Elijah Meeks

D3.js IN ACTION

MEAP

MANNING

MEAP Edition
Manning Early Access Program
D3.js in Action
Version 5

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thank you for purchasing the MEAP for *D3.js in Action*. To get the most benefit from this book, you'll want to have some established skills in programming, with experience in HTML5 and basic knowledge about CSS and DOM, or be transitioning from dealing with data in R/Python/SQL with the desire to build more sophisticated shareable applications.

When I first started experimenting with D3 two years ago, it was out of necessity. Flash was dead, and I needed an information visualization library that was feature rich and, hopefully, long lived. D3 proved to be more than that, with robust capabilities not only for data visualization, but for creating complex applications on the web.

Since that time, there have been numerous examples, some really great introductions to the library, and a few cookbooks to help new users learn the basics and get specific tasks done with D3. In contrast, *D3.js in Action* is my attempt to produce an exhaustive, highly informative, deep dive into the library, one that covers the fundamental structures of how D3 works with data to produce the stunning products we're all so impressed by. D3 does a lot, and not just with graphics, and I've tried to explain in detail the whole library.

Along with all that, I also wanted to focus on two specific spheres that D3 handles really well: networks and maps. As a result, this book spends a full chapter on each, dealing with the variety of features and functions in D3 that let you create the most amazing network visualizations and online interactive mapping applications. This is along with chapters that focus on general charts and D3-specific layouts more broadly.

Finally, throughout *D3.js in Action* you'll see an approach that embraces the functionality available in modern browsers.

It's a big book for a big library, and I hope you find it as useful to read as I did to write it. Please be sure to post any questions, comments, or suggestions you have about the book in the Author Online forum. Your feedback is essential in developing the best book possible.

— Elijah Meeks

brief contents

1

An introduction to D3.js

1.1 What is D3?

Data-Driven Documents is a brand name. It's what "D3" stands for. But it's also a class of applications that have been offered on the web in one form or another for years. Whether as interactive dashboards, rich internet applications, or dynamically driven content, we've been building and dealing with data-driven documents for quite some time. So in one sense, the D3.js library is an iterative step in a chain of various technologies used for data-driven documents, but in another sense, it is a radical one.

D3.js comes out of a need for robust data visualization on the web, but does more than that because of its robus design. Coming out of the data visualization program at Stanford Computer Science, Mike Bostock worked with Jeff Heer and Vadim Ogievetsky to create Protovis, which like D3.js is a JavaScript library designed for information visualization but which was designed to provide compatibility with older browsers. Bostock also developed Polymaps, another JavaScript library which provided vector and tile mapping capability in a lightweight form. These earlier endeavors would inform the creation of D3.js, which focused on modern standards and modern browsers. As Bostock describes it, "This avoids proprietary representation and affords extraordinary flexibility, exposing the full capabilities of web standards such as CSS3, HTML5 and SVG." This is the radical nature of D3.js. It will not run on Internet Explorer 6, and while that cost may be too much to bear for certain developers, the wide adoption of standards on modern browsers has finally afforded the capacity to break with the past. In that regard, D3 is a sign of the new capabilities that let web developers deliver dynamic and interactive content seamlessly in the browser.

The iterative nature of D3.js comes from its resemblance to earlier methods of deploying rich interactivity to the web, such as Flash using ActionScript3. As I or any other former Flash developer can tell you, the period after which Steve Jobs condemned Flash--but before the maturation of JavaScript engines and browsers--was a difficult one. You simply could not build

the kind of high-performance rich Internet applications in the browser like you could in a compiled Flash runtime. The performance of Flash more than outweighed its proprietary nature and the sometimes seemingly willful obtuseness of Adobe. Flash for animation and RIAs is still common on the Internet, and especially for internal webapps, for this very reason. D3.js provides the same performance, but integrated into web standards and the document object model at the core of HTML.

With that in mind, let's take a look at the basic principles of data visualization and how D3 works. We'll do this by establishing the basic principles of how D3 selections and data-binding work, followed by understanding how it interacts with SVG and the DOM, then we'll look at data types that you'll commonly encounter. Finally, we'll use D3 to create some simple DOM and SVG elements.

1.1.1 Data Visualization is More Than Just Data Visualization

You may think of data visualization as limited to pie charts, line charts, and the variety of charting methods popularized by Tufte and deployed in research. One of the core strengths of d3.js is that it allows for the creation of vector graphics for traditional charting, but also the creation of geospatial and network visualizations as well as traditional HTML elements like tables, lists, and paragraphs. This broad-based approach to data visualization, where a map or a network graph or a table is just another kind of representation of data, is at the core of the D3.js library's appeal for application development. By requiring a break with the practice of supporting long-obsolete browsers, D3.js affords developers with the capacity not only to make richly interactive applications, but applications that are styled and served just like traditional web content. This makes them more portable, more amenable to the growing linked data web, and more easily maintained by large teams.

The decision on Bostock's part to deal broadly with data, and create a library capable of presenting maps as easily as charts as easily as networks as easily as ordered lists, also means that a developer need not split their effort trying to understand the abstractions and syntax of one library for maps, and another for dynamic text content, and another for data visualization. Instead, the code for running an interactive force-directed network layout is not only very close to pure JavaScript, it's also very similar to representing dynamic points of interest on a D3.js map. Not only are the methods the same, the very data could be the same, formulated in one way for lists and paragraphs and spans while formulated in another way for geospatial representation. The class of data-driven document is already a broad one on its face, and becomes even more all-encompassing when one also treats images and text as data.

1.1.2 D3 is About Selecting and Binding

Throughout this chapter, we'll see numerous code snippets that you can run in your browser to make changes to the graphical appearance of elements on your web site. At the end of a chapter is a simple application written in D3 that explains the basics of the code you're running in JavaScript. But before that we'll be exploring the basic principles of web development using D3, and you'll just see this pattern of code over and over again:

```
d3.selectAll("circle.a").style("fill", "red").attr("cx", 100)
```

This will make every circle on your page with the class of "a" turn red and move it so that its center is 100 pixels to the right of the left side of your <svg> canvas. Likewise, this code:

```
d3.selectAll("div").style("background", "red").attr("class", "b")
```

Will make every div on your web page turn red and change its class to "b". But before you can change your circles and divs, you'll need to make them, but before you do that, it's best for you to understand what's happening in this pattern.

The first part of that line of code, `d3.selectAll()`, is part of the core functionality necessary for understanding D3: selections. Selections can be made with `d3.select()`, which selects a single element, but you'll use `d3.selectAll()`, which can be used to select multiple elements, more often. Selections are a group of one or more web page elements that may be associated with a set of data, like so:

```
d3.selectAll("div.market").data([1,5,11,3])
```

Which would bind the elements in the array [1,5,11,3] to elements with the class of "market". This association is known in D3 as **binding data** and so a selection should be thought of as a set of web page elements and a corresponding, associated set of data. Sometimes there are more data elements than DOM elements, or vice verse, in which case there are functions in D3 designed to create or remove elements that you can use to generate content. Selections and data-binding will be explained in detail in chapter 2. Selections might not include any data binding, and won't for most of the examples in this chapter, but it's this inclusion that allows for the powerful information visualization techniques that D3 affords. A selection can be made on any elements in a web page, which includes items in a list, circles, or even regions on a map of Africa. Just as the elements can take a number of shapes, the data associated with those elements (where applicable) can take many forms.

Imagine you have a set of data such as the price and size of a few houses, and a set of web page elements, whether graphics or traditional elements, that you want to represent that data, whether with text or through size and color A selection, then, is the group of all of them together, upon which you perform some actions such as moving them, changing their color, or updating the values in the data. You can, and will, work with the data and the web page elements separately, but the real power of D3 comes from leveraging selections to use data and web page elements together.

1.1.3 D3 is about deriving the appearance of web page elements from bound data

Once you have a selection, you can then use D3 to modify the appearance of web page elements to reflect differences in the data. You may want to make the length of a line equal to the value of the data, or change the color to particular color that corresponds to a class of data. You might want to hide or show elements as they correspond to a user's navigation of a dataset.

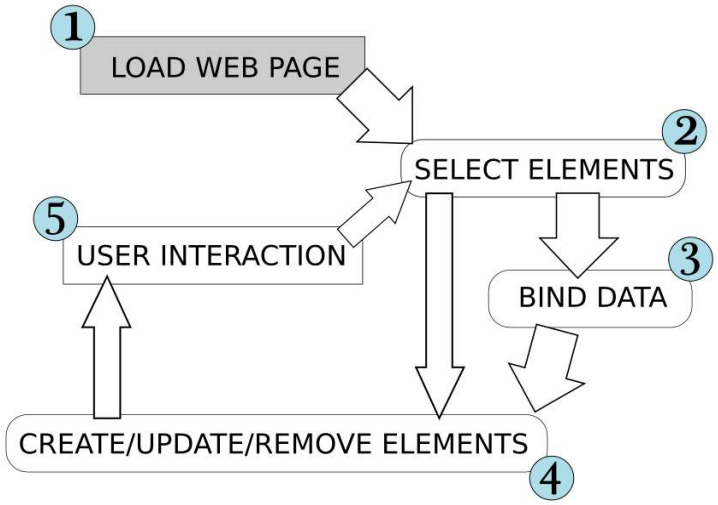

Figure 1.1 A page utilizing D3 is typically built in such a way that the page loads with styles, data, and content as defined in traditional HTML development (1) with its initial display using D3 selections of HTML elements (2) either using data binding (3) or without it to modify the structure and appearance of the page (4) . The changes in structure prompt user interaction (5) which causes new selections with and without data-binding to further alter the page. Step 1 is shown differently because it only happens once (when you load the page) whereas every other step may happen multiple times, depending on user interaction.

This is done by using selections to reference the data bound to an element in a selection. D3 is built to iterate through the elements in your selection and perform the same action using the bound data, which results in different graphical effects. While the action you perform is the same, the effect is different because it is based on the variation in the data. We'll first see data binding in process at the end of the chapter, and in much more detail throughout this book.

1.1.4 Web page elements can now be divs, countries, flowcharts

We've grown accustomed to thinking of web pages as consisting of text elements with containers for pictures or videos or embedded applications. But as you grow more familiar with D3, you'll begin to recognize that every element on the page can be treated with the same high-level abstractions. The most basic element on a web page, a that represents a rectangle into which you can drop paragraphs and lists and tables, can be selected and modified in the same way you could select and modify a country on a web map, or individual circles and lines that make up a complex data visualization.

This doesn't always hold true. To gain access to the ability to select items on a web page, you have to ensure that they are built in a manner that makes them a part of the traditional structure of a web page. You cannot select items in a Java applet, or in a Flash runtime, nor could you select the labels on an embedded Google map, but if you create these elements so

that they exist as elements in your web page, then you give yourself tremendous flexibility. To get a taste of this, look at Chapter 7, where we build robust mapping applications in D3, and you'll see the `d3.select()` syntax being used to update the appearance of a mapping application in the same manner as it's being used here and elsewhere to create and move circles or elements.

1.2 Leveraging HTML5

We've come a long way from the days when animated gifs and frames were the pinnacle of dynamic content on the web. In figure 1.2 below, we can see why gifs never caught on for robust data visualization on the web. GIFs, like the infoviz libraries designed to use VML or canvas, are still necessary for earlier browsers, but D3 is designed for modern browsers that don't need the helper libraries necessary for backward compatibility. This means that D3 development isn't for everyone, but if your audience can be assumed to have access to a modern web browser, it also brings with it a significant reduction in the cost necessary not only to code for older browsers but to learn and keep updated on the various libraries that support backward compatibility with those older browsers.

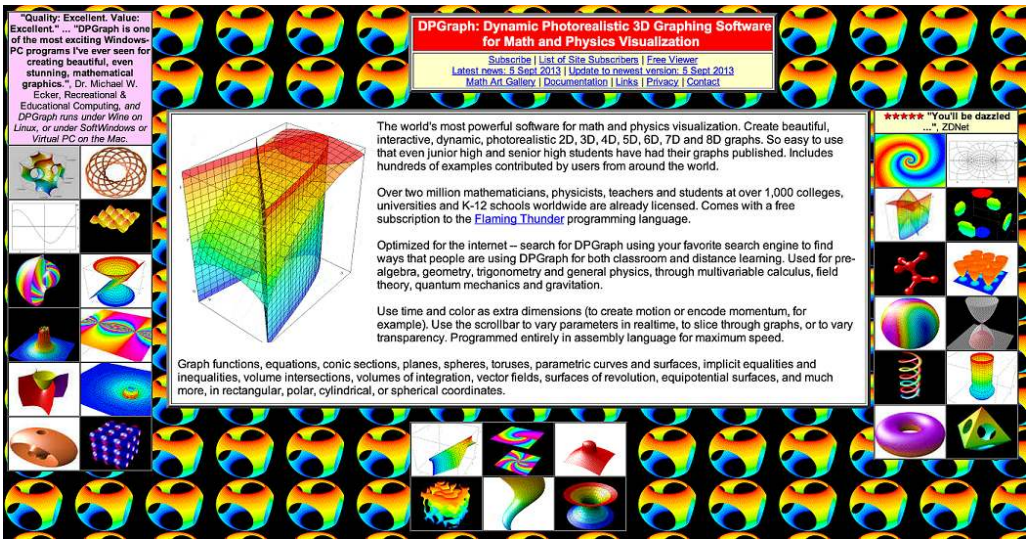

Figure 1.2 Before GIFs were weaponized to share cute animal behavior, they were your only hope for animated data visualization on the web. Few examples from the 90s like dpgraph.com still exist, but there are more than enough GIFs on this single page to remind us of the dangers of GIFs.

A modern browser typically means one that can not only display SVG graphics and obey CSS3 rules, but that does so with great performance. Along with CSS (Cascading Style Sheets) and Scalable Vector Graphics (SVG), we can break down HTML5 into the Document Object Model

(DOM) and JavaScript. The following sections will treat with each of them and include code you can run to see how D3 leverages their functionality to create interactive and dynamic web content.

1.2.1 The DOM

A web page is structured according to the Document Object Model, or DOM. You need a passing familiarity with the DOM to do web development, and it's beyond what we can cover in this book. Instead, we'll take a quick look at DOM elements and structure in a simple web page in your browser and touch on the basics of the DOM. To get started, you'll need a web server that you can access from the computer that you're using to code. With that in place, you can download the D3 library from d3js.org (d3.js or d3.min.js for the minified version) and place that in the directory where you'll make your web page. We'll create a simple page called d3ia.html in the text editor with the following contents:

```
<!doctype html>
<html>
<script>d3v3.min.js</script>#a
<body> #b
<div id="someDiv" style="width:200px;height:100px;border:black 1px solid;"> #c
<input id="someCheckbox" type="checkbox" /> #d
</div>
</body>
<html>
```

#a a child element of <html>
#b a child element of <html>
#c a child element of <body>
**#d a child element of **

Basic HTML like the kind seen here follows the DOM. It defines a set of nested elements, starting with an `<html>` element with all its child elements and their child elements and so on. In the above example, the `<script>` and `<body>` elements are children of the <html> element and the `<div>` element is a child of the `<body>` element. The <script> element loads the D3 library here, or it can have inline JavaScript code, whereas any content in the <body> element shows up onscreen when you navigate to this page.

There are three categories of information about each element that determine its behavior and appearance: styles, attributes, and properties. **Styles** can determine transparency, color, size, borders and so on. **Attributes** typically refer to classes, IDs, and interactive behavior, though some attributes can also determine appearance, depending on which type of element you're dealing with. **Properties** typically refer to states, such as the "checked" property of a checkbox, which is true if the box is checked and false if the box is unchecked. D3 has three corresponding functions to modify these values. If we wanted to modify the HTML elements above, we could do so using D3 functions that abstract this process:

```
d3.select("#someDiv").style("border", "5px darkgray dashed")
d3.select("#someDiv").attr("id", "newID")
d3.select("#someCheckbox").property("checked", true)
```

Like many D3 functions of this kind, if you don't signify a new value, then the function will return the existing value. You'll see this in action throughout this book, and later in the chapter as we write more code, but for now just remember that these three functions allow you to change how an element appears and interacts.

The DOM also determines the order of drawing of elements onscreen, with child elements drawn after and inside parent elements. While there is some control over drawing elements above or below each other with traditional HTML using `z-index`, this isn't available for SVG elements (though it might be implemented at some point using the `render-order` attribute).

EXAMINING THE DOM IN THE CONSOLE

Navigate to d3ia.html and we can get some exposure to how D3 works. The page isn't very impressive, with just a single black-outlined rectangle. You could modify the look and feel of this web page by updating d3ia.html, but you'll find that it is very easy to modify the page by using your web browser's developer console. You'll find this useful for testing changes to classes or elements before implementing them in your code. Open up the developer console and you'll have two very useful screens shown in figures 1.3 and 1.4, which we'll go back to over and over.

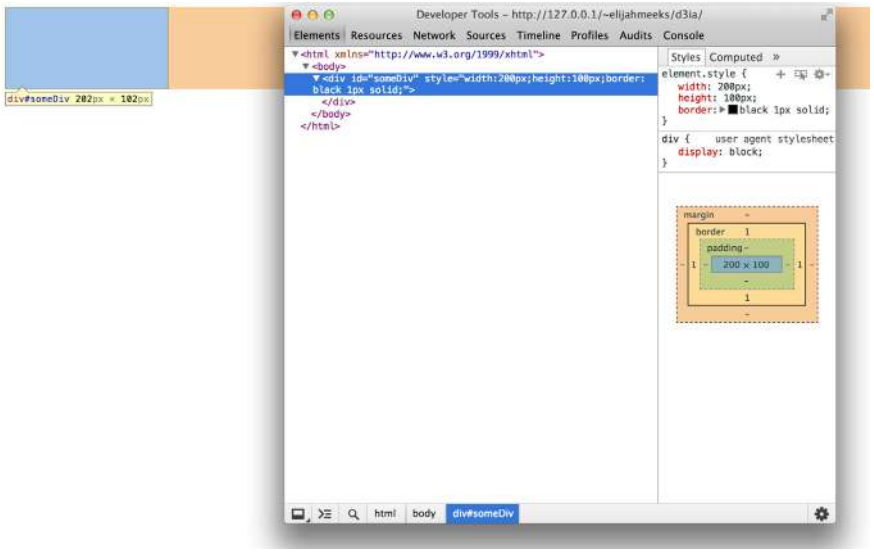

Figure 1.3 The developer tools in Chrome places the JavaScript console on the rightmost tab labeled 'Console' with the element inspector available using the hourglass on the bottom left or by browsing the DOM in the 'Elements' tab.

The element inspector allows you to look at the elements that make up your web page by navigating through the DOM (represented as nested text where each child element is shown indented). Or by selecting an element on screen graphically, typically represented as a magnifying glass or cursor icon.

The other screen you'll want to use quite often is the console (Figure 1.x), which allows you to write and run JavaScript code right on your web page:

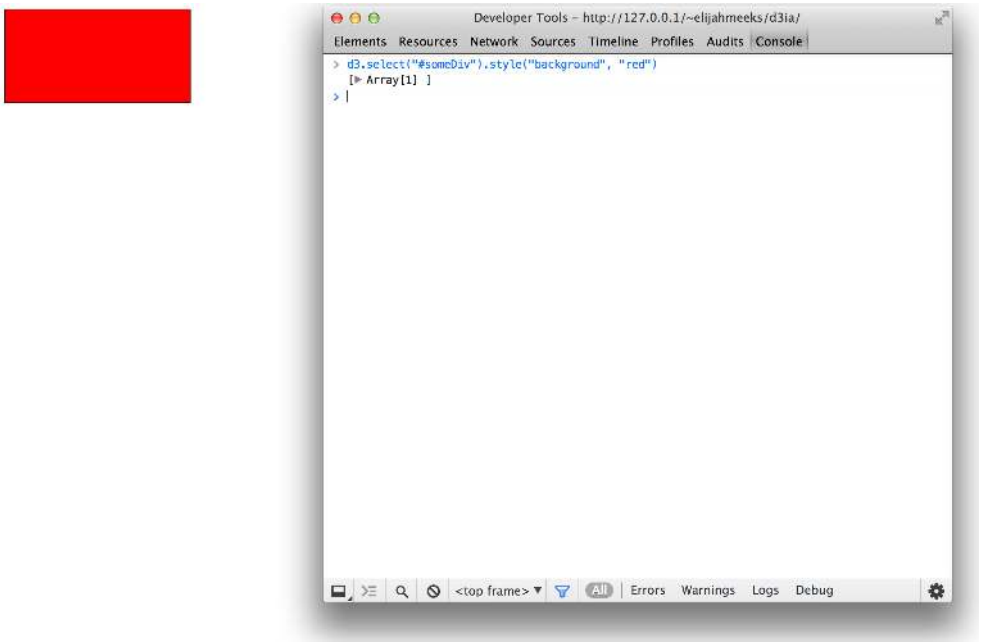

Figure 1.4 You can run JavaScript code in the console, and also call global variables or declare new ones as necessary. Any code you write in the console and changes made to the web page are lost as soon as you reload the page.

The examples in this book use Google Chrome and its developer console, but you could use Safari's developer tools or Firebug in Firefox or whatever developer console you're most comfortable with to follow along. You can see and manipulate DOM elements such as the `<div>` or `<body>` above by clicking on the element inspector or looking at the DOM as represented in HTML. You can click on one of these elements and change its appearance by modifying it in the console.

There are three categories of information about each element that determine its behavior and appearance: styles, attributes, and properties. **Styles** can determine transparency, color, and size. **Attributes** typically refer to classes, IDs, and interactive behavior, though some attributes can also determine appearance, depending on which type of element you're dealing

with. **Properties** typically refer to states, such as the "checked" property of a checkbox, which is true if the box is checked and false if the box is unchecked.

If you want, you can even delete elements in the console. Give it a try, select the div either in the DOM or visually and press delete. Now your webpage is very lonely. Hit refresh so that your page reloads the HTML and your div comes back. You can adjust the size and color of your div by adding new styles or changing the existing one, so you can increase the width of the border and making it dashed by changing the border style to "black 5px dashed". Or you can add content to the div in the form of other elements or simply text by right-clicking on the element and selecting "edit as HTML" as shown in figure 1.5 and 1.6.

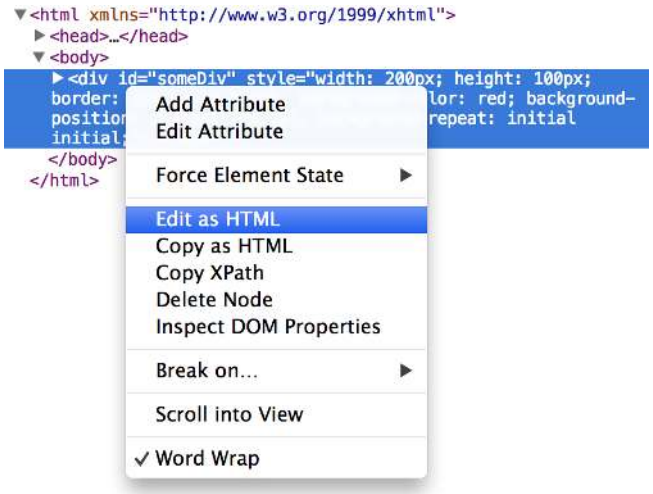

Figure 1.5 Rather than adding or modifying individual styles and attributes, you have the ability to rewrite the HTML code as you would in a text editor. As with any changes, these only last until you reload the page.

And then writing in between the opening and closing HTML whatever you'd like:

```
background-position: initial initial; ba
repeat: initial initial;">
    Here's some text to put in my div
```

Figure 1.X Changing the content of a DOM element is as simple as adding text between the opening and ending brackets of the element.

Any changes you make, regardless of whether they're well-structured or not, will be reflected on the web page. In figure 1.7 we see the results of modifying the HTML, which is rendered immediately on our page.

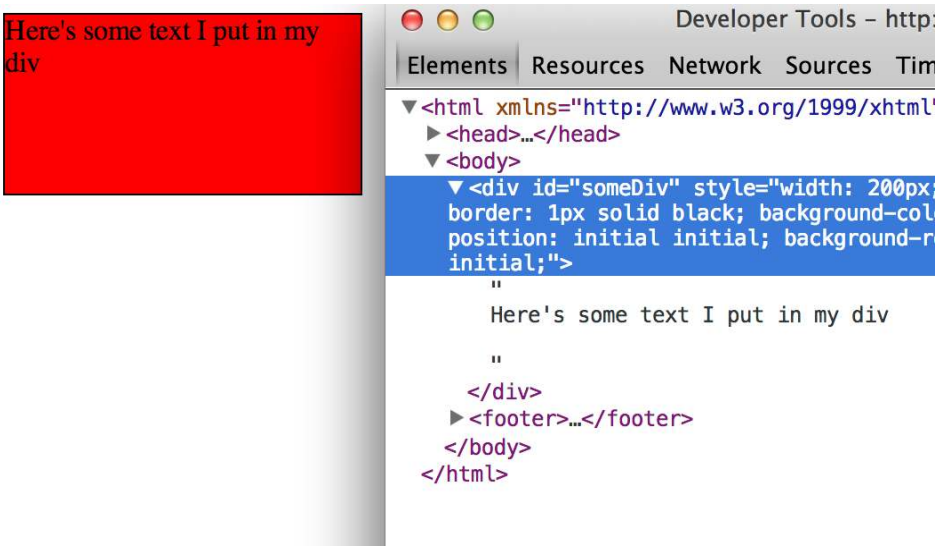

Figure 1.7 The page is updated as soon as you finish making your changes. Writing HTML manually in this way is really only useful for planning how you might want to dynamically update the content.

In this way, you could slowly and painstakingly create a web page in the console. We're not going to do that. Instead, we're going to use D3 to create elements on the fly with size, position, shape and content based on your data.

1.2.2 Coding in the Console

You'll do a lot of your coding in the IDE of your choice, but one of the great things about web development is that you can test JavaScript code changes by using your console. Later we'll focus on writing JavaScript, but for now, just to demonstrate how the console works, copy the following code into your console and hit enter and you should see the effect show in figure 1.8.

```
d3.select("div").style("background","lightblue").html("Something else maybe")
```

Figure 1.8 The D3 select syntax modifies style using the .style() function and traditional HTML content using the .html() function.

Now, if all D3 could do was select HTML elements and change their style and content like this, then it wouldn't be much of a library. To do more, we have to move away from traditional HTML and focus on a special type of elements in the DOM: Scalable Vector Graphics.

1.2.3 SVG

A major value of HTML5 is the integrated support for SVG or scalable vector graphics. These allow for simple mathematical representation of images that scale and are amenable to animation and interaction. Part of D3's attractiveness is that it provides an abstraction layer for drawing SVG. This is because SVG drawing can be a little confusing. SVG drawing instructions for complex shapes, known as `<path>` elements, is written a bit like the old LOGO programming language worked. You start at a point on a canvas and draw a line from that point to another, and if you want it to curve you can give the SVG drawing code coordinates on which take make that curve. So if you want to draw the line on the right, you would create a <path> element in an <svg> canvas element in your web page, and you would set the "d" attribute of that <path> element to equal the text on the left:

```
M50,485.7142857142857L58.33333333333332,473.80952
38095238C66.66666666666666,461.90476190476187,83.
33333333333331,438.0952380952381,99.9999999999999
9,433.3333333333333C116.66666666666666,428.571428
57142856,133.33333333333334,442.85714285714283,15
0,445.2380952380952C166.66666666666669,447.619047
6190476,183.33333333333331,438.0952380952381,199.
99999999999997,421.4285714285714C216.666666666666
63,404.7619047619047,233.33333333333331,380.95238
09523809,250,383.3333333333333C266.6666666666667,
385.71428571428567,283.33333333333337,414.2857142
857143,300,430.95238095238096C316.66666666666674,
447.6190476190476,333.33333333333337,452.38095238
09524,350,450C366.6666666666667,447.6190476190476
,383.3333333333333,438.0952380952381,400,442.8571
4285714283C416.66666666666663,447.6190476190476,4
33.3333333333333,466.66666666666663,449.999999999
99994,469.047619047619C466.66666666666663,471.428
57142857144,483.3333333333333,457.1428571428571,4
91.66666666666663,450L500,442.8571428571429
```

But you would almost never want to create SVG by manually writing drawing instructions like this. Instead, you'll want to use D3 to do the drawing with a variety of helper functions, or rely on other SVG elements that represent simple shapes (known as geometric or graphical primitives) using more readable attributes. We'll update d3ia.html to look like Listing 1.1 to include the necessary elements to display SVG, as well as some examples of the various shapes you might use:

Listing 1.1 A sample web page with SVG elements

```
<!doctype html>

<html>
<script src="d3.v3.min.js" type="text/JavaScript">
</script>
<body>
<div id="infovizDiv">
<svg style="width:500px;height:500px;border:1px lightgray solid;">
  <path d="M 10,60 40,30 50,50 60,30 70,80" style="fill:black;stroke:gray;stroke-
width:4px;" />
  <polygon style="fill:gray;" points="80,400 120,400 160,440 120,480 60,460" />
<g>
<line x1="200" y1="100" x2="450" y2="225"
style="stroke:black;stroke-width:2px;"/>
<circle cy="100" cx="200" r="30"/>
<rect x="410" y="200" width="100" height="50"
style="fill:pink;stroke:black;stroke-width:1px;" />
</g>
</svg>
</div>
</body>
</html>
```

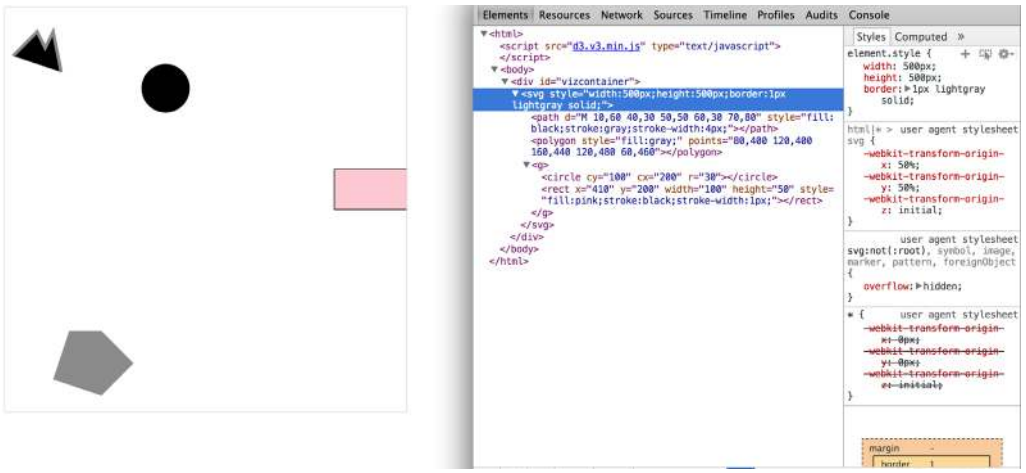

Figure 1.9 Inspecting the DOM of a web page with an SVG canvas reveals the nested graphical elements as well as the style and attributes that determine their position. Notice that the circle and rectangle exist as child elements of a group.

You can inspect the elements just like you would the traditional elements we looked at earlier, as you can see in figure 1.9, and you can manipulate these elements using traditional JavaScript selectors `like document.getElementById` or with D3, removing them or changing the style like so:

```
d3.select("circle").remove()
d3.select("rect").style("fill", "purple")
```

#a deletes the circle
#b changes the rectangle color to purple

Now, refresh your page and let's take a look at the new elements. You're already familiar with divs, and it's always useful to put an SVG canvas in a div because you'll still want access to the parent container for layout and styling. Now, let's take a look at the other elements we've added:

<SVG>

This is your actual canvas on which everything is drawn. The top-left corner is 0,0 and the canvas will clip anything drawn beyond its defined height and width (in the case of our example, anything beyond 500,500 which we can see with the rectangle). An <svg> element can be styled with CSS to have different borders and backgrounds. The <svg> element can also be dynamically resized using the viewBox attribute, which is more complex and beyond the scope of the overview here.

You can use CSS (which we'll touch on later in this chapter) to style your SVG canvas or use D3 to add inline styles like so:

```
d3.select("svg").style("background", "darkgray")
```

#a infoviz is always cooler on a dark background

REMEMBER: The x-axis is drawn left to right, but the y-axis is drawn top to bottom, so you'll see that the circle is set 200 pixels to the right and 100 pixels down.

We'll look at a few of the SVG elements below. Each of which will include some examples that you can paste into the <svg> element in your HTML to see on your web page.

<canvas>

There's a second mode of drawing available with HTML5 using <canvas> elements to draw bitmaps. We won't be going into in detail here but will touch on this method later in Chapter 2. Canvas creates static graphics drawn in a manner similar to SVG that can then be saved as images. There are four main reasons to use canvas:

Compatibility - which we won't worry about because if you're using D3, then you're coding for a modern browser.

Creating static images - You can draw your data visualization with canvas to save views as snapshots for thumbnail and gallery views.

Large amounts of data - SVG creates individual elements in the DOM and while this is great for attaching events and styling, it can overwhelm a browser and cause significant slowdown.

WebGL - Canvas allows you to use webGL to draw, so that you can create 3D objects. There are also ways to create 3D objects like globes and polyhedrons using SVG, which we'll get into a bit in Chapter 7 as we examine geospatial information visualization.

<CIRCLE> <RECT> <LINE> <POLYGON>

SVG provides a set of common shapes, each of which has **attributes** that determine their size and position in such a way that is easier to deal with than the generic "d" attribute we saw above. These attributes vary depending on the element that you're dealing with, so that the `<rect>` has "x" and "y" attributes that determine the shape's top-left corner, as well as "height" and "width" attributes that determine its overall form. In comparison, the `<circle>` element has a "cx" and a "cy" attributes that determine the center of the circle, and an "r" attribute that determines the radius of the circle. The `<line>` element has "x1" and "y1" attributes that determine the starting point of the line and "x2" and "y2" attributes that determine its end point. There are other simple shapes that are similar to these, such as the `<ellipsis>` and there are more complex shapes, like the `<polygon>` that has a "points" attribute that holds a set of comma-separated XY coordinates in clockwise order that determines the area bounded by the polygon.

Infoviz Vocabulary: Geometric Primitive

Accomplished artists can draw anything with vector graphics, but you're probably not looking at D3 because you're an artist. Instead, you're dealing with graphics with more pragmatic goals in mind. From that perspective, it's important to understand the concept of geometric primitives (also known as graphical primitives). Geometric primitives are simple shapes such as points, lines, circles and rectangles. These simple shapes, which can be combined to make more complex graphics, are particularly useful for visually displaying information.

Primitives are also useful for understanding complex information visualization that you see out in the world. Dendrograms like that seen in Figure 1.20 are far less intimidating when you realize they are just circles and lines. Interactive timelines are easier to understand and create when you think of them as collections of rectangles and points. Even geographic data, which primarily comes in the form of polygons, points and lines, is less confusing when you break it down into its most basic graphical structures.

Each of these attributes can be hand-edited in HTML to adjust their size, form and position. Open up your element inspector and click on the `<rect>`. Change its "width" to 25 and its "height" to 25 as shown in Figure 1.10.

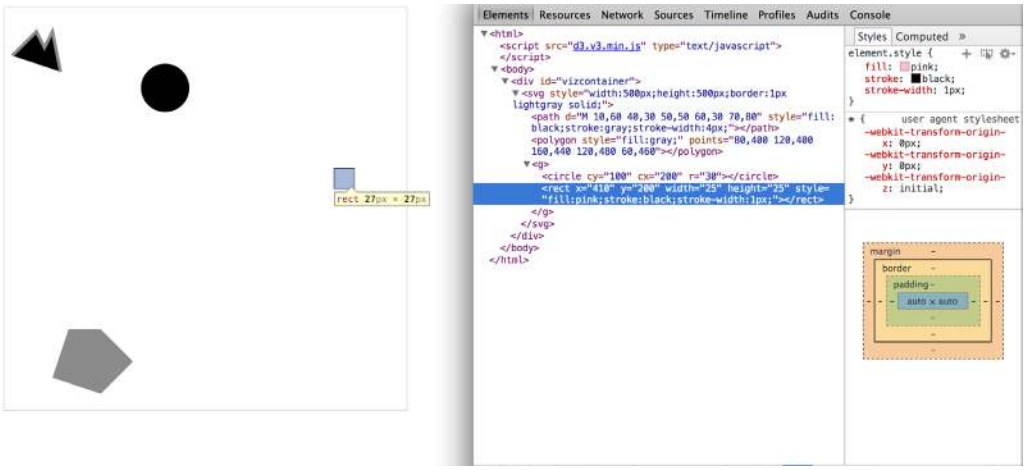

Figure 1.10 Modifying the height and width attributes of a <rect> element will change the appearance of that element. Inspecting the element also shows how the stroke adds to the computed size of the element.

Now you've learned why there is no SVG `<square>`. The color, stroke and transparency of any shape can be changed by adjusting the style of the shapes, with "fill" determining the color of the area of the shape while "stroke" and "stroke-width" and "stroke-dasharray" determine its outline.

Notice, though, that the inspected element has a measurement of 27px x 27px. That's because the 1px stroke is drawn on the outside of the shape. That makes sense, once you know the rule, but if you change the stroke-width to "2px" it will still be 27px x 27px. That's because the stroke is drawn evenly over the inside and outside border. This may not seem too big a deal, but it's something to remember when you're trying to line up your shapes later on.

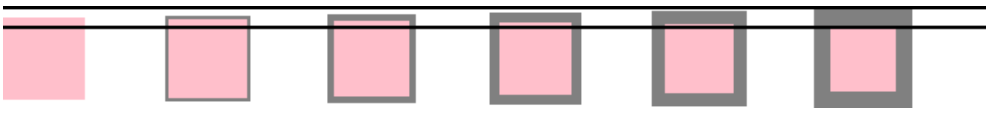

Figure 1.11 The same 25x25 <rect> with no, 1px, 2px, 3px, 4px and 5px stroke. Though these are being drawn on a retina screen using half-pixels, the second and third report the same width and height (27px x 27px) as do the fourth and fifth (29px x 29px).

Go ahead and change the style parameters of the rectangle from to:

`"fill:purple;stroke-width:5px;stroke:cornflowerblue;".`

Congratulations, you've now successfully visualized the complex and ambiguous phenomenon known as "ugly".

<TEXT>

SVG provides the capacity to write text as well as shapes. SVG text, though, does not have the formatting support found in HTML elements, and so it's primarily used for labels. If you do want to do some basic formatting, you can nest <tspan> elements within <text> elements.

<G>

The <g> or group element is distinct from the above SVG elements in that it has no graphical representation and does not exist as a bounded space. Instead, it is a logical grouping of elements. You'll want to use <g> elements extensively when creating graphical objects that are made up of several shapes and text. For instance, if you wanted to have a circle with a label above it and move the label and the circle at the same time, then you would place it inside a <g> element:

```
<g>
<circle r="2"/>
<text>This circle's Label</text>
</g>
```

Moving a <g> around your canvas requires you to adjust the "transform" attribute of the <g> element. The "transform" attribute is more intimidating than the various x/y attributes of shapes, since it accepts a structured description in text of how you want to transform a shape. One of those structures is "translate()" which will accept a pair of coordinates that will move the element to the x and y position defined by the values in "translate(x,y)". So if you want to move a <g> element 100 pixels to the right and 50 pixels down, then you need to set its "transform" attribute to transform="translate(100,50)". The transform attribute also accepts a "scale()" setting, so that you can change the rendered scale of the shape. We can see these settings in action by modifying the example above with the results shown in figure 1.12.

```
<g>
<circle r="2"/>
<text>This circle's Label</text>
</g>
<g transform="translate(100,50)">
<circle r="2" />
<text>This circle's Label</text>
</g>
<g transform="translate(100,50) scale(2.5)">
<circle r="2"/>
<text>This circle's Label</text>
</g>
```

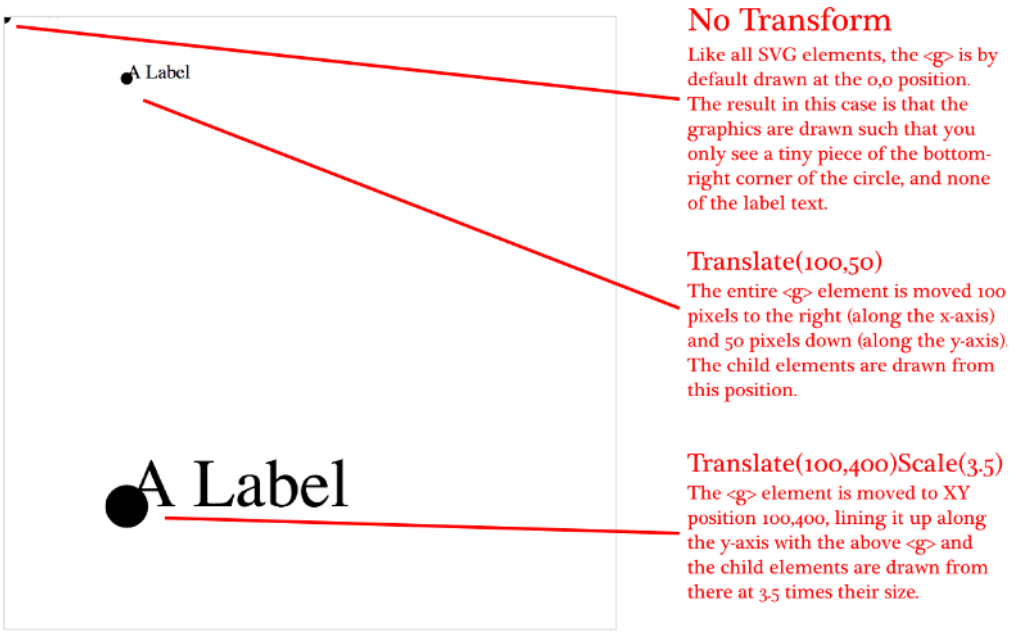

Figure 1.12 All SVG elements can be affected by the Transform attribute, but this is particularly salient when working with <g> elements, which require this approach to adjust their position. The child elements are drawn from using the position of their parent <g> as their relative 0,0 position. The scale() setting in the transform attribute then affects the scale of any of the size and position attributes of the child elements.

<PATH>

A path is an area determined by its "d" attribute. Paths can be open or closed, meaning the last point connects the first if closed and does not if open. The open or closed nature of a path is determined by the absence or presence of the letter "Z" at the end of the text string in the "d" attribute. It can still be filled either way. You can see the difference in figure 1.13.

```
<path style="fill:none;stroke:gray;stroke-width:4px;" d="M 10,60 40,30 50,50 60,30
70,80" transform="translate(0,0)" />
  <path style="fill:black;stroke:gray;stroke-width:4px;" d="M 10,60 40,30 50,50
60,30 70,80" transform="translate(0,100)" />
  <path style="fill:none;stroke:gray;stroke-width:4px;" d="M 10,60 40,30 50,50
60,30 70,80Z" transform="translate(0,200)" />
  <path style="fill:black;stroke:gray;stroke-width:4px;" d="M 10,60 40,30 50,50
60,30 70,80Z" transform="translate(0,300)" />
```

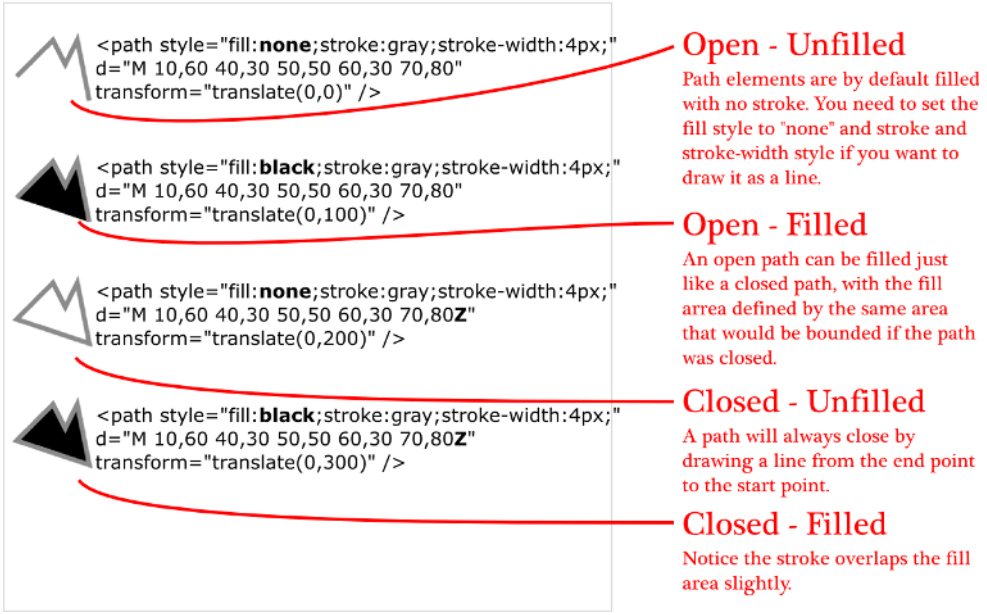

Figure 1.13 Each path shown here uses the same coordinates in its “d” attribute, with the only differences between them being the presence or absence of “Z” at the end of the text string defining the “d” attribute, the settings for fill and stroke, and the position via the “transform” attribute.

While there are times when you may want to write that "d" attribute yourself, it’s more likely that your experience crafting SVG will come in one of three ways: using geometric primitives like circles, rectangles or polygons; drawing SVG using a vector graphics editor like Adobe Illustrator or Inkscape; or drawing SVGs parametrically using hand-written constructors or built-in constructors in D3.Most of this book will focus on using D3 to create SVG, but don’t overlook the possibility of creating SVG in another application, such as Inkscape or Adobe Illustrator, and pasting it into your HTML, and then manipulating it using D3. This will allow you to integrate hand-drawn elements into your work. We’ll see this workflow later on in Chapter 3.

1.2.4 CSS

Cascading Style Sheets (CSS) are used to style the elements in the DOM. A stylesheet can exist as a separate .css file that you include in your HTML page or can be embedded directly in the HTML page. Stylesheets refer to an ID or Class or type of element and determine the appearance of that element. The terminology used to define the style is known as a CSS selector and is the same type of selector used in the `d3.select()` syntax. You can set inline styles (that are applied to only a single element) by using

`d3.select("#someElement").style("opacity", .5)` to set the opacity of an element to 50%. Let's update our d3ia.html to include a stylesheet as seen in Listing 1.2.

Listing 1.2 A sample web page with a stylesheet

```
<!doctype html>
<html>
<script src="d3.v3.min.js" type="text/JavaScript">
</script>
<style>
.inactive, .tentative {
  stroke: darkgray;
  stroke-width: 2px;
  stroke-dasharray: 5 5;
}

.tentative {
  opacity: .5;
}

.active {
  stroke: black;
  stroke-width: 4px;
  stroke-dasharray: 1;
}

circle {
  fill: red;
}

rect {
  fill: darkgray;
}
</style>
<body>
<div id="infovizDiv">
<svg style="width:500px;height:500px;border:1px lightgray solid;">
  <path d="M 10,60 40,30 50,50 60,30 70,80" />
  <polygon style="fill:gray;" />
<g>
<circle class="active tentative" cy="100" cx="200" r="30"/>
<rect class="active" x="410" y="200" width="100" height="50" />
</g>
</svg>
</div>
</body>
</html>
```

The results stack on each other, so when we examine the rectangle element, as shown in figure 1.14, we can see that its style is set by the reference to `rect` in the stylesheet as well as the reference to it having the class attribute of `active`.

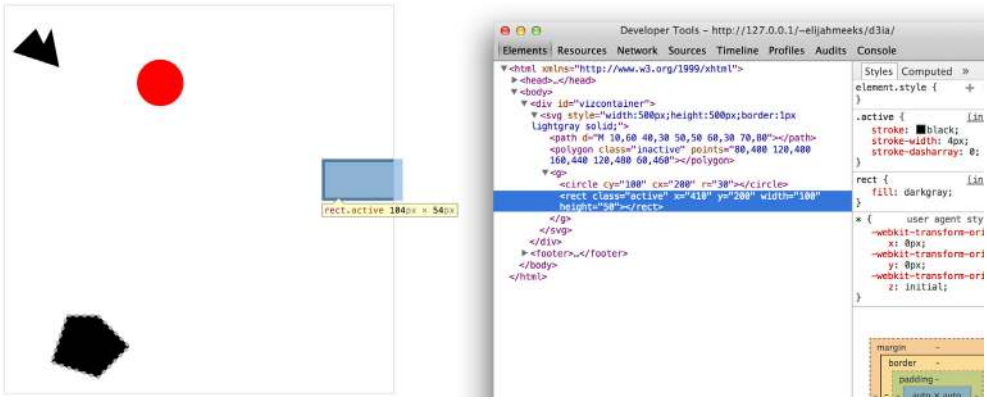

Figure 1.14 Examining an SVG rectangle in the console shows that it inherits its fill style from the CSS style applied to <rect> types and its stroke style from the .active class.

Stylesheets can also refer to a state of the element, so that with `:hover` you can change the way an element looks when the user mouses over that element. There are other complex CSS selectors that you can find out about in more detail in a book devoted to that subject. For this book, we'll focus mostly on using CSS classes and IDs for selection and to change style. The most useful way to do this is to have CSS classes associated with particular stylistic changes and then change the class of an element. You can change the class of an element, which is an attribute of an element, by selecting and modifying the class attribute. The circle shown in figure 1.15 is affected by two overlapping classes: .active and .tentative.

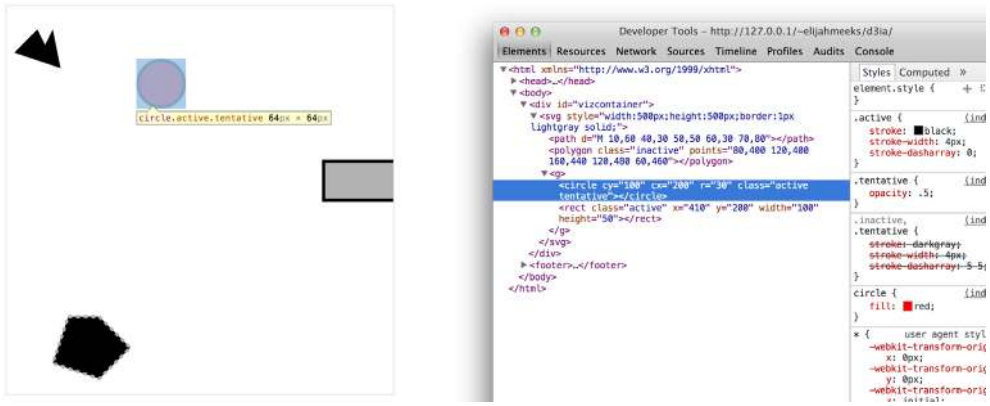

Figure 1.15 The SVG circle has its fill value set by its type in the stylesheet, with its opacity set by its membership in the .tentative class and its stroke set by its membership in the .active class. Notice that the stroke settings from the .tentative class are overwritten by the stroke settings in the later declared .active class.

Here we have a couple overlapping classes defined, with tentative, active, and inactive all applying different style changes to our shape. There are times when an element is just in one of these classes, in which case you could overwrite the class attribute entirely:

```
d3.select("circle").attr("class", "tentative");
```

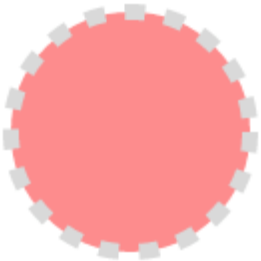

Figure 1.16 An SVG circle with fill style determined by its type and its opacity and stroke settings determined by its membership in the tentative class.

The results, as we see in Figure 1.16, are what we would expect. But this overwrites the entire class attribute to the value you set. But elements can have multiple classes and there are times when something is both active and tentative or inactive and tentative, so let's reload the page and take advantage of the helper function `d3.classed(), which` allows you to add or remove a class from the classes in an element:

```
d3.select("circle").classed("active", true);
```

By using `.classed()`, we don't overwrite the existing attribute, but rather append or remove the named class from the list. Here we can see the results of two classes with conflicting styles defined. The active style overwrites the tentative style because it occurs later in the stylesheet. Another rule to remember is that more specific rules overwrite more general rules. There's more to CSS, but there are other books for that.

By defining style in your stylesheet and changing appearance based on class membership, you create code that is more maintainable and readable. You'll need to use inline styles to set the graphical appearance of a set of elements to a variety of different values, such as changing the fill color to correspond to a color ramp based on the data bound to that set of elements. We'll see that functionality in action later as we deal with bound data. But as a general rule setting inline styles should only be used when you can't use traditional classes and states defined in a stylesheet.

1.2.5 JavaScript

D3, like many information visualization libraries in JavaScript, provides a host of helpful functions to abstract the process of creating and modifying elements to make it more programmer friendly. On top of that, it provides mechanisms to link data and web page elements in a way that makes the drawing and updating of these SVG elements reusable and

maintainable. But these mechanisms are also applicable to more traditional HTML elements like paragraphs and divs.

As a result, a web application written in D3 can accomplish much of the user interface functionality that users expect without relying on libraries like jQuery. This is because the latest version of JavaScript has built-in so much of the functionality that once required the custom code to sort arrays or move elements. If you read the solutions in StackOverflow, you may think that being a JavaScript developer requires being a jQuery developer, but look more closely and you'll see that unless you're developing for an audience using a browser from the Bush administration, which by necessity means it won't support the key features of D3, or you need to use a plugin that requires jQuery, then you might ust as easily write the same functionality in plain JavaScript.

There are two things that you should be aware of right off the bat, method chaining and arrays:

METHOD CHAINING

D3 examples, like many examples written in JavaScript, use method chaining extensively. Method chaining, also known as function chaining, is facilitated by returning the method itself with the successful completion of functions associated with a method. One way to think of method chaining is to think of how we talk and refer to each other. Imagine you were talking to someone at a party, and you asked about another guest:

"What's her name?"
"Her name is Lindsay."
"Where does she work?"
"She works at Tesla."
"Where does she live?"
"She lives in Cupertino."
"Does she have any children?"
"Yes, she has a daughter."
"What's her name?"

Do you think the answer to that last question would be "Lindsay"? Of course not, we'd expect the answer to refer to Lindsay's daughter, even though all the previous questions referred to Lindsay. Method chaining is like that. Method chaining is used a lot in D3 examples, which means you'll see something like this written on one line or formatted (but functionally identical) to something written on multiple lines:

```
d3.selectAll("div").data(someData).enter().append("div").html("Wow").append("span"
).html("Even More Wow").style("font-weight", "900")
```

It's the same as what's below. The only change is in the use of line breaks, which JavaScript ignores:

```
d3.selectAll("div") #a
.data(someData) #b
```

```
.enter() #c
.append("div") #d
.html("Wow") #e
.append("span") #f
.html("Even More Wow") #g
.style("font-weight", "900"); #h
```

#a Returns the Function 1, a selection
#b Sets the data on Function 1 and returns Function 1
#c Returns Function 2, the selection.enter() function
#d Sets .append() behavior on function 2 and returns function 3, a selection
#e Sets .html() for function 3 and returns function 3
#f Sets the append() behavior on function 3 and returns function 4
#g Sets the html() for function 4 and returns function 4
#h Sets the font-weight style on function 4 & returns function 4

You could write each line separately, declaring the different variables as you go, and achieve the same effect. It might also make method chaining make a little more sense to you if you haven't been exposed to it before:

```
var function1 = d3.selectAll("div");
function1.data(someData);
var function2 = function1.enter();
var function3 = function2.append("div");
function3.html("Wow");
var function4 = function3.append("span");
function4.html("Even More Wow");
function4.style("font-weight", "900");
```

We can see this in action by running this in our console. This is the first time we've used the `.data()` function, which along with `.select()` is at the core of developing with D3. When you use `.data()` you're binding each element in your selection to each item in an array. If you have more items in your array than elements in your selection, then you can use the `.enter()` function to define what to do with each extra element. In the above function, we're selecting all the elements in the `<body>` and the `.enter()` function tells D3 to `.append()` a new div when there are more elements in the array than elements in the selection. Given that our d3ia.html page already has one div, this means if we bind an array with more than one value, D3 will append, or add, a div for each value in the array beyond the first.

There's a corresponding `.exit()` function that defines how to respond when there are fewer values in an array than there are elements in a selection. For now, we're just going to run the code as it appears in the examples, and in later chapters we'll get into much more detail on the way selections and binding work.

With this example, we're not doing anything with the data in the array and are only creating elements based on the size of the array (one `<div>` for each element in the array). For this to work, you need to give someData a value. With that in place, you can run your code:

```
var someData = ["filler", "filler", "filler", "filler"];
d3.select("body").selectAll("div").data(someData).enter().append("div").html("Wow"
).append("span").html("Even More Wow").style("font-weight", "900");
```

Figure 1.17 By binding an array of four values to a selection of elements on the page, we can see that the .enter() function created three new elements to reflect the size mismatch between the data array and the selection.

The result, as seen in figure 1.17, is the addition of three lines of text. It might surprise you that there are three lines, given that the array has four values, but that's because while the data was bound to the existing `<div>` element on the page, the actions defined to change the contents were only applied to the `.enter()` function, which means that they were only applied to the newly created `<div>` elements.

Figure 1.18 Inspecting the DOM shows that the new elements have been created with unformatted content followed by the child element with style and content set by your code.

When we inspect the DOM as shown in figure 1.18, we can see that the method chaining operated in the manner described above. A `<div>` was added, it's HTML content was set to "Wow" and a `<span>` element with a different style was appended to the `<div>` and its HTML content was set to "Even More Wow". Despite how exciting this all may or may not seem (six wows is exciting in my book, but people are hard to impress) there's a lot more we can do. But first, we need to examine the array object we're binding, and focus on JavaScript arrays and array functions.

ARRAYS AND ARRAY FUNCTIONS

D3 is all about arrays and so it's important to understand the structure of arrays and the options available to you to prepare those arrays for binding to data. Your array might just be an array of string or number literals, such as this:

```
someNumbers = [17, 82, 9, 500, 40]
someColors = ["blue", "red", "chartreuse", "orange"]
```

Or it may be an array of JSON objects, which will become more common as you want to do more interesting things with D3:

```
somePeople = [{name: "Peter", age: 27}, {name: "Sulayman", age: 24}, {name:
"K.C.", age: 49}]
```

In either case, one example of an extremely useful array is `.filter()`, which returns an array wherein the elements in that array satisfy a test you provide. For instance, here's how to create an array out of `someNumbers` that had values greater than 40:

```
someNumbers.filter(function(el) {return el >= 40})
```

Likewise, here's how you could create an array out of someColors with words shorter than five letters:

```
someColors.filter(function(el) {return el.length < 5})
```

The function `.filter()` is a method of an array and accepts a function that iterates through the array with the variable you've named. In the function above, I named that variable "el" and the function runs a test on each value by testing on "el". When that test evaluates true, the element is kept in our new array.

The result of this `.filter()` function, which you can see in figure 1.19, returns either the element or nothing depending on if it satisfies the test, building a new array only made up of the elements that do.

Figure 1.19 Running JavaScript in the console allows you to test your code. Here we've created a new array called smallerNumbers that consists of only three values, which we can then use as our data in a selection to update and create new elements.

```
smallerNumbers = someNumbers.filter(function(el) {return el >= 40 ? this : null})

d3.select("body").selectAll("div").data(smallerNumbers).enter().append("div").html
(function (d) {return d})
```

The resulting code creates two new divs from your three value array smallerNumbers (remember that one div already exists and so the `.enter()` function does not trigger though data is bound to it) and the contents of the div are the values in your array. This is done through an anonymous function (sometimes referred to in D3 examples as an "accessor") in your `.html()` function and is another key aspect of utilizing D3. Any anonymous function

called when setting the `.style()`, `.attr()`, `.property()`, `.html()` or other function of a selection can provide you with the data bound to that selection. As you explore examples, you'll see this function deployed over and over again:

```
.style("background", function(d,i) {return d})
.attr("cx", function(d,i) {return i})
.html(function(d,i) {return d})
```

In every case, the first variable (typically represented with the letter "d" but you can declare it as whatever you want) will contain the data value bound to that element, and the second variable will return the array position of the value bound to that element. This may seem a bit strange, but you'll get used to it as you see it utilized in a variety of ways in the upcoming chapters.

There are, of course, many other array functions, and much more you can do with JavaScript than was covered here, but that's the subject of several other books.

1.3 Data Standards

Standardization of methods of displaying data has been fed by and feeds into standardization of methods of formatting that data. Data can be formatted in a variety of manners for a variety of purposes, but it tends to fall within a few recognizable classes: Tabular data, nested data, network data, geographic data, raw data and objects.

1.3.1 Tabular Data

Tabular data refers to data appearing in columns and rows typically found in a spreadsheet or a table in a database. While you invariably end up creating arrays of objects in D3, it is often more efficient and simply easier to pull in data in tabular format. Tabular data is delimited with a particular character, and that delimiter determines its format. So you can have a CSV, which stands for Comma-Separated Values, where the delimiter is a comma, or tab-delimited values, or a semicolon or a pipe symbol acting as the delimiter. For instance, you may have a spreadsheet of user information that includes age and salary. If you export it in a delimited form, it will look like table 1.1.

Table 1.1 Delimited data can take different forms expressing the same data. Here we see a dataset storing name, age, and a salary of two people using commas, spaces or the bar symbol to delimit the different fields.

name,age,salary	name age salary	name|age|salary
Sal,34,50000	Sal 34 50000	Sal|34|50000
Nan,22,75000	Nan 22 75000	Nan|22|75000

D3 provides three different functions to deal with pulling in tabular data: `d3.csv()`, `d3.tsv()` and `d3.dsv()`. The only difference between them is `d3.csv()` is built for comma-delimited files,

`d3.tsv()` is built for tab-delimited files and `d3.dsv()` allows you to declare the delimiter. We'll see them in action throughout the book.

1.3.2 Nested Data

Data that appears in a nested manner, wherein objects exist as children of objects recursively, is very common. Many of the most intuitive layouts in D3 are based on nested data, which can be represented as trees, such as the one in figure 1.20, or packed in circles or boxes. Data is not often output in such a format, and requires a bit of scripting to organize it as such, but the flexibility of representation of such a format is worth the effort. We will see hierarchical data in detail in Chapter 4 as we look at various popular D3 layouts.

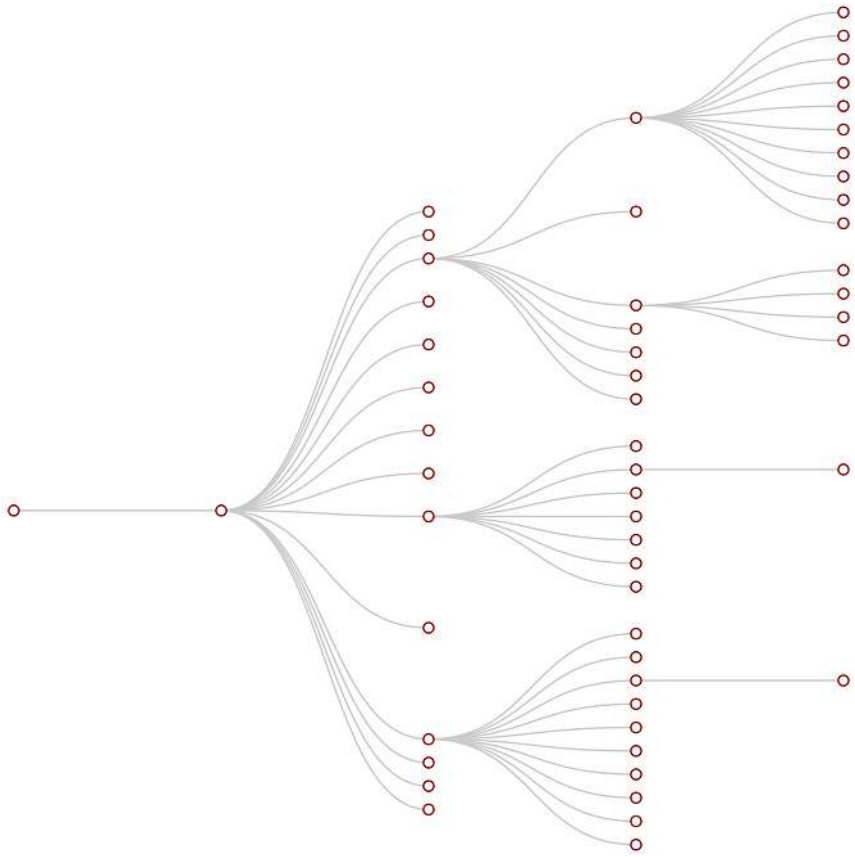

Figure 1.20 Nested data represents parent/child relationships of objects, typically with each object having an array of child objects, and is represented in a number of forms, such as this dendrogram. Notice that each object can only have one parent.

1.3.3 Network Data

Networks are everywhere. Whether they are the raw output of our social networking streams or transportation networks or a flowchart, networks are a powerful method of delivering to users an understanding of complex systems. Networks are often represented as node-link diagrams like that seen in figure 1.21. Like geographic data, there are many standards for network data, but this text will focus only on two forms: node/edge lists and connected arrays. As with geodata, network data in many forms can be easily transformed into these data types by using a freely available network analysis tool like Gephi (available at gephi.org). We will examine network data and network data standards as we deal with network visualization in Chapter 6.

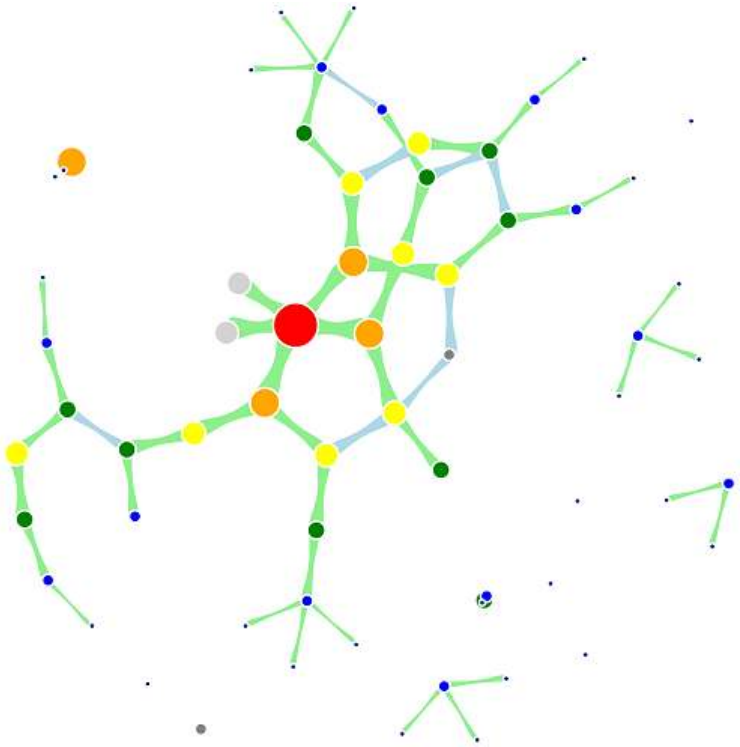

Figure 1.21 Network data consists of objects and the connections between them. The objects are typically referred to as nodes or vertices, while the connections are referred to as edges or links. Networks are often represented using force-directed algorithms such as the example here, that arrange the network in such a way to pull connected nodes toward each other.

1.3.4 Geographic Data

Geographic data consists of data that refers to locations either as points or shapes, and is used to create the variety of online maps seen on the web today, such as the map of the

United States in figure 1.22. The incredible popularity of on-line mapping means that you can get access to a massive amount of publicly accessible geodata for any project. There are a few standards but the focus in this book will be on two: the GeoJSON and topoJSON standards. While geodata may come in many forms, the use of readily available geographic information systems (GIS) tools like Quantum GIS allow developers to transform it into this format for ready delivery to the web. We'll look at geographic data closely in Chapter 7.

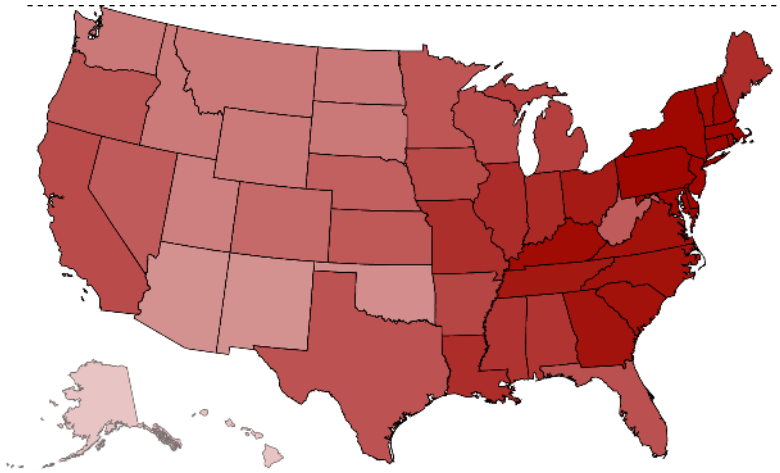

Figure 1.22 Geographic data stores the spatial geometry of objects, such as states. Each of the states in this image is represented as a separate feature with an array of values indicating their shape. Geographic data can also consist of points, such as for cities, or lines, such as for roads.

1.3.5 Raw Data

As we'll deal with in chapter two, everything is data, and this includes images or blocks of text. An important thing to remember with D3 is that while information visualization typically is thought of as using shapes encoded by color and size to represent data, sometimes the best way to represent it is with linear narrative text, an image or a video. If you plan to develop applications for an audience that needs to understand complex systems but consider the manipulation of text or images to be somehow separate from the representation of numerical or categorical data as shapes, then you arbitrarily reduce your capability to communicate. The layouts and formatting that come along with dealing with text and images, typically thought of as tied to older modes of web publication, are possible in D3, and we'll deal with that throughout this book, but especially in Chapter 8.

1.3.6 Objects

There are two types of data points that you'll be using with D3: Literals and Objects. A literal, such as a string literal like "Apple" or "beer" or a number literal like 64273 or 5.44, is pretty straightforward. A JavaScript object, expressed using JavaScript Object Notation (JSON), is

not so straightforward but something that you need to understand if you plan to do sophisticated data visualization.

Let's say you have a dataset that consists of individuals from an insurance database, where you need to know how old someone is, whether they're employed, their name, and their children, if any. A JSON object that represents each individual in such a database would be expressed as follows:

```
{name: "Charlie", age: 55, employed: true, childrenNames: ["Ruth", "Charlie Jr."]}
```

Each object is surround by { } and has attributes that have a string, number, array, boolean or object as their value. You can assign an object to variable and access its attributes by referring to them, like so:

```
var person = {name: "Charlie", age: 55, employed: true, childrenNames: ["Ruth",
"Charlie Jr."]};

person.name // Charlie
person["name"] // Charlie
person.name = "Charles" // Sets name to Charles
person["name"] = "Charles" // Sets name to Charles
person.age < 65 // true
person.childrenNames // ["Ruth", "Charlie Jr."]
person.childrenNames[0] // "Ruth"
```

Objects can be stored in arrays and associated with elements using d3.select() syntax. But objects can also be iterated through like arrays using a for loop:

```
for (x in person) {console.log(x); console.log(person[x]);}
```

The x in the loop is the attribute name, and `person[x]` is the same as using the `person["name"]` syntax above to show the value of that attribute of the object.

If your data is stored as JSON, then you can import it using `d3.json()`, which is demonstrated many times in later chapters. But remember that whenever we use `d3.csv()` or other functions to pull in data formatted in other ways, D3 imports the data as an array of JSON objects. We'll look at objects more extensively as we use them later.

1.4 Infoviz Standards Expressed in D3

The chances are good that you're interested in D3 because of data or information visualization. Information visualization has never been so popular as it is today. The wealth of maps and charts and complex representations of systems and datasets is not just present in the workplace but in our entertainment and everyday lives. With this popularity comes a growing library of classes and sub-classes of representation of data and information using visual means, as well as aesthetic rules to promote legibility and comprehension. Your audience, whether it is the general public, or academics, or decision makers, has grown accustomed to what we once would have thought of as incredibly abstract and complicated representations of trends in data. This is why libraries like D3 aren't just popular among data scientists but also among journalists, artists, scholars, IT professionals and even fan communities.

But the wealth of options can seem overwhelming and the relative ease of modifying a dataset to appear in a streamgraph or treemap or even a simple histogram tends to promote the idea that information visualization is more about style than substance. Fortunately, there are well-established rules for what charts and methods to use for different types of data from different systems. While I can't possibly cover every aspect of this in the book, I'll touch on things that will be useful to consider as we create more complicated information visualizations. While some developers are using D3 to revolutionize the use of color and layout, most simply want to create visual representations of data that support practical concerns. Because D3 is being developed in this mature information visualization environment, it contains numerous helper functions to let developers worry about interface and design rather than color and axes.

Still, to properly deploy information visualization, you should be familiar with what to do and what not to do. The best way to do so is to review the work of established designers and information visualization practitioners. This requires you to have a firm understanding not only of your data but of your audience. While there is an entire library of works that deal with these issues, here are a few that I've found useful and can get you oriented on the basics:

Edward Tufte	***The Visual Display of Quantitative Information***
	Envisioning Information
Isabel Meirelles	***Designing for Information***
Christian Swineheart	***Pattern Recognition***

These are by no means the only or most applicable texts for learning data visualization, but I've found them very useful, especially for getting started. The best advice is to pare down and establish fundamental, even basic, data visualization practices that clearly represent the trends that are salient to your audience. When in doubt, simplify--it is often better to present a histogram than a streamgraph, or a hierarchical network layout (like a dendrogram) than a force-directed one. The more visually complex methods of displaying data tend to inspire more excitement, but can also promote an audience seeing what they want to see or focusing on the aesthetics of the graphics rather than the data.

Infoviz Tip: Kill Your Darlings

One of the best pieces of advice when it comes to working in information visualization comes from the practice of writing: "Kill your darlings". Just as writers may become enamored of certain scenes or characters, we become enamored of a particularly elegant or sophisticated-looking graphics. Our love of a cool chart or animation can distract us from the goal of communicating the structure and patterns in the data. So, remember to save your harshest criticism for your most beloved pieces, because you may find, much to your chagrin, that they're not as useful and informative as you think they are.

One thing to keep in mind while reading up on data visualization is that the literature is often focused on static charts. With D3 you'll be making interactive and dynamic visualizations and not just static ones. You'll make a dynamic (or animated) data visualization before you finish this chapter, and using D3 to make a chart interactive is incredibly simple. Just a few interactive touches can not only make a visualization more readable but significantly more engaging, and a user who feels like they're exploring rather than reading, even if only with a few mouseover events or a simple click to zoom, will find the content of the visualization more compelling than that in a static page. But this added complexity requires an investment in learning principles of interface design and user experience. We'll get into this in more detail in Chapter 9 and Chapter 12.

1.5 Your First D3 App

Throughout this chapter, you've seen various lines of code and hopefully the effect of those lines of code on the growing d3ia.html sample page we've been building. But I've avoided explaining the code in too much detail so that we could concentrate on the principles at work in D3. By now, you've probably realized that it's simple enough to build an application from scratch that uses D3 to create and modify elements. Let's put it all together and see how it works. First, let's start with a clean HTML page that doesn't have any styles defined or existing divs.

```
<!doctype html>
<html>
<head>
<script>d3v3.min.js</script>
</head>
<body>
</body>
<html>
```

1.5.1 Hello World With Divs

With that in place, you can use D3 as an abstraction layer for adding traditional content to the page. While you can write JavaScript inside the your .html file or in its own .js file. For now, let's just put some code in the console and see how it works. Later, we'll focus on the various commands and get into them in more detail as they're applied to layouts and interfaces. We can get started with a very simple piece of code that uses D3 to write to your web page:

```
d3.select("body").append("div")
.style("border", "1px black solid")
.html("hello world");
```

We can adjust the element on the page and give it some interactivity with the inclusion of the .on() function:

```
d3.select("div")
.style("background-color", "pink")
.style("font-size", "24px")
.attr("id", "newDiv")
.attr("class", "d3div")
.on("click", function() {console.log("You clicked a div")})
```

The `.on()` method allows you to create an event listener for the currently selected element or set of elements. It accepts the variety of events that can happen to an element, such as "click", "mouseover", "mouseout" and so on. If you click on your div, you'll notice that it gives a response in your console as shown in figure 1.23.

Figure 1.23 Using console.log() you can easily test to see if an event is properly firing. Here we create a and assign an onclick event handler using the .on() syntax. When we click on that element and fire the event, the action is noted in the console.

1.5.2 Hello World with Circles

You probably didn't pick up this book just to learn how to add divs to a webpage, though, and likely want to deal graphics like lines and circles. To append shapes to a page with D3, remember that you need to have an SVG canvas element somewhere in your page's DOM. You could either add this SVG canvas to the page as you were writing the HTML. You could append it using the D3 syntax we've learned:

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

But for now let's adjust our d3ia.html page to start with an SVG canvas:

```
<!doctype html>
<html>
<head>
<script>d3v3.min.js</script>
</head>
<body>
<div id="vizcontainer">
<svg style="width:500px;height:500px;border:1px lightgray solid;" />
</div>
</body>
<html>
```

Once you have an SVG canvas on your page, you can append various shapes to it using the same select() and append() syntax we've been using in 1.5.1 for `<div>` elements:

```
d3.select("svg").append("line").attr("x1", 20).attr("y1",
20).attr("x2",400).attr("y2",400).style("stroke", "black").style("stroke-
width","2px");

d3.select("svg").append("text").attr("x",20).attr("y",20).text("HELLO");

d3.select("svg").append("circle").attr("r",
20).attr("cx",20).attr("cy",20).style("fill","red");

d3.select("svg").append("circle").attr("r",
100).attr("cx",400).attr("cy",400).style("fill","lightblue");

d3.select("svg").append("text").attr("x",400).attr("y",400).text("WORLD");
```

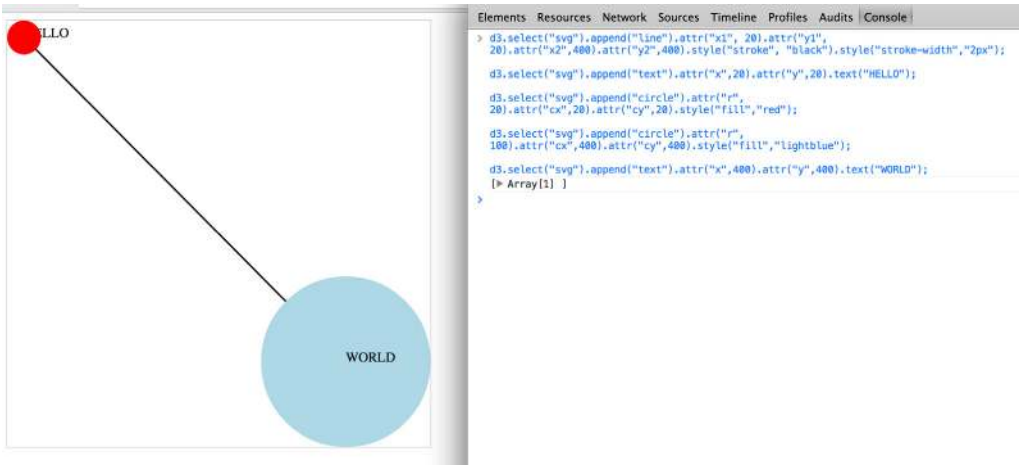

Figure 1.24 The results of running the above code in the console is the creation of two circles, a line, and two text elements. The order in which these elements are drawn results in the first label covered by the circle drawn later.

Notice that your circles are drawn over the line and the text is drawn above or below the circle depending on the order that you run your commands as you can see in figure 1.24. Recall that this is because the draw order of SVG is based on its DOM order. There are some methods we'll learn later on to adjust that order.

1.5.3 A Conversation with D3

There are so many examples of writing Hello World with languages that I thought we should give the world a chance to respond. Let's add the same big circle and little circle from before but this time, when we add text, notice the inclusion of the `.style("opacity")` setting that makes our text invisible. We've also given each of them a `.attr("id")` setting so that the text

near the small circle has an "id" attribute with the value of "a" and the text near the large circle has an "id" attribute with the value of "b".

```
d3.select("svg").append("circle").attr("r",
20).attr("cx",20).attr("cy",20).style("fill","red");

d3.select("svg").append("text").attr("id", "a").attr("x",20).style("opacity",
0).attr("y",20).text("HELLO WORLD");

d3.select("svg").append("circle").attr("r",
100).attr("cx",400).attr("cy",400).style("fill","lightblue");

d3.select("svg").append("text").attr("id",
"b").attr("x",400).attr("y",400).style("opacity", 0).text("Uh, hi.");
```

Two circles, no line, and no text. So now we make the text appear using the `.transition()` method with the `.delay()` method and we should have an end state like the one seen in figure 1.25.

```
d3.select("#a").transition().delay(1000).style("opacity", 1);
d3.select("#b").transition().delay(3000).style("opacity", .75);
```

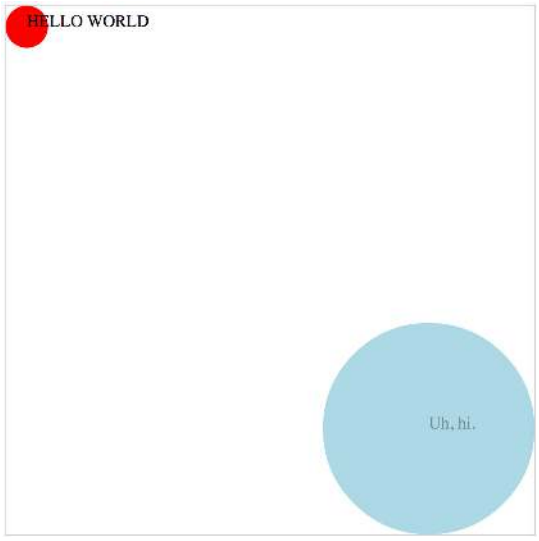

Figure 1.25 Transition behavior when associated with a delay will result in a pause before the application of the attribute or style.

Congratulations, you've made your first dynamic data visualization. The `.transition()` method indicates that you don't want your change to be instantaneous, and by chaining it with the `.delay()` method we indicate how many milliseconds we want to wait before implementing the style or attribute changes that appear in the chain after that .delay() setting.

We'll get a bit more ambitious later on but before we finish here, let's look at the another `.transition()` setting. Along with being able to set a `.delay()` before which the new style or attribute is applied, you can set a .duration() over which the change is applied. The results in your browser should move the shapes in the direction of the arrows in figure 1.26.

```
d3.selectAll("circle").transition().duration(2000).attr("cy", 200);
```

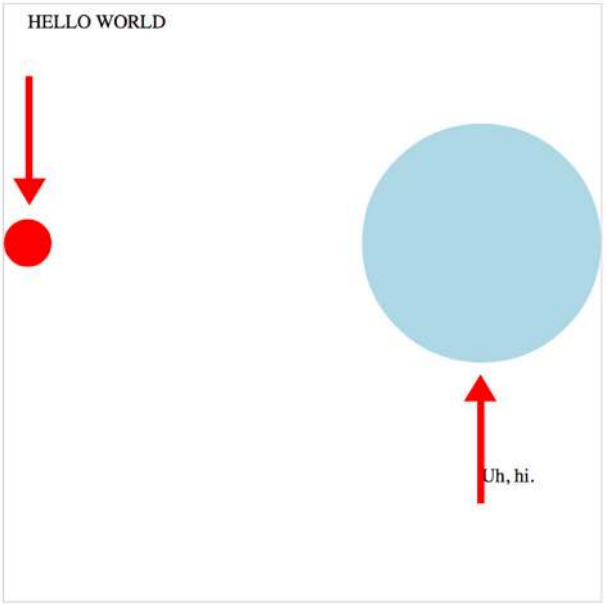

Figure 1.26 Transition behavior when associated with position will make the shape graphically move to its new position over the course of the assigned duration. Since we used the same "y" position for both circles, this results in the first circle moving down and the second circle moving up the the "y" position we've set, which is between the two circles.

The `.duration()` method, as you can see, adjusts the setting over the course of the amount of time (again, in milliseconds) that you set it for.

That covers the basics of how D3 works and how it's designed, and these fundamental concepts will come back again and again throughout the following chapters, where we'll learn about more complicated variations on representing and manipulating data.

1.6 Summary

In this chapter you've received a brief overview as to what D3 is and how it is particularly well-suited for developers building web applications for the modern browser. To do so, I've highlighted the the standardizations and advances that allow this all to happen. Data standards, practices for dealing with data, and the improved performance of JavaScript and HTML5 all factor into making this possible.

D3.js is just another JavaScript library, one of thousands, but it is also indicative of a change in our expectations of what a web page can do. While you may initially explore D3 as a method for building one-off data visualizations of a highly-processed static dataset, it has built in so much more power and functionality than that leverages the functionality of modern web standards to allow you to create interactive data-driven documents. These documents will improve your ability to communicate with every audience with reusable and robust methods.

D3 itself has many strengths, some of which are simply its being positioned as a library for modern web development, but D3 application development brings with it more than that: As you begin to understand the processes and functions that D3 uses to manipulate, compute, and represent data, even for one-off sites, you'll find that these methods will translate immediately to your work in all aspects of web development, whether dealing with traditional layout and design issues or in more complex interactive applications.