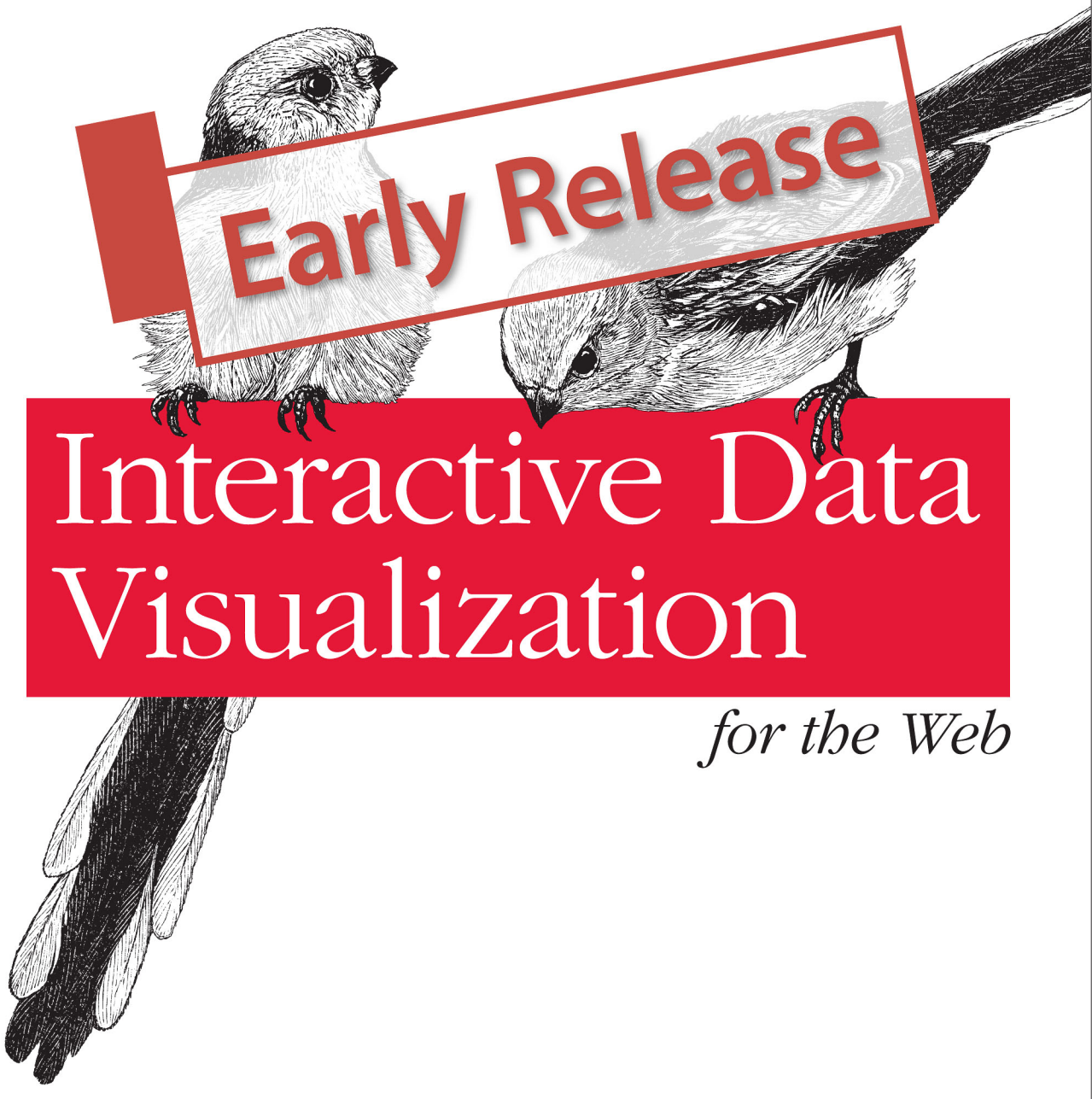
An Introduction to Designing With D3



Interactive Data Visualization

for the Web

O'REILLY®

Scott Murray

Interactive Data Visualization for the Web

Scott Murray

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Interactive Data Visualization for the Web

by Scott Murray

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meghan Blanchette
Production Editor:
Copyeditor:

Proofreader: FIX ME!
Indexer:
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

March 2013: First Edition

Revision History for the First Edition:

2012-11-20 Early Release Version 1

See *http://oreilly.com/catalog/errata.csp?isbn=9781449339739* for release details.

ISBN: 978-1-449-33973-9

[?]

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

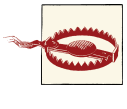
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

CHAPTER 1

Introduction

Why data visualization?

Our information age more often feels like an era of information overload. Excess amounts of information are overwhelming; raw data becomes useful only when we apply methods of deriving insight from it.

Fortunately, we humans are intensely visual creatures. Few of us can detect patterns among rows of numbers, but even young children can interpret bar charts, extracting meaning from those numbers' visual representations. For that reason, data visualization is a powerful exercise. Visualizing data is the fastest way to communicate it to others.

Of course, visualizations, like words, can be used to lie, mislead, or distort the truth. But when practiced honestly and with care, the process of visualization can help us see the world in a new way, revealing unexpected patterns and trends in the otherwise-hidden information around us. At its best, data visualization is expert storytelling.

More literally, visualization is a process of *mapping* information to visuals. We craft rules that interpret data and express its values as visual properties. For example, this humble bar chart is generated from a very simple rule: Larger values are *mapped* as taller bars.

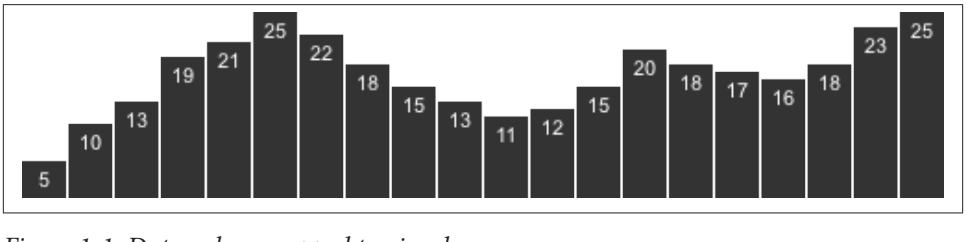


Figure 1-1. Data values mapped to visuals

More complex visualizations are generated from data sets more complex than the sequence of numbers above and more complex sets of mapping rules.

Why write code?

Mapping data by hand can be satisfying, yet is slow and tedious. So we usually employ the power of computation to speed things up. The increased speed enables us to work with much larger data sets of thousands or millions of values; what would have taken years of effort by hand can be mapped in a moment. Just as important, we can rapidly experiment with *alternate mappings*, tweaking our rules and seeing their output re-rendered immediately. This loop of write/render/evaluate is critical to the iterative process of refining a design.

Sets of mapping rules function as *design systems*. The human hand no longer executes the visual output; the computer does. Our human role is to conceptualize, craft, and write out the rules of the system, which is then finally executed by software.

Unfortunately, software (and computation generally) is extremely bad at understanding what, exactly, people want. (To be fair, many humans are also not good at this challenging task.) Because computers are binary systems, everything is either on or off, yes or no, this or that, there or not there. Humans are mushier, softer creatures, and the computers are not willing to meet us halfway — we must go to them. Hence the inevitable struggle of learning to write software, in which we train ourselves to communicate in the very limited and precise syntax that the computer can understand.

Yet we continue to write code because seeing our visual creations come to life is so rewarding. We practice data visualization because it is exciting to see what has never before been seen. It is like summoning a magical, visual genie out of an inscrutable data bottle.

Why interactive?

Static visualizations can offer only pre-composed "views" of data, so multiple static views are often needed to present a variety of perspectives on the same information. The number of dimensions of data are limited, too, when all visual elements must be present on the same surface at the same time. Representing multidimensional data sets fairly in static images is notoriously difficult. A fixed image is ideal when alternate views are neither needed nor desired, and required when publishing to a static medium, such as print.

Dynamic, interactive visualizations can empower people to explore the data for themselves. The basic functions of most interactive visualization tools have changed little since 1996, when Ben Shneiderman of the University of Maryland first proposed a "Visual Information-Seeking Mantra": *Overview first, zoom and filter, then details-on-demand.*

> Shneiderman, Ben. *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*, July 1996, Department of Computer Science Human-Computer Interaction Laboratory, and Institute for Systems Research University of Maryland, College Park, Maryland

This design pattern is found in most interactive visualizations today. The combination of functions is successful because it makes the data accessible to different audiences, from those who are merely browsing or exploring the data set, to those who approach the visualization with a specific question in search of an answer. An interactive visualization that offers an overview of the data alongside tools for "drilling down" into the details, may successfully fulfill many roles at once, addressing the different concerns of different audiences, from those new to the subject matter to those already deeply familiar with the data.

Of course, interactivity can also encourage engagement with the data in ways that static images cannot. With animated transitions and well-crafted interfaces, some visualizations can make exploring data feel more like playing a game. Interactive visualization can be a great medium for reaching those who don't consider learning to be a *fun* task.

Why on the web?

Visualizations aren't truly visual unless they are *seen*. Getting your work out there for others to see is critical, and publishing on the web is the quickest way to reach a global audience.

Best of all, everything covered in this book can be done with freely accessible tools, so the only investment required is your time. And everything we'll talk about uses open-source, web-standard technologies. By avoiding proprietary software and plug-ins, you can ensure that your projects are accessible on the widest possible range of devices, from typical desktop computers to tablets and even phones. The more accessible your visualization, the greater your audience and your impact.

What This Book Is

This book is a practical introduction to merging three practices — data visualization, interactive design, and web development — using D3, a powerful tool for custom, web-based visualization.

These chapters grew out of my own process of learning how to use D3. Many people, including myself, come to D3 with backgrounds in design, mapping, and data visualization, but not programming and computer science.

D3 has a bit of an unfair reputation for being hard to learn. D3 itself is not so complicated, but it operates in the domain of the web, and the web *is* complicated. Using D3 comfortably requires some prior knowledge of the web technologies with which it interacts, such as HTML, CSS, JavaScript, and SVG. Many people (myself included) are self-taught when it comes to web skills. This is great, because the barrier to entry is so low, but problematic because it means we probably didn't learn each of these technologies from the ground up — more often, we just hack something together until it seems to work, and call it a day. Yet successful use of D3 requires understanding some of these technologies in a fundamental way.

Since D3 is written in JavaScript, learning to use D3 often means learning a lot about JavaScript. For many datavis folks, D3 *is* their introduction to JavaScript (or even web development generally). It's hard enough to learn a new programming language, let alone a new tool built on that language. D3 will enable you to do great things with JavaScript that you never would have even attempted. The time you spend learning both the language and the tool will provide an incredible payoff.

My goal is to reduce that learning time, so you can start creating awesome stuff sooner. We'll take a ground-up approach, starting with the fundamental concepts and gradually adding complexity. I don't intend to show you how to make specific kinds of visualizations so much as to help you understand the workings of D3 well enough to take those building blocks and generate designs of your own creation.

Who You Are

You may be an absolute beginner, someone new to data vis, web development, or both. (Welcome!) Perhaps you are a journalist interested in new ways to communicate the data you collect during reporting. Or maybe you're a designer, comfortable drawing static infographics but ready to make the leap to interactive projects on the web. You could be an artist, interested in generative, data-based art. Or a programmer, already familiar with JavaScript and the web, but excited to learn a new tool and pick up some visual design experience along the way.

Whoever you are, I hope that you

- have heard of this new thing called the "World Wide Web"
- are a bit familiar with HTML, the DOM, and CSS
- may even have a little programming experience already
- have heard of jQuery or written some JavaScript before

- aren't scared by unknown initialisms like CSV, SVG, or JSON
- want to make useful, interactive visualizations

If anything above is unknown or unclear, don't fear. You may just want to spend more time with the **Technology Fundamentals** chapter, which covers what you really need to know before diving into D3.

What This Book Is Not

That said, this is definitely not a computer science textbook, and it is not intended to teach the intricacies of any one web technology (HTML, CSS, JavaScript, SVG) in depth.

In that spirit, I may gloss over some technical points, grossly oversimplifying important concepts fundamental to computer science in ways that will make true software engineers recoil. That's fine, because I'm writing for artists and designers here, not engineers. We'll cover the basics, and then you can dive into the more complex pieces once you're comfortable.

I will deliberately *not* address every possible approach to a given problem, but will typically present what I feel is the simplest solution, or, if not the simplest, then the most understandable.

My goal is to teach you the fundamental concepts and methods of D3. As such, this book is decidedly *not* organized around specific example projects. Everyone's data and design needs will be different. It's up to you to integrate these concepts in the way best suited to your particular project.

Using Sample Code

If you are a mad genius, then you can probably learn to use D3 without ever looking at any sample code files, in which case you can skip the rest of this section.

Since you're still with me, you are probably still very bright but not mad, in which case you should undertake this book with the full set of accompanying code samples in hand. Before you go any further, please download the sample files from here:

https://github.com/alignedleft/d3-book

Normal people will want to click the "Downloads" link, then click again to download the compressed `.zip` archive with all the files. Hardcore geeksters will want to clone the repository using git. If that last sentence sounds like total gibberish, please use the first option.

Within the download, you'll notice there is a folder for each chapter that has code to go with it:

```
chapter_05
chapter_06
chapter_07
chapter_08
…
```

Files are organized by chapter, so in Chapter 9 when I reference `01_bar_chart.html`, know that you can find that file in the corresponding location:

```
d3-book/chapter_9/01_bar_chart.html
```

You are welcome to copy, adapt, modify and reuse the example code in these tutorials for any non-commercial purpose.

Thank You

Finally, this book has been hand-crafted, carefully written, and pedagogically fine-tuned for maximum effect. Thank you for reading it. I hope you learn a great deal, and even have some fun along the way.

CHAPTER 2

Introducing D3

D3 — also referred to as D^3 or d3.js — is a JavaScript library for creating data visualizations. But that kind of undersells it.

The abbreviation D3 references the tool's full name, *Data-Driven Documents*. The *data* is provided by you, and the *documents* are web-based documents, meaning anything that can be rendered by a web browser, like HTML and SVG. D3 does the *driving*, in the sense that it connects the data to the documents.

Of course, the name also functions as a clever allusion to the network of technologies underlying the tool itself: the W3, or World Wide Web, or, today, simply "the web."

D3's primary author is the brilliant Mike Bostock, although there are a few other dedicated contributors. The project is entirely open source and freely available.

D3 is released under a BSD license, so you may use, modify, and adapt the code for noncommercial or commercial use at no cost.

D3's official home on the web is d3js.org.

What it Does

Fundamentally, D3 is an elegant piece of software that facilitates generation and manipulation of web documents with data. It does this by

- *loading* data into the browser's memory
- *binding* data to elements within the document, creating new elements as needed
- *transforming* those elements by interpreting each element's bound datum and setting its visual properties accordingly, and finally
- *transitioning* elements between states in response to user input.

Learning to use D3 is simply a process of learning the syntax used to tell it how you want it to load and bind data, and transform and transition elements.

The *transformation* step is most important, as this is where the *mapping* happens. D3 provides a structure for applying these transformations, but, as we'll see, you define what the mapping rules are. Should larger values make taller bars or brighter circles? Will clusters be sorted on the x axis by age or category? What color palette is used to fill in countries on your world map? All of the visual design decisions are up to you. You provide the concept, you craft the rules, and D3 executes it — without telling you what to do. (Yes, it's like the opposite of Excel's pushy "Chart Wizard".)

What it Doesn't Do

- D3 doesn't generate predefined or "canned" visualizations for you. (D3's *layouts* are a slight exception to this, as they help you achieve common visualization forms more quickly.)
- D3 doesn't even try to support older browsers. This helps keep the D3 codebase clean, free of hacks to support old versions of Internet Explorer, for example. The philosophy is that by creating more compelling tools and refusing to support older browsers, we encourage more people to upgrade (rather than forestall the process, thereby requiring us to continue to support those browsers, and so on — a vicious cycle). D3 wants us to move forward.
- D3 doesn't handle bitmap map tiles, such as those provided by Google Maps or Cloudmade. D3 is great with anything vector — SVG images or GeoJSON data — but was never intended to work with traditional map tiles. When geomapping, you can either go all-SVG and stick with D3, use a different tool with tiles, or figure out how to integrate D3 for the data layer and the other tool for a base map tile layer. (See "Alternatives" below.) This question comes up a lot in the D3 community, and as of today, there is no easy answer.
- D3 doesn't hide your original data. Since D3 code is executed on the client-side (meaning, in the user's web browser, as opposed to on the web server), the data you want visualized must be sent to the client. If your data can't be shared, then don't use D3. Alternatives include using proprietary tools (like Flash) or pre-rendering visualizations as static images and sending those to the browser. (If you're not interested in sharing your data, though, why would you bother visualizing it? The purpose of visualization is to communicate the data, so you may sleep better at night by choosing openness and transparency, rather than having nightmares about data thieves.)

Origins and Context

The first web browsers rendered static pages; interactivity was limited to clicking on links. In 1996, Netscape introduced the first browser with JavaScript (originally known as ECMAScript), a new scripting language that could be interpreted *by the browser while the page was being viewed*.

This doesn't sound as groundbreaking as it turned out to be, but this enabled web browsers to evolve from merely passive *browsers* to dynamic frames for interactive, networked experiences. This shift ultimately enabled every intra-page interaction we have on the web today. Without JavaScript, D3 would never exist, and web-based data visualizations would be limited to pre-rendered, non-interactive GIFs. (Yuck. Thank you, Netscape!)

Jump ahead to 2005, when Jeffrey Heer (then of the University of California, Berkeley), Stuart Card (of the Palo Alto Research Center), and James Landay (of the University of Washington, Seattle) introduced prefuse, a toolkit for bringing data visualization to the web. prefuse (spelled with all lowercase letters) was written in Java, a compiled language, whose programs could run in web browsers via a Java plug-in.

Note that *Java* is a completely different programming language than *JavaScript*, despite their similar names. (Hey, it was the 90s, and coffee-related words like "java" were really cool. Wouldn't you rather go by "JavaScript" than "ECMAScript"?) Just remember: JavaScript is a scripting language that is interpreted by the web browser. Java programs must be compiled and saved as "applets" to be used on the web, and then are executed by a Java plug-in, not by the browser itself. In practice, this tends to mean that Java applets stand alone in a little box somewhere within the browser window (much like Flash Player content). JavaScript, however, is native to the browser, so it can operate anywhere on the entire page.

Java has now fallen out of favor, in part because its applets are slow to load. Nonetheless, prefuse was a breakthrough application — the first to make web-based visualization accessible to less-than-expert programmers. Until prefuse came along, any datavis on the web was very much a custom affair.

Two years later, in 2007, Jeff Heer introduced Flare, a similar toolkit, but written in ActionScript, so its visualizations could be viewed on the web through Adobe's Flash Player. Flare, like prefuse, relied on a browser plug-in, but the Flash is much faster than Java ever was — it loads in an instant, and interaction is quick and fluid.

Flare was a huge improvement, but as web browsers continued to evolve, it was clear that interactive visualizations could be created with native browser technology, no plug-ins required.

By 2009, Jeff Heer had moved to Stanford, where he was advising a graduate student named Michael Bostock. Together, in Stanford's Vis Group, they created Protovis, a visualization toolkit that expanded on prefuse and Flare, but relied exclusively on native browser technologies. Protovis is written in JavaScript. (By the way, if you have used Protovis, be sure to reference Mike's introduction to D3 for Protovis users.)

Protovis made generating interactive visualizations as straightforward as possible, even for users without prior programming experience. Yet to do so, it had to create an abstract representation layer. The designer could address this layer using Protovis' syntax, but it wasn't accessible through standard methods, so debugging was difficult. Learning Protovis involved learning skills specific to a particular tool that managed data and visuals in a unique way. D3, by contrast, is closer to pure JavaScript, so the skills gained while learning D3 will be useful beyond the realm of datavis.

In 2011, Mike Bostock, Vadim Ogievetsky, and Jeff Heer officially announced D3, the next evolution in web visualization tools. Unlike Protovis, D3 doesn't provide an intermediate, abstract layer in which to operate. Rather, it operates directly on the web document itself. This means easier debugging, easier experimentation, and of course far more visual possibilities. The only downside to this approach is a slightly steeper learning curve. But this book will make that as painless as possible.

If you're familiar with any of these groundbreaking tools, you'll appreciate that D3 descends from a prestigious lineage. And if you have any interest in the philosophy underlying D3's elegant technical design, I highly recommend Mike, Vadim, and Jeff's InfoVis paper which so clearly articulates the need for this kind of tool. The paper encapsulates years' worth of learning and insights made while developing visualization tools.

Alternatives

D3 may not be perfect for every project. Sometimes you just need a quick chart, and you don't have time to code it from scratch. Or you may need to support older browsers, and can't rely on recent technologies like SVG.

For those situations, it's good to know what other tools are out there. Here is a brief, non-comprehensive list of D3 alternatives, all of which use web-standard technologies (mostly JavaScript) and are free to download and use.

Easy Charts

Flot — A plotting library for jQuery that uses the HTML canvas element and supports older browsers, even all the way back to Internet Explorer 6.

gRaphaël — A charting library based on Raphaël (see below). Uses canvas and also supports older browsers.

Highcharts JS — A JavaScript-based charting library with several predesigned themes and chart types. Uses SVG for modern browsers and falls back on VML for old versions of IE. (Free only for noncommercial use.)

JavaScript InfoVis Toolkit — The JIT provides several preset visualization styles for your data.

jqPlot — A plug-in for charting with jQuery. Supports IE7 and newer.

Peity — A jQuery plug-in for very simple and very *tiny* bar, line, and pie charts. Supports only recent browsers.

Timeline.js — A library specifically for generating interactive timelines.

Graph Visualizations

A "graph" is just data with a networked structure (e.g., B is connected to A, and A is connected to C).

Arbor.js — A library for graph visualization using jQuery.

Sigma.js — A very lightweight library for graph visualization.

Geomapping

I distinguish between *mapping* (all visualizations are maps) and *geomapping* (visualizations that include geographic data, or geodata, such as traditional maps). D3 has a lot of geomapping functionality, but you should know about these other tools.

Kartograph — A JavaScript-and-Python combo for gorgeous, entirely vector-based mapping by Gregor Aisch.

Leaflet — A library for tiled maps, designed for smooth interaction on both desktop and mobile devices. Some support for displaying data layers of SVG on top of the map tiles. (Reference Mike's demo "Using D3 with Leaflet".)

Modest Maps — The granddaddy of tiled map libraries, now available for several platforms. Modest Maps has been succeeded by Polymaps.

Polymaps — A library for making tiled maps. Supports SVG for data layers overlaid on the tiles.

Almost From Scratch

These tools, like D3, provide methods of drawing visual forms, but without pre-designed visual templates. If you enjoy the creative freedom of starting from scratch, you may enjoy these.

Processing.js — A native JavaScript implementation of Processing, the fantastic programming language for artists and designers new to programming. Processing is written in Java, so exporting Processing sketches to the web traditionally involved clunky applets. Thanks to Processing.js, regular Processing code can run natively, in the browser. Renders using canvas, so only modern browsers are supported.

Paper.js — A framework for rendering vector graphics to canvas. Also, its website is one of the most beautiful on the Internet, and their demos are unbelievable. (Go play with them now.)

Raphaël — Another library for drawing vector graphics to canvas. Popular due to its friendly syntax and support for older browsers.

Three-Dimensional

D3 is not the best at 3D, simply because web browsers are historically two-dimensional beasts. But with increased support for WebGL, there are now more opportunities for 3D web experiences.

PhiloGL — A WebGL framework specifically for 3D visualization.

Three.js — A library for generating any sort of 3D scene you could imagine, produced by Google's Data Arts team. You could spend all day exploring the mind-blowing demos on their site.

Tools built on D3

For when you want to use D3 without actually writing any D3 code, you can choose one of the many tools built on top of D3!

Cubism — A D3 plug-in for visualizing time series data, written by Mike Bostock. (Also one of my favorite demos.)

Dashku — A tool for real-time updated dashboard and widgets by Paul Jensen.

NVD3 — Reusable charts with D3.

XXXXXXXXXX - STATUS OF NVD3 IS IN FLUX; MAY NEED TO REMOVE THIS OR UPDATE WITH NEW URL

Rickshaw — A toolkit for displaying time series data.

Tributary — A great tool for experimenting with live coding using D3, by Ian Johnson.

CHAPTER 3
Technology Fundamentals

Solid familiarity with the following concepts will make your time with D3 a lot less frustrating and a lot more rewarding. Consider this a brief refresher course on Web-Making 101.

The Web

If you're brand-new to making web pages, you will now have to think about things that regular people blissfully disregard every day, such as: How does the web actually work?

We think of the web as a bunch of interlinked pages, but it's really a collection of conversations between web servers and web clients ("browsers").

The following scene is a dramatization of a typical such conversation that happens whenever you or anyone else clicks a link or types an address into your browser (meaning, this brief conversation is had about 88 zillion times every day):

> CLIENT: I'd really like to know what's going on over at somewebsite.com. I better call over there to get the latest info. [Silent sound of Internet connection being established.]
>
> SERVER: Hello, unknown web client! I am the server hosting somewebsite.com. What page would you like?
>
> CLIENT: This morning, I am interested in the page at: somewebsite.com/news/
>
> SERVER: Of course, one moment.
>
> *Code is transmitted from SERVER to CLIENT.*
>
> CLIENT: I have received it. Thank you!
>
> SERVER: You're welcome! Would love to stay on the line and chat, but I have other requests to process. Bye!

Clients contact servers with *requests*, and servers respond with data. But what is a "server" and what is a "client"? In a nutshell:

Web servers are Internet-connected computers running server software, so called because they *serve* web documents as requested. Servers are typically always-on and always-connected, but web developers often also run *local* servers, meaning they run on the same computer that you're working on. *Local* means here; *remote* means somewhere else.

There are lots of different server software packages, but Apache is the most common. Web server software is not pretty, and no one ever wants to look at it.

In contrast, web *browsers* can be very pretty, and we spend a lot of time looking at them. Regular people recognize names like Firefox, Safari, Chrome, and Internet Explorer, all of which are browsers or *web clients*.

Every web page, in theory, can be identified by its URL (Uniform Resource Locator) or URI (Uniform Resource Identifier). Most people don't know what *URL* stands for, but they recognize them when they see them. By obsolete convention, URLs commonly begin with *www*, as in www.calmingmanatee.com, but with a properly configured server, the *www* part is wholly unnecessary.

Complete URLs consist of three parts:

- An indication of the *communication protocol*, such as HTTP or HTTPS
- The *domain name* of the resource, such as `calmingmanatee.com`
- Any additional locating information, such as the path of the requested file, or any query parameters

A complete URL, then, may look like:

```
http://alignedleft.com/tutorials/d3/
```

Note that the protocol is separated from the domain name by a colon and two forward (regular) slashes. Why two slashes? No reason. The inventor of the web regrets the error.

HTTP stands for HyperText Transfer Protocol, and it's the most common protocol for transferring web content from server to client. The "S" on the end of HTTPS stands for *Secure*. HTTPS connections are used whenever information should be encrypted in transit, such as for online banking or e-commerce.

Let's briefly step through the process of what happens when a normal person goes to visit a website.

1. User runs the web browser of her choice, then types a URL into the address bar, such as `alignedleft.com/tutorials/d3/`. Since she did not specify a protocol, HTTP is assumed, and "http://" is prepended to the URL.

2. The browser then attempts to connect to the server behind alignedleft.com across the network.
3. The server associated with alignedleft.com acknowledges the connection and is taking requests. ("I'll be here all night.")
4. The browser sends a request for the page that lives at `/tutorials/d3/`.
5. The server sends back the HTML content for that page.
6. As the client browser receives the HTML, it discovers references to *other files* needed to assemble and display the entire page, including CSS stylesheets and image files. So it contacts the same server again, once per file, requesting the additional information.
7. The server responds, dispatching each file as needed.
8. Finally, all the web documents have been transferred over. Now the client performs its most arduous task, which is to *render* the content. It first parses through the HTML, to understand the structure of the content. Then it reviews the CSS selectors, applying any properties to matched elements. Finally, it plugs in any image files and executes any JavaScript code.

Can you believe that all that happens every time you click a link? It's a lot more complicated than most people realize, but it's important to understand that client-server conversations are fundamental the web.

HTML

Hypertext Markup Language is used to structure content for web browsers. HTML is stored in plain text files with the *.html* suffix. A simple HTML document looks like this:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <h1>Page Title</h1>
        <p>This is a really interesting paragraph.</p>
    </body>
</html>
```

HTML is a complex language with a rich history. This overview will address only the current iteration of HTML (formerly known as HTML5) and will only touch on what is immediately relevant for our practice with D3.

Content + Structure

The core function of HTML is to enable you to "mark up" content, thereby giving it structure. Take, for example, this raw text:

```
Amazing Visualization Tool Cures All Ills A new open-source tool
designed for visualization of data turns out to have an
unexpected, positive side effect: it heals any ailments of the viewer. Leading
scientists report that the tool, called D3000, can cure even the
following symptoms: fevers chills general malaise It is achieves this
end with a patented, three-step process. Load in data. Generate a visual
representation. Activate magic healing function.
```

Reading between the lines, we can infer that this is a very exciting news story. But as unstructured content, it is very hard to read. By adding structure, we can differentiate between the headline, for example, and the body of the story.

Amazing Visualization Tool Cures All Ills

A new open-source tool designed for visualization of data turns out to have an unexpected, positive side effect: it heals any ailments of the viewer. Leading scientists report that the tool, called D3000, can cure even the following symptoms:

- fevers
- chills
- general malaise

It is achieves this end with a patented, three-step process.

1. Load in data.
2. Generate a visual representation.
3. Activate magic healing function.

Above, we have the same raw text content, but with a *visual structure* that makes the content more accessible.

HTML is a tool for specifying *semantic structure*, or attaching hierarchy, relationships, and *meaning* to content. (HTML doesn't address the visual representation of a document's structure — that's CSS's job.) Here is our story with each chunk of content replaced by a *semantic description* of what that content is.

Headline

Paragraph text

- Unordered list item
- Unordered list item

- Unordered list item

Paragraph text

1. Numbered list item
2. Numbered list item
3. Numbered list item

This is the kind of structure we specify with HTML markup.

Adding Structure with Tags

The process of "marking up" is a process of adding *tags* to content. HTML tags begin with < and end with >, as in , which is the tag indicating a paragraph of text. Each new tag creates a new *element* in the document structure.

When tags occur in pairs, closing tags are indicated with a slash that closes or ends the element, as in . Thus, a paragraph of text may be marked up as

```
<p>This is a really interesting paragraph.</p>
```

Most tags can be *nested*. For example, here we use the em tag to add *emphasis*.

```
<p>This is a <em>really</em> interesting paragraph.</p>
```

Nesting tags introduces hierarchy to the document. In this case, em is a child of p, since it is contained by p. (Conversely, p is `em`'s parent.)

Tags may be nested, but they cannot overlap closures of their parent tags, as doing so would disrupt the hierarchy. For example:

```
<p>This could cause <em>unexpected</p>
<p>results</em>, and is best avoided.</p>
```

Some tags never occur in pairs, such as img, which references an image file. Although HTML no longer requires it, you may sometime see such tags written in *self-closing* fashion, with a trailing slash before the closing bracket.

```
<img src="photo.jpg" />
```

As of HTML5, the self-closing slash is optional, so this code is equivalent:

```
<img src="photo.jpg">
```

Common Tags

There are hundreds of different HTML tags. Here are some of the most common. We'll cover additional tags in later chapters. (Reference the excellent Mozilla Developer Network documentation for a complete listing.)

- `<!DOCTYPE html>` — The standard document type declaration. Must be the first thing in the document.
- `html` — Surrounds all HTML content in a document.
- `head` — The document `head` contains all metadata about the document, such as its `title` and any references to external stylesheets and scripts.
- `title` — The title of the document. Browsers typically display this at the top of the browser window, and use this title when bookmarking a page.
- `body` — Everything not in the `head` should go in the `body`. This is the primary visible content of the page.
- `h1`, `h2`, `h3`, `h4` — These tags let you specify heading of different levels. `h1` is a top-level heading, `h2` is below that, and so on.
- `p` — A paragraph!
- `ul`, `ol`, `li` — Unordered lists are specified with `ul`, most often used for bulleted lists. Ordered lists (`ol`) are often numbered. Both `ul` and `ol` should include `li` tags to specify list items.
- `em` — Indicates emphasis. Typically rendered in *italics*.
- `strong` — Indicates additional emphasis. Typically rendered in **boldface**.
- `a` — A link! Typically rendered as underlined, blue text, unless otherwise specified.
- `span` — An arbitrary `span` of text, typically within a larger containing element like `p`.
- `div` — An arbitrary *division* within the document. Used for grouping and containing related elements.

Our earlier example could be given semantic structure using some of these tags like so:

```
<h1>Amazing Visualization Tool Cures All Ills</h1>
<p>A new open-source tool designed for visualization of data turns out to have an unexpected, posi
<ul>
    <li>fevers</li>
    <li>chills</li>
    <li>general malaise</li>
</ul>
<p>It is achieves this end with a patented, three-step process.</p>
<ol>
    <li>Load in data.</li>
    <li>Generate a visual representation.</li>
    <li>Activate magic healing function.</li>
</ol>
```

When viewed in a web browser, that markup is rendered as:

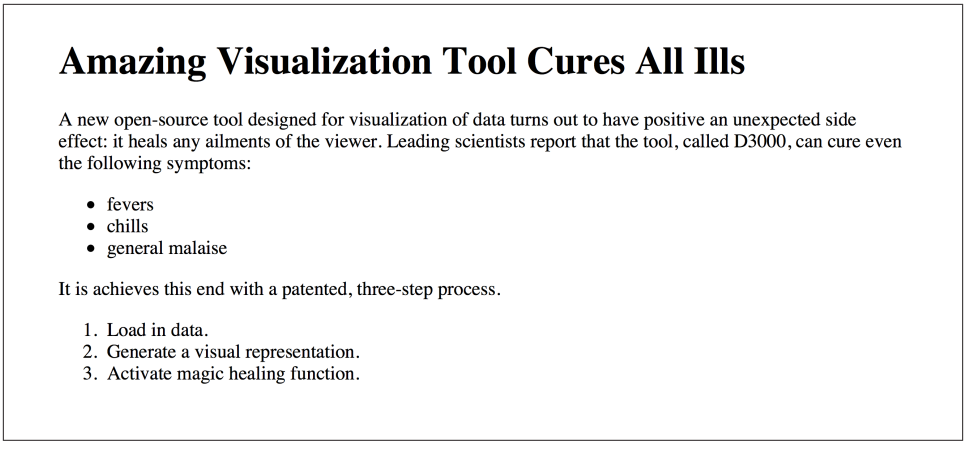


Figure 3-1. Typical default rendering of simple HTML

Notice that we specified only the *semantic* structure of the content; we didn't specify any visual properties, such as color, type size, indents, or line spacing. Without such instructions, the browser falls back on its *default styles*, which, frankly, are not too exciting.

Attributes

All HTML elements can be assigned *attributes* by including property/value pairs in the opening tag.

```
<tagname property="value"></tagname>
```

The name of the property is followed by an equals sign, and the value is enclosed within double quotation marks.

Different kinds of tags can be assigned different attributes. For example, the `a` link tag can be given an `href` attribute, whose value specifies the URL for that link. (`href` is short for "HTTP reference".)

```
<a href="http://d3js.org/">The D3 website</a>
```

Some attributes can be assigned to *any* type of element, such as `class` and `id`.

Classes and IDs

Classes and IDs are extremely useful attributes, as they can be referenced later to identify specific pieces of content. Your CSS and JavaScript code will rely heavily on classes and IDs to identify elements. For example:

```
<p>Brilliant paragraph</p>
<p>Insightful paragraph</p>
<p class="awesome">Awe-inspiring paragraph</p>
```

These are three very uplifting paragraphs, but only one of them is truly awesome, as I've indicated with `class="awesome"`. The third paragraph becomes part of a *class* of *awesome* elements, and it can be selected and manipulated along with other class members. (We'll get to that in a moment.)

Elements can be assigned multiple classes, simply by separating them with a space:

```
<p class="uplifting">Brilliant paragraph</p>
<p class="uplifting">Insightful paragraph</p>
<p class="uplifting awesome">Awe-inspiring paragraph</p>
```

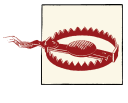
Now, all three paragraphs above are `uplifting`, but only the last one is both `uplifting` *and* `awesome`.

IDs are used in much the same way, but there can only be one ID per element, and each ID value can be used only once on the page. For example:

```
<div id="content">
    <div id="visualization"></div>
    <div id="button"></div>
</div>
```

IDs are useful for when a single element has some special quality, like a `div` that functions as a button or as a container for other content on the page.

As a general rule, if there will be only *one* such element on the page, you can use an `id`. Otherwise, use a `class`.



Class and ID names cannot begin with numerals; they must begin with alphabetic characters. So `id="1"` won't work, but `id="item1"` will. The browser will not give you any errors; your code simply won't work, and you will go crazy trying to figure out why.

Comments

As code grows in size and complexity, it is good practice to include comments. These are friendly notes that you leave for yourself to remind you why you wrote the code the way you did. If you are like me, you will revisit projects only weeks later and have lost all recollections of it. Commenting is an easy way to reach out and provide guidance and solace to your future (and very confused) self.

In HTML, comments are written as:

```
<!-- Your comment here -->
```

Anything between the `<!--` and `-->` will be ignored.

DOM

The term Document Object Model refers to the hierarchical structure of HTML. Each bracketed tag is an *element*, and we refer to elements' relative relationships to each other in human terms: parent, child, sibling, ancestor, and descendant. For example, in this HTML:

```
<html>
    <body>
        <h1>Breaking News</h1>
        <p></p>
    </body>
</html>
```

`body` is the parent element to both of its children, `h1` and `p` (which are siblings to each other). All elements on the page are descendants of `html`.

Web browsers parse the DOM in order to make sense of a page's content. As coders building visualizations, we care about the DOM because our code must navigate its hierarchy in order to apply styles and actions to its elements. We don't want to make *all* the `div` elements blue; we need to know how to select just the `div`s of the class `sky` and make *them* blue.

Developer Tools

In any browser, you can select "View Source" to see the original HTML source of the page. (In Safari, the option is under the View menu. In Chrome, it's under View > Developer > View Source.)

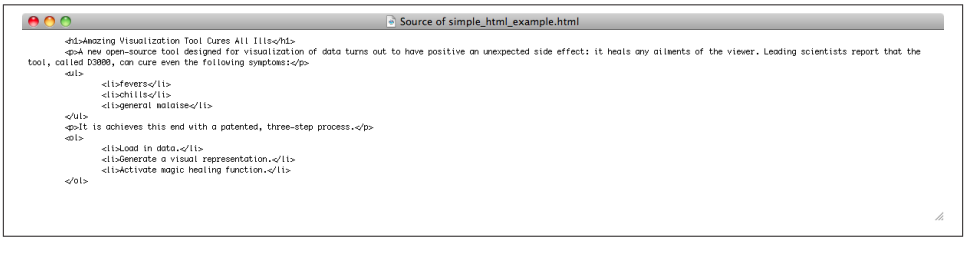


Figure 3-2. Safari's plain source view window

That gets you the raw HTML, but if any D3 or JavaScript code has been executed, the current DOM may be vastly different.

Fortunately, your browser's developer tools enable you to see the current state of the DOM. Safari and Chrome share the same rendering engine (WebKit) as well as a similar web inspector. In Safari, you must enable the developer tools in Safari > Preferences > Advanced > Show Develop in menu bar. Then, in the Develop menu, choose Show Web Inspector. In Chrome, just select View > Developer > Developer Tools.

The Elements tab of the inspector shows you the *current state of the DOM*. This is useful because your code will modify DOM elements dynamically. In the web inspector, you can watch elements as they change.

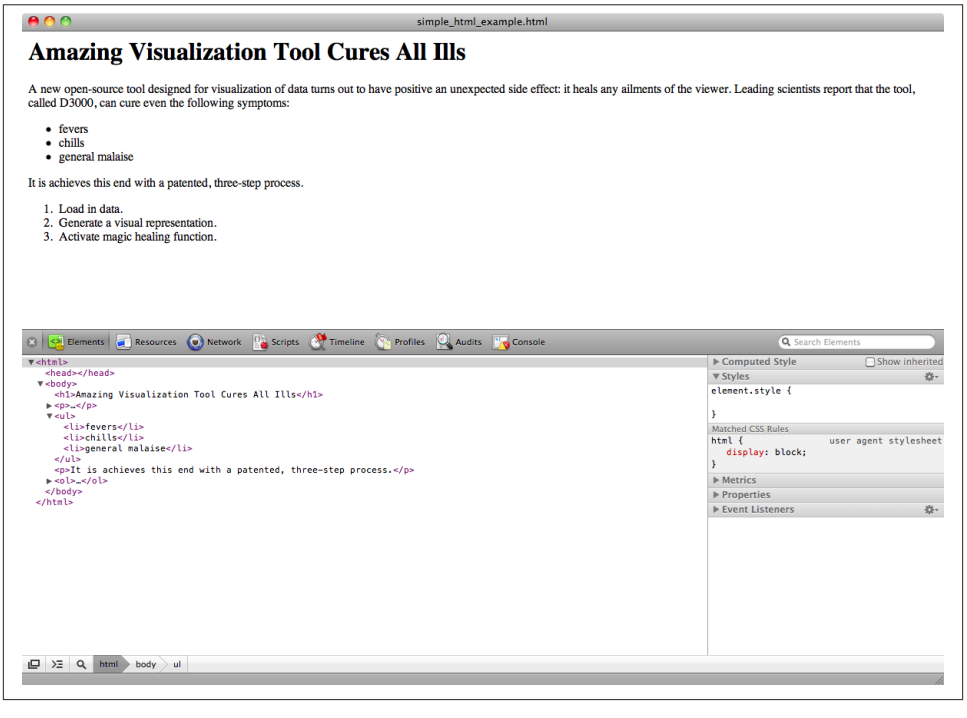


Figure 3-3. Safari's web inspector

If you look closely, you'll already see some differences between the raw HTML and the DOM, including the fact that Safari generated the required `html`, `head`, and `body` elements. (I was lazy and didn't include them in my original HTML.)

One more thing: Why am I focusing on Chrome and Safari? Why not Firefox, Internet Explorer, Opera, or any of the other many browsers out there? For one, it's best to develop your projects using a browser with the broadest support for web standards. That eliminates IE, as its current release (version 9) leaves a lot to be desired, standards-wise.

Second, for my way of working, Safari and Chrome have the easiest-to-use and most functional developer tools. Firefox is a great browser, but it currently doesn't have dev tools built-in. (Firefox requires installation of Firebug, a plug-in.) I recommend you try them all and decide what works best for you.

Rendering and the Box Model

Rendering is the process by which browsers, after parsing the HTML and generating the DOM, apply *visual rules* to the DOM contents and draw those pixels to the screen.

The most important thing to keep in mind when considering how browsers render content is:

To a browser, everything is a box.

Paragraphs, `div`s, `span`s — all are boxes in the sense that they are two-dimensional rectangles, with properties that any rectangle can have, such as width, height, and positions in space. Even if something looks curved or irregularly shaped, rest assured, to the browser, it is merely another rectangular box.

You can see these boxes with the help of the web inspector. Just mouse over any element and the box associated with that element is highlighted in blue.

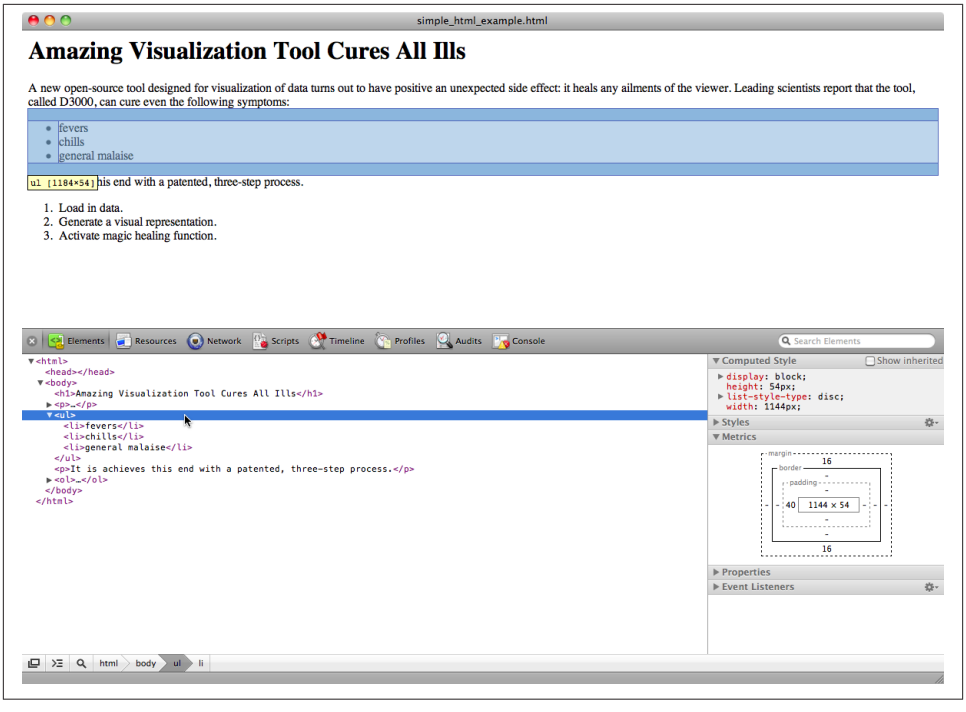


Figure 3-4. Inspector with element box highlighted

There's a lot of information about the `ul` unordered list here. Note that the list's total dimensions (width and height) are shown in the yellow box at the element's lower left corner. Also, the list's position in the DOM hierarchy is indicated with chevrons in the lower left corner of the inspector: `html > body > ul`

The box for the `ul` expands to fill the width of the entire window because it is a *block-level* element. (Note how under "Computed Style" is listed `display: block`.) This is in contrast to *inline* elements, which rest *in line* with each other, not stacked on top of each other like blocks. Common inline elements include: `strong`, `em`, `a`, and `span`.

By default, block-level elements expand to fill their container elements and force any subsequent sibling elements further down the page. Inline elements do not expand to fill extra space, and happily exist side-by-side, next to their fellow inline neighbors. (Discussion question: Which kind of element would you rather be?)

CSS

Cascading Style Sheets are used to style the visual presentation of DOM elements. A simple CSS stylesheet looks like this:

```
body {
    background-color: white;
    color: black;
}
```

CSS styles consist of *selectors* and *properties*. Selectors are followed by properties, grouped in curly brackets. A property and its value is separated by a colon, and the line is terminated with a semicolon, like so:

```
selector {
    property: value;
    property: value;
    property: value;
}
```

The same properties can be applied to multiple selectors at once by separating the selectors with a comma, as in:

```
selectorA,
selectorB,
selectorC {
    property: value;
    property: value;
    property: value;
}
```

For example, you may want to specify that both `p` paragraphs and `li` list items should use the same font size, line height, and color:

```
p,
li {
    font-size: 12px;
    line-height: 14px;
    color: orange;
}
```

Collectively, this whole chunk of code (selectors + brackets properties) is called a *CSS rule*.

Selectors

D3 uses CSS-style selectors to identify elements on which to operate, so it's important to understand how to use them.

Selectors identify specific elements to which styles will be applied. There are several different kinds of selectors. We'll use only the simplest ones in this book.

Type selectors are the easiest. They match DOM elements with the same name.

```
h1          /* Selects level 1 headings              */
p           /* Selects all paragraphs                */
strong      /* Selects all strong elements           */
em          /* Selects all em elements               */
div         /* Selects all divs                      */
```

Descendant selectors match elements that are contained by (or "descended from") another element. We will rely heavily on descendant selectors to apply styles.

```
h1 em       /* Selects em elements contained in an h1 */
div p       /* Selects p elements contained in a div  */
```

Class selectors match elements of any type that have been assigned a specific class. Class names are preceded with a period.

```
.caption    /* Selects elements with class "caption"  */
.label      /* Selects elements with class "label"    */
.axis       /* Selects elements with class "axis"     */
```

Since elements can have more than one class, you can target elements with multiple classes by stringing the classes together, as in:

```
.bar.highlight  /* Could target highlighted bars      */
.axis.x         /* Could target an x-axis             */
.axis.y         /* Could target a y-axis              */
```

`.axis` could be used to apply styles to both axes, for example, while `.axis.x` would apply only to the x-axis.

ID selectors match the single with a given ID. (Remember, IDs can be used only once each in the DOM!) IDs are preceded with a hash mark.

```
#header     /* Selects element with ID "header"       */
#nav        /* Selects element with ID "nav"          */
#export     /* Selects element with ID "export"       */
```

Selectors get progressively more useful as you combine them in different ways to target specific elements. You can string selectors together to get very specific results. For example:

```
div.sidebar /* Selects divs with class "sidebar", but
               not other elements with that class     */
#button.on  /* Selects element with ID "button", but
               only when the class "on" is applied    */
```

Remember, since the DOM is dynamic, classes and IDs can be added and removed, so you may have CSS rules that apply only in certain scenarios.

For details on additional selectors, see the Mozilla Developer Network.

Properties and Values

Groups of property/value pairs cumulatively form the styles.

```
margin: 10px;
padding: 25px;
background-color: yellow;
color: pink;
font-family: Helvetica, Arial, sans-serif;
```

At the risk of stating the obvious, notice that each property expects a different kind of information. `color` wants a color, `margin` requires a measurement (here in `px` or pixels), and so on.

By the way, colors can be specified in several different formats:

- named colors — `orange`
- hex values — `#3388aa` or `#38a`
- RGB values — `rgb(10, 150, 20)`
- RGB with alpha transparency — `rgba(10, 150, 20, 0.5)`

You can find exhaustive lists of properties online; I won't try to list them here. Instead, I'll just introduce relevant properties as we go.

Comments

```
/* By the way, this is what a comment looks like
   in CSS.  They start with a slash-asterisk pair,
   and end with an asterisk-slash pair.  Anything
   in between will be ignored. */
```

Referencing Styles

There are three common ways to apply CSS style rules to your HTML document.

1. Embed the CSS in your HTML.

If you embed the CSS rules in your HTML document, you can keep everything in one file. In the document `head`, include all CSS code within a `style` element.

```
<html>
    <head>
        <style type="text/css">

            p {
                font-size: 24px;
                font-weight: bold;
                background-color: red;
                color: white;
            }

        </style>
    </head>
```

```
    <body>
        <p>If I were to ask you, as a mere paragraph, would you say that I have style?</p>
    </body>
</html>
```

That HTML page with CSS renders as:

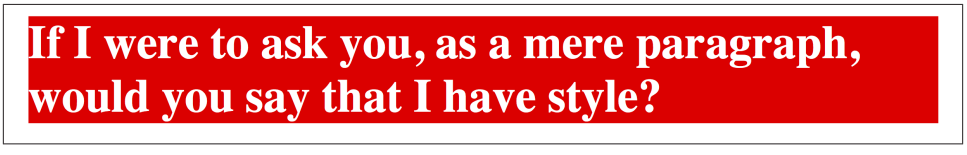


Figure 3-5. Rendering of an embedded CSS rule

Embedding is the simplest option, but I generally prefer to keep different kinds of code (e.g., HTML, CSS, JavaScript) in separate documents.

2. Reference an external style sheet from the HTML.

To store CSS outside of your HTML, save it in a plain text file with a *.css* suffix, like `style.css`. Then use a `link` element in the document `head` to reference the external CSS file, like so:

```
<html>
    <head>
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <p>If I were to ask you, as a mere paragraph, would you say that I have style?</p>
    </body>
</html>
```

This example renders exactly the same as the prior example.

3. Attach inline styles

A third method is to attach style rules *inline* directly to elements in the HTML. You can do this by adding a `style` attribute to any element. Then include the CSS rules within the double quotation marks.

```
<p style="color: blue; font-size: 48px; font-style: italic;">Inline styles are kind of a hassle</p
```

Figure 3-6. Rendering of an inline CSS rule

Since inline styles are attached directly to elements, there is no need for selectors.

Inline styles are messy and hard to read, but they are useful for giving special treatment to a single element, when that style information doesn't make sense in a larger stylesheet. We'll learn how to apply inline styles programmatically with D3 (which is much easier than typing them in by hand, one at a time).

Inheritance, Cascading, and Specificity

Many style properties are *inherited* by an element's descendants unless otherwise specified. For example, this document's style rule applies to the `div`:

```
<html>
    <head>
        <title></title>
        <style type="text/css">

            div {
                background-color: red;
                font-size: 24px;
                font-weight: bold;
                color: white;
            }

        </style>
    </head>
    <body>
        <p>I am a sibling to the div.</p>
        <div>
            <p>I am a descendant and child of the div.</p>
        </div>
    </body>
</html>
```

Yet when this page renders, the styles intended for the `div` (red background, bold text, and so on) are *inherited* by the second paragraph, since that `p` is a descendant of the styled `div`.

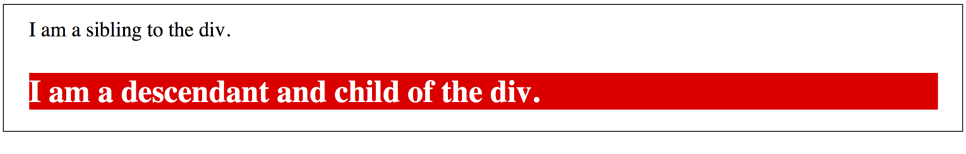


Figure 3-7. Inherited style

Inheritance is a great feature of CSS, as child adopt the styles of their parents. (There's a metaphor in there somewhere.)

Finally, an answer to the most pressing question of the day: Why are they called *Cascading* Style Sheets? It's because selector matches *cascade* from the top down. When more than one selector applies to an element, the *later* rule generally overrides the *earlier* one. For example:

```
p {
    color: blue;
}

p.highlight {
    color: black;
    background-color: yellow;
}
```

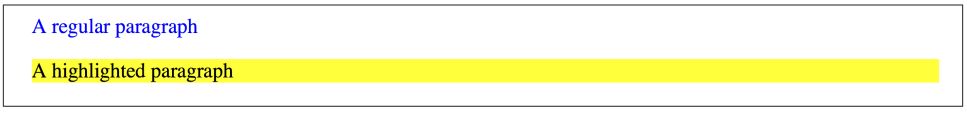


Figure 3-8. CSS cascading and inheritance at work

These rules set the text of all paragraph elements in the document to be blue *except* for those with the class of `highlight` applied, which will be black *and* have a yellow background. The rules for `p` are applied first, but then the rules for `p.highlight` override the less-specific `p` rules.

Later rules *generally* override earlier ones, but not always. The true logic has to do with the *specificity* of each selector. The `p.highlight` selector would override the `p` rule even if it were listed first, simply because it is a more specific selector. If two selectors have the same specificity, then the later one will be applied.

This is one of the main causes of confusion with CSS. The rules for calculating specificity are inscrutable, and I won't cover them here. To save yourself headaches later, keep your selectors clear and easy to read. Start with general selectors on top, and work your way down to more specific ones, and you'll be all right.

JavaScript

JavaScript is the scripting language that can make pages dynamic by manipulating the DOM after a page has already loaded in the browser. As I mentioned earlier, getting to know D3 is also a process of getting to know JavaScript. So we'll dig in deeper as we go, but here is a taste to get you started.

Hello, Console

Normally, we write JavaScript code in a text file, and then load that file to the browser in a web page. But you can also just type JavaScript code directly into your browser! This is an easy and quick way to get your feet wet and test out code.

We'll also use the JavaScript console for debugging, as it's an essential tool for seeing what's going on with your code. (Later, you may want to read more information on debugging HTML, CSS, and JavaScript with the web inspector and console.)

In Safari's Develop menu, choose Show Error Console, or open the web inspector and click the Console tab. In Chrome, just select View > Developer > JavaScript Console.

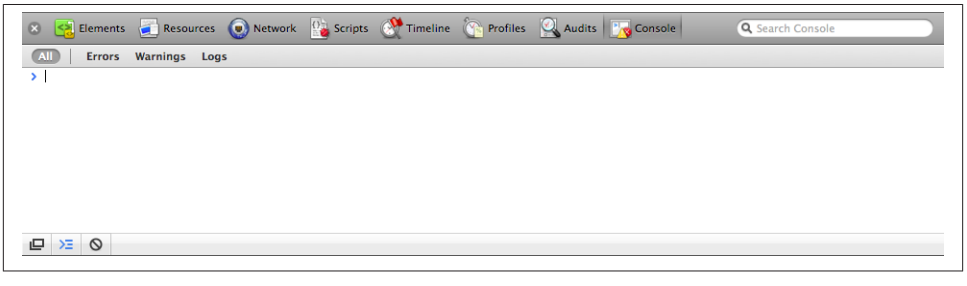


Figure 3-9. A fresh JavaScript console… delicious!

The console accepts one line of code at a time, and it always spits back the result of whatever you input. For example, if you enter the number `7`, the console returns the mind-numbingly obvious result of `7`. Brilliant.

Other times, you'll want your script to print values out to the console automatically, so you can keep an eye on what's going on. As you'll see in some examples below, you can use `console.log("something");` to do that.

Please follow along, and type the following examples into the console to see how it works.

Variables

Variables are containers for data. A simple variable holds one value.

```
var number = 5;
```

In the statement above, `var` indicates you are declaring a new variable, the name of which is `number`. The equals sign is an *assignment operator* because it takes the value on the right (`5`) and *assigns* it to the variable on the left (`number`). So when you see something like

```
defaultColor = "hot pink";
```

try to read = not as "equals," but as "is set to." So the statement above could be stated in plain English as "The variable defaultColor *is set to* hot pink."

As you've just seen, variables can store numeric values as well as strings of text, so called because they are formed by "stringing together" individual characters of text. Strings must be enclosed by quotation marks. True or false (Boolean) values can also be stored:

```
var thisMakesSenseSoFar = true;
```

Also note that in JavaScript, statements are concluded with a semicolon.

You can try making some variables yourself in the console. For example, type `var amount = 200`, then hit enter, then on a new line just enter `amount` and press enter. You should see `200` returned to you in the console — proof that JavaScript remembered the value you assigned to `amount`!

JS is a Loosey Goosey Language

If you're new to JavaScript, know that it is a *loosely typed* language, meaning you don't have to specify what *type* of information will be stored in a variable in advance. Many other languages, like Java (which is completely different from Java*Script*!), require you to declare a variable's type, such as `int`, `float`, `boolean`, or `String`.

```
//Declaring variables in Java
int number = 5;
float value = 12.3467;
boolean active = true;
String text = "Crystal clear";
```

JavaScript, however, automatically *types* a variable based on what kind of information you assign to it. (Note that `''` or `""` indicate string values. I prefer double quotation marks `""`, but some people like singles `''`.)

```
//Declaring variables in JavaScript
var number = 5;
var value = 12.3467;
var active = true;
var text = "Crystal clear";
```

How boring — `var`, `var`, `var`, `var`! — yet handy, as we can declare and name variables before we even know what type of data will go into them. You can even change a variable's type on the fly without JavaScript freaking out on you.

```
var value = 100;
value = 99.9999;
value = false;
value = "This can't possibly work.";
value = "Argh, it does work! No errorzzzz!";
```

Other Variable Types

A variable is a datum, the smallest building block of data. The variable is the foundation of all other data structures, which are simply different configurations of variables.

Now we'll address some of these more complex forms of data, including arrays, objects, and arrays of objects. You may want to skip this section for now, but reference it later, once you're ready to load your own data into D3.

Arrays. An array is a sequence of values, conveniently stored in a single variable.

Keeping track of related values in separate variables is inefficient:

```
var numberA = 5;
var numberB = 10;
var numberC = 15;
var numberD = 20;
var numberE = 25;
```

Rewritten as an array, those values are much simpler. Hard brackets `[ ]` indicate an array, while each value is separated by a comma:

```
var numbers = [ 5, 10, 15, 20, 25 ];
```

Arrays are ubiquitous in data visualization, so you will become very comfortable with them. You can access (retrieve) a value in an array by using *bracket notation*:

```
numbers[2]  //Returns 15
```

The numeral in the bracket refers to a corresponding position in the array. Remember, array positions begin counting at zero, so the first position is `0`, the second position is `1`, and so on.

```
numbers[0]  //Returns 5
numbers[1]  //Returns 10
numbers[2]  //Returns 15
numbers[3]  //Returns 20
numbers[4]  //Returns 25
```

Some people find it helpful to think of arrays in spatial terms, as though they have rows and columns, like in a spreadsheet:

Position	Value
0	5
1	10
2	15
3	20
4	25

Arrays can contain any type of data, not just integers.

```
var percentages = [ 0.55, 0.32, 0.91 ];
var names = [ "Ernie", "Bert", "Oscar" ];

percentages[1]  //Returns 0.32
names[1]        //Returns "Bert"
```

Although I don't recommend it, different types of values can even be stored within the same array.

```
var mishmash = [ 1, 2, 3, 4.5, 5.6, "oh boy", "say it isn't", true ];
```

See what I mean about being loosely typed? JavaScript is *very* loose.

Objects. Arrays are great for simple lists of values, but with more complex data sets, you'll want to put your data into an object. For our purposes, think of a JavaScript object as a custom data structure. We use curly brackets `{}` to indicate an object. In between the brackets, we include *indices* and *values*. A colon `:` separates each index and its value, and a comma separates each index/value pair.

```
var fruit = {
    kind: "grape",
    color: "red",
    quantity: 12,
    tasty: true
};
```

To reference each value, we use *dot notation*, specifying the name of the index:

```
fruit.kind      //Returns "grape"
fruit.color     //Returns "red"
fruit.quantity  //Returns 12
fruit.tasty     //Returns true
```

Think of the value as "belonging" to the object. Oh, look, some fruit. "What kind of fruit is that?" you might ask. As it turns out, `fruit.kind` is `"grape"`. "Are they tasty?" Oh, definitely, because `fruit.tasty` is `true`.

Objects + Arrays. You can combine these two structures to create arrays of objects, or objects of arrays, or objects of objects or, well, basically whatever structure makes sense for your data set. (JavaScript is loosey goosey to the max.)

Let's say we have acquired a couple more pieces of fruit, and we want to expand our catalogue accordingly. We use hard brackets `[]` on the outside, to indicate an array, followed by curly brackets `{}` and object notation on the inside, with each object separated by a comma.

```
var fruits = [
    {
        kind: "grape",
        color: "red",
```

```
        quantity: 12,
        tasty: true
    },
    {
        kind: "kiwi",
        color: "brown",
        quantity: 98,
        tasty: true
    },
    {
        kind: "banana",
        color: "yellow",
        quantity: 0,
        tasty: true
    }
];
```

To access this data, we just follow the trail of indices down to the values we want. Remember, [] means array, and {} means object. `fruits` is an array, so first we use bracket notation to specify an array index:

```
fruit[1]
```

Next, each array element is an object, so just tack on a dot and an index:

```
fruit[1].quantity   //Returns 98
```

Here's a map of how to access every value in the `fruits` array of objects:

```
fruits[0].kind      ==  "grape"
fruits[0].color     ==  "red"
fruits[0].quantity  ==  12
fruits[0].tasty     ==  true

fruits[1].kind      ==  "kiwi"
fruits[1].color     ==  "brown"
fruits[1].quantity  ==  98
fruits[1].tasty     ==  true

fruits[2].kind      ==  "banana"
fruits[2].color     ==  "yellow"
fruits[2].quantity  ==  0
fruits[2].tasty     ==  true
```

Yes, that's right, we have `fruits[2].quantity` bananas.

JSON. At some point in your D3 career, you will encounter JavaScript Object Notation. You can read up on the details, but JSON is basically a specific syntax for organizing data as JavaScript objects. The syntax is optimized for use with JavaScript (obviously) and AJAX requests, which is why you'll see a lot of web-based APIs that return data formatted as JSON. It's faster and easier to parse with JavaScript than XML, and of course D3 works well with it.

All that, and it doesn't look much weirder than what we've already seen:

```
var jsonFruit = {
    "kind": "grape",
    "color": "red",
    "quantity": 12,
    "tasty": true
};
```

The only difference here is that our indices are now surrounded by double quotation marks `""`, making them string values.

GeoJSON. Just as JSON is just a formalization of existing JavaScript object syntax, GeoJSON is a formalized syntax of JSON objects, optimized for storing geodata. All GeoJSON object are JSON objects, and all JSON objects are JavaScript objects.

GeoJSON can store points in geographical space (typically as longitude/latitude coordinates), but also shapes (like lines and polygons) and other spatial features. If you have a lot of geodata, it's worth it to parse it into GeoJSON format for best use with D3.

We'll get into the details of GeoJSON when we talk about geomaps, but for now, just know that this is what simple GeoJSON data could look like:

```
var geodata = {
    "type": "FeatureCollection",
    "features": [
        {
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [ 150.1282427, -24.471803 ]
            },
            "properties": {
                "type": "town"
            }
        }
    ]
};
```

(Confusingly, longitude is always listed before latitude. Get used to thinking in terms of lon/lat instead of lat/lon.)

Mathematical Operators

You can add, subtract, multiply, and divide values using the following operators, respectively:

```
+   //Add
-   //Subtract
*   //Multiply
/   //Divide
```

For example, type `8 * 2` into the console, hit enter, and you should see `16`. More examples:

```
1 + 2       //Returns 3
10 - 0.5    //Retuns 9.5
33 * 3      //Returns 99
3 / 4       //Returns 0.75
```

Comparison Operators

You can compare values against each other using the following operators:

```
==  //Equal to
!=  //Not equal to
<   //Less than
>   //Greater than
<=  //Less than or equal to
>=  //Greater than or equal to
```

These all compare some value on the left to some value on the right. If the result is true, then `true` is returned. If the result is false, `false` is returned.

Try it! Type each of the following into the console and see what you get:

```
3 == 3
3 == 5
3 >= 3
3 >= 2
100 < 2
298 != 298
```

(JavaScript also offers the `===` and `!==` operators, which perform equality comparisons without type coercion, but we don't need to go into that fine point here.)

Control Structures

Whenever your code needs to make a decision or repeat something, you need a control structure. There are lots to choose from, but we are primarily interested in `if` statements and `for` loops.

if() Only

An `if` statement uses comparison operators to determine if a statement is true or false.

```
if (test) {
    //Code to run if true
}
```

If the test between parantheses is `true`, then the code between the curly brackets is run. If the test turns up `false`, then the bracketed code is ignored, and life goes on. (Technically, life goes on either way.)

```
if (3 < 5) {
    console.log("Eureka! Three is less than five!");
}
```

In the example above, the bracketed code will always be executed, because `3 < 5` is always true. `if` statements are more useful when comparing variables or other conditions that change.

```
if (someValue < anotherValue) {
    //Set someValue to anotherValue, thereby restoring
    //a sense of balance and equilibrium to the world.
    someValue = anotherValue;
}
```

for() Now

You can use `for` loops to repeatedly execute the same code, with slight variations. A `for` loop uses this syntax:

```
for (initialization; test; update) {
    //Code to run each time through the loop
}
```

They are so-called because they loop through the code *for* as many times as specified. First, the initialization statement is run. Then, the test is evaluated, like a mini-`if` statement. If the test is true, then the bracketed code is run. Finally, the update statement is run, and the test is re-evaluated.

The most common application of a `for` loop is to increase some variable by `1` each time through the loop. The test statement can then control how many times the loop is run by referencing that value. (The variable is often named `i`, purely by convention, because it is short and easy to type.)

```
for (var i = 0; i < 5; i++) {
    console.log(i);  //Prints value to console
}
```

The `for` loop above prints the following to the console:

```
0
1
2
3
4
```

The first time through, a variable named `i` is declared and set to zero. `i` is less than five, so the bracketed code is run, printing the current value of `i` (zero) to the console. Finally, the value of `i` is increased by one. (`i++` is shorthand for `i = i + 1`.) So now `i` is `1`, so the test again evaluates as true, the bracketed code is run again, but this time the value printed to the console is one.

As you can see, reading prose descriptions of loops is about as interesting as executing them by hand yourself. This is why we invented computers. So I'll skip to the end and point out that after the final iteration, `i` is increased to five, after which the test returns false, and the loop is over.

It's important to note that counting began at zero, and not one. That is, the "first" value of `i` was `0`. Why not `1`? This is another arbitrary convention, but it nicely parallels how computers count arrays of values.

What Arrays Are Made for()

Code-based data visualization would not be possible without arrays and the mighty `for()` loop. Together, they form a data geek's dynamic duo. (If you do not consider yourself a "data geek," then may I remind you that you are reading a book titled "Interactive Data Visualization for the Web.")

An array organizes lots of data values in one convenient place. Then `for()` can quickly "loop" through every value in an array and perform some action with it — such as, express the value as a visual form. D3 often manages this looping for us, such as with its magical `data()` method.

Note this example, which loops through each of the values in an array called `numbers`:

```
var numbers = [ 8, 100, 22, 98, 99, 45 ];

for (var i = 0; i < numbers.length; i++) {
    console.log(numbers[i]);  //Print value to console
}
```

See that `numbers.length`? That's the beautiful part. `length` is a property of every array. In this case, `numbers` contains six values, so `numbers.length` resolves to `6`, and the loop runs six times. If `numbers` were ten positions long, the loop would run ten times. If it were ten million positions long… yeah, you get it. This is what computers are good at: taking a set of instructions and executing them over and over. And this is at the heart of why data visualization can be so rewarding — you design and code the visualization system, and the system will respond appropriately, even as you feed it different data. The system's mapping rules are consistent, even when the data are not.

Functions

Functions are chunks of code that do things!

More specifically, functions are special because they can take arguments or parameters as input, and then return values as output. Parentheses are used to *call* (execute) a function. If that function requires any arguments (input values), then they are *passed* to the function by including them in the parentheses.

Whenever you see something like this, you know it's a function:

```
calculateGratuity(38.40);
```

In fact, you've already seen functions at work with `console.log`, as in:

```
console.log("Look at me; I can do stuff!");
```

But the best part about functions is that you can define your own. There are several ways to define functions in JavaScript, but here's the simplest, which I'll use throughout this book:

```
var calculateGratuity = function(bill) {
    return bill * 0.2;
}
```

This declares a new variable named `calculateGratuity`. Then, instead of assigning a simple number or string, we store an entire function in the variable! In the parentheses, we name `bill`, another variable to be used only by the function itself. `bill` is the expected input. When called, the function will take that input, multiply it by `0.2`, and *return* the result as its output.

So now if we call

```
calculateGratuity(38.40);
```

the function returns `7.68`. Not bad — a 20% tip!

Of course, you could store the output in order to use it elsewhere later, as in:

```
var tip = calculateGratuity(38.40);
console.log(tip);  //Prints 7.68 to the console
```

There are also *anonymous* functions which, unlike `calculateGratuity`, don't have names. Anonymous functions are used all the time with D3, but I'll introduce them later.

Comments

```
/* JavaScript supports CSS-style comments like this. */

// But double-slashes can be used as well.
// Anything following // on the same line will be ignored.
// This is helpful for including brief notes to yourself
// as to what each line of code does, as in:

console.log("Brilliant");  //Prints "Brilliant" to the console
```

Referencing Scripts

Scripts can be included directly in HTML, between two `script` tags

```
<body>
    <script type="text/javascript">
        alert("Hello, world!");
    </script>
</body>
```

or stored in a separate file with a *.js* suffix, and then referenced somewhere the HTML (typically in the `head`):

```
<head>
    <title>Page Title</title>
    <script type="text/javascript" src="myscript.js"></script>
</head>
```

JavaScript Gotchas

As a bonus, and at no extra charge, I would like to share with you my top four JavaScript Gotchas: things that, had I known earlier, would have saved me many hours of late-night debugging sessions, anxiety-induced panic, and increased cortisol levels. You may want to come back and reference this section later.

1. Dynamic Typing

I do love how your fingers flutter across the keyboard, but that's not what I'm talking about. No, JavaScript is a *loosely typed* language, meaning you don't have to specify what *type* of information will be stored in a variable in advance. Many other languages, like Java (which is completely different from Java'Script'!), require you to declare a variable's type, such as `int`, `float`, `boolean`, or `String`.

```
//Declaring variables in Java
int number = 5;
float value = 12.3467;
boolean active = true;
String text = "Crystal clear";
```

JavaScript, however, automatically *types* a variable based on what kind of information you assign to it. (Note that `''` or `""` indicate string values. I prefer double quotation marks `""`, but some people like singles `''`.)

```
//Declaring variables in JavaScript
var number = 5;
var value = 12.3467;
var active = true;
var text = "Crystal clear";
```

How boring — `var`, `var`, `var`, `var`! — yet handy, as we can declare and name variables before we even know what type of data will go into them. You can even change a variable's type on-the-fly without JavaScript freaking out on you.

```
var value = 100;
value = 99.9999;
value = false;
value = "This can't possibly work.";
value = "Argh, it does work! No errorzzzz!";
```

I mention this because one day a numeric value will be accidentally stored as a string, and it will cause some part of your code to behave strangely, and basically I don't want you to come crying to me when that happens. Thus, know that whenever a variable's type is in doubt, you can employ the `typeof` operator:

```
typeof 67;              //Returns "number"
var myName = "Scott";
typeof myName;          //Returns "string"
myName = true;
typeof myName;          //Returns "boolean"
```

2. Variable Hoisting

Contrary to what you would expect, JavaScript code is usually, but not always, executed in linear, top-to-bottom order. For example, in this code, when would you expect the variable `i` to be established?

```
var numLoops = 100;
for (var i = 0; i < numLoops; i++) {
    console.log(i);
}
```

In many other languages, `i` would be declared right when the `for` loop begins, as you'd expect. But thanks to a phenomenon known as *variable hoisting*, in JavaScript, variable declarations are *hoisted* up to the top of the function context in which they reside. So in our example, `i` is actually declared *before* the `for` loop even begins. The following is equivalent:

```
var numLoops = 100;
var i;
for (i = 0; i < numLoops; i++) {
    console.log(i);
}
```

This can be problematic when you have conflicting variable names, as a variable that you thought existed only later in the code is actually present right at the beginning.

3. Function-level Scope

In programming, the concept of *variable scope* helps us identify which variables are accessible in which contexts. Generally, it is a bad idea to have every value accessible from everywhere else, because you'd end up with so many conflicts and accidental value changes that you would just go crazy.

Many languages use *block-level scope*, in which variables exist only within the current "block" of code, usually indicated by curly braces. With block-level scope, our `i` would exist only within the context of the `for` loop, for example, so any attempts to read the value of `i` or change `i` outside of the loop would fail. This is nice because you could establish other variables from within your loop and know that they wouldn't conflict with variables that exist elsewhere.

In JavaScript, however, variables are scoped at the function level, meaning they are accessible anywhere within the *function* (not block) in which they reside.

This is primarily something to be aware of if you're used to other languages. Bottom line: You can keep values contained by wrapping them within functions.

4. Global Namespace

Speaking of variable conflicts, please do me a favor: Open any web page, activate the JavaScript console, and type `window`.

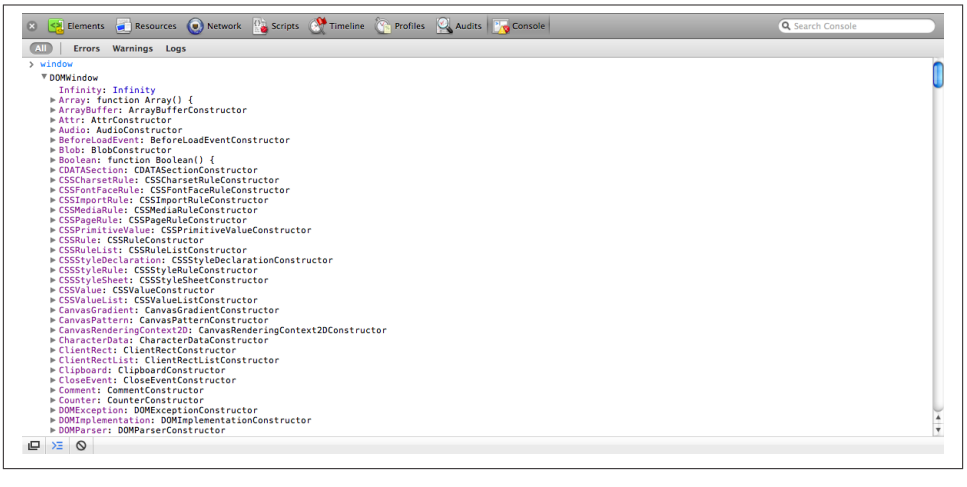


Figure 3-10. The global namespace

I know it feels like you just entered the matrix, but this is actually called "the global namespace," which might sound even cooler.

`window` is the topmost object in the browser's hierarchy of JavaScript elements, and all of these objects and values you see beneath `window` exist at the *global* level. What this means is that every time you declare a new variable, you are adding a new value to `window`. Or, as righteous JavaScript coders like to say, *you are polluting the global namespace*. (Surprisingly, most people who write things like this in online forums are actually quite friendly in person.)

For example, try typing this into the console: `var zebras = "amazing"`

Then type `window` again, and scroll all the way down to the bottom, where you should now see `zebras`:

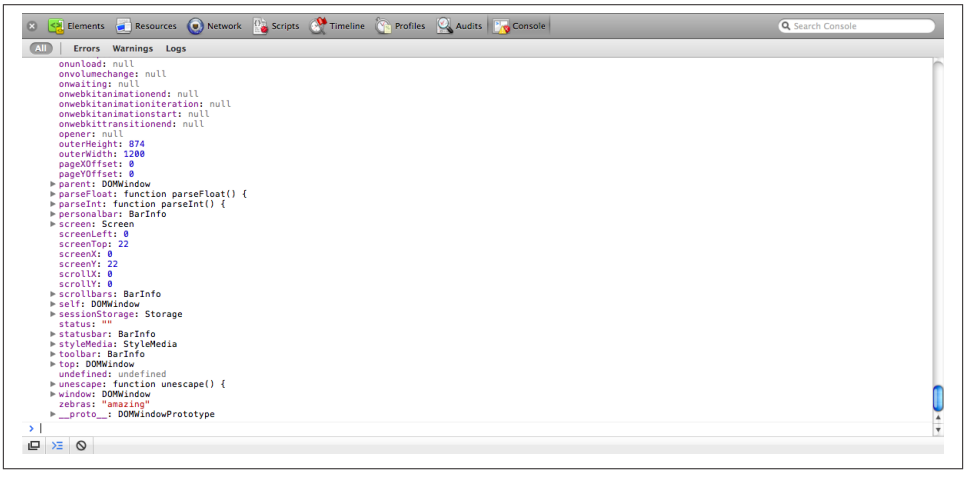


Figure 3-11. The global namespace, now with zebras

What's so wrong with adding values to window? As you get started, nothing at all. But as your projects grow in complexity, and especially if you begin to incorporate other non-D3 JavaScript code (such as jQuery, Facebook "Like" buttons, or Google Analytics tracking code), at some point you're bound to run into a conflict, because you're using the variable `zebras` for your project, but zebraTracker.js is *also* using a variable with the same name! The result: chaos. Or at least some bad data, which could lead to unexpected behavior or errors.

There are two easy workarounds (and, to clarify, you probably don't have to worry about this until later):

1. Declare variables only within other functions. This is not usually feasible, but due the function-level scope will prevent local variables from conflicting with others.
2. Declare a single global object, and attach all of your would-be global variables to that object. For example:

```
var Vis = {};  //Declare empty global object
Vis.zebras = "still pretty amazing";
Vis.monkeys = "too funny LOL";
Vis.fish = "you know, not bad";
```

All of your weird animal-related variables are no longer polluting the global namespace, but instead they all exist as values stored within your single, global object, `Vis`. Of course, `Vis` could be named whatever you like, but `Menagerie` is harder to type. Regardless, the only naming conflict you'll have with other scripts is if one of them *also* wants to have a global object named `Vis`, and what are the odds of that?

SVG

D3 is most useful when used to generate and manipulate visuals as Scalable Vector Graphics. Drawing with `div`s and other native HTML elements is possible, but a bit clunky and subject to the usual inconsistencies across different browsers. Using SVG is more reliable, visually consistent, and faster.

Here's a small circle and its SVG code:

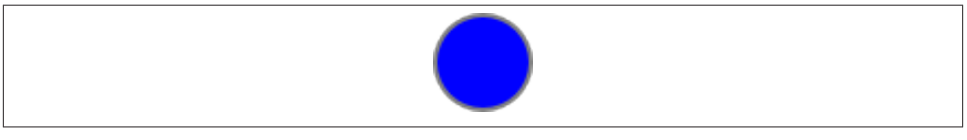


Figure 3-12. A small SVG circle

```
<svg width="50" height="50">
    <circle cx="25" cy="25" r="22"
     fill="blue" stroke="gray" stroke-width="2"/>
</svg>
```

Vector drawing software like Illustrator can be used to generate SVG files, but we need to learn how to generate them with code.

The SVG Element

SVG is a text-based image format. Each SVG image is defined using markup code similar to HTML, and SVG code can be included directly within any HTML document, or inserted dynamically into the DOM. Every web browser supports SVG *except* Internet Explorer versions 8 and older. SVG is XML-based, so you'll notice that elements that don't have a closing tag must be self-closing. For example:

```
<element></element> <!-- Uses closing tag -->
<element/>          <!-- Self-closing tag -->
```

Before you can draw anything, you must create an SVG element. Think of the SVG element as a canvas on which your visuals are rendered. (In that respect, SVG is con-

ceptually similar to HTML's `canvas` element.) At a minimum, it's good to specify `width` and `height` values. If you don't specify these, the SVG will behave like a typically greedy, block-level HTML element and take up as much room as it can within its enclosing element.

```
<svg width="500" height="50">
</svg>
```

Note that *pixels* are the default measurement units, so we can specify dimensions of `500` and `50`, not `500px` and `50px`. We could have specified `px` explicitly, or any number of other supported units, including `em`, `pt`, `in`, `cm`, and `mm`.

Simple Shapes

There are a number of visual elements that you can include between those `svg` tags, including `rect`, `circle`, `ellipse`, `line`, `text`, and `path`. (Sorry, `zebras` is not valid in this context.)

If you're familiar with computer graphics programming, you'll recognize the usual pixel-based coordinates system in which `0,0` is the top-left corner of the drawing space. Increasing `x` values move to the right, while increasing `y` values move down.

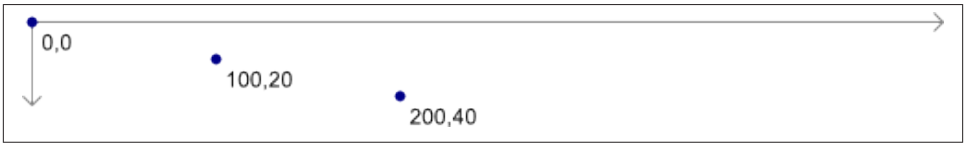


Figure 3-13. The SVG coordinates system

`rect` draws a rectangle. Use `x` and `y` to specify the coordinates of the upper-left corner, and `width` and `height` to specify the dimensions. This rectangle fills the entire space of our SVG:

```
<rect x="0" y="0" width="500" height="50"/>
```

Figure 3-14. An SVG rect

`circle` draws a circle. Use `cx` and `cy` to specify the coordinates of the *center*, and `r` to specify the radius. This circle is centered in the middle of our 500-pixel-wide SVG because its `cx` ("center-x") value is 250.

```
<circle cx="250" cy="25" r="50"/>
```

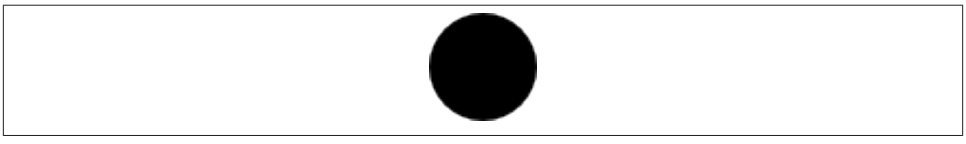


Figure 3-15. An SVG circle

`ellipse` is similar, but expects separate radius values for each axis. Instead of `r`, use `rx` and `ry`.

```
<ellipse cx="250" cy="25" rx="100" ry="25"/>
```

Figure 3-16. An SVG ellipse

`line` draws a line. Use `x1` and `y1` to specify the coordinates of one end of the line, and `x2` and `y2` to specify the coordinates of the other end. A `stroke` color must be specified for the line to be visible.

```
<line x1="0" y1="0" x2="500" y2="50" stroke="black"/>
```

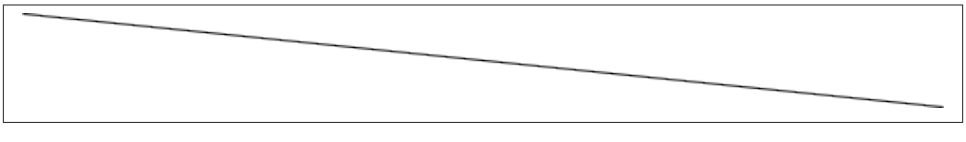


Figure 3-17. An SVG line

`text` renders text. Use `x` to specify the position of the left edge, and `y` to specify the vertical position of the type's *baseline*.

```
<text x="250" y="25">Easy-peasy</text>
```

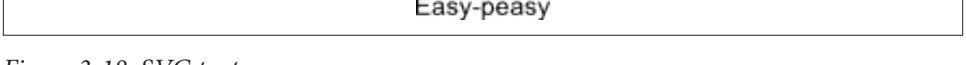


Figure 3-18. SVG text

`text` will inherit the CSS-specified font styles of its parent element unless specified otherwise. (More on styling text in a moment.) We could override that formatting as follows:

```
<text x="250" y="25" font-family="serif"
 font-size="25" fill="gray">Easy-peasy</text>
```

Figure 3-19. More SVG text

Also note that when any visual element runs up against the edge of the SVG, it will be clipped. Be careful when using `text` so your descenders don't get cut off (ouch!). You can see this happen when we set the baseline (`y`) to 50, the same as the height of our SVG:

```
<text x="250" y="50" font-family="serif"
 font-size="25" fill="gray">Easy-peasy</text>
```

Figure 3-20. More SVG text

`path` is for drawing anything more complex than the shapes above (like country outlines for geomaps), and will be explained separately. For now, we'll work with simple shapes.

Styling SVG Elements

SVG's default style is a black fill with no stroke. If you want anything else, you'll have to apply styles to your elements. Common SVG properties are:

- `fill` — A color value. Just as with CSS, colors can be specified as named colors, hex values, or RGB or RGBA values.
- `stroke` — A color value.
- `stroke-width` — A numeric measurement (typically in pixels).
- `opacity` — A numeric value between 0.0 (completely transparent) and 1.0 (completely opaque).

With `text`, you can also use these properties, which work just like in CSS:

- `font-family`

- `font-size`

In another parallel to CSS, there are two ways to apply styles to an SVG element: either directly (inline) as an attribute of the element, or with a CSS style rule.

Here are some style properties applied directly to a `circle` as attributes:

```
<circle cx="25" cy="25" r="22"
 fill="yellow" stroke="orange" stroke-width="5"/>
```

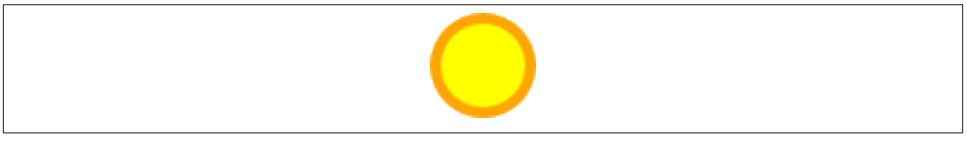


Figure 3-21. An SVG circle

Alternatively, we could strip the style attributes, assign the `circle` a class (just as if it were a normal HTML element)

```
<circle cx="25" cy="25" r="22" class="pumpkin"/>
```

and then put the `fill`, `stroke`, and `stroke-width` rules into a CSS style that targets the new class:

```
.pumpkin {
    fill: yellow;
    stroke: orange;
    stroke-width: 5;
 }
```

The CSS approach has a few obvious benefits:

1. You can specify a style once and have it be applied to multiple elements.
2. CSS code is easier to read than inline attributes.
3. For those reasons, the CSS approach may be more maintainable and make design changes faster to implement.

Using CSS to apply SVG styles, however, can be disconcerting for some. `fill`, `stroke`, and `stroke-width`, after all, are *not* CSS properties. (The nearest CSS equivalents are `background-color` and `border`.) If it helps you remember which rules in your stylesheet are SVG-specific, consider including `svg` in those selectors:

```
svg .pumpkin {
    /* ... */
 }
```

Layering and Drawing Order

There are no "layers" in SVG, and no real concept of depth. SVG does not support CSS's `z-index` property, so shapes can only be arranged within the two-dimensional x/y plane.

And yet, if we draw multiple shapes, they overlap:

```
<rect x="0" y="0" width="30" height="30" fill="purple"/>
<rect x="20" y="5" width="30" height="30" fill="blue"/>
<rect x="40" y="10" width="30" height="30" fill="green"/>
<rect x="60" y="15" width="30" height="30" fill="yellow"/>
<rect x="80" y="20" width="30" height="30" fill="red"/>
```

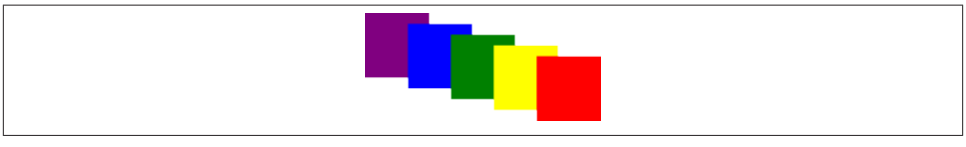


Figure 3-22. Overlapping SVG elements

The order in which elements are coded determines their depth order. The purple square appears first in the code, so it is rendered first. Then, the blue square is rendered "on top" of the purple one, then the green square on top of that, and so on.

Think of SVG shapes as being rendered like paint on a canvas. The pixel-paint that is applied later obscures any earlier paint, and thus appears to be "in front."

This aspect of drawing order becomes important when you have some visual elements that should not be obscured by others. For example, you may have axes or value labels that appear on a scatterplot. The axes and labels should be added to the SVG last, so they appear in front of any other elements.

Transparency

Transparency can be useful when elements in your visualization overlap but must remain visible, or you want to de-emphasize some elements while highlighting others.

There are two ways to apply transparency: use an RGB color with alpha, or set an `opacity` value.

You can use `rgba()` anywhere you specify a color, such as with `fill` or `stroke`. `rgba()` expects three values between 0 and 255 for red, green, and blue, plus an alpha (transparency) value between 0.0 and 1.0.

```
<circle cx="25" cy="25" r="20" fill="rgba(128, 0, 128, 1.0)"/>
<circle cx="50" cy="25" r="20" fill="rgba(0, 0, 255, 0.75)"/>
<circle cx="75" cy="25" r="20" fill="rgba(0, 255, 0, 0.5)"/>
<circle cx="100" cy="25" r="20" fill="rgba(255, 255, 0, 0.25)"/>
<circle cx="125" cy="25" r="20" fill="rgba(255, 0, 0, 0.1)"/>
```

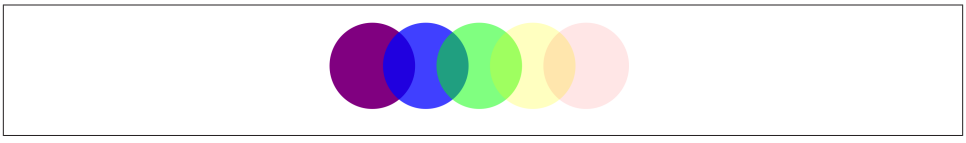


Figure 3-23. RGBA SVG shapes

Note that with `rgba()`, transparency is applied to the `fill` and `stroke` colors independently. The following circles' `fill` is 75% opaque, while their `stroke`s are only 25% opaque.

```
<circle cx="25" cy="25" r="20"
        fill="rgba(128, 0, 128, 0.75)"
        stroke="rgba(0, 255, 0, 0.25)" stroke-width="10"/>
<circle cx="75" cy="25" r="20"
        fill="rgba(0, 255, 0, 0.75)"
        stroke="rgba(0, 0, 255, 0.25)" stroke-width="10"/>
<circle cx="125" cy="25" r="20"
        fill="rgba(255, 255, 0, 0.75)"
        stroke="rgba(255, 0, 0, 0.25)" stroke-width="10"/>
```

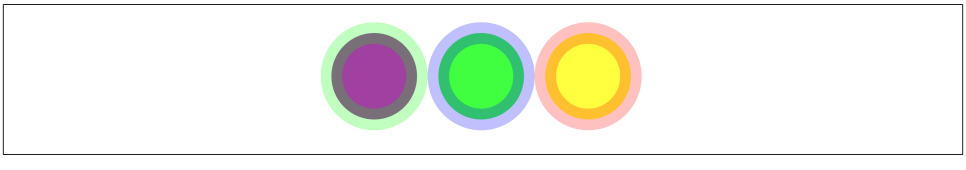


Figure 3-24. More RGBA SVG shapes

To apply transparency to an entire element, set an `opacity` attribute. Here are some completely opaque circles

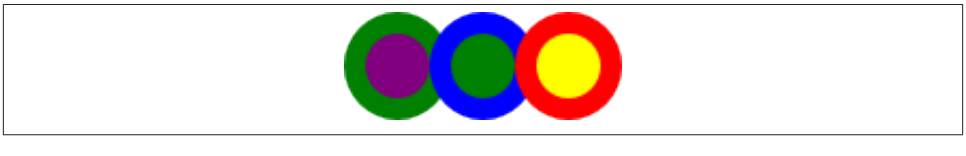


Figure 3-25. Opaque circles

followed by the same circles, with `opacity` values:

```
<circle cx="25" cy="25" r="20" fill="purple"
        stroke="green" stroke-width="10"
        opacity="0.9"/>
<circle cx="65" cy="25" r="20" fill="green"
        stroke="blue" stroke-width="10"
```

```
        opacity="0.5"/>
<circle cx="105" cy="25" r="20" fill="yellow"
        stroke="red" stroke-width="10"
        opacity="0.1"/>
```

Figure 3-26. Semi-opaque circles

You can employ `opacity` on an element that also has colors set with `rgba()`. When doing so, the transparencies are multiplied. The following circles use the same RGBA values for `fill` and `stroke`. The first circle below has no element `opacity` set, but the other two do:

```
<circle cx="25" cy="25" r="20"
        fill="rgba(128, 0, 128, 0.75)"
        stroke="rgba(0, 255, 0, 0.25)" stroke-width="10"/>
<circle cx="65" cy="25" r="20"
        fill="rgba(128, 0, 128, 0.75)"
        stroke="rgba(0, 255, 0, 0.25)" stroke-width="10"
        opacity="0.5"/>
<circle cx="105" cy="25" r="20"
        fill="rgba(128, 0, 128, 0.75)"
        stroke="rgba(0, 255, 0, 0.25)" stroke-width="10"
        opacity="0.2"/>
```

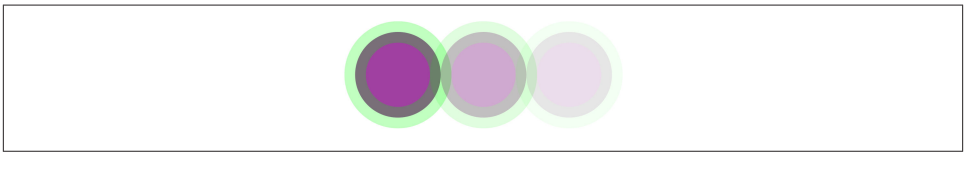


Figure 3-27. More opaque circles

Notice how the third circle's `opacity` is `0.2` or 20%. Yet its purple fill already has an alpha value of `0.75` or 75%. The purple area, then, has a final transparency of 0.2 times 0.75 = 0.15 or 15%.

A Note on Compatibility

Older browsers don't support SVG. So, generally speaking, Internet Explorer version 8 and older will not display SVG images at all. This is a little annoying, because people may visit your page only to see emptiness in place of a gorgeous visualization. On the other hand, D3 also does not work with older browsers (again, meaning IE8 and older), so it never would have worked in the first place.

In general, your best bet is to encourage people to upgrade and use a web browser that has been built in the last few years. The more people move to modern browsers, the better their web experiences will be, and the larger our potential audiences.

That said, perhaps your project is required to work in older browsers, or at least notify them of why the piece isn't working. I recommend using Modernizr or a similar JavaScript tool to detect whether or not the browser supports SVG. If it does, then you can load your D3 code and proceed as normal. If SVG is *not* supported, then you can display a static, non-interactive version of your visualization alongside a message explaining that a current browser is needed. (Be nice and provide links to the Chrome, Firefox, and Safari download pages.)

caniuse.com is a fantastic resource for supported browser features. See their list of browsers with SVG support.

People have tried to get D3 working in older browsers, sometimes by mashing it up with Raphael to render to canvas, or other hacky means. So it may be possible, but I don't recommend it.

CHAPTER 4
Setup

Getting set up with D3 is pretty straightforward — a simple matter of downloading the latest version, creating an empty page in which to write your code, and finally setting up a local web server.

Downloading D3

Start by creating a new folder for your project. Call it whatever you like, but maybe something like `project-folder`.

Within that folder, I recommend creating a sub-folder called `d3`. Then download the latest version of D3 into that sub-folder. You can do that by visiting d3js.org, right-clicking the link for `d3.v2.js`, and choosing "Save linked file as…" or something similar. As of this writing (Fall 2012), the current version of D3 is 2.10.0.

D3 is also provided in a "minified" version, `d3.v2.min.js`, from which whitespace has been removed for smaller file sizes and faster load times. The functionality is the same, but typically you'd use the regular version while working on a project (for friendlier debugging), and then switch to the minified version once you've launched the project publicly (for optimized load times). The choice is up to you, but in this book, I'll use the standard version.

A third and final option is to download the entire D3 repository, which gives you not just the JavaScript files, but also all of the component source code and a massive folder with tons of examples. You can browse the repository contents first, or just download the whole thing as a compressed ZIP file. Of course, once you've downloaded everything, make a copy of the file `d3.v2.js` and move that into `project-folder/d3/`.

Referencing D3

Now create a simple HTML page within your project folder named `index.html`. Remember, HTML documents are just plain text files, so you can use the text editor of your choice. Free editors like TextEdit and Notepad are fine, but your life may be easier if you use an editor designed specifically for working with code, like Coda, Espresso, or Sublime Text (among many, many others).

If your editor gives you the option to set the file encoding, choose Unicode (UTF-8).

Your folder structure should now look something like this:

```
project-folder/
    d3/
        d3.v2.js
        d3.v2.min.js (optional)
    index.html
```

Paste in the following into your new HTML file:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>D3 Test</title>
        <script type="text/javascript" src="d3/d3.v2.js"></script>
    </head>
    <body>
        <script type="text/javascript">
            // Your beautiful D3 code will go here
        </script>
    </body>
</html>
```

Things to note:

1. The `meta` tag identifies the encoding for this file as `utf-8`, which is needed to ensure that the browser can parse D3's functions and data properly.
2. The first `script` tag sets the reference to `d3.v3.js`. You should edit this file path as needed if you're using the minified version of D3 or decided to locate `d3.v2.js` somewhere other than the `d3` directory.
3. The second `script` tag, in the `body`, is where you will soon key in all your beautiful code. And it will be beautiful.

Done! Your D3 template files and folders are all set up. You may want to make a copy of this template for each new project down the line.

Setting up a Web Server

In some cases, you can view local HTML files directly in your web browser. However, some browsers have security restrictions that prevent them from loading local files via JavaScript. Meaning, if your D3 code is trying to pull in any external data files (like CSVs or JSON), it will fail with no good explanation. This isn't D3's fault; it's just a browser quirk, and I don't mean quirky in a cute kind of way.

For this reason, it is much more reliable to load your page via a web server. While you *could* use a remote web server, it is much, much faster to store and host everything locally (meaning, on the same computer, the one right in front of you). It is a strange idea, to use your local computer to host and serve files to itself, but you can think about it as the different programs talking to each other: The browser program requests files from the server program, which responds by serving them back.

The good news is that it's quite easy to get a local server up and running. Here are a couple of ways to do that.

1. MAMP, WAMP, & LAMP

This option takes longer, but is best if you like dragging and dropping to install things, and want to avoid scary things like the terminal.

All of the AMPs in this section stand for Apache (the web server software), MySQL (popular database software), and PHP (a popular web scripting language). We are really interested only in Apache, the web server, but it usually comes bundled with the other two, since they all work well together.

On a Mac, you can download and install MAMP or XAMPP for Mac.

The Windows equivalents are WampServer and XAMPP for Windows.

If you use Linux, then all of this is probably already installed on your machine, but you could still download XAMPP for Linux.

Installation for each of these packages varies somewhat, so follow the documentation carefully. (I've found MAMP to be the easiest to install.)

Each package will designate one folder as the web server directory, so only the files *within that folder* will be served. You should find out what that folder is, and move your D3 `project-folder` into it.

Once the local server is up and running, you can view any pages within the server directory by opening a browser (on the same computer, of course) and pointing it to `localhost`, as in:

```
http://localhost/
```

Yes, instead of `www.something.com`, you just use `localhost`, which tells the browser to request a page from *this machine*. Depending on your AMP configuration, you may need to append a port number to the URL, as in:

```
http://localhost:8888/
```

If the server's port number is `8888` and your project folder is called `project-folder`, then you could view your D3 template page by going to:

```
http://localhost:8888/project-folder/
```

2. Terminal + Python

If you're comfortable entering commands in the terminal and your machine has Python installed, then running a mini-Python server is a super quick option. If you're reading this section, I'm going to assume you know some basic terminal commands for your platform (as it's slightly different for Windows and Mac OS X or Linux).

To run a Python web server:

1. Open up a terminal window.
2. Via command line, navigate into the directory that you want served, e.g. `project-folder`.
3. Enter `python -m SimpleHTTPServer 8888 &`

This will activate the server on port `8888`. Switch back to your browser and visit:

```
http://localhost:8888/
```

You should see the blank "D3 Test" page. Given that `body` of the page is empty, it won't look like much. Select "View source" to confirm it's loading properly.

Diving In

All set? Great! Let's start working with data.

CHAPTER 5
Data

Data is an extremely broad term, only slightly less vague than the nearly all-encompassing *information*. What is data? (What *isn't* data?) What kinds of data are there, and what can we use with D3?

Broadly speaking, data is structured information with potential for meaning.

In the context of programming for visualization, data is stored in a digital file, typically in either text or binary form. Of course, potentially every piece of digital emphemera may be considered "data" — not just text, but bits and bytes representing images, audio, video, databases, streams, models, archives, and anything else.

Within the scope of D3 and browser-based visualization, however, we will limit ourselves to *text-based data*. That is, anything that can be represented as numbers and strings of alpha characters. If you can get your data into a `.txt` plain text file, a `.csv` comma-separated value file, or a `.json` JSON document, then you can use it with D3.

Whatever your data, it can't be made useful and visual until it is *attached* to something. In D3 lingo, the data must be *bound* to elements within the page. Let's address how to create new page elements first. Then attaching data to those elements will be a cinch.

Generating Page Elements

Typically, when using D3 to generate new DOM elements, the new elements will be circles, rectangles, or other visual forms that represent your data. But to avoid confusing matters, we'll start with a simple example and create a lowly `p` paragraph element.

Begin by creating a new document with our simple HTML template from the last chapter:

```
<!DOCTYPE html>
<html lang="en">
```

```
    <head>
        <meta charset="utf-8">
        <title>D3 Test</title>
        <script type="text/javascript" src="d3/d3.v2.js"></script>
    </head>
    <body>
        <script type="text/javascript">
            // Your beautiful D3 code will go here
        </script>
    </body>
</html>
```

Open that page up in your web browser, and pop open the web inspector. You should see something like:

Figure 5-1. Web inspector

Back in your text editor, replace the comment between the `script` tags with:

```
d3.select("body").append("p").text("New paragraph!");
```

Save and refresh, and voilà! There is text in the formerly empty browser window, and the following in the web inspector:

Figure 5-2. Web inspector, again

See the difference? Now in the DOM, there is a new paragraph element that was generated on-the-fly! This may not be exciting yet, but you will soon use a similar technique to dynamically generate tens or hundreds of elements, each one corresponding to a piece of your data set.

Let's walk through what just just happened. To understand that first line of D3 code, you must first meet your new best friend, *chain syntax*.

Chaining Methods

D3 smartly employs a technique called *chain syntax*, which you may recognize from jQuery. By "chaining" methods together with periods, you can perform several actions in a single line of code. It can be fast and easy, but it's important to understand how it works, to save yourself hours of debugging headaches later.

By the way, *functions* and *methods* are just two different words for the same concept: a chunk of code that accepts an argument as input, performs some action, and returns some other information as output.

This code

```
d3.select("body").append("p").text("New paragraph!");
```

may look like a big mess, especially if you're new to programming. So the first thing to know is that JavaScript, like HTML, doesn't care about whitespace and line breaks, so you can put each method on its own line for legibility:

```
d3.select("body")
    .append("p")
    .text("New paragraph!");
```

Both I and your optometrist highly recommend putting each method on its own indented line. But everyone has their own coding style; use whatever indents, line breaks, and whitespace (tabs or spaces) are most legible for you.

One Link at a Time

Let's deconstruct each line in this chain of code.

`d3` — References the D3 object, so we can access its methods. Our D3 adventure begins here.

`.select("body")` — Give the `select()` method a CSS selector as input, and it will return a reference to the first element in the DOM that matches. (Use `selectAll()` when you need more than one element.) In this case, we just want the `body` of the document, so a reference to `body` is handed off to the next method in our chain.

`.append("p")` — `append()` creates whatever new DOM element you specify and appends it to the end (but *just inside*) of whatever selection it's acting on. In our case, we

want to create a new `p` within the `body`. We specified `"p"` as the input argument, but this method also sees the reference to `body` that was passed down the chain from the `select()` method. So an empty `p` paragraph is *appended* to the `body`. Finally, `append()` hands off a reference to the new element it just created.

`.text("New paragraph!")` — `text()` takes a string and inserts it between the opening and closing tags of the current selection. Since the previous method passed down a reference to our new `p`, this code just inserts the new text between `<p>` and `</p>`. (In cases where there is existing content, it will be overwritten.)

`;` — The all-important semicolon indicates the end of this line of code. Chain over.

The Hand-off

Many, but not all, D3 methods return a selection (or, really, reference to a selection), which enables this handy technique of method chaining. Typically, a method returns a reference to the element that it just acted upon, but not always.

So remember: When chaining methods, order matters. The output type of one method has to match the input type expected by the next method in the chain. If adjacent inputs and outputs are mismatched, the hand-off will function more like a dropped baton in a middle-school relay race.

When sussing out what each function expects and returns, the API reference is your friend. It contains detailed information on each method, including whether or not it returns a selection.

Going Chainless

Our sample code could be rewritten without chain syntax as:

```
var body = d3.select("body");
var p = body.append("p");
p.text("New paragraph!");
```

Ugh! What a mess. Yet there will be times you need to break the chain, such as when you are calling so many functions that it doesn't make sense to string them all together. Or just because you want to organize your code in a way that makes more sense to you.

Now that you know how to generate new page elements with D3, it's time to attach data to them.

Binding data

What is binding, and why would I want to do it to my data?

Data visualization is a process of *mapping* data to visuals. Data in, visual properties out. Maybe bigger numbers make taller bars, or special categories trigger brighter colors. The mapping rules are up to you.

With D3, we *bind* our data input values to elements in the DOM. Binding is like "attaching" or associating data to specific elements, so that later you can reference those values to apply mapping rules. Without the binding step, we have a bunch of data-less, un-mappable DOM elements. No one wants that.

In a Bind

We use D3's `selection.data()` method to bind data to DOM elements. But there are two things we need in place first, before we can bind data:

1. The data
2. A selection of DOM elements

Let's tackle these one at a time.

Data

D3 is smart about handling different kinds of data, so it will accept practically any array of numbers, strings, or objects (themselves containing other arrays or key/value pairs). It can handle JSON (and GeoJSON) gracefully, and even has a built-in method to help you load in CSV files.

But to keep things simple, for now we will start with a boring array of five numbers. Here is our sample data set:

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

If you're feeling adventurous, or already have some data in CSV or JSON format that you want to play with, here's how to do that. Otherwise, just skip ahead to "Please Make Your Selection."

Loading CSV Data

CSV stands for comma-separated values. A CSV data file might look something like this:

```
Food,Deliciousness
Apples,9
Green Beans,5
Egg Salad Sandwich,4
Cookies,10
Vegemite,0.2
Burrito,7
```

Each line in the file has the same number of values (two, in this case), and values are separated by a comma. The first line in the file often serves as a header, providing names for each of the "columns" of data.

If we saved the CSV data above into a file called `food.csv`, then we could load the file into D3 by using the `d3.csv()` method:

```
d3.csv("food.csv", function(data) {
    console.log(data);
});
```

`csv()` takes two arguments: a string representing the path of the CSV file to load in, and an anonymous function, to be used as a *callback function*. The callback function is "called" only *after* the CSV file has been loaded and parsed successfully. So if `food.csv` can't be found, or there is some other error, it will never be called.

When called, the anonymous function is handed the result of the CSV loading and parsing process; that is, the data. Here I'm naming it `data`, but this could be called whatever you like. You should use this callback function to do all the things you can only do *after* the data has been loaded. In the example above, we are just logging the value of the `data` array to the console, to verify it. (See `01_csv_loading_example.html` in the example code.)

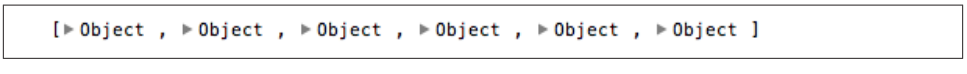


Figure 5-3. Array logged to console

You can see that `data` is an array (because of the hard brackets `[]` on either end) with six elements, each of which is an object. By toggling the disclosure triangles next to each object, we can see their values.

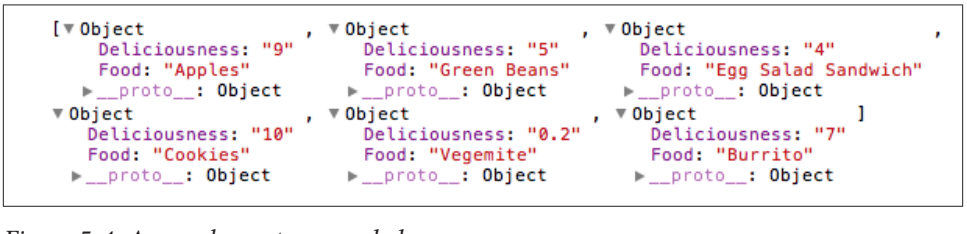


Figure 5-4. Array elements expanded

Aha! Each object has both a `Food` property and a `Deliciousness` property, the values of which correspond to the values in our CSV! (There is also a third property, `__proto__`, but that has to do with how JavaScript handles objects, and you can ignore it for now.) D3 has employed the first row of the CSV for property names, and subsequent rows for values. You may not realize it, but this just saved you a *lot* of time.

One more thing to note is that each value from the CSV is stored as a string, even the numbers. (You can tell because 9 is surrounded by quotation marks, as in `"9"` and not simply `9`.) This may cause unexpected behavior later, if you try to reference your data as a numeric value, but it is still typed as a string.

Verifying your data is a great use of the `csv()` callback function, but typically this is where you'd call other functions that construct the visualization, now that the data is available, as in:

```
var dataset;  //Declare global var

d3.csv("food.csv", function(data) {

    //Hand CSV data off to global var,
    //so it's accessible later.
    dataset = data;

    //Call some other functions that
    //generate your visualization, e.g.:
    generateVisualization();
    makeAwesomeCharts();
    makeEvenAwesomerCharts();
    thankAwardsCommittee();

});
```

One more tip: If you have *tab*-separated data in a TSV file, try the `d3.tsv()` method, which otherwise behaves exactly as above.

Loading JSON Data

We'll spend more time talking about JSON later, but for now all you need to know is that the `d3.json()` method works the same way as `csv()`. Load your JSON data in this way:

```
d3.json("waterfallVelocities.json", function(json) {
    console.log(json);  //Log output to consle
});
```

Here I've named the parsed output `json`, but it could be called `data` or whatever you like.

Please Make Your Selection

The data is locked and loaded. As a reminder, we are working with this array:

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

Now you need to decide what to select. That is, what elements will your data be associated with? Again, let's keep it super simple and say that we want to make a new paragraph for each value in the data set. So you might imagine something like this would be helpful

```
d3.select("body").selectAll("p")
```

and you'd be right, but there's a catch: The paragraphs we want to select *don't exist yet*. And this gets at one of the most common points of confusion with D3: How can we select elements that don't yet exist? Bear with me, as the answer may require bending your mind a bit.

The answer lies with `enter()`, a truly magical method. See this code, which I'll explain:

```
d3.select("body").selectAll("p")
    .data(dataset)
    .enter()
    .append("p")
    .text("New paragraph!");
```

View the example code `02_creating_paragraphs.html` and you should see five new paragraphs, each with the same content, like so:

Figure 5-5. Dynamic paragraphs

Here's what's happening.

`d3.select("body")` — Finds the `body` in the DOM and hands off a reference to the next step in the chain.

`.selectAll("p")` — Selects all paragraphs in the DOM. Since none exist yet, this returns an empty selection. Think of this empty selection as representing the paragraphs that *will soon exist*.

`.data(dataset)` — Counts and parses our data values. There are five values in our array called `dataset`, so everything past this point is executed five times, once for each value.

`.enter()` — To create new, data-bound elements, you must use `enter()`. This method looks at the current DOM selection, and then at the data being handed to it. If there are more data values than corresponding DOM elements, then `enter()` *creates a new placeholder element* on which you may work your magic. It then hands off a reference to this new placeholder to the next step in the chain.

`.append("p")` — Takes the empty placeholder selection created by `enter()` and appends a `p` element into the DOM. Hooray! Then it hands off a reference to the element it just created to the next step in the chain.

`.text("New paragraph!")` — Takes the reference to the newly created `p` and inserts a text value.

Bound and Determined

All right! Our data has been read, parsed, and bound to new `p` elements that we created in the DOM. Don't believe me? Take another look at `02_creating_paragraphs.html` and whip out your web inspector.

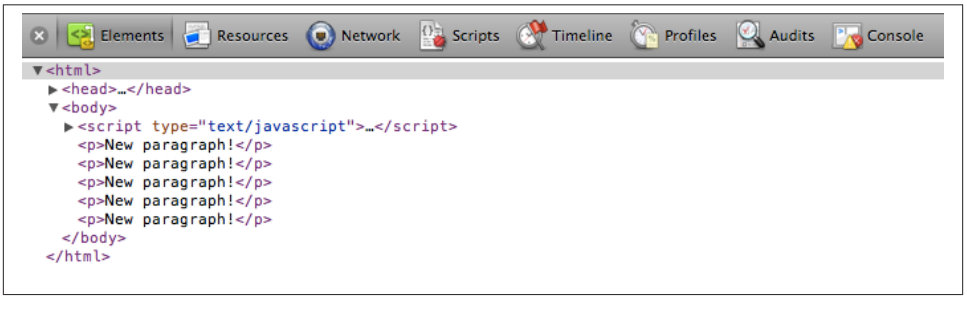


Figure 5-6. New `p` elements in the web inspector

Okay, I see five paragraphs, but where's the data? Switch to the JavaScript console, type in the following code, and hit enter:

```
console.log(d3.selectAll("p"))
```

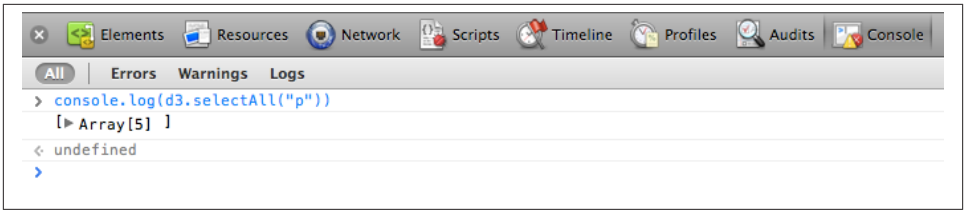


Figure 5-7. Logging to the console, from the console

An array! Or, really, an array containing another array. Click the gray disclosure triangle to reveal its contents:

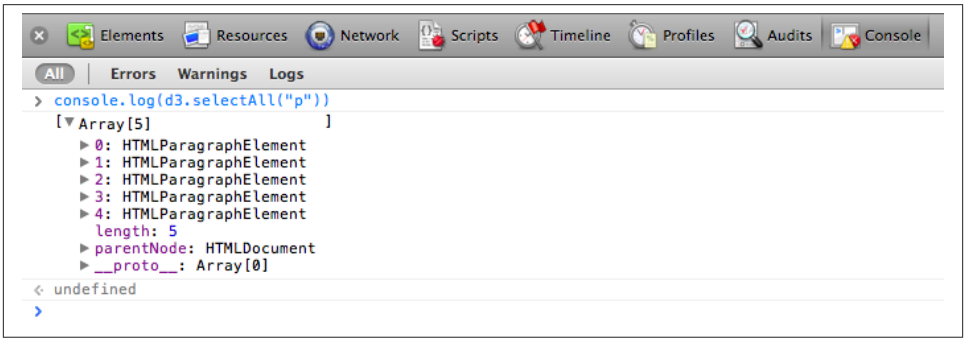


Figure 5-8. Array of arrays, expanded

You'll notice the five `HTMLParagraphElement`s, numbered 0 through 4. Click the disclosure triangle next to the first one (number zero).

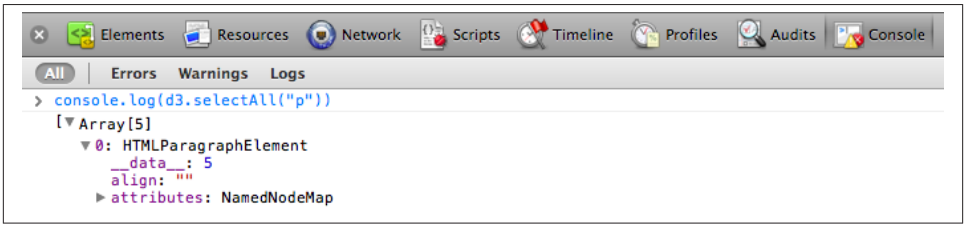


Figure 5-9. The p element, expanded

See it? Do you see it? I can barely contain myself. There it is:

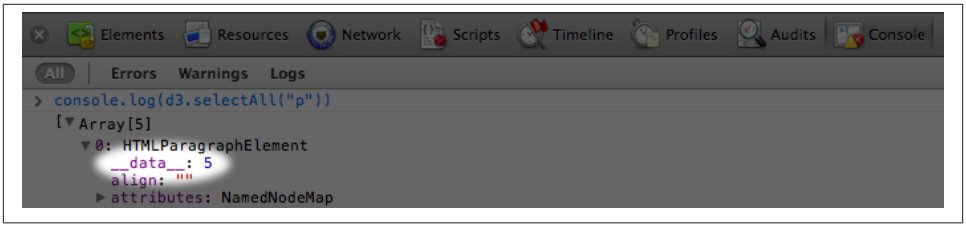


Figure 5-10. Finally, bound data

Our first data value, the number `5`, is showing up under the first paragraph's `__data__` attribute. Click into the other paragraph elements, and you'll see they also contain `__data__` values: 10, 15, 20, and 25, just as we specified.

You see, when D3 binds data to an element, that data doesn't exist in the DOM, but it does exist in memory as a `__data__` attribute of that element. And the console is where you can go to confirm whether or not your data was bound as expected.

The data is ready. Let's do something with it.

Using your data

We can see that the data has been loaded into the page and is bound to our newly created elements in the DOM, but can we *use* it? Here's our code so far:

```
var dataset = [ 5, 10, 15, 20, 25 ];

d3.select("body").selectAll("p")
    .data(dataset)
    .enter()
    .append("p")
    .text("New paragraph!");
```

Let's change the last line to:

```
    .text(function(d) { return d; });
```

Now test out that new code in `03_creating_paragraphs_text.html`. You should see:

Figure 5-11. More dynamic paragraphs

Whoa! We used our data to populate the contents of each paragraph, all thanks to the magic of the `data()` method. You see, when chaining methods together, anytime after you call `data()`, you can create an anonymous function that accepts `d` as input. The magical `data()` method ensures that `d` is set to the corresponding value in your original data set, given the current element at hand.

The value of "the current element" changes over time as D3 loops through each element. For example, when looping through the third time, our code creates the third `p` tag, and `d` will correspond to the third value in our data set (or `dataset[2]`). So the third paragraph gets text content of "15".

High-functioning

In case you're new to writing your own functions (a.k.a. methods), the basic structure of a function definition is:

```
function(input_value) {
    //Calculate something here
    return output_value;
}
```

The function we used above is dead simple, nothing fancy

```
function(d) {
    return d;
}
```

and it's wrapped within D3's `text()` function, so whatever our function returns is handed off to `text()`.

```
.text(function(d) {
    return d;
});
```

Our anonymous function is executed first. Then its result is handed off to `text()`. Then `text()` finally works its magic (by inserting its input argument as text within the selected DOM element).

But we can (and will) get much fancier, because you can customize these functions however you want. Yes, this is both the pleasure and pain of writing your own JavaScript. We can define our own custom functions however we want. Maybe you'd like to add some extra text, as in

```
.text(function(d) {
    return "I can count up to " + d;
});
```

which produces this result, as seen in example file `04_creating_paragraphs_counting.html`:

Figure 5-12. Still more dynamic paragraphs

Data Wants to be Held

You may be wondering why you have to write out `function(d) { … }` instead of just `d` on its own. For example, this won't work:

```
.text("I can count up to " + d);
```

In this context, without wrapping `d` in an anonymous function, `d` has no value. Think of `d` as a lonely little placeholder value that just needs a warm, containing hug from a kind, caring function's parantheses. (Extending this metaphor further, yes, it is creepy that the hug is being given by an *anonymous* function — stranger danger! — but that only confuses matters.)

Here is `d` being gently and appropriately held by a function:

```
.text(function(d) {  // <-- Note tender embrace at left
    return "I can count up to " + d;
});
```

The reason for this syntax is that `.text()`, `attr()`, and many other D3 methods take a *function* as an argument. For example, `text()` can take either simply a static string of text as an argument *or* a function. Both of these are valid ways to use `text()`:

```
.text("someString")
.text(someFunction())
```

When you write

```
.text(function(d) {
    return d;
})
```

you are defining an anonymous function. If D3 sees a function there, it will *call* that function, while handing off the current datum `d` as the function's argument. Without the function in place, D3 can't know to whom it should hand off the argument `d`. When no one is there to receive the value of `d`, D3 may get confused and even start crying. (D3 is more emotional than you'd expect.)

At first, this may seem silly and like a lot of extra work to just get at `d`, but the value of this approach will become clear as we work on more complex pieces.

Beyond Text

Things get a lot more interesting when we explore D3's other methods, like `attr()` and `style()`, which allow us to set HTML attributes and CSS properties on selections, respectively.

For example, adding one more line to our code

```
.style("color", "red");
```

produces this result, as seen in `05_creating_paragraphs_with_style.html`:

Figure 5-13. Red paragraphs

All the text is now red; big deal. But we could use a custom function to make the text red only if the current datum exceeds a certain threshold. So we revise that last line to use a function instead of a string:

```
.style("color", function(d) {
    if (d > 15) {   //Threshold of 15
        return "red";
    } else {
        return "black";
    }
});
```

See the resulting change in `06_creating_paragraphs_with_style_functions.html`:

Figure 5-14. Dynamically styled paragraphs

Notice how the first three lines are black, but once `d` exceeds the arbitrary threshold of 15, the text turns red.

Okay, we've got data loaded in and dynamically DOM elements bound to that data. I'd say we're ready to start drawing with data!

CHAPTER 6

Drawing with Data

It's time to start drawing with data.

Let's continue working with our simple data set for now:

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

Drawing divs

We'll use this to generate a super-simple bar chart. Bar charts are essentially just rectangles, and an HTML `<div>` is the easiest way to draw a rectangle. (Then again, to a web browser, *everything* is a rectangle, so you could easily adapt this example to use `span`s or whatever element you prefer.)

Formally, a chart with vertically oriented rectangles is a *column* chart, and one with horizontal rectangles is a *bar* chart. In practice, most people just call them all bar charts, as I'll do from now on.

This `div` could work well as a data bar:

Figure 6-1. A humble `div`

```
<div style="display: inline-block;
            width: 20px;
            height: 75px;
            background-color: teal;"></div>
```

(Among web standards folks, this is a semantic no-no. Normally, one shouldn't use an empty `div` for purely visual effect, but I am making an exception for the sake of this example.)

Because this is a `div`, its `width` and `height` are set with CSS styles. Except for `height`, each bar in our chart will share the same display properties, so I'll put those shared styles into a class called `bar`:

```
div.bar {
    display: inline-block;
    width: 20px;
    height: 75px;   /* We'll override height later */
    background-color: teal;
}
```

Now each `div` needs to be assigned the `bar` class, so our new CSS rule will apply. If you were writing the HTML code by hand, you would write:

```
<div class="bar"></div>
```

Using D3, to add a class to an element, we use the `selection.attr()` method. It's important to understand the difference between `attr()` and its close cousin, `style()`. `attr()` sets DOM attribute values, while `style()` applies CSS styles directly to an element.

Setting Attributes

`attr()` is used to set an HTML attribute and its value on an element. An HTML attribute is any property/value pair that you could include between an element's <> brackets. For example, these HTML elements

```
<p class="caption">
<select id="country">
<img src="logo.png" width="100px" alt="Logo" />
```

contain a total of five attributes (and corresponding values), all of which could be set with `attr()`:

```
Attribute   |   Value
------------|------------
class       |   caption
id          |   country
src         |   logo.png
width       |   100px
alt         |   Logo
```

To assign a class of `bar`, we can use:

```
.attr("class", "bar")
```

A Note on Classes

Note that an element's *class* is stored as an HTML attribute. The class, in turn, is used to reference a CSS style rule. This may cause some confusion because there is a difference between setting a *class* (from which styles are inferred) and applying a *style* directly to an element. You can do both with D3. Although you should use whatever approach makes the most sense to you, I recommend using *classes* for properties that are shared by multiple elements, and applying *style* rules directly only when deviating from the norm. (In fact, that's what we'll do in just a moment.)

I also want to briefly mention another D3 method, `classed()`, which can be used to quickly apply or remove classes from elements. The line of code above could be rewritten as:

```
.classed("bar", true)
```

This line simply takes whatever selection is passed to it and applies the class `bar`. If `false` were specified, it would do the opposite, removing the class of `bar` from any elements in the selection:

```
.classed("bar", false)
```

Back to the Bars

Putting it all together with our data set, here is the complete D3 code so far:

```
var dataset = [ 5, 10, 15, 20, 25 ];

d3.select("body").selectAll("div")
    .data(dataset)
    .enter()
    .append("div")
    .attr("class", "bar");
```

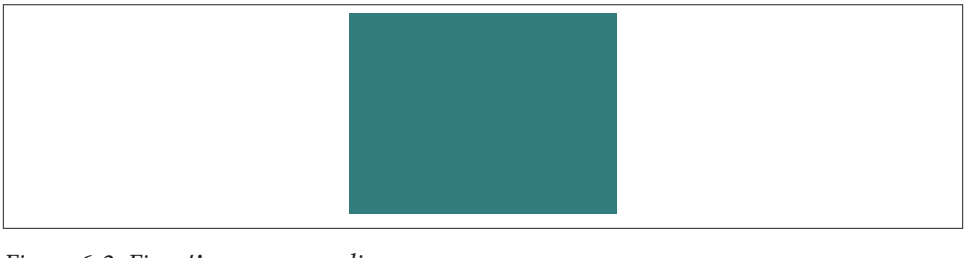


Figure 6-2. Five `div`s masquerading as one

To see what's going on, look at `01_drawing_divs.html` in your browser, view the source, and open your web inspector. You should see five vertical `div` bars, one generated for each point in our data set. However, with no space between them, they look like one big rectangle.

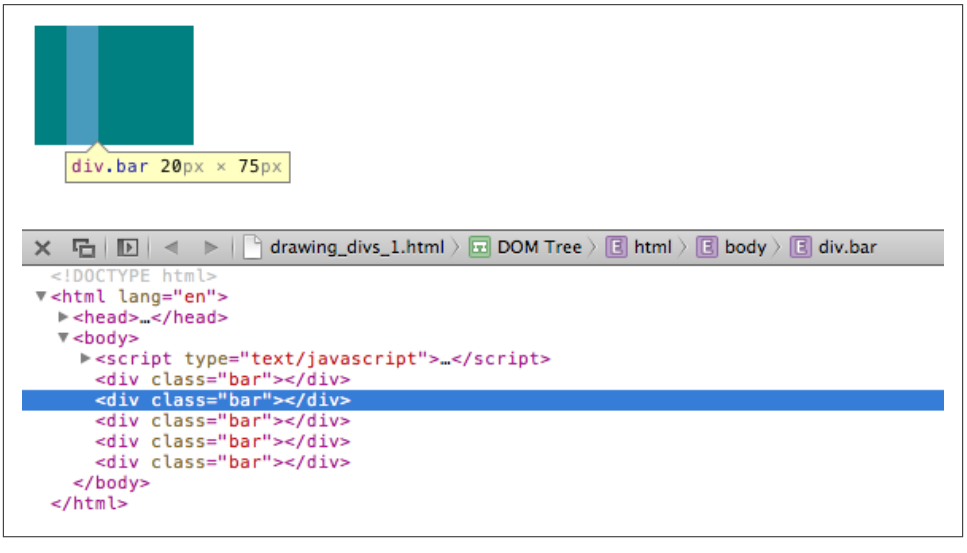


Figure 6-3. Five `div`s masquerading as one, as seen through the web inspector

Setting Styles

The `style()` method is used to apply a CSS property and value directly to an HTML element. This is the equivalent of including CSS rules within a `style` attribute right in your HTML, as in:

```
<div style="height: 75px;"></div>
```

To make a bar chart, the height of each bar must be a function of its corresponding data value. So let's add this to the end of our D3 code:

```
.style("height", function(d) {
    return d + "px";
});
```

Figure 6-4. A small bar chart!

See that code in `02_drawing_divs_height.html`. You should see a very small bar chart!

When D3 loops through each data point, the value of `d` will be set to that of the corresponding value. So we are setting a `height` value of `d` (the current data value) while appending the text `px` (to specify the units are pixels). The resulting heights are `5px`, `10px`, `15px`, `20px`, and `25px`.

This looks a little bit silly, so let's make those bars taller

```
.style("height", function(d) {
    var barHeight = d * 5;  //Scale up by factor of 5
    return barHeight + "px";
});
```

and add some space to the right of each bar, to space things out:

```
margin-right: 2px;
```

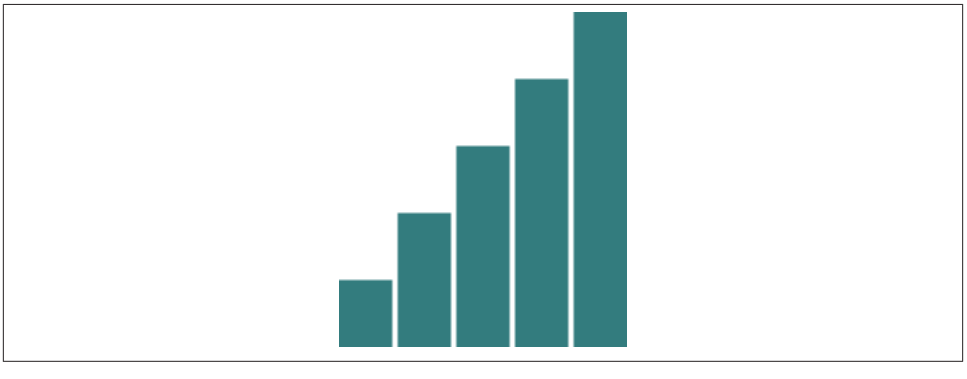


Figure 6-5. A taller bar chart

Nice! We could go to SIGGRAPH with that chart.

Try out the sample code `03_drawing_divs_spaced.html`. Again, view the source and use the web inspector to contrast the original HTML against the final DOM.

The Power of data()

This is exciting, but real-world data is never this clean:

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

Let's make our data a bit messier, as in `04_power_of_data.html`:

```
var dataset = [ 25, 7, 5, 26, 11 ];
```

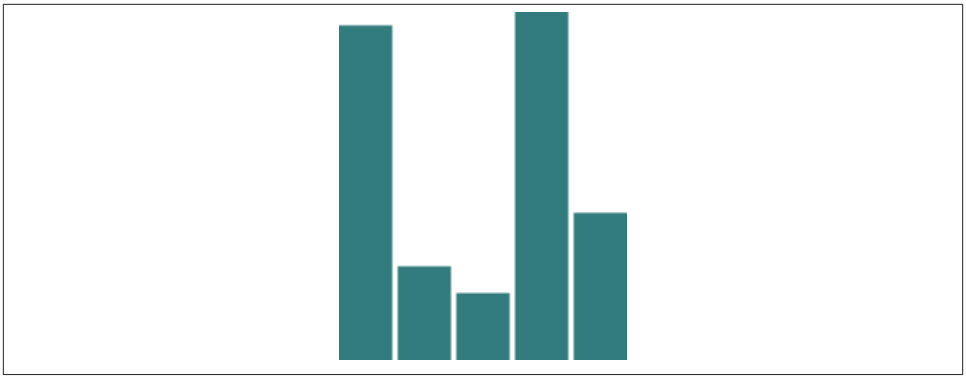


Figure 6-6. New data values

We're not limited to five data points, of course. Let's add many more! (See `05_power_of_data_more_points.html`.)

```
var dataset = [ 25, 7, 5, 26, 11, 8, 25, 14, 23, 19,
                14, 11, 22, 29, 11, 13, 12, 17, 18, 10,
                24, 18, 25, 9, 3 ];
```

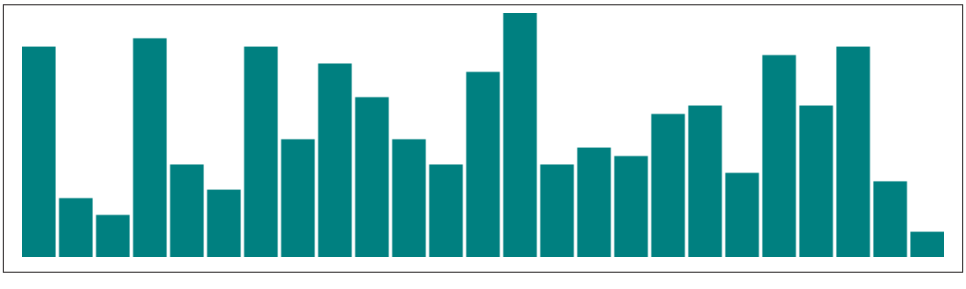


Figure 6-7. Lots more data values

25 data points instead of five! How does D3 automatically expand our chart as needed?

```
d3.select("body").selectAll("div")
    .data(dataset)  // <-- The answer is here!
    .enter()
    .append("div")
    .attr("class", "bar")
    .style("height", function(d) {
        var barHeight = d * 5;
        return barHeight + "px";
    });
```

Give `data()` ten values, and it will loop through ten times. Give it one million values, and it will loop through one million times. (Just be patient.)

That is the power of `data()` — being smart enough to loop through the full length of whatever data set you throw at it, executing each method beneath it in the chain, while updating the context in which each method operates, so `d` always refers to the current datum at that point in the loop.

That may be a mouthful, and if it all doesn't make sense yet, it will soon. I encourage you to make a copy of `05_power_of_data_more_points.html`, tweak the `dataset` values, and note how the bar chart changes.

Remember, the *data* is driving the visualization — not the other way around.

Random Data

Sometimes it's fun to generate random data values, whether for testing purposes or just pure geekiness. That's just what I've done in `06_power_of_data_random.html`. Notice that each time you reload the page, the bars render differently.

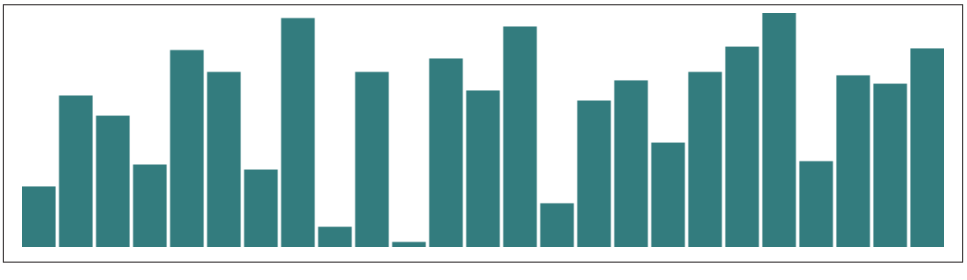
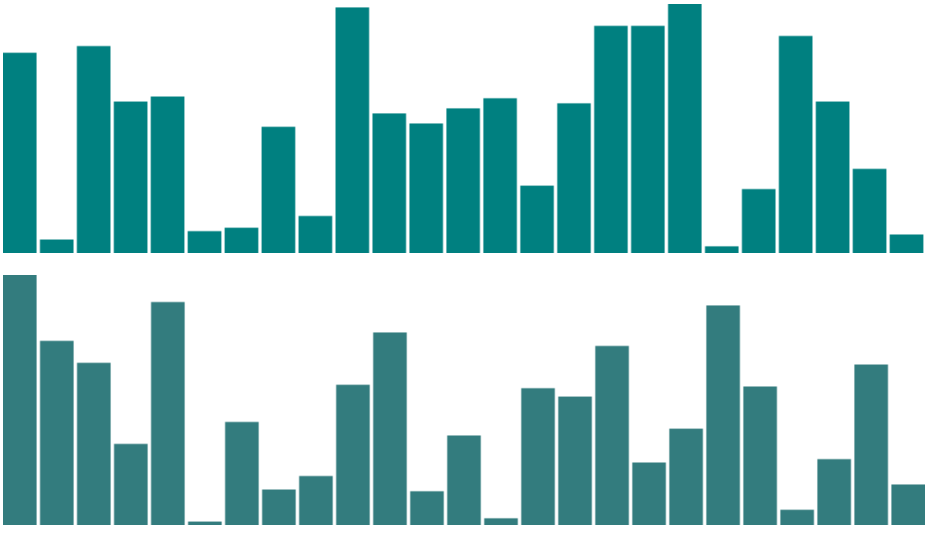


Figure 6-8. Bar charts with random values



View the source, and you'll see this code:

```
var dataset = [];                        //Initialize empty array
for (var i = 0; i < 25; i++) {           //Loop 25 times
    var newNumber = Math.random() * 30;  //New random number (0-30)
    dataset.push(newNumber);             //Add new number to array
}
```

That code doesn't use any D3 methods; it's just JavaScript. Without going into too much detail, the code above:

1. Creates an empty array called `dataset`.
2. Initiates a `for` loop, which is executed 25 times.
3. Each time, it generates a new random number with a value between zero and 30.
4. That new number is appended to the `dataset` array. (`push()` is an array method that appends a new value to the end of an array.)

Just for kicks, open up the JavaScript console and enter `console.log(dataset)`. You should see the full array of 25 randomized data values.

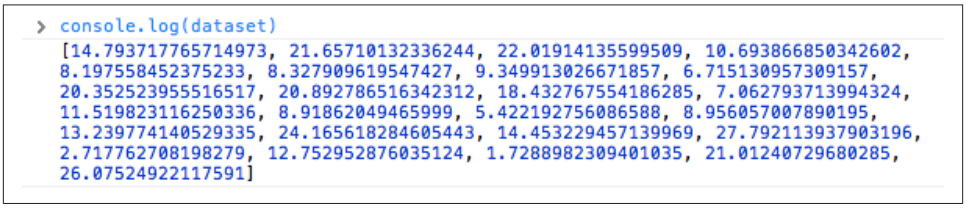


Figure 6-9. Random values in console

Notice that they are all decimal or floating point values (14.793717765714973), not whole numbers or integers (14) like we used initially. For this example, decimal values are fine, but if you ever need whole numbers, you can use JavaScript's `Math.round()` method. For example, you could wrap the random number generator from this line

```
var newNumber = Math.random() * 30;
```

as follows:

```
var newNumber = Math.round(Math.random() * 30);
```

Try it out in `07_power_of_data_rounded.html`, and use the console to verify that the numbers have indeed been rounded to integers:

```
> console.log(dataset)
  [23, 19, 18, 16, 24, 29, 1, 5, 13, 4, 29, 23, 11, 9, 16, 10, 15, 4, 28, 23, 13,
  19, 20, 20, 27]
```

Figure 6-10. Random integer values in console

That's about all we can do visually with `div`s. Let's expand our visual possibilities with SVG.

Drawing SVGs

For a quick refresher on SVG syntax, see the "SVG" portion of the "Technology Fundamentals" chapter.

One thing you may notice about SVG elements is that all of their properties are specified as *attributes*. That is, they are included as property/value pairs within each element tag, like this:

```
<element property="value"></element>
```

Hmm, that looks strangely like HTML!

```
<p class="eureka">Eureka!</p>
```

We have already used D3's handy `append()` and `attr()` methods to create new HTML elements and set their attributes. Since SVG elements exist in the DOM, just as HTML elements do, we can use `append()` and `attr()` in exactly the same way to generate SVG images!

Create the SVG

First, we need to create the SVG element in which to place all our shapes.

```
d3.select("body").append(.png");
```

That will find the document's `body` and append a new `svg` element just before the closing `</body>` tag. While that code will work, may I suggest a slight modification?

```
var svg = d3.select("body").append(.png");
```

Remember how most D3 methods return a reference to the DOM element on which they act? By creating a new variable `svg`, we are able to capture the reference handed back by `append()`. Think of `svg` not as a "variable" but as a "reference pointing to the SVG object that we just created." This reference will save us a lot of code later. Instead of having to search for that SVG each time — as in `d3.select(.png")` — we just say `svg`.

```
svg.attr("width", 500)
   .attr("height", 50);
```

Alternatively, that could all be written as one line of code:

```
var svg = d3.select("body")
            .append(.png")
            .attr("width", 500)
            .attr("height", 50);
```

See `08_drawing_svgs.html` for that code. Inspect the DOM and notice that there is, indeed, an empty SVG element.

To simplify your life, I recommend putting the width and height values into variables at the top of your code, as in `09_drawing_svgs_size.html`. View the source, and you'll see:

```
//Width and height
var w = 500;
var h = 50;
```

I'll be doing that with all future examples. By *variabalizing* the size values, they can be easily referenced throughout your code, as in:

```
var svg = d3.select("body")
            .append(.png")
            .attr("width", w)   // <-- Here
            .attr("height", h); // <-- and here!
```

Also, if you send me a petition to make "variabalize" a real word, I will gladly sign it.

Data-driven Shapes

Time to add some shapes. I'll bring back our trusty old data set

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

and then use `data()` to iterate through each data point, creating a `circle` for each one:

```
svg.selectAll("circle")
    .data(dataset)
    .enter()
    .append("circle");
```

Remember, `selectAll()` will return empty references to all `circle`s (which don't exist yet), `data()` binds our data to the elements we're about to create, `enter()` returns a placeholder reference to the new element, and `append()` finally adds a `circle` to the DOM. In this case, it appends those `circle`s to the end of the SVG element, since our initial selection is our reference `svg` (as opposed to the document `body`, for example).

To make it easy to reference all of the `circle`s later, we can create a new variable to store references to them all:

```
var circles = svg.selectAll("circle")
                 .data(dataset)
                 .enter()
                 .append("circle");
```

Great, but all these circles still need positions and sizes. Be warned: The following code may blow your mind.

```
circles.attr("cx", function(d, i) {
            return (i * 50) + 25;
        })
       .attr("cy", h/2)
       .attr("r", function(d) {
            return d;
       });
```

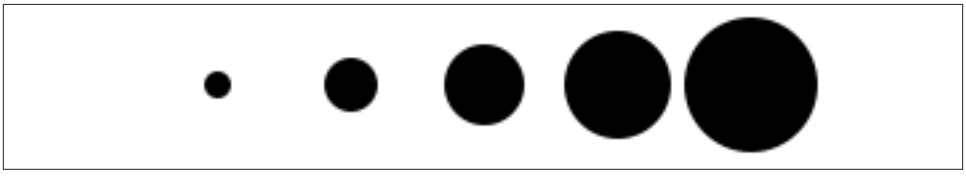


Figure 6-11. Row of data circles

Feast your eyes on the demo, `10_drawing_svgs_circles.html`. Let's step through the code, one line at a time.

```
circles.attr("cx", function(d, i) {
            return (i * 50) + 25;
        })
```

This takes the reference to all `circle`s and sets the `cx` attribute for each one. (Remember that, in SVG lingo, `cx` is the x position value of the *center* of the circle.) Our data has already been bound to the `circle` elements, so for each `circle`, the value `d` matches the corresponding value in our original data set (5, 10, 15, 20, or 25).

Another value, `i`, is also automatically populated for us. (Thanks, D3!) `i` is a numeric index value of the current element. Counting starts at zero, so for our "first" circle `i == 0`, the second circle's `i == 1` and so on. We're using `i` to push each subsequent circle over to the right, because each subsequent loop through, the value of `i` increases by one.

```
(0 * 50) + 25   //Returns 25
(1 * 50) + 25   //Returns 75
(2 * 50) + 25   //Returns 125
(3 * 50) + 25   //Returns 175
(4 * 50) + 25   //Returns 225
```

To make sure `i` is available to your custom function, you must include it as an argument in the function definition, `function(d, i)`. You must also include `d`, even if you don't use `d` within your function (as in the case above).

On to the next line.

```
.attr("cy", h/2)
```

`cy` is the y position value of the center of each circle. We're setting `cy` to `h` divided by two, or one-half of `h`. You'll recall that `h` stores the height of the entire SVG, so `h/2` has the effect of aligning all `circle`s in the vertical center of the image.

```
.attr("r", function(d) {
    return d;
});
```

Finally, the radius `r` of each `circle` is simply set to `d`, the corresponding data value.

Pretty Colors, Oooh!

Color fills and strokes are just other attributes that you can set using the same methods. Simply by appending this code

```
.attr("fill", "yellow")
.attr("stroke", "orange")
.attr("stroke-width", function(d) {
    return d/2;
});
```

we get the following colorful circles, as seen in `11_drawing_svgs_color.html`:

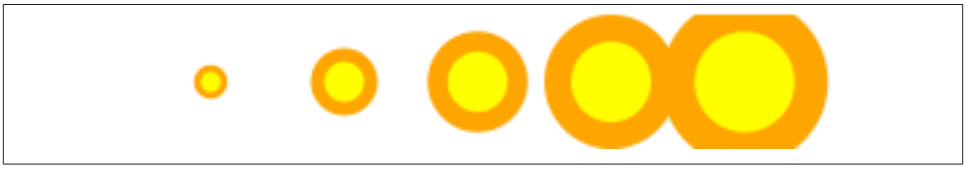


Figure 6-12. Colorful data circles

Of course, you can mix and match attributes and custom functions to apply any combination of properties. The trick with data visualization, of course, is choosing appropriate *mappings*, so the visual expression of your data is understandable and useful for the viewer.

Making a Bar Chart

Now we'll integrate everything we've learned so far to generate a simple bar chart as an SVG image.

We'll start by adapting the `div` bar chart code to draw its bars with SVG instead, giving us more flexibility over the visual presentation. Then we'll add labels, so we can see the data values clearly.

The Old Chart

See the `div` chart, updated with some new data, in `12_making_a_bar_chart_divs.html`:

```
var dataset = [ 5, 10, 13, 19, 21, 25, 22, 18, 15, 13,
                11, 12, 15, 20, 18, 17, 16, 18, 23, 25 ];

d3.select("body").selectAll("div")
    .data(dataset)
    .enter()
    .append("div")
    .attr("class", "bar")
    .style("height", function(d) {
        var barHeight = d * 5;
        return barHeight + "px";
    });
```

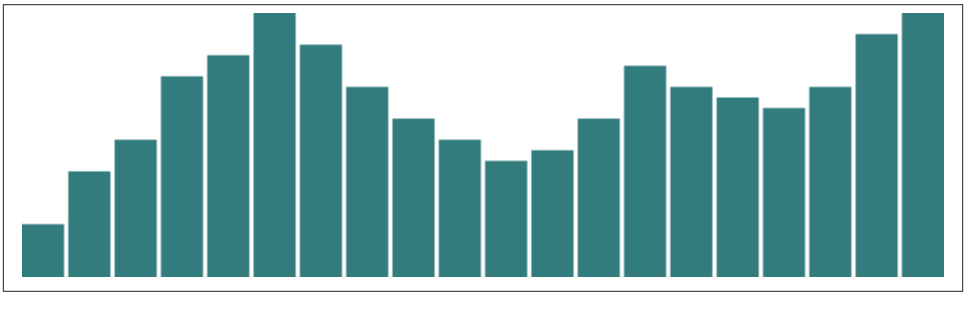


Figure 6-13. Bar chart with `div`s

It may be hard to imagine, but we can definitely improve on this simple bar chart made of `div`s.

The New Chart

First things first, we need to decide on the size of the new SVG:

```
//Width and height
var w = 500;
var h = 100;
```

(Of course, you could name `w` and `h` something else, like `svgWidth` and `svgHeight`. Use whatever is most clear to you. JavaScript programmers, as a group, are fixated on efficiency, so you'll often see single-character variable names, code written with no spaces, and other hard-to-read, yet programmatically efficient, syntax.)

Then, we tell D3 to create an empty SVG element and add it to the DOM:

```
//Create SVG element
var svg = d3.select("body")
            .append(.png")
            .attr("width", w)
            .attr("height", h);
```

To recap, this inserts a new `<svg>` element just before the closing `</body>` tag, and assigns the SVG a width and height of 500 by 100 pixels. This statement also puts the result into our new variable called `svg`, so we can easily reference the new SVG without having to reselect it later using something like `d3.select(.png")`.

Next, instead of creating `div`s, we generate `rect`s and add them to `svg`.

```
svg.selectAll("rect")
   .data(dataset)
   .enter()
   .append("rect")
   .attr("x", 0)
   .attr("y", 0)
   .attr("width", 20)
   .attr("height", 100);
```

This code selects all `rect`s inside of `svg`. Of course, there aren't any yet, so an empty selection is returned. (Weird, yes, but stay with me. With D3, you always have to first select whatever it is you're about to act on, even if that selection is momentarily empty.)

Then, `data(dataset)` sees that we have 20 values in the data set, so it calls `enter()` 20 times. `enter()`, in turn, returns a placeholder selection for each data point that does not yet have a corresponding `rect` — which is to say, all of them.

For each of the 20 placeholders, `append("rect")` inserts a `rect` into the DOM. As we learned in the "Technology Fundamentals" chapter, every `rect` must have `x`, `y`, `width`, and `height` values. We use `attr()` to add those attributes onto each newly created `rect`.

Beautiful, no?

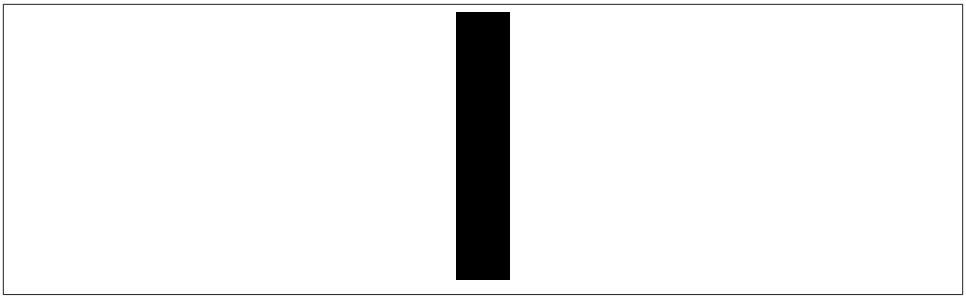


Figure 6-14. One lonely bar

Okay, maybe not. All of the bars are there (check the DOM of `13_making_a_bar_chart_rects.html` with your web inspector), but they all share the same `x`, `y`, `width`, and `height` values, with the result that they all overlap. This isn't a visualization of data yet.

Let's fix the overlap issue first. Instead of an `x` of zero, we'll assign a dynamic value that corresponds to `i`, or each value's position in the data set. So the first bar will be at zero, but subsequent bars will be at `21`, then `42`, and so on.

```
.attr("x", function(d, i) {
    return i * 21;  //Bar width of 20 plus 1 for padding
})
```

Figure 6-15. Twenty bars

See that code in action with `14_making_a_bar_chart_offset.html`.

That works, but it's not particularly flexible. If our data set were longer, then the bars would just run off to the right, past the end of the SVG! Since each bar is 20 pixels wide, plus 1 pixel of padding, then a 500-pixel wide SVG can only accommodate 23 data points. Note how the 24th bar here gets clipped:

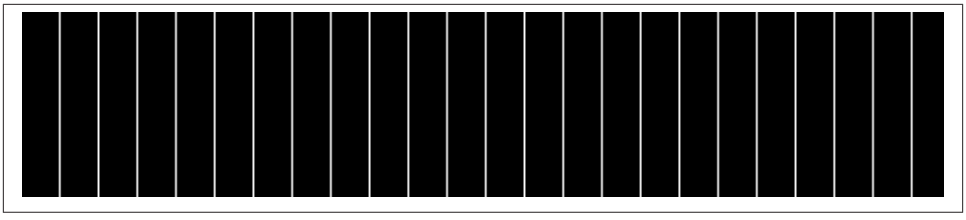


Figure 6-16. Twenty-four bars

It's good practice to use flexible, dynamic coordinates — heights, widths, x values, and y values — so your visualization can scale appropriately along with your data.

As with anything else in programming, there are a thousand ways to achieve that end. I'll use a simple one. First, I'll amend the line where we set each bar's `x` position:

```
.attr("x", function(d, i) {
    return i * (w / dataset.length);
})
```

Notice how the `x` value is now tied directly to the width of the SVG (`w`) and the number of values in the data set (`dataset.length`). This is exciting, because now our bars will be evenly spaced, whether we have 20 data values

Figure 6-17. Twenty evenly spaced bars

or only five:

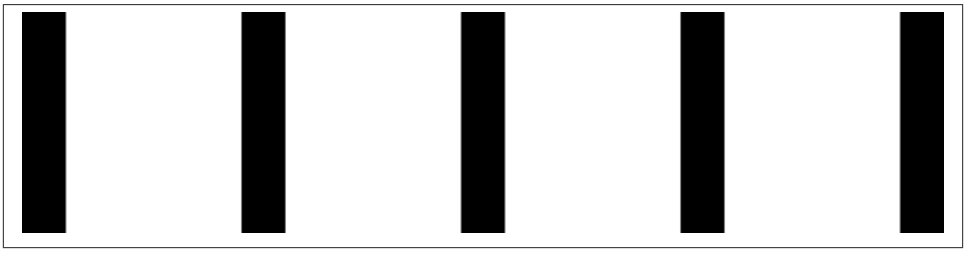


Figure 6-18. Five evenly spaced bars

See that code so far in `15_making_a_bar_chart_even.html`.

Now we should set the bar *widths* to be proportional, too, so they get narrower as more data is added, or wider when there are fewer values. I'll add a new variable near where we set the SVG's width and height

```
//Width and height
var w = 500;
var h = 100;
var barPadding = 1;  // <-- New!
```

and then reference that variable in the line where we set each bar's `width`. Instead of a static value of `20`, the width will now be set as a fraction of the SVG width and number of data points, minus a padding value:

```
.attr("width", w / dataset.length - barPadding)
```

Figure 6-19. Twenty evenly spaced bars with dynamic widths

It works! (See `16_making_a_bar_chart_widths.html`.) The bar widths and x positions scale correctly whether there are 20 points, only five

Figure 6-20. Five evenly spaced bars with dynamic widths

or even 100:

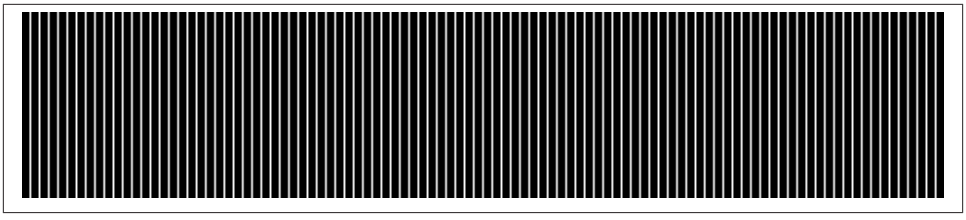


Figure 6-21. One hundred evenly spaced bars with dynamic widths

Finally, we encode our data as the *height* of each bar. You would hope it were as easy as referencing the `d` data value when setting each bar's `height`:

```
.attr("height", function(d) {
    return d;
});
```

Figure 6-22. Dynamic heights

Hmm, that looks funky. Maybe we can just scale up our numbers a bit?

```
.attr("height", function(d) {
    return d * 4;  // <-- Times four!
});
```

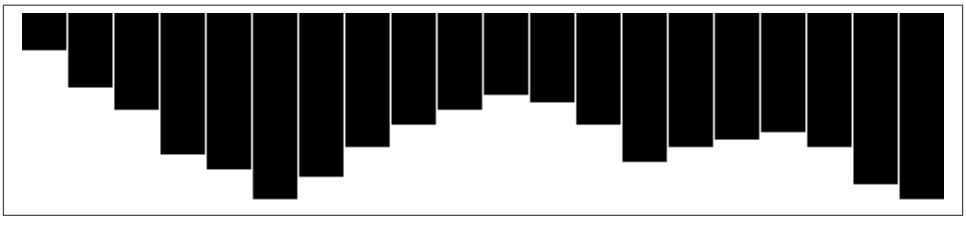


Figure 6-23. Dynamic heights, magnified

Alas, it is not that easy! We want our bars to grow upward from the bottom edge, not down from the top — but don't blame D3, blame SVG.

You'll recall that, when drawing SVG `rect`s, the `x` and `y` values specify the coordinates of the *upper-left corner*. That is, the origin or reference point for every `rect` is its top-left. For our purposes, it would be soooooo much easier to set the origin point as the bottom-left corner, but that's just not how SVG does it, and, frankly, SVG is pretty indifferent about your feelings on the matter.

Given that our bars do have to "grow down from the top," then where is "the top" of each bar in relationship to the top of the SVG? Well, the top of each bar could be expressed as a relationship between the height of the SVG and the corresponding data value, as in:

```
.attr("y", function(d) {
    return h - d;  //Height minus data value
})
```

Then, to put the "bottom" of the bar on the bottom of the SVG, each `rect`s height can be just the data value itself:

```
.attr("height", function(d) {
    return d;  //Just the data value
});
```

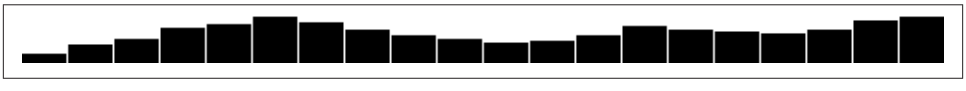


Figure 6-24. Growing down from above

Let's scale things up a bit by changing `d` to `d * 4`. (Note: Later we'll learn about D3 *scales*, which offer better ways to accomplish this.)

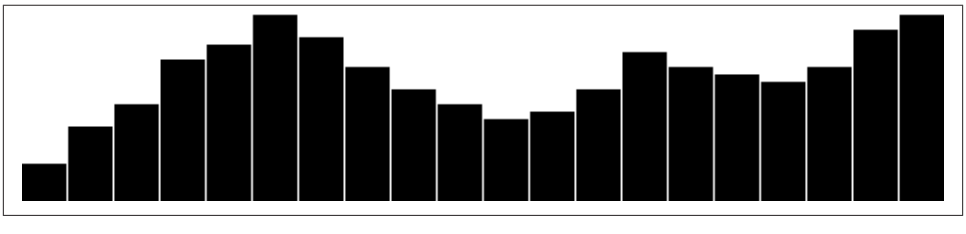


Figure 6-25. Growing bigger from above

The working code for our growing-down-from-above, SVG bar chart is in `17_making_a_bar_chart_heights`.

Color

Adding color is easy. Just use `attr()` to set a `fill`:

```
.attr("fill", "teal");
```

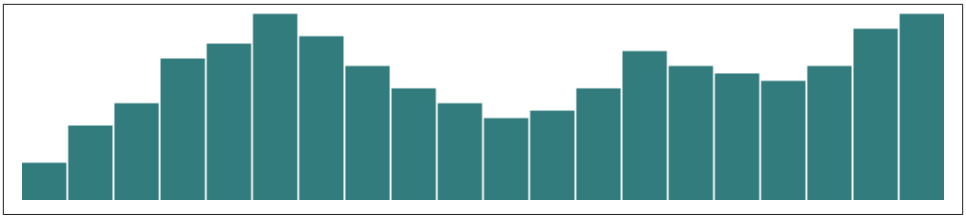


Figure 6-26. Teal bars

Find that all-teal bar chart in `18_making_a_bar_chart_teal`.

Teal is nice, but you'll often want a shape's color to reflect some quality of the data. That is, you may want to *encode* the data values as color. (In the case of our bar chart, that makes a *dual encoding*, in which the same data value is encoded in two different visual properties: both height and color.)

Using data to drive color is as easy as writing a custom function that again references `d`. Here, we replace `"teal"` with a custom function:

```
.attr("fill", function(d) {
    return "rgb(0, 0, " + (d * 10) + ")";
});
```

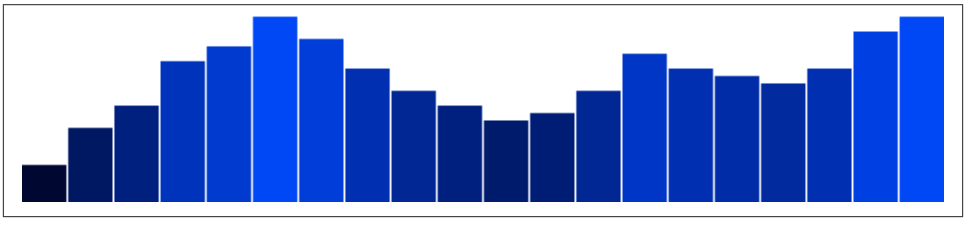


Figure 6-27. Data-driven blue bars

See the code in `19_making_a_bar_chart_blues.html`. This is not a particularly useful visual encoding, but you can get the idea of how to translate data into color. Here, `d` is multiplied by 10, and then used as the blue value in an `rgb()` color definition. So the greater values of `d` (taller bars) will be more blue. Smaller values of `d` (shorter bars) will be less blue (closer to black). The red and green components of the color are fixed at zero.

Labels

Visuals are great, but sometimes you need to show the actual data values as text within the visualization. Here's where value labels come in, and they are very, very easy to generate with D3.

You'll recall from the SVG primer that you can add `text` elements to an SVG element. We'll start with:

```
svg.selectAll("text")
   .data(dataset)
   .enter()
   .append("text")
```

Look familiar? Just as we did for the `rect`s, here we do for the `text`s. First, select what you want, bring in the data, enter the new elements (which are just placeholders at this point), and finally append the new `text` elements to the DOM.

We'll extend that code to include a data value within each `text` element by using the `text()` method

```
.text(function(d) {
     return d;
})
```

and then extend it further, by including `x` and `y` values to position the text. It's easiest if I just copy and paste the same x/y code we used above for the bars:

```
.attr("x", function(d, i) {
     return i * (w / dataset.length);
})
.attr("y", function(d) {
     return h - (d * 4);
});
```

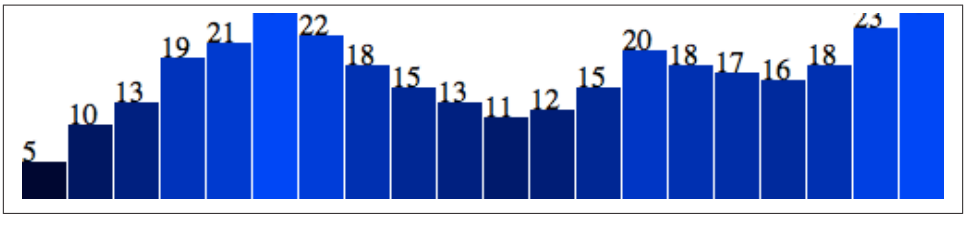


Figure 6-28. Baby value labels!

Aha! Value labels! But some are getting cut off at the top. Let's try moving them down, inside the bars, by adding a small amount to the `x` and `y` calculations:

```
.attr("x", function(d, i) {
     return i * (w / dataset.length) + 5;  // +5
})
.attr("y", function(d) {
     return h - (d * 4) + 15;              // +15
});
```

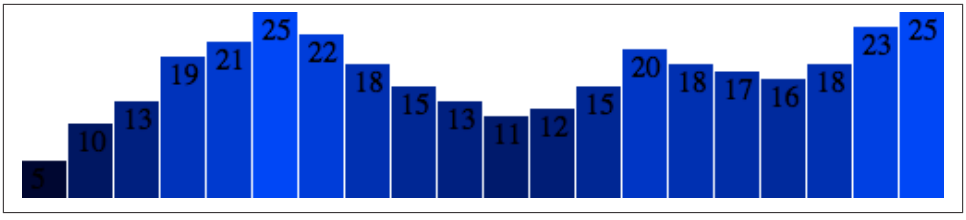


Figure 6-29. In-bar value labels

Better, but not legible. Fortunately, we can fix that:

```
.attr("font-family", "sans-serif")
.attr("font-size", "11px")
.attr("fill", "white");
```

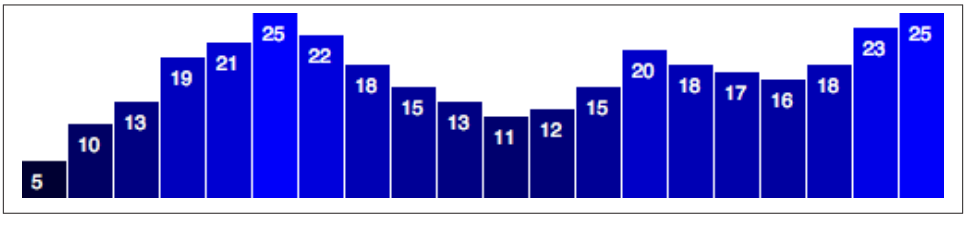


Figure 6-30. Really nice value labels

Fantasti-code! See `20_making_a_bar_chart_labels.html` for that brilliant visualization.

If you are not typographically obsessive, then you're all done. If, however, you are like me, you'll notice that the value labels aren't perfectly aligned within their bars. (For example, note the "5" in the first column.) That's easy enough to fix. Let's use the SVG `text-anchor` attribute to center the text horizontally at the assigned `x` value:

```
.attr("text-anchor", "middle")
```

Then, let's change the way we calculate the `x` position by setting it to the left edge of each bar *plus* half the bar width:

```
.attr("x", function(d, i) {
    return i * (w / dataset.length) + (w / dataset.length - barPadding) / 2;
})
```

And I'll also bring the labels up one pixel for perfect spacing, as you can see in `21_making_a_bar_chart_aligned.html`:

```
.attr("y", function(d) {
    return h - (d * 4) + 14;  //15 is now 14
})
```

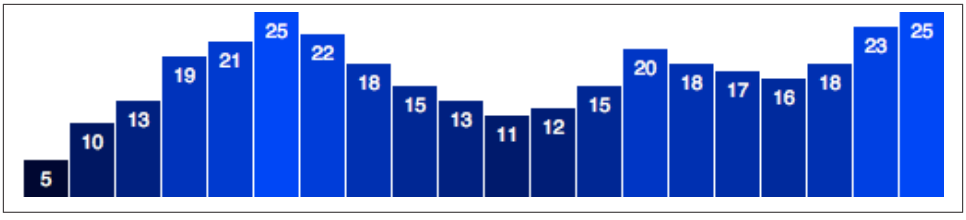


Figure 6-31. Centered labels

Done!

Making a Scatterplot

So far, we've drawn only bar charts with simple data — just one-dimensional sets of numbers.

But when you have two sets of values to plot against each other, you need a second dimension. The scatterplot is a common type of visualization that represents two sets of corresponding values on two different axes: horizontal and vertical, x and y.

The Data

As you saw in the "Technology Fundamentals" section on JavaScript, you have a lot of flexibility around how to structure a data set. For our scatterplot, I'm going to use an array of arrays. The primary array will contain one element for each data "point." Each of those "point" elements will be another array, with just two values: one for the x value, and one for y.

```
var dataset = [
                [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
                [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]
              ];
```

Remember, `[]` means array, so nested hard brackets `[[]]` indicate an array within another array. We separate array elements with commas, so an array containing three other arrays would look like: `[[],[],[]]`

We could rewrite our data set with more whitespace so it's easier to read, like so:

```
var dataset = [
                  [ 5,     20 ],
                  [ 480,   90 ],
                  [ 250,   50 ],
                  [ 100,   33 ],
                  [ 330,   95 ],
                  [ 410,   12 ],
                  [ 475,   44 ],
```

```
            [ 25,    67 ],
            [ 85,    21 ],
            [ 220,   88 ]
        ];
```

Now you can see that each of these 10 rows will correspond to one point in our visualization. With the row `[5, 20]`, for example, we'll use `5` as the x value, and `20` for the y.

The Scatterplot

Let's carry over most of the code from our bar chart experiments, including the piece that creates the SVG element:

```
//Create SVG element
var svg = d3.select("body")
            .append(.png")
            .attr("width", w)
            .attr("height", h);
```

Instead of creating `rect`s, however, we'll make a `circle` for each data point:

```
svg.selectAll("circle")  // <-- No longer "rect"
   .data(dataset)
   .enter()
   .append("circle")     // <-- No longer "rect"
```

Also, instead of specifying the `rect` attributes of `x`, `y`, `width`, and `height`, our `circle`s need `cx`, `cy`, and `r`:

```
      .attr("cx", function(d) {
           return d[0];
      })
      .attr("cy", function(d) {
           return d[1];
      })
      .attr("r", 5);
```

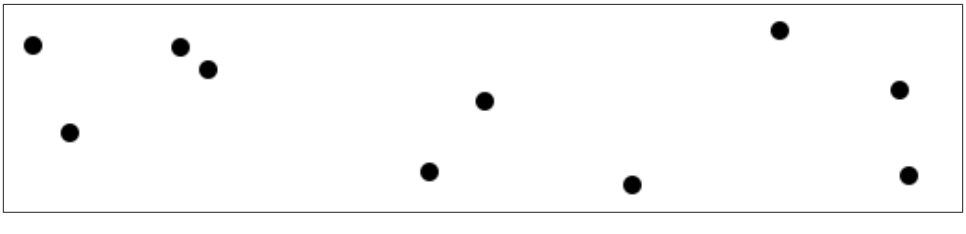


Figure 6-32. Simple scatterplot

See the working scatterplot code in `22_scatterplot.html`.

Notice how we access the data values and use them for the `cx` and `cy` values. When using `function(d)`, D3 automatically hands off the current data value as `d` to your function. In this case, the current data value is one of the smaller, sub-arrays in our larger `dataset` array.

When each single datum `d` is itself an array of values (and not just a single value, like `3.14159`), you need to use bracket notation to access its values. Hence, instead of `return d`, we have `return d[0]` and `return d[1]`, which return the first and second values of the array, respectively.

For example, in the case of our first data point `[5, 20]`, the first value (array position `0`) is `5`, and the second value (array position `1`) is `20`. Thus:

```
d[0] == 5
d[1] == 20
```

By the way, if you ever want to access any value in the larger data set (outside of D3, say), you can do so using bracket notation. For example:

```
dataset[5] == [410, 12]
```

You can even use multiple sets of brackets to access values within nested arrays:

```
dataset[5][1] == 12
```

Don't believe me? Take another look at the scatterplot page `22_scatterplot.html`, open your JavaScript console, type in `dataset[5]` or `dataset[5][1]`, and see what happens.

Size

Maybe you want the circles to be different sizes, so their radii correspond to their y values. Instead of setting all `r` values to `5`, try:

```
.attr("r", function(d) {
    return Math.sqrt(h - d[1]);
});
```

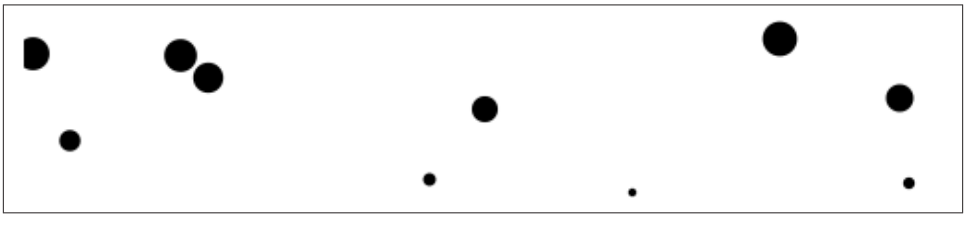


Figure 6-33. Scatterplot with sized circles

See `23_scatterplot_sqrt.html` for the code. `Math.sqrt()` is JavaScript's square root function. Here, we arbitrarily subtract the datum's y value `d[1]` from the SVG height `h`, and then take the square root. The effect is that circles with greater y values (those circles lower down) have smaller radii.

This is neither pretty nor useful, but it illustrates how to use `d`, along with bracket notation, to reference data values and set `r` accordingly.

Labels

Let's label our data points with `text` elements. I'll adapt the label code from our bar chart experiments, starting with:

```
svg.selectAll("text")  // <-- Note "text", not "circle" or "rect"
   .data(dataset)
   .enter()
   .append("text")     // <-- Same here!
```

This looks for all `text` elements in the SVG (there aren't any yet), and then appends a new `text` element for each data point. Then use the `text()` method to specify each element's contents:

```
      .text(function(d) {
           return d[0] + "," + d[1];
      })
```

This looks messy, but bear with me. Once again, we're using `function(d)` to access each data point. Then, within the function, we're using *both* `d[0]` *and* `d[1]` to get both values within that data point array.

The plus `+` symbols, when used with strings, such as the comma between quotation marks `","`, act as *append* operators. So what this one line of code is really saying is: Get the values of `d[0]` and `d[1]` and smush them together with a comma in the middle. The end result should be something like `5,20` or `25,67`.

Next, we specify *where* the text should be placed with `x` and `y` values. For now, let's just use `d[0]` and `d[1]`, the same values that we used to specify the `circle` positions.

```
      .attr("x", function(d) {
           return d[0];
      })
      .attr("y", function(d) {
           return d[1];
      })
```

Finally, add a bit of font styling with:

```
      .attr("font-family", "sans-serif")
      .attr("font-size", "11px")
      .attr("fill", "red");
```

Figure 6-34. Scatterplot with labels

It may not be pretty, but we got it working! See `24_scatterplot_labels.html` for the latest.

Next Steps

Hopefully, some core concepts of D3 are becoming clear: Loading data, generating new elements, and using data values to derive attribute values for those elements.

Yet the image above is barely passable as a data visualization. The scatterplot is hard to read, and the code doesn't use our data flexibly. To be honest, we haven't yet improved on — gag — Excel's *Chart Wizard!*

Not to worry: D3 is way cooler than Chart Wizard (not to mention Clippy), but generating a shiny, interactive chart involves taking our D3 skills to the next level. To use data flexibly, we'll learn about D3's *scales* in the next chapter. And to make our scatterplot easier to read, we'll learn about *axis generators* and axis labels.

This would be a good time to take a break and stretch your legs. Maybe go for a walk, or grab a coffee or a sandwich. I'll hang out here (if you don't mind), and when you get back, we'll jump into D3 scales!

CHAPTER 7
Scales

"Scales are functions that map from an input domain to an output range."

That's Mike Bostock's definition of D3 scales.

The values in any data set are unlikely to correspond exactly to pixel measurements for use in your visualization. Scales provide a convenient way to map those data values to new values useful for visualization purposes.

D3 scales are *functions* whose parameters you define. Once they are created, you call the scale function, pass it a data value, and it nicely returns a scaled output value. You can define and use as many scales as you like.

It may be tempting to think of a scale as something that appears visually in the final image — like a set of tick marks, indicating a progression of values. *Do not be fooled!* Those tick marks are part of an *axis*, which is a *visual representation* of a scale. A scale is a mathematical relationship, with no direct visual output. I encourage you to think of scales and axes as two different, yet related, elements.

This chapter addresses only *linear* scales, since they are most common and easiest understood. Once you understand linear scales, the others — ordinal, logarithmic, square root, and so on — will be a piece of cake.

Apples and Pixels

Imagine that the following data set represents the number of apples sold at a roadside fruit stand each month:

```
var dataset = [ 100, 200, 300, 400, 500 ];
```

First of all, this is great news, as the stand is selling 100 additional apples each month! Business is booming. To showcase this success, you want to make a bar chart illustrating the steep upward climb of apple sales, with each data value corresponding to the height of one bar.

Until now, we've used data values directly as display values, ignoring unit differences. So if 500 apples were sold, the corresponding bar would be 500 pixels tall.

That could work, but what about next month, when 600 apples are sold? And a year later, when 1,800 apples are sold? Your audience would have to purchase ever-larger displays, just to be able to see the full height of those very tall apple-bars! (Mmm, apple bars!)

This is where scales come in. Because apples are not pixels (which are also not oranges), we need scales to translate between them.

Domains and Ranges

A scale's *input domain* is the range of possible input data values. Given the apples data above, appropriate input domains would be either 100 and 500 (the minimum and maxmium values of the data set) or zero and 500.

A scale's *output range* is the range of possible output values, commonly used as display values in pixel units. The output range is completely up to you, as the information designer. If you decide the shortest apple-bar will be 10 pixels tall, and the tallest will be 350 pixels tall, then you could set an output range of 10 and 350.

For example, create a scale with an input domain of `100,500` and an output range of `10,350`. If you handed the low input value of `100` to that scale, it would return its lowest range value, `10`. If you gave it `500`, it would spit back `350`. If you gave it `300`, it would hand `180` back to you on a silver platter. (`300` is in the center of the domain, and `180` is in the center of the range.)

We can visualize the domain and range as corresponding axes, side-by-side:

```
Input domain 100 300 500 10 180 350 Output range
```

One more thing: To prevent your brain from mixing up the *input domain* and *output range* terminology, I'd like to propose a little exercise. When I say "input," you say "domain." Then I say "output," and you say "range." Ready? Okay:

- Input! Domain!
- Output! Range!
- Input! Domain!
- Output! Range!

Got it? Great.

Normalization

If you're familiar with the concept of *normalization*, it may be helpful to know that, with a linear scale, that's all that is really going on here.

Normalization is the process of mapping a numeric value to a new value between 0 and 1, based on the possible minimum and maximum values. For example, with 365 days in the year, day number 310 maps to about 0.85, or 85% of the way through the year.

With linear scales, we are just letting D3 handle the math of the normalization process. The input value is normalized according to the domain, and then the normalized value is scaled to the output range.

Creating a Scale

D3's scale function generators are accessed with `d3.scale` followed by the type of scale you want. I recommend opening up the sample code page `01_scale_test.html` and typing each of the following into the console.

```
var scale = d3.scale.linear();
```

Congratulations! Now `scale` is a function to which you can pass input values. (Don't be misled by the `var` above. Remember that in JavaScript, variables can store functions.)

```
scale(2.5);  //Returns 2.5
```

Since we haven't set a domain and a range yet, this function will map input to output on a 1:1 scale. That is, whatever we input will be returned unchanged.

We can set the scale's input domain to `100,500` by passing those values to the `domain()` method as an array. Note the hard brackets indicating an array:

```
scale.domain([100, 500]);
```

Set the output range in similar fashion, with `range()`:

```
scale.range([10, 350]);
```

These steps can be done separately, as above, or chained together into one line of code:

```
var scale = d3.scale.linear()
                    .domain([100, 500])
                    .range([10, 350]);
```

Either way, our scale is ready to use!

```
scale(100);  //Returns 10
scale(300);  //Returns 180
scale(500);  //Returns 350
```

Typically, you will call scale functions from within an `attr()` method or similar, not on their own. Let's modify our scatterplot visualization to use dynamic scales.

Scaling the Scatterplot

To revisit our data set from the scatterplot:

```
var dataset = [
                [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
                [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]
              ];
```

You'll recall that this `dataset` is an array of arrays. We mapped the first value in each array onto the x axis, and the second value onto the y axis. Let's start with the x axis.

Just by eyeballing the x values, it looks like they range from 5 to 480, so a reasonable input domain to specify might be `0,500`, right?

…

Why are you giving me that look? Oh, because you want to keep your code flexible and scalable, so it will continue to work even if the data change in the future. Very smart! Remember, if we were building a data dashboard for the roadside apple stand, we'd want our code to accommodate the enormous projected growth in apple sales. Our chart should work just as well with 5 apples sold as 5 million.

d3.min() and d3.max()

Instead of specifying fixed values for the domain, we can use the convenient array functions `d3.min()` and `d3.max()` to analyze our data set on the fly. For example, this loops through each of the x values in our arrays and returns the value of the greatest one:

```
//Calculates max x value and returns 480
d3.max(dataset, function(d) {
    return d[0];  //References first value in each sub-array
});
```

Both `min()` and `max()` can take one or two arguments. The first argument must be a reference to the array of values you want evaluated, which is `dataset`, in this case. If you have a simple, one-dimensional array of numeric values, like `[7, 8, 4, 5, 2]`, then no second argument is needed, as it's obvious how to compare the values against each other. For example:

```
var simpleDataset = [7, 8, 4, 5, 2];
d3.max(simpleDataset);  // Returns 8
```

The `max()` function simply loops through each value in the array, and identifies the largest one.

But our `dataset` is not just an array of numbers; it is an array of arrays. Using only `d3.max(dataset)` may produce unexpected results.

```
var dataset = [
                [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
                [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]
              ];
d3.max(dataset);  // Returns [85, 21], which is not helpful for us right now
```

In order to tell `max()` which *specific* values we want compared, we must include a second argument: an *accessor function*.

```
d3.max(dataset, function(d) {
    return d[0];
});
```

The accessor function is an anonymous function to which `max()` hands off each value in the data array, one at a time, as `d`. The accessor function specifies *how to access* the value to be used for the comparison. In this case, our data array is `dataset`, and we want to compare only the x values, which are the first values in each sub-array, meaning in position `[0]`. So our accessor function looks like this:

```
function(d) {
    return d[0];  //Return the first value in each sub-array
}
```

Note that this looks suspiciously similar to the syntax we used when generating our scatterplot circles, which also used anonymous functions to retrieve and return values:

```
.attr("cx", function(d) {
    return d[0];
})
.attr("cy", function(d) {
    return d[1];
})
```

This is a common D3 pattern. Soon you will be very comfortable with all manner of anonymous functions crawling all over your code.

Seting up Dynamic Scales

Putting together all of the above, let's create the scale function for our x axis:

```
var xScale = d3.scale.linear()
                     .domain([0, d3.max(dataset, function(d) { return d[0]; })])
                     .range([0, w]);
```

First, notice that I named it `xScale`. Of course, you can name your scales whatever you want, but a name like `xScale` helps me remember what this function does.

Second, notice that both the domain and range are specified as two-value arrays in hard brackets.

Third, notice that I set the low end of the input domain to zero. (Alternatively, you could use `min()` to calculate a dynamic value.) The upper end of the domain is set to the maximum value in `dataset` (which is currently 480, but could change in the future).

Finally, observe that the output range is set to `0` and `w`, the SVG's width.

We'll use very similar code to create the scale function for the y axis:

```
var yScale = d3.scale.linear()
                     .domain([0, d3.max(dataset, function(d) { return d[1]; })])
                     .range([0, h]);
```

Note that the `max()` function here references `d[1]`, the y value of each sub-array. Also, the upper end of `range()` is set to `h` instead of `w`.

The scale functions are in place! Now all we need to do is use them.

Incorporating Scaled Values

Revisiting our scatterplot code, we now simply modify the original line where we created a `circle` for each data value

```
.attr("cx", function(d) {
    return d[0];  //Returns original value bound from dataset
})
```

to return a scaled value (instead of the original value):

```
.attr("cx", function(d) {
    return xScale(d[0]);  //Returns scaled value
})
```

Likewise, for the y axis, this

```
.attr("cy", function(d) {
    return d[1];
})
```

is modified as:

```
.attr("cy", function(d) {
    return yScale(d[1]);
})
```

For good measure, let's make the same change where we set the coordinates for the text labels, so these lines

```
.attr("x", function(d) {
    return d[0];
})
.attr("y", function(d) {
    return d[1];
})
```

become this:

```
.attr("x", function(d) {
    return xScale(d[0]);
})
.attr("y", function(d) {
    return yScale(d[1]);
})
```

And there we are!

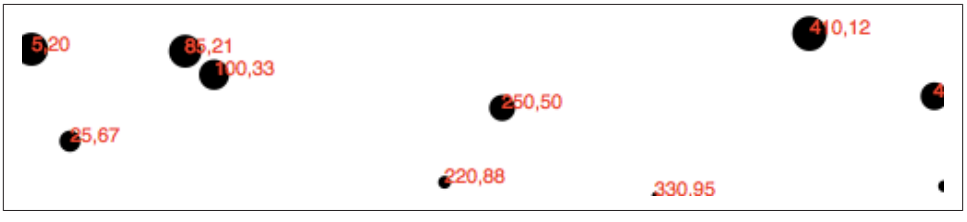


Figure 7-1. Scatterplot using x and y scales

Check out the working code in `02_scaled_plot.html`. Visually, it is disappointingly similar to our original scatterplot! Yet we are making more progress than may be apparent.

Refining the Plot

You may have noticed that smaller y values are at the top of the plot, and the larger y values are toward the bottom. Now that we're using D3 scales, it's super easy to reverse that, so greater values are higher up, as you would expect. It's just a matter of changing the output range of `yScale` from

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

to

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

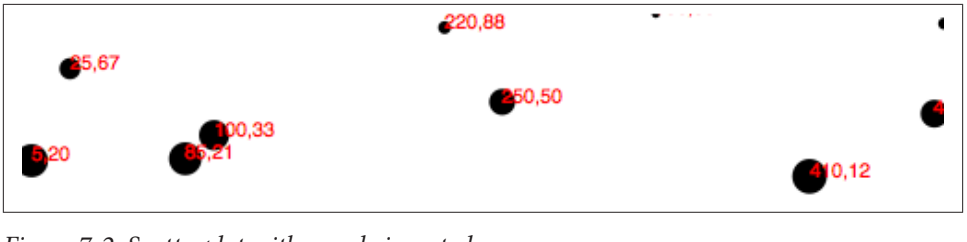


Figure 7-2. Scatterplot with y scale inverted

See `03_scaled_plot_inverted.html` for that code.

Yes, now a *smaller* input to `yScale` will produce a *larger* output value, thereby pushing those `circle`s and `text` elements down, closer to the base of the image. I know, it's almost too easy!

Yet some elements are getting cut off. Let's introduce a `padding` variable:

```
var padding = 20;
```

Then we'll incorporate the `padding` amount when setting the range of both scales. This will help push our elements in, away from the edges of the SVG, to prevent them from being clipped.

The range for `xScale` was `range([0, w])`, but now it's

```
.range([padding, w - padding]);
```

The range for `yScale` was `range([h, 0])`, but now it's

```
.range([h - padding, padding]);
```

This should provide us with 20 pixels of extra room on the left, right, top, and bottom edges of the SVG. And it does!

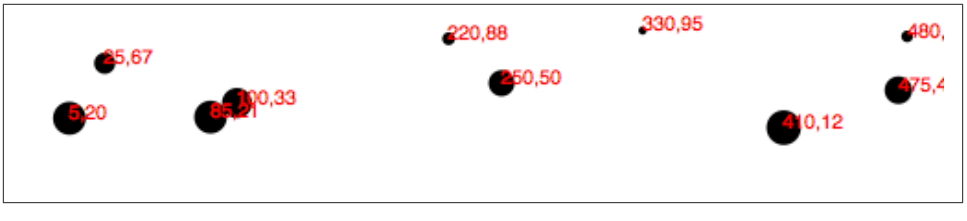


Figure 7-3. Scatterplot with padding

But the text labels on the far right are still getting cut off, so I'll double the amount of `xScale`'s padding on the right side by multiplying by two:

```
.range([padding, w - padding * 2]);
```

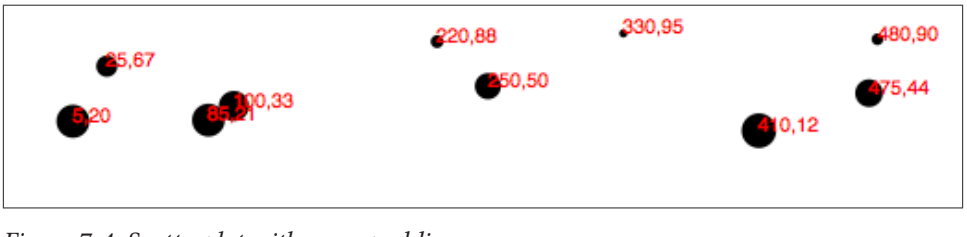


Figure 7-4. Scatterplot with more padding

Better! Reference the code so far in `04_scaled_plot_padding.html`. But there's one more change I'd like to make. Instead of setting the radius of each `circle` as the square root of its y value (which was a bit of a hack, and not visually useful), why not create another custom scale?

```
var rScale = d3.scale.linear()
                     .domain([0, d3.max(dataset, function(d) { return d[1]; })])
                     .range([2, 5]);
```

Then, setting the radius looks like this:

```
.attr("r", function(d) {
    return rScale(d[1]);
});
```

This is exciting, because we are guaranteeing that our radius values will *always* fall within the range of `2,5`. (Or *almost* always: See reference to `clamp()` below.) So data values of `0` (the minimum input) will get circles of radius `2` (or a diameter of 4 pixels). The very largest data value will get a circle of radius `5` (diameter of 10 pixels).

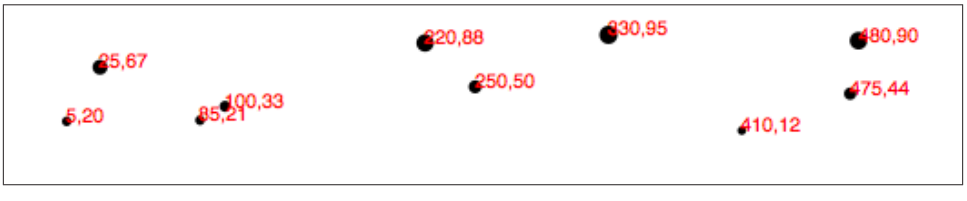


Figure 7-5. Scatterplot with scaled radii

Voila: Our first scale used for a visual property other than an axis value! (See `05_scaled_plot_radii.html`.)

Finally, just in case the power of scales hasn't yet blown your mind, I'd like to add one more array to the data set: `[600, 150]`

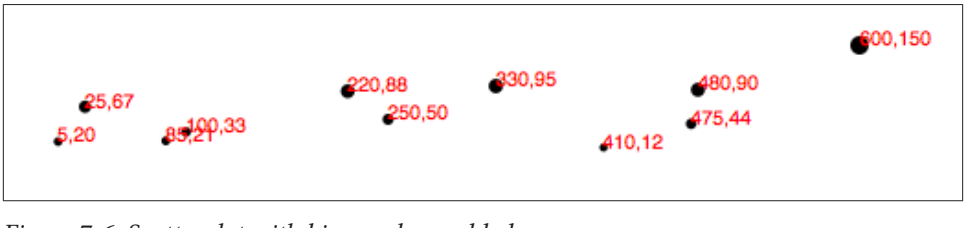


Figure 7-6. Scatterplot with big numbers added

Boom! Check out `06_scaled_plot_big.html`. Notice how all the old points maintained their relative positions, but have migrated closer together, down and to the left, to accommodate the newcomer in the top right corner.

And now, one final revelation: We can now very easily change the size of our SVG, and *everything scales accordingly*. Here, I've increased the value of `h` from `100` to `300` and made *no other changes*:

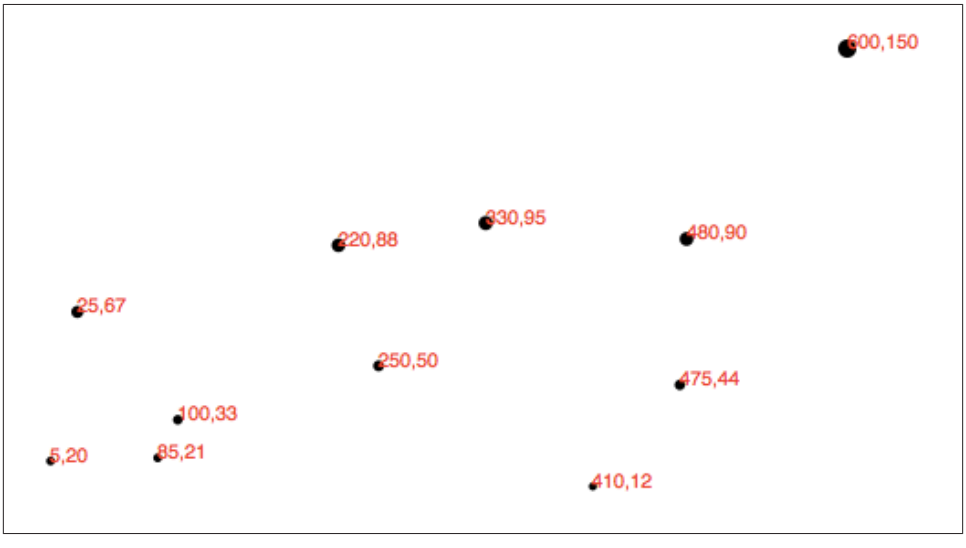


Figure 7-7. Large, scaled scatterplot

Boom, again! See `07_scaled_plot_large.html`. Hopefully, you are seeing this and realizing: No more late nights tweaking your code because the client decided the graphic should be 800 pixels wide instead of 600. Yes, you will get more sleep because of me (and D3's brilliant built-in methods). Being well-rested is a competitive advantage. You can thank me later.

Other Methods

`d3.scale.linear()` has several other handy methods that deserve a brief mention here:

- `nice()` — This tells the scale to take whatever input domain that you gave to `range()` and expand both ends to the nearest round value. From the D3 wiki: "For example, for a domain of [0.20147987687960267, 0.996679553296417], the nice domain is [0.2, 1]." This is useful for normal people, who are not computers and find it hard to read numbers like 0.20147987687960267.
- `rangeRound()` — Use `rangeRound()` in place of `range()`, and all values output by the scale will be rounded to the nearest whole number. This is useful if you want shapes to have exact pixel values, to avoid the fuzzy edges that may arise with antialiasing.

- `clamp()` — By default, a linear scale *can* return values outside of the specified range. For example, if given a value outside of its expected input domain, a scale will return a number also outside of the output range. Calling `.clamp(true)` on a scale, however, forces all output values to be within the specified range. Meaning, excessive values will be rounded to the range's low or high value (whichever is nearest).

Other Scales

In addition to `linear` scales (discussed above), D3 has several other scale methods built-in:

- `identity` — A 1:1 scale, useful primarily for pixel values
- `sqrt` — A square root scale
- `pow` — A power scale (good for the gym)
- `log` — A logarithmic scale
- `quantize` — A linear scale with discrete values for its output range, for when you want to sort data into "buckets"
- `quantile` — Similar to above, but with discrete values for its input domain (when you already have "buckets")
- `ordinal` — Ordinal scales use non-quantitative values (like category names) for output; perfect for comparing apples and oranges

Now that you have mastered the power of scales, it's time to express them visually as, yes, *axes*!

CHAPTER 8
Axes

Having mastered the use of D3 scales, we now have this scatterplot:

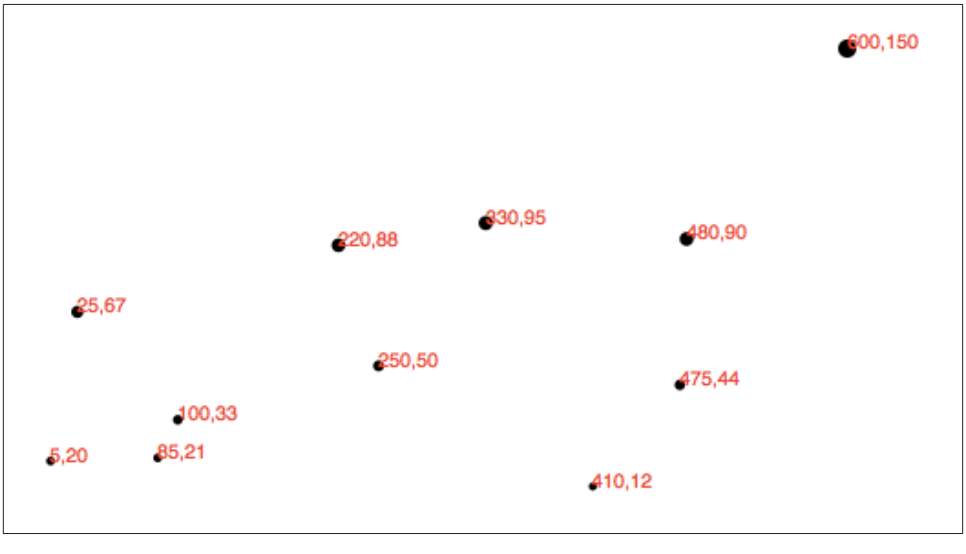


Figure 8-1. Large, scaled scatterplot

Let's add horizontal and vertical axes, so we can do away with the horrible red numbers cluttering up our chart.

Introducing Axes

Much like its scales, D3's *axes* are actually *functions* whose parameters you define. Unlike scales, when an axis function is called, it doesn't return a value, but generates the visual elements of the axis, including lines, labels, and ticks.

Note that the axis functions are SVG-specific, as they generate SVG elements. Also, axes are intended for use with quantitative scales (as opposed to ordinal ones).

Setting up an Axis

Use `d3.svg.axis()` to create a generic axis function:

```
var xAxis = d3.svg.axis();
```

At a minimum, each axis also needs to be told on what *scale* to operate. Here we'll pass in the `xScale` from the scatterplot code:

```
xAxis.scale(xScale);
```

We can also specify where the labels should appear relative to the axis itself. The default is `bottom`, meaning the labels will appear below the axis line. (Although this is the default, it can't hurt to specify it explicitly.) Possible orientations for horizontal axes are `top` and `bottom`. For vertical axes, use `left` and `right`.

```
xAxis.orient("bottom");
```

Of course, we can be more concise and string all this together into one line:

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

Finally, to actually generate the axis and insert all those little lines and labels into our SVG, we must *call* the `xAxis` function. This is similar to the scale functions, which we first configured by setting parameters, and then later *called*, to put them into action.

I'll put this code at the end of our script, so the axis is generated after the other elements in the SVG, and therefore appears "on top":

```
svg.append("g")
    .call(xAxis);
```

This is where things get a little funky. You may be wondering why this looks so different from our friendly scale functions. Here's why: Since an *axis* function actually draws something to the screen (by appending SVG elements to the DOM), we need to specify *where* in the DOM it should place those new elements. This is in contrast to scale functions like `xScale()`, for example, which calculate a value and return those values, typically for use by yet another function, without impacting the DOM at all.

So what we're doing with the code above is to first reference `svg`, the SVG image in the DOM. Then, we `append()` a new `g` element to the end of the SVG. In SVG-land, a `g` element is a *group* element. Group elements are invisible; unlike `line`, `rect`, and `circle`, and they have no visual presence themselves. Yet they help us in two ways: First, `g` elements can be used to contain (or "group") other elements, which keeps our code nice and tidy. Second, we can apply *transformations* to `g` elements, which affects how visual elements within that group (such as `line`s, `rect`s, and `circle`s) are rendered. We'll get to transformations in just a minute.

So we've created a new `g`, and then finally, the function `call()` is called on our new `g`. So what is `call()`, and who is it calling?

D3's `call()` function takes a *selection* as input and hands that selection off to any *function*. In this case, the selection is our new `g` group element. While the `g` isn't strictly necessary, we are using it because the axis function is about to generate lots of crazy lines and numbers, and frankly I just feel better keeping those things contained in one place. (Fun fact: Axes are real party animals, but respond well to discipline.) `call()` hands off `g` to the `xAxis` function, so our axis is generated *within* `g`.

If we were messy people who loved messy code, we could also rewrite the snippet above as this exact equivalent:

```
svg.append("g")
    .call(d3.svg.axis()
                .scale(xScale)
                .orient("bottom"));
```

See, you could cram the whole axis function within `call()`, but it's usually easier on our brains to define functions first, then call them later.

In any case, here's what that looks like. See code example `01_axes.html`.

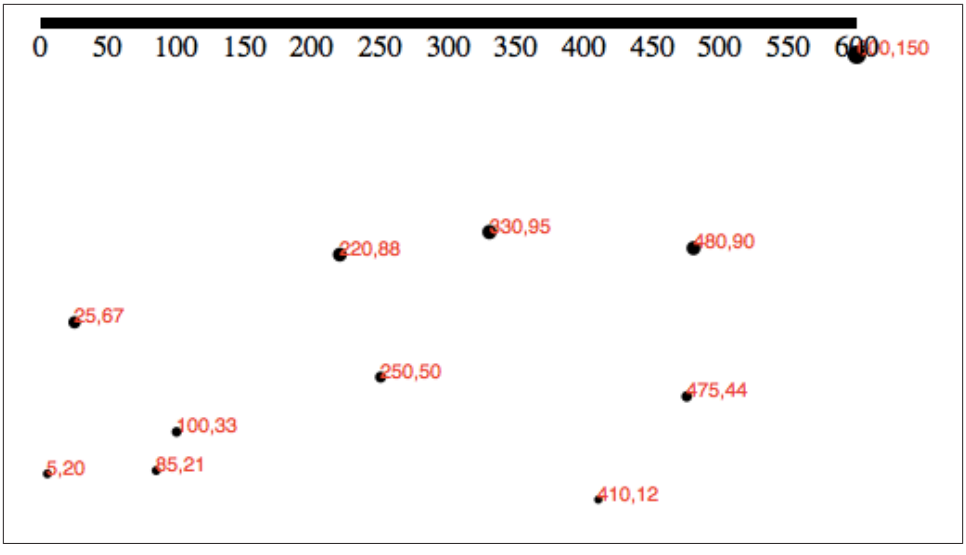


Figure 8-2. Simple, but ugly axis

Cleaning it Up

Technically, that is an axis, but it's neither pretty nor useful. To clean it up, let's first assign a class of `axis` to the new `g` element, so we can target it with CSS:

```
svg.append("g")
    .attr("class", "axis") //Assign "axis" class
    .call(xAxis);
```

Then, we introduce our first CSS styles, up in the `<head>` of our page:

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

.axis text {
    font-family: sans-serif;
    font-size: 11px;
}
```

See how useful it is to group all the axis elements within one `g` group? Now we can very easily apply styles to anything within the group the simple CSS selector `.axis`. The axes themselves are made up of `path`, `line`, and `text` elements, so those are the three elements that we target in our CSS. The `path`s and `line`s can be styled with the same rules, while `text` gets its own rules around font and font size.

You may notice that when we use CSS rules to style SVG elements, only SVG attribute names — not regular CSS properties — should be used. This is confusing because many properties share the same names in both CSS and SVG, but some do not. For example, in regular CSS, to set the color of some text, you would use the `color` property, as in:

```
p {
    color: olive;
}
```

That will set the text color of all `p` paragraphs to be `olive`. But try to apply this property to an SVG element, as with

```
text {
    color: olive;
}
```

and it will have no effect, because `color` is not a property recognized by SVG. Instead, you must use SVG's equivalent, `fill`:

```
text {
    fill: olive;
}
```

If you ever find yourself trying to style SVG elements, but for some reason *the stupid CSS code just isn't working*, may I suggest you take a deep breath, pause, and then review your *property names* very closely to ensure you're using SVG names, not CSS ones.

The `shape-rendering` property is another weird SVG attribute you should know. We use it here to make sure our axis and its tick mark lines are pixel-perfect. No blurry axes for us!

Here's what the chart looks like after our CSS clean-up:

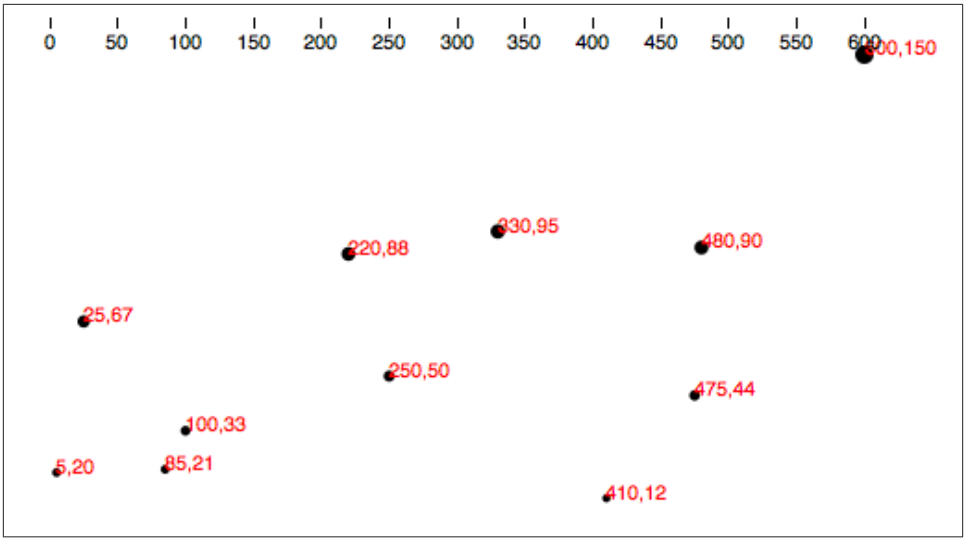


Figure 8-3. Cleaner axis

That's better, but the top horizontal line of the axis is cut off, and the axis itself should be down at the base of the chart anyway. Here's where SVG *transformations* come in. By adding one line of code, we can `transform` the entire axis group, pushing it to the bottom:

```
svg.append("g")
    .attr("class", "axis")
    .attr("transform", "translate(0," + (h - padding) + ")")
    .call(xAxis);
```

Note that we use `attr()` to apply `transform` as an attribute of `g`. SVG transforms are quite powerful, and can accept several different kinds of transform definitions, including scales and rotations. But we are keeping it simple here with only a *translation* transform, which simply pushes the whole `g` group over and down by some amount.

Translation transforms are specified with the easy syntax of `translate(x,y)`, where `x` and `y` are, obviously, the number of horizontal and vertical pixels by which to translate the element. So, in the end, we would like our `g` to look like this in the DOM:

```
<g class="axis" transform="translate(0,280)")
```

As you can see, the `g.axis` isn't moved horizontally at all, but it is pushed 280 pixels down, conveniently to the base of our chart. We specify as much in this line of code:

```
    .attr("transform", "translate(0," + (h - padding) + ")")
```

Note the use of `(h - padding)`, so the group's top edge is set to `h`, the height of the entire image, minus the `padding` value we created earlier. `(h - padding)` is calculated to be `280`, and then connected to the rest of the string, so the final transform property value is `translate(0,280)`.

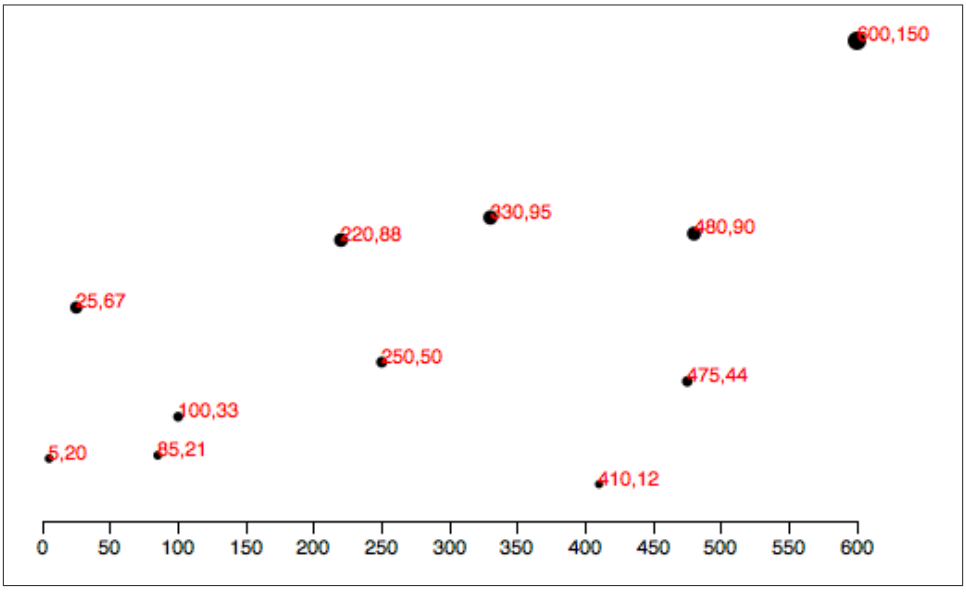


Figure 8-4. Nice, clean axis

Much better! Check out the code so far in `02_axes_bottom.html`.

Check for Ticks

Some ticks spread disease, but D3's ticks communicate information. Yet more ticks are not necessarily better, and at a certain point they begin to clutter your chart. You'll notice that we never specified how many ticks to include on the axis, nor at what intervals they should appear. Without clear instruction, D3 has auto-magically examined our scale `xScale` and made informed judgements about how many ticks to include, and at what intervals (every 50, in this case).

As you would expect, you can customize all aspects of your axes, starting with the rough number of ticks, using `ticks()`:

```
var xAxis = d3.svg.axis()
                  .scale(xScale)
                  .orient("bottom")
                  .ticks(5);  //Set rough # of ticks
```

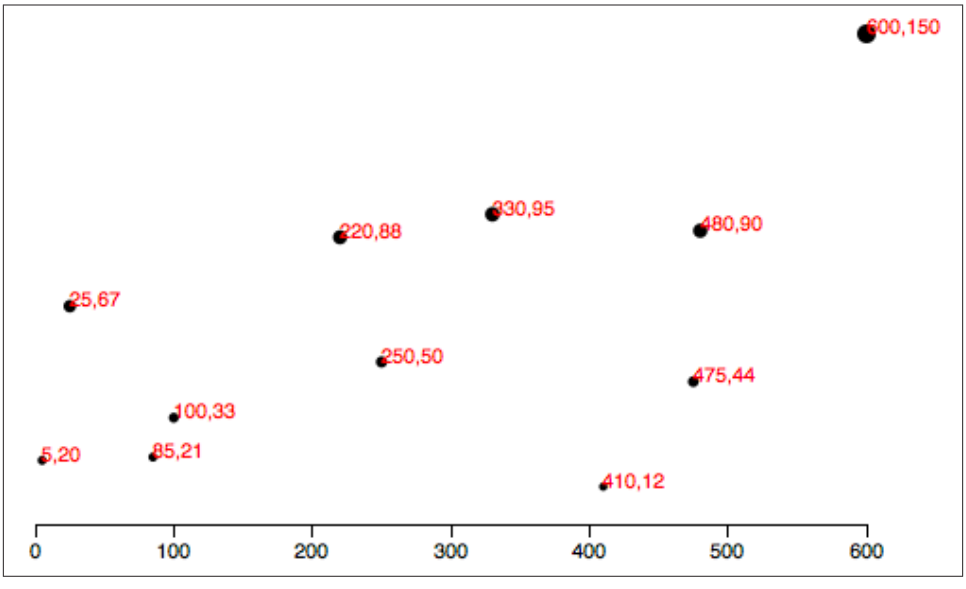


Figure 8-5. Fewer ticks

See `03_axes_clean.html` for that code.

You probably noticed that, while we specified only five ticks, D3 has made an executive decision and ordered up a total of seven. That's because D3 has got your back, and figured out that including only *five* ticks would require slicing the input domain into less-than-gorgeous values — in this case, 0, 150, 300, 450, and 600. D3 inteprets the `ticks()` value as merely a suggestion, and will override your suggestion with what it determines to be the most clean and human-readable values — in this case, intervals of 100 — even when that requires including slightly more or fewer ticks than you requested. This is actually a totally brilliant feature that increases the scalability of your design; as the data set changes, and the input domain expands or contracts (bigger numbers or smaller numbers), D3 ensures that the tick labels remaing clear and easy to read.

Y Not?

Time to label the vertical axis! By copying and tweaking the code we already wrote for the `xAxis`, we add this near the top of of our code

```
//Define Y axis
var yAxis = d3.svg.axis()
                  .scale(yScale)
                  .orient("left")
                  .ticks(5);
```

and this, near the bottom:

```
//Create Y axis
svg.append("g")
    .attr("class", "axis")
    .attr("transform", "translate(" + padding + ",0)")
    .call(yAxis);
```

Note that the labels will be oriented `left` and that the `yAxis` group `g` is translated to the right by the amount `padding`.

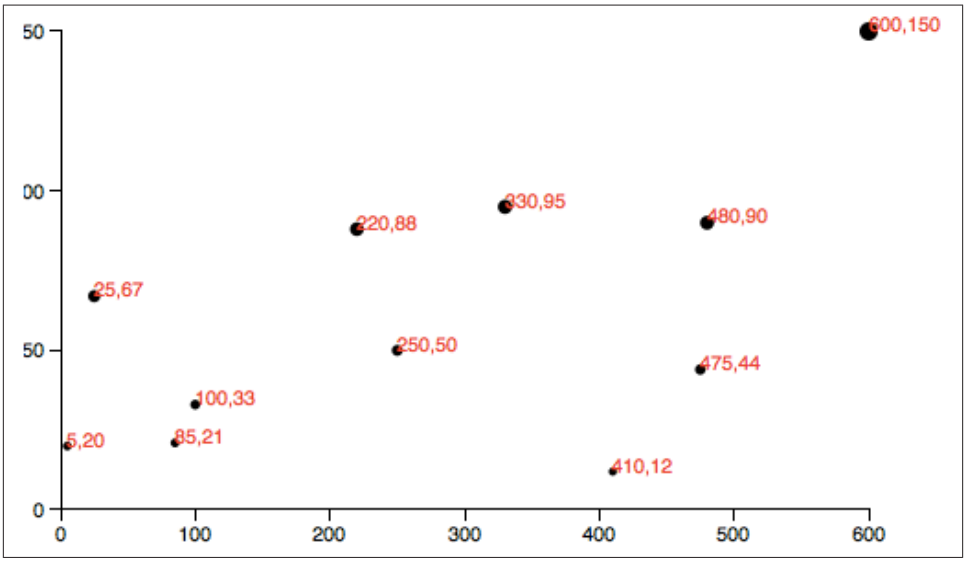


Figure 8-6. Initial Y axis

This is starting to look like a real chart! But the `yAxis` labels are getting cut off. To give them more room on the left side, I'll bump up the value of `padding` from 20 to 30:

```
var padding = 30;
```

Of course, you could also introduce separate `padding` variables for each axis, say `xPadding` and `yPadding`, for more control over the layout.

See the updated code in `04_axes_y.html`. Here's what it looks like:

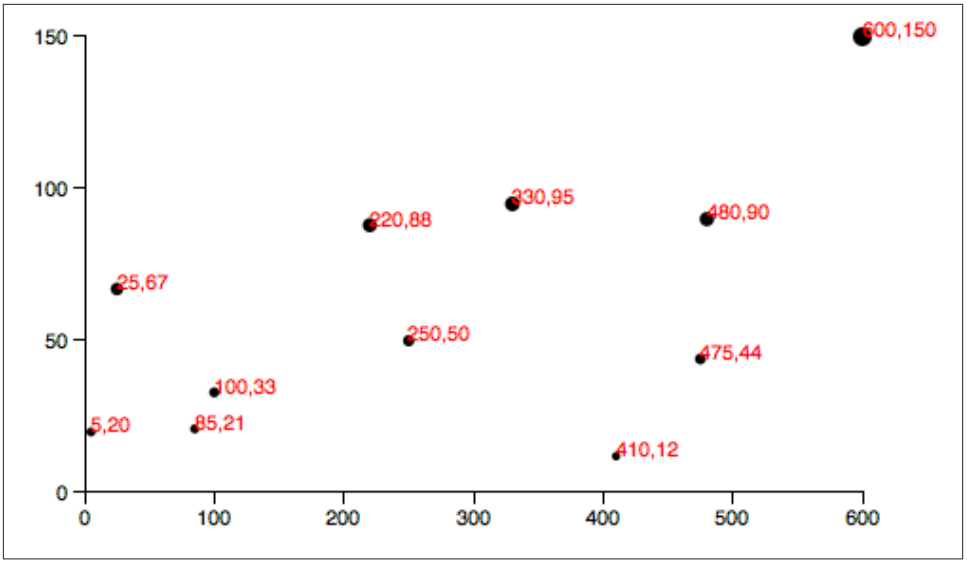


Figure 8-7. Scatterplot with Y axis

Final Touches

I appreciate that so far you have been very quiet and polite, and not at all confrontational. Yet I still feel as though I have to win you over. So to prove to you that our new axis are dynamic and scalable, I'd like to switch from using a static data set to using randomized numbers:

```
//Dynamic, random dataset
var dataset = [];
var numDataPoints = 50;
var xRange = Math.random() * 1000;
var yRange = Math.random() * 1000;
for (var i = 0; i < numDataPoints; i++) {
    var newNumber1 = Math.round(Math.random() * xRange);
    var newNumber2 = Math.round(Math.random() * yRange);
    dataset.push([newNumber1, newNumber2]);
}
```

This code initializes an empty array, then loops through 50 times, chooses two random numbers each time, and adds ("pushes") that pair of values to the `dataset` array.

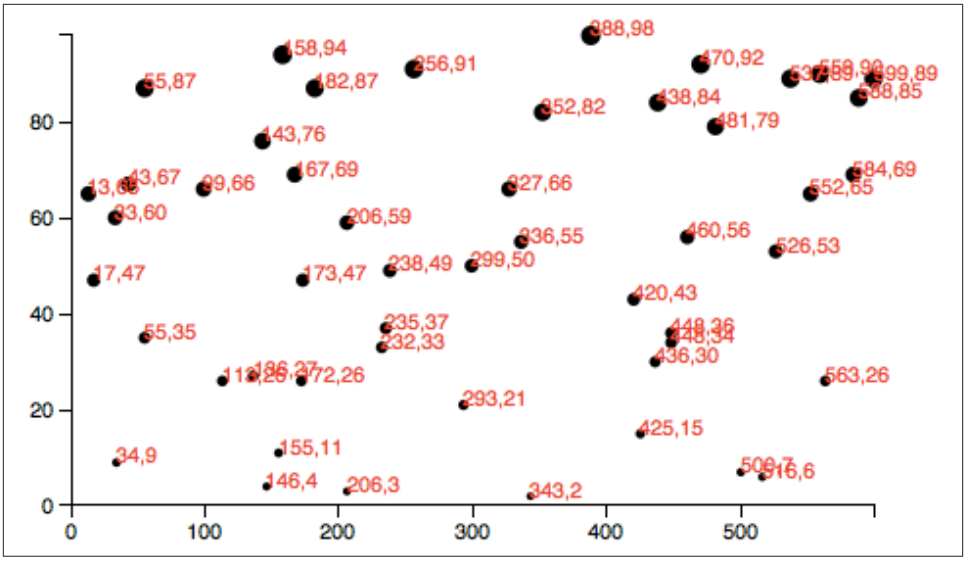


Figure 8-8. Scatterplot with random data

Try out that randomized data set code in `05_axes_random.html`. Each time you reload the page, you'll get different data values. Notice how both axes scale to fit the new domains, and ticks and label values are chosen accordingly.

Having made my point, I think we can finally cut those horrible, red labels, by commenting out the relevant lines of code:

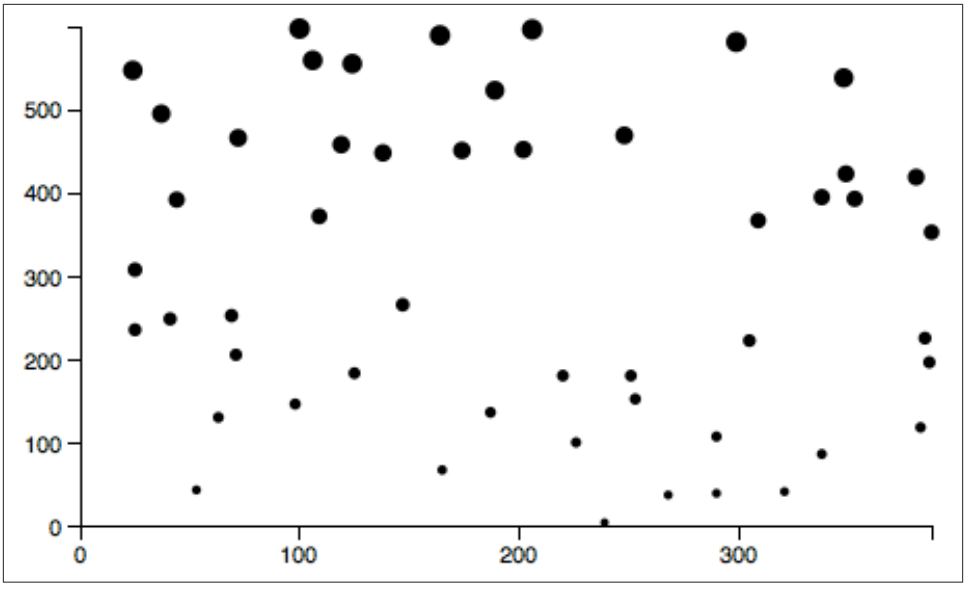


Figure 8-9. Scatterplot with random data and no red labels

Our final scatterplot code lives in `06_axes_no_labels.html`.

Formatting Tick Labels

One last thing: So far, we've been working with integers — whole numbers — which are nice and easy. But data is often messier, and in those cases, you may want more control over how the axis labels are formatted. Enter `tickFormat()`, which enables you to specify how your numbers should be formatted. For example, you may want to include three places after the decimal point, or display values as percentages, or both.

To use `tickFormat()`, first define a new number-formatting function. This one, for example, says to treat values as percentages with one decimal point precision. That is, if you give this function the number `0.23`, it will return the string `23.0%`. (See the reference entry for d3.format() for more options.)

```
var formatAsPercentage = d3.format(".1%");
```

Then, tell your axis to use that formatting function for its ticks, e.g.:

```
xAxis.tickFormat(formatAsPercentage);
```

Development tip: I find it easiest to test these formatting functions out in the JavaScript console. For example, just open any page that loads D3, such as `06_axes_no_labels.html`, and type your format rule into the console. Then test it by feeding it a value, as you would with any other function:

```
> var formatAsPercentage = d3.format(".1%");
  undefined
> formatAsPercentage(0.54321)
  "54.3%"
>
```

Figure 8-10. Testing `format()` in the console

You can see here that a data value of `0.54321` is converted to `54.3%` for display purposes — perfect!

You can play with that code in `07_axes_format.html`. Obviously, a percentage format doesn't make sense with our scatterplot's current data set, but as an exercise, you could try tweaking how the random numbers are generated, to make more appropriate, non-whole number values, or just experiment with the format function itself.

CHAPTER 9

Updates, Transitions, and Motion

Until this point, we have used only static data sets. But real-world data almost always *changes* over time. And you may want your visualization to reflect those changes.

In D3 terms, those changes are handled by *updates*. The visual adjustments are made pretty with *transitions*, which can employ *motion* for perceptual benefit.

We'll start by generating a visualization with one data set, and then changing the data completely.

Modernizing the Bar Chart

Let's revisit our trusty old bar chart.

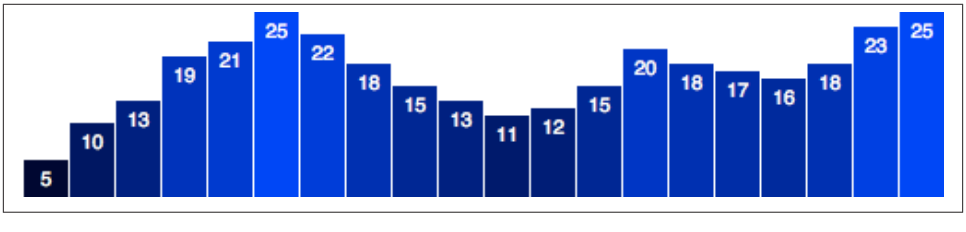


Figure 9-1. The bar chart, as seen last

If you examine the code in `01_bar_chart.html`, you'll see that we used this static dataset:

```
var dataset = [ 5, 10, 13, 19, 21, 25, 22, 18, 15, 13,
                11, 12, 15, 20, 18, 17, 16, 18, 23, 25 ];
```

Since then, we've learned how to write more flexible code, so our chart elements resize to accommodate different sized data sets (meaning, shorter or longer arrays) and different data values (smaller or larger numbers). We accomplished that flexibility using D3 scales, so I'd like to start by bringing our bar chart up to speed.

Ready? Okay, just give me a sec…

Aaaaaand, done! Thanks for waiting.

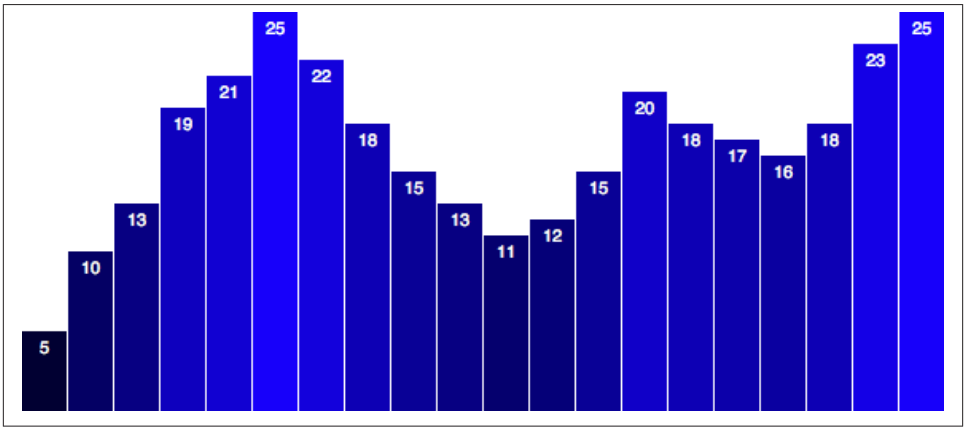


Figure 9-2. A scalable, flexible bar chart

This looks pretty similar, but a lot has changed under the hood. You can follow along by opening up `02_bar_chart_with_scales.html`.

To start, I adjusted the width and height, to make the chart taller and wider:

```
var w = 600;
var h = 250;
```

Next, I introduced an *ordinal scale* to handle the left/right positioning of bars and labels along the x axis.

```
var xScale = d3.scale.ordinal()
                .domain(d3.range(dataset.length))
                .rangeRoundBands([0, w], 0.05);
```

This may seem like gobbledegook, so I'll walk through it one line at a time.

Ordinal Scales, Explained

First, in this line

```
var xScale = d3.scale.ordinal()
```

we declare a new variable called `xScale`, just as we had done with our scatterplot. Only here, instead of a *linear* scale, we create an *ordinal* one. Ordinal scales are typically used for ordinal data, typically categories with some inherent *order* to them, such as:

- freshman, sophomore, junior, senior
- grade B, grade A, grade AA
- strongly dislike, dislike, neutral, like, strongly like

We don't have true ordinal data for use with this bar chart. Instead, we just want the bars to be drawn from left to right using the same order in which values occur in our data set. D3's ordinal scale is useful in this situation, when we have many visual elements (vertical bars) that are positioned in an arbitrary order (left to right), but must be evenly spaced. This will become clear in a moment.

```
                .domain(d3.range(dataset.length))
```

This next line of code sets the input domain for the scale. Remember how linear scales need a two-value array to set their domains, as in `[0, 100]`? For a linear scale, that array would set the low and high values of the domain. But ordinal domains are, well, ordinal, so they don't think in linear, quantitative terms. To set the domain of an ordinal scale, you typically specify an array with the category names, as in:

```
.domain(["freshman", "sophomore", "junior", "senior"])
```

For our bar chart, we don't have explicit categories, but we could assign each data point or bar an ID value corresponding to its position within the `dataset` array, as in 0, 1, 2, 3, and so on. So perhaps our domain statement could read:

```
.domain([0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

It turns out there is a very simple way to quickly generate an array of sequential numbers: the `d3.range()` method.

While viewing `02_bar_chart_with_scales.html`, go ahead and open up the console, and type in:

```
d3.range(10)
```

You should see the output array:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

How nice is that? D3 saves you time once again (and the hassle of extra `for()` loops).

Coming back to our code, it should now be clear what's happening here:

```
                .domain(d3.range(dataset.length))
```

1. `dataset.length`, in this case, is evaluted as `20`, since we have 20 items in our dataset.

2. `d3.range(20)` is then evaluated, which returns this array: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]`
3. Finally, `domain()` sets the domain of our new ordinal scale to those “”

This may be somewhat confusing because we are using numbers (0, 1, 2…) as ordinal values, but ordinal values are typically non-numeric.

Round Bands Are All the Range These Days

The great thing about `d3.scale.ordinal()` is it supports *range banding*. Instead of returning a continuous range, as any quantitative scale (like `d3.scale.linear()`) would, ordinal scales use *discrete* ranges, meaning the output values are determined in advance, and could be numeric or not.

We could use `range()` like everyone else, *or* we can be smooth and use `rangeBands()`, which takes a low and high value, and automatically divides it into even chunks or “bands,” based on the length of the domain. For example,

```
.rangeBands([0, w])
```

this says “calculate even bands starting at 0 and ending at w, then set this scale’s range to those bands.” In our case, we specified 20 values in the domain, so D3 will calculate:

```
(w - 0) / xScale.domain().length
(600 - 0) / 20
600 / 20
30
```

In the end, each band will be `30` “wide.”

It is also possible to include a second parameter, which includes a bit of spacing between each band. Here I’ve used `0.2`, meaning that 20% of the width of each band will be used for spacing in between bands.

```
.rangeBands([0, w], 0.2)
```

We can be even smoother and use `rangeRoundBands()`, which is the same as `rangeBands()`, except that the output values are rounded to the nearest whole pixel, so 12.3456 becomes just 12, for example. This is helpful for keeping visual elements lined up precisely on the pixel grid, for clean, sharp edges.

So, our final line of code in that statement is:

```
.rangeRoundBands([0, w], 0.05);
```

This gives us nice, clean pixel values, with a teensy bit of visual space between them.

Referencing the Ordinal Scale

Later in the code (and I recommend viewing the source), when we create the `rect` elements, we set their horizontal, x axis positions like so:

```
//Create bars
svg.selectAll("rect")
   .data(dataset)
   .enter()
   .append("rect")
   .attr("x", function(d, i) {
        return xScale(i);         // <-- Set x values
   })
   …
```

Note that, since we include `d` and `i` as parameters to the anonymous function, D3 will automatically pass in the correct values. Of course, `d` is the current datum, and `i` is its index value. So `i` will be passed 0, 1, 2, 3, and so on.

Coincidentally (hmmm!), we used those same values (0, 1, 2, 3…) for our ordinal scale's input domain. So when we call `xScale(i)`, `xScale()` will look up the ordinal value `i` and return its associated output (band) value. (You can verify all this for yourself in the console. Just try typing `xScale(0)` or `xScale(5)`.)

Even better, setting the widths of these bars just got a lot easier. Before using the ordinal scale, we had:

```
.attr("width", w / dataset.length - barPadding)
```

We don't even need `barPadding` anymore, since now the padding is built into `rangeRoundBands()`. Setting the width of each `rect` needs only this:

```
.attr("width", xScale.rangeBand())
```

Nice! Isn't it nice when D3 does the math for you?

Other Updates

I've made several other updates in `02_bar_chart_with_scales.html`, including creating a new linear scale `yScale` to calculate vertical values. You already know how to use linear scales, so you can skim the source and note how that's being used to set the bar heights.

Updating Data

Okay, once again, we have this amazing bar chart, flexible enough to handle any data we can throw at it.

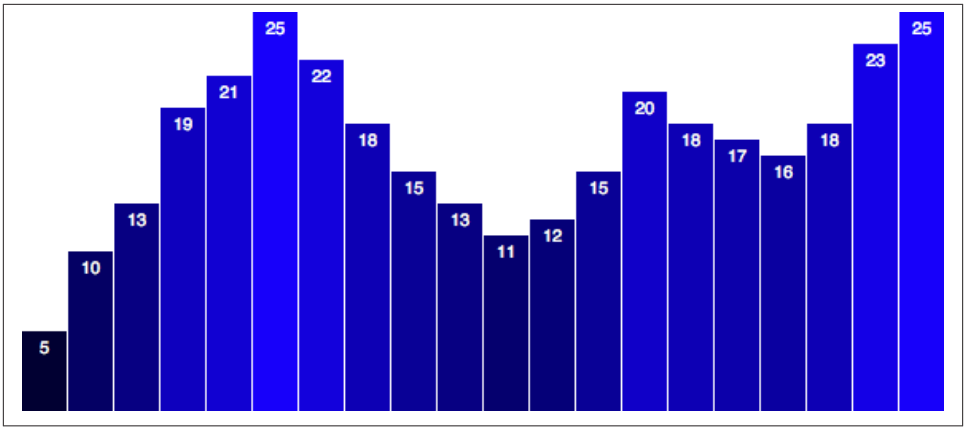


Figure 9-3. The bar chart

Or is it? Let's see.

The simplest kind of update is when all data values are updated at the same time *and* the number of values stays the same.

The basic approach in this scenario is:

1. Modify the values in your data set
2. Re-bind the new values to the existing elements (thereby overwriting the original values)
3. Set new attribute values as needed to update the visual display

Before any of those steps can happen, though, some *event* needs to kick things off. So far, all of our code has executed immediately upon page load. We *could* have our update run right after the initial drawing code, but it would happen imperceptibly fast. To make sure we can observe the change as it happens, we will separate our update code from everything else. We will need a "trigger," something that happens *after* page load to apply the updates. How about a mouse click?

Interaction via Event Listeners

Any DOM element can be used, so rather than design a fancy button, I'll add a simple `p` paragraph to the HTML's `body`:

```
<p>Click on this text to update the chart with new data values (once).</p>
```

Then, down at the end of our D3 code, let's add:

```
d3.select("p")
    .on("click", function() {
        //Do something  on click
    });
```

This selects our new `p`, and then adds an *event listener* to that element. Huh?

In JavaScript, events are happening all the time. Not exciting events, like huge parties, but really insignificant events like `mouseover` and `click`. Most of the time, these insignificant events go ignored (just as in life, perhaps). But if someone is *listening*, then the event will be *heard*, and can go down in posterity, or at least trigger some sort of DOM interaction. (Rough JavaScript parallel of classic koan: If an event occurs, and no listener hears it, did it ever happen at all?)

An event *listener* is an anonymous function that *listens* for a specific event *on* a specific element or elements. D3's method `selection.on()` provides a nice shorthand for adding event listeners. As you can see, `on()` takes two arguments: the event *type* (`"click"`) and the listener itself (the anonymous function).

In this case, the listener listens for a `click` event occurring on our selection `p`. When that happens, the listener function is executed. You can put whatever code you want in between the brackets of the anonymous function:

```
d3.select("p")
    .on("click", function() {
        //Do something mundane and annoying on click
        alert("Hey, don't click that!");
    });
```

We'll talk a lot more about interactivity in the next chapter.

Changing the Data

Instead of generating annoying pop-ups, I'd rather simply update `dataset` by overwriting its original values. This is Step 1, from above.

```
dataset = [ 11, 12, 15, 20, 18, 17, 16, 18, 23, 25,
            5, 10, 13, 19, 21, 25, 22, 18, 15, 13 ];
```

Step 2 is to re-bind the new values to the existing elements. We can do that by selecting those `rect`s and simply calling `data()` one more time:

```
svg.selectAll("rect")
   .data(dataset);     //New data successfully bound, sir!
```

Updating the Visuals

Finally, Step 3 is to update the visual attributes, referencing the (now-updated) data values. This is super easy, as we simply copy and paste the relevant code that we've already written. In this case, the `rect`s can maintain their horizontal positions and widths; all we really need to update are their `height`s and `y` positions. I've added those lines below:

```
svg.selectAll("rect")
   .data(dataset)
   .attr("y", function(d) {
        return h - yScale(d);
   })
   .attr("height", function(d) {
        return yScale(d);
   });
```

Notice this looks almost exactly like the code that generates the `rect`s initially, only without `enter()` and `append()`.

Putting it all together, here is all of our update code in one place:

```
//On click, update with new data
d3.select("p")
    .on("click", function() {

        //New values for dataset
        dataset = [ 11, 12, 15, 20, 18, 17, 16, 18, 23, 25,
                    5, 10, 13, 19, 21, 25, 22, 18, 15, 13 ];

        //Update all rects
        svg.selectAll("rect")
           .data(dataset)
           .attr("y", function(d) {
                return h - yScale(d);
           })
           .attr("height", function(d) {
                return yScale(d);
           });

    });
```

Check out the revised bar chart in `03_updates_all_data.html`! Here's how it looks to start:

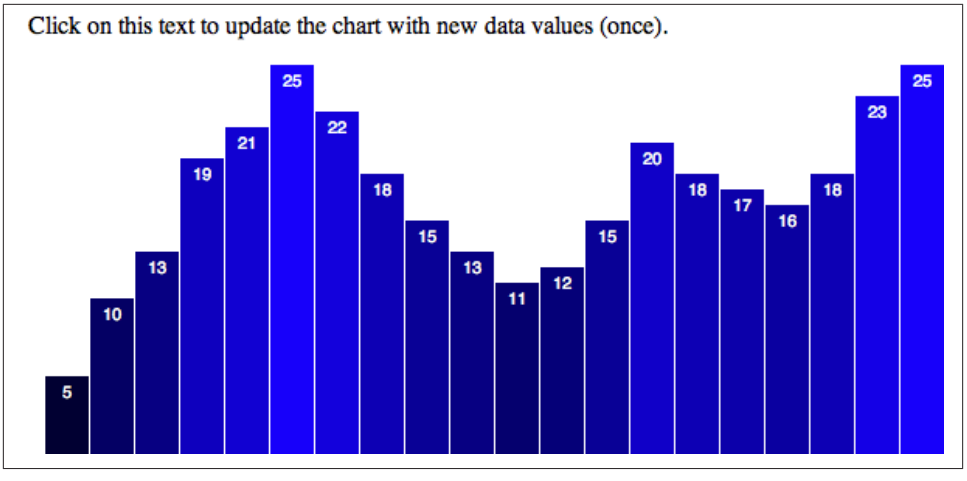


Figure 9-4. Updateable bar chart

Then click anywhere on the paragraph, and it turns into this:

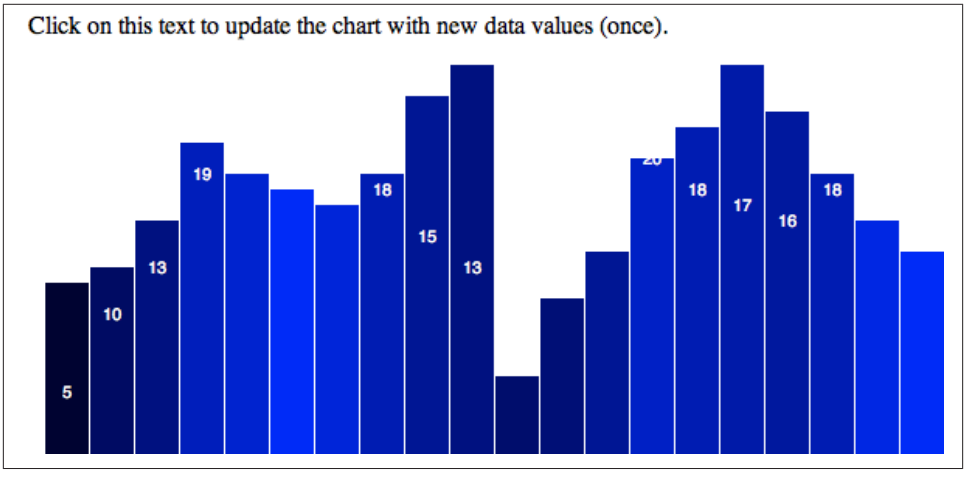


Figure 9-5. Bar chart, data updated

Good news: The values in `dataset` were modified, re-bound, and used to adjust the `rect`s. Bad news: It looks weird, because we forgot to update the labels, and also the bar colors. Good news (again, because I always like to end with good news): This is easy to fix!

To ensure the colors change on update, we just copy and paste in the line where we set the `fill`:

```
svg.selectAll("rect")
   .data(dataset)
   .attr("y", function(d) {
        return h - yScale(d);
   })
   .attr("height", function(d) {
        return yScale(d);
   })
   .attr("fill", function(d) {   // <-- Down here!
        return "rgb(0, 0, " + (d * 10) + ")";
   });
```

To update the labels, we use a similar pattern, only here we adjust their text content and x/y values:

```
svg.selectAll("text")
   .data(dataset)
   .text(function(d) {
        return d;
   })
   .attr("x", function(d, i) {
        return xScale(i) + xScale.rangeBand() / 2;
   })
   .attr("y", function(d) {
        return h - yScale(d) + 14;
   });
```

Take a look at `04_updates_all_data_fixed.html`. You'll notice it looks the same to start, but click the trigger and now the bar colors and labels update correctly!

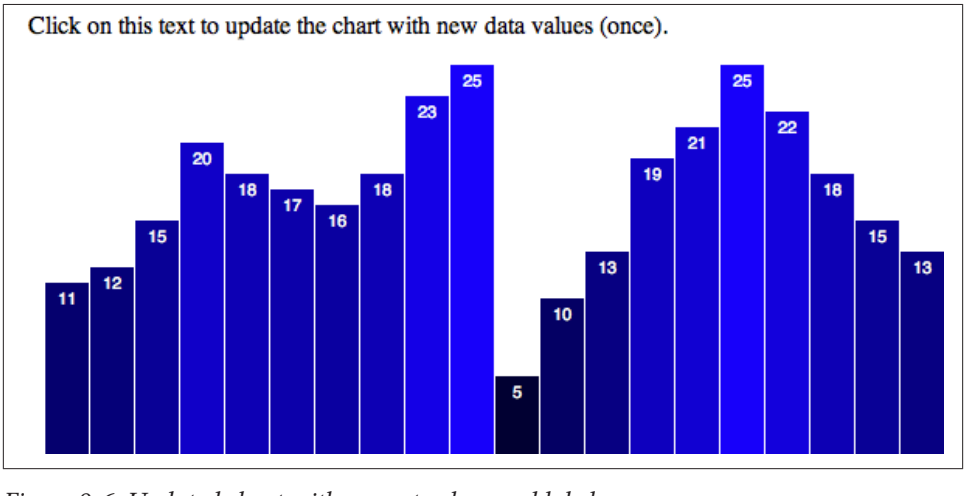


Figure 9-6. Updated chart with correct colors and labels

Very nice!

Transitions

Life transitions can be scary: the first day of school, moving to a new city, quitting your day job to do freelance data visualization full-time. But D3 transitions are fun, beautiful, and not at all emotionally taxing.

Making a nice, super smooth, animated transition is as simple as adding one line of code:

```
.transition()
```

Specifically, add this link in the chain below where your selection is made, and *above* where any attribute changes are applied.

```
//Update all rects
svg.selectAll("rect")
   .data(dataset)
   .transition()    // <-- This is new! Everything else here is unchanged.
   .attr("y", function(d) {
        return h - yScale(d);
   })
   .attr("height", function(d) {
        return yScale(d);
   })
   .attr("fill", function(d) {
        return "rgb(0, 0, " + (d * 10) + ")";
   });
```

Now run the code in `05_transition.html`, and click the text to see the transition in action. Note that the end result looks the same visually, but the transition from the chart's initial state to its end state is much, much nicer.

Isn't that *insane*? I'm not a psychologist, but I believe it is literally insane that we can add a single line of code, and D3 will *animate* our value changes for us over time.

Without `transition()`, D3 evaluates every `attr()` statement immediately, so the changes in height and fill happen right away. When you add `transition()`, D3 introduces the element of time. Rather than applying new values all at once, D3 *interpolates* between the old values and the new values, meaning it normalizes the beginning and ending values, and calculates all their in-between states. D3 is also smart enough to recognize and interpolate between different attribute value formats. For example, if you specified a height of `200px` to start, but transition to just `100` (without the `px`). Or if a `blue` fill turns `rgb(0,255,0)`. You don't need to fret about being consistent; D3 takes care of it.

Do you believe me yet? This is really insane. And super helpful.

duration(), or How Long is This Going to Take?

So the `attr()` values are interpolated over time, but how *much* time? It turns out the default is 250 milliseconds, or one-quarter second. (1000 milliseconds = 1 second.) That's why the transition in `05_transition.html` is so fast.

Fortunately, you can control how much time is spent on any transition by — again, I kid you not — adding a single line of code:

```
.duration(1000)
```

The `duration()` must be specified *after* the `transition()`, but before the `attr()` statements to which the transition applies. And durations are always specified in milliseconds, so `duration(1000)` is a one-second duration.

Here is that line in context:

```
//Update all rects
svg.selectAll("rect")
   .data(dataset)
   .transition()
   .duration(1000)  // <-- Now this is new!
   .attr("y", function(d) {
        return h - yScale(d);
   })
   .attr("height", function(d) {
        return yScale(d);
   })
   .attr("fill", function(d) {
        return "rgb(0, 0, " + (d * 10) + ")";
   });
```

Open up `06_duration.html` and see the difference. Now make a copy of that file, and try plugging in some different numbers to slow or speed the transition. For example, try `3000` for a three-second transition, or `100` for one-tenth of a second.

The actual durations you choose will depend on the context of your design and what triggers the transition. In practice, I find that transitions of around 150ms are useful for providing minor interface feedback (such as hovering over elements), while about 1000ms is ideal for many more significant visual transitions, such as switching from one view of the data to another. 1000ms (one second) is not too long, not too short.

In case you're feeling lazy, I made `07_duration_slow.html`, which uses `5000` for a five-second transition.

With such a slow transition, it becomes obvious that the value labels are not transitioning smoothly along with the bar heights. As you now know, we can correct that oversight by adding only two new lines of code, this time in the section where we update the labels:

```
//Update all labels
svg.selectAll("text")
```

```
    .data(dataset)
    .transition()        // <-- This is new,
    .duration(5000)      //     and so is this.
    .text(function(d) {
         return d;
    })
    .attr("x", function(d, i) {
         return xScale(i) + xScale.rangeBand() / 2;
    })
    .attr("y", function(d) {
         return h - yScale(d) + 14;
    });
```

Much better! Note in `08_duration_slow_labels_fixed.html` how the labels now animate smoothly along with the bars

ease()-y Does It

With a 5000ms, slow-as-molasses transition, we can also perceive the *quality* of motion. In this case, notice how the animation begins very slowly, then accelerates, then slows down again as the bars reach their destination heights. That is, the rate of motion is not *linear*, but *variable*.

The quality of motion used for a transition is called *easing*. In animation terms, we think about elements *easing* into place, moving from here to there.

With D3, you can specify different kinds of easing by using `ease()`. The default easing is `"cubic-in-out"`, which produces the gradual acceleration and deceleration we see in our chart. It's a good default, because you generally can't go wrong with a nice, smooth transition.

Contrast that smoothness to `09_ease_linear.html`, which uses `ease("linear")` to specify a linear easing function. Notice how the rate of motion is constant. That is, there is no gradual acceleration and deceleration — the elements simply begin moving at an even pace, and then they stop abruptly. (Also, I lowered the duration down to 2000ms.)

`ease()` must also be specified after `transition()`, but before the `attr()` statements to which the transition applies. `ease()` can come before or after `duration()`, but this sequence makes the most sense to me:

```
…   //Selection statement(s)
.transition()
.duration(2000)
.ease("linear")
…   //attr() statements
```

Fortunately, there are several other built-in easing functions to choose from. Some of my favorites are:

`circle`: Gradual ease in and acceleration until elements snap into place

`elastic`: The best way to describe this one is "sproingy"

`bounce`: Like a ball bouncing, then coming to rest

Sample code files 10, 11, and 12 illustrate the above. The complete list of easing functions is on the D3 wiki.

Wow, I already regret telling you about `bounce`. Please only use `bounce` if you are making a satirical inforgraphic that is mocking other bad graphics. Pereceptually, I don't think there is a real case to be used for `bounce` easing in visualization. `cubic-in-out` is the default for a reason.

Please Do Not delay()

While `ease()` controls the quality of motion, `delay()` specifies when the transition begins.

`delay()` can be given a static value, also in milliseconds, as in:

```
…
.transition()
.delay(1000)     //1000ms or 1 second
.duration(2000)  //2000ms or 2 seconds
…
```

As with `duration()` and `ease()`, the order here is somewhat flexible, but I like to include `delay()` before `duration()`. That makes more sense to me, since the delay happens first, followed by the transition itself.

See `13_delay_static.html`, in which clicking the text triggers first a 1000ms delay (in which nothing happens), followed by a 2000ms transition.

Static delays can be useful, but more exciting are delay values that we calculate dynamically. A common use of this is to generate staggered delays, so some elements transition before others. Staggered delays can assist with perception, as it's easier for our eyes to follow an individual element's motion when it is slightly out of sync with its neighboring elements' motion.

To do this, instead of giving `delay()` a static value, we give it a function, in typical D3 fashion:

```
…
.transition()
.delay(function(d, i) {
    return i * 100;
})
.duration(500)
…
```

Just as we've seen with other D3 methods, when given an anonymous function, the datum bound to the current element is passed into `d`, while the index position of that element is passed into `i`. So, in this case, as D3 loops through each element, the delay for each element is set to `i * 100`, meaning each subsequent element will be delayed 100ms *more* than the preceding element.

All this is to say that we now have staggered transitions. Check out the beautifully animated bars in `14_delay_dynamic.html` and see for yourself.

Note that I also decreased the duration to 500ms to make it feel a bit snappier. Also note that `duration()` sets the duration for each *individual transition*, not for all transitions in aggregate. So, for example, if 20 elements have 500ms transitions applied with no delay, then it will all be over in 500ms, or one-half second. But if a 100ms delay is applied to each subsequent element (`i * 100`), then the total running time of all transitions will be 2400ms:

```
Max value of i times 100ms delay plus 500ms duration =
19 * 100 + 500 =
2400
```

Since these delays are being calculated on a per-element basis, if you added more data, then the total running time of all transitions will increase. This is something to keep in mind if you have a dynamically loaded data set whose array length is variable. If you suddenly loaded 10,000 data points instead of 20, you could spend a long, long time watching those bars wiggle around (1,000,400 seconds or 11.5 days to be precise). Suddenly, they're not so cute anymore.

Fortunately, we can *scale* our delay values dynamically to the length of the data set. This isn't fancy D3 stuff; it's just math.

See `15_delay_dynamic_scaled.html`, in which 30 values are included in the dataset. If you get out your stopwatch, you'll see that the total transition time is 1.5 seconds, or around 1500ms.

Now see `16_delay_dynamic_scaled_fewer.html`, which uses exactly the same transition code, but with only 10 data points. Notice how the delays are slightly longer (well, 200% longer), so the total transition time is the same: 1.5 seconds! How is this possible?

```
…
.transition()
.delay(function(d, i) {
    return i / dataset.length * 1000;   // <-- Where the magic happens
})
.duration(500)
…
```

The two sample pages above use the same delay calculations here. Instead of multiplying `i` by some static amount, we first divide `i` by `dataset.length`, in effect normalizing the

value. Then, that normalized value is multiplied by `1000`, or 1 second. The result is that the maximum amount of delay for the last element will be `1000`, and all prior elements will be delayed by some amount less than that. A max delay of `1000` plus a duration of `500` equals 1.5 seconds total transition time.

This approach to delays is great because it keeps our code *scalable*. The total duration will be tolerable whether we have only 10 data points or 10,000.

Randomizing the Data

Just to illustrate how cool this is, let's repurpose our random number generating code from chapter 5 here, so we can update the chart as many times as we want, with new data each time.

Down in our click-update function, let's replace the static `dataset` with a randomly generated one:

```
//New values for dataset
var numValues = dataset.length;                    //Count original length of dataset
dataset = [];                                      //Initialize empty array
for (var i = 0; i < numValues; i++) {              //Loop numValues times
    var newNumber = Math.round(Math.random() * 25); //New random integer (0-25)
    dataset.push(newNumber);                       //Add new number to array
}
```

This will overwrite `dataset` with an array of random integers with values between 0 and 25. The new array will be the same length as the original array.

Then, I'll update the paragraph text:

```
<p>Click on this text to update the chart with new data values as many times as you like!</p>
```

Now open up `17_randomized_data.html`, and you should see:

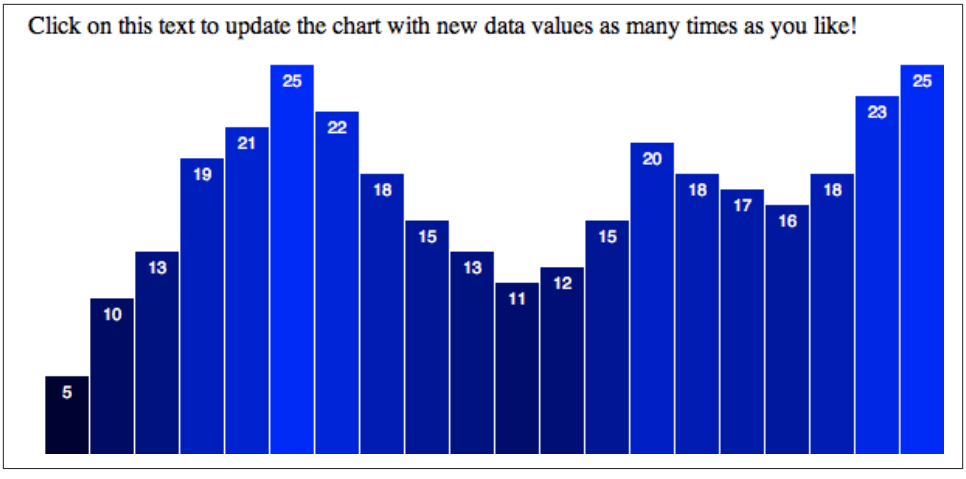


Figure 9-7. Initial view

Every time you click the paragraph at top, the code will:

1. Generate new, random values
2. Bind those values to the existing elements
3. Transition elements to new positions, heights, and colors, using the new values!

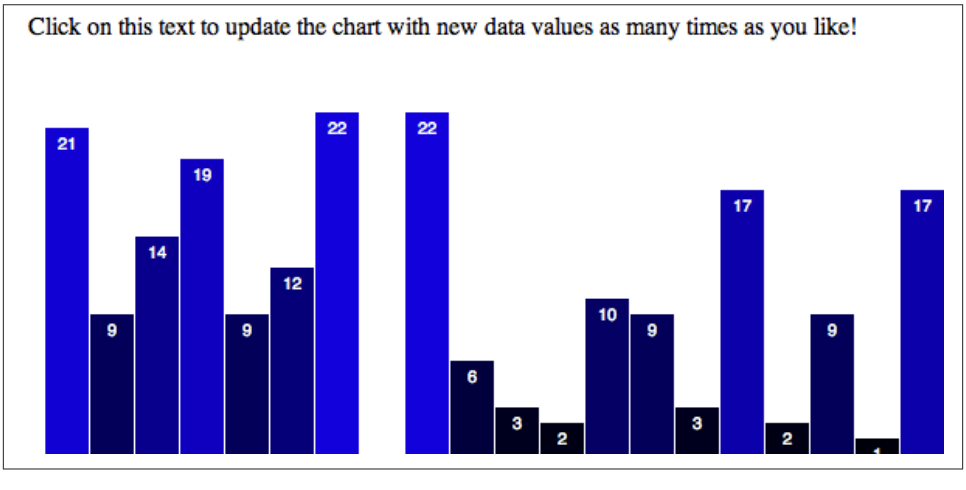
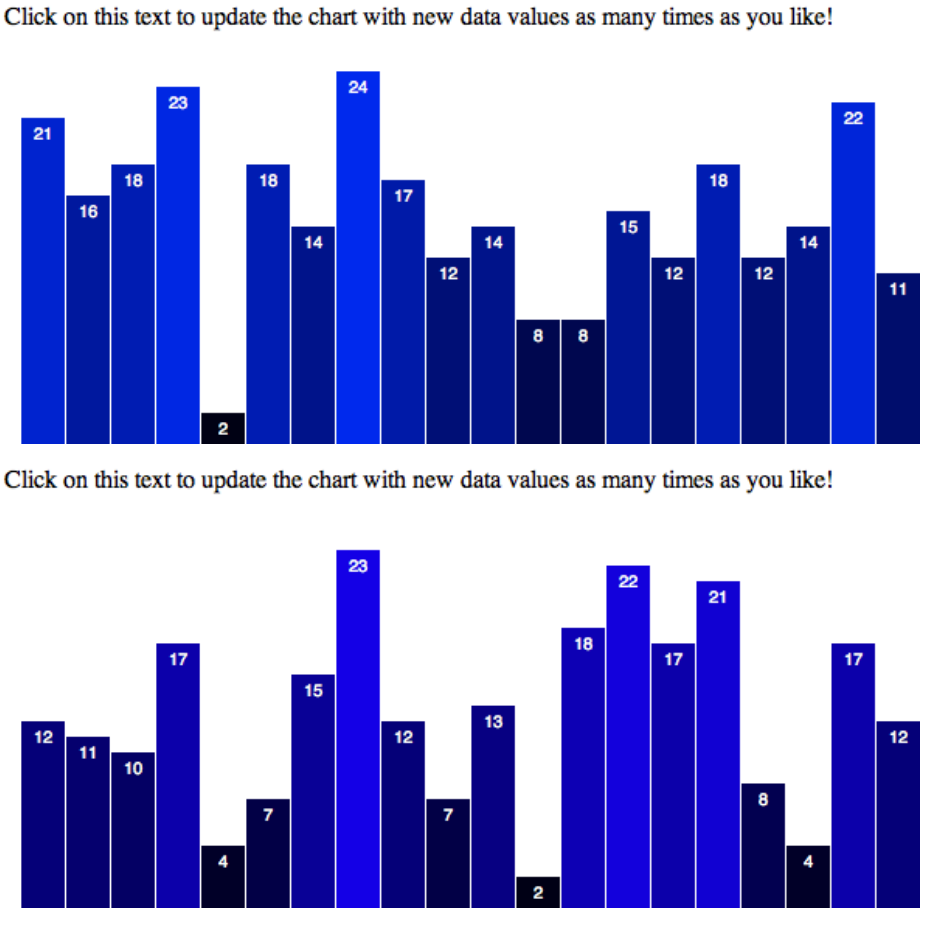


Figure 9-8. Random data applied



Pretty cool! If this does not feel pretty cool or even a little cool to you, please turn off your computer and go for a short walk to clear your head, thereby making room for all the coolness to come. If, however, you are sufficiently cooled by this knowledge, read on.

Updating Scales

Astute readers may take issue with this line from above:

```
var newNumber = Math.round(Math.random() * 25);
```

25? Why 25? In programming, this is referred to as a *magic number*. I know, it sounds fun, but the problem with magic numbers is that it's difficult to tell why they exist (hence, the "magic"). Instead of `25`, something like `maxValue` would be more meaningful:

```
var newNumber = Math.round(Math.random() * maxValue);
```

Ah, see, now the magic is gone, and I can remember that `25` was acting as the *maximum value* that could be calculated and put into `newNumber`. As a general rule, it's best to avoid magic numbers, and instead store those numbers inside variables with meaningful names, like `maxValue` or `numberOfTimesWatchedTheMovieTopSecret`.

More importantly, I now remember that I arbitrarily chose `25` because values larger than that exceeded the range of our chart's scale, so those bars were cut off. For example, here I replaced `25` with `50`:

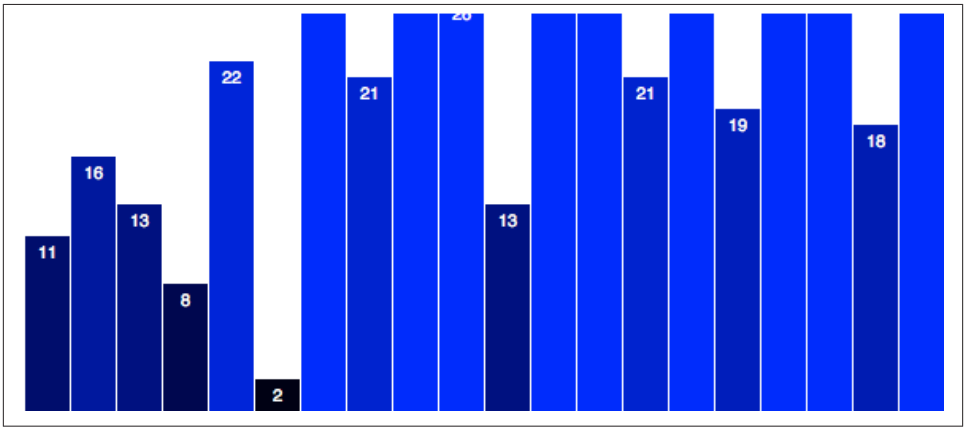


Figure 9-9. Too tall! We used the wrong magic number!

The real problem is not that I chose the *wrong* magic number; it's that our *scale* needs to be updated whenever the data set is updated! Whenever we plug in new data values, we should also recalibrate our scale to ensure that bars don't get too tall or too short.

Updating a scale is easy. You'll recall we created `yScale` with this code:

```
var yScale = d3.scale.linear()
                .domain([0, d3.max(dataset)])
                .range([0, h]);
```

The *range* can stay the same (since the visual size of our chart isn't changing), but after the new dataset has been generated, we should update the scale's *domain*:

```
//Update scale domain
yScale.domain([0, d3.max(dataset)]);
```

This sets upper end of the input domain to the largest data value in `dataset`. Down below, when we update all the bars and labels, we already reference `yScale` to calculate their positions, so no other code changes are necessary!

Check it out in `18_dynamic_scale.html`. I went ahead and replaced our magic number `25` with `maxValue`, which I set here to `100`. So now when we click to update, we get random numbers between 0 and 100. If the *maxmimum* value in the data set is 100, then `yScale`'s domain will go up to 100, as we see here:

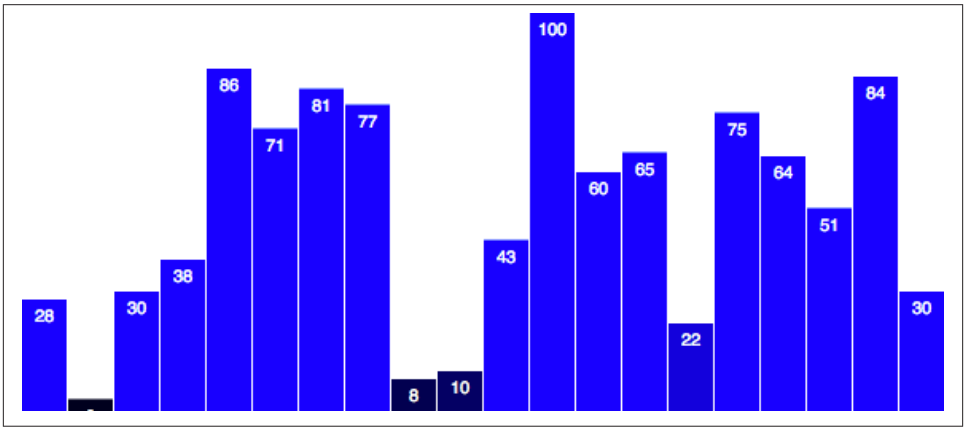


Figure 9-10. Random data, but the y axis scale automatically accommodates

But since the numbers are random, they won't always reach that maximum value of 100. Here, they top out at 85:

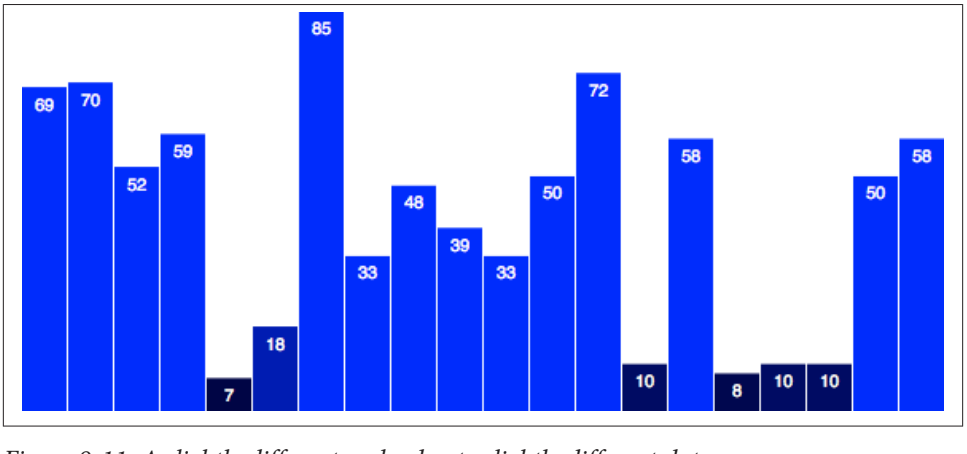


Figure 9-11. A slightly different scale, due to slightly different data

Note that the height of the 100 bar in the first chart is *the same* as the height of the 85 bar here. The data is changing; the scale input domain is changing; the output visual range does *not* change.

Updating Axes

This bar chart doesn't have any axes, but our scatterplot from the last chapter does. I've brought it back, with a few tweaks, in `19_axes_static.html`.

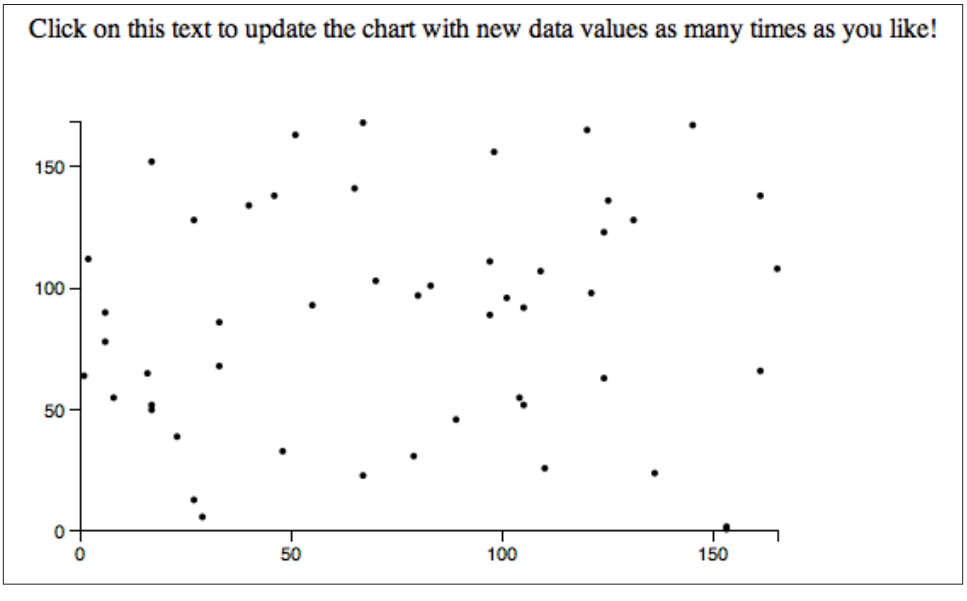


Figure 9-12. Updated scatterplot, now with data updates and dynamic scales!

To summarize the changes to the scatterplot:

- You can now click the text at top to generate and update with new data
- Animated transitions used after data updates
- I eliminated the staggered delay, and set all transitions to occur over a full second (1000ms)
- Both the x and y axis scales are updated, too
- Circles now have a constant radius

Try clicking the text and watch all those little dots zoom around. Cute! I sort of wish they represented some meaningful information, but hey, random data can be fun, too.

What's *not* happening yet is that the axes aren't updating. Fortunately, that is simple to do.

First, I am going to add the class names `x` and `y` to our x and y axes, respectively. This will help us select those axes later:

```
//Create X axis
svg.append("g")
    .attr("class", "x axis")    // <-- Note x added here
    .attr("transform", "translate(0," + (h - padding) + ")")
    .call(xAxis);

//Create Y axis
svg.append("g")
    .attr("class", "y axis")    // <-- Note y added here
    .attr("transform", "translate(" + padding + ",0)")
    .call(yAxis);
```

Then, down in our click function, we simply add:

```
//Update X axis
svg.select(".x.axis")
    .transition()
    .duration(1000)
    .call(xAxis);

//Update Y axis
svg.select(".y.axis")
    .transition()
    .duration(1000)
    .call(yAxis);
```

For each axis, we:

1. Select the axis
2. Initiate a transition
3. Set the transition's duration
4. Call the appropriate axis generator

Remember that each axis generator is already referencing a scale (either `xScale` or `yScale`). Since those scales are being updated, the axis generators can calculate what the new tick marks should be.

Open up `20_axes_dynamic.html` and give it a try.

Once again, `transition()` handles all the interpolation magic for you — watch those ticks fade in and out! Just beautiful, and you barely had to lift a finger.

each() Transition Starts and Ends

There will be times when you want to make something happen at the start or end of a transition. In those times, you can use `each()` to execute arbitrary code for each element in the selection.

`each()` expects two arguments:

1. Either `"start"` or `"end"`
2. An anonymous function, to be executed either at the start of a transition, or as soon as it has ended

For example, here is our circle-updating code, with two `each()` statements added:

```
//Update all circles
svg.selectAll("circle")
   .data(dataset)
   .transition()
   .duration(1000)
   .each("start", function() {      // <-- Executes at start of transition
       d3.select(this)
         .attr("fill", "magenta")
         .attr("r", 3);
   })
   .attr("cx", function(d) {
        return xScale(d[0]);
   })
   .attr("cy", function(d) {
        return yScale(d[1]);
   })
   .each("end", function() {        // <-- Executes at end of transition
       d3.select(this)
         .attr("fill", "black")
         .attr("r", 2);
   });
```

You can see this in action in `21_each.html`.

Now you click the trigger, and immediately each circle's fill is set to magenta, and its radius set to 3. Then the transition is run, per usual. When complete, the fills and radii are restored to their original values.

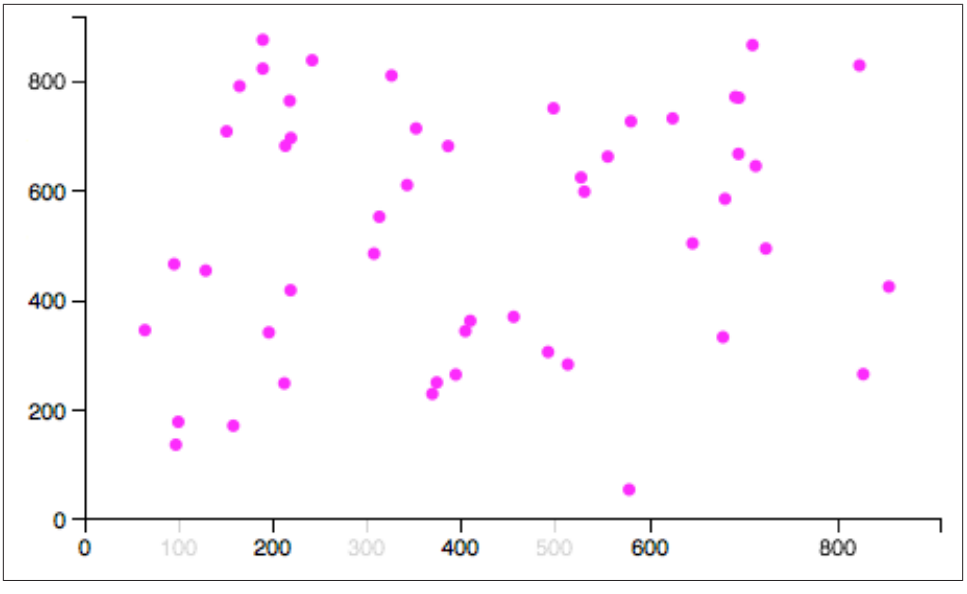


Figure 9-13. Hot pink circles in mid-transition

Something to note is that within the anonymous function passed to `each()`, the context of `this` is maintained as "the current element." This is handy, because then `this` can be referenced with the function to easily re-select the current element and modify it, as done here:

```
.each("start", function() {
    d3.select(this)             // Selects 'this', the current element
      .attr("fill", "magenta")  // Sets fill of 'this' to magenta
      .attr("r", 3);            // Sets radius of 'this' to 3
})
```

Start Carefully

Warning: You may be tempted to throw another transition in here, resulting in a smooth fade from black to magenta. Don't do it! Or do it, but note that this will break:

```
.each("start", function() {
    d3.select(this)
      .transition()             // New transition
      .duration(250)            // New duration
      .attr("fill", "magenta")
      .attr("r", 3);
})
```

If you try this, and I recommend that you do, you'll find that the circles do indeed fade to pink, but they no longer change positions in space. That's because, by default, *only one transition can be active on any given element at any given time*. Newer transitions interrupt and override older transitions.

This might seem like a design flaw, but D3 operates this way on purpose. Imagine if you had several different buttons, each of which triggered a different view of the data, and a visitor was clicking through them in rapid succession. Wouldn't you want an earlier transition to be interrupted, so the last-selected view could be put in place right away?

(If you're familiar with jQuery, you'll notice a difference here. By default, jQuery queues transitions, so they execute one after another, and calling a new transition doesn't automatically interrupt any existing ones. This sometimes results annoying interface behavior, like menus that fade in when you mouse over them, but won't start fading out until *after* the fade-in has completed.)

In this case, the code above "breaks" because the first (spatial) transition is begun, then `each("start", …)` is called on each element. Within `each()`, a second transition is initiated (the fade to pink), overriding the first transition, so the circles never make it to their final destinations (though they look great, just sitting at home).

Because of this quirk of transitions, just remember that `each("start", …)` should be used only for immediate transformations, with no transitions.

End Gracefully

`each("end", …)`, however, *does* support transitions. By the time `each("end", …)` is called, the primary transition has already ended, so initiating a new transition won't cause any harm.

See `22_each_combo_transition.html`. Within the first `each()` statement, I bumped the pink circle radius size up to 7. In the second, I added two lines for transition and a duration:

```
.each("end", function() {
    d3.select(this)
      .transition()            // <-- New!
      .duration(1000)          // <-- New!
      .attr("fill", "black")
      .attr("r", 2);
});
```

Watch that transition: so cool! Note the sequence of events:

1. You click the `p` text
2. Circles turn pink and increase in size immediately
3. Circles transition to new positions

4. Circles transition to original color and size

Also, try clicking on the `p` trigger several times in a row. Go ahead, just go nuts, click as fast as you can. Notice how each click interrupts the circles' progress. (Sorry, guys!) You're seeing each new transition request override the old one. The circles will never reach their final positions and fade to black unless you stop clicking and give them time to rest.

Transitionless each()

You can also use `each()` outside of a transition, just to execute arbitrary code for each element in a selection. Outside of transitions, just omit the `"start"` or `"end"` parameter, and only pass in the anonymous function.

Although I didn't do this above, you can also include references to `d` and `i` within your function definition, and D3 will hand off those values, as you'd expect.

```
…
.each(function(d, i) {
    //Do something with d and i here
});
```

Containing Visual Elements with Clipping Paths

On a slightly related note, you may have noticed that during these transitions, points with low x or y values would exceed the boundaries of the chart area, and overlap the axis lines:

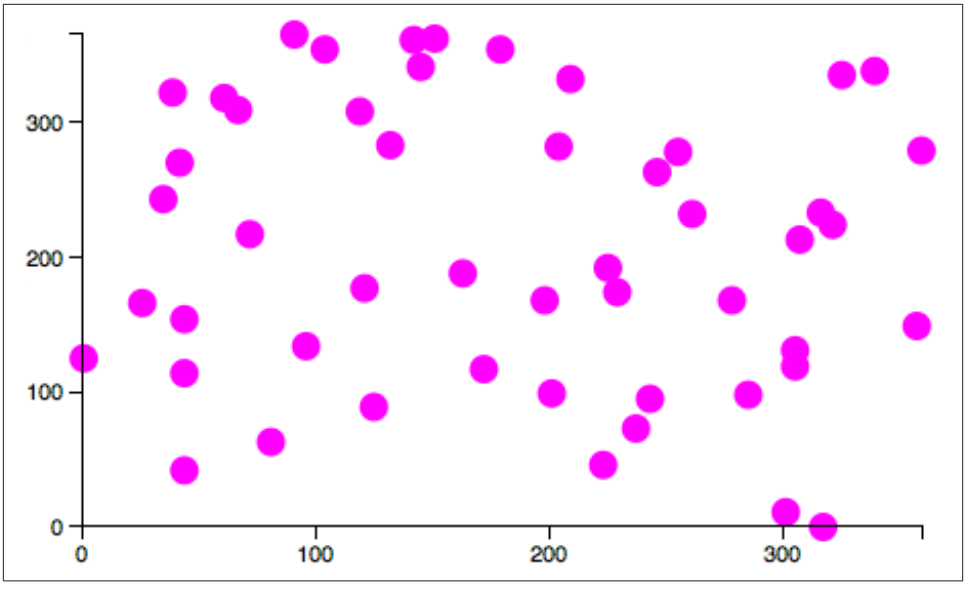


Figure 9-14. Points exceeding the chart area

Fortunately, SVG has support for *clipping paths*, which may be more familiar to you as *masks*, as in Photoshop or Illustrator. A clipping path is an SVG element that contains visual elements which, together, make up the clipping path or mask to be applied to other elements. When a mask is applied to an element, only the pixels that land within that mask's shape are displayed.

Much like `g`, a `clipPath` has no visual presence of its own, but it contains visual elements (which are used to make the mask). For example, here's a simple `clipPath`:

```
<clipPath id="chart-area">
    <rect x="30" y="30" width="410" height="240"></rect>
</clipPath>
```

Note that the outer `clipPath` element has been given an ID of `chart-area`. We can use that ID to reference it later. Within the `clipPath` is a `rect`, which will function as the mask.

So there are three steps to using a clipping path:

1. Define the `clipPath` and give it an ID
2. Put visual elements within the `clipPath` (usually just a `rect`, but this could be `circle`s or any other visual element)
3. Add a reference to the `clipPath` from whatever element(s) you wish to be masked

Continuing with the scatterplot, I'll define the clipping path with this new code (steps 1 and 2):

```
//Define clipping path
svg.append("clipPath")                  //Make a new clipPath
    .attr("id", "chart-area")           //Assign an ID
    .append("rect")                     //Within the clipPath, create a new rect
    .attr("x", padding)                 //Set rect's position and size…
    .attr("y", padding)
    .attr("width", w - padding * 3)
    .attr("height", h - padding * 2);
```

I want all of the `circle`s to be masked by this `clipPath`. I could add a `clipPath` reference to every single circle, but it's much easier and cleaner to just put all the `circle`s into a `g` group, and then add the reference to that. (This is step 3 above.)

So, I will modify this code…

```
//Create circles
svg.selectAll("circle")
   .data(dataset)
   .enter()
   .append("circle")
   …
```

…by adding three new lines, creating a new `g`, giving it an arbitrary ID, and finally adding the reference to the `chart-area` `clipPath`:

```
//Create circles
svg.append("g")                             //Create new g
   .attr("id", "circles")                   //Assign ID of 'circles'
   .attr("clip-path", "url(#chart-area)")   //Add reference to clipPath
   .selectAll("circle")                     //Continue as before…
   .data(dataset)
   .enter()
   .append("circle")
   …
```

Notice that the attribute name is `clip-path`, while the element name is `clipPath`. Argh! I know; it drives me crazy, too.

View the sample page `23_clip-path.html` and open the web inspector. Let's look at that new `rect`.

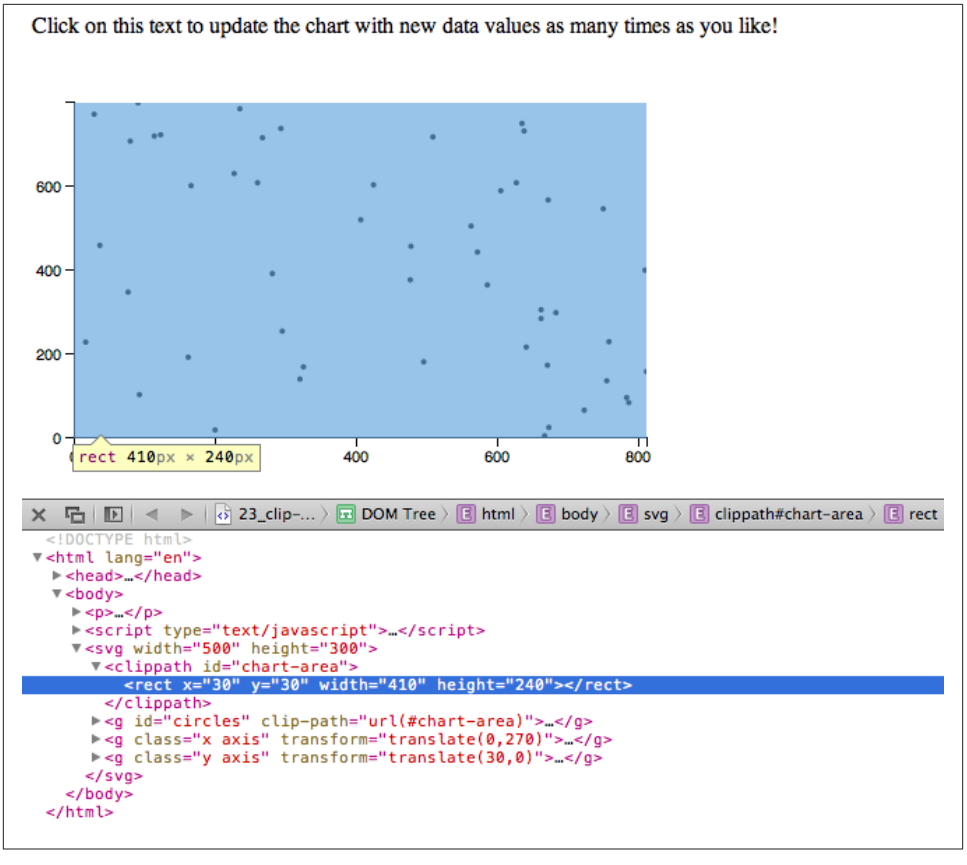


Figure 9-15. The dimensions of a `rect` within a `clipPath`

Since `clipPath`'s have no visual rendering (they only mask other elements), it's helpful to highlight them in the web inspector, which will then outline the path's position and size with a blue highlight. So here we can see that the `clipPath` `rect` is in the right place, and is the right size.

Notice, too, that now all the `circle`s are grouped within a single `g` element, whose `clip-path` attribute references our new clipping path, using the slightly peculiar syntax `url(#chart-area)`. Thanks for that, SVG specification.

The end result is that our `circle`'s pixels get clipped when they get too close to the edge of the chart area. Note the points at the extreme top and right edges above.

The clipping is easier to see mid-transition:

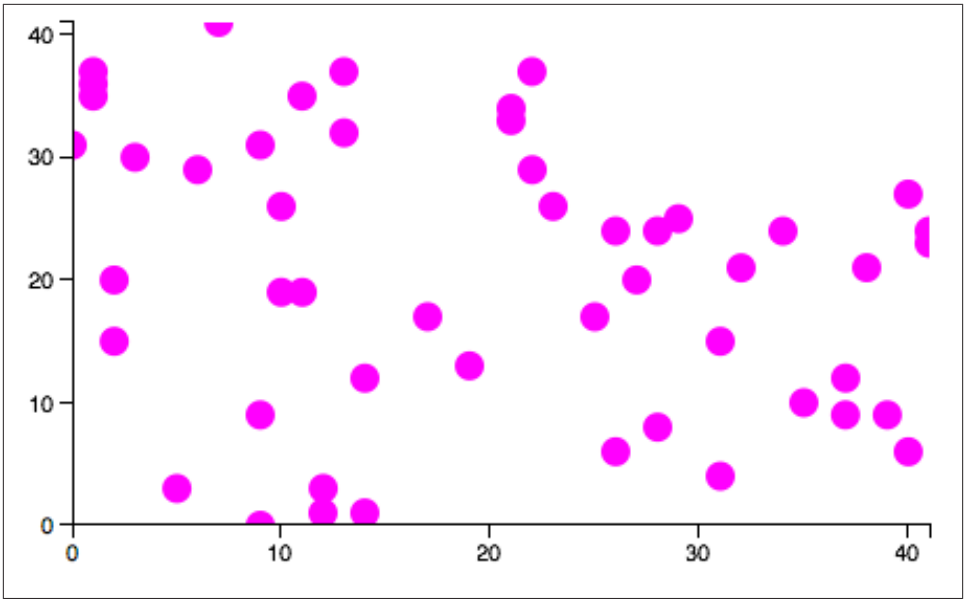


Figure 9-16. Points contained within the chart area

Voila! The points no longer exceed the chart boundaries.

Other Kinds of Data Updates

Until now, when updating data, we have taken the "whole kit-and-kaboodle" approach: changing values in the data set array, and then re-binding that revised data set, overwriting the original values bound to our DOM elements.

That approach is most useful when *all* the values are changing, and when the length of the data set (i.e., the number of values) stays the same. But as we know, real-life data is messy, and calls for even more flexibility, such as when you only want to update one or two values, or you even need to add or subtract values. D3, again, comes to the rescue.

Adding Values (and Elements)

Let's go back to our lovely bar chart, and say that some user interaction (a mouse click) should now trigger adding a *new* value to our data set. That is, the length of the array `dataset` will increase by one.

Well, generating a random number and pushing it to the array is easy enough:

```
//Add one new value to dataset
var maxValue = 25;
var newNumber = Math.round(Math.random() * maxValue);
dataset.push(newNumber);
```

Making room for an extra bar will require recalibrating our x axis scale. That's just a matter of updating its input domain to reflect the new length of `dataset`:

```
xScale.domain(d3.range(dataset.length));
```

That's the easy stuff. Now prepare to bend your brain yet again, as we dive into the depths of D3 *selections*.

Select…

By now, you are comfortable using `select()` and `selectAll()` to grab and return DOM element selections. And you've even seen how, when these methods are chained, they grab selections within selections, and so on, as in:

```
d3.select("body").selectAll("p");   //Returns all 'p' elements within 'body'
```

When storing the *results* of these selection methods, the most specific result — meaning, the result of the last selector in the chain — is the reference handed off to the variable. For example:

```
var paragraphs = d3.select("body").selectAll("p");
```

Now `paragraphs` contains a selection of all `p` elements in the DOM, even though we traversed through `body` to get there.

The twist here is that the `data()` method *also* returns a selection. Specifically, `data()` returns references to all elements to which data was just bound, which we call the *update* selection.

In the case of our bar chart, this means that we can select all the bars, then re-bind the new data to those bars, and grab the update selection all in one fell swoop.

```
//Select…
var bars = svg.selectAll("rect")
    .data(dataset);
```

Now the update selection is stored in `bars`.

Enter…

When we changed our data values, but not the *length* of the whole data set, we didn't have to worry about an update selection — we simple re-bound the data, and transitioned to new attribute values.

But now we have *added* a value. So `dataset.length` was originally 20, but now it is 21. How can we address that new data value, specifically, in order to draw a new `rect` for it? Stay with me here; your patience will be rewarded.

The genius of the update selection is that it contains within it references to *enter* and *exit* sub-selections.

Entering elements are those that are new to the scene. It's considered good form to welcome such elements to the neighborhood with a plate of cookies.

Whenever there are more data values than corresponding DOM elements, the *enter* selection contains references to those elements *which do not yet exist*. You already know how to access the enter selection: by using `enter()` after binding the new data, as we do when first creating the bar chart. You have already seen this code:

```
svg.selectAll("rect")   //Selects all rects (as yet non-existent)
   .data(dataset)       //Binds data to selection, returns update selection
   .enter()             //Extracts the enter selection, i.e. 20 placeholder elements
   .append("rect")      //Creates a 'rect' inside each of the placeholder elements
   …
```

You have already seen this sequence of `selectAll()` > `data()` > `enter()` > `append()` many times, but only in the context of creating many elements at once, when the page first loads.

Now that we have added one value to `dataset`, we can use `enter()` to address the one new corresponding DOM element, without touching all the existing `rect`s. Following the `Select…` code above, I'll add:

```
//Enter…
bars.enter()
    .append("rect")
    .attr("x", w)
    .attr("y", function(d) {
        return h - yScale(d);
    })
    .attr("width", xScale.rangeBand())
    .attr("height", function(d) {
        return yScale(d);
    })
    .attr("fill", function(d) {
        return "rgb(0, 0, " + (d * 10) + ")";
    });
```

Remember, `bars` contains the update selection, so `bars.enter()` extracts the enter selection from that. In this case, the enter selection is one reference to one new DOM element. We follow that with `append()` to create the new `rect`, and all the other `attr()` statements as usual, except for this line:

```
    .attr("x", w)
```

You may notice that this sets the horizontal position of the new `rect` to be just past the far right edge of the SVG. I want the new bar to be created just out of sight, so I can use a nice, smooth transition to move it gently into view.

Update…

We made the new `rect`; now all that's left is to update all `rect`s' visual attributes. Again, `bars` here stores the complete update selection (which includes the enter selection).

```
//Update…
bars.transition()
    .duration(500)
    .attr("x", function(d, i) {
        return xScale(i);
    })
    .attr("y", function(d) {
        return h - yScale(d);
    })
    .attr("width", xScale.rangeBand())
    .attr("height", function(d) {
        return yScale(d);
    });
```

This has the effect of taking *all* the bars and transitioning them to their new x, y, width, and height values! Don't believe me? See the working code in `24_adding_values.html`.

Here's the initial chart:

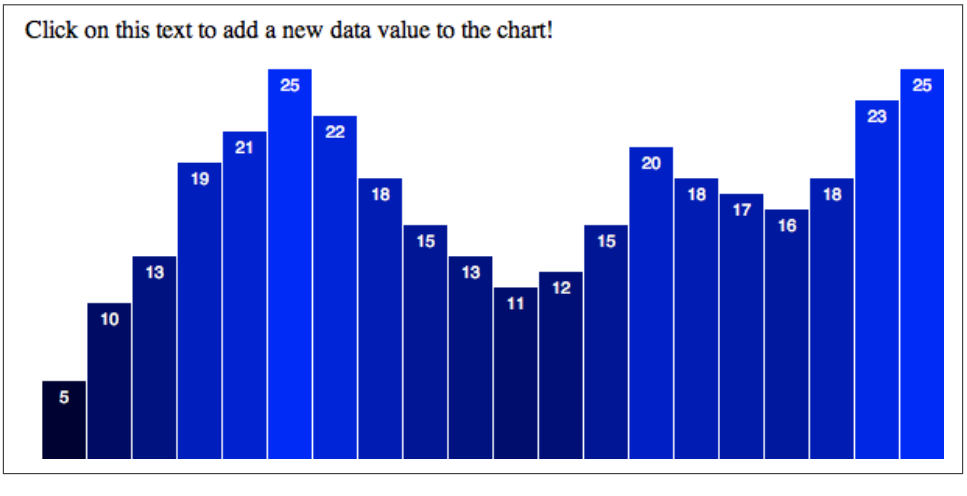


Figure 9-17. Intial bar chart

Then, the chart after one click on the text. Note the new bar on the right!

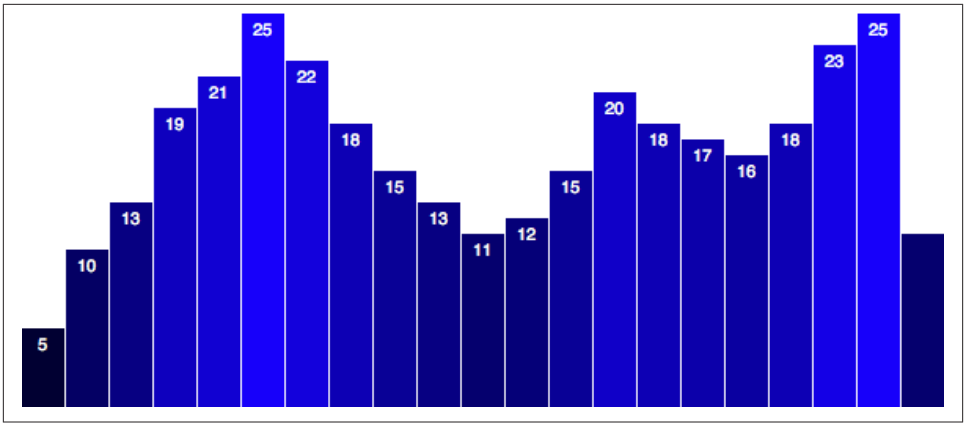


Figure 9-18. After one click

After two clicks…

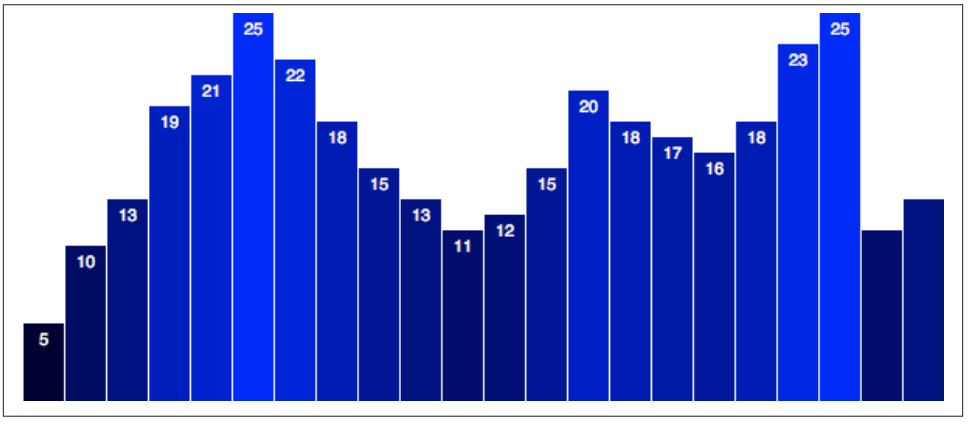


Figure 9-19. After two clicks

…then three…

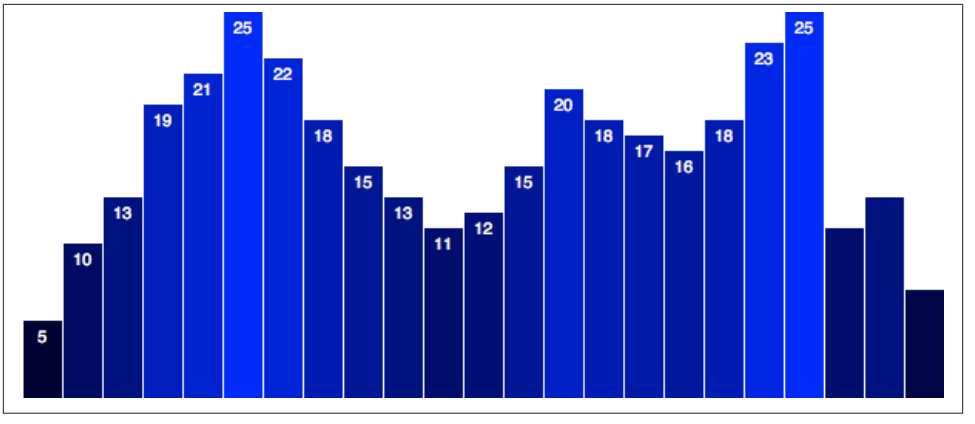


Figure 9-20. After three clicks

…then several more:

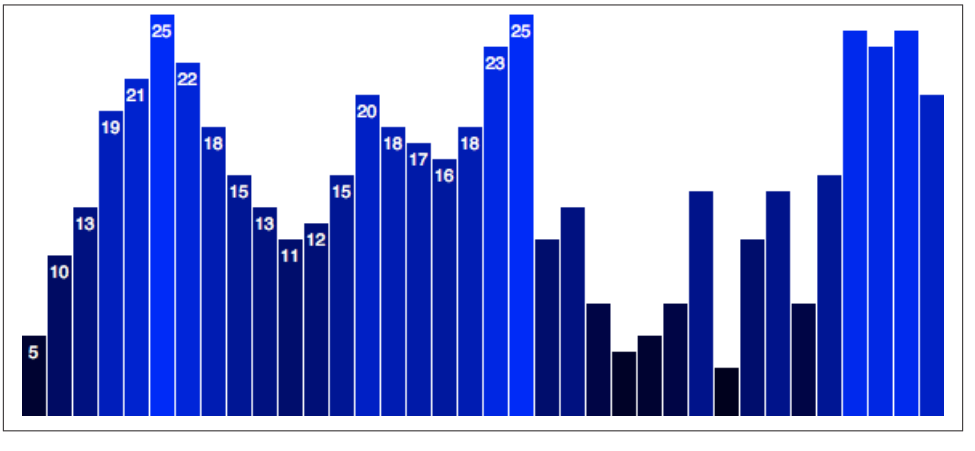


Figure 9-21. After many clicks

Not only are new bars being created, sized, and positioned, but on every click, *all other bars* are rescaled and moved into position as well.

What's *not* happening is that new value labels aren't being created and transitioned into place. I leave that as an exercise for you to pursue.

Removing Values (and Elements)

Removing elements is easier.

Whenever there are more DOM elements than data values, the *exit* selection contains references to those elements without data. As you've already guessed, we can access the exit selection with `exit()`.

First, I'll change our trigger text to indicate we're removing values:

```
<p>Click on this text to remove a data value from the chart!</p>
```

Then, on click, instead of generating a new random value and adding it to `dataset`, we'll use `shift()`, which removes the first element from the array:

```
//Remove one value from dataset
dataset.shift();
```

Exit…

Exiting elements are those that are on their way out. We should be polite and wish these elements a safe journey.

So we grab the exit selection, transition the exiting element off to the right side, and, finally, remove it:

```
//Exit…
bars.exit()
    .transition()
    .duration(500)
    .attr("x", w)
    .remove();
```

`remove()` is a special transition method that waits until the transition is complete, and then deletes the element from the DOM forever. (Sorry, there's no getting it back.)

Making a Smooth Exit

Visually speaking, it's good practice to perform a transition first, rather than simply `remove()` elements right away. In this case, we're moving the bar off to the right, but you could just as easily transition `opacity` to zero, or apply some other visual transition.

That said, if you ever need to just get rid of an element ASAP, by all means, you can use `remove()` without calling a transition first.

Okay, now try out the code in `25_removing_values.html`. Here's the inital view:

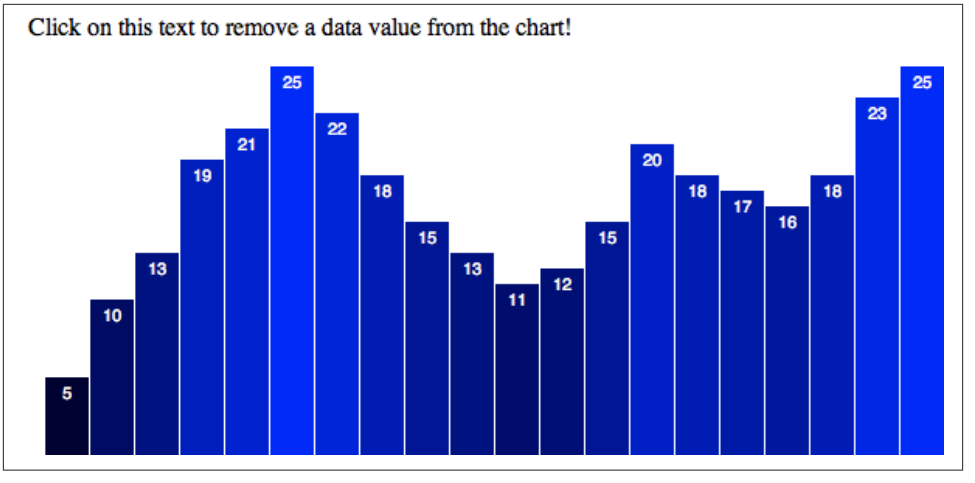


Figure 9-22. Intial bar chart

Then, after one click on the text. Note the loss of one bar:

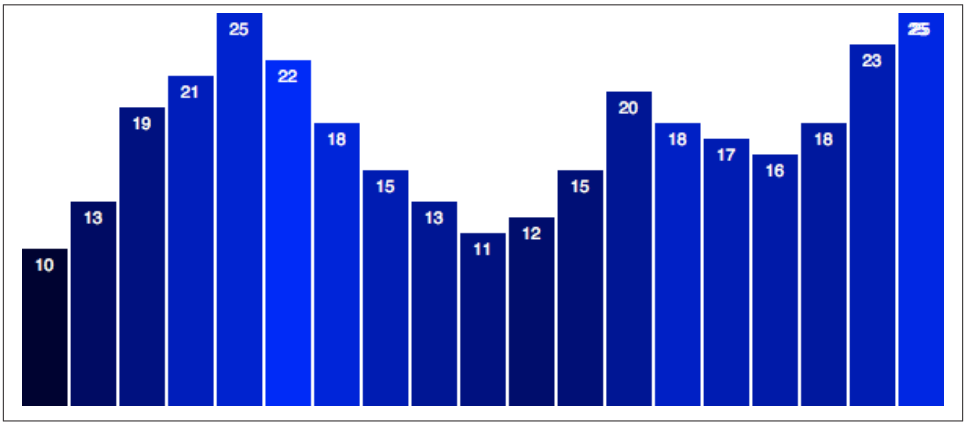


Figure 9-23. After one click

After two clicks…

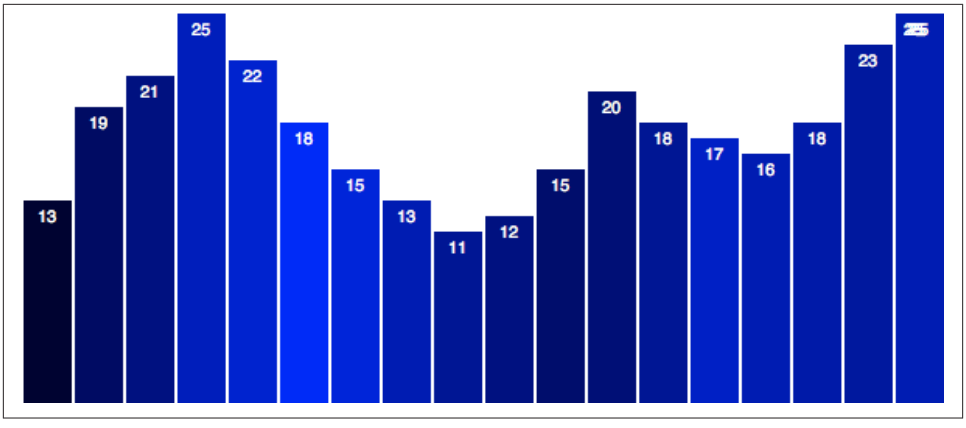


Figure 9-24. After two clicks

…then three…

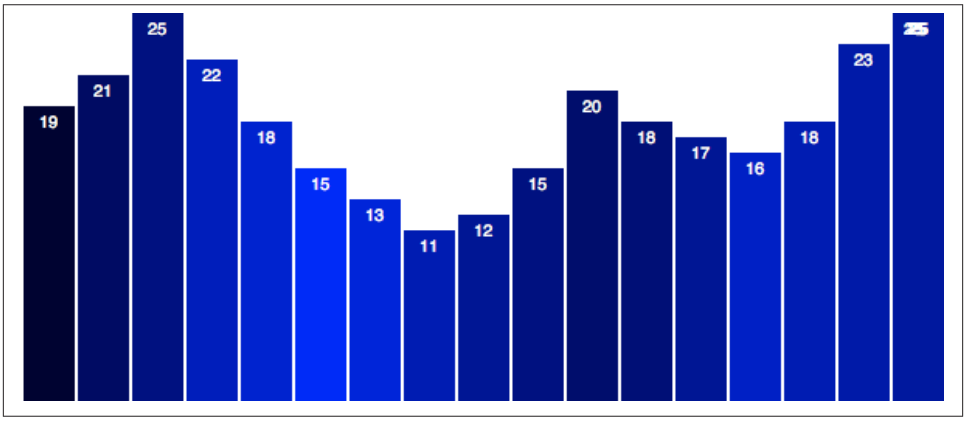


Figure 9-25. After three clicks

…then several more:

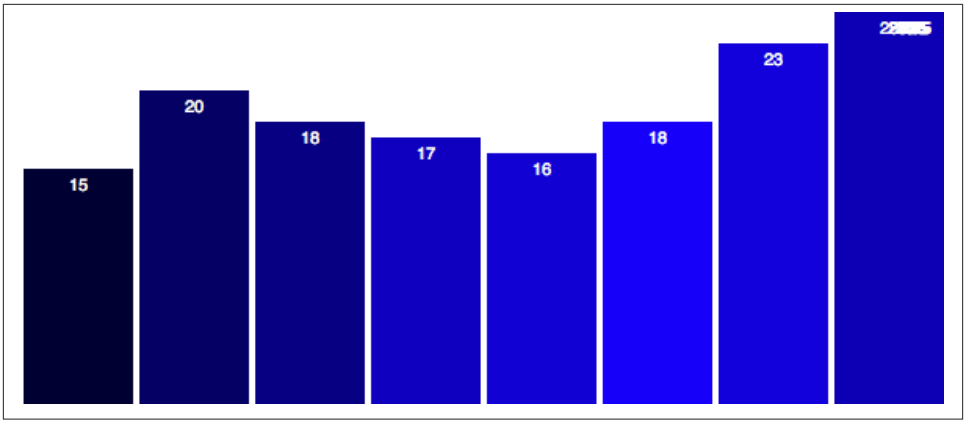


Figure 9-26. After many clicks

On each click, one bar moves off to the right, and then is removed from the DOM. (You can confirm this with the web inspector.)

But what's not working as expected? For starters, the value labels aren't being removed, so they clutter up the top right of our chart. Again, I will leave fixing this aspect as an exercise to you.

More importantly, although we are using the `Array.shift()` method to remove the *first* value from the `dataset` array, it's not the *first* bar that is removed, is it? Instead, the last bar in the DOM, the one visually on the far right, is always removed. Although the data values are updating correctly (note how they move to the left with each click — 5, 10, 13, 19, and so on), the bars are assigned new values, rather than "sticking" with their intial values. That is, the anticipated *object constancy* is broken — the "5" bar becomes the "10" bar, and so on, yet perceptually we would prefer that the "5" bar simply scoot off to the left and let all the other bars keep their original values.

Why, why, oh, why is this happening?! Not to worry; there's a perfectly reasonable explanation. The key to maintaining object constancy is, well, keys. (On a side note, Mike Bostock has a very eloquent overview of the value of object constancy, which I recommend.)

Data Joins With Keys

Now that you understand update, enter, and exit selections, it's time to dig deeper into data joins.

A "data join" happens whenever you bind data to DOM elements. That is, every time you call `data()`.

The default join is by *index order*, meaning the first data value is bound to the first DOM element in the selection, the second value is bound to the second element, and so on.

But what if the data values and DOM elements are not in the same order? Then you need to tell D3 how to join or pair values and elements. Fortunately, you can define those rules by specifying a *key function*.

This explains the problem with our bars. After we remove the first value from the `dataset` array, we re-bind the new `dataset` on top of the existing elements. Those values are joined in index order, so the first `rect`, which originally had a value of 5, is now assigned 10. The former 10 bar is assigned 13, and so on. In the end, that leaves one `rect` element without data — the last one on the far right.

We can use a *key function* to control the data join with more specificity and ensure that the right data value gets joined and bound to the right `rect` element.

Preparing the Data

Until now, our data set has been a simple array of values. But in order to use a key function, each value must have some "key" associated with it. Think of the key as a means of identifying the value without looking at the value itself, since the values themselves may change or exist in duplicate form. (If there were two values of `3`, how could you tell them apart?)

Instead of an array of values, let's use an array of *objects*, each of which can contain both a key value and the actual data value:

```
var dataset = [ { key: 0, value: 5 },
                { key: 1, value: 10 },
                { key: 2, value: 13 },
                { key: 3, value: 19 },
                { key: 4, value: 21 },
                { key: 5, value: 25 },
                { key: 6, value: 22 },
                { key: 7, value: 18 },
                { key: 8, value: 15 },
                { key: 9, value: 13 },
                { key: 10, value: 11 },
                { key: 11, value: 12 },
                { key: 12, value: 15 },
                { key: 13, value: 20 },
                { key: 14, value: 18 },
                { key: 15, value: 17 },
                { key: 16, value: 16 },
                { key: 17, value: 18 },
                { key: 18, value: 23 },
                { key: 19, value: 25 } ];
```

Remember, hard brackets [ ] indicate an array, while curly brackets {} indicate an object.

Note that the data values here are unchanged from our original `dataset`. What's new are the keys, which just enumerate each object's original position within the `dataset` array. (By the way, your chosen key name don't have to be `key` — the name can be anything, like `id`, `year`, or `fruitType`. I am using "key" here for simplicity.)

Updating All References

The next step isn't fun, but it's not hard. Now that our data values are buried within objects, we can no longer just reference `d`. (Ah, the good old days.) Anywhere in the code where we want to access the actual data *value*, we now need to specify `d.value`. When we use anonymous functions within D3 methods, `d` is handed whatever is in the current position in the array. In this case, each position in the array now contains an object, such as `{ key: 12, value: 15 }`. So to get at the value `15`, we now must write `d.value` to reach *into* the object and grab that `value` value. (I hope you see a lot of value in this paragraph.)

First, that means a change to the `yScale` definition:

```
var yScale = d3.scale.linear()
                .domain([0, d3.max(dataset, function(d) { return d.value; })])
                .range([0, h]);
```

In the second line, we used to have simply `d3.max(dataset)`, but that only works with a simple array. Now that we're using objects, we have to include an *accessor function* that tells `d3.max()` how to get at the correct values to compare. So as `d3.max()` loops through all the elements in the `dataset` array, now it knows not to look at `d` (which is an object, and not easily compared to other objects), but `d.value` (a number, which is easily compared to other numbers).

Note we also need to change the second reference to `yScale`, down in our click-update function:

```
yScale.domain([0, d3.max(dataset, function(d) { return d.value; })]);
```

Next up, everywhere `d` is used to set attributes, we must change `d` to `d.value`. For example, this

```
…
.attr("y", function(d) {
    return h - yScale(d);        // <-- d
})
…
```

becomes this:

```
…
.attr("y", function(d) {
    return h - yScale(d.value);  // <-- d.value!
})
…
```

Key Functions

Finally, we define a key function, to be used whenever we bind data to elements:

```
var key = function(d) {
    return d.key;
};
```

Notice that, in typical D3 form, the function takes `d` as input. Then this very simple function specifies to take the `key` value of whatever `d` object is passed into it.

Now, in all four places where we bind data, we replace this line

```
.data(dataset)
```

with this:

```
.data(dataset, key)
```

One note: Rather than defining the key function first, then referencing it, you could of course simply write the key function directly into the call to `data()` like so:

```
.data(dataset, function(d) {
    return d.key;
})
```

But in this case, you'd have to write that four times, which is redundant, so I think defining the key function once at the top is cleaner.

That's it! Consider your data joined.

Exit Transition

One last tweak: Let's set the exiting bar to scoot off to the left side instead of the right:

```
//Exit…
bars.exit()
    .transition()
    .duration(500)
    .attr("x", -xScale.rangeBand())  // <-- Exit stage left
    .remove();
```

Great! Check out the sample code, with all of those changes, in `26_data_join_with_key.html`. The inital view is unchanged

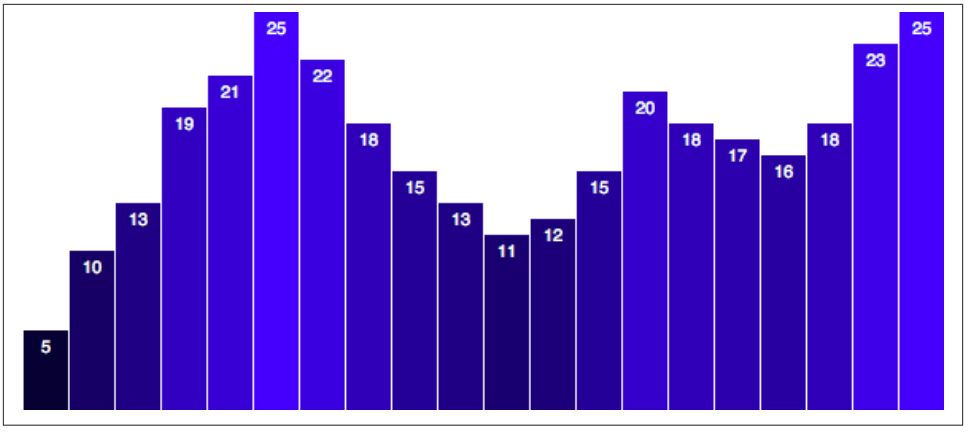


Figure 9-27. Intial bar chart

but try clicking the text, and voila: The left-most bar slides cleanly off to the left, all other bars' widths rescale to fit, and then the exited bar is deleted from the DOM! (Again, you can confirm this by watching the `rect`s disappear one-by-one in the web inspector.)

Here's the view after one bar is removed:

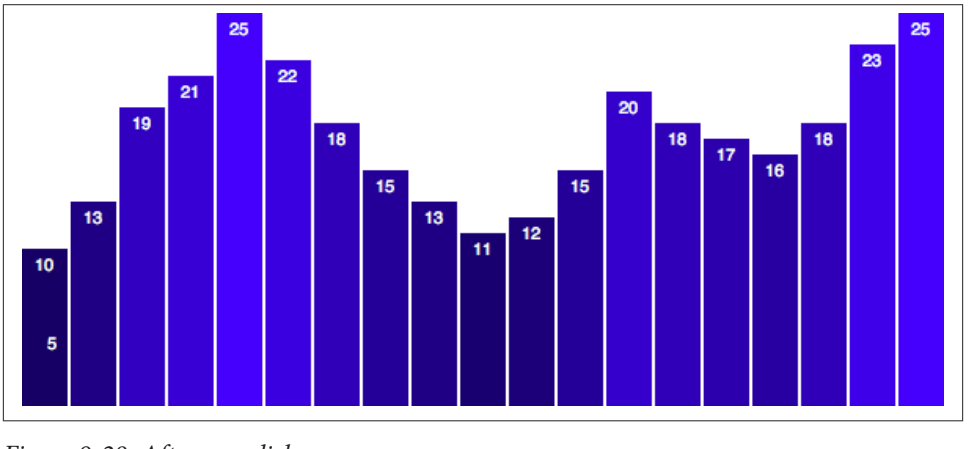


Figure 9-28. After one click

After two…

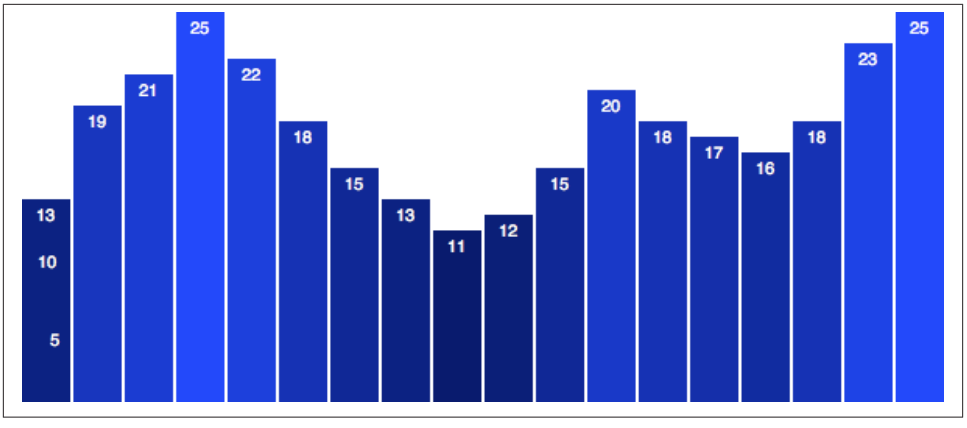


Figure 9-29. After two clicks

…then three…

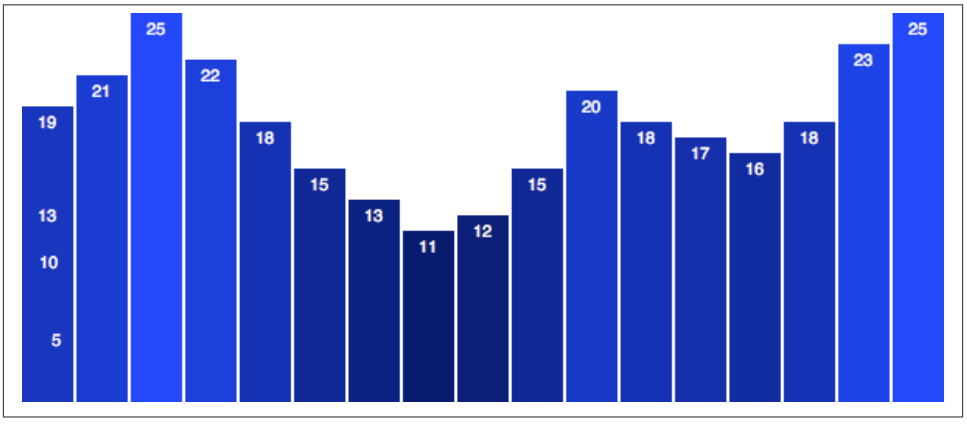


Figure 9-30. After three clicks

…then several more:

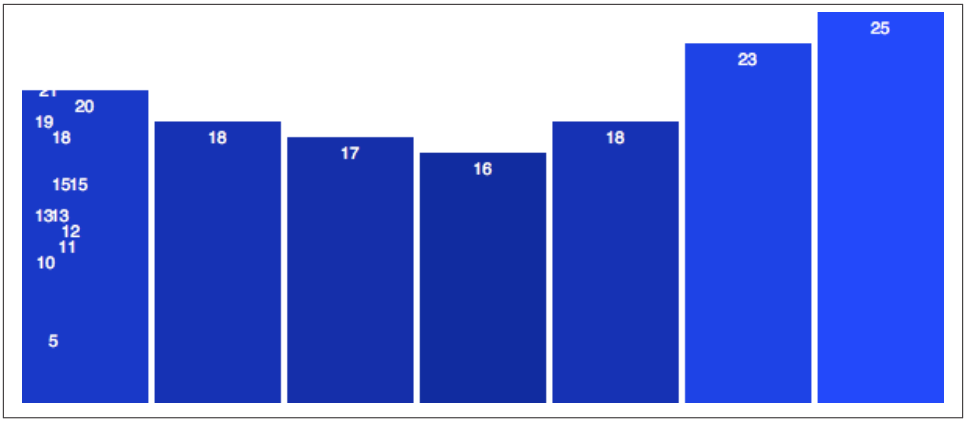


Figure 9-31. After many clicks

This is working better than ever. The only hitch is that the labels aren't exiting to the left, and also are not removed from the DOM, so they clutter up the left side of the chart. Again, I leave this to you; put your new D3 chops to the test and clean up those labels.

Add and Remove: Combo Platter

We could stop there and be super satisfied with our newfound skills. But why not go all the way, and adjust our chart so data values can be added *and* removed?

This is easier than you might think. First, we'll need two different triggers for the user interaction. I'll split the one paragraph into two, and give each a unique ID, so we can tell which one is clicked:

```
<p id="add">Add a new data value</p>
<p id="remove">Remove a data value</p>
```

Later, down where we set up the click function, `select()` must become `selectAll()`, now that we're selecting more than one `p` element:

```
d3.selectAll("p")
    .on("click", function() { …
```

Now that this click function will be bound to both paragraphs, we have to introduce some logic to tell the function to behave differently depending on which paragraph was clicked. There are many ways to achieve this; I'll go with most straightforward one.

Fortunately, within the context of our anonymous click function, `this` refers to the element that was clicked — the paragraph. So we can get the ID value of the clicked element by selecting `this` and inquiring using `attr()`:

```
d3.select(this).attr("id")
```

That statement will return `add` when `p#add` is clicked, and `remove` when `p#remove` is clicked. Let's store that value in a variable, and use it to control an `if` statement:

```
//See which p was clicked
var paragraphID = d3.select(this).attr("id");

//Decide what to do next
if (paragraphID == "add") {
    //Add a data value
    var maxValue = 25;
    var newNumber = Math.round(Math.random() * maxValue);
    var lastKeyValue = dataset[dataset.length - 1].key;
    console.log(lastKeyValue);
    dataset.push({
        key: lastKeyValue + 1,
        value: newNumber
    });
} else {
    //Remove a value
    dataset.shift();
}
```

So, if `p#add` is clicked, we calculate a new random value, and then lookup the key value of the last item in `dataset`. Then we create a new object with an incremented key (to ensure we don't duplicate keys; insert locksmith joke here) and the random data value.

The best part: No additional changes are needed! The enter/update/exit code we wrote is already flexible enough to handle adding *or* removing data values — that's the beauty of it.

Try it out in `27_adding_and_removing.html`! You'll see that you can click to add or remove data points at will! Of course, real-world data isn't created this way, but you can imagine these data updates being triggered by some other event — such as data refreshes being pulled from a server — and not mouse clicks.

Recap

To review:

- `data()` binds data to elements, but also returns the *update selection*.
- The update selection may contain *enter* and *exit* selections, which can be accessed via `enter()` and `exit()`.
- When there are *more values than elements*, an enter selection will reference the placeholder, not-yet-existing elements.
- When there are *more elements than values*, an exit selection will reference the elements without data.
- Data joins determine how values are matched with elements.

- By default, data joins are performed by index, meaning in order of appearance.
- For more control over data joins, you can specify a key function.

One last note on data updates: In the bar chart example above, we used this sequence:

1. Enter
2. Update
3. Exit

While this worked well for us, this order isn't set in stone. Depending on your design goals, you may want to update first, then enter new elements, and finally exit old ones. It all depends — just remember that once you have the update selection in hand, you can reach in to grab the enter and exit selections anytime. The order in which you do so is flexible and completely up to you.

Fantastic. You are well on your way to becoming a D3 wizard. Now let's get to the really fun stuff: interactivity!