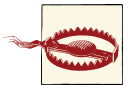
CHAPTER 3
Technology Fundamentals

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



Beware! This is a pretty dense chapter, packed with years' worth of web development knowledge, and nothing in here is specific to D3. I recommend skimming just the sections on information that is new to you, and skipping the rest. You can always reference this chapter later as questions arise.

The Web

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

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

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

> CLIENT: I'd really like to know what's going on over at *somewebsite.com*. I better call over there to get the latest info. [Silent sound of Internet connection being established.]
>
> SERVER: Hello, unknown web client! I am the server hosting *somewebsite.com*. What page would you like?
>
> CLIENT: This morning, I am interested in the page at *somewebsite.com/news/*.
>
> SERVER: Of course, one moment.

Code is transmitted from SERVER to CLIENT.

CLIENT: I have received it. Thank you!

SERVER: You're welcome! Would love to stay on the line and chat, but I have other requests to process. Bye!

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

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

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

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

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

Complete URLs consist of four parts:

- An indication of the *communication protocol*, such as HTTP or HTTPS
- The *domain name* of the resource, such as *calmingmanatee.com*
- The *port number*, indicating over which port the connection to the server should be attempted
- Any additional locating information, such as the path of the requested file, or any query parameters

A complete URL, then, might look like this: *http://alignedleft.com:80/tutorials/d3/*.

Typically, the port number is excluded, as web browsers will try to connect over port 80 by default. So the preceding URL is functionally the same as the following: *http://alignedleft.com/tutorials/d3/*

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

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

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

1. User runs the web browser of her choice, then types a URL into the address bar, such as *alignedleft.com/tutorials/d3/*. Because she did not specify a protocol, HTTP is assumed, and "http://" is prepended to the URL.
2. The browser then attempts to connect to the server behind *alignedleft.com* across the network, via port 80, the default port for HTTP.
3. The server associated with *alignedleft.com* acknowledges the connection and is taking requests. ("I'll be here all night.")
4. The browser sends a request for the page that lives at */tutorials/d3/*.
5. The server sends back the HTML content for that page.
6. As the client browser receives the HTML, it discovers references to *other files* needed to assemble and display the entire page, including CSS stylesheets and image files. So it contacts the same server again, once per file, requesting the additional information.
7. The server responds, dispatching each file as needed.
8. Finally, all the web documents have been transferred over. Now the client performs its most arduous task, which is to *render* the content. It first parses through the HTML to understand the structure of the content. Then it reviews the CSS selectors, applying any properties to matched elements. Finally, it plugs in any image files and executes any JavaScript code.

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

HTML

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

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
```

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

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

Content Plus Structure

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

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

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

Amazing Visualization Tool Cures All Ills

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

- fevers
- chills
- general malaise

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

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

That has the same raw text content, but with a *visual structure* that makes the content more accessible.

HTML is a tool for specifying *semantic structure*, or attaching hierarchy, relationships, and *meaning* to content. (HTML doesn't address the visual representation of a

document's structure—that's CSS' job.) Here is our story with each chunk of content replaced by a *semantic description* of what that content is.

Headline

Paragraph text

- Unordered list item
- Unordered list item
- Unordered list item

Paragraph text

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

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

Adding Structure with Elements

"Marking up" is the process of adding *tags* to create *elements*. HTML tags begin with < and end with >, as in , which is the tag indicating a paragraph of text. Tags usually occur in pairs, in which case adding an opening and closing pair of tags creates a new *element* in the document structure.

Closing tags are indicated with a slash that closes or ends the element, as in . Thus, a paragraph of text may be marked up like the following:

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

Some elements can be *nested*. For example, here we use the em element to add *emphasis*.

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

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

When elements are nested, they cannot overlap closures of their parent elements, as doing so would disrupt the hierarchy. For example:

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

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

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

As of HTML5, the self-closing slash is optional, so the following code is equivalent to the preceding code:

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

Common Elements

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

`<!DOCTYPE html>`
: The standard document type declaration. Must be the first thing in the document.

`html`
: Surrounds all HTML content in a document.

`head`
: The document `head` contains all metadata about the document, such as its `title` and any references to external stylesheets and scripts.

`title`
: The title of the document. Browsers typically display this at the top of the browser window and use this title when bookmarking a page.

`body`
: Everything not in the `head` should go in the `body`. This is the primary visible content of the page.

`h1`, `h2`, `h3`, `h4`
: These let you specify headings of different levels. `h1` is a top-level heading, `h2` is below that, and so on.

`p`
: A paragraph!

`ul`, `ol`, `li`
: Unordered lists are specified with `ul`, most often used for bulleted lists. Ordered lists (`ol`) are often numbered. Both `ul` and `ol` should include `li` elements to specify list items.

`em`
: Indicates emphasis. Typically rendered in *italics*.

`strong`
: Indicates additional emphasis. Typically rendered in **boldface**.

`a`
: A link. Typically rendered as underlined, blue text, unless otherwise specified.

`span`
: An arbitrary `span` of text, typically within a larger containing element like `p`.

`div`
: An arbitrary *division* within the document. Used for grouping and containing related elements.

Our earlier example could be given semantic structure by marking it up using some of these element's tags:

```
<h1>Amazing Visualization Tool Cures All Ills</h1>
<p>A new open-source tool designed for visualization of data turns out to have
an unexpected, positive side effect: it heals any ailments of the viewer.
Leading scientists report that the tool, called D3000, can cure even the
following symptoms:</p>
<ul>
    <li>fevers</li>
    <li>chills</li>
    <li>general malaise</li>
</ul>
<p>It achieves this end with a patented, three-step process.</p>
<ol>
    <li>Load in data.</li>
    <li>Generate a visual representation.</li>
    <li>Activate magic healing function.</li>
</ol>
```

When viewed in a web browser, that markup is rendered as shown in Figure 3-1.

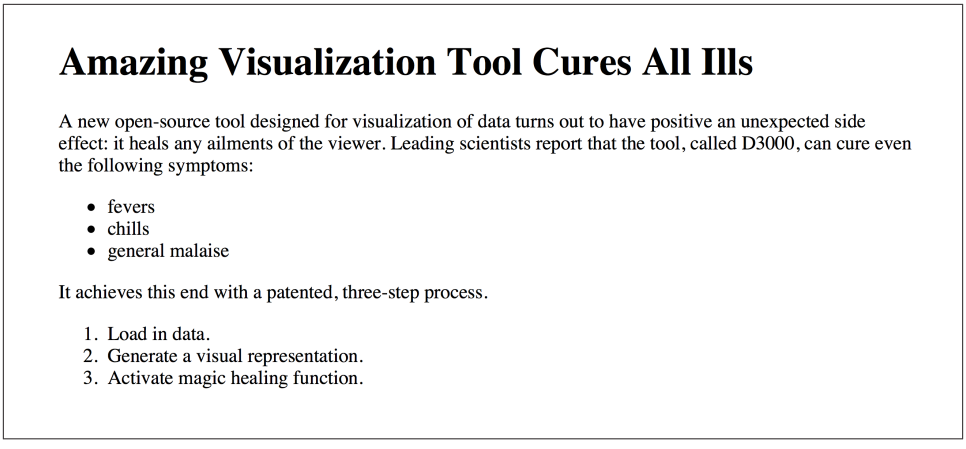


Figure 3-1. Typical default rendering of simple HTML

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

Attributes

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

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

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

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

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

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

Classes and IDs

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

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

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

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

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

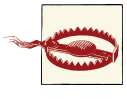
Now, all three paragraphs are `uplifting`, but only the last one is both `uplifting` *and* `awesome`.

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

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

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

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



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

Comments

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

In HTML, comments are written in the following format:

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

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

DOM

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

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

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

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

Developer Tools

In the olden days, the web development process went like this:

1. Write some code in a text editor.
2. Save the files.
3. Switch to a browser window.
4. Reload the page, and see if your code worked.
5. If it didn't work, take a guess at what went wrong inside the magical black box of the web browser, then go back to step 1.

Browsers were notoriously secretive about what went on *inside* the rendering engine, which made debugging a total nightmare. (Seriously, in the late 1990s and early 2000s, I literally had nightmares about this.) Fortunately, we live in a more enlightened age, and every modern-day browser has built-in *developer tools* that expose the inner workings of the beast and enable us to poke around under the hood (to mix incompatible metaphors).

All this is to say that developer tools are a big deal and you will rely on them heavily to both test your code and, when something breaks, figure out what went wrong.

Let's start with the simplest possible use of the developer tools: viewing the raw source code of an HTML page (see Figure 3-2).

Every browser supports this, although different browsers hide this option in different places. In Chrome 23.0, it's under View→Developer→View Source. In Firefox 17.0, look under Tools→Web Developer→Page Source. In Safari 6.0, it's under Develop→Show Page Source (although you must first set the "Develop" menu to display under Safari→Preferences→Advanced). Going forward, I'm going to assume that you're using the newest version of whatever browser you choose.

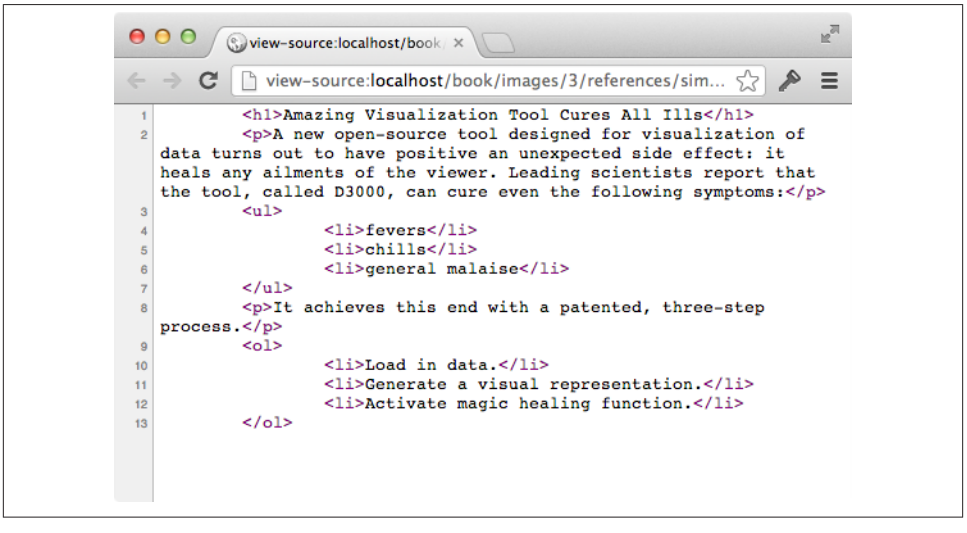


Figure 3-2. Looking at the source code in a new window

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

Fortunately, your browser's developer tools enable you to see the current state of the DOM. And, again, the developer tools are different in every browser. In Chrome, find them under View→Developer→Developer Tools. In Firefox, try Tools→Web Developer. In Safari, first enable the developer tools (in Safari→Preferences→Advanced). Then, in the Develop menu, choose Show Web Inspector. In any browser, you can also use the corresponding keyboard shortcut (as shown adjacent to the menu item) or right-click and choose "inspect element" or something similar.

Until recently, Safari and Chrome shared the same developer tools, but with Safari 6.0, Apple completely redesigned their dev tools, much to the dismay of many web-developing Safari fans. (The new tools are very hard to navigate, and I don't think I'm the only one who feels that way.) Whichever browser you use might look a bit different from my screenshots, but the functionality will be very similar.

Figure 3-3 shows the Elements tab of Chrome's web inspector. Here we can see the current state of the DOM. This is useful because your code will modify DOM elements dynamically. In the web inspector, you can watch elements as they change.

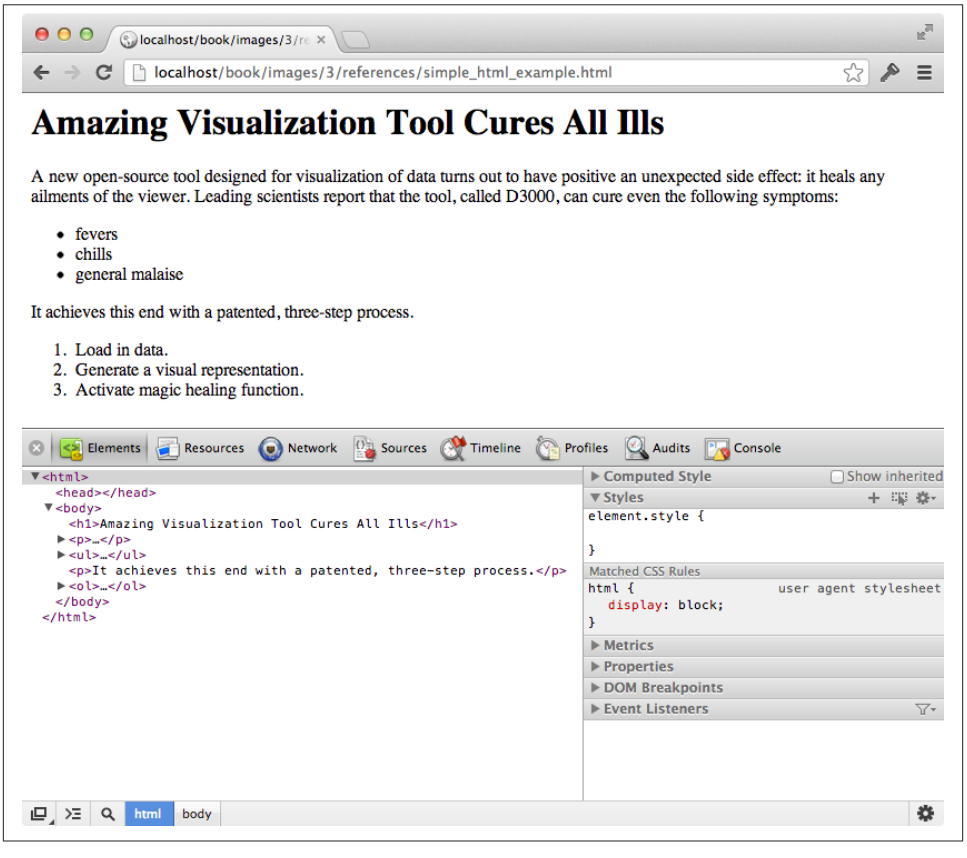


Figure 3-3. Chrome's web inspector

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

One more thing: why am I focusing on Chrome, Firefox, and Safari? Why not Internet Explorer, Opera, or the many other browsers out there? For one, it's best to develop your projects using a browser with the broadest support for web standards. Internet Explorer made huge progress with versions 9 and 10, but Chrome, Firefox, and Safari are understood to have the broadest standards support, and they are updated more frequently.

Second, you're going to spend a lot of time using the developer tools, so you should develop with a browser that has tools you enjoy using. I was pretty devoted to Safari until the 6.0 update changed everything. Now I'm going back and forth between Chrome and Firefox's new dev tools. I recommend you try them all and decide what works best for you.

Rendering and the Box Model

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

The most important thing to keep in mind when considering how browsers render content is this: to a browser, everything is a box.

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

You can see these boxes with the help of the web inspector. Just mouse over any element, and the box associated with that element is highlighted in blue, as shown in Figure 3-4.

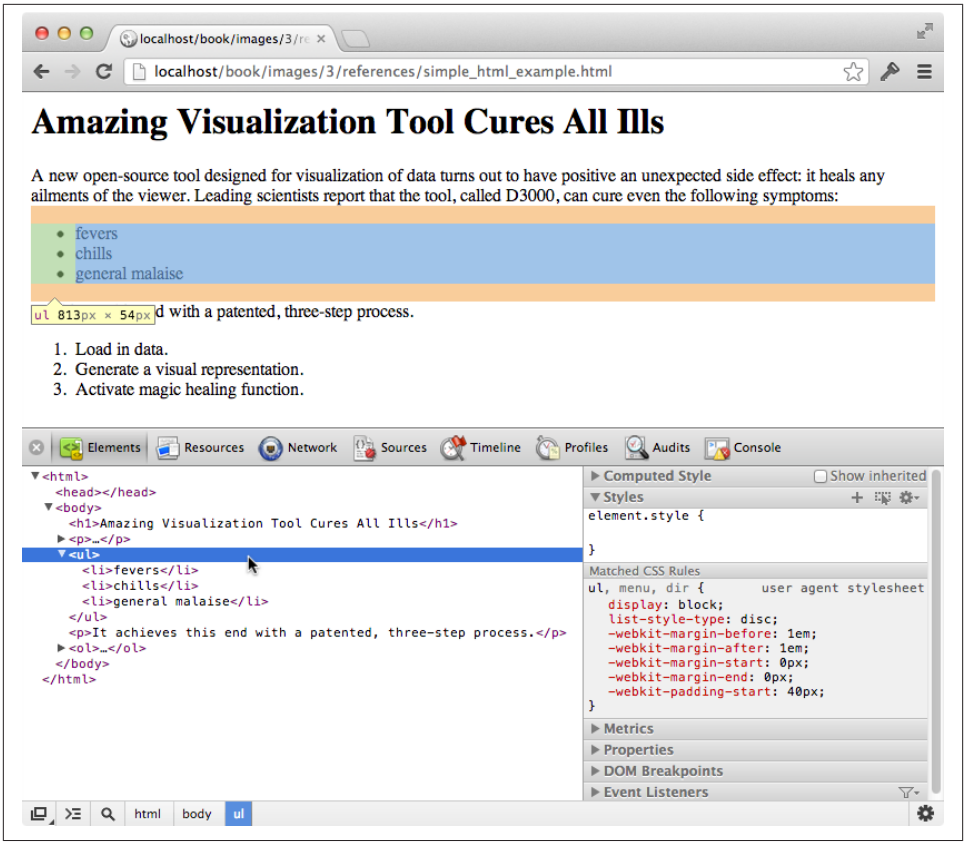


Figure 3-4. Inspector with element box highlighted

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

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

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

CSS

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

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

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

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

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

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

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

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

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

Selectors

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

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

Type selectors

These are the simplest. They match DOM elements with the same name:

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

Descendant selectors

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

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

Class selectors

These match elements of any type that have been assigned a specific class. Class names are preceded with a period, as shown here:

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

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

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

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

ID selectors

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

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

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

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

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

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

Properties and Values

Groups of property/value pairs cumulatively form the styles:

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

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

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

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

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

Comments

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

Referencing Styles

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

Embed the CSS in your HTML.

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

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

            p {
                font-size: 24px;
```

```
                font-weight: bold;
                background-color: red;
                color: white;
            }

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

That HTML page with CSS renders as shown in Figure 3-5.

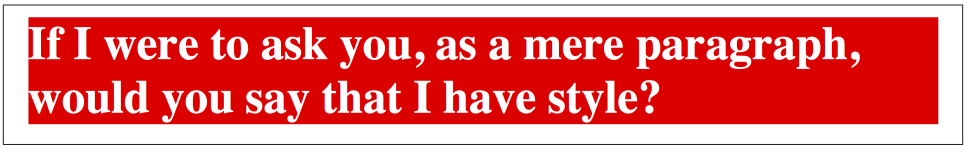


Figure 3-5. Rendering of an embedded CSS rule

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

Reference an external stylesheet from the HTML.

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

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

This example renders exactly the same as the prior example.

Attach inline styles.

A third method is to attach style rules *inline* directly to elements in the HTML. You can do this by adding a `style` attribute to any element. Then include the CSS rules within the double quotation marks. The result is shown in Figure 3-6.

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

Figure 3-6. Rendering of an inline CSS rule

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

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

Inheritance, Cascading, and Specificity

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

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

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

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

Yet when this page renders, the styles intended for the `div` (red background, bold text, and so on) are *inherited* by the second paragraph, as shown in Figure 3-7, because that `p` is a descendant of the styled `div`.

Figure 3-7. Inherited style

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

Finally, an answer to the most pressing question of the day: why are they called Cascading Style Sheets? It's because selector matches cascade from the top down. When more than one selector applies to an element, the later rule generally overrides the earlier one. For example, the following rules set the text of all paragraph elements in the document to be blue *except* for those with the class of `highlight` applied, which will be black *and* have a yellow background, as shown in Figure 3-8. The rules for `p` are applied first, but then the rules for `p.highlight` override the less specific `p` rules.

```
p {
    color: blue;
}

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

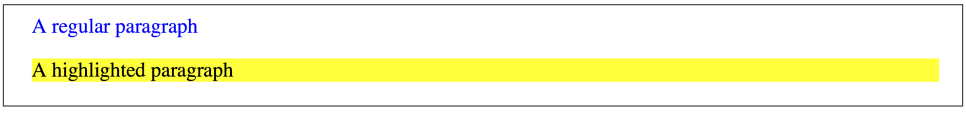


Figure 3-8. CSS cascading and inheritance at work

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

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