different analytical slices and you will probably need to consider different methods to appropriately convey the stories for each one.

There are a number of ways of classifying the variety of methods for visualizing data but here is a suggested taxonomy:

- Comparing categorical values
- Assessing hierarchies and part-of-a-whole relationships
- Showing changes over time
- Mapping geo-spatial data
- Charting and graphing relationships

Of course, there are often overlapping functional or storytelling features inherent to the chart types that sit under these method headings. For instance, a stacked area chart shows changes over time but also facilitates the categorical comparison of its different layers. That would be an example of a chart type that spans across two method classifications. However, the principle focus of this chart type is telling a story over time and so we would consider it belonging to the "showing changes over time" method. The comparisons it enables represent an additional but secondary focus.

As we saw in the previous chapter, the forming of data questions really helps you articulate the range of analytical stories you are wishing to portray. In our demonstration exercise, when we were looking to show the results of our analysis, we were essentially responding to the question "how have the medal-winning performances of China and Germany compared over the past five events?"

The story being expressed was about showing changes over time: that defines our method. The selection, ultimately, of a line chart—a type belonging to this category of visualization methods—was evidently a suitable match as we specifically wanted to show the detail of the continuous transition across all five event years.

Had the focus been more about a comparison of all countries and the combined, aggregate picture of results over time we might have chosen a stream graph or an area chart. Both of these are chart types that would typically fall within the method of "showing changes over time".

Alternatively, had we sought to demonstrate the stark comparison of the medals won at the earliest and most recent events in our dataset, we might have chosen a method for comparing categorical values. In this case, the use of a slopegraph or a bar chart would have been more suitable, as shown here:

We will cover this issue in more depth in Chapter 5, *Taxonomy of Data Visualization Methods*.

Considering the physical properties of our data

Now, we're looking to narrow down our search further by thinking about which types of charts will most effectively accommodate the variables of data we're looking to portray.

As discussed in the previous chapter, learning about the physical properties of your data gives you an important sense of the shape and size of your data. As you refine your editorial focus you will develop an understanding of the data variables you may seek to display graphically.

The quantity and nature of the variables you are using will have a significant influence on reducing the range of suitable chart types you might be able to use within the method family you have chosen. As discussed earlier, this process of eliminating choices can only be of help to us as we move forward.

Referring back, once again, to our demonstration for the Olympics project, the data we were looking to use for our final story was event year (quantitative interval-scale), medal totals (quantitative ratio-scale), and country (categorical nominal). We had a good sense of the range and distribution of values held against each variable, we were just highlighting two countries and we wanted to show the full five-event transition. The best solution, therefore, was to use the line chart as we have just seen.

In Chapter 5, *Taxonomy of Data Visualization Methods*, we go into much more detail about this taxonomy and the range of chart types that sit underneath each of these five headings. You will see a gallery of some of the most common, contemporary methods. Each example is accompanied by a description of the chart and an outline of the data variables that each one can realistically accommodate. This will give you a good sense of some of the common data representation techniques. It could act almost like a creative menu for you to refer back to when seeking ideas and potential solutions.

Determining the degree of accuracy in interpretation

Now we start to step into the minefield. You can be assured that this section will have been the most revised and rewritten across the entire book.

Having identified the general visualization method and started to filter down further to identify the most suitable chart types, we now have to consider another key issue.

This judgment gets to the very heart of the form/function or art/science fault lines that exist in this field—to what degree of accuracy do you wish readers to be able interpret values from your visualization?

You might ask in response, why would you *ever* not wish to maximize the precision of interpretation? Surely, the mission is to deliver as much accuracy through our representation as possible?

Well yes, of course we do, but the inclusion of terms such as "maximize" and "as possible" allude to the specter of alternative influences. You see, for certain contexts, as we outlined in the early chapters, you might be seeking to explore different aesthetic forms of representation. And yes, sometimes this might involve certain sacrifices in terms of the precision of interpretation.

To frame this section, we first need to learn about visual variables. A visual variable is the specific form we assign to data in order to represent it visually. It could be the length or height of a bar, the position of a point on an axis, the color of a county on a map, or the connection between two nodes in a network.

Each of the chart types that we come to take as common representation methods are based on the deployment of a single or, more commonly, combination of several visual variables at once. Using multiple variables, in particular, enables a designer to efficiently express extra layers of meaning behind the properties of a single mark, as the next example demonstrates.

Image from "How Much Money do the Movies we Love Make?" (http://vallandingham.me/vis/movie/), created by Jim Vallandingham

In this bubble chart visualization, each mark involves a combination of several visual variables representing a range of different data variables:

- The position on the x axis represents a film's profit.
- The position on the y axis represents the average review percentage rating of each film.
- The circle area represents the film's budget.
- The circle's color (hue) represents movie genre.
- Users interacting with this web-based design will also discover a text label displaying the raw values by hovering over one of the bubbles. Text is not universally treated as a visual variable but it is still worth acknowledging.

If you let your imagination run free and try to conceive as many visible properties as possible that might be capable of representing series of categorical, ordinal, or quantitative data, you will realize that there are many potential approaches. In fact, in a fun recent experiment (http://blog.visual.ly/45-ways-to-communicate-two-quantities/), visualization designer Santiago Ortiz proposed over 40 different ways to potentially represent just two simple numbers.

Indeed, if you fully release your creative inhibitions, you can take things beyond the visible or physical, as some of Santiago's suggestions did, and consider how our other senses might be exploited in order to represent/interpret data through channels such as sound, smell, touch, and taste. Just imagine how some of the inherent variable qualities of our other sensory mechanisms could be capable of distinguishing categories and values of data.

Anyway, back to more practical thoughts, for now, and a brief history lesson.

Once upon a time there was a man called Jacques Bertin. Bertin was a clever chap. In fact he was a pioneering thought-leader within data visualization. Building on the earlier studies from the Gestalt School of Psychology—mentioned in Chapter 1, *The Context of Data Visualization*—his book *Semiologie Graphique* (1967) is one of the subject's founding texts and represents one of the earliest and most comprehensive attempts to theorize how we perceive and interpret different representations of data through shape, pattern, and color.

Bertin determined that there were three main aims behind your choice of data representation, moving from high-to low-level acts of graphical interpretation. This is still an extremely potent way of organizing our thoughts and reasoning our selection of the most effective visual variables. These aims are as follows:

- The highest level of Bertin's interpretive acts concerned whether we are able to visually discriminate between different data marks or data series: can we actually see and read the data being presented. We must make sure that the way we visually distinguish different categorical and quantitative values is legible and is in no way hidden by way of unnecessary clutter, noise, or distraction.
- The second act refers to being able to satisfactorily judge the relative order or ranking of values in terms of their magnitude. This is basic pattern matching where we seek to determine the general hierarchy of the values being displayed: where is the most and where is the least, which is the biggest and which is the smallest.
- The lowest-level act relates to judging values. Studies have shown how the effectiveness of different visual variables can be ranked based on which most accurately support comparison and pattern perception. Bertin was the first to propose such a hierarchy and his work has been tested, developed, and refined by Cleveland and McGill (*Journal of the American Statistical Association, Vol. 79, No. 387. September, 1984, pp. 531-554*) and then by Jock MacKinlay.

Here, in the following presentation, we see the most recent version created by MacKinlay. Each column represents the three main data types (note that there is no distinction between ratio and interval-scale types of quantitative variables). Within each column you have an ordering of the most accurate and least accurate visual variables according to their interpretive precision:

	Categorical (Nominal)	Categorical (Ordinal)	Quantitative (All)
Most Accurate	Position	Position	Position
	Color Hue	Density	Length
	Texture	Color Saturation	Angle
	Connection	Color Hue	Slope
	Containment	Texture	Area
	Density	Connection	Volume
	Color Saturation	Containment	Density
	Shape	Length	Color Saturation
	Length	Angle	Color Hue
	Angle	Slope	Texture
	Slope	Area	Connection
	Area	Volume	Containment
Least Accurate	Volume	Shape	Shape

Image recreated from "Ranking of Perceptual Tasks" (Automating the Design of Graphical Presentations of Relational Information, ACM Transactions on Graphics, Vol.5, No.2, April 1986) by Jock MacKinlay.

The studies by Bertin, Cleveland and McGill, and then MacKinlay focus on the fact that our visual system isn't capable of absolute measurements. Therefore, frameworks like this simply propose a guide to understand which variables will be better at delivering relative measurements but with highest accuracy. In other words, the higher up the column the easier it will be for your reader to accurately interpret values represented by those variables.

So, looking at that table, you might ask why you would ever *not* use position as the visual variable for your data: That will surely maximize the efficiency and accuracy of your data communication for all data types?

It is unfortunately not as simple as that. If it was, I don't think I would need to write this book. As we've seen earlier, we rarely only have just one data variable to communicate. You will therefore often need multiple visual variables to communicate multiple data variables.

However, above that, and returning to our section introduction, how accurate do we really need the interpretations to be? Do we actually need to facilitate the reading of exact values from every visualization we create?

Alternatively, can we allow ourselves more creative freedom by recognizing that in some cases just being able to facilitate the relative order of values may be sufficient for the context and requirements of the design?

If you recall in the first stage of our methodology we discussed the importance of trying to define, as early as possible, the functional and tonal quality of your intended design. The tonal judgment, in particular, is the important matter right now for this is what separated those pieces that matched an analytical and pragmatic style from those that were more abstract or emotive.

In this image we see a selection of visualization styles and demonstration of the fine balance being judged between design creativity and interpretive accuracy based on the contextual requirements. Let's take a closer look at each one at a time:

- The "wind map" on the left-hand side of the image doesn't aim to facilitate the reading of exact values. The use of pattern density to indicate the strength of the wind, as shown in the hierarchy table, focuses on delivering a sense of those areas with strong wind (as well as its direction)and the areas where there is little wind. The elegance of the resulting design makes for a compelling visual that draws users to interact and learn about the patterns.
- The "OECD Better Life Index" display in the middle shows a glyph chart based around floral shapes with the petals sized according to different quality of life-indicator values. We don't find it that easy to determine precise values from these shapes but we do get a sense of the big values, the medium values, and the small values. This is an attractive alternative to a very pragmatic and analytical display such as a bar chart, or even just a table of numbers. At this primary layer of the tool's interface, the balance achieved between design creativity and interpretive accuracy was judged to be ideal, with the added feature of interactivity to enable more detailed annotation and accurate value-reading.
- The example of an area chart and bar chart, on the right-hand side of the image, provides a contrasting context. Here we might be talking about an analytical experience where the accuracy and efficiency of exchange is paramount. The idea of design innovation or novel creativity is not important. In these cases, you will be looking to prioritize the deployment of the higher-ranking visual variables enabling a reasonably easier experience in reading the values.

In each of these cases, we see a different balancing act taking place, a series of trade-offs between the interpretive accuracy and the design aesthetic to arrive at the right solution for the given context.

Creating an appropriate design metaphor

Maintaining consistency with our defined purpose—the requirements that come from understanding what triggered the project as well as the tonal and functional choices behind our intent—should be seen as a proposed pathway, not a final destination.

It may be that our initial thoughts around what we would hope to achieve are revised as we learn more about the data and the stories we can and may wish to tell. For instance, we might have initially thought there could be a rich narrative emerging that could have been portrayed quite powerfully and emotionally. As you learn more about your data and its potential deployment representation-wise, it may be that you realize a more analytical approach is more suitable.

Likewise, when we discover more about the extent of variety and potential in a dataset, a topic we thought would lead to an explanatory piece may actually evolve to be a more exploratory piece.

We are never fixed to our choices, but the quicker they are defined and the clearer we are in our conviction the better the design will be served.

This is important to recognize because it may be that when you start compiling your potential representation solution, you start to see things in a different light. This is particularly possible when the designer starts to consider integrating an extra layer of meaning through visual metaphor.

Visual metaphors are about integrating a certain visual quality in your work that somehow conveys that extra bit of connection between the data, the design, and the topic. It goes beyond just the choice of visual variable, though this will have a strong influence.

Deploying the best visual metaphor is something that really requires a strong design instinct and a certain amount of experience.

In this next example, designer Moritz Stefaner was commissioned to analyze and visualize how the clients and customers of a German start-up muesli company combined the ingredients they offer. The end result was a static visualization based on a radial network or chord diagram (on the right), showing ingredients grouped by category (base mueslis, fruit, nuts, sweets, and so on):

Images from "Müsli Ingredient Network" (http://moritz.stefaner.eu/projects/musli-ingredient-network/), created by Moritz Stefaner for mymuesli (http://mymuesli.com)

When describing his design process, Moritz noted how, when he worked on different sketches of the data, the matrix chart (example on the left-hand side) revealed some particularly strong stories that were otherwise missed in the radial diagram (on the right-hand side).

However, the radial diagram was the solution picked for the final piece. So why was this? Aside from some printing-readability issues that undermined the matrix chart, he commented: "from a visual point of view, it does not look very tasty." The radial diagram looked more appetizing—more edible—and fitted more strongly with the metaphor of a fruit bowl. The matrix chart solution looked more like fungi!

The decision to sacrifice certain qualities of interpretive enlightenment—offered by the matrix chart—was justified by the designer's instinct to enhance the visual metaphor emerging from the design of the radial diagram.

Referring back to the "wind map" again, here we saw a visual design that seemed consistent with the subject matter: it looks and feels like wind would look like in our imagination. In the piece that we saw in Chapter 2, *Setting the Purpose and Identifying Key Factors*, titled *Iraq's Bloody Toll*, this was a powerful story made even more impactive through the subtle but emphatic visual metaphor created by the arrangement of the chart and the color scheme.

These examples reinforce the value of Edward Tufte's message, from earlier, about the importance of

making judgments through our own lens and based on our own word of authority.

Choosing the final solution

From the options and influences that we have just processed, we should be able to narrow down and identify the right data representation specification for our visualization.

It may be that this specification matches neatly with an established chart type and we can essentially "pick it off the shelf" and deploy it. This could be considered our top-down approach. In Chapter 5, *Taxonomy of Data Visualization Methods*, we'll see a gallery of contemporary solutions. From this selection you may identify a particular solution that fits with your context both structurally and metaphorically.

Alternatively, we may wish to custom-build a solution from the bottom up, carefully constructing a design one visual variable at a time, accommodating the range of data variables we want to show, and the style with which we want to show it. Of course, every chart type had to originate from somewhere and be invented at some point. It may be that as we construct our unique solution, we end up slipping back towards realizing that a tried-and-trusted option remains the best choice.

As I introduced at the start of this section, getting the right data representation solution can be hard, particularly in light of all these competing influencing factors. The best way to enhance your skill on this front is through practice, developing your experience, and learning from others. Get into a discipline of curating great examples that you come across in the field and try to discover how others have tackled similar subjects or maybe even similar dataset "shapes".

The fundamental challenge is being able to handle the temptation (or pressures, depending on your viewpoint) to focus on achieving aesthetic innovation or novelty. This is understandable. Many times I hear clients and training delegates expressing a desire to simply move beyond the bar chart, line chart, and pie chart in order to create something different.

We are such taste-driven beings; it is simply human nature but sometimes this is seeking innovation for the sake of it. There is no point pursuing something different on the flawed basis of just trying to stand out from the crowd or to put more "bums on seats" if the resulting experience for the reader is one of ultimate frustration. The disappointment caused by aesthetics that obstruct and obscure the discovery of insights about the subject matter will outweigh any good will created by an initially positive impression.

The key is not to set out to achieve an attractive and attention-grabbing work—let those qualities emerge as a by-product of good design. Focus instead on delivering the appropriate functional elements by employing the most suitable data representation.

Over time, the more experience you gather as a designer, the more natural these judgments will become.

The visualization anatomy – data presentation

The presentation of data involves thinking about pretty much every other design feature that might be included in our visualization. Here, we are determining the following:

- The use of color
- The potential of interactive features
- The explanatory annotation
- The architecture and arrangement

The decisions we make about these layers should be focused on delivering extra meaning, intuitiveness, and depth of insight to our readers or users.

One of the key concepts throughout our judgment of presentation-related design options is to seek to make the visible, invisible. In contrast to data representation, where our objective is to make the invisible stories and insights, visible, data presentation features should almost feel invisible so that the portrayal of the data maintains visual dominance. Therefore, try to bear the following two things in mind:

- **Visual inference means data inference**: If it looks like data, it should be data. If it isn't data then you've incorrectly conveyed a sense of representation where there isn't any and design refinement is required. An example might be the use of a color to represent a certain sentiment. If that color is used on a bar chart or is picked for the background of a label or call-out, but it is no longer connected to the representation of any sentiment meaning, this may trick the reader who has programmed their visual sense to spot this inference.
- **Facilitating the resemblance of data**: Let the data breathe. We talked about this in the discussion about Jacques Bertin's interpretative acts, but the presentation layers of your visualization will have a great impact on this. Ensuring a reader can discriminate between data categories and values is usually influenced by the background artifacts and surrounding apparatus. Throughout your design, make sure your data stands out clearly as the principle visual component.

The use of color

Here is a quote from Maureen Stone (http://www.perceptualedge.com/articles/b-eye/choosing_colors.pdf):

> *"Color used well can enhance and clarify a presentation. Color used poorly will obscure, muddle, and confuse."*

We've already touched on various aspects of using color as a potentially important visual variable for the representation of data, but the deployment of color for a visualization project naturally extends further. Given the depth and breadth of the field of color theory, it is important to consider it separately from our other design choices. The preceding quote emphasizes the value of doing this.

When deployed poorly, the use of color can create unnecessary decoration that can distract and compete undeservedly for attention in ways that will undermine the clarity and accessibility of the information exchange.

Conversely, with effective use of color we can deliver an attractive, synthesized design that most efficiently taps into the preattentive nature of the eye and the brain.

We are seeking to create layers of visual prominence that help us instantly achieve a sense of the important messages and features. Take a look at a landscape painting and witness the depth that is created through color, the separation between foreground and background that helps elevate prominent features and relegate contextual properties.

The best advice for guiding your decisions about using color is to refer to the two key rules shown at the start of this section—make sure it is used unobtrusively and it does not mislead by implying representation when it shouldn't be.

As with all design layers, the sensible objective here should be to strive for elegance rather than novelty, eye-candy, or attractiveness. To achieve this, it is important to be aware of the different functions, choices, and potential issues surrounding color deployment.

To represent data

One of most common mistakes used in relation to color is seen when it is being deployed to represent quantitative data. Specifically, when the "hue" property of color is used.

Take a look at this spectrum of colors: if these squares were representing quantitative data, which would be the biggest? How about the smallest? Which is bigger, red or blue?

As you will realize, there is no convention or association that determines a relationship between color (hue) and any sense of hierarchy or order of magnitude. We don't see one color as being inherently bigger or smaller than the other, and so to use this to represent quantitative data is a mistake.

In the following pair of images, on the left-hand side, even with a color legend explaining the value bands being depicted by the different colors, there is no preattentive association that allows us to efficiently determine the values being represented on the map. Referring back to Bertin's interpretive acts, we can't even easily establish a general sense of big, medium, and small values without having to constantly move to-and-fro the map and the legend. By contrast, the map on the right-hand side uses a single hue and uses a sequential color scheme that represents the highest values (dark) to lowest values (light) in a logical and immediately understandable way:

Image (left) republished from the freely licensed media file repository Wikimedia Commons, source:http://en.wikipedia.org/wiki/File:FrancePopulationDensity1968.png

Image (right) from "The Good and The Bad [2012]" (http://www.theusrus.de/blog/the-good-the-bad-22012/) by Martin Theus

What we can see demonstrated in this example is that, for quantitative data, one of the best ways to effectively depict a range of quantitative values is through the lightness property of color: that is, a scheme which goes from the most intense color through to increasing amounts of white. This is sometimes called a sequential color scheme:

As we can see clearly in this next display, we inherently and automatically attach a sense of order to such sequential scales. Of course, without a key it might be difficult for us to precisely pick out the absolute values that each color band represents, but we can certainly determine major patterns that lead to judgments of data order within and across both sample maps:

Image (cropped) from "Unemployment, 2004 to present" (http://projects.flowingdata.com/america/unemployment/raw.html), by Nathan Yau.

That idea, of surfacing the general patterns of the highest and lowest values, is really what the main purpose of color is when used to represent quantitative variables.

There are other types of color scheme used for situations that require us to represent two quantitative variables or to highlight two extremes of a single variable. These are known as diverging schemes.

While there is a variety of different ways to construct diverging color schemes, typically, the extreme ends of the spectrum are presented as darker and distinguished by strongly contrasting color hues. Alternative approaches might involve exploiting established color metaphors or might already be intuitively understood or easily learned.

The next image is an example of where preprogrammed understanding of color representation can be utilized. In this case, we see the respective strength of party political support across the U.S., with the Republicans represented by their established red and Democrats in blue. This is a topological map that displays calculated contours to show the general spread of support for each party. An added dimension to this particular piece is the use of an extra representation—color transparency—to represent population density, thus adjusting the display to accommodate the lack of population uniformity.

Image from "Isarithmic Maps of Public Opinion Data" (http://dsparks.wordpress.com/2011/10/24/isarithmic-maps-of-public-opinion-data/), by David B Sparks

It isn't just on maps, of course, where properties of color can be important to distinguish quantitative values. One of the most popular methods for coloring involves the traffic light metaphor of red, amber, and green. This is commonly used in corporate settings to indicate good, average, or bad performance thresholds.

However, it is important to know that around 10 percent of the population (particularly males) has a red-green color deficiency. The use of an approach such as the traffic light colors will therefore potentially alienate a significant proportion of your intended audience. An effective alternative is to switch green for blue, so positive values are now shown as blue and negative are still in red, as we see in the following horizon chart:

Image from "Unemployment Rate: variation from the county average" (http://warksobservatory.files.wordpress.com/2012/07/unemployment-horizon-chart.pdf), by Spencer Payne/Warwickshire Observatory

To check your chosen color schemes against the potential impact of different color deficiencies, use an application such as Vis Check (http://vischeck.com), which is a free online tool to simulate what a color-blind person would see when looking at your images.

As we've already explained in the data representation section, one of the key functions of a visual variable is to facilitate resemblance—the discrimination of data—and the use of color (hue) to distinguish between categorical variables is a particularly strong aid.

In the next example, we see a project created to display the status of various indicators surrounding how different states around the U.S. handle gay rights issues.

There are seven distinct categories of data distinguished by a unique color. The color itself has no meaning; it is purely a means of helping to separate out the various tracks of issues. The lightness of the color does add an extra layer of information, indicating where maximum (darker) and limited (lighter) rights are in place, and the absence of any color as well as the presence of a cross-hatching pattern further encodes extra meaning:

Image from "Gay rights in the US, state by state" (http://www.guardian.co.uk/world/interactive/2012/may/08/gay-rights-united-states), by Guardian in America Interactive

As we saw in the earlier image showing the political persuasion map of the U.S., the use of color for categorical data also allows us to maximize the implication of metaphorical or representative association.

However, regardless of whether the color depiction of categories is arbitrary or embodies more meaningful association, one of the key rules we need to obey is that the eye is only really capable of distinguishing up to a maximum of twelve different color classifications. This is just one of the many fascinating aspects of color that can be discovered from great books such as *Visual Thinking by Design*, by Colin Ware.

If you have more than twelve categories you may need to find ways of combining classifications to avoid this issue. You'll see this in effect on images such as the many subway maps around the cities of the world. As more extended lines and routes emerge, there are fewer remaining distinct color options that will help to emphasize, indicate, and separate these new markings.

There are also many definitions about the emotional or cultural significances behind color representation. It is naturally advantageous to exploit universal visual languages, but only when they are definitely universal! You need to be sensitive to the potential differing perceptions of color meaning across the regions of world. For most colors there is contradictory association and so referring to a resource such as *Color Meanings by Culture* (http://www.globalization-group.com/edge/resources/color-meanings-by-culture) will represent time well spent.

To bring the data layer to the fore

In addition to the representation of data, we also look to employ color to help create visual depth and a sense of hierarchy in our designs. In the first chapter, we saw the demonstration of color and imagery being at fault for the lack of clarity in the diagram showing the placement and outcome of penalties taken by a selected footballer.

The clutter that can occur between background presentation and the foreground data representation makes it a real challenge to efficiently establish a sense of visual hierarchy. The brain and the eyes otherwise have to work especially hard to draw any insight.

What we are trying to establish is a clear sense of the most important signals brought to the foreground and the less important contextual or decorative elements pushed into the background.

We saw this effect successfully demonstrated by our proposed solution for the Olympics demonstration in the previous chapter. Here, the main focus surrounding the narrative of China and Germany's transition over time was achieved by promoting their series of values strongly into the foreground through color. The rest of the value series for the remaining countries, as well as the chart apparatus, were relegated subtly into background but were still visible and available for reference.

We see a similar effect demonstrated by this following image taken from a typical dashboard display. By their very nature, dashboards are deployed in situations whereby the efficiency and accuracy of detecting key message as signals is a key aim:

In this example, we see a limited, rather monochromatic color scheme applied across all properties —values, charts, labels, and titles. Through deploying this soft palette, it enables the key signals to jump in to the foreground as the most important visuals: the red indicators (alerting a need for further investigation), the blue headline bars (best performance), and the very subtle markers on the sparklines to represent the highest (blue) and lowest (orange) weekly levels.

When it comes to learning about the potential of color to create a sense of hierarchy, we can take

inspiration from the effective deployment of color witnessed in other contexts. We can see examples from the best designs in advertising, website, product, and video games where creating intuitive, hierarchical displays are often vital components of their purpose and experience.

When it comes to judging background colors, there is no definitive set of rules about whether light (typically white) or dark (typically black) colors are better or worse. It is always a contextual judgment based on the intended style of the project as well as the palette of colors from which you intend to represent data. It is essentially a judgment about the legibility of contrast between foreground and background chart properties.

As a general piece of advice, try not to use strong, highly saturated colors when covering large areas. Don't force the eye to have to constantly contend with and process dominant colors. Instead, give yourself the option of using strong colors to highlight and draw attention to the data layer.

Another important property to take notice of, in the relationship between foreground and background, is the careful deployment of chart apparatus, such as the axes, gridlines, tick marks, borders, titles—any chart property you may use to frame and reference your data.

Don't be afraid to remove or dampen the visible presence of such elements, particularly as the defaults in many tools are set to black. We are automatically tempted to make things darker, bolder, more prominent, more imprisoned. Where possible, minimize, dampen, or even remove some of these chart properties because we want to let the data stand out and facilitate our "seeing" of its qualities.

This extends to elements like titles. The following are two contrasting title designs for a visualization project that was undertaken about the history of Olympic speed. The first title shows a very rich and colorful image comprising a mosaic of all the posters down the years:

When this version was incorporated on to the main design, it was immediately clear that it was too visually prominent, drawing too much attention away from the main data display. By contrast, the second version was much subtler and worked far better as a cohesive part of the final display:

Image from "Pursuit of Faster" (http://www.visualisingdata.com/index.php/2012/07/new-visualization-design-project-the-pursuit-of-faster/), by Andy Kirk and Andrew Witherley

There are many deeper and more specific aspects of color theory around the contrast or relationship between two colors. For example, typically it can be seen that blue on black is hard for many to discriminate, as is yellow on white. There are also issues to consider about the unexpected by-product of illusions being created between different arrangements of colors and shades. Color theory is a huge field and we can only reasonably scratch the surface in this book.

To conform to design requirements

The final factor concerning color involves the necessity to incorporate an organization's visual identity and conforming to predefined color palettes. Wherever possible, you would always seek to avoid the restrictions to color choice, but often this will simply not be possible. Just imagine some of the major corporations in the world and their brand identities and you'll immediately be able to envision the definitive color palettes.

The use of predetermined color schemes in visualization is to be expected, especially because it helps maintain consistency and recognition of brand. For a designer, it can be a hindrance and so it reemphasizes the importance of identifying this requirement in your early part of the methodology.

Here is an example from the Guardian newspaper. This bubble hierarchy diagram shows the breakdown of UK Government spending by department. The image contains a wide range of colors but they hold no quantitative or categorical meaning. Aside from helping to distinguish the different clusters, they perform a largely decorative function that makes the piece more attractive to engage with and help reinforce the organization's visual identity, which is typically a very colorful spectrum:

Image from "UK public spending by government department, 2008/09" (http://www.guardian.co.uk/news/datablog/2010/may/17/uk-public-spending-departments-money-cuts), by Michael Robinson and Jenny Ridley for the Guardian

Many organizations such as the Guardian and also the New York Times have developed such a strong visual identity from their respective works, consistently observing defined color palettes, that you can now immediately identify their work from the style this color usage perpetrates.

Creating interactivity

At its best, a static visualization is like a powerful photograph—a carefully conceived, arranged, and executed vision that manages to portray the sequence or motion of a story without the actual deployment of movement. In my humble opinion, delivering such an immersive experience through static design is the most elegant demonstration of data visualization.

That said, enhancements in technology over the past decade have created incredible opportunities for talented developers to construct powerful interactive visualizations. From the rapid diffusion of fast access to the Web, the development of advanced software, and programming environments, through to the immediate access to millions of live data records and the range of responsive platforms on which we can now access information, the richness of potential interactivity can lead to some incredible work.

We are currently witnessing a generation of outstanding interactive visualization projects, representing a paradigm shift in the levels of creativity, innovation, and user experience. Where once we were trapped by the limitations of a single sheet of paper, the limited real estate offered by a VDU and the slow speed of our Internet connections, now there are few, if any, genuine barriers to the potential of interactive visualizations.

Like we suggested earlier, the very best examples of interactivity manage to make the visible, invisible. That is, the functions of interactivity blend into the design so seamlessly and intuitively that the apparatus of interaction is inseparable from the data portrayal—we no longer view it as a tool wrapped around a data visualization.

Inevitably, the potential development of an interactive design requires technical capabilities. There is no way of avoiding that. Otherwise, the option to build an interactive will simply be a non-starter. Other constraints such as platform compatibility, data loading speed, and server capacity need to be factored in as well. Your ambitions may be lofty and impressive but you need to be realistic about what you can actually accomplish and this should have already been determined.

Referring back to your early thoughts about the purpose of your project, you also need to carefully consider the motivation and intention of this design. Specifically, what functional experience are you trying to create for your audience: is it an exploratory, explanatory, or maybe a combined design?

Remember, just because you can create interactivity doesn't always mean it enhances the user experience of engaging with data visualization. Don't compromise the essence of your visual communication by abandoning a static design just for the novelty of creating interactivity.

Conversely, if the complexity and variety of the data structures that you are working with make it incompatible with a static portrayal, that's exactly the situation that warrants interactive features.

If you have decided that facilitating interactivity is required for your visualization, you have many different features and functions to contemplate deploying.

The following interactive Sankey diagram is a perfect demonstration of a project that effectively integrates a host of useful interactive features that maximize the exploratory potential of the subject matter. It was developed to present a breakdown of the flow of different sources and types of fuels, from supply through transformation, and to end usage:

Image from "Energy Technology Perspectives 2012 Online Data" at http://www.iea.org/etp/explore
© OECD/IEA 2012, developed by Raureif GmbH and Christian Behrens

Notice, through the annotated tips, the range of different actions you can trigger in order to see the data from many different perspectives. That is a key factor behind the deployment of interactivity—being able to take on multiple views of a subject matter to really understand the stories emerging.

Here is a brief outline of the variety of types of functions and features you should consider building into your interactive work:

Manipulating variables and parameters

The complexity of some data frameworks often means we are trying to find ways of showing many dimensions of stories within a single display or to facilitate different combinations of variables for exploratory visual analysis. The ability to select, filter, exclude, or modify certain variables is a valuable way of letting the user interact with different slices of the data. Furthermore, grouping and sorting options are common facilities for extracting new insights.

In the Sankey diagram example, you can isolate any of the vertical segments to see the breakdown and flow of those individual components across the entire system. You can also modify the variable of time using the slider to see changes across numerous yearly milestones.

A technique called "brushing"—highlighting a set of data marks—is another powerful way of focusing in on a subset view of our presented data, particularly with scatter plot type displays.

Adjusting the view

In contrast to manipulating variables, this is more about adjusting the user's lens or window into the subject. When we have hierarchical or high-resolution data, the ability to perform vertical exploration through the different layers of detail is an important feature. This can be particularly valuable in map-based visualizations where you may wish to pan around the landscape and zoom through different levels of magnification. You would see the benefit of this in a project such as the "Wind Map" that we saw earlier, enabling the user to dive into different parts of the country or those areas with strong winds that would be interesting to see in more detail.

An additional element of view adjustment is to create different horizontal tabs or panels of data. For example, if you wanted to show your data stories via a series of explanatory sequences. Collapsible devices such as concertinas allow for extra detail to be neatly organized hidden away from the default view and revealed when required. In the energy flow example, we also see a full-screen option that enables users to optimize the screen space occupied by the interactive.

Annotated details

We'll learn more about annotation shortly but, in interactive terms, this is about creating extra layers of data detail through interactive events such as hovering or clicking. This is particularly useful if you want to reveal actual data values or extra detail about a given category or event.

In the earlier section we discussed the degree of accuracy in interpretation and we saw an example of an interactive bubble chart. As you hovered over the bubbles, you saw a pop-up text display with the raw numbers. The availability of this type of detail, just a click or hover away from view, might give us greater creative license. By having the backup of absolute data accuracy through these values, we might give ourselves that extra confidence to choose a less precise but potentially more creative visual variable for use in our data representation display. It's almost like having a perceptual safety net.

Animation

When we have time-series based data, there is great potential for us to portray our visualization through animation, creating a shifting scene of data as it unravels a compelling data story.

The use of features such as Play, Pause, and Reset can be enhanced by offering manually controllable time sliders (seen in the earlier energy flow example) as well as chapter navigation to skip through key milestones.

The following example below, depicting the expansion of post offices across the U.S. through the years 1700 to 1900 is a perfect demonstration of the potential power of animated data presentation. While the individual frames are interesting in their own right, the real power of this portrayal comes through the emerging story of the social history of population growth and migration across the country. In the following screenshot, we see the striking moment in 1846 when the first post office on the West Coast. This is an event that would have been lost without the animated version:

Image from "Posted: Visualizing US Expansion Through Post Offices" (http://blog.dwtkns.com/2011/posted/), created by Derek Watkins

One thing to bear in mind is that our memory capability is ill-equipped for remembering the previous scenes of an animated story. If the facilitation of comparison is important, then animation may not be the best method and something like a panel of small multiples will be more effective. The deployment of features such as trailing certain categories or the amplification of new values as they change significantly over time can also help compensate for this shortcoming.

The potential development of an interactive or animated visualization is a very exciting prospect for a talented developer and designer. However, without wishing to sound like the boring parent who doesn't let you go outside to play, the suitability and need for interactivity does need to be thoroughly reasoned and justified.

I will reemphasize the point made earlier: just because you can, doesn't mean to say you should. Interactive visualizations expand the creative opportunities but should be deployed to enhance the

clarity and accessibility of data perception, not compromise it. Poorly considered clicks, sliders, filters, and menus can create unnecessary distraction and may delay access to the data and the key insights.

Annotation

Here is a quote from Amanda Cox (http://eyeofestival.com/speaker/amanda-cox/):

> *"The annotation layer is the most important thing we do... otherwise it's a case of here it is, you go figure it out."*

Our next layer is one that can often be neglected. However, as this quote suggests, annotating visualization is such an important features of our design. It is about taking care of your audience, recognizing who they are, what they might know already, and what they don't know.

Done well, annotation can help explain and facilitate the viewing and interpretive experience. It is the challenge of creating a layer of user assistance and user insight: how can you maximize the clarity and value of engaging with this visualization design?

As discussed in the first chapter, a key objective for effective data visualization design is the facilitation of accessibility into a subject through intuitive design. The degree of accessibility is enhanced through the effective inclusion of useful explanation across all features of your visualization solution. We shouldn't assume that readers or users are instantly and easily going to be able to navigate their way around our designs and so we need to carefully consider the best ways to assist them; explained as follows:

- **Titles**: A compelling title can help to attract an audience and articulate the focus of the visualization subject matter. Sometimes, especially in explanatory visualizations, you can look to exploit this prominent space to tell readers about a key insight or headline. However, make sure it is an accurate reflection of the content of the visualization otherwise it will be misleading.
- **Introductions**: These are really important instructive elements to explain the project's background and context, describing the background motivation and what your intentions are in terms of how it should be used.
- **User guides**: While intuitive accessibility is stated as an overall goal, many projects often warrant further explanation, particularly with interactive pieces and those that have inherently complex subjects or frameworks.

  In this next project, titled *Political Moneyball* and created by the Wall Street Journal, we see a demonstration of exceptional care for the audience's understanding of how to optimize this visualization's use. Not only does it include thorough written annotation and labeling to help users understand all the features of this incredibly immersive tool, but there is also a video tutorial to offer that extra degree of support. The designers of this piece astutely recognize the potential depth and interpretive complexity of the subject matter and I imagine also want to do justice to their efforts to bring this deep subject to fruition.

Image from "Political Moneyball" (http://graphics.wsj.com/political-moneyball/#), created by Andrew Garcia Phillips and Sarah Slobin of the Wall Street Journal

- **Labels**: In the interactive section, we discussed the potential of labels to reveal extra details about data values. As we see in the previous project, labeling is an incredibly simple but useful device to help explain matters. Often, these are hidden and interactively revealed through selection or by hovering.
- **Captions and narrative**: In addition to the potential use of the title to offer a key headline, sometimes you may wish to surface important insights and findings to help fast-track the reader's interpretation process. You might draw out the good and the bad or maybe the expected or unexpected. You should also consider the potential value, in certain projects, of the "what next?" question—what should the user do with this information? what actions need to be taken?
- **Visual annotation**: Annotation goes beyond just written explanations and we should consider how to use chart or graphic devices to help draw out important insights visually. Simple options include features such as gridlines, axes labels, and tick marks. In Chapter 3, *Demonstrating Editorial Focus and Learning About Your Data*, we saw an example of effective visual annotation. Here, reference lines and background shading is used effectively to help the reader achieve distinction between different tiers of interpretation, as you explore the relationship between what countries spend on education and the military.

Image from "In Numbers: Education Around the World" (http://visualdata.dw.de/specials/bildung/en/index.html), created by Gregor Aisch for Deutsche Welle

- **Legends and keys**: Always explain the use of color schemes or the varying size of shapes in terms of their categorical or quantitative representation.
- **Units**: You should include details of the units of values being displayed to ensure you don't create ambiguities and potential misinterpretation. As with many of these annotated features, this is an obvious requirement, something we've had drilled into us since our school days, but you'd be surprised how often they can be left out.
- **Data sources**: It is vital to include detailed references about from where you have accessed your data or any other sourced element (such as imagery). Where you have chance to offer a more detailed narrative, you may wish to explain what treatment you have applied to the data in terms of its quality or analytical transformation.
- **Attribution**: Don't forget to acknowledge those who have either contributed directly, influenced the construction of the design, or those people whose work has acted as a source inspiration.

The final thing to mention about annotation is that this is likely to be the first time we have to consider our typography selections. There are, of course, plenty of established guides and sources of literature to help influence your choice of fonts for all pieces of written annotation. However, this is another aspect of design that you will be able to ultimately judge best using your own design instinct. Many

designers have their favorites and like to maintain this identity but also many projects may be required to observe certain visual identity rules like we outlined in the color section.

Arrangement

You have established how you are going to represent your data, you've identified your visual identity through color, the choices around static or interactive design have been rationalized, and you have identified the range of annotation requirements.

For our final layer, we need to consider how to arrange our design in terms of the layout, placement, and organization of all visible elements. How can we piece everything together most effectively?

As we've just discussed in relation to annotation, our intention with the arrangement and architecture of our design is to deliver as intuitive an experience as possible. The level of intuitiveness and smooth access into the subject matter is strongly influenced by the logic and implied meaning behind the arrangement of our chart elements, the interactive features, and annotation devices.

The key overall aim is to reduce the amount of work the eye has to undertake to navigate around the design and to decipher the sequence and hierarchy of the display. For the brain, once again, we're looking to minimize the amount of thinking and "working out" that goes on. We therefore need to carefully consider the choices we make around the size, positioning, grouping, and sorting of all that we show. As with all visualization design layers, we need to be able to justify the decisions we make about every visible property presented.

Here is a simple, but effective, demonstration of the careful consideration of arrangement. It is just one example out of many we could refer to from the projects shown in this book.

Observe the positioning of the chapter navigation slider across the top, the size of the space afforded to the main map display, the narrative found on the right-hand side, the proximity of the legend to the data, and the location of the pan and zoom device—all these decisions are very deliberate and designed to maximize the logic and meaning behind the layout of this project's data, its interactive features and annotated elements.

Image from "The Growth of Newspapers Across the U.S.: 1690-2011" (http://www.stanford.edu/group/ruralwest/cgi-bin/drupal/visualizations/us_newspapers), created by Rural West Initiative, Bill Lane Center for the American West, Stanford University.

On the matter of arrangement, it is important to mention an important paper produced by Edward Segal and Jeff Heer of the Stanford Vis group and titled *Narrative Visualization: Telling Stories with Data* (http://vis.stanford.edu/papers/narrative).

As the title suggests, this article provides an excellent outline of the different design strategies for arranging and structuring the layout of your visualizations that will help maximize the potential telling of stories through data.

Summary

In this chapter, we have assessed the major design decisions we need to take across the key layers of a data visualization's anatomy. We have developed a better understanding of the choices we need to make and a sense of how to rationalize them depending on the context of our project.

We have worked through the challenge of selecting the appropriate data representation solution and then pulled apart the different aspects of data presentation in order to create a cohesive design concept.

Some might consider this to be an overly systematic or scientific way to tackle a task that has significant elements of creativity. However, this approach is consistent with the central theme of this book, to equip you with effective tactics and strategies to move gracefully through this potentially tricky design concepting process.

Remember that even the masters of the subject struggle to avoid making mistakes and ending up in creative blind alleys. A worthwhile design is rarely arrived at instantly and without the need for iterations and rethinks.

A key message is not to put yourself under too much pressure to get things right, straightaway and every time. Hopefully, following this framework of decisions and options will give you as good a chance as possible to get to your solution quicker and more efficiently through practice and experience.

In the next chapter, we will look closely at the taxonomy of different visualization methods and the various common chart types techniques that fall under each category. This will help cement your understanding of the data representation layer, undoubtedly the most important design feature to get right, and provide you with a convenient gallery of options to consider for your own projects.

Chapter 5. Taxonomy of Data Visualization Methods

In the previous chapter, we learned about the anatomy of a data visualization. This provided a framework to recognize and consider all the different design decisions you're faced with as well as guidance for how you might rationalize the choices you make.

In this chapter, we specifically revisit the data representation layer. This was described as the most critical layer of the visualization design task. It is also probably the most difficult to master.

In this chapter, we'll look at a taxonomy of data visualization methods as defined by the primary communication purpose. Within this taxonomy we will see an organized collection of some of the most common chart types and graphical methods being used today.

By exploring this array of chart types you will get a better understanding of the relationship between the stories you are trying to portray, the physicality of your data, and the visual variables through which we can represent them. It will provide you with a catalog of reference, offering ideas, and inspiration for when you face this stage in the design of your own projects.

The gallery of solutions presented does not intend or pretend to cover every subtle variation of chart or graph design. A creative field like data visualization simply does not lend itself to finite classification.

However, it should help you feel better equipped to more efficiently determine the most suitable representation solution for your specific problem context.

Data visualization methods

The common definition for taxonomy comes from the biological sciences and refers to the organization into groups of members that share similar characteristics. In this case, the members are chart types and the shared characteristic is based on the primary data portrayal function.

Selecting the appropriate visualization method will be influenced by the definition work you undertook earlier in the methodology to clarify the intention of your visualization communication.

It is about starting the journey towards identifying the most suitable way to answer your main data questions: how are you going to show, what it is you want to say.

Here is an outline of the primary communication purpose of each method classification:

Method classification	**Communication purpose**
Comparing categories	To facilitate comparisons between the relative and absolute sizes of categorical values. The classic example would be the bar chart.
Assessing hierarchies and part-to-whole relationships	To provide a breakdown of categorical values in their relationship to a population of values or as constituent elements of hierarchical structures. The example here would be the pie chart.
Showing changes over time	To exploit temporal data and show the changing trends and patterns of values over a continuous timeframe. A typical example is the line chart.
Plotting connections and relationships	To assess the associations, distributions, and patterns that exists between multivariate datasets. This collection of solutions reflects some of the most complex visual solutions and usually focuses on facilitating exploratory analysis. A common example would be the scatter plot.
Mapping geo-spatial data	To plot and present datasets with geo-spatial properties via the many different mapping frameworks. A popular approach would be the choropleth map.

Once we have selected the appropriate method, we then start to work through the other key ingredients of the data representation selection process, as outlined in Chapter 4, *Conceiving and Reasoning Visualization Design Options*.

Weighing up the following factors helps us to narrow down the variety of options within each method classification to find the most suitable choice of specific chart type or graphical method:

- Does it accommodate the physical properties of your data?
- Does it facilitate the desired degree of accuracy?
- Is it potentially capable of conveying a certain metaphorical and design consistency with our

subject matter?

Choosing the appropriate chart type

Attempting to organize chart types based on their primary portrayal method is not new (see http://queue.acm.org/detail.cfm?id=1805128 and http://www.visualizing.org/stories/taxonomy-data-visualization). The classifications presented in this chapter reveal a personal view—informed by knowledge, experience, and instinct—of a logical way to organize thinking about the relationship between data variables, visual variables, and chart frameworks.

The examples presented are intended to cover the most typical and contemporary approaches being used today. It is an arbitrary statement, but you should find that on 95 percent of occasions one or several of the chart types shown will cover your requirements. The remaining 5 percent will probably require custom-built solutions for very specific data shapes and contexts.

Note that many of the chart types presented hold numerous presentational characteristics and could belong to more than just one classification of method. For example, an area chart shows changes over time and enables comparison between categories. As the chart types have been organized based on their primary method, the prominence of the time-series nature of this example would lend itself more towards the "showing changes over time" method category.

As you go through the chapter, you will see the following information:

- The popular and alternative names used to describe each chart.
- The type and quantity of typical data variables you would normally use with each type of chart. On most occasions *any* categorical or quantitative variable is suitable though more specific variable types (nominal, ordinal, ratio-, interval-scale) are proposed where applicable.
- The visual variables that have been used to represent data (optional variables are italicized) in each chart.
- A brief description of each type's functional purpose and application.
- An example to illustrate what each chart looks like. Many of these have been seen elsewhere in this book so it will hopefully cement your understanding.

Remember, you may often need a combination of different visualization methods and multiple chart types blended together to forge a multidimensional story.

Comparing categories

The following examples present chart types that facilitate the comparison of categorical values.

Dot plot

Data variables: 2 x categorical, 1 x quantitative.

Visual variables: Position, color-hue, symbol.

Description: A dot plot compares categorical variables by representing quantitative values with a single mark, such as a dot or symbol. The use of sorting helps you to clearly see the range and distribution of values. You can also combine multiple categorical value series on to the same chart distinguishing them using color or variation in symbol. Beyond two series things do start to get somewhat cluttered and hard to read.

Bar chart (or column chart)

Data variables: 1 x categorical, 1 x quantitative-ratio.

Visual variables: Length/height, color-hue.

Description: Bar charts convey data through the length or height of a bar, allowing us to draw accurate comparisons between categories for both relative and absolute values. When using length as the visual variable to represent a quantitative value it is important to show the full extent of this property so always start the bar from the zero point on the axis. The use of color can help draw

attention to the values of specific categories in accordance with your narrative, as shown in the following screenshot:

Floating bar (or Gantt chart)

Data variables: 1 x categorical-nominal, 2 x quantitative.

Visual variables: Position, length.

Description: A floating bar chart—sometimes labeled a Gantt chart because of similarities in appearance—helps to show the range of quantitative values. It presents a bar stretching from the lowest to the highest values (therefore the starting position is not the zero point). Using such charts enables you to identify the diversity of measurements within a category and view overlaps and outliers across all categories. A Gantt chart is shown in the following screenshot:

Pixelated bar chart

Data variables: Multiple x categorical, 1 x quantitative.

Visual variables: Height, color-hue, symbol.

Description: The proposed name of "pixelated bar chart" is more an intuitive description than an established type. These charts provide a dual layer of resolution: a global view of a bar chart (showing aggregate totals) and a local view of the detail that sits beneath the aggregates (demonstrated by the pixels shown within each bar). Typically, these charts are interactive and offer an ability to hover over or click on the constituent pixels/symbols to learn about the stories at this more detailed resolution.

Image from "The Sexperience 1000" (http://sexperienceuk.channel4.com/the-sexperience-1000), created by Channel 4/Mint Digital

Histogram

Data variables: 1 x quantitative-interval, 1 x quantitative-ratio.

Visual variables: Height, width.

Description: Histograms are often mistaken for bar charts but there are important differences. Histograms show distribution through the frequency of quantitative values (y axis) against defined intervals of quantitative values(x axis). By contrast, bar charts facilitate comparison of categorical values. One of the distinguishing features of a histogram is the lack of gaps between the bars, as shown in the following image:

Slopegraph (or bumps chart or table chart)

Data variables: 1 x categorical, 2 x quantitative.

Visual variables: Position, connection, color-hue.

Description: A slopegraph creates an effective option for comparing two (or more) sets of quantitative values when they are associated with the same categorical value. They especially provide a neat way of showing a before and after view or comparison of two different points in time. In the following example, we see the total points won for teams in the English Premier League across two comparable seasons. The layout creates a combined view of rank and absolute value based on position on the vertical axis, with a link joining the associated values to highlight the transitional change. Color can be used to further emphasize upward or downward changes:

Radial chart

Data variables: Multiple x categorical, 1 x categorical-ordinal.

Visual variables: Position, color-hue, color-saturation/lightness, texture.

Description: A radial chart displays data around a concentric, circular layout. The example shown in the following image shows the status across a number of different categorical measures relating to gay rights for each state in the U.S., arranged to indicate approximate geographical relationships. A slight visual shortcoming associated with a radial chart is the fractionally distorted prominence of the segments on the outside rings which end up being larger (due to arc length) than those on the inside. Often radial charts are used for showing data over time but this only works when the sequence is continuous (such as a 24 hour clock).

Image from "Gay rights in the US, state by state" (http://www.guardian.co.uk/world/interactive/2012/may/08/gay-rights-united-states), created by Guardian in America Interactive

Glyph chart

Data variables: Multiple x categorical, multiple x quantitative.

Visual variables: Shape, size, position, color-hue.

Description: A glyph chart is based on a shape (in the following example, a flower) being the main artifact of representation. The physical properties of the shape (through a feature such as a petal) represent different categorical variables; they are sized according to the associated quantitative value and distinguished through color. While absolute magnitude judgments are not easily achieved nor intended, the hierarchy of the data (big, medium, and small values) is possible to interpret and the typical deployment of interactivity enables further exploration.

Image from "OECD Better Life Index" (http://oecdbetterlifeindex.org), created by Moritz Stefaner (htpp://moritz.stefaner.eu) in collaboration with Raureif GmbH (http://raureif.net)

Sankey diagram

Data variables: Multiple x categorical, multiple x quantitative.

Visual variables: Height, position, link, width, color-hue.

Description: Sankey diagrams are used to convey the idea of flow. They portray constituent quantities of a series of associated categorical values, across a number of "stages", with the ongoing associations represented by connecting bands. The width of these links indicates the proportional flow from one stage to another. They are useful for showing situations where elements transform and divide over key events, as shown here displaying the breakdown of different fuels, how they are transformed and then ultimately used.

Image from "Energy Technology Perspectives 2012 Online Data" at http://www.iea.org/etp/explore
© OECD/IEA 2012, developed by Raureif GmbH and Christian Behrens

Area size chart

Data variables: 1 x categorical, 1 x quantitative-ratio.

Visual variables: Area, color-hue.

Description: This type of chart doesn't have an obvious name, so Area size chart is a best attempt! It is a very simple visual device that deploys the visual variable of area—normally a rectangle or circle—to compare two (or maybe several) values. Normally these values will vary in size quite dramatically to convey a certain shock at the disparity. The subject matter may relate to a part-of-a-whole comparison (portion judgment) but more typically involves separate, independent categories (comparative judgment).

Small multiples (or trellis chart)

Data variables: Multiple x categorical, multiple x quantitative.

Visual variables: Position, any visual variable.

Description: Small multiples are not really a separate chart type but an arrangement approach that facilitates efficient and effective comparisons to be made across a multipanel display of small chart elements. These displays exploit the capacity of our visual system to rapidly scan across a trellis of small similar charts and to be capable of easily and immediately spotting patterns. These are particularly useful for comparing categories that have a broad range of values. They also work very well for showing snapshots of events that change over time. One of the earliest examples of this approach was *The Horse in Motion* by Eadweard Muybridge to show the different stages of a horse's movement over a time frame-by-frame. A trellis chart is shown in the following image:

Word cloud

Data variables: 1 x categorical, 1 x quantitative-ratio.

Visual variables: Size.

Description: Word clouds depict the frequency of words used in a given set of text. The font size indicates the quantity of each word's usage. Color is often just used as decoration (which you'll notice actually distorts the visual prominence). While it's fair to say they are becoming something of a ubiquitous visual commodity, they can be useful for exploring datasets for the first time in order to identify key terms being used. If you feel compelled to use word clouds, the best advice is to ensure the underlying text being used is carefully prepared in advance to reduce the noise. A word cloud is shown here:

design
data
visualization
image
visual
methodology
3462ot
important
subject
process
something
one
make
like
people
different
interpret
way
see
simple
take
think
new
practice
value
effective
patterns
work
things
knowledge
definition
science
many
perception
information
understanding
much
form
key
just
creative
created
chart
look
message
efficient
means
values
effectively
representation
receiver
objectives
effectiveness
provide
example
eyes
single
bar
function
better
four
solutions
need
across
reader
stage
discovery
communicate
creating
pattern
book
importance
thinking
use
never
based
source
general
involved
discipline
significant
functions
well
understand
level
field
terms
role
read
world
needs
mean
eye
rather
everything
instinct
two
fundamentally
create
sets
insights
help
decisions
ways
display
line
next
often
charts
brain
skills
right
presentation
interactive
reading
chapter
experience
techniques
exchange
approach
demonstration
side
intended
however
amount
sense
first
quote
another
able
greater
communication
throughout
indeed
consider
determine
digital
attributes
also
set
now
comes
beyond
already
objective
purpose
become
included
require
more
time
data
visualization
design
image
visual
methodology
process
something
one
make
like
people
different
interpret
way
see
simple
take
think
new
practice
important
subject
3462ot
value
effective
patterns
work
things
knowledge
definition
science
many
perception
information
understanding
much
form
key
just
creative
created
chart
look
message
efficient
means
values
effectively
representation
receiver
objectives
effectiveness
provide
example
eyes
single
bar
function
better
four
solutions
need
across
reader
stage
discovery
communicate
creating
pattern
book
importance
thinking
use
never
based
source
general
involved
discipline
significant
functions
well
understand
level
field
terms
role
read
world
needs
mean
eye
rather
everything
instinct
two
fundamentally
create
sets
insights
help
decisions
ways
display
line
next
often
charts
brain
skills
right
presentation
interactive
reading
chapter
experience
techniques
exchange
approach
demonstration
side
intended
however
amount
sense
first
quote
another
able
greater
communication
throughout
indeed
consider
determine
digital
attributes
also
set
now
comes
beyond
already
objective
purpose
become
included
require
more
time

Assessing hierarchies and part-to-whole relationships

The following examples present chart types that help us to assess hierarchical and part-to-whole relationships.

Pie chart

Data variables: 1 x categorical, 1 x quantitative-ratio.

Visual variables: Angle, area, color-hue.

Description: Pie charts are probably the most contentious chart type and attract much negative sentiment. While we know it is harder to accurately interpret angles and judge the area of segments compared to other visual variables, the negativity is arguably more a reflection of their relentless misuse. The inclusion of too many categories and colors, 3D decoration, and poorly executed arrangement are often to blame for this. Usually, a simple bar chart will suffice to demonstrate the part-to-whole relationship. However, if you are determined to use a pie chart, always start the first slice from the vertical position (to establish a sense of baseline), minimize the number of categories being displayed (ideally maximum of three), and arrange the segments as logically as possible. Variations include the donut chart, which is essentially the same chart but with the center removed (to accommodate labels or nested donut charts).

Stacked bar chart (or stacked column chart)

Data variables: 2 x categorical, 1 x quantitative-ratio.

Visual variables: Length, color-hue, position, color-saturation/lightness.

Description: Stacked bars are fairly self-explanatory. They can be based on the stacks of absolute values or standardized to show part of a whole breakdown, as in following example. Colors and

position differentiate the value categories. Where the categorical values are ordinal in nature, it helps to sequence the values logically, for example when you have sentiment data such as the Likert scale of disagree (reds) through to agree (blues). This sequencing helps draw out the contrasting composition of the sentiment from all categories. The only drawback of a stacked chart is the difficulty in being able to accurate read bar lengths, as there is no common baseline.

Square pie (or unit chart or waffle chart)

Data variables: 1 x categorical, 1 x quantitative-ratio.

Visual variables: Position, color-hue, symbol.

Description: There are several titles for this type of chart but the common technique involves a grid of units (may be squares or symbols) to represent parts of a whole. This may be for a percentage comparison (square pie) or an absolute quantity (unit chart, waffle chart). The use of color and symbol establishes the visual composition of the categorical and quantitative values. An example is shown here:

Tree map

Data variables: Multiple x categorical-nominal, 1 x quantitative-ratio.

Visual variables: Area, position, color-hue, color-saturation/lightness.

Description: Tree maps take the concept of a whole population and divide up portions of rectangular spaces within to represent organized, clustered constituent units sized according to their relative value. As well as arrangement, various properties of color are typically used to provide additional layers of quantitative or categorical insight. Here is an example:

Image from "Newsmap" (http://newsmap.jp/), created by Marcos Weskamp

Circle packing diagram

Data variables: 2 x categorical, 1 x quantitative-ratio.

Visual variables: Area, color-hue, position.

Description: As the title suggests, this type of chart seeks to pack together constituent circles into an overall circular layout that represents the whole. Each individual circle represents a different category and is sized according to the associated quantitative value. Other visual variables, such as color and position, are often incorporated to enhance the layers of meaning of the display. Note that you can't tessellate circles and so the combined view never creates a perfect fit (there are always gaps). The algorithms used to form the arrangement will often utilize certain overlapping properties to maintain the accuracy of the respective part-to-whole area sizes.

Image from "Gates Foundation Educational Spending" (http://vallandingham.me/vis/gates/), created by Jim Vallandingham

Bubble hierarchy

Data variables: Multiple x categorical, 1 x quantitative-ratio.

Visual variables: Area, position, color-hue.

Description: This technique is used to portray organization and structure through a hierarchical display. In the following example, we see the use of circles to represent the constituent departments, sized according to their quantitative value and colored to visually distinguish the different departments.

Image from "UK public spending by government department, 2008/09" (http://www.guardian.co.uk/news/datablog/2010/may/17/uk-public-spending-departments-money-cuts), created by Michael Robinson and Jenny Ridley for the Guardian

Tree hierarchy

Data variables: 2 x categorical, 1 x quantitative-ratio.

Visual variables: Angle/area, position, color-hue.

Description: Similar to the bubble hierarchy, this technique presents the organization and structure of data through a hierarchical tree network. In the following example, portraying the structure of a book, the effect is quite abstract but every visual property is serving the purpose of representing just the data - the quantitiative properties and hierarchical arrangement of all the book's elements:

Image from "Literary Organism" (http://itsbeenreal.co.uk/index.php?/wwwords/literary-organism/), created by Stefanie Posavec

Showing changes over time

The following examples show alternative ways of graphically showing changes over time:

Line chart

Data variables: 1 x quantitative-interval, 1 x quantitative-ratio, 1 x categorical.

Visual variables: Position, slope, color-hue.

Description: Line charts are something we should all be familiar with. They are used to compare a continuous quantitative variable on the x axis and the size of values on the y axis. The vertical points are joined up using lines to show the shifting trajectory through the resulting slopes. Line charts can help unlock powerful stories of the relative or (maybe) related transition of categorical values. Unlike bar charts, the y axis doesn't need to start from zero because we are looking at the relative pattern of the data journey.

An example is shown in the following image:

Sparklines

Data variables: 1 x quantitative-interval, 1 x quantitative-ratio.

Visual variables: Position, slope.

Description: Sparklines aren't necessarily a variation on the line chart, rather, a clever use of them. They were conceived by Edward Tufte and are described as "intense, word-sized graphics". They take advantage of our visual perception capabilities to discriminate changes even at such a low resolution in terms of size. They facilitate opportunities to construct particularly dense visual displays of data in small space and so are particularly applicable for use on dashboards. An example is shown here:

Area chart

Data variables: 1 x quantitative-interval, 1 x categorical, 1 x quantitative-ratio.

Visual variables: Height, slope, area, color-hue.

Description: As you can see in the following example, a number of visual properties are involved in area charts. The vertical position and connecting slope of the horizon (like a line chart) shows the progression of the values over time and the color area underneath the chart helps to emphasize these changes. Unlike a standard line chart, an area chart should have the y axis starting at zero to ensure the area judgment is being interpreted accurately.

Horizon chart

Data variables: 1 x quantitative-interval, 1 x categorical, 2 x quantitative-ratio.

Visual variables: Height, slope, area, color-hue, color-saturation/lightness.

Description: This is a variation on the area chart, modified to include (and cope with) both positive and negative values. Rather than presenting negative values beneath the x axis, the negative area is mirrored on to the positive side and then colored differently to indicate its negative polarity. The result is a chart that occupies a single row of space, which helps to accommodate multiple stories onto a single display and facilitates comparison to pick out local and global patterns of change over time.

Image from "Unemployment Rate: variation from the county average" (http://warksobservatory.files.wordpress.com/2012/07/unemployment-horizon-chart.pdf), created by Spencer Payne/Warwickshire Observatory

Stacked area chart

Data variables: 1 x quantitative-interval, 1 x categorical, 1 x quantitative-ratio.

Visual variables: Height, area, color-hue.

Description: A stacked area chart provides a compositional view of categories to show their changes over time. As the title suggests, these are based on stacks of area charts differentiated by color and present either absolute aggregates or percentage aggregates. Note that the quantitative values are represented by the height (derived from top and bottom positions) of the area stacks at any given point. Sometimes the resulting shapes of the middle sections can be slightly misleading and misinterpreted due to the lack of a common baseline position.

Stream graph

Data variables: 1 x quantitative-interval, 1 x categorical, 1 x quantitative-ratio.

Visual variables: Height, area, color-hue.

Description: The stream graph operates in a similar fashion to a stacked area chart, allowing multiple series to be layered as streams of area with quantitative values expressed through the height of the individual stream at any given time. It has no baseline x axis and so there is no concept of negative or positive values, purely aggregates. Its functional purpose is really to highlight peaks and troughs—it has a particularly organic feel and is suited to displays intended to show "ebb and flow" stories. Many stream graphs will offer interactivity to allow you to explore and isolate individual layers. An example is shown in the following screenshot:

Image from "German Energy Landscape" (http://visualization.geblogs.com/visualization/germanenergy), created by Gregor Aisch

Candlestick chart (or box and whiskers plot, OHLC chart)

Data variables: 1 x quantitative-interval, 4 x quantitative-ratio.

Visual variables: Position, height, color-hue.

Description: The candlestick chart is commonly used in financial contexts to reveal the key statistics about a stock market for a given timeframe (often daily). In the following example, we see stock market changes by day based on the OHLC measures—opening, highest, lowest, and closing prices. The height of the central bar indicates the change from the opening to closing price and the color tells us if this is an increase or decrease. This is a chart that clearly requires a certain amount of experiential learning in order to read it efficiently. However, one you've achieved this you will see how extremely dense and powerful these displays are. They are similar in concept to the "box and whiskers plot", which focus on the statistical distribution of a set of values (showing upper and lower quartiles as well as the median).

Barcode chart

Data variables: 1 x quantitative-interval, 3 x categorical.

Visual variables: Position, symbol, color-hue.

Description: These are very compact displays that depict a sequence of events or milestones over the course of time using a combination of symbols and color. In the following example, we see the key events during two football matches. Demonstrating similar qualities to those of a sparkline, barcode charts (named because they look like barcodes, funnily enough) convey a significant amount of data packed into a small space. Once again, as you familiarize yourself with how to read these charts, they do unlock a terrific amount of narrative.

Image from "Umbro World Cup Poster" (http://www.mikemake.com/Umbro-s-World-Cup-Poster), created by Michael Deal

Flow map

Data variables: Multiple x quantitative-interval, 1 x categorical, 1 x quantitative-ratio.

Visual variables: Position, height/width, color-hue.

Description: Similar in many ways to the Sankey diagram, a flow map portrays the flow of a quantitative value as it is transformed over time and/or space. In this famous example, showing the march of Napoleon's army in the Russian campaign of 1812, the thickness of the main band indicates the size of the army as it moves over time and geography towards Moscow. The geographical accuracy of the plot is preserved in this chart but we don't see (or need to see) the full map detail. Notice too that the freezing temperatures are presented in the line chart below the main display, providing a further layer of the detail behind this story:

The following map shows Napoleon's famous Russian campaign:

Images republished from the freely licensed media file repository Wikimedia Commons, source: http://commons.wikimedia.org/wiki/File:Minard.png

Plotting connections and relationships

We now look at the different visualization techniques used to plot connections and relationships:

Scatter plot

Data variables: 2 x quantitative.

Visual variables: Position, color-hue.

Description: A scatter plot is a combination of two quantitative variables plotted on to the x and y axes in order to reveal patterns of correlations, clustering, and outliers. This is a very important chart type, in particular, for when we are familiarizing with and exploring a dataset. An sample scatter plot is shown in the following image:

Bubble plot

Data variables: 2 x quantitative, 2 x categorical.

Visual variables: Position, area, color-hue.

Description: A bubble plot extends the potential of a scatter plot through multiple encoding of the data mark. In the following example, we see the marks becoming circles of varying size and then colored according to their categorical relationship. Often, you will see a further layer of time-based

data applied to convey motion with the plot animated over time.

Image from "How Much Money do the Movies we Love Make?" (http://vallandingham.me/vis/movie/), created by Jim Vallandingham

Scatter plot matrix

Data variables: 2 x quantitative, 2 x categorical.

Visual variables: Position, color-hue.

Description: Similar to the small multiples chart that we saw earlier, a scatter plot matrix takes advantage of the eye's rapid capability to spot patterns across multiple views of the same type of chart. In the following case, we have a panel of multiple combined scatter plots:

Image from "Scatterplot Matrix" (http://mbostock.github.com/d3/ex/splom.html), created by Mike Bostock

Heatmap (or matrix chart)

Data variables: Multiple x categorical, 1 x quantitative-ratio.

Visual variables: Position, color-saturation.

Description: With further similarities to small multiples, heatmaps enable us to perform rapid pattern matching to detect the order and hierarchy of different quantitative values across a matrix of categorical combinations. The use of a color scheme with decreasing saturation or increasing lightness helps create the sense of data magnitude ranking.

Parallel sets (or parallel coordinates)

Data variables: Multiple x categorical, multiple x quantitative-ratio.

Visual variables: Position, width, link, color-hue.

Description: Parallel sets offer a unique way of visually exploring and analyzing datasets. The technique involves plotting all your data on to a series of axes, one for each of the variables you are interested in examining. This creates pathways that show the connections between the breakdown of values contained within your data for each variable. They are useful for learning about the potential correlations and consistencies that exist in our datasets. You'll notice certain similarities with the function of Sankey diagrams.

An example of parallel sets is given in the following image:

Image from "Parallel Sets" (http://eagereyes.org/parallel-sets), created by Robert Kosara and Caroline Ziemkiewicz

Radial network (or chord diagram)

Data variables: Multiple x categorical, 2 x quantitative-ratio.

Visual variables: Position, connection, width, color-hue, color-lightness, symbol, size.

Description: A radial network or chord diagram creates a framework for comparing complex relationships between categorical values. The use of a radial layout offers the opportunity to move beyond the restrictions of an x and y axis pairing. The key explanatory property is the connections that exist between components, sometimes sized (thickness) and colored to incorporate extra layers of detail. In the following example, we see additional levels of detail represented by the encoded size of text and icons. One thing to note is that the length (and therefore prominence) of a line can slightly mislead by inferring significance (a more important relationship) when it is simply a by-product of the arrangement around the radial layout.

Image from "Müsli Ingredient Network" (http://moritz.stefaner.eu/projects/musli-ingredient-network/), created by Moritz Stefaner for mymuesli (http://mymuesli.com)

Network diagram (or force-directed/node-link network)

Data variables: Multiple x categorical-nominal, 1 x quantitative-ratio.

Visual variables: Position, connection, area, color-hue.

Description: At first glance, network diagrams, similar to the one shown in the following example, can look quite daunting through their visual complexity and apparent clutter (indeed, often they are described as "hairballs"). Their intention and value is to facilitate exploration of complex data frameworks based on the existence or quantifiable strength of relationships, connections, and logical organization. The typical purpose of these graphs is to enable the viewer to get a sense of patterns—picking out the elements that are of interest, observing clusters and gaps, dominant nodes and sparse connections. There are many derivatives of network diagrams with variations influenced by the data being used as well as the imagination and technical capabilities of the designer.

Image created by Joe Parry (http://key-lines.com/)

Mapping geo-spatial data

This final set of examples explores the different ways of mapping geo-spatial data

Choropleth map

Data variables: 2 x quantitative-interval, 1 x quantitative-ratio.

Visual variables: Position, color-saturation/lightness.

Description: As described in the previous chapter, choropleth maps color the constituent geographic units (such as states or counties) based on quantitative values using a sequential or diverging scheme of saturation/lightness. While these are popular techniques, there is a recognized shortcoming caused by the fact that populations are not uniformly distributed. There is a potential distorting effect created by the prominence of larger geographic areas which may not be proportionately representative of the population of data. Make sure you choose your color classifications carefully to ensure you accurately represent the chronological prominence of increasing quantities. An example of a choropleth map is shown in the following screenshot:

Image (cropped) from "Unemployment, 2004 to present" (http://projects.flowingdata.com/america/unemployment/raw.html), created by Nathan Yau

Dot plot map

Data variables: 2 x quantitative-interval.

Visual variables: Position.

Description: A dot plot map essentially displays a geographical scatter plot of records, combining the longitude and latitude to position marks on the map. In the following example, we also see this data being gradually plotted over time to reveal a story of geographical spread:

Image from "Posted: Visualizing US Expansion Through Post Offices" (http://blog.dwtkns.com/2011/posted/), created by Derek Watkins

Bubble plot map

Data variables: 2 x quantitative-interval, 1 x quantitative-ratio, 1 x categorical-nominal.

Visual variables: Position, area, color-hue.

Description: This type of mapping plots differently-sized circular markers over given geographical coordinates to indicate the magnitude of a quantitative value. Whereas the dot plot maps were like geographical scatter plots, these are essentially bubble charts overlayed on to a map. The main

contention with these designs tend to be that the spread of bubbles, depending on their size, can reach far beyond their geographical point and end up bleeding into other circles. Normally, the colors used include a relatively high transparency setting in order to accommodate the potential overlaps and "halos" are often used to distinguish outer edges. An example of a bubble plot map is shown in the following screenshot:

Image from "Visualizing Yahoo! Mail" (http://www.periscopic.com/#/work/yahoo-mail/), created by Periscopic

Isarithmic map (or contour map or topological map)

Data variables: Multiple x quantitative, multiple x categorical.

Visual variables: Position, color-hue, color-saturation, color-darkness.

Description: This is a technique for overcoming the flaws associated with the choropleth map and involves combining color-hue (to represent a political party), with color saturation (to represent the dominance of party persuasion), with a final dimension of color-darkness to represent the density of population. Algorithms are applied to help smooth the representation through the contour effect and this creates an elegant end result; as shown here:

Image from "Isarithmic Maps of Public Opinion Data" (http://dsparks.wordpress.com/2011/10/24/isarithmic-maps-of-public-opinion-data/), created by David B Sparks

Particle flow map

Data variables: Multiple x quantitative.

Visual variables: Position, direction, thickness, speed.

Description: A particle flow map uses animation to portray the motion of data across geography and over time. In the following example, we see the motion of the currents that drive the world's oceans. These careful and highly sophisticated constructions combine multiple variables of location, size, speed, and direction to create a compelling design that perfectly captures the nature of the subject matter.

Image from "Perpetual Ocean" (http://www.nasa.gov/topics/earth/features/perpetual-ocean.html), created by NASA/Goddard Space Flight Center Scientific Visualization Studio

Cartogram

Data variables: 2 x quantitative-interval, 1 x quantitative-ratio.

Visual variables: Position, size.

Description: Where a choropleth map takes a location and gives it a shade of color to represent a value, a cartogram takes a location and resizes the geographic shape to represent a value. The result is a distorted and skewed view of reality in the form of a reconfigured atlas. As with many of the chart types outlined here, the purpose is not to enable exact readings, rather to highlight the highly inflated, deflated, and unchanged shapes and sizes. They do rely on a certain predeveloped familiarity of (for example) a country's position, its shape, and its size. The most effective deployment of such charts tends to be when they are interactive and you can unlock all the benefits of exploratory analysis. An example is given here:

Image from "All Cancer Deaths: 2002" via Worldmapper (http://www.worldmapper.org/images/)
©Copyright SASI Group (University of Sheffield)

Dorling cartogram

Data variables: 2 x categorical, 1 x quantitative-ratio.

Visual variables: Position, size, color-hue.

Description: A Dorling cartogram (named after Professor Danny Dorling who invented them) deploys a uniform shape (typically a circle) to represent a geographical location and then sizes this according to a quantitative variable. In the following example, we see a portrayal of countries represented by circles, sized according to that country's CO2 emissions and colored to distinguish the continents. As before, we may struggle to easily identify places that have now been transformed in shape, size, and position but effective annotation can generally compensate for that.

Image from "An atlas of pollution: the world in carbon dioxide emissions" (http://image.guardian.co.uk/sys-files/Guardian/documents/2011/02/10/CarbonWeb.pdf), created by Michael Robinson, Mark McCormick and Paul Scruton

Network connection map

Data variables: 2 x quantitative-interval, 1 x categorical-nominal.

Visual variables: Position, link, color-hue.

Description: Similar to the network diagram that we saw before, the intention of a network connection map is to facilitate the exploration of complex geographical connections. A connection map joins up related locations to form a pattern that enables discovery of hubs, overlaps, clusters, and gaps—pretty much the same focus as that of the network diagram but this time with the platform of geographical coordinates. The exhaustiveness of certain datasets means an image of the atlas is almost fully formed from the resulting patterns and there is no need for the actual mapping layer to be visible. In the following image, we see the world's flight paths with those involving Toronto highlighted in orange.

Image from "Toronto Flight Lines" (http://www.biodiaspora.com/), created by Bio.Diaspora 2012

Summary

This chapter has showcased a fairly comprehensive range of visualization chart types across a taxonomy of different methods.

These examples should be considered as representative of the majority of the relevant and popular approaches being used today. However, you should appreciate that this is not an exhaustive nor restrictive collection of options and does not cover all the derivatives that are possible.

The purpose of this chapter was to help you to understand the challenges and options involved in rationalizing your data representation solutions. Hopefully, the information provided about the relationship between the physicality of your data variables and the chart types that can accommodate them will give you a fast track solution.

Additionally, you may have learnt more about the different roles played by visual variables across these examples, and it may inspire you to have the confidence to consider constructing your own unique solutions.

From what we have covered in this chapter and the previous chapter, you should be fairly clear about the design choices you are going to make. We now need to move beyond concept and towards production.

In Chapter 6, *Constructing, Launching, and Evaluating the Data Visualization*, we take the methodology through to the finish with a discussion about some of the most important tools and resources available for creating and launching visualizations. We will then discuss evaluation and the ongoing challenge of developing your visualization design capabilities.

Chapter 6. Constructing and Evaluating Your Design Solution

The work we have undertaken over the past two chapters has helped us to shape and refine our design concept leading to a visual specification that we believe will most effectively deliver against the requirements of our project. This completes our preparation work and we now move away from concepting and towards construction.

In this final chapter—and last stage of the methodology—we look at the broad variety of options for building our solution and the remaining important tasks to undertake before launching.

We will run through a selection of the most common and useful software applications and programming environments to help you select the most appropriate tool to match your design requirements and technical capabilities.

We will look at some of the key considerations around testing, finishing, and launching a design solution as well as the important matter of evaluating the success of your project post-launch.

Finally, we wrap things up with a discussion about the best ways for you to continue to learn, develop, and refine your data visualization design skills as you seek to master this fascinating and rewarding discipline.

For constructing visualizations, technology matters

The importance of being able to rationalize options has been a central theme of this book. As we reach the final stage of this journey and we are faced with the challenge of building our visualization solution, the keyword is, once again, choice.

The intention of this book has been to focus on offering a handy strategy to help you work through the many design issues and decisions you're faced with.

Up to now discussions about issues relating to technology and technical capability have been kept to a minimum in order to elevate the importance of the preparatory and conceptual stages. You have to work through these challenges regardless of what tools or skills you have.

However, it is fair to say that to truly master data visualization design, it is inevitable that you will need to achieve technical literacy across a number of different applications and environments.

All advanced designers need to be able to rely on a symphony of different tools and capabilities for gathering data, handling, and analyzing it before presenting, and launching the visual design. While we may have great concepts and impressively creative ideas, without the means to convert these into built solutions they will ultimately remain unrealized. The following example, tracking 61 years of tornado activity in the US, demonstrates a project that would have involved a great amount of different analytical and design-based technical skills and would not have been possible without these:

Image from "Tornado tracks" (http://uxblog.idvsolutions.com/2012/05/tornado-tracks.html), created by John Nelson/IDV Solutions.

In contrast to most of the steps that we have covered this far, the choices we make when it comes to producing the final data visualization design are more heavily influenced by capability and access to resources than necessarily the suitability of a given tool. This is something we covered earlier when identifying the key factors that shape what may or may not be possible to achieve.

To many, the technology side of data visualization can be quite an overwhelming prospect—trying to harness and master the many different options available, knowing each one's relative strengths and weaknesses, identifying specific function and purpose, keeping on top of the latest developments and trends, and so on.

Acquiring a broad technical skillset is clearly not easily accomplished. We touched on the different capability requirements of data visualization in Chapter 2, *Setting the Purpose and Identifying Key Factors*, in the *The "eight hats" of data visualization design* section. This highlighted the importance of recognizing your strengths and weaknesses and where your skillset marries up with the varied and numerous demands of visualization design. In order to accommodate the absence of technical skills, in particular, you may need to find a way to collaborate with others or possibly scale down the level of your ambition.

Visualization software, applications, and programs

The scope of this book does not lend itself to provide a detailed dissection and evaluation of the many different possible tools and resources available for data visualization design. There are so many to choose from and it is a constantly evolving landscape—it feels like each new month sees an additional resource entering the fray. To help, you can find an up-to-date, curated list of the many technology options in this field by visiting http://www.visualisingdata.com/index.php/resources/.

Unlike other design disciplines, there is no single killer tool that does everything. To accommodate the agility of different technical solutions required in this field we have to be prepared to develop a portfolio of capabilities.

What follows is a selection of just some of the most common, most useful, and most accessible options for you to consider utilizing and developing experience with. The tools presented have been classified to help you understand their primary purpose or function.

Charting and statistical analysis tools

This category covers some of the main charting productivity tools and the more effective visual analytics or Business Intelligence (BI) applications that offer powerful visualization capabilities.

Microsoft Excel (http://office.microsoft.com/en-gb/excel/) is ubiquitous and has been a staple diet for many of us number crunchers for most of our working lives. Within the data visualization world, Excel's charting capabilities are somewhat derided largely down to the terrible default settings and the range of bad-practice charting functions it enables. (3D cone charts, anyone? No, thank you.)

However, Excel does allow you to do much more than you would expect and, when fully exploited, it can prove to be quite a valuable ally. With experience and know-how, you can control and refine many chart properties and you will find that most of your basic charting requirements are met, certainly those that you might associate more with a pragmatic or analytical tone.

Sample screenshot of Excel's charting capabilities

Excel can also be used to serve up chart images for exporting to other applications (such as Illustrator, see later). Search online for the work of Jorge Camoes (http://www.excelcharts.com/blog/), Jon Peltier (http://peltiertech.com/), and Chandoo (http://chandoo.org/) and you'll find some excellent visualization examples produced in Excel.

Tableau (http://www.tableausoftware.com/) is a very powerful and rapid visual analytics application that allows you to potentially connect up millions of records from a range of origins and formats. From there you can quickly construct good practice charts and dashboards to visually explore and present your data. It is available as a licensed desktop or server version as well as a free-to-use public version.

Tableau is particularly valuable when it comes to the important stage of data familiarization. When you want to quickly discover the properties, the shapes and quality of your data, Tableau is a great solution. It also enables you to create embeddable interactive visualizations and, like Excel, lets you export charts as images for use in other applications.

Sample screenshot of Tableau's charting capabilities

There are many excellent Tableau practitioners out there whose work you should check out, such as Craig Bloodworth (http://www.theinformationlab.co.uk/blog/), Jérôme Cukier (http://www.jeromecukier.net/), and Ben Jones (http://dataremixed.com/), among many others.

While the overall landscape of BI is patchy in terms of its visualization quality, you will find some good additional solutions such as QlikView (http://www.qlikview.com/uk), **TIBCO Spotfire** (http://spotfire.tibco.com/), **Grapheur** (http://grapheur.com/), and **Panopticon** (http://www.panopticon.com/).

You will also find that there are many chart production tools available online. Google has created a number of different ways to create visualizations through its **Chart Tools** (https://developers.google.com/chart/) and **Visualization API** (https://developers.google.com/chart/interactive/docs/reference) environments. While you can exploit these tools without the need for programming skills, the API platforms do enable developers to enhance the functional and design options themselves.

Additionally, **Google Fusion Tables** (http://www.google.com/drive/start/apps.html) offers a convenient method for publishing simple choropleth maps, timelines, and a variety of reasonably interactive charts.

Sample screenshot of Google Fusion Table's charting capabilities

Other notable browser-based tools for analyzing data and creating embeddable or exportable data

visualizations include **DataWrapper** (http://datawrapper.de/) and **Polychart** (http://polychart.com/).

One of the first online offerings was Many Eyes, created by the IBM Visual Communications Lab in 2007, though ongoing support and development has lapsed. Many Eyes introduced many to **Wordle** (http://www.wordle.net/) a popular tool for visualizing the frequency of words used in text via "word clouds". Note, however, the novelty of this type of display has long since worn off for many people (especially please stop using it as your final PowerPoint slide in presentations!).

Programming environments

The ultimate capability in visualization design is to have complete control over the characteristics and behavior of every mark, property, and user-driven event on a chart or graph. The only way to fundamentally achieve this level of creative control is through the command of one or a range of programming languages.

Until recent times one of the most important and popular options was **Adobe Flash** (http://www.adobe.com/uk/products/flash.html), a powerful and creative environment for animated and multimedia designs. Flash was behind many prominent interactive visualization designs in the field. However, Apple's decision to not support Flash on its mobile platforms effectively signaled the beginning of the end. Subsequently, most contemporary visualization programmers are focusing their developments on a range of powerful JavaScript environments and libraries.

D3.js (http://d3js.org/) is the newest and coolest kid in town. Launched in 2011 from the Stanford Visualization Group that previously brought us Protovis (no longer in active development) this is a JavaScript library that has rapidly evolved into to *the* major player in interactive visualization terms.

D3 enables you to take full creative control over your entire visualization design (all data representation and presentation features) to create incredibly smooth, expressive, and immersive interactive visualizations. Mike Bostock, the key creative force behind D3 and who now works at the New York Times, has an incredible portfolio of examples (http://bost.ocks.org/mike/) and you should also take a look at the work and tutorials provided by another D3 "hero", Scott Murray (http://alignedleft.com/).

Sample screenshot of D3.js development environment

D3 and Flash are particularly popular (or have been popular, in the latter's case) because they are suitable for creating interactive projects to work in the browser.

Over the past decade, **Processing** (http://processing.org/) has reigned as one of the most important solutions for creating powerful, generative, and animated visualizations that sit outside the browser, either as video, a separate application, or an installation. As an open source language it has built a huge following of creative programmers, designers, and artists look to optimize the potential of data representation and expression. There is a large and dynamic community of experts, authors, and tutorial writers that provide wonderful resources for anyone interested in picking up capabilities in this environment.

There are countless additional JavaScript libraries and plugins that offer specialist capability, such as **Paper.js** (http://paperjs.org/) and **Raphaël** (http://raphaeljs.com/), to really maximize your programming opportunities.

Moving briefly away from interactive programming environments we turn to **R** (http://www.r-project.org/), a highly extensible, open source language for statistical analysis and graphical techniques. R has developed into a powerful and versatile method for creating static charts and graphics that transcend the creative limitations of software packages such as Excel. There is a large

active online community, which can really help with the challenge of going through the learning process. To demonstrate R's worth, the New York Times use this extensively in their data sketching and static graphic workflows. Check out **Mondrian** (http://rosuda.org/mondrian/) and **Wolfram Mathematica** (http://www.wolfram.com/mathematica/) for other powerful, statistical graphing capabilities.

Quadrigram (http://www.quadrigram.com/) is an innovative visual programming environment designed to enable anyone working with data to create powerful, flexible, and custom visualizations. It is intended to be accessible for people with limited technical and programming experience.

In support of open interactive journalism, the **Miso** project (http://misoproject.com/) developed by the Guardian and Bocoup, is a fairly recent arrival. It provides open source tools for developers and non-developers alike to facilitate the quick creation of impressive, extensible interactive data visualizations. There also exists an option for developers to get under the hood and extend and expand the computational methods.

Other notable programming tools to mention include **Nodebox** (http://nodebox.net/), which is a Python-based tool for creating generative, static, animated, or interactive visualizations, and **KendoUI** (http://www.kendoui.com/) for building interactive HTML5-based data visualizations for both web and mobile applications.

Finally, in recent times, we have witnessed the rise of **WebGL** (http://www.chromeexperiments.com/webgl/), a new web technology for rendering interactive two- and three-dimensional graphics. The utilization of this standard has so far seen more experimentation than particularly solid visualization work, but it certainly offers new capabilities for pushing the creative boundaries of data representation.

Tools for mapping

The great opportunity that exists these days for plotting geo-spatial data onto maps is matched by the range of tools available to accomplish it.

Powerful options come in different shapes and forms through **Arc GIS** (http://www.esri.com/software/arcgis), **Indiemapper** (http://indiemapper.com/), **Instant Atlas** (http://communities.instantatlas.com/), **Geocommons** (http://geocommons.com/), and **CartoDB** (http://cartodb.com/). Across these options you will find the ability to create, rich interactive visualizations of geo-spatial data and full-on mapping applications, typically offering flexible licensing and pricing plans from free/trial through to premium levels depending on your needs.

Sample screenshot of Instant Atlas mapping capabilities

For those designers with developer skills wishing to have greater creative control and freedom over their mapping solutions, there are a number of open source mapping frameworks and libraries such as **Polymaps** (http://polymaps.org/), **Kartograph** (http://kartograph.org/), **Leaflet** (http://leafletjs.com/), and **OpenStreetMap** (http://www.openstreetmap.org/).

Additionally, **TileMill** (http://mapbox.com/tilemill/) offers an extremely versatile and accessible application for making elegant data-driven maps whether you are a beginner designer or more-established cartographer.

Other specialist tools

Not all visualizations are interactive, of course, and some of the finest visualization works we see are static pieces. Infographics in particular are typically manually crafted designs, comprising a blend of different visual design elements (such as charts, illustrations, and diagrams). As we have already mentioned, often the chart elements we use for our static work originate from tools such as Excel, Tableau, or R with images imported to help construct a final work.

The vast majority of statics are produced using **Adobe Illustrator** (http://www.adobe.com/uk/products/illustrator.html), the long-established and all-powerful creative package that has been *the* graphic and illustration tool for many years. There is now an open source alternative called **Inkscape** (http://inkscape.org/) ,which offers an impressive array of features that offer a viable alternative for many peoples' needs.

For many people (perhaps those with limited access to varied resources) **PowerPoint** (http://office.microsoft.com/en-gb/powerpoint/) or **Keynote** (http://www.apple.com/uk/iwork/keynote/) provide a perfectly adequate platform for their data presentation needs. Another Adobe package, **InDesign** (http://www.adobe.com/uk/products/indesign.html) provides a further means for creating and publishing final static works.

Elsewhere, for network visualizations, exploratory graphs, and analysis of complex systems check out **KeyLines** (http://key-lines.com/) and **Gephi** (https://gephi.org/).

If you're looking to create advanced motion graphics, modeling, simulation, and visual effects then **Maya 3D** (http://usa.autodesk.com/maya/) and **Adobe After Effects** (http://www.adobe.com/uk/products/aftereffects.html) are incredibly powerful, industry standard production platforms.

Finally, to showcase your static work, once you've created your final designs and want to publish and share image files, sites such as **closr.it** (http://www.closr.it/) or **zoom.it** (http://zoom.it/) enable navigable, zoom-able windows to host large, detailed images.

The construction process

So, you've selected the tools you'll need to build your design and you are now well in to the execution stage. We're not far from the finishing line but it's not yet time for you to lower your guard, lose your focus, or cease your momentum.

You see, this is the part of the design process where stresses and strains emerge—the ill-timed bugs, dataset problems, functional failures, unwanted interference. During this stage it is important that you keep your cool and see your tasks through as efficiently as possible.

As you work through the construction process, it is important to focus on getting the functional elements of your solution working first before spending too much time achieving your desired aesthetic or incorporating technical flair. It is always very tempting to spend too much time, too soon on things that shouldn't really be given such priority. Just remind yourself that there is no point running out of time trying to make something look good when it doesn't yet function. Remember, it will be easier to make something that is functional, beautiful, than it is to make something beautiful, functional.

As we mentioned earlier in the book, you will rarely create a worthy project without the need for iteration. While we have to present the sequences of the methodology in this book in linear fashion, there is always going to be movement forward and backwards between stages. This is something that should be accepted but also embraced—it is part and parcel of any creative process. While a methodological approach to this challenge gives you structure and a neat framework of concerns to work through, iteration gives you the creative breathing room to allow different ideas to blossom and influences to take hold. It is something that you should be prepared to do and plan for.

You clearly want to avoid long iterative cycles but smaller ones can really help you explore, clarify, and refine your potential solutions. It may be that you end up following two or three parallel options to quite an advanced stage and then see which emerges as the strongest. Indeed, some clients will state a need for evidence of alternatives before committing themselves. For these client-based projects, you need to maintain open dialog throughout to avoid any inconsistency in interpretation from either party. Do your absolute best to eradicate the possibility of last minute surprises about a solution not matching requirements or expectations. That is a sting in the tail nobody wants!

As you work through your construction stages, in particular, there will be points when you recognize a need to make certain sacrifices. There may be things you intend to include but can't justify them. Trade-offs are a constant necessity caused by time or resource constraints.

Some of the things we find hardest to drop are the most irrational. We often find ourselves in a sense of denial. This may come from a desire to include features that you have slaved over or become overly precious about.

We saw an example of this in the discussion about color in Chapter 4, *Conceiving and Reasoning*

Visualization Design Options. Here we saw an initially conceived title format for an Olympics project that was formed out of thumbnail images of all the historical event posters; this image is shown here:

When it came to incorporating this title into the final piece, it was clear that it drew too much attention away from the rest of the visualization. Despite this being obvious, because of the time and energy spent on making this title image, it was hard to relinquish. Thankfully, a sensible voice determined that we should drop it and find a simpler solution. Simple advice? Take the hit and just get over it!

Image from "Pursuit of Faster" (http://www.visualisingdata.com/index.php/2012/07/new-visualization-design-project-the-pursuit-of-faster/), by Andy Kirk and Andrew Witherley

As we approach the maturing stages of our development work, this idea of getting input from others becomes more important. It can be quite a tough moment to convince yourself that something is ready to be judged (in a prelaunch setting) but it is invaluable to test out people's responses to what you are creating.

You want people who are informed about the context of the work and also about the challenges involved in creating a visualization. You also need to trust them to give you constructive and reasonable feedback, otherwise it may prove a wasted effort.

You should be seeking feedback on a number of dimensions of your design in order to determine if the intention of your solution is consistent with the audience experience:

- What is their instinctive reaction? Positive, negative, intrigue, confusion, or just a plain "so what?"
- Can they understand how to read the graphic or use the tool? Does it have clear explanations and intuitive design in terms of visual hierarchy and structural arrangement?
- Can they derive insight from it? Maybe throw them some test questions to assess the visualization's ability to effectively inform.
- Does it work functionally? Can they find any errors, mistakes, programmatic errors, or any other design flaw that undermines the clarity, accuracy, or performance of the solution?

There are plenty of evaluation methodologies and techniques, probably much more sophisticated than this, but these are just some of the most useful prompts for you to gather feedback against before finalizing your work.

Approaching the finishing line

Here is a quote from Antoine de Saint-Exupery:

> *"You know you've achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away."*

The finishing line is now getting ever closer. However, apart from those projects where there is a clear finite deadline to work to, the judgment of when a design is actually finished is not necessarily always obviously recognizable. A deadline provides this finality, but open-ended projects need their own completion point to be determined. It is natural to keep tweaking, refining, and enhancing your piece but eventually you need to call out something as being completed.

A useful signpost to note your progress was proposed by designer Martin Wattenberg (co-developer on the "Wind Map" project that we saw earlier). Martin describes the subtle but telling change in your role as you shift from debugging a design (programmatically or figuratively) to finding yourself becoming an enthusiastic user, engaging with your own work to unearth insights.

As the quote at the start of this section expresses, another viewpoint is to step back and away from your design and challenge everything that you have included. Justify to yourself (and/or to others) the reason why features or design choices need to stay, but also determine what elements you can eliminate, those that don't add any communicative or functional value. It's not necessarily about striving for minimalism; rather the most elegant and clear form.

As well as challenging all our design choices we also need to switch mindsets more towards the Project Manager or Administrator's perspective and undertake some important checks. Sometimes, when you're close to finishing you would prefer to stick your head in the oven than seek issues that need addressing, but you've got to continue to strive for optimal accuracy and intercept any potential mistakes.

Simple errors can completely undermine quality—an extra zero in a value, the mislabeling of a country, an emboldened font when it wasn't wanted, and so on. It might be the smallest and most innocent of mistakes, but that can be enough to tarnish the rest of your work with doubt in the eyes of your audience.

Try to see this as the final push. Paying attention to the finer details of your work will safeguard the project's integrity (and by extension your own, as the designer). Hopefully, much of the user-testing and evaluation work outlined just before will help in the identification of any problems in accuracy. People with a freshness of perspective can often provide great value on this front.

Whether it is them or you looking for these characteristics, here are a few things you need to watch out for:

- **Data and statistical accuracy**: Scan through a good-sized sample of all your visualized data

values to ensure there aren't any erroneous items or incorrect outliers. Check the rigor of all your statistics and calculations.

- **Visualization accuracy**: Make sure that the way you have represented your data is functioning effectively and does not mislead the user or reader. Do all your representation choices accurately portray the data values they're associated with?
- **Functional accuracy**: More concerned with interactive pieces—do all the functions and features on your design perform as you intended?
- **Visual inference**: As we stated before, visual inference should equal data inference. If it looks like data, it should be data. If something looks significant, maybe through its positioning or color choice, then it should be significant. If there is any decorative element or other artifact that appears to be implying something it is not meant to, remove it.
- **Formatting accuracy**: Check the consistency of your typography, in terms of type, style, and size. Make sure your color usage is consistent down to the RGB or CMYK code level.
- **Annotation accuracy**: Read through all your titles, labels, introductory text, credits, captions, and check any units that you have included. It's not just about spelling or grammatical errors but checking to see if things make sense and are succinctly expressed.

Post-launch evaluation

The exciting and also probably anxious moment has arrived and your visualization has now been launched in to the wild!

How, where, and what this launch actually looks like clearly covers a very broad range of possibilities—it might be a chart in a report, a presentation to a board meeting, an infographic in a newspaper, or an interactive web-based project.

Regardless of how this piece exists, in an ideal world you would now seek to assess the visualization's effectiveness and impact in a post-launch setting. I say in an ideal world because sometimes you simply don't have sufficient capacity or resources to allocate to the post-launch evaluation.

However, you should still care to seek an assessment of how well your project has served its purpose. Has the reaction and consequence of the work been consistent with its intent and reason for being created, as we determined earlier in the process? It is important to recall the following terms of reference because they frame the type of feedback we seek:

- Was there a positive reaction to the piece we created?
- Did it deliver the appropriate tone of voice?
- Did it reach the intended audience type and volume?
- Were users able to effectively consume or discover insights?
- Where we had a set idea of the intended consequences of this work, were they experienced?
- What problems did people experience, if any?

To obtain feedback of this type and breadth we must consider multiple channels. Each of the following options provides an incremental level of evaluative value but consequently also requires a proportionate increase in the amount of time, effort, and probably cost to obtain:

- **Metrics and benchmarks**: For web-based visualizations there are a number of easily obtainable measures to indicate the reach and popularity of your project. The traditional analytic measures for page views, visits, and visitors can now be easily supplemented with social media metrics such as Tweet counts, Facebook likes, Google+ shares, and so on. These are very simple, cheap, and accessible indicators to help you form a basic understanding of your design's utilization. What you need to think about is: what does success look like? What are the relative benchmarks of performance against these measures that will inform your overall satisfaction?
- **Client or customer feedback**: Of course, the most tangible form of feedback for many projects will come from those who have asked or commissioned you (and hopefully paid you) to create the solution. You'll learn in no uncertain terms whether or not what you created fell short, matched, or exceeded their expectations. Sometimes, you have to judge yourself against the requirements outlined to you and not the resulting performance. After all, you can only respond to the brief you were given.
- **Peer review**: Sometimes the most important and constructive evaluation can come from peers,

perhaps expert practitioners or thought leaders. In the visualization field, there are many examples of bloggers who will conduct a review and critique of new work. Getting visitor hits is one thing but receiving a positive review and mention from a peer is worth its weight in gold.

- **Unstructured feedback**: This type of evidence might come via online comments forms, reaction on social media, or through anecdotal channels (e-mails, in-person conversations, perhaps overheard comments) to add a layer of qualitative reaction and evidence of success or failure.
- **Invite user assessment**: Rather than placing value on anecdotal or reactionary and opportune feedback, you could be more proactive by offering simple mechanisms for users to provide more structured qualitative responses, perhaps through small-scale questionnaires.
- **Formal case studies**: Taking things to a more advanced level of evaluation (almost academic in its nature), case studies can take many forms using techniques such as interviews, observations, and controlled experiments, where you might set tasks, manipulate conditions, and record responses. These will often be undertaken by an independent observer to offer that degree of integrity.

No matter through which of these methods you obtain your evaluation feedback, you should be prepared for and accept criticism. Of course, in this digital age everyone is a critic—and too often anonymous—but you should always welcome constructive feedback and use this to fuel your development.

Finally, from your personal point of view, how effective did *you* think it went? Your own satisfaction is very important because this is what also drives your future decisions and development. Often we'll know best whether something could be considered to be an effective outcome and a satisfying process. Even if the results are very positive, there may be many things you thought could have gone better:

- Did you accomplish the outcomes you wanted?
- Did you create something you were satisfied with?
- Were you satisfied with how you rationalized the choices?
- Maybe you hated the project, the client, or the subject matter
- Perhaps you spent far too long on the work and you haven't been paid or rewarded sufficiently for the time you invested
- Maybe you regret consuming so much caffeine late at night

Try not to weigh yourself down with too many thoughts of regret around "wish I'd not done this" or "wish I'd managed to include that". Instead, put all reflection to best use as a learning experience to inform your development and preparedness for future opportunities.

Developing your capabilities

The project is over. You can take a deep breath and relax. Well, at least for an hour until your next project is lined up!

For you the bigger picture now is to consider your ongoing development in this discipline, learning from each experience, and building up your expertise.

The single most important message that I want to put across in this book is the value of practice, experience, and ongoing self-improvement. Data visualization is such a multidimensioned and rapidly evolving craft that cannot be mastered overnight.

Earlier we looked at the framework of the "eight hats". Through assessing yourself against this collection of capabilities, skills, and attitudes you can self-determine where your strengths and weaknesses may be and then look to address them. There are several strategies to help ensure your development continues.

Practice, practice, practice!

When it comes to developing your practical design skills the major piece of advice is simple—practice, practice, and more practice. There are so many different variables and subtle challenges involved in every project that you can't fail to learn from each project that you undertake.

We've just reinforced that data visualization is a craft. You need to continue to exercise your creative and analytical muscle to stay in good shape.

If time permits, try forcing yourself to stick to a practice agenda: maybe, do small personal projects every week then a bigger project every month. You might never launch the work in public but just testing yourself against the challenges of gathering data, analyzing, and presenting it will help maintain your development.

An especially ideal opportunity for practice exists through the frequent data visualization contests that are held these days, often with added incentive of prizes for the best-judged work. These typically involve a basic design brief, a published dataset, and a timeframe to create a compelling solution. A great value of these contests comes from seeing all the other solutions that are submitted. This lets you learn how others have tackled the same problem but maybe in different ways as compared to your own approach.

I have already stressed the importance of maintaining a written record of how you have tackled your design projects. It is worth repeating because it will really help you identify areas for improvement both in terms of effectiveness and efficiency. It will also be a useful reference guide should you ever need to take on a similar problem or comparable dataset.

Also, keep all your trash! Whether it is sketches on paper or little developments on the computer that you deemed redundant, where possible, keep them because you never know when they might come in useful.

Earlier in this chapter, we profiled the importance of technology and the potential limitations of your own capabilities. It is up to you to decide how far and the direction you may wish to take your technical skills. You may not always have the time or opportunity but if you are really serious about advancing your visualization design skills, you should try to push yourself beyond your comfort zone. Rather than relying on the same old tools, pushing them to do things that they're not really designed to do; try out new software, applications, and programming environments. Accept that there will be relatively steep learning curves involved but that the rewards could be great.

Evaluating the work of others

One of the most effective ways of sharpening your visualization design "eye" is by evaluating other designers' work. Not necessarily through providing formal feedback, but just testing your reaction and analysis of the designs you see.

Try to take on the dual mindset of a user and of a designer, in order to undertake a forensic assessment of what has been produced and how well it works using the following prompts:

- What one word describes what is your immediate, instinctive reaction? Is it positive or negative sentiment?
- If it is not necessarily an "instant" piece, does it have the qualities of a "slow-burner", seemingly becoming more appealing after a certain duration?
- What purpose do you think the designer had in mind? Does the style and function of the end product match the likely intention?
- We rarely create these pieces in perfect project conditions, so consider what type of inherent factors might have surrounded and influenced this project? Does a sense of sympathy with the possible influencing factors of the design process effect your impression?
- Work through the five design layers and ask yourself how well each has been executed and what improvements could have been made?

Also think about the general design considerations we outlined about creating accessibility through intuitive design, as well as the idea of reward versus effort, and see how these qualities are achieved.

Eventually, with enough practice, you will develop your critical eye and will become much faster, more informed, and fairer in your judgments of other peoples' work. This will be a great way to educate your own design techniques and refine your own style.

Publishing and sharing your output

One of the contemporary ways of developing your capabilities is to publish yourself. A platform such as a blog will create an ideal means of sharing your work and your ideas.

Posting your design work and building up a public portfolio of your projects creates a virtual shop window. You can take advantage of this format and share narratives about your design process, explaining to people how and why you arrived at the various solutions.

Writing articles, publishing critiques of work, and facilitating discussions are also great ways of promoting yourself. It helps you to learn about the subject. As you write about a topic, you are forced into developing a conviction, to structure arguments, and learn about different perspectives. It really does sharpen your views remarkably, even if (initially, at least) the only visitors to your site are your devoted parents, out of duty.

Eventually, through hard work and dedication, you will create an interested audience and this opens up wonderful opportunities for developing connections with other practitioners, creating rich networks around the world, across demographics, and beyond your subject field.

If you don't have the energy, time, or enthusiasm for a commitment such as a blog, then there are plenty of online galleries and communities through which you can share and publicize your work.

Immerse yourself into learning about the field

Over the past few years we have seen a relative explosion in the amount of online content covering the subject of data visualization, infographics, and data-driven journalism. Websites, blogs, designer sites, and online galleries are now bursting at the seams with interesting articles, new tools, latest projects, and endless amount of inspiration. Social media too is a wonderful platform to learn about key opinions and contemporary developments. The visualization field is particularly active on Twitter where you will find a very spirited and positive community.

Immersing yourself in the array of online resources will keep you up-to-date with contemporary developments. The following is a list of just a small selection of some of the best websites that you should visit and keep a track of. They have been loosely organized by their general remit, though many offer a wide variety of value:

Latest projects, trends, articles, announcements, and developments:

- Visualising Data (http://www.visualisingdata.com/)
- Information Aesthetics (http://infosthetics.com/)
- Flowing Data (http://flowingdata.com/)
- DataVisualization.ch (http://datavisualization.ch/)
- Visual.ly (http://blog.visual.ly/)

Discourse around data visualization:

- Perceptual Edge (http://www.perceptualedge.com/blog/)
- The Functional Art (http://www.thefunctionalart.com/)
- Eager Eyes (http://eagereyes.org/)
- Fell In Love With Data (http://fellinlovewithdata.com/)
- Michael Babwahsingh (http://michaelbabwahsingh.com/)

Design narratives, process, and project critique:

- Charts 'n Things (http://chartsnthings.tumblr.com/)
- The Why Axis (http://thewhyaxis.info/)
- Junk Charts (http://junkcharts.typepad.com/)
- Graphic Sociology (http://thesocietypages.org/graphicsociology/)
- National Geographic (http://juanvelascoblog.com/)

Design or technical tutorials, advice, and much more!:

- Scott Murray (http://alignedleft.com/)
- Jérôme Cukier (http://www.jeromecukier.net/)
- Jim Vallandingham (http://vallandingham.me/)
- Gregor Aisch (http://vis4.net/blog/)
- Naomi Robbins, Forbes (http://blogs.forbes.com/naomirobbins/)

Visualization communities, designers, design agencies and general inspiration:

- Visualizing.org (http://visualizing.org/)
- Information is Beautiful Awards (http://www.informationisbeautifulawards.com/)
- Any New York Times design (via http://www.nytimes.com/)
- Guardian datablog (http://www.guardian.co.uk/news/datablog)
- Stamen (http://content.stamen.com/)
- Pitch Interactive (http://www.pitchinteractive.com/beta/index.php)
- Periscopic (http://www.periscopic.com/)
- Moritz Stefaner (http://well-formed-data.net/)
- Santiago Ortiz (http://moebio.com/)
- Tulp Interactive (http://tulpinteractive.com/)

It should go without saying that an intimate appreciation of the many books about and around the subject is vital for learning this craft. Of course, you have already shown great wisdom in choosing this book but there are so many fascinating and invaluable titles to choose from. You can find a list of the most influential titles by visiting http://www.visualisingdata.com/index.php/resources/.

It is also important to expose yourself to influences from outside the specific boundaries of this field. You can pick up a great deal of inspiration from reading about graphic design, architecture, product design, typography, move-making, video game design, and journalism—all areas from which we can translate, transport ideas, and learn.

Another critical layer of learning comes from the world of academia and the value of keeping abreast of latest research and studies, from which many tools and best practices naturally emerge. The open access movement is gathering pace and making academic literature much more accessible to those not directly affiliated with academic institutions.

Conferences are also naturally a great way to keep in touch with the very latest developments, hearing from great speakers, and seeing inspirational presentations of case studies and examples. You also get to interact with other similarly passionate practitioners, something that can prove very rewarding.

Beyond these options, clearly a further avenue is through formal training and these days this comes in all shapes and sizes—from online tutorials, video tutorials, and webinars, through to in-person training courses and undergraduate or postgraduate programs. Scour the Internet to find the right solution for you.

Whichever way you go about developing your skills and knowledge, you can be sure there will be plenty of support from across the field. Data visualization is blessed with a wonderful positive and supportive community of incredibly talented and humble people, so you will always be met with a warm welcome.

Summary

We have now come to the end of this design journey. Hopefully, you got to your destination smoothly and you didn't experience many frights along the way!

In this chapter, we have focused on the execution stage of the visualization design process, bringing form to all your preparatory efforts, and transforming your concept into a produced work. We have introduced some of the most useful technologies to give you a flavor of the variety of tools being used for the different stages or types of visualization design.

We have looked at some of the important final steps to take before launching your design, the importance of running final checks across all elements, and conducting testing to get an evaluation of your solution before launch.

Once launched, it is then your prerogative to seek evidence of the impact of your work and we outlined a number of different tactics for undertaking this.

Finally, we suggested some strategies for you to consider pursuing to continue to develop your data visualization skills, knowledge, and experience. This will give you the best chance of taking your capabilities forward and achieving success in this thoroughly exciting field.

Good luck to all of you with the visualization challenges you take on in the future and thank you so much for taking the time to read this book. I hope it helps in any way possible!

Part 2. Module 2

Social Data Visualization with HTML5 and JavaScript

Leverage the power of HTML5 and JavaScript to build compelling visualizations of social data from Twitter, Facebook, and more

Chapter 1. Visualizing Data

A scant few years ago this book would not have been possible. The rapid expansion in social media, data processing, and web technologies has enabled a fusion of divergent fields. From this fusion we can create fascinating displays of data about exotic topics. The beauty that is inherited in data can be exposed in a fashion that is accessible to the masses. Visualizations such as the following word map (http://gigaom.com/2013/07/19/the-week-in-big-data-on-twitter-visualized/), can unlock hidden information while delighting users with an extraordinary experience:

Berkeley
commands
Useful
Unix
Nate
degree
Information
online
Want
fork
Master
Big
000
line
never
Becoming
http
working
master
DataScience
60
Silver
That'll
Analytics
bigdata
@kdnuggets
command
@petewarden
University
Online
unix
100
master's
Berkeley
School
@mikeloukides
Kaggle
100K
now
first
Master's
UC
new
amp
Next
@timoreilly
students
60K
job
BigData
trust
looking
quick
need
scientists
Cost
program
Program
@gigaom
Berkeley's
Python
Top
analytics
love
Curriculum
@antirez
Rise
@greda
post
California
gt
Metromap
announces
Summer
Reading
Reda
awk
lunch
launches
Greg
like
Tabs
Required
Matter
Insights
Trust
Spartz
de
Looking
jobs
News
Hog
tuff
job
always
Business
UC
@wired
know
Dark
@2Uinc
datascience
tools
years
MT
Whale
Applied
@UCBerkeley
education
Education
Intervention
Cloudera
good
Competitive
Competitive
Tech
July
Ten
Masters
Masters
@berkeleyISchool
sort
today
Today's
intro
aggregating
Calls
still
New
Becoming
grouping
shopping
Sort
Become
Calls
List
Klaugen
big
file
20
Top
Python
analytics
Berkeley's
July
Tech

There's a lot of data out there

It shouldn't come as a surprise to anybody that the amount of data humans are recording is growing at an amazing rate. Every few years the data storage company EMC produces a report on just how much data is being preserved (http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf). In 2012, it was estimated that between 2005 and 2020 the amount of data stored globally will grow from 130 to 40, 000 exabytes. That works out at 5.2 terabytes for each person on the planet. It is such a staggering amount of information that understanding how much of it exists is difficult. By 2020, it will work out to 11 spindles of 100 DVDs per person. If we switch to Blu-ray discs, which have a capacity of 50 GB, the stack of them required to store all 40, 000 Exabytes would still reach far beyond the orbit of the moon.

The growth in data is inevitable as people put more of their lives online. The adoption of smartphones has turned everybody into a photographer. **Instagram**, a popular image sharing site, gathers some 40 million photos a day. One wonders how many photos of people's meals the world really needs. In the past few months there has been an explosion of video clip sharing sites such as Vine and Instagram, which generate massive amounts of data. A myriad of devices are being created to extend the reach of smartphones beyond gathering photographic data. The latest generation of smartphones include temperature, humidity, and pressure sensors in addition to the commonplace GPS, gyroscopic, geomagnetic, and acceleration sensors. These allow for recording an accurate representation of the world around the user.

An increase in the number of sensors is not a trend that is limited to smartphones. The price of sensors and radios has reached a tipping point where it is economical to create standalone devices that record and transmit data about the world. There was a time when building an array of temperature sensors that report back to a central device was the realm of large SCADA systems. One of my first jobs was testing a collection of IP-enabled monitoring devices at a refinery. At the time, the network hardware alone was worth millions. That same system can be built for a few hundred dollars now. A trip to a crowdsourcing site such as Kickstarter or Indiegogo will find countless Bluetooth or Wi-Fi enabled sensor devices. These devices may find your lost keys or tell you when to water your tomatoes. A huge number of them exist, which suggests that we're entering into an age of autonomous devices reporting about the world. A sort of Internet of things is emerging.

At the same time, the cost per gigabyte of storing data is decreasing. Cheaper storage makes it economical to track data that would have previously been thrown away. In the 1970s, BBC had a policy of destroying recordings of TV programs once they reached a certain age. This resulted in the loss of more than a hundred episodes of the cult classic *Doctor Who*. The low data density of storage media available in the 1960s meant that retaining complete archives was cost-prohibitive. Such deletion now would be unimaginable as the cost of storing video has dropped substantially. The cost for storing a gigabyte of information on Amazon's servers is on the order of a penny-a-month and can be even cheaper if the right expertise are available in house. The Parkinson's law states the following:

Work expands so as to fill the time available for its completion.

In a restatement of this law, in our case, it would be "*the amount of data will grow to fill the space available to it*."

The growth in data has made our lives more difficult. While the amount of data has been growing, our ability to understand them has remained more or less stagnant. The tools available to refine and process large quantities of data have not kept pace. Running simple queries against gigabytes of data is a time-consuming process. Queries such as "list all the tweets that contain the word 'Pepsi'" cannot be realistically completed on anything but a cluster of machines working in parallel. Even when the result is returned, the number of matching records is too large to be processed by a single person or even a team of people.

The term "Big Data" is commonly used to describe the sorts of very large datasets that are becoming more common. Like most terms that have become marketing terms, Big Data is defined differently by different people and companies. In this book we'll think of it as any quantity where running simple queries using traditional database tools on consumer grade hardware is difficult due to computational, storage, or retrieval limits.

Understanding the world of Big Data is a complex proposition. Visualizing data in a meaningful way is going to be one of the great problems of the coming decade. What's more, is that it is going to be a problem that will need to be addressed in domains that have not been traditionally data-rich.

Consider a coffee shop; this is not a company that one would expect would produce a great deal of data. However, consumers who are hungry for data are starting to demand to know from whence the beans for their favorite coffee came, for how long they were roasted, and how they were brewed. A similar program called **ThisFish** already exists that allows consumers to track the origin of their seafood (http://thisfish.info) all the way back to when it was caught. Providing data about its coffee in an easily accessible form becomes a selling feature for the coffee shop. The following screenshot shows a typical label from a coffee shop showing the source of the beans, roasting time, and organic certification:

People are very interested in data, especially data about their habits. But as interested as people are in data, nobody wants to trawl through an Excel file. They would like to see data presented to them in an accessible and fun way.

Getting excited about data

The truth is that data is interesting! It's amazingly interesting because it tells a story. The issue is that most of the time that story is hidden behind a raft of seemingly uninteresting numbers. It takes some skill to extract the key data and display it to people in a meaningful way. Humans are visual creatures and are more readily able to process images than tables of numbers.

The best data visualizations arise from a sense of passion in the subject of your visualization. Don't we all work better if the subject of our work is something in which we're really interested? Great visualizations don't just educate their viewers, they delight their users. They present data in a novel way that is still easily understood by the audience. Great visualizations strip away the excess information to reveal a kernel of information. At the same time, great visualizations have a degree of beauty to them. Don't be fooled into thinking that this beauty serves no purpose. In a world of ever shortening attention spans, there is still a place for beauty. We still stop and pause for a moment when presented with as aesthetically pleasing visualization. The extra few seconds that the beauty buys you may be what keep people interested long enough to take in your meaning.

Even the most benign data has a story worth telling. To most, there is very little that seems less interesting than tax revenue statistics. However, there have been some very compelling stories found within that raft of data. The data tells a story about which companies are avoiding paying tax revenues. It tells another story about which cities have the highest per capita income. Within that boring data are countless interesting stories that can be extracted though a passionate application of data visualization.

Data is a lot of things, but it is never boring. You can get excited about data too and uncover the hidden stories in any dataset. In every dataset, there is an interesting conclusion waiting to be exposed by a data sleuth such as yourself. You should share your excitement with others in the form of data visualizations.

Data beyond Excel

By far, the most popular data manipulation and visualization tool in the world is Microsoft Excel. Excel has been around for almost three decades, and during that time has grown to be the de facto tool on which businesses rely to perform data analytics. Excel has the ability to sort and group data and to create graphs for the resulting information.

As we saw previously, the amount of data in the world is huge. The first step in most data visualizations is to filter and aggregate the data down into a dataset that contains the key insights you want to share with your users. If it sounds like extracting, meaning that it is an opinionated process, that's because it is. Presenting an unbiased visualization is just about impossible. That's okay, though. Not everybody is an expert on your data, and guiding others to your conclusions is valuable.

You'll find that the data you have from which to derive visualizations is hardly ever in a format you can use right off the bat. You will need to manipulate the data to get it into a form you can use. If your source dataset is small enough and your manipulations sufficiently trivial, you may be able to do your preprocessing in Microsoft Excel. Excel provides a suite of tools for sorting, filtering, and summarizing data. There are numerous books and articles available on how to work with data in Excel as well as how to create graphs, but we won't delve into it here.

The problem with Excel is that it is old news. Everybody has seen the rather pedestrian graphs you get out of Excel. With the exception of a couple, these are the same charts which were produced by Excel 95. Where is the excitement about data? It seems to be missing. If you create your visualizations wholly in Excel, your users are going to miss out on your enthusiasm for data.

Swiss army knives are famous for having a dozen different features. You can use the same tool to open a bottle of wine as you use for removing stones from the shoes of horses (a far more common application around most parts). When you build a tool to be multi-functional, you end up with a tool that does nothing particularly well. Simply looking at the length of the help index for Microsoft Excel should tell you that Excel falls solidly into the category of multi-functional tools. You can do your accounting with Excel or track how quickly you can run a 5K; you can even build graphs with that data. But what you can't do is build really good graphs. For that, you're going to need specialized tools with a narrower focus on data visualizations.

Social media data

We've talked a lot about visualizations and data, but the other part of this book's title is to do with social media. Unless you've been living in a cave without Internet, you will be at least slightly aware of the social media wave that has swept the planet in the last decade. Has it really been only a decade? Facebook was founded in 2004. While one can point to examples from before 2004, I would argue that Facebook was the first social media site to enter into the common consciousness of the population.

Defining what exactly makes a site a social media site is difficult. There needs to be some aspect of social interaction on the site and some sort of a connection between the users. To avoid labeling any site with a comment section as a social media site, the primary purpose of the site must be to enable the interactions between users. Content on these sites is typically user-generated rather than being created by the owners of the site. Social media sites enable interaction between users with similar interests.

Why should I care?

The role that social media now plays in our world cannot be understated. Even if you avoid membership in all social media sites and believe that social media has no impact upon your life, it does. A great example of the real-world impact of social media is how reliant news media has become on social media. Earlier this year, the Associated Press' Twitter account was hijacked and several messages were sent suggesting that the White House had been attacked by terrorists. While the news was quickly rebuffed, stock markets declined sharply on the news. Had the subterfuge not been so quickly discovered, the real-world consequences could have been far worse.

Data from social media provides a context for events happening around the world. One has simply to look at the trends on Twitter to pick out the important news stories of the day. As newspaper subscriptions drop, the number of people on Twitter, Facebook, and other sites grow. Traditional news outlets have started to integrate commenting and sharing on their stories via social media. The commentary on the story often becomes the story instead of simply providing a meta-story. Many commentators have pointed to the importance of Twitter in the Arab Spring and even in protests in the US. Social media is quickly overtaking more traditional sources of news and is becoming a driving force for society.

Social media is not limited to person-to-person interactions. More and more, it is being used by businesses to connect with their customers. Frequently, the best way to get service from a faceless corporation is to post a message on their Facebook page or send them a tweet. I have certainly had the experience of tweeting about a company or a service only to have their social media people reach out to me. Anything that empowers companies to develop better relations with their customers is a powerful tool and likely to have a long life.

From the perspective of visualizations, social media is a phenomenal source of interesting data. I can think of very little that is as compelling as a source of data as the social interactions between people.

Humans evolved to be social animals so we have a built-in interest in what is happening in our social circles. In addition to their websites, many social media sites have APIs that promote building applications that use their data. The theory is that if they can enable an ecosystem for their valuable data, people are far more likely to visit their site frequently and third-party applications may even draw in new users.

Social media is the very definition of Big Data. Facebook has something like a billion users, each of which may generate a dozen pieces of data a day. Twitter, LinkedIn, and Facebook have all created their own database technologies after having found the amount of data with which they have to deal, to be too large for traditional databases. Fortunately, there is little need to work with the full scale of social data. Narrower sets of data can be accessed through the various data access APIs. The key is to shift as much of the filtering and aggregation to the social media sites as possible. By exploring the available information, it is possible to draw interesting conclusions and expose information through visualizations that aren't typically apparent to users.

HTML visualizations

The final piece in the puzzle is HTML5. When I was young, a new version of HTML meant another long-winded specification from the World Wide Web consortium. The specification process for a new version of HTML would take several years and would be planned out by a committee with members from large technical organizations such as Microsoft and IBM. While there is an HTML5 specification, it is not as formal as previous iterations. The term HTML5 has come to describe a collection of future-oriented technologies that can be used to create powerful web applications.

HTML5 includes specification for diverse features such as the following:

- Web workers (multi-threaded JavaScript)
- Touch events for touchscreen devices
- Micro-data formats
- Canvas
- Scalable Vector Graphics
- Camera API
- Geolocation API
- Offline data

Through these new APIs and features, HTML5 has become a major player not just on browsers, but also on mobile devices and on the desktop. Through toolkits such as PhoneGap (http://phonegap.com/), HTML and Microsoft's WinJS JavaScript can be used as primary development languages on iPhones, Android, Windows Phone, and even Blackberry. The native APIs are bound to JavaScript equivalents opening up the camera, GPS, and filesystem to JavaScript applications. HTML5 can also be used as a development platform for Windows 8-style applications (previously known as Metro). On non-Windows platforms, desktop applications can be developed in HTML5 using a toolkit-like Adobe Air (http://www.adobe.com/products/air.html). HTML5 offers a multi-platform development environment that allows taking skills from the Web to tablet to desktop.

The offline data tools remove the dependence on having a web server to serve content to your application. Embedding data directly on the client machine instead of having to pull it down repeatedly from a server allows for applications to be truly mobile—the network is no longer crucial.

HTML5 has been hugely beneficial to visualization developers. Canvas and SVG both offer enticing functionalities. CSS3 also allows for a greater degree of flexibility around styling. Before HTML5 came onto the scene, interactive data visualizations in a browser could best be achieved using third-party tools such as Java Applets or Adobe Flash. The adoption rates for these technologies, while high, still cut off a large number of users. Even with high adoption rates, the versions of these tools being run in the wild were frequently archaic. Neither Java Applets, nor Adobe Flash is available on the increasingly popular mobile platforms. HTML5, on the other hand, is now supported in some form on the vast majority of smart phones.

One of the best features of developing a visualization in HTML is that it is possible to allow users to

interact with the visualization. Famous visualizations such as the *London Underground Map* have been crippled by being drawn on a static piece of paper. Interactions provide a whole new level of user engagement—previously impossible. It should not surprise you if users of interactive visualizations find ways to manipulate the visualization to derive whole new conclusions.

The industry support for HTML and JavaScript technologies is impressive. All the technology giants have invested heavily in developing browsers and development tools based on HTML5 and JavaScript. The pace of change in the web development sphere is stunning. There is not a week that goes by when I fail to hear about an innovative new JavaScript library or a new take on a development platform. The ready availability of cloud-based hosting has enabled startups to flourish on the web.

When choosing a tool in which to develop visualizations, HTML provides an excellent option. Broad support, good tooling, and a well-known API ensure that developing will be a pleasure. Well, maybe not a pleasure, but at least relatively painless. HTML and JavaScript are the lingua franca for all web developers. No matter if development is being done with Ruby on Rails, ASP.NET, or even Wordpress as a backend, the frontend is always going to be written in HTML and JavaScript. This gives a big pool of developers from which talent may be pulled.

Summary

Communicating information to users is tricky. The problem is compounded by the huge quantity of data that is now available at the click of a mouse or the punch of a key. As a visualization developer, it is your role to sort through clouds of irrelevant data to extract the bits in which you're interested and then to present that data to your users in an interesting way. People are interested in data, but they are rarely interested in sorting through reams of tabular data. Visualizations are frequently the best tools for presenting that data to your users.

The confluence of readily accessible, high quality, social data from social media sites coupled with new visualization tools present a never before seen opportunity to create interesting visualizations. Through the passion of developers who can see beyond the standard Microsoft Excel graphs and tables, there is a future for not just static visualizations but also interactive, fun visualizations that will delight users while they explore previously invisible aspects of data.

In the next chapter we'll examine some of the ways in which we can create visualizations using modern web development tools.

Chapter 2. JavaScript and HTML5 for Visualizations

In the previous chapter I mentioned that there have been some developments in HTML5 that have made visualizations far easier. This chapter is going to explore a couple of them, namely the following:

- Canvas
- Scalable Vector Graphics (SVG)

If you're familiar with the functionality of these two tools, you might want to give this chapter a skip.

Canvas

Canvas is one of the features of HTML5 that gets the most play in technical articles and in demos. The things that can be done with canvas are very impressive, so it is no surprise that it appears so frequently. Canvas provides a low-level bitmap interface for drawing. You can think of it as Microsoft Paint in a browser. The images that are generated on the canvas are all raster images, meaning that the images are built from a grid of pixels rather than a set of geometric objects, as is the case in a vector image. Interaction with elements drawn on a canvas must be done through filters and global transformations; precise control is problematic if not impossible.

Creating a canvas element on your page is simple. You simply need to add an HTML element such as the following:

```
<canvas width="200" height="200">Alternate text here</canvas>
```

Tip

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.

You can also find all the code examples for this book at https://github.com/stimms/SocialDataVisualizations.

This will create a square canvas of 200 x 200 pixels. Should the browser being used not support the canvas element, the alternate text will be shown. Fortunately, it is pretty rare that you'll see this warning, as canvas has wide support. By the middle of 2013 canvas was available to 87 percent of Internet users. For the latest numbers, check out http://caniuse.com/#feat=canvas. This site also has browser support information for other HTML5 features. There is even support for mobile browsers on iOS, Android, BlackBerry, and Windows Phone. The only commonly used browsers that don't have support for canvas are versions of Internet Explorer prior to Version 9.

If you're in a situation where your target demographic makes heavy use of older Internet Explorer versions, all is not lost. There is a handy JavaScript **polyfill**; a downloadable piece of code that provides canvas functionality to older versions of Internet Explorer. The library is available at https://code.google.com/p/explorercanvas. To make use of it, a conditional include may be used, as shown in the following code snippet:

```
<head>
<!--[if lt IE 9]><script src="excanvas.js"></script><![endif]-->
…
</head>
```

This will only include `excanvas.js` if the browser running is an older version of Internet Explorer.

Excanvas does not fully support canvas, with features such as shadows and 3D missing, but the vast majority of the features are available. There is also a performance hit to using the JavaScript version of canvas, so, if you make use of animations, they may not be as smooth as they would on a modern browser. This is a small price to pay for reaching the remaining few percent of users. As browsers are updated, this problem will become less and less important.

Drawing simple shapes on the canvas is easy, but there is also the power to draw some very complex objects, including drawing in 3D. We'll limit our discussion to some of the simpler shapes and functions that would be useful in creating visualizations.

Before we get into drawing, we hold the introduction of the coordinate system for canvas. The origin of the coordinate system is located in the top-left corner and grows towards the bottom-right. To draw on the canvas, as shown in the following figure, JavaScript is used:

The first step in drawing is to get a handle for the instance of the canvas to which you would like to draw. You can, of course, have as many canvas elements on a page as you wish. Here, we'll create a canvas with an ID of `demo`:

```
<canvas height="200" width="200" id="demo"></canvas>
```

We can now select the element using standard JavaScript methods:

```
var demoCanvas = document.getElementById("demo");
```

Alternately, if you're making use of a CSS selector library, such as jQuery, you can select the element using that:

```
var demoCanvas = $("#demo")[0];
```

The next step is to get a reference to the drawing context itself. The context contains a collection of methods for drawing and will be what we use for all canvas operations:

```
var context = demoCanvas.getContext("2d");
```

Canvas supports a number of basic shapes from which it is possible to build complex objects. The simplest shape is the lowly rectangle. This can be created by calling the `strokeRect()` function:

```
context.strokeRect(20, 30, 100, 50);
```

This will create a rectangle that starts at coordinates (20, 30) with a width of 100 pixels and a height of 50 pixels. In fact, this will draw the previously shown rectangle but shifted 20 pixels to the right and 30 pixels down. So, the signature for this method is to give the x, y coordinates for the starting point of the rectangle followed by the width and height. In addition to the `strokeRect()` function, there are `fillRect` and `clearRect`. When drawing on canvas, one can draw a stroke which is a line, draw a filled structure, or clear the content of an area.

If rectangles aren't your style, perhaps you would be more interested in drawing a circle? Canvas actually considers a circle to be a special case of an arc. Thus, to draw a circle with a radius of 50px, you need to specify not just the center point and radius, but also the starting and ending angle. The code for such a circle is as follows:

```
context.arc(75, 75, 50, 0, 2*Math.PI);context.fill();
```

Here, the center point is given as (75, 75) and the radius 50. The fourth term is the starting angle which we give as 0 and the `2*Math.PI` is the ending angle—the whole way round the circle. An optional final argument determines if the arc is to be drawn anti-clockwise. It defaults to `false` or `clockwise`.

Alerting the final parameter gives different arcs, as shown in the following screenshot:

As you can see, all of the angles used in canvas are denoted in radians. To convert from an angle in degrees to one in radians, you can multiply it by `Pi/180`.

For more complex shapes, canvas supports straight line or a path. To draw a line, you set a starting coordinate, then give an ending point, and then call the `stroke()` function. Canvas will extrapolate where to draw the line, as shown in the following code snippet:

```
context.beginPath();
context.moveTo(0,0);
context.lineTo(50,120);
context.closePath();
```

```
context.stroke();
```

Here we start a new path; the pen is moved to (0, 0), then a line is drawn to (50, 120), and then the path is ended. It is important to begin and end your paths, or subsequent calls to `stroke()` functions will result in a continuation from the last point. You can think of it like using a pen; `beginPath` puts the pen on the piece of paper, `moveTo` temporarily lifts the pen and moves it, `lineTo` moves the pen to the destination and draws a line behind it, and finally `closePath` picks the pen back up off the paper. Without picking the pen up, the next time you draw a line to somewhere, the pen will already be on the paper and you'll get an extra line.

If you think the syntax for drawing a line is a bit arcane, you're not alone. The multiple calls enable you to build more complex lines with several segments. By using a loop, we can build relatively complex shapes, as shown in the following code snippet:

```
var width = 200;
var height = 200;
context.moveTo(0, 0);
for(i = 0; i> 100; I += 4)
{
   context.lineTo(i, height-i);
   context.lineTo(width - 1-i, height - 1 -i);
   context.lineTo(width - 2-i, i+2);
   context.lineTo(i+3, i+3);
}
context.stroke();
```

This code will generate a spiral. On each iteration of the loop, we move towards the center of the image by 4 pixels, one pixel on each edge. The result is as shown in the following screenshot:

The canvas maintains a state between calls to it, so, if you set the color of the fill using `context.fillStyle`, all subsequent calls to fill a shape will take on the same fill style. As you can probably imagine, this is a common source of bugs when building a visualization using canvas. It is especially problematic when calling functions to operate on the canvas.

Fortunately, there is an easy solution: the context state can be saved and restored when entering and

exiting a function. It is polite to keep your functions from messing around with a global state and will certainly reduce the number of bugs:

```
function fillCircle(context, x, y, radius){
   context.save();
   context.fillStyle = "orange";
   context.beginPath();
   context.arc(x, y, radius, 0, 2* Math.PI);
   context.fill();
   context.restore();
}
```

In the first line of the function, the current drawing context is pushed onto a stack and then it is restored on the last line of the function. By doing this, we can make as many changes to the context within the method as we like and rest assured that the context of the caller will not be corrupted.

Canvas has support for a full palette of colors and also allows for transparency and even gradients. So far, the examples have used color names. These names date way back to the mid 1990s and actually hail from the X11 windowing system. However, there are two other ways to specify a color for canvas. The first is by using a hexadecimal string that specifies the values for red, green, and blue as two-digit hexadecimals digits. The larger the value, the higher is the intensity of that color, as shown in the following figure:

The final, and my preference, is to use decimal RGB values, as shown in the following figure, which I think is far more readable and also easier to build programmatically:

A slight variant on the decimal format is to use the `rgba` function instead of `rgb`. This adds an extra parameter, which is a decimal number between 0 and 1, which denotes the opacity. 1 is fully opaque

and 0 is completely transparent.

As canvas is a raster-based drawing system, it is possible to include most other raster files in it. Raster file formats include JPEG, PNG, GIF, and BMP. Being able to import an existing image can be a very handy tool for visualizations.

However, there are some caveats to using images on canvas. When you're including an image, you can't just load it from a URL directly. First, a JavaScript Image object needs to be created and the source of that image needs to be set to the URL. This Image object can then be used on the canvas:

```
Var image = new Image();
image.src = "logo.png";
```

Requesting images from another domain can be tricky. In order to maintain security in the browser requesting data, other domains are restricted. For images, you can request permission from the hosting domain to use the image by setting the `crossOrigin` property on the image:

```
Var image = new Image();
image.crossOrigin = "anonymous";
image.src =
"http://codinghorror.typepad.com/.a/6a0120a85dcdae970b0128776ff992970c-pi";
```

Here, the value of the `crossOrigin` policy has been set to `anonymous`. This means that the browser won't pass on any authentication information to the server hosting the image. You can also set a value of user-credentials if you do want to pass credential information. Support for `crossOrigin` on images is relatively new, so you may do better to host the images on the same domain as the canvas.

Because loading an image can take some time, it is not advisable to set the value of `src` on an image and immediately attempt to draw it on the canvas. Instead, you should hook into the `onload` event on the image:

```
function drawImage()
{
   var img = new Image();
   img.src = 'worksonmymachine.png';
   img.onload = function(){
      var context = document.getElementById('example').getContext('2d');
      context.drawImage(img, 0, 0);
   }
}
```

Using the `onload` function will prevent an empty image from being rendered to the canvas. If you're loading multiple images, the order in which they are loaded is indeterminate. You may wish to check that all images are loaded before continuing. Complex dependency chains can be managed using jQuery's `Deferred` functionality. The second and third parameter to `drawImage` is, predictably, the coordinates at which to draw the image. There are more advanced versions of `drawImage` that allow for scaling and cropping of an image before drawing it to the canvas.

The final feature of canvas which we'll look at is the transformation. When composing more complex scenes or visualizations, it is frequently easier to build the object at a different scale, location, or orientation. Transformations provide a mechanism for altering the shapes you draw on canvas.

A function, in this context the scaling function, will multiply every coordinate by the x or y scaling factor. Canvas provides for independently scaling the x and y values. This means that it is easy to stretch a shape in just one direction:

```
var colours = ["rgba(255,0,0,.5)", "rgba(0,255,0,.5)", "rgba(0,0,255,.5)"];
for(var i = 0; i< 3; i++)
{
    context.fillStyle = colours[i];
    context.scale(i+1,i+1);
    context.fillRect(10, 10, 50, 50);
}
```

The preceding code, which simply draw a series of three rectangles, will produce an output that looks like the following screenshot:

You'll note that our scaling didn't just create larger images, it also multiplied the coordinates of the rectangles. The other thing to note is that the size increase doesn't seem to be uniform. That's because canvas is stateful. If you apply several scale functions in a series without resetting the scaling, each one will build on the last scaling. You can keep a track of your transformations and apply a reversing function. For instance, if you have scaled by 2, you can scale by the multiplicative inverse of 2, which is 1/2, to get back to the original scaling. It is more likely easier to save the context and restore it using the `save` and `restore` functions we spoke of earlier.

If we modify our function, you can see that the resulting image is very different, as shown in the following screenshot:

Frequently, when applying a scaling transformation, you'll want to apply a translation transformation to move the starting coordinates to where you expect them to be. You can do this by employing a translate transformation:

```
var x = y =10;
var width = height = 100;
for(var i = 2; i><= 0; i--)
{
   var scalingFactor = i+1;
   context.save();
   context.fillStyle = colours[i];
   context.translate(x * -i, y *-i);
   context.scale(scalingFactor, scalingFactor);
   context.fillRect(x, y, width, height);
   context.restore();
}
```

On the highlighted line, we shift the canvas over so it is as if the rectangle is drawn at the origin. The order of the application of transformations is important, as shown in the following screenshot:

On the left-hand side, you can see the output of our code and on the right-hand side, the result of

swapping the `translate` and `scale` operations.

There are a lot of other great features of canvas. Going into everything is just too much for a book of this length; it is most likely a book in its own right.

Scalable Vector Graphics

Scalable Vector Graphics, or as they are more commonly known, SVGs, are another feature relatively new to HTML. They fulfill a similar role to canvas, but differ by the fact that they are vector-based instead of raster-based. This means that every image is made up of a series of basic shapes. This might sound a bit like canvas, after all, we created all our canvas images using basic shapes. The difference is that with SVG, the basic shapes remain as distinct objects after they've been drawn. With canvas, the source commands used to create the image are lost as soon as it is rendered. The information about the source of the pixel is lost in a jumble of canvas commands.

Conveniently, SVGs are stored as XML files. While I would typically not even consider the idea of storing anything but the combination to my safety deposit box, in such an inaccessible file format it does integrate nicely with HTML. Raster images are typically linked in a separate file from the HTML. SVGs can be embedded directly into the HTML document. This technique can be used to reduce the number of server trips necessary to render a page on a user agent. However, the real advantage is that it allows for the SVG to be integrated into the HTML **Document Object Model** (**DOM**), allowing you to manipulate the SVG using the same techniques you might use to manipulate any other element.

The source for a simple SVG may look like the following:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect width="50" height="150" x="20" y="20" stroke="black"stroke-width="2"
fill="#a3a3a3" />
</svg>
```

You can paste this into any HTML document and you'll get a simple rectangle that looks like the following screenshot:

The code is quite easy to understand, and the syntax should be familiar to anybody who has built a website. We first open new SVG element. Without any explicit sizing information the SVG fills its container. Inside of the SVG, we create a new rectangle with a width of 50px and a height of 150px. The outline of the rectangle is black and has a width of 2px while the inside is filled with a grey color.

The primitives you can use to build your images should also be somewhat familiar, now that you've seen canvas in action. `rect` and `path` remain unchanged from canvas. However, SVG differs in its treatment of circles and provides an actual `<circle>` tag as well as an `<elipse>` tag for round shapes with two foci. `<polygon>` and `<polyline>` tags provide for drawing free-form straight-edged shapes with the polygon being a filled shape and the polyline being just a line. Should you desire a more curvy shape, SVG provides a `path` element that allows for defining complex curves and arcs. It is very tricky to build a curved path by hand. Typically, for curved paths you'll want to make use of an editor or an SVG library. Finally, SVG has support for writing text using the aptly named `<text>` element.

Building multiple elements in SVG is as simple as adding another child to the SVG , as shown in the following code snippet:

```
>svg xmlns="http://www.w3.org/2000/svg" version="1.1"<
  >rect width="50" height="150" x="20" y="20" stroke="black" stroke-width="2"
fill="#a3a3a3" /<
  >rect width="50" height="120" x="75" y="50" stroke="black" stroke-width="2"
fill="#a3a3a3"/<
  >rect width="50" height="90" x="130" y="80" stroke="black" stroke-width="2"
fill="#a3a3a3"/<
  >rect width="50" height="60" x="185" y="110" stroke="black" stroke-width="2"
fill="#a3a3a3"/<
>/svg<
```

This results in an image that looks like the following:

As you can see, the code is quite repetitive. On every single rectangle, we specify the `stroke` and `fill` information. This repetition can be eliminated in two ways. The first is to use a group of elements to define the styling information. SVG provides a generic grouping container which is denoted by the `<g>` tag. The styling information can be applied to that container instead of the individual elements:

```
>g stroke="black" stroke-width="2" fill="#a3a3a3"<
  >rect width="50" height="150" x="20" y="20"  /<
  >rect width="50" height="120" x="75" y="50" /<
  >rect width="50" height="90" x="130" y="80" /<
  >rect width="50" height="60" x="185" y="110" /<
```

```
>/g<
```

An alternative is to use CSS to do the styling for you:

```
>style<
  rect {
    fill: #f3f3f3;
    stroke: black;
    stroke-width: 2;
  }
>/style<
>svg xmlns="http://www.w3.org/2000/svg" version="1.1" id="graph1"<
  >rect width="50" height="150" x="20" y="20" /<
  >rect width="50" height="120" x="75" y="50" /<
  >rect width="50" height="90" x="130" y="80" /<
  >rect width="50" height="60" x="185" y="110" /<
>/svg<
```

In the preceding code, the styling information is attached directly to all elements of type `rect`. Typically, you would want to avoid using broad selectors such as these, as they will apply to all the SVG elements on the page. It is better to narrow the selectors either by making them apply to just that one SVG or, preferably, by assigning a class to the SVG elements you wish to style. It is generally preferred to style your elements, even those that are part of an SVG, using CSS. It is likely that your SVG will contain multiple rectangles that you don't want to take on the same styling.

The style properties (`fill, stroke`, and so on) used in the CSS do not differ from those used in in the SVG markup. More advanced CSS selectors are also available, such as `nth-child`, which selects just children matching a specific pattern. Consider the following code snippet:

```
rect:nth-child(even)
{
   fill:#878787;
}
```

Adding the preceding code in our example will, very simply, create a zebra striping effect on our graph, as shown in the following screenshot:

We can even get fancy and use CSS to add some interaction to our graph by simply specifying a

`:hover` pseudo selector in the CSS and changing the color under the cursor:

```
rect:hover
{
   fill: rbg(87,152,176);
}
```

The following screenshot shows the resulting graph:

Of course, having the SVG as part of the DOM opens other doors to factors other than styling. It is also possible to manipulate the elements of an SVG using JavaScript. You can even assign event listeners to the SVG elements.

By making use of the fantastic jQuery library, we can easily add event listeners to the nodes in the graph SVG we've been building so far:

```
$("rect").click(function(){
    alert($(this).attr("height"));
});
```

If you've never seen jQuery before, what is happening here is that we're selecting all the `rect` elements on the page, and when a click is fired, it opens an alert box with the height of the column on which we clicked.

Throughout this book, we'll be making extensive use of the jQuery library and this style of lambda-based programming. If you're not familiar with jQuery, it would be advisable to take a break and read some tutorials, such as http://try.jquery.com/.

We've covered all the basic functionality of SVGs, but there is one advanced feature I'd like to mention: filters. **Filters** are transformations that can be applied to elements of an SVG. These filters go beyond the scaling and translation transformations we saw in canvas, although both `scale` and `translate` are supported in SVG. There are about 20 of these filters and each one performs different transformations. We won't be able to go into each, but we'll look at a couple of them.

One of the most common requirements in visualization is to give things a 3D feel. Full 3D can be very difficult, but we can trick the eye by using shadows. These shadows can be created using a

combination of three different filters: offset, Gaussian blur, and blend.

To use a filter, we start by defining it. A `filter` element can be defined as a number of sequentially applied filters. To figure out what filters we need for a shadow, we can work backwards from the properties of a shadow. The first thing to notice is that shadows are offset from the things casting them. For this, we can use an offset filter that will shift the element in one direction or another. Where you want to shift the element depends on where your light source is. For our purposes, let's say that it is above and to the left of the SVG. This will cast shadows down and to the right:

```
>defs<
    >filter id="shadowFilter" width="175%" height="175%"<
     >feOffset result="offsetImage" in="SourceAlpha" dx="5" dy="5"/<
    >/filter<
>/defs<
```

On the filter line, we need to specify a height for our filtered element, which is greater than the original. If we don't do so, much of our shadow will be cut off as it extends beyond the boundary of the source object. Here we have given our filter an ID so it is easily applied later. You'll also note that we've specified an `in` and `out` property for `feOffset`. This allows us to chain the filters together. In our case, we're taking `SourceAlpha`, which is just the alpha, or the `transparency` property from the original image.

Let's apply this filter to just one element of our graph so we can see what is happening to it. I've removed the other styling so as not to confuse matters. The filter is applied by using the `filter` attribute and giving it the ID of the filter created previously:

```
<rect width="50" height="60" x="185" y="110" filter="url(#shadowFilter)"/>
```

The following will be the result:

Shadows are also fuzzier than the original image. This can be achieved by using a Gaussian blur filter:

```
<feGaussianBlur result="blurredOffset" in="offsetImage" stdDeviation="8" />
```

Gaussian filters randomly move points inside your image based on a normal distribution function. You

might wish to play around with the standard deviation to achieve different blur effects; I found somewhere in the 8-12 range to be good for shadows:

```
<feBlend in="SourceGraphic" in2="blurOut" mode="normal" />
```

Finally, we want to take the blurred black box we've created and combine it with the original:

```
<feBlend in="SourceGraphic" in2="blurredOffset" mode="normal" />
```

Applying this filter on hover gives a very convincing pop-up effect when users hover over the image, as shown in the following screenshot:

SVG provides for easy manipulation of parts of an image though the styling tools that are already well known to you from your work with CSS. At the same time, being able to attach events to the image allows the creation of impressive user interaction.

Which one to use?

Deciding whether to use canvas or SVG can be a difficult problem. It mostly comes down to which one feels more comfortable. Those with a background in computer graphics or animation are more likely going to be happier with canvas, as the `redraw` loop of canvas will be familiar. Canvas is better suited to redrawing entire scenes or even if you plan on using 3D elements. If your visualization makes use of textures or rendered images, canvas' ability to draw them to the canvas directly is almost certainly going to be advantageous. For visualizations that have some reliance on maintaining a fast frame rate, canvas is generally high performing.

On the other hand, SVG can be a much simpler technology to use. Each element in an SVG can be individually manipulated, which makes small animations far easier. The integration with the DOM allows for events to be fired on interaction with a single element of the SVG. To achieve that in the canvas, you must manually track what is being drawn at that location. That SVG can also be styled using CSS, which allows for components to be more easily reusable on sites with different themes.

For the purposes of this book, we're going to focus on SVGs. The resolution independence of SVG coupled with the ease of use and fantastic support libraries makes it a logical choice. I don't believe there is a visualization we'll be creating that cannot be created with canvas but the effort would be far greater. This is doubly so for the interactive visualizations.

Summary

You should now be able to make an informed decision between SVG and canvas-build simple static images. We're going to take a break from the visual aspects in the next chapter and talk about the OAuth protocol, which is used by many social media sites to protect their data.

Chapter 3. OAuth

Creating visualizations is really only half the battle; the other half is getting high quality data to drive the visualization. There are a lot of potential data sources that you may wish to exploit. Almost every country has a national statistics organization tasked with gathering and analyzing economic and social statistics. In the past few years, many governments have begun adopting Open Data initiatives. Many businesses are also centered on providing usable data; think about the amount of data provided by various stock exchanges. You may even have access to internal company data you're using to drive your visualization, or your visualization may be part of a larger application that will provide you with data.

Another source, and the source we're interested in for this book, is social media. Social media sites have a plethora of data available to their users. The vast majority of social media sites provide APIs for accessing data in a programmatic fashion. Frequently, this data is personalized for your user account. For instance, Twitter filters the tweets you see by the users you're following, and Facebook similarly shows you updates from your friends. Some data is restricted, such that only certain groups of people can see it. You may not want the whole world seeing your Facebook updates, so you set permissions to only allow friends to see it. In order to personalize the data, most social networking sites require authentication and authorization for their APIs.

Typically, the data you want to show users is related to their own social media account. In most cases, your visualization will not be hosted on the same system as the data you're looking to use, as shown in the following figure. What's more important is that the consumer of your visualization, the end user, is likely to be on yet another system:

This means that there needs to be some way to get the social media data from the social media site to your visualization, and then on to the end user. On the surface, there seems to be the following two options:

- Ask the user for their social media password and use that to authenticate and check authentication against the social media API
- Ask the user to retrieve the data required for the visualization and send that on to your

visualization

Neither option is particularly desirable. Users are likely to be resistant to giving out their passwords, especially to something as important as their social media sites. You probably don't want to take ownership of their password either, as that is an extra security headache for you. Most users are insufficiently tech-savvy to extract the data needed from their social media sites and send it onto your application. Even those who have the technical skills are not going to be impressed with the amount of work that will entail. Certainly, standard export mechanisms could be established and refined to make it easier for users, but before we develop a system for this, perhaps is there a third way?

The trick seems to be in finding a way to get credential information from the end user to the social media site without telling the intervening visualization site too many details. As with many computer problems, a parallel problem and solution may be found outside of the world of computing.

Many electronic garage door openers and home security systems offer guest codes. These codes may have restrictions on them, such as they only operate on certain days of the week or a certain number of times. The purpose of these accounts is to provide limited access to your house for, say, cleaning people or trade people. A similar concept is purported to exist in high-end automobiles: a valet key causes the car to operate in a restricted mode. A similar mechanism could be used to grant your visualization access to a restricted portion of the social media site.

The OAuth protocol provides a mechanism for authenticating applications to make use of your social media data and functionality without needing to know your password.

In this chapter we're going to look at, on a high level, how OAuth works and then try authenticating with a few social networks using our existing credentials.

Authentication versus authorization

Frequently, there is confusion between the concepts of authentication and authorization. **Authentication** is the act of ensuring that somebody is who they say they are, while **authorization** is the act of ensuring that the person has the rights to perform an action. The concepts are related and have frequently been part of the same step. OAuth breaks the relationship between the two. Although there is typically an authentication step where logging into the server is required, the way that authentication is performed is not prescribed. If the user is already logged into the server site, the authorization step may be transparent to the user. The server can use any method it wishes to perform authentication. This opens the door to allowing for multi-factor authentication or even additional delegation of authentication. For instance, your visualization could request information from Stack Overflow that uses OpenID to delegate authentication. Thus, when a user requests access to Stack Overflow data, the user might actually need to log in with their Google account that would pass authentication details to Stack Overflow, which in turn would pass authorization credentials to your visualization.

It is important to keep in mind the differences between authentication and authorization when acquiring data for your visualization.

The OAuth protocol

Before the introduction and extensive update of OAuth, each service with which you needed to interact provided their own authorization protocol. These methods diverged significantly. Every time you wanted to make use of a new API, a new authorization scheme had to be learned and implemented. This made interacting with a significant number of services very difficult.

OAuth was created to solve the problem of a lack of standardization around authorization with different sites. The creation of Version 1.0 took about two years and was contributed to by a number of major industry players, as well as smaller interested parties.

OAuth versions

There are currently two major versions of OAuth in the wild: 1.0a and 2.0. Version 1.0a is a security update to the 1.0 specification which corrected a session fixation attack. The 2.0 specification diverged significantly from 1.0a and, unfortunately, there remains a mixture of services that use different protocols in the wild. There is some political debate about the security of the 2.0 specification, which has resulted in a number of companies remaining on the 1.0a version. The biggest difference is that OAuth 1.0 was designed to be usable over an unencrypted connection. When OAuth 1.0 was created, SSL certificates were prohibitively expensive for many small providers. Thus, a complicated series of signed requests were specified which allows even unsecure connections to act as if they were secure.

Now SSL certificates have become cheaper, and hardware fast enough, that acquiring an SSL certificate is well within the means of even the smallest startup. Thus, Version 2.0 of OAuth relies on SSL to provide much of its security. It is not safe to use OAuth 2.0 over an unencrypted link. For our purposes, we can remain largely ignorant of the other differences between OAuth versions. However, you should be aware that the authorization and authentication tools you write for one social media site may not work on others. It is even possible that due to implementation differences, your authorization methods will not work between two different sites that both fully support OAuth 2.0.

OAuth defines some different names for the players we saw earlier in the chapter, so let's make use of these terms. An updated view of the previous figure looks like the following:

Resource Owner is the person who "owns" the data. For example, you "own" your updates on Facebook. **Client** is the site requesting access to data and **Server** is the one that has both the data and the credential information for the resource owner. The data that is owned is typically called the **protected resource,** as it is what the OAuth layer is protecting. I've left the icons in place from earlier for clarity, but the client doesn't have to be a visualization, neither does the server have to be a social media site.

You've likely already made use of the OAuth protocol, even if you didn't know it. If you've ever granted an application permission to use your Twitter account, you've used OAuth. The official

Twitter applications such as **TweetDeck,** make use of OAuth for authorization. Each application requests access to specific functionality from Twitter. If you're a Twitter user, you can see the applications you've authorized to use your Twitter account from the **Settings** panel. Each one of these applications has been granted permission to interact with Twitter as if it was you, as shown in the following screenshot:

In the preceding screenshot, you'll notice that there is a button next to each application that allows revoking that application's access. This is one of the great features of OAuth—applications never know your password, so you can remove their ability to act as you without having to change your password. If you do wish to change your Twitter password, you only need to do it with Twitter and not with all of the services to which you have granted permission to act on your behalf.

If Twitter were to become aware that, say, www.wordpress.com had been compromised, they could revoke the application's access for all their users at once. Keeping credentials is a difficult problem and not one which many developers wish to take on. If credentials can be retained by a reliable company such as Twitter, that removes a common point of failure.

Let's dig a bit more into how OAuth actually works. In order to understand what's happening, it is useful to walk through an example. In this example, our visualization will request some information from Facebook using its Graph API. The **Graph** API is the interface that Facebook have provided developers with to access the social graph, which is really just a collection of properties about a user. Facebook is an OAuth 2.0 site, so this example will use the workflow as described in OAuth 2.0.

Our visualization would like to access information from Facebook. The first time we load the visualization, we're presented with a screen that allows users to click on a button to get access to the Facebook information, as shown in the following screenshot:

When we first set up our visualization site, we will have requested an authentication token from Facebook. This token is granted by Facebook to our site alone. As part of registering the application with Facebook, we will have entered a domain from which the token can be used. This grants some security over preventing others from using our tokens. Servers may have extensive checks in play to qualify an application to access their protected data.

Our site will generate a request to the Facebook OAuth endpoint which would include the generated token. A typical URL will look like the following:

```
https://www.facebook.com/dialog/oauth?client_id=
591498037422486&redirect_uri=http://visualization.com/&response_type=token
```

Tip

A more detailed example of authenticating against Facebook can be found in Chapter 7, *Facebook*.

The visualization now redirects directly to the Facebook login page, which will ask for your login information, as shown in the following screenshot:

Once you've entered that information and correctly authenticated with Facebook, you'll be redirected back to the page you specified in the `redirect_uri` parameter. This will typically be your original page. Appended to the URI will be a very lengthy token generated by Facebook and used in subsequent communications with their API:


```
http://localhost:8080/Visualization1.html#access_d8Ava9mMBALMG0FQp22SEn5La7mtC27
evICrZB5ToVHdJRZC2FYdFmnIsveVKhcikSeVZAEAZAHBliKEeGvrKHnb5FnU5VoCooy49FIoJuzca3o
HeYuNUZAatdgjUEr2tDzZBB8CJGmPkHdmNe3RyS1l9XAcTKwGGVAy6FB0gZDZD&expires_in=6667
```


Depending on the permissions your visualization has requested, Facebook may prompt you to grant these permissions explicitly to the visualization. The granting of permissions to applications is they key function of OAuth. Facebook has about 50 different permissions that your application can request. This includes the following:

- `email`
- `user_likes`
- `user_location`

The prompt to grant the application access to your Facebook information looks like the following screenshot:

In this example, **Example Visualization** has requested access to my location, which is protected by the `user_location` permission. In Facebook's case, the granted permissions are encoded in the token to be used, but this is not explicitly laid out in the OAuth protocol.

Once permissions are assigned, Facebook will redirect the user back to your `redirect_uri`, allowing you to leverage the token to query the Facebook API, as shown in the following screenshot:

For some OAuth sites, it is recommended that an additional call be performed at this stage to ensure that the returned token matches the current application. By sending our application's ID and the returned token from the authentication step, we are provided with some validation details that can be used to confirm the login has not been hijacked. Not every site requires this step; it is simply an added security suggested by Facebook. With the credentials, we can make calls to the Graph API. In this case, the visualization makes a very simple request to retrieve the authenticated user's first and last name. You can see how the request is formulated in the Chapter 7, *Facebook*.

That's it! So OAuth delegates the authorization and authentication steps to a third party without the need for complex tools. OAuth uses nothing but normal HTTP and SSL. The workflow for OAuth is almost entirely represented in the following figure:

The variety of OAuth dialects and requirements can be difficult to understand. Finding just the right combination of tokens to receive proper authentication with different sites is almost as difficult as the mishmash of techniques used prior to OAuth. OAuth has a reputation for being difficult to understand and inconsistent . This can be attributed to the OAuth standard not being a standard in the same way that HTTP is a standard. With HTTP, if you comply with the standard, you can be assured that your service will be able to interact with all others. OAuth does not offer that same level of interactivity guarantee.

If you're going to be using a lot of different data sources, perhaps if you're creating a mashup, it may be advisable to use a third-party service to communicate with the OAuth servers. Even if you're only using a single data source, you may not wish to complicate your development process with figuring out OAuth. Companies such as DailyCred and OAuth.io provide a service that abstracts away the difficulties of dealing with OAuth. They allow for authentication with numerous different OAuth providers through a consistent API. The hard work of fitting into the various OAuth APIs is handled by them leaving you free to concentrate on building your visualization.

Authenticating against Facebook with OAuth.io is as simple as running the following lines of code:

```
OAuth.initialize('<Your Public key>');
OAuth.redirect('facebook', "callback/url");
```

The example visualization used in this chapter is closer to 70 lines of code.

Of course, these services have a monetary cost to them and also provide an additional failure point. As with all things, care must be taken to ensure that a good solution for OAuth is selected.

Summary

While not every social media site makes use of OAuth, knowledge of how OAuth works and how it can be used to facilitate API usage will most likely improve your experience in developing visualizations. You should now be able to explain how OAuth works. In the next chapter we'll take a look at libraries for visualization.

Chapter 4. JavaScript for Visualization

In Chapter 2, *JavaScript and HTML5 for Visualizations*, we looked at the advantages offered by building our visualizations using scalable vector graphics. It should, however, have been clear that building SVGs by manipulating the underlying XML is a frustrating and time consuming exercise. Although there are countless XML manipulation tools, it would be nice to take advantage of the power of an API which is specifically designed for building SVGs instead of a more general language.

There are a number of JavaScript libraries that have been created for the manipulation of SVGs. svg.js (http://www.svgjs.com/) and Raphaël (http://raphaeljs.com/), both deserve mentions as being excellent tools for drawing. The demos on the Raphaël website are particularly impressive. d3.js offers functionality designed specifically for visualizations and we'll look at that too.

Raphaël

To draw a simple rectangle making use of **Raphaël** is much more comfortable than building the same rectangle in XML. The library can be included either from the disc or from a CDN such as CloudFlare, as shown in the following code:

```
<html>
  <body>…</body>
  <script src="//cdnjs.cloudflare.com/ajax/libs/raphael/2.1.0/raphael-min.js">
</script>
</html>
```

We can draw any shape in the following manner:

```
function drawRectangle()
{
  var paper = Raphael("visualization", 320, 200);
  paper.rect(50, 20, 50, 150);
}
```

This code finds the element with an ID of `visualization` and appends an SVG to it with a dimension of 320 x 200 pixels. It then inserts a new rectangle at `(50, 20)`, with width `50` and height `150`.

If we wanted to create a simple column graph with Raphaël, it would not be difficult. Let's give it a shot. The first thing we'll need is some data. We can use a JavaScript array for now, but in the real world this information could be retrieved from a web service in either JSON or XML form. In this example, we'll pick some months and the associated value, as shown in the following code:

```
var data = [{month: "May", value: 5},
  {month: "June", value: 4},
…
  {month: "September", value: 8}];
```

Now, let's update the function we used above to draw rectangles to deal with data values. First, we'll change the method signature, as shown in the following code:

```
function drawGraphColumn(paper, item, currentColumn, maximumValue)
{…}
```

This function takes in SVG on which to draw, the current item, and a maximum value that is used to calculate the appropriate height. In the body of the function, we'll start by calculating the bar height, as shown in the following code:

```
var barHeight = (500 * (item.value/maximumValue));
```

We've hard coded the maximum height at 500 px, and each bar is simply a percentage of that height equal to the item's value as a percentage of the maximum. We'll use that value to draw the bar, as

shown in the following code:

```
var rectangle = paper.rect(currentColumn*30, 500 - barHeight, 20, barHeight);
rectangle.attr("fill", "rgb("+ (item.value * 40) + " ," + (item.value * 20) +
"," + item.value * 20 + ")");
rectangle.attr("stroke-width", 2);
```

The rectangle is offset based on which column it is avoiding overlapping rectangles. We set the color to be a function of the item value so that the color varies by height.

The function is called by passing each one of our data elements into it sequentially:

```
var maximumValue = 0;
$.each(data, function(index, item)
{
  if(item.value > maximumValue) maximumValue = item.value;
});
$.each(data, function(index, item){
  drawGraphColumn(paper, item, index, maximumValue);
});
```

Here, we first calculate the maximum value from the array. Then, call the `drawGraphColumn()` function we defined above for each element. Looping over the data array is done using jQuery's `each` operator which applies the given function to each element in the array. The resulting graph looks like the following:

Raphaël is a general purpose SVG library. This means that apart from being suited to building visualizations, it can be used to create more generic drawings. In the same way that we were looking for an API for manipulating SVGs, which was better suited than manipulating XML by hand, it would be nice to have a library which is targeted at building visualizations. d3.js is a library which is specifically designed to make building visualizations using SVG easier.

d3.js

d3.js brings a number of functions and a coding style that makes creating even simple graphs like the one above simpler. Let's recreate the above graph using d3 and see how it differs. The first thing to do is introduce an SVG element to the page. With Raphaël, we did this using the constructor; in d3, we'll append an SVG element explicitly, as shown in the following code:

```
var graph = d3.select("#visualization")
  .append("svg")
  .attr("width", 500)
  .attr("height", 500);
```

Immediately, you'll see that the style of the JavaScript used differs greatly from Raphaël. d3 relies heavily on the use of method chaining. If you're new to this concept, it is quick to pick up. Each call to a method performs some action, then returns an object, and the next method call operates on this object. So, in this case, the `select` method returns the `div` with the `ID` of `visualization`. Calling `append` on the selected `div` adds an SVG element and then returns it. Finally, the `attr` methods set a property inside the object and then return the object.

At first, method chaining may seem odd, but as we move on you'll see that it is a very powerful technique, and cleans up the code considerably. Without method chaining, we end up with a lot of temporary variables.

Next, we need to find the maximum element in the data array. In the previous example, we used a jQuery `each` loop to find that element. d3 has built in array functions which make this much cleaner, as shown in the following code:

```
var maximumValue = d3.max(data, function(item){ return item.value;});
```

There are similar functions for finding minimums and means. None of the functions are anything you couldn't get by using a JavaScript utility library, such as underscore.js or lodash. However, it is convenient to make use of the built-in versions.

The next pieces we make use of are d3's scaling functions, as shown in the following code:

```
var yScale = d3.scale.linear()
  .domain([maximumValue, 0])
  .range([0, 300]);
```

Scaling functions serve to map from one dataset to another. In our case, we're mapping from the values in our data array to the coordinates in our SVG. We use two different scales here: linear and ordinal. The linear scale is used to map a continuous domain to a continuous range. The mapping will be done linearly, so if our domain contained values between `0` and `10`, and our range value ranges between `0` and `100`, then a value of `6` would map to `60`, `3` to `30`, and so forth. It seems trivial, but with more complicated domains and ranges, scales are very helpful. As well as linear scales, there are power and logarithmic scales which may fit your data better.

In our example data, our `y` values are not continuous; they're not even numeric. For this case, we can make use of an ordinal scale, as shown in the following code:

```
var xScale = d3.scale.ordinal()
  .domain(data.map(function(item){return item.month;}))
  .rangeBands([0,500], .1);
```

Ordinal scales map a discrete domain into a continuous range. Here, `domain` is the list of months and the range the width of our SVG. You'll note that instead of using `range` we use `rangeBands`. Range bands split the range up into chunks, each to which each range item is assigned. So, if our domain was `{May, June}` and the range `0` to `100`, then May onwards we would receive a band from `0` to `49`, and June from `50` to `100` would be `june`. You'll also note that `rangeBands` takes an additional parameter; in our case `0.1`. This is a padding value that generates a sort of no man's land between each band. This is ideal for creating a bar or column graph, as we may not want the columns touching. The padding parameter can take a value between `0` and `1` as a decimal representation of how much of the range should be reserved for padding. A value of `0.25` would reserve 25 percent of the range for padding.

There are also a family of built-in scales that deal with providing colors. Selecting colors for your visualization can be challenging, as the colors have to be far enough apart to be discernible. If you're color challenged like me, then the scales `category10`, `category20`, `category20b`, and `category20c` may be for you. You can declare a color scale, as shown in the following code:

```
var colorScale = d3.scale.category10()
  .domain(data.map(function(item){return item.month;}));
```

The previous code will assign a different color to each month out of a set of 10 pre-calculated possible colors.

Finally, we need to actually draw our graph, as shown in the following code:

```
var graphData = graph.selectAll(".bar")
  .data(data);
```

We select all of the `.bar` elements inside the graph using `selectAll`. Hang on! There aren't any elements inside the graph that match the `.bar` selector. Typically, `selectAll` will return a collection of elements matching the selector just as the `$` function does in jQuery. In this case, we're using `selectAll` as a short hand method of creating an empty d3 collection which has a `data` method and can be chained.

We next specify a set of data to union with the data from the existing selection of elements. d3 operates on collection objects without using looping techniques. This allows for a more declarative style of programming, but can be difficult to grasp immediately. In effect, we're creating a union of two sets of data, the currently existing data (found using `selectAll`), and the new data (provided by the `data` function). This method of dealing with data allows for easy updates to the data elements, should further elements be added or removed later.

When new data elements are added you can select just those elements by using `enter()`. This prevents repeating actions on existing data. You don't need to redraw the entire image, just the portions that have been updated with new data. Equally, if there are elements in the new dataset which didn't appear in the old one, then they can be selected with `exit()`. Typically, you want to just remove those elements which can be done by the following code:

```
graphData.exit().remove()
```

When we create elements using the newly generated dataset, the data elements will actually be attached to the newly created DOM elements. Creating the elements involves calling `append`, as shown in the following code:

```
graphData.enter()
  .append("rect")
  .attr("x", function(item){ return xScale(item.month);})
  .attr("y", function(item){ return yScale(item.value);})
  .attr("height", function(item){ return 300 - yScale(item.value);})
  .attr("width", xScale.rangeBand())
  .attr("fill", function(item, index){return colorScale(item.month);});
```

The following diagram shows how `data()` works with new and existing dataset:

You can see in the previous code how useful method chaining has become. It makes the code much shorter and more readable than assigning a series of temporary variables, or passing the results into

standalone methods. The scales also come into their own here. The x coordinate is found simply by scaling the month we have using the ordinal scale. Because that scale takes into account the number of elements as well as the padding, nothing more complicated is needed.

They coordinate is similarly found using previously defined `yScale`. Because the origin in an SVG is in the top left, we have to take the inverse of the scale to calculate the height. Again this is a place where we wouldn't generally be using a constant except for the brevity of our example. The width of the column is found by asking the `xScale` for the width of the bands. Finally, we set the color based on the color scale so it appears like the following graph:

This d3 version of the graph is actually far more capable than the Raphaël version. We eliminated a lot of the magic numbers that were present in the Raphaël through the use of scales.

Let's continue to enhance our graph and explore some of the other features of d3.js.

Custom color scales

The d3 graph we generated above is very colorful, since we made made use of one of the built in color scales. However, most of the time you must have least some thematic consistency in visualizations. You can achieve this consistency across visualizations by leveraging custom scales.

Let's start with a simple example: alternating color scales. In order to make our new color scale, a drop in replacement for the existing `category10` scale we need to use a little bit of JavaScript fun to inject a new scale function into d3. We start with attaching the function to d3's scale namespace, as shown in the following code:

```
d3.scale.alternatingColorScale = function()
{
```

JavaScript allows for monkey patching of objects so this function will actually show up as being part of d3. We'll start implementing the function by setting up the domain and range. We define the `domain` and `range` functions which serve as getters and setters for the domain and range:

```
var domain, range;
  scale.domain = function(x){
    if(!arguments.length) return domain;
    domain = x;
    return scale;
  }
  scale.range = function(x){
    if(!arguments.length) return range;
    range = x;
    return scale;
  }
```

Finally, we'll set up the mapping function which is called when using the scale, as shown in the following code:

```
  function scale(x){
    return range[domain.indexOf(x)%range.length];
  }
  return scale;
}
```

This scale is applied by the following code:

```
var colorScale = d3.scale.alternatingColor()
  .domain(data.map(function(item){return item.month;}))
  .range(["#423A38", "#47B8C8", "#BDB9B1"]);
```

This results in a graph which appears more consistent and is shown in the following graph:

I picked the simplest condition on which to customize our scale, but more complicated and informative scales can be used. A scale which uses a threshold is shown in the following graph:

This is easily done by altering the `scale` function and passing the value instead of the key (month) into the function.

Labels and axes

Up until now, we've build the graphs without much attention to what we're graphing. It would be great to put some labels on the graph, so people can decode our data with ease. Fortunately d3 provides an `axis()` function which makes adding the axis a snap.

We'll start with the x axis, as shown in the following code:

```
var xAxis = d3.svg.axis().scale(xScale).orient("bottom");
  graph.append("g")
  .attr("transform", "translate(20,300)")
  .attr("text-anchor", "middle")
  .call(xAxis);
```

We start here by using the `axis` function to create an axis. We pass in our `xScale` to give it a hint about where the ticks on the scale should be placed. We next append a `g` element to our graph. `g` is an SVG element which acts as a container to hold other elements. You can place any other shape inside the element `g`, and then perform transformations on them as a whole. We do just that in the next step. The axis is, by default, at the top of the graph, so we shift it down and slightly to the right to better line up. The `text-anchor` property sets where the x coordinate of the text should be anchored. By default it is left, but as we have the middle of each bar we set `text-align:middle`. Finally, we pass in the `xAxis`.

The `yAxis` is added just as easily with the following code:

```
var yAxis = d3.svg.axis().scale(yScale).orient("left");
  graph.append("g")
  .attr("transform", "translate(20,6)")
  .call(yAxis);
```

The translation here is a bit more complex, and we're just accounting for the fact that we gave the scale the left orientation, which causes it to be drawn left of the minimum scale value. As our minimum scale value is `0`, it is drawn off screen, as shown in the following graph:

We've managed to add axis and labels in a matter of a few lines of code. With a more general SVG library, this would take a significant amount of work. There are a lot of configuration options for the axis function too. You can set the number of ticks, the labels, and the format of the ticks.

Summary

While we've looked at a great deal of d3, we've only just scratched the surface of what d3.js can do. It is a large and powerful library. There are many books written on the subject of d3, so we can't cover it all. A great resource for d3 is Swizec Teller's book *Data Visualization with d3.js, Packt Publishing*. There are tons of additional functions that we'll uncover, as we develop some of the visualizations in the rest of the book. We'll also visit some further applications of the functions presented in this chapter.

Chapter 5. Twitter

Twitter is a service that really grew up on having an open and available API. Initially, there were no Twitter clients. Communication with Twitter was limited to SMS and later the website. During the development of Twitter, the developers apparently racked up hundreds of dollars in SMS charges, testing, and building the system. Twitter grew in popularity on the back of hundreds of developers who built tools like **TweetDeck** and **Tweetree** using open Twitter APIs or the Twitter RSS feeds. As such, Twitter offers a rich API that can be used to build up applications, and in our case, visualizations.

Before we get into building visualizations, let's take a look at the Twitter API, and how we can make use of it. There is a great deal of documentation on the API available at https://dev.twitter.com. If you want more information, this should be your first stop for additional research.

For our purposes, Twitter offers two different models for retrieving data. The first is a typical RESTful model, where the client makes requests to Twitter for specific resources that are returned over HTTP in JSON format. This API is likely to be similar to others you have seen before. It is stateless, meaning that no information is retained server side between requests, and follows the best practices of HTTP. If you're attempting to consume data from Twitter in a web browser, then this is the option for you. The second option is the streaming API. This method makes use of a persistent HTTP connection to which Twitter will send messages as they occur. It is a generally poor idea to make use of the streaming API from a browser, so you'll need to have some sort of an intermediary server between the browser and the API, as shown in the following figure:

Having a server is, unfortunately, a requirement of all visualizations that make use of Twitter, even the RESTful API, as Twitter does not support authentication using a pure browser-based solution. We'll get into all that later, but first we'll need to set up a developer account with Twitter.

Getting access to the APIs

If you remember in Chapter 3, *OAuth*, one of the requirements was getting an application key for each site with which we want to talk. This applies to Twitter, so let's go and do that.

Open up a browser, head over to https://dev.twitter.com, and click on the **Sign in** link. If you already have a Twitter account, then you can use that here to sign in. If not then you can sign up for a new Twitter account. Don't worry, it is all free.

Once you're signed in, then in the top right-hand corner there should be a link to **My applications,** as shown in the following screenshot:

Clicking on that will take you to a page where you can go about setting up your first application. You'll need to enter some information about the application. You can enter whatever you choose for most fields, but the callback URL should be http://127.0.0.1:8080/twitter1.html. This is the URL to which Twitter will direct you once the OAuth phase is complete. We're using a localhost value here, but in production you would want to use the publicly-facing URL for your visualization. The following screenshot shows the **Application Details** window:

Application Details

Name: *

ExampleVisualization

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description: *

An example of how to use Twitter data to build a visualization

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website: *

http://blog.simontimms.com

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL:

http://127.0.0.1:8080/twitter1.html

Where should we return after successfully authenticating? For @Anywhere applications, only the domain specified in the callback will be used. OAuth 1.0a applications should explicitly specify their `oauth_callback` URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Tip

You cannot use a localhost domain here, but if you would rather not see an IP address, then you can use a URL shortening service to create an alias for your localhost URL. Make sure your URL shortener preserves query parameters, or you won't be able to login correctly.

Once your application has been created, you'll be able to see the various settings from the same **My applications** option that we used previously. The key information for our purpose is `OAuth settings`, as shown in the following figure:

OAuth settings

Your application's OAuth settings. Keep the "Consumer secret" a secret. This key should never be human-readable in your application.

Access level	Read-only About the application permission model
Consumer key	`Abdtj1FfuLoOMT08ofQqbw`
Consumer secret	`RImXET4AtUkaQHHgjIPNg87JN6L91GnkDszhYna8uE`

These keys will be used for user authorization with our application. Should the secret key be leaked—for instance, you might have pasted it into a book you're writing—it can be reset using the **Reset keys** link. Doing so will prevent readers of your book from pretending to be you and committing unspeakable evils in your name.

Setting up a server

As I mentioned, Twitter does not allow direct access to their authentication structure from a browser, we'll need to make use of a server. Fortunately, we can develop against a server on our own computer —no need for an external server. We're using a lot of JavaScript in this book, so let's keep that theme going and host our site locally using node.js. Any other HTTP server will also work.

Installing node.js is pretty simple. If you're on Windows, then there is an installer available from http://nodejs.org. On OS X, there is a `.pkg`-based installer available on the same site, or it can be installed using Homebrew. If you're using Linux, it is preferable to compile from source. However, if you're using a distribution with a built-in packaging system, such as **apt** or **yum**, then there is a node.js package that can be installed with either of the two commands:

```
sudo yum install nodejs            #Fedora or RedHat
sudo apt-get install nodejs        #Debian or Ubunt
```

node.js is a piece of server-side software that is designed to perform all of its I/O tasks asynchronously. This means that an operation such as writing to disk is handled without blocking the main thread. When the I/O is complete, the main thread is notified. One of the most common applications is its usage as an HTTP server. This functionally comes in the box in the form of the HTTP module, but the interface provided by that module is pretty lightweight. Instead, we'll make use of the Express framework. Express is a lightweight framework which provides some infrastructure around routing, sessions and serving content, and templating. It can be installed using the node package manager `npm` as shown in the following command:

```
npm install express
```

We'll make use of Express going forward.

OAuth

OAuth can, of course, be manually configured and controlled, but we stand on the shoulders of giants for good reason. It is much easier for us to make use of an already built OAuth library. Fortunately, node has such a library, creatively called **OAuth**. Even with this library, you'll see that interacting with an OAuth 1.0a endpoint is complicated. To install it, drop to the command line again and use the node package manager:

```
npm install oauth
```

This library can perform both OAuth 1.0a and OAuth 2.0 operations. As Twitter is an OAuth 1.0a endpoint, we'll be making use of that.

The first thing to do is set up our Express application. Express provides application templates, but they are an overkill for the simple application in this chapter. If you're planning on creating a more complicated application in future, you'll want to look more into app generation and directory structure for you, application. We start by requiring `express` and creating a new app using the loaded module, as shown in the following code:

```
var express = require("express");
var app = express();
var oAuth = require('oauth');
```

Require is a library which permits the dynamic loading of JavaScript libraries. It is the most common way to bring in external modules in a node application. Next, we configure a number of settings in `express`, as shown in the following code:

```
app.configure(function() {
  app.use(express.bodyParser());
  app.use(express.cookieParser() );
  app.use(express.session({ secret: "a secret key"}));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

`bodyParser` allows `express` for simple parsing of the body of requests sent to the server. On the next line, the `cookieParser` is set up. Like the `bodyParser`, this allows for parsing of cookies and populates the request object with values retrieved from cookies, in our case, information for the session. Next, we set up the session capabilities. This allows us to share information from request to request. In its default configuration, it uses in-memory storage to hold session information. This means that restarting your application will erase session information. If you're hosting your visualization on a farm of machines, you'll need to make use of an external data storage mechanism such as **MongoDB** or **Redis**. We pass in a secret key that is used in generating the `session` hash. It should be a random string. Using the `app.router` will instruct express to listen to route requests, which we'll define in a second. Finally our `.html` and `.js` files are going to be in a directory called `public`, so we'll instruct `express` to serve out the contents of that directory as static resources.

We now want to make use of the OAuth library. This can be done with a function as shown in the following code:

```
function getOAuth(){
  var twitterOauth = new oAuth.OAuth(
  'https://api.twitter.com/oauth/request_token',
  'https://api.twitter.com/oauth/access_token',
  consumerKey,
  consumerSecretKey,
  '1.0A',
  null,
  'HMAC-SHA1');
  return twitterOAuth;
}
```

We create an OAuth object associated with Twitter. We give the two end-points, and then the consumer key and consumer secret which we received from Twitter earlier. The requirement to embed the consumer secret for OAuth 1.0a is why client-side code cannot be used to retrieve information from Twitter. The consumer secret cannot be leaked to outsiders by sending it to the client. `1.0A` is passed in as the version of OAuth; no authorization callback is required so `null` is given as the sixth parameter. The final parameter is the signature method: Twitter uses `HMAC-SHA1`.

Next, we'll set up a route in the Express application to request the OAuth tokens from Twitter:

```
app.get('/requestOAuth', function(req, res){
  function recieveOAuthRequestTokens(error, oauth_token,
oauth_token_secret,results) {
    if (!error){
      req.session.oAuthVars = { oauth_token: oauth_token,oauth_token_secret:
oauth_token_secret}; res.redirect('https://api.twitter.com/oauth/authorize?
oauth_token=' + oauth_token);
    }
  requestOAuthRequestTokens(recieveOAuthRequestTokens);
});
function requestOAuthRequestTokens(onComplete){
  getOAuth().getOAuthRequestToken(onComplete);
}
```

Here, we hook up the `/requestOAuth` route to first request an OAuth token, and then use that to redirect the users to the sign in page on Twitter. We build an anonymous function and pass that into OAuth, because node is highly asynchronous. The callback model allows the main node thread to serve another request, while waiting for Twitter to get back to it with OAuth tokens. Once we have the OAuth tokens, we save them in a session state for use in the next step, and redirect to the Twitter authorization page.

Twitter will redirect the user once they have authenticated to the URL we defined when setting up the application. In our case, this will be served by the route `/receiveOAuth`, as shown in the following code:

```
app.get('/receiveOAuth', function(req, res){
```

```
  if(!req.session.oAuthVars){
    res.redirect("/requestOAuth");
    return;
  }
  if(!req.session.oAuthVars.oauth_access_token){
    var oa = getOAuth();
    oa.getOAuthAccessToken( req.session.oAuthVars.oauth_token,
req.session.oAuthVars.oauth_token_secret, req.param('oauth_verifier'),
    function(error, oauth_access_token, oauth_access_token_secret,tweetRes) {
      req.session.oAuthVars.oauth_access_token = oauth_access_token;
      req.session.oAuthVars.oauth_access_token_secret =
oauth_access_token_secret;
      GetRetweets(res, req.session.oAuthVars.oauth_access_token,
req.session.oAuthVars.oauth_access_token_secret);
    });
  }
  else
    GetRetweets(res, req.session.oAuthVars.oauth_access_token,
req.session.oAuthVars.oauth_access_token_secret);
});
```

This code takes the OAuth tokens passed back by Twitter's `redirect`, and performs the final step which is looking up the access tokens. Once we have these access tokens, they can be used to call the API—here done in the `GetRetweets` function. We'll save all the tokens generated in the session so that user's don't have to continually grant access to the Twitter API.

Tired of tokens yet? You should be! This exchange to set up OAuth 1.0a uses an awful lot of tokens. Fortunately, we're done with tokens and OAuth. Now we can get onto building a visualization with Twitter data!

Visualization

There is a whole bunch of APIs made available to us by Twitter. We should start perhaps by inventing something we would like to visualize and then decide if the data is available and how we would show it. I'm curious about which of the people I follow tweet the most. Some accounts such as `@kellabyte` seem to always be tweeting and others like @ericevans hardly at all.

Server side

Let's start by getting the data on the server side. In node.js, I set up a new route using the following code:

```
app.get('/friends', function(req, res){
  if(!req.session.oAuthVars || !req.session.oAuthVars.oauth_access_token){
    res.redirect('/requestOAuth');
    return;
  }
  var cursor = -1;
  receiveUserListPage(res, req.session.twitterVars.user_id,
req.session.oAuthVars.oauth_access_token,
req.session.oAuthVars.oauth_access_token_secret, cursor, new Array());
});
```

First, we check to make sure that we have the appropriate tokens available in the session. If not then we redirect back to the `requestOAuth` page which will start up the whole OAuth workflow. Next, we set an initial cursor value. Twitter limits the number of results which come back from its services. This avoids dumping a million records out to the consumer, which is not something either party is likely to want. For the API call, we'll be using the limit set to 20. However, Twitter also provides a continuation token which they call a cursor. By calling the service again with this token, the next page of results is returned. An initial value of `-1` gives the first page. The cursor along with all the required tokens is passed into `receiveUserListPage`, which will perform the actual lookups.

Tip

Rate Limits

Twitter limits the number of request you can send to their service. When developing a visualization, you may bump into these limits. Wait for 15 minutes and try again. In production, try caching your data so you don't have to query Twitter so frequently.

`receiveUserListPage` looks like the following code:

```
function receiveUserListPage(res, user_id, oauth_access_token,
oauth_access_token_secret, currentCursor, fullResults){
  var oauth = getOAuth();
  oauth.get( 'https://api.twitter.com/1.1/friends/list.json?
skip_status=true&user_id=' + user_id + "&cursor=" + currentCursor,
  oauth_access_token,
  oauth_access_token_secret,
  function (e, data, oaRes){
  var jsonData = JSON.parse(data);
  if(jsonData.errors){
    projectResults(res, fullResults);
    return;
  }
  fullResults = _.union(fullResults,
  _.map(jsonData.users,
```

```
  function(item){return { name: item.name,
    count: item.statuses_count
  }}));
  if(jsonData.next_cursor == 0){
    projectResults(res, fullResults);
  }
  else
    ReceiveUserListPage(res, user_id, oauth_access_token,
oauth_access_token_secret, jsonData.next_cursor, fullResults);
  }
});
}

function projectResults(res, fullResults)
{
  var selectedResults = _.first(_.sortBy(fullResults, function(item){return
item.count;}).reverse(), 10);
  res.end(JSON.stringify(selectedResults));
}
```

We start off by getting a reference to the OAuth library, then we use the current cursor and the current `user_id` to query Twitter. The API call we're using returns a set of 20 users from the list of people who I follow. The results are returned as a string so we parse them into an object using `JSON.parse`. If the resulting object contains a field called `errors`, then we've likely hit the rate limit, so we return everything we've pull down so far. Because the rate limit is only `15` for this API call, if you follow more than 300 people, you'll hit the limit.

If we have results, we append them to our current set of data. We use the underscore's `map` function to select only two of the fields from. This saves on bandwidth and makes debugging easier, as the objects returned from Twitter are very heavyweight with dozens of useless fields. If `next_cursor` is equal to `0`, then it means that we've reached the end of the list and can return the current set of names and counts. Otherwise we recourse into the function, giving it the new cursor, set of names, and items. Once we hit a case where we can return, we call `projectResults` which sends the 10 users with the most tweets to the client formatted as JSON.

Tip

Underscore.js

The underscore JavaScript library is a small library that makes working with arrays far easier. It adds set functions such as `union` and `intersect`, as well as functional programming concepts such as map and reduce. It can be downloaded from http://underscorejs.org/.

Client side

The client side visualization code can be placed in the public directory that we previously directed Express to serve out as static content.

I'd like to show the most active tweeters visually. A nice way of doing this is to use a bubble chart, and make the bubbles bigger the more tweets they have. Let's build up the code:

```
function visualize(data){
  var graph = d3.select(".visualization")
  .append("svg")
  .attr("width", 1024)
  .attr("height", 768);
  var colorScale = d3.scale.category10();
  calculateBubbles(data, 1024, 768);
  var currentX = 0;
  graph.selectAll(".bubble")
  data(data)
  enter()
  append("circle")
  .style("fill", function(x,y){return colorScale(y);})
  .attr("cx",  function(d){return d.cx;})
  .attr("cy", function(d){ return d.cy;})
  .attr("r", 0)
  .attr("opacity", .5)
  .transition()
  .duration(750)
  .attr("r", function(d){return d.radius;});
  graph.selectAll(".label")
  .data(data)
  .enter()
  .append("text")
  .text(function(d){return d.name + "(" + d.count + ")";})
  .attr("x", function(d){return d.cx;})
  .attr("y", function(d){return d.cy;})
  .attr("text-anchor", "middle");
}
```

Much of this code will look familiar now that you know a bit about d3. The data array which is passed in is what is retrieved from our node service. First, we create an SVG element on the page of an arbitrary size. Then, we set up a color scale so that our visualization will be nicely colored. The `calculateBubbles` function is a helper function which will calculate the locations of the bubbles. It augments our data array with the x and y coordinates for the circle as well as its radius. We won't get into that here, but the code is available on GitHub. For each of the top tweeters we create a bubble. We color it using the color scale, and set the location using the pre-calculated values from the data array. Initially, we set the radius to `0`, but then we use a transition to grow the circles for a nice loading effect.

For each one of the circles, we want to label what the circle represents. This is done by adding a text

element at the center of the circle.

The resulting graph, based on the 10 most active people I follow, looks like the following figure:

Every one of these people has tweeted over 35,000 times.

Summary

You should now be able to set up a new application to query Twitter, create the proper OAuth tokens using the OAuth library on node.js, and build a bubble chart. The Twitter API is rich and has many more potential visualizations lurking in it. I'm sure that we could come up with a couple of dozen potential visualizations. There is no better way to learn than through experimenting with the API, so don't be afraid to get messy.

In the next chapter we'll take a look at visualizations of data on the popular question and answer site, Stack Overflow. Their API is largely an open one which doesn't require authentication for most queries, so we should have a brief reprieve from having to use OAuth and even node.js.

Chapter 6. Stack Overflow

In 2008, the programming question market on the Internet was dominated by a company called Experts Exchange. Many were dissatisfied with the culture on the site and the requirement that people had to be registered to view answers. Programmers Jeff Atwood and Joel Spolsky launched the "question and answer" site, Stack Overflow. Since then, the site has, taken off, quickly growing to become one of the top 100 sites on the Internet. Users can ask and answer questions on the site about a wide variety of programming topics. Answering a question well or asking a well thought-out question wins reputation points, which are prominently displayed. Although, it's not a social media site like Facebook and Twitter, Stack Overflow's content is all user-created and user-moderated. Stack Overflow offers an API against which you can query for all sorts of interesting information.

Authenticating

Much of the query API is available without authenticating. However, if you want private information about users or want to write to the site, then you'll need to authenticate. There is also a much higher request limit for authenticated applications. Without authenticating, a single IP address is limited to 300 requests a day. With an authenticated application, this limit is raised to 10,000 requests.

Tip

Rate limits

Many social media sites use rate limits in their APIs. These limits are in place to prevent you from overloading the site, and also to save you from asking for too much data. Twitter processes more than 4,000 tweets a second. Without very special preparation, your infrastructure would quickly be overwhelmed if you were to process them all.

Again, this is a site that makes use of **OAuth** to authorize users. However, they make use of OAuth 2.0, which is much easier than the OAuth 1.0a we used in the previous chapter. We'll limit ourselves to making use of public information to avoid authenticating. Should you wish to authenticate, I promise it is easier than Twitter. You can find instructions at https://api.stackexchange.com/docs/authentication. Stack Overflow uses the same authorization system as Facebook, so the example from the OAuth chapter should work perfectly.

Creating a visualization

Many of the questions on Stack Overflow have a large number of answers. The site is not optimized to show the latest answers; the answers are ranked by being the most accepted answer then randomly. This is done to give all answers a chance at being shown near the top which should, in theory, encourage people to vote for the best answer instead of just the first answer shown.

For this visualization, I would like to show how a question has been answered over time. Are more recent answers likely to get a higher score? Is the first answer always the best?

Let's start by pulling down the data for an individual question which has a large number of answers. To do this, we'll make use of the questions API. All of the API endpoints are hosted on https://api.stackexchange.com. We're going to make use of the latest API which is Version 2.1. This is also encoded into the URI, as is the specific endpoint and the ID. Within the question API, we're interested in the answers, so we can query specifically for them, giving us a URI of https://api.stackexchange.com/2.1/questions/{id}/answers.

In the query string, we'll specify the site against which we want to query. Stack Exchange hosts several dozen question and answer sites modeled on Stack Overflow, all of which are served from the same API endpoint, so it is necessary to filter just for Stack Overflow by passing in `site=stackoverflow`:

```
function retrieveQuestionAnswers(id){
  var page = 1;
  var has_more = true;
  var results = [];
  while(has_more) {
    $.ajax(https://api.stackexchange.com/2.1/questions/ + id + "/answers?
site=stackoverflow&page=" + page,{
        success: function(json){
          has_more = json.has_more;
          results = results.concat(json.items);},
        failure: function() {
          has_more = false;},
          async: false
        });
        page++;
    }
  return results;
}
```

Twitter provided us with the continuation tokens that we could pass back to Twitter to request the next page of data. Stack Overflow takes a different approach and assigns page numbers, allowing us to browse through the results with ease. Embedded in the response for every API call is a token called `has_more`, which is true whenever there are more pages of data that match the current query.

In this code, we make use of the continuation token and the page number to perform as many queries

as necessary to retrieve all the answers. We are making use of the jQuery function `ajax`, instead of the more common `getJson` function, because we would like to retrieve the data synchronously. We do this because we want the entire dataset at one time. If your visualization allows for data to be added dynamically then you can relax the `async:false` requirement.

What's returned is an array of objects, each one of which represents an answer to a question. If we give the `retrieveQuestionAnswers` method an ID such as `901115`, then we'll get back an array of 50 answers. These come back over the course of two requests and the code above merges them together into the results array which is returned.

Each `Answer` contains a number of fields. A list of the fields returned by default can be found at https://api.stackexchange.com/docs/types/answer. For the purpose of our visualization, we're most interested in when the answer was originally suggested, its score, and also whether it was chosen as the accepted answer. These bits of information can be found in the fields: `creation_date`, `score`, and `is_accepted`. We'll ignore the rest of the fields for now.

Now that we have some basic data, we can start thinking about the visualization. We're trying to convey the relationship between the age of a question and its score. This sounds a lot like a use for a scatter plot. The data points stand on their own and can be placed along two axis, date and points. My theory before starting on this that answers that are older will tend to have a higher score, because they've been around longer to gather points. People are programmed to believe that numbers going up are positive, so let's play to that and plot points versus age which will, if my theory holds, have higher values on the right.

Of course, a scatter plot is boring and nothing we couldn't generate outside of Excel. We'll add some interactivity to it, but to start, we'll still need a simple scatter plot.

This is easily done with a couple of scales and some circles, as shown in the following code:

```
var graph = d3.select("#graph");
var axisWidth = 50;
var graphWidth = graph.attr("width");
var graphHeight = graph.attr("height");
var xScale = d3.scale.linear()
  .domain([0, d3.max(data, function(item){ return item.age;})])
  .range([axisWidth,graphWidth-axisWidth]);
var yScale = d3.scale.log()
  .domain([d3.max(data, function(item){return item.score;}),1])
  .range([axisWidth,graphHeight-axisWidth]);
```

This gives a very flat graph with the majority of the data being close to zero, while the scale is skewed by a high outlier with a score over 2000, as can be seen in the following figure:

This can be ameliorated by using a logarithmic scale. Any time you use a non-standard scale like logarithmic, you'll want to put in axis labels to prevent causing confusion or misleading the consumer of the visualization.

```
var yAxis = d3.svg.axis()
  .scale(yScale)
  .orient('left')
  .tickValues([1,5,10,50,100,500,1000,2000])
  .tickFormat(function(item){return item;});
graph.append("g")
  .attr("transform", "translate(" + axisWidth +",0)")
  .call(yAxis);
graph.append("text")
  .attr("x", "0")
  .attr("y", graphHeight/2)
  .attr("transform", "rotate(90, 0, " + graphHeight/2 + ")")
  .text("Score");
```

The labels in this graph are manually assigned to give the best spread. You can automatically assign labels, but I found them to be declared at odd places. I also defined a function to format the labels, otherwise they had a tendency to be formatted using scientific notation (*2 * 10^3*). Finally, I appended some text as an axis label. I also added an age axis that lists the age of the answer in days.

```
var xAxis = d3.svg.axis().scale(xScale).orient('bottom');
graph.append("g")
  .attr("transform", "translate(0," + (graph.attr("height") - axisWidth)  +")")
  .call(xAxis);
graph.append("text")
  .attr("x", graphWidth/2)
  .attr("y", graphHeight-5)
  .style("text-anchor", "middle")
  .text("Age in days");
```

The only special thing worth noting in this code is that the label is rotated using a transform, as it appears along a vertical axis. The resulting graph looks like this figure:

Now that we have a basic visualization, we can start spiffing it up with some interaction.

The simplest interaction we can add is to pop up a label when somebody moves the mouse pointer over one of the points.

This can be done by using the `on()` function of `d3`. This function can tie event listeners to the elements created as part of an SVG. To start, we add to the end of circle appending from above, as shown in the following code:

```
//append circle
.on("mouseover", function(item){
  showTip(item);
});
```

Here, the `showTip()` function will be called whenever the user hovers the mouse over one of the circles in the above graph. The `item` parameter, which is passed into the event handler, is the item from the data collection that is attached to the hovered circle. If you need additional information about the event, and we do, then that can be found attached to the global variable `d3.event`.

In the event handler, we first highlight the selected circle by ensuring all other circles are black and then making the selected one blue:

```
function showTip(item){
  d3.selectAll(".score").attr("fill", "black");
  d3.select(d3.event.srcElement).attr("fill", "blue");
```

It may also be useful to change the size of the circle to draw even more attention to it. This can be done by simply updating its attributes. Next, we hide the previous tip and set the inner contents of the tip to take values from the selected data element:

```
  d3.select("#tip").style("opacity", 0);
  d3.select("#count").text(item.score);
  d3.select("#age").text(Math.floor(item.age));
  d3.select("#profileImage").attr("src",
  item.owner.profile_image);
  d3.select("#profileName").text(item.owner.display_name);
```

Finally, we move the tool tip to be next to the circle and have it fade in:

```
  d3.select("#tip").style("left", d3.event.x + "px");
  d3.select("#tip").style("top", d3.event.y + "px");
  d3.select("#tip").transition().duration(400).style("opacity", .75);
}
```

The end result looks like the following diagram:

Adding interactivity to your visualization allows you to present far more data than would normally be possible. Hiding data so that it can only be seen by moving the mouse over, or clicking on it prevents overwhelming your users while still providing the maximum amount of information.

Filters

The data returned by our query isn't exactly what we want. For instance, we don't care about `last_edit_date` or even the `last_activity_date`, but we do care about the number of up and down votes. By pulling extra data back, we're wasting bandwidth and slowing down the visualization for our users. Fortunately, Stack Overflow has a solution for that in the form of filters.

Tip

Deep queries

If you find that you need to explore the Stack Overflow data in greater depth than is provided for by the API, you can download a dump of the entire site at http://www.clearbits.net/creators/146-stack-exchange-data-dump. This dump is provided every three months, and currently clocks in at 13.4 GB compressed. With this dump, you can run much more complex queries without the fear of hitting a rate limit.

Filters govern what data is returned from the API, and can be used to either add or remove fields. They are statically created so you should only need to create them once and there is no need to create a new filter each time you query the site, or even each time your application is launched. In fact, I actually make use of the API explorer provided by Stack Exchange to create my filters ahead of time. The URL for creating filters is https://api.stackexchange.com/docs/create-filter.

In the **include** field, you can place a semi-colon to include a delimited series of names. Everything which is part of the answer object is prefaced by answer, so the answer owner would be referred to as `answer.owner`. The default filter is quite inclusive so as a base filter I've used the special `none` filter. This includes no fields unless they are explicitly included. Using the `none` filter as a base is the best practice to reduce excess queries, as shown in the following figure:

If you do start with the `none` filter, be sure to add the tokens `.items` and `.has_more` to the include list. Without items, the items collection—which holds either questions, answers, or users depending on the query—isn't included and `has_more` is needed to tell if there are additional pages. For our purposes, the following filter is perfect:

```
answer.answer_id;answer.owner;answer.score;answer.down_vote_count;answer.upvote_
```

```
count;answer.creation_date;shallow_user.profile_image;shallow_user.display_name;
.items;.has_more
```

The `create` filter returns an alpha-numeric string which can then be used in our query to filter it appropriately. The URL against which we're querying becomes the following:

```
"https://api.stackexchange.com/2.1/questions/" + id + "/answers?
site=stackoverflow&filter=!2BjddbKa0El(rE-eV_QT8)5M&page=" + page
```

By using a filter, I was able to reduce the payload returned from the API to 3kB from 22kB. This is a significant saving, especially over low bandwidth connections.

Summary

You should now be able to query against the Stack Exchange API for not just the Stack Overflow but for all the Stack Exchange sites. You should also have some idea of how to add interactivity to your visualizations through the use of `d3`. In the next chapter we'll take a look at using Facebook as a source of data to visualize.

Chapter 7. Facebook

Facebook is the 900 lbs (408.233 kg) gorilla of the social media world. Literally created in a university dorm, Facebook has grown to have 1.1 billion active users. That's one out of every seven people on the planet. It has impressive sway to say the least. No book about using social media APIs could be complete without an investigation of how to use Facebook's API.

Creating an app

As you might expect for such a large site, there are numerous APIs available for building applications related to Facebook. The simplest involve integrating "post to Facebook" buttons in websites or mobile apps, and the most complicated allow you to actually run the code on Facebook's servers as an app. We're going to make use of the Graph API.

The Graph API provides an HTTP-based method of accessing information, what Facebook calls the social graph. The graph is really just the relationship between various users and their data. It is a graph and a collection of nodes and edges as opposed to the bar graph-style of chart.

To get started, we'll register an application with Facebook, much as we did with Twitter and would have to do with Stack Overflow if we wanted to make use of authentication. To do this we'll head over to http://developers.facebook.com and click on the **Apps** link in the top menu bar. From there, click on **Create New App**. You'll be presented with the **Create New App** dialog box as shown in the following screenshot:

The app name can be anything you like, the app namespace is used to give your app a location on Facebook such as apps.facebook.com/NiftyVisualization. It is largely unnecessary for our purposes. The app category is wholly up to you and should be determined based on what it is that you're visualizing. **Heroku** is a cloud-based hosting provider partnered with Facebook to provide hosting space for Facebook applications. If you don't have hosting already, Heroku is a reasonable alternative and does support node.js; however, using it is outside the scope of this book.

Once you've filled out the details of your application, you'll be asked to confirm that you're a human by solving a CAPTCHA puzzle. You'll now be taken to the edit page where you can fill in the last few details before testing out your access to the API. It looks like the following screenshot:

Here, you'll need to fill in at least one app domain. This value is checked by the API when you sign in to ensure that your app is being used from an authorized domain. Unfortunately, you can't access Facebook's Graph API from a file that isn't served from a domain. This means that just going to `file://c:/code/visualization.html` won't allow you to access Facebook's API. Fortunately not all is lost, using `localhost` is permitted, but that does mean we have to run an HTTP server. We can make use of the same `Node.js` installation which we used in previous chapters.

The site URL in the website with the Facebook login should be set to be the return URL for your OAuth key exchange. We can actually set this to anything because we're going to use AJAX to do the authentication, and our users won't ever actually move away from our initial page.

Using the API

It is perfectly possible to manually perform OAuth to authenticate and authorize your visualization with Facebook. However, Facebook has been kind enough to provide a very usable JavaScript SDK. The API abstracts away the process of logging into a function call. To make use of the API, we first need to include it in our visualization. To do this, simply include the following script inside one of your `script` tags:

```
(function(d){
  var js;
  var id = 'facebook-jssdk';
  var ref = d.getElementsByTagName('script')[0];
  if (d.getElementById(id)) {return;}
  js = d.createElement('script'); js.id = id; js.async = true;
  js.src = "//connect.facebook.net/en_US/all.js";
  ref.parentNode.insertBefore(js, ref);
}(document));
```

This code will create a new `script` tag in your document and set its source to a file on the Facebook site. Adding a `script` tag like this will cause the browser to load the contents of that script file and execute it. Because we're loading the script in an asynchronous fashion, we'll need to wait for it to be loaded before we make use of it. The SDK calls a hook, `fbAsyncInit`, once it is initialized. We just need to tie a function to that hook, as shown in the following code:

```
window.fbAsyncInit = function() {
  FB.init({
  appId      : '525498574499442',
  channelUrl : '//localhost:8080/channel',
  status     : true,
  cookie     : true,
  xfbml      : true  // parse XFBML
  });
};
```

This will provide the SDK with the App ID available from the developer's webpage. Also, here I've provided `channelUrl` that is used to solve some cross-domain issues that appear on some browsers. Setting the status will have the `init` method fetch the value of `status`. `cookie` will enable cookie support. Finally `xfbml` enables Facebook markup language. What's that? It is a collection of specially formatted HTML elements that are controlled by the Facebook SDK.

For instance, if we would like to show a login button (how convenient that we do want to show a login button) then we can simply add the following code:

```
<fb:login-button show-faces="true" width="200" max-
rows="1"scope="user_birthday,email,friends_birthday">
</fb:login-button>
```

When an unauthenticated user opens the page, then a login button will show. When an authenticated

user reaches the page, they will be shown their own login information. You'll note the `scope` attribute; this is used to give Facebook a list of permissions you're requesting. Here we've asked for the logged in user's birthday, e-mail ID, and their friends' birthdays. When signing in, the user will be prompted by Facebook to allow your visualization access to that permission. There are about three dozen different permissions which can be requested from Facebook that govern everything from retrieving information about the logged in user, to their friends, events, and RSVPs. It is well worth poking around in here to discover interesting aspects to visualize.

The final piece in the authentication puzzle is to provide a function for the login button to call once it has logged the user in:

```
FB.Event.subscribe('auth.authResponseChange', function(response) {
  if (response.status === 'connected') {
    //use SDK here
  } else if (response.status === 'not_authorized') {//not authorized
    FB.login({scope: "user_birthday,email,friends_birthday"});
  } else { //not logged in
    FB.login({scope: "user_birthday,email,friends_birthday"});
  }
});
```

This event is triggered whenever there is a change in the authorization response, such as when we get authentication back from the login button.

Retrieving data

Before we go about retrieving data we should probably decide what data we wish to visualize. The amount of data which is available about the logged in user is not all that great (at least it isn't for me but I hardly use Facebook). That leaves us with looking at our friends. I found devices my friends use to access Facebook to be quite interesting. Are they more Android users or iOS? This information is available as part of the friends' collection. To retrieve this information we can use the `FB.api()` method:

```
FB.api('/me?fields=friends.fields(devices)', function(response){
  for(i = 0; len = response.friends.data.length; i< len; i++){
    var friend = response.friends.data[i];
    if(friend.devices)
      for(j = 0; j< friend.devices.length; j++)
    if(friend.devices[j].hardware != "iPad")
      operatingSystems[friend.devices[j].os]++;
  }
});
```

Into the `api()` method we pass a URL to be retrieved. In this case, we request the special URL `/me` which refers to the currently logged in users. We also provide a filter so that only the friends' collection is retrieved and, in fact, only the device's collections are retrieved for each friend. In the callback, we're just counting up the number of Android versus iOS devices. iPads and iPhones are separate devices to Facebook, but we don't want to count iOS as an access method twice, so we ignore any iPads. Once this code executes, we end up with a collection of device counts. For my friends I got the following:

```
{Android: 28, iOS: 36}
```

Visualizing

One of the more effective visualization techniques is to show the relative strengths of different categories by showing a scaled image. We saw this technique applied in the Twitter chapter using bubbles. We can take that to the next step by using images instead of just circles.

The first step is to locate logos for Android and iOS that are already SVGs. As it turns out, Wikipedia is a great source for this and their images are all licensed under creative commons, meaning we can use them in our visualizations. One of the really great features of SVG is that you can easily merge two images together through the use of definitions. If you open up an SVG like the Android logo at http://upload.wikimedia.org/wikipedia/commons/e/e1/Android_dance.svg, you can copy all the markup under a `<defs>` tag in another image. I took the Android and Apple logos and moved them into my raw markup. If I wanted to display them, I could use the `<use>` tag and reference the definitions by ID. It looks like the following code:

```
<defs>
  <g id="appleLogo">
    <!--various shapes needed to build the Apple logo-->
  </g>
  <g id="androidLogo">
    <!--various shapes needed to build the Android logo-->
  </g>

</defs>
<use x="0" y="10" xlink:href="#appleLogo"/>
<use x="512" y="10" xlink:href="#androidLogo" />
```

This will create an Apple logo next to an Android logo in our SVG. Knowing that we can leverage `d3` to build and scale the logos as appropriate, we're lucky, in that both of the SVGs we have are 256 px square, so they look to be approximately the same size before we've translated them. The `d3` is relatively simple, as shown in the following code:

```
var visualization = d3.select("#visualization");
visualization.selectAll(".logo").data(operatingSystems)
.enter().append("use")
.attr("xlink:href", function(item){ return "#" + item.os + "Logo";})
.attr("transform", function(item, index){
  return "translate("  + 300 * index + " 0),scale(" + (item.users /
operatingSystems[0].users) + ")";
});
```

We start by selecting the SVG then instead of appending shapes, we append using statements. The `xlink:href` attribute takes the value of the definition to include. Next, we scale and translate the logos so they are next to each other and the appropriate size. We set the first logo to be the baseline size, and every subsequent logo is drawn as a percentage of that. This only works because our numbers are quite close. With highly divergent numbers, a more robust strategy would be needed. With some additional text elements appended, the result is the following figure:

android
28
apple
36

Summary

You should now have a grasp of how to make use of the Facebook API to retrieve data. This data can then be visualized using any technique. In the next chapter we'll take a look at the upstart Google+ social network and see how we can leverage the data present there for visualizations.

Chapter 8. Google+

Of the major social networking sites, Google+ is the newest entrant. Although an upstart, it does have a large user base claiming to have more than 350 million active accounts (http://ca.ign.com/articles/2013/05/02/report-google-bigger-than-twitter-with-359-million-active-users). This is not Google's first attempt at breaking into the billion-dollar social media market. They have, in the past, created Google Buzz, Google Friend Connect, and Orkut in an attempt to gain a large user base. All but Orkut have since been mothballed and its user base is almost entirely located in Brazil. Google has purposefully avoided creating a write API in the hope that it eliminates automatically-posted spam. Google+ provides a read-only API that we can leverage to create visualizations; however, the API is very limited in comparison to other such APIs—while this book is being written you cannot even list the members of a circle.

Creating an app

Google+ is another OAuth 2.0 site so we, of course, need to get an application key as the first step to creating any visualization. This also means that we will need a return URL, so again we'll need to set up an HTTP server to run the visualization.

The first step is to log into https://code.google.com/apis/console using your Google account. Should you not have such an account, you can also create one from that page. Once on the site, you'll be presented with a giant button allowing you to create an application project. This console actually governs the access to all of Google's APIs, and there are quite a few.

Next, you'll be presented with a huge list of the various APIs. If you scroll way down, you'll eventually find Google+ (use the search, it will save hours of scrolling). Toggle the switch to the "on" position. You may need to agree to a couple of user agreements. Be sure to read the entire agreement as you always do.

The next step is to request a new key, as shown in the following screenshot. This can be done from the **API Access** tab in which you should click on **Create an OAuth 2.0 client ID…**. In the dialog that opens, you'll need to fill in an application name and a URL. This is not the URL for the OAuth exchange; that comes in the next tab. On this tab, enter a URL from which the OAuth request may originate and to which it should return. For our purposes, `http://localhost:8080` will be the domain:

Create Client ID

Client ID Settings

Application type

Web application
Accessed by web browsers over a network.

Service account
Calls Google APIs on behalf of your application instead of an end-user. Learn more

Installed application
Runs on a desktop computer or handheld device (like Android or iPhone).

Authorized Redirect URIs (fewer options)

One per line. For example: `https://example.com/path/to/callback`

```
http://localhost:8080/oauth2callback
```

Authorized JavaScript Origins

One per line. For example: `https://example.com`

```
http://localhost:8080
```

Create client ID | Back | Cancel | Learn more

We'll then receive a couple of keys ready for use in our application. The client ID is the field you want to use in your scripts.

Retrieving data

As with Facebook, we could do manual authentication against the OAuth 2.0 endpoint, but let's make use of Google's provided API. Hooking it up is very simple:

```
document.addEventListener("DOMContentLoaded",
 function() {
    var po = document.createElement('script');
    po.type = 'text/javascript'; po.async = true;
    po.src = 'https://plus.google.com/js/client:plusone.js';
    var s = document.getElementsByTagName('script')[0];
    s.parentNode.insertBefore(po, s);
  });
```

This runs once the document is ready and loads the API script from Google's servers by placing a new `script` tag on the page just before the tag including jQuery. The loaded document contains a number of JavaScript functions that can be used to interact with Google APIs but not specifically Google+ APIs—that happens after login.

To add a login button, we add the following HTML code:

```
<button class="g-signin"
   data-scope="https://www.googleapis.com/auth/plus.login"
   data-requestvisibleactions="http://schemas.google.com/AddActivity"
   data-clientId="988552574579.apps.googleusercontent.com"
   data-callback="onSignInCallback"
   data-theme="dark"
   data-cookiepolicy="single_host_origin">
</button>
```

This will generate the login button. The various `data-` properties attached to this button are processed by the script that we loaded from Google. The login is scoped to Google+ instead of one of the other Google APIs. The Client ID should be set to the one that was retrieved when creating the app. Most importantly, a callback function is assigned that will be activated when the request to log in succeeds. The callback will dynamically load the Google+ API:

```
function onSignInCallback(authResult) {
      gapi.client.load('plus','v1', function(){
        if (authResult['access_token']) {
          $('#gConnect').hide();
                          retrieveFriends();
        } else if (authResult['error']) {
          console.log('There was an error: ' + authResult['error']);
          $('#gConnect').show();
        }
        console.log('authResult', authResult);
      });
    }
```

Once the API for Google+ is loaded, we can make use of it, as we are doing in the highlighted line.

This function also hides the sign in button, so users don't attempt to sign in more than once. `retreiveFriends` is simple and will just send off a request to retrieve a list of friends:

```
retrieveFriends: function(){
      var request = gapi.client.plus.people.list({
        'userId': 'me',
        'collection': 'visible'
      });
      request.execute(retrieveFriendsCallback);
    }
```

Now that we have a list of friends, we can set about building a simple visualization using them.

Visualization

`d3` purposefully steers away from providing concrete visualizations. There is no single function you can call to get a bar chart or a scatter plot. Instead, it provides tools around creating the visualizations; these tools in turn provide a high degree of flexibility and empowers the creation of unique visualizations. One of the more powerful tools is the layout mechanism. Layouts provide some of the boilerplate code that would have to be written to achieve a certain sort of visualization.

We're going to make use of the **Force-directed graph** layout. Force-directed graphs provide a way of visualizing data that is interconnected. The strength of the bonds between nodes is frequently a function of how closely related the nodes are.

Our first step is to transform our data into a list of nodes and edges. As the API returns such limited data, we'll only be able to establish relationships between you and your friends. Those relationships will make up the edges or links, and the friends, the nodes:

```
var nodes = [];
 var links = [];
var centerNode = { name: "Me"};
nodes.push(centerNode);
for(i = 0; i< data.items.length; i++){
   var node = { name: data.items[i].displayName, image:
data.items[i].image.url};
   nodes.push(node);
   links.push({source: centerNode, target: node});
}
```

Now that we have the nodes and links, we can create a force layout using them:

```
var graph = d3.select("#graph");
var force =
d3.layout.force().charge(-120).linkDistance(100).size([500,500]).nodes(nodes).st
art();
```

The `charge` and `linkDistance` functions govern how widely the nodes disperse themselves. For the links, we draw a simple line to represent them:

```
var link =
graph.selectAll(".link").data(links).enter().append("line").attr("class",
"link");
```

The nodes are a bit more complicated, because for each one we need to set a picture taken from the Google+ data, the initial location, as well as the dimensions. We also need to attach an event handler to the nodes so that when dragged, the `force.drag` action is fired:

```
var node = graph.selectAll(".node").data(nodes)
           .enter()
            .append("image")
             .attr("xlink:href", function(d){ return d.image;})
```

```
            .attr("class", "node")
            .attr("r", 15)
            .attr("x", 250)
            .attr("y", 250)
            .attr("width", 50)
            .attr("height", 50)
            .call(force.drag);
```

Finally, we need to instruct `d3` what action should be taken on each tick when animating the graph:

```
force.on("tick", function () {calculatePosition(link, node);});
```

This will result in a graphic that shows my links to friends on Google+, as shown in the following screenshot. If you click and drag a node, it will move and all the nodes will rebalance themselves to account for the movement:

Summary

The `limited` API of Google+ does limit some of the visualizations we can create. There has been a rumor for years that Google will surface additional functionality in Google+, but I have seen no real action so far. You should now be able to authenticate against Google+ and retrieve data from it. You should also be able to make use of the graphically pleasing force-directed layout from `d3` as well as any of the other available layouts.

Part 3. Module 3

Learning d3.js Data Visualization

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

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

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

What is D3.js?

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

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

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

What's ES2016?

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

Tip

For a really good rundown of all the new toys you have with ES2016, check out this nice guide by the folks at Babel.js, which we will use extensively throughout this book:

https://babeljs.io/docs/learn-es2015/.

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

Getting started with Node and Git on the command line

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

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

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

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

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

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

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

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

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

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

Note

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

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

command:

```
$ git checkout chapter1
```

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

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

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

```
$ npm install
```

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

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

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

```
$ npm start
```

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

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

localhost:8080/src/

Elements Network Sources Timeline Profiles »

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

html body script

Styles Event Listeners DOM Breakpoints Properties $scope

Filter

```
element.style {
}
script {                    user agent stylesheet
  display: none;
}
```

margin
border
padding
auto × auto

Console Emulation Rendering

<top frame> Preserve log

Filter Regex Hide network messages

All Errors Warnings Info Logs Debug Handled

>

A quick Chrome Developer Tools primer

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

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

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

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

backface-visibility
background
background-attachment
background-blend-mode
background-clip
background-color
background-image
background-origin
background-position
background-position-x
background-position-y
background-repeat
background-repeat-x
background-repeat-y
background-size
background: ;
}

Breakpoints Properties $scope

body { user agent stylesheet
display: block;
margin: ▶ 8px;
}

margin 8
border -
padding -
8 - - 884 × 0 - - 8
-
-
8

The obligatory bar chart example

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

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

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

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

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

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

Uncaught Error: Cannot find module "d3"

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

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

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

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

```
$ npm install
```

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

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

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

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

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

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

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

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

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

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

The preceding code is not really all that interesting, but wouldn't it be awesome if you never had to type that out again? Well! Because you're a total boss and are learning ES2016 like all the cool kids,

you won't ever have to. Let's create our first child class!

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

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

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

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

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

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

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

Note

Hoorah! Another fancy new ES2016 feature!

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

If you have no idea what I'm talking about here, don't worry. It just means that you should define variables with `let` because they're more likely to act as you think they should and are less likely to leak into ot1her parts of your code. It will also throw an error if you use it before it's defined, which

can help with troubleshooting and preventing sneaky bugs.

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

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

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

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

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

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

Now the fun begins!

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

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

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

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

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

population value.

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

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

Replace it with this line:

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

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

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

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

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

Note

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

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

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

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

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

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

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

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

```
        .call(yAxis);
```

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

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

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

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

Tip

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

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

Keep this in mind when you're tinkering.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

.bar {
  fill: steelblue;
}
```

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

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

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

Tip

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

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

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

Summary

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

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

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

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

Chapter 2. A Primer on DOM, SVG, and CSS

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

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

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

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

DOM

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

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

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

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

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

Manipulating the DOM with D3

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

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

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

Whoa, that's a lot of code!

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

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

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

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

Let's rewrite the previous example in D3:

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

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

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

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

Selections

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

Note

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

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

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

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

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

Let's make a table!

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

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

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

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

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

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

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

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

This will give us the outer container. In order to see this work, we need to tell `index.js` to load our new class. Open that up now. You'll notice that all of our code from the last chapter is still there. How

messy! Let's start by cleaning it up using ES2016 modules. Move the code for the `BasicChart` class into `basic-chart.js`, and move `BasicBarChart` into `basic-bar-chart.js`. Lastly, you need to let `BasicBarChart` know where the `BasicChart` class is, so put the following line at the top of `basic-bar-chart.js`:

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

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

Note

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

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

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

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

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

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

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

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

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

Note

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

command inside the `learning-d3/` directory:

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

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

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

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

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

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

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

Go to the command line and type this:

```
$ npm start
```

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

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

Woo! A `table` element!

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

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

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

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

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

    // Each element in "header" is a string.
```

```
    this.header.forEach(function(value){
      tableHeader.append('th').text(value);
    });

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

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

    return table;
  }
}
```

Now your table should look like this:

one two three four five six

q w e r t y

Elements Console Sources Network Tir

```
…<!DOCTYPE html>
  <html>
  ▶ <head>…</head>
  ▼ <body>
      <div id="chart"></div>
      <script src="/assets/bundle.js"></script>
    ▼ <table class="table">
      ▼ <thead>
        ▼ <tr>
            <th>one</th>
            <th>two</th>
            <th>three</th>
            <th>four</th>
            <th>five</th>
            <th>six</th>
          </tr>
        </thead>
      ▼ <tbody>
        ▼ <tr>
            <td>q</td>
            <td>w</td>
            <td>e</td>
            <td>r</td>
            <td>t</td>
            <td>y</td>
          </tr>
        </tbody>
      </table>
    </body>
  </html>
```

What exactly did we do here?

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

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

Okay, time to finally play with some selections!

Selections example

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

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

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

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

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

Tip

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

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

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

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

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

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

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

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

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

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

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

Manipulating content

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

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

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

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

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

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

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

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

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

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

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

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

Tip

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

Joining data to selections

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

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

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

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

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

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

Note

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

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

An HTML visualization example

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

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

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

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

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

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

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

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

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

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

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

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

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

```
}
```

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

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

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

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

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

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

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

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

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

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

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

After execution, the `rows` property will hold an array of `<tr>` elements, each bound to its respective

place in the dataset. The first `<tr>` element holds the first datum, the second holds the second datum, and so on.

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

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

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

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

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

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

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

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

```
this.tableBody.selectAll('tr')
```

```
      .sort((a, b) => d3.ascending(a.Group, b.Group));
```

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

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

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

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

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

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

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

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

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

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

Note

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

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

That's a much nicer table!

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

Scalable Vector Graphics

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

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

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

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

Drawing with SVG

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

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

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

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

let d3 = require('d3');

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

export function renderDailyShowGuestTable() {
  let url =
      'https://cdn.rawgit.com/fivethirtyeight/data/master/daily-show-
guests/daily_show_guests.csv';

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

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

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

Note

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

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

Manually adding elements and shapes

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

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

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

Text

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

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

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

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

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

Our image looks like this:

Shapes

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

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

```
svg.append('line')
  .attr({x1: 10,
    y1: 10,
    x2: 100,
    y2: 100,
    stroke: 'blue',
    'stroke-width': 3
  });
```

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

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

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

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

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

Our image now looks like this:

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

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

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

Let's try adding a circle:

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

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

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

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

```
  });
```

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

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

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

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

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

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

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

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

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

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

Transformations

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

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

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

According to Wikipedia, an affine transformation is any transformation that preserves points, straight lines, and planes, while keeping sets of parallel lines parallel. They don't necessarily preserve distances but do preserve ratios of distances between points on a straight line. This means that if you take a rectangle, you can use affine transformations to rotate it, make it bigger, and even turn it into a

parallelogram; however, no matter what you do, it will never become a trapezoid.

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

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

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

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

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

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

Our picture looks like what is shown here:

Let's rotate the eye by 45 degrees:

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

That's not what we wanted at all!

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

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

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

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

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

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

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

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

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

Much better! We get exactly what we wanted:

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

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

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

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

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

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

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

Using paths

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Line

To create a line, we use the `d3.svg.line()` generator and define the `x` and `y` accessor functions.

Accessors tell the generator how to read the `x` and `y` coordinates from data points.

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

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

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

Now we get to define a simple path generator:

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

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

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

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

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

Our graph looks as follows:

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

```
d="M0,220L48.888888888888886,0L97.77777777777777,219.99999999999994L146.66666666
666666,440L195.55555555555554,220.00000000000006L244.44444444444446,0L293.333333
3333333,219.99999999999991L342.22222222222223,440L391.1111111111111,220.00000000
000009L440,0"
```

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

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

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

Let's try something else:

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

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

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

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

Area

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

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

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

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

```
    (chart.margin.left + chart.margin.right)) +
    ', ' + chart.margin.top + ')');
```

Now we define an area generator and draw an area:

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

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

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

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

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

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

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

Arc

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

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

Let's draw an arc:

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

```
    ')');

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

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

Huzzah, a simple arc. Rejoice!!

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

Symbol

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

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

As always, we start with a path generator:

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

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

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

Now we can draw some symbols:

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

Note

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

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

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

Chord

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

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

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

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

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

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

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

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

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

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

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

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

The `.source()` and `.target()` accessors tell us that the first item in every pair is the source and the second is the target. For `startAngle`, we remember that a full circle is `2Pi` and divide it by the

number of sections. Finally, to pick a section, we multiply by the current datum. The `endAngle` accessor is more of the same, except with the datum offset by 1:

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

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

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

Diagonal

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

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

coordinates.

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

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

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

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

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

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

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

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

Axes

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

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

Tip

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

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

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

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

We also need a linear scale:

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

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

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

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

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

The result doesn't look good at all.

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

For now, adding this code will be sufficient:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

0 10 20 30 40 50 60 70 80 90 100
0 20 40 60 80 100
0 10 20 30 40 50 60 70 80 90 100
a e i o u

CSS

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

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

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

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

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

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

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

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

We can invoke CSS with D3 in three ways:

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

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

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

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

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

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

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

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

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

Tip

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

https://mdn.io/stroke-dasharray

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

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

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

Colors

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

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

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

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

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

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

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

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

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

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

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

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

```
  let arc = d3.svg.arc()
```

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

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

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

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

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

Finally, we draw the two color wheels:

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

Wow! That's quite a bit of code.

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

For each circle, we append a `<g>` element and move it into position. Then we use `colors.range()`

to get a full list of colors and join it as data. For every new color, we create another `<g>` element and select all the `<path>` elements it contains.

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

The result looks as follows:

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

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

Summary

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

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

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

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

Chapter 3. Making Data Useful

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

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

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

Thinking about data functionally

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

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

Note

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

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

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

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

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

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

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

Built-in array functions

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

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

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

Note

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Data functions of D3

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
      direction = (direction + 1) % 4;
      max[1] = y;
    }
  });

  return spiral;
```

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

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

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

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

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

Note

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

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

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

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

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

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

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

Put the following generators inside this function.

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

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

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

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

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

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

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

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

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

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

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

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

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

Suppose you start the development server via the following line:

```
$ npm start
```

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

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

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

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

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

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

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

Consider this code:

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

Change it to the following:

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

return results;
```

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

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

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

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

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

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

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

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

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

Note

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Loading data

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

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

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

The core

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

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

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

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

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

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

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

Convenience functions

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

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

All of these follow this pattern:

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

Note

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

Scales

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

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

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

The following figure is borrowed from Wikipedia:

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

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

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

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

Much better!

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

Ordinal scales

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

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

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

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

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

  ordinal() {
  }
}
```

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

```
  ordinal() {
    let data = d3.range(30),
      colors = d3.scale.category10(),
      points = d3.scale.ordinal().domain(data)
        .rangePoints([0, this.height], 1.0),
     bands = d3.scale.ordinal().domain(data)
      .rangeBands([0, this.width], 0.1);
  }
```

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

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

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

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

```
let data = d3.range(30),
  colors = d3.scale.category10(),
  points = d3.scale.ordinal().domain(data)
    .rangePoints([0, this.height], 1.0),
 bands = d3.scale.ordinal().domain(data)
  .rangeBands([0, this.width], 0.1);

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

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

```
        height: 10})
    .style('fill', (d) => colors(d));
```

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

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

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

The picture looks like this:

Quantitative scales

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

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

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

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

A drawing function will help us avoid code repetition:

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

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

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

Continuous range scales

We can draw using the following code:

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

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

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

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

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

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

drawSingle(line2)
  .attr('transform', `translate(0, ${this.height / 12})`)
```

```
      .style('stroke', colors(1));

    let power = d3.scale.pow().exponent(0.2).domain(extent).range([ this.height
/ 2, 0]),
      line3 = line1.y((d) => power(weierstrass(d)));

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

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

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

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

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

Now our picture looks as follows:

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

Discrete range scales

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

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

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

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

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

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

The result looks like this:

Time

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

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

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

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

Formatting

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

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

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

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

The same formatter works the opposite way as well:

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

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

Time arithmetic

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

- `d3.time.interval`: Here, `interval` can be a second, minute, hour, and so on. It returns a new time interval. For instance, `d3.time.hour` will be an hour long.
- `d3.time.interval(Date)`: This is an alias for `interval.floor()`, which rounds `Date` down so that more specific units than the interval are set to zero.
- `interval.offset(Date, step)`: This will move the date by a specified number of steps to the

correct unit.

- `interval.range(Date_start, Date_stop)`: This will return every interval between the two specified dates.
- `d3.time.intervals`: Here, an interval is seconds, minutes, hours, and so on. These are helpful aliases for `interval.range`.

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

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

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

Note

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

http://momentjs.com.

Geography

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

D3 gives us three tools for geographic data:

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

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

Getting geodata

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

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

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

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

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

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

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

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

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

Next, we transform the files with three simple commands:

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

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

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

Drawing geographically

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

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

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

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

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

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

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

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

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

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

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

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

Note

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

Next, we define a geographic projection in the constructor:

```
   let projection = d3.geo.equirectangular()
```

```
        .center([8, 56])
        .scale(800);
```

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

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

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

Now we load our data using ES2016 promises:

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

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

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

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

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

Note

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

The equivalent code in ES5 would be as follows:

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

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

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

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

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

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

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

Now we can start drawing:

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

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

We add some styling to `index.css`:

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

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

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

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

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

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

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

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

Add some more style definitions as follows:

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

.land {
     fill: #ede9c9;
     stroke: #79bcd3;
     stroke-width: 2;
}

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

.urban {
     fill: #e1c0a3;
}
```

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

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

Using geography as a base

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

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

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

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

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

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

Promise.all([p1, p2, p3, p4, p5]).then((values) => {
  let [sea, land, cultural, airports, renditions] = values;
  draw(sea, land, cultural);

  addRenditions(airports, renditions);
});
```

The function loads the two datasets and then calls (the yet-nonexistent) `addRenditions` to draw them. We use `d3.text` instead of `d3.csv` for `airports.dat` because it doesn't have a header line, so we have to parse it manually.

In `addRenditions`, we first wrangle the data into JavaScript objects—airports into a dictionary by ID—and use that to get the latitude and longitude of each destination and arrival airport:

```
function addRenditions(_airports, renditions) {
  let airports = {},
```

```
      routes;

    d3.csv.parseRows(_airports).forEach(function (airport) {
      var id = airport[4];
      airports[id] = {
        lat: airport[6],
        lon: airport[7]
      };
    });

    routes = renditions.map((v) => {
      let dep = v['Departure Airport'];
      let arr = v['Arrival Airport'];
      return {
        from: airports[dep],
        to: airports[arr]
      };
    }).filter((v) => v.to && v.from).slice(0, 50);
  }
```

We used `d3.csv.parseRows` to parse CSV files into arrays and manually turned them into dictionaries. The array indices aren't very legible, unfortunately, but they make sense when you look at the raw data:

```
1,"Goroka","Goroka","Papua New Guinea","GKA","AYGA",
    -6.081689,145.391881,5282,10,"U"
2,"Madang","Madang","Papua New Guinea","MAG","AYMD",
    -5.207083,145.7887,20,10,"U"
```

We then map each rendition flight so that we just have a dictionary of arrival and departure coordinates. We filter out any results where either `to` or `from` is missing, which are likely cases when our map function isn't able to match the airport short codes. Also, because it's a really big dataset and drawing all of it looks a bit messy, we've limited it to the first 50 objects in the array using `Array.prototype.slice`.

Next, we'll actually draw the lines, using our projection to translate the latitude and longitude coordinates into something that can fit on our screen:

```
let lines = chart.selectAll('.route')
  .data(routes)
  .enter()
    .append('line')
    .attr('x1', (d) => projection([d.from.lon, d.from.lat])[0])
    .attr('y1', (d) => projection([d.from.lon, d.from.lat])[1])
    .attr('x2', (d) => projection([d.to.lon, d.to.lat])[0])
    .attr('y2', (d) => projection([d.to.lon, d.to.lat])[1])
    .classed('route', true);
```

The routes won't show up until we style them. Add the following code to `index.css`:

```
.route {
  stroke-width: 2px;
```

```
  stroke: goldenrod;
}
```

This screenshot displays the result:

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

```
    let projection = d3.geo.equirectangular()
    .center([8, 56])
    .scale(200);
```

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

Note

We solved what is commonly referred to as the "too many markers" problem—that is, when zoomed out, data on a map looks cluttered—by simply limiting the amount of data that can be shown. This is admittedly a pretty cheap way out; a better workaround is to either cluster the map data (which wouldn't be all that easy with lines like the ones we have) or provide UI elements to toggle aspects of the dataset. We'll look at interactivity in the coming chapters; hold on to your hats!

Summary

You've made it through the chapter on data!

We really got to the core of what D3 is about—that is, data wrangling. While learning about data wrangling, you saw some interesting properties of prime numbers, learned all about loading external data, and effectively used scales to avoid calculations. We played with promises and generators along the way.

Lastly, we made a cool map to learn how simple geographic data can be once you get a hand on a good source and transform it into a better format.

Chapter 4. Defining the User Experience – Animation and Interaction

Animation is like chilli sauce. A little goes a really long way and can really help to spice up a graphic while leading the viewer through the content; if there is too much, it's all anyone will notice. Good UX — short for user experience, which is one of the computer idioms you employ throughout your projects — is more like guacamole. If it's good, it's a nice subtle touch which will improve the overall quality of your output and make everyone happy; if it is bad, it will taint everything and ruin the whole burrito.

In this chapter, we'll discuss both animation and user interaction, with an eye towards using both to improve the quality of your data visualizations. We'll also use D3's behaviors to make that map from the last chapter look awesome. Throughout the chapter, we'll discuss why or why not animation or interactivity should be used in a particular scenario.

The ability to display data creatively with D3 is one of the best reasons for using it; interaction and animation allow you to not only display data but also *explain* data. How you use UX throughout your interface design determines whether you are building an *exploratory* graphic, wherein the user is given access to all of the data and has the ability to change how it's displayed through sorting, filtering and so on, or an *explanatory* graphic, where minimal interactivity guides the user through the relevant data. In reality, you'll probably mix both approaches, but understanding which type of interaction you want to have with the reader at what point is helpful when planning your projects.

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

Animation

The first question to ask is, why would animation improve this project?

If you're making something that isn't really designed to communicate data and is just designed to trip people out at your local warehouse rave, then "because it would make it look cool" is a totally valid response. Please don't let me discourage you from running rainbow color interpolators through that spiral in the last chapter if you think it'd be fun (because, speaking from personal experience, creating crazy animated art with D3 is a highly enjoyable use of a Saturday afternoon).

If, however, you're rendering data, a bit more consideration is probably necessary. What is your data doing? If it's a value increasing over time, animating a line going upwards from left-to-right makes more sense than fading in the line all at once.

Previously, we set attributes on our various SVG objects as we wanted them to appear once the image was finally rendered. Now, we'll use animation to guide viewers through our graphic, using the narrative focus it provides as a way of helping them interpret the data we're displaying. To do this, we need to animate the relevant properties of each SVG object.

Animation with transitions

D3 transitions are one way of accomplishing this. Transitions use the familiar principle of changing a selection's attributes, except that changes are applied over time.

To slowly turn a rectangle red, we use the following line of code:

```
d3.select('rect').transition().style('fill', 'red');
```

We start a new transition with `.transition()` and then define the final state of each animated attribute. By default, every transition takes 250 milliseconds; you can change the timing with `.duration()`. New transitions are executed on all properties simultaneously unless you set a delay using `.delay()`.

Delays are handy when we want to make transitions happen in sequence. Without a delay, they are all executed at the same time, depending on the internal timer. For single objects, nested transitions are much simpler than carefully calibrated delays.

We've already used these way back in Chapter 1, *Getting Started with D3, ES2016, and Node.js* — remember how our bar chart bars grew to accommodate the data? I've reproduced the relevant section from the following `BasicBarChart.js`:

```
    this.chart.selectAll('rect')
        .data(data)
        .enter()
        .append('rect')
        .attr('class', 'bar')
        .attr('x', (d) => x(d.name))
        .attr('width', x.rangeBand())
        .attr('y', () => y(this.margin.bottom))
        .attr('height', 0)
       .transition()
        .delay((d, i) => i*200)
        .duration(800)
        .attr('y', (d) => y(d.population))
        .attr('height', (d) => this.height - y(d.population))
```

We initialize the transition, set each datum's delay to be 200ms later than the last and then run the 800ms transition, increasing the `y` value and bar height to the bar's value.

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

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

This is handy when making instant changes before or after a transition. Just keep in mind that

transitions run independently and you cannot rely on transitions outside the current callback being in any particular state.

Note

Whoops, we've actually done something silly with our animation here. Even though it was a nifty demonstration of how to stagger animations using `.delay()`, you generally shouldn't do this with ordinal scale charts. Why? Staggering gives the impression that the *x* axis is changing over time — in fact, we're not charting time series data at all in this so it makes more sense for all the bars to rise at the same time. It's really subtle but these are the sorts of things worth considering as you begin to use animation.

Interpolators

To calculate values between the initial and final states of a transition, D3 uses a type of function called an interpolator, which maps the `[0,1]` domain to a target range, which can be a color, a number, or a string. These make it easy to blend between two values, because the interpolator will return the iterations between the values supplied to it. Under the hood, scales are based on these same interpolators.

D3's built-in interpolators can interpolate between almost any two arbitrary values, most often between numbers or colors, but also between strings. This sounds odd at first but it's actually pretty useful. To let D3 pick the right interpolator for the job, we just write `d3.interpolate(a, b)` and the interpolation function is chosen depending on the type of `b`. `a` is the initial value, and `b` is the final value.

If `b` is a number, `a` will be coerced into a number and `.interpolateNumber()` will be used. You should avoid interpolating to or from a zero value because values will eventually be transformed into a string for the actual attribute and very small numbers might turn into scientific notation. CSS and HTML don't quite understand 1e-7 (the digit 1 with seven zeroes before the decimal place), so the smallest number you can safely use is 1e-6.

If `b` is a string, D3 checks whether it's a CSS color, in which case it is transformed to a proper color, just like the ones in Chapter 2, *A Primer on DOM, SVG, and CSS*. `a` is transformed into a color as well and then D3 uses `.interpolateRgb()` or a more appropriate interpolator for your color space.

Something even more amazing happens when the string is not a color. D3 can handle that too! When it encounters a string, D3 parses it for numbers, then uses `.interpolateNumber()` on each numerical piece of the string. This is useful for interpolating mixed style definitions.

For instance, to transition a font definition, you might do something like this:

```
d3.select('svg')
.append('text')
.attr({x: 100, y: 100})
.text("I'm growing!")
.transition()
.styleTween('font', () =>
 d3.interpolate('12px Helvetica', '36px Comic Sans MS'));
```

We used `.styleTween()` to manually define a transition. It is most useful when we want to define the starting value of a transition without relying on the current state. The first argument defines which style attribute to transition and the second is the interpolator.

You can use `.tween()` to do this for attributes other than the style.

Every numerical part of the string was interpolated between the starting and ending values and the string parts changed to their final state immediately. An interesting application of this is interpolating

path definitions—you can make shapes change in time. How cool is that?

Keep in mind that only strings with the same number and location of control points (numbers in the string) can be interpolated. You can't use interpolators for everything. Creating a custom interpolator is as simple as defining a function that takes a single `t` parameter and returns the start value for `t = 0` and the end value for `t = 1` and blends values for anything in between.

For example, the following code shows the `interpolateNumber` function of D3:

```
function interpolateNumber(a, b) {
  return function(t) {
    return a + t * (b - a);
  };
}
```

It's as simple as that!

You can even interpolate whole arrays and objects, which work like compound interpolators of multiple values. We'll use those soon.

Easing

Easing tweaks the behavior of interpolators by controlling the time (`t`) argument. We use this to make our animations feel more natural, to add some bounce elasticity, and so on. Mostly, we use easing to avoid the artificial feel of linear animation.

Let's make a quick comparison of the easing functions provided by D3 and see what they do.

First create the file `chapter4.js` and a new class that extends `BasicChart`. You know the drill.

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

export class chapter4 extends BasicChart {
  constructor(data) {
    super(data);

  }
}
```

Next, we need an array of easing functions and a scale to place them along the vertical axis. Put this in the constructor under `super(data)`:

```
let eases = ['linear', 'poly(4)', 'quad', 'cubic', 'sin', 'exp',
'circle', 'elastic(10, -5)', 'back(0.5)', 'bounce', 'cubic-in',
'cubic-out', 'cubic-in-out', 'cubic-out-in'],
y = d3.scale.ordinal().domain(eases).rangeBands([50, 500]);
```

You'll notice that `poly`, `elastic`, and `back` take arguments since these are just strings so we'll have to change them into real arguments manually later. The `poly` easing function is just a polynomial, so `poly(2)` is equal to **quad** and `poly(3)` is equal to **cubic**. Or, for those of us who stopped paying attention towards the end of our secondary school math, the higher the `poly` argument value, the deeper the curve — for instance, `poly(4)` (equivalent to **quart**) has a fair bit of delay at the beginning, the end or both, depending on where you set the easing (see below). The higher the number, the more dramatic the delay. Have a play with it, do what feels right.

The `elastic` easing function simulates a rubber band and the two arguments control tension. You should play with the values to get the effect you want. The `back` easing function is supposed to simulate backing into a parking space. The argument controls how much overshoot there's going to be.

The easings at the end (`cubic-in`, `cubic-out`, and so on) are functions that we create ourselves by combining the following modifiers:

- `-in`: It does nothing
- `-out`: It reverses the easing direction
- `-in-out`: It copies and mirrors the easing function from `[0, 0.5]` and `[0.5, 1]`
- `-out-in`: It copies and mirrors the easing function from `[1, 0.5]` and `[0.5, 0]`

You can add these to any easing function so play around. Now, we're going to render a bunch of circles animated using each easing:

```
eases.forEach((ease) => {
  let transition = svg.append('circle')
    .attr({cx: 130, cy: y(ease), r: y.rangeBand()/2-5})
    .transition()
    .delay(400)
    .duration(1500)
    .attr({cx: 400});
});
```

We loop over the list with an iterator that creates a new circle and uses the `y()` scale for vertical placement and `y.rangeBand()` for the circle size. In this way, we can easily add or remove examples. Transitions start with a delay of just under half a second to give us a chance to see what's going on. A duration of `1500` milliseconds and a final position of `400` should give us enough time and space to see the easing.

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

```
if (ease.indexOf('(') > -1) {
  let args = ease.match(/[0-9]+/g),
    type = ease.match(/^[a-z]+/);
  transition.ease(type, args[0], args[1]);
} else {
  transition.ease(ease);
}
```

This code checks for parentheses in the `ease` string, parses out the easing function and its arguments, and feeds them to `transition.ease()`. Without parentheses, `ease` is just the easing type.

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

```
svg.append('text')
.text(ease)
.attr({x: 10, y: y(ease)+5});
```

Replace `index.js` with the following:

```
import {chapter4} from './chapter4';
new chapter4();
```

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

Note

In order to avoid repeating the preceding four lines over and over again, I'll take it on faith that you know how to import and instantiate the classes we're creating at this point, in addition to knowing how to load the developer server. I'll therefore leave out that part from here on in.

The visualization is a cacophony of dots:

The screenshot doesn't quite showcase the animation so you should try this one in the browser. You could also take a look at the easing curves at http://easings.net/.

Easings are a nice finishing touch to put on most animations. Most things in the real world don't have constant acceleration; a good rule of thumb is to match whichever element you're animating with an easing appropriate for its size in relation to the page. In other words, small elements should generally move faster than large elements and have tighter `-in` and `-out` easings. The key thing is to think about how stuff should logically move instead of just slapping a 1-second fade-in on everything.

Timers

D3 uses timers to schedule transitions. Even an immediate transition will start after a delay of 17ms.

Far from keeping timers all to itself, D3 lets us use timers so that we can take animation beyond the two-keyframe model of transition. If you are not familiar with animation terminology, keyframes define the start or end of a smooth transition.

We use `d3.timer()` to create a timer. It takes a function, a delay, and a starting mark. After the set delay (in milliseconds) from the mark, the function will be executed repeatedly until it returns `true`. The mark should be a date converted into milliseconds since Unix timestamp (`Date.getTime()` is sufficient), or you can let D3 use `Date.now()` by default.

Let's animate the drawing of a parametric function to work just like the Spirograph toy you might have had as a kid.

We'll create a timer, let it run for a few seconds, and use the millisecond mark as the parameter for a parametric function.

Create a new class in `chapter4.js`:

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

Here's a good function from Wikipedia's article on parametric equations at http://en.wikipedia.org/wiki/Parametric_equations:

```
let position = (t) => {
  let a = 80, b = 1, c = 1, d = 80;
  return {x: Math.cos(a*t) - Math.pow(Math.cos(b*t), 3),
          y: Math.sin(c*t) - Math.pow(Math.sin(d*t), 3)};
};
```

This function returns a mathematical position based on the parameter, going from zero up. You can tweak the Spirograph by changing the `a`, `b`, `c`, and `d` variables — there are examples in the same Wikipedia article.

This function returns positions between `-2` and `2`, so we need scales to make it visible on the screen:

```
let tScale = d3.scale.linear().domain([500, 25000])
  .range([0, 2*Math.PI]),
  x = d3.scale.linear().domain([-2, 2]).range([100, this.width-100]),
  y = d3.scale.linear().domain([-2, 2]).range([this.height-100,100]);
```

`tScale` translates time into parameters for the function; `x` and `y` calculate the final position on the

image.

Now we need to define `brush` to fly around and pretend that it's drawing and also a variable to hold the `previous` position so that we can draw straight lines:

```
let brush = chart.append('circle')
  .attr({r: 4}),
  previous = position(0);
```

Next, we need to define an animation `step` function to move the brush and draw a line between the previous and current points:

```
let step = (time) => {
  if (time > tScale.domain()[1]) {
    return true;
  }

  let t = tScale(time),
    pos = position(t);

  brush.attr({cx: x(pos.x), cy: y(pos.y)});

  this.chart.append('line')
      .attr({x1: x(previous.x),
        y1: y(previous.y),
        x2: x(pos.x),
        y2: y(pos.y),
        stroke: 'steelblue',
        'stroke-width': 1.3});

  previous = pos;
};
```

The first condition stops the timer when the current value of the `time` parameter is beyond the domain of `tScale`. Then, we use `tScale()` to translate the time into our parameter and get a new position for the brush.

Then, we move the brush — there is no transition because we are performing the transition ourselves already — and draw a new steel blue line between the previous position and the current position (`pos`).

We conclude by setting a new value for the previous position.

All that's left now is to create a timer:

```
let timer = d3.timer(step, 500);
```

That's it. Half a second after a page refresh, the code will begin drawing a beautiful shape and finish 25 seconds later.

Starting out, it looks like this:

Getting the whole picture takes a while so this isn't the best way to draw Spirographs. Since we're using time as a parameter, a smoother curve (with more points) takes more time.

Animation with CSS transitions

Another way of animating things with D3 is by using CSS transitions. If you simply wish to animate a `transform` on an element (particularly if you don't need a lot of control over sequencing), CSS transitions are easier. They have the added benefit of being much better for performance due to not forcing the browser to repaint on every tick, which means that the GPU does all the work and everything runs much smoother. Lastly, you can use CSS media queries to target how an animation works on different devices.

Let's make a basic bar chart and use CSS transitions to do the heavy lifting.

Create a new class in `chapter4.js`, as shown here:

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

And fill it with the following basic bar chart code, It seems like a lot but is actually really similar to what we used way back in `chapter1.js`:

```
export class PrisonPopulationChart extends BasicChart {
  constructor(path) {
    super();

    this.margin.left = 50;
    let d3 = require('d3');

    let p = new Promise((res, rej) => {
      d3.csv(path, (err, data) => err ? rej(err) : res(data));
    });

    require('./index.css');

    this.x = d3.scale.ordinal().rangeBands([this.margin.left, this.width], 0.1);

    p.then((data) => {
      this.data = data;
      this.drawChart();
    });

    return p;
  }

  drawChart() {
    let data = this.data;
    data = data.filter((d) => d.year >= d3.min(data, (d) => d.year) && d.year <=
d3.max(data, (d)=> d.year));

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

```
    this.x.domain(data.map((d) => d.year));
    this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

    this.xAxis =
d3.svg.axis().scale(this.x).orient('bottom').tickValues(this.x.domain().filter((
d, i) => !(i % 5)));
    this.yAxis = d3.svg.axis().scale(this.y).orient('left');

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

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

    this.bars = this.chart.append('g').classed('bars', true).selectAll('rect')
        .data(data)
        .enter()
        .append('rect')
        .style('x', (d) => {
          return this.x(d.year);
        })
        .style('y', () => this.y(0))
        .style('width', this.x.rangeBand())
        .style('height', 0);

    // Run CSS animation
    setTimeout(()=> {
      this.bars.classed('bar', true)
      .style('height', (d) => this.height - this.y(+d.total) )
        .style('y', (d) => this.y(+d.total));
    }, 1000);
  }
}
```

The only thing that's a bit surprising this time is that we load the data using a promise and then return it from the constructor. This is a bit tricky — what's being returned isn't the resolved promise data but, rather, the promise object itself. The resolved data is taken care of in the constructor when we use `.then()` and attached to the class as a property. We will use this ability to return unresolved promises later on in the chapter to let our child classes know when the data has resolved.

We're using a dataset of the UK prison population from 1900 to 2015, which is available at https://github.com/aendrew/learning-d3/blob/chapter4/src/data/uk_prison_data_1900-2015.csv or in the `src/` data folder of the book's repo.

Change `index.js` to look like this:

```
import {PrisonPopulationChart} from './chapter4';
new PrisonPopulationChart('data/uk_prison_data_1900-2015.csv');
```

You might have noticed that we now only use the constructor to load in data; all the D3 work is in the `drawChart()` class method. Although the earlier chapters were pretty light on object-oriented programming concepts, we're going to start writing tighter classes now that we're dealing with user interaction. Breaking down a project into smaller functions is important for several reasons: it helps prevent you repeating yourself when constructing the stages of a user's journey and it allows each piece of the project to be more easily tested. We'll get into automated testing later but, for now, start thinking of ways you can construct your classes so that each major operation is in its own method.

Tip

As a taste of some of the cool things ES2015 classes allow you to do, did you see how I set the margin to a new value in the constructor? Our `BasicChart` class is smart enough to pick up the new value and adjust the chart created by our parent class accordingly.

After the constructor, we move onto `drawChart()`. The first part is setting up the scales and axes in the same way as before:

```
let data = this.data;

this.y = d3.scale.linear().range([this.height,
  this.margin.bottom]);
this.x.domain(data.map((d) => d.year));
this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

this.xAxis = d3.svg.axis().scale(this.x).orient('bottom')
  .tickValues(this.x.domain().filter((d, i) => !(i % 5)));

this.yAxis = d3.svg.axis().scale(this.y).orient('left');

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

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

If you hadn't noticed, we defined the `x` scale back up in the constructor:

```
this.x = d3.scale.ordinal().rangeBands([this.margin.left, this.width], 0.1);
```

This is so that we get immediate access to it from a child class that we will create later on in the chapter. It's a standard ordinal scale using `.rangeBands`. We also filter the *x* axis ticks so we get a tick every 5 years.

Note

Pop-quiz! Why did we use an ordinal scale for years instead of a time or linear scale?

While we could have used either, an ordinal scale makes the most sense because we're creating a bar chart of years — time and linear scales will render the tick on the left edge of the bar, which looks silly.

Then, we set all the bars:

```
this.bars = this.chart.append('g').classed('bars', true)
  .selectAll('rect')
    .data(data)
    .enter()
    .append('rect')
    .style('x', (d) => this.x(d.year))
    .style('y', () => this.y(0))
    .style('width', this.x.rangeBand())
    .style('height', 0);
```

This puts them into a group element with the class `.bars`. I've set the `y` value to zero as well as the height. This sets the initial state or the first keyframe. Next, we added the following right after that:

```
// Run CSS animation
setTimeout(()=> {
  this.bars.classed('bar', true)
  .style('height', (d) => this.height - this.y(d.total) )
    .style('y', (d) => this.y(d.total));
}, 1000);
```

This adds the `.bar` class to our bars and sets both its height and `y` value to their end keyframe values. Note how we've used `.style()` instead of `.attr()` — this is so that the new values are recorded in the `style` attribute tag, which means that our CSS transition will affect it. We do this separately in a `setTimeout` delay because we don't want the transition to be applied on the initial render and, without the delay, the user won't see the initial state.

Lastly, add the following to `index.css`:

```
.bar {
  transition: all 1s ease-out;
}
```

This simply tells the web browser to animate any style property changes over one second and use the `ease-out` easing.

You should now have a CSS-driven bar chart!

Note

For more on the CSS transitions API, visit this Mozilla Developer Network article: https://developer.mozilla.org/Web/CSS/CSS_Transitions/Using_CSS_transitions.

When should you use CSS transitions instead of the D3 transition API? There are two big use cases where it really makes a lot of sense: when you have to animate a lot of SVG elements and when your audience is viewing your work on a cellphone. It's often really difficult to create smooth animations using JavaScript-based interpolations on mobile devices because several things affect execution — most notably page scroll, which has made making fancy long form-style articles particularly nightmarish to get working on a mobile device in the last few years (though, luckily, this has changed as of iOS 8).

Generally, D3's transition API is easier to use, more powerful and doesn't mix animation logic with styling logic. It's also pretty fast — you shouldn't be afraid to use it by default, you can switch particular features to CSS transitions if you feel the performance benefit would improve the user experience.

Lastly, it makes a lot of sense to animate user interface interactivity using CSS transitions as UI elements tend to be simpler than charts and their animations shouldn't impact on performance when animating the SVG graphic.

Note

What other ways are there to animate SVG? So glad you asked!

In the beginning, there was **SMIL** or **Synchronized Multimedia Integration Language**. To use SMIL, you use `<animate>` tags, which you put in your SVG markup. If this sounds gross already, it is. Luckily, Chrome 45 depreciated it, which means you never have to worry about learning it. The morbidly curious can visit: https://mdn.io/SVG_animation_with_SMIL.

In the future, an alternative to CSS transitions will be the Web Animations API. It's currently not supported by anything from Microsoft or Apple but it's been in Chrome since version 36 and Firefox since version 41. It has the *nicest* syntax, is all based on JavaScript and works well with D3. Here's an example:

```
d3.selectAll('.bar').each(function(d, i){
  this.animate([
    {transform: `translate(${x(i)}, ${y(d)})`}
  ],{
    duration: 1000,
    iterations: 5,
    delay: 100
  });
});
```

Alas, as always, web development is a toy chest full of things you can't reliably use *just quite yet*. If you want to play with it while it's still being standardized, check out the fantastic polyfill at https://github.com/web-animations/web-animations-js that enables the use of web animations in most modern browsers. I'll hopefully be able to write more about the Web Animations API in the next edition of this book!

Interacting with the user

This is it. This is where all of the UX tidbits that I've been dropping throughout the chapter and all the ES2015 ideas that you've been learning come together — let's make a simple explanatory graphic that uses interaction to walk the viewer through data.

The first step in any visualization involving user interactivity is to plan exactly what you want the visualization to do, how you want your viewers to interact with it, and what you want to say about the data. What is the data's story? What's the best way to tell it?

We have the numerical product of over a century of incarceration in a western country in the prison population dataset. There are many ways we can look at this data. We can look at how the prison population has risen versus the overall population growth or we can look at how the prison population has risen or fallen in relation to known historical events. Often, you'll need more than one chart — for instance, when I used this data in a project for *The Times*, the piece had no less than five charts and one map, with the reader being walked through each graphic in sequence. This is where we start to get into actual data journalism territory, which is far beyond the scope of this short section. Suffice to say, however, that it helps to write down these things in either bullet point or paragraph form before you start writing any code. The real work is often done long before the first line of JavaScript is ever written.

In this particular instance, because we have a century of data, we're going to look at a few notable historical points. The graphic will have five states which will be navigated through a series of five buttons:

1. Initial view — years 1900 to 2015. This provides a general overview of how the prison population has risen over time.
2. Zoom 1900 to 1930. Highlights 1914–1918. The text explains how the population rose due to the end of World War I.
3. Zoom 1930 to 1960. Highlights 1939-1945. The text explains how the population rose after World War II.
4. Zoom 1960 to 1990. Discusses the rise of the consumer society.
5. Zoom 1990 to 2015. Highlights 1993 and explains the sharp rise after the murder of James Bulger.

We're keeping the user interface deliberately simple but remember that simpler is often better, particularly when building for an audience on mobile devices (sliders are much harder to use on touch devices than buttons, for instance).

Basic interaction

Much like elsewhere in JavaScript land, the principle for interaction is simple—attach an event listener to an element and do something when it's triggered. We add and remove listeners to and from selections with the `.on()` method, an event type (for instance, `click`), and a listener function that is executed when the event is triggered.

We can set a capture flag which ensures our listener is called first and all other listeners wait for our listener to finish. Events bubbling up from children elements will not trigger our listener.

You can rely on the fact there will only ever be a single listener for a particular event on an element because old listeners for the same event are removed when new ones are added. This is very useful when trying to eliminate unpredictable behavior.

Just like other functions acting on element selections, event listeners get the current datum and index and set the `this` context to the DOM element. The global `d3.event` lets you access the actual event object.

We're going to create a new class that extends our last chart:

```
export class InteractivePrisonPopulationChart extends PrisonPopulationChart {
  constructor(path) {
    let p = super(path);
    this.scenes = require('./data/prison_scenes.json');
    this.scenes.forEach((v, i) => v.cb = this['loadScene' + i].bind(this));

    p.then(() => this.addUIElements());
  }
}
```

Creating a scaffold like this is really useful when starting a more involved project like this. Additionally, I have pulled out the chapter data from a JSON file and assigned a callback function to each scene. You can often get away with having just one `draw` function that does something clever to load each scene but it's often easier to start out thinking in terms of discrete segments so that you don't end up with a single function with a ton of conditional logic.

Tip

Pay attention to the `.bind(this)` call in that ugly `forEach` loop I wrote. Without that, we wouldn't have access to our class methods via `this` in the scene functions!

A good rule of thumb when writing ES2015 classes is: when in doubt and throwing a lot of `TypeErrors`, check whether `this` is what you think it should be!

We gain the advantage of having all that work we just did relegated to the `super()` call because we're extending `PrisonPopulationChart`, in effect concentrating all the rendering code in a separate class. You don't necessarily need to create a totally separate class for interaction; I did it

here to simplify everything. Also, note how we assign whatever the parent constructor returns to a variable? This is because we returned a promise in the parent constructor, which allows us to use `.then()` to ensure that the data is loaded and attached to `this.data` before we continue.

Let's start by giving ourselves some space underneath the chart. Add the following to the end of the constructor:

```
    this.height = window.innerHeight / 2;
    this.chart.attr('height', this.height);
    this.svg.attr('height', this.height + 50);
    this.margin.right = 10;
    this.margin.bottom = 10;
```

This gives us roughly half of the window to work with. You'll want to experiment a bit to see what the best combination is or possibly set the element size using CSS media queries.

Next, let's create our UI elements — in this case, five buttons. Add a function called `addUIElements`, shown as follows:

```
addUIElements() {
  this.buttons = d3.select('#chart')
    .append('div')
    .classed('buttons', true)
      .selectAll('.button')
      .data(this.scenes).enter()
      .append('button')
      .classed('scene', true)
      .text((d) => d.label)
      .on('click', d => d.cb())
      .on('touchstart', d => d.cb());

  this.words = d3.select('#chart').append('div');
  this.words.classed('words', true);
}
```

There is nothing new here — create a new button for each element in the chapters array and run its callback function when any button is clicked or tapped. We also drop a plain ol' `div` into the chart area, which is where we'll put all of our text describing each scene.

Note

Haven't run into `touchstart` before? Think of it as the `mousedown` event of touch. Other useful touch events are `touchmove`, `touchend`, `touchcancel`, and `tap`. Mozilla's documentation explains touch events in more detail at https://developer.mozilla.org/Web/Guide/API/DOM/Events/Touch_events.

We need a function to clear selected bars — let's do something a little bit different in terms of sequencing animation and use promises. Although D3 doesn't use promises for animation natively (or for much in general, really), it's an idiom becoming widely used throughout the JavaScript world, particularly by the Angular 2 community. Add the following method to your class:

```
  clearSelected() {
    return new Promise((res, rej) => {
      d3.selectAll('.selected').classed('selected', false);
      res();
    });
  }
```

This returns a new promise, which resolves after we've removed the `.selected` class from all the bar elements.

We also need a method to select specific bars:

```
  selectBars(years) {
    this.bars.filter((d) => years.indexOf(
     Number(d.year)) > -1).classed('selected', true);
  }
```

We now need to create our chart's update function. Let's make it return a promise like our animation functions:

```
updateChart(data = this.data) {
  return new Promise((res, rej) => {
    let bars = this.chart.selectAll('.bar').data(data);

    this.x.domain(data.map((d) => d.year));
    this.y.domain([0, d3.max(data, (d) => Number(d.total))]);

    this.chart.selectAll('.axis.x').call(
      d3.svg.axis().scale(this.x).orient('bottom')
     .tickValues(this.x.domain().filter((d, i) => !(i % 5))));
    this.chart.selectAll('.axis.y')
      .call(this.yAxis);

    // Update
    bars.style('x', (d) => this.x(d.year))
      .style('width', this.x.rangeBand())
      .style('height', (d) => this.height - this.y(+d.total) )
      .style('y', (d) => this.y(+d.total))

    // Add
    bars.enter()
      .append('rect')
      .style('x', (d) => this.x(+d.year))
      .style('width', this.x.rangeBand())
      .style('height', (d) => this.height - this.y(+d.total) )
      .style('y', (d) => this.y(+d.total))
      .classed('bar', true);

    // Remove
    bars.exit().remove();

    res();
  });
```

```
}
```

We do the standard D3 update, add and remove routine, setting the scales to the selected data. Finally, we resolve the promise by calling `res()` at the end. Note how we take an argument which defaults to the class `data` property.

Note

Default argument values are a new feature in ES2016! This is such a nice addition to the language, as somebody who has written so much existence checking logic for basic functions it'd make you cry.

Let's look at that again:

```
updateChart(data = this.data)
```

This means the `data` argument will equal the class' internal `data` property if no argument is supplied. The next time that you feel yourself reaching for `typeof argumentVar !== 'undefined'`, give default arguments a try.

Lastly, let's set up our states. The first one is easy:

```
  loadScene0() {
    this.clearSelected().then(() => this.updateChart());
    this.words.html('');
  }
```

We clear the selected bars and then update the chart. We set `words` to be an empty string. We need to clear both the selected bars and the words on the first scene even if they originate in that way because we're allowing the viewer to navigate the scenes in a non-linear order — they can always come back to the first scene, which means we have to clear anything created by another scene:

```
loadScene1() {
  let scene = this.scenes[1];
  this.clearSelected().then(() => {
    this.updateChart(this.data.filter((d) =>
      d3.range(scene.domain[0], scene.domain[1]).indexOf(Number(d.year)) > -1))
        .then(() => this.selectBars(d3.range(1914, 1918)));
  });
  this.words.html(scene.copy);
}
```

This is somewhat more interesting. We do the same thing as we did in the first scene but, in the `then()` callback, we create a range from the domain given in the scene array. This gives us an array of the years with which we can filter the bars. Lastly, we set the text in the interactive dialog (helpfully written by my colleague, Sam Joiner) to form a meaningful sentence explaining the change.

Note

Copy is what people in the newspaper and advertising businesses call text content. I don't mean it in

the duplicate sense here!

The next bit is rather repetitive:

```
loadScene2() {
  let scene = this.scenes[2];
  this.clearSelected().then(
    () => {
      this.updateChart(this.data.filter((d) => d3.range(scene.domain[0],
scene.domain[1]).indexOf(Number(d.year)) > -1))
        .then(() => this.selectBars(d3.range(1939, 1945)));
  });
  this.words.html(scene.copy);
}

loadScene3() {
  let scene = this.scenes[3];
  this.clearSelected().then(
    () => this.updateChart(this.data.filter((d) => d3.range(scene.domain[0],
scene.domain[1]).indexOf(Number(d.year)) > -1))
  );
  this.words.html(scene.copy);
}

loadScene4() {
  let scene = this.scenes[4];
  this.clearSelected().then(
    () => {
      this.updateChart(this.data.filter((d) => d3.range(scene.domain[0],
scene.domain[1]).indexOf(Number(d.year)) > -1))
        .then(() => this.selectBars([1993]));
    }
  );
  this.words.text(scene.copy);
}
```

And there you have it — your first interactive data visualization!

Prison numbers fall, and then – as men return from the battlefields of the First World War – start to rise to new heights.

Behaviors

In the last section, we created an *explanatory* graphic that used interaction to guide the user through the data. Often, however, the goal is just to make a dataset interactive and give the user some way of manipulating it, in other words, an *exploratory* graphic.

D3's behaviors save a boatload of time in setting up the more complex interactions in a chart. Additionally, they're designed to handle differences in input devices so you only have to implement a behavior once to have it work both with a mouse and on touch devices. The two currently supported behaviors are drag and zoom, both of which will get you pretty far.

Drag

Instead of having the user click buttons in the last example, what if we just let them drag the chart area to see the UK's prison population increase? It involves a bit more work from the user but it also gives them the ability to navigate freely through the chart — which may be desirable in some circumstances.

Let's extend the last chart we created and override the `.addUIElements` method. Create a child class that extends `InteractivePrisonPopulationChart` and set it up as shown below, thus overriding the parent `.addUIElements` method and avoiding a bunch of unnecessary buttons.

```
export class DraggableInteractivePrisonChart extends
InteractivePrisonPopulationChart {
  constructor(path) {
    let p = super(path);
    this.x.rangeBands([this.margin.left, this.width * 4]);
  }
  addUIElements() {}
}
```

The first thing that we need to do is create a hit box — we want dragging on the bars area to result in the bars being dragged. Alas, `g` container elements aren't clickable so we need to get the dimensions of `g.bars`, placing an invisible `rect` element of that size on top of it. Yet further alas, the dimensions of `g.bars` won't make any sense until the CSS transition has finished. Luckily, we can listen to the `transitionend` event to see if this has occurred. Add this to the `addUIElements` function, as shown here:

```
  let bars = d3.select('.bars').on('transitionend', ()=> {
    let dragContainer = this.chart.append('rect')
      .classed('bar-container', true)
      .attr('width', bars.node().getBBox().width)
      .attr('height', bars.node().getBBox().height)
      .attr('x', 0)
      .attr('y', 0)
      .attr('fill-opacity', 0);

});
```

`SVGElement.getBBox()` is a tremendously useful function that gives you the `x`, `y`, height and width of a particular element. Note, however, that you can't use this on D3 selections — only SVG elements! That's why we get the underlying element out of the selection by using `selection.node()`.

Now we need to set up the drag behavior. Still inside the `transitionend` callback, add the following:

```
let drag = d3.behavior.drag().on('drag', () => {
  let barsTransform = d3.transform(bars.attr('transform'));
  let xAxisTransform = d3.transform(
    d3.select('.axis.x').attr('transform'));
  bars.attr('transform', `translate(${barsTransform.translate[0] + d3.event.dx},
```

```
0)`);
  d3.select('.axis.x').attr('transform',
    `translate(${xAxisTransform.translate[0] + d3.event.dx},
${xAxisTransform.translate[1]})`);
});
```

Firstly, we get the current transform values of the *x* axis and the bars themselves. `d3.transform()` is a very helpful function for getting a transform matrix into a useful object form so we use that twice. We then translate both the axes and bars their current translation distance, plus the distance the user drags across our hit box, provided by `d3.event.x`. We also provide the `y` translate of the axis since we set that way back up in our parent class.

Finally, after our call above but still inside the `transitionend` event callback, add the following to instantiate the drag behavior on our container.

```
dragContainer.call(drag);
```

Now you can drag!

Note

You may have noticed that the bars drag behind the *x* axis which looks a bit unsightly. You can't add backgrounds to `g` elements as you can with `div` so you'll have to append a white `rect` element to the axis. The technique for getting the `rect` width and height is the same as it is for the hit box — I'll leave fixing this as an exercise for you.

Zoom

Despite the name, the zoom behavior lets you do more than just zoom—you can also pan! Like the drag behavior, zoom automatically handles both mouse and touch events and then triggers the higher-level zoom event. Yes, this means pinch-to-zoom works! That's pretty awesome, if you ask me.

Remember that map from Chapter 3, *Making Data Useful*, the one with the rendition flights?

Let's commit a crime against computational efficiency and make it zoom and pan.

I am warning you that this will be very rudimentary and painfully slow. This is not how you'd make a real explorable map, just an example to let us play with zooming. In real life, you would use tiling, progressive detailing, and other tricks. You also wouldn't write everything in the constructor, as we did in this example.

Tip

After even one example organizing all the functions into class methods, doesn't this already feel atrociously messy? Hopefully, you can see why it's so much better to organize intelligently, plan and organize your classes — you can always revisit a project and being able to pick your code back up and understand it as quickly as possible is both enormously important and very much facilitated by writing clean and extensible code.

Let's go back to the map. Jump to the end of the `GeoDemo draw` function in `chapter3.js` and add a call to `zoomable`; we'll define this next.

```
zoomable();
```

While you're in `draw`, turn off the river and cities layers to help with performance.

`zoomable` sets up the behavior on the chart so put this at the end of the constructor. Next, we'll define what the behavior actually is:

```
function zoomable() {
  chart.call(
    d3.behavior.zoom()
    .translate(projection.translate())
    .scale(projection.scale())
    .on('zoom', () => onzoom())
  );
}
```

We defined a zoom behavior with `d3.behavior.zoom()` and immediately called it on the whole image.

We set the current `.translate()` vector and `.scale()` to whatever the projection was using. The `zoom` event calls our `onzoom` function.

Let's define it as follows:

```
function onzoom() {
  projection
    .translate(d3.event.translate)
    .scale(d3.event.scale);

  d3.selectAll('path')
  .attr('d', d3.geo.path().projection(projection));

}
```

Firstly, we told our projection that the new translation vector was in `d3.event.translate`.

The translation vector pans the map with a transformation as in Chapter 2, *A Primer on DOM, SVG, and CSS*. `d3.event.scale` is just a number the projection uses to scale itself, effectively zooming the map.

Next, we recalculate all the routes using the changed projection:

```
  d3.selectAll('line.route')
    .attr('x1', (d) => projection([d.from.lon, d.from.lat])[0])
    .attr('y1', (d) => projection([d.from.lon, d.from.lat])[1])
    .attr('x2', (d) => projection([d.to.lon, d.to.lat])[0])
    .attr('y2', (d) => projection([d.to.lon, d.to.lat])[1]);
```

The positioning function is exactly the same as in `addRenditions` because geographic projections handle panning out of the box. The thickness stays more or less the same throughout.

You can now explore the world!

Have patience, though, it's *reaaaaal* slow. Redrawing everything on every move does that. For a more performant and zoomable map, we'd have to use data with less detail when zoomed out, draw a sensible number of lines, and possibly avoid drawing parts of the map that fall out of the image anyway. Another way in which we could make it more performant is by transforming the entire map area instead of reprojecting on each event. This not only requires less computing but it also allows the GPU to do some of the work. Let's give that a shot!

Replace the entirety of `zoomable` with the following snippet:

```
function zoomable() {
  chart.call(d3.behavior.zoom()
    .center([chart.attr('width') / 2, chart.attr('height') / 2])
    .scale(projection.scale())
    .on('zoom', () => onzoom()));
```

```
}
```

All we've done here is replace the translate line with a line telling it to always zoom from the center of the chart — it works without doing this but it's a bit janky. Next, replace `onzoom` with the following:

```
function onzoom() {
  let scaleFactor = d3.event.scale / projection.scale();
  chart.attr('transform', `translate(${d3.event.translate})
scale(${scaleFactor})`);
  d3.selectAll('line.route').each(function() {
    d3.select(this).style('stroke-width', `${ 2 / scaleFactor}px`);
  });
}
```

We're doing two things here: we're getting the real scale factor by dividing the scale given by the zoom event by the initial scale as it was defined for the projection (in this case, 150). We then use this to scale and translate the `g` element holding all of our geometry. We also divide the initial stroke width (2px, set in `index.css`) by the scaling factor to prevent the route lines from getting big and blocky as we zoom in.

That's miles better, isn't it? D3 gives you a lot of freedom in how you implement things; understanding which is the best method for any particular project comes from practice and knowing your audience.

Brushes

Brushes are similar to zoom and drag and are a simple way to create complex behavior— they enable users to select a part of the canvas.

Strangely, they aren't considered as a behavior but fall under the `.svg` namespace, perhaps because they are mostly meant for visual effects.

To create a new brush, we call `d3.svg.brush()` and define its `x` and `y` scales using `.x()` and `.y()`. We can also define a bounding rectangle.

Time for an example!

We're going back yet again to our all-singing, all-dancing prison population graph. This is the last example in which I will use it, I promise. We are going to let the user zoom in on a group of bars by selecting them with a brush, zooming out on a right-click.

Begin by creating a new class that extends `InteractivePrisonPopulationChart`, like so:

```
export class SelectableInteractivePrisonChart extends
InteractivePrisonPopulationChart {
  constructor(path) {
    super(path);
  }

  addUIElements() {}
  brushstart() {}
  brush() {}
  brushend() {}
  rightclick() {}
}
```

This is almost the same setup that we used for the drag example. Add the following to `addUIElements`:

```
    this.chart.append('g')
      .classed('brush', true)
      .call(d3.svg.brush().x(this.x).y(this.y)
      .on('brushstart', this.brushstart.bind(this))
      .on('brush', this.brushmove.bind(this))
      .on('brushend', this.brushend.bind(this)));
```

We made a new grouping element for the brush and called a freshly constructed `d3.svg.brush()` with both scales defined. The `.brush` class helps with styling. We also `bind` the local context to each event callback so that we still have access to our class methods.

Next, we define listeners for the `brushstart`, `brush`, and `brushend` events as part of our class. We're not doing anything with `brushstart` so let's skip straight to `brushmove`:

```
  brushmove() {
    let e = d3.event.target.extent();
    d3.selectAll('.bar').classed('selected', (d) =>
      e[0][0] <= this.x(d.year)
      && this.x(d.year) <= e[1][0]
    );
  }
```

`brushmove` is where the real magic happens. Firstly, we find the selection's boundaries by using `d3.event.target.extent()`.

`d3.event.target` returns the current brush and `.extent()` returns a set of two points — the upper-left and bottom-right corners.

Then, we go through all the bars and turn the `.selected` class on or off, depending on whether the bar's position lies within the bounding box.

Note

Note that the values returned by `extent` are wholly reliant on the type of scale you provide to it — if you give it a linear scale, you'll compare the unscaled datum value against the extent value. In an ordinal scale like we have here, we have to scale the datum value before comparing.

Next, we define what happens when the mouse button is released:

```
brushend() {
  let selected = d3.selectAll('.selected');

  // Clear brush object
  d3.event.target.clear();
  d3.select('g.brush').call(d3.event.target);

  // Zoom to selection
  let first = selected[0][0];
  let last = selected[0][selected.size() - 1]
  let startYear = d3.select(first).data()[0].year;
  let endYear = d3.select(last).data()[0].year;
  this.clearSelected().then(() => {
    this.updateChart(this.data.filter((d) =>
      d3.range(startYear, endYear).indexOf(Number(d.year)) > -1));
  });

  let hitbox = this.svg
    .append('rect')
    .classed('hitbox', true)
    .attr('width', this.svg.attr('width'))
    .attr('height', this.svg.attr('height'))
    .attr('fill-opacity', 0);

  hitbox.on('contextmenu', this.rightclick.bind(this));
}
```

Quite a lot happens here. Firstly, we clear the brush overlay, then we figure out the first and last element selected by the brush. From that, we get the start and end years, which we supply to `d3.range`, redrawing the chart just like we did in `InteractivePrisonPopulationChart`. Lastly, we add a hit box, which we'll use to listen to the `contextmenu` event (`contextmenu` being the right-click variant of the click event).

Our HTML needs some more styling definitions, like this:

```
.brush .extent {
  stroke: #fff;
  fill-opacity: .125;
  shape-rendering: crispEdges;
}

.bar.selected {
  stroke: black;
}
```

In this case, we're still using the CSS transitions from the first example. It's preferable to use CSS transitions rather than what D3 can do in this case — brushes sometimes have problems with D3 transitions and change properties immediately.

When you select some elements, the image will look like this:

Summary

Wow, what a fun chapter!

You've made things jump around the page, almost killed your computer and patience with a zoomable map, and made one supremely awesome bar graph. Well done!

In this chapter, we've animated with transitions, interpolators and timers, and then we learned how to do some of that with CSS. We then learned the difference between explanatory and exploratory visualizations, and used interactivity to create the former. We then made some exploratory visualizations by using behaviors with some of our previous projects. Things are starting to look pretty snazzy, aren't they?

In the next chapter, we'll be looking at creating a whole boatload of really pretty charts using D3's black magic — layouts. Combining the skills you've learned in this chapter and the next one will mean that you're able to produce some truly fantastic charts. I hope that you're ready — this is where stuff starts getting really cool!

Chapter 5. Layouts – D3's Black Magic

Most of us look at the Internet for inspiration and code samples. You find something that looks great, you look at the code, and your eyes glaze over. It doesn't make any sense.

The usual culprit is D3's reliance on layouts for anything remotely complicated. The black magic of taking some data, calling a function, and—voilà—visualization! This elegance makes layouts look deceptively difficult, but they make things a lot easier when you get the hang of them.

In this chapter, we'll go in, guns blazing, with everything you've learned so far to create 11—count 'em! 11!—visualizations of the same dataset.

What are layouts and why should you care?

D3 layouts are modules that transform data into drawing rules. The simplest layout might only transform an array of objects into coordinates, like a scale.

But we usually use layouts for more complex visualizations, drawing a force-directed graph or a tree for instance. In these cases, layouts help us separate calculating coordinates from putting pixels on a page screen. This not only makes our code cleaner but also lets us reuse the same layouts for vastly different visualizations.

Theory is boring. Let's dig in.

Built-in layouts

By default, D3 comes with 12 built-in layouts that cover most common visualizations. They can be split roughly into **normal** and **hierarchical** layouts. Normal layouts represent data in a flat hierarchy, while hierarchical layouts generally present data in a tree-like structure. The normal layouts are as follows:

- Histogram
- Pie
- Stack
- Chord
- Force

The hierarchical layouts are as follows:

- Partition
- Tree
- Cluster
- Pack
- Tree map

To see how they behave, we're going to make an example for each type. We'll start with the humble pie chart and histogram and then progress to force-directed graphs and fancy trees. We'll be using the same dataset for all examples so that we can get a feel of how different presentations affect the perception of data.

We're getting pretty good at this by now, so we're going to make these examples particularly magnificent. That's going to create a lot of code, so every time we come up with something reusable, we'll put it in a **helpers.js** file as a function. We'll be exporting this as a big ol' bucket of functions instead of a class this time, mainly because we will have to deal with much less weirdness due to this changing context.

Let's create an empty `helpers.js` file. We'll be adding functions to this file willy-willy and then importing them under the same namespace elsewhere. We could have done this in a bunch of different ways, for instance, exporting an object with a bunch of different methods, or creating a class or a factory function. However, doing it this way means that we can either selectively choose functions from the module without importing them all, or import all of them during development with a single statement. For now, just add this line at the top:

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

Meanwhile, let's once again extend `BasicChart`, creating a class that will create all our different representations of the same chart. Add the following code to a new file called `chapter5.js`:

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

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

export class PoliticalDonorChart extends BasicChart {
  constructor(chartType, ...args) {
    super();
    require('./chapter5.css');

    let p = new Promise((res, rej) => {
      d3.csv('data/uk_political_donors.csv',
        (err, data) => err ? rej(err) : res(data));
    });


    p.then((data) => {
      this.data = data;
      this[chartType].call(this, ...args);
    });

    return p;
  }
}
```

You'll notice that we're importing everything from `helpers.js` under the `helpers` namespace. As we add helper functions, they'll automatically be added to the `helpers` object we're importing from that file. Otherwise, there is nothing unusual:

1. First, we load the CSS file that we'll make in a moment.
2. Then we call `super`.
3. We create a new `Promise`.
4. We have D3 load and parse the CSV data.
5. Then we resolve the promise if everything is okay, calling the method specified by the constructor's first argument.

The rest of the constructor's arguments are then passed as arguments to the method we're calling.

Note

The `...args` bit in the constructor's arguments is a new feature in ES2016, called the **rest parameter**. It collects every argument after the ones specifically defined as an array. We then use another new ES2016 feature, called the **spread operator**, in `this[chartType].call(this, ...args)` to destructure the array into its individual values. This lets us add as many arguments as we want to each chart method we're writing.

The dataset

The dataset that we are going to use in all of these examples is a list of donations to various political entities of the UK: who received the donation, the amount, whom it was from, the type of donor, and the date it was made. Data like this is important, as it can indicate individual actors influencing the political process. This alone might not be enough for a story, but at the very least, it can point you towards groups or individuals worth pursuing further. It's in the book's repository at `src/data/uk_political_donors.csv`, or you can get it from https://github.com/leilahaddou/graph-data/blob/master/pef.csv.

Note

It's worth noting that I got this dataset from Leila Haddou's totally awesome tutorial on Neo4j, a network graph database. If you like the force-directed graph later on in this chapter, Neo4j might just totally blow your mind. Give it a shot! It is at http://leilahaddou.github.io/neo4j-tutorial.html.

Normal layouts

Time to draw! As mentioned, we'll begin with normal layouts. They display data in a flat hierarchy.

Using the histogram layout

We are going to use the `histogram` layout to create a bar chart of the number of donations received. The layout itself will handle everything from collecting values in bins to calculating heights, widths, and positions of the bars.

Histograms usually represent a probability distribution over a continuous numerical domain, but the names of donation recipients are ordinal. To bend the `histogram` layout to our will, we will have to turn names into numbers—we'll use a scale.

Since it feels like this could be useful in other examples, we'll put the code in `helpers.js`:

```
export function uniques(data, name) {
  let uniques = [];
  data.forEach((d) => {
    if (uniques.indexOf(name(d)) < 0) {
      uniques.push(name(d));
    }
  });
  return uniques;
}

export function nameId(data, name) {
  let uniqueNames = uniques(data, name);
  return d3.scale.ordinal()
  .domain(uniqueNames)
  .range(d3.range(uniqueNames.length));
}
```

These are two simple functions:

- `uniques`: This goes through the data and returns a list of unique names. We help it with the `name` accessor.
- `nameId`: This creates an ordinal scale that we'll be using to convert names into numbers. Expect to see both of these a lot in this chapter.

Now we can tell the histogram how to handle our data with `nameId`. Add the following method to your `PoliticalDonorChart` class:

```
  histogram() {
    let data = this.data;

    let nameId = helpers.nameId(data, (d) => d.EntityName);
    let histogram = d3.layout.histogram()
                        .bins(nameId.range())
                        .value((d) => nameId(d.EntityName))(data);

    this.margin = {top : 10, right : 40, bottom : 100, left : 50};
  }
```

Using `d3.layout.histogram()`, we create a new histogram and use `.bins()` to define the upper threshold for each bin. Given `[1,2,3]`, values under `1` go into the first bin, values between `1` and `2` into the second, and so on.

The `.value()` accessor tells the histogram how to find values in our dataset.

Another way to specify bins is by specifying the number of bins you want and letting the histogram uniformly divide a continuous numerical input domain into bins. For such domains, you can even make probability histograms by setting `.frequency()` to `false`. You can limit the range of considered bins with `.range()`.

Finally, we use the layout as a function on our data to get an array of objects with the following schema:

```
{
    0: {
        DonorName: String, // e.g. "Ændrew Rininsland"
        DonorStatus: String, // e.g. "Individual"
        ECRef: String, // e.g. "c1234567"
        EntityName: String, // e.g. "Green Party"
        ReceivedDate: Date, // e.g. "27/01/15" (dd/mm/yy)
        Value: String // e.g. "£1,234"
    },
    dx: Number,
    x: Number,
    y: Number
}
```

Note

It's worth reiterating that we define the histogram and then immediately run it on our dataset. Look at the following line:

```
let histogram = d3.layout.histogram()
  .bins(nameId.range())
  .value((d) => nameId(d.EntityName))(data);
```

It can also be written like this:

```
let histogramLayout = d3.layout.histogram()
  .bins(nameId.range())
  .value((d) => nameId(d.EntityName));
let histogram = histogramLayout(data);
```

We just avoid the temporary variable for something we use only once here by immediately executing the layout and then storing the output in a variable.

The bin width is in the `dx` property, `x` is the horizontal position, and `y` is the height. We access elements in bins with normal array functions. Note that they're all strings, even the amount and date. We'll have to handle this in our code using `d3.format`.

Using this data to draw a bar chart should be easy by now. We'll define a scale for each dimension, label both axes, and place some rectangles for bars.

Next, add two scales:

```
let x = d3.scale.linear()
  .domain([0, d3.max(histogram, (d) => d.x)])
  .range([this.margin.left, this.width-this.margin.right]);

let y = d3.scale.log().domain([1, d3.max(histogram, (d) => d.y)])
  .range([ this.height - this.margin.bottom, this.margin.top ]);
```

The use of a logarithmic scale for the vertical axis will make it such that the number of donations received by the biggest entities doesn't totally flatten everything else.

Next, put a vertical axis on the left:

```
let yAxis = d3.svg.axis()
     .scale(y)
     .tickFormat(d3.format('f'))
     .orient('left');

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

We create a grouping element (the `'g'`) for every bar and its label:

```
    let bar = this.chart.selectAll('.bar')
      .data(histogram)
      .enter()
      .append('g')
      .classed('bar', true)
      .attr('transform', (d) => `translate(${x(d.x)},${y(d.y)})`);
```

Moving the group into position, as shown in the following code, means less work when positioning the bar and its label:

```
    bar.append('rect')
      .attr({
        x: 1,
        width: x(histogram[0].dx) - this.margin.left - 1,
        height: (d) => this.height - this.margin.bottom - y(d.y)
      })
      .classed('histogram-bar', true);
```

Because the group is in place, we can put the bar a pixel from the group's edge. All bars will be `histogram[0].dx` units wide, and we'll calculate heights using the `y` position of each datum and the total graph height. Lastly, we create the labels:

```
    bar.append('text')
```

```
      .text((d) => d[0].EntityName)
      .attr({
        transform: (d) => `translate(0, ${this.height - this.margin.bottom -
y(d.y) + 7}) rotate(60)`
      });
```

We move labels to the bottom of the graph, rotate them by 60 degrees to avoid overlap, and set their text to the `EntityName` property of the datum.

Now create a new file, `chapter5.css`, and add this code:

```
.axis path, .axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
.axis text {
  font-size: 0.75em;
}
rect.histogram-bar {
  fill: steelblue;
  shape-rendering: crispEdges;
}
.bar text {
  font-size: 0.4em;
}
```

Don't forget to `require` the CSS file in your JavaScript. Replace `index.js` with the following line:

```
require('./index.css');
import {PoliticalDonorChart} from './chapter5';
new PoliticalDonorChart('histogram');
```

Our bar chart looks like this:

Hmm… It seems that the incumbent party got the most donations. This is totally unsurprising. Let's filter for only Members of Parliament (MPs, for those outside the Commonwealth).

Replace the first line of your `histogram` method with the following:

```
let data = this.data.filter((d) => d.EntityName.match(' MP'));
```

Much better! Note that we start at `1` instead of `0` as we're using a logarithmic scale. This means that MPs with only one donor appear as if they have none. You'd want to explain that to your audience if sticking to the log scale in such an instance, or more likely filter out individuals with only one donation:

Ed Miliband, then-leader of the opposition Labour Party, had the most donations in this particular set of data

Baking a fresh 'n' delicious pie chart

The preceding bar chart reveals that Ed Miliband received the most donations during this period. Let's find out who's making him so popular.

We are going to use the pie chart layout to cut the donations to *The Rt Hon Edward Miliband MP* into slices, showing how many donations he got from each donor. After filtering the dataset for donations going to Miliband, we have to categorize the entries by givers, and finally we feed them into the pie chart layout to generate a pie chart.

We can use the `histogram` layout to put data into bins depending on the `DonorName` property. Let's add a function to `helpers.js`:

```
export function binPerName (data, name) {
   let nameIds = nameId(data, name);
   let histogram = d3.layout.histogram()
    .bins(nameIds.range())
    .value((d) => nameIds(name(d)) );
   return histogram(data);
}
```

Similar to the `uniques` and `nameId` functions, `binPerName` takes the data and a `name` accessor and returns histogram data.

Now create a `pie` method in your `PoliticalDonorChart` class:

```
  pie(name) {
    let filtered = this.data.filter((d) => d.EntityName === name);
    let perDonor = helpers.binPerName(filtered,
      (d) => d.DonorName);
  }
```

Entries in the `perDonor` variable will tell us exactly how many donations were received by the name specified by the function argument.

To bake a pie, we call the `pie` layout and give it a `value` accessor:

```
    let pie = d3.layout.pie()
      .value((d) => d.length)(perDonor);
```

The `pie` layout is now full of slice objects, each holding the `startAngle` and `endAngle` values and the original value.

The entries look like this:

```
{
  data: Array[135],
  endAngle: 2.718685950221936,
  startAngle: 0,
```

```
  value: 135
}
```

We could have specified a `.sort()` function to change how slices are organized and a `.startAngle()` or `.endAngle()` function to limit the pie's size.

All that's left to do now is drawing a pie chart. We'll need an `arc` generator (just like the ones in Chapter 2, *A Primer on DOM, SVG, and CSS*) and some colors to tell the slices apart.

Finding 24 distinct colors that look great together is hard; luckily for us, `@ponywithhiccups` has jumped into the challenge and made the pick. Thank you!

Let's add these colors to `helpers.js`:

```
export const color = d3.scale.ordinal().range(['#EF3B39', '#FFCD05', '#69C9CA',
'#666699', '#CC3366',
  '#0099CC', '#999999', '#FBF5A2', '#6FE4D0', '#CCCB31', '#009966', '#C1272D',
'#F79420', '#445CA9',
  '#402312', '#272361', '#A67C52', '#016735', '#F1AAAF','#A0E6DA', '#C9A8E2',
'#F190AC', '#7BD2EA',
  '#DBD6B6']);
```

The `color` scale is an ordinal scale without a domain. We export it as a constant (that is, a variable that won't change) using the `const` keyword. Exporting variables that aren't constants isn't allowed.

Note

If you try to redefine the `color` constant, Babel will throw this error:

```
Module build failed: SyntaxError: ../learning-d3/src/helpers.js: Line 39:
"color" is read-only
```

Use constants for variables that really shouldn't ever be modified. Note, however, that you can still update your color scale by using `.range` and `.domain` as setters (that is, by supplying them a new array to replace the one you defined in `helpers.js`).

To make sure that the donors always get the same color, a function in `helpers.js` will help us fixate the domain, as shown in the following code:

```
export function fixateColors (data) {
  color.domain(uniques(data, (d) => d.DonorName));
}
```

Now, we can define the `arc` generator in our pie method and fixate the colors:

```
    let arc = d3.svg.arc()
      .outerRadius(150)
      .startAngle((d) => d.startAngle)
      .endAngle((d) => d.endAngle);
```

```
helpers.fixateColors(filtered);
```

A group element will hold each arc and its label, as shown in the following code:

```
let slice = this.chart.selectAll('.slice')
           .data(pie)
           .enter()
           .append('g')
           .attr('transform', 'translate(300, 300)');
```

To make positioning simpler, we move every group to the center of the pie chart. Creating slices works the same as in Chapter 2, *A Primer on DOM, SVG, and CSS*:

```
slice.append('path')
  .attr({
    d: arc,
    fill: (d) => helpers.color(d.data[0].DonorName)
  });
```

We get the color for a slice with `d.data[0].DonorName`. The original dataset is in `.data`, and all the `.DonorName` properties in it are the same. That's what we grouped by.

Labeling your pie chart

Labels take a bit more work. They need to be rotated into place and sometimes flipped so that they don't appear upside down. Labeling an arc will be handy later as well, so let's make a general function in `helpers.js`:

```
export function arcLabels(text, radius) {
  return function (selection) {
    selection.append('text')
      .text(text)
      .attr('text-anchor', (d) => tickAngle(d) > 100 ? 'end' : 'start')
      .attr('transform', (d) => {
        let degrees = tickAngle(d);
        let turn = `rotate(${degrees}) translate(${radius(d) + 10}, 0)`;
        if (degrees > 100) {
          turn += `rotate(180)`;
        }
        return turn;
      });
  }
}
```

Here, we're creating a "factory"-style function that generates another function operating on a D3 selection. This means we can use it with `.call()` while still defining our own parameters.

We'll give `arcLabels` a `text` accessor and a `radius` accessor, and it will return a function that we can use with `.call()` on a selection to make labels appear in just the right places. The meaty part appends a text element, tweaks its `text-anchor` element depending on whether or not we're going to flip it, and rotates the element to a particular position with the help of a `tickAngle` function.

Let's add the contents of the `tickAngle` function:

```
export function tickAngle (d) {
  let midAngle = (d.endAngle - d.startAngle) / 2;
  let degrees = (midAngle + d.startAngle) / Math.PI * 180 - 90;
  return degrees;
}
```

The `helpers.tickAngle` calculates the middle angle between `d.startAngle` and `d.endAngle` and transforms the result from radians to degrees so that SVG can understand it.

This is basic trigonometry, so I won't go into details, but your favorite high schooler should be able to explain the math.

We use `arcLabels` back in the `pie` method:

```
slice.call(helpers.arcLabels((d) => `${d.data[0].DonorName} (${d.value})`,
arc.outerRadius()));
```

Lastly, we initialize the constructor in `index.js`:

```
new PoliticalDonorChart('pie', 'The Rt Hon Edward Miliband MP');
```

And our delicious pie is done, as shown in the following screenshot:

Showing popularity through time with stack

D3's official docs say:

> *"The stack layout takes a two-dimensional array of data and computes a baseline; the baseline is then propagated to the above layers, so as to produce a stacked graph."*

Not clear at all, but I am hard pressed to come up with anything better. The `stack` layout calculates where one layer ends and another begins. An example should help.

We're going to make a layered timeline of donations over the 5-year sample, stretching as far back as 2010, with the width of each layer telling us how many donations went where at a certain time. This timeline is called a **streamgraph**.

To label layers, we're going to create a `mouseover` behavior that highlights a layer and shows a tooltip with the donation recipient's name.

Let's begin the binning. Create a new method called `streamgraph` in `PoliticalDonorChart`:

```
streamgraph() {
  let time = d3.time.format('%d/%m/%y');
  let data = this.data;
  let extent = d3.extent(data.map((d) => time.parse(d.ReceivedDate)));
  let timeBins = d3.time.days(extent[0], extent[1], 12);
}
```

To parse timestamps into `Date` objects, we specified a format for strings such as `27/04/15`. Then, we used this format to find the earliest and latest time with `d3.extent`. Telling `d3.time.days()` to go from start to finish with a step of 14 days creates a list of bins.

We use the `histogram` layout to munge our dataset into a more useful form:

```
  let perName = helpers.binPerName(data, (d) => d.EntityName);
  let timeBinned  = perName.map((nameLayer) => {
    return {
      to: nameLayer[0].EntityName,
      values: d3.layout.histogram()
            .bins(timeBins)
            .value((d) => time.parse(d.ReceivedDate))(nameLayer)
    }
  });
```

You already know what `helpers.binPerName` does.

To bin data into time slots, we mapped through each layer of the `name` accessors and turned it into a two-property object. The `.to` property tells us whom the layer represents, and `.values` is a histogram of time slots where entries tell us how much karma the user many donations somebody got

in a certain 12-day period.

Time for a `stack` layout:

```
let layers = d3.layout.stack()
  .order('inside-out')
  .offset('wiggle')
  .values((d) => d.values)(timeBinned);
```

The `d3.layout.stack()` creates a new `stack` layout. We told it how to order layers with `.order('inside-out')` (you should also try `default` and `reverse`) and decided how the final graph looks with `.offset('wiggle')`. The `wiggle` minimizes changes in slope. Other options include `silhouette`, `zero`, and `expand`. Try them!

Once again, we told the layout how to find values with the `.values()` accessor. Our `layers` array is now filled with objects like this one:

```
{to: " The Rt Hon Edward Miliband MP",
   values: Array[50]}
```

The `values` is an array of arrays. Entries in the outer array are time bins that look like this:

```
{dx: 1036800000,
   length: 1,
   x: Object(Thu Oct 13 2011 00:00:00 GMT+0200 (CEST)),
   y: 1,
   y0: 140.16810522517937}
```

The important parts of this array are as follows: `x` is the horizontal position, `y` is the thickness, and `y0` is the baseline. The `d3.layout.stack` will always return these.

To start drawing, we need some margins and two scales:

```
this.margin = {
  top: 220,
  right: 50,
  bottom: 0,
  left: 50
};

let x = d3.time.scale()
  .domain(extent)
  .range([this.margin.left, this.width - this.margin.right]);

let y = d3.scale.linear()
  .domain([0, d3.max(layers, (layer) => d3.max(layer.values, (d) => d.y0 +
d.y))])
  .range([this.height - this.margin.top, 0]);
```

The tricky thing was finding the vertical scale's domain. We found it by going through each value of every layer, looking for the maximum `d.y0+d.y` value—baseline plus thickness.

We'll use an `area` path generator for the layers:

```
let offset = 100;
let area = d3.svg.area()
  .x((d) => x(d.x))
  .y0((d) => y(d.y0) + offset)
  .y1((d) => y(d.y0 + d.y) + offset);
```

Nothing too fancy. The baselines define bottom edges and adding the thickness gives the top edge. Fiddling determined that both should be pushed down by 100 pixels.

Let's draw an axis first:

```
let xAxis = d3.svg.axis()
  .scale(x)
  .tickFormat(d3.time.format('%b %Y'))
  .ticks(d3.time.months, 2)
  .orient('bottom');

this.chart.append('g')
  .attr('transform', `translate(0, ${this.height - 100})`)
  .classed('axis', true)
  .call(xAxis)
  .selectAll('text')
    .attr('y', 5)
    .attr('x', 9)
    .attr('dy', '.35em')
    .attr('transform', 'rotate(60)')
    .style('text-anchor', 'start');
```

Same as usual—we defined an axis, called it on a selection, and let D3 do its thing. We made it more readable with a custom `.tickFormat()` function and used `.ticks()` to say that we want a new tick every 2 months.

Okay, now for the streamgraph, add the following code:

```
this.chart.selectAll('path')
  .data(layers)
  .enter()
  .append('path')
  .attr('d', (d) => area(d.values))
  .style('fill', (d, i) => helpers.color(i))
  .call(helpers.tooltip((d) => d.to, this.chart));
```

Not much is going on. We used the `area` generator to draw each layer, defined colors with `helpers.color`, and called a `tooltip` function, which we'll define in `helpers.js` later.

The streamgraph looks like this:

Jan 2010
Mar 2010
May 2010
Jul 2010
Sep 2010
Nov 2010
Jan 2011
Mar 2011
May 2011
Jul 2011
Sep 2011
Nov 2011
Jan 2012
Mar 2012
May 2012
Jul 2012
Sep 2012
Nov 2012
Jan 2013
Mar 2013
May 2013
Jul 2013
Sep 2013
Nov 2013
Jan 2014
Mar 2014
May 2014
Jul 2014
Sep 2014
Nov 2014
Jan 2015
Mar 2015
May 2015

Adding tooltips to our streamgraph

It looks pretty, but it is useless. Let's add that `tooltip` function to the `helpers.js` tooltip:

```
export function tooltip (text, chart) {
  return function (selection) {
    selection.on('mouseover.tooltip', mouseover)
      .on('mousemove.tooltip', mousemove)
      .on('mouseout.tooltip', mouseout);
  }
}
```

We defined event listeners with a `.tooltip` namespace so that we can define multiple listeners on the same events.

The `mouseover` function will highlight streams and create tooltips, `mousemove` will move tooltips, and `mouseout` will put everything back to normal.

Let's put the three listeners inside the inner function of our tooltip helper:

```
    function mouseover(d) {
      let path = d3.select(this);
      path.classed('highlighted', true);
  }
```

That's the simple part of `mouseover`. It selects the current area and changes its class to `highlighted`. This will make it lighter and add a red outline.

In the same function, add the meaty part:

```
      let mouse = d3.mouse(chart.node());
      let tool = chart.append('g')
        .attr({
          'id': 'nameTooltip',
          transform: `translate(${mouse[0] + 5},${mouse[1] + 10})`
        });

      let textNode = tool.append('text')
        .text(text(d)).node();

      tool.append('rect')
        .attr({
          height: textNode.getBBox().height,
          width: textNode.getBBox().width,
          transform: 'translate(0, -16)'
        });

      tool.select('text')
        .remove();

      tool.append('text').text(text(d));
```

It is longer and with a dash of magic, but not scary at all!

First, we find the mouse's position. Then we create a group element and position it down and to the right of the mouse. We add a text element to the group and call SVG's `getBBox()` function on its node. This gives us the text element's bounding box and helps us size the background rectangle.

Finally, we remove the text, because it's covered by the background, and add it again. We might be able to avoid all this trouble by using HTML *divs*, but I wanted to show you pure SVG tooltips. Hence, consider the following code:

```
    function mousemove () {
        let mouse = d3.mouse(chart.node());
        d3.select('#nameTooltip')
        .attr('transform', `translate(${mouse[0] + 15},
          ${mouse[1] + 20})`);
    }
```

The `mousemove` listener in the following code is much simpler. It just finds the `#nameTooltip` element and moves it to follow the cursor:

```
    function mouseout () {
        let path = d3.select(this);
        path.classed('highlighted', false);
        d3.select('#nameTooltip').remove();
    }
```

The `mouseout` function selects the current path, removes its `highlighted` styling, and removes the tooltip.

Voilà! Tooltips!

Very rudimentary—they don't understand edges and they won't break any hearts with their looks, but they get the job done. Let's add some CSS to `chapter5.css`:

```
#nameTooltip {
  font-size: 1.3em;
}

#nameTooltip rect {
  fill: white;
}

#nameTooltip text {
  fill: #000;
  stroke: #000;
  color: #000;
}

path.highlighted {
  fill-opacity: 0.5;
  stroke: red;
```

```
  stroke-width: 1.5;
}
```

And suddenly, we have a potentially useful streamgraph on our hands!

Highlighting connections with chord

We've seen how many donations people have and when they got it, but there's another gem hiding in the data—connections. We can visualize who is donating to whom using the `chord` layout.

We're going to draw a chord diagram—a circular diagram of connections. Chord diagrams are often used in genetics and have even appeared on covers of magazines (http://circos.ca/intro/published_images/).

Ours is going to have an outer ring showing how much money is being donated and chords showing where that money is going.

First, we need a matrix of connections for the chord diagram, and then we'll go the familiar route of path generators and adding elements. The matrix code will be useful later, so let's put it in `helpers.js`:

```
export function connectionMatrix (data) {
  let nameIds = nameId(allUniqueNames(data), (d) => d );
  let uniques = nameIds.domain();
  let matrix = d3.range(uniques.length).map(
    () => d3.range(uniques.length).map(() =>  0));
  data.forEach((d) => {
    matrix[nameIds(d.DonorName)][nameIds(d.EntityName)] +=
Number(d.Value.replace(/[^\d\.]*/g, ''));
  });

  return matrix;
}
```

Let's also create a function that returns unique names:

```
export function allUniqueNames(data) {
  let donors = uniques(data, (d) => d.DonorName);
  let donees = uniques(data, (d) => d.EntityName);
  return uniques(donors.concat(donees), (d) => d);
}
```

We begin with the familiar `uniques` list and the `nameId` scale. Then we create a zero matrix and loop through the data to increment by the value of each donation (which we quickly clean up by removing the pound symbol and the comma before changing it to a `Number`). Rows are *from whom*, columns are *to whom*. For example, if the fifth cell in the first row holds `10`, it means the first person or organization has given £10 to the fifth person. This is called an **adjacency matrix**.

Meanwhile, back in `PoliticalDonorChart`, create a `chord` method:

```
chord(filterString) {
    let filtered = this.data.filter((d) => d.EntityName.match(filterString || '
MP') );
    let uniques = helpers.uniques(filtered, (d) => d.DonorName);
```

```
    let matrix = helpers.connectionMatrix(filtered);
}
```

We create the matrix from our data, which we've filtered for individual MPs. We can also provide an argument in the constructor to filter based on it. Moreover, we create another array of unique names, this time for MPs. This allows us to have something to connect to.

We're going to need `uniques` for labels, and it would be nice to have the `innerRadius` and `outerRadius` variables handy:

```
    let innerRadius = Math.min(this.width, this.height) * 0.3;
    let outerRadius = innerRadius * 1.1;
```

Time to make the `chord` layout do our bidding:

```
    let chord = d3.layout.chord()
      .padding(.05)
      .sortGroups(d3.descending)
      .sortSubgroups(d3.descending)
      .sortChords(d3.descending)
      .matrix(matrix);
```

It is a little different from others. The `chord` layout takes data via the `.matrix()` method and can't be called as a function.

We started with `d3.layout.chord()` and put a `.padding()` method between groups, which improves readability. To improve readability further, everything is sorted. The `.sortGroups` sorts groups on the edge, `.sortSubgroups` sorts chord attachments in groups, and `.sortChords` sorts the chord drawing order so that smaller chords overlap bigger ones.

In the end, we feed data into the layout with `.matrix()`:

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

We add a centered group element so that all our coordinates are relative to the center from now on.

The drawing of the diagram happens in three steps—arcs, labels, and chords—as shown in the following code:

```
    let group = diagram.selectAll('.group')
      .data(chord.groups)
        .enter().append('g');

    let arc = d3.svg.arc()
      .innerRadius(innerRadius)
      .outerRadius(outerRadius);

    group.append('path')
      .attr('d', arc)
```

```
      .attr('fill', (d) => helpers.color(d.index));
```

This creates the outer ring. We use `chord.groups` to get group data from the layout, create a new grouping element for every chord group, and then add an arc. We use `arc_labels` from the pie example to add the labels:

```
group.call(helpers.arcLabels(
  (d) => uniques[d.index], () => outerRadius + 10));
```

Even though the radius is constant, we have to define it as a function using the following code because we didn't make `arcLabels` flexible enough for constants. Nobody ain't got time for that, though—we still have seven charts to make in this chapter!

```
    diagram.append('g')
           .classed('chord', true)
           .selectAll('path')
           .data(chord.chords)
           .enter()
           .append('path')
           .attr('d', d3.svg.chord().radius(innerRadius))
           .attr('fill', (d) => helpers.color(d.target.index));
```

We get chord data from `chord.chords` and use a `chord path` generator to draw the chords. We pick colors with `d.target.index` because the graph looks better, but chord colors are *not* informative.

We add some CSS to `chapter5.css` to make the chords easier to follow:

```
.chord path {
      stroke: black;
      stroke-width: 0.2;
      opacity: 0.6;
}
```

Finally, replace the new call in `index.js` with the following:

```
new PoliticalDonorChart('chord');
```

And now we have a very busy and hard-to-read diagram!

Professor Tom Burns
Crockett & Jones Ltd
The Skills Funding Agency
Mrs Hilda Worth
Gail Rebuck
Carpet & Flooring Megastore
Mark Latham
Mr Ross Pullan
Mr Paul Baker
Maria Fernandes
Elaine Drage
John Drage
R Collard Ltd
Lance Mason Ltd
Longhill Environmental Ltd
Mr Tim Butcher Butcher
Packt Publishing Limited
Betterworld Ltd
Lord Michael Levy
Signature Tax Ltd
Sonnie Leong
Dr Gurnek Bains
Margaret Thornton
Stewart Thornton
John Kennedy
Mr Ray Bloom
Martin Littler
Nicolas Stanley
Mr Theodore Fras
Mr Lennart Perhag
Great Yarmouth W
Centrax Industries L
Mr Martyn Rose
The Arts Club
Tim Harris
Mr Jeremy Isaacs
Stephen Stefanou
Golden Kinsella
Transport Tours
Mr John Moores
Mr William Salomon
Carus Pharmaceuticals Ltd
Henry Davis
James O'Keefe
Ascroft Contractors Ltd
Marc Pennick
Creative Coalition Campaign Secretariat
Camira Fabrics Ltd
Mr Scott Saunders
Industry of Sound
Metas Ltd
P & G Investments LLP
Chat Group Ltd
Richard Greer
Ms Joanne Rowling
Simon Chappell
Mark Deniston
Alan Parker
Reza Tabrizi
Mr Rupert Read
Dr Prasad Rao
Beauchamp Estates Limited
Mr Adrian Christmas
AB Produce Plc
Esher and Walton Connect
Unison
Ms Helena Doughty
Commercial Services Design Ltd
Mr Alan Bekhor
Ohotels Group Limited
Mr Mal Ogg
Women2Win
Trevor Chinn
Mr Kishorkant Bhatessa
Susan Anstey
Duncan Kenworthy
Nanrows Ltd
Andrew McGuiness
Pasta Foods
Analysis, Research & Communications (ARC) Ltd
Sarah Barr
Mr Frederick Forsyth
Farz-u-Rasool
Mr John Chatfeild-Roberts
Thompsons Solicitors LLP
Ms Diana Irving
Suzanne Mozzi
Mr David Evans
Mr Stephen Brennink
Bingham Capital
Ms Jo Evans
Mr Trevor Evans of Watfo
Mr Paul Nyners
Union of Construct
Mr Robertson
Michael Sanzone
Pasternco Ltd
Mr Killian Hurle
Patrick Stewart
Mr Jayant Trade
Maritime and Trade
Mr Jayant Trade
Prime Bank
Peter Bonsonne
Mr Ashley Ward
Mr Ashley Ward
Pooja Singh
Mr Sukhna Singh
Mr Kwame Kolade
Mr Kwame Kolade
Mr Kwame Kolade

First of all, chord colors don't mean anything! They just make it easier to distinguish chords. Furthermore, this graph shows how big each donation is by the size of the chord.

A chord chart is suboptimal for this type of data because donations are pretty unidirectional—MPs typically don't give money back to donors. In the preceding chart, you see your arcs kind of tapering towards nowhere; this is because we don't have an item on the chart for where all the donations are going. Fixing it's a bit hacky, but easy enough. We put this code block before the line where we define `uniques` in our `chord` method:

```
let uniqueMPs = helpers.uniques(filtered, (d) => d.EntityName);
uniqueMPs.forEach((v) => {
  filtered.push({
    DonorName: v,
    EntityName: v,
    Value: '9001'
  });
```

```
});
```

This just adds another entry and gives it a value that's over nine thousand (nine thousand?!), which is just an arbitrary number I chose that gives the entries a bit of size but not enough to distort the graphic. Feel free to play around with it.

Lastly, add the following code at the bottom of `chord` to make the entries you just added bold:

```
this.chart.selectAll('text').filter(function(d) {
  return d3.select(this).text().match(filterString)})
  .style('font-weight', 'bold');
```

There we go! It makes slightly more sense now! From this, you can tell that *Unite* (Britain's largest union, for those outside the UK) gave a lot of money to Ed Miliband's campaign, and *Silvergate Investments* gave a lot to David Miliband. You'll notice that both are listed twice; clearly this data needs some cleanup. As you build things in D3, you'll often find situations like this. You can do either one of two things: handle all these edge cases in your code, or clean up the data using something like OpenRefine. It's much better to clean up messy data before it gets to D3 as it results in cleaner and easier-to-read code with less hacky conditional logic.

Tip

For a good tutorial on using OpenRefine, visit http://schoolofdata.org/handbook/recipes/cleaning-data-with-refine/.

Unite
David Sainsbury
David Claydon
Silvergate Investments
Silvergate Investments Ltd
Anthony Bailey
GMB
GLC Ltd
Tony Blair
USDAW
Gulam Noon
UNITE the union
David Puttnam
Sue Nye
Paul Ettinger
Professor John Dainton
Dr G S Bains
Robert Gavron
Kevin McGrath
Patil Diagnostics Ltd
Clive Hollick
Waheed Alli
Sir Sigmund Sternb
Parry Mitchell
Trevor Beattie
Community
The Rt Hon Edwa
David Miliband MP
Stefanos Stafanou
Clive Jones
Andrew McGuiness
Duncan Kenworthy
Faiz-ul Rasool
Gabrielle Dellal
Trevor Chinn
Lord David Evans
Sonny Leong
of Construction, Allied Trades and Technicians
Qhotels Group Limited
Lord Evans of Watford
Patrick Stewart
Trust Reservations Ltd
Alan Parker
Richard Greer
Mark Deniston
Reza Tabrizi
James O'Keefe
Stephen Kinsella
Stefanos Stefanou
Dr Gurnek Bains
Martin Littler
Margaret Thornton
Stewart Thornton
Sonnie Leong
Nicolas Stanley
Maria Fernandes
Mark Latham
Sonia Klein
Lord na Moonie
Mr S Akhtar
Flora Fraser Soros
Emily Thornberry
David McMillen
Anwar Ansari
Ann Limb
Professor Tom Burns
Testhouse Limited
Landlegend Ltd
Ahmed Noor
Hamish Sandison
David Rowntree
Gail Rebuck
Peter Droussiotis
Paul Callaghan
chel Reeves MP
Anthony Burton

Drawing with force

The `force` layout is the most complicated of the non-hierarchical layouts. It lets you draw complex graphs using physical simulations—force-directed graphs if you will. Everything you draw will have built-in animation.

We're going to draw a graph of connections between donors and politicians. Every donor and politician will be a node, the size of which will correspond to the user's total donations. Links between nodes will tell us who is donating to whom.

To make things clearer, we're going to add tooltips and make sure that hovering over a node highlights the connected nodes.

Let's begin!

As in the chord example, we begin with a matrix of connections. We aren't going to feed this directly to the `force` layout, but we will use it to create the kind of data it enjoys. Start by changing the constructor in `index.js` to the following:

```
new PoliticalDonorChart('force');
```

Then add a `force` method to your chart class:

```
  force(filterString =  ' MP ') {
    let filtered = this.data.filter(
      (d) => d.EntityName.match(filterString) );
    let nameId = helpers.nameId(
      helpers.allUniqueNames(filtered), (d) => d);
    let uniques = helpers.allUniqueNames(filtered);
    let matrix = helpers.connectionMatrix(filtered);
  }
```

D3's `force` layout expects an array of nodes and links. Let's make them:

```
let nodes = uniques.map((name) => new Object({
  name: name,
  totalDonated: matrix[nameId(name)].reduce(
    (last, curr) => last + curr, 0),
  totalReceived: matrix.reduce((last, curr) => last + curr[nameId(name)], 0)
}));

let links = filtered.map((d) => {
  return {
    source: nameId(d.DonorName),
    sourceName: d.DonorName,
    target: nameId(d.EntityName),
    targetName: d.EntityName,
    amountDonated: matrix[nameId(d.DonorName)][nameId(d.EntityName)]
  }
});
```

We're defining the bare minimum of what we need, and the layout will calculate all the hard stuff.

The `nodes` tell us who they represent, and `links` connect a `source` object to a `target` object with an index into the `nodes` array. The layout will turn them into proper references, as shown in the following code. Every link also contains a `count` object that we'll use to define its strength:

```
let force = d3.layout.force()
  .nodes(nodes)
  .links(links)
  .charge((node) => node.totalDonated ? -50 : 0)
  .gravity(0.05)
  .size([this.width, this.height]);

 force.start();
```

We create a new `force` layout with `d3.layout.force()`; just like the `chord` layout, it isn't a function either. We feed in the data with `.nodes()` and `.links()`. The `charge` is set so that nodes that are donators repel each other, which will help prevent everything from just clumping in the center. The `gravity` setting pulls the graph towards the center of the image; we defined its strength with `.gravity()`. We tell the `force` layout the size of our picture with `.size()`. No calculation happens until `force.start()` is called, but we need the results to define a few scales for later.

There are a few more parameters to play with: the overall `.friction()` (the smallest `.linkDistance()` value the nodes stabilize to) and `.linkStrength()` for link stretchiness. Play with them. The `nodes` members now look like this:

```
{index: 0,
  name: "The Rt Hon Edward Miliband MP",
  px: 497.0100389553633,
  py: 633.2734045531992,
  weight: 100,
  x: 499.5873097327753,
  y: 633.395804766377}
```

The `weight` tells us how many links connect with this node, `px` and `py` state its previous position, and `x` and `y` state the current position.

The `links` members are a lot simpler:

```
{count: 2
  source:1 Object
  target: Object}
```

Both the `source` and `target` objects are a direct reference to the correct node. Now that the layout has made its first calculation step, we have the data needed to define some scales:

```
let distance = d3.scale.linear()
   .domain(d3.extent(d3.merge(matrix)))
   .range([300, 100]);
```

```
let given = d3.scale.linear()
    .domain(d3.extent(matrix, (d) => d3.max(d)))
   .range([2, 35]);
```

We're going to use the `given` scale for node sizes and `distance` for link lengths. Nodes that either receive or donate more will appear bigger, and nodes representing donors will be closer to the nodes they donated to based on the donated amount:

```
force.linkDistance((d) => distance(d.amountDonated));
force.start();
```

We use `.linkDistance()` to dynamically define link lengths according to the `.count` property. To put the change in effect, we restart the layout with `force.start()`.

Finally! Time to put some ink on paper—well—pixels on screen!

```
let link = this.chart.selectAll('line')
  .data(links)
  .enter()
    .append('line')
    .classed('link', true);
```

Links are simple. Go through the list of links and draw a `line`.

Next, draw a circle for every donor node and a square for every recipient:

```
let node = this.chart.selectAll('.node')
  .data(nodes)
  .enter()
    .append((d) => {
      return document.createElementNS(
        'http://www.w3.org/2000/svg', d.totalDonated > 0 ?
        'circle' : 'rect');
    })
    .classed('node', true);

this.chart.selectAll('circle.node')
  .attr({
    r: (d) => given(d.totalDonated),
    fill: (d) => helpers.color(d.index),
    class: (d) => 'name_' + nameId(d.name)
  })
  .classed('node', true);

this.chart.selectAll('rect.node')
  .attr({
    width: (d) => given(d.totalReceived),
    height: (d) => given(d.totalReceived),
    fill: (d) => helpers.color(d.index),
    class: (d) => 'name_' + nameId(d.name)
  })
  .classed('node', true);
```

The only unusual bit in the preceding code is us using a callback for the `.append` method, which requires a new element to be returned. Because we aren't able to use `.append` itself when we override its accessor (since we're conditionally making the element a circle or square depending on whether the node is a donor or recipient of a donation), we have to set the namespace and everything…

We add tooltips with the familiar `helpers.tooltip` function, and `force.drag` will automatically make the nodes draggable:

```
node.call(helpers.tooltip((d) => d.name, this.chart));
node.call(force.drag);
```

After all that work, we still have to do the updating on every tick of the `force` layout animation:

```
force.on('tick', () => {
  link.attr('x1', (d) => d.source.x)
  .attr('y1', (d) => d.source.y)
  .attr('x2', (d) => d.target.x)
  .attr('y2', (d) => d.target.y);

  this.chart.selectAll('circle.node').attr('cx', (d) => d.x)
  .attr('cy', (d) => d.y);

  this.chart.selectAll('rect.node').attr('x', function(d) {
    return d.x - this.getBBox().width / 2;
  })
  .attr('y', function(d) {
    return d.y - this.getBBox().height / 2
  });
});
```

On a `tick` event, we move every `link` endpoint and `node` to its new position. Simple!

Add some styling to `chapter5.css`:

```
.link {
  stroke: lightgrey;
  stroke-width: 0.3;
}
```

And voilà! We get a force-directed graph of election donations.

Running this example looks silly because it spins around a lot before settling down. But once it stabilizes, the graph looks something like this:

Drag and pull it around the screen to see different connections.

We should have added some code to print names next to the highlighted nodes, but the example was long enough. Let's say that's left as an exercise for the reader.

We will now move on to hierarchical layouts!

Hierarchical layouts

All hierarchical layouts are based on an abstract hierarchy layout designed for representing hierarchical data—data within data within data within data within.... all the way down. As mentioned earlier, imagine a tree, or an org chart if you don't go outside very much.

All of the common code for the `partition`, `tree`, `cluster`, `pack`, and `treemap` layouts is defined in `d3.layout.hierarchy()`, and they all follow similar design patterns. The layouts are so similar that the official documentation very obviously copy-pastes most of its explanations which are practically identical. Let's avoid that by looking at the common stuff first, and then we will focus on the differences.

First of all, we need some hierarchical data. There's an implicit hierarchy in our data, insomuch that donations flow downwards, but it's not as well-defined as it would be for, say, organizational units of most businesses or the filesystem tree of a hard drive. The result is a scheme that works well with three of the layouts but looks slightly contrived for the other two—apologies in advance for that, but hopefully, it gives you the right idea.

We'll have a root node called `donations`, which will contain the names of people or groups that have donated in the last election cycle. For the `tree` and `cluster` layouts, each of those will contain nodes for everyone they have donated to. For the `partition`, `pack`, and `treemap` layouts, child nodes will tell us who contributed to the parent's total received donations.

The final data structure will look like this:

```
{
    "name": "donations",
    "children": [
        {
            "name": "Unite",
            "count": 5,
            "children": [
                {
                    "name": "Ed Balls",
                    "count": 2,
                    "children": []
                }, {
                    "name": "Ed Miliband",
                    "count": 6,
                   "children": []
              }
}
```

While it could potentially go on forever, that wouldn't make sense in our case. We have only three levels in this example, but you can go down as many levels as you like.

The default accessor expects a `.children` property, but we could have easily done something crazy,

such as dynamically generating a fractal structure in a custom accessor.

As usual, there's a `.value()` accessor that helps layouts to find data in a node. We'll use it to set the amount either donated or received.

To run a hierarchical layout, we call `.nodes()` with our dataset. This immediately returns a list of nodes that you can't get to later. For a list of connections, we call `.links()` with a list of our nodes. Nodes in the returned list will have some extra properties calculated by the layout. Most layouts tell us where to put something with `.x` and `.y`, and then use `.dx` and `.dy` to tell us how big the layout should be.

All hierarchical layouts also support sorting with `.sort()`, which takes a sorting function, such as `d3.ascending` or `d3.descending`.

Enough of theory! Let's add a data munging function to `helpers.js`:

```
export function makeTree(data, filterByDonor, name1, name2) {
  let tree = {name: 'Donations', children: []};
  let uniqueNames = uniques(data, (d) => d.DonorName)

  tree.children = uniqueNames.map((name) => {
    let donatedTo = data.filter((d) => filterByDonor(d, name));
    let donationsValue = donatedTo.reduce((last, curr) => {
          let value = Number(curr.Value.replace(/[^\d\.]*/g, ''));
          return value ? last + value : last;
      }, 0);

    return {
      name: name,
      donated: donationsValue,
      children: donatedTo.map((d) => {
        return {
          name: name2(d),
          count: 0,
          children: []
        };
      })
    };
  });

  return tree;
}
```

Wow, there's a lot going on here! We avoided recursion because we know our data will never nest more than two levels deep.

The `tree` holds an empty root node at first. We use `helpers.uniques` to get a list of names. Then we map through the array and define the children of the root node by counting everyone's donations and using `helpers.binPerName` to get an array of children.

The code is wibbly-wobbly because we use `filterByDonor`, `name1`, and `name2` for data accessors, but making this function flexible makes it useful in all hierarchical examples.

Drawing a tree

The `tree` layout displays data in a tree using the Reingold-Tilford tidy algorithm. We'll use it to display our dataset in a large circular tree, with every node connected to its parent by a curvy line.

We begin the method by fixating colors, turning data into a tree, and defining a way to draw curvy lines:

```
  tree(filterString = ' MP') {
    let filtered = this.data.filter(
      (d) => d.EntityName.match(filterString) );
    helpers.fixateColors(filtered);

    let tree = helpers.makeTree(filtered,
      (d, name) => d.DonorName === name,
      (d) => d.EntityName,
      (d) => d.EntityName || '');

    tree.children = tree.children.filter(
      (d) => d.children.length > 1)

    let diagonal = d3.svg.diagonal.radial()
      .projection((d) => [d.y, d.x / 180 * Math.PI]);
}
```

You know `fixateColors` from before. We defined `makeTree` about one page ago, and we talked about the `diagonal` generator in Chapter 2, *A Primer on DOM, SVG, and CSS*:

```
    let layout = d3.layout.tree()
        .size([360, this.width / 5]);
    let nodes = layout.nodes(tree);
    let links = layout.links(nodes);
```

We create a new tree layout by calling `d3.layout.tree()`. Defining its size with `.size()` and executing it with `.nodes().size()` tells the layout how much room it's got—in this case, we're using `x` as an angle (`360` degrees) and `y` as a radius, though the layout itself doesn't really care about that.

To avoid worrying about centering later on, we put a grouping element center stage:

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

First, we are going to draw the links, and then the nodes and their labels:

```
    let link = chart.selectAll('.link')
      .data(links)
        .enter()
        .append('path')
        .attr('class', 'link')
        .attr('d', diagonal);
```

You should be familiar with this by now; go through the data and append new paths shaped with the `diagonal` generator:

```
    let node = chart.selectAll('.node')
      .data(nodes)
      .enter().append('g')
      .attr('class', 'node')
      .attr('transform', (d) => `rotate(${d.x - 90})translate(${d.y})`);
```

For every node in the data, we create a new grouping element and move it into place using `rotate` for angles and `translate` for radius positions.

Now it's just a matter of adding a circle and a label:

```
    node.append('circle')
      .attr('r', 4.5)
      .attr('fill', (d) => helpers.color(d.name))
      .on('mouseover', function(d) {
        d3.select(this.nextSibling).style('visibility', 'visible');
      })
      .on('mouseout', function(d) {
        d3.select(this.nextSibling).style('visibility', 'hidden');
      });
    node.append('text')
      .attr('dy', '.31em')
      .attr('text-anchor', (d) => d.x < 180 ? 'start' : 'end')
      .attr('transform', (d) => d.x < 180 ? 'translate(8)' :
'rotate(180)translate(-8)')
      .text((d) => d.depth > 1 ? d.name : d.name.substr(0, 15) + (d.name.length
> 15 ? '...' : ''))
      .style({
        'font-size': (d) => d.depth > 1 ? '0.6em' : '0.9em',
        'visibility': (d) => d.depth > 0 ? 'hidden' : 'visible'
      });
```

Every node is colored with the user's native color, and the text is transformed similarly to the earlier pie and chord examples. Finally, we've made leaf nodes' text smaller to avoid overlap.

Our tree looks something like this:

It's rather big, so you should try it out in the browser. Just remember that the inner ring represents users, people, or groups giving karma donations to the politicians in the outer ring.

Showing clusters

The `cluster` layout is the same as the `tree` layout, except that the leaf nodes line up.

Code-wise, this example is the same as the last, so we won't go through it again. Really, the only difference is that we don't have to flip labels at certain angles. You can look at the code in `chapter5.js` of the book's repo.

We end up with a very tall graph that looks something like this:

Partitioning a pie

Now we're getting somewhere! The next three layouts fit our data perfectly—we're taking three looks at how our core politicians' donations are structured.

The `partition` layout creates adjacency diagrams, where you don't draw nodes with links between them but next to each other so that it looks as if the children partition the parent.

We are going to draw a two-layer donut chart. Users will go on the first layer and the layer on top will show us where the donations are coming from.

We begin by munging the dataset and fixating colors; it's the same as before:

```
  partition(filterString = ' MP') {
    let filtered = this.data.filter(
      (d) => d.EntityName.match(filterString) );
    let tree = helpers.makeTree(filtered,
      (d, name) => d.DonorName === name,
      (d) => d.EntityName,
      (d) => d.EntityName || '');

    helpers.fixateColors(filtered);
}
```

Then we use the `partition` layout:

```
    let partition = d3.layout.partition()
      .value((d) => d.parent.donated)
      .sort((a, b) => d3.descending(a.parent.donated, b.parent.donated))
      .size([2 * Math.PI, 300]);

    let nodes = partition.nodes(tree);
```

We used `.value()` to tell the layout that we care about the `.donated` values, and we'll get a better picture if we `.sort()` the output. Similar to the `tree` layout, `x` will represent angles—this time in radians—and `y` will be radii.

We need an `arc` generator as well, as shown in the following code:

```
    let arc = d3.svg.arc()
      .innerRadius((d) => d.y)
      .outerRadius((d) => d.depth ? d.y + d.dy / d.depth : 0);
```

The generator will use each node's `.y` property for the inner radius and add `.dy` for the outer radius. Fiddling shows that the outer layer should be thinner. Hence, we are dividing it by the tree depth.

Notice that there's no accessor for `.startAngle` and `.endAngle`, which are stored as `.x` and `.dx`. It's easier to just fix the data:

```
    nodes = nodes.map((d) => {
      d.startAngle = d.x;
      d.endAngle = d.x + d.dx;
      return d;
    });
    nodes = nodes.filter((d) => d.depth);
```

It is as simple as mapping the data, defining angle properties, and then filtering the data to make sure that the root isn't drawn.

We use the familiar grouping trick to center our diagram:

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

The preparation work is done. It's drawing time!

```
    let node = chart.selectAll('g')
      .data(nodes)
      .enter()
      .append('g');

    node.append('path')
    .classed('partition', true)
    .attr({
      d: arc,
      fill: (d) => helpers.color(d.name)
    });
```

An arc is drawn for every node. The color is chosen as usual. Lastly, we add tooltips because adding labels is a bit too messy on this one:

```
node.call(helpers.tooltip(function (d) { return d.nick; }));
```

Add some more CSS to `chapter5.css`:

```
path.partition {
  stroke: white;
  stroke-width: 1;
}
```

Finally, instantiate in `index.js`. We're going to continue filtering by the `Miliband` brothers:

```
new PoliticalDonorChart('partition', 'Miliband');
```

The adjacency diagram looks like this:

The outer segments represent either Ed or David Miliband (depending on the color), and the inner segments represent the donors

Packing it in

The `pack` layout uses packing to visually represent hierarchies. It stuffs children nodes into their parents, trying to conserve space and sizing each node so that it's the cumulative size of its children.

Conceptually, it's very similar to the `treemap` layout, so I'm going to skip all of the code and just show you the image. To see the code that generated it, check out the `pack` method in `chapter5.js`:

It looks nice, but the pack layout probably isn't the best way to visualize this particular dataset

The code is rather familiar—generate a tree, fixate the colors, create the layout, tweak a few parameters, get computed nodes, draw the nodes, and add tooltips. Simple!

It looks very pretty, but it's not too informative. Adding labels wouldn't help much either because most of the nodes are too small.

Subdividing with treemap

The `treemap` layout subdivides nodes with horizontal and vertical slices, essentially packing children into their parents, just like the `pack` layout, but using rectangles. As a result, node sizes on every level can be compared directly, making this one of the best layouts for analyzing cumulative effects of subdivisions.

We are going to have some fun with this example. Tooltips will name the parent—parents are almost completely obscured by the children—and moving the mouse arrow over a node will make unrelated nodes become lighter, making the graph less confusing (at least in theory).

It's also a cool effect and a great way to end this chapter on layouts. But we begin with the boring stuff—prepare data and fixate the colors:

```
  treemap(filterString = ' MP') {
    let filtered = this.data.filter(
      (d) => d.EntityName.match(filterString) );
    let tree = helpers.makeTree(filtered,
      (d, name) => d.DonorName === name,
      (d) => d.EntityName,
      (d) => d.EntityName || '');

    helpers.fixateColors(filtered);
}
```

Creating the `treemap` layout follows familiar patterns:

```
    let treemap = d3.layout.treemap()
      .size([this.width, this.height])
      .padding(3)
      .value((d) => d.parent.donated)
      .sort(d3.ascending);

    let nodes = treemap.nodes(tree)
      .filter((d) => d.depth);
```

We added some padding with `.padding()` to give the nodes room to breathe. Every node will become a group element holding a rectangle. The leaves will also hold a label:

```
    let node = this.chart.selectAll('g')
      .data(nodes)
      .enter()
      .append('g')
      .classed('node', true)
      .attr('transform', (d) => `translate(${d.x},${d.y})`);

    node.append('rect')
      .attr({
        width: (d) => d.dx,
        height: (d) => d.dy,
```

```
      fill: (d) => helpers.color(d.name)
    });
```

Now, for the first fun bit. Let's fit labels into as many nodes as they can possibly go:

```
  let leaves = node.filter((d) => d.depth > 1);

  leaves.append('text')
    .text((d) => {
      let name = d.name.match(/([^\s]+\s[^\s]+) MP$/)
        .shift().split(' ');
      return `${name[0].substr(0, 1)}. ${name[1]}`;
    })
    .attr('text-anchor', 'middle')
    .attr('transform', function (d) {
      let box = this.getBBox();
      let transform = `translate(${d.dx / 2},${d.dy / 2 + box.height / 2})`;
      if (d.dx < box.width && d.dx > box.height && d.dy > box.width) {
        transform += 'rotate(-90)';
      } else if (d.dx < box.width || d.dy < box.height) {
        d3.select(this).remove();
      }

      return transform;
    });
```

Finally! That was some interesting code!

We found all the leaves and started adding text. To fit labels into nodes, we get their size with `this.getBBox()`. Then we move them to the middle of the node, and check for fit. We also do a bit of regex and array manipulation to the text so that it removes everything except the last name and the initial before that. This is not the most sure-fire way to sanitize those strings—it's much better to use something like OpenRefine to clean up your data beforehand and put it in the format you ultimately want to present it as—but it works for our purpose right now.

If the label is too wide but fits vertically, we rotate it; otherwise, we remove the label after verifying again that it doesn't fit. Checking the height is important because some nodes are very thin.

We add tooltips with `helpers.tooltip`:

```
leaves.call(helpers.tooltip((d) => d.parent.name, this.chart));
```

Another fun bit—partially hiding nodes from different parents:

```
    leaves.on('mouseover', (d) => {
      let belongsTo = d.parent.name;
      this.chart.selectAll('.node')
        .transition()
        .style('opacity', (d) => {
          if (d.depth > 1 && d.parent.name !== belongsTo) {
            return 0.3;
```

```
        }

        if (d.depth == 1 && d.name !== belongsTo) {
          return 0.3;
        }

        return 1;
      });
    })
    .on('mouseout', () => {
      d3.selectAll('.node')
        .transition()
        .style('opacity', 1);
    })
    .on('click', (d) => alert(d.name));
```

We used two mouse event listeners: one creates the effect, another removes it. The `mouseover` listener goes through all the nodes and lightens those with a different parent or those that aren't parents (that is, `d.parent.name` and `d.name` are different). This listener removes all changes. We also have it do an alert popup with the full label text if we click on any of the `rect` elements.

Note

The `alert()` is without a doubt the most disgusting way of presenting information to the user. Even after ignoring the indescribably obnoxious ways in which this browser "feature" was used by Internet marketers in the 90s, it should still never be used, given their `alert()` method's visual similarity to operating system alerts, which can confuse the recipient. Furthermore, you can't style them, and they steal focus from whatever else is going on in the browser. You shouldn't even use them for debugging; `console.log` and `console.dir` are far more descriptive for dumping variables.

I used them in this example purely for demonstrative purposes; in all actuality, a much better way to handle click events would be to update a text element beneath the treemap. I'll leave this as an exercise for the reader.

After this, add a few more lines of CSS to `chapter5.css`:

```
.name text {
  font-size: 1.5em;
}

.name rect {
  fill: white;
}
```

The end result looks like an abstract painting:

Each grouping is a single donor, with the size of its contents reflecting the size of the donation.

Summary

Despite the near-mythical power of D3 layouts, they turn out to be nothing more than helpers that turn your data into a collection of coordinates.

After going all-out with these examples, we used almost every trick we've explained so far. We even wrote so much code that we had to make a separate library! With a bit of generalization, some of those functions could be layouts of their own. There's a whole world of community-developed layouts for various types of charts. The `d3-plugins` repository on GitHub (https://github.com/d3/d3-plugins) is a good place to start exploring.

You now understand what all the default layouts are up to, and I hope you're already thinking about using them for purposes beyond the original developers' wildest dreams.

In the next chapter, you'll learn how to use D3 *outside* of the browser. That's right folks, we're headed to Server Town! In doing so, we'll strip D3 right down to its bare bones and use it to render things that aren't even SVG!

Chapter 6. D3 on the Server with Node.js

Here's where we start to get really funky with D3. Not only can it render beautiful charts on the frontend, but we can also use D3 to generate things before they even get to the user's browser. This is really the cutting edge of D3, so realize that the skills you've learned in the preceding chapters will serve you in 95 percent of situations and don't sweat it if you want to stick to the frontend for now. This chapter will still be here for that rainy afternoon when you want to try to figure out how **Heroku** works.

We've added this chapter to the second edition because it uses D3 in a really abstract sense and we can start to tie up a lot of the stray concepts we've started discussing in the book

Readying the environment

Ever written server apps in PHP? If so, you're in for a treat. JavaScript web applications are a million times easier to deploy thanks to **Platform as a Service** (**PaaS**) webhosting providers, and you can manage an entire fleet of servers using a few simple tools. Instead of fighting with a huge monolithic Apache or Nginx configuration, you can deploy a new instance for every app you create, which sandboxes them and allows for much more compact infrastructure. We'll discuss how to deploy to Heroku later in the chapter; for now, we're just going to test everything locally.

Tip

What is Heroku and do you have to use it? Heroku is a way of deploying applications that use "12-factor app" principles (for the specifics, visit http://12factor.net). Without going too deep into the 12-factor app philosophy, the idea is that you try to create *stateless* applications that use web services in place of a large, monolithic piece of server infrastructure (for instance, a Linux-based virtual server running both the webserver and database processes). I use Heroku in this instance because it's simple to deploy to and free to use in limited capacities, but you can also deploy the code we'll write in this chapter on any server infrastructure that has Node.js installed.

You may not know it, but practically everything you need to write a server application resides in our project directory. If you've never done Node.js development before, it's very similar to what we've done in the preceding chapters with the browser, but with its own APIs and concepts. The JavaScript engine running on both Google Chrome and Node.js is called "V8", so you don't have to learn a totally different set of languages or skills in order to start immediately building server applications.

We're going to install a few more dependencies via `npm`:

```
$ npm install express body-parser --save
```

`express` is the leading Node.js web server library; it nicely abstracts Node.js's ability to open ports and serve content into an easy-to-use API that is very light and fast. However, because Node.js uses the CommonJS module loading standard, we need to add a new set of build instructions to our `webpack.config.js`. We need to make the object being exported an array (so both configurations run), so make it look as follows (the first config has been truncated for space reasons):

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

module.exports = [
  { ... }, // This is the first config; leave it be!
  {
      name: 'server',
      entry: './src/chapter6.js',
      target: 'node',
      output: {
          path: path.resolve(__dirname, 'build'),
          filename: 'server.js'
      },
```

```
      externals: {
        canvas: 'commonjs canvas'
      },
      module: {
        loaders: [
          {
            test: /\.js?$/,
            exclude: 'node_modules',
            loader: 'babel'
          },
          {
            test: /\.json$/,
            loader: 'json-loader'
          }
        ]
      }
  }
];
```

It's pretty much the same; we've just changed the target to `node`. The only thing that's different is we've told Webpack to treat `node-canvas` (which we'll use later in the chapter) as an external library instead of trying to bundle it.

Note that this will still emit all your other code during a build; if it's going slowly, feel free to temporarily disable the first `config` by commenting it out.

All aboard the Express train to Server Town!

Okay, let's get into the nitty-gritty right away and I'll explain what's going on. Add all of this into a new file called `chapter6.js`:

```
import express from 'express';
import bodyParser from 'body-parser';
import d3 from 'd3';
import {readFile} from 'fs';

import {nearestVoronoi} from './helpers';

let app = express();

app.use(bodyParser.urlencoded());
```

Here we're just importing all of our libraries; nothing to see here... Oh wait — didn't I say we have to use CommonJS because we're in Nodeville? No — because we're using Babel and Webpack to transpile all of our code, we can still use the lovely ES2015 module loading syntax without issue — even for Node.js modules that aren't themselves in ES6!

Anyway, we essentially instantiate Express, assign it to `app` and tell it to use the `urlencoded` form data body parser for `POST` requests (we installed this at the same time as `express`).

Add the following code:

```
app.get('/', (req, res) => { res.send('Hi there!'); });

app.listen(process.env.PORT || 8081, () => {
  console.log("We're up and running on " + http://localhost:(process.env.PORT ||
8081);)
});
```

This sets up a route for `GET` (that is, a normal visit to a page, as opposed to a form submission, for instance) on the application's root path. Then it starts listening for requests on either port `8081` or *whatever that $PORT environment variable is set to*. I've emphasized that last line because it's a key tenant in building web applications like this — keep all configuration in environment variables, so you never have to worry about storing plaintext passwords in your source code.

Note

What are these "environment variables" I keep mentioning?

Your shell keeps a number of variables persistent that it uses to do things — you might be familiar with `$PATH`, which is a list of directories the shell looks in for executable code. Environment variables can also be used by web servers to hold configuration details, which can then be consumed by web applications (such as the one we're making!).

One example of where this is useful is database connection details — you generally don't want to version control sensitive details such as database passwords, so instead you provide them to your web server as environment variables, which your application then uses to connect to the database. There are a ton of other benefits to this, but suffice it to say, making your applications configurable through environment variables is a key aspect of writing good JavaScript server applications.

Now go back to the command line and type the following:

```
$ npm run server
```

Then visit `http://localhost:8081` and you should be greeted with the following text:

Congrats, you are now a bonafide web applications developer!

Well, not quite, but, *rapidly getting there!*

One thing worth noting at this point is that all instructions in a closure execute simultaneously, just like in the browser. In this way, JavaScript is said to be "asynchronous" — by executing everything simultaneously, you can't rely on something to be blocking like you can in PHP, for instance. Although it's a bit hard to wrap your head around this when starting in Node.js, this has really exciting ramifications for server code, making it very easy and very quick to scale horizontally.

Let's do something way less lame, and let's break out a brand new D3 feature while we're at it.

Proximity detection and the Voronoi geom

A **Voronoi geom** chops a geographic shape into discrete regions around points, such that no section overlaps and the entirety of the area is covered. Anything within a particular point's section is closer to that point than to any other point. We're going to use this to figure out what the closest major airport is to your current location, which we'll supply via the HTML5 Geolocation API.

Replace your call to `app.get` in `chapter6.js` with the following:

```
app.get('/', (req, res) => {
  res.send(`<!doctype html>
    <html>
    <head>
      <title>Find your nearest airport!</title>
    </head>
    <body>
      <form method="POST" action="/">
        <h1>Enter your latitude and longitude, or allow your browser to check.
</h1>
        <input type="text" name="location" /> <br />
        <input type="submit" value="Check" />
      </form>
      <script type="text/javascript">
      navigator.geolocation.getCurrentPosition(function(position) {
        document.getElementById('latlon').value = position.coords.latitude + ','
+ position.coords.longitude;
      });
      </script>
    </body>
    </html>`);
});
```

Tip

You'll notice we put view code inside of our web server logic, which isn't great, but it works for our purposes. You'll want to use Express's views system for anything more elaborate than what we're doing.

If you haven't recently, restart the server by press *Ctrl + C* and then running the following:

```
$ npm run server
```

It's worth noting that, unlike in the frontend, we need to restart the server every time there's a change. If something isn't working as expected, try restarting it.

This sends the web browser a basic HTML document that asks them to fill in latitude and longitude as a comma-separated value. Or, if they accept the browser's request to use the HTML5 Geolocation API, it will auto-populate.

Next we need to create a new function for the Voronoi calculations. Inside of `helpers.js`, add the following:

```
export function nearestVoronoi(location, points) {
  let nearest = {};
  let projection = d3.geo.equirectangular();

  location = location.split(/,\s?/);

  let voronoi = d3.geom.voronoi(
    points.map((point) => {
      let projected = projection([point.longitude, point.latitude]);
      return [projected[0], projected[1], point];
    }))
    .filter((d) => d);

  voronoi.forEach((region) => {
    if (isInside(projection([location[1], location[0]]), region)) {
      nearest = {
        point: region.point[2],
        region: region
      };
    }
  });

  if (nearest === {}) throw new Error('Nearest not findable');
  else return nearest;
}
```

Lots and lots going on here; let's unpick it a bit.

We have two function arguments, `location` and `points`: `location` is a comma-separated latitude/longitude pair, `points` will be an array with all of our airports when we supply it from our server code in `chapter6.js`.

We then start things off by splitting our latitude/longitude string into an array, and setting up an equirectangular projection like we did with our first map all the way back in Chapter 3, *Making Data Useful*.

Finally, we get to configuring our Voronoi geom. Despite how complex this might look, it's pretty simple underneath — you ultimately just supply it with a bunch of points and it calculates the max size of each region. We do this via an `Array.prototype.map`, where we first project the longitude and latitude, then return these as an array. We also include the original `airport` object in the array so we know which point corresponds to which airport.

Tip

You can include as many additional array elements as you want and they'll all show up in each Voronoi region's `.point` object. The Voronoi geom only reads the first two elements of the array when constructing the Voronoi regions.

We also filter by the datum's identity, as follows:

```
    .filter((d) => d);
```

This is because the Voronoi geom will return a point as `undefined` if there are any duplicates (such as there are in this particular dataset), which will mess up other stuff (particularly if you want to then use a path generator to draw your Voronoi regions). This strips out all the undefined items.

Our Voronoi regions in hand, we then do a `.forEach` loop to check whether our user's location is inside of each region. Once we find which Voronoi region our point is interior to, we return that. If none are found, we throw an error.

The last thing we need to do is write the `isInside` function, which we'll also put in `helpers.js`. This is taken from `substack/point-in-polygon` on GitHub, which you can install via npm if you want to save yourself some typing:

```
export function isInside(point, polygon) {
  let x = Number(point[0]), y = Number(point[1]);

  let inside = false;
  for (let i = 0, j = polygon.length - 1; i < polygon.length; j = i++) {
      let xi = polygon[i][0], yi = polygon[i][1];
      let xj = polygon[j][0], yj = polygon[j][1];

      let intersect = ((yi > y) != (yj > y))
          && (x < (xj - xi) * (y - yi) / (yj - yi) + xi);
      if (intersect) inside = !inside;
  }

  return inside;
}
```

Without getting too deep into the preceding code, it ultimately draws lines out from a test point and looks at how many boundaries it crosses. It's based on the code from the following URL, which goes into much greater detail: https://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html.

Now we just need to wire this all up back in `chapter6.js`. Instead of a `GET` request, this time we're going to handle a `POST` request (which will contain our current location) to the same address.

```
app.post('/', (req, res) => {
  let location = req.body.location;

  let airportPromise = new Promise((res, rej) =>
    readFile('src/data/airports.dat', 'utf8', (err, data) => err ? rej (err) :
res(data)));
});
```

Pretty straightforward — when Express receives a `POST` request (such as when you submit a form), `body-parser` will intercept the form data and assign it to `req.body`. Next, we create a new

`Promise`, where we use `fs.readFile` to load in our airport database.

Note

We use `fs.readFile` because Node doesn't know how to `use d3.csv`, which uses `XMLHttpRequest`, which is only in the browser. Because the file we want is on the same computer as our server application, we can just load it in from the disk instead.

Let's resolve that promise and feed our helper functions some data:

```
airportPromise.then((airportData) => {
  let points = d3.csv.parseRows(airportData)
    .filter((airport) => !airport[5].match(/\N/) && airport[4] !== '')
    .map((airport) => {
      return {
        name: airport[1],
        location: airport[2],
        country: airport[3],
        code: airport[4],
        latitude: airport[6],
        longitude: airport[7],
        timezone: airport[11]
      }
    });

  let airport = nearestVoronoi(location, points);
})
.catch((err) => console.log(err));
```

This is pretty straightforward; we're just reformatting our data so it's a bit nicer to work with. Note that we filter out airports without both the airport codes, which means we'll only get large international airports.

Tip

It's worth noting the `airports.dat` data set not only includes airports ranging in size from tiny Moose Jaw Municipal Airport to the gargantuan Beijing International, but also includes some international rail stations — for instance, London St. Pancras International. Which, *I guess* kind of makes sense? Trains are *sort* of like ground planes, right?

After that, we put both our location and point arrays into our `nearestVoronoi` function, which we assign to `airport`. We also log any errors to the console via `Promise.catch`, because otherwise we won't have any idea what's going on if it fails.

The only thing left to do now is return a HTML document to the web browser if we're successful. Add this after `let airport`:

```
res.send(`<!doctype html>
<html>
<head>
```

```
  <title>Your nearest airport is: ${airport.point.name}</title>
</head>
<body style="text-align: center;">
  <h1>
    The airport closest to your location is: ${airport.point.name}
  </h1>
  <table style="margin: 0 auto;">
  <tr>
    ${Object.keys(airport.point).map((v) => `<th>${v}</th>`).join('')}
  </tr>
  <tr>
    ${Object.keys(airport.point).map((v) => `<td>${airport.point[v]}
</td>`).join('')}
    </tr>
  </body>
  </html>`);
```

It's not going to win any prizes for beauty, but it does summarize the data with a minimal amount of code.

Okay, go back to your terminal. Hit *Ctrl* + *C* to kill off Node.js if it was still running from earlier, and then run the following from the root of your project directory:

```
$ npm run server
```

It'll take a second for Webpack to churn through everything, but once it's done, visit `http://localhost:8081` to see a screen like the following:

Either enter in a latitude/longitude pair, or let your web browser take your current position instead. Click `Submit` to see the results:

The airport closest to your location is: City

name	location	country	code	latitude	longitude	timezone
City	London	United Kingdom	LCY	51.505278	0.055278	Europe/London

Hey, that's actually kind of cool! And we've figured something out using D3 that didn't even involve drawing anything!

By now you should be seeing D3 less as a bunch of magical tools that turn data into pretty visual things, and more just as a collection of functions that output mathematics a particular way. While D3 is certainly the most interesting when it's drawing things in a web browser, it's such a powerful library that you can use it for things far removed from its original use case of rendering SVG in the DOM.

Rendering in Canvas on the server

How about we do another one of those things right now? As mentioned before, the output from our little server app is pretty dull. Let's render a map using Canvas!

For this to work on the server, we're going to need to install `node-canvas`, which uses Cairo as a dependency. Assuming you're in Mac OS X and have Homebrew installed, run the following:

```
$ brew install pkg-config cairo libpng jpeg giflib
```

If you're not an OS X user with Homebrew installed, I suggest visiting https://github.com/Automattic/node-canvas and following the instructions there.

Tip

Alternatively, you can skip this entirely if you don't care to test locally, as we'll be deploying this all to Heroku later on in the chapter.

Next, add `node-canvas` to our app's dependencies:

```
$ npm install canvas earth-topojson --save
```

It's worth noting that this is a super weird way to use Canvas compared to how we can in the browser; normally we'd just rely on the browser's built-in Canvas renderer, but we don't have that luxury on the server. Note however, that the Canvas code we're writing for `node-canvas` will work the same way in the browser, in case you want to use it there.

We also install `earth-topojson`, which is a quick and dirty way of getting some basic political boundaries into a project.

Create a new function in `chapter6.js`:

```
function drawCanvasMap(location, airports) {
  let Canvas = require('canvas');
  let topojson = require('topojson');

  let canvas = new Canvas(960, 500);
  let ctx = canvas.getContext('2d');
  let projection = d3.geo.mercator()
    .center([location.split(/,\s?/)[1], location.split(/,\s?/)[0]])
    .scale(500);
}
```

This sets up `Canvas` and creates a new Mercator projection centered on the user's location. `ctx` is your canvas context; it's where and how you do your drawing.

Next, add the following to `drawCanvasMap`:

```
  let boundaries = require('earth-topojson/110m.json');
  let airport = nearestVoronoi(location, airports);
  let airportProjected = projection([airport.point.longitude,
airport.point.latitude]);

  let path = d3.geo.path()
    .projection(projection)
    .context(ctx);

  ctx.beginPath();
  path(topojson.feature(boundaries, boundaries.objects.countries));
  ctx.stroke();

  ctx.fillStyle = '#f00';
  ctx.fillRect(airportProjected[0] - 5 , airportProjected[1] - 5, 10, 10);

  return canvas.toDataURL();
```

The first three lines are what you'd expect — first we load in our boundaries, then we use our Voronoi function from earlier to calculate the nearest airport. From that we get a point, which we project into our coordinate system.

Then it's time to draw: we create a geo path generator, and supply `ctx` to it using the `.context` method. This is really pretty cool, because D3 does all the hard work in drawing paths in Canvas for us.

With our path generator doing its thing, we tell `ctx` to start drawing a path. We then run the path generator on our geometry, and add a stroke to it. Unlike what we normally do in D3, we don't chain these methods — we run them one right after the other, or *procedurally*.

We then draw a tiny little square on our closest airport, because that's how we do!

Finally, we use one of Canvas' super awesome features, which is outputting directly to a base64-encoded PNG data string. We return this data URL, because we'll be updating our `POST` method to output it.

Inside the call to `app.post`, add the following line above the call to `res.send`:

```
let canvasOutput = drawCanvasMap(location, points);
```

Then replace the call to `res.send` with the following:

```
    res.send(`<!doctype html>
      <html>
      <head>
        <title>Your nearest airport is: ${airport.point.name}</title>
      </head>
      <body style="text-align: center;">
        <h1>The airport closest to your location is: ${airport.point.name}</h1>
        <img style="width: 480px; height: 250px;" src="${canvasOutput}" />
```

```
        <table style="margin: 0 auto;">
        <tr>
          ${Object.keys(airport.point).map((v) => `<th>${v}</th>`).join('')}
        </tr>
        <tr>
          ${Object.keys(airport.point).map((v) => `<td>${airport.point[v]}
</td>`).join('')}
        </tr>
      </body>
      </html>`);
```

This simply adds an image tag, with the `src` attribute set to our data string. If you look at it by right-clicking and selecting **Inspect Element**, you can see what our function is actually outputting, as follows:

This is what an image rendered to a base64 string looks like

Hit *Ctrl + C* if you still have Node.js running and then restart it by typing the following:

```
$ npm run server
```

Open up `localhost:8081`, enter a lat/long pair, and now your results page has a handy little static map!

Pretty cool, huh? Canvas, despite being a bit weird to use in D3, is a super powerful technology, particularly when rendering huge amounts of data (any DOM-based display language tends to get really slow after about 1000 elements; Canvas is effectively just a 2D drawing plane, so never has that problem).

Deploying to Heroku

A server app isn't very useful without a server!

Luckily, **Heroku** provides free plans for limited use and is super easy to deploy to. At the moment, they allow 18 hours of uptime per day on the free plan, with your machine downcycling when it isn't active (in effect, this means that your server generally won't ever run out of uptime hours provided it isn't being hit with traffic constantly).

Start by creating an account at http://www.heroku.com and install the Heroku Toolbelt from http://toolbelt.heroku.com. Once you've done so, go to the root of your project folder and type the following:

```
$ heroku create
```

This will create a new Git remote and set up your app at a random URL, like https://calm-dusk-16214.herokuapp.com/.

Next, create a new file named `Procfile`. Heroku looks at this when you deploy to know how to run your app. Add the following contents:

```
web: node build/server.js
```

Save, then make sure you have the latest bundle built, as follows:

```
$ npm run server
```

Hit *Ctrl + C* after it builds; we don't need to run it any more.

Because the servers you get from Heroku are really basic by default, we need to find a way of installing Cairo and any other dependencies node-canvas might have. Run the following:

```
$ heroku config:add BUILDPACK_URL=https://github.com/mojodna/heroku-buildpack-multi.git#build-env
```

This tells Heroku to use a custom buildpack when deploying. A buildpack is effectively just a recipe for configuring a server a certain way. We define our buildpacks in a new file called `.buildpacks` (notice the dot at the beginning). Add the following to it:

```
https://github.com/mojodna/heroku-buildpack-cairo.git
https://github.com/heroku/heroku-buildpack-nodejsNode.js.git
```

Finally, add everything to a commit:

```
$ git add . && git commit -am "Time for Heroku"
```

We're now going to deploy! Assuming you're working from the master branch, type the following:

```
$ git push heroku master
```

Tip

Heroku always deploys from the master branch. If you've checked out the `chapter6` branch in Git, type the following instead:

```
$ git push heroku chapter6:master
```

Visit the URL provided by `heroku create` — and you should see your app in all its glory, online and accessible to anyone on the Internet! Congratulations, you've just written a pretty awesome webapp!

Didn't think you'd get a crash course on writing backend code in a book about data visualization, did you?!

Summary

In this chapter, first we set up Webpack to produce a separate bundle for the server, then we wrote a simple webapp using Express and D3's Voronoi geom to find the nearest airport to a user. We then upgraded our server app to draw a map using D3 and Canvas, which we then outputted to the user as a PNG.

Wasn't that all really pretty weird but also kind of fun? Writing server-side code is like that, but it can also be really nicely cathartic after spending a bunch of time doing frontend development, which tends to be really finicky due to having to support so many devices. If nothing else, hopefully you've begun to see how processing on the backend can improve the performance of your projects by offloading some of the processing work from the user's machine.

There's clearly quite a lot more you could learn about this topic that I just simply don't have room to cover here. We didn't go into scalability at all (which is super important when building things for large audiences, such as in the newsroom), nor into how to properly architect a non-trivial application — for that I'd recommend installing a few Express-based Yeoman generators and seeing how they scaffold projects (I'm particularly a fan of `generator-angular-fullstack`), or checking out Alexandru Vlăduțu's *Mastering Web Application Development with Express*, *Packt Publishing*, 2014.

You now have a pretty full toolbox for confronting a very wide array of data visualization challenges. In the next chapter, we'll add two more tools to it — unit testing and strong typing — in order to help you have more confidence in the work you produce.

Chapter 7. Designing Good Data Visualizations

Data visualization is a tool that can be used in many ways. As you've seen while building examples throughout the book, data visualization is sometimes used to communicate information in a novel or interesting way; sometimes data visualization provides clarity, other times it's just used to make cool things.

Regardless of whether you're a journalist wanting to highlight a change in GDP, a scientist needing to communicate the results of an experiment, or a software engineer looking to integrate visualization into a product, chances are you'll want data visualization that is clear, concise, and does not mislead. Although the examples in this chapter will mainly be from a news media context, many of the points we'll discuss apply in a similar way to data visualization in general.

In this chapter, we'll look at a few general principles to keep in mind while building data visualizations, and I'll give some examples of good data visualization as well. Note that I'm in no way a data visualization expert, per se — I'm a developer and a journalist with a degree of learned design experience, and my thoughts on what constitute "good data visualization" are very much influenced by my background in building explanatory data-driven graphics for titles like *The Times*, *The Economist*, and *The Guardian*. These are very fast-paced newsrooms, and the goal when visualizing data is generally to communicate the important bits of a dataset to an audience instead of letting them explore the data. Although you might not have the same demands in your use of D3 as that of a newsroom developer, much of what I'll be discussing also applies if you're using D3 in academia, publishing, or elsewhere — the skill of being able to quickly and succinctly communicate information is incredibly valuable no matter your profession.

With that caveat out of the way, let's get on with discussing what exactly comprises good data visualization.

Clarity, honesty, and sense of purpose

There are two big schools of thinking in terms of data visualization at the moment: there's the ultra-minimalist philosophy espoused by Alberto Cairo and Edward Tufte, where the primary goal of data visualization is to reduce confusion, and then there are those who use data to create beautiful things that uphold design over communication. If you couldn't tell by the title of this section, I generally believe the former is far more appropriate in most cases. As somebody wishing to visually communicate data, the absolute worst thing you can do is mislead an audience, whether intentionally or not — not only do you lose credibility with your audience once they discover how they've been misled, but you also increase public skepticism over the ability of data to communicate the truth.

Here's a contemporary example. In September 2015, the U.S. Congress held a hearing on Planned Parenthood, the American reproductive and women's health group. During the hearing, Republican congressman Jason Chaffetz showed the following chart, created by the anti-abortion group Americans United for Life:

There are many problems with this chart, not least the complete lack of *y* axes. PolitiFact redrew the chart with corrected axes and it came out like this:

Vox took it a step further and drew the rest of Planned Parenthood's services:

As you can see, while there has definitely been a decrease in cancer screenings and prevention services (as well as contraceptives, for that matter), and a slight rise in the number of abortions, there has also been a dramatic increase in spending for STI/STD treatment and prevention. As Alberto Cairo commented on the original chart:

> *"That graphic is a damn lie … Regardless of whatever people think of this issue, this distortion is ethically wrong."*

The public backlash about this one misleading chart led it to being named "2015's Most Misleading Chart" by Quartz. Regardless of what the creators of the chart originally intended to demonstrate with it, any hope of achieving that goal was obliterated once viewers felt they were being misled.

Note

For a more thorough discussion of everything wrong with Chaffetz's chart, I highly recommend the commentary by both PolitiFact and Vox, at http://www.politifact.com/truth-o-meter/statements/2015/oct/01/jason-chaffetz/chart-shown-planned-parenthood-hearing-misleading-/ and http://www.vox.com/2015/9/29/9417845/planned-parenthood-terrible-chart respectively.

In the preceding quote, Cairo makes an interesting point in that communicating data carries with it certain fundamental ethical requirements. This is how *data visualization* differs from *data art* — in the latter, what's ethically required of the *artist* is to purposefully and honestly communicate their emotions, beliefs, fears, and so on. This is a long way off from the ethical requirement of the *visualizer*, which is to communicate specific qualities of the data through visual methods. Taking this a step further, the ethics of data journalism compel the journalist to tell the audience what the data really means and how it relates to that audience.

In your projects, decide where your scope lies. Are you acting in the role of a data journalist, with a desire to walk the reader through a specific bunch of numbers and figures? Are you acting as a data visualizer, perhaps creating a dashboard designed to quickly and effectively summarize a very large, multivariate dataset? Or do you want to build something fun and entertaining that leverages data merely as a method by which to achieve that aim? All three of these roles are perfectly acceptable, and there is room for work ranging from incisively explained line-charts all the way through to *objets d'art* that give us a better understanding of our size and place in the universe. But whatever you do, be clear with your intentions and never mislead.

Helping your audience understand scale

A big part of visualizing data is conveying scale and differences in magnitude. The following few examples do this particularly well.

To start with, please view John Burn-Murdoch's graphic on high-speed elevators for the at http://www.ft.com/cms/s/2/1392ab72-64e2-11e4-ab2d-00144feabdc0.html.

The following screengrab doesn't really do it justice:

If the above were the live visualization, you would see the elevators in each building endlessly rise and fall, with a counter beneath tracking how many times the elevator has gone up and down while you were looking at the page. A nice bit of easing at the top and bottom makes you feel like the little magenta square traveling along the line is a real elevator, subject to physics in the same way a big metal cage rapidly moving up and down the world's tallest buildings would be. Although this printed version only communicates one dimension — the relative heights of each elevator and building — the interactive version is able to use animation to convey a second quality; that is, the speed of the

elevator. In this, one can really see the power of using the digital medium to communicate data in ways a static print version will never be able to. Scale is demonstrated by tying visual content to time; in this case, merely saying "the elevator can do a return trip in 2 minutes" would not be nearly as effective as demonstrating what "2 minutes" feels like to the reader.

Another good use of animation to explain scale is Hans Rosling's Gapminder project (http://www.gapminder.org/world/), which allows viewers to see how the world has changed over time. In his excellent TED talk about understanding the world from a data-driven perspective, Rosling discusses how, if looking at measures such as life expectancy and GDP per capita, the world is getting substantially better as time goes on, and that our perceptions of other countries are often rooted in a degree of ignorance as to how similar we generally are.

As time goes on, you can watch as the bubbles in the chart migrate from the bottom-left quadrant to the upper-right. It's worth comparing the fairly-dry exploratory visualization to the TED talk (https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen). Although

depicting similar data, the latter is far more enjoyable simply due to Rosling's narration, with excitement evident in his voice as he explains how the world is improving and changing as societies develop. Good explanatory data visualization doesn't have to be limited to text output.

Tip

You can see an example of the preceding chart recreated using D3 by visiting Mike Bostock's implementation at http://bost.ocks.org/mike/nations/.

A much more elaborate example of embracing the digital medium to help convey scale is The Guardian's *Homan Square: A Portrait of Chicago's Detainees* interactive by Spencer Ackerman, Zach Stafford and the Guardian US interactive team. If you haven't seen it, please visit and prepare to have your mind utterly blown: http://www.theguardian.com/us-news/ng-interactive/2015/oct/19/homan-square-chicago-police-detainees.

It starts with a big collection of faceless grey silhouettes, each representing a single person in custody at a secretive police warehouse in Chicago:

As you scroll, the faces fly around the screen to make up different configurations, such as this bubble

chart:

And even this isomorphic map:

Note

The Guardian's U.S. interactive team did a fantastic Q&A about the process behind this visualization that is well worth reading — https://source.opennews.org/en-US/articles/how-we-made-homan-square-portrait/.

Although the data morphs in many different ways, you still feel an attachment to each point, remembering how they were originally displayed — they're not merely pixels on a screen; each portrait represents another person whose life has been impacted (a feeling reinforced when they single out individual portraits as case studies later on in the piece).

These examples are phenomenal for communicating scale to the reader. One of the biggest reasons data visualization is such a powerful tool is that it helps answer the questions "how big is 'big'?" and "how small is 'small'?" As reporting becomes increasingly reliant on data, it becomes very easy to mislead the reader by over- or under-emphasizing scale (indeed, this is a big reason why the chart in the preceding section was considered so dishonest). This can often be mitigated by designing visual output in such a way that the viewer feels some attachment to the visual stimuli onscreen.

As a final example, please visit this graphic by Ana Becker of *The Wall Street Journal* that attempts to visualize what your chances are of winning the Powerball lottery jackpot — it's another one where the following screenshots really won't do it justice. Also, try not to hit the **jump ahead** button for at least a little while, http://graphics.wsj.com/lottery-odds/:

After a while of continuous scrolling, the screen starts to look like this:

At this point, you realize there's no chance that you're going to ever physically scroll to the bottom, and so either click the **jump ahead** link or grab the scrollbar. If at that point you're still convinced you can plan your retirement based on your lottery earnings, probably nothing will change your mind.

In a sense, scale is emphasized through comparing the initial "coin-toss" probability to the probability of picking the winning Powerball numbers. Comparing the coin-toss's 1 in 2 with the Powerball's 1 in 292,201,337 is impossible to do in one screen (and especially so in print), because no matter what sort of scale you use, the latter completely dwarfs the former. Making use of the browser's practically unlimited vertical screen real estate is a very effective way of tying visual elements to a physical property (much like the elevator speeds interactive earlier), insomuch that the physical effort involved in scrolling down to just 1 percent of the page (much less, count how many dots that is) very effectively demonstrates how mind-bogglingly big a number like 292m is in the context of comparing probabilities.

Using color effectively

One of the biggest choices you'll make when building data visualizations is choosing what colors to use to represent what. While it's certainly possible to convey a lot of information through purely monochromatic charts, using the wide range of color representable through a digital display can be a very effective way of depicting another property dimension of the data you're visualizing.

If you plan to use color to communicate information, there are a few things to consider. The first is whether a user will be able to discern the pattern necessary to understand what the colors mean. If you have a legend explaining what color corresponds to what, try turning it off and thinking about whether you're still able to understand what the color combination means. Is it intending to show increasing intensity? Diverging values? Or just that it has a particular quality (in which case, do you really need a legend)?

Secondly, pay some attention to people who are colorblind. The most common color combination to use for choropleth maps is generally the "stoplight" color scheme: green for low values, yellow for medium values, and red for high values (or the inverse, depending on whether a high value is a good thing or not). There's a problem with this, though — for people who are red-green colorblind, the highest and lowest values look nearly identical.

Also, while still on the topic of stoplight colors, when discussing sensitive topics like immigration, a fair degree of care should be taken when color-coding anything green or red — if red means "bad" or "severe", is it actually that way? Or is it displaying red merely because the color scale has been constrained to the values in the data set? Colors are much more emotive, and it's easy to unintentionally present an opinion when using them. In general, using a sequential color scheme (scaled to the average of what one would expect that value to be) is a much safer bet.

A very good way of finding a color scheme for a map (and quite a lot else) is using ColorBrewer (http://colorbrewer2.org), a tool built by Cynthia Brewer and Mark Harrower at Penn State. It provides a bunch of different pre-built color schemes for representing data, and you can choose the type of relationship you want to depict. Additionally, it helpfully allows filtering out color schemes that aren't colorblind-friendly, or even printer- and photocopier-friendly.

Tip

You can use ColorBrewer schemes directly in D3! There's a file in the D3 repo at `lib/colorbrewer/colorbrewer.js`, but at the time of writing it has been removed in releases and isn't really easy for us to use with modular JavaScript. Instead, install it separately via:

```
$ npm install colorbrewer --save
```

And then `require` it as normal. For the above **7-class BuGn** scale (look next to the **EXPORT** pane for the scheme's name), you'd write:

```
let BuGn = require('colorbrewer').BuGn[7];
```

Lastly, listen to natural cues from your data: if you're doing a visualization of political parties, it makes sense to color-code the data at least somewhat similarly to the party colors (as boring as it is to always color-code political data to party colors, the novelty of not doing it this way is easily outweighed by the cognitive dissonance it causes).

Understanding your audience (or "trying not to forget about mobile")

Your audience is one of the most critical things to consider when beginning a new data visualization project. This has two parts: the first is from an editorial perspective (what is the audience's background knowledge of this topic? What types of charts will the audience be able to recognize and properly read? How do these charts work within the broader contexts of this story and other work published?), while the second is technological (what platforms and devices will be used to consume this content?).

It's really important to tentatively sketch out any bespoke data visualization before you start writing code, and this can take many forms. On one hand, it never hurts to figure out the rough shape of your data before committing to a particular visualization style — frequently I get asked for pie charts with a few small outlier values highlighted, which doesn't work (the rest of the chart dwarfs the outliers). You don't necessarily need to get a pencil and paper out for this — pasting your data into Excel and playing with its default charts often helps before committing the data idea to code.

The second way you should sketch out your visualization is on a component or an interaction level. This is where understanding which devices your audience use is important. If you have previous work out there, look at its analytics. To see what percentage of readers use a specific browser in *Google Analytics*, look under **Audience** | **Technology**. Pay attention also to **Audience** | **Mobile** | **Overview**, which will tell you what proportion of your audience is comprised of mobile and desktop. If you have no analytic data to work with (for instance, if you're launching a new project), it's generally a smart bet to assume that half of your audience will be on mobile, and so you should design your project accordingly.

Developing for an array of screen resolutions is called **responsive design**, and there are two major workflows: mobile-first and desktop-first. Traditionally, designs begin at desktop size and are then scaled down for mobile, but this often means that mobile support is an afterthought and it's often problematic to shrink down large elements later on. Mobile-first design starts at the smallest possible size and scales up, which is often easier as you get the more difficult versions of the site designed first and out of the way, and you can let things grow as screen resolution increases. Whether you need to do either depends on your audience — it's quite possible it predominantly uses one platform or the other, reducing the need to accommodate both; again, check your analytics. That said, while ensuring things work well cross-device and cross-browser takes a lot of additional work, it should be a standard of excellence that you strive towards as you create things for broader consumption.

If creating a mobile-first design, take out some paper and a black marker and draw a bunch of phone-sized shapes. Draw squares where you want each user-interface element (buttons, dropdown boxes, radio buttons, and so on) or chart to go — how you distinguish each element from the others is entirely up to you, just make sure that whatever vocabulary you use is shared by those you work with. You don't need to be super artistic with it, just be detailed enough to express how you think each user

interaction (clicking/tapping, swiping, dragging, and so on) should *feel* within the project. Sketch out roughly how you think each element should *flow* on the different devices. Then do the same for a larger desktop screen size.

A rough drawing versus the finished page. Prototypes are meant to be discussion pieces you can use to solicit feedback from colleagues and refine throughout the development process. Don't worry if the end result looks nothing like your original sketches, or if (like me) you can't draw a straight line to save your life.

If possible, run your pen-and-paper prototypes past a few people to see whether your user interactions feel natural.

Some principles for designing for mobile and desktop

Mobile and desktop computing differ in some key ways, and understanding these is crucial for building data visualizations that are effective on both platforms.

Mobile:

- Has multiple screen orientations.
- Has uniformly-small screen dimensions.
- Does not have a pointer; interaction is derived from touch gestures. There is no "hover" state.
- Relies on often inconsistent data availability.
- Has significantly less computing power than comparable laptops.
- Keyboard is visible on-screen when in use; it is not used for navigation.

Desktop:

- Generally has only one screen orientation.
- Has variably-large screen dimensions (and is often connected to huge screens).
- Uses a pointer (and keyboard) to interact. The "hover" state is usable.
- Generally has reliable data availability
- Has significantly more computational power than comparable phones and tablets.
- Keyboard can be used without impacting what's visible on the screen; additionally, can be used for navigation.

I really don't have the space to go into a full course about how to do responsive design right here, but a few basic principles to keep in mind will get you started.

Columns are for desktops, rows are for mobile

On some level, it's fairly safe to assume that most people will view your work in portrait mode on mobile. This means that you effectively have one column, and every element in your one column is in a full row extending perpendicularly to the column's direction. Paragraphs should always flow vertically in a single column (instead of in multiple columns positioned left to right), and it's not an awful idea to rotate bar charts 90º so that your bars aren't squished together in one narrow horizontal row. Much like the *WSJ* lottery probabilities graphic above, make sure use of the infinite vertical space you have to scroll.

On desktop, however, a single paragraph spanning the entire horizontal width of the display is very hard to read, as it requires the eye to move back and forth quite a lot. Using columns is often preferable, as it not only makes better use of the horizontal space, but it also improves readability. Form elements in particular often look much better when grouped into columns.

The good news is that Flexbox makes it really easy to switch the orientation of groups of elements, provided you don't have to support older versions of Internet Explorer. If you aren't using flexbox already, take the effort to do so — it will make your life so much easier (at least, once you figure out

how to use it).

Tip

Want to learn flexbox the fun way? Try out Flexbox Froggy, a game that teaches you how to position elements using Flexbox and Frogs. http://flexboxfroggy.com/.

Still find flexbox hideously frustrating? You're in good company. Try out Flexbox Grid, which uses Twitter Bootstrap-like classes to create responsive grids using flexbox. http://flexboxgrid.com/.

Be sparing with animations on mobile

Animation on mobile is really tricky, not only because you have reduced graphics processing power, but also because scroll events have traditionally messed with JavaScript execution timing. If you don't totally disable animation on mobile, try to only use CSS transitions, as these are more performant than iterating through properties via JavaScript. When in doubt, disable animation on mobile.

Realize similar UI elements react differently between platforms

Things like radio buttons, sliders, and checkboxes are available both on mobile and desktop, but some are easier to use on mobile than others. In general, web browsers draw all of these elements slightly too small for comfort on most mobile devices. Where possible (for instance, select dropdowns), make individual form elements stretch the entire device width on mobile, or in the case of things such as checkboxes and radio buttons, use the `for` property of the `<label>` element to make labels tap-able and scale these horizontally.

Avoid "mystery meat" navigation

Pointer-based devices have the benefit of being able to use the cursor's "hover" state to reveal information (for instance, labels on buttons). While this can allow for more minimalist-looking interfaces on desktop, it's a really terrible anti-pattern referred to as "mystery meat navigation" when on mobile. When you hover over a button, you're not committing to clicking it. However, because mobile devices lack a hover state, users must commit to a user interaction to understand what a button actually does. Given how slow the mobile reading experience can be, users tend to be a lot more cautious before committing to any action that might cause the page to reload.

The solution is simple: unless using incredibly clear iconography, label your buttons on mobile.

Tip

For some good mobile icons, try Font Awesome (https://fortawesome.github.io/Font-Awesome/) and Google's Material Icons (https://design.google.com/icons).

Be wary of the scroll

A common user interaction idiom on desktop (popularized by the *New York Times'* "Snowfall" long-

form piece) is to tie animations to the browser's scroll event. This is frequently fraught with peril on mobile, because scrolling triggers a memory-intensive redraw that often blocks JavaScript execution. While newer mobile operating systems handle this better than before (I'm looking mainly at you, iOS), it's often unsafe to assume that a scroll-dependent animation will play properly and not feel non-performant and "janky" on mobile. Again, this has improved quite a lot with newer devices, but it's still never a bad idea to tie animation states to tap events (possibly delivered via a button) on mobile.

Summary

Hopefully by now, you feel like you have a secure understanding of the work you want to do using D3 to visualize data. We went through some examples and laid some good ground rules for building high-quality data visualizations that not only inform an audience, but also look pretty spectacular while doing so. We also discussed how to make sure your work functions well on mobile.

In the next chapter, we'll tie things up by helping you feel more confident with the visualizations you produce by using type checking and automated testing to ensure that nothing goes wonky at the worst possible time. Stay tuned…

Chapter 8. Having Confidence in Your Visualizations

When you're building things for as big an audience as the Web provides, a very real fear is that a software glitch prevents data from displaying correctly. When developing projects on a tight timeline, testing is one aspect that often gets neglected as the deadline gets closer and closer, and often things need to be viewed by other people even earlier (editors, managers, and other people further on down the line — possibly even lawyers) in turn emphasizing output over process. Let me be clear—if you want to ensure that your visualizations are of a high quality, you need to take steps to make sure that they are well tested and functioning properly. On some level, doing this is an exercise in managing complexity.

The next chart depicts project complexity over time. As you can see, complexity increases somewhat exponentially as time passes by. Adding more team members, more lines of code, and/or more dependencies increases the project's complexity dramatically. Meanwhile, the step chart depicts how tooling processes improve in response to complexity. Although it's possible to implement all of these at the beginning of every project, it's often better to do so incrementally in response to the project's demands. For instance, generally everyone will start with version control because it helps in collaboration, can revert mistakes, and provides a project with history (plus it's really easy to set up). Then, say you add a few more team members, or make a project open source. Now you have code flying at you from all directions. Having a way to automatically test whether that code will break anything or not starts to become incredibly useful.

A bit later, imagine a lot of bad code still getting past the automated testing; having manual code reviews might help this further. Each of these improvements to the process take time to both implement and use, and whether they'll benefit your project or not is hugely dependent on how big your team is and how well you trust every member of it (or, if working alone, how well you trust yourself not to introduce errors).

As time passes by, complexity increases somewhat exponentially, but tooling increases in a step pattern; source: Martin Probst and Alex Eagle, https://youtu.be/yy4c0hzNXKw?t=245

The preceding chart is from Martin Probst and Alex Eagle's talk on TypeScript at AngularConnect 2015. They touch upon a lot of the same topics as this chapter and it's worth watching (https://www.youtube.com/watch?v=yy4c0hzNXKw):

- In this chapter, we'll focus on a few technologies that help manage project complexity. We start by talking about linting. **Linting** is when you run your code against preset rules to ensure that it conforms to a set of standards.
- Then we will move on to **static** type checking. This is a tool for ensuring that the data flowing through your application doesn't act unexpectedly.
- We will end this chapter with **automated** testing. This is what it says on the tin: you write tests for your code that must all pass. Failing tests can be used to diagnose bugs or other issues with the code.

You honestly don't need to do any of these things to make stuff with D3, but getting into the habit of using these tools when necessary will both improve the quality of your code and make you a better developer. Let's dive in!

Linting all the things

A **linter** is a piece of software that runs source code past a set of rules and then causes a stink if your code breaks any of those rules. Now, I know what you're thinking: "my boss/manager/editor/significant other is *already* giving me more feedback than I'd care for. Why do I need *yet another* thing to do that?" Glad you asked!

Linting rules are often based on industry best practices, and most open source projects have a customized rule set corresponding to their community guidelines. This simultaneously ensures that code looks consistent even when delivered by a multitude of people and lets contributors know when they're doing something that is a little confusing or error prone in their code. Note, however, that all of these rules are just opinions—you don't *have* to write your code following them, but it tends to help everyone else out if you do.

If you've been following along with the GitHub repository for this book, then, perhaps, you've noticed a hidden file called `.eslintrc`, or noticed `eslint` in `package.json`. ESLint is currently the best linter for ES62016 code, and it works very similarly to its predecessors, JSHint and JSCS. It is configured via the `.eslintrc` file, which contains both the specific rules to be checked against (or a set of defaults to extend, such as what we've used) and information about the code's environment (for instance, NodeJS has different global variables than the browser).

To run ESLint against the current project, type this line:

```
$ npm run lint
```

Although there won't be any linting errors in the repository by the time this book goes to print, here's an example of what the output looked like at the writing stage of the book:

```
> learning-d3@1.0.0 lint /Users/aendrew/Sites/learning-d3
> ./node_modules/eslint/bin/eslint.js src/*.js


/Users/aendrew/Sites/learning-d3/src/chapter2.js
  44:12  error  "d" is defined but never used  no-unused-vars

/Users/aendrew/Sites/learning-d3/src/chapter3.js
  139:11  error  "lines" is defined but never used  no-unused-vars
  379:9   error  Unexpected constant condition      no-constant-condition
  387:9   error  Unexpected constant condition      no-constant-condition

/Users/aendrew/Sites/learning-d3/src/chapter4.js
   80:9   error  "timer" is defined but never used  no-unused-vars
  178:30  error  "rej" is defined but never used    no-unused-vars
  185:30  error  "rej" is defined but never used    no-unused-vars

/Users/aendrew/Sites/learning-d3/src/chapter5.js
  227:53  error  Irregular whitespace not allowed  no-irregular-whitespace
  338:9   error  "link" is defined but never used  no-unused-vars
  354:33  error  "d" is defined but never used     no-unused-vars
  357:32  error  "d" is defined but never used     no-unused-vars

/Users/aendrew/Sites/learning-d3/src/voronoi-airports.js
   3:25  error  "location" is defined but never used  no-unused-vars
  13:7   error  "path" is defined but never used      no-unused-vars

✖ 13 problems (13 errors, 0 warnings)
```

Tsk-tsk! What a lot of errors we have here!

We use npm to run ESLint in this instance, which is effectively an alias for the following line:

```
$ node ./node_modules/eslint/bin/eslint.js src/*.js
```

You can also install ESLint globally and then just run it anywhere.

Although this is helpful, linting is way more useful when you see it all the time while developing. Let's make ESLint scream at us while we're using `webpack-dev-server`. First, install `eslint-loader`:

```
$ npm install eslint-loader --save-dev
```

Next, we add `eslint-loader` as a preloader to our Webpack config, so it runs on our code *before* Babel does its thing with it. In `webpack.config.js`, add the following code in the module section of each build item:

```
preLoaders: [
  {test: /\.js$/, loader: "eslint-loader", exclude: /node_modules/}
]
```

Then run `webpack-dev-server`:

```
$ npm start
```

Ta-da! Now, you can get instant feedback about how messy all of your code is every time you hit save! You're utterly thrilled by this. I can feel it!

Linting is a very light way of managing complexity. Generally, a linting failure won't necessarily cause anything other than a message to appear on the developer's screen. It's up to you and your team to maintain a sense of discipline in terms of not committing code until it passes linting.

Static type checking with TypeScript and Flow

Static type checking is where you have a process that looks at how variables are being used, and then throws a wobbly if you do something weird. By this, I mean that it looks at the *type* of each variable and uses *type annotations* (bits of text defining what type a variable is when the variable itself is defined) to ensure that functions don't mutate a variable in an unexpected way. This is called **static typing**, and it is a feature built into many robust languages, such as C++ and Java. While JavaScript's dynamic typing (also shared by lots of other web languages, such as PHP and Ruby) is helpful in some ways and enables a certain style of programming, it can also be incredibly frustrating due to its ability to introduce silent errors. Because we're using a transpiler to transform our JavaScript anyway (throughout the book, this has been Babel, though that doesn't necessarily have to be the case), we can introduce static type checking to JavaScript at the same time if we so want.

There are two big players in statically typed JavaScript right now: Facebook's Flow project and Microsoft's TypeScript. Which you use is somewhat dependent upon when and how you plan to employ type checking, though my personal favorite is TypeScript for reasons that I'll get into later.

Although it's a level of tooling complexity above both version control and linting, in all honesty, to get the most from any of those three tools, it really helps to use them from the outset and be really disciplined with their use (particularly if you have two or more developers on your project). Flow is nice in that it is much easier to work into a Babel-based workflow than TypeScript, but TypeScript has a ton of benefits that extend beyond just ensuring that types don't change, and is particularly well-suited for working with D3.

Tip

Warning!

Lots of boring details about configuring stuff ahead! Wait, though! I have a solution!

The following sections will describe how to get started with either TypeScript or Flow, and give a bit of light config info, so you can get a sense of how it all fits together. Alas! Configuring things is really dull and should be avoided. If you want to dive right into playing with this stuff, you have two options. You can simply check out the `chapter8` branch of the book repository by doing the following:

```
$ git stash save && git checkout origin/chapter8
```

Alternatively, you can install my handy `strong-d3` Yeoman generator, which will quickly scaffold out a brand new D3 project using either Flow or TypeScript. To install it via npm, run this line:

```
$ npm install --global generator-strong-d3 yo
```

Create a new directory for your project, switch into it, and run the generator, replacing `myProject` with whatever you want to call what you're working on:

```
$ mkdir myProject && cd myProject && yo strong-d3 myProject
```

Answer the questions it asks and you're on your way! For full usage instructions, visit github.com/aendrew/generator-strong-d3.

The new kid on the block – Facebook Flow

If you have an existing ES2016 project and you want to introduce type checking into it retroactively, Flow might be the ticket. Unlike TypeScript, which totally replaces Babel, Flow lets you selectively add type checking to your projects. In many cases, this is all you'll need.

We're not actually going to use Facebook's normal Flow implementation (which starts up a whole server and is way too heavy for our uses); instead, we're going to use a Babel plugin that uses Flow-style syntax, `babel-plugin-typecheck`. Install that first:

```
$ npm install --save-dev babel-plugin-typecheck@2
```

Next, change your `.babelrc` file to the following:

```
{
  plugins: ["typecheck"]
}
```

Let's test this out. Start up `webpack-dev-server` with this command:

```
$ npm start
```

Then open `index.js`. Clear it out and add the following code:

```
function giveMeANumber(base: number): number {
  return base * Math.random();
}
alert(giveMeANumber('hi!'));
```

What we do here is create a function that multiplies a random number by whatever number the function is supplied. As you can see, the function declaration looks a bit different:

```
function giveMeANumber(base: number): number {
```

What we're doing here is saying that the argument base should be a number and the function should always return a number. We then try to completely disregard what we've just said it should be and try to supply the function with a string:

```
alert(giveMeANumber('hi!'));
```

If you open this up in your web browser, your console should let you know you've messed up:

```
▶Uncaught TypeError: Value of argument 'base' violates contract, expected number got  index.js:1
string
```

Although this is a truly trivial example, imagine you had a massive project with tons of code, and this

was your first day as a new developer on the project. Instead of possibly introducing a bug while learning the ins and outs of the code, Flow will smack you down straightaway, letting you know that you're using the code wrong. Plus, because the type definitions are right in the code, you can go straight to that function, look at the type annotation, and realize that you need to give it a number instead of a string. Combining it with a modern editor that lets you do this in a keystroke results in a very nice development experience.

TypeScript – the current heavyweight champion

That development experience is made even better with TypeScript, which is somewhat more stable and more widely used than Flow (at least at the time of writing this book). There are many upsides to using TypeScript, which I'll get into in just a moment. But before that, be forewarned that there's slightly more effort involved with getting TypeScript set up. This is because it's a transpiler in its own right (instead of just a type checker), which means we can't use Babel anymore.

The good news, however, is that TypeScript compiles modern JavaScript very similarly to how Babel does, and we just need to tweak a few things before we can begin using it. In practice, unless you're using a lot of Babel plugins, you probably won't notice the difference between the two. Further, we're actually going to use both simultaneously, importing TypeScript modules into Babel and vice versa like it ain't no *thang*.

First, let's install some more stuff. Modern JavaScript development is like 50 percent coding, 20 percent being confused about what libraries to use, and 30 percent installing those libraries. So let's get to it:

```
$ npm install typescript@1.8 ts-loader --save-dev
```

This will install both the TypeScript transpiler and the Webpack loader. Next, we're going to install a utility called `typings`, which helps us download prewritten type definitions for libraries such as D3:

```
$ npm install --g typings
```

Next, generate a `typings.json` file:

```
$ typings init
```

Then install the D3 TypeScript definition:

```
$ typings install d3 --save
```

Next, let's update our Webpack config. Under `loaders`, add the following:

```
{
  test: /\.ts$/,
  loader: 'ts-loader'
}
```

This will use TypeScript to load all files ending in `.ts`. You can go back through all your old files and rename them to `.ts`, or you can just do as we are doing here and use *both* Babel and TypeScript. Once you get used to TypeScript, you'll probably want to start with it from the beginning, but this time around, we're going to mash up both ES6 and TypeScript for great awesomeness.

First, let's add some lines to `index.js` to import our forthcoming TypeScript class and instantiate it with some data:

```
import {TypeScriptChart} from './chapter8.ts';
let data = require('./data/chapter1.json');
new TypeScriptChart(data);
```

Importing a TypeScript file into Babel is now as difficult as specifying the `.ts` extension.

Next, create a new file in `src/` called `chapter8.ts` and add this code at the top:

```
/// <reference path="../typings/main.d.ts" />
import * as d3 from 'd3';
```

This both includes your TypeScript definitions and imports D3 from `node_modules`. From here, we can start using TypeScript as we would normally.

Tip

One worthwhile thing to do at this point is to create a `tsconfig.json` file, which is a TypeScript config file. With this, you can prevent a number of annoying behaviors from the TypeScript transpiler, and some TypeScript IDE integrations make extensive use of it (particularly `atom-typescript`). For an example `tsconfig.json` file, look at the `chapter8` branch of the book's repository.

Let's create a brand new chart, using our old friend `BasicChart` as a base. "But wait a minute, that's not in TypeScript?" you ask. Eh, give it a shot anyway!

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

Run Webpack:

```
$ webpack
```

And you'll get the following error:

```
    ERROR in ./src/chapter8.ts
    (2,26): error TS2307: Cannot find module './basic-chart'.
```

Eh, was worth a try! How do we solve this?

Create a new file in `src/` named `basic-chart.d.ts`. We are going to write a really simple, ambient type definition for our `BasicChart` class, which will allow us to add type checking to our existing class code without changing it or converting it to TypeScript. This is similar to how we consume D3 with TypeScript—we use the `Typings` utility to install D3's TypeScript definition, which gives us all the advantages of TypeScript with D3, all without any extra effort on Mr. Bostock's behalf (the definition is maintained separately by the fantastic folks at a project called `DefinitelyTyped`).

Here it is in one go:

```
import * as d3 from 'd3';
export declare class BasicChart {
  constructor(data?: any);
```

```
  data: Array<any>|Object;
  svg: d3.Selection<SVGElement>;
  chart: d3.Selection<SVGGElement>;
  width: number;
  height: number;
  margin: {
    left: number;
    top: number;
    right: number;
    bottom: number;
  }
}
```

This probably looks a bit different from how we've been doing things, so bear with me.

First off, we import everything from the `d3` module and call it `d3`. This is how you import D3 3.5.x as an ES2016 module, by the way. Although it didn't matter whether we used the shorter `require()` when our code was transpiled with Babel earlier in the book, in TypeScript you really want to use the ES6 module syntax.

Next, we declare the `BasicChart` class and export it. All we do then is list its properties (and methods if it had any), noting what types each property should take. All the numerical properties at the end of it are pretty self-explanatory. Let's take a quick look at the first four:

```
  constructor(data?: any);
  data: any;
```

We put our constructor and the arguments it takes. It has one optional argument (specified by the question mark), and we're not really prescriptive in terms of what it should be, so we use the `any` type. Because we then directly assign the argument to the class' `data` property, I've set that as `any`.

Tip

If you set `compilerOptions.noImplicitAny` in your `tsconfig.json` file to `false`, any variable without a type definition will be given the `any` type. If you're having trouble getting your code to work with types initially and just want to add types to existing code on an ad hoc basis, setting `noImplicitAny` to `false` might be something worth trying.

If we did some transformation on the data first—changing it to, say, an array of numbers—we can write something like this:

```
data: Array<number>
```

We supply the number argument to the `Array` interface, which is what we're doing with these weirdo angled brackets.

Similarly, we define the `svg` and `chart` properties using the D3 selection interface, containing `SVGElements`:

```
  svg: d3.Selection<SVGElement>;
  chart: d3.Selection<SVGGElement>;
```

Tip

Where on earth did I get `d3.Select<SVGElement>` from? Although we could have just specified these as the `any` type as an easy way out, TypeScript is more powerful when you give variables as specific a type annotation as you possibly can. For instance, the D3 definition declares several interfaces corresponding to the various types of variables created by D3, `Selection` being only one.

Throughout this book, I've attempted to be very neutral on the topic of editing environments and IDEs, not caring whether you use **Atom**, **Sublime Text**, **vi**, or something awful like Notepad. However, you *really really* should use an IDE if you're going to work with TypeScript. I use Atom (with the `atom-typescript` plugin). My environment tells me what interfaces the D3 module has while I'm typing, and I can just browse through them. Not only that, the autocomplete will tell me what arguments to specify for the functions I'm using, because a TypeScript definition effectively acts as machine-readable API documentation for the editor. It is super helpful!

Save and run Webpack again. It should compile without issues. Go back to `chapter8.js`, and we'll start filling out our new chart class. Let's kick it old school and redo the first chart that we did in this book as TypeScript.

```
import * as d3 from 'd3';
import {BasicChart} from './basic-chart';
export class TypeScriptChart extends BasicChart{
  constructor(data: Array<ITypeScriptChartDatum>) {
    super(data);
  }
}
```

Okay, this looks familiar. But what's that `ITypeScriptChartDatum` thing? Things that start with a capital `I` in TypeScript are generally interfaces, or reusable sets of typings. As you can see, our constructor takes an array of these. Let's create that interface now. Put this code at the bottom of the file, outside of your class:

```
interface ITypeScriptChartDatum {
  population: Array<IPopulation>;
  name: string;
}
```

If you remember, our data is comprised of an array of objects containing a name string and an array of population measurements. We don't need to create a new interface for the population array items, but it makes things a bit more readable if we do. Add the following code afterwards:

```
interface IPopulation {
  module_name: Array<any>;
  module_type: string;
  value: string;
  demography: {
```

```
    "04M": string;
    "04F": string;
    "511M": string;
    "511F": string;
    "1217M": string;
    "1217F": string;
    "1859M": string;
    "1859F": string;
    "60M": string;
    "60F": string;
  }
}
```

We go back to our constructor, and let's start fleshing out our class a bit more. Let's define our class' `data` property as an array of objects that are strings and numbers. Below the constructor, add this line:

```
data: Array<{name: string; population: number}>
```

It's good to do this because otherwise TypeScript will see the `any` type we used in our parent class and not know how the data is structured (as a result, it will need a bunch of type annotations in things such as D3 data accessors, which gets annoying).

Let's try to directly assign our constructor argument to `this.data` inside our constructor, as follows:

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

If you run the TypeScript compiler now, you'll get the following error:

```
    ERROR in ./src/chapter8.ts
    (15,7): error TS2322: Type 'ITypeScriptChartDatum[]' is not assignable to
type '{ name: string; population: number; }[]'.
      Type 'ITypeScriptChartDatum' is not assignable to type '{ name: string;
population: number; }'.
        Types of property 'population' are incompatible.
          Type 'IPopulation[]' is not assignable to type 'number'.
```

Note that TypeScript, even if it's screaming at you for whatever reason, will still compile the JS files, which is why sometimes it works even if the compiler throws errors.

But not today! Let's munge that data like it's going out of style! Replace that last line with:

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

No more errors! Alright! Let's set some happy little scales inside the constructor:

```
this.x = d3.scale.ordinal().rangeRoundBands(
  [this.margin.left, this.width - this.margin.right], 0.1);
this.y = d3.scale.linear().range(
  [this.height, this.margin.bottom]);
this.x.domain(this.data.map((d) => d.name ));
this.y.domain([0, d3.max(this.data, (d) => d.population)]);
```

Nothing new here, except that now we're throwing a TypeScript error, saying that the `x` and `y` properties do not exist. What?!

What we need to do here is define the types of these class properties. Just as with `data`, we need to tell TypeScript what form these should take. Add the following two lines beneath the constructor but still inside the class:

```
x: d3.scale.Ordinal<string, number>;
y: d3.scale.Linear<number, number>;
```

What we do here is use the `Ordinal` and `Linear` interfaces for D3 to let TypeScript know what kinds of scales these are supposed to be. Because our *x* axis is ultimately going to be a bunch of place names mapped to a location on the screen, we provide `string` as the domain argument and `number` as the range argument (the argument signature for the `Linear` interface helpfully provided by my IDE's autocompletion). Meanwhile, the *y* axis is a linear scale that maps numbers to numbers. Cool, no more errors!

Let's add the axes:

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

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

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

Nothing surprising here at all! Finally, let's draw the bars:

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

```
    this.bars.transition()
      .delay((d, i) => i*200)
      .duration(800)
      .attr('y', (d) => this.y(d.population))
      .attr('height', (d) => this.height - this.y(d.population));
```

Because we're saving the reference to our bars selection as a class property, we need to define what it is. Beneath the two scale type annotations, add the following code:

```
bars: d3.Selection<{name: string; population: number}>;
```

Woo! We've built our first strictly typed D3 chart using TypeScript!

This was a ludicrously shallow overview of TypeScript that barely scratched the surface, but hopefully, you're using a good editor and are already feeling how powerful it can be. To be able to use TypeScript effectively, you should know at least how to typecast each variable as well as create things such as interfaces. If TypeScript interests you as a technology, I highly recommend the handbook at http://www.typescriptlang.org/Handbook.

Behavior-driven development with Karma and Mocha Chai

All of these will take you pretty far towards having more confidence in your visualizations, but another step you can take to be even more of a rock star is adding automated testing to your projects.

There are many reasons to write automated tests. If you have a product that needs to render charts reliably and the chart rendering is merely a part of a much larger application, you will likely want to use automated testing to ensure that changes to the application don't break your charts. Likewise, if you've created an open source project that receives a lot of pull requests from various people who use your library, you might want tests to ensure that none of this outside code causes regressive bugs. Beyond this, automated tests are great if you want to be able to show your editor proof that your chart is working and is accurate, or if you merely want to gain more confidence in your data visualization work.

There are fundamentally two ways by which you can approach testing—you can build your project and add testing after the fact, possibly needing to refactor parts so that it's more easily testable; or you can write your tests at the very beginning of your project, before any code, and *then* build your project, ensuring that it passes the tests you've created at each step of the way.

The latter approach is called **test-driven development** (**TDD**), and it should be seen as the hallmark of having reached some degree of skill with JavaScript. An extension of it is called **behavior-driven development** (**BDD**), which tends to be more focused on user interface interactions. BDD tests tend to be less brittle, as they focus more on how a feature is functioning than how it works.

I personally find the syntax of BDD-style testing frameworks much easier to read and write, which is what I'll use here, via the Mocha Chai library.

There are many types of automated tests you can do, but we're going to mainly focus on unit and functional testing.

Unit testing is when you test each of your project's functions in an isolated fashion, which requires you to write your code in such a way that side effects are minimized. If you remember from Chapter 3, *Making Data Useful*, one aim of functional programming is to not have your functions produce side effects, and unit tests are a way of both ensuring and verifying that this is in fact the case. TDD focuses on writing unit tests for each part of your application as part of the initial development process—you're simultaneously quality-assuring your code as you write it. One upshot of this is you can be less reliant on ad hoc testing methods; that is, instead of switching between the web browser and your code editor every time you hit *save* to see whether something worked or not, you can switch to your command line's test runner, which will explicitly tell you whether that thing worked or not. This can often be much faster, which helps offset the time spent on writing the tests before anything else.

Functional testing, on the other hand, is more comparable to looking at how the application behaves in a consumer context. Imagine you buy a new phone. The phone has had each of its components tested at a very high level at the factory, and that whole process is opaque to you; you assume that it passed all tests because it made it out of the factory. However, you still test it in your own ways to make sure you like it: how does it feel in your hand? Is it light and flimsy-feeling or do the materials used make it feel like a premium product? Is the touch interface responsive? How do the buttons feel? Is the screen bright enough? How long does the battery last?

BDD is more geared towards testing the latter, and it's particularly helpful with data visualizations. In some ways, D3 does some of the unit testing work for you. Because your project is using a release of D3 that passes all of its tests (that is, "the phone has left the factory"), you don't need to worry so much about D3 doing anything wrong. Rather, your bigger concerns are preventing silent errors due to dynamic typing (that is, concatenating the numbers `"1"` and `"1"` as strings and getting `"11"` instead of adding them together to get `"2"`) and ensuring that user manipulation of data doesn't cause errors. This is where BDD comes in.

Setting up your project with Mocha and Karma

We're going to use two things to write our unit tests: Karma and Mocha. Karma is the process that makes all our tests run inside of a browser, and Mocha is the test framework we're going to use. Chai is an add-on to Mocha that allows us to write BDD-style test assertions.

Install all of them in your project first, along with a few other goodies:

```
$ npm install mocha chai karma karma-webpack karma-chrome-launcher karma-nyan-reporter karma-sourcemap-loader karma-mocha--save-dev
```

Tip

Why do we keep using `--save-dev` instead of just `--save` in this chapter? We use `--save-dev` because these are developer tools that only run in the developer environment. They shouldn't be part of the distributed code.

We need to create a config file for Karma. You can either run through the wizard by running this:

```
$ ./node_modules/karma/bin/karma init
```

And then spend the next four hours reading documentation about how to set up Karma, or you can just grab the prebuilt `karma.conf.js` file in the `chapter8` of the book repo (which has lots of comments explaining what I've done). Alternatively, you can use `generator-strong-d3` to scaffold things out really easily—generally, saving time by using known good build environments is a sound strategy. Configuring build tools is a massive time sink that can usually be avoided by using good boilerplate code.

Let's get a feel of BDD-style development with Mocha and Chai by updating our TypeScript class from earlier with a few new behaviors.

Testing behaviors first – BDD with Mocha

We really aren't going to write a full test suite because we have very little space left in the book and writing about automated testing to any significant degree could fill way more pages than we really have left together. But what we'll do is update our `TypeScriptChart` class to sort the bar chart either by population or alphabetically. However, we will do so in a BDD fashion.

To start, create a new folder called `test/` and create a file called `chapter8.spec.js`. The convention is to name your test files the same as the file it's testing, but with `.spec` before the file extension. Before we write any code, it's often helpful to write out our goals in plain English:

- The chart should sort data alphabetically ascending by region name
- The chart should sort data alphabetically descending by region name
- The chart should sort data ascending by population
- The chart should sort data descending by population

As you'll see, our assertions in Mocha will resemble these statements quite closely. Open up `chapter8.spec.js` and add the following code:

```
import {TypeScriptChart} from '../src/chapter8.ts';
let data = require('../src/data/chapter1.json');
let chart = new TypeScriptChart(data);
describe('ordering a TypeScriptChart', () => {
  describe('alphabetically', () => {
    it('should sort data ascending', () => {});
    it('should sort data descending', () => {});
  });

  describe('by population', () => {
    it('should sort data ascending', () => {});
    it('should sort data descending', () => {});
  });
});
```

First, we import our chart library and our data and set up our parent `describe` block. The `describe` block is used to organize your tests. Having one parent `describe` block per `.spec` file that then contains more logical groupings is a good convention to follow. We then scaffold our assertions using `it`. Similar to `describe`, `it` groups a number of tests together (the assertions that we'll soon write in each `it` statement's callback function). Think of each `it` statement as a test and the assertions contained therein as the parts of the test needed for it to verify what the test says.

In our tests here, we're using the actual data that the chart will display. Because these are behavior tests, we're just testing to ensure that the behavior that we expect actually occurs. Often, instead of loading in a dataset, you'll create a mock data sample, which you'll use to test each function or method. Using mock data is often much faster and helps ensure that the tests are functionally correct; in other words, we ensure that they're not passing merely because of an edge-case bug resulting from, for instance, a dirty dataset. Which testing method you use is entirely up to you and how detailed you

want to be with your testing.

Before we add assertions, we need to make our code do the behavior we're testing. Update your test to resemble this:

```
describe('ordering a TypeScriptChart', () => {
  describe('alphabetically', () => {
    it('should sort data ascending', () => {
      chart.order('alphabetical', 'asc');
    });
    it('should sort data descending', () => {
      chart.order('alphabetical', 'desc');
    });
  });

  describe(' by population', () => {
    it('should sort data ascending', () => {
      chart.order('population', 'asc');
    });
    it('should sort data descending', () => {
      chart.order('population', 'desc');
    });
  });
});
```

As you can see, our chart now seems to have an `order` method, which takes two arguments: ordering `type` and `direction`. Our humble `TypeScriptChart` class doesn't have one of these yet, but we'll get to that shortly. First, we need to write our assertions. Update the first nested `describe` block to resemble the following:

```
  describe('alphabetically', () => {
    it('should sort data ascending', () => {
      chart.order('alphabetical', 'asc');
      chart.x.domain().should.have.length(8);
      chart.x.domain()[0].should.equal('burundi');
      chart.x.domain()[7].should.equal('yemen');
    });
    it('should sort data descending', () => {
      chart.order('alphabetical', 'desc');
      chart.x.domain().should.have.length(8);
      chart.x.domain()[0].should.equal('yemen');
      chart.x.domain()[7].should.equal('burundi');
    });
  });
```

We look at the `x` scale's domain to see whether the sort has worked or not because that's what we use to update things such as the *x* axis. From querying our dataset, we know that the first item alphabetically is Burundi and the last is Yemen. In each test, we first verify that the domain has the right number of items, and then verify that the first item is what it should be and the last is what it should be. If we wanted, we could have checked the entire array, but that would be more work and make the test slightly more brittle. Let's do the second block now:

```
  describe('by population', () => {
    it('should sort data ascending', () => {
      chart.order('population', 'asc');
      chart.x.domain().should.have.length(8);
      chart.x.domain()[0].should.equal('liberia');
      chart.x.domain()[7].should.equal('syria');
    });
    it('should sort data descending', () => {
      chart.order('population', 'desc');
      chart.x.domain().should.have.length(8);
      chart.x.domain()[0].should.equal('syria');
      chart.x.domain()[7].should.equal('liberia');
    });
  });
```

Note

It suffices to say that these really aren't the best tests. Because you're not using mock data, it's not entirely obvious where strings such as `syria` and `liberia` are coming from. Our dataset is small enough that we can get away with it, though.

From the preceding lines, you can get a taste for what Mocha Chai BDD syntax is like. There are also a few other types of syntax you can use, depending on your preference and testing style, but I find the Chai BDD style easiest to read. As well, I use the "Should" variant; the other variant, "Expect", would look like this:

```
expect(chart.x.domain()).to.have.length(8);
```

Cool! Now that we've written our assertions, it's time to write some *actual* code! Open up `chapter8.ts` and add the following function to your `TypeScriptChart` class:

```
  order(type, direction) {
    this.data = this.data.sort((a, b) => {
      switch(type) {
        case 'population':
          return direction.indexOf('asc') ?
            b.population - a.population : a.population - b.population;
        case 'alphabetical':
          return direction.indexOf('asc') ?
            (a.name < b.name ? 1 : a.name > b.name ? - 1 : 0) :
            (a.name > b.name ? 1 : a.name < b.name ? - 1 : 0);
      }
    });
  this.redraw();
}
```

We still need to write the `redraw` function to update all the scales; put this below the preceding code:

```
  redraw() {
    this.bars.data(this.data)
        .enter()
        .append('rect')
```

```
      .attr('class', 'bar')
      .attr('x', (d) => this.x(d.name))
      .attr('width', this.x.rangeBand())
      .attr('y', () => this.y(this.margin.bottom))
      .attr('height', 0);

  this.bars.transition()
    .delay((d, i) => i*200)
    .duration(800)
    .attr('y', (d) => this.y(d.population))
    .attr('height', (d) => this.height - this.y(d.population));

  this.x.domain(this.data.map((d) => d.name ));
  let xAxis = d3.svg.axis().scale(this.x).orient('bottom');
  this.chart.select('.x.axis').call(xAxis);
}
```

We are finally done! Jump back to your terminal and run Karma:

```
$ ./node_modules/karma/bin/karma start
```

Look at that! 4/4 passed! Nyan Cat loves us!

```
01 03 2016 00:28:22.958:WARN [karma]: No captured browser, open http://localhost:9876/
01 03 2016 00:28:22.969:INFO [karma]: Karma v0.13.21 server started at http://localhost:9876/
01 03 2016 00:28:22.975:INFO [launcher]: Starting browser Chrome
01 03 2016 00:28:24.188:INFO [Chrome 48.0.2564 (Mac OS X 10.11.3)]: Connected on socket /#N17wa4xXuBXISqFOAAAA with id 87436581

 4   _,------,
 4   _|   /\_/\
 0   ~|_( ^ .^)
 0    ""  ""

     4 total   4 passed   0 failed   0 skipped
```

It doesn't always look like that though. If we had run Karma before we wrote our `redraw` function, the output would have looked like this:

```
4   _,------,
0   _|  /\_/\
4   ~|_( x .x)
0    ""  ""

Failed Tests:
ordering a TypeScriptChart
  alphabetically
    should sort data ascending
      Chrome 48.0.2564 (Mac OS X 10.11.3)
        1) TypeError: this.redraw is not a function
          at TypeScriptChart.order (/Users/aendrew/Sites/learning-d3/test/chapter8.spec.js:152:15 <- webpack:///src/chapter8.ts:62:0)
          at Context.<anonymous> (/Users/aendrew/Sites/learning-d3/test/chapter8.spec.js:58:14 <- webpack:///test/chapter8.spec.js:9:12)

    should sort data descending
      Chrome 48.0.2564 (Mac OS X 10.11.3)
        2) TypeError: this.redraw is not a function
          at TypeScriptChart.order (/Users/aendrew/Sites/learning-d3/test/chapter8.spec.js:152:15 <- webpack:///src/chapter8.ts:62:0)
          at Context.<anonymous> (/Users/aendrew/Sites/learning-d3/test/chapter8.spec.js:64:14 <- webpack:///test/chapter8.spec.js:15:12)

  by population
    should sort data ascending
      Chrome 48.0.2564 (Mac OS X 10.11.3)
        3) TypeError: this.redraw is not a function
          at TypeScriptChart.order (/Users/aendrew/Sites/learning-d3/test/chapter8.spec.js:152:15 <- webpack:///src/chapter8.ts:62:0)
          at Context.<anonymous> (/Users/aendrew/Sites/learning-d3/test/chapter8.spec.js:73:14 <- webpack:///test/chapter8.spec.js:24:12)

    should sort data descending
      Chrome 48.0.2564 (Mac OS X 10.11.3)
        4) TypeError: this.redraw is not a function
          at TypeScriptChart.order (/Users/aendrew/Sites/learning-d3/test/chapter8.spec.js:152:15 <- webpack:///src/chapter8.ts:62:0)
          at Context.<anonymous> (/Users/aendrew/Sites/learning-d3/test/chapter8.spec.js:79:14 <- webpack:///test/chapter8.spec.js:30:12)


  4 total   0 passed   4 failed   0 skipped
```

Yikes, Nyan Cat isn't looking so good now.

In some ways, the latter is far more useful. It tells us where our tests are failing, which we can use to quickly debug problems with our code. This is the whole point of tests—to be able to pinpoint exactly where something's broken when something goes wrong during development.

Once again, there's absolutely no way I can go into a topic this big in the space of one chapter, but hopefully, you get a sense of how you can start building testing into your projects. If this is something that interests you, I highly recommend doing quite a bit more reading about the philosophy behind automated testing and BDD. Although writing tests is fairly easy, writing *good* tests takes quite a lot of skill to master.

Even though automated testing is a good tool for giving yourself confidence in your data visualizations, don't trust it absolutely. It's possible to write a whole lot of tests that don't really test anything, and things like confirmation bias have a tendency to creep into test writing. That aside, it can be a tremendously helpful tool for producing world-class code.

Tip

Next time you're feelin' like you're an awesome automated test writer, do this fun 2-minute interactive quiz from NYT:

http://www.nytimes.com/interactive/2015/07/03/upshot/a-quick-puzzle-to-test-your-problem-solving.html.

Then, if you get it wrong, write more negative test assertions.

Summary

As much as it pains me, I believe it is nearly time to bid each other adieu. I truly hope you've learned some things and enjoyed the preceding eight chapters—writing a textbook is a fine balance of getting to the quick 'n' dirty learning bits while also having some fun along the way. Regardless, if you have somehow read this thing from beginning to end, my hats off to you, as we have covered an absolutely mind-boggling array of technologies and approaches to software development.

We started off with some super basic stuff, talking about the DOM and CSS. Then we deep-dived into SVG and you learned how to build super pretty web vector graphics. After that, we started doing some really neat stuff with D3—remember that chapter on layouts when we made, like, a gazillion charts?!

We've done some cool stuff with Node.js and Canvas, we've made a boatload of cool maps, and we even put some stuff on Heroku because we're web development ninjas! Hee-yah! So much stuff!

Lastly, we talked about making truly great projects with all of this technology and being confident in our work. Whether it's looking to others for inspiration or wiring up a solid test suite to make sure that our work is accurate, data visualization is a craft. It is one craft that will only continue to grow in importance as our world becomes all the more "data rich," but one that requires a degree of precision. Don't worry if you get some things wrong, but do try your hardest to get everything right.

We are in a truly amazing era of the Web, where the restrictions that were once preventing developers from building brilliant things are slowly being eliminated, by smarter approaches, bolder decision-making, and better tooling. It is all moving *frighteningly* fast, but don't let that hold you back from trying new things and playing with code that is on the cutting edge. There is no other field in computing where you can build things that work so *universally* and so *instantaneously*. Although web development can be really difficult (hello, responsive design!), never has any one set of technologies been so utterly crucial to the way the world consumes information. Having finished this book, you now have in possession a massive war chest of tools you can try to use. Some of them (particularly the ones I sprinted through, that is, Node, Canvas, TypeScript, and Mocha) you clearly need to learn a lot more about to use effectively, but hopefully, I've given you a few things to try out in your journey to understand and make use of all this stuff. At the very least, you should have everything you need to start visualizing your world with D3 and be able to share it with the world through *the magic of the Internet*.

Finally, if you get frustrated and need either a hand or a place to express your annoyance that this *thing* you've been working on is still broken two days later, come join us on the D3 Slack channel—there's usually somebody or the other around who's been there and can help.

-Æ.

Appendix A. Bibliography

This learning path has been prepared for you to help you make compelling visualizations for your data on the Modern Web and dig into deeper concepts of Data Visualization. It comprises of the following Packt products:

- Data Visualization: a successful design process, Andy Kirk
- Social Data Visualization with HTML5 and JavaScript, Simon Timms
- Learning d3.js Data Visualization, Second Edition, Ændrew Rininsland, Swizec Teller

Index

A

- abstract visualization
  - about / Emotive and abstract
- access
  - to APIs, getting / Getting access to the APIs
- adjacency matrix / Highlighting connections with chord
- Adobe After Effects
  - URL / Other specialist tools
- Adobe Air
  - URL / HTML visualizations
- Adobe Flash
  - about / Programming environments
  - URL / Programming environments
- Adobe Illustrator
  - about / Other specialist tools
  - URL / Other specialist tools
- analytical visualization
  - about / Pragmatic and analytical
- animation
  - about / Animation
  - with transitions / Animation with transitions
- animation, with CSS transitions
  - about / Animation with CSS transitions
- animation, with transitions
  - about / Animation with transitions
  - interpolators / Interpolators
  - easing / Easing
  - timers / Timers
- annotation
  - about / Annotation
- Anscombe's quartet / Visualization as a discovery tool
- Any New York Times design
  - URL / Immerse yourself into learning about the field
- API
  - using / Using the API
- api() method / Retrieving data
- app
  - creating / Creating an app
- arc / Arc
- Arc GIS

    - URL / Tools for mapping
- area / Area
- area charts
    - about / Area chart
    - data variables / Area chart
    - visual variables / Area chart
- area size chart
    - about / Area size chart
    - data variables / Area size chart
    - visual variables / Area size chart
- arrow functions
    - URL / The obligatory bar chart example
- Atom, Sublime Text / TypeScript – the current heavyweight champion
- authenticating
    - about / Authenticating
- authentication
    - versus authorization / Authentication versus authorization
- authorization
    - versus authentication / Authentication versus authorization
- axes / Axes
- axis() function / Labels and axes
- axis function / Labels and axes

B

- .bar elements / d3.js
- background colors / To bring the data layer to the fore
- bar chart
  - about / Bar chart (or column chart)
  - data variables / Bar chart (or column chart)
  - visual variables / Bar chart (or column chart)
- barcode charts
  - about / Barcode chart
  - data variables / Barcode chart
  - visual variables / Barcode chart
- BDD
  - with Mocha / Testing behaviors first – BDD with Mocha
- behavior-driven development
  - with Karma / Behavior-driven development with Karma and Mocha Chai
  - with Mocha Chai / Behavior-driven development with Karma and Mocha Chai
- behavior-driven development (BDD), / Behavior-driven development with Karma and Mocha Chai
- behaviors
  - about / Behaviors
  - drag / Drag
  - zoom / Zoom
  - brushes / Brushes
- Big Data
  - about / There's a lot of data out there
- bodyParser / OAuth
- brushes behavior
  - about / Brushes
- brushing / Creating interactivity
- bubble chart / Determining the degree of accuracy in interpretation
- bubble hierarchy
  - about / Bubble hierarchy
  - data variables / Bubble hierarchy
  - visual variables / Bubble hierarchy
- bubble plot
  - about / Bubble plot
  - data variables / Bubble plot
  - visual variables / Bubble plot
- bubble plot maps
  - about / Bubble plot map
  - data variables / Bubble plot map
  - visual variables / Bubble plot map

- built-in array functions
  - about / Built-in array functions
- built-in layouts / Built-in layouts
- Business Intelligence (BI) / Charting and statistical analysis tools

C

- candlestick chart
  - about / Candlestick chart (or box and whiskers plot, OHLC chart)
  - data variables / Candlestick chart (or box and whiskers plot, OHLC chart)
  - visual variables / Candlestick chart (or box and whiskers plot, OHLC chart)
- canvas
  - about / Canvas
  - creating / Canvas
  - square canvas, creating / Canvas
  - polyfill / Canvas
  - Excanvas / Canvas
  - arcs / Canvas
  - stroke() function / Canvas
  - context.fillStyle / Canvas
  - colors, palette / Canvas
  - crossOrigin property / Canvas
  - crossOrigin policy / Canvas
  - onload function / Canvas
  - drawImage / Canvas
  - translation transformation, applying / Canvas
  - translate operation / Canvas
  - scale operation / Canvas
  - and SVGs, selecting between / Which one to use?
  - rendering, on server / Rendering in Canvas on the server
- CartoDB
  - URL / Tools for mapping
- cartogram
  - about / Cartogram
  - data variables / Cartogram
  - visual variables / Cartogram
- Cascading Style Sheets (CSS) / CSS
- Chandoo
  - URL / Charting and statistical analysis tools
- characteristics, data visualization
  - data and statistical accuracy / Approaching the finishing line
  - visualization accuracy / Approaching the finishing line
  - functional accuracy / Approaching the finishing line
  - visual inference / Approaching the finishing line
  - formatting accuracy / Approaching the finishing line
  - annotation accuracy / Approaching the finishing line
- charge function / Visualization
- charting tools, data visualization / Charting and statistical analysis tools

- Charts 'n Things
  - URL / Immerse yourself into learning about the field
- Chart Tools
  - URL / Charting and statistical analysis tools
- chart type
  - selecting / Choosing the appropriate chart type
- chord / Chord
  - connections, highlighting / Highlighting connections with chord
  - URL / Highlighting connections with chord
- choropleth map
  - about / Choropleth map
  - data variables / Choropleth map
  - visual variables / Choropleth map
- Chrome Developer Tools primer
  - about / A quick Chrome Developer Tools primer
- circle packing diagrams
  - about / Circle packing diagram
  - data variables / Circle packing diagram
  - visual variables / Circle packing diagram
- closr.it
  - about / Other specialist tools
- clusters
  - displaying / Showing clusters
  - pie, partitioning / Partitioning a pie
  - pack layout / Packing it in
  - treemap, used for subdividing / Subdividing with treemap
- cognitive scientist
  - about / The cognitive scientist
- collaborators, data visualization
  - initiator / The initiator
  - data scientist / The data scientist
  - journalist / The journalist
  - computer scientist / The computer scientist
  - designer / The designer
  - cognitive scientist / The cognitive scientist
  - communicator / The communicator
  - project manager / The project manager
- color
  - about / The use of color
  - used, for representing data / To represent data
  - used, for bringing data layer to fore / To bring the data layer to the fore
  - used, for conforming design requirements / To conform to design requirements
- ColorBrewer

    - URL / Using color effectively
- colors / Colors
    - selecting / Using color effectively
- communicator
    - about / The communicator
- computer scientist
    - about / The computer scientist
- construction process, data visualization / The construction process
- content
    - manipulating / Manipulating content
- context.fillStyle / Canvas
- continuous range scales
    - about / Continuous range scales
- cookieParser / OAuth
- create filter
    - about / Filters
- Cross-Origin Resource Sharing (CORS) / Loading data
- crossOrigin policy / Canvas
- crossOrigin property / Canvas

D

- d3 / Visualization
- D3
  - URL / What is D3.js?
  - manipulating with / Manipulating the DOM with D3
  - data functions / Data functions of D3
- D3 (Data-Driven Documents)
  - about / What is D3.js?
- d3-plugins repository
  - URL / Summary
- D3.js
  - URL / Programming environments, What is D3.js?
  - about / Programming environments, What is D3.js?
- d3.js
  - about / d3.js
  - data() / d3.js
  - of graph / d3.js
  - custom color scales / Custom color scales
  - labels / Labels and axes
  - axis function / Labels and axes
- data
  - about / Exploiting the digital age, There's a lot of data out there
  - familiarizing / Preparing and familiarizing yourself with your data
  - preparing / Preparing and familiarizing yourself with your data
  - representing, color used / To represent data
  - beyond Excel / Data beyond Excel
  - retrieving / Retrieving data, Retrieving data
  - joining, to selections / Joining data to selections
  - loading / Loading data
  - core / The core
  - convenience functions / Convenience functions
- data familiarization, mechanics
  - acquisition / Preparing and familiarizing yourself with your data
  - examination / Preparing and familiarizing yourself with your data
- data functionally
  - about / Thinking about data functionally
- data functions
  - about / Data functions of D3
- data growth
  - about / There's a lot of data out there
- data scientist
  - about / The data scientist

- dataset
  - about / The dataset
- data visualization
  - about / Exploiting the digital age, Clarity, honesty, and sense of purpose
  - as discovery tool / Visualization as a discovery tool
  - principles / The bedrock of visualization knowledge
  - defining / Defining data visualization
  - skills, for masses / Visualization skills for the masses
  - design objectives / Visualization design objectives
  - function, establishing / Establishing intent – the visualization's function
  - intent, establishing / Establishing intent – the visualization's tone
  - potential motives / Establishing intent – the visualization's tone
  - collaborators / The "eight hats" of data visualization design
  - desgn concepts / Data visualization design is all about choices
  - tips / Some helpful tips
  - methods / Data visualization methods
  - tecnology concepts / For constructing visualizations, technology matters
  - construction process / For constructing visualizations, technology matters, The construction process
  - programs / Visualization software, applications, and programs
  - applications / Visualization software, applications, and programs
  - software / Visualization software, applications, and programs
  - statistical analysis tools / Charting and statistical analysis tools
  - charting tools / Charting and statistical analysis tools
  - programming environments / Programming environments
  - mapping tools / Tools for mapping
  - specialist tools / Other specialist tools
  - finishing line, approaching / Approaching the finishing line
  - post-launch evaluation / Post-launch evaluation
  - user capabilities, developing / Developing your capabilities
  - design skills, practicing / Practice, practice, practice!
  - designers' work, evaluating / Evaluating the work of others
  - output, sharing / Publishing and sharing your output
  - latest projects / Immerse yourself into learning about the field
  - trends / Immerse yourself into learning about the field
  - articles / Immerse yourself into learning about the field
  - announcements / Immerse yourself into learning about the field
  - developments / Immerse yourself into learning about the field
  - design narratives / Immerse yourself into learning about the field
  - project critique / Immerse yourself into learning about the field
  - technical tutorials / Immerse yourself into learning about the field
  - visualization communities / Immerse yourself into learning about the field
- data visualization, tips

    - sketching / Some helpful tips
    - note taking / Some helpful tips
    - time management / Some helpful tips
- DataVisualization.ch
    - URL / Immerse yourself into learning about the field
- data visualization methodology
    - about / The data visualization methodology
- DataWrapper
    - URL / Charting and statistical analysis tools
- deep queries
    - about / Filters
- desgn concepts, data visualization / Data visualization design is all about choices
- designer
    - about / The designer
- design metaphor
    - creating / Creating an appropriate design metaphor
- design objectives, data visualization
    - about / Visualization design objectives
    - function / Strive for form and function
    - form / Strive for form and function
    - selection, justifying / Justifying the selection of everything we do
    - accessibility, creating through intuitive design / Creating accessibility through intuitive design
    - never decieve receiver / Never deceive the receiver
- design requirements
    - conforming, color used / To conform to design requirements
- destructuring
    - URL / Drawing geographically
- diagonal / Diagonal
- digital age
    - exploiting / Exploiting the digital age
- discovery tool, data visualization / Visualization as a discovery tool
- discrete range scales
    - about / Discrete range scales
- diverging schemes
    - about / To represent data
- Document Object Model (DOM) / Scalable Vector Graphics
    - about / DOM
    - manipulating with / Manipulating the DOM with D3
    - selections / Selections
    - table!, creating / Let's make a table!
- domain-specific language (DSL)
    - about / What is D3.js?

- Dorling cartogram
  - about / Dorling cartogram
  - data variables / Dorling cartogram
  - visual variables / Dorling cartogram
- dot plot
  - about / Dot plot
  - data variables / Dot plot
  - visual variables / Dot plot
- dot plot maps
  - about / Dot plot map
  - data variables / Dot plot map
  - visual variables / Dot plot map
- drag behavior
  - about / Drag
- drawGraphColumn() function / Raphaël
- drawImage / Canvas

E

- <elipse> tag / Scalable Vector Graphics
- each operator / Raphaël
- Eager Eyes
  - URL / Immerse yourself into learning about the field
- easing function
  - about / Easing
- ECMAScript 2016 (ES2016)
  - about / What's ES2016?
  - URL / What's ES2016?
  - Git, on command line / Getting started with Node and Git on the command line
  - Node, on command line / Getting started with Node and Git on the command line
  - Chrome Developer Tools primer / A quick Chrome Developer Tools primer
  - obligatory bar chart example / The obligatory bar chart example
- editorial focus
  - significance / The importance of editorial focus
- EMC
  - URL / There's a lot of data out there
- emotive visualization
  - about / Emotive and abstract
- environment
  - preparing / Readying the environment
- environment variables
  - about / All aboard the Express train to Server Town!
- Excanvas / Canvas
- Excel / Preparing and familiarizing yourself with your data
  - about / Data beyond Excel
- exhibiting data visualization
  - about / When the function is to exhibit data
  - example / When the function is to exhibit data
- explanatory data visualization
  - about / When the function is to explain
  - example / When the function is to explain
- exploratory data visualization
  - about / When the function is to explore
  - example / When the function is to explore
- express train
  - about / All aboard the Express train to Server Town!

F

- 12-factor app
  - URL / Readying the environment
- Facebook Flow
  - about / The new kid on the block – Facebook Flow
- Fell In Love With Data
  - URL / Immerse yourself into learning about the field
- filters
  - about / Scalable Vector Graphics, Filters
  - using / Scalable Vector Graphics
  - applying, to one element / Scalable Vector Graphics
  - applying, on hover / Scalable Vector Graphics
- Flash / Programming environments
- Flexbox and Frogs
  - URL / Columns are for desktops, rows are for mobile
- Flexbox Grid
  - URL / Columns are for desktops, rows are for mobile
- floating bar
  - about / Floating bar (or Gantt chart)
  - data variables / Floating bar (or Gantt chart)
  - visual variables / Floating bar (or Gantt chart)
- Flow
  - static type checking / Static type checking with TypeScript and Flow
- FlowingData
  - URL / Immerse yourself into learning about the field
- flow maps
  - about / Flow map
  - data variables / Flow map
  - visual variables / Flow map
- Font Awesome
  - URL / Avoid "mystery meat" navigation
- for-each statement / What exactly did we do here?
- Force-directed graph layout / Visualization
- force.drag action / Visualization
- force layout
  - used, for drawing / Drawing with force
- function
  - establishing, of data visualization / Establishing intent – the visualization's function
- Functional Art
  - URL / Immerse yourself into learning about the field

G

- Gapminder project
  - URL / Helping your audience understand scale
- geo-spatial data
  - mapping / Mapping geo-spatial data
- Geocommons
  - URL / Tools for mapping
- geodata
  - getting / Getting geodata
- geography
  - about / Geography
  - geodata, getting / Getting geodata
  - drawing geographically / Drawing geographically
  - using, as base / Using geography as a base
- Gephi
  - URL / Other specialist tools
- Gestalt Laws / The bedrock of visualization knowledge
- GetRetweets function / OAuth
- GitHub
  - URL / What is D3.js?, The obligatory bar chart example, Getting geodata, Using geography as a base
- glyph chart
  - about / Determining the degree of accuracy in interpretation, Glyph chart
  - data variables / Glyph chart
  - visual variables / Glyph chart
- Google+
  - app, creating / Creating an app
  - data, retrieving / Retrieving data
  - visualization / Visualization
- Google Fusion Tables
  - about / Charting and statistical analysis tools
  - URL / Charting and statistical analysis tools
- Google Refine / Preparing and familiarizing yourself with your data
- Graph API / OAuth versions
  - about / Creating an app
- Grapheur
  - URL / Charting and statistical analysis tools
- Guardian datablog
  - URL / Immerse yourself into learning about the field

H

- hankerin
  - URL / An HTML visualization example
- Haskell
  - URL / Thinking about data functionally
- heatmap
  - about / Heatmap (or matrix chart)
  - data variables / Heatmap (or matrix chart)
  - visual variables / Heatmap (or matrix chart)
- helpers.js file / Built-in layouts
- Heroku
  - about / Creating an app
  - deploying to / Deploying to Heroku
  - URL / Deploying to Heroku
- Heroku Toolbelt
  - URL / Deploying to Heroku
- hierarchical layouts
  - about / Hierarchical layouts
- histogram layout
  - using / Using the histogram layout
- histograms
  - about / Histogram
  - data variables / Histogram
  - visual variables / Histogram
- Homebrew
  - URL / Rendering in Canvas on the server
- horizon chart
  - about / Horizon chart
  - data variables / Horizon chart
  - visual variables / Horizon chart
- hover
  - filters, applying / Scalable Vector Graphics
- HTML visualization
  - example / An HTML visualization example
- HTML visualizations
  - about / HTML visualizations
- hue saturation lightness (HSL) / Colors

I

- Illustrator / Charting and statistical analysis tools
- InDesign
  - about / Other specialist tools
  - URL / Other specialist tools
- Indiegogo
  - about / There's a lot of data out there
- Indiemapper
  - URL / Tools for mapping
- Infographics
  - about / Other specialist tools
- Information is Beautiful Awards
  - URL / Immerse yourself into learning about the field
- Infosthetics
  - URL / Immerse yourself into learning about the field
- initiator
  - about / The initiator
- Inkscape
  - about / Other specialist tools
  - URL / Other specialist tools
- Instagram
  - about / There's a lot of data out there
- Instant Atlas
  - URL / Tools for mapping
- interactivity
  - creating / Creating interactivity
- interactivity, features
  - parameter manipulation / Creating interactivity
  - variable manipulation / Creating interactivity
  - view, adjusting / Creating interactivity
  - annotated details / Creating interactivity
  - animation / Creating interactivity
- interpolators
  - URL / Line
  - about / Interpolators
- Isarithmic maps
  - about / Isarithmic map (or contour map or topological map)
  - data variables / Isarithmic map (or contour map or topological map)
  - visual variables / Isarithmic map (or contour map or topological map)
- iterators and generators
  - URL / Data functions of D3

J

- journalist
  - about / The journalist
- Junk Charts
  - URL / Immerse yourself into learning about the field

K

- Karma
  - behavior-driven development (BDD) / Behavior-driven development with Karma and Mocha Chai
  - used, for settingup project / Setting up your project with Mocha and Karma
- Kartograph
  - URL / Tools for mapping
- KendoUI
  - about / Programming environments
  - URL / Programming environments
- KeyLines
  - URL / Other specialist tools
- Keynote
  - about / Other specialist tools
  - URL / Other specialist tools
- Kickstarter
  - about / There's a lot of data out there

L

- Law of Proximity / The bedrock of visualization knowledge
- Law of Similarity
  - about / The bedrock of visualization knowledge
- Laws of Perceptual Organization
  - about / The bedrock of visualization knowledge
- layouts
  - about / What are layouts and why should you care?
  - built-in layouts / Built-in layouts
  - normal / Normal layouts
  - histogram layout, using / Using the histogram layout
  - hierarchical layouts / Hierarchical layouts
- Leaflet
  - URL / Tools for mapping
- Leila Haddou
  - URL / The dataset
- let keyword
  - URL / The obligatory bar chart example
- Likert scale of disagree
  - about / Stacked bar chart (or stacked column chart)
- line / Line
- line charts
  - about / Line chart
  - data variables / Line chart
  - visual variables / Line chart
- linkDistance function / Visualization
- linter
  - about / Linting all the things
- lShowing
  - about / Showing popularity through time with stack

M

- Many Eyes / Charting and statistical analysis tools
- mapping tools, data visualization / Tools for mapping
- matrix chart / Creating an appropriate design metaphor
- Maya 3D
  - URL / Other specialist tools
- methods, data visualization
  - categories comparison / Data visualization methods
  - hierarchies, assessing / Data visualization methods, Assessing hierarchies and part-to-whole relationships
  - part-to-whole relationships / Data visualization methods, Assessing hierarchies and part-to-whole relationships
  - changes over time, displaying / Data visualization methods, Showing changes over time
  - relationships, plotting / Data visualization methods, Plotting connections and relationships
  - connections, plotting / Data visualization methods, Plotting connections and relationships
  - geo-spatial data, mapping / Data visualization methods, Mapping geo-spatial data
- Microsoft Excel
  - about / Charting and statistical analysis tools
- Mike Bostock
  - URL / Helping your audience understand scale
- Miso project
  - about / Programming environments
  - URL / Programming environments
- mobile and desktop, designing
  - about / Some principles for designing for mobile and desktop
  - columns and rows / Columns are for desktops, rows are for mobile
  - animations / Be sparing with animations on mobile
  - UI elements / Realize similar UI elements react differently between platforms
  - mystery meat navigation / Avoid "mystery meat" navigation
  - scroll / Be wary of the scroll
- Mocha
  - behavior-driven development (BDD) / Behavior-driven development with Karma and Mocha Chai
  - used, for setting up project / Setting up your project with Mocha and Karma
  - behavior-driven development (BDD) with / Testing behaviors first – BDD with Mocha
- moment.js
  - URL / Time arithmetic

N

- network connection map
  - about / Network connection map
  - data variables / Network connection map
  - visual variables / Network connection map
- network diagram
  - about / Network diagram (or force-directed/node-link network)
  - data variables / Network diagram (or force-directed/node-link network)
  - visual variables / Network diagram (or force-directed/node-link network)
- node.js
  - installing / Setting up a server
  - about / Setting up a server
- Nodebox
  - URL / Programming environments
  - about / Programming environments
- none filter
  - about / Filters
- NYT
  - URL / Testing behaviors first – BDD with Mocha

O

- OAuth
  - protocol / The OAuth protocol
  - versions / OAuth versions
  - 1.0a / OAuth versions
  - 2.0 / OAuth versions
  - Resource Owner / OAuth versions
  - protected resource / OAuth versions
  - TweetDeck / OAuth versions
  - Graph API / OAuth versions
  - visualization site, setting up / OAuth versions
  - Facebook OAuth endpoint, request sending to / OAuth versions
  - redirect_uri parameter / OAuth versions
  - third-party OAuth providers / OAuth versions
  - about / OAuth, Authenticating, Using the API
- OAuth 1.0a / OAuth versions
- OAuth 2.0 / OAuth versions
- OAuth protocol
  - versions / OAuth versions
- OECD Better Life Index
  - about / Determining the degree of accuracy in interpretation
- on() function / Creating a visualization
- onload event / Canvas
- onload function / Canvas
- OpenRefine
  - URL / Highlighting connections with chord
- OpenStreetMap
  - URL / Tools for mapping
- ordinal scales
  - about / Ordinal scales

P

- <polygon> tag / Scalable Vector Graphics
- <polyline> tag / Scalable Vector Graphics
- pack layout / Packing it in
- Panopticon
  - URL / Charting and statistical analysis tools
- Paper.js
  - URL / Programming environments
- parallel sets
  - about / Parallel sets (or parallel coordinates)
  - data variables / Parallel sets (or parallel coordinates)
  - visual variables / Parallel sets (or parallel coordinates)
- parameters, for data visualization project
  - about / Key factors surrounding a visualization project
  - aim / Key factors surrounding a visualization project
  - time pressures / Key factors surrounding a visualization project
  - costs / Key factors surrounding a visualization project
  - client pressures / Key factors surrounding a visualization project
  - format / Key factors surrounding a visualization project
  - technical capabilities / Key factors surrounding a visualization project
- parametric equations
  - URL / Timers
- particle flow map
  - about / Particle flow map
  - data variables / Particle flow map
  - visual variables / Particle flow map
- paths, Scalable Vector Graphics (SVG)
  - using / Using paths
  - line / Line
  - area / Area
  - arc / Arc
  - symbol / Symbol
  - chord / Chord
- Perceptual Edge
  - URL, for blog / Immerse yourself into learning about the field
- Periscopic
  - URL / Immerse yourself into learning about the field
- PhoneGap
  - about / HTML visualizations
  - URL / HTML visualizations
- pie
  - partitioning / Partitioning a pie

- pie chart
  - about / Baking a fresh 'n' delicious pie chart
  - labeling / Labeling your pie chart
- pie charts
  - about / Pie chart
  - data variables / Pie chart
  - visual variables / Pie chart
- Pitch Interactive
  - URL / Immerse yourself into learning about the field
- pixelated bar chart
  - about / Pixelated bar chart
  - data variables / Pixelated bar chart
  - visual variables / Pixelated bar chart
- Platform as a Service (PaaS) / Readying the environment
- PolitiFact and Vox
  - URL / Clarity, honesty, and sense of purpose
- Polychart
  - URL / Charting and statistical analysis tools
- polyfill / Canvas
- polyfill
  - URL / Animation with CSS transitions
- Polymaps
  - URL / Tools for mapping
- pragmatic visualization
  - about / Pragmatic and analytical
- principles, data visualization / The bedrock of visualization knowledge
- Processing
  - URL / Programming environments
  - about / Programming environments
- programming environments, data visualization / Programming environments
- project, data visualization
  - purpose / The reason for existing
  - intended effect / The intended effect
  - parameters / Key factors surrounding a visualization project
- projection plugin
  - URL / Drawing geographically
- project manager
  - about / The project manager
- protected resource / OAuth versions
- prototype / Built-in array functions
- Protovis
  - about / Programming environments
- proximity

- detecting / Proximity detection and the Voronoi geom

Q

- QlikView
  - URL / Charting and statistical analysis tools
- Quadrigram
  - URL / Programming environments
  - about / Programming environments
- quantitative scales
  - about / Quantitative scales

R

- R
  - about / Programming environments
  - URL / Programming environments
- radial chart
  - about / Radial chart
  - data variables / Radial chart
  - visual variables / Radial chart
- radial network
  - about / Radial network (or chord diagram)
  - data variables / Radial network (or chord diagram)
  - visual variables / Radial network (or chord diagram)
- rangeBands / d3.js
- Raphaël
  - URL / Programming environments
  - about / Raphaël
  - drawGraphColumn() function / Raphaël
- rate limits
  - about / Authenticating
- red green blue (RGB) / Colors
- redirect_uri parameter / OAuth versions
- requestOAuth page / Server side
- Resource Owner / OAuth versions
- responsive design / Understanding your audience (or "trying not to forget about mobile")
- rest parameter / Built-in layouts
- retreiveFriends / Retrieving data
- rgba function / Canvas

S

- Sankey diagram / Creating interactivity
  - about / When the function is to explain, Sankey diagram
  - data variables / Sankey diagram
  - visual variables / Sankey diagram
- Scalable Vector Graphics (SVG)
  - about / Scalable Vector Graphics
  - drawing with / Drawing with SVG
  - elements, adding manually / Manually adding elements and shapes
  - shapes, adding manually / Manually adding elements and shapes
  - text / Text
  - shapes / Shapes
  - transformations / Transformations
  - paths, using / Using paths
  - diagonal / Diagonal
  - axes / Axes
  - Cascading Style Sheets (CSS) / CSS
  - colors / Colors
- scale operation / Canvas
- scales
  - about / Scales, Helping your audience understand scale
  - ordinal scales / Ordinal scales, Quantitative scales
  - quantitative scales / Quantitative scales
  - continuous range scales / Continuous range scales
  - discrete range scales / Discrete range scales
- scatter plot
  - about / Scatter plot
  - data variables / Scatter plot
  - visual variables / Scatter plot
- scatter plot matrix
  - about / Scatter plot matrix
  - data variables / Scatter plot matrix
  - visual variables / Scatter plot matrix
- scatterplot matrix visualization
  - about / When the function is to explore
- script tag
  - about / Using the API
- selections
  - about / Selections
  - example / Selections example
  - data, joining to / Joining data to selections
- select method / d3.js

- server
  - setting up / Setting up a server
  - canvas, rendering / Rendering in Canvas on the server
- showTip() function / Creating a visualization
- sketching / Some helpful tips
- skills, data visualization / Visualization skills for the masses
- slopegraph
  - about / Slopegraph (or bumps chart or table chart)
  - data variables / Slopegraph (or bumps chart or table chart)
  - visual variables / Slopegraph (or bumps chart or table chart)
- social media data
  - about / Social media data
- sparklines
  - about / Sparklines
  - data variables / Sparklines
  - visual variables / Sparklines
- specialist tools, data visualization / Other specialist tools
- spread operator / Built-in layouts
- square pie
  - about / Square pie (or unit chart or waffle chart)
  - data variables / Square pie (or unit chart or waffle chart)
  - visual variables / Square pie (or unit chart or waffle chart)
- stacked area chart
  - about / Stacked area chart
  - data variables / Stacked area chart
  - visual variables / Stacked area chart
- stacked bar chart
  - about / Stacked bar chart (or stacked column chart)
  - data variables / Stacked bar chart (or stacked column chart)
  - visual variables / Stacked bar chart (or stacked column chart)
- static type checking
  - with TypeScript and Flow / Static type checking with TypeScript and Flow
- static typing
  - about / Static type checking with TypeScript and Flow
- statistical analysis tools, data visualization / Charting and statistical analysis tools
- steam graph
  - about / Stream graph
  - visual variables / Stream graph
- stories
  - finding, visual analysis used / Using visual analysis to find stories
  - telling, example / An example of finding and telling stories
  - finding, example / An example of finding and telling stories
- stream graph

    - data variables / Stream graph
- streamgraph / Showing popularity through time with stack
    - tooltips, adding / Adding tooltips to our streamgraph
- stroke() function / Canvas
- stroke-dasharray
    - URL / CSS
- strokeRect() function / Canvas
- SVGs
    - about / Scalable Vector Graphics
    - Document Object Model (DOM) / Scalable Vector Graphics
    - multiple elements, building / Scalable Vector Graphics
    - CSS, using / Scalable Vector Graphics
    - and canvas, selecting between / Which one to use?
- symbol / Symbol
    - URL / Symbol
- Synchronized Multimedia Integration Language (SMIL)
    - URL / Animation with CSS transitions

T

- Tableau / Preparing and familiarizing yourself with your data
  - URL / Charting and statistical analysis tools
  - about / Charting and statistical analysis tools
  - practitioners / Charting and statistical analysis tools
- TED talk
  - URL / Helping your audience understand scale
- test-driven development (TDD) / Behavior-driven development with Karma and Mocha Chai
- text-anchor property / Labels and axes
- The Why Axis
  - URL / Immerse yourself into learning about the field
- ThisFish
  - about / There's a lot of data out there
  - URL / There's a lot of data out there
- TIBCO Spotfire
  - URL / Charting and statistical analysis tools
- TileMill
  - URL / Tools for mapping
- time
  - about / Time
  - formatting / Formatting
  - arithmetic / Time arithmetic
- timers
  - about / Timers
- tooltips
  - adding, to streamgraph / Adding tooltips to our streamgraph
- translate operation / Canvas
- tree
  - drawing / Drawing a tree
- tree hierarchy
  - about / Tree hierarchy
  - data variables / Tree hierarchy
  - visual variables / Tree hierarchy
- treemap layout / Subdividing with treemap
- tree maps
  - about / Tree map
  - data variables / Tree map
  - visual variables / Tree map
- trellis chart
  - about / Small multiples (or trellis chart)
  - data variables / Small multiples (or trellis chart)
  - visual variables / Small multiples (or trellis chart)

- Tulp Interactive
  - URL / Immerse yourself into learning about the field
- TweetDeck / OAuth versions
- Twitter
  - access to APIs, getting / Getting access to the APIs
  - server, setting up / Setting up a server
  - OAuth / OAuth
  - visualization / Visualization
  - about / Creating a visualization
- TypeScript
  - static type checking / Static type checking with TypeScript and Flow
  - about / TypeScript – the current heavyweight champion

U

- Underscore.js / Server side
- US Census Bureau
  - URL / Getting geodata
- user interaction
  - about / Interacting with the user
  - basic interaction / Basic interaction

V

- Vine
  - about / There's a lot of data out there
- Vis Check
  - URL / To represent data
- Visual.ly
  - URL / Immerse yourself into learning about the field
- visual analysis
  - used, for finding stories / Using visual analysis to find stories
- visual annotation / Annotation
- Visualising Data
  - URL / Immerse yourself into learning about the field
- visualization
  - about / Pragmatic and analytical
  - Twitter / Visualization
  - creating / Creating a visualization
  - Google+ / Visualization
- visualization, Twitter
  - server side / Server side
  - client side / Client side
- visualization anatomy, data presentation
  - about / The visualization anatomy – data presentation
  - color / The use of color
  - interactivity, creating / Creating interactivity
  - annotation / Annotation
  - arrangement / Arrangement
- visualization anatomy, data representation
  - about / The visualization anatomy – data representation
  - correct visualization method, selecting / Choosing the correct visualization method
  - physical properties, considering for data / Considering the physical properties of our data
  - degree of accuracy, determining in interpretation / Determining the degree of accuracy in interpretation
  - design metaphor, creating / Creating an appropriate design metaphor
  - final solution, selecting / Choosing the final solution
- visualization method
  - selecting / Choosing the correct visualization method
- visualization techniques / Visualizing
- Visualizing.org
  - URL / Immerse yourself into learning about the field
- visual variable / Determining the degree of accuracy in interpretation
- Voronoi geom
  - about / Proximity detection and the Voronoi geom

W

- W3C
  - URL / Transformations
- WebGL
  - about / Programming environments
  - URL / Programming environments
- wind map / Determining the degree of accuracy in interpretation
- Word clouds
  - about / Word cloud
  - data variables / Word cloud
  - visual variables / Word cloud
- word clouds / Charting and statistical analysis tools
- Wordle
  - URL / Charting and statistical analysis tools

Z

- zoom.it
  - about / Other specialist tools
- zoom behavior
  - about / Zoom