O'REILLY®

D3 for the Impatient

Interactive Graphics for Programmers and Scientists

Philipp K. Janert

O'REILLY®

D3 for the Impatient

If you're in a hurry to learn D3.js, the leading JavaScript library for web-based graphics and visualization, this book is for you. Written for technically savvy readers with a background in programming or data science, the book moves quickly, emphasizing unifying concepts and patterns. Anticipating common difficulties, author Philipp K. Janert teaches you how to apply D3 to your own problems.

Assuming only a general programming background, but no previous experience with contemporary web development, this book explains supporting technologies such as SVG, HTML5, CSS, and the DOM as needed, making it a convenient one-stop resource for a technical audience.

- Understand D3 selections, the library's fundamental organizing principle
- Learn how to create data-driven documents with data binding
- Create animated graphs and interactive user interfaces
- Draw figures with curves, shapes, and colors
- Use the built-in facilities for heatmaps, tree graphs, and networks
- Simplify your work by writing your own reusable components

"This book will satisfy your pressing urge to just get started with D3 already! "

—**Scott Murray**
Author of *Interactive Data Visualization for the Web*

"A comprehensive guide that quickly explains both the common patterns and the inner workings of this powerful framework."

—**Giuseppe Verni**
Principal Engineer at Qualcomm

Philipp K. Janert, physicist and programmer, has been working with data and graphs since 1992. The author of *Data Analysis with Open Source Tools* and *Feedback Control for Computer Systems* (both from O'Reilly), he holds a PhD from the University of Washington.

DATA / WEB DEVELOPMENT

US $49.99 CAN $65.99

ISBN: 978-1-492-04677-6

Twitter: @oreillymedia
facebook.com/oreilly

D3 for the Impatient

Interactive Graphics for Programmers and Scientists

Philipp K. Janert

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

D3 for the Impatient
by Philipp K. Janert

Copyright © 2019 Philipp K. Janert. All rights reserved.

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://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jennifer Pollock
Editor: Corbin Collins
Production Editor: Deborah Baker
Copy Editor: Sonia Saruba
Proofreader: Charles Roumeliotis

Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Illustrators: Philipp K. Janert and Rebecca Demarest

May 2019: First Edition

Revision History for the First Edition
2019-05-02: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492046776* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *D3 for the Impatient*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04677-6

[LSI]

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 element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/janert/d3-for-the-impatient*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it 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: "*D3 for the Impatient* by Philipp K. Janert (O'Reilly). Copyright 2019 Philipp K. Janert, 978-1-492-04677-6."

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*.

O'Reilly Online Learning

O'REILLY®

For almost 40 years, *O'Reilly* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

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 *https://oreil.ly/D3-for-the-Impatient*.

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

I would like to thank Mike Loukides and Scott Murray for supporting this project enthusiastically from the very beginning. Giuseppe Verni, Jane Pong, Matt Kirk, Noah Iliinsky, Richard Kreckel, Sankar Rao Bhogi, Scott Murray, and Sebastien Martel read parts or all of the manuscript, tested the examples, and made many important suggestions. Matt, Scott, and Sebastien further answered questions and

shared their insights through extensive correspondence. Special thanks go to Giuseppe Verni, who read the entire manuscript with particularly keen interest and dedication, and offered lots of helpful advice.

The title is a belated homage to *Unix for the Impatient* by Paul W. Abrahams and Bruce R. Larson (Addison-Wesley Professional).

CHAPTER 1
Introduction

D3.js (or just D3, for Data-Driven Documents) is a JavaScript library for manipulating the Document Object Model (DOM) tree in order to represent information visually. It has become a de facto standard for information graphics on the web.

Its popularity notwithstanding, D3 has a reputation for having a steep learning curve. I don't think this is because D3 is complicated (it is not), and not even because it has a large API (it does, but the API is well structured and extremely well designed). Instead, I believe that many of the difficulties experienced by new users are due to *incorrect assumptions*. Because D3 is used to create some impressive graphics, it is easy—even natural—to regard it as a "graphics library" that facilitates handling of graphical primitives and provides high-level support for common plot types. Approaching D3 with this expectation, the new user is unpleasantly surprised at the verbosity required to set something as elementary as the color of an element—and what's that "selection" business anyway? Why can't I simply get a canvas-like element and be done with it?

The mistake here is that D3 is *not* a graphics library: instead, it is a JavaScript library to manipulate the DOM tree! Its basic building blocks are not circles and rectangles, but nodes and DOM elements; and the typical activity does not involve the painting of a graphical shape on a canvas, but the styling of an element through attributes. The "current location" is not so much given through an xy-coordinate on a canvas, but through a selection of nodes in a DOM tree.

This leads to what I believe is the second major challenge for many new users: D3 is a web technology, and relies on a collection of other web technologies; the DOM API and event model, Cascading Style Sheet (CSS) selectors and properties, the JavaScript object model, and of course Scalable Vector Graphics (SVG). In many cases, D3 provides only a relatively thin layer around these technologies, and its own design frequently reflects the underlying APIs. This makes for a large and somewhat heterogeneous environment. If you are already familiar with the complete stack of modern web technologies that is known as HTML5, you will feel right at home, but if you are *unfamiliar* with them, then the lack of a distinct, unified abstraction layer can be very confusing.

Thankfully, you don't need to study all these underlying technologies in depth: D3 does make them easier to use, and it provides a considerable amount of unification and abstraction on top of them. The only area where it is definitely not enough to "wing it" is SVG. You must have an adequate understanding of SVG—not only of its representational elements, but also of the structural elements that control how information is organized within a graph. I tried to put together the necessary information in Appendix B. If you are not familiar with SVG, then I suggest that you work through this appendix before perusing the rest of this book—you will be glad you did!

Whom This Book Is For

This book is intended for *programmers* and *scientists* who want to add D3 to their toolbox. I assume that you are reasonably proficient as a programmer and comfortable working with data and graphics. At the same time, I don't expect that you have more than a cursory knowledge of contemporary professional web development.

Here is what you should already have:

- Knowledge of at least one or two programming languages (but not necessarily JavaScript), and the confidence to pick up the syntax of a new language from its language reference.
- Familiarity with contemporary programming concepts; that is, not just loops, conditionals, and common data structures, but also closures and higher-order functions.
- Basic understanding of XML and the hierarchical structure it imposes on documents. I expect that you know of the DOM and

how it treats the elements of a web page as nodes in a tree, but I do not assume that you are familiar with either the original DOM API or any one of its modern replacements (like jQuery).

- Exposure to simple HTML and CSS (you should be able to recognize and use `<body>` and `<p>` tags, and so on), as well as some familiarity with CSS syntax and mechanisms.

But in particular, the reader I have in mind is *impatient*: proficient and capable, but frustrated by previous attempts to wrap his or her head around D3. If this describes your situation, this book is for you!

Why D3?

Why should D3 be of interest to programmers and scientists—or in fact anybody who is not primarily a web developer? The following reasons stand out:

- D3 provides a convenient way to deliver graphics over the web. If you work with data and visualizations, you know the drill: you create your figures in the plotting program of your choice, then save the results as PNG or PDF, and then create a web page with `<img>` tags, so that other people can see your work. Wouldn't it be nice if you could create and publish your figures in one step?
- More importantly, D3 makes it easy and convenient to create *animated* and *interactive* graphics. This point cannot be overemphasized: scientific visualization has as much to benefit from animation and interactivity as any other field—but this goal has in the past been notoriously difficult to achieve. It frequently required intricate and unbefitting technologies (ever tried your hand at Xlib programming?) or specialized and often expensive commercial packages. D3 lets you leave these challenges behind and puts you into the present day for your visualization needs.
- Graphics aside, D3 is an accessible, easy-to-learn, and easy-to-use framework for general-purpose DOM handling. If you have the occasional need to manipulate the DOM, then D3 may be all you need, without the obligation to master all the other frameworks and APIs for web programming. The design of the library itself is also interesting as a model for the facilities it pro-

vides "out of the box" to handle common data manipulation and visualization tasks.

But more than anything, I believe D3 to be an *enabling technology* that quite generally broadens the range of solutions available to its users. The most interesting applications of D3 are possibly those that have not yet been invented.

What Is in This Book

This book tries to be a *comprehensive*, yet *concise*, introduction to D3, covering most major parts of functionality in sufficient depth.

- It tries to be a convenient, *one-stop* resource by including both API reference documentation as well as background information on ancillary topics that might not be familiar to the typical reader (such as SVG, JavaScript, and the DOM, but also color spaces or the HTML Canvas element).
- The book emphasizes *mechanisms and design concepts*, rather than ready-made recipes. The assumption is that readers will want to learn D3 deeply enough that they can apply it to their own, possibly novel and unforeseen, purposes.

Basically, the hope is that this book prepares you to do things with D3 that I would never have thought of!

. . . and What Is Not

This book intentionally restricts itself to D3 alone, its capabilities and mechanisms. This implies a number of omissions:

- No extensive case studies or cookbook-style recipes
- No introductions to data analysis, statistics, or visual design
- No mention of JavaScript frameworks other than D3
- No discussion of contemporary web development in general

I would like to emphasize the last two items. This book is strictly about D3 *only*, without any reference to or dependence on other JavaScript frameworks or libraries. This is quite intentional: I wanted to make D3 accessible to those readers as well who are unfamiliar or uncomfortable with JavaScript's rich but heterogenous eco-

system. For the same reason, this book does not discuss other topics in contemporary web development. In particular, you will find *no discussion of browser compatibility* and related topics. The assumption is that you are generally using a modern, up-to-date, JavaScript-enabled browser that is capable of rendering SVG.[1]

One other omission concerns D3 support for geographic and geospatial information. Although important, this topic seems sufficiently well-contained that it should not be too difficult to learn from the D3 Reference Documentation once the D3 basics are clear.

How to Read This Book

This book presents a continuous, progressive narrative, with new material systematically introduced from chapter to chapter. That being said, the later chapters in particular can be read in any order after the necessary groundwork has been laid in the first half of the book. Here is my suggested roadmap:

1. Unless you already have a solid foundation in SVG, I strongly recommend that you read Appendix B. Without this knowledge, nothing else will make much sense.
2. Everyone should read Chapter 2 as a tutorial warm-up and to set expectations for the topics that will be discussed.
3. Chapter 3 is required reading. Selections are *the* principal organizing concept in D3. Not only do selections represent a handle on the DOM tree, but they also manage the association between DOM elements and the data set. Pretty much every D3 program begins by taking a selection, and understanding them and their capabilities is mandatory when working with D3.
4. Strictly speaking, Chapter 4 on event handling, interactivity, and animations is optional. But because these are among the most exciting capabilities offered by D3, it would be a shame to skip it.
5. Chapter 5 is important because it explains some fundamental D3 design concepts (such as components and layouts) and

1 This is in the spirit of D3 itself. As the D3 website states: "D3 is not a compatibility layer, so if your browser doesn't support standards, you're out of luck" (*https://github.com/d3/d3/wiki*).

introduces generally useful techniques (like SVG transformations and custom components).

6. The remaining chapters can largely be read in any order, as and when the need for that particular topic arises. In particular, I would like to draw attention to Chapter 7 and its detailed description of the inconspicuous but extremely versatile scale objects, and to the variety of functions for array handling in Chapter 10.

Conventions

This section explains some specific conventions used in the remainder of this book.

Conventions of the D3 API

The D3 API uses some uniform conventions that greatly enhance its usability. Some of them are common JavaScript idioms and not specific to D3, but may not be familiar to non-JavaScript programmers. By setting them out explicitly in one place, subsequent discussions don't need to be cluttered with redundant information.

- D3 is primarily an access layer to the DOM tree. As a rule, D3 makes no attempt to encapsulate underlying technologies, and instead provides convenient, but otherwise generic, handles on them. For example, D3 does not introduce "circle" or "rectangle" abstractions of its own, but instead gives the programmer direct access to the SVG facilities for creating graphical shapes. The advantage of this approach is that D3 is tremendously adaptable and not tied to one particular technology or version. The disadvantage is that programmers need to have knowledge of the underlying technologies, in addition to D3, since D3 itself does not provide a complete abstraction layer.
- Because JavaScript does not enforce formal function signatures, all function arguments are technically optional. Many D3 functions use the following idiom: when called *with* appropriate arguments, these functions act as *setters* (setting the corresponding property to the supplied value); when called *without* arguments, these functions act as *getters* (returning the current

value of the property). To entirely *remove* a property, call the appropriate setter while supplying `null` as argument.

- When called as setters, functions typically return a reference to the current object, thus enabling method chaining. (This idiom is so intuitive and consistent that it will rarely be mentioned explicitly again.)
- Instead of a value, many D3 setters can take an *accessor function* as argument, which is expected to return a value that will be used to set the property in question. The parameters expected by accessor functions are not the same across all of D3, but a set of related D3 functions will always call accessor functions in a uniform way. The details of accessor arguments are documented with the respective D3 functions.
- Some important D3 facilities are implemented as *function objects*. They perform their primary task when called as a function, but they are also objects, with member functions and internal state (examples are *scale objects*, see Chapter 7; and *generators* and *components*, see Chapter 5). It is a common pattern to instantiate such an object, configure it using its member functions, and finally invoke it to complete its purpose. Frequently, the final invocation does not use explicit function-call syntax, but instead employs one of JavaScript's methods for "synthetic" function calls: the function object is passed to another function (such as `call()`), which supplies the required arguments and finally evaluates the function object itself.

Conventions for the API Reference Tables

Throughout the book, you will find tables showing parts of the D3 API in a reference format. Entries in these tables are sorted by relevance, keeping related functions together.

- D3 functions are either called on the global `d3` object, or as member functions of some D3 object; some functions are available both ways. If a function is called through an object, this object is referred to as the *receiver* of the method call. Inside the member function, the receiver is the object pointed to by the `this` variable.

- All API reference tables indicate the type of the receiver in the caption. The tables do not refer explicitly to the object's prototype.
- Function signatures *attempt* to indicate the type of each argument, but many functions accept such a wide variety of different argument types that no unambiguous notation is practical. Read the textual description for full details. Where they are used, *brackets* indicate an array. Optional function arguments are not indicated explicitly.

Conventions for the Code Examples

The code examples are intended to demonstrate D3 features and mechanisms. In order to bring out their respective point most clearly, the examples are stripped to the bare essentials. I dispensed with most "niceties," such as pleasing colors or semantically interesting data sets. Colors are usually primary, and most data sets are small and simple.

On the other hand, each example is complete in itself, can be executed as is, and will create the associated graph. With few exceptions, I don't show code fragments. I found that it is better to display simple examples in full, rather than show only the "interesting bits" from a longer example; this way, there is no danger that the overall context is getting lost. Everything is executable and ready to be extended and embellished at will.

Naming conventions

The examples use the following naming conventions for variables:

- First-letter acronyms for individual objects: `c` for "circle," `p` for point, and so on. Append an "s" for collections: `cs` will be an array of circles, `ps` an array of points.
- Frequently occurring quantities have their own notation: pixels are denoted with `px`, scale objects with `sc`. Generators and components are function objects that "make" something and thus are called `mkr`.
- The letter `d` is used generically to indicate "the current thing" in anonymous functions. When working with D3 selections, `d` is usually an individual data point bound to a DOM element;

when working with arrays, `d` is an array element (as in `ds.map( d => +d )`).

- Data sets are called `data` or `ds`.
- Selections representing either an `<svg>` or a `<g>` element are common and, when assigned to a variable, are denoted as `svg` or `g`.

Source file organization

Beginning with Chapter 3, I adopt a convention whereby for each code listing, the page is expected to already contain an `<svg>` element with a unique `id` and with properly set `width` and `height` attributes. The example code then selects this SVG element by its `id` attribute and often assigns this selection to a variable for future reference:

```
var svg = d3.select( "#fig" );
```

This avoids the ambiguity of using a more general selector (such as `d3.select( "svg" )`) and makes it easy to include several examples in a single HTML page.

To each graph, there corresponds a JavaScript function creating the SVG elements that make up the figure dynamically. By convention, the function names begin with `make...` and continue with the value of the `id` attribute of the destination SVG element.

With the exception of the examples in Chapter 2, there is one HTML page and one JavaScript file per chapter. (With rare exceptions, I don't include JavaScript code in an HTML page directly.)

Platform, JavaScript, and browser

To run the examples, you need to run a local or hosted web server (see Appendix A). The examples should work in any contemporary, JavaScript-enabled browser. Several versions of JavaScript are currently extant.[2] With three exceptions, the code examples use only "classic" JavaScript (ES5, released in 2009/2011) without any further frameworks or libraries. The three features that require a newer version of JavaScript (ES6, released in 2015) are:

2 See *https://en.wikipedia.org/wiki/ECMAScript*.

- The concise *fat arrow notation* for anonymous functions (see Appendix C), which is used throughout the examples.
- Destructuring assignment (`[a, b] = [b, a]`), which is used in a few places.
- Several examples access remote resources using D3 wrappers for the Fetch API (see Chapter 6); these depend on the JavaScript `Promise` object.

CHAPTER 2

Let's Make Some Graphs, Already!

Let's work through some examples that will demonstrate some of the capabilities of D3, and enable you to begin solving some real-world problems immediately—if only by adapting the examples. The first two examples in this chapter will show you how to create standard *scatter* and *xy-plots*, complete with axes, from data files. The plots won't be pretty, but will be fully functional, and it should be easy to tidy up their looks and to apply the concepts to other data sets. The third example is less complete. It is mostly a demo to convince you how easy it is to include event handling and animation in your documents.

A First Example: A Single Data Set

To begin our exploration of D3, consider the small, straightforward data set in Example 2-1. Plotting this simple data set using D3 will already bring us in contact with many essential concepts.

Example 2-1. A simple data set (examples-simple.tsv)

```
x       y
100     50
200     100
300     150
400     200
500     250
```

As I already pointed out in Chapter 1, D3 is a JavaScript library for manipulating the DOM tree in order to represent information visu-

ally. This suggests that any D3 graph has at least *two* or *three* moving pieces:

- An HTML file or document, containing a DOM tree
- A JavaScript file or script, defining the commands that manipulate the DOM tree
- Frequently, a file or other resource, containing the data set

Example 2-2 shows the HTML file in its entirety.

Example 2-2. An HTML page defining an SVG element

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script src="d3.js"></script>                                    ❶
  <script src="examples-demo1.js"></script>                        ❷
</head>

<body onload="makeDemo1()">                                        ❸
  <svg id="demo1" width="600" height="300"
       style="background: lightgrey" />                            ❹
</body>
</html>
```

Yes, that's right. The HTML file is basically empty! All the action takes place in JavaScript. Let's quickly step through the few things that actually do happen in this document:

❶ First, the document loads the D3 library *d3.js*.

❷ Then, the document loads our own JavaScript file. This file contains all the commands that define the graph we are preparing.

❸ The `<body>` tag defines an `onload` event handler, which will be triggered when the `<body> element` has been completely loaded by the browser. The `makeDemo1()` event-handler function is defined in our JavaScript file *examples-demos1.js*.[1]

1 Defining an event handler via the `onload` tag is sometimes frowned upon because of the way it embeds JavaScript code in HTML. See Appendices A and C for modern alternatives.

❹ Finally, the document contains an SVG element of 600×300 pixels. The SVG element has a light-gray background to make it visible, but is otherwise empty.

The third and last piece is the JavaScript file, shown in Example 2-3.

Example 2-3. Commands for Figure 2-1

```
function makeDemo1() {                                          ❶
    d3.tsv( "examples-simple.tsv" )                             ❷
        .then( function( data ) {                               ❸ ❹
            d3.select( "svg" )                                  ❺
                .selectAll( "circle" )                          ❻
                .data( data )                                   ❼
                .enter()                                        ❽
                .append( "circle" )                             ❾
                .attr( "r", 5 ).attr( "fill", "red" )           ❿
                .attr( "cx", function(d) { return d["x"] } )    ⓫
                .attr( "cy", function(d) { return d["y"] } );
        } );
}
```

If you put these three files (the data file, the HTML page, and the JavaScript file), together with the library file *d3.js* into a common directory, and then load the page into a browser, the browser should render a graph equivalent to Figure 2-1.[2]

Let's step through the JavaScript commands and discuss them:

❶ The script defines only a single function, the `makeDemo1()` callback, to be invoked when the HTML page is fully loaded.

❷ The function loads (or "fetches") the data file, using the `tsv()` function. D3 defines several functions to read delimiter-separated-value file formats. The `tsv()` function is intended for tab-separated files.

❸ The `tsv()` function, like all functions in the JavaScript Fetch API, returns a JavaScript `Promise` object. A `Promise` is an object

2 You should be able to load the page and the associated JavaScript file by pointing the browser to the local directory. But the browser may refuse to load the data file in this way, hence it is usually necessary to run a web server when working with D3. See Appendix A for some advice in this regard.

that packages a result set and a callback, and invokes the callback when the result set is complete and ready for processing. A `Promise` provides the `then()` function to register the desired callback to invoke. (We will have more to say about JavaScript promises in "JavaScript Promises" on page 115.)

❹ The callback to be invoked when the file is loaded is defined as an anonymous function, which receives the content of the data file as argument. The `tsv()` function will return the contents of the tab-separated file as an array of JavaScript objects. Each line in the file results in one object, with property names defined through the header line in the input file.

❺ We select the `<svg>` element as the location in the DOM tree to contain the graph. The `select()` and `selectAll()` functions accept a CSS selector string (see "CSS Selectors" on page 36 in Chapter 3) and return matching nodes: `select()` only the first match, and `selectAll()` a collection of all matching nodes.

❻ Next, we select *all* `<circle>` elements inside the `<svg>` node. This may seem absurd: after all, there aren't any `<circle>` elements inside the SVG! But `selectAll("circle")` simply returns an *empty* collection (of `<circle>` elements), so this is not a problem. The odd-looking call to `selectAll()` fulfills an important function by creating a *placeholder* (the empty collection) which we will subsequently fill. This is a common D3 idiom when populating a graph with new elements.

❼ Next we associate the collection of `<circle>` elements with the data set via the call to `data(data)`. It is essential to realize that the two collections (DOM elements on the one hand, and data points on the other) are not affiliated with each other as collections "in bulk." Instead, D3 attempts to establish a one-to-one correspondence between DOM elements and data points: *each data point is represented through a separate DOM element*, which in turn takes its properties (such as its position, color, and appearance) from the information of the data point. In fact, it is a fundamental feature of D3 to establish and manage such one-to-one correspondences between individual data points and their associated DOM elements. (We will investigate this process in much more detail in Chapter 3.)

❽ The `data()` function returns a collection of elements that have been associated with individual data points. In the current case, D3 cannot associate each data point with a `<circle>` element because there aren't any circle elements (yet), and hence the collection returned is empty. However, D3 also provides access to all surplus data points that could not be matched with DOM elements through the (confusingly named) `enter()` function. The remaining commands will be invoked for each element in this "surplus" collection.

❾ First, a `<circle>` element is appended to the collection of `<circle>` elements inside the SVG that was selected on line 6.

❿ Some fixed (that is, not data-dependent) attributes and styles are set: its radius (the `r` attribute) and the fill color.

⓫ Finally, the position of each circle is chosen based on the value of its affiliated data point. The `cx` and `cy` attributes of each `<circle>` element are specified based on the entries in the data file: instead of providing a fixed value, we supply *accessor functions* that, given an entry (that is, a single-line record) of the data file, return the corresponding value for that data point.

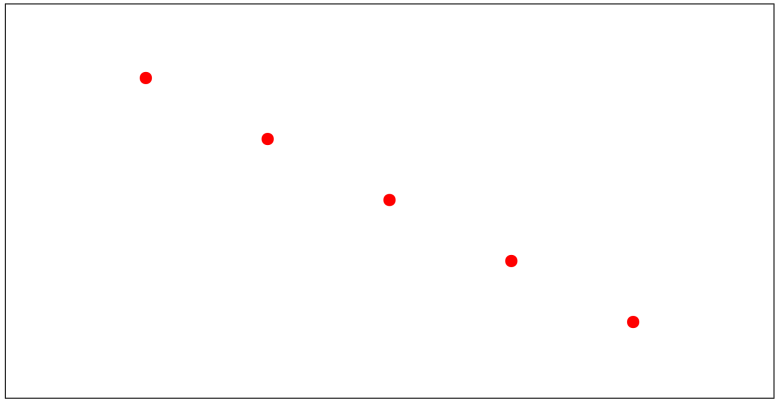


Figure 2-1. A plot of a simple data set (see Example 2-3)

To be honest, that's a remarkably painful process for such a simple graph! Already, you see here what will become ever more apparent: D3 is not a graphics library, much less a plotting tool. Instead, it is a library for manipulating the DOM tree, in order to represent infor-

mation visually. You will find yourself operating on parts of the DOM tree (via selections) all the time. You will also notice that D3 is not exactly parsimonious with keystrokes, requiring the user to manipulate attribute values and to write accessor functions one by one. At the same time, I think it is fair to say that the code, although verbose, is clean and rather straightforward.

If you actually experiment with the examples, you may encounter some additional surprises. For example, the `tsv()` function is picky: columns *must* be tab separated, whitespace is *not* ignored, the header line *must* be present, and so on. Finally, on closer inspection of the data set and the graph you should realize that the figure is actually not correct—it's upside down! That's because SVG uses "graphical coordinates" with the horizontal axis running left to right as usual, but with the vertical axis running top to bottom.

Let's recognize these initial impressions as we continue our exploration with the second example.

A Second Example: Two Data Sets

For our second example, we will use the data set in Example 2-4. It looks almost as innocuous as the previous one, but on closer inspection, it reveals some additional difficulties. Not only does this file contain *two* data sets (in columns `y1` and `y2`), but the numeric ranges of the data require attention. In the previous example, the data values could be used directly as pixel coordinates, but values in the new data set will require scaling to transform them to meaningful screen coordinates. We will have to work a little harder.

Example 2-4. A more complicated data set (examples-multiple.tsv)

```
x       y1      y2
1.0     0.001   0.63
3.0     0.003   0.84
4.0     0.024   0.56
4.5     0.054   0.22
4.6     0.062   0.15
5.0     0.100   0.08
6.0     0.176   0.20
8.0     0.198   0.71
9.0     0.199   0.65
```

Plotting Symbols and Lines

The page we use is essentially the same that we used previously (Example 2-2), except that the line:

```
<script src="examples-demo1.js"></script>
```

must be replaced with:

```
<script src="examples-demo2.js"></script>
```

referencing our new script, and the `onload` event handler must give the new function name:

```
<body onload="makeDemo2()">
```

The script itself is shown in Example 2-5, and the resulting figure is shown in Figure 2-2.

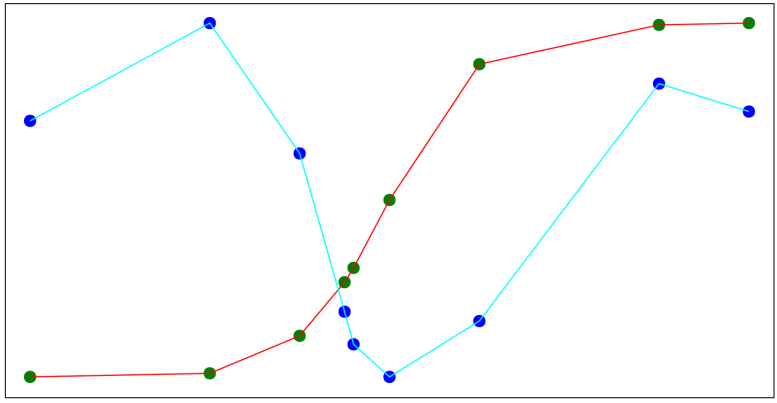


Figure 2-2. A basic plot of the data set in Example 2-4 (also see Example 2-5)

Example 2-5. Commands for Figure 2-2

```
function makeDemo2() {
    d3.tsv( "examples-multiple.tsv" )
        .then( function( data ) {
            var pxX = 600, pxY = 300;                          ❶

            var scX = d3.scaleLinear()                         ❷
                .domain( d3.extent(data, d => d["x"] ) )       ❸
                .range( [0, pxX] );
            var scY1 = d3.scaleLinear()                        ❹
                .domain(d3.extent(data, d => d["y1"] ) )
                .range( [pxY, 0] );                            ❺
            var scY2 = d3.scaleLinear()
```

```
            .domain( d3.extent(data, d => d["y2"] ) )
            .range( [pxY, 0] );

        d3.select( "svg" )                                          ❻
            .append( "g" ).attr( "id", "ds1" )                      ❼
            .selectAll( "circle" )                                  ❽
            .data(data).enter().append("circle")
            .attr( "r", 5 ).attr( "fill", "green" )                 ❾
            .attr( "cx", d => scX(d["x"]) )                         ❿
            .attr( "cy", d => scY1(d["y1"]) );                      ⓫

        d3.select( "svg" )                                          ⓬
            .append( "g" ).attr( "id", "ds2" )
            .attr( "fill", "blue" )                                 ⓭
            .selectAll( "circle" )                                  ⓮
            .data(data).enter().append("circle")
            .attr( "r", 5 )
            .attr( "cx", d => scX(d["x"]) )
            .attr( "cy", d => scY2(d["y2"]) );                      ⓯

        var lineMaker = d3.line()                                   ⓰
            .x( d => scX( d["x"] ) )                                ⓱
            .y( d => scY1( d["y1"] ) );

        d3.select( "#ds1" )                                         ⓲
            .append( "path" )                                       ⓳
            .attr( "fill", "none" ).attr( "stroke", "red" )
            .attr( "d", lineMaker(data) );                          ⓴

        lineMaker.y( d => scY2( d["y2"] ) );                        ㉑

        d3.select( "#ds2" )                                         ㉒
            .append( "path" )
            .attr( "fill", "none" ).attr( "stroke", "cyan" )
            .attr( "d", lineMaker(data) );

//          d3.select( "#ds2" ).attr( "fill", "red" );              ㉓
    } );
}
```

❶ For later reference, assign the size of the embedded SVG area to variables (px for pixel). Of course it is possible to leave the size specification out of the HTML document entirely and instead set it via JavaScript. (Try it.)

❷ D3 provides *scale objects* that map an input *domain* to an output *range*. Here we use linear scales to map the data values from their natural domain to the pixel range of the graph, but the

library also includes logarithmic and power-law scales, and even scales that map numeric ranges to colors for false-color or "heatmap" graphs (see Chapters 7 and 8). Scales are function objects: you call them with a value from the domain and they return the scaled value.

❸ Both domain and range are specified as two-element arrays. The `d3.extent()` function is a convenience function that takes an array (of objects) and returns the greatest and smallest values as a two-element array (see Chapter 10). To extract the desired value from the objects in the input array, we must supply an accessor function (similar to what we did in the final step in the previous example). To save on keystrokes, here (and in most of the following!) we make use of JavaScript's *arrow functions* (or "fat arrow notation"—see Appendix C).

❹ Because the three columns in the data set have different ranges, we need three scale objects, one for each column.

❺ For the vertical axes, we invert the output range in the definition of the scale object to compensate for the upside-down orientation of the SVG coordinate system.

❻ Select the `<svg>` element to add symbols for the first data set.

❼ This is new: before adding any graphical elements, we append a `<g>` element and give it a unique identifier. The final element will look like this:

```
<g id="ds1">...</g>
```

The SVG `<g>` element provides a logical grouping. It will enable us to refer to *all* symbols for the first data set together and to distinguish them from symbols for the second data set (see Appendix B and the sidebar "The <g> Element Is Your Friend" on page 112).

❽ As before, we create an empty placeholder collection using `selectAll( "circles" )`. The `<circle>` elements will be created as children of the `<g>` element just added.

❾ Fixed styles are applied directly to each `<circle>` element.

❿ An accessor functions picks out the appropriate column for the horizontal axis. Note how the scale operator is applied to the data before it is returned!

⓫ An accessor function picks out the appropriate column for the first data set, again scaled properly.

⓬ The familiar procedure is used again to add elements for the second data set—but note the differences!

⓭ For the second data set, the fill color is specified as a `fill` attribute on the `<g>` element; this appearance will be inherited by its children. Defining the appearance on the parent allows us to change the appearance of all children in one fell swoop at a later time.

⓮ Again, an empty collection is created to hold the newly added `<circle>` elements. This is where the `<g>` parent element is more than a convenience: if we would invoke `selectAll( "circle" )` on the `<svg>` element at this point, we would not obtain an empty collection, but instead receive the `<circle>` elements from the *first* data set. Instead of adding new elements, we would modify the existing ones, overwriting the first data set with the second. The `<g>` element allows us to distinguish clearly between elements and their mapping to data sets. (This will become clearer once we will have studied the D3 `Selection` abstraction systematically in Chapter 3.)

⓯ The accessor function now picks out the appropriate column for the second data set.

⓰ To distinguish the two data sets more clearly, we want to connect the symbols belonging to each data set with straight lines. Lines are more complicated than symbols because each line (line segment) depends on *two* consecutive data points. D3 helps us here: the `d3.line()` factory function returns a function object, which, given a data set, produces a string suitable for the `d` attribute of the SVG `<path>` element. (See Appendix B to learn more about the `<path>` element and its syntax.)

⓱ The line generator requires accessor functions to pick out the horizontal and vertical coordinates for each data point.

⓲ Select the `<g>` element for the first data set by the value of its `id` attribute. An ID selector string consists of the hashmark `#`, followed by the attribute value.

⓳ A `<path>` element is added as child of the `<g>` group of the first data set…

⓴ … and its `d` attribute is set by invoking the line generator on the data set.

㉑ Instead of creating a new line generator from scratch, we reuse the existing one by specifying a new accessor function, this time for the second data set.

㉒ A `<path>` element for the second data set is appended at the appropriate place in the SVG tree and populated.

㉓ Because the fill style for the symbols of the second data set was defined on the parent element (not on the individual `<circle>` elements themselves) it is possible to change it in a single operation. Uncomment this line to make all circles for the second data set red. Only appearance options are inherited from the parent: it is not possible to change, for example, the radius of all circles in this way, or to turn circles into rectangles. These kinds of operations require touching every element individually.

At this point, you might begin to get a feel for what working with D3 is like. Sure, it is verbose, but much of it feels like an assembly-type job where you simply snap together premade components. The method chaining, in particular, can resemble the construction of a Unix pipeline (or playing with LEGO, for that matter). The components themselves tend to emphasize mechanism over policy: that makes them reusable across a wide range of intended purposes, but leaves a greater burden on the programmer or designer to create semantically meaningful graphical assemblies. One final aspect that I would like to emphasize is that D3 tends to defer decisions in favor of "late binding": for example, in the way that it is customary to pass accessor functions as arguments, rather than requiring that the

appropriate columns be extracted from the original data set before being passed to the rendering framework.

Adding Graph Elements with Reusable Components

Figure 2-2 is bare-bones: it shows the data but nothing else. In particular, it doesn't show the scales—which is doubly important here, because the two data sets have quite different numerical ranges. Without scales or axes it is not possible to read quantitative information from the graph (or any graph, for that matter). We therefore need to add axes to the existing graph.

Axes are complex graphical elements because they must manage tick marks and tick labels. Thankfully, D3 provides an axis facility that will, given a scale object (which defines domain and range and the mapping between them), generate and draw all the required graphical elements. Because the visual axis component consists of many individual SVG elements (for the tick marks and labels), it should always be created inside its own `<g>` container. Styles and transformations applied to this parent element are inherited by all parts of the axis. This is important because all axes are initially located at the origin (the upper-left corner) and must be moved using the `transform` attribute to their desired location. (Axes will be explained in more detail in Chapter 7.)

Besides adding axes and new D3 functionality for generating curves, Example 2-6 also demonstrates a different style of working with D3. The code in Example 2-5 was very straightforward, but also verbose, and included a great deal of code replication: for example, the code to create the three different scale objects is almost identical. Similarly, the code to create symbols and lines is mostly duplicated for the second data set. The advantage of this style is its simplicity and linear logical flow, at the cost of higher verbosity.

In Example 2-6 there is less redundant code because duplicated instructions have been pulled into local functions. Because these functions are defined inside of `makeDemo3()`, they have access to the variables in that scope. This helps to keep the number of arguments required by the local helper functions small. This example also introduces *components* as units of encapsulation and reuse, and demonstrates "synthetic" function invocation—all important techniques when working with D3.

Example 2-6. Commands for Figure 2-3

```
function makeDemo3() {
    d3.tsv( "examples-multiple.tsv" )
        .then( function( data ) {
            var svg = d3.select( "svg" );                                    ❶

            var pxX = svg.attr( "width" );                                   ❷
            var pxY = svg.attr( "height" );

            var makeScale = function( accessor, range ) {                    ❸
                return d3.scaleLinear()
                    .domain( d3.extent( data, accessor ) )
                    .range( range ).nice();
            }
            var scX  = makeScale( d => d["x"],  [0, pxX] );
            var scY1 = makeScale( d => d["y1"], [pxY, 0] );
            var scY2 = makeScale( d => d["y2"], [pxY, 0] );

            var drawData = function( g, accessor, curve ) {                  ❹
                // draw circles
                g.selectAll( "circle" ).data(data).enter()
                    .append("circle")
                    .attr( "r", 5 )
                    .attr( "cx", d => scX(d["x"]) )
                    .attr( "cy", accessor );

                // draw lines
                var lnMkr = d3.line().curve( curve )                         ❺
                    .x( d=>scX(d["x"]) ).y( accessor );

                g.append( "path" ).attr( "fill", "none" )
                    .attr( "d", lnMkr( data ) );
            }

            var g1 = svg.append( "g" );                                      ❻
            var g2 = svg.append( "g" );

            drawData( g1, d => scY1(d["y1"]), d3.curveStep );                ❼
            drawData( g2, d => scY2(d["y2"]), d3.curveNatural );

            g1.selectAll( "circle" ).attr( "fill", "green" );                ❽
            g1.selectAll( "path" ).attr( "stroke", "cyan" );

            g2.selectAll( "circle" ).attr( "fill", "blue" );
            g2.selectAll( "path" ).attr( "stroke", "red" );

            var axMkr = d3.axisRight( scY1 );                                ❾
            axMkr( svg.append("g") );                                        ❿

            axMkr = d3.axisLeft( scY2 );
```

```
            svg.append( "g" )
                .attr( "transform", "translate(" + pxX + ",0)" )  ⓫
                .call( axMkr );                                   ⓬

            svg.append( "g" ).call( d3.axisTop( scX ) )
                .attr( "transform", "translate(0,"+pxY+")" );     ⓭
        } );
}
```

❶ Select the `<svg>` element to draw on and assign it to a variable so that it can be used later without having to call `select()` again.

❷ Next, query the `<svg>` element for its size. Many D3 functions can work as setters as well as getters: if a second argument is supplied, the named property is set to the specified value, but if the second argument is missing, the current value of the property is returned instead. We use this feature here to obtain the `<svg>` element's size.

❸ The `makeScale()` function is simply a convenient wrapper to cut down on the relative verbosity of the D3 function calls. Scale objects are already familiar from the previous listing (Example 2-5). The `nice()` function on a scale object extends the range to the nearest "round" values.

❹ The `drawData()` function bundles all commands necessary to plot a single data set: it creates both the circles for individual data points as well as the lines connecting them. The first argument to `drawData()` must be a `Selection` instance; typically, a `<g>` element as container for all the graphical elements representing a single data set. Functions that take a `Selection` as their first argument and then add elements to it are known as *components* and are an important mechanism for encapsulation and code reuse in D3. This is the first example we see; the axis facility later in this example is another. (See Chapter 5 for the full discussion.)

❺ The `d3.line()` factory is already familiar from Example 2-5. It can accept an algorithm that defines what kind of curve should be used to connect consecutive points. Straight lines are the default, but D3 defines many other algorithms as well—you can also write your own. (See Chapter 5 to learn how.)

❻ Create the two `<g>` container elements, one for each data set.

❼ Invoke the `drawData()` function while supplying one of the container elements, as well as an accessor describing the data set and the desired curve shape. Just to demonstrate what capabilities are available, the two data sets are drawn using different curves: one with step functions and the other with natural cubic splines. The `drawData()` function will add the necessary `<circle>` and `<path>` elements to the `<g>` container.

❽ For each container, select the desired graphical elements to set their color. Although it would have been easy enough to do so, choosing the color was quite intentionally *not* made part of the `drawData()` function. This reflects a common D3 idiom: creating DOM elements is kept separate from configuring their appearance options!

❾ The axis for the first data set is drawn on the left side of the graph; remember that by default all axes are rendered at the origin. The `axisRight` object draws tick labels on the *right* side of the axis, so that they are outside the graph if the axis is placed on the graph's right side. Here, we use it on the left side and allow for the tick labels to be inside the graph.

❿ The factory function `d3.axisRight( scale )` returns a function object that generates the axis with all its parts. It requires an SVG container (typically a `<g>` element) as argument, and creates all elements of the axis as children of this container element. In other words, it is a *component* as defined earlier. (See Chapter 7 for details.)

⓫ For the axis on the right side of the graph, the container element must be moved to the appropriate location. This is done using the SVG `transform` attribute.

⓬ This is new: instead of calling the `axMkr` function explicitly with the containing `<g>` element as argument, the `axMkr` function is passed as argument to the `call()` function instead. The `call()` function is part of the `Selection` API (see Chapter 3); it is modeled after a similar facility in the JavaScript language. It invokes its argument (which must be a function), while supplying the

current selection as argument. This form of "synthetic" function invocation is quite common in JavaScript, and it is a good idea to get used to it. One advantage of this style of programming is that it supports method chaining, as you can see in the next item…

⓭ … where we add the horizontal axis at the bottom of the graph. Just to show what's possible, the order of function calls has been interchanged: the axis component is invoked first, the transformation is applied second. At this point we have also dispensed with the ancillary `axMkr` variable. This is probably the most idiomatic way to write this code.[3]

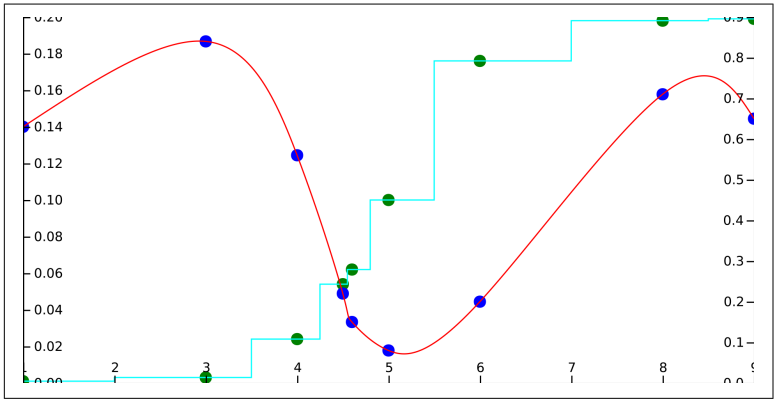


Figure 2-3. An improved plot, including coordinate axes and using different types of curves (compare Figure 2-2 and Example 2-6)

The resulting graph is shown in Figure 2-3. It is not pretty, but fully functional, showing two data sets together with their respective scales. The first step to improve the appearance of the graph would be to introduce some padding inside the SVG area, around the actual data, to make room for the axes and tick labels. (Try it!)

3 In fact, the `drawData()` function would typically be called this way, too: `g1.call( drawData, d => scY1(d["y1"]), d3.curveStep )`

A Third Example: Animating List Items

Our third and last example may appear a bit more whimsical than the other two, but it nevertheless demonstrates two important points:

- D3 is not limited to generating SVG. On the contrary, it can manipulate *any* part of the DOM tree. In this example, we will use D3 to manipulate ordinary HTML list elements.
- D3 makes it easy to create responsive and animated documents; that is, documents that can respond to user events (such as mouse clicks) and that change their appearance over time. Here, we will only give the briefest preview of the amazing possibilities; we will revisit the topic in much more detail in Chapter 4.

Creating HTML Elements with D3

For a change, and because the actual script is very short, the JavaScript commands are included directly in the page and are not kept in a separate file. The entire page, including the D3 commands, for the current example is shown in Example 2-7 (also see Figure 2-4).

Example 2-7. Using D3 for nongraphical HTML manipulation (see Figure 2-4)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script src="d3.js"></script>
  <script>
function makeList() {
    var vs = [ "From East", "to West,", "at Home", "is Best" ];    ❶

    d3.select( "body" )                                           ❷
        .append( "ul" ).selectAll( "li" )                         ❸
        .data(vs).enter()                                         ❹
        .append("li").text( d => d );                             ❺
}
  </script>
</head>

<body onload="makeList()" />
</html>
```

Structurally, this example is almost equivalent to the example that opened this chapter (in particular Example 2-3), but with a few differences in detail:

❶ The data set is not loaded from an external file but is defined inside the code itself.

❷ The HTML `<body>` element is selected as the outermost container of interest.

❸ The code appends an `<ul>` element and then creates an empty placeholder for the list items (using `selectAll( "li" )`).

❹ As before, the data set is bound to the selection, and the collection of data points without matching DOM elements is retrieved.

❺ Finally, a list item is appended for each data point, and its contents (which, in this example, is the text for the list item) are populated with values from the data set.

All of this is quite analogous to what we have done before, except that the resulting page is plain, textual HTML. D3 turns out to be a perfectly good tool for generic (that is, nongraphical) DOM manipulations.

Figure 2-4. A bulleted list in HTML (see Examples 2-7 and 2-8)

Creating a Simple Animation

It does not take much to let this document respond to user events. Replace the `makeList()` function in Example 2-7 with the one in Example 2-8. You can now toggle the color of the text from black to red and back by clicking on a list item. Moreover, the change won't take effect immediately; instead, the color of the text will change continuously over a couple of seconds.

Example 2-8. Animation in response to user events

```
function makeList() {
    var vs = [ "From East", "to West,", "at Home", "is Best" ];

    d3.select( "body" )
        .append( "ul" ).selectAll( "li" )
        .data(vs).enter()
        .append("li").text( d => d )                                    ❶
        .on( "click", function () {                                     ❷
            this.toggleState = !this.toggleState;                       ❸ ❹
            d3.select( this )                                           ❺
                .transition().duration(2000)                            ❻
                .style("color", this.toggleState?"red":"black");        ❼
        } );
}
```

❶ Up to this point, the function is identical to the one from Example 2-7.

❷ The `on()` function registers a callback for the named event type (`"click"` in this case), with the current element as the DOM `EventTarget`. Each list item can now receive click events and will pass them to the supplied callback.

❸ We need to keep track of the current toggle state separately for each list item. Where else to keep this information than on the element itself? JavaScript's permissive nature makes this supremely easy: simply add a new member to the element! D3 assigns the active DOM element to `this` before invoking the callback, and so provides access to the current DOM element.

❹ This line makes use of JavaScript's permissive nature in another way as well. The first time the callback is invoked (for each list item), the `toggleState` has not been assigned yet. It therefore has the special value `undefined`, which evaluates to `false` in a Boolean context, making it unnecessary to initialize the variable explicitly.

❺ In order to operate on it using method chaining, the current node needs to be wrapped in a selection.

❻ The `transition()` method interpolates smoothly between the current state of the selected elements (the current list item, in

this case) and its desired final appearance. The transition interval is specified as 2000 milliseconds. D3 can interpolate between colors (via their numerical representation) and many other quantities. (Interpolations will be discussed in Chapter 7.)

❼ Finally, the new text color is selected based on the current state of the status variable.

CHAPTER 3

The Heart of the Matter: Selecting and Binding

D3 is a JavaScript library for manipulating the DOM tree in order to represent information visually. This makes it different from other graphics or plotting libraries: a conventional graphics library operates on a "canvas" and places lines, circles, and other graphical objects directly onto this canvas. But because D3 uses the DOM tree to display information, it must provide capabilities to operate on the DOM tree—in addition to the customary management of shapes, coordinates, colors, and so on. Specifically, it must allow the user to:

1. Specify where in the DOM tree a change should take place and which elements will be affected; the user must be able to *select* a node (or set of nodes).
2. Associate individual records from the data set with specific elements or nodes in the DOM tree; enable the user to *bind* or *join* a data set to a selection of nodes.
3. Change the size, position, and appearance of DOM elements according to the values of the data associated with them.

The first and last item in this list are common activities in contemporary web development, and users familiar with the jQuery library, for instance, should feel quite at home. (But if you are *not* familiar with jQuery and the particular style of programming popularized by it, D3 can seem very peculiar indeed!)

The second item, however, is different. The idea to establish a tight association between individual data records and individual DOM elements, so that the appearance of the DOM elements can change according to the data bound to them, seems fairly unique to D3. This idea, and the particular way it is implemented, is central to D3.

The operations to select nodes, bind data to them, and update their appearance are bundled in the `Selection` abstraction. A thorough understanding of its concepts and capabilities is essential to be productive with D3. (The `Selection` abstraction also contains functionality to associate event handlers with DOM elements. We will treat this topic in Chapter 4 in the context of animating graphics.)

Selections

Selections are ordered collections of DOM elements, wrapped in a `Selection` abstraction. The `Selection` abstraction provides an API to query and modify the elements it contains. The `Selection` API is declarative and supports method chaining, making it possible to manipulate the DOM tree without explicit looping over individual nodes.

Creating Selections

You typically obtain an initial selection instance by using one of the selection functions that operate on the global `d3` object (see Table 3-1). You can then create subsets of this initial selection using member functions that operate on a `Selection` instance (see Table 3-2).

Selection methods accept a CSS selector string (see "CSS Selectors" on page 36) and return a collection of matching DOM elements in document order. The `select(selector)` functions return only the *first* matching element, while the `selectAll(selector)` methods return *all* matching elements. Both functions return an empty collection if no matching elements are found or if the selector is `null` or `undefined`. (Using CSS selector strings as selection criterion is the common case; we will discuss some additional options later in this chapter.)

Table 3-1. Global functions to create selections

Function	Description
`d3.select(selector)`	• Searches the entire document and returns a `Selection` containing the first element matching the selector. Returns an empty selection if no elements match the selector, or if the selector is `null` or `undefined`. • The selector can either be a CSS selector string or a `Node` object.
`d3.selectAll(selector)`	• Searches the entire document and returns a `Selection` of all elements matching the selector, in document order. Returns an empty selection if no elements match the selector, or if the selector is `null` or `undefined`. • The selector can either be a CSS selector string or an array of `Node` objects.
`d3.create(name)`	Given the name of an element, creates a `Selection` containing a detached element of the given name in the current document.

As already indicated, selection methods nest when chained: any subsequent selection action will only act on the results of the previous one. (Technically, the previous action returns a new selection object, which becomes the target for the next action, and so on.) For example, the following code (taken straight from the D3 Reference Documentation) will select the first bold element in all paragraphs in the document:

```
bs = d3.selectAll( "p" ).select( "b" );
```

whereas the following snippet will select all `<circle>` elements inside the element with the ID `id123`:

```
cs = d3.select( "#id123" ).selectAll( "circle" );
```

You can obtain an initial selection by querying the global `d3` object, which amounts to searching the entire document. Of course, `Selection` objects can be assigned to variables (as in the snippets just shown), passed to functions (as in Example 2-6), and so on.

Table 3-2. Methods to create subselections from selections (sel is a Selection object)

Function	Description
`sel.select(selector)`	• Searches each element of the original selection for matches and returns a `Selection` containing the first element matching the selector. Returns an empty selection if no elements match the selector, or if the selector is `null` or `undefined`. • The selector can either be a CSS selector string, or an accessor function. The accessor must return a DOM `Element` instance or `null` if no match is found. • Data bound to elements in the original selection is *retained* by the elements returned from this function.
`sel.selectAll(selector)`	• Searches each element of the original selection for matches and returns a `Selection` of all elements matching the selector. Returns an empty selection if no elements match the selector, or if the selector is `null` or `undefined`. • The selector can either be a CSS selector string or an accessor function. The accessor must return an array of DOM `Element` instances or an empty array if there are no matches. • Results are returned in source order (original selection). • Data bound to the elements in the source is *not* propagated to the elements in the returned selection.
`sel.filter(selector)`	Similar to `sel.selectAll(selector)`. The selector can be a CSS selector string or an accessor function. If it is an accessor, then it must return a Boolean that indicates whether the current node should be retained in the selection or not.

Three types of selectors can be used with the `select()` and `selectAll()` functions:

- A CSS selector string (see "CSS Selectors" on page 36). This is the common case.
- The selection functions on the global `d3` object accept either a node (for `select()`) or a collection of nodes (for `selectAll()`).
- When selection functions are invoked on a `Selection` object, they can accept an accessor function as the selector. For `select()`, this function must return a DOM `Element` instance or `null` if there is no match; for `selectAll()`, it must return an array of elements, possibly empty if there are no matches; for

`filter()`, it must return a Boolean value indicating whether the current element should be retained.

Using a node as the selector may seem strange: why call `select()` if I already have a node in hand? If this is the case, then you call `select()` not to select a node, but as a convenient way to wrap the node in a `Selection` object, to make the `Selection` API available for it. This is commonly done inside of event handlers, when you want to bring the `Selection` API to work on the current node that has received the event. (You already saw an example of this in Example 2-8; further examples can be found in Figures 4-2 and 4-4.)

Understanding Selections

Most D3 programming begins with making a selection of elements from a document; this selection will then be modified. Selections are therefore the base material when working with D3. Two questions present themselves at this point:

- What actually is a selection?
- What can I do with a selection?

Technically, a `Selection` is a JavaScript wrapper around an ordered collection of DOM elements together with a set of methods to manipulate this collection. The API is declarative and supports method chaining, hence it is generally unnecessary to handle the elements of the collection explicitly.

Conceptually, a selection is a handle on (all or part of) the DOM tree, together with a set of operations that constitute the three activities listed earlier: selecting elements, binding data, and modifying appearance and behavior.

It is probably best to think of selections as opaque abstraction and operate on them only through the provided API. Besides functions to create selections based on specified criteria, most operations in the `Selection` API can be grouped into two sets: those that operate on the *elements* of a selection (for example, by changing the attributes of an element), and those that operate on the entire *selection* itself (for example, by adding or removing elements). Last, but not least, there is a small set of operations to manage the association

between a data set and the elements of a selection. We will turn to that topic next.

CSS Selectors

CSS selectors are a mechanism for specifying parts of the DOM tree. Like some other web technologies, the basics are simple enough, but the far reaches of the spec can appear arcane. Here, I focus only on those parts that seem the most relevant when working with D3 (as opposed to web development in general).

There are several different types of selectors:

`type`
: Type or element selectors consist of the (not case-sensitive) tag name, such as `p` or `svg`. A special case is the *universal* selector `*` (mostly useful as part of selector combinations, such as `div *`, meaning *all* descendants of `<div>` elements).

`#id`
: Selects a single element by the value of the element's `id` attribute. To form a selector, the attribute value is preceded by a hashmark `#`. The attribute value must be *unique* within the document.

`.class`
: Selects elements based on the value of their `class` attribute. Multiple elements in a document can have the same class, and a single element can have multiple classes (separated by whitespace in the `class` attribute). To form a selector, the class name is prepended by a dot.

`:pseudo-class`
: Pseudo-classes select elements based on their *state* or *position* within the document. Several dozen pseudo-classes are recognized. Some of the better known ones are `:hover` and `:visited` (the latter only for links). Important for us are pseudo-classes that match based on the position of an element within the document, such as `:first-child`, `:first-of-type`, and so on. Positions can be constants, the key words `even` and `odd`, or even a combination of offset and increment. Pseudo-class selectors are always preceded by a single colon.

`[attribute]`
: Elements can be selected based on the presence of attributes and their values. For example, `"[width]"` selects elements having a `width` attribute, whereas `"[width='600']"` selects only elements with a width of 600. Special syntax allows matching down to substrings of the attribute values. Attribute selectors are enclosed in brackets.

`::pseudo-element`
: Allows selecting *parts* of an element, such as `::first-line`. Pseudo-element selectors are preceded by a double colon.

Selectors can be *combined*. The semantics of the combination depends on on the form of the combination:

- Class and pseudo-class selectors can be combined with any of the type selectors: `p.nav` or `g:first-child` (logical `AND`).
- Selectors can be joined with a comma into "groups." A group matches any element that matches at least one of the group elements (logical `OR`).
- Selectors separated by a *whitespace* form a descendant relationship: `p b` matches any bold element that is a direct or indirect descendant of a paragraph element.
- Selectors can be joined using one of the "combinators" `>`, `+`, or `~` to express more specific child or sibling relationships. For example, `p > b` matches only bold elements that are a *direct child* of a paragraph element. Similarly, `+` matches the *next* sibling, and `~` matches *any* sibling.

Most important for our purposes are type and ID selectors. (Don't forget to include the hashmark `#` when attempting to select on an ID!) Classes can be used to "tag" certain elements so that they can be identified later. The D3 axis facility makes use of this (see Figure 7-2; also see Examples 5-8 and 7-2 for more).

But don't forget pseudo-classes, either: they can be remarkably convenient. You may also want to consider attribute-based selectors to keep state directly within HTML or SVG elements, rather than in a separate data structure. Selector combinations are less commonly used when working with D3, because nested selections provide a more idiomatic alternative (for example, `d3.selectAll( "p" ).selectAll( "b" )`).

The Mozilla Development Network (MDN) provides a more comprehensive guide to selectors on their site, MDN Introduction to CSS Selectors, as well as a helpful reference (including all pseudo-classes) at MDN CSS Selector Reference.

Binding Data

The `data()` method accepts an array of arbitrary values or objects and attempts to establish a one-to-one correspondence between the entries of this array and the elements in the current selection. (Remember that `data()` must be called through a `Selection` object, which defines the "current selection.")

Unless a key has been provided (described later), the `data()` function will attempt to match up data entries and selection elements by their *position* in their respective containers: the first data point with the first selected DOM element, the second data point with the second DOM element, and so on (see Figure 3-1). There is no requirement that the number of elements in both collections must match; in fact, they commonly do not. If the numbers do not agree, there will either be a surplus of data points, or a surplus of DOM elements. (We will come back to this point.)

If a data point has been associated with a DOM element, the data point itself is stored in the `__data__` property of the selection element. The relationship between data point and selection element is therefore persistent and will continue until explicitly overwritten (by calling `data()` again with a different data set as argument). Because the data point is stored inside the DOM element, the data is available to the methods that modify the attributes and appearance of the DOM element.

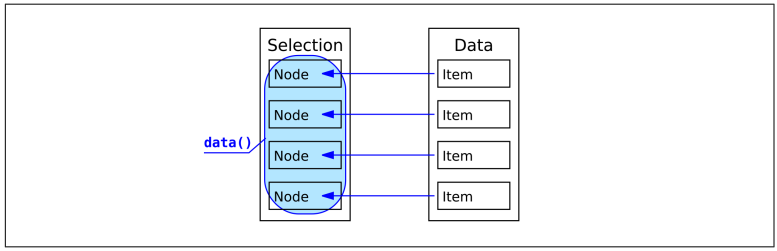


Figure 3-1. Binding data to a selection when there are as many data points as DOM elements

The `data()` method returns a new `Selection` object containing those elements that were successfully bound to entries in the data set. The `data()` method also populates the so-called "enter" and "exit" selections, which contain the unmatched (surplus) data points or DOM elements, respectively (see Table 3-3).

Unmatched Items: The Enter and Exit Selections

If the number of data points and DOM elements is not equal, there will be a surplus of unmatched items, either of data points or DOM elements, but not both. Because items are matched starting at the beginning, the unmatched items will always be the trailing items in their respective collections. If items are joined on a key (see the next section), matching can fail in additional ways.

After a call to `data(data)`, collections of any unmatched items are accessible through the `enter()` and `exit()` methods (see Figure 3-2).[1] Without a preceding call to `data()`, these methods return empty collections. The `exit()` method actually returns a `Selection` of DOM elements, but the `enter()` method only returns a collection of placeholder elements (by construction, there can be no actual nodes for surplus data points).

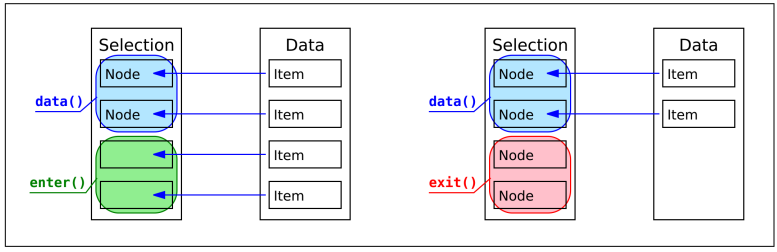


Figure 3-2. Binding data to a selection when the number of data points does not agree with the number of elements in the selection

1 The names of these functions can be understood if you think of binding as going through three phases. During the "entry" phase, DOM elements are created for any unmatched data items. During the "update" phase, DOM elements are styled based on the data bound to them. During the "exit" phase, surplus DOM elements (that have no data bound to them) are removed from the graph. If you get confused, just remember that `enter()` returns the surplus data items and `exit()` returns the surplus DOM elements.

The collections of surplus items can be used to create or remove items as necessary to make the graph match up with the data set. Given a set of surplus data items (as returned by `enter()`), you can create the needed DOM elements:

```
d3.select( "svg" ).selectAll( "circle" )
    .data( data ).enter()
    .append( "circle" ).attr( "fill", "red );
```

Similarly, given a selection of surplus DOM elements (as returned by `exit()`), you can remove these items as follows:

```
d3.select( "svg" ).selectAll( "circle" )
    .data( data ).exit()
    .remove();
```

The `data()` method itself returns the `Selection` of DOM elements that were successfully bound to data points; it can be used to update the appearance of these DOM elements. All three activities are used together in the *General Update Pattern* that will be discussed later in this chapter.

Table 3-3. Methods to bind data to a selection (sel is a Selection object)

Function	Description
`sel.data( data, key )`	• Establishes a one-to-one relationship between the items in the supplied array of arbitrary values (the data set) and the current selection. If no `key` has been provided, matching is done in order; otherwise, items are bound to nodes with a matching key. • Returns a selection of nodes with data bound to them. • If called without arguments, the function returns an array of data items in the current selection.
`sel.enter()`	• Returns a selection of "placeholder" items for those data items that were not bound to selection elements in a previous call to `data(data)`. • Returns an empty selection when called without a preceding call to `data(data)`.
`sel.exit()`	• Returns a selection of those DOM elements that were not bound to data items in a previous call to `data(data)`. • Returns an empty selection when called without a preceding call to `data(data)`.

Joining on a Key

Sometimes joining data and nodes by simply aligning them in order is not enough. In particular when updating an *existing* set of nodes with *new* data, it may matter that the *correct* DOM elements receive new values.

Figure 3-3 shows an example: when the user clicks into the graph, the positions of the circles are updated with new data. Because the graph uses a smooth, animated transition to move the circles from their old positions to new ones, it is important that each circle receives the new data associated with it. Example 3-1 shows the commands to create Figure 3-3.

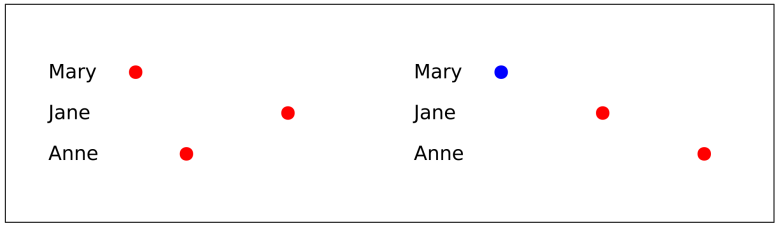


Figure 3-3. Updating a graph with new data: the appearance of the graph before the update (left) and after (right) (see Example 3-1)

Example 3-1. Commands for Figure 3-3

```
function makeKeys() {
    var ds1 = [["Mary", 1], ["Jane", 4], ["Anne", 2]];                    ❶
    var ds2 = [["Anne", 5], ["Jane", 3]];                                 ❷

    var scX = d3.scaleLinear().domain([0, 6]).range([50, 300]),
        scY = d3.scaleLinear().domain([0, 3]).range([50, 150]);

    var j = -1, k = -1;                                                   ❸

    var svg = d3.select( "#key" );                                        ❹

    svg.selectAll( "text" )                                               ❺
        .data(ds1).enter().append( "text" )
        .attr( "x", 20 ).attr( "y", d=>scY(++j) ).text( d=>d[0] );

    svg.selectAll("circle").data(ds1).enter().append( "circle" )          ❻
        .attr( "r", 5 ).attr( "fill", "red" )
        .attr( "cx", d=>scX(d[1]) ).attr( "cy", d=>scY(++k)-5 );

    svg.on( "click", function() {
        var cs = svg.selectAll( "circle" ).data( ds2, d=>d[0] );          ❼
```

```
        cs.transition().duration(1000).attr("cx", d=>scX(d[1]) ); ❽
        cs.exit().attr( "fill", "blue" );                         ❾
    } );
}
```

❶ The original data set.

❷ The new data set. Note that it is incomplete: only two out of the three items will be updated with new data. Furthermore, the order of the items that are present is different than in the original data set.

❸ Integers to track the vertical position of the text label and circle.

❹ The active `<svg>` element as `Selection`, assigned to a variable for future reference.

❺ Create the text labels…

❻ … and the circles at their initial positions.

❼ Inside the `click` event handler, the new data set is bound to the selection of `circle` elements. Notice the second argument to the `data()` function: this function defines the *key* on which data items will be joined.

❽ A smooth, animated transition from the old positions to the new ones.

❾ The `exit()` selection is now populated with Mary's node, since in the last call to `data()`, no data point was bound to this node. We give this circle a different color to make it stand out.

This example shows how joining on a key is accomplished: you simply supply a second argument to the `data(data, key)` function. This additional argument must be an accessor function, which returns the desired key value as a string for each node or data point. This function will be evaluated for all items in the data set *and* for the data points bound to all nodes in the current selection, and items with matching keys will be bound to each other. Nonmatching items, either in the data set or the selection, populate the `enter()` and `exit()` selections as usual. If there are *duplicate keys*, either in

the data set or the selection, only the first occurrence of the key (in collection order) is bound. Duplicates in the data set are placed in the `enter()` selection, duplicates in the current selection end up in the `exit()` selection. (See Figure 3-4.)

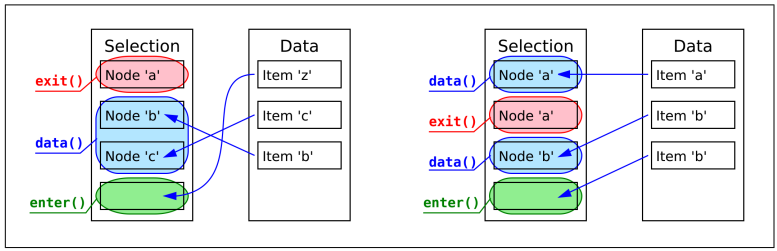


Figure 3-4. Binding data on a nontrivial key. All three of the data(), enter(), and exit() selections may be populated (left). If there are duplicate keys, either in the data set or selection, the corresponding items are placed in the enter() or exit() selections, respectively (right).

The General Update Pattern

A particular situation arises when an existing graph must be *updated* repeatedly with new data—for example, because data is only becoming available over time or because the graph must respond to user input. In this case, it is not enough to simply create additional graph elements corresponding to new inputs; the new elements must also be *merged back* with the existing elements to get the whole graph ready for the next iteration. The complete sequence of steps is therefore:

1. Bind new data to an existing selection of elements.
2. Remove any surplus items that do not have matching data associated anymore (the `exit()` selection).
3. Create and configure all items associated with data points that did not exist before (the `enter()` selection).
4. Merge the remaining items from the original selection with the newly created items from the `enter()` selection.
5. Update all items in the combined selection based on the current values of the bound data set.

The example in Example 3-2 defines two data sets; clicking into the graph area replaces the current data set with the other one and updates the graph accordingly.

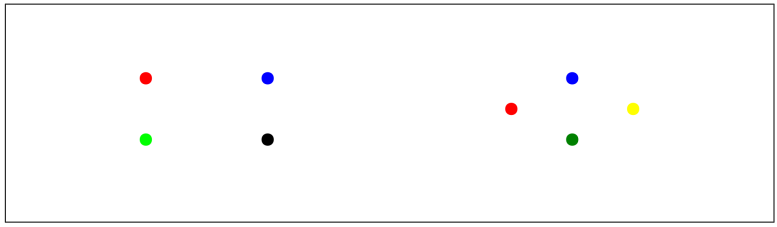


Figure 3-5. The General Update Pattern: the appearance of the graph before the click (left) and after (right) (see Example 3-2)

Example 3-2. Commands for Figure 3-5. The lines marked 6-10 constitute the General Update Pattern.

```
function makeUpdate() {
    var ds1 = [ [2, 3, "green"], [1, 2, "red"], [2, 1, "blue"],          ❶
                [3, 2, "yellow"] ];
    var ds2 = [ [1, 1, "red"], [3, 3, "black"], [1, 3, "lime"],
                [3, 1, "blue"]];

    var scX = d3.scaleLinear().domain([1, 3]).range([100, 200]),         ❷
        scY = d3.scaleLinear().domain([1, 3]).range([50, 100]);

    var svg = d3.select( "#update" );                                    ❸

    svg.on( "click", function() {                                        ❹
        [ ds1, ds2 ] = [ ds2, ds1 ];                                     ❺

        var cs = svg.selectAll( "circle" ).data( ds1, d=>d[2] );         ❻

        cs.exit().remove();                                              ❼

        cs = cs.enter().append( "circle" )                               ❽
            .attr( "r", 5 ).attr( "fill", d=>d[2] )
            .merge( cs );                                                ❾

        cs.attr( "cx", d=>scX(d[0]) ).attr( "cy", d=>scY(d[1]) );        ❿
    } );

    svg.dispatch( "click" );                                             ⓫
}
```

❶ The two data sets. Each entry consists of the *x* and *y* coordinates, followed by the color. We will also use this color string as key when binding the data.

❷ Scales to map the data values to screen coordinates.

❸ Obtain a convenient handle on the `<svg>` element.

❹ Register a `click` event handler for this `<svg>` element. All relevant action will take place inside this event handler.

❺ In response to a user click, swap the data sets, replacing the current one with its alternate.

❻ *Bind* the new data set to the (existing) `<circle>` elements in the graph, using the color name as key.

❼ *Remove* those elements that are no longer bound to data (the `exit()` selection).

❽ *Create* new elements for those data points that are new in this data set (the `enter()` selection).

❾ *Merge* the existing elements retained from the earlier selection into the selection of newly created elements, and treat the combination as the "current" selection going forward.

❿ *Update* all the elements in the combined selection using the bound data values.

⓫ This statement generates a synthetic `click` event. This triggers the event handler and therefore populates the graph when the page is first loaded. (We will revisit events in Chapter 4.)

The purpose of the `merge()` function may not be immediately apparent. But keep in mind that the enter and update selections (as returned by `enter()` and `data()`, respectively) are *separate* selections. If we want to operate on their elements without duplicating code, then we must combine them first. Moreover, a plain concatenation of the enter and update selections would not be appropriate;

the `merge()` operation is designed to operate on the particular data representation of these two selections.[2]

Manipulating Selections

The `Selection` abstraction contains many operations to either operate on the elements of a selection individually, or to manipulate the overall selection itself.

Accessor Functions

In many places, D3 methods will accept a function instead of a value. Typically, these functions are invoked for each item in the selection (or collection) and have access to the item itself and often some other information. They must return a value that is acceptable in the place where they are called. In this way, it is possible to compute a value dynamically for each item.

An example will make this clear. To make every element in a selection red, we simply say:

```
sel.attr( "fill", "red" );
```

But if we want to choose the color dynamically, for example, based on the data bound to the element, we could say:

```
sel.attr( "fill",
          function(d, i, ns) {return d<0 ? "red":"green"} );
```

Or, using "fat arrow notation," which is particularly convenient for this purpose:

```
sel.attr( "fill", d => d<0 ? "red" : "green" );
```

2 It may aid comprehension to provide a rough sketch of what is happening under the covers. The exit selection contains an array having the same number of elements as the old data set, but with only those entries populated for which there is no corresponding data point in the new data set. Both enter and update selections contain arrays having an entry for each data point in the new data set, but with only those entries populated that are newly added or retained from previously, respectively. The `merge()` function combines these complementary arrays into one. If both arrays have a nonnull entry in the same position, one of them will be clobbered. All of this is implemented using the JavaScript `Array` type, which allows for "holes" of unassigned, undefined values at arbitrary index positions. It is probably also best to consider all of this information as implementation detail and subject to change. Just remember the role of the `merge()` function as part of the General Update Pattern.

The parameters passed to these functions depend a little on circumstances; I will point out the differences as we go along. For methods of the `Selection` abstraction, the parameters are always: the data point `d` bound to the current element, the element's index `i` in the current selection, and the array `nodes` containing the nodes in the current selection,[3] while `this` is set to point to the current DOM element itself:[4]

```
function( d, i, nodes ) {
    this === nodes[i];                  // true!
}
```

As with all JavaScript functions, all arguments are optional. The return type is not mandated, but will generally be fairly obvious from the context. When evaluated for a selection, accessors are usually expected to return a value (often a string), or a DOM node.

Operating on Elements of a Selection

The `Selection` abstraction provides several methods to manipulate aspects of the individual DOM elements contained in the selection (see Table 3-4).

Table 3-4. Methods operating on elements of a selection (sel is a Selection object)

Function	Description
`sel.attr(name, value)`	Sets the attribute named `name` to the supplied value.
`sel.style(name, value, priority)`	Sets the named style property to the supplied value. An override priority may be specified by supplying the string `"important"` (without exclamation point) as the third parameter. Note that style length properties usually require units (in contrast to SVG attributes).
`sel.property(name, value)`	Sets the named property to the supplied value. (This is intended for HTML elements that have properties that are not accessible as attributes, such as the checkbox `checked` property.)

3 Strictly speaking, the `nodes` argument contains the current *group*, and `i` is the index within the group. We will discuss groups toward the end of this chapter.

4 If you want to access `this` in an accessor, you must use the `function` keyword; you cannot use an arrow function.

Function	Description
`sel.classed(value, flag)`	The `value` argument should be a whitespace-separated string of class names. If the `flag` parameter evaluates to `true`, the `class` attribute is set to this string; if `flag` evaluates to `false`, the classes are unassigned.
`sel.text(value)`	Sets the "text content" to the supplied value: use this to set the actual text for `<text>` elements. (This method replicates a corresponding property on the DOM `Node` interface.)
`sel.html(value)`	Sets the "inner HTML" to the supplied value: this is the HTML with its markup inside of, but not including, the current element. (This method replicates a corresponding property on the DOM `Element` interface.)
`sel.datum(value)`	Sets the data bound to this element to the supplied value.
`sel.each(function)`	Invokes the supplied accessor function for each element in the selection.

All function arguments are optional (as always in JavaScript). The functions in Table 3-4 can be used to set, get, or clear an attribute (or property,[5] or style, …), depending on the `value` argument:

- If no `value` is supplied, the function returns the current value for the first nonnull element in the selection.
- If a `null` value is supplied, the attribute (or property) is removed from the element.
- If a constant `value` is supplied and not `null`, the attribute (or property) is set to the supplied value.
- If a function is supplied, it is evaluated for every element in the selection and its return value is used to modify the current element.

5 The distinction between attributes and properties is subtle. Basically, an HTML element has *attributes*, a JavaScript `Node` object has *properties*. Most attributes map to properties and vice versa, but the names don't always agree exactly, and the momentaneous value associated with an attribute or property may depend on the dynamic state of the page and therefore may differ between the two representations. The SVG specification distinguishes between properties as attributes that can be modified through CSS, in contrast to attributes in general, which cannot.

With the exception of the last two entries, the functions in Table 3-4 are thin wrappers around the equivalent functionality in the DOM API; you may want to compare the appropriate reference documentation for the precise semantics of some of the terms. (See, for example, the MDN Node Reference and the MDN Element Reference.)

Operating on Selections Themselves

The `Selection` abstraction provides several functions to operate on the entire selection, for example, to add, remove, or reorder elements (see Table 3-5).

Table 3-5. Methods that operate on the entire selection (sel is a Selection object)

Function	Description
`sel.append(item)`	• Appends an additional element to each element in the current selection. • If `item` is a string, a new element of the indicated type (tag name) is created and appended. • If `item` is an accessor function, it is evaluated for each element in the current selection and should return a DOM `Element` (new or existing) to be appended. • Returns a new selection containing the appended elements.
`sel.insert(item, before)`	• Inserts an element before the first element matching the `before` selector for each element in the current selection. • If `item` is a string, a new element of the indicated type (tag name) is created and inserted. • If `item` is an accessor function, it is evaluated for each element in the current selection and should return a DOM `Element` (new or existing) to be inserted. • The `before` argument may be a CSS selector string or an accessor function. If it is an accessor, then it must return a DOM `Node` (not a `Selection`) from the current selection; the new element will be inserted before this `Node`. If `before` is missing or `null`, the item is appended at the end of the selection. • Returns a new selection containing the inserted elements.

Function	Description
`sel.merge(selection)`	Merges the current selection with the supplied selection and returns the combined selection. The two selections are expected to have undefined elements in complimentary positions; if both selections have a nonnull entry in the same position, the current selection prevails. This function is not intended to concatenate arbitrary selections; instead, its primary purpose is to merge the `enter` selection back into the current selection in the General Update Pattern.
`sel.remove()`	Removes the selected elements from the document and returns a selection containing the removed elements.
`sel.sort(comparator)`	Takes a function of two arguments (`a` and `b`), which is called with the data bound to the elements of this selection. The comparator should return -1 if `a` is less than `b`, +1 if `b` is less than `a`, and zero otherwise. Returns a new selection with each group sorted according to the comparator; also reinserts elements into the current selection accordingly. When no comparator is supplied, the selection is sorted in ascending order.
`sel.call(function, arguments)`	Invokes the supplied function exactly once, passing in the current selection and any supplied arguments. Returns the current selection and so enables method chaining (the primary application for this method).
`sel.nodes()`	Returns the nonnull nodes in this selection as an array of DOM `Node` objects.
`sel.node()`	Returns the first nonnull node in this selection as a DOM `Node` object.
`sel.size()`	Returns the number of nonnull elements in this selection.
`sel.empty()`	Returns true if there are no nonnull elements in this selection.

The `node()` and `nodes()` functions are a way to obtain a reference to the actual DOM `Node` instances in the selection. This is occasionally useful; we will see examples in Chapter 4.[6]

We have seen the `append()` function in action several times before. The next example will demonstrate how to use `insert()` and `sort()`. Initially, an unordered list is populated from a static data set. If you mouse over the list, two more items are inserted into it. If you then mouse click into the list, the items are sorted, in descending order, according to their text content (see Example 3-3).

6 Be warned that additional concerns arise when accessing the DOM API directly, rather than through D3. In particular, XML namespaces often need to be taken into account explicitly; see Chapter 6 for an example.

Example 3-3. Inserting elements into a selection and sorting them

```
function makeSort() {
    var data = [ "Jane", "Anne", "Mary" ];

    var ul = d3.select( "#sort" );
    ul.selectAll( "li" ).data( data ).enter().append( "li" )          ❶
        .text( d=>d );

    // insert on mouse enter
    var once;                                                         ❷
    ul.on( "mouseenter", function() {                                 ❸
        if( once ) { return; }
        once = 1;
        ul.insert( "li", ":nth-child(2)" )                            ❹
            .datum( "Lucy" ).text( "Lucy" );
        ul.insert( "li", ":first-child" )                             ❺
            .datum( "Lisa" ).text( "Lisa" );
    } );

    // sort on click
    ul.on( "click", function() {                                      ❻
        ul.selectAll( "li" ).sort( (a,b)=>( a<b?1:b<a?-1:0 ) );       ❼
    } );
}
```

❶ An unordered list is populated from the data set.

❷ The variable `once` makes sure that the new items are only added to the list once.

❸ Register the first of two event handlers for the list: if the mouse pointer enters the area occupied by the list, the callback is invoked.

❹ The position where the new list items are to be inserted is specified through pseudo-classes. The `:nth-child()` pseudo-class starts counting at 1 (so that `:nth-child(1)` equals `:first-child`). Observe that we need to both set the data bound to each element (using `datum()`) and the visible text (using `text()`) separately when using `insert()`.

❺ Another element is added in front of the entire list, pushing the previously added element from the second to the third position. (Positions in pseudo-classes are evaluated at the time the pseudo-class is applied.)

❻ A click event handler is registered as the second event handler on the list.

❼ Upon a mouse click, the list elements are sorted, in descending order, based on the value of the data bound to them.

Remember that `insert()` does not bind data; an explicit call to `datum()` is required to add data to elements added using `insert()`.

As a rule, methods that operate on individual elements of a selection (Table 3-4) return the *current* selection, whereas methods that operate on the entire selection (Table 3-5) return a *new* selection—but there are some exceptions. For reference, here are the functions that return *new* selections:

- `select()`, `selectAll()`
- `data()`, `enter()`, `exit()`
- `append()`, `insert()`, `remove()`
- `merge()`, `filter()`, `sort()`
- `create()`

Shared Parent Information Among Selections with Groups

Selections maintain one additional piece of information that is not exposed explicitly in the API, namely which members of the selection share a common parent *in the previous selection* (not necessarily in the document). This is best understood through an example. Consider the following HTML table:

```
<table>
    <tr>
        <td>A</td><td>B</td>
    </tr>
    <tr>
        <td>C</td><td>D</td>
    </tr>
</table>
```

and select all cells within all rows (for example, to color them):

```
d3.selectAll("tr").selectAll("td").attr(..., (d,i,ns)=>{ ... });
```

The first argument `d` to the accessor function is obviously the data bound to each cell. But what should the index `i` refer to? Because the cells were selected as elements of their respective *rows*, the index `i` holds the position of each cell *within its row*—in other words, its *column*. This makes it exceedingly easy to shade the cells by columns, for example. In the same spirit, the third argument `ns` contains the elements (nodes) for the current row (not for the entire table).

Let me repeat that this information about shared parents refers to the originating *selection*, not the *document*. For example, instead of creating a selection of rows first, you could select the cells directly from the document:

```
d3.selectAll("td").attr(..., (d, i, ns) => { ... });
```

Now, the index `i` will be the running number of the cells (from 0 to 3, in this case), and `ns` is the collection of all cells. Selections maintain only a single level of ancestry. If each cell contained an unordered list, then in the following snippet:

```
d3.selectAll("tr").selectAll("td")
    .selectAll("li").attr( ..., (d,i) => { ... } );
```

the index `i` would be the position of each list item within its list. The information about the column information would be lost at that point. (It would, of course, still be available in the second selection in this chain, the one containing the table cells.)

All of this is extremely straightforward and intuitive—the less you think about it, the easier it is. By and large, D3 simply does what you expect it to do. None of this functionality is exposed explicitly (through individual functions), but you may find in code and documentation references to "groups"—the name for the internal representation of the common parent information. (All children of a common parent form a group.)

More detail on this topic can be found in two dedicated blog posts by Mike Bostock, which are recommended reading: *https://bost.ocks.org/mike/nest/* and *https://bost.ocks.org/mike/selection/*.

CHAPTER 4

Events, Interactivity, and Animation

When rendered by a browser, SVG elements can receive user events and can be manipulated as a whole (for example, to change their position or appearance). This means that they behave essentially like widgets in a GUI toolkit. That's an exciting proposition: to regard SVG as a *widget set* for *graphics*. This chapter discusses the options available to create those essential features of a user interface: interactivity and animation.

Events

An important aspect of the DOM is its *event model*: essentially any DOM element can *receive events* and invoke an appropriate handler. The number of different event types is very large; most important for our purposes are user-generated events (mouse clicks or movements, as well as keystrokes; see Table 4-1).[1]

Table 4-1. Some important user-generated event types

Function	Description
`click`	Any mouse button is pressed and released on an element.
`mousemove`	The mouse is moved while over an element.

1 See the MDN Event Reference for more information.

Function	Description
`mousedown, mouseup`	A mouse button is pressed or released over an element.
`mouseenter, mouseleave`	The mouse pointer is moved onto or off of an element.
`mouseover, mouseout`	The mouse pointer is moved onto or off of an element, or any of its children.
`keydown, keyup`	Any key is pressed or released.

D3 treats event handling as part of the `Selection` abstraction (see Table 4-2). If `sel` is a `Selection` instance, then you use the following member function to register a callback as event handler for the specified event type:

```
sel.on( type, callback )
```

The `type` argument must be a string indicating the event type (such as `"click"`). Any DOM event type is permitted. If a handler was already registered for the event type via `on()`, it is removed before the new handler is registered. To explicitly remove the handler for a given event type, provide `null` as the second argument. To register multiple handlers for the same event type, the type name may be followed by a period and an arbitrary tag (so that the handler for `"click.foo"` does not overwrite the one for `"click.bar"`).

The callback is a function invoked when an event of the specified type is received by any element of the selection. The callback will be invoked in the same way as any other accessor is invoked in the context of a selection, being passed the data point `d` bound to the current element, the element's index `i` in the current selection, and the nodes in the current selection, while `this` contains the current element itself.[2] The actual event instance is *not* passed to the callback as an argument, but it is available in the variable:

```
d3.event
```

When an event occurs, this variable contains the raw DOM event instance (not a D3 wrapper!). The information provided by the event object itself depends on the event type. For mouse events, naturally the *location* of the mouse pointer when the event occurred is of particular interest. The event object contains the mouse coordi-

2 If you want to access `this` in a callback, you must use the `function` keyword to define the callback; you cannot use an arrow function.

nates with respect to three different coordinate systems,[3] but none directly provides the information that would be most useful, namely the position with respect to the containing parent element! Thankfully, they can be obtained using:

```
(((“d3.mouse() function”)))d3.mouse( node )
```

This function returns the mouse coordinates as a two-element array `[x, y]`. The argument should be the enclosing container element (as a DOM `Node`, not as `Selection`). When working with SVG, you can supply *any* element (as a `Node`), and the function will calculate the coordinates relative to the nearest ancestor SVG element.

Table 4-2. Some important methods, variables, and functions related to event handling (sel is a Selection object)

Function	Description
`sel.on( types, callback )`	Adds or removes a callback for each element in the selection. The `types` argument must be a string consisting of one or more event type names, separated by whitespace. An event type may be followed by a period and an arbitrary tag to allow multiple handlers to be registered for a single event type. • If a callback is specified, it is registered as the event handler; any existing event handler is removed first. • If the callback argument is `null`, any existing handler is removed. • If the callback argument is missing, the currently assigned handler is returned.
`d3.event`	Contains the current event, if any, as a DOM `Event` object.
`d3.mouse( parent )`	Returns a two-element array containing the mouse coordinates relative to the specified parent.
`sel.dispatch( type )`	Dispatches a custom event of the specified type to all elements in the current selection.

3 They are: `screen`, relative to the edge of the physical screen; `client`, relative to the edge of the browser window; and `page`, relative to the edge of the document itself. Due to the placement of the window on the screen, and to the scrolling of the page within the browser, these three will generally differ.

Exploring Graphs with the Mouse

For someone working analytically with data, these features offer some exciting opportunities because they make it easy to *explore graphs interactively*: point the mouse at a spot on the graph and get additional information about the data point located there. Here is a simple example. If you call the function in Example 4-1, while supplying a CSS selector string (see "CSS Selectors" on page 36) that identifies an `<svg>` element, the current mouse pointer location (in pixel coordinates) will be shown in the graph itself. Moreover, the location of the textual display is not fixed but will move together with the mouse pointer.

Example 4-1. Given a CSS selector string, this function will continuously display the mouse position in pixel coordinates whenever the user moves the mouse.

```
function coordsPixels( selector ) {
    var txt = d3.select( selector ).append( "text" );                    ❶
    var svg = d3.select( selector ).attr( "cursor", "crosshair" )        ❷
        .on( "mousemove", function() {
            var pt = d3.mouse( svg.node() );                             ❸
            txt.attr( "x", 18+pt[0] ).attr( "y", 6+pt[1] )               ❹
                .text( "" + pt[0] + "," + pt[1] );
        } );
}
```

❶ Create the `<text>` element to display the coordinates. It is important to do this *outside* of the event callback; otherwise, a new `<text>` element will be created every time the user moves the mouse!

❷ Change the shape of the mouse cursor while over the `<svg>` element. This is not required, of course—but it is a fitting effect (and also demonstrates how the mouse cursor can be changed through attributes; see Appendix B).

❸ Obtain the mouse coordinates, relative to the upper-left corner of the `<svg>` element, using the `d3.mouse()` convenience function.

❹ Update the text element created earlier. In this example, both the displayed text content of the element and its position are updated: slightly to the right of the mouse position.

Displaying the mouse coordinates is, of course, neither new nor particularly exciting. But what is exciting is to see just *how* easy it is to implement such behavior in D3!

Case Study: Simultaneous Highlighting

The next example is more interesting. It addresses a common problem when working with multivariate data sets: how to link two different views or projections of the data visually. One way is to select a region of data points in one view with the mouse and simultaneously highlight the corresponding points in all other views. In Figure 4-1, points are highlighted in *both* panels according to their distance (in pixel coordinates) from the mouse pointer in the *lefthand panel*. Because this example is more involved, we will first discuss a simplified version (see Example 4-2).

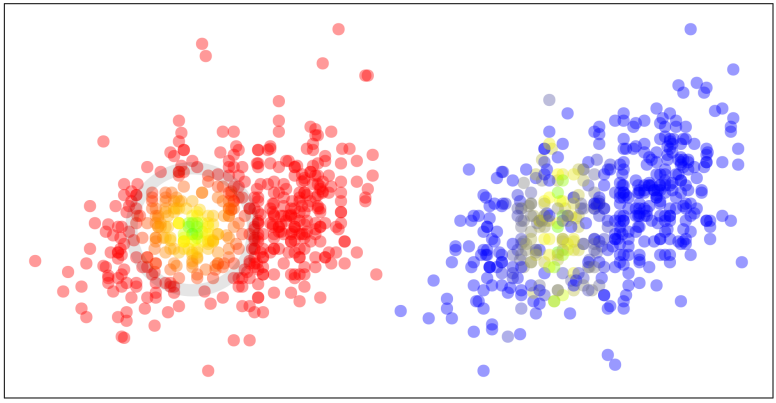


Figure 4-1. Data points belonging to the same record are simultaneously highlighted in both panels, based on the distance of the points in the lefthand panel to the mouse pointer.

Example 4-2. Commands for Figure 4-1

```
function makeBrush() {
    d3.csv( "dense.csv" ).then( function( data ) {                ❶
        var svg1 = d3.select( "#brush1" );                        ❷
        var svg2 = d3.select( "#brush2" );
```

```
        var sc1=d3.scaleLinear().domain([0,10,50])                    ❸
            .range(["lime","yellow","red"]);
        var sc2=d3.scaleLinear().domain([0,10,50])
            .range(["lime","yellow","blue"]);

        var cs1 = drawCircles(svg1,data,d=>d["A"],d=>d["B"],sc1);     ❹
        var cs2 = drawCircles(svg2,data,d=>d["A"],d=>d["C"],sc2);

        svg1.call( installHandlers, data, cs1, cs2, sc1, sc2 );       ❺
    } );
}

function drawCircles( svg, data, accX, accY, sc ) {
    var color = sc(Infinity);                                         ❻
    return svg.selectAll( "circle" ).data( data ).enter()
        .append( "circle" )
        .attr( "r", 5 ).attr( "cx", accX ).attr( "cy", accY )
        .attr( "fill", color ).attr( "fill-opacity", 0.4 );
}

function installHandlers( svg, data, cs1, cs2, sc1, sc2 ) {
    svg.attr( "cursor", "crosshair" )
        .on( "mousemove", function() {
            var pt = d3.mouse( svg.node() );

            cs1.attr( "fill", function( d, i ) {                      ❼
                var dx = pt[0] - d3.select( this ).attr( "cx" );
                var dy = pt[1] - d3.select( this ).attr( "cy" );
                var r = Math.hypot( dx, dy );

                data[i]["r"] = r;                                     ❽
                return sc1(r); } );                                   ❾

            cs2.attr( "fill", (d,i) => sc2( data[i]["r"] ) ); } )     ❿

        .on( "mouseleave", function() {
            cs1.attr( "fill", sc1(Infinity) );                        ⓫
            cs2.attr( "fill", sc2(Infinity) ); } );
}
```

❶ Load the data set and specify the callback to invoke when the data is available (see Chapter 6 for more information about fetching data). The file contains three columns, labeled `A`, `B`, and `C`.

❷ Select the two panels of the graph.

❸ D3 can smoothly interpolate between colors. Here we create two color gradients (one for each panel). (See Chapter 7 to learn more about interpolation and scale objects.)

❹ Create the circles representing data points. The newly created circles are returned as `Selection` objects. Following a general D3 convention, columns are specified in the function call by providing accessor functions.

❺ Call the function `installHandlers()` to register the event handlers. This line of code uses the `call()` facility to invoke the `installHandlers()` function, while supplying the `svg1` selection and the remaining parameters as arguments. (We encountered this already in Example 2-6; also see the discussion regarding *components* in Chapter 5.)

❻ Initially, the circles are drawn with the "maximum" color. To find this color, evaluate the color scale at positive infinity.

❼ For each point in the panel on the left, calculate its distance to the mouse pointer…

❽ … and store it, as an additional column, in the data set. (This will be our mechanism of communication between the two panels of the figure.)

❾ Return the appropriate color from the color gradient.

❿ Use the additional column in the data set to color the points in the panel on the right.

⓫ Restore the points to their original colors when the mouse leaves the lefthand panel.

This version of the program works well and solves the original problem. The improved version of the `installHandlers()` function shown in Example 4-3 allows us to discuss some additional techniques when writing this kind of user interface code.

Example 4-3. An improved version of the installHandlers() function in Example 4-2

```
function installHandlers2( svg, data, cs1, cs2, sc1, sc2 ) {
    var cursor = svg.append( "circle" ).attr( "r", 50 )                    ❶
        .attr( "fill", "none" ).attr( "stroke", "black" )
        .attr( "stroke-width", 10 ).attr( "stroke-opacity", 0.1 )
        .attr( "visibility", "hidden" );                                   ❷

    var hotzone = svg.append( "rect" ).attr( "cursor", "none" )            ❸
        .attr( "x", 50 ).attr( "y", 50 )
        .attr( "width", 200 ).attr( "height", 200 )
        .attr( "visibility", "hidden" )                                    ❹
        .attr( "pointer-events", "all" )

        .on( "mouseenter", function() {                                    ❺
            cursor.attr( "visibility", "visible" ); } )

        .on( "mousemove", function() {                                     ❻
            var pt = d3.mouse( svg.node() );
            cursor.attr( "cx", pt[0] ).attr( "cy", pt[1] );

            cs1.attr( "fill", function( d, i ) {
                var dx = pt[0] - d3.select( this ).attr( "cx" );
                var dy = pt[1] - d3.select( this ).attr( "cy" );
                var r = Math.hypot( dx, dy );

                data[i]["r"] = r;
                return sc1(r); } );

            cs2.attr( "fill", (d,i) => sc2( data[i]["r"] ) ); } )

        .on( "mouseleave", function() {
            cursor.attr( "visibility", "hidden" );
            cs1.attr( "fill", sc1(Infinity) );
            cs2.attr( "fill", sc2(Infinity) ); } )
}
```

❶ In this version, the actual mouse pointer itself is hidden and replaced with a large, partially opaque circle. Points within the circle will be highlighted.

❷ Initially, the circle is hidden. It will only be shown once the mouse pointer enters the "hot zone."

❸ The "hot zone" is defined as a rectangle inside the lefthand panel. The event handlers are registered on this rectangle,

meaning that they will only be invoked when the mouse pointer is inside of it.

❹ The rectangle is hidden from view. By default, DOM elements that have their `visibility` attribute set to `hidden` do not receive mouse pointer events. To overcome this, the `pointer-events` attribute must be set explicitly. (Another way to make an element invisible is to set its `fill-opacity` to `0`. In this case, it will not be necessary to modify the `pointer-events` attribute.)

❺ When the mouse enters the "hot zone," the opaque circle that acts as the pointer is displayed.

❻ The `mousemove` and `mouseleave` event handlers are equivalent to the ones in Example 4-2, except for the additional commands to update the circle acting as a cursor.

The use of an active "hot zone" in this example is of course optional, but it demonstrates an interesting technique. At the same time, the discussion of the `pointer-events` attribute suggests that this kind of user interface programming may involve unexpected challenges. We will come back to this point after the next example.

The D3 Drag-and-Drop Behavior Component

Several common user interface patterns consist of a combination of events and responses: in the drag-and-drop pattern, for instance, the user first selects an item, then moves it, and finally releases it again. D3 includes a number of predefined *behavior components* that simplify the development of such user interface code by bundling and organizing the required actions. In addition, these components also unify some details of the user interface.

Consider a situation like the one in Figure 4-2, showing the following SVG snippet:

```
<svg id="dragdrop" width="600" height="200">
  <circle cx="100" cy="100" r="20" fill="red" />
  <circle cx="300" cy="100" r="20" fill="green" />
  <circle cx="500" cy="100" r="20" fill="blue" />
</svg>
```

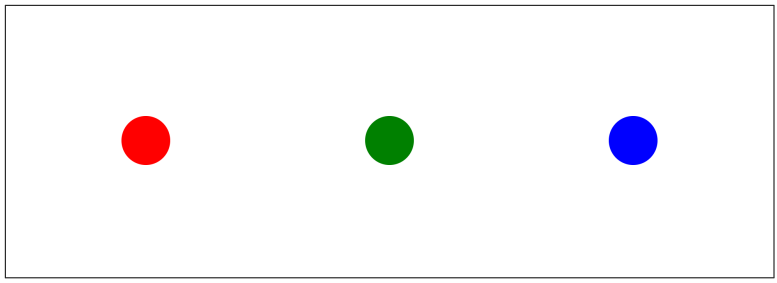


Figure 4-2. The initial configuration for the drag-and-drop behavior

Now let's enable the user to change the position of the circles with the mouse. It is not difficult to add the familiar drag-and-drop pattern by registering callbacks for `mousedown`, `mousemove`, and `mouseup` events, but Example 4-4 uses the D3 `drag` behavior component instead. As explained in Example 2-6, a component is a function object that takes a `Selection` instance as argument and adds DOM elements to that `Selection` (also see Chapter 5). A *behavior* component is a component that installs required event callbacks in the DOM tree. At the same time, it is also an object that has member functions itself. The listing uses the drag component's `on( type, callback )` member function to specify the callbacks for the different event types.

Example 4-4. Using the drag-and-drop behavior

```
function makeDragDrop() {
    var widget = undefined, color = undefined;

    var drag = d3.drag()                                                    ❶
        .on( "start", function() {                                          ❷
            color = d3.select( this ).attr( "fill" );
            widget = d3.select( this ).attr( "fill", "lime" );
        } )
        .on( "drag", function() {                                           ❸
            var pt = d3.mouse( d3.select( this ).node() );
            widget.attr( "cx", pt[0] ).attr( "cy", pt[1] );
        } )
        .on( "end", function() {                                            ❹
            widget.attr( "fill", color );
            widget = undefined;
        } );

    drag( d3.select( "#dragdrop" ).selectAll( "circle" ) );                 ❺
}
```

❶ Create a `drag` function object using the factory function `d3.drag()`, then invoke the `on()` member function on the returned function object to register the required callbacks.

❷ The `start` handler stores the current color of the selected circle; then changes the selected circle's color and assigns the selected circle itself (as a `Selection`) to `widget`.

❸ The `drag` handler retrieves the current mouse coordinates and moves the selected circle to this location.

❹ The `end` handler restores the circle's color and clears the active `widget`.

❺ Finally, invoke the `drag` component operation while supplying a selection containing the circles to install the configured event handlers on the selection.

A more idiomatic way to express this would use the `call()` function rather than invoking the component operation explicitly:

```
d3.select( "#dragdrop" ).selectAll( "circle" )
    .call( d3.drag()
           .on( "start", function() { ... } )
           .on( "drag", function() { ... } )
           .on( "end", function() { ... } ) );
```

The event names in Example 4-4 may come as a surprise: these are not standard DOM events, but D3 pseudoevents. The D3 `drag` behavior combines both mouse and touch-screen event handling. Internally, the `start` pseudoevent corresponds to either a `mousedown` or a `touchstart` event, and similar for `drag` and `end`. Furthermore, the `drag` behavior prevents the browser's *default action* for certain event types.[4] D3 includes additional behaviors to assist with zooming and when selecting parts of a graph with a mouse.

4 If you try to implement the current example without using the D3 `drag` facility, you may occasionally observe spurious user interface behavior. This is likely the browser's default action interfering with the intended behavior. The remedy is to call `d3.event.preventDefault()` in the `mousemove` handler. See Appendix C for more information.

Notes on User Interface Programming

I hope the examples so far have convinced you that creating *interactive* graphs using D3 need not be difficult—in fact, I believe D3 makes them feasible even for ad hoc, one-off tasks and explorations. At the same time, as the discussion after the previous two examples shows, graphical user interface programming is still a relatively complex problem. Many components, each with its own rules, participate and can interact in unexpected ways. Browsers may differ in their implementation. Here are some reminders and potential surprises (also see Appendix C for background information on DOM event handling):

- Repeated calls to `on()` for the same event type on the same `Selection` instance clobber each other. Add a unique tag to the event type (separated by a period) to register multiple event handlers.
- If you want to access `this` in a callback or accessor function, you *must* use the `function` keyword, you *cannot* use an arrow function. This is a limitation of the JavaScript language (see Appendix C). Examples can be found in the `installHandlers()` function in Examples 4-2 and 4-3, and several times in Example 4-4.
- Browser default behavior may interfere with your code; you may need to prevent it explicitly.
- Generally, only visible, painted elements can receive mouse pointer events. Elements with their `visibility` attribute set to `hidden`, or with both `fill` and `stroke` set to `none`, do not receive pointer events by default. Use the `pointer-events` attribute for fine-grained control over the conditions under which elements will receive events. (See MDN Pointer-Events.)
- In a similar spirit, a `<g>` element has no visual representation, and hence does not generate pointer events. Nevertheless, it may be appropriate to register an event handler on a `<g>` element because events generated by any of its (visible) children will be delegated to it. (Use an invisible rectangle or other shape to define active "hot zones," as in Example 4-3.)

Smooth Transitions

An obvious way to respond to events is to apply some change to the figure's appearance or configuration (for example, to show a before-and-after effect). In this case, it is often useful to let the change take place *gradually*, rather than instantaneously, to draw attention to the change that is taking place and to allow users to discern additional detail. For example, users may now be able to recognize which data points are most affected by the change, and how (see Figure 3-3 and Example 3-1 for an example).

Conveniently, the D3 `Transition` facility does all the work for you. It replicates most of the `Selection` API, and you can change the appearance of selected elements using `attr()` or `style()` as before (see Chapter 3). But now the new settings do not take effect immediately; instead, they are applied gradually over a configurable time span (see Example 2-8 for an early example).

Under the covers, D3 creates and schedules the required *intermediate configurations* to give the appearance that the graph is changing smoothly over the desired duration. To do so, D3 invokes an *interpolator* that creates the intermediate configurations between the starting and end points. The D3 interpolation facility is fairly smart and able to interpolate automatically between most types (such as numbers, dates, colors, strings with embedded numbers, and more —see Chapter 7 for a detailed description).

Creating and Configuring Transitions

The workflow to create a transition is simple (also see Table 4-3):

1. Before creating a transition, make sure any data has been bound and all elements that are supposed to be part of the transition have been created (using `append()` or `insert()`)—even if they are initially set to be invisible! (The `Transition` API allows you to change and remove elements, but it does not provide for the creation of elements as part of the transition.)
2. Now select the elements you wish to change using the familiar `Selection` API.
3. Invoke `transition()` on this selection to create a transition. Optionally, call `duration()`, `delay()`, or `ease()` for more control over its behavior.

4. Set the desired end state using `attr()` or `style()` as usual. D3 will create the intermediate configurations between the current values and the indicated end states, and apply them over the duration of the transition.

Often, these commands will be part of an event handler, so that the transition starts when an appropriate event occurs.

Table 4-3. Functions to create and terminate a transition (sel is a Selection object; trans is aTransition object)

Function	Description
`sel.transition( tag )`	Returns a new transition on the receiving selection. The optional argument may be a string (to identify and distinguish this transition on the selection) or a `Transition` instance (to synchronize transitions).
`sel.interrupt( tag )`	Stops the active transition and cancels any pending transitions on the selected elements for the given identifier. (Interrupts are not forwarded to children of the selected elements.)
`trans.transition()`	Returns a new transition on the same selected elements as the receiving transition, scheduled to start when the current transition ends. The new transition inherits the current transition's configuration.
`trans.selection()`	Returns the selection for a transition.

In addition to the desired end point, a `Transition` also allows you to configure several aspects of its behavior (see Table 4-4). All of these have reasonable defaults, making explicit configuration optional:

- A *delay* that must pass before the change begins to take effect.
- A *duration* over which the setting will change gradually.
- An *easing* that controls how the rate of change will differ over the transition duration (to "ease into" and "out of" the animation). By default, the easing follows a piecewise cubic polynomial with "slow-in, slow-out" behavior.
- An *interpolator* to calculate the intermediate values (this is rarely necessary, because the default interpolators handle most common configurations automatically).
- An *event handler* to execute custom code when the transition starts, ends, or is interrupted.

Table 4-4. Functions to configure a transition or to retrieve the current setting if called without argument (trans is a Transition object)

Function	Description
`trans.delay( value )`	Sets the delay (in milliseconds) before the transition begins for each element in the selection; the default is 0. The delay can be given as a constant or as a function. If it is a function, the function will be invoked once for each element, before the transition begins, and should return the desired delay. The function will be passed the data bound to the element `d` and its index in the selection `i`.
`trans.duration( value )`	Sets the duration (in milliseconds) of the transition for each element in the selection; the default is 250 milliseconds. The duration can be given as a constant or as a function. If it is a function, the function will be invoked once for each element, before the transition begins, and should return the desired duration. The function will be passed the data bound to the element `d` and its index in the selection `i`.
`trans.ease( fct )`	Sets the easing function for all selected elements. The easing must be a function, taking a single parameter between 0 and 1, and returning a single value, also between 0 and 1. The default easing is `d3.easeCubic` (a piecewise defined cubic polynomial with "slow-in, slow-out" behavior).
`trans.on( type, handler )`	Adds an event handler on the transition. The type must be `start`, `end`, or `interrupt`. The event handler will be invoked at the appropriate point in the transition's lifecycle. This function behaves similarly to the `on()` function on a `Selection` object see Table 4-2).

Using Transitions

The `Transition` API replicates large parts of the `Selection` API. In particular, all functions from Table 3-2 (that is, `select()`, `selectAll()`, and `filter()`) are available. From Table 3-4, `attr()`, `style()`, `text()`, and `each()` carry over, as well as all functions from Table 3-5 *except* `append()`, `insert()`, and `sort()`. (As was pointed out earlier, all elements participating in a transition must exist before the transition is created. For the same reason, none of the functions for binding data from Table 3-3 exist for transitions.)

Basic transitions are straightforward to use, as we have already seen in an example in an earlier chapter (Example 3-1). The application

in Example 4-5 is still simple, but the effect is more sophisticated: a bar chart is updated with new data, but the effect is *staggered* (using `delay()`) so that the bars don't all change at the same time.

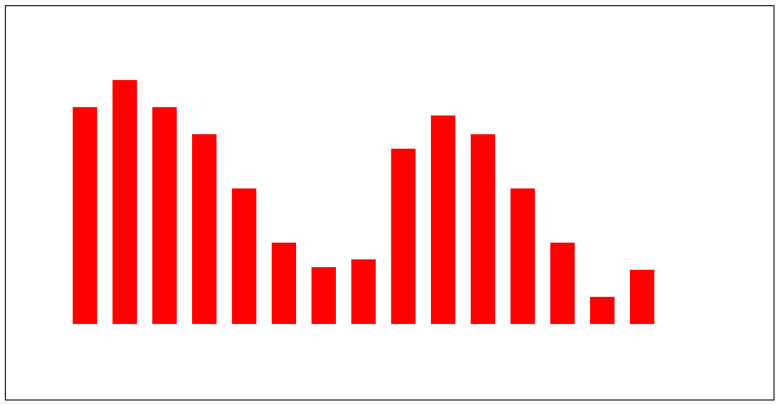


Figure 4-3. When this bar chart is updated with a new data set, the updates are applied consecutively, left to right.

Example 4-5. Using transitions (see Figure 4-3)

```
function makeStagger() {
    var ds1 = [ 2, 1, 3, 5, 7, 8, 9, 9, 9, 8, 7, 5, 3, 1, 2 ];      ❶
    var ds2 = [ 8, 9, 8, 7, 5, 3, 2, 1, 2, 3, 5, 7, 8, 9, 8 ];
    var n = ds1.length, mx = d3.max( d3.merge( [ds1, ds2] ) );      ❷

    var svg = d3.select( "#stagger" );

    var scX = d3.scaleLinear().domain( [0,n] ).range( [50,540] );   ❸
    var scY = d3.scaleLinear().domain( [0,mx] ).range( [250,50] );

    svg.selectAll( "line" ).data( ds1 ).enter().append( "line" )    ❹
        .attr( "stroke", "red" ).attr( "stroke-width", 20 )
        .attr( "x1", (d,i)=>scX(i) ).attr( "y1", scY(0) )
        .attr( "x2", (d,i)=>scX(i) ).attr( "y2", d=>scY(d) );

    svg.on( "click", function() {                                   ❺
        [ ds1, ds2 ] = [ ds2, ds1 ];                                ❻

        svg.selectAll( "line" ).data( ds1 )                         ❼
            .transition().duration( 1000 ).delay( (d,i)=>200*i )    ❽
            .attr( "y2", d=>scY(d) );                               ❾
    } );
}
```

❶ Define two data sets. To keep things simple, only the y values are included; we will be using the array index of each item for its horizontal position.

❷ Find the number of data points, and the overall maximal value across both data sets.

❸ Two scale objects that map the values in the data set to vertical, and their index positions in the array to horizontal pixel coordinates.

❹ Create the bar chart. Each "bar" is realized as a thick line (rather than a `<rect>` element).

❺ Register an event handler for `"click"` events.

❻ Interchange the data sets.

❼ Bind the (updated) data set `ds1` to the selection…

❽ … and create a transition instance. Each bar will take one second to attain its new size, but will start only after a *delay*. The delay is dependent on the horizontal position of each bar, growing left to right. This has the effect that the "update" seems to sweep across the chart.

❾ Finally, set the new vertical length of each line. This is the end point for the transition.

Hints and Techniques

Not all transitions are as straightforward as the ones we have seen so far. Here are some additional hints and techniques.

Strings. The D3 default interpolators will interpolate numbers that are embedded in strings, but leave the rest of the string alone because there is no generally useful way to interpolate between strings. The best way to achieve a smooth transition between strings is to cross-fade between *two* strings in the same location. Assume that two suitable `<text>` elements exist:

```
<text id="t1" x="100" y="100" fill-opacity="1">Hello</text>
<text id="t2" x="100" y="100" fill-opacity="0">World</text>
```

Then you can cross-fade between them by changing their opacity (possibly changing the duration of the transition):

```
d3.select("#t1").transition().attr( "fill-opacity", 0 );
d3.select("#t2").transition().attr( "fill-opacity", 1 );
```

An alternative that may make sense in certain cases is to write a custom interpolator to generate intermediate string values.

Chained transitions. Transitions can be chained so that one transition begins when the first one ends. The subsequent transitions inherit the earlier transition's duration and delay (unless they are overridden explicitly). The following code will turn the selected elements first to red, then to blue:

```
d3.selectAll( "circle" )
    .transition().duration( 2000 ).attr( "fill", "red" )
    .transition().attr( "fill", "blue" );
```

Explicit starting configuration. Unless you plan to use a custom interpolator (see next), it is important that the starting configuration is set explicitly. For example, don't rely on the default value (`black`) for the `fill` attribute: unless the fill attribute is set explicitly, the default interpolator will not know what to do.

Custom interpolators. Using the methods in Table 4-5, it is possible to specify a *custom interpolator* to be used during the transition. The methods to set a custom interpolator take a *factory function* as argument. When the transition starts, the factory function is invoked for each element in the selection, being passed the data `d` bound to the element and the element's index `i`, with `this` being set to the current DOM `Node`. The factory must return an interpolator function. The interpolator function must accept a single numeric argument between 0 and 1 and must return an appropriate intermediate value between the starting and the end configuration. The interpolator will be called after any easing has been applied. The following code uses a simple custom color interpolator without easing (see Chapter 8 to learn about more flexible ways to operate on colors in D3):

```
d3.select( "#custom" ).selectAll( "circle" )
    .attr( "fill", "white" )
    .transition().duration( 2000 ).ease( t=>t )
    .attrTween( "fill", function() {
        return t => "hsl(" + 360*t + ", 100%, 50%)"
    } );
```

The next example is more interesting. It creates a rectangle centered at the position (100, 100) in the graph and then rotates the rectangle smoothly around its center. (D3 default interpolators understand some SVG transformations, but this example demonstrates how to write your own interpolator in case you need to.)

```
d3.select( "#custom" ).append( "rect" )
    .attr( "x", 80 ).attr( "y", 80 )
    .attr( "width", 40 ).attr( "height", 40 )
    .transition().duration( 2000 ).ease( t=>t )
    .attrTween( "transform", function() {
        return t => "rotate(" + 360*t + ",100,100)"
    } );
```

Transition events. Transitions emit custom events when they start, end, and are interrupted. Using the `on()` method, you can register an event handler on a transition, which will be called when the appropriate lifecycle event is emitted. (See the D3 Reference Documentation for details.)

Easings. Using the `ease()` method, you can specify an *easing*. The purpose of an easing is to "stretch" or "compress" the time seen by the interpolator, allowing the animation to "ease into and out of" the movement. This often enhances the visual affect of an animation dramatically. As a matter of fact, "slow-in, slow-out" has been recognized by animators at Disney as one of the "Twelve Principles of Animation" (see "Principles of Animation" on page 74). But at other times, when they don't match well with the way the user *expects* an object to behave, easings can be downright confusing. The balance is definitely subtle.

An easing takes a parameter in the interval [0, 1] and maps it to the same interval, starting for $t = 0$ at 0 and ending for $t = 1$ at 1. The mapping is typically nonlinear (otherwise, the easing is just the identity). The default easing is `d3.easeCubic`, which implements a version of "slow-in, slow-out" behavior.

Technically, an easing is simply a mapping that is applied to the time parameter t before it is passed to the interpolator. This makes the distinction between the easing and the interpolator somewhat arbitrary. What if a custom interpolator itself mangles the time parameter in some nonlinear way? From a practical point of view, it seems best to treat easings as a *convenience feature* that adds "slow-in, slow-out" behavior to standard interpolators. (D3 includes a confusingly

large range of different easings, some of which considerably blur the distinction between an easing and what should be considered a custom interpolator.)

Don't overuse transitions. Transitions can be overused. A general problem with transitions is that they usually can't be interrupted by the user: the resulting forced wait can quickly lead to frustration. When transitions are employed to allow the user to track the effects of a change, they aid understanding (see Figure 3-3 for a simple example). But when they are used just "for effect," they easily become tiring once the initial cuteness wears off. (Figure 4-3 can serve as a cautionary example in this spirit!)

Table 4-5. Methods to specify custom interpolators (trans is a Transition object)

Function	Description
`trans.attrTween( name, factory )`	Sets a custom interpolator for the named attribute. The second argument must be a factory method that returns an interpolator.
`trans.styleTween( name, factory )`	Sets a custom interpolator for the named style. The second argument must be a factory method that returns an interpolator.
`trans.tween( tag, factory )`	Sets a custom interpolator to be invoked during transitions. The first argument is an arbitrary tag to identify this interpolator, the second argument must be a factory method that returns an interpolator. The effect of the interpolator is not restricted; it will be invoked purely for its side effects.

Principles of Animation

Classic animators at the Walt Disney Company have identified "Twelve Basic Principles of Animation" for creating credible, engaging animations. These were not intended for information visualization, of course, but a surprising number of them seem relevant, or at least worthy of consideration in the current context, although sometimes only in a very general way (such as "Staging," "Exaggeration," and "Appeal"). But what seems most noteworthy is that a remarkable four (out of twelve) emphasize the importance of *easing* in animations in some way (most obviously "slow-in, slow-out," but indirectly also "Anticipation," "Follow-Through," and "Squash-and-Stretch"). It is worth experimenting with an applica-

tion such as Example 4-5 to see how much of the overall effect is due to easing and timing (another one of the twelve principles).

Further reading:

- *The Illusion of Life: Disney Animation* by Ollie Johnston and Frank Thomas (Disney Editions, 1995)
- Abbreviated version online: *FrankandOllie.com*
- Also see the Wikipedia entry.

Animation with Timer Events

Transitions are a convenience technique to transform a configuration smoothly into another, but they are not intended as framework for general animations. To create those, it is generally necessary to work on a lower level. D3 includes a special timer that will invoke a given callback *once per animation frame*, that is, every time the browser is about to repaint the screen. The time interval is not configurable because it is determined by the browser's refresh rate (about 60 times per second or every 17 milliseconds, for most browsers). It is also not *exact*; the callback will be passed a high-precision timestamp that can be used to determine how much time has passed since the last invocation (see Table 4-6).

Table 4-6. Functions and methods for creating and using timers (t is a Timer object)

Function	Description
`d3.timer( callback, after, start )`	Returns a new timer instance. The timer will invoke the callback perpetually once per animation frame. When invoked, the callback will be passed the apparent elapsed time since the timer started running. (Apparent elapsed time does not progress while the window or tab is in the background.) The numeric `start` argument may contain a timestamp, as returned by `d3.now()`, at which the timer is scheduled to begin (it defaults to now). The numeric `after` argument may contain a delay, in milliseconds, which will be added to the start time (it defaults to 0).
`d3.timeout( callback, after, start )`	Like `d3.timer()`, except that the callback will be invoked exactly once.

Function	Description
`d3.interval( callback, interval, start )`	Similar to `d3.timer()`, except that the callback will only be invoked every `interval` milliseconds.
`t.stop()`	Stops this timer. Has no effect if the timer is already stopped.
`d3.now()`	Returns the current time, in milliseconds.

Example: Real-Time Animations

Example 4-6 creates a smooth animation by updating a graph for every browser repaint. The graph (see the left side of Figure 4-4) draws a line (a *Lissajous curve*[5]) that slowly fades as time goes on. In contrast to most other examples, this code does not use binding—mostly because there is no data set to bind! Instead, at each time step, the next position of the curve is calculated and a new `<line>` element is added to the graph from the previous position to the new one. The opacity of all elements is reduced by a constant factor, and elements whose opacity has fallen so low as to be essentially invisible are removed from the graph. The current value of the opacity is stored in each DOM `Node` itself as a new, "bogus" property. This is optional; you could instead store the value in a separate data structure keyed by each node (for example, using `d3.local()`, which is intended for this purpose), or query the current value using `attr()`, update, and reset it.

Example 4-6. Real-time animation (see the left side of Figure 4-4)

```
function makeLissajous() {
    var svg = d3.select( "#lissajous" );

    var a = 3.2, b = 5.9;                // Lissajous frequencies
    var phi, omega = 2*Math.PI/10000;    // 10 seconds per period

    var crrX = 150+100, crrY = 150+0;
    var prvX = crrX, prvY = crrY;

    var timer = d3.timer( function(t) {
        phi = omega*t;

        crrX = 150+100*Math.cos(a*phi);
        crrY = 150+100*Math.sin(b*phi);
```

5 See *http://mathworld.wolfram.com/LissajousCurve.html*.

```
        svg.selectAll( "line" )
            .each( function() { this.bogus_opacity *= .99 } )
            .attr( "stroke-opacity",
                   function() { return this.bogus_opacity } )
            .filter( function() { return this.bogus_opacity<0.05 } )
            .remove();

        svg.append( "line" )
            .each( function() { this.bogus_opacity = 1.0 } )
            .attr( "x1", prvX ).attr( "y1", prvY )
            .attr( "x2", crrX ).attr( "y2", crrY )
            .attr( "stroke", "green" ).attr( "stroke-width", 2 );

        prvX = crrX;
        prvY = crrY;

        if( t > 120e3 ) { timer.stop(); } // after 120 seconds
    } );
}
```

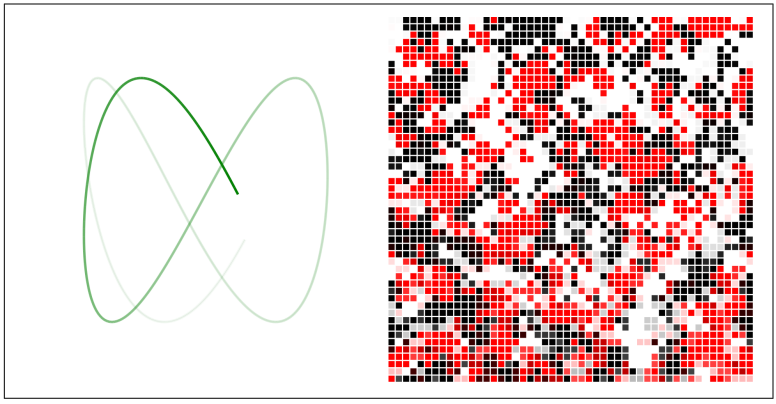


Figure 4-4. Animations: a Lissajous figure (left, see Example 4-6), and a voter model (right, see Example 4-7)

Example: Smoothing Periodic Updates with Transitions

In the previous example, each new data point to be displayed was calculated in real time. That's not always possible. Imagine that you need to access a remote server for data. You might want to poll it periodically, but certainly not for each repaint. In any case, a remote fetch is always an asynchronous call and needs to be handled accordingly.

In such a situation, transitions can help to create a better user experience by smoothing out the time periods between updates from the data source. In Example 4-7, the remote server has been replaced by a local function to keep the example simple, but most of the concepts carry over. The example implements a simple *voter model*:[6] at each time step, each graph element randomly selects one of its eight neighbors and adopts its color. The update function is called only every few seconds; D3 transitions are used to update the graph smoothly in the meantime (see the right side of Figure 4-4).

Example 4-7. Using transitions to smooth out periodic updates (see the right side of Figure 4-4)

```
function makeVoters() {
    var n = 50, w=300/n, dt = 3000, svg = d3.select( "#voters" );

    var data = d3.range(n*n)                                          ❶
        .map( d => { return { x: d%n, y: d/n|0,
                              val: Math.random() } } );

    var sc = d3.scaleQuantize()                                       ❷
        .range( [ "white", "red", "black" ] );

    svg.selectAll( "rect" ).data( data ).enter().append( "rect" )     ❸
        .attr( "x", d=>w*d.x ).attr( "y", d=>w*d.y )
        .attr( "width", w-1 ).attr( "height", w-1 )
        .attr( "fill", d => sc(d.val) );

    function update() {                                               ❹
        var nbs = [ [0,1], [0,-1], [ 1,0], [-1, 0],
                    [1,1], [1,-1], [-1,1], [-1,-1] ];
        return d3.shuffle( d3.range( n*n ) ).map( i => {
            var nb = nbs[ nbs.length*Math.random() | 0 ];
            var x = (data[i].x + nb[0] + n)%n;
            var y = (data[i].y + nb[1] + n)%n;
            data[i].val = data[ y*n + x ].val;
        } );
    }

    d3.interval( function() {                                         ❺
        update();
        svg.selectAll( "rect" ).data( data )
            .transition().duration(dt).delay((d,i)=>i*0.25*dt/(n*n))
```

6 See *http://mathworld.wolfram.com/VoterModel.html*.

```
            .attr( "fill", d => sc(d.val) ) }, dt );
}
```

❶ Creates an array of n^2 objects. Each object has a random value between 0 and 1, and also knowledge of its x and y coordinates in a square. (The odd `d/n|0` expression is a shorthand way to truncate the quotient on the left to an integer: the bit-wise OR operator forces its operands into an integer representation, truncating the decimals in the process. This is a semi-common JavaScript idiom that is worth knowing.)

❷ The object returned by `d3.scaleQuantize()` is an instance of a *binning scale*, which splits its input domain into equally sized bins. Here, the default input domain `[0,1]` is split into three equally sized bins, one for each color. (See Chapter 7 for more detail about scale objects.)

❸ Binds the data set and then creates a rectangle for each record in the data set. Each data record contains information about the position of the rectangle, and the scale object is used to map each record's value property to a color.

❹ The actual update function that computes a new configuration when called. It visits each element of the array in random order. For each element, it randomly selects one of its eight neighbors and assigns the neighbor's value to the current element. (The purpose of the arithmetic is to convert between the element's array index and its (x, y) coordinates in the matrix representation, while taking into account periodic boundary conditions: if you leave the matrix on the left, you loop back in on the right and vice versa; same for top and bottom.)

❺ The `d3.interval()` function returns a timer that invokes the specified callback at a configurable frequency. Here, it calls the `update()` function every `dt` milliseconds and then updates the graph elements with the new data. The updates are smoothed by a transition, which is delayed according to the position of the element in the array. The delay is short compared with the duration of the transition. The effect is that the update sweeps from top to bottom across the figure.

CHAPTER 5

Generators, Components, Layouts: Drawing Curves and Shapes

In this chapter, we will explore some basic *graphical* building blocks provided by D3. In the previous chapters (Chapters 3 and 4), we learned how D3 manipulates the DOM tree in order to represent information visually, but we did not really concentrate on graphical objects. They will be the topic for the present chapter. This will also give us the opportunity to understand how different D3 facilities are structured and how they work together. We will also learn mechanisms to organize our own code for convenience and reuse.

Generators, Components, and Layouts

SVG provides only a small set of built-in geometric shapes (like circles, rectangles, and lines; see Appendix B). Anything else has to be built up painstakingly (and rather painfully) either from those basic shapes or by using the `<path>` element and its turtle graphics command language. How does D3 assist with this?

It is worth remembering that D3 is a JavaScript library for manipulating the DOM tree, not a graphics package. It doesn't paint pixels, it operates on DOM elements. It should therefore come as no surprise that the way D3 facilitates graphics operations is by organizing and streamlining the way DOM elements are handled.

To produce complex figures, D3 employs three different styles of helper functions. These can be distinguished by the scale they act

on: *generators* generate individual attributes, *components* create entire DOM elements, and *layouts* determine the overall arrangement of the whole figure (see Figure 5-1). Pay close attention, because none of them does *quite* what one might expect.

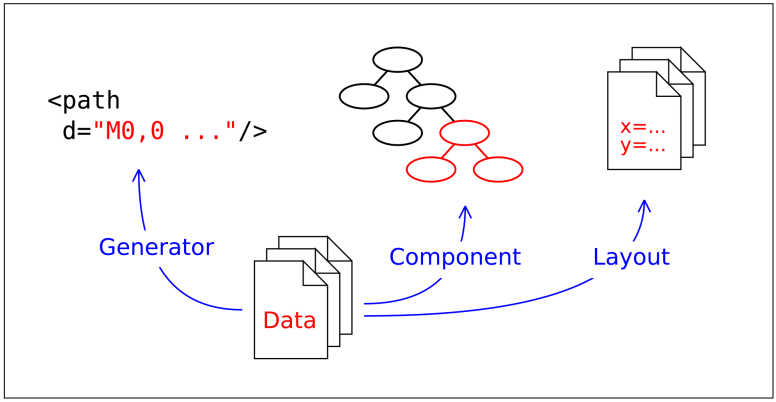


Figure 5-1. Different D3 constructs to represent data visually: generators create a command string for <path> elements, components modify the DOM tree, and layouts add pixel coordinates and other information to the data set itself.

Generators

Generators are wrappers around the `<path>` element's turtle graphics command language: they free the programmer from having to compose the cryptic command string. Generators typically consume a data set and *return a string that is suitable for the `<path>` element's `d` attribute*. But they don't create the `<path>` element: it's up to the surrounding code to make sure such an element exists. We will see examples of generators in this chapter, and in Chapter 9.

Components

Components are functions that inject newly created elements into the DOM tree. They always take a `Selection` as argument, to indicate where in the DOM tree the new elements will be added. Often the selection will be a `<g>` element as a container for the newly created DOM elements. Components return nothing; *they are usually invoked synthetically using the `call()` function* of the destination `Selection`. Components do modify the DOM tree: they are the most "active" tools in the D3 toolbox.

The `axMkr` in Example 2-6 was an example of a component, as was the drag-and-drop behavior component in Example 4-4.

Layouts
: Layouts consume a data set and calculate the pixel coordinates and angles that graphical elements should have to represent the data set in some nontrivial way. For example, in Chapter 9 we will see a layout that consumes a hierarchical data set and calculates the pixel coordinates for each node in order to represent the data set as a tree diagram. Layouts *return a data structure that can be bound to a `Selection`*. But they don't create or place any graphical elements themselves; they merely calculate the coordinates where the elements should go. It's up to the calling code to make use of this information. We'll see examples later in this chapter, and several more in Chapters 8 and 9.

All of these helpers are implemented as *function objects*. They perform their primary task when invoked as a function. But they also have state and expose an API of member functions to modify and configure their behavior. The general workflow is therefore as follows:

1. Create an instance of the desired helper.
2. Configure it as necessary (for example, by specifying accessor functions for the data set) using its member functions.
3. Invoke it in the appropriate evaluation context.

Frequently, all three steps are bundled into one, so that the instance is created, configured, and evaluated as part of an `attr()`, `call()`, or `data()` invocation. (See Example 2-6 for some examples.)

Finally, keep in mind that the same patterns are available and convenient for your own code. The *Component* pattern, in particular, is frequently useful to reduce code duplication, even for tiny, single-page projects. (We'll come back to this toward the end of this chapter.)

Symbols

Symbols are predefined shapes that can be used to represent individual data points in a graph (for example, in a scatter plot). This section introduces two different mechanisms to create symbols for use

in an SVG. The first uses the D3 symbol generator and the SVG `<path>` element. The second employs the SVG `<use>` tag, which allows you to reuse a fragment of an SVG document elsewhere in the document.

Using D3 Built-Ins

D3 defines seven built-in symbol types for use with the `d3.symbol()` generator; see Figure 5-2. Example 5-1 demonstrates how you can create and use them; the resulting graph is shown in Figure 5-3.

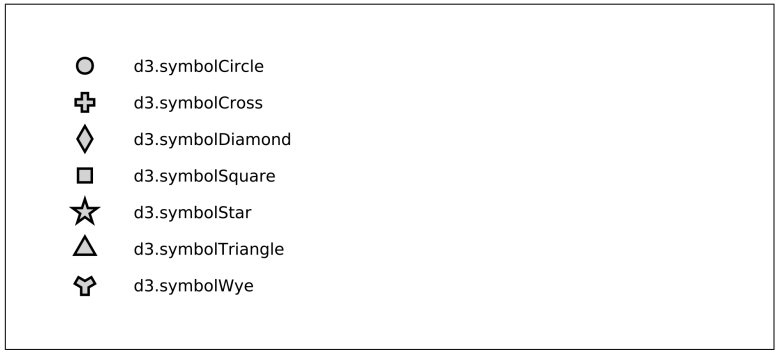


Figure 5-2. Predefined D3 symbol shapes. (If the name of the last symbol confuses you, try pronouncing it!)

The symbol generator is a bit different than the other D3 shape generators because it does not consume a data set. All you can configure are the symbol type and size (see Table 5-1). In particular, there are no provisions to fix the *position* of the symbol on the graph! Instead, all symbols are rendered at the origin, and you use SVG transforms to move them into their final positions. This is a common idiom when working with D3 and SVG; we will see it often. Some useful information on SVG transformations is in "SVG Transformations" on page 90.

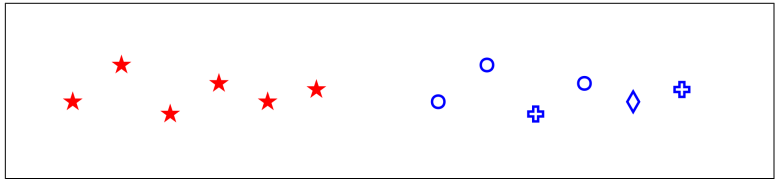


Figure 5-3. Using predefined symbols to represent data

Example 5-1. Commands for Figure 5-3

```
function makeSymbols() {
    var data = [ { "x":  40, "y":   0, "val": "A" },                    ❶
                 { "x":  80, "y":  30, "val": "A" },
                 { "x": 120, "y": -10, "val": "B" },
                 { "x": 160, "y":  15, "val": "A" },
                 { "x": 200, "y":   0, "val": "C" },
                 { "x": 240, "y":  10, "val": "B" } ];

    var symMkr = d3.symbol().size(81).type( d3.symbolStar );            ❷
    var scY = d3.scaleLinear().domain([-10,30]).range([80,40]);         ❸

    d3.select( "#symbols" ).append( "g" )                               ❹
        .selectAll( "path" ).data(data).enter().append( "path" )        ❺
        .attr( "d", symMkr )                                            ❻
        .attr( "fill", "red" )
        .attr( "transform",                                             ❼
               d=>"translate(" + d["x"] + "," + scY(d["y"]) + ")" );

    var scT = d3.scaleOrdinal(d3.symbols).domain(["A","B","C"]);        ❽

    d3.select( "#symbols" )
        .append( "g" ).attr( "transform", "translate(300,0)" )          ❾
        .selectAll( "path" ).data( data ).enter().append( "path" )
        .attr( "d", d => symMkr.type( scT(d["val"]) )() )               ❿
        .attr( "fill", "none" )                                         ⓫
        .attr( "stroke", "blue" ).attr( "stroke-width", 2 )
        .attr( "transform",                                             ⓬
               d=>"translate(" + d["x"] + "," + scY(d["y"]) + ")" );
}
```

❶ Define a data set, consisting of *x* and *y* coordinates and an additional "value."

❷ The `d3.symbol()` factory function returns a symbol generator instance. Here, we immediately configure the size of the symbols and select the star shape as the type to use. (The size parameter is proportional to the symbol's *area*, not its radius.)

❸ Conveniently, the x values in the data set will work as pixel coordinates, but we need a scale object to translate the y values to vertical positions. The inverted `range()` interval compensates for the upside-down graphical coordinates used by SVG.

❹ Create an initial selection and append a `<g>` element. This `<g>` will hold the symbols on the left side of the graph and keep them separate from the ones on the right.

❺ Bind the data and create a new `<path>` element for each data point.

❻ Populate the `d` attribute of the previously created `<path>` element using the symbol generator. Because the symbol generator is already fully configured, we don't need to do anything else at this point. The symbol generator is not evaluated, but instead is supplied as a function; it will automatically be invoked for each data point by the selection.

❼ Move each newly created `<path>` element to its final position through an SVG transform, using the scale object to calculate the vertical offset.

❽ For the right side of the graph, we will change the shape of the the symbol according to the third column in the data set. To do so, we need to associate the values from the data set with symbol shapes. An ordinal (or discrete) scale is essentially a hashmap that associates each value in the input domain with a value in the array `d3.symbols` of available symbol shapes. (See Chapter 7 to learn more about the different scale objects.)

❾ Append a new `<g>` element for the second set of symbols, and shift it to the right. This shift will also apply to all children of the `<g>` element.

❿ We can reuse the symbol generator instance created earlier. Here, every time the symbol generator is invoked, its type is set explicitly, based on the value in the data set.

⓫ For a change, the symbols are not filled; only the outlines are shown.

⓬ The transform to move each symbol into position is exactly the same as before. There is no need to change it, because the entire containing `<g>` element has been moved to avoid overprinting the stars on the left side.

This example demonstrates the basic techniques; many further variations are possible. For example, not only the shape but also the size and color of each symbol can be varied dynamically. To choose a color, the ordinal scale will come in handy again, like so:

```
d3.scaleOrdinal(["red","green","blue"]).domain(["A","B","C"]);
```

Or choose contrasting colors for interior and boundary (`fill` and `stroke`) to achieve a different effect.

Table 5-1. Methods of the symbol generator (sym is a symbol generator)

Function	Description
`d3.symbol()`	Returns a new symbol generator. Unless configured otherwise, the generator will create a circle of 64 square pixels.
`sym()`	Returns a string, suitable as value of the d attribute of a `<path>` element, that will render a symbol of the desired shape.
`sym.type(shape)`	Sets the symbol shape. The argument should be one of the predefined symbol shapes shown in Figure 5-2. Returns the current value if called without argument.
`sym.size(area)`	Sets the symbol size. The argument is interpreted as the approximate *area* of the symbol (in pixels, before any scale transformation is applied). Returns the current value if invoked without an argument. The default size is 64 square pixels.
`d3.symbols`	An array (not a function), containing the predefined symbol shapes.

Custom Symbols

The way generators are used in Example 5-1 may appear a bit opaque, but there is really no magic here. Ultimately, all that happens is that the generator instance returns a string of commands using the `<path>` element's command syntax. You can do that. For example, if you define the following function:[1]

```
function arrow() {
    return "M0 0 L16 0 M8 4 L16 0 L8 -4";
}
```

then you can replace step 6 in Example 5-1 with:

```
.attr( "d", arrow )
```

1 Internally, the D3 symbol generator uses an HTML5 `<canvas>` element for performance reasons, but there is no strict need to do so.

Now try applying a data-dependent *rotation*, using an appropriate SVG transform, to the generated `<path>` element! There are many more options.

SVG Fragments as Symbols

The `<use>` tag lets you reuse arbitrary SVG fragments within a document and hence provides an alternative for creating reusable symbols in SVG.[2] It is recommended (but not required) that components intended for reuse are defined inside of the SVG document's `<defs>` section. (The `<defs>` section must be part of the SVG document; do not place it into the header of the HTML page, for example!) Example 5-2 shows a document that defines a reusable component inside its `<defs>` section, and Example 5-3 shows how one might use it. (See Figure 5-4 for the result.)

Example 5-2. An SVG defining a symbol in the <defs> section (also see Example 5-3 and Figure 5-4).

```
<svg id="usedefs" width="275" height="100">
  <defs>                                                    ❶
    <g id="crosshair">                                      ❷
      <circle cx="0" cy="0" r="2" fill="none"/>             ❸
      <line x1="-3" y1="0" x2="-1" y2="0" />
      <line x1="1" y1="0" x2="3" y2="0" />
      <line x1="0" y1="-1" x2="0" y2="-3" />
      <line x1="0" y1="1" x2="0" y2="3" />
    </g>
  </defs>
</svg>
```

❶ Open a `<defs>` section as a child of the `<svg>` element.

❷ Create a `<g>` element that will be the container for all elements making up the symbol, and assign an `id` to it. The `<use>` tag will refer to this symbol via the identifier specified here.

❸ Add the graphical elements for the symbol. Remember that the symbol can be rescaled when it is being used, hence its absolute size is not important here.

2 Although current browsers don't seem to have a problem with the `<use>` tag, I have found that not all SVG tools handle it properly. Be aware.

Example 5-3. Creating symbols with <use> tags (also see Figure 5-4).

```
function makeCrosshair() {
    var data = [ [180, 1], [260, 3], [340, 2], [420, 4] ];          ❶

    d3.select( "#usedefs" )
        .selectAll( "use" ).data( data ).enter().append( "use" )    ❷
        .attr( "href", "#crosshair" )                               ❸
        .attr( "transform",                                         ❹
               d=>"translate("+d[0]+",50) scale("+2*d[1]+")" )
        .attr( "stroke", "black" )                                  ❺
        .attr( "stroke-width", d=>0.5/Math.sqrt(d[1]) );            ❻
}
```

❶ A minimal data set. The first component of each record will be used for the horizontal position, the second one for the symbol size.

❷ Select the SVG element and bind data as usual, creating a `<use>` element for each record in the data set.

❸ Set the `href` attribute for every newly created `<use>` element to the symbol identifier. The final element in the page will look something like: `<use href="#crosshair" ...>`.

❹ Use the `transform` attribute to choose the symbol's position and size.

❺ Set the stroke color. It is necessary to do so explicitly, because the default value of the `stroke` attribute is `none`!

❻ Set the stroke width. To obtain strokes of roughly equal width in the figure, the stroke width must be chosen so as to *cancel* the effect of the earlier scale transformation.

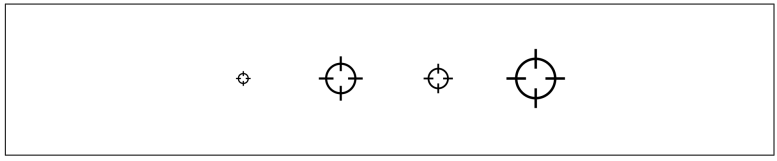


Figure 5-4. Creating symbols with <use> tags (also see Example 5-3)

Of course, the contents of the `<defs>` section is no different than the rest of the document, which means that it, too, can be generated

dynamically using D3. For example, the following snippet creates a diagonal cross (also called a "saltire") that can be invoked through `<use>` tags:[3]

```
var d = d3.select("svg").append("defs")
    .append("g").attr("id", "saltire");
d.append("line")
    .attr("x1",-1).attr("y1",1).attr("x2",1).attr("y2",-1);
d.append("line")
    .attr("x1",-1).attr("y1",-1).attr("x2",1).attr("y2",1);
```

Yet another way is to load the appropriate SVG from a file and insert it into the current document (see Chapter 6 for an example).

SVG Transformations

You can apply a *transform* to any SVG element by setting the element's `transform` attribute to an appropriate value. Although you can specify an arbitrary affine transformation directly by supplying the individual matrix elements explicitly (see Appendix B), SVG also provides an alternative, human-readable syntax for the common operations of scaling, rotation, and translation.

In this human-readable format, transformations are specified using syntax that resembles JavaScript function calls:

```
scale( fx, fy )     // if fy is omitted, fx is used for both
rotate( phi, x, y ) // rotate around (x,y), (0,0) if missing
translate( dx, dy )
```

You may build up a transformation as a combination of these "functions":

```
<rect transform="translate(10,20) scale(2.0) rotate(30)" />
```

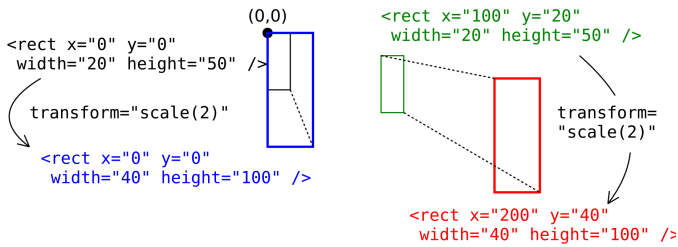
Notice that the individual transformations of a combination are always applied *from right to left* (as in matrix multiplication). In the above snippet, the rectangle is first rotated by 30 degrees (*clockwise*—remember that the y-axis points down!), *then* scaled up by a factor of 2, and finally shifted right and down (orientation of the y-axis, again). It is permissible to invoke the same "function" multiple times in a single attribute value:

3 Another way to create such a symbol is to rotate `d3.symbolCross` by 45 degrees using an SVG transformation!

```
<rect x="100" y="200"
      transform="translate(100,200)
                 rotate(30)
                 translate(-100,-200)" />
```

This creates a rectangle at (100, 200), shifts it into the origin, rotates it (around the origin), then shifts the rotated rectangle back to its initial location.

This all may seem straightforward and intuitive enough, but it can lead to confusing results because it is easy to forget that the human-readable "functions" are nothing more than shortcuts for the underlying matrix operations. The single biggest source of confusion is that *scaling and rotating objects that are not located at the origin has side effects*. It is easy to believe, for example, that `transform="scale(2)"` simply scales up the object by a factor of two, but that's not what's happening! Instead, *all* of the object's coordinates will be scaled up by a factor of two, including its location. Unless the object is located at the origin, `transform="scale(2)"` therefore results in a translation (in addition to the scaling). The same effect occurs if you translate an object and then scale it: the translation is scaled in addition to the item size. (See the figure in this sidebar.)



Effect of a scale transformation on an object located at the origin (indicated through a dot) and on an object away from the origin

Fortunately, you can avoid all these confusions by adhering to the following workflow:

1. Create objects only at the origin.
2. Apply any desired scalings and rotations while the object is still at the origin.

3. Only then move (translate) the object to its intended location.

The `<g>` element is frequently useful in the context of transformations, because any transformation applied to this element is applied to all of its children. This makes it possible to assemble a complex graphical object from its constitutents, and then move the entire aggregate to its desired location at once. This is the common pattern when working with D3 components (see Example 5-8 for a typical example).

One final observation: although I have not found an explicit reference in the SVG 2 Draft Standard, it appears that transformations are applied *last* when an element is rendered. This implies that, for instance, even an explicitly specified stroke width will be affected by a scale transformation. (For example, `stroke-width="1" transform="scale(2)"` results in an outline that is two pixels wide.) You should take this effect into account when working with transformations.

Lines and Curves

Drawing lines and curves follows precisely the workflow outlined at the beginning of this chapter: you instantiate a generator, feed it a data set, and then invoke it to obtain a string suitable for the `d` attribute of a `<path>` element.

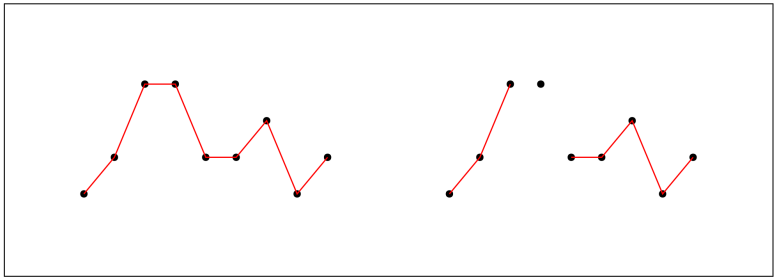


Figure 5-5. Creating lines to connect dots. On the right side, one of the data points has been marked as "undefined" and breaks the line into two segments.

Example 5-4. Creating lines (see Figure 5-5)

```
function makeLines() {
    // Prepare a data set and scale it properly for plotting
```

```
    var ds = [ [1, 1], [2, 2], [3, 4], [4, 4], [5, 2],
               [6, 2], [7, 3], [8, 1], [9, 2] ];
    var xSc = d3.scaleLinear().domain([1,9]).range([50,250]);
    var ySc = d3.scaleLinear().domain([0,5]).range([175,25]);
    ds = ds.map( d => [xSc(d[0]), ySc(d[1])] );                    ❶

    // Draw circles for the individual data points
    d3.select( "#lines" ).append( "g" ).selectAll( "circle" )      ❷
        .data( ds ).enter().append( "circle" ).attr( "r", 3 )
        .attr( "cx", d=>d[0] ).attr( "cy", d=>d[1] );

    // Generate a line
    var lnMkr = d3.line();                                         ❸
    d3.select( "#lines" ).append( "g" ).append( "path" )           ❹
        .attr( "d", lnMkr(ds) )                                    ❺
        .attr( "fill", "none" ).attr( "stroke", "red" );           ❻
}
```

❶ Here, we apply the scale operations to the entire data set ahead of time, rather than invoking them on an as-needed basis later, in order to keep the subsequent code simpler.

❷ We use `<circle>` elements to mark the individual data points. (We could have used symbols instead, but the `transform` syntax required to move them into position is clumsier.)

❸ Generate an instance of the line generator.

❹ Remember that a line is ultimately realized as a `<path>` element, hence we must first create a `<path>` element that will subsequently be configured.

❺ Invoke the line generator, with the data set as argument.

❻ Fix the presentational attributes.

Three aspects of the line generator are configurable (see Table 5-2). Using the `x()` and `y()` methods, you can set the accessor functions that pick out the *x* and *y* coordinates from the data set. The accessors are called with three arguments:

```
function( d, i, data ) { ... }
```

where `d` is the current record in the data set, `i` is its index, and `data` is the entire data set itself. If you don't supply your accessors, then

the default is used, which selects the first and second column from a multidimensional array. (This was done in Example 5-4.)

Another configuration option allows you to mark certain points as "undefined," by passing an accessor to the `defined(accessor)` function. The accessor must return a Boolean value; if it is `false` for some data points, then the generated line will exclude those points, resulting in a break that separates the line into segments. An example should make this clear. If you replace the line:

```
var lnMkr = d3.line();
```

in Example 5-4 with:

```
var lnMkr = d3.line().defined( (d,i) => i==3 ? false:true );
```

then the point at index position 3 is treated as undefined, resulting in the graph on the right side of Figure 5-5. You can use this feature to exclude certain data points, for example, those that fall outside the desired plot range or where some quantity changes discontinuously, without having to remove them from the data set itself.

The shape and nature of the curve that is drawn to connect the individual points can be changed through curve factories. (They will be discussed in the next section.) Finally, be aware that the line generator will connect points in the sequence in which they *appear in the data set*. This implies that you may need to sort points (for example, by their *x* coordinates) before passing them to the line generator. The following snippet uses the `sort()` function of JavaScript's `Array` type to sort the data set from Example 5-4 by the *x* coordinates of the data points:

```
ds = ds.sort( (a,b) => a[0]>b[0] )
```

As usual, the line generator's member functions in Table 5-2 can act both as getters and setters. When called without arguments, they act as getters and return the current value. When called as setters, they return the current generator instance, in order to allow method chaining.

Table 5-2. Methods of the line generator (mkr is a line generator)

Function	Description
`d3.line()`	Returns a new line generator instance.
`mkr(data)`	When called with a data set, returns a string, suitable for the `d` attribute of a `<path>` element, that will draw a line through the points in the data set.

Function	Description
`mkr.x(accessor)`, `mkr.y(accessor)`	Set accessor functions for the *x* and *y* coordinates of each data point. Each accessor will be called with three arguments: the current record `d` in the data set, its index `i`, and the entire data set `data`. The accessors must return the *x* or *y* coordinates for the current data point, respectively. The default accessors select the first two columns in a two-dimensional data set.
`mkr.defined(accessor)`	Sets an accessor that allows to mark arbitrary points as undefined. The accessor is called with same three arguments as the coordinate accessors, but must return a Boolean value; points for which the accessor returns `false` are treated as undefined.
`mkr.curve(curve)`	Sets the curve factory to be used.

Built-In Curves

In addition to straight lines, D3 offers a variety of other curve shapes to connect consecutive points—and you can also define your own. You choose a different curve shape by passing the appropriate *curve factory* to the line generator, like so:

```
var lnMkr = d3.line().curve( d3.curveLinear );
```

The built-in curve factories (see Figure 5-6) fall into two major groups:

- Curves that are completely determined by the data set and that pass exactly through all data points (see Table 5-3).
- Curves that depend on an additional curvature or stiffness parameter and that may not pass exactly through the data points (see Table 5-4).

The built-in curves are primarily intended for information visualization; they were not developed for scientific curve plotting. Curves are drawn in the order of the points in the data set; data points are not reordered based on the *x* coordinates! The built-in curves make no direct provisions for curve fitting, for filtering noisy data, or for weighting data points according to their local error. If you need such functionality, you will have to implement it yourself, for example, by implementing a custom curve generator (see the next section), or by first processing the input data separately. The D3 Reference Documentation contains additional details and references to the original literature.

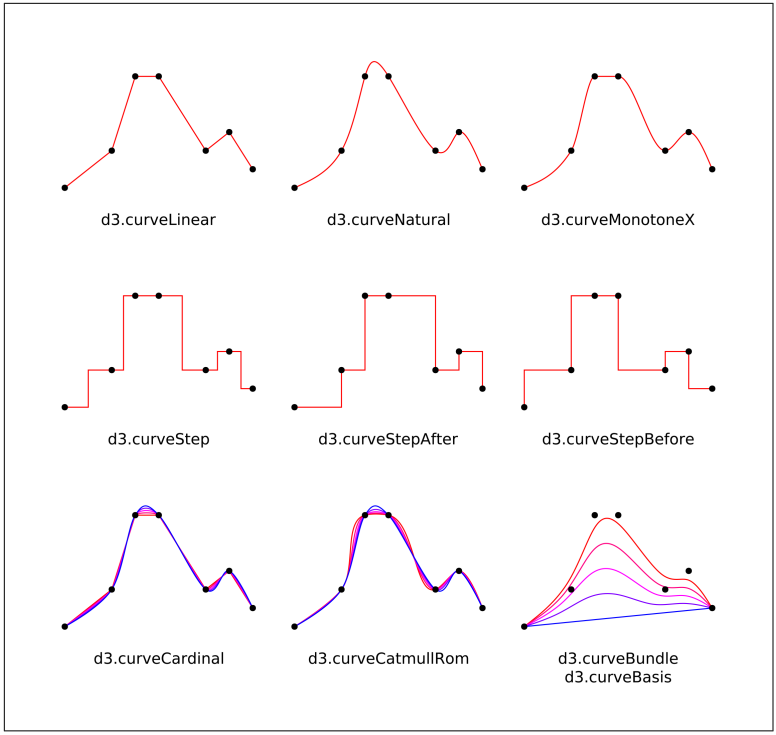


Figure 5-6. Built-in curve factories. The values of the adjustable parameter for the bottom row are 0.0 (blue), 0.25, 0.5, 0.75, and 1.0 (red).

Curves without an adjustable parameter

In addition to straight lines and step functions, D3 also provides several spline implementations to connect consecutive points. Natural splines match direction and curvature where line segments meet, and have zero curvature at the ends of the data set. The curve created by `d3.curveMonotoneX` does not overshoot the data points in the vertical when passing through the points in the horizontal direction (similarly for `d3.curveMonontoneY`).

Table 5-3. Factories for curves without an adjustable parameter

Straight lines	Splines	Step
`d3.curveLinear`	`d3.curveNatural`	`d3.curveStep`
`d3.curveLinearClosed`	`d3.curveMonotoneX`	`d3.curveStepAfter`
	`d3.curveMonotoneY`	`d3.curveStepBefore`

Curves with an adjustable parameter

Both cardinal and Catmull-Rom splines pass through the data points exactly, but relax the condition that the curvatures should match at all line segment joints. Instead, they impose additional constraints on the local tangents at the joints. Both depend on an adjustable parameter that controls their shape; this parameter should take values between 0 and 1. Cardinal and Catmull-Rom splines are identical to each other if this parameter is zero; they differ in their behavior if the parameter value increases. The curve factories provide methods to set the parameter, for example:

```
d3.curveCardinal.tension(0.5);
d3.curveCatmullRomClosed.alpha(0.25);
```

The curves created by `d3.curveBundle` are not required to pass through the data points exactly. When the adjustable parameter is zero, the resulting curve coincides with the curve generated by `d3.curveBasis`.

Except for `d3.curveBundle`, the adjustable curve types provide *open* and *closed* variants (such as `d3.curveCardinalOpen` or `d3.curveBasisClosed`). The open curve factories use the end points of the data set for the determination of the curve shape, but do not extend the drawn curve to the actual end points (the curves stop at the second-to-last point). For the closed curves, the ends of the data set are connected, so that the first and last point are treated as neighbors. The resulting line forms a geometrically closed curve.

Table 5-4. Factories for curves that depend on an adjustable parameter

Curve factory	Adjustable parameter	Default value
`d3.curveCardinal`	`cardinal.tension(t)`	0
`d3.curveCatmullRom`	`catrom.alpha(a)`	0.5
`d3.curveBundle`	`bundle.beta(b)`	0.85
`d3.curveBasis`		

Splines

Splines are piecewise polynomial curves that are joined together smoothly. The various spline variants differ in details of how the segments are joined, and in the treatment of the endpoints. Some splines contain an adjustable parameter that controls the curvature of the resulting curve.

Because splines are constructed from polynomials, they can be evaluated cheaply (without the need to compute transcendental functions). This explains their popularity in computer graphics and similar applications. Moreover, keep in mind that D3 curves are implemented using the SVG `<path>` element. This element supports Bézier curves (in fact, those are the only curves, other than straight lines and elliptical arcs, that the `<path>` element can draw natively). Bézier curves themselves are also constructed from polynomials, hence it is possible to represent splines through Bézier curves exactly, which makes it natural to use splines with the `<path>` element.

Custom Curves

If you are dissatisfied with the built-in curves, you can provide your own curve algorithms by supplying a custom curve factory to the `line` generator—for example, to implement a fitting or smoothing method.[4] A curve factory is a function that accepts a single argument and returns an object that implements the curve API. When D3 invokes the factory function, it supplies a `d3.path` object as the argument. The `d3.path` facility exposes a turtle graphics interface similar to the command language of the `<path>` element. Your implementation of the curve API must populate this object according to your algorithm. The curve API, which your custom code must implement, consists of three functions: `lineStart()`, `lineEnd()`, and `point(x, y)`—their semantics will be explained in a moment. Eventually, D3 will turn the path object into a string that can be used to populate the `d` attribute of a `<path>` element.

Let's consider a very simple example, just to explain the mechanism. Example 5-5 implements a curve factory that draws a horizontal line at the *median* of the vertical positions of all the data points for each line segment. It behaves exactly like the built-in factories; for example, you could use it in Example 5-4 like so:

```
var lnMkr = d3.line().curve( curveVerticalMedian );
```

4 This section is probably best skipped on first reading.

Example 5-5. A simple curve factory

```
function curveVerticalMedian( context ) {
    return {
        lineStart: function() {
            this.data = [];                                          ❶
        },

        point: function( x, y ) {
            this.data.push( [x, y] );                                ❷
        },

        lineEnd: function() {
            var xrange = d3.extent( this.data, d=>d[0] );            ❸
            var median = d3.median( this.data, d=>d[1] );

            context.moveTo( xrange[0], median );                     ❹
            context.lineTo( xrange[1], median );                     ❺
        }
    };
}
```

❶ The `lineStart()` function is called at the *beginning* of every line segment (remember that every point data point marked as "undefined" ends the current and forces a new line segment). Here, an array is initialized that will hold the data points.

❷ The `point(x, y)` function is called for every data point with the coordinates of that point. In this case, the coordinates of each point are simply appended to the end of the array.

❸ The `lineEnd()` function is called at the *end* of each line segment. Now that all the data points for the current line segment have been collected, the horizontal extent of the data set and the median of the vertical positions can be found, and with this information, we are now ready to draw the desired line.

❹ The `context` variable is a `d3.path` instance, which is supplied to the curve factory by the D3 infrastructure. This object exposes functions similar to the command language of the `<path>` element. All drawing has to be done using this object. Here, we invisibly place the pen at the beginning of the line and…

❺ … draw a visible line to its end.

This is the basic workflow. With a little more work, you can change the `lineEnd()` function to calculate and draw, for example, a linear regression line, a locally weighted interpolation such as LOESS, or a kernel density estimate.[5]

If the resulting curve is not a straight line, then it will generally be necessary to build it up from individual pieces in a loop. In the following snippet, the horizontal plot interval is broken into 100 steps:

```
for( var t = xmin; t <= xmax; t += 0.01*(xmax-xmin) ) {
    context.lineTo( t, y(t) );
}
```

Here, `xmin` and `xmax` are the end points of the plot interval, and `y(t)` is the computed value of the curve at position `t`.

To calculate the median or a linear regression, the *entire* data set must be known before computation and drawing can begin. Other curves can be calculated and drawn incrementally—straight lines connecting consecutive points, for example. In this case, the main logic moves from the `lineEnd()` function to the `point()` function: for each new data point, the next part of the curve is calculated and the appropriate `d3.path` methods are called to draw it.

Circles, Arcs, and Pie Charts: Working with Layouts

So far, we have not encountered an example for a D3 *layout* yet. Layouts consume a data set and calculate where graph elements should be placed to visualize the data. They return a data structure containing this information, but don't actually create or place any DOM elements; instead, it is up to the surrounding code to do this. Some D3 generators are specifically designed to work with the data structure returned by a particular layout. Although both can be used separately, understanding the *layout* helps to understand the *generator*.

The D3 pie layout provides a particularly clear example to demonstrate how this combined process works, and I will describe it here in some detail. Pie charts are sometimes frowned upon, but the D3 pie layout is of interest nevertheless, not only to explain the D3 layout workflow, but also as an extremely versatile method for creating

5 See, for example, *Data Analysis with Open Source Tools* by Philipp K. Janert (O'Reilly).

any sort of circular graph, not just classical pie charts. For example, it's a fun exercise to use the pie layout to create color wheels in order to explore different color spaces! (See Chapter 8 for color spaces available in D3; Chapter 9 discusses D3 layouts for tree diagrams.)

Example 5-6 defines a data set describing, say, an election outcome: for each candidate, it gives the name and number of votes. Figure 5-7 shows a pie chart of this data set.

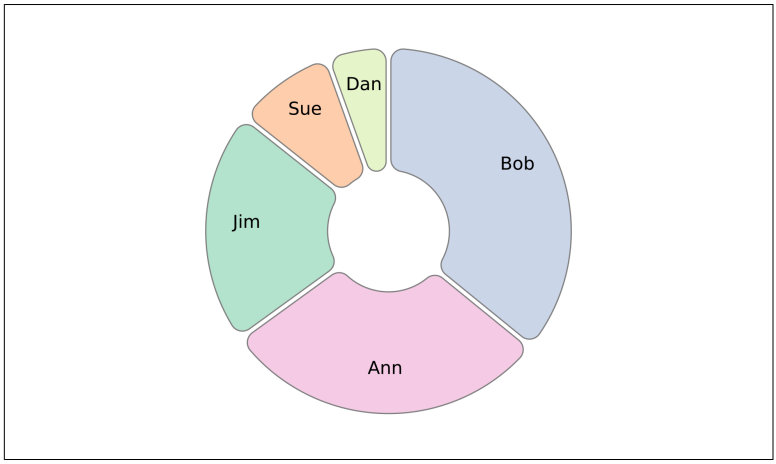


Figure 5-7. A pie chart (see Example 5-6)

Example 5-6. Using the pie layout and arc generator (see Figure 5-7)

```
function makePie() {
    var data = [ { name: "Jim", votes: 12 },
                 { name: "Sue", votes:  5 },
                 { name: "Bob", votes: 21 },
                 { name: "Ann", votes: 17 },
                 { name: "Dan", votes:  3 } ];

    var pie = d3.pie().value(d=>d.votes).padAngle(0.025)( data ); ❶

    var arcMkr = d3.arc().innerRadius( 50 ).outerRadius( 150 )    ❷
        .cornerRadius(10);

    var scC = d3.scaleOrdinal( d3.schemePastel2 )                 ❸
        .domain( pie.map(d=>d.index) )                            ❹

    var g = d3.select( "#pie" )                                   ❺
        .append( "g" ).attr( "transform", "translate(300, 175)" )

    g.selectAll( "path" ).data( pie ).enter().append( "path" )    ❻
```

```
        .attr( "d", arcMkr )                                          ❼
        .attr( "fill", d=>scC(d.index) ).attr( "stroke", "grey" );

    g.selectAll( "text" ).data( pie ).enter().append( "text" )        ❽
        .text( d => d.data.name )                                     ❾
        .attr( "x", d=>arcMkr.innerRadius(85).centroid(d)[0] )        ❿
        .attr( "y", d=>arcMkr.innerRadius(85).centroid(d)[1] )
        .attr( "font-family", "sans-serif" ).attr( "font-size", 14 )
        .attr( "text-anchor", "middle" );
}
```

❶ This line creates an instance of the pie layout, configures it, and invokes it on the data set, all in one step. The returned data structure `pie` is an array of objects with one object for each record in the original data set. Each object contains a reference to the original data, start and end angles of the associated pie slice, and so on.

❷ Creates and configures an arc generator (but does not invoke it).

❸ Creates a scale object that associates a color with each `pie` element. The colors are drawn from the built-in scheme `d3.schemePastel2`, which defines a set of gentle colors, suitable as background colors, but that nevertheless provide good visual contrast to each other.

❹ Ordinal scales look up objects by their string representation. Unfortunately, JavaScript's default `toString()` method does not return a unique identifier for an object, only a generic constant. For this reason, we can't map from the elements of the `pie` array directly and have to choose a uniquely identifying member of each element.

❺ Selects the destination element in the page, appends a `<g>` element as the common container of the chart components, and moves it into position. (The pie chart generated by the arc generator will be centered at the origin.)

❻ Bind the `pie` data structure…

❼ … and invoke the arc generator on each element.

❽ The remaining commands create the text labels in each pie slice.

❾ Each object in the `pie` array has a member `data` that contains a reference to the corresponding record in the original data set.

❿ The `centroid()` function on the arc generator returns the coordinates of a location that is centered inside each pie slice as a suitable position for a label. Here I change the inner radius of the arc generator in order to push the labels further toward the rim.

This is all. If you compare the API for the pie and the arc generator (Tables 5-5 and 5-6), you will notice how closely the default settings for the arc generator match the data structure returned by the pie layout, leading to the rather compact code in Example 5-6.

Table 5-5. Methods of the pie chart layout operator (mkr is an instance of the layout operator)

Function	Description
`d3.pie()`	Returns a new layout operator in default configuration.
`mkr( data )`	Takes an array of arbitrary records and produces an array of objects. Each output object represents one pie slice and has the following members: • `data` : a reference to the corresponding record in the original data set • `value` : the numeric value represented by the current pie slice • `index` : a positive integer giving the position of the current segment when following the sequence of segments in order of increasing angle around the chart • `startAngle`, `endAngle` : beginning and end of the pie slice, in radians, measured clockwise from the top (noon) position • `padAngle` : padding for the current pie slice The segments in the returned array are always sorted according to the sequence of values in the input data set; they are not necessarily sorted in order of ascending angular positions in the chart.

Function	Description
`mkr.value( arg )`	Numeric value or accessor function to return the relevant numerical value that the current slice represents. Defaults to: `d => d`.
`mkr.sort( comparator )`	Sets the comparator that is used to sort elements of the original data set. The sorting affects the assignment of elements to positions around the chart; it does not affect the order of segments in the returned array.
`mkr.sortValues( comparator )`	Sets the comparator that is used to sort elements of the original data set by their value. The sorting affects the assignment of elements to positions around the chart; it does not affect the order of segments in the returned array. By default, slices are sorted by value in descending order.
`mkr.startAngle( arg ),` `mkr.endAngle( arg )`	Numeric value or accessor function to return the overall start and end angle of the pie chart.
`mkr.padAngle( arg )`	Numeric value or accessor function to return any padding to be inserted between adjacent slices.

The arc generator can be used by itself to draw arbitrary circle segments even if they are not used as part of a pie chart, but in this case it is up to the user to provide the necessary information in a suitable form. The arc generator will fail unless start and end angles and inner and outer radius have been fixed; there are no default values. These parameters can be set either through API calls or be provided as an object when the generator is evaluated. In the latter case, the parameters are extracted from the object by the configured accessor functions. These techniques may also be mixed (see Example 5-7).

Example 5-7. Different ways to configure the arc generator. In all cases, g is a suitable <g> selection.

```
// API configuration
var arcMkr = d3.arc().innerRadius( 50 ).outerRadius( 150 )
    .startAngle( 1.5*Math.PI ).endAngle( 1.75*Math.PI );
g.append( "path" ).attr( "d", arcMkr() );

// Object with default accessors
var arcSpec = { innerRadius: 50, outerRadius: 150,
                startAngle: 0, endAngle: 0.25*Math.PI };
g.append( "path" ).attr( "d", d3.arc()( arcSpec ) );

// Mixed, nondefault accessors
var arcMkr = d3.arc().innerRadius( 50 ).outerRadius( 150 )
    .startAngle( d => Math.PI*d.start/180 )
```

```
    .endAngle( d => Math.PI*d.end/180 );
g.append( "path" ).attr( "d", arcMkr( { start: 60, end: 90 } ) );
```

Table 5-6. Methods of the arc generator (mkr is an arc generator)

Function	Description
`d3.arc()`	Returns an arc generator in default configuration.
`mkr( args )`	Returns a string suitable for the `<path>` element's `d` attribute, provided the start and end angle and the inner and outer radius have been defined either through API calls or by providing an object as the argument that can be interpreted by the configured accessors. Values set through API calls take precedence.
`mkr.innerRadius( arg )`, `mkr.outerRadius( arg )`	Sets the inner and outer radius to a constant value, or sets an accessor that extracts and returns them from the arguments provided when the generator is invoked. Defaults: `d => d.innerRadius`, `d => d.outerRadius`.
`mkr.startAngle( arg )`, `mkr.endAngle( arg )`	Sets the start and end angle to a constant value, or sets an accessor that extracts and returns them from the arguments provided when the generator is invoked. Angles are measured clockwise from the top (noon) position and must not be negative. Defaults: `d => d.startAngle`, `d => d.endAngle`.
`mkr.centroid()`	Returns the coordinates of a location that is centered inside the generated slice as a two-element array.
`mkr.cornerRadius( arg )`	Sets the radius by which the corners of a slice will be rounded, or sets an accessor that extracts and returns them. Default: `0`.

Other Shapes

D3 includes facilities for the generation and layout of other shapes:

- The `d3.area()` generator is related to the `d3.line()` generator, except that it creates areas that are bounded by *two* boundary lines.
- The `d3.lineRadial()` and `d3.areaRadial()` generators work with *polar coordinates* to make circular line and area plots.
- The `d3.stack()` layout is intended to be used with the `d3.area()` generator for the creation of stacked graphs (where several components are graphed together in such a way that the cumulative value of a set of quantities is displayed).

- The `d3.chord()` layout shows the mutual flows between nodes in a network as a circular arrangement. It is intended to be used together with the `d3.arc()` and the specialized `d3.ribbon()` generator.
- The `d3.links()` generator is used to draw the branches in tree diagrams of hierarchical data sets. It is intended to be used together with the `d3.tree()` and `d3.cluster()` layouts.
- The `d3.pack()`, `d3.treemap()`, and `d3.partition()` layouts prepare hierarchical data to be displayed in containment hierarchies.

The last two items of this list will be discussed in Chapter 9. Check the D3 Reference Documentation for information about the other items.

Writing Your Own Components

Compared to the variety of generators and layouts, D3 does not contain many built-in *components*. (The axis facility from Example 2-6 is one example, and the drag-and-drop behavior from Example 4-4 is another.) But components are the primary means to organize and simplify your own work.

Components are functions that take a `Selection` as argument and do something to it.[6] They return nothing; all their results are side effects.

Writing a component is useful whenever you want to repeatedly create several DOM elements together, in particular when their positioning relative to each other is important. But a component may also simply be a way of saving keystrokes on a recurring set of commands, or a method to encapsulate a bit of messy code. In many ways, components are to the DOM tree what functions are to code: a reusable set of actions that are always performed together.

6 Despite the name, components are not *things*, they are *actions*, although often the action results in a thing being added to the DOM tree.

A Simple Component

Let's say that we want to use, in multiple locations in a graph, a rectangle with rounded corners and a text label centered inside of it (see Figure 5-8). In this kind of situation, it is convenient to keep the relative positioning of the parts *within* the component (text and border) separate from the positioning of the entire component itself. The `sticker()` function in Example 5-8 creates both the rectangle and the text inside of it centered *at the origin*. It is then up to the calling code to move both of them together to the final position.

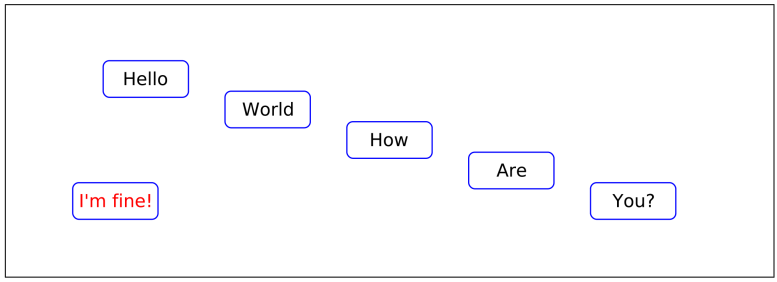


Figure 5-8. A reusable component

Example 5-8. Commands for Figure 5-8

```
function sticker( sel, label ) {                                    ❶
    sel.append( "rect" ).attr( "rx", 5 ).attr( "ry", 5 )            ❷
        .attr( "width", 70 ).attr( "height", 30 )
        .attr( "x", -35 ).attr( "y", -15 )
        .attr( "fill", "none" ).attr( "stroke", "blue" )
        .classed( "frame", true );                                  ❸

    sel.append( "text" ).attr( "x", 0 ).attr( "y", 5 )              ❹
        .attr( "text-anchor", "middle" )
        .attr( "font-family", "sans-serif" ).attr( "font-size", 14 )
        .classed( "label", true )
        .text( label ? label : d => d );                            ❺
}

function makeSticker() {
    var labels = [ "Hello", "World", "How", "Are", "You?" ];
    var scX = d3.scaleLinear()
        .domain( [0, labels.length-1] ).range( [100, 500] );
    var scY = d3.scaleLinear()
        .domain( [0, labels.length-1] ).range( [50, 150] );

    d3.select( "#sticker" )                                         ❻
        .selectAll( "g" ).data( labels ).enter().append( "g" )
```

```
        .attr( "transform",
               (d,i) => "translate(" + scX(i) + "," + scY(i) + ")" )
        .call( sticker );

    d3.select( "#sticker" ).append( "g" )                          ❼
        .attr( "transform", "translate(75,150)" )
        .call( sticker, "I'm fine!" )
        .selectAll( ".label" ).attr( "fill", "red" );              ❽
}
```

❶ A component is simply a function that takes a `Selection` instance as its first argument. The remaining parameters are arbitrary.

❷ Create and configure the rectangle as a child of the supplied selection. The rectangle is centered at the origin.

❸ Assign a CSS class to the rectangle: this will make it easier to identify it later for styling. The second parameter to `classed()` is important: `true` indicates that the class name should be assigned to the element, whereas `false` indicates that the class name should be removed.

❹ Create and configure the text element and assign it a class name as well.

❺ If a second argument has been supplied when the `sticker()` function was called, use it as the label. Otherwise, use the data bound to the current node. This makes it possible to use this component regardless of whether data is bound to the target selection or not.

❻ To use this component *with* bound data, create a `<g>` element for each data point as usual, and then invoke the `sticker()` function using `call()`. This will execute the `sticker()` function, while supplying the current selection as the first argument. In this case, the "current selection" contains the newly created `<g>` elements (with their bound data).

❼ To use this component *without* bound data, again append a `<g>` element as the container for the sticker, and invoke `sticker()` via `call()`, but this time you must supply a label explicitly. The

`call()` facility will simply forward any arguments past the first one when invoking the supplied function.

❽ The CSS class name makes it easy to select part of the component to change its appearance. Keep in mind, however, that `selectAll()` returns a *new* selection: any subsequent calls in the chain of function calls will *only* apply to the elements that matched the `".labels"` selector!

Working with Components

A few general comments about working with components:

- A component is usually created inside of a `<g>` element as the common parent to all parts of the component: for example, so that the entire component can be moved to its final location through an SVG transformation applied to the containing `<g>` element. This `<g>` element is usually *not* created by the component, but by the calling code. If this seems surprising, keep in mind that the component does not return the elements it creates, but adds them directly to the DOM tree. To do so, it needs to know *where* to add them, and the calling code must supply this information—hence the surrounding code creates the `<g>` element as the destination for the component to add its results.
- Although a component can be invoked using regular function call syntax, with the destination `Selection` as the first argument, it is more idiomatic to invoke it "synthetically" using the `call()` function. The `call()` function will automatically inject the destination `Selection` as the first argument and return the same `Selection` instance, thus enabling method chaining. If you need to gain access to the new DOM elements that were created by the component, use `selectAll()` with an appropriate selector on the `Selection` returned by `call()`. The typical call sequence therefore looks like this:

```
d3.select("svg").append("g").attr( "transform", ... )
    .call( component )
    .selectAll( "circle" ).attr( "fill", ... )...
```

  Also see the very end of Example 5-8 for an example.
- Although it is possible to pass in appearance options (like color and so on) as additional parameters to the component, it is usu-

ally more idiomatic to apply them later on using the usual mechanisms of the `Selection` API (as was done toward the end of Example 5-8). This has the advantage that the appearance of the component is not fixed by its author, but can be changed—as desired—by the user.

A Component to Save Keystrokes

Whereas the sticker component in the previous section can reasonably claim to be reusable, a component can also make sense as nothing more than an ad hoc labor-saving device to save keystrokes in a local scope. For example, imagine that you need to add `<text>` elements in several spots, all of which should use different text alignments and font sizes. Let's also assume a situation where it is not possible or practical to collect these presentational attributes in a stylesheet or on a common parent. An ad hoc component like the following can relieve you from having to type the attribute names and function calls for every `<text>` element explicitly:

```
function texter( sel, anchor, size ) {
    return sel.attr( "text-anchor", anchor )
        .attr( "font-size", size )
        .attr( "font-family", "sans-serif" );
}
```

And you use it simply as shorthand where appropriate:

```
d3.select( ... )
    .append( "text" ).call( texter, "middle", 14 ).text( ... );
```

Similar ad hoc components might be useful to set the fill and stroke colors of an object in one fell swoop, or to create a circle with its center coordinates and radius in a single call.

SVG Transformations as Components

SVG transformations are extremely useful—but, unfortunately, they are also a bit verbose, in particular when you need to build up the argument string dynamically. A labor (and space) saving device would be welcome! (This section is probably best skipped on first reading.)

It's easy enough to write a component for that purpose (in this section, I'll restrict myself to *translations* for simplicity; extensions to more general SVG transformations are straightforward):

```
function trsf( sel, dx, dy ) {
    return sel.attr( "transform", "translate("+dx+","+dy+")" );
}
```

And you would invoke it like any other component:

```
d3.select( ... ).append("g").call(trsf, 25, 50).append( ... )...
```

This is fine as far as it goes, but what if the offsets should depend on data bound to a selection? What we want to do is this (compare, for instance, Example 5-8):

```
d3.select( ... ).data( data ).enter().append( "g" )
    .call( trsf, (d,i)=>..., (d,i)=>... ).append( "circle" )...
```

But the simple `trsf()` component just introduced will not work in that situation. In order to handle *accessor functions* as arguments, the component must distinguish whether these arguments are functions or constants, and evaluate them in the former case. One problem that arises if we want to stay within the published `Selection` API (instead of poking under the hood) is that the API functions only allow for *one* accessor function to be evaluated simultaneously, but our `trsf()` takes two. Hence we need to evaluate the first accessor for all elements of the selection, store the results, then evaluate the second accessor, and finally combine the intermediate results to create the actual `transform` attribute value. The following code makes use of the `d3.local()` facility, intended for just this kind of application. It is essentially a hashmap, but expects a DOM `Node` as key. In this way, it is possible to store an intermediate value per `Node`:

```
function trsf( sel, dx, dy ) {
    var dxs = d3.local(), dys = d3.local();
    sel.each( function(d,i) {
        dxs.set( this,
                 typeof dx==="function" ? dx(d,i) : dx||0 ) });
    sel.each( function(d,i) {
        dys.set( this,
                 typeof dy==="function" ? dy(d,i) : dy||0 ) });

    return sel.attr( "transform", function() {
        return "translate(" +
            dxs.get(this) + "," + dys.get(this) + ")"} );
}
```

A simpler but less clean method is to store the intermediate result in the node itself by adding additional "bogus" properties to it:

```
sel.each( function(d,i) {
    this.bogus_dx = typeof dx==="function" ? dx(d,i):dx||0 } );
```

Either way, the `trsf()` component can now be called as intended. For example, the following code could be added to Example 5-8:

```
var vs = [ "This", "That" ];
d3.select( "#sticker" ).append( "g" )
    .selectAll( "g" ).data( vs ).enter().append( "g" )
    .call( trsf, (d,i) => 300 + scX(i), (d,i) => scY(i) )
    .call( sticker );
```

The <g> Element Is Your Friend

When reading the SVG spec, the purpose of the `<g>` element may appear obscure. What is *that* supposed to be good for? As it turns out, the `<g>` element has a number of practical uses when working with SVG and with D3 specifically. In particular:

- Grouping elements with the `<g>` element allows for more specific, discriminating selections without having to rely on additional information (like `id` or `class` attributes) or CSS selector magic. For example, if all `<circle>` elements that represent a specific data set are contained in a `<g>` element, then one can select those, and *only* those, `<circle>` elements as follows:

  ```
  var cs = d3.select( "g" ).selectAll( "circle" );
  ```

  Other `<circle>` elements, possibly representing other data sets or having some other function, will not be selected.
- The power and convenience of SVG transforms only becomes apparent when used together with the `<g>` element. Inside of a `<g>` element, you can construct a complicated component at the origin, worrying only about the *relative* placement of its constituents. Only when the entire component is finished, is it moved into its final position using a transform. Creating reusable components in SVG relies on this process.
- Grouping elements using the `<g>` element may reduce code duplication because it is possible to apply appearance options to the `<g>` element as the common parent, rather than having to touch each (visible) element individually. In a similar spirit, it can be convenient to register an event handler on a parent `<g>` element, rather than on each child element.

CHAPTER 6

Files, Fetches, Formats: Getting Data In and Out

Unless you define your data inside your script (as in Example 5-1, for example) or have your script generate it (as in Examples 4-6 and 4-7), you will have to get the data *into* your script somehow. This involves two separate steps: fetching the data from its location (which may be the local filesystem, a remote server, or another resource, such as a web service) and parsing it into a usable data structure. If you want to create text labels from data, you will need to do the opposite and format data for textual output. This chapter describes the facilities D3 offers to help with these tasks. Discussions of file formats are usually a drag—I'll try to make it brief.

Fetching a File

The JavaScript Fetch API is a modern replacement for the venerable `XMLHttpRequest` object—the technology that first enabled web pages to exchange data with servers asynchronously, thus giving rise to AJAX and the entire contemporary, "dynamic" web experience. D3 wraps this Fetch API, replicates some of its methods, and adds functionality that is convenient when working with web pages or tabular data. Some artifacts of the underlying API remain visible through D3; for this reason, it is frequently worth consulting the Fetch API reference as well.

Table 6-1 lists all functions provided by D3 that can retrieve ("fetch") a resource (such as a file) from a URL. Of course the

resource need not be a file—it can be anything, as long as it is describable by a URL (like a server that generates data on demand).

- All functions take a specification of the desired resource as a string containing a URL.
- All functions return a `Promise` object (see "JavaScript Promises" on page 115).
- All functions take an optional `RequestInit` object. The elements permitted in this object and their values are defined by the Fetch standard; they control various aspects of the remote communication, such as permission and caching issues. Some of these may be relevant even in relatively simple applications; we will discuss some of them toward the end of this section.
- The convenience functions that parse tabular data may also take a conversion function that will be applied to the data as it is read. (We will discuss conversion functions in the next section.)

Table 6-1. Methods for retrieving a resource (all methods return a Promise object)

Function	Description
`d3.text( url, init )`	Fetches the specified resource and treats it as UTF-8 decoded string. [a]
`d3.json( url, init )`	Fetches the specified resource and parses it as JSON into an object. [a]
`d3.dsv( delimiter, url, init, converter )`	Takes a delimiter (such as `","`) and a URL as required parameters. Fetches the specified resource, which must include a descriptive header line, and parses it as delimiter-separated values; the result will be an array of objects. An optional conversion function may be specified as the last parameter.
`d3.csv( url, init, converter )`	Fetches the specified resource, which must include a descriptive header line, and parses it as comma-separated values; the result will be an array of objects. An optional conversion function may be specified as the last parameter.
`d3.tsv( url, init, converter )`	Fetches the specified resource, which must include a descriptive header line, and parses it as tab-separated values; the result will be an array of objects. An optional conversion function may be specified as the last parameter.
`d3.html( url, init )`	Fetches the specified resource and parses it into an `HTMLDocument` element.
`d3.svg( url, init )`	Fetches the specified resource and parses it into an `SVGDocument` element.

Function	Description
`d3.xml( url, init )`	Fetches the specified resource and parses it into a `Document` element.
`d3.image( url, init )`	Fetches the specified resource and parses it into an `HTMLImageElement` element.
`d3.blob( url, init )`	Fetches the specified resource and treats it as a `Blob` object.[a]
`d3.buffer( url, init )`	Fetches the specified resource and treats it as an `ArrayBuffer` object.[a]

[a] This function replicates part of the Fetch API.

JavaScript Promises

A JavaScript `Promise` is an object that coordinates asynchronous function calls and their associated callbacks. The `then( onSuccess, onFailure )` member function takes two callbacks, one of which is invoked, depending on whether the original asynchronous call succeeded or failed. Both callbacks are expected to take a single argument when called (the *fulfillment value* or the *rejection reason*, the details of which are determined by the context in which the `Promise` object was created):

```
promise.then(function(value ) { /* ... handle success */ },
             function(reason) { /* ... handle failure */ });
```

Both callbacks are optional; no error results if either one is omitted.

Callbacks can be attached to a `Promise` even *after* the underlying asynchronous call has occurred; the `Promise` will make sure that the appropriate callback is invoked regardless. It is possible to invoke `then()` multiple times on the same `Promise` object to register multiple callbacks for a single asynchronous call.

The `then()` function returns a new `Promise` object, encapsulating the outcome of the callback that was provided. This makes it possible to chain asynchronous calls:

```
promise.then( handler1 ).then( handler2 );
```

The exact semantics of `Promise` objects can be surprisingly complex; you may need to check the appropriate reference documentation for more involved applications.[1]

1 For example, the MDN Promises Guide.

Examples

Let's consider a few examples. Assuming that you have a simple JSON file, like this one:

```
{ "val": 5, "txt": "Hello" }
```

then you can read it (and access its properties) like so:

```
d3.json("simple.json").then(res=>console.log(res.val,res.txt));
```

JSON parsers are picky; make sure your JSON is correctly formed! (In particular, JSON property keys *must be double-quoted strings*, in marked contrast to the syntax for JavaScript object initializers.)

It is also easy to fetch a bitmap image and attach it to the document (or page):

```
d3.image( "image.jpg" ).then( function(res) {
    d3.select( "#figure" ).append( () => res ) } );
```

This code assumes that the page contains a placeholder with an appropriate `id` attribute (for example, `<div id="figure">...</div>`). Note the argument to the `append()` function: it is a function, taking no arguments, and returning the result of the fetch! The reason for this roundabout route is that `append()` can handle either a string or a *function returning a node*, but not a node by itself. The result of the fetch is a node, hence it is necessary to "wrap it into a function" like this.

Finally, let's assume that you have an SVG file, for example, containing the definition of a symbol that you would like to reuse:

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg">
  <defs>
    <g id="heart">
      <path d="M0 -3 A3 6.6 -35 1 0 0 6 A3 6.6 35 1 0 0 -3Z" />
    </g>
  </defs>
</svg>
```

You can insert the `<defs>` section into the current document and use the defined symbol like this:

```
d3.svg( "heart.svg" ).then( function(res) {
    d3.select("svg").insert( ()=>res.firstChild,":first-child");
    d3.select("svg").append( "use" ).attr( "href", "#heart" )
        .attr( "transform", "translate(100,100) scale(2)" );
} );
```

Again, note how the result of the fetch is "wrapped into a function." In this snippet, `res` is a DOM `SVGDocument` instance (not a D3 data type), hence you must use the native DOM API to extract the `<defs>` element (the first and only child of this SVG document).

The external SVG file in this example contains declarations for XML namespaces. We have not worried about XML namespaces so far (because D3 for the most part takes care of them for us). Here, they are required, because otherwise the SVG parser will not work correctly.

Controlling Fetches with the RequestInit Object

Most of the time, the functions in Table 6-1 just work without any further tweaking. But the simplicity and convenience of the API hides underlying complexity (and sometimes obstructs proper diagnosis if something goes wrong). The means to control aspects of the remote communication is the `RequestInit` object that all functions in Table 6-1 accept as an optional parameter.[2]

Caching. Browsers may *cache* resources fetched from a remote location, with the consequence that changes to the remote resource will not become visible in the browser. In particular during development, this can become a major nuisance! A simple way to prevent all browser caching of a remote resource is to set the `cache` property to `no-store`:

```
d3.svg( "heart.svg", { cache: "no-store" } ).then( ... );
```

While good practice during development, preventing browser-side caching is wasteful when used in production. The details of resource caching can be complex; you may want to check the appropriate reference.[3]

Third-party resources and CORS. When attempting to load resources using the Fetch API from third-party websites, you may occasionally encounter strange failures and permission issues. The browser refuses to complete the request when made from within the JavaScript runtime—even though the resource may be readily available

2 See the MDN Fetch Reference for a list of all legal parameters.

3 For example: MDN Request Cache Reference and Cache Control for Fetch.

using a command-line tool or even by pointing the browser itself to the URL. The cause is likely to be the browser's *same-origin policy* and the *cross-origin resource sharing* (CORS) mechanism, which limit JavaScript access to third-party resources.[4]

The CORS protocol is strange in the way it splits responsibility between browser and server (under certain conditions, the server will send the requested resource to the browser, but the browser will refuse to make it accessible to its own JavaScript runtime!), and browsers differ in the CORS policy they implement. The `mode` property of the `RequestInit` object contains additional information.[5]

CORS relies on an interplay between browser and server. In particular, the server *must* be configured to send the appropriate header information. If it isn't, there is nothing you can do. You will have to download the required resource separately and serve it from your own server, or access the resource through a proxy server.

Writing a file. Occasionally, it is necessary to *write* a file—for example, to save a graph as an SVG file. That is not easy, though, because the local filesystem is not accessible from within JavaScript. It is, however, possible to upload a file (or any data, for that matter) to a server. Example 6-1 shows how this can be accomplished. (A very similar function was used to capture the graphs for this book.)

Example 6-1. A JavaScript function to upload all SVG figures in a page to a server

```
function upload() {
    var out = new FormData();                                          ❶

    d3.selectAll( "svg" ).each( function() {                           ❷
        var id = d3.select( this ).attr( "id" );                       ❸
        if( id ) {
            out.set( "filename", id );
            out.set( "data", this.outerHTML );                         ❹

            d3.text( "http://localhost:8080/upload",                   ❺
                     { method: "POST", body: out } )
                .then( function(r) { console.log("Succ:", id) },       ❻
                       function(r) { console.log("FAIL:", id) } );
```

4 The best short introduction I am aware of is Spring's "Understanding CORS".

5 See the MDN Request.mode Reference.

```
        } } );
}
```

❶ Creates a `FormData` object as the container for the data to upload.

❷ Invoke the following anonymous function for every SVG element in the page.

❸ Grab the value of the element's `id` attribute—it will later become the filename. If no `id` attribute exists, skip the upload.

❹ The `outerHTML` of a page element is the element's content together with the tags that make up the element itself. In this case, this constitutes the `<svg>` tag itself and all of its children. The `innerHTML` is just the contents without the enclosing tags.)

❺ Upload the `FormData` element, using the HTTP POST method, to a suitable server. Notice how the `RequestInit` object is used to hold the payload and method specification.

❻ Print a confirmation message to the browser console.

Of course, all this assumes that a server is listening at the specified URL that can handle the uploaded data and do something useful with it (for example, save it to disk).

Parsing and Writing Tabular Data

D3 provides functions to parse (and write) strings containing delimiter-separated data (see Tables 6-2 and 6-3). They are primarily intended for files of the `"text/csv"` MIME type (as laid down in RFC 4180), commonly used by spreadsheet programs. Some notes on parsing more general file formats follow in the next subsection.

The library supports two different styles to represent a data set:

- The functions `parse()` and `format()` treat each record as an *object*.
  - — Names of the object properties are taken from the file's first (or header) line, which *must* be present.
  - — The data set is returned as an *array of objects*.

- The functions `parseRows()` and `formatRows()` treat each record as an *array* (of columns).
  - — The entire file, including its first line, is assumed to contain data.
  - — The data set is returned as an *array of arrays*.

Use `parseRows()` if the input files does not contain a header line with metadata. The `Array` object returned by `parse()` provides an additional member variable `columns`, which contains a list of the original column names in the order of the input file.

Table 6-2. Methods to parse and format delimiter-separated data (p is a parser-formatter instance)

Function	Description
`d3.dsvFormat( delim )`	Returns a parser-formatter instance. The mandatory argument specifies the delimiter to use; it should be a single character.
`p.parse( string, converter )`	Parses the input string and returns an array of objects. The first record of the input is expected to contain column names that will be used as property keys within the created objects. If supplied, the optional conversion function will be invoked on each record after it has been split into fields; it should return an object.
`p.parseRows( string, converter )`	Parses the input string and returns an array of arrays. If supplied, the optional conversion function will be invoked on each record after it has been split into fields; it should return an array.
`p.format( data, columns )`	Takes an array of objects and returns a delimiter-separated string. The ordered array of property names to be included in the output is optional; if it is omitted, all properties are included (in arbitrary order).
`p.formatRows( data )`	Takes an array of arrays and returns a delimiter-separated string.

Field Value Conversions

Whether you use `parse()` or `parseRows()`, the field values are *strings*; values are not automatically converted to numbers. This does cause problems occasionally when other parts of the program

require numerical input; for this reason it is good practice to always convert input to numbers explicitly.

You can supply an optional second argument to either `parse()` or `parseRows()` to perform such conversions or other desired cleanups. This function will be invoked for each input line *after* the line has been split into fields and turned into an object or array. It is therefore not intended to parse each row into fields but to apply conversions to the individual field values.

The conversion function will be passed three arguments:

- The field values in the current row (as object or array)
- The line number of the current row (starting at zero, not counting the header line)
- An array of column names (for `parse()` only)

The conversion function should return an object or array representing the current line (or `null` or `undefined` to skip the current line).

For data files containing only strings, numbers, and dates, you can use the built-in conversion function `d3.autoType()`, which converts entries that "look like" numbers or dates to the appropriate type. For more complicated situations, you have to write your own conversion function. Consider the following CSV file:

```
Year,Month,Name,Weight (kg)
2005,1,Peter,86.3
2007,7,Paul,72.5
```

The following code will turn it into an array of objects with lowercase member names and using appropriate data types:

```
d3.text( "csv.csv" ).then( function(res) {                    ❶
    var data = d3.csvParse( res, (d,i,cs) => {
        return {
            date: new Date( d.Year, d.Month-1 ),              ❷
            name: d.Name,                                     ❸
            weight: +d["Weight (kg)"]                         ❹
        };
    } );
    console.log( data );
} );
```

❶ Load as plain text, parse, and convert in callback.

❷ Combine two columns into `Date` type. (JavaScript's month index is zero-based.)

❸ Convert property name to lowercase.

❹ Eliminate invalid property name, convert value to number.

Parsing Input Containing Arbitrary Delimiters

The preceding code snippet uses the convenience function `d3.csvParse()`, which assumes a comma-separated file. Because comma- and tab-separated files are so common, D3 provides a set of shorthands for them (see Table 6-3). For arbitrary delimiters you must first instantiate a parser-formatter instance using `d3.dsvFormat(delim)`, then invoke `parse()`, `format()` (or `parseRows()`, and `formatRows()`) on this instance, while supplying the input string to parse (or the array to format into a string). The explicit delimiter argument is mandatory; it should be a single character. On the other hand, you can parse a resource when fetching it in one fell swoop using `d3.csv()`. Here are three ways to achieve the same effect (`data` will always be an array of objects, use `d3.csvParseRows()` or `parser.parseRows()` otherwise):

```
d3.csv( "csv.csv" ).then( function(res) {
    var data = res;
} );

d3.text( "csv.csv" ).then( function(res) {
    var data = d3.csvParse( res );
} );

d3.text( "csv.csv" ).then( function(res) {
    var parser = d3.dsvFormat( "," );
    var data = parser.parse( res );
} );
```

Table 6-3. Shorthands for comma- and tab-separated files. The functions are equivalent to those in Table 6-2.

Comma-delimited	Tab-delimited
`d3.csvParse( string, converter )`	`d3.tsvParse( string, converter )`
`d3.csvParseRows( string, converter )`	`d3.tsvParseRows( string, converter )`
`d3.csvFormat( data, columns )`	`d3.tsvFormat( data, columns )`

Comma-delimited	Tab-delimited
`d3.csvFormatRows( data )`	`d3.tsvFormatRows( data )`

Generating Tabular Output

The functions `format()` and `formatRows()` implement the inverse operation: serializing a data structure into a string. The input must be an array of objects (for `format()`) or an array of arrays (for `formatRows()`). The `format()` function takes as additional, optional arguments a list of object property names to be included in the output; if this is omitted, a union of *all* property names found across the entire input is used. Fields in the created string are separated using the specified delimiter, records are separated using newlines (`\n`), and fields are quoted as necessary.

Using Regular Expressions to Parse Whitespace-Separated Data

Even if data files do not conform to the formats required by the methods just described, the infrastructure provided by them can still be useful. For example, consider the case of a data file with columns separated by whitespace: any combination of tabs and spaces. This is a situation calling for regular expressions, and the following snippet shows how to use them in conjunction with the overall framework:

```
d3.text( "txt.txt" ).then( function(res) {
    var parser = d3.dsvFormat( "" );                          ❶
    var rows = parser.parseRows( res, function(d, i, cs) {    ❷
        return d[0].split( /\s+/g ).map( x => +x );           ❸ ❹
    } );
    console.log( rows );
} );
```

❶ Create a parser-formatter instance, selecting a delimiter character that you are certain does *not* occur in the input file. (The empty string seems to work, but the ASCII NUL character `"\0"`, or any other character that you are sure won't occur, present alternatives.)

❷ Because the delimiter does not occur in the input, no separation into columns is performed (but the input is correctly broken into lines or records)…

❸ … so that the array `d` of "column" values has only a single element. The `split()` function is invoked on it with a regular expression that matches any combination of whitespace characters.

❹ Finally, all resulting field values are converted to numbers.

Formatting Numbers

JavaScript does not have built-in routines to convert arbitrary scalars into formatted strings, comparable to the family of `printf()` functions familiar from many other programming languages. D3 provides a remedy: a sophisticated formatting facility modeled after similar functionality in Python 3. This section describes how to convert numbers into human-readable strings; routines for formatting timestamps will be discussed in Chapter 10.

In full generality, the workflow to format a value involves three steps:

1. Obtain a *locale* object (or use the current "default locale").
2. Use the locale object to obtain a formatter instance for the intended output format.
3. Apply the formatter to a numeric value to obtain the value's formatted, human-readable string representation.

Of course you can bundle all three steps into a single statement without assigning the intermediate objects to individual variables. For example, using the default locale, you might simply say:

```
var str = d3.format( ".2f" )( Math.PI );
```

Locales

There are two functions to obtain a locale object (see Table 6-4). Both require a locale definition as input. A locale definition specifies details such as the currency symbol, prevailing number formats, names of months and weekdays, and so on (see the D3 Reference Documentation for details). Locale definitions in a format suitable for D3 are available from *https://unpkg.com*, a repository for content packaged using the JavaScript package manager *npm*. The following snippet shows how to fetch and use a new locale definition. It prints

the string `3,1316` to the console, following the German convention of using a comma (not a point) as a decimal indicator:

```
d3.json( "https://unpkg.com/d3-format/locale/de-DE.json" ).then(
    function( res ) {
        var loc = d3.formatLocale( res );
        console.log( loc.format( ".4f" )( Math.PI ) );
    },
    function( err ) {
        throw err;
    }
);
```

Table 6-4. Factory methods for creating locale objects

Function	Description
`d3.formatLocale( def )`	Takes a locale definition and returns a locale object.
`d3.formatDefaultLocale( def )`	Takes a locale definition and returns a locale object, but also sets the default locale to the supplied definition.

Formatters

A locale object serves as a factory for formatters. Once you have chosen a locale (either a specific one or the default locale), you use it to obtain a formatter instance by providing the intended output format as a string. The formatter is a function object, which takes a single number and returns a formatted string. (It is not possible to create a string containing multiple formatted values simultaneously, in contrast to `printf()`.) All the factory functions that produce a formatter instance are listed in Table 6-5.

Table 6-5. Factory methods to create number formatter instances (loc is a locale instance)

Function	Description
`d3.format( fmt )`	Returns a formatter instance for the current default locale using the format specification in the string `fmt`.
`loc.format( fmt )`	Returns a formatter instance for the receiver locale using the format specification in the string `fmt`.
`d3.formatPrefix( fmt, scale )`	Returns a formatter instance for the current default locale. Quantities will be expressed as multiples of the supplied "scale" argument, which should be an engineering power $10^{\pm3}$, $10^{\pm6}$, $10^{\pm9}$, … The scale is represented through SI prefixes in the output string.

Function	Description
`loc.formatPrefix( fmt, scale )`	Same as `d3.formatPrefix()`, but for the receiver locale.

In addition to the usual formatter, there is also a special formatter that expresses all quantities in a *fixed* engineering unit. For instance, it will express all quantities in "kilos," "millis," or whichever scale you choose. (See the D3 Reference Documentation or the Wikipedia Metric Prefixes page for all available prefixes.) This behavior is *not* indicated through the supplied output format specifier; instead, you must use a special factory function to obtain a formatter with this behavior. For example:

```
d3.formatPrefix( ".4f", 10e-3 )(Math.PI) === "3141.5927m"
d3.formatPrefix( ".4f", 10e3 )(Math.PI)  === "0.0031k"
```

A formatter object can be reused to format several values. This snippet turns an array of numbers into an array of formatted strings:

```
var f = d3.format( ".3f" );
[ Math.E, Math.PI ].map( f );
```

Format or Conversion Specifiers

The format specifier string can consist of up to nine different fields:

```
[[fill]align][sign][symbol][zero][width][,][.precision][type]
```

See Table 6-6 for the permissible values and their effects. Be aware that you don't use `%` in the format specification (in contrast to what's customary for `printf()`).

Table 6-6. Conversion specifiers for numbers

Field	Specifier	Description
fill	any character	Used for padding when aligning values
align	>	Right-align value within available space (default if missing)
	<	Left-align value within available space
	^	Center value within available space
	=	Right-align value, left-align sign and symbol

Field	Specifier	Description
sign	-	Minus sign for negative values, nothing otherwise (default if missing)
	+	Minus sign for negative values, + otherwise
	(	Parentheses for negative values, nothing otherwise
	blank	Minus sign for negative values, a blank space otherwise
symbol	$	Insert currency symbol per the locale definition
	#	Prefix binary, octal, or hexadecimal numbers by 0b, 0o, or 0x, respectively
zero	0	When a 0 is present, it sets the > and the = flags, overriding other settings.
width	number	Defines the minimum field width. If the value does not exhaust the width, the value will be padded. If not present, the width will be determined by the content.
,	,	When a comma is present, a grouping separator will be used.
precision	number	Number of digits to the right of the decimal (for f and %); number of significant digits (for e, g, r, s, p, and missing type). Defaults to 6, but equals 12 when the type indicator is missing. Ignored for integer formats (b, o, d, x, and X).
type	e	Exponential notation
	f	Floating point notation
	g	Decimal notation if the resulting string has fewer than *precision* significant digits, otherwise exponential notation
	r	Decimal notation, rounded to significant digits
	s	Decimal notation with an SI prefix, rounded to significant digits
	%	Multiply by 100, then decimal notation with a percent sign
	p	Multiply by 100, round to significant digits, then decimal notation with a percent sign
	b	Binary notation, rounded to integer
	o	Octal notation, rounded to integer
	d	Decimal notation, rounded to integer
	x	Hexadecimal notation, using lowercase letters, rounded to integer
	X	Hexadecimal notation, using uppercase letters, rounded to integer
	c	Convert integer to the corresponding Unicode character
	n	Shorthand for ,g (with grouping separator)
	missing	Like g, but omits trailing zeros

D3 includes some functions that can help construct format specifiers. First among these is the function `d3.formatSpecifier(fmt)` that parses the specifier `fmt` into its constituent fields. You can now

inspect individual values, even change some of the fields (possibly programmatically), and then glue the values together again to obtain a new specifier based upon the old one. Other functions (`d3.precisionFixed()`, `d3.precisionPrefix()`, and `d3.precisionRound()`) help you find the proper value for the `precision` field in a specifier, given the finest resolution (that is, the smallest difference between consecutive values) that you still would like to be visible. These functions are used internally (for instance, to determine the appropriate format for axis tick marks), but they are available for general uses as well.

CHAPTER 7

Values to Visuals: Interpolations, Scales, and Axes

Any form of visual data representation involves a *mapping* between the original data values and their visual depiction. Information can be encoded in different forms, depending on circumstances and purpose: the *location* of a symbol within a graph is probably the most common form of visual representation, but the *size* of a symbol or its *color* are also frequently used.

In D3, instances of the scale object can be used to map data values from the input domain to the output range. The scale abstraction is convenient as it provides a unified API to map between pretty much any possible combination of input and output values: numbers to colors, strings to sizes, objects to positions.

Axes with regularly spaced and labeled markers (*tick marks* and *labels*) are an important element of well-designed visualizations, because they allow the observer to associate quantitative information with the graphical presentation—or, more generally, to "map back," from the graphical elements to the original problem domain. Once a scale object has been set up and configured for a specific visualization problem, it can also be used to create the required tick marks or grid lines, and D3 provides functions to do so.

To produce a proper output value for *any* legal input value, scale objects need to be able to *interpolate* between values. D3 provides some general facilities to interpolate not only between numbers but also between colors, strings, and even more general objects. We will

briefly describe them first, followed by a discussion of the various scale objects. Finally, you will see how to use scale objects to create axes and tick marks.

Interpolation

The need to interpolate smoothly between two values arises frequently, for example, to create smooth animations, but also to design continuous color gradients. As a major convenience to users, D3 interpolates seamlessly not only between numbers but also between dates, colors, strings with embedded numbers, and even complex data types containing any of these elements. This behavior can seem quite magical but occasionally leads to unexpected results when the limitations of the interpolation model are not appreciated. Hence, a brief look under the covers is in order.

How Universal Interpolation Works

The first thing to understand is that D3 bases its decisions primarily on the *end point* of the interpolation interval; the starting point is coerced to conform to the end point. The value returned from the interpolation will also be of the same *type* as the end point. (Note that for items 2 and 4 below, for performance reasons, the interpolators for dates, arrays, and objects return a reference to the same object on each invocation and only change the object's internal state.)

1. Constants, null values, and booleans are not interpolated; the value at the end point prevails.
2. Numbers, colors (either as CSS strings or as D3 `color` objects), and JavaScript `Date` objects are interpolated individually. Numbers and dates are interpolated linearly, and color interpolation depends on the color space.
3. If the end point is a string (other than a color specification), it is searched for embedded numbers. If a number is found in the equivalent position (in a regular-expression sense) in the starting point, then the numbers are interpolated. The remainder of the string is taken from the end point.
4. If the end point is an array or object, it is searched recursively for embedded values. Interpolation takes place if a value is

found in the equivalent position in the starting point, otherwise the end point prevails.

The predominance of the end point implies that when interpolating between, for example, the strings `"10px serif"` and `"22px Helvetica"`, only the *numbers* will be interpolated, but the rest of the string will come from the end point. Possibly contrary to expectations, the font does not switch at the midpoint of the interval; instead Helvetica is used throughout.

Finally, keep in mind that although D3 interpolation is fairly smart, it is not semantic. Nested data structures are searched for embedded numbers (or other types), but you shouldn't try to interpolate between two CSS class *names* and expect all the numeric values in the corresponding CSS *styles* to be interpolated!

Implementation Notes and Custom Interpolators

A D3 interpolator is a function that takes a number between 0 and 1 as its only argument, and returns an appropriate interpolated value. It should return the start and end points of the interpolation interval when evaluated at 0 or 1, respectively.

D3 provides individual factory functions for all types that can be interpolated, or you can use the generic factory function:

```
d3.interpolate( start, end )
```

which returns an appropriate interpolator instance based on the type of the end point of the interpolation interval, as explained earlier.

The standard interpolator for numbers uses a linear interpolation: $v = (1 - t)\,a + t\,b = a + (b - a)\,t$ with $0 \le t \le 1$, where a and b are the start and end points of the interpolation interval, respectively. Of course you can write your own interpolators that use other mappings. (Compare the discussion of custom interpolators in the context of D3 transitions in Chapter 4 for examples.)

D3 provides two convenience methods to create custom interpolators based on splines: `d3.interpolateBasis()` and `d3.interpolateBasisClosed()`. Each takes an array of values that *must be numbers* and treats them as nodes that are (horizontally) equally spaced. The variant `d3.interpolateRgbBasis()` is intended for color-space interpolation; see Examples 8-1 and 8-4.

Scales

Scale objects map one set of values into another—primarily to map data values into pixel coordinates (or pixel colors, for that matter), but of course they can also be used for *any* sort of mapping task.

D3 provides a variety of predefined scales that fall into three broad groups, depending on the nature of the source and destination values:

Continuous scales
: Map a continuous numerical region to another.

Binning scales
: Map a continuous numerical region to a set of discrete bins.

Discrete (or ordinal) scales
: Map a set of discrete values to another set of discrete values.

The originating space is called the source or input *domain*, and the final space is called the destination or output *range*. This naming convention must be committed to memory:

> Scale objects map the input *domain* to the output *range*.

Sometimes, instantiating scale objects appears unnecessary. It seems easy enough to index directly into an array of shapes or colors, or to perform some linear transformation on the fly. But scale objects offer some advantages that such ad hoc solutions don't:

- Scale objects bundle information in one place. If the input domain or output range changes, only the scale object needs to be updated.
- Scale objects provide additional services. Ordinal scales, for example, are more robust than indexing directly into an array, because they will cycle through the array, reusing elements if the number of input values exceeds the length of the array.
- Although their definition is more verbose, the actual invocation of scale objects is extremely lightweight, thus simplifying nested code in particular.

Scale objects are a convenient abstraction—use them!

Continuous Scales: Numbers to Numbers

All "continuous" scales (those that map a continuous numeric region to another) support a common set of operations (see Table 7-1). You obtain continuous scale instances from one of the following factory functions. (In the formulas, sgn(x) is the sign function, and $|x|$ indicates the absolute value of x.)

`d3.scaleLinear()`

- Returns a continuous scale object with input domain `[0, 1]` and output range `[0, 1]`, and clamping disabled.
- The domain will be mapped to the range using $y = a\,x + b$ for appropriately calculated values of a and b.

`d3.scalePow()`

- Returns a continuous scale object with input domain `[0, 1]` and output range `[0, 1]`, exponent $k = 1$, and clamping disabled.
- The domain will be mapped to the range using $y = a\ \mathrm{sgn}(x)\ |x|^k + b$ and appropriately calculated values of a and b.
- The scale object exposes an additional method `exponent(k)` to set or read the exponent; the exponent may be negative.

`d3.scaleSqrt()`

A convenience function equivalent to `d3.scalePow().exponent(0.5)`. (This comes in handy for finding the radius of a symbol when encoding a value through the symbol's area.)

`d3.scaleLog()`

- Returns a continuous scale object with input domain `[1, 10]` and output range `[0, 1]`, base $b = 10$, and clamping disabled.
- The domain will be mapped to the range using $y = a \log(|x|) + c$, for appropriate values of a and c, with the logarithm being taken with respect to the current base.
- The scale object exposes an additional method `base(b)` to set or read the base of the logarithm.
- For logarithmic scales, the input domain interval must be *strictly* positive or negative; it must not include or cross 0. A

strictly negative input domain is legal; the absolute value will be applied to all arguments before logarithms are taken.

`d3.scaleTime()`

- Returns a continuous, linear scale object for timestamps. The input domain values are interpreted as `Date` objects; `invert()` returns a `Date`.
- Tick marks are generated at intervals that are meaningful for timestamps (such as 1, 5, 15, and 30 minutes).
- The `ticks()` method can accept either a count of desired tick marks or a time interval (see Table 10-5) to indicate the spacing between tick marks.

`d3.scaleUtc()`

Like `d3.scaleTime()`, except that timestamps are interpreted as UTC.

Table 7-1. Methods of continuous scale objects (sc is a continuous scale instance)

Function	Description
`sc( x )`	Given a (numeric) value `x` from the input domain, returns the corresponding value from the output range. If clamping is active, the returned value is restricted to the defined output range.
`sc.invert( y )`	Given a value `y` from the output range, returns the corresponding value from the input domain. If clamping is active, the returned value is restricted to the defined input domain.
`sc.domain( [ invalues ] )`	Sets the input domain to the supplied array. If the array elements are not numbers, they will be coerced to numbers. The array must contain at least two elements. If it contains more, then they are paired with the corresponding range elements, and interpolation takes place for each subinterval separately. If the array contains more than two elements, they must be sorted (in either ascending or descending order).
`sc.range( [ outvalues ] )`	Sets the output range to the supplied array; the array elements need not be numeric. The array must contain at least two elements. If it contains more elements, then they are paired up with the corresponding domain elements, and interpolation takes place for each subinterval separately.

Function	Description
`sc.clamp( boolean )`	Activates clamping if the supplied value evaluates to `true`, disables it if the value is `false`. If clamping is active, then return values are restricted to the specified intervals. (For most scale objects, clamping is disabled by default.)
`sc.ticks( count )`	Returns an array of equally spaced, numerically "round" values from the input domain. The optional `count` argument is interpreted as a *hint* for the number of returned elements; it defaults to 10.
`sc.tickFormat( count, format )`	Returns a formatter object (see Chapter 6) to render the values returned by the `ticks()` function to the appropriate precision as human-readable strings. The `count` value should be the same as the one supplied to the `ticks` function. A custom format can be supplied to override the default.
`sc.nice( count )`	Extends the input domain so that its endpoints fall onto numerically "round" values. The optional `count` argument is interpreted as a *hint* for the number of returned elements; it defaults to 10.
`sc.copy()`	Returns a clone of the current scale object.

- Some of the functions in Table 7-1 can act as getters and setters. They return the current value when invoked without argument as getters. When invoked with an argument as setters, they return the current scale object to facilitate method chaining.
- Scale objects support *clamping*. Without clamping, the value may be extrapolated to values outside the defined input domain and output range intervals. When clamping is active, then calculated values that would fall outside the interval are replaced with the nearest interval boundary.
- If domain and range have more than two values, then the subintervals will be paired up and interpolation takes place in each subinterval. If domain and range have an unequal number of elements, the shorter one prevails and surplus elements are ignored. The implementation requires that values in the input domain must be *sorted*, either in ascending or descending order.

Examples 7-2 and 7-3 at the end of this chapter demonstrate many aspects of continuous scales.

Sequential and diverging scales

Finally, there are also the `d3.scaleSequential(interpolator)` and `d3.scaleDiverging(interpolator)` factory functions. The scale objects they create are not defined by a domain and a range, but by a domain and an interpolator instance. Their API is more restricted than that of a regular scale object (for example, it does not include an `invert()` function; check the D3 Reference Documentation for more detail).

Sequential and diverging scales are really intended for color scales, where they are used in conjunction with color-space interpolators (see Example 8-1). They are not meant as framework for arbitrary mappings between numeric ranges. If this is what you want (for example, to create probability plots or similar graphs) then you need to write your own mapping functions without making use of the D3 scale object infrastructure.

Binning Scales: Numbers to Bins

D3 provides three different types of scale objects that map a continuous interval of numbers into a discrete set of "bins." When invoked, these scales find the bin for the supplied argument and return the value or object associated with that bin. The returned value itself can be arbitrary; it need not be a number. For instance, in Examples 4-7 and 7-3, binning scales are used to map numeric values to sets of discrete colors. The number of bins is determined by the number of available return values: one bin is created per return value.

The scale objects differ in the way the bins are defined (see Figure 7-1):

- The `d3.scaleQuantize()` factory returns a scale object that breaks the input domain into *equally sized bins*.
- The `d3.scaleThreshold()` factory returns a scale object that requires *a user-supplied set of threshold values* to divide the input domain into bins.
- The `d3.scaleQuantile()` factory returns a scale object that ingests a data set and *calculates the bins based on quantiles* of this data set.

The scale objects created by these three factories interpret the arguments of the `domain()` function in different ways: `d3.scaleQuantize()` as extent of the input domain, `d3.scaleThreshold()` as threshold values that separate bins, and `d3.scaleQuantile()` as records in the data set.

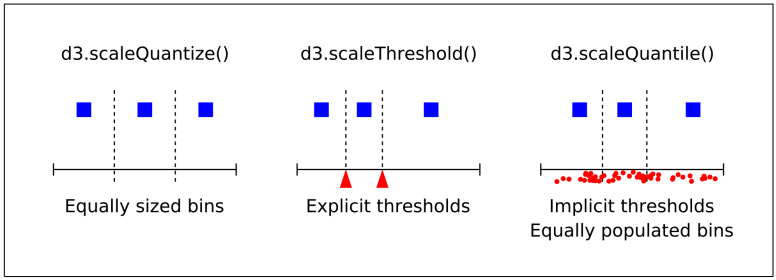


Figure 7-1. Binning scales differ in the way they divide the input domain into bins.

Binning scales provide a smaller API than the scales that map to a continuous range (see Table 7-2). In particular, binning scales do not support clamping: values outside the domain are mapped to either the first or last bin, respectively. You obtain a binning scale object from one of the following factory functions:

`d3.scaleQuantize()`

- Returns a binning scale that divides a contiguous, numeric input domain into a set of equally sized bins.
- The input domain must be specified as a pair of numeric values. It defaults to `[0, 1]`.
- The output range elements are defined using the `range()` method. The default is `[0, 1]`.
- The returned scale object provides `nice()`, `tick()`, and `tickFormat()` functions that are equivalent to the ones on continuous scales (see Table 7-1).

`d3.scaleThreshold()`

- Returns a binning scale based on explicit thresholds in the input domain. The thresholds need not be numbers, but they must be sortable.

- The thresholds are defined using the `domain()` method. The argument to `domain()` must be an array of values that *must be sorted in ascending order*.
- The output range elements are defined using the `range()` method. The number of range elements must be *one greater* than the number of threshold elements.
- The default domain is `[0.5]`, the default range is `[0, 1]` (so that the scale maps inputs less than 0.5 to 0, and inputs greater or equal than 0.5 to 1).

`d3.scaleQuantile()`

- Returns a binning scale that divides a contiguous, numeric input domain into a set of bins based on the quantiles of a supplied data set.
- The data set is supplied as an array of numeric values using the `domain()` method. The input array need not be sorted; `NaN`, `undefined`, and `null` elements are ignored. The scale object retains a copy of the entire data set.
- The output range elements must be defined using the `range()` method. The number of range elements determines the quantiles that are computed. (If you provide an array of four range elements, then the calculated bins will be quartiles, and so on.)
- Both domain and range are empty by default; they must be explicitly defined.
- The scale object exposes a `quantiles()` method that returns an array of computed threshold values. Quantiles are calculated according to the "R-7" method.[1]

Table 7-2. Methods of binning scale objects (sc is a binning scale instance). Scale objects returned by d3.scaleQuantize also support methods to configure tick marks and labels.

Function	Description
`sc( x )`	Given a (numeric) value `x` from the domain, returns the corresponding value from the range. If `x` is outside the domain, returns either the first or last range value.

1 See *https://en.wikipedia.org/wiki/Quantile*.

Function	Description
`sc.invertExtent( y )`	Given a value `y` from the range, returns a two-element array containing the boundaries of the corresponding bin. If `y` is not an element of the range, the bin boundaries are `NaN` or `undefined` (depending on the type of receiver instance).
`sc.domain( [ invalues ] )`	The argument must be an array of numeric values. The scale object will create bin boundaries based on the supplied values; the details depend on the type of the receiver. Scale objects created by `d3.scaleQuantize()` interpret them as the extent of the input domain, those created by `d3.scaleThreshold()` as threshold values that separate bins, and those created by `d3.scaleQuantile()` as records in the data set.
`sc.range( [ outvalues ] )`	Sets the range to the supplied array. The array must not be empty, and the array elements may be of any type. One bin will be created for each element in the array.
`sc.copy()`	Returns a clone of the current scale object.

Discrete or Ordinal Scales: Keys to Sequence

The scales generated by `d3.scaleOrdinal()` map discrete values to discrete values (see Table 7-3). They differ from ordinary hashmaps in that you don't need to establish every key/value relationship individually. Instead, you register an array of domain values and an array of range values, and the scale object automatically matches them up, in order. If there are fewer output range values than there are elements in the input domain, then range elements are reused cyclically. In fact, you don't even need to specify the input domain values ahead of time: by default, the scale instance associates each new domain value with the next unused range element. The relationship is established the first time the domain value is encountered and remembered when it occurs again. The values in the input domain can be arbitrary and need not be numeric. Internally, they are coerced to strings and the lookup occurs based on their string representation. Ordinal scale objects are obtained from the following factory function:

`d3.scaleOrdinal( [ outvalues ] )`
: Returns a scale object with empty input domain and the specified output range. If no range is supplied as argument, the scale object will always return `undefined` until a range has been defined.

Mapping categorical values to discrete colors or shapes are typical applications for discrete scale objects (for applications, see Examples 5-1 and 5-6). There are also two factory functions that create specialized scale objects (`d3.scalePoint()` and `d3.scaleBand()`) that offer additional convenience functionality for creating certain types of scatter plots or bar charts and histograms. (See Example 10-1 for an application of `d3.scaleBand()`.)

Table 7-3. Methods of ordinal scale objects (sc is an ordinal scale instance)

Function	Description
`sc( x )`	Given a value `x` from the input domain, returns the corresponding element from the output range. If `x` is not found in the domain, then it is added to the domain and associated with the next range element (this is the default). Alternatively, if an explicit value has been specified for unknown inputs, then this value is returned.
`sc.domain( [ invalues ] )`	Sets the input domain to the supplied array. Elements of the domain can be of any type; the scale object will identify them by their string representation. When the scale object is invoked for a value that is not contained in this array, the scale object will return the `unknown` value. If the `unknown` value has been set to the special value `d3.scaleImplicit`, then unrecognized input values are added to the domain when first encountered and associated with the next range element.
`sc.range( [ outvalues ] )`	Sets the output range to the supplied array; the array elements need not be numeric. If there are fewer range elements than domain values, the elements in the range will be reused cyclically.
`sc.unknown( value )`	Sets the output value for unknown inputs. If the argument is the special value `d3.scaleImplicit`, then unknown inputs are added to the domain and associated with the next range element—this is the default. (Note that `d3.scaleImplicit` is a special *value*, not a factory function.)
`sc.copy()`	Returns a clone of the current scale object.

Axes

Axes are important as a way to infer quantitative information from the location of graph elements. In D3, scale objects are used to map between domain values in the data set and pixel coordinates, and hence it is only natural that the D3 axes components are wrappers around scale objects.

The Constituents of an Axis

Coordinate axes on a graph are complex constructs. They usually consist of:

- A straight line, indicating the actual axis
- A set of tick marks along the axis
- A label for each tick mark

In D3, these constituent parts are realized as a collection of SVG elements (also see Figure 7-2):

- A `<path>` element of class `domain` for the axis.
- A set of `<g>` elements of class `tick`, one for each tick mark. Each of these `<g>` elements contains a `<line>` element for the tick mark and a `<text>` element for the associated label.

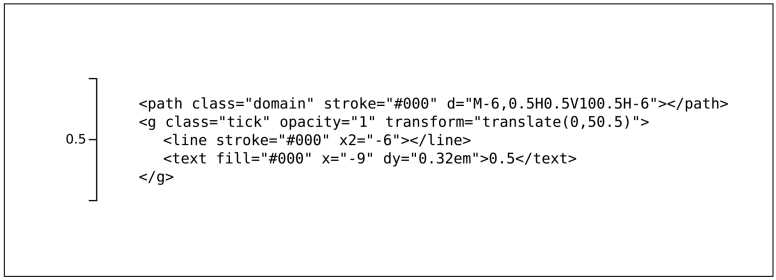


Figure 7-2. A short axis and its constituent SVG elements, as created by the axis component

The D3 axis facility distinguishes between inner and outer tick marks. The outer tick marks are those at the two ends of the actual axis, and are implemented as part of the `<path>` element. The configuration of the inner tick marks and labels is delegated to the underlying scale object. Several of the methods in Table 7-5 in fact simply forward their arguments to the scale object; when in doubt about the behavior, check the scale object's documentation.

The D3 axis component will create and configure all elements of an axis and insert them into the DOM tree. (Remember that a component is a function object that will create elements and add them to the DOM tree—see Chapter 5.) As is common when using components, the surrounding code must provide a *common container*, typi-

cally a <g> element, as parent for the elements of the axis. Axes are always *rendered at the origin* initially and must be moved to their final position through an SVG transform on the common parent. The shared <g> element is also a convenient location for any formatting instructions that apply to all axes elements. (In fact, D3 already injects instructions regarding font type and size into it.) Just remember that the axis component itself does not create the shared container.

Creating an Axis and Moving It into Position

The basic workflow for creating an axis is as follows (see Table 7-4):

1. Instantiate a scale object.
2. Instantiate an axis component (this is a function object).
3. Invoke the axis component while supplying a suitable <g> element as the argument.
4. Use an SVG transform to move the <g> element to its desired location. (This step can happen before or after invoking the axis component.)

These steps are demonstrated in the following snippet:

```
var sc = d3.scaleLinear();
var axMkr = d3.axisBottom(sc);

var g = d3.select( ... ).append( "g" ).attr( "transform", ... );
axMkr( g );
```

If you want to be more idiomatic, you can use JavaScript's synthetic function invocation to combine the last two lines into a single, unbroken chain of method calls:

```
d3.select( ... ).append("g").attr("transform", ...).call(axMkr);
```

Table 7-4. Functions for the creation and basic configuration of axis components (axMkr is an axis component instance)

Function	Description
`d3.axisTop( scale )`	Returns an axis instance for the supplied scale object. The default tick size is 6, the default padding 3. The generated axis is intended for the *top* of the graph; tick marks and labels will be *above* the axis.

Function	Description
`d3.axisRight( scale )`	As `d3.axisTop()`, but the generated axis is intended for the *right side* of the graph; tick marks and labels will be *to the right* of the axis.
`d3.axisBottom( scale )`	As `d3.axisTop()`, but the generated axis is intended for the *bottom* of the graph; tick marks and labels will be *below* the axis.
`d3.axisLeft( scale )`	As `d3.axisTop()`, but the generated axis is intended for the *left side* of the graph; tick marks and labels will be *to the left* of the axis.
`axMkr( selection )`	Creates the elements of the axis as children of the supplied selection, which must be a selection of container elements (`<svg>` or `<g>`). If the selection contains multiple nodes, an axis is created in each. The axis is always rendered at the origin; use transformations to move it into position. Returns nothing (`undefined`).
`axMkr.scale( scale )`	Sets the scale object and returns the axis instance; returns the current scale object when called without arguments.

Customizing Tick Marks and Their Labels

The axis component provides some functions to control the configuration of ticks and tick labels (see Table 7-5), but if you want finer-grained control, you will need to do it yourself. The advantage is, of course, that you can create any appearance you desire; the downside is that you have to write the code. Example 7-1 demonstrates some techniques that may be useful in this context. The example uses a *single* scale object, but creates *four* axes with different appearances; the resulting graph is shown in Figure 7-3.

Table 7-5. Methods to configure tick marks and labels (axMkr is an axis component instance)

Function	Description
`axMkr.ticks( t, fmt )`	A convenience function to set the parameters to be forwarded to the `ticks()` and `tickFormat()` functions of the underlying scale object. The parameter `t` can either be an integer indicating the desired number of tick marks or a time interval (see Table 10-5). The `fmt` argument should be a format string suitable for the creation of a formatter object. (This method always returns the current axis component instance and can therefore not be used as a getter for the current tick configuration values.)

Function	Description
`axMkr.tickValues( [values] )`	If an array of values is specified, then those values take precedence over values calculated by the underlying scale object. Supply `null` to clear previously set values.
`axMkr.tickFormat( format )`	Takes a formatter instance (*not* a format specification) for formatting the tick labels. Supply `null` to clear a previously set value.
`axMkr.tickSize( size )`	Sets the size of both inner and outer tick marks. Defaults to 6.
`axMkr.tickSizeInner( size )`	Sets the size of the actual, inner tick marks. Defaults to 6.
`axMkr.tickSizeOuter( size )`	Sets the size of the markers at both ends of the straight line indicating the actual axis. Defaults to 6. Outer tick marks may coincide with the first and last inner tick marks.
`axMkr.tickPadding( size )`	Sets the padding between tick marks and tick labels. Defaults to 3.

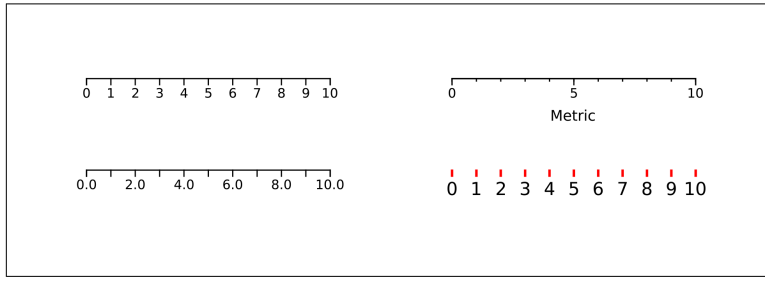


Figure 7-3. Changing the appearance of tick marks and tick labels (see Example 7-1)

Example 7-1. Customizing tick marks (see Figure 7-3)

```
function makeTicks() {
    var sc = d3.scaleLinear().domain( [0,10] ).range( [0,200] );  ❶

    // top left: default settings
    d3.select( "#ticks" ).append( "g" )                           ❷
        .attr( "transform", "translate( 50,50)" )
        .call( d3.axisBottom(sc) );

    // bottom left: additional decimal in labels
    d3.select( "#ticks" ).append( "g" )                           ❸
        .attr( "transform", "translate( 50,125)" )
        .call( d3.axisBottom(sc).tickFormat( d3.format(".1f") ) )
        .selectAll( "text" )
```

```
        .filter( (d,i)=>i%2!=0 ).attr( "visibility", "hidden" );

    // top right: minor and major tick marks, addtl label for axis
    d3.select( "#ticks" ).append( "g" )                                    ❹
        .attr( "transform", "translate(350,50)" )
        .call( d3.axisBottom(sc).tickSize(3).tickFormat( ()=>"" ) );
    d3.select( "#ticks" ).append( "g" )
        .attr( "transform", "translate(350,50)" )
        .call( d3.axisBottom( sc ).ticks(2) )
        .append( "text" ).text( "Metric" )
        .attr( "x", sc(5) ).attr("y", 35 )
        .attr( "font-size", 12 ).attr( "fill", "black" );

    // bottom right: custom appearance
    var g = d3.select( "#ticks" ).append( "g" )                            ❺
        .attr( "transform", "translate(350,125)" )
        .call( d3.axisBottom(sc).tickPadding( 5 ) );
    g.select( ".domain" ).attr( "visibility", "hidden" );
    g.selectAll( ".tick" ).select( "line" )
        .attr( "stroke", "red" ).attr( "stroke-width", 2 );
    g.selectAll( "text" ).attr( "font-size", 14 );
}
```

❶ Create a single scale object that maps an input domain consisting of the interval [0, 10] to a (pixel) range of [0, 200].

❷ The plain axis, as generated by the axis component with default settings.

❸ Show an additional decimal in each tick label. The `tickFormat()` function requires a formatter object, not a format specification, and hence the `d3.format()` factory function is called to produce a formatter instance that will include an additional digit in the tick label. (See Chapter 6 for more information on formatting numbers.) To make room for the additional digit, every second tick label needs to be hidden; the `filter()` method of the `Selection` API helps to select the appropriate axis elements (see Table 3-2).

❹ This version distinguishes between short, unlabeled *minor* tick marks and the labeled *major* tick marks. In fact, the axis is drawn twice: once without labels and with short ticks, and then a second time with longer, labeled, but *fewer* tick marks. To suppress the generation of labels the first time, a function is supplied as the formatter object that always returns an empty

string. Furthermore, an overall text label is added to describe the entire axis: notice how the scale object is used to calculate its position. (The axis component has already set the `text-anchor` attribute to `middle` for the entire axis.)

❺ This final version is mostly intended to demonstrate some additional techniques. In several places, it uses the class names that the axis component creates (such as `domain` and `tick`) to select parts. Because this example uses a larger font size for the tick labels, it is necessary to adjust their padding.

Examples

It's time to put all these ideas together and work through some examples that demonstrate scales and axes in action.

Falling Long-Distance Costs: Linear and Logarithmic Scales

Figure 7-4 shows a data set (the cost of a typical long-distance phone call in the US over time) plotted in two different ways. The outer graph uses linear scales, whereas the inset uses semilog axes to reveal the almost exponential decline in communication costs during the twentieth century.

The commands can be found in Example 7-2. Both parts of the graph obviously have a lot in common, and the shared code has been placed into a function. This function is a closure: it can access the `data` variable from its enclosing scope, so that the data set does not need to be passed in as an argument. The function is also a component, because it takes a `Selection` instance as the first argument and inserts new DOM elements into it. The component creates a plot at the origin; it is up to the calling function to move the generated plot to its final destination. (Also see Chapter 5 for more information about custom components.)

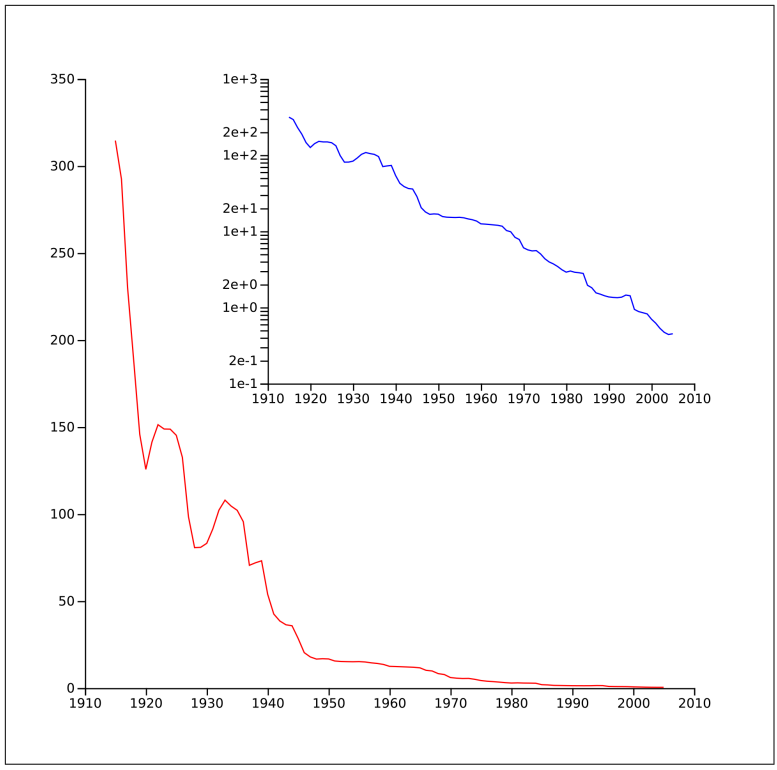


Figure 7-4. Using linear and semilogarithmic coordinate axes, in a figure with an inset (see Example 7-2)

Example 7-2. Commands for Figure 7-4

```
function makeSemilog() {
    d3.text( "cost.csv" ).then( res => {
        var data = d3.csvParseRows( res, d => [ +d[0], +d[1] ] ); ❶

        function draw( sel, scX, scY, width, height ) {          ❷
            scX = scX.domain( d3.extent( data, d=>d[0] ) ).nice() ❸
                .range( [ 0, width ] );
            scY = scY.domain( d3.extent( data, d=>d[1] ) ).nice()
                .range( [ height, 0 ]);

            var ds = data.map( d=>[ scX(d[0]), scY(d[1]) ] );    ❹
            sel.append( "path" ).classed( "curve", true )
                .attr( "d", d3.line()(ds) ).attr( "fill", "none" )

            sel.append( "g" )                                    ❺
                .call( d3.axisBottom( scX ).ticks(10, "d") )
                .attr( "transform", "translate( 0,"+height+")" );
```

```
            sel.append( "g" ).call( d3.axisLeft( scY ) );          ❻
        }

        d3.select( "#semilog" ).append( "g" )                      ❼
            .attr( "transform", "translate( 50, 50 )" )
            .call( draw,d3.scaleLinear(),d3.scaleLinear(),500,500 )
            .select( ".curve" ).attr( "stroke", "red" );

        d3.select( "#semilog" ).append( "g" )                      ❽
            .attr( "transform", "translate( 200, 50 )" )
            .call( draw,d3.scaleLinear(),d3.scaleLog(),350,250 )
            .select( ".curve" ).attr( "stroke", "blue" );
    } );
}
```

❶ Read the data file and convert it to an array of arrays. Convert the entries to numbers.

❷ The first argument to the `draw()` function is a `Selection` object. All new elements created by the function will be added to this selection. When the function is invoked by `call()`, the current selection will automatically be supplied as the first argument.

❸ Configure the scale objects that are passed to the function. Notice that the `nice()` function must be called *after* the domain has been set.

❹ Rescale the data set, then pass it to a line generator. Because the data set matches the format expected by the line generator, it is not necessary to define accessors here: the default accessors will do. The path element is assigned the CSS class `curve` to make it distinguishable later on.

❺ Append a `<g>` element as container for the horizontal axis, then invoke an axis component using one of the scale objects that were configured earlier. Move the axis into position.

❻ Repeat for the vertical axis. Because the entire graph is created at the origin, there is no need to move the vertical axis from its default location.

❼ Finally, append a new `<g>` element as container for the entire graph, move it into position, and invoke the `draw()` function on

this selection. The `call()` function will automatically inject the newly created `<g>` element (the "current selection") as the first argument when calling the `draw()` function. The `call()` function also returns the `<g>` element, hence we can select the path element representing the data set by its CSS class attribute to modify its color. Notice that it is not necessary to pass the color to the `draw()` function—the appearance options can be fixed later.

❽ Repeat, but this time supply a logarithmic scale object for the vertical axis to create a semilogarithmic plot.

Server Load: Time Series and Discrete Colors

Figure 7-5 shows a server's CPU utilization in percent, as a function of time, over a period of almost two-and-a-half hours. For easier reference, idle loads up to 35 percent are shown in green, and overload conditions exceeding 75 percent are in red. The load is recorded every minute, and the beginning of the data file looks like this:

```
timestamp,load
08:01:00,29
08:02:00,29
08:03:00,30
08:04:00,22
08:05:00,20
...
```

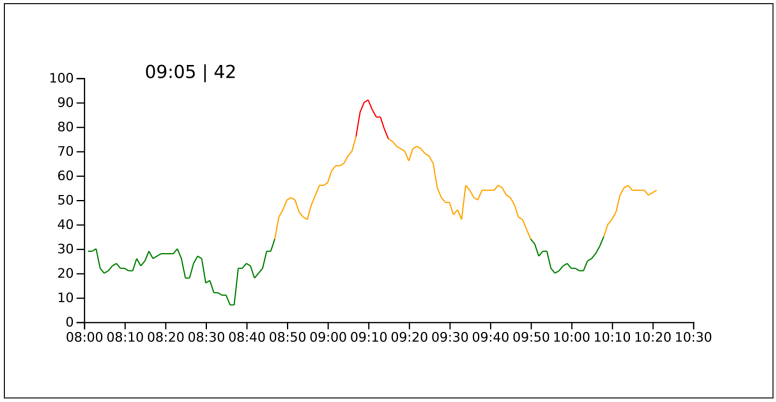


Figure 7-5. A time-series plot. Top left: the mouse position, in coordinates of the data set.

The figure was created using the commands in Example 7-3. This example demonstrates some of the D3 facilities for handling *time and date* information, as well as some other useful techniques.

Example 7-3. Commands for Figure 7-5

```
function makeTimeseries() {
    d3.text( "load.csv" ).then( res => {
        // prepare data
        var parse = d3.utcParse( "%H:%M:%S" );                          ❶
        var format = d3.utcFormat( "%H:%M" );

        var data = d3.csvParse( res, function( d ) {                    ❷
            return { ts: parse(d.timestamp), val: +d.load }
        } );

        // create scale objects
        var scT = d3.scaleUtc()                                         ❸
            .domain( d3.extent( data, d=>d.ts ) ).nice()
            .range( [ 50, 550 ] );
        var scY = d3.scaleLinear()
            .domain( [0, 100] ).range( [ 250, 50 ] );
        var scC = d3.scaleThreshold()                                   ❹
            .domain( [35, 75] ).range( ["green","orange","red"] );

        data = d3.pairs( data,                                          ❺
                        (a,b) => { return { src: a, dst: b } } );

        // draw data and axes
        var svg =
            d3.select( "#timeseries" ).attr( "cursor","crosshair" );

        svg.selectAll("line").data(data).enter().append("line")         ❻
            .attr( "x1", d => scT(d.src.ts) )
            .attr( "x2", d => scT(d.dst.ts) )
            .attr( "y1", d => scY(d.src.val) )
            .attr( "y2", d => scY(d.dst.val) )
            .attr( "stroke", d=>scC( (d.src.val + d.dst.val)/2 ) );

        svg.append( "g" ).attr( "transform", "translate(50,0)" )        ❼
            .call( d3.axisLeft(scY) );
        svg.append( "g" ).attr( "transform", "translate(0,250)" )
            .call( d3.axisBottom(scT).tickFormat( format )
                   .ticks( d3.utcMinute.every(10) ) );

        // display mouse position
        var txt = svg.append("text").attr("x",100).attr("y",50)         ❽
            .attr("font-family","sans-serif").attr("font-size",14);
        svg.on( "mousemove", function() {
            var pt = d3.mouse( svg.node() )
```

```
            txt.text( format( scT.invert( pt[0] ) ) + " | " +          ❾
                      d3.format( ">2d" )( scY.invert(pt[1]) ) );       ❿
        } );
    } );
}
```

❶ Create functions to parse and format timestamps. The `parse()` function parses a string and returns a JavaScript `Date` object.

❷ Parse the input data and apply a conversion function. The latter turns every record in the data file into an object with two properties: `ts`, which is a JavaScript `Date`, and a number `val`. (See Chapter 6 for more information about reading and parsing files.)

❸ Create the necessary scale objects. For the horizontal axis, first the interval bounded by the minimum and maximum values in the data set is found using `d3.extent()` (see Chapter 10), then this interval is extended to "round" values using `nice()`.

❹ Colors are assigned using fixed thresholds. The thresholds are set using the `domain()` function of `d3.scaleThreshold()`. Notice that there are, of course, only *two* thresholds separating the *three* colors used in the graph.

❺ The original data set consists of an array of *points*, but to draw it with *lines*, it is transformed into an array of *pairs* of consecutive points, so that each pair marks the beginning and end of a line. The `d3.pairs()` function takes an array and combines any two consecutive elements using the provided callback. In this case, the callback creates a new object, with `src` and `dst` properties, and assigns the two original data points to these properties. Each object now represents a line segment. (See Chapter 10 for more information on `d3.pairs()` and other convenience functions that operate on arrays.)

❻ Using the transformed data set, creating the individual lines for the graph is now easy. The color of each line is based on the average data value at its two end points.

❼ Create the two axes. The horizontal axis, representing time, uses the format function created earlier to turn JavaScript `Date`

objects into tick labels. It also uses the D3 *time interval* facility: `d3.utcMinute` represents a time interval of one minute. The `every()` function retains only the indicated multiple of the original interval, 10 minutes in this case. The time scale object can interpret this information and generate tick marks with the desired frequency. You may want to remove the calls to `tickFormat()` and `ticks()` to observe the default behavior for a time axis. (See Chapter 10 to learn more about the D3 time intervals.)

❽ The remaining commands display the current mouse position, but in the coordinates of the original data set, not in pixel coordinates. (See Example 4-1 for an example that displays the current mouse position in pixel coordinates, instead.)

❾ Inside the `mousemove` event handler, the pixel coordinates for the current mouse position are obtained using `d3.mouse()`, but are then converted into the coordinates of the data set using the `invert()` function on the scale objects for the horizontal and vertical axis.

❿ The horizontal (time) coordinate is formatted using the `format()` function defined earlier, but to format the vertical position, a numerical formatter is created using `d3.format()` and evaluated immediately. The format specifier `">2d"` creates a right-aligned text label that has space for two digits. (See Chapter 6 to learn about formatting numbers.)

CHAPTER 8

Colors, Color Scales, and Heatmaps

Colors can serve different purposes in visualization: they may simply serve to make a graph more interesting or more pleasing, or they may help to reinforce and emphasize aspects of the data, or they may be primary carriers of information themselves. This chapter will explain how colors are represented in D3 and then discuss various color schemes and how they can be used to display information. The chapter ends with a description of false-color plots that strictly rely on color to represent data.

Colors and Color Space Conversions

Specifying an individual color in D3 is easy: you provide a string with either the name of a predefined color, or the color's red, green, blue (RGB) or hue, saturation, lightness (HSL) components using the CSS3 syntax (see Appendix B). But the string format is not very convenient if you want to manipulate colors programmatically. For such purposes, you can obtain a `color` object using one of the factory functions in Table 8-1. A `color` object can be used wherever a color specification is expected: its `toString()` function will be called automatically and return a representation of the color in CSS3 format.

The returned `color` objects provide only a minimal API; they mostly serve as containers for their channel information. Every `color` object exposes its three components as suitably named properties, one for each channel. In addition, each object also has an `opacity` property for the alpha channel.

The functions in Table 8-1 accept as arguments one of the following and return a `color` object in the requested color space:

- A CSS3 color string
- Another `color` object (for conversion to another color space)
- A set of three (or four, if an opacity value is specified) components

The exception is the generic `d3.color()` factory, which takes a CSS3 string or another `color` object and returns an RGB or HSL `color` object, depending on the input.

Table 8-1. Factory functions to create `color` objects in different color spaces, the color components of the returned object, and their legal ranges

Function	Components[a]	Comments
`d3.rgb()`	`r, g, b`	• 0 ≤ r, g, b ≤ 255 strictly
`d3.hsl()`	`h, s, l`	• h: any value (positive or negative), the hue will be interpreted modulo 360 • 0 ≤ s, l ≤ 1 strictly
`d3.lab()`	`l, a, b`	• 0 ≤ l ≤ 100 strictly • -100 ≤ a, b ≤ 100 approximately
`d3.hcl()`	`h, c, l`	• h: any value (positive or negative), the hue will be interpreted modulo 360 • -125 ≤ c ≤ 125 approximately • 0 ≤ l ≤ 100 approximately
`d3.color()`	`r, g, b` or `h, s, l`	Input must be a CSS3 string or another object; output is RGB, unless the input is HSL.

[a] All color objects also have an `opacity` component, with values from 0 (transparent) to 1 (opaque).

The allowed parameter ranges are in general different for each color space. The functions in Table 8-1 return `null` if their inputs are obviously illegal (such as negative RGB components), but even a nonnull return value does not guarantee that the returned color is *displayable*. You can use the function `displayable()` on the returned color instance to find out. If a color is not displayable, then its `toString()` method will substitute a "suitable" displayable color

(such as white or black when the required color is too bright or too dark) instead.

Color Spaces

In general, three coordinates are required to specify a color (not counting transparency). In the RGB model, the color space is represented as a cube, spanned by the three component axes at right angles. The RGB space is closest to the technical hardware realization but is notoriously unintuitive (quick, what does `#b8860b` look like?).

In the HSL model, the RGB cube is deformed into a double cone, with the axis of the cone running along the space diagonal (from black to white) of the original color cube. The three components are now interpreted as cylindrical coordinates: hue measures the angle around the cylinder axis, lightness the position along the axis (from black at the bottom to white at the top), and saturation is the radial distance from the axis (with greys making up the axis, and fully saturated colors the surface of the cone). Fully saturated, maximally bright colors are found at half height, along the "belly" of the double cone.

Compared to RGB, the HSL coordinates have a more intuitive interpretation, but both RGB and HSL suffer from the same defect: their color components represent the intensities of the rendering device but do not take into account the nonuniform nature of human *color perception*. This leads to a variety of inconsistencies. Considering named CSS3 colors only, you may want to compare the three colors DarkKhaki (`#bdb76b`), Gold (`#ffd700`), and Yellow (`#ffff00`). Of these, DarkKhaki has a greater lightness (in the HSL space) than the other two, although it appears darker than either of them. Yellow and Gold have equal lightness but appear quite different. Other examples where the lightness value does not match perception are easily found.

Color spaces based on *perceived* intensities help to avoid these problems. Two of these are the CIELAB (or LAB) and the hue, chroma, luminance (HCL) color spaces. In the LAB space, the coordinates again span a parallelepiped, with l measuring the lightness, and the a and b coordinates expressing the position along two axes, one running from red to green, and the other from blue to yellow. (You will notice that these two axes are approximately orthogonal on the standard color wheel.) The HCL space casts the

same information into cylindrical coordinates, similar to the HSL space, with chroma expressing how colorful a color appears, and luminance being a measure of its radiance. The HCL space provides the same convenient intuition as the familiar HSL space, but color comparisons tend to work better than when using HSL components.

A practical problem when using color components that express human perception, rather than technical capability, is that it is fairly easy to construct colors that cannot be rendered by standard hardware. In contrast to RGB and HSL, the ranges of the components are not limited to a fixed set of values, and care must be taken that the resulting color is in fact displayable.

Color Schemes

D3 comes with a fairly large number of color schemes for use with scale objects. It can be difficult to make sense of the plethora of built-in schemes, hence here is a survey of what is available.[1]

Cartographic Schemes

The following schemes are based on work originally intended for use in cartographic maps (the kind you find in an atlas) to depict political or other thematic information. They are best suited for coloring *areas*, but work less well for lines or shapes with minute detail. Because they are designed to be easy on the eyes, they are also often a good choice to *enhance* the perception of information that is already represented by other means.

Categorical schemes
: Nine nonsemantic, categorical schemes, each with 8 to 12 entries. Adjacent colors tend to have strong contrasts, except for the scheme `d3.schemePaired`, which consists of six pairs, each consisting of a light and a dark shade of comparable hue. (See Figures 5-7 and 9-3 for examples.)

Diverging schemes
: Nine diverging schemes, each having different dark and saturated colors of different hue at the ends, and passing through a

1 See *https://github.com/d3/d3-scale-chromatic* for visual representations.

light and pale version of grey, white, or yellow in the center. Use these to show deviations from a baseline in either direction. (See Figure 8-1.)

Single-hue monotonic schemes
: Six monotonic schemes, each running from a light and pale white or grey to a dark and saturated color, using a single hue. (See Figure 9-2 for an application.)

Two- or three-hue monotonic schemes
: Twelve continuous schemes, each running from a light and pale white or grey color over one or two intermediate but "similar" colors (such as blue and green, or yellow, green, and blue) to a dark and saturated color.

These color schemes come in two variants:

- A continuous variant (as an interpolator) for use with `d3.scaleSequential()`
- A discrete variant (as an array of discrete color values) for use with `d3.scaleOrdinal()` or one of the binning scales (such as `d3.scaleThreshold()`)[2]

Most of the discrete schemes come in several versions, each containing a different number of equally spaced, discrete elements (between 3 and about 10, check the D3 Reference Documentation regarding the exact number for each scheme).

False-Color Schemes

The following schemes are intended for false-color plots or heatmaps. These schemes are only available as continuous interpolators (and not in discrete versions).

Multihue schemes
: Five multihue schemes, all running from a very dark (nearly black) color to yellow or white. These schemes are careful to vary saturation *and* lightness simultaneously (which might be a better idea in principle than in practice).

2 Having a separate discrete color scheme, as opposed to just evaluating the corresponding interpolator at a discrete set of values, allows mapping *nonnumerical* categorical values to colors as well.

Rainbow schemes

Two rainbow schemes that are perceptually more uniform than the standard color wheel in HSL space. For one of these, its two halves are available separately, one running over the warm (red) side of the color wheel, and the other over the cool (blue, green) side.

The "sinebow" scheme `d3.interpolateSinebow` is very interesting. In the standard rainbow, the RGB components vary in a regular, piecewise linear fashion as the hue ranges from 0 to 360. In the sinebow, this piecewise linear behavior has been replaced by three sine functions, which are phase-shifted by 120 degrees relative to each other. The result is a bright but much more perceptually uniform rainbow (see Figure 8-1). Highly recommended for its appearance and conceptual simplicity.[3]

Palette Design

The design of color schemes or palettes for data visualization is a difficult topic. Here are some considerations and recommendations.[4]

- Distinguish whether color is used to enhance the perception of some information that is also available in the graph in another form, or whether the color itself is the primary carrier of essential information. The shading of topographic maps, for example, merely emphasizes the elevation information available through contour lines, whereas in a true false-color plot like Figure 8-2, all relevant information is encoded strictly in color.
- Decide whether the data has an inherent ordering, and if so, ensure that the color scheme supports it. For example, we conventionally and intuitively associate blue with lower, and red with higher values; a well-designed graph should take this into account. (The built-in color schemes are inconsistent in this regard!) Sometimes perceptual ordering among colors may be undesirable (for example, for certain kinds of categorical data).

3 Also see *http://basecase.org/env/on-rainbows*.

4 I have a bit more to say about this topic in Appendix D of my book *Gnuplot in Action* (Manning Publications, Second Edition).

- Be aware of semantics conventionally associated with certain colors (like the traffic-light colors red, yellow, and green).
- Avoid very dark and very light colors, because they tend to hide details ("white on white" or "black on black"). Avoid screaming colors: strike a balance between the garish and the imperceptible. (Perceptually uniform color spaces, such as HCL, can help with this.)
- Place strong gradients where you want them. In many cases, relatively small changes in some part of the input domain are much more interesting than changes elsewhere. By making sure that strong visual changes in the color scheme are located where changes in the data are most relevant, you can enhance perception of important details.
- In a similar spirit, you may want to use noncontinuous changes to indicate a semantic threshold in the data. (Topographic maps, for example, usually use continuous gradients, but always have a sharp transition to indicate coast lines.)
- Whenever color is used to convey information (as opposed to purely aesthetic purposes), the graph should contain a "color box" that explicitly resolves the mapping between colors and values. No matter how intuitive a color scheme may appear, it is impossible to reconstruct this information reliably from the graph alone.
- Finally, keep in mind that about 8% of men and 0.5% of women are at least partially affected by color-blindness, with red/green blindness the most common.

Other Color Schemes

The sheer number and apparent variety of built-in color schemes may appear to exhaust all possibilities, but depending on context and purpose, other schemes may be more suitable (see "Palette Design" on page 158). It can be interesting to peruse collections of existing color schemes for ideas.[5] There is no reason to use the built-

5 A very large and diverse repository of palettes, not all of which are intended for scientific visualization, can be found at *http://bit.ly/2Wamh09*. Of particular interest in the present context is the work done by Kenneth Moreland and by the Generic Mapping Tools project, especially the GMT Haxby palette.

in schemes if they don't seem appropriate or are hard to adapt to a specific application.

Color Scales

In order to utilize color to represent information, it is often convenient to combine color with a scale object (see Chapter 7): specifically, to use colors to define the scale object's output range. This section collects a few simple applications and techniques.

Discrete Colors

Discrete color scales can be realized using either binning scales or a discrete scale object. Binning scales, such as those created by `d3.scaleQuantize()` or `d3.scaleThreshold()`, make sense when a continuous range of numbers is to be represented through a discrete set of colors. The first divides the input domain into equally sized bins, the latter expects the user to explicitly define the threshold values that separate bins—see Examples 4-7 and 7-3. Remember that the `domain()` function is semantically overloaded for binning scales (see Chapter 7 for details):

```
var sc1 = d3.scaleQuantize().domain( [0, 1] )
    .range( [ "black", "red", "yellow", "white"] );
var sc2 = d3.scaleThreshold().domain( [ -1, 1] )
    .range( [ "blue", "white", "red" ] );
```

Discrete scales, as created by `d3.scaleOrdinal()`, are appropriate when a discrete set of input values should be displayed using different colors. A discrete scale object can establish an explicit relationship between specific values from the input domain and the colors to represent them:

```
var sc = d3.scaleOrdinal( [ "green", "yellow", "red"] )
    .domain( [ "good", "medium", "bad" ] );
```

Alternatively, when no domain has been specified, a discrete scale object assigns the next unused color to each new symbol as it arises. Examples can be found in Examples 5-6 and 9-5.

Their implementation through scale objects aside, colors for discrete color schemes should be chosen according to their purpose. Sometimes the goal is merely to make it easier to distinguish different graphical elements without implying either semantics or ordering: the built-in categorical schemes are of this kind (see Figures 5-7 and

9-3 for examples). At other times, the colors should convey a semantic meaning, although not necessarily monotonic increments or a strict ordering: a red/yellow/green "traffic light" scheme as in Figure 7-5 is an example. And finally, sometimes distinct color schemes are intended to represent a monotonic sequence of steps (as in the right side of Figure 9-2). In the latter case, binning scales may be used together with the built-in monotonic or diverging scales to map a continuous range of values to a specific set of fixed colors with a clear sense of ordering.

Color Gradients

The ability of D3 to interpolate colors makes it particularly easy to generate color gradients. The examples in this section are mostly intended to demonstrate the syntax for a few simple, yet typical cases (see Example 8-1 and Figure 8-1).

Example 8-1. Creating color scales (see Figure 8-1)

```
sc1 = d3.scaleLinear().domain( [0, 3, 10] )                          ❶
    .range( ["blue", "white", "red"] );

sc2 = d3.scaleLinear().domain( [0, 5, 5, 10] )                       ❷
    .range( ["white", "blue", "red", "white"] );

sc3 = d3.scaleSequential( t => "" + d3.hsl( 360*t, 1, 0.5 ) )        ❸
        .domain( [0, 10] );

sc4 = d3.scaleSequential( t => d3.interpolateSinebow(2/3-3*t/4) )    ❹
        .domain( [0, 10] );

sc5 = d3.scaleDiverging( t => d3.interpolateRdYlGn(1-t) )            ❺
        .domain( [0, 2, 10] );

sc6 = d3.scaleSequential( d3.interpolateRgbBasis(                    ❻
    ["#b2d899","#ffffbf","#bf9966","#ffffff"] ) ).domain( [0,10] );
```

❶ A simple blue/white/red gradient using default color space interpolation—but notice the asymmetrical position of the white band!

❷ A gradient with a sharp transition in the middle.

❸ The much-maligned "standard HSL rainbow"—mostly to show the syntax (not because it is a particularly good color scale).

❹ A much better rainbow, using the built-in "sinebow" interpolator. Here, it is traversed backward (so that it runs from blue to red, conveying customary low-to-high semantics), and its range is restricted to prevent it from wrapping around and reusing colors.

❺ An example of a diverging scale. It employs the built-in red/yellow/green interpolator, but changes its direction so that large values are represented as red. The domain of a diverging scale must specify *three* numeric values; the middle value is mapped to the interpolator position $t = 0.5$. Here, the yellow color, representing the center of the returned range of colors, is displaced to the value 2 in the input domain.

❻ An example for the kind of color scheme conventionally used in topographic maps. In contrast to other schemes, the lightness does not increase monotonously, but exhibits additional, systematic variation (the green and brown are relatively dark, the yellow and white relatively light), which may provide a visual aide. This scheme uses the `d3.interpolateRgbBasis` interpolator, which constructs a spline through all supplied colors (which are assumed to be equally spaced). The advantage of the spline interpolator is not so much a smoother color scheme, but that the user does not have to specify the location of the intermediate colors using `domain()`.

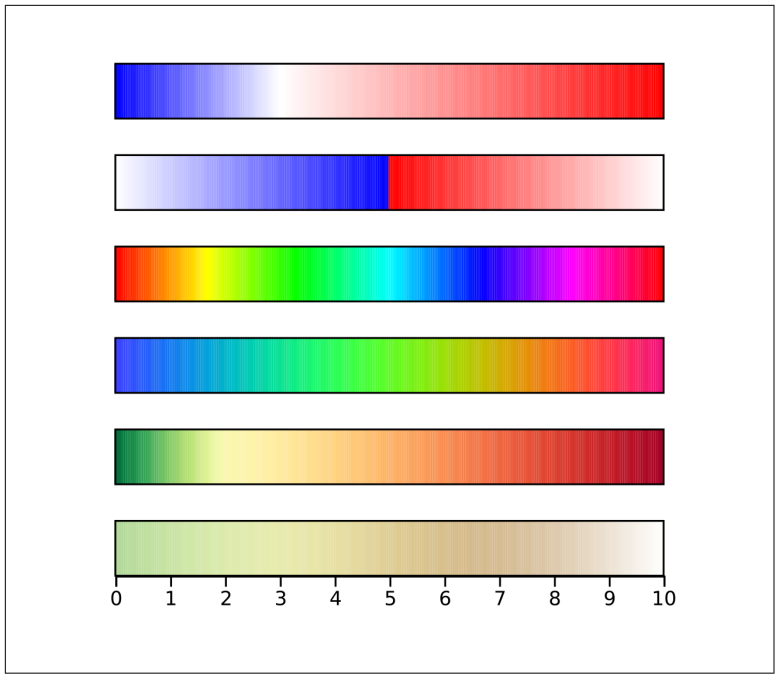


Figure 8-1. Color scales (see Example 8-1)

Making a Color Box

Any graph that relies on color to convey meaning (as opposed to using color merely to enhance appearance and perception) needs a *key* that clearly displays which values are mapped to which colors. Sometimes a textual description is sufficient, but a visual *color box* is usually much clearer. D3 does not have a built-in component for this purpose, but it is easy enough to create one. The one shown in Example 8-2 was used to generate the instances in Figure 8-1; it can serve as a model for general applications.

Example 8-2. Commands to create a color box component (compare Figure 8-1)

```
function colorbox( sel, size, colors, ticks ) {                    ❶
    var [x0, x1] = d3.extent( colors.domain() );                   ❷
    var bars = d3.range( x0, x1, (x1-x0)/size[0] );

    var sc = d3.scaleLinear()                                      ❸
        .domain( [x0, x1] ).range( [0, size[0] ] );
    sel.selectAll( "line" ).data( bars ).enter().append( "line" ) ❹
```

```
        .attr( "x1", sc ).attr( "x2", sc )
        .attr( "y1", 0 ).attr( "y2", size[1] )
        .attr( "stroke", colors );

    sel.append( "rect" )                                          ❺
        .attr( "width", size[0] ).attr( "height", size[1] )
        .attr( "fill", "none" ).attr( "stroke", "black" )

    if( ticks ) {                                                 ❻
        sel.append( "g" ).call( d3.axisBottom( ticks ) )
            .attr( "transform", "translate( 0," + size[1] + ")" );
    }
}
```

❶ In addition to the target selection, the component takes the following arguments: the size (in pixels) of the color box as a two-element array, the actual color scale, and—optionally—a regular scale for creating tick marks.

❷ This creates an array that contains, for each pixel in the desired width of the color bar, the corresponding value from the input domain.

❸ This scale maps the values of the original domain to pixel coordinates in the colorbox.

❹ Create a one-pixel wide, colored line for each entry in `bars`.

❺ Put a box around the colors…

❻ … and add a set of tick marks if an appropriate scale object was passed in.

False-Color Graphs and Related Techniques

Whenever data can naturally be projected into a two-dimensional plane, then false-color plots or "heatmaps" are an attractive option, possibly together with (or as an alternative to) contour lines.

Heatmaps

A form of visualization that relies particularly heavily on color coding is the *false-color plot* or *heatmap*. For data points on a two-dimensional grid, the observed value is represented as color: this is the way elevation is commonly shown in topographic maps.

The number of individual graph elements (data points or pixels) in a false color plot can quickly become large, even for grids of quite modest size. For performance reasons, it may therefore be advisable (or even necessary) to use the HTML5 `<canvas>` element when creating such plots in D3 (see "The HTML5 <canvas> Element" on page 167). Example 8-3 demonstrates a simple use of the `<canvas>` element; Figure 8-2 shows the resulting figure.

Example 8-3. A false-color graph of parts of the Mandelbrot set. This graph was created using an HTML5 canvas element (see Figure 8-2).

```
function makeMandelbrot() {
    var cnv = d3.select( "#canvas" );                                    ❶
    var ctx = cnv.node().getContext( "2d" );

    var pxX = 465, pxY = 250, maxIter = 2000;                            ❷
    var x0 = -1.31, x1 = -0.845, y0 = 0.2, y1 = 0.45;

    var scX = d3.scaleLinear().domain([0, pxX]).range([x0, x1]);         ❸
    var scY = d3.scaleLinear().domain([0, pxY]).range([y1, y0]);

    var scC = d3.scaleLinear().domain([0,10,23,35,55,1999,2000])         ❹
        .range( ["white","red","orange","yellow","lightyellow",
                 "white","darkgrey"] );

    function mandelbrot( x, y ) {                                        ❺
        var u=0.0, v=0.0, k=0;
        for( k=0; k<maxIter && (u*u + v*v)<4; k++ ) {
            var t = u*u - v*v + x;
            v = 2*u*v + y;
            u = t;
        }
        return k;
    }

    for( var j=0; j<pxY; j++ ) {                                         ❻
        for( var i=0; i<pxX; i++ ) {
            var d = mandelbrot( scX(i), scY(j) );
            ctx.fillStyle = scC( d );
            ctx.fillRect( i, j, 1, 1 );
        }
    }
}
```

❶ Select the `<canvas>` element from the page and use it to obtain a drawing context for simple, two-dimensional graphics. Notice that `getContext()` is not part of D3, but is part of the DOM

API, hence you must first obtain the underlying DOM `Node` from the D3 `Selection` using the `node()` method.

❷ Fix some configurational parameters: the size of the canvas in pixels, the maximum number of steps for the Mandelbrot iteration, and the region of interest in the complex plane.

❸ Create two scales that map pixel coordinates to locations in the complex plane.

❹ Create a scale to map the number of iteration steps to a color. The appearance of the graph depends strongly on the placement of the intermediate values.

❺ This function implements the actual Mandelbrot iteration: for a given point $x + iy$ in the complex plane, perform the iteration until either the squared distance from the origin exceeds 2^2 or until the maximum number of iterations is exceeded; return the number of steps taken. (See the Wikipedia page on Mandelbrot sets.)

❻ The double loop runs over all pixels of the canvas. The pixel coordinate is transformed to a location in the complex plane, which is passed to the `mandelbrot()` function. Its return value is converted to a color, which is used to paint a filled, 1×1 rectangle (that is, a pixel) on the canvas.

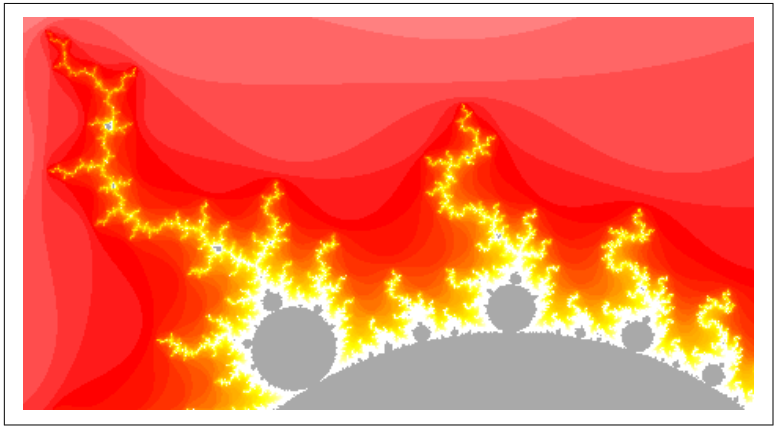


Figure 8-2. Using the HTML5 <canvas> element (see Example 8-3)

The HTML5 <canvas> Element

The HTML5 `<canvas>` element is a device for creating *bitmap* images. Like SVG, it can be scripted from within the browser, but unlike SVG, it offers only low-level facilities. In particular, it does not maintain distinct graph elements that can be manipulated individually, the way the DOM tree does. A canvas image is a "flat" bitmap.

It is important to understand that the `<canvas>` element is primarily a placeholder. To draw anything, you must first obtain a *drawing context* using the `getContext()` API call. Different types of drawing contexts are available; each exposes a different API to support different types of programming (such as accelerated 3D graphics).

We will only be concerned with the simple `CanvasRenderingContext2D` rendering context. It provides only a few graphics primitives, among them are ways to draw filled and unfilled rectangles or text, and a `path` facility that uses a kind of turtle graphics similar to the SVG `<path>` element. The stroke or fill color must be set (as a context property) before drawing anything.

The most basic workflow to draw on an HTML5 canvas from D3 therefore looks like this:

```
var cnv = d3.select( "#canvas" );
var ctx = cnv.node().getContext( '2d' );

ctx.fillStyle = color;
ctx.fillRect( x, y, w, h );
```

The value of `color` should be a CSS3 color specification, `x` and `y` are the position of the top-left corner of the rectangle, and `w` and `h` its width and height.

Overall, the `<canvas>` element is straightforward to use. A tutorial is available from MDN: MDN Canvas Tutorial.

Contour Lines

An alternative (or possible addition) to false-color plots, suitable mostly for smoothly varying data, uses *contour lines*: curves representing levels of equal elevation (as in a topographic map). D3 provides a *layout* to compute the location of such lines (see Table 8-2).

Table 8-2. Functions for calculating contour lines (conMkr is a contours layout instance)

Function	Description
`d3.contours()`	Returns a new contours layout instance with default settings.
`conMkr( [data] )`	Calculates contours for the supplied data set. The data must be formatted as a one-dimensional array, such that the element at the position `[i, j]` in the grid occupies the array element with index `[i, j*cols]`. Returns an array of GeoJSON objects, sorted by the threshold value they represent.
`conMkr.size( [cols, rows] )`	Sets the number of columns and rows as a two-element array.
`conMkr.thresholds( args )`	The argument should be an array of values for which contour lines should be calculated. If the argument is a single integer `n`, then approximately `n` contour lines will be calculated with suitable separation.
`conMkr.contour( [data], threshold )`	Generates a single contour for the supplied data set at the specified threshold value. Returns a single GeoJSON object.

The required representation of the data is peculiar. Data points are expected to lie on a regular, rectangular `cols×rows` grid, stored as a *single, one-dimensional array* of numbers, with the rows forming a contiguous sequence. The item at array index `i + j*cols` therefore represents the point at location `[i, j]`. There are no provisions to associate actual ("domain") coordinates with each data point. One consequence is that the resulting graph will be `cols×rows` pixels in size; if you want a different size, you must apply a scaling transformation.

When the contours layout mechanism is invoked on the data, it returns an array of GeoJSON objects, one for each contour.[6] You can then use the `d3.geoPath()` generator to generate a command string suitable for the `d` attribute of a regular `<path>` element. (This process is equivalent to the one discussed in Chapter 5.) Each of the contour objects also exposes a `value` property containing the threshold value represented by the contour.

6 GeoJSON is a standard for representing geographical features; it is defined in RFC 7946.

If the contour lines are filled with color, the resulting graph is again a false-color plot or heatmap. In Example 8-4, both representations are used together: first, a large number of filled contours is drawn to create a smooth colored background. Then, a few contour lines are drawn on top of this background at specific threshold values (see Figure 8-3).

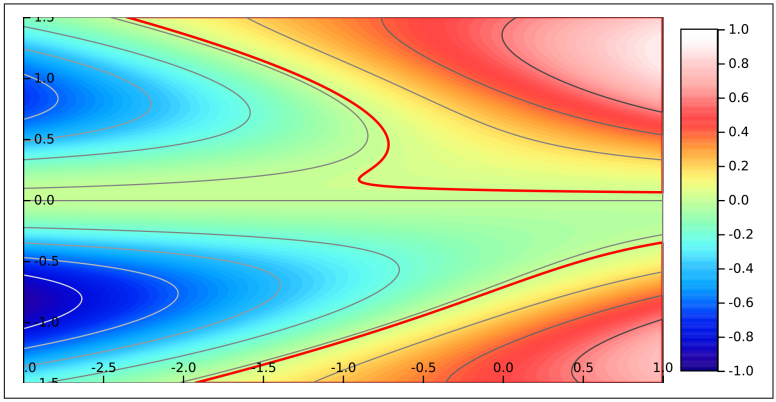


Figure 8-3. A false-color plot of a smooth function. This graph was created using the D3 contours layout mechanism (see Example 8-4).

Example 8-4. Creating a false-color plot using the contours layout mechanism (see Figure 8-3)

```
function makeContours() {
    d3.json( "haxby.json" ).then( drawContours );                    ❶
}

function drawContours( scheme ) {
    // Set up scales, including color
    var pxX = 525, pxY = 300;                                        ❷
    var scX = d3.scaleLinear().domain([-3, 1]).range([0, pxX]);
    var scY = d3.scaleLinear().domain([-1.5, 1.5]).range([pxY, 0]);
    var scC = d3.scaleSequential(
        d3.interpolateRgbBasis(scheme["colors"]) ).domain([-1,1])    ❸
    var scZ = d3.scaleLinear().domain( [-1, -0.25, 0.25, 1] )
        .range( [ "white", "grey", "grey", "black" ] );

    // Generate data
    var data = [];                                                   ❹
    var f = (x, y, b) => (y**4 + x*y**2 + b*y)*Math.exp(-(y**2))
    for( var j=0; j<pxY; j++ ) {
        for( var i=0; i<pxX; i++ ) {
            data.push( f( scX.invert(i), scY.invert(j), 0.3 ) );
        }
```

```
    }

    var svg = d3.select( "#contours" ), g = svg.append( "g" );      ❺
    var pathMkr = d3.geoPath();                                     ❻

    // Generate and draw filled contours (shading)
    var conMkr = d3.contours().size([pxX, pxY]).thresholds(100);    ❼
    g.append("g").selectAll( "path" ).data( conMkr(data) ).enter()
        .append( "path" ).attr( "d", pathMkr )                      ❽
        .attr( "fill", d=>scC(d.value) ).attr( "stroke", "none" )

    // Generate and draw contour lines
    conMkr = d3.contours().size( [pxX,pxY] ).thresholds( 10 );      ❾
    g.append("g").selectAll( "path" ).data( conMkr(data) ).enter()
        .append( "path" ).attr( "d", pathMkr )
        .attr( "fill", "none" ).attr( "stroke", d=>scZ(d.value) );

    // Generate a single contour
    g.select( "g" ).append( "path" )                                ❿
        .attr( "d", pathMkr( conMkr.contour( data, 0.025 ) ) )
        .attr( "fill", "none" ).attr( "stroke", "red" )
        .attr( "stroke-width", 2 );

    // Generate axis
    svg.append( "g" ).call( d3.axisTop(scX).ticks(10) )             ⓫
        .attr( "transform", "translate(0," + pxY + ")" );
    svg.append( "g" ).call( d3.axisRight(scY).ticks(5) );

    // Generate colorbox
    svg.append( "g" ).call( colorbox, [280,30], scC )               ⓬
        .attr( "transform", "translate( 540,290 ) rotate(-90)" )
        .selectAll( "text" ).attr( "transform", "rotate(90)" );
    svg.append( "g" ).attr( "transform", "translate( 570,10 )" )
        .call( d3.axisRight( d3.scaleLinear()
                             .domain( [-1,1] ).range( [280,0] ) ) );
}
```

❶ Load the colors for the color scheme from a file, passing the result to the `drawContours()` function, which does the actual work.[7]

❷ Set the size of the resulting figure in pixels. Remember that there needs to be one data point per pixel.

7 This color scheme is based on the GMT Haxby palette from the Generic Mapping Tools project.

❸ The colors loaded from the file are used together with the `d3.interpolateRgbSpline` interpolator to create a color scale object (compare the last item in Example 8-1). A separate scale object (`scZ`) will determine the color of the *contour lines* to make sure they are clearly visible against the changing color background.

❹ Generate the data by evaluating the function `f(x,y)` for each pixel coordinate. The `invert()` function of the scale object is used to obtain the domain coordinates for each pixel location.

❺ Obtain a handle on the DOM tree, and append a `<g>` element as overall container for the main plot.

❻ Create a `d3.geoPath` generator instance. This is a function object; when invoked on a GeoJSON object, it will return a string suitable for the `d` attribute of a `<path>` element.

❼ Create a contours layout instance, and set the number of pixels in the graph and the data. The chosen number of contours is large: when each is filled with color, they will yield a smooth heatmap.

❽ For each contour, append a `<path>` element, and fill it with color according to the `value` property on the contour object. The `pathMkr` will be invoked on each contour and return a string suitable for the `d` attribute of the `<path>` element.

❾ Reconfigure the contours layout to create approximately 10 contours, then invoke it to produce unfilled contour lines.

❿ Just to show how it is done: the `contour()` function can be used to generate a *single* contour at a specific threshold value.

⓫ Use the original scale object to add coordinate axes to the graph…

⓬ … and add a color box to show the numeric values corresponding to the colors in the heatmap. This code reuses the `colorbox` component from Example 8-2, then uses an SVG transformation to create vertical orientation.

CHAPTER 9

Trees and Networks

This chapter describes facilities to arrange entire *collections* of graph elements simultaneously while taking into account certain constraints between them and with regard to their relative placement. First, we will deal with hierarchical data sets and the tree-like graphs that express their specific topology. Then we will deal with sets of data points subject to more general constraints, such as networks.

Trees and Hierarchical Data Structures

Hierarchical, tree-like data structures occur frequently, and D3 includes a number of different *layouts* to represent them as graphs. As explained in Chapter 5, layouts ingest a data set, calculate the appropriate sizes and positions, and add them to the input data set in the form of additional member variables. But layouts don't create any graphical elements; it is up to the calling code to make use of the information added by the layout.

Preparing the Data

The D3 hierarchical layouts require the data to be represented as a tree of D3 `Node` instances (see Tables 9-1 and 9-2).[1] If the data is in a hierarchical format already, it can be converted immediately using `d3.hierarchy()`. The only requirement is that every node in the

1 Do not confuse D3 `Node` instances, used to represent generic, hierarchical data structures, with the DOM `Node` interface.

original data structure has knowledge about its own children. If the data is instead provided as a tabular set of links between parent and child nodes, then `d3.stratify()` can construct the required tree of `Node` instances (check the D3 Reference Documentation).

Table 9-1. Methods of D3 Node instances (node is a D3 Node instance)

Function	Description
`d3.hierarchy( data, accessor )`	Creates a tree of `Node` instances from the supplied data set; returns the root node. The accessor function is invoked for each node in the original data set and should return an array of child nodes; it defaults to `d => d.children`.
`node.ancestors()`	Returns an array of ancestor nodes, starting with the current node.
`node.descendants()`	Returns an array of descendant nodes, starting with the current node.
`node.leaves()`	Returns an array of leaf (that is, childless) nodes.
`node.path( target )`	Returns an array of nodes that constitute the shortest path between the current and the target node.
`node.links()`	Returns an array of links for the current node. Each entry in the returned array is an object having `source` and `target` properties, which will be set to the parent and child node instances, respectively.
`node.count()`	For every node in the subtree starting at and including the current node, calculates the number of leaf nodes below each node and assigns the value to each node's `value` property. Leaf nodes count as one. Returns the current node.
`node.sum( accessor )`	Traverses the subtree starting at and including the current node in post-order traversal, evaluating the accessor function for each node, with the node`s `data` property as argument. The accessor must return a nonnegative number. For each node, the accessor's return value is added to the cumulative sum of all descendant nodes, and the result is assigned to the node's `value` property. Returns the current node.
`node.sort( comparator )`	Sorts the children of the receiver node and of all descendants in preorder traversal. The comparator is passed two nodes `a` and `b`, and must return a value less than, equal to, or greater than zero if `a` is less than, equal to, or greater than `b`, respectively.

Function	Description
`node.each( function )`	Invokes the supplied function on the current node and each descendant node. Each node is passed as the argument to the function. (Also check the D3 Reference Documentation for `node.eachAfter()` and `node.eachBefore()`.)
`node.copy()`	Returns a deep copy of the subtree starting at the given node.

The functions `count()`, `sum()`, and `sort()` modify the tree. They must be called explicitly *before* any code that makes use of their results.

Table 9-2. Properties of D3 Node instances (node is a D3 Node instance)

Function	Description
`node.data`	The record from the original data set, corresponding to the current node, as object.
`node.depth`	Distance from the root node; the root node has depth zero.
`node.height`	Distance from the deepest accessible leaf node; leaf nodes have height zero.
`node.parent`	Parent node; `null` for the root node.
`node.children`	An array of child nodes; `undefined` for leaf nodes.
`node.value`	Placeholder for cumulative information computed by `count()` or `sum()` (see text).

Link and Node Diagrams for Trees

One way to represent a hierarchical data structure is as a *tree diagram*, showing the links and nodes explicitly. D3 provides two layouts for such diagrams: `d3.tree` and `d3.cluster`. Of these, `d3.tree` builds relatively compact trees, moving outward from the root node and placing each new generation of children at the same distance from the root. In contrast, `d3.cluster` builds trees in which all leaf nodes are displayed at the same depth (see Figure 9-1).

Both layouts offer the same set of configuration options (see Table 9-3). The size of the resulting graph can be configured in two ways: either by setting the maximum size of the entire graph using `size()` or by setting the minimal size of each node using `nodeSize()`. A minimal example can be found in Example 9-1. The use of the link generator in this example is entirely optional; you may instead want to use straight lines to represent the edges.

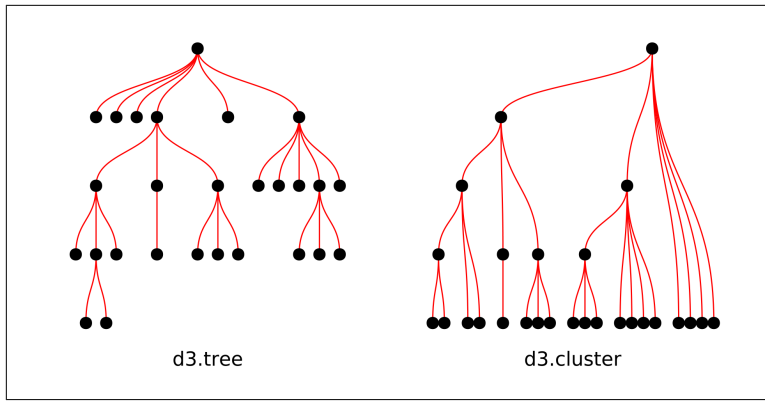


Figure 9-1. The same hierarchical data set, visualized using two different layouts

Table 9-3. Methods of the d3.tree and d3.cluster tree layouts (t is a tree layout instance)

Function	Description
`d3.tree()`	Returns a new `tree` layout instance. In the final arrangement, all nodes at the same depth will be at the same pixel distance from the root node.
`d3.cluster()`	Returns a new `cluster` layout instance. In the final arrangement, all leaf nodes will be at the same pixel distance from the root node.
`t( node )`	Takes the root node of a tree of D3 `Node` objects and calculates the pixel positions of the nodes below the node passed as the argument. Returns the input hierarchy, with the member variables `x` and `y` added to each node.
`t.size( [x, y] )`	Takes an array of two elements, interpreted as the width and height of the resulting figure, in pixels. Nodes will be arranged such that the entire tree graph fits into a rectangle of the indicated size.
`t.nodeSize( [x, y] )`	Takes an array of two elements, interpreted as the width and height of a single node, in pixels. In order to avoid overlapping, nodes will be spaced at least that far apart.

Example 9-1. Commands for the left side of Figure 9-1

```
function makeTree() {
    d3.json( "filesys.json" ).then( function(json) {
        var nodes = d3.hierarchy(json, d=>d.kids);                    ❶
        d3.tree().size( [250,225] )( nodes );                         ❷

        var g = d3.select( "#tree" ).append( "g" )                    ❸
```

```
            .attr( "transform", "translate(25, 25)" );

        var lnkMkr = d3.linkVertical().x( d=>d.x ).y( d=>d.y );    ❹
        g.selectAll( "path" ).data( nodes.links() ).enter()        ❺
            .append( "path" ).attr( "d", d=>lnkMkr(d) )
            .attr( "stroke", "red" ).attr( "fill", "none" );

        g.selectAll("circle").data( nodes.descendants() ).enter()  ❻
            .append("circle").attr( "r", 5 )
            .attr( "cx", d=>d.x ).attr( "cy", d=>d.y );
    } );
}
```

❶ Construct a tree of D3 `Node` objects from the input data. The supplied accessor function is invoked for each element in the input and returns an array of child nodes (stored in the input in the property `kids`).

❷ Invoke the `d3.tree()` layout mechanism on the tree of nodes and add position information to each node. The resulting coordinate values are expected to fit into a rectangle of the specified size. The layout operator modifies its input, hence it is not necessary to capture its return value.

❸ Append a `<g>` element as container for the tree and move it to the desired position.

❹ Create a link generator and configure its accessor functions. Given a data set, the link generator produces a command string for the `d` attribute of a `<path>` element (as described in Chapter 5). Here, we use a generator that produces lines that meet each node with a vertical tangent.

❺ Create a `<path>` element for every element returned by `node.links()` and invoke the link generator on it.

❻ Obtain an array of nodes using `node.descendants()` and mark each node by a circle.

Depending on the input data and the chosen layout algorithm, it may be necessary to *sort* the nodes before passing them to the layout. Sorting arranges the order in which a node's children are accessed when visiting that node. In Figure 9-1, this amounts to a left-to-right ordering of each node's children (and hence their

respective subtrees). The nodes of the tree on the left side of Figure 9-1 are arranged in the order of the input file, but for the tree on the right, the nodes are resorted by their height (distance from the leaf nodes) before invoking the `d3.cluster()` layout (the rest of the code is identical to Example 9-1):

```
var nodes = d3.hierarchy( json, d=>d.kids )
    .sort( (a,b) => b.height - a.height );
d3.cluster().size( [250,225] )( nodes );
```

It is important to realize that the `x` and `y` properties added by `d3.tree()` or `d3.cluster()` do not carry semantics—they are just coordinates in an arbitrary two-dimensional space. To create trees that grow horizontally, you interchange the coordinates when drawing the graph, like this:

```
d3.linkHorizontal().x( d=>d.y ).y( d=>d.x );
```

and equivalently for all other graphics commands; for example, you need to interchange width and height information, and so on. You can even produce a *radial* tree as shown on the left in Figure 9-2 by interpreting the coordinates as angle and radius (see Example 9-2).

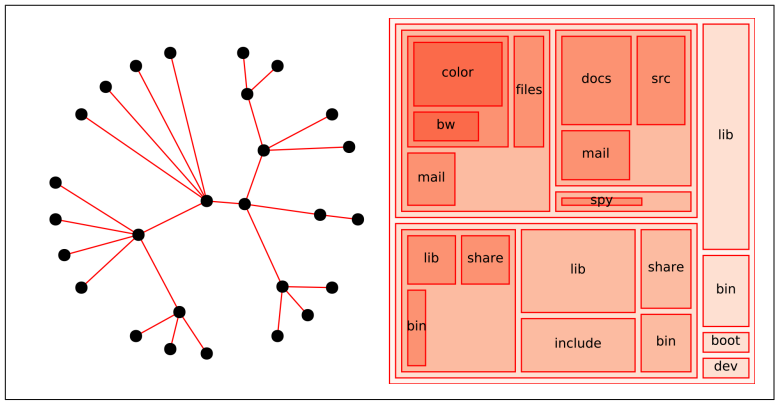


Figure 9-2. Left: The data set from Figure 9-1, shown in a radial layout (see Example 9-2). Right: a treemap diagram of the same data set (see Example 9-3).

Example 9-2. Commands for the left side of Figure 9-2

```
function makeRadial() {
    d3.json( "filesys.json" ).then( function(json) {
        var nodes = d3.cluster().size( [2*Math.PI, 125] )(
            d3.hierarchy( json, d=>d.kids )
```

❶

```
            .sort( (a,b)=>b.height-a.height )
        );

        var g = d3.select( "#radial" ).append( "g" )
            .attr( "transform", "translate(150, 150)" );                    ❷

        var h = function( r, phi ) { return  r*Math.sin(phi) }              ❸
        var v = function( r, phi ) { return -r*Math.cos(phi) }

        g.selectAll( "line" ).data( nodes.links() ).enter()
            .append( "line" ).attr( "stroke", "red" )
            .attr( "x1", d=>h(d.source.y, d.source.x) )                     ❹
            .attr( "y1", d=>v(d.source.y, d.source.x) )
            .attr( "x2", d=>h(d.target.y, d.target.x) )
            .attr( "y2", d=>v(d.target.y, d.target.x) );

        g.selectAll( "circle" ).data( nodes.descendants() ).enter()
            .append( "circle" ).attr( "r", 5 )
            .attr( "cx", d=>h(d.y, d.x) )
            .attr( "cy", d=>v(d.y, d.x) );
    } )
}
```

❶ For the radial tree, the range of the two coordinates is chosen suitably for polar coordinates.

❷ In this case, the origin of the resulting tree lies at the center of the graph, and the containing `<g>` element is placed accordingly.

❸ Define some local functions to handle polar coordinates.

❹ In this example, the edges are represented as straight lines using `<line>` elements. This makes it necessary to access the coordinates of both the `source` and `target` node in each link explicitly. (Link generators do this by default.)

Instead of using straight lines, a radial link generator could have been used for the edges:

```
g.selectAll("path").data( nodes.links() ).enter().append("path")
    .attr( "d", d3.linkRadial().angle(d=>d.x).radius(d=>d.y) )
    .attr( "stroke", "red" ).attr( "fill", "none" );
```

Area Graphs for Containment Hierarchies

Sometimes it is of interest to "roll up" the information contained in a hierarchy: a directory in a filesystem, for example, may contain a

certain number of files itself, but indirectly it also contains all the files in all of its subdirectories. The challenge is to visualize all *three* pieces of information simultaneously:

- The individual size of each component
- The cumulative size for each component and all of its children
- The parent/child hierarchy

One way to do so is with a *treemap* as shown on the right side of Figure 9-2. It uses *area* to indicate size (both individual and cumulative) and geometric *enclosure* to represent the hierarchy itself.

The D3 `Node` abstraction provides two useful member functions to help with such problems. The function `count()` returns the number of leaf nodes in the subtree below the receiver node (including the receiver itself, hence for leaf nodes `count()` returns 1). In contrast, the function `sum()` evaluates an accessor function for each node, which must return a nonnegative number. The `sum()` function then calculates the cumulative sum of these numbers for the subtree below the current node. Both `count()` and `sum()` assign their results to the member variable `value` on each node. The functions `count()` and `sum()` must be invoked explicitly before evaluating any function that requires the `value` property to be set on the nodes.

D3 includes several layouts to turn this information into graphical arrangements. Both the `d3.treemap()` and the `d3.partition()` layout use rectangles to represent nodes, whereas the `d3.pack()` layout uses circles. The way these layouts are invoked is quite similar; Example 9-3 uses the `d3.treemap()` layout for demonstration.

Example 9-3. Creating a treemap (see the right side of Figure 9-2)

```
function makeTreemap() {
    d3.json( "filesys.json" ).then( function(json) {
        var sc = d3.scaleOrdinal( d3.schemeReds[8] );

        var nodes = d3.hierarchy(json, d=>d.kids).sum(d=>d.size)  ❶
            .sort((a,b) => b.height-a.height || b.value-a.value); ❷

        d3.treemap().size( [300,300] ).padding(5)(nodes);         ❸

        var g = d3.select( "#treemap" ).append( "g" );

        g.selectAll( "rect" ).data( nodes.descendants() ).enter()
```

```
            .append( "rect" )
            .attr( "x", d=>d.x0 ).attr( "y", d=>d.y0 )
            .attr( "width", d=>d.x1-d.x0 )                          ❹
            .attr( "height", d=>d.y1-d.y0 )
            .attr( "fill", d=>sc(d.depth) ).attr( "stroke", "red" );

        g.selectAll( "text" ).data( nodes.leaves() ).enter()        ❺
            .append( "text" )
            .attr( "text-anchor", "middle" ).attr( "font-size", 10 )
            .attr( "x", d=>(d.x0+d.x1)/2 )
            .attr( "y", d=>(d.y0+d.y1)/2+2 )
            .text( d=>d.data.name );
    } );
}
```

❶ Create a tree of D3 `Node` objects from the input file, and evaluate the `sum()` function, treating the `size` member of each node as its value.

❷ Sort the nodes by descending height and then descending value (as recommended for treemaps and similar layouts).

❸ Invoke the layout mechanism, specifying some additional padding around each element.

❹ The `d3.treemap()` layout calculates *two corner positions* for each rectangle, not width and height as required by the `<rect>` element!

❺ Only the leaf nodes receive a text label.

Force-Based Particle Arrangements

If the number of elements that need to be arranged in a graph is large, or if the constraints they must fulfill are complicated, it may not be clear how to find an optimal arrangement—or even what one might look like! If this is the case, then an *iterative* relaxation scheme may be helpful: start with the elements in an arbitrary initial configuration, apply the constraints, and let the elements move in response to them. D3 includes a facility to simulate the behavior of a collection of elements, subject to constraints, in order to determine a satisfactory arrangement.

Simulation How-To

I will not describe the complete API here in detail. Instead, I will provide an overview and roadmap that should help to make sense of the reference documentation.

Simulation setup

A simulation operates on an array of nodes or particles.[2] A particle is simply an arbitrary, possibly empty, object. Unless they already exist, the simulation will create the following members: `x`, `y` (position), `vx`, `vy` (velocity), and `index` (numerical index as identifier). If a particle has members `fx`, `fy` (fixed position), then the particle will be treated as stationary: it will interact with the other particles, but at the end of each iteration step, its position will be reset to `fx`, `fy` and its velocity to zero. Particles can have any (or no) properties; a unique identifier will often be helpful (in particular for establishing links between particles in a network).

A simulation automatically begins to run "in the background" once it has been created, taking one simulation step per *animation frame* (see Chapter 4). It can be stopped with `stop()` and restarted with `restart()`. Once stopped, it can be advanced manually using `tick()`. An event handler can be registered with `on()` to be invoked after each step of the iteration (to create a smooth animation) or at its end (to display the final configuration). A simulation may take a while to complete (about five seconds when using the default settings).

A simulation does not return its results explicitly. Instead it updates the array of simulation particles with new positions and velocities at each simulation step.

Controlling convergence

The simulation is intended to create a "layout," hence it is supposed to converge to a final configuration and then stop. Convergence is mainly controlled through a parameter called *alpha*. By default,

2 In this context, the term "node" designates a generic node in a data structure, but does not refer to a DOM `Node` or to a D3 `Node` as described earlier in this chapter. To avoid confusion, I'll use the term *particle* exclusively—in contrast to the convention of the D3 Reference Documentation.

alpha is reduced by a constant factor at each step; if it falls below the threshold set with `alphaMin()`, the simulation stops. Most (not all) interactions multiply the changes to be applied at each time step by the current value of the alpha parameter, so that the incremental changes decrease as the simulation progresses.[3] The method `alphaDecay()` is used to control how quickly the alpha parameter is reduced; to let the simulation run forever, keep the alpha parameter constant. The default decay rate is such that the simulation will end after 300 simulation steps. There are no built-in provisions to stop the simulation if the incremental changes to the particle positions in each step become small; such functionality would have to be added externally.

To aid convergence (and to avoid spurious oscillation and instability), all particles are subject to ordinary friction (an interaction that is proportional in magnitude but opposite in direction to a particle's velocity). The friction coefficient can be set using `velocityDecay()`.

Constraints and interactions

D3 defines several predefined constraints or interactions acting on and between particles. The D3 documentation refers to all of them as "forces," but it is important to understand that these are not necessarily forces in the sense of Newton's Law $F = ma$. I will use the term *interaction* exclusively to avoid confusion.

Interactions are created as instances and passed to the simulation. Each instance can be configured separately. An interaction instance is added to the simulation using `force()`. This function takes an arbitrary tag name and an interaction instance. It is possible to add multiple instances of the same interaction type (with different configurations) to a single simulation.

The properties of individual *particles* are configured through *interaction* instances. For several interaction types it is possible to specify the strength of the interaction either as an overall constant or as a per-particle accessor function that can return a different value for each particle.

3 The effect of reducing alpha is similar, but not exactly equal to, reducing the length of the simulation time step.

Some interactions are implemented as (Newtonian) forces, others as (soft) constraints. For constraints, it is possible to apply them iteratively several times per overall simulation step in order to drive the system closer to a configuration satisfying the constraint.

Effects from different interaction instances are not combined into a single "total force" on each particle. Instead the effect of each interaction instance is applied separately to all particles before the effects of the next interaction instance are calculated.

Built-In Interactions

D3 includes several built-in interactions. You use the following factory functions to obtain an interaction instance, which in turn can be configured and passed to the simulation object.

`d3.forceCenter( x, y )`
: An overall adjustment (translation), made to the positions of all particles after each simulation step, to keep the center of mass of the system at the indicated position.

`d3.forceCollide( radius )`
: A soft-core repulsion between pairs of particles. It is linear in the amount of overlap between the two particles (as measured by the distance of their centers), and zero if they do not overlap. The particle radius can be set globally (for all particles) or by defining an accessor function to allow individual particles to have different radii. A global strength parameter can be set. The constraint can be iterated to reinforce the constraint.

`d3.forceLink( [ links ] )`
: A soft constraint between pairs of particles. If two particles form a "link," the constraint acts to keep the separation between these two particles at a specific distance. The constraint grows linearly with the deviation from the desired distance (Hooke's Law, harmonic oscillator interaction). The desired distance and the strength of the interaction can be set globally or as accessors for each link individually. Links are specified as an array of objects, each of which must have a `source`, a `target`, and an `index` property. The first two must identify particles from the running simulation, either as object references or via an arbitrary identifier. In the latter case, the `id()` function must be used to set an accessor function, which, when invoked on a simulation parti-

cle, returns the ID used in the link object (see Example 9-5 for an example).

`d3.forceManyBody()`
: An interaction between pairs of particles that is proportional to the inverse square of their distance (Coulomb's Law, gravitation). It can be either attractive or repulsive, according to the sign of the strength parameter (positive values lead to attraction, negative values to repulsion). The strength can be set globally or via a per-particle accessor.[4] The interaction range can be truncated at both limits (short-range and long-range). Furthermore, an approximation is used by default where more distant particles are first aggregated into clusters, and the interaction is calculated between those clusters, not between individual particles. This approximation is controlled by a parameter called `theta`.

`d3.forceX( x )`, `d3.forceY( y )`, `d3.forceRadial( r, x, y )`
: A soft constraint intended to anchor all particles along a one-dimensional, straight or circular line, by driving a single component of a particle's location toward an absolute value. The first two interactions have the effect of driving the *x* and *y* coordinates to the indicated value, respectively. The third one acts to minimize the particle's distance from the nearest point on a circle with the indicated radius and center (that is, it acts to fix the radial component of the particle's location while leaving its angle coordinate unchanged). The magnitude of the constraint is proportional to the projection of the distance onto the chosen coordinate axis. (In other words, the magnitude of the first interaction is proportional to `x - particle.x`.) The strength can be set globally or via a per-particle accessor.

Those are the available interaction types that can be introduced into a simulation using `force()`. In addition, per-particle fixed positions (via the `fx`, `fy` properties on the particle object) and the overall friction (via `velocityDecay()`) also influence the particles' behavior. Finally, it is possible to create custom interaction types that can be

4 The configuration will not settle into a stable configuration if some particles have a positive strength parameter while others have a negative one. Particles with negative strength values will try to get away from other particles, but the particles with positive values will follow them, leading to a nonphysical runaway situation.

used with the overall D3 simulation framework (but that's a serious undertaking).

Examples

After all this theory, two quick examples will demonstrate how simulations can be used.

Creating a network layout

Example 9-5 loads a list of particles and connections from a JSON file, some of which is shown in Example 9-4. A simulation is then run until completion to generate the arrangement of particles in Figure 9-3. The figure is created by the event listener that is triggered when the simulation comes to an end.

Example 9-4. Parts of a JSON file defining a network

```
{ "ps": [ { "id": "A" },
          { "id": "B" },
          { "id": "C" },
          ... ],
  "ln": [ { "source": "A", "target": "B" },
          { "source": "A", "target": "C" },
          { "source": "A", "target": "D" },
          ... ]
}
```

Example 9-5. Using a force-based simulation to arrange the nodes of a network (see Figure 9-3)

```
function makeNetwork() {
    d3.json( "network.json" ).then( res => {
        var svg = d3.select( "#net" )
        var scC = d3.scaleOrdinal( d3.schemePastel1 )

        d3.shuffle( res.ps ); d3.shuffle( res.ln );

        d3.forceSimulation( res.ps )
            .force("ct", d3.forceCenter( 300, 300 ) )
            .force("ln",
                   d3.forceLink( res.ln ).distance(40).id(d=>d.id) )
            .force("hc", d3.forceCollide(10) )
            .force("many", d3.forceManyBody() )
            .on( "end", function() {
                svg.selectAll( "line" ).data( res.ln ).enter()
                    .append( "line" ).attr( "stroke", "black" )
```

```
                    .attr( "x1", d=>d.source.x )
                    .attr( "y1", d=>d.source.y )
                    .attr( "x2", d=>d.target.x )
                    .attr( "y2", d=>d.target.y );

                svg.selectAll("circle").data(res.ps).enter()
                    .append("circle")
                    .attr( "r", 10 ).attr( "fill", (d,i) => scC(i) )
                    .attr( "cx", d=>d.x ).attr( "cy", d=>d.y )

                svg.selectAll("text").data(res.ps).enter()
                    .append("text")
                    .attr( "x", d=>d.x ).attr( "y", d=>d.y+4 )
                    .attr( "text-anchor", "middle" )
                    .attr( "font-size", 10 )
                    .text( d=>d.id );
            } )
    } );
}
```

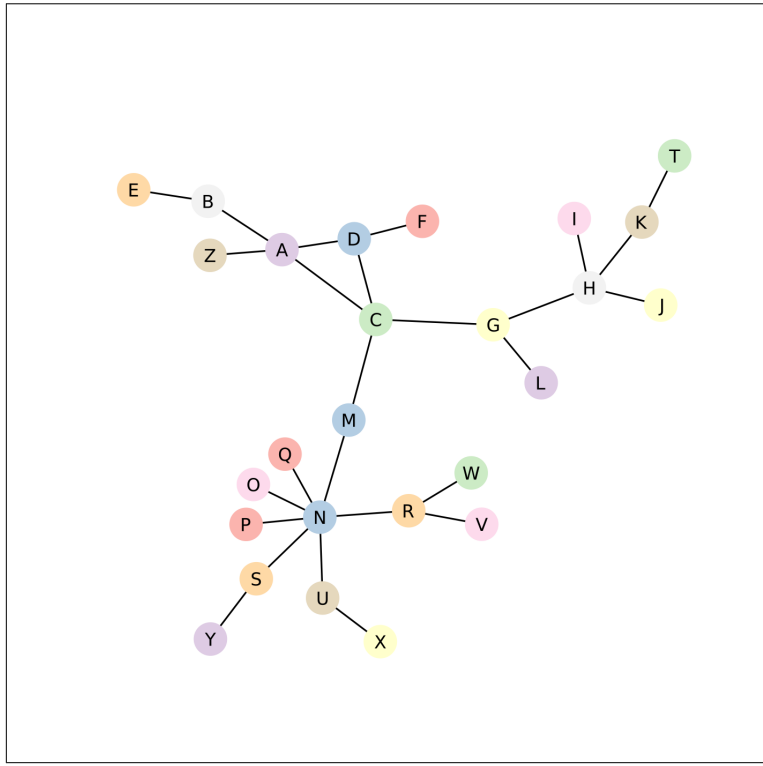


Figure 9-3. The nodes of a network, arranged through a force-based simulation (see Example 9-5)

The simulation uses four different interactions. The purpose of the link interaction and the soft-core repulsion is clear. The long-range, repulsive "many body" interaction ensures that the network is spread out and that the particles don't clump together in a tight cluster. The "centering" interaction finally fixes the *overall position* of the entire cluster within the SVG element. Without it, only the relative, but not the absolute, coordinates of the particles would be determined.

The simulation outcome is nondeterministic. If the final arrangement of the network seems unsatisfactory, then running the simulation again may produce a better result. In this case, the order of the particles and links should be shuffled to present the simulation with a different starting configuration (for example, using `d3.shuffle()`, see Chapter 10).

Animated particles

The simulation framework can be used to create animations of physical systems. In Example 9-6, two particles connected by a spring are bouncing off of each other. The initial configuration of the particles lets them spin around their common center of mass.

Example 9-6. Using a force-based simulation to create animated graphics

```
function makeSimul() {
    var ps = [ { x: 350, y: 300, vx: 0, vy: 1 },
               { x: 250, y: 300, vx: 0, vy: -1 } ];
    var ln = [ { index: 0, source: ps[0], target: ps[1] } ];

    var cs1 = d3.select( "#simul" ).select( "#c1" );
    var cs2 = d3.select( "#simul" ).select( "#c2" );

    var sim = d3.forceSimulation( ps )
        .alphaDecay( 0 ).alphaMin( -1 ).velocityDecay( 0 )
        .force("ln", d3.forceLink(ln).distance(50).strength(0.01))
        .on( "tick", function() {
            cs1.attr( "cx", ps[0].x ).attr( "cy", ps[0].y );
            cs2.attr( "cx", ps[1].x ).attr( "cy", ps[1].y );
        } );
}
```

For simplicity, Example 9-6 assumes that the SVG element already contains two `<circle>` elements that it can use. An event handler

has been installed that updates the position of the circles after each simulation step.

The simulation parameters have been chosen to let the simulation run without converging (no alpha decay and no friction). Nevertheless, you will notice that the oscillation is damped and disappears after a while. The reason is that the implementation of `d3.forceLink()` uses an anticipatory update algorithm, which is more stable but less physically realistic. To create a physically accurate simulation, you would have to write your own interaction (or modify the existing one).

CHAPTER 10

Utilities: Arrays, Statistics, and Timestamps

D3 includes some ancillary features that mostly augment JavaScript's built-in functionality. In this chapter, I'll introduce two topics because they are so common and commonly useful: operations on arrays, including basic descriptive statistics for strictly numerical arrays, and support for working with dates and timestamps.

Structural Array Manipulations

D3 includes some functions that change the structure of arrays (of arbitrary types). Table 10-1 summarizes some of the most useful of these functions and those that are used elsewhere in this book.

Table 10-1. D3 functions to create and manipulate JavaScript arrays

Function	Description
`d3.range(start, stop, step)`	Returns an array of uniformly spaced numbers, between `start` (inclusive) and `stop` (exclusive), obtained by repeatedly adding `step` to `start`. The step size need not be an integer and may be negative. If only a single argument is supplied, it is interpreted as `stop`; in this case `start` defaults to 0 and `step` defaults to 1. If two arguments are supplied, they are interpreted as `start` and `stop`; `step` again defaults to 1.

Function	Description
`d3.shuffle( array, low, high )`	Performs an in-place random shuffle on a subarray, bounded by array indices `low` (inclusive) and `high` (exclusive). If the bounds are omitted, the entire array is shuffled. Returns the array.
`d3.cross(a, b, reducer)`	Returns the Cartesian product of the arrays `a` and `b` as a *one-dimensional* array. The `reducer` function is invoked for each pair of input elements; its return value is entered into the cross product. The default reducer forms the two-element array of its inputs: `(u, v) => [u, v]`.
`d3.merge( [array] )`	Takes an array of arrays and concatenates their elements into a single array. Nested data structures are not followed.
`d3.pairs(array, reducer)`	Invokes the `reducer` function on each adjacent pair of elements and collects the reducer's return value into a one-dimensional array. The returned array has one item less than the input array. The default reducer forms the two-element array of its inputs: `(u, v) => [u, v]`. Returns an empty array if the input array has fewer than two elements.
`d3.transpose(matrix)`	Takes a two-dimensional array of arrays and returns its transpose as a two-dimensional array of arrays.
`d3.zip( array1, array2, … )`	Takes an arbitrary number of arrays. Returns an array of arrays, where the first array contains the first element from all arguments, the second array contains the second element, and so on. The returned array is truncated to the length of the shortest of the arguments. If only a single array is supplied, returns an array of single-argument arrays.

The table is not complete; it omits some functions to search an array efficiently (using binary search) and functions that are primarily used internally by D3 (for example, to generate tick marks). Check the D3 Reference Documentation for further details.

Descriptive Statistics for Numerical Arrays

Table 10-2 summarizes some functions that calculate basic descriptive statistics for a data set. These functions are primarily intended for *numerical* values. Undefined values (`null`, `undefined`, and `NaN`) are ignored; they are also not counted toward the number of elements in the array (for example, when calculating the mean).

Table 10-2. Methods to calculate basic descriptive statistics on arrays of numerical values. Every function can take an optional accessor function, which is passed the current array element.

Function	Description
`d3.min( array, accessor ),` `d3.max( array, accessor )`	Returns the smallest or largest array element, or `undefined` if the array is empty.
`d3.extent( array, accessor )`	Returns both the smallest and largest array element as a two-element array `[min, max]`. Returns `[undefined, undefined]` if the input is empty.
`d3.sum( array, accessor )`	Returns the sum of the array elements, or 0 if the array is empty.
`d3.mean( array, accessor )`	Returns the mean of the array elements, or `undefined` if the array is empty.
`d3.variance( array, accessor ),` `d3.deviation( array, accessor )`	Returns the sample variance s^2_{n-1} and its square root, respectively. Returns undefined if the input has fewer than two values.
`d3.median( array, accessor )`	Returns the median, or 0 if the input is empty. The array need not be sorted.[a]
`d3.quantile( array, p, accessor )`	Returns the *p*-quantile, where $0 \leq p \leq 1$. *The input array must be sorted.*[a]

[a] Median and quantiles are calculated using the R-7 method: *https://en.wikipedia.org/wiki/Quantile*.

Histograms

D3 includes a way to generate histograms of numerical data. The histogram facility is a layout: it returns an array of bins. Each bin is an array containing the original data points associated with this bin; its `length` property therefore gives the number of elements per bin. Each bin also exposes the properties `x0` and `x1`, which contain the lower and upper bound of the bin, respectively. (See Table 10-3 and the D3 Reference Documentation for further details.)

Table 10-3. Functions and methods for the creation of histograms (h is a histogram instance)

Function	Description
`d3.histogram()`	Returns a new histogram layout operator.
`h( array )`	Computes the histogram for the supplied array of values.
`h.value( accessor )`	Sets the value accessor. The accessor will be called for each element in the input array, being passed the element `d`, the index `i`, and the array itself as three arguments. The default accessor assumes that the input values are sortable; if this is not the case, the accessor should return a corresponding sortable value for each element of the data set.
`h.domain( [min, max] )`	Sets the minimum and maximum value to be considered when constructing the histogram; values strictly outside this interval will be ignored. The interval can be specified as an array or as an accessor function that will be invoked on the array of values. If the input data is not sortable, the interval should be specified on the corresponding sortable values.
`h.thresholds( count ),` `h.thresholds( [boundary] ),` `h.thresholds( fct )`	If the argument is an integer, it is interpreted as the desired number of equally sized bins to create. If it is an array, its elements are treated as bin boundaries; if there are *n* boundaries, the resulting histogram will have *n+1* bins. If the argument is a function, it is expected to generate an array of bin boundaries. The default is `d3.thresholdSturges`.
`d3.thresholdSturges()`	Computes the number of bins as $1 + \lceil \log_2 n \rceil$, where *n* is the number of data points.
`d3.thresholdScott()`	Computes the bin width as $3.5\,\sigma / \sqrt[3]{n}$, where σ is the sample standard deviation and *n* is the number of data points.
`d3.thresholdFreedmanDiaconis()`	Computes the bin width as $2\,\mathrm{IQR} / \sqrt[3]{n}$, where IQR is the sample inter-quartile range and *n* is the number of data points.

Figure 10-1 shows a typical histogram. The code in Example 10-1 uses a *band scale* for the horizontal positioning of the bins. A band scale maps a discrete set of values (bins) to a continuous range of numbers (pixels). It also exposes further useful information, such as each bin's calculated bandwidth (in pixels). The padding around each bin can be set (as a fraction of the bin width). The call to `round()` forces the bin boundaries to coincide with (integer) pixel

coordinates, thus avoiding anti-aliasing effects and leading to sharper edges.

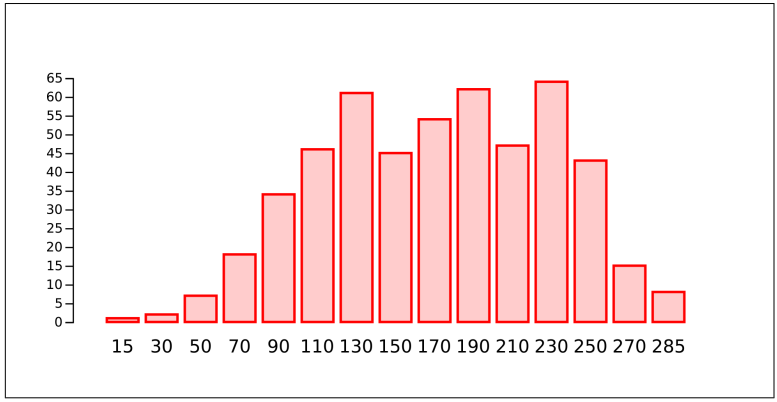


Figure 10-1. A histogram, created using the D3 histogram layout and a band scale

Example 10-1. Commands for Figure 10-1

```
function makeHisto() {
    d3.csv( "dense.csv" ).then( function( data ) {
        var histo = d3.histogram().value( d=>+d.A )( data );

        var scX = d3.scaleBand().padding( 0.2 ).round( true )
            .range( [15, 515] ).domain( histo );

        var scY = d3.scaleLinear().range( [200, 0] )
            .domain( [0, d3.max( histo, d=>d.length ) ] ).nice();

        var g = d3.select( "#histo" )
            .append( "g" ).attr( "transform", "translate( 40,50 )" )

        g.selectAll( "rect" ).data( histo ).enter()
            .append( "rect" ).attr( "width", scX.bandwidth() )
            .attr( "x", scX ).attr( "y", d=>scY(d.length) )
            .attr( "height", d => 200-scY(d.length) )
            .attr( "fill", "red" ).attr( "fill-opacity", 0.2 )
            .attr( "stroke", "red" ).attr( "stroke-width", 2 )

        g.selectAll( "text" ).data( histo ).enter().append( "text" )
            .attr( "text-anchor", "middle" )
            .attr( "font-family", "sans-serif" )
            .attr( "font-size", 14 )
            .attr( "x", d => scX(d)+0.5*scX.bandwidth() )
            .attr( "y", 225 )
            .text( d=>(d.x0+d.x1)/2 );
```

```
        g.append( "g" ).call( d3.axisLeft(scY) );
    } );
}
```

Working with Dates and Timestamps

D3 includes some functionality to perform arithmetic on dates. In particular, it provides various ways to generate collections of equally spaced time intervals: for example, to generate tick marks for coordinate axes and so on. D3 does not introduce its own date/time abstraction; instead, all pertinent functions are built on (and work with) the native JavaScript `Date` data type and therefore share its limitations (in particular with regard to high-precision date calculations). See the following sidebar for some additional notes and warnings.

The JavaScript Date Type

Internally, JavaScript `Date` represents time as *milliseconds* (not seconds) since the Unix epoch. Leap seconds are ignored.

To obtain a `Date` object, you must call the constructor using the `new` keyword; merely invoking the `Date()` function will produce a human-readable string representation:

```
var date   = new Date();    // a Date object
var string = Date();        // a human-readable string
var millis = Date.now();    // milliseconds since the epoch
```

The `Date` constructor can be called with different arguments (the month index is zero-based as usual; the string representation has to follow ISO 8601 format):

```
new Date();
new Date( millis );
new Date( isostring );
new Date( year,monthIndex,day,hour,minute,second,millis );
```

It is possible to *subtract* a date or number from another date: the dates will be converted to milliseconds implicitly. When attempting to *add* a number to a date, however, both arguments will instead be converted to strings that are then concatenated. Use the unary prefix `+` to coerce the `Date` to a number, then use the constructor to obtain a new `Date` object from the result, in milliseconds. Keep in

mind that all numeric arguments have to be expressed in *milliseconds*, not seconds:

```
var now = new Date();
var then = now - 1000;
var later = new Date( +now + 10000 );
```

Nonequality comparison works as expected, but equality comparison requires particular care because neither the == nor the === operator will convert its arguments to numbers. This results in a comparison of *object identities* instead of *value equality*. To ensure value comparison, force both arguments to numbers:

```
if( later > now ) { ... } ;                    // true

var now2 = new Date( +now );
if( now == now2 ) { ... }                      // false
if( +now == +now2 ) { ... }                    // true
```

Most difficulties in performing calculations on date/time information stem from the way the continuous time dimension is divided into various intervals (such as hours, days, and months), which are not all of the same length (months) and do not necessarily align with each other (weeks and months). D3 deals with this problem in the following way:

- At the beginning of a calculation, the desired time interval (that is, the granularity of the current calculation) must be chosen and remains fixed for the entire calculation: you first choose whether you want to consider days, weeks, months, and so on.
- Timestamps are generally *truncated* to the chosen interval. (A timestamp may be truncated to midnight of the current day or the first day of the current month if the chosen interval is the day or the month, respectively.)
- The calculation then involves the addition and subtraction of multiples of the number of milliseconds corresponding to the chosen time interval.

Here are some typical examples (see Table 10-4 for a complete list of functions). Notice the locale-dependent result in the last line:

```
var now         = new Date();                          ❶
var nextMonth   = d3.timeMonth.offset( now, 1 );       ❷
var daysBetween = d3.timeDay.count( now, nextMonth );  ❸
var weekStarts  = d3.timeWeek.range( now, nextMonth ); ❹
```

❶ Create a baseline for the current set of calculations.

❷ The moment one month later. As opposed to other time-handling functions, `offset()` does *not* truncate its argument, hence the resulting timestamp preserves the current time of day.

❸ The number of days between today and the corresponding day next month. The result depends on the number of days in the current month.

❹ The variable `weekStarts` contains an array with four (or five, depending on the value of `now`) timestamps, each marking the *beginning* of a week (considered in the US to be Sunday—if you choose a different locale, the results of this calculation may be different).

Table 10-4. Calculations on time intervals (itvl is an interval instance). All functions can accept a number of milliseconds in place of a Date instance.

Function	Return type	Description
`itvl( date )`	`Date`	Equivalent to `itvl.floor( date )`.
`itvl.floor( date )`	`Date`	Returns a new `Date` equal to the latest interval boundary before or equal to the argument.
`itvl.ceil( date )`	`Date`	Returns a new `Date` equal to the earliest interval boundary after or equal to the argument.
`itvl.round( date )`	`Date`	Returns a new `Date` equal to the closest interval boundary to the argument.
`itvl.offset( date, n )`	`Date`	Returns a new `Date` equal to the time that is `n` intervals after `date` (if `n` is positive) or before `date` (if `n` is negative). The `date` is not truncated. If `n` is not an integer, it is replaced by `Math.floor(n)`. If `n` is omitted, it defaults to `1`.
`itvl.count( date1, date2 )`	Number	Returns the number of interval boundaries that are strictly later than `date1` and earlier or equal to `date2`.
`itvl.range( start, stop, step )`	`[ Date ]`	Returns an array of `Date` objects, representing every interval boundary equal or later than `start` and strictly earlier than `stop`. If `step` is specified, then only every `step` boundary will be included. If `step` is not an integer, it is replaced by `Math.floor(n)`.

Function	Return type	Description
`itvl.filter( fct )`	Interval	Returns a new interval that is a filtered subset of the receiver. The supplied function will be supplied a `Date` and should return `true` only if that date is to be retained in the new interval.
`itvl.every( step )`	Interval	A filtered interval retaining only every `step` of the receiver interval. For example, `d3.timeMinute.every(15)` returns an interval representing quarters of the hour, starting on the full hour.

D3 defines a set of built-in intervals; see Table 10-5. In addition to the intervals in the table, D3 also defines intervals for weeks starting on any day of the week (like `d3.timeMonday` for weeks starting on Monday, and so on). You may also define your own custom intervals using the function `d3.timeInterval()`.

Table 10-5. Built-in intervals

Local time	UTC time
`d3.timeMillisecond`	`d3.utcMillisecond`
`d3.timeSecond`	`d3.utcSecond`
`d3.timeMinute`	`d3.utcMinute`
`d3.timeHour`	`d3.utcHour`
`d3.timeDay`	`d3.utcDay`
`d3.timeWeek`	`d3.utcWeek`
`d3.timeMonth`	`d3.utcMonth`
`d3.timeYear`	`d3.utcYear`

Parsing and Formatting Timestamps

Formatting timestamps (that is, instances of the JavaScript `Date` object) follows the same workflow that is used for numbers (see Chapter 6):

1. Obtain a locale object (or use the current default locale—see Table 10-6).
2. Given a locale object, instantiate a formatter by supplying a format specification.

3. Invoke the formatter with a `Date` instance as the argument to obtain a formatted string.

Table 10-6. Methods for obtaining a locale object for timestamps

Function	Description
`d3.timeFormatLocale( def )`	Takes a locale definition and returns a locale object suitable for timestamp conversions.
`d3.timeFormatDefaultLocale( def )`	Takes a locale definition and returns a locale object, but also sets the default locale for timestamp conversions.

In addition to functions that *format* a `Date` object into a string (see Table 10-7), there are also functions that *parse* a string according to a format specification and return an equivalent `Date` object (see Table 10-8). This is useful if you have a timestamp as a string (if you have the individual components instead, use `new Date( ... )`). The parser is unable to parse its input and returns `null` if the input string does not match the specified format *exactly*. Different versions of formatters and parsers treat their input as local or UTC time, respectively.

Table 10-7. Methods for formatting timestamps (loc is a locale instance)

Function	Description
`d3.timeFormat( fmt )`	Returns a timestamp formatter instance for the current default locale using the format specification in the string `fmt`. The returned formatter will interpret its argument as local time.
`d3.utcFormat( fmt )`	Returns a timestamp formatter instance for the current default locale using the format specification in the string `fmt`. The returned formatter will interpret its argument as UTC time.
`loc.format( fmt )`	Returns a timestamp formatter instance for the receiver locale using the format specification in the string `fmt`. The returned formatter will interpret its argument as local time.
`loc.utcFormat( fmt )`	Returns a timestamp formatter instance for the receiver locale using the format specification in the string `fmt`. The returned formatter will interpret its argument as UTC time.
`d3.isoFormat( date )`	A formatter instance for the ISO 8601 format `%Y-%m-%dT%H:%M:%S.%LZ`. Takes a date object and returns a formatted string.

Some examples of functions from Tables 10-7 and 10-8:

```
var now = new Date();
console.log( d3.timeFormat( "%a %B" )( now ) );
```

```
console.log( d3.isoFormat( now ) );

var then = d3.timeParse( "%Y-%m-%d" )( "2018-10-25" );
```

Also see Example 7-3 for an example.

Table 10-8. Methods for parsing timestamps (loc is a locale instance)

Function	Description
`d3.timeParse( fmt )`	Returns a timestamp parser instance for the current default locale using the format specification in the string `fmt`. The returned formatter will interpret its argument as local time.
`d3.utcParse( fmt )`	Returns a timestamp parser instance for the current default locale using the format specification in the string `fmt`. The returned formatter will interpret its argument as UTC time.
`loc.parse( fmt )`	Returns a timestamp parser instance for the receiver locale using the format specification in the string `fmt`. The returned formatter will interpret its argument as local time.
`loc.utcParse( fmt )`	Returns a timestamp parser instance for the receiver locale using the format specification in the string `fmt`. The returned formatter will interpret its argument as UTC time.
`d3.isoParse( string )`	A parser instance for the ISO 8601 format `%Y-%m-%dT%H:%M:%S.%LZ`. Takes a string and returns a `Date` instance.

The syntax for the format specifier is based on the `strftime()` and `strptime()` family of functions in the standard C library. See Table 10-9 (replicated here from the D3 Reference Documentation for convenience—check there for additional details).

Table 10-9. Conversion specifiers for time and date information

	Description
%a	Abbreviated weekday name [a]
%A	Full weekday name [a]
%b	Abbreviated month name [a]
%B	Full month name [a]
%c	The locale's date and time, such as %x, %X [a]
%d	Zero-padded day of the month as a decimal number [01,31]
%e	Space-padded day of the month as a decimal number [1,31]; equivalent to %_d
%f	Microseconds as a decimal number [000000, 999999]
%H	Hour (24-hour clock) as a decimal number [00,23]
%I	Hour (12-hour clock) as a decimal number [01,12]
%j	Day of the year as a decimal number [001,366]

	Description
%m	Month as a decimal number [01,12]
%M	Minute as a decimal number [00,59]
%L	Milliseconds as a decimal number [000, 999]
%p	Either AM or PM [a]
%Q	Milliseconds since Unix epoch
%s	Seconds since Unix epoch
%S	Second as a decimal number [00,61]
%u	Monday-based (ISO 8601) weekday as a decimal number [1,7]
%U	Sunday-based week of the year as a decimal number [00,53]
%V	Monday-based (ISO 8601) week of the year as a decimal number [01, 53]
%w	Sunday-based weekday as a decimal number [0,6]
%W	Monday-based week of the year as a decimal number [00,53]
%x	The locale's date, such as `%-m/%-d/%Y` [a]
%X	The locale's time, such as `%-I:%M:%S %p` [a]
%y	Year without century as a decimal number [00,99]
%Y	Year with century as a decimal number
%Z	Time zone offset, such as `-0700`, `-07:00`, `-07`, or `Z`
%%	The literal percent sign %

[a] Output may be affected by choice of locale.

APPENDIX A

Setup, Tools, Resources

Setup

To work with D3, you need to run a web server, either locally or hosted, to serve pages, JavaScript files, and other resources (such as data files). In principle, it is possible to load a local page using the *file:* protocol and any JavaScript files referenced by it. But browsers may prevent your JavaScript code from loading other resources, such as data files, in this way, depending on the browser's cross-origin resource sharing (CORS) policy. Browsers are inconsistent in this regard; it is probably best to sidestep the issue by always using a web server when working with D3.

Setting up a web server need not be a challenge: several minimal web servers can be run without further configuration from the command line,[1] and many programming languages include ready-to-use web server modules as well. The D3 website recommends *http-server*, which is a Node.js package. If you have the Node runtime and the *npm* package manager installed, you can install and run a web server using:[2]

```
npm install -g http-server
http-server ./project -p 8080
```

1 For an extensive list, see *https://gist.github.com/willurd/5720255*.

2 If you encounter permission errors, these instructions may be helpful: *http://bit.ly/2XHjY4K*. Debian-based systems may also need to install the `nodejs-legacy` package.

Because they are part of its standard distribution, Python's web server modules are ubiquitous, but can be quite slow, even for development work (the `-d` argument requires Python 3.7 or later):

```
python -m http.server -d ./project 8080  # Python 3
python -m SimpleHTTPServer 8080          # Python 2: current dir
```

The *busybox* set of tools should be installed on all Debian-derived Linux distributions by default. Its built-in web server works well and is very fast:

```
busybox httpd -h ./project -p 8080
```

In all of these examples (except Python 2), the server will be serving files from the *project/* directory below the current directory, and listen on port 8080. The files in the server directory will be available at *http://localhost:8080*.

To include D3 in your project, download the current version of the library from *https://d3js.org*, unpack the archive, and then place the library file into your project directory. Two functionally equivalent versions exist: the full, human-readable version *d3.js* and the "minified" version *d3.min.js*, which reduces the download size to less than half.

Here is a minimal HTML document that follows the current recommendations for HTML5 document structure and that references both the D3 library and a JavaScript file called *script.js* (it also includes an empty SVG element to show the syntax):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Document Title</title>

  <script src="d3.js"></script>
  <script src="script.js"></script>
</head>

<body>
  <svg id="fig1" width="500" height="300" />
</body>
</html>
```

Alternatively, you can reference D3 directly at the project website or another public repository:

```
<script src="https://d3js.org/d3.v5.min.js"></script>
```

To begin executing JavaScript code once the page has loaded completely, register a JavaScript function in your code as event handler for the `load` event. The recommended way to do this is to include the following line in your JavaScript file:

```
window.addEventListener( "load", main )
```

This will invoke the JavaScript function called `main()` once the page has finished loading. (The function name is arbitrary, of course.) You can repeat this line to call different functions when the page has loaded.

Tools

SVG files are XML files and therefore can be manipulated by the usual tools for handling XML. (This includes your favorite text editor.)

Inkscape is a well-established editor for SVG files, but it is complex. Several other editors for vector graphics can handle SVG, but usually do not allow manipulating it on the DOM tree level.

Several command-line tools exist to convert SVG into other formats, such as PDF or PNG, among them *rsvg-convert* and *svgcairo-py3*. I had the best results with *svgcairo* (this is how most of the figures in this book were prepared).[3]

I have found that browsers often offer the best compliance, in particular when it comes to newer or more advanced features of the SVG spec. One way to take advantage of this is to run a browser in *headless* mode. In headless mode, the browser executes all commands (including all SVG directives and all JavaScript code) as usual, but does not render the result to the screen. Instead it is generally possible to save a "screenshot" of the fully rendered page to file—this may be the best way to convert complicated SVG images that are not handled well by other tools. Headless mode is usually enabled through a command-line flag (`-headless` for Firefox, `--headless` for Chrome). Check your browser's documentation for details.

3 Be aware that *svgcairo* requires *csstiny* and *cssselect* to handle style sheet information. If these two modules are not present, *svgcairo* will continue to work, but will *silently* ignore all style information.

An interesting tool is SVG Crowbar (*https://nytimes.github.io/svg-crowbar*). It is a bookmarklet that helps to extract and download SVG documents from web pages (for Chrome only).[4]

Resources

The principal source of information regarding D3 is the project homepage at *https://d3js.org*. Here you will find an extensive gallery of D3 example applications, links to further documentation, and information regarding D3 development.

The D3 Reference Documentation is very clear and detailed. Check there for up-to-date information and for additional details not mentioned in this book.

Books

- *Interactive Data Visualization for the Web* by Scott Murray, (O'Reilly, Second Edition) is a popular introduction to D3, targeted at readers who are less technically experienced.
- *D3.js in Action* by Elijah Meeks (Manning Publications, Second Edition) is another popular introduction to D3, which places D3 in the context of a general data visualization workflow.

Websites, Example Collections, and Galleries

Visit the following websites for examples, advice, and further information regarding D3:

- *https://github.com/d3/d3/wiki/Gallery*
- *https://observablehq.com/*
- *https://bl.ocks.org/*
- *https://blockbuilder.org/*

4 I thank Jane Pong for this recommendation.

APPENDIX B

An SVG Survival Kit

Introduction

The Scalable Vector Graphics (SVG) format is a vector image format for creating two-dimensional graphics, with support for interactivity and animation. SVG files are XML-based text files that can be manually edited and (at least in principle) searched. SVG produces a DOM tree (not a "flat," canvas-like image).

SVG images can be standalone documents (which is rare), or included in a web page. Contemporary browsers can handle `<svg>...</svg>` elements included in HTML documents. SVG files can also be included like other images: `<img src="file.svg" />`. In this case, it is mandatory that the SVG file declares the appropriate namespaces:

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg">
```

A W3C SVG Working Group was formed in 1998, a first standard (SVG 1.0) was released in 2001. Work is in progress for an all-new version (SVG 2) that will integrate SVG with HTML5. A draft standard for SVG 2 became a W3C Candidate Recommendation in 2016.

General Overview

SVG defines a number of renderable *graphics elements*, such as basic shapes, text elements, and lines and curves. The appearance of such

elements can be controlled through *presentational attributes*, which control size, position, color, and so on.

SVG further defines several *structural elements* that are used to organize the information within an SVG document. Using structural elements, it is possible to group other elements together into complex units, which can then be reused or transformed together.

SVG makes it possible to apply *transformations* to translate, rotate, or stretch elements. A transformation is specified for an individual element or for a group of elements through the `transform` attribute.

In general, SVG elements are *rendered* in document order, with the consequence that earlier elements appear to be visually "behind" later elements in the document. Elements that occur later in the document will occlude elements that are mentioned earlier.

SVG uses *graphical coordinates*, with the origin in the top left corner and the vertical axis running *downward*. Positions and sizes are measured in dimensionless *user coordinates*. The absolute size of the graph (and the elements it contains) is determined by the rendering device.

Finally, SVG documents can receive *user events* and invoke appropriate *event handlers*. Documents can change in response to events, thus providing a form of interactive graphical user interface (GUI).

Shapes

SVG provides explicit tags for a set of predefined simple shapes. Each shape has a set of specific attributes that control size and position. All shapes accept the usual presentational attributes to modify the shape's appearance. (See Table B-1. Not included are `<polyline>` and `<polygon>`, because typically the `<path>` element is used for such shapes; see the next section.)

Table B-1. SVG shapes

Tag	Specific attributes	Description
`<rect>`	`x`, `y`	Coordinates of upper-left corner
	`width`, `height`	Width and height
	`rx`, `ry`	Horizontal and vertical corner radius
`<circle>`	`cx`, `cy`	Center Coordinates
	`r`	Radius

Tag	Specific attributes	Description
<ellipse>	cx, cy	Center coordinates
	rx, ry	The horizontal and vertical radius
<line>	x1, y1	Starting point coordinates
	x2, y2	End point coordinates

Path

The <path> element is the most low-level drawing command in SVG. It can draw arbitrary shapes using a command language implementing a form of turtle graphics. The <path> element has only a single specific attribute (in addition to the usual representational attributes, of course):

d

The value of the d attribute consists of a string of whitespace-separated commands and coordinates. (See Table B-2.) A command is specified through a single letter. All commands come in two forms: as an uppercase letter or a lowercase letter. An uppercase letter indicates that the following set of coordinates are *absolute* coordinates, whereas a lowercase letter indicates that the following set of coordinates is to be interpreted as *relative* coordinates. Coordinates in the <path> element are always unitless. Negative coordinates are legal.

It is customary to let path specifications begin with an M command to place the pen at a well-defined position.

Commands exist to draw straight lines, Bézier curves, and ellipsoidal arcs. In particular the latter ones require a large number of parameters; see the SVG reference for complete details. In practice, command strings are commonly generated automatically, although it is also possible to write them explicitly. Two examples:

```
<path d="M40 50 L70 60 L70 40 Z" />
```

```
<path d="M40 100 L70 110 Q80 100 70 90 Z"
      fill="none" stroke="black" />
```

Table B-2. Commands for the d attribute of the <path> element

Command letter	Coordinates	Description
M, m	x y, dx dy	Move to (invisible—no line drawn).
L, l	x y, dx dy	Line to
V, v	y, dy	Vertical line
H, h	x, dx	Horizontal line
Z, z		Line to the first point; forming a proper (line) miter join with it. (This command can occur in the middle of the command string, not just at the end.)
Q, q	cx cy x y	Quadratic Bézier curve, from the current position to x, y, with control point at cx, cy.
T, t	multiple points	Quadratic Bézier curve through multiple control points.
C, c	cx1 cy1 cx2 cy2 x y	Cubic Bézier curve, from the current position to x, y, with control points at cx1, cy1 and cx2, cy2.
S, s	multiple points	Cubic Bézier curve through multiple control points.
A, a	rx ry phi arg sweep x y	Ellipsoidal arc from the current position (the starting point of the arc) to x, y (the end point). The parameters rx, ry, and phi determine the size of the ellipsis and its rotation relative to the coordinate system. The remaining parameters are binary flags that fix one out of the four possible solutions to this algebraic problem uniquely.

Text

Text can be included in an SVG image using the <text> element. The <text> element may include <tspan> elements, which allow the styling or positioning of pieces of text relative to the surrounding text. The <textPath> element can be used to render text along a path defined by a <path> element.

With the exception of text-anchor, which is only applicable to <text> elements, the attributes in Table B-3 apply to both <text> and <tspan> elements.

Table B-3. Attributes of the <text and <tspan> elements

Attribute	Description
x, y	Absolute position
dx, dy	Position relative to default text position (*not* relative to the parent element)
rotate	Rotation angle for each glyph, clockwise in degrees

Attribute	Description
`text-anchor`	Controls the alignment of the text relative to its origin. Legal values are `start`, `middle`, and `end`. (For scripts that are inherently left-to-right, this is equivalent to left alignment, centered, and right alignment, respectively.)

Text elements behave exactly like other graphical elements when it comes to color: the color of each glyph's body is controlled through the `fill` attribute, and the `stroke` attribute controls the color of an outline around the glyph. Specifying a separate stroke color only makes sense for sufficiently large font size; for most ordinary font sizes you will want to use `stroke="none"`.

The `<text>` element does not include a mechanism for line breaks: multiline text must be constructed from individual `<text>` elements.

Further mechanisms exist to control the rendering of each glyph in a text individually and to select fonts and font properties. Check the SVG reference for details.

Presentational Attributes

SVG defines a large number of presentational attributes; Table B-4 lists the most commonly used ones. Additional attributes exist to control the treatment of line ends and line joins (miter joins).

The most frequently used presentational attributes are probably `stroke` and `fill`, which control the color that is used to draw the the outline and interior of shapes and other elements. Because the default values are not attractive, it is generally necessary to set these two attributes explicitly. Two issues frequently cause confusion:

- SVG does not have a `color` attribute, instead you must use `fill` and `stroke`.
- The default value for `fill` is `black`; the default value for `stroke` is `none`. (This means that setting `fill="none"` on an element without also updating its `stroke` attribute makes it invisible!)

Table B-4. Common presentational attributes

Attribute	Description
`stroke`	The color of an element's outline. Defaults to `none`.
`stroke-width`	The width of an element's outline. Defaults to `1`.

Attribute	Description
`stroke-opacity`	The opacity of an element's outline, as a floating point number between 0 (fully transparent) and 1 (fully opaque). Defaults to `1`.
`stroke-dasharray`	Controls the pattern of dashes and gaps used to draw an element's outline.
`fill`	The color of an element's interior. Defaults to `black`.
`fill-opacity`	The opacity of an element's interior, as a floating point number between 0 (fully transparent) and 1 (fully opaque). Defaults to `1`.
`font-family`	The font family to render a text element, such as: `Times, Times New Roman, Georgia, serif`, or `Verdana, Arial, Helvetica, sans-serif`, or `Lucida Console, Courier, monospace`.[a]
`font-size`	The font size for rendering a text element. The size can be given in user coordinates (without explicit units), in points, as a percentage, in pixels, or `"ems"`.[a]
`font-style`	The font style to render a text element. Legal values are `normal`, `italic`, and `oblique`.[a]
`font-weight`	The font weight to render a text element. Legal values are `normal`, `bold`, `bolder`, `lighter`, `100`, `200`, `300`, `400` (the same as `normal`), `500`, `600`, `700` (the same as `bold`), `800`, `900`.[a]
`cursor`	The shape of the cursor that is displayed when the mouse pointer is over an element. Legal values include `auto` (the default), `default`, `cross hair`, `none`. Further predefined shapes exist to indicate drag-and-drop or resize actions. It is also possible to load and use an image.
`opacity`	The opacity of an element or a group of elements (when applied to the SVG `<g>` element, for instance), as a floating point number between 0 (fully transparent) and 1 (fully opaque). Defaults to `1`.
`display`	Controls the rendering of an element. When set to `none`, the element and its children will be removed from the rendering tree. They will not be rendered and therefore will be invisible; they will also not be able to receive events. If applied to a `<tspan>` element, the element is ignored for the purposes of text layout. The `display` property is not inherited by children and therefore has no effect when applied to a container element.
`visibility`	Controls the visibility of an element. When set to `visible`, the element is displayed. When set to `hidden` or `collapse`, the element is invisible, but may still receive events and (if applied to a `<tspan>` element) will still take up room during text layout. The `visibility` property is inherited.

[a] The `font-` attributes are modeled after the corresponding CSS properties. Check a CSS reference for further details.

Color

Colors are specified using CSS3 color syntax. The standard allows for a great variety of color formats; Table B-5 shows some of them.

(See MDN CSS <color> for more detail and the complete list of named colors.)

Table B-5. Different ways to specify color

Attribute	Description
`colorname`	One of the 148 predefined colornames, such as `red`. (The names correspond to only 140 distinct colors, because 8 colors have aliases.) Named colors do not allow transparency.
#RRGGBB	An RGB value as hexadecimal string, where each color channel is given as a hexadecimal number between `00` and `FF` (not case sensitive).
#RRGGBBAA	As before, but including transparency information: `00` fully transparent, `FF` fully opaque.[a]
`rgb(r, g, b)`	An RGB value using function call notation. The arguments must either *all* be integers between 0 and 255 (inclusive) or *all* be percentages between `0%` and `100%`. The percentage sign is mandatory when using percentages. Examples: `rgb( 255, 0, 0 )`, `rgb( 100%, 0%, 0% )`.
`rgba(r, g, b, a)`	As before, but including transparency information. The opacity parameter `a` can be a floating-point number between 0 and 1 *or* a percentage between `0%` and `100%`.[a] Examples: `rgba( 255, 0, 0, 50% )`, `rgba( 100%, 0%, 0%, 0.5 )`.
`hsl(h, s, l)`	An HSL (hue, saturation, lightness) value. The `h` argument should be an integer and is interpreted as an angle between 0 and 360 degrees (0: red, 120: green, 240: blue). Negative angles and angles greater than 360 are legal. The `s` and `l` arguments *must* be percentages between `0%` and `100%`; the percent sign is mandatory. Maximally bright colors correspond to a lightness value of `50%`; `0%` is black, `100%` is white. Example: `hsl( 0, 100%, 50% )`.
`hsla(h, s, l, a)`	As before, but including transparency information. The opacity parameter `a` can be a floating-point number between 0 and 1 or a percentage between `0%` and `100%`.[a] Example: `hsla( 0, 100%, 100%, 0.5 )`.

[a] A zero in the alpha channel always expresses a fully transparent color.

In addition, the keyword `none` is a legal value for `stroke` and `fill`. It signifies that no color will be applied.

Transformations

SVG elements can be translated, rotated, stretched, and sheared using the `transform` attribute on an element.

- In conjunction with the `<g>` element, transformations allow moving complex groups of elements to their final position

(without having to update the positional attributes in each element).

- Transformations enable visual effects that are not directly supported by the existing graphical elements (such as an ellipse with tilted axes or diagonally oriented text).

Transforms are applied to elements using the `transform` attribute. Table B-6 lists the possible values for this attribute. The `transform` attribute can contain a sequence of transformations, which will be applied right-to-left (mathematical notation)! Some examples:

```
<ellipse cx="0" cy="0" rx="10" ry="20" transform="rotate(30)" />
<text x="0" y="0"
      transform="translate(100,100) rotate(15)">Hello</text>
```

Table B-6. SVG transforms

Operation	Description
`translate(dx, dy)`	Translates the current element by `dx` and `dy`.
`rotate(phi, x, y)`	Rotates the current element in counterclockwise direction by the angle `phi`, given in degrees, around the point with coordinates `x` and `y`. If the coordinates are omitted, the element is rotated around the graph's origin.
`scale(fx, fy)`	Scales the current element in horizontal and vertical direction by the supplied scale factors. The scale factors should be floating-point numbers. If `fy` is omitted, `fx` is used for both directions.
`skewX(phi)`, `skewY(phi)`	Apply shear transformations along the horizontal or vertical axis.
`matrix(a, b, c, d, e, f)`	Apply a general affine transformation (see notes).

Reflections can be expressed as scale transformations with negative factors: `scale(-1,1)` is a reflection about the vertical axis, `scale(1,-1)` is a reflection about the horizontal one, and `scale(-1,-1)` is a point reflection through the origin.

The general affine transformation in two dimensions can be expressed in matrix notation as follows:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

All transformations in Table B-6 can be expressed in terms of this transformation:

```
translate(u,v) = matrix(1, 0, 0, 1, u, v)
scale(g,h)     = matrix(g, 0, 0, h, 0, 0)
rotate(q)      = matrix(cos(q), sin(q), -sin(q), cos(q), 0, 0 )
rotate(q,u,v)  = translate(u,v) rotate(q) translate(-u,-v)
skewX(q)       = matrix(1, 0, tan(q), 1, 0, 0)
skewY(q)       = matrix(1, tan(q), 0, 1, 0, 0)
```

The *origin* for SVG transformations (important for rotations and scale operations) is the origin of the SVG itself; by default it is located in the upper-left corner. However, when the `transform` attribute is used on the *outermost* `<svg>` element, it is interpreted as a CSS (not SVG) transform, and therefore takes the *center* of the element as origin. For more information about the importance of the origin and for practical advice when using SVG transformations, see "SVG Transformations" on page 90.

Structural Elements and Document Organization

SVG defines a number of structural elements. These elements are not rendered themselves, but only serve to organize other elements within the document. (See Table B-7.)

Of particular importance is the `<g>` (group) element. The `<g>` is a container element that acts as a common parent for its children. It combines its children into a composite element that can henceforth be treated as a unit. (The function of the `<g>` element is therefore comparable to the "group/ungroup" functionality familiar from many drawing programs.) Presentational attributes set on the `<g>` element are inherited by its children. Transformations applied to the `<g>` element are applied to all of its children. This makes it possible to move a set of graphical elements to a different position as a unit (without having to update the position of each element individually). When rotated, the content of the `<g>` element is rotated rigidly about a common center.

Table B-7. SVG structural elements

Element	Description
`<svg>`	The `<svg>` element is a container that delimits an SVG document or document fragment. A document fragment delimited by `<svg>` elements can be directly included in an HTML5 document.
`<g>`	Groups elements into a composite element that can be transformed or reused as a unit. Any presentation attribute defined on `<g>` is inherited by its children.
`<defs>`	A `<defs>` element is a container element that can be used to collect definitions of styles or reusable components inside a document. Its use is recommended, not mandatory.
`<use>`	The `<use>` element references another element and renders a copy of this element in place of the `<use>` element. The referenced element may be a container element, in which case its entire content is being copied.

The following snippet shows how several of the elements in Table B-7 might be used together in an SVG document:

```
<defs>
  <g id="doublecircle">
    <circle cx="0" cy="0" r="3" fill="red" />
    <circle cx="0" cy="0" r="5" fill="none" stroke="red" />
  </g>
</defs>
<use x="10" y="20" href="#doublecircle" />
```

Coordinates, Scaling, and Rendering

By default, the origin of an SVG graph lies in the upper-left corner, with the horizontal axis running left to right, and the vertical axis pointing *down*. Lengths in SVG are usually specified as pure numbers without units; a single unit is treated as equivalent to one pixel. The other length units familiar from CSS (`em`, `ex`, `px`, `pt`, `cm`, `mm`, and `in`) are available, but rarely used. Current CSS and SVG standards fix the pixel density at 96 dpi, thus defining the physical length of one pixel.

An SVG graph is, in principle, infinitely large. The `width` and `height` attributes set on the `<svg>` tag do *not* define the size of the figure, but only the size of the *viewport*. The viewport is an area of the given size in which part or all of the SVG figure will be shown: essentially, a "window" onto the overall SVG.

By default, the viewport will show the part of the SVG image directly underneath it, with the origin of the figure in the top-left corner of

the viewport and with the same units of length used for the viewport and the figure. To select a specific part of the entire SVG image to be shown in the viewport, the `viewBox` attribute on the `<svg>` element must be set explicitly. Using the `viewBox` attribute, a rectangular area of the entire figure can be specified, which will then be scaled to fit into the viewport. (This is also the proper way to rescale SVG figures.) The details of this mapping (from the area specified by the `viewBox` to the visible area of the viewport) are controlled by the `preserveAspectRatio` attribute, which controls not only the aspect ratio, but also the alignment of the selected area within the viewport.

SVG and CSS

SVG elements can be styled using CSS mechanisms and syntax. All presentational attributes (see Table B-4) can be set through styles.

CSS style directives can be included in an SVG document, using `<style>...</style>` tags inside the `<defs>` section:

```
<svg>
  <defs>
    <style>
      circle { fill: blue }
      .strong { fill: red }
    </style>
  </defs>

  <circle cx="100" cy="100" r="5" />
  <rect class="strong" x="200" y="200" width="50" height="20" />
</svg>
```

It is also possible to link an external stylesheet, for example, by including a line like this in the `<head>...</head>` section of the HTML page:

```
<link href="styles.css" rel="stylesheet">
```

Finally, inline style directives are legal, but less commonly used:

```
`<circle style="fill: green" />`
```

Resources

The tutorial on the Mozilla Development Network (MDN) contains more detail, but moves relatively quickly and can be recommended as a place to start: MDN SVG Tutorial. On the MDN, you can also

find a reference for elements, MDN SVG Element Reference, and attributes, MDN SVG Attribute Reference.

The SVG 2 Draft Standard itself is quite readable and, of course, the definitive source of information.

APPENDIX C

Hitchhiker's Guide to JavaScript and the DOM

JavaScript

If you have used a language from the C, C++, C#, and Java family, then the JavaScript syntax will feel immediately familiar. And if you have used any dynamically typed language (like Perl, Python, or Ruby) before, then *most* of JavaScript's semantics will appear familiar as well.

That being said, JavaScript holds more than its fair share of surprises. Following is a list of "gotchas"—features of the JavaScript language that you are likely to encounter in practice (when working with D3 in particular) that might not become obvious just by reading the code or skimming the reference. (This section does *not* pretend to be a comprehensive introduction to the JavaScript language; see the pointers at the end of this appendix for some recommendations.)

Hosted language. JavaScript was originally designed to run in a *hosted environment*, specifically, in the browser. This means that many services usually provided by the operating system are unavailable; in particular, the filesystem is inaccessible (and so is the standard output channel). The network, on the other hand, can be reached. The purpose of the *Node.js* project is to provide a standalone JavaScript runtime environment that can run outside of a hosted environment ("on the server," as opposed to in the browser).

The host environment provides a number of services. Of particular practical importance is the global `console` object, which gives the programmer access to the *developer console*. (Browsers differ in how you open the developer console from the user perspective—right-clicking into a page and selecting "Inspect" is a common way.) Using `console.log()`, it is possible to write data structures and other information to the screen, but the `console` object provides further useful functionality, such as formatted output and timers for simple profiling tasks.

All code running in a web browser silently creates a DOM `Window` instance. It can be accessed through the global `window` variable.

Semicolon insertion. JavaScript will attempt to insert a semicolon terminating a statement. However, the precise rules governing this feature are complex.[1] It is often recommended not to rely on semicolon insertion, but to insert them explicitly where appropriate.

Basic data types. Strings can be either surrounded by single or double quotes; both types of quotation are entirely equivalent. The two different quotation styles can enclose each other: `"That's good"` or `'Say "Go"'`. The usual escape sequences are available, both in single- and double-quoted strings. There is no separate character type. The plus `+` operator concatenates strings.

There are two distinct types to indicate "void" or "null": `null` and `undefined`. Of these, `undefined` is only used to indicate a missing initialization (in variable declarations, for nonexistent function arguments and object properties, and as the return value from functions without an explicit `return` statement). The `null` value is generally used to indicate the absence of a legal value (and is therefore most like `void`, `null`, `nil`, `None`, and `undef` in other languages).

JavaScript numbers are standard, IEEE 754 double-precision floats (including support for the `NaN` special value), familiar from every other contemporary programming language.

Boolean expressions. The following expressions (and only these!) evaluate to `false` in a Boolean context (they are "falsy"):

1 A good summary of the rules can be found in Item 6 of *Effective JavaScript* by David Herman (Addison-Wesley Professional).

```
0, "", false, null, undefined, NaN
```

All other values, including the empty array and the empty object, evaluate to `true` (they are "truthy").

Boolean operators short-circuit. The return value of the entire Boolean expression is the last evaluated expression.

Type conversions. Like many other languages, JavaScript will convert types in mixed expressions. Unfortunately, the JavaScript rules for implicit conversions are unusually complex, inconsistent, and painful. In particular, the `+` operator behaves differently than the other binary operators, and the final result depends on the order of arguments.

The binary operators `-`, `*`, `/`, and `%` first convert both arguments to numbers, *even if both arguments are strings*. In contrast, the `+` operator converts both arguments to strings, *if at least one of the arguments is a string*. Furthermore, if there is more than one `+` operator, the expression is evaluated left to right, and hence the order of arguments matters. Finally, when used as *unary* (prefix) operators, both `+` and `-` transform the following string to a number. Some examples:

```
"1" - "2"       // evaluates to -1 (number)
1 + "-2"        // evaluates to "1-2" (string)
1 + +"-2"       // evaluates to -1 (number)

1 + 2 + "3"     // evaluates to "33" (string)
1 + "2" + 3     // evaluates to "123" (string)
1 + 2 + +"3"    // evaluates to 6 (number)
1 + 2 + -"3"    // evaluates to 0 (number)
```

Expressions that evaluate to `true` in Boolean context have a numeric value of 1 when used in numeric context; expressions that evaluate to `false` have a numeric value of 0. The exception is `undefined`, which converts to `NaN`.

Also check "Converting Safely Between Strings and Numbers" on page 227.

Equality comparison. There are two operators each for testing equality or inequality:

- Normal comparison using `==` and `!=` will apply the conversion rules (see previous item) to their arguments before comparison.

- Strict comparison using `===` and `!==` will only consider arguments as equal if they are of the *same type* and have equal value.

The recommendation is to use *only* strict comparison `===` and `!==`.

Variable declaration and scope. Variables declared without any further qualifier automatically become *global* variables.

The `var` keyword is used to restrict the scope of a variable to the enclosing *function*. It is important to realize that in JavaScript the pertinent scope is defined by the enclosing function, not the enclosing block. In the following snippet, *both* `y` and `z` are visible in the entire function; the declaration of the variable `z` is *hoisted* to the top of the function. Nevertheless, `z` is only *initialized* inside of the conditional block:

```
function f(x) {
    var y = ...;
    if( ... ) {
        var z = ...;
    }
}
```

The initialization is an optional part of the declaration statement. Two forms of declaration and initialization are commonly used for multiple variables:

```
var a = 1,
    b = 2;
```

(also on a single line: `var a = 1, b = 2;`) or:

```
var a = 1;
var b = 2;
```

Multiple declarations of one variable with the same name inside the same function (that is, the same scope) are legal. However, the `var` keyword is ignored in subsequent statements: no new variable is declared, merely a new value is assigned to the existing variable.

Recent additions to the JavaScript language introduce the additional keywords `let` (to declare a block-level variable) and `const` (to declare a constant value).

Functions. There are two ways to define functions:

- Via a *function declaration*:

```
function f(x) { ... }
```

- Via a *function expression*:

```
var f = function(x) { ... };
```

Function expressions do not need to be assigned to a variable. They can be used as anonymous functions (for example, to define a callback). Even when defined as an anonymous function, it is still possible to give the expression a name. This name is only visible inside the function body—this is useful to define recursive, anonymous functions:

```
promise.then( function f(x) { ...;
                              f(x);          // recursive call
                              ...; } );
```

Functions are objects. It is therefore possible to add properties to a function:[2]

```
function f(x) {
    f.author = "Joe Doe";
    ...;
}
```

All function arguments are optional. Arguments that have not been supplied when the function was called have the `undefined` value. Inside a function body, all supplied arguments are also available in a read-only, array-like data structure called `arguments`. You can obtain the number of supplied arguments using `arguments.length`, and access each argument by its index, as in `arguments[0]`.

The `return` keyword is necessary to return a value from a function; a function without an explicit `return` statement (or with an empty `return`) returns `undefined`.

JavaScript supports closures.

The this variable. The variable `this` is automatically available in every function body. If the function was called as a method (that is, as a member function of some object), then `this` refers to that

2 This feature is frequently used by D3, in particular for generators, components, and layouts (see Chapter 5). The following discussion on StackExchange is informative: *http://bit.ly/2UC9guz*.

object (to the *receiver* of that method call). If the function was called as a nonmember function, then `this` refers to the *global object*.

The `this` variable is defined for each function scope. Nested function definitions therefore reset (or clobber) the `this` variable. This is a problem when defining callbacks and other nested functions inside of member functions:

```
var obj = {
  a: "Hello!",

  f: function() {
    console.log(this.a);                              // Hello!
    console.log(function() { return this.a; }());     // undefined

    var that = this;
    console.log(function() { return that.a; }()); } // Hello!
};
```

In the second call to `console.log()`, an anonymous function is defined and immediately evaluated (note the trailing parentheses!), and the function's return value is printed to the console. Because this anonymous function defines its own scope, `this` is reassigned and no longer refers to the enclosing object. (Specifically, `this` now points to the global object, which does not have a member `a`, hence the function returns `undefined`.) The usual remedy is to save the contents of the `this` variable to another variable, traditionally called `that`, in the outer scope, as demonstrated by the last two lines in the code snippet.

JavaScript provides several facilities for "synthetic" function calls. These facilities accept an additional parameter that is used to populate the `this` variable. Check the reference documentation for `call()`, `apply()`, and `bind()`.

Arrow functions. Arrow functions (or "fat arrow notation") are a recent addition to JavaScript (introduced in ES6, released in 2015). They simplify the definition of small anonymous functions. To summarize the most important parts of their syntax:

- Enclose the parameters in parentheses and the function body in braces:

  ```
  (a, b, c) => { statements }
  ```

- If there is only a single parameter, the parentheses are optional:

  ```
  a => { statements }
  ```

- If there are no parameters, use empty parentheses:

```
() => { statements }
```

- If the function body consists of a single expression, then both the braces and the `return` statement are optional, and the expression value will be used as the return value from the function:

```
(a, b, c) => expression;
```

Arrow functions don't have their own `this` or `arguments` variable; instead the values of both `this` and `arguments` are retained from the enclosing scope. Therefore arrow functions cannot be used in situations where you need `this`; if you do need `this`, the `function` keyword must be used to define the function or callback.

Collection data types. Traditionally, JavaScript has not had a genuine collection type: neither an array nor a dictionary (hashmap). The `Array` data type is, in fact, an *object*, whose keys are restricted to (string representations of) integer values. In addition, the `Array` data type supports a number of methods that implement typical array operations (such as `push()`, `pop()`, `splice()`, `sort()`, and so on.) The number of elements in the array is available as a member variable `length` (not a function).

The implementation of `Array` as an object with restricted keys has the peculiar effect in that eliminating an element (as in `delete a[1]`) does *not* shrink the overall length of the collection; instead it creates a "hole": an undefined entry without a key. The treatment of holes by various JavaScript facilities is inconsistent.[3] The proper way to remove an element from an array without leaving holes is to use the `splice()` function. (The `delete` operator is intended to remove elements from objects.)

Negative indices into arrays are not counted from the back of the array (as in Perl, Python, Ruby, and so on), but are interpreted as strings. Assigning to a negative index `a[-1] = 3;` results in the addition of a new *property* with the property name `"-1"` to the array object. To count elements from the back of the array, use the func-

3 See *Speaking JavaScript* by Axel Rauschmayer (O'Reilly), page 283, for a summary.

tions `slice()` and `splice()` (for read and write access, respectively).

JavaScript does not have a proper dictionary (or hashmap) type, either. Instead, objects are used for that purpose. The main problem is that objects automatically contain a number of built-in, inherited functions and members, in addition to the keys that were explicitly assigned. This can lead to problems if you inadvertently try to use a property name that already exists.

Recent versions of JavaScript contain a proper `Map` data type. D3 also contains several collection and container data types, and some additional utility functions to work on arrays.

Objects. The preferred way to create an object (instance) is through an *object literal*:

```
var obj = { a: 1,
            b: [ 1, 2, 3 ],
            c: function(x) { return x*x }
          };
```

Note the specific syntax of object literals (colons between key and value, commas between different entries). A missing comma (or a semicolon in place of a comma) can lead to obscure parsing errors.

The members of an object are accessible through dot notation and bracket notation. Both forms are equivalent, except that dot notation requires the key to be a valid JavaScript identifier (a sequence of alphanumeric characters, including `_` and `$`, not starting with a digit), whereas bracket notation allows any string:

```
obj.a += 2;          OR        obj["a"] += 2;
obj.f(3);            OR        obj["f"](3);
```

JavaScript objects and types are not "closed": their members and interface can be changed dynamically. It is therefore possible to find an object of a standard type (such as an `Array`) that nevertheless has additional members.[4]

The JavaScript object, type, and inheritance model is unlike anything currently practiced in other mainstream programming languages. However, the D3 API effectively encapsulates most of the

4 This feature is used by D3, in particular to decorate arrays returned from layouts with additional members; see Chapter 5 and Chapter 9.

underlying life cycle and hierarchy management, making it relatively safe and easy to ignore the unfamiliar aspects.

Mathematical functions. The usual mathematical functions are accessible as members of the global `Math` object:

```
Math.PI;
Math.sin(0.1);
```

Unavailable features. Several features familiar from other programming languages are not built into JavaScript or its standard library. This has led to a profusion of third-party libraries and frameworks. Programs that use only JavaScript, without the use of any such libraries or frameworks, are referred to as *Vanilla JavaScript* (or, jokingly, as using the "VanillaJS Framework").

JavaScript provides almost no mechanisms for encapsulation and modularization. Closures, in the form of Immediately Invoked Function Expressions (IIFE), are sometimes used to create local scopes.

JavaScript does not support threads. Instead, events are placed into a queue and are processed sequentially. In order not to block user-facing functionality, it is therefore important to handle long-running tasks (specifically I/O) *asynchronously* (via callbacks and promises—see "JavaScript Promises" on page 115).

JavaScript does not have a facility for producing formatted output in the spirit of `printf()` and similar functions. D3 includes an alternative for numerical and date/time values (see Chapters 6 and 10, respectively).

Converting Safely Between Strings and Numbers

Converting between strings and numbers is a common task for any program that reads data from a file in order to operate on it. JavaScript offers a somewhat confusing array of methods, all with slightly different trade-offs.

String to number:

`Number(str)`
: If the variable `str`, after stripping leading and trailing whitespace, is exactly representable as a number, then `Number(str)` returns a number of the represented value. If any character is

encountered that cannot be converted, `NaN` is returned. Convertible are `"123"`, `" 123 "`, `"1.0"`, and `"1.e3"`. Not convertible are `"123a"` and `"1-2"`. The empty string, a string consisting only of whitespace, and the value `null` are converted to `0`. The Boolean values `true` and `false` convert to `1` and `0`, respectively.

`parseFloat(str)`
: This is a global function, not a member function of an object. After stripping leading whitespace, it converts the string to an equivalent numerical value until it encounters any character that is not convertible, and returns the value up to that point. If no conversion at all took place, a value of `NaN` is returned. That is, `"123a"` converts to `123`. Boolean values, the empty string, and the value `null` all convert to `NaN`.

`+str`
: The unary prefix operator `+` has the same behavior as the `Number()` function. In arithmetic expressions, it binds tighter than other arithmetic operators: `3 * +"2"` evaluates to `6`. The unary prefix operator `-` acts like unary `+`, but in addition changes the sign of the numeric result.

  Notice that `Number()` is more permissive with regard to special values (such as the empty string, `null`, and the Booleans), but that `parseFloat()` is more permissive with regard to *trailing* nonconvertible characters. All three methods convert `undefined` to `NaN`.

Number to string:

`String(num)`
: Returns a string representation of its argument. Note that no string formatting is available—for example, it is not possible to restrict the number of digits to the right of the decimal point. The values `null`, `true`, and `false` convert to the strings `"null"`, `"true"`, and `"false"`.

`"" + num` *or* `'' + num`
: An equivalent shorthand for `String(num)`.

`num.toString()`
: If `num` is a variable, with a value that is neither `null` nor `undefined`, then the `toString()` member function will return a string representation of that value. Attempting to invoke `toString()` on a numeric literal (as in `1.toString()`) will lead

to a syntax error, while attempting to invoke `toString` on `null` or `undefined` will lead to a runtime error. A receiver value of `NaN` is permitted and leads to the result `"NaN"`.

JavaScript does not include facilities for formatted output (comparable to the `printf()` family of functions, familiar from many other languages), but D3 provides replacement functionality (see Chapter 6).

Two cautionary comments about other available mechanisms, not mentioned so far:

- Never use the `Number()` or `String()` functions together with the `new` keyword. Doing so will not create primitive types (numbers or strings) but "wrapper objects" with different behavior.
- The function `parseInt(str, radix)` converts a string to an integer, using the specified radix (which must be between 2 and 36). The function stops when encountering a nonconvertible character, hence it may give incorrect results if the string is given in exponential notation (because of the `e` in exponential notation).

Finally, some comments about converting objects to primitive types (strings or numbers):

- The method `valueOf()` is inherited by all objects and converts the receiving object to a primitive value. If the object represents a number, then `valueOf()` returns a numeric type of the corresponding value; otherwise, it returns the object itself. (Using the latter value in an arithmetic expression will yield a value of `NaN` for the expression.) The function `valueOf()` is usually not called explicitly by programmer code.
- The method `toString()` is also available on all objects; it is called (usually implicitly) to obtain a string representation of the object. However, its default implementation does not return a descriptive string uniquely identifying the object, but instead returns a generic constant. (It is therefore not possible to use objects directly as keys in hashmaps.)
- The easiest way to obtain a formatted string representation of an arbitrary object is to use `JSON.stringify()`.

The DOM

Originally, a web page was a file with HTML markup in it: the browser would download such a file and render its contents. This traditional point of view is no longer valid. It is more accurate and helpful to think of a document as a dynamic *data structure* that is maintained and updated by the browser. An initial version of it may in fact have been downloaded, but as an in-memory data structure, it can of course change at any time; its current state is continuously rendered by the browser. It is this visual representation rendered by the browser that the user sees and interacts with.

The Document Object Model (DOM) is a standardized, object-oriented API that exposes this data structure and enables programmers to query and manipulate it. The DOM represents the elements of a document (like a web page or an XML document) as *nodes* in a *tree* (the *DOM tree*). In principle, the DOM specification is language independent, but we will only consider its JavaScript implementation. Because JavaScript is running inside the browser, it is the obvious technology to manipulate documents dynamically on the client side.

The traditional DOM API is notoriously verbose and awkward. Various libraries (such as jQuery) have been developed to provide a streamlined wrapper for the original API and to add further functionality. D3 replicates much DOM functionality as well (in addition to its graphics capabilities). In fact, D3 wraps the DOM quite effectively, but to make sense of the available features, it is helpful to know a little about the DOM itself.

Classes

A few of the most important classes or interfaces that you will encounter include:

`EventTarget`
: Anything that can receive events (technically, anything that can have an `EventListener` attached to it via the `addEventListener()` member function). Almost anything in the DOM implements `EventTarget`.

`Window` *(implements* `EventTarget`*)*
: An abstraction for the browser window, its size, and GUI elements (such as title, menu, and scroll bars).

`Node` *(implements* `EventTarget`*)*
: A *general* abstraction for anything in a DOM tree. For example, `Document`, `Text`, and `Comment` all implement `Node`, but do not implement `Element`. Even attributes can be represented as `Node`.

`Element` *(implements* `Node`*)*
: An actual page element. More specific subclasses (such as `HTMLElement` or `SVGElement`) exist for specific document types, each having many specific subtypes representing the particular page elements of that document.

`Document` *(implements* `Node`*)*
: An abstraction for the entire page or document loaded into the browser; in other words, a `Document` provides a handle on the entire DOM tree. Browsers automatically create a global variable `document` that represents the current page.

Be aware that many commonly used API elements are implemented as *properties*, not *methods*.

DOM Events

An important part of the DOM API is its support for *events*. Events can be user events (such as mouse or keyboard actions), resource events (when a resource has finished loading), network events, and many more.

An application can respond to events through *event listeners* (or handlers). An event listener is attached to a page element (technically, to an `EventTarget` instance), receives events for that element, and invokes the appropriate callback. There are basically two ways to register an event handler with a page element:

- Use `EventTarget.addEventListener( type, callback, options )`. Using this method repeatedly for the same target object, it is possible to add more than one listener to a single element for a given event type; a corresponding function allows you to remove listeners at runtime. This is the recommended way to add listeners to elements.

- Inline, using syntax like `<p onclick="handler()">` (where `handler()` is a JavaScript function). This method is discouraged, mostly because it mixes markup and code, but also because it only allows a single handler per event type and element.

Traditionally, there was a third method that added listeners as properties directly to the page elements; this is now obsolete. D3 provides its own way of adding event listeners to the D3 `Selection` abstraction via the `on()` member function.

When using `addEventListener()`, the handler is registered for the element on which this method is invoked. The `type` argument is a string identifying what kind of event the handler should respond to. The number of defined events is quite large.[5] The callback is a function that is being passed the current `Event` instance as a single argument and must return nothing. Inside the callback, `this` points to the element that the listener was added to. The `options` argument is optional; usually the defaults are fine.

Event Propagation

If you peruse the DOM reference documentation as it pertains to event handling, you may find yourself bewildered by some oddly conspicuous features using an unfamiliar and not very descriptive terminology. Most of them are related to *event propagation*, and their relative prominence in the APIs reflects the historical uncertainty about the best design. Today's DOM event handling facilities are the product of a convoluted gestation process, and the resulting API contains traces of several competing design ideas (and their respective, and sometimes conflicting, jargons). Because it can be difficult to find all the information in one place, here is a summary.

Imagine two page elements nested inside each other (such as a table cell inside a table), each of which has an event listener for the same event type attached to it (let's say, both listen for `click` events). If the user now clicks into the table cell, in which *order* will the event handlers be invoked: outside in, or inside out? The answer depends on how the event listeners were registered. Events first propagate from the outermost element inward toward the innermost event target

5 For a complete list of events, see the MDN Event Reference.

(this is known as the *capture* phase), then travel outward again to the top-level document (this is the *bubble* phase). Through the options parameter supplied to `addEventListener()`, you can select when the associated callback should be invoked. By default, event handlers are invoked during the *bubble* phase, in other words, inside out. (Some event types, associated with particular elements, don't propagate or "bubble."[6])

This kind of event propagation is convenient, because it makes it possible to register a single event listener on a common *parent*, rather than on each element individually (this is known as *event delegation*). The propagation of events can be prevented by invoking the `stopPropgation()` or `stopImmediatePropagation()` methods on the `Event` instance passed to the event listener callback: `stopPropagation()` will prevent any *parent* handlers from being invoked (but other handlers registered on the current event target will be executed), whereas `stopImmediatePropagation()` will prevent *any* further handlers (on parents or the current object) from being invoked. (You may also still find references to the `cancelBubble` property, which is now obsolete.)

The `Event` object instance passed to the event listener callback carries information about the elements receiving and handling the event. The property `eventTarget` points to the element that dispatched the event (that is, the innermost element), whereas `currentTarget` points to the element on which the current event handler was registered. (The property `relatedTarget` is mostly relevant for certain mouse events that involve more than one element, such as dragging.)

Preventing *default actions* is a separate, but related concept. Browsers may take certain actions "by default" when an event occurs (such as following a link when it is clicked, or opening the "Print" dialog when the "Print" menu entry has been selected). If an event is *cancelable*, then calling `preventDefault()` on it will prevent the browser from taking whatever default action is usually associated with this event; the event has been *canceled*. Not all events (or event types) are cancelable; examine the `cancelable` property on the

6 See *https://en.wikipedia.org/wiki/DOM_events*.

event object. Calling `preventDefault()` on a noncancelable event has no effect.

Finally, marking an event listener as "passive" allows for a certain optimization when handling scrolling. (This is a new, specialized feature.

The Browser as Development Environment

Every modern browser comes with built-in development tools. The way you access them is browser-specific; right-clicking into the page and selecting "Inspect" or "Inspect Element" is a common method. This will open a panel with various tabs, providing access to the different tools (you can expect to find a debugger, a profiler, network, memory, storage monitors, and more). Three are indispensable:

The console
: The console is a message area where the JavaScript runtime will place error messages. The program can write to it using the `console` object. The console is fairly smart: for example, it can suppress identical loglines or collapse complex data structures. Console output can be navigated with the mouse (for example, to expand collapsed data structures).

The command line
: The console window provides an interactive prompt to enter and evaluate JavaScript code; output will be directed to the console. All the elements of the page will be available as data structures and can be inspected and manipulated.

The element inspector
: The element or page inspector shows the current version of the DOM tree as a hierarchical collection of tags and attributes. This is invaluable when trying to track down problems in the document structure.

Resources

- The Mozilla Development Network (MDN) is a comprehensive, frequently updated, yet also somewhat overwhelming resource for all kinds of web technologies. The quality of the content can be uneven.

- The information at *http://www.w3schools.com*, often ranked very highly in search-engine results, tends to be less detailed and up-to-date.
- The book *Learning Web Design* by Jennifer Niederst Robbins (O'Reilly, Fourth Edition) is particularly strong on HTML and CSS, but also briefly covers JavaScript and the DOM.

JavaScript

- A Re-introduction to JavaScript on MDN is a good, concise overview of JavaScript for readers with previous programming experience.
- Among the many JavaScript books, I found the following one particularly suitable for readers coming to JavaScript from other programming languages, because it moves rather quickly and emphasizes JavaScript's pecularities: *Speaking JavaScript: An In-Depth Guide for Programmers* by Axel Rauschmayer (O'Reilly).
- The JavaScript Guide and JavaScript Reference, both on MDN, are useful as references on individual topics.

Index

C

D

E

F

G

H

I

T

U

V

W

X

Y

About the Author

Philipp K. Janert was born and raised in Germany. He obtained a PhD in theoretical physics from the University of Washington in 1997 and has been working in the tech industry since, as programmer, scientist, and applied mathematician. He is the author of *Data Analysis with Open Source Tools* (O'Reilly), *Feedback Control for Computer Systems* (O'Reilly), and *Gnuplot in Action* (Manning, Second Edition).

Colophon

The animal on the cover of *D3 for the Impatient* is the purple-crowned fairy hummingbird (*Heliothryx barroti*), a bird that lives in the treetops and along the edges of humid lowland forests in a range that extends from Guatemala south to northernmost coastal Peru.

This hummingbird has metallic green feathers above, bright white below, with long dark wings and tails; they have a black stripe around their eyes. Males have a metallic-violet cap that gives the species its name. On average, purple-crowned fairy hummingbirds weigh nearly 2 ounces (about as much as a US nickel), and are 4 inches long.

Along with the standard hummingbird flower nectar diet, these birds hawk for spiders and other bugs in midair. As is also common with hummingbirds, after mating, the females of this species build nests, incubate their eggs, and raise young on their own. From egg-laying to the fledglings' first hover from the nest averages 40 days.

The purple-crowned fairy hummingbird is known to humans sharing its territory as a bold, inquisitive visitor. However, they have been observed attacking other hummingbirds, driving at them with their beaks and making the insistent chipping sound that signals hummingbird aggression.

As a result of millennia of coevolution with these birds, some flowers are configured to use them for pollination. During feeding, hummingbirds' heads are in the right position to be dusted with the flower's pollen, so that pollination occurs when they feed at the next flower. However, when confronted with long, tubular flowers containing nectar that their short bills cannot reach, the purple-crowned fairy hummingbird has a workaround: though its beak is

short, it's sharp, and by piercing the base of the flower, it accesses the nectar. This of course thwarts the flower's pollination strategy.

With a good-sized range and population, this hummingbird's conservation status on the IUCN Red List is that of being of Least Concern.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover image is a color illustration by Karen Montgomery, based on an engraving from H. G. Adams's *Humming Birds Described and Illustrated* (1856). The cover fonts are Gilroy and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.