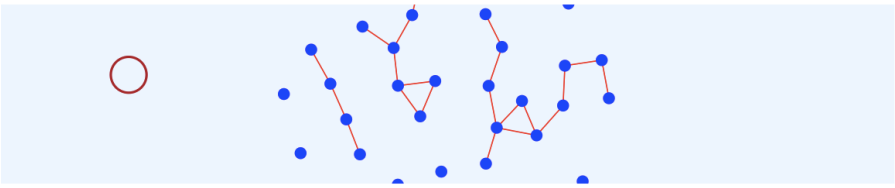
D3 for R Users

Joyce Robbins

2020-12-07

Contents

Welcome to D3



Adapted from Build Your Own Graph!

This guide serves as a companion text to Scott Murray's *Interactive Data Visualization for the Web, 2nd edition*–henceforth *IDVW2*–a required text for GR5702. Be sure to get the second edition, which is a comprehensive update to D3 version 4. The first edition uses D3 version 3, which is not compatible. (The current version of D3 is actually v6. However, since differences between v4 and v5/v6 are minimal, unless otherwise indicated in this guide, the code in *IDVW2* will work with either.)

We rely on the text heavily but also deviate from it in several ways. *IDVW2* is written for graphics designers not data science students so the pain points are somewhat different.

D3 is a JavaScript library, not a standalone language, so any time we refer to D3 we really mean D3/JavaScript, though it is not necessary to know JavaScript well before beginning; we will learn as we go. Most of the JavaScript we use is covered in *IDVW2*, though we also use some newer JavaScript options from ES5 and ES6, such as `.map()`, `.filter()`, arrow functions and template literals, that make coding easier (and more like R!)[1] We use different examples, though you are strongly encouraged to study Murray's code examples in addition to reading the text. Particularly through the first half, we don't follow the text in order, so always refer to this guide first which will direct you to the pages of the text that you should read.

[1] Ok, they're not actually that new, but it takes a while for new JavaScript to catch on, mainly due to concern with maintaining compatibility with older browsers. Since D3 itself is not compatible with very old browsers, and since we can't focus on everything at once, we are not going to concern ourselves with browser compatibility. If you are interested in this, caniuse.com is very helpful for looking up what works where.

This is very much a work-in-progress so please submit issues on GitHub to provide feedback and edit or add text by submitting pull requests. (Click the icon at the top of each page to get started. More detailed instructions are available on edav.info. If you would just like to view the source code, click the icon.)

0.1 Workflow

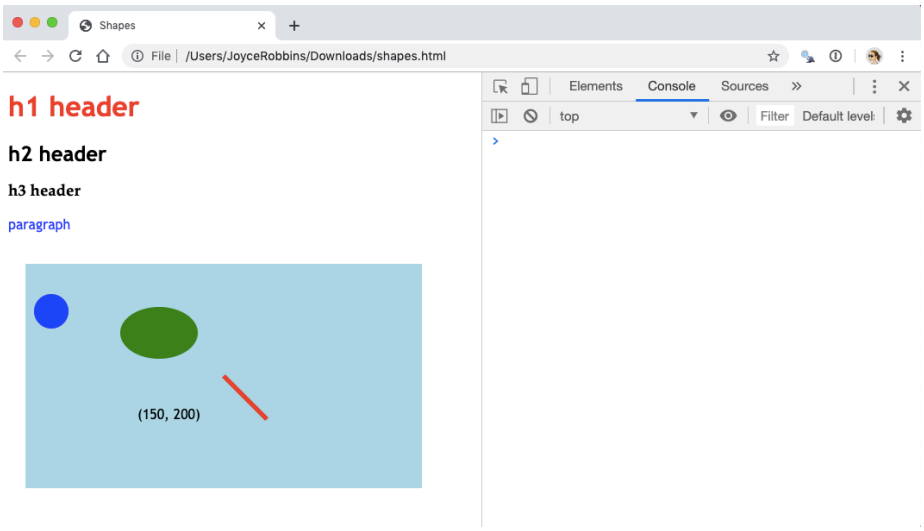
A big hurdle to learning a new language is just getting setup. Often authors forget to mention what your programming environment should look like, what should be open on the screen. I will try not to do that and be as clear as possible so you know where you should be entering the code in the pages that follow. This task is somewhat complicated by the fact that we will be using a variety of workflow options. This section will serve as a reference guide; future sections will link back here as appropriate.

All of our workflows require Google Chrome, so if you don't have it already, download and install it.

0.1.1 JavaScript Console

With this workflow we will open a web page–either online or local–in Chrome and run JavaScript in the Console. To view the Console, open Chrome DevTools by clicking *View, Developer, JavaScript Console* if you have a menu bar in Chrome, using a keyboard shortcut (Mac: option+command+j; Windows, Linux, Chrome OS: control+shift+J), or employing another one of the many options for doing so. The Console is one piece of a suite of tools available in the browser.

With the DevTools open, your screen will look like this:



The next chapter, Jump in the deep end, employs this workflow.

0.1.2 This book in the Console

If you're not reading the `.pdf` version, you can open DevTools on this very page. This is very convenient because not only to you not have to leave this book to practice D3, you can copy code blocks and paste them in the Console. In addition to opening DevTools (see above), close the side bar by clicking on the icon above to give yourself more screen space.

Let's try it out.

Open the JavaScript Console

Scroll so that both the blue rectangle above and the code chunk below are visible on your screen. Toggle the sidebar, open the Console, and then move the mouse onto the code block so the icon appears. Click on it to copy the code, paste it in the Console, and then press return.

```
d3.select("svg#demo")
  .append("circle")
    .attr("cx", "-25")
    .attr("cy", "100")
    .attr("r", "20")
    .attr("fill", "red")
  .transition()
  .duration(3000)
    .attr("cx", "325")
  .remove();
```

Pretty neat.

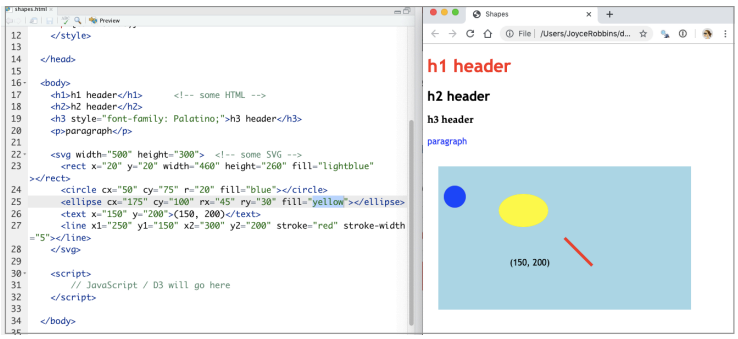
0.1.3 Text editor

This is a very basic local setup in which the same `.html` is open both in a text editor (if you don't want to stray too far from home, use RStudio) and in a web browser (we will use Chrome), each on one half of your screen. The workflow is: make changes to the file in the text editor, save the changes and then refresh the page in the browser to see the updates. Keyboard shortcuts for saving and refreshing (on the Mac, command-s and command-r respectively) are very helpful.

Let's try an example:

Download a copy of `shapes.html` by opening this page and clicking *File, Save Page As…* Open the file in a text editor of your choice on one half of your screen. On the other half of your screen open the same file in Chrome. As you make changes to the `.html` file, save the file and then refresh the browser to see the effects.

Your screen should look like this:



Chapter 1

Jump in the deep end

Let's skip the explanations and start coding in D3 right now. Why? So you can see the benefits and know what you're working toward when you get stuck in the weeds. Then we'll go back and start learning step by step. In this chapter we will work in the JavaScript Console (help).

1.1 Get ready

1. If you don't have it, install the Chrome browser.

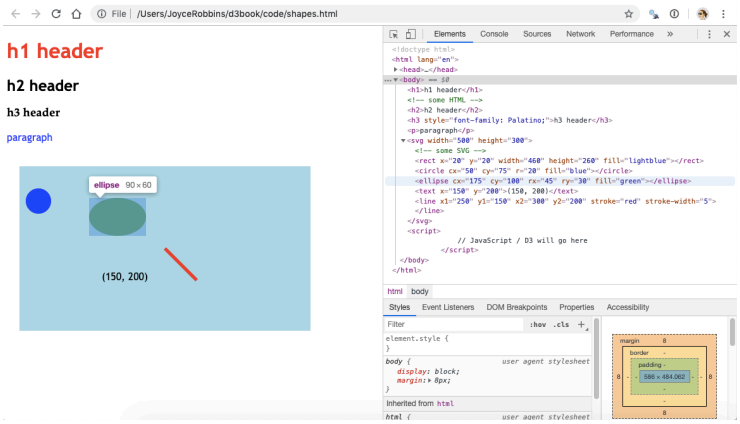
2. Download a copy of `shapes.html` by opening the following page and then saving with *File, Save Page As…*: `shapes.html`. (Or download a zip of the whole repo. Clicking here will start the download. Or fork and clone the repo).

3. If Chrome is your default browser, open `shapes.html` by double clicking it. Otherwise, open it with *File, Open File…* in Chrome.

1.2 Elements tab

1. Open Chrome DevTools (help).

2. Hover the mouse over various elements in the `<body> ... </body>` section. Observe the highlighted sections in the rendered web page on the left of the screen. Click on the mini black triangles to the left of the `<body>` and `<svg>` tags if needed to open these sections of the DOM tree. Your screen should look like this:



3. Now try the reverse: right click on elements on the web page, choose "Inspect" and see what is highlighted in the Elements pane. Get comfortable with the connection between the code on the right and the rendered elements on the left.

1.3 Console tab

1. Switch to the Console tab, next to the Elements tab. Let's practice running some code. Note that the code is unrelated to the `shapes.html` web page that we have open.

   We will spend a lot of time in the Console since it's interactive – think R console. Eventually we will switch to including JavaScript/D3 in `.html` or `.js` files and use the Console only for testing things out or debugging.

2. Type the following lines of code at the prompt (`>`), press enter after each line–that is, after the semicolon (`;`)–and see what happens:

```
3 + 4;

"3" + "4";

 x = [1, 2, 3];

x[1];

x + 1;
```

```
y = {a: 3, b: 4};

y["b"];
```

1.4 Modify elements

1. Now we'll start using D3 to manipulate elements on the page. Try the following, by entering one line at a time in the Console as before:

```
d3.select("circle").attr("cx", "200");

d3.select("circle").attr("cx", "500");

d3.select("circle").attr("cx", "100");

d3.select("circle").attr("r", "30");

d3.select("circle").attr("r", "130");

d3.select("circle").attr("r", "3");

d3.select("circle").attr("fill", "red");

d3.select("circle").attr("fill", "aliceblue");

d3.select("circle").attr("fill", "lightseagreen");
```

   Note that "select" and "attr" are separate operations chained together with "." – think pipe (%>%) operator.

2. Refresh the page. What happened?
3. Go to Elements. Look at the value of the `y1` attribute of the SVG `<line>` element. Go back to the Console and enter the following:

```
d3.select("line").attr("y1", "10");
```

4. Switch back to Elements and observe. What happened?
5. Stay in Elements and refresh the page. What happened to `y1`?
6. Return to the Console to make style changes to the HTML elements:

```
d3.select("h1").style("color", "purple");

d3.select("h2").style("font-size", "50px");

d3.select("h2").style("font-family", "Impact");
```

1.5 Add transitions

1. Try these:

```
d3.select("circle").transition().duration(2000).attr("cx", "400");

d3.select("ellipse").transition().duration(2000).attr("transform", "translate (400, 400)");

d3.select("line").transition().duration(2000).attr("x1", "400");

d3.select("line").transition().duration(2000).attr("y1", "250");

d3.select("p").transition().duration(2000).style("font-size", "72px");
```

2. Experiment with more transitions.

1.6 Add interactivity

1. Set up a function to turn the fill color to yellow:

```
function goyellow() {d3.select(this).attr("fill", "yellow")};
```

2. Add an event listener to the circle that will be trigger a call to `goyellow()` on a mouseover:

```
d3.select("circle").on("mouseover", goyellow);
```

3. Test it out.
4. Add the same event listener to the ellipse. Test it out.
5. Create a function `goblue()` that changes the fill color to blue.
6. Add event listeners to the circle and ellipse that will trigger a call to `goblue()` on a *mouseout*. Test out your code.

7. Try out a click event. (Note the use of an anonymous function.)

```
d3.select("line").on("click", function()
  {d3.select(this).attr("stroke-width", "10");});
```

8. Try another click event. What's happening?

   v6

```
d3.select("svg").on("click", function(event)
  {d3.select("text").text(`(${d3.pointer(event)})`)});
```

   v5

```
d3.select("svg").on("click", function()
  {d3.select("text").text(`(${d3.mouse(this)})`)});
```

Chapter 2

Web tech

Read *IDVW2*, Chapter 3: Technology Fundamentals

There is a lot of material in this chapter. It is worth making the effort to learn it now and start D3 with a solid foundation of elementary HTML/CSS/SVG/JavaScript.

Here we examine `shapes.html` from Chapter 1 to see how the various technologies are combined into a single document.

2.1 HTML

Note that `shapes.html` has an HyperText Markup Language or `.html` extension; HTML in fact provides the structure for the document. It has a `<head>` and `<body>` section.

In the `<head>` section we use `<script>` tags to link to the D3 library:

```
<script src="https://d3js.org/d3.v6.js"></script>
```

HTML content is enclosed between opening an closing **tags** such as `<h1>` and `</h1>`.

HTML class and ID **attributes** are included inside the opening tags:

```
<h1 class="myclass" id="myid">This is an h1 header.</h1>
```

2.2 CSS

CSS (Cascading Style Sheets) is used for styling web pages, and more importantly for our purposes, selecting elements on a page or in a graphic. We will

generally work with internal style sheets since it's simpler when starting out to have everything in one document. External style sheets, however, are generally the preferred method for web design.

2.2.1 Internal style sheet

`shapes.html` has an *internal style sheet*: CSS style information appears in the `<head>` section marked off with `<style>` tags:

```
<style type="text/css">
    h1 {color:red;}     /* CSS styling */
    p {color:blue;}
</style>
```

Here we specify that all HTML `<h1>` headers should be red and all HTML paragraphs `<p>` should be blue. This is an example of an *internal style sheet*. Later we will consider alternatives: *external style sheets* and *inline styling*.

Styling for coder designed classes is also specified in this section. For example, we could style a "formal" class as such:

```
<style type="text/css">
    .formal {color: red;
        font-size: 30px;
        font-family: Lucida Calligraphy;
        }
</style>
```

Note that classes are defined by the "." before the name.

2.2.2 External style sheets

External style sheets are `.css` files that contain styling information and are linked to with a `<link>` tag in the `<head>` section of an HTML document:

```
<head>
    <link rel="stylesheet" href="style.css">
</head>
```

External style sheets are the preferred way of styling as they can easily be modified without changing the web page; in fact, the motivation for CSS came from a desire in the early days of the internet to separate styling from content.

Developers have the option now of choosing premade themes, which are shared through external style sheets. They can be quite complex. The `.css` file for the Minty theme from Bootswatch, for example, contains over 10,000 lines.

CSS Zen Garden demonstrates the power of external style sheets: the same HTML document takes on very different looks depending on the stylesheet to which it is linked.

2.2.3 Inline styling

With inline styling, styling is added to each tag individually:

```
<span style="color: white; background-color: fuchsia; font-family: impact;
      font-size: 24px; border-style: solid; border-color: limegreen;
      border-width: 3px">
      Styled inline
</span>
```

Styled inline

This is how early web pages were styled. To take a step back in time, use developer tools to view the source code for the main page of www.dolekemp96.org, an old web site that has been maintained for historical purposes. As you can see, it's a tedious way of writing content, which internal and external style sheets eliminate.

Although you will not be adding inline styling manually, you will notice that when we select elements and change the styling with D3, the modifications are made inline. In other words, we do not make changes to the elements directly, not via a style sheet.

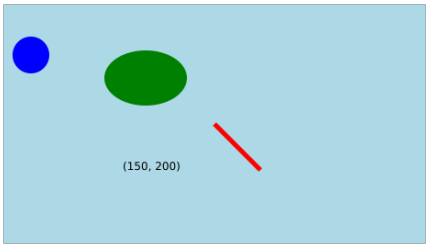
2.3 SVG

SVG (Scalable Vector Graphics) is a human readable graphics format that facilitates manipulation of individual elements. You may be familiar with `.svg` files. Here we have SVG graphics within `<svg>` tags in the `<body>` section of the HTML document:

```
<svg width="500" height="300">  <!-- some SVG -->
    <rect x="20" y="20" width="460" height="260" fill="lightblue"></rect>
    <circle cx="50" cy="75" r="20" fill="blue"></circle>
    <ellipse cx="175" cy="100" rx="45" ry="30" fill="green"></ellipse>
    <text x="150" y="200">(150, 200)</text>
    <line x1="250" y1="150" x2="300" y2="200" stroke="red" stroke-width="5"></line>
</svg>
```

Rendered:

‘



(150, 200)

There are very few SVG tags that you'll need to know, and once we get going with D3, you will not have to code any SVG manually. It is worth doing a little to become familiar with the format and in particular to get used to the new location of the origin.

2.4 JavaScript

JavaScript is the most common language for making web pages interactive. Code is executed when pages are opened or refreshed. So far we have run JavaScript in the Console, but have not included it in the web page itself. When we do so, it will be between `<script>` tags in the `<body>` section of the HTML document, or in a separate `.js` file.

We will learn JavaScript on an as-needed basis. In terms of data, we will begin with simple arrays:

```
var x = [3, 5, 1, 6, 7]
```

In the Just Enough JS chapter, we cover more complex data structures and some methods for data manipulation.

javascript.info is an excellent resource for expanding your knowledge beyond the basics.

2.5 D3

D3 (Data Driven Documents) is a JavaScript library well suited to interactive graphics. As such, it is also included between `<script>` tags in the `<body>` section. For D3 to work, you must link to the D3 library in the `<head>` section of the document.

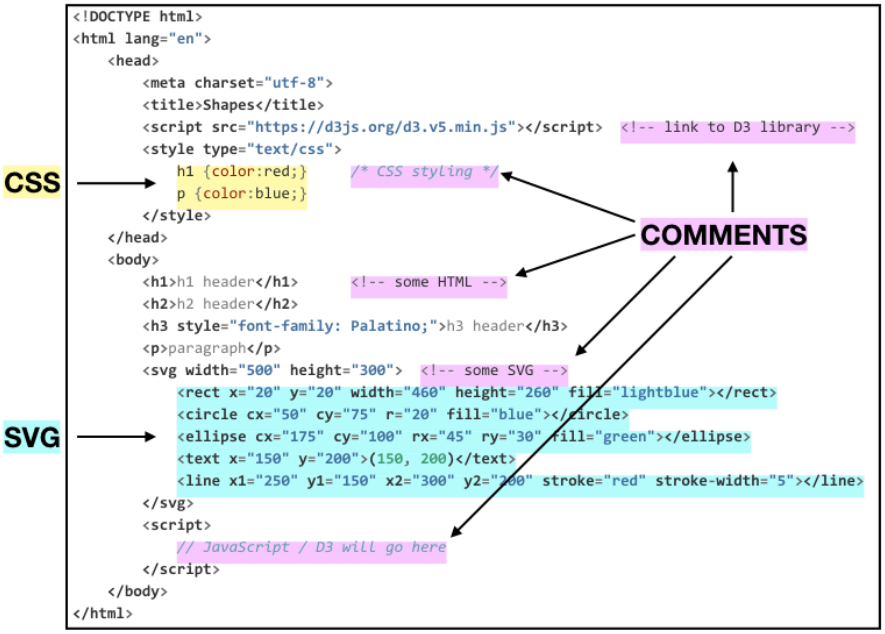
There seems to be a misconception that D3 is a high level language. It is not. You will be working on the pixel level to create graphics, including drawing your own axes and doing other things that you're not used to doing if you've been working in R or Python.

> *On the bright side, after D3, you will gain a new appreciation for base R graphics. You will write code such as* `plot(iris$Sepal.Length, iris$Sepal.Width, pch = 16, col = iris$Species, las = 1, xlab = "Sepal.Length", ylab = "Sepal.Width")` *and think: wow, there are axes! Amazing!*

It is legitimate to ask why you need to know D3 as a data scientist. Many if not most of you will not be coding in JavaScript from the ground up in your future careers. However, it's a great way to learn how interactive graphics work under the hood, and will give you a solid foundation which you can draw on to tweak visualizations that you build with high level tools such as Plotly.

2.6 HTML tree

While `shapes.html` appears as a single consistent document, it is actually comprised of multiple languages. HTML, CSS, and SVG are already there, and we will be adding JavaScript / D3 soon.



Of note:

- An HTML document is composed of lines or sections set off with tags. In particular `<style> ... </style>`, `<svg> ... </svg>`, and `<script> ... </script>` indicate the inclusion of CSS, SVG, and JavaScript/D3 respectively.
- For D3 to work, you must link to a D3 library. To link to the online version, copy and paste the `<script>` line from https://d3js.org. Alternatively, you can also download a copy from the same site and reference your local copy with:

```
<script src="d3.js"></script>
```

- There are two main sections. The `<head>` section contains the *title*, *link to D3 library*, and *internal CSS*. The `<body>` section contains HTML elements (`<h1>`, `<p>`, etc.), SVGs (between `<svg>`/`</svg>`tags) and JavaScript/D3 scripts (between `<script>`/`<script>`tags).

  Do not assume that if it works that it is correct; today's browsers can be very forgiving.

- Comment syntax varies with language:

- `<!-- single or multiline HTML or SVG comment -->`
- `/* single or multiline CSS comment */`
- `// single line JavaScript comment`
- `/* JavaScript multiline comment */`

2.7 Exercise : shapes

Download a copy of `shapes.html` by opening this page and clicking *File, Save Page As…* Set yourself up to work locally in a text editor help. (Developer Tools should not be open; we will not be using the Console.)

1. Add an additional circle to the svg.
2. Add styling to the internal style sheet to style circles.
3. Add two additional paragraphs use the `<p>` tag.
4. Add an ID attribute to one of the circles.
5. Add a class attribute to two of the `<p>` tags.
6. Use the internal style sheet to style paragraphs of the class you created in 5.
7. Adjust additional elements as desired.

Chapter 3

D3 in the Console

Read *IDVW2*, Chapter 6: Drawing with Data. Skip pp. 89-96 as we will not be drawing bar charts with the `div`approach.

3.1 Selections

3.1.1 Select by tag

The ability to select elements on a page is key to being able to manipulate them. `d3.select()` will select the first match; `d3.selectAll()` will select all matches.

```
d3.select("svg").select("circle");
```

selects the first circle in the order in which circles appear in the `<svg>` grouping. If there were more than one circle we could select them all with:

```
d3.select("svg").selectAll("circle");
```

We can select HTML elements by tag in the same way:

```
d3.select("body").select("h1");
d3.select("body").selectAll("h1");
```

3.1.2 Select by class

Classes are selected by adding a "." before the class name:

```
d3.select("svg").selectAll("circle.apple")
```

This provides one method of selecting a certain collection of elements of the same type.

3.1.3 Select by ID

IDs differ from classes in that they are unique identifies. IDs are selected by adding a "#" before the ID:

```
d3.select("svg").select("circle#henry");
```

3.1.4 Store selections

It is often helpful to store selections for later use. Here we store the svg selection in `mysvg`:

```
var mysvg = d3.select("svg");
```

> *The JavaScript community is moving toward using* `let` *and* `const` *instead of* `var`*; we, however, will stick with* `var` *to be consistent with* IDVW2. *Of course you're welcome to use* `const` *and* `let` *instead, and if so, may find these articles helpful: Let It Be - How to declare JavaScript variables and ES2015 const is not about immutability.*

Store circle selection in a variable:

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

var circ = svg.selectAll("circle");
```

3.2 Modify existing elements

Try out the code in this section with a downloaded copy of five_green_circles.html opened in Chrome and the Console visible.

3.2.1 Modify attributes

link to get or set attribute API

```
d3.select("circle").attr("r");          // see radius

d3.select("circle").attr("r", "10");    // set radius to 10
```

3.2.2 Modify styles

link to get or set style API

```
d3.select("h1").style("color");

d3.select("h1").style("color", "blue");
```

> *It is often difficult to remember whether to use* `.attr()` *or* `.style()` *In general, properties such as position on the SVG, class, and ID are* attributes*, while decorative properties such as color, font, font size, etc. are* styles*. However, in some cases, you can use either. For example, the following both make the circle blue:*

```
d3.select("circle").attr("fill", "blue");

d3.select("circle").style("fill", "blue");
```

The first will add a `fill="blue"` *attribute to the* `<circle>` *tag, while the latter will add* `style="fill: blue;"`*. All is well and good until you find yourself with both in the same tag, in which case the* `style` *property will take precedence. The bottom line: don't mix the two options because it can cause problems.*

> *To further complicate matters,* `.style()` *is just shorthand for* `.attr("style", "...")` *so the following are in fact equivalent:*

```
d3.select("circle").style("fill", "blue");

d3.select("circle").attr("style", "fill: blue;");
```

In other words, style is an attribute!

3.2.3 Modify text

This section is interactive. Hover over code as directed to observe effects.

HTML text

```
<p id="typo" class="fancy">Manhatten</p>
```

Manhatten

Hover to execute this code (and fix the typo):

```
d3.select("#typo").text("Manhattan");
```

SVG text

```
<svg width="500" height="100">
  <rect width="500" height="100" fill="#326EA4"></rect>
  <text id="svgtypo" x="50" y="70" fill="white" font-weight="bold" font-size="40px">
     Web scrapping is fun.</text>
</svg>
```

Hover on this SVG to execute the code below it (and fix the typo):

Web scrapping is fun.

```
d3.select("#svgtypo").text("Web scraping is fun.");
```

> *The SVG `<text>` tag can be tricky. It differs from HTML text tags (`<p>`, `<h1>`, `<h2>`, etc.) in that it has `x` and `y` attributes that allow you to position text on an SVG canvas. Unlike HTML, the fill attribute controls the color of the text. Compare:*

```
d3.select("p").style("color", "red");   // HTML

d3.select("text").attr("fill", "red");  // SVG
```

3.2.4 Move SVG text

```
<svg width="600" height="100">
  <rect width="600" height="100" fill="#326EA4"></rect>
  <text id="moveleft" x="200" y="70" fill="white" font-weight="bold" font-size="40px">
      I want to move left.</text>
</svg>
```

Hover on this SVG to execute the code below it:

I want to move left.

```
d3.select("#moveleft").attr("x", "20").text("Thanks, now I'm happy!");
```

3.3 Add elements

3.3.1 HTML

Continue trying out code with `five_green_circles.html` open in Chrome.

The following adds a `<p>` tag but doesn't change how the page looks, since there's no text associated with it.

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

To add text, use `.text()`:

```
d3.select("body").append("p").text("This is a complete sentence.");
```

> *To debug adding an element, go to the Elements tab to see what was added and where. If an element is in the wrong place in the HTML tree, it will not be visible.*

3.3.2 SVG

Likewise, here we add a `<circle>` to the `<svg>`, but we can't see it since it has no attributes.

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

Adding attributes will create visible circles:

```
d3.select("svg").append("rect").attr("x", "0").attr("y", "0")
    .attr("width", "500").attr("height", "400").attr("fill", "lightblue");

d3.select("svg").append("circle").attr("cx", "200")
    .attr("cy", "100").attr("r", "25").attr("fill", "orange");

d3.select("svg").append("circle").attr("cx", "300")
    .attr("cy", "150").attr("r", "25").attr("fill", "red");
```

We can use a saved selection to assist in creating a new element:

(*IDVW2*, pp. 97-98)

```
mysvg = d3.select("svg");

mysvg.append("circle").attr("cx", "250").attr("cy", "250").attr("r", "50")
  .attr("fill", "red");
```

3.4 Remove elements

These methods will remove matching elements in order, starting with the first find in the document.

3.4.1 HTML

```
d3.select("p").remove();
```

3.4.2 SVG

```
d3.select("svg").select("circle").remove();

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

3.5 Exercise : green circles

Download and open a fresh copy of five_green_circles.html in Chrome. Open Developer Tools open and do the following in the Console with D3:

1. Select the circle with ID “henry” and make it blue.
2. Select all circles of “apple” class make them red.
3. Select the first circle and add an orange border (“stroke”), and stroke width (“stroke-width”) of 5.
4. Select all circles of “apple” class and move them to the middle of the svg.

3.6 Exercise : blue circles

Download and open a fresh copy of six_blue_circles.html in Chrome. Open Developer Tools and execute Steps 1-4 one at a time in the Console. After Step 4, refresh the page to go back to Step 1 if so desired. (You do not need to create a loop as in the visual.)

> *This exercise is provided as a challenge. It's fine to skip this exercise and move on to the next section.*

1. Move all the circles to the right.
2. Move them back to the left *and* change their color.
3. In a text editor, add an id to the third circle in `six_blue_circles.html`, save the file, and then in the Console, move only that circle to the right.
4. Move all the circles to the middle of the screen, *then* move them all to the same location.

3.7 Bind data... *finally!*

(*IDVW2*, pp. 98-108)

To follow along with the code in this section, download and open six_blue_circles.html.

Bind data:

```
d3.select("svg").selectAll("circle").data([90, 230, 140, 75, 180, 25]);
```

Check data binding:

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

Set x-coordinate of each circle to data value using arrow function:

```
d3.select("svg").selectAll("circle").attr("cx", d => d);
```

Set x-coordinate of each circle to data value with a JavaScript function:

```
d3.select("svg").selectAll("circle").attr("cx", function(d) {return d;});
```

We'll bind a new set of data to the circles, this time storing the dataset in a variable:

```
var dataset = [50, 80, 110, 140, 170, 200];
```

We'll also store a selection of all circles before binding the data:

```
var circ = d3.select("svg").selectAll("circle");
```

And now, the data bind:

```
circ.data(dataset);
```

Nothing appears to have happened; the circles remain the same and there is no evidence of any changes looking at the circles in the DOM (see Elements tab).

We can check that the data are indeed bound with:

```
circ.data();  // now we see data
```

Modify elements w/ stored selections, bound data:

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

circ.attr("cx", function(d) {return d/2;});

circ.attr("cx", function(d) {return d/4;}).attr("r", "10");
```

Same as above, using arrow functions:

```
circ.attr("cx", d => d);

circ.attr("cx", d => d/2);

circ.attr("cx", d => d/4).attr("r", "10");
```

Note that if we bind a new set of data to the DOM elements, the original set will be overwritten:

```
var newdata = [145, 29, 53, 196, 200, 12];

circ.data(newdata);

circ.transition()
    .duration(2000)
    .attr("cx", d => 2*d);
```

3.8 Exercise : data bind

Download and open a fresh copy of six_blue_circles.html in Chrome and practice binding data to the circles and modifying the circles based on the data as in the examples above.

Chapter 4

Update, Enter, and Exit

Read: *IDVW2*, Chapter 9, pp. 178-184; Chapter 12, pp. 231-249

4.1 Lecture slides

D3 Data Bind

4.2 Remove some elements

a.k.a. more DOM elements than data values

We'll start with six circles and remove some.

Download and open a fresh copy of six_blue_circles.html in Chrome.

Let's bind four data values to the six circles:

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

svg.selectAll("circle")
    .data([123, 52, 232, 90]);
```

Click the black triangle to view the `_enter`, `_exit`, and `_groups` fields.

We can store the selection in a variable:

```
var circ = svg.selectAll("circle")
    .data([123, 52, 232, 90]);
```

Let's look at the exit selection:

```
circ.exit();
```

Try this:

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

What happened and why?

Now try this:

```
circ.exit().attr("fill", "purple");
```

What happened and why?

What do you think this will do? Try it.

```
circ.exit().transition().duration(2000).remove();
```

Create a new variable `circ2` and compare it to `circ`:

```
var circ2 = d3.selectAll("circle");

circ.data();

circ2.data();

circ.exit();

circ2.exit();
```

What's going on?

4.3 Add some elements

a.k.a. more data values than DOM elements

We'll start with six circles and add some.

Let's bind new data to the circles:

```
var circ = svg.selectAll("circle")
      .data([123, 52, 232, 90, 34, 12, 189, 110]);
```

And look at the enter selection:

```
circ.enter();
```

How many placeholders are in the enter selection?

Let's add circles for each of these placeholders:

```
circ.enter()
    .append("circle")
      .attr("cx", "100")
      .attr("cy", (d, i) => i * 50 + 25)
      .attr("r", "20")
      .attr("fill", "blue");
```

Try this:

```
circ.transition()
  .duration(3000)
  .attr("cx", "400");
```

What do you need to do to act on *all* of the circles?

```
svg.selectAll("circle")
  .transition()
  .duration(2000)
  .attr("cy", (d, i) => (i * 50) + 25)
  .attr("cx", "200");
```

4.4 Data / enter / append

We'll start with nothing–not even an SVG–and add elements with the data / enter / append sequence.

Work in the Console on this page (help).

> *Open the JavaScript Console*

The SVG will be added here:

```
var svg = d3.select("div#dea")
  .append("svg")
  .attr("width", "400")
  .attr("height", "250");
```

Create an array of values:

```
var specialdata = [75, 150, 200];
```

Add rectangles:

```
  svg.selectAll("rect")
    .data(specialdata)
    .enter()
    .append("rect")
      .attr("x", d => d)
      .attr("y", d => d)
      .attr("width", "50")
      .attr("height", "30")
      .attr("fill", "pink");
```

4.4.1 Labels

Note that we can also label the rectangles with the data value:

```
  svg.selectAll("text")
      .data(specialdata)
      .enter()
      .append("text")
      .attr("x", d => d + 25)
      .attr("y", d => d + 25)
      .text(d => d)
      .attr("fill", "blue")
      .attr("text-anchor", "middle");
```

4.5 Exercise : horizontal bar chart

1. Create a new html file (try to recreate the template without looking… or save a copy of this one) and open it in your text editor.

   If you create a new file in RStudio, choose "Text File" and use the `.html` *file extension when you save it. Do not choose "R HTML".*

Add a script that adds an svg element and horizontal bars of the lengths (in pixels) specified in `bardata`. Create the bars with the data / enter / append sequence.

```
    var bardata = [300, 100, 150, 225, 75, 275];
```

4.6 Merge selections

a.k.a. combining update and enter selections with `.merge()`

Open six_blue_circles.html in Chrome. (You do not need to download it first.)

Run the following code in the Console:

```
var svg = d3.select("svg");
var circ = svg.selectAll("circle")
  .data([123, 52, 232, 90, 34, 12, 189, 110]);

var allcirc = circ.enter()  // 2 placeholders
        .append("circle")  // placeholders -> circles
          .attr("cx", "100")  // acts on enter selection only
          .attr("cy", (d, i) => (i - 5) * 50)
          .attr("r", "20")
          .attr("fill", "red");
```

Now try to predict what the following code will do. Were you right?

```
allcirc.transition()
        .duration(3000)
        .attr("cx", "400")
        .attr("fill", "purple");
```

Refresh the page and then copy and paste the following into the Console and run.

```
var svg = d3.select("svg");
var circ = svg.selectAll("circle")
  .data([123, 52, 232, 90, 34, 12, 189, 110]); // update selection

var allcirc = circ.enter()  // 2 placeholders
        .append("circle")  // placeholders -> circles
          .attr("cx", "100")  // acts on enter selection only
          .attr("cy", (d, i) => (i - 5) * 50)
          .attr("r", "20")
```

```
            .attr("fill", "red")
            .merge(circ);  // combines enter and update selections
```

And now, the following code (same as before). What changed? Why?

```
allcirc.transition()
        .duration(3000)
        .attr("cx", "400")
        .attr("fill", "purple");
```

Note the pattern:

Store the data bind in X.

Y = X.enter().append() *attributes* .merge(X)

Do more stuff with Y.

4.7 Exercise : merge

Open the bar chart you created in the previous exercise in Chrome, or this one and work in the Console. (You don't have to download it.)

1. Change the data to any six other values and update the lengths of the bars.

2. Bind a new dataset, `newbardata` to the bars, update the bar lengths, and remove any extra bars.

   ```
   newbardata = [250, 125, 80, 100];
   ```

3. Bind a new dataset, `reallynewbardata`, to the bars, then add additional bars so each data value has a bar. Make the outline (stroke) of the new bars a different color.

   ```
   reallynewbardata = [300, 100, 250, 50, 200, 150, 325, 275];
   ```

4. Use `.merge()` to combine the update and enter selections into one selection and then transition the height of all of the bars to half their current height.

5. Add text labels inside the bars at the right end with the length of the bar in pixels.

4.8 Groups

Open six_blue_circles.html in Chrome. (You do not need to download it first.)

Run this code in the Console:

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

var specialdata = [100, 250, 300];

var bars = svg.selectAll("rect")
      .data(specialdata)
      .enter()
      .append("rect")
        .attr("x", d => d)
        .attr("y", d => d)
        .attr("width", "50")
        .attr("height", "30")
        .attr("fill", "red");
```

What's going on?

Refresh the page, and try the following instead:

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

var specialdata = [100, 250, 300];

var bars = svg.append("g")
      .attr("id", "rects")
      .selectAll("rect")
      .data(specialdata)
      .enter()
      .append("rect")
        .attr("x", d => d)
        .attr("y", d => d)
        .attr("width", "50")
        .attr("height", "30")
        .attr("fill", "red");
```

Compare:

```
d3.select("svg")
  .select("g#rects")
  .selectAll("rect")
  .attr("fill", "purple");
```

and

```
d3.select("svg")
  .selectAll("rect")
  .attr("fill", "purple");
```

4.9 General Update Pattern

Open Developer Tools on this page.

Create a function in the Console:

```
function changedata(data) {
  d3.select("svg#gup")
    .selectAll("rect")
    .data(data)
    .attr("width", d => d);
    }
```

Test it out:

```
changedata([258, 373, 278, 9, 72, 96]);
```

What happens if there are too many data values?

```
changedata([196, 360, 283, 390, 46, 56, 152]);
```

Let's use the enter selection to add new bars in this case:

```
function changedata(data) {
  var bars = d3.select("svg#gup")
    .selectAll("rect")
    .data(data);    // bars is the update selection

  bars.enter()
    .append("rect")
      .attr("x", "30")  // until merge, acts on
      .attr("y", (d, i) => i * 50) // enter selection only
      .attr("height", "35")
      .attr("fill", "lightgreen")
    .merge(bars) // merge in the update selection
      .attr("width", d => d); // acts on all bars
  }
```

What happens if we have more bars than data values?

```
changedata([325, 116, 25]);
```

Let's add to the function to remove the extra bars in this case:

```
function changedata(data) {
  var bars = d3.select("svg#gup")
    .selectAll("rect")
    .data(data);    // bars is the update selection

  bars.enter()
    .append("rect")
      .attr("x", "30")  // until merge, acts on
      .attr("y", (d, i) => i * 50) // enter selection only
      .attr("height", "35")
      .attr("fill", "lightgreen")
    .merge(bars) // merge in the update selection
      .attr("width", d => d); // acts on all bars

  bars.exit()
    .remove();
  }
```

Try:

```
changedata([271, 49, 389]);
```

A fancy exit:

```
function changedata(data) {
  var bars = d3.select("svg#gup")
    .selectAll("rect")
    .data(data);    // bars is the update selection

  bars.enter()
    .append("rect")
      .attr("x", "30")  // until merge, acts on
      .attr("y", (d, i) => i * 50) // enter selection only
      .attr("height", "35")
      .attr("fill", "lightgreen")
    .merge(bars) // merge in the update selection
      .attr("width", d => d); // acts on all bars
```

```
  bars.exit()
    .attr("fill", "red")
    .transition()
    .duration(2000)
    .attr("width", "0")
    .remove();
  }
```

```
changedata([234, 129, 432, 286, 49, 372]);

changedata([401, 23, 173]);
```

VOILA! We have created the D3 General Update Pattern!

It is covered in *IDVW* in the "Other Kinds of Data Updates" section on pp. 178-186 in Chapter 9. (The earlier part of Chapter 9 deals with data updates in which the number of DOM elements remains the same.)

Note that the General Update Pattern changed with D3 Version 4 so avoid examples from Version 3.

Also available here: general_update_pattern.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>EDAV5_1</title>
    <script src="https://d3js.org/d3.v6.min.js"></script>
  </head>

  <body>

    <script id="s1">

// Create svg and initial bars

var svg = d3.select("body")
  .append("svg")
    .attr("width", "500")
    .attr("height", "400");

var bardata = [300, 100, 150, 225, 75, 275];

var bars = svg.selectAll("rect")
  .data(bardata);
```

```
bars.enter().append("rect")
  .attr("x", "30")
  .attr("y", (d, i) => i*50)
  .attr("width", d => d)
  .attr("height", "35")
  .attr("fill", "lightgreen");

// General Update Pattern

function update(data) {

  var bars = svg.selectAll("rect")    // data join
    .data(data);

    bars.enter()
      .append("rect")    // add new elements
        .attr("x", "30")
        .attr("y", (d, i) => i*50)
        .attr("width", d => d)
        .attr("height", "35")
        .attr("fill", "yellow")
      .merge(bars)    // merge
        .transition()
        .duration(2000)
        .attr("width", d => d)
        .attr("fill", "orange");

    bars.exit().remove();    // remove extra elements
    }

    </script>

  </body>

</html>
```

4.10 Exercise: : functions

Open general_update_pattern.html and practice running the `update()` function with different datasets in the Console.

For example:

```
update([100, 200, 300]);
```

4.11 Exercise : vertical bar chart

Change the bar chart in general_update_pattern.html to a vertical bar chart.

Chapter 5

Just Enough JS

Basics: *IDVW*, pp. 36-52

objects, arrays, arrays of objects, functions (and other things)

5.1 Arrays of arrays

Open the JavaScript Console

```
// try me in the Console
var array_dataset = [[100, 75, 30], [200, 125, 20]];

d3.select("svg#arrays")
  .selectAll("circle")
  .data(array_dataset)
  .enter()
  .append("circle")
    .attr("cx", d => d[0])
    .attr("cy", d => d[1])
    .attr("r", d => d[2])
    .attr("fill", "red");
```

svg#arrays

5.2 Arrays of objects

```
// Try me in the Console
var object_dataset = [
  {cx: 100, cy: 150, fill: `red`},
  {cx: 200, cy: 100, fill: `blue`}
  ];

d3.select("svg#objects")
  .selectAll("circle")
  .data(object_dataset)
  .enter()
  .append("circle")
    .attr("cx", d => d.cx)
    .attr("cy", d => d.cy)
    .attr("r", "30")
    .attr("fill", d => d.fill);
```

svg#objects

5.3 .map()

What's the issue?

In R many operations are vectorized:

```
sqrt(3)
```

```
## R output ##  [1] 1.732051
```

```
x <- c(3, 5, 7)
sqrt(x)
```

```
## R output ##  [1] 1.732051 2.236068 2.645751
```

Not so in JavaScript:

```
Math.sqrt(3);     // Try me in the Console
```

```
var x = [3, 5, 7];     // Try me in the Console
Math.sqrt(x);          // Doesn't work...
```

5.3.1 Simple arrays

Use `.map()` to operate on each array element separately. The concept is similar to `lapply()` or `purrr::map()`, but unlike in R, it's needed for simple arrays.

R

```
x <- c(3, 5, 7)
sqrt(x)
```

```
## R output ##  [1] 1.732051 2.236068 2.645751
```

JavaScript

Do something to every element of a simple array:

```
var x = [3, 5, 7];     // try me
x.map(Math.sqrt);
```

```
[4, 10, 12].map(d => d*3);     // try me
```

```
[4, 10, 12].map(function(d) {return d*3;});     // try me
```

```
[10, 20, 30, 40].map((d, i) => d*i);     // try me
```

5.3.2 Arrays of arrays

Do something to the first item of every element of a nested array:

```
[[1, 2], [3, 4]].map(d => Math.sqrt(d[0]))  // try me
```

Sum up all items in each element of the array:

```
[[1, 2, 3], [4, 5, 6]].map(d => d[0] + d[1] + d[2]); // try me
```

Created a nested array out of a simple array:

```
[10, 20, 30].map(d => [d, Math.pow(d, 2)]);
```

5.3.3 Create arrays of objects

Create an array of objects out of a simple array (note the parentheses around the object):

```
[10, 20, 30].map(d => ({n: d, nsq: Math.pow(d, 2)}));  // try me
```

```
[10, 20, 30].map((d, i) => ({index: i, value: d}));  // try me
```

5.4 Sorting

Sorting in JavaScript sorts by character and modifies the original array[1]:

```
var x = [3, 1, 5, 12, 7];      // try me
x.sort();
```

```
x;      // try me
```

If we want to sort by number, not character, we can pass the comparator function `d3.ascending` to `.sort()`:

```
var x = [3, 1, 5, 12, 7];      // try me
x.sort(d3.ascending);
```

`d3.sort()` will produce the same results:

```
var y = [3, 1, 5, 12, 7];      // try me
d3.sort(y);
```

5.5 D3 statistics

link to API

D3 brings us back to familiar ground with functions that take an *array* and return a single value. Here are D3 functions with the same names and behavior as their R equivalents:

[1] If you don't wish for that to happen, you'll need to make a deep copy.

R	D3
`min(x)`	`d3.min(x)`
`max(x)`	`d3.max(x)`
`sum(x)`	`d3.sum(x)`
`mean(x)`	`d3.mean(x)`
`median(x)`	`d3.median(x)`

A few with different names:

R	D3
`range(x)`	`d3.extent(x)`
`var(x)`	`d3.variance(x)`
`sd(x)`	`d3.deviation(x)`

Quantiles works a bit differently in D3: *the value returned depends on the sort order of x.* Therefore it is crucial to sort the array first. Next, `p` takes a single value, not an array as in R.

R	D3
`quantile(x)`	`d3.quantile(x, p)`

Thus for a single quantile we have:

```
var x = [12, 34, 1, 43, 90, 72];      // try me
d3.quantile(x.sort(d3.ascending), .25);
```

https://github.com/d3/d3/blob/master/API.md#statistics

5.6 D3 + .map()

D3 statistics functions combined with `.map()` can be helpful in a variety of situations.

Vectorizing a parameter, for example to mimic `quantile(x)` in R:

R

```
x <- c(1, 12, 34, 43, 72, 90);
quantile(x)
```

```
## R output ##     0%   25%   50%   75%  100%
## R output ##   1.00 17.50 38.50 64.75 90.00
```

JavaScript

```
var x = [1, 12, 34, 43, 72, 90];       // try me
[0, .25, .5, .75, 1].map(p => d3.quantile(x, p));
```

Sum up the first item of all elements in an array of arrays:

R

```
l <- list(c(100, 200, 40), c(300, 150, 20))
sum(purrr::map_dbl(l, ~.x[1]))
```

```
## R output ##  [1] 400
```

JavaScript

```
var dataset = [[100, 200, 40], [300, 150, 20]];      // try me
d3.sum(dataset.map(d => d[0]));
```

Sum up all items in each array to create a simple array:

R

```
l <- list(c(100, 200, 40), c(300, 150, 20))
purrr::map_dbl(l, ~sum(.x))
```

```
## R output ##  [1] 340 470
```

JavaScript

```
var dataset = [[100, 200, 40], [300, 150, 20]];      // try me
dataset.map(d => d3.sum(d));
```

Chapter 6

Scales and Axes

6.1 Scales

6.1.1 Lecture slides

Scales

6.1.2 Practice

See: *IDVW2*, Chapter 7: Scales

Practice creating an ordinal scale in the Console:

> *Open the JavaScript Console*

```
var ordscale = d3.scaleBand()
  .domain([0, 1, 2, 3, 4])
  .range([0, 100]);
```

```
ordscale(1);
```

Try other numbers: `ordscale(3);`, `ordscale(2.5);`, `ordscale(7);`, etc.

Add inner padding and try again.

See diagram here: https://github.com/d3/d3-scale#band-scales

> *Be sure to use `d3.scaleBand()`, not `d3.scaleOrdinal()` for this use case.

6.1.3 Examples

`d3.scaleBand()`

IDVW2 Chapter 9, pp. 150-153

Vertical bar chart with labels

`d3.scaleBand()` used to create an `xScale` function to convert bar numbers to pixels. Change the `w` parameter and observe how the bars are resized to fit on the SVG.

d3.scaleBand.html

`d3.scaleLinear()`

Vertical bar chart with labels

`d3.scaleLinear()` is added to create a `yScale` function to convert bar heights to pixels. Change the data and observe how the bars are resized to fit on the SVG.

d3.scaleLinear.html

6.2 Margins

6.2.1 Lecture slides

Margins

"Margin convention"

```
var w = 500;
var h = 400;
var margin = {top: 25, right: 0, bottom: 25, left: 25};
var innerWidth = w - margin.left - margin.right;
var innerHeight = h - margin.top - margin.bottom;
```

6.2.2 Example

Full example, vertical bar chart with margins: margins.html

6.3 Axes

See: *IDVW2*, Chapter 8: Axes

6.3.1 Lecture slides

Axes

6.3.2 Example

Full example: vertical bar chart with axes (open in browser): axes.html

To download and open locally: [axes.html] (https://raw.githubusercontent.com/jtr13/d3book/master/code/axes.html){target="_blank"}

Practice changing the data and seeing what happens.

Chapter 7

Interactivity

Read: *IDVW2*, Chapter 10 Interactivity

7.1 Lecture slides

interactivity.pdf

7.2 Binding event listeners

```
## R output ##  NULL
```

Open Developer Tools and try this code in the Console:

```
d3.select("svg")
  .on("click", function () {
    d3.select("svg")
      .append("text")
        .attr("x", "100")
        .attr("y", "40")
        .text("Hello World");
        });
```

7.3 What is *this*?

In the context of event handlers, "this" is the element that received the event, a.k.a. what you clicked on if it's a click event.

Examples from the first chapter:

```
d3.select("line")
  .on("click", function() {
    d3.select(this)
      .attr("stroke-width", "10");
      });
```

v6

```
d3.select("svg")
  .on("click", function(event) {
    d3.select("text")
      .text(`(${d3.pointer(event).map(Math.round)})`)
      });
```

v5

```
d3.select("svg")
  .on("click", function() {
    d3.select("text")
      .text(`(${d3.mouse(this).map(Math.round)})`)
      });
```

We can separate the function and the event listener:

```
function goyellow() {
  d3.select(this)
    .attr("fill", "yellow")
    };
```

```
d3.select("circle")
  .on("mouseover", goyellow);
```

Try this in the Console:

v6

```
d3.select("svg")
  .on("click", function (event) {
    console.log(d3.pointer(event).map(Math.round));
    });
```

v5

```
// won't work on this page
d3.select("svg")
  .on("click", function () {
    console.log(d3.mouse(this));
    });
```

7.3.1 Location on SVG

v6 `d3.pointer(event)`

v5 `d3.mouse(this)`

7.3.2 Attribute of an element

```
d3.select(this).attr("id");
```

7.3.3 Value of radio button

`d3.select(this).node().value;` (string)

`+d3.select(this).node().value;` (number)

7.4 Add / remove "buttons"

(HTML paragraphs are used as buttons in this example.)

HTML:

```
<p id="add">Add an element</p>
<p id="remove_left">Remove bar (left)</p>
<p id="remove_right">Remove bar (right)</p>
```

JavaScript:

```
d3.selectAll("p")
    .on("click", function () {
      var paraID = d3.select(this).attr("id");
      if (paraID == "add") {
          var newvalue = Math.floor(Math.random()*400);
          bardata.push(newvalue);
          } else if (paraID == "remove_left") {
          bardata.shift();
```

```
          } else {
          bardata.pop();
          };
      update(bardata);
      });
```

7.5 Putting it all together

Vertical bar chart with add / remove buttons and general update pattern

vertical_bar.html

7.6 Dependent event listeners

In these examples, the behavior or existence of one event listener depends on another.

7.6.1 Global variable example

Here the circle click behavior depends on the value of the radio button: if the "Move left" radio button is checked, the circle will move left *when clicked*. If the "Move right" radio button is checked, the circle will move right *when clicked*.

A global variable is used to keep track of the radio button value. The event listener on the circle conditions the behavior on the value of this global variable.

```
## R output ##  NULL
```

```
// global variable keeps track of which radio button is clicked
var action = "left";
d3.select("div#rad")
  .selectAll("input")
  .on("click", function() { action = d3.select(this).node().value; });

// circle click behavior depends on value of "action"
d3.select("svg#radio").select("circle")
  .on("click", function () {
    if (action == "left") {
      var cx_new = +d3.select(this).attr("cx") - 50;
      if (cx_new < 20) cx_new = 20;
      } else {
```

```
      var cx_new = +d3.select(this).attr("cx") + 50;
      if (cx_new > 280) cx_new = 280;
      }
    d3.select(this)
      .transition()
      .duration(500)
      .attr("cx", cx_new);
      });
```

7.6.2 Turn off event listener

In this example, the event listeners on the squares are turned on or off depending on the value of the radio button. Event listeners can be removed by setting the behavior to `null`.

```
## R output ##  NULL
```

```
// movement function
var jump = function () {
      d3.select(this).transition().duration(500)
      .attr('y', '0')
      .transition().duration(500).ease(d3.easeBounce)
      .attr('y', '75');
};

// initial setup: add event listener to red square
d3.select("svg#radio2")
  .select("rect#red")
  .on("click", jump);

// switch event listeners if radio button is clicked
d3.select("div#rad2").selectAll("input")
  .on("click", function () {
  if (d3.select(this).node().value == "blue") {
    d3.select("svg#radio2").select("rect#blue").on("click", jump);
    d3.select("svg#radio2").select("rect#red").on("click", null);
    } else {
    d3.select("svg#radio2").select("rect#red").on("click", jump);
    d3.select("svg#radio2").select("rect#blue").on("click", null);
    }
});
```

Chapter 8

Transitions

Read *IDVW2*, Chapter 9: transitions section (pp. 158-178)

8.1 Examples

Open Developer Tools and try in the Console:

```
d3.select("svg")
  .selectAll("circle")
  .transition()
  .duration(2000)
  .attr("cx", "275");
```

```
d3.select("svg")
  .selectAll("circle")
  .transition()
  .duration(2000)
  .attr("cx", "25")
  .attr("fill", "green");
```

8.2 Do this

Run simultaneous transitions on *different* selections:

```
d3.select("svg").selectAll("circle#henry").transition()
    .duration(2000).attr("cx", "275");
```

```
d3.select("svg").selectAll("circle.apple").transition()
    .duration(2000).attr("cx", "25");
```

Run sequential transitions on the same selection in one chain:

```
d3.select("svg").selectAll("circle")
  .transition().duration(2000).attr("cx", "275")
  .transition().duration(2000).attr("cx", "25");
```

Transition from *something* to *something*:

```
d3.select("svg").append("circle")
    .attr("cx", "200")
    .attr("cy", "100")
    .attr("r", "5")
    .attr("fill", "lightblue")
    .transition()
    .duration(4000)
    .attr("r", "25")
    .attr("fill", "blue");
```

8.3 Not this

DO NOT run two transitions on the same selection at the same time (see p. 172).

(What works in the Console *will not work* in a script.)

```
d3.select("svg").selectAll("circle").transition()
    .duration(2000).attr("cx", "250");
d3.select("svg").selectAll("circle").transition()
    .duration(2000).attr("cx", "75");
```

DO NOT transition from *nothing* to something:

```
d3.select("svg").append("circle")
    .transition()
    .duration(2000)
    .attr("cx", "200")
    .attr("cy", "100")
    .attr("r", "25")
    .attr("fill", "red");
```

DO NOT store a selection with a transition (it's no longer a selection with the transition):

Try this:

```
var circ = d3.select("svg")
  .selectAll("circle")
  .data([50, 95, 100, 200, 50, 150, 250])
  .enter()
  .append("circle")
    .attr("cx", d => d)
    .attr("cy", "100")
    .attr("fill", "blue")
    .attr("r", "0")
    .transition()
    .duration(2000)
    .attr("r", "25");
```

And then this:

```
circ.attr("fill", "green");
```

DO NOT put a transition before a merge:

```
d3.select("svg")
  .selectAll("circle")
  .transition()
  .duration(2000)
    .attr("cx", "300")
  .merge("oldcirc")
    .attr("fill", "green");
```

BE AWARE that not everything transitions (for example, text doesn't.)

8.4 Strategy

Example 1

Think carefully about what you want to happen, and then decide what goes before and after the transition.

Plan what you want to happen:

1. New bars appear with orange fill.

2. All bars in proper location.
3. Fill color for all bars transitions to blue.

Add code:

```
var bars = d3.selectAll("svg")
  .selectAll("rect")
  .data(dataset);

bars.enter()                    // 1. new bars
  .append("rect")
    .attr("fill", "orange")
  .merge(bars)                  // 2. all bars in location
    .attr("x", ...)
    .attr("y", ...)
    .attr("width", ...)
    .attr("height", ...)
  .transition()
  .duration(2000)
    .attr("fill", "blue");   // 3. all transition to blue
```

Example 2

Plan

1. A new bar appears from the right side.
2. Once it reaches its spot, it turns purple.

(Try this one.)

```
var bars = d3.selectAll("svg")
  .append("g")
  .selectAll("rect")
  .data([125]);

bars.enter()
  .append("rect")
    .attr("x", "325")
    .attr("y", d => 200 - d )
    .attr("width", "30")
    .attr("height", d => d)
    .attr("fill", "blue")
  .transition()
```

```
  .duration(3000)
    .attr("x", "50")   // transition only affect "x"
  .transition(2000)
  .duration(2000)
    .attr("fill", "purple");  // 2nd transition
```

Further reading: Working with Transitions.

Chapter 9

Object Constancy

9.1 No object constancy

Transitions

Off On

Add an element

Remove bar (right)

Remove bar (left)

Of note:

- Rather than smoothly transitioning off to the left, all bars are resized when “Remove bar (left)” is clicked
- When ”Remove bar (right) is clicked, the bar on the right immediately disappears, and then the remaining bars transition to their new places to the right.

9.2 Object constancy

Slides:

object_constancy.pdf

Example: object_constancy.html

Of note:

- Bars now smoothly transition off to the left and right

9.2.1 Practice joining data by key

Download and open keys.html

(or open this online version) in a new tab.

Try the following:

1.

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

var dataset = [{key: 12, x: 100, y: 200},
               {key: 16, x: 250, y: 300}];

svg.selectAll("text")
  .data(dataset, d => d.key)
  .exit()
  .remove();
```

Then:

```
svg.selectAll("text")
  .attr("x", d => d.x)
  .attr("y", d => d.y);
```

2.

(Refresh)

```
var dataset = [{key: 23, x: 300, y: 150},
               {key: 5, x: 450, y: 270}];

var databind = svg.selectAll("text")
  .data(dataset, d => d.key)

databind.exit().remove();
```

Then:

```
databind.enter().append("text")
  .attr("x", d => d.x)
  .attr("y", d => d.y)
  .text(d => `key: ${d.key}`);
```

3. Experiment with other data binds.

Chapter 10

Reading files

As you've surely noticed by this point, many things in JavaScript operate on an *asynchronous* basis. Code is not executed linearly from beginning to end but rather in response to various triggers. For example, event listeners behave asynchronously: code will execute only if a mouse click event occurs.

The benefit to reading files asynchronously is that we don't have to wait to while a file loads for other things to happen. It would be very frustrating to navigate to a new web page and have to wait for all the scripts to finish before we could do anything on the page.

10.1 Promises

Loading data is one area where D3 v5 introduces major changes from D3 v4. While v4 uses callbacks, v5 switches to promises, as promises facilitate cleaner and more flexile code than callbacks.

The concept is simple. We want to control what code needs to wait until data loaded to be executed and what doesn't. We can do that with the following structure:

```
var rowConverter = function (d) {
  return {
    disp: +d.disp,
    mpg: +d.mpg,
    carname: d.carname,
    cylcolor: d.cylcolor
    }
};
```

```
d3.csv("data/mtcars.csv", rowConverter)
  .then(function(data) {

// stuff that requires the loaded data

  })
  .catch(function(error) {

// error handling

  });
```

The row converter function is used to select variables and change data types ("+" converts to floating point). `d3.csv()` returns a promise. If the promise is resolved, the `.then()` function will execute; if the promise is rejected, the `.catch()` function will execute.

> *Forget the mindset that you read files and store them file in variables for later use. It doesn't work that way here. The data is read in and acted on immediately. If most of the code requires loaded data, then most of the code will appear in the `.then()` method.*

A simple example of loading data in **v5** can be found in this block. In contrast to the example above, an anonymous row converter function (with arrow functions) is used instead of calling a separate row converter function. Note as well that it's not necessary to include all variables in the row converter as this author has done. To test, fork the block and delete all the variables that aren't used, so that the row converter (line 56) becomes:

```
d => ({
            HighwayMpg: parseInt(d.HighwayMpg),
            Horsepower: parseInt(d.Horsepower),
        })
```

You will see that the code still works.

For more about `d3.csv()`, see the `d3.fetch` API.

10.2 Local server

For security reasons, Chrome does not let you read local files. To be able to do so, you can run a local server. One option is http-server. Follow the instructions to install `http-server`, navigate in Terminal to the directory with your html file, and then enter `http-server`. You should get a message like this:

```
Starting up http-server, serving ./
Available on:
  http://127.0.0.1:8080
  http://192.168.1.54:8080
Hit CTRL-C to stop the server
```

Copy and paste the URL in the browser and you should see your page with data loaded. As indicated, Control-c will stop the server.

10.3 Other local options

A simple way to avoid this issue is to upload data files to GitHub and read them from there. There are other workarounds, including opening Chrome from the command line with the `--allow-file-access-from-files` flag.

10.4 Hosting online

An alternative to the options above are to avoid the issue by hosting your code online. Options for doing so are covering in the chapter on sharing D3 online.

Chapter 11

Share D3 online

There are a number of ways you can share your D3 code online. Even if you're not sharing, there are advantages to an online setup, for example, not having to set up a local server as described in the chapter on reading files.

11.1 VizHub

My top recommendation is to use vizhub.com, created by Curran Kelleher, to host d3 visualizations online. Watch the video on the home page, "How to Use VizHub" for a quick how-to tutorial. Note however that using ES6 imports to import D3 modules separately, such as `import { select } from 'd3';` as demoed in the video is completely optional; it is sufficient to link to the full D3 library in the `<head>` section of `index.html` as we've been doing:

```
<script src="https://d3js.org/d3.v6.js"></script>
```

As such you will not have `index.js` and generated `bundle.js` files as in the example.

When browsing examples, remember that different versions of D3 are not compatible, so be sure to look at v6 examples, or know what you need to update from v4 and v5 examples. In addition, as with all code sharing sites, remember that not all examples are good examples. Examples by Curran Kelleher, the creator of VizHub, are good models to follow.

11.2 bookdown

Including D3 code directly in a bookdown book hosted online (such as GitHub Pages) has the advantage that everyone is in one document. On the downside

the book must be knit to view the html which happens quickly without R code but is still an extra step.

There are a few workflow options:

11.2.1 Everything in the .Rmd

Include a link to the D3 library and then all of your code between script tags, and then your D3 between a second set of script tags:

```
<script src="https://d3js.org/d3.v6.js"></script>
<script>
...
</script>
```

(You don't need `<html>`, `<head>`, and `<body>` tags.)

11.2.2 Put D3 in a separate .js file

In your Rmd file include the link to the D3 library as well as a link to a file with your D3 code:

```
<script src="https://d3js.org/d3.v6.js"></script>
<div id="mysvg"></div>
<script src="myjs.js"></script>
```

You can then create an identical `.html` file for testing purposes only. This has the advantage of not having to render the full bookdown book each time you would like to observe changes to your D3 code.

11.2.3 Put D3 in a separate .js file

Create your entire visualization in a separate `.html` file and then include in your `.Rmd` file with `<iframe>`:

```
<iframe src="mybarchart.html" width="400" height="300"></iframe>
```

11.3 Observable

Observable, created by D3 author Mike Bostock, is the official D3 web tool for creating and sharing D3 code. It is a powerful, popular tool–all new D3 code examples are now presented in Observable–but program flow is different than it is for stand-alone JavaScript. If you're interested in learning more, see the collection of Observable Tutorials.

11.4 Blockbuilder 2015-2020

As an aside, blockbuilder.org was a very popular service for sharing D3 code online, but by the end of 2020, will no longer allow users to create new blocks, and going forward will serve as an archive of previously created ones. If you use it for this purpose, remember to pay attention to the version of D3 in use; some authors have included the version number in the block titles but this practice is not widespread. Remember as well that not all examples are good examples, both in terms of the graph choices and the quality of the code.

A few reputable block authors, in terms of quality of D3 code, are:

Mike Bostock (creator of D3)

Ian Johnson (creator of blockbuilder.org)

Curran Kelleher (data viz consultant / educator)

Malcolm Maclean (author of D3 Tips and Tricks, v5.x, lots of simple v5 examples with v5 in title)

Kent Russell (creator of interactive `parcoords` package in R)

Chapter 12

Line charts

Read: *IDVW2*, Chapter 11 Using Paths

12.1 Lecture slides

line_charts.pdf

12.2 SVG `<line>` element

(Use for two points only.)

```
<line x1="0" y1="80" x2="100" y2="20" stroke="black" />
```

```
var x1 = 0;
var y1 = 80;
var x2 = 100;
var y2 = 20;

d3.select("svg")
  .append("line")
  .attr("x1", x1)
  .attr("x2", x2)
  .attr("y1", y1)
  .attr("y2", y2);
```

12.3 SVG <path> element

(Use if you have more than two points.)

```
<svg width = "500" height = "400">
  <path d="M 50 400 L 100 300 L 150 300 L 200 33 L 250 175
     L 300 275 L 350 250 L 400 125" fill="none"
      stroke="red" stroke-width="5">
  </path>
</svg>
```

`d` attribute:

M = move to

L = line to

More options: https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/d

12.4 SVG editors

(a digression)

ex. SVG-edit

Save plots as SVG files in R

Save plots as SVG files:

```
svg("xsquared.svg")
plot(1:10, (1:10)^2, axes=F)
dev.off()

library(svglite)
svglite("xsquared2.svg")
plot(1:10, (1:10)^2, axes=F)
dev.off()
```

Set graphics device to svg in code chunk options:

```
{r dev="svglite"}
```

12.5 Back to line charts

Format that we have:

Day	High Temp
April 1	60
April 2	43
April 3	43
April 4	56
April 5	45
April 6	62
April 7	49

Format that we need looks something like this:

```
<path class="line" fill="none" d="M0,149.15254237288136L71.42857142857143,264.40677966
```

12.5.1 Step 1: Create a line generator

Expects data in an array of 2-dimensional arrays, that is, an array of (x,y) pairs:

```
var dataset = [ [0, 60], [1, 43], [2, 43], [3, 56], [4, 45], [5, 62], [6, 49] ];

var mylinegen = d3.line()
```

Test it in the Console:

```
mylinegen(dataset);
```

Add an ordinal scale for x:

```
var xScale = d3.scaleBand()
    .domain(d3.range(dataset.length))
    .range([0, 500])
```

... and a linear scale for y:

```
var yScale = d3.scaleLinear()
    .domain([d3.min(dataset, d => d[1]) - 20,
             d3.max(dataset, d => d[1]) + 20])
    .range([400, 0]);
```

*Why `d[1]` instead of `d`? (See p. 122)

Add accessor functions `.x()` and `.y()`:

```
mylinegen
    .x(d => xScale(d[0]))
    .y(d => yScale(d[1]));
```

Test again:

```
mylinegen(dataset);
```

Now let's add a `<path>` element with that `d` attribute: (this step is just for learning purposes…)

```
var mypath = mylinegen(dataset);

d3.select("svg").append("path").attr("d", mypath)
    .attr("fill", "none").attr("stroke", "red")
    .attr("stroke-width", "5");
```

12.5.2 Step 2: Put the line generator to work

Now let's do it the direct way: bind the *datum* and calculate the path in one step:

```
d3.select("svg").append("path")
    .datum(dataset)
    .attr("d", mylinegen)
    .attr("fill", "none")
    .attr("stroke", "teal")
    .attr("stroke-width", "5");
```

Finally, we'll add a class and style definitions:

```
<style>
  .linestyle {
    fill: none;
    stroke: teal;
    stroke-width: 5px;
  }
</style>
```

The `append("path")` line becomes:

```
svg.append("path")
    .datum(dataset)
    .attr("d", mylinegen)
    .attr("class", "linestyle");
```

Putting it all together, we have:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Line generator</title>
    <script src="https://d3js.org/d3.v6.js"></script>
    <style type ="text/css">
      .linestyle {
        fill: none;
        stroke: teal;
        stroke-width: 5px;
      }
    </style>
  </head>
  <body>
    <script>
      var w = 500;
      var h = 400;
      var svg = d3.select("body").append("svg")
          .attr("width", w).attr("height", h);
      var dataset = [ [0, 60], [1, 43], [2, 43], [3, 56],
          [4, 45], [5, 62], [6, 49] ];
      var xScale = d3.scaleBand()
        .domain(d3.range(dataset.length))
        .range([0, w]);
      var yScale = d3.scaleLinear()
        .domain([d3.min(dataset, d => d[1]) - 20,
                 d3.max(dataset, d => d[1]) + 20])
        .range([h, 0]);
      var mylinegen = d3.line()
        .x(d => xScale(d[0]))
        .y(d => yScale(d[1]));
      svg.append("path")
         .datum(dataset)
         .attr("d", mylinegen)
         .attr("class", "linestyle");
    </script>
```

```
  </body>
</html>
```

And another example with axes:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title></title>
        <script src="https://d3js.org/d3.v6.js"></script>
        <style type ="text/css">
      .line {
        fill: none;
        stroke: red;
        stroke-width: 1.5px;
      }
        </style>
    </head>
    <body>
    <svg width="600" height="400"></svg>
    <script>
      var svg = d3.select("svg")
      var margin = {top: 20, right: 50, bottom: 30, left: 50}
      var width =  +svg.attr("width") - margin.left - margin.right
      var height = +svg.attr("height") - margin.top - margin.bottom
      var g = svg.append("g").attr("transform", `translate(${margin.left}, ${margin.top})`);
      var parseTime = d3.timeParse("%d-%b-%y");
      var xScale = d3.scaleTime().range([0, width]);
      var yScale = d3.scaleLinear()
        .domain([20, 80])
        .range([height, 0]);
      var line = d3.line()
        .x(d => xScale(d.date))
        .y(d => yScale(d.high));
      var data =
      [{"date":"1-Apr-18","high":60},
      {"date":"2-Apr-18","high":43},
      {"date":"3-Apr-18","high":43},
      {"date":"4-Apr-18","high":56},
      {"date":"5-Apr-18","high":45},
      {"date":"6-Apr-18","high":62},
      {"date":"7-Apr-18","high":49}];
      data.forEach(function(d) {
            d.date = parseTime(d.date);
```

```
      });

      xScale
        .domain(d3.extent(data, d => d.date));

      g.append("g")
          .attr("transform", `translate(0, ${height})`)
          .call(d3.axisBottom(xScale).ticks(7));
      g.append("g")
          .call(d3.axisLeft(yScale))
      g.append("path")
          .datum(data)
          .attr("class", "line")
          .attr("fill", "none")
          .attr("d", line);
    </script>
    </body>
</html>
```

(Also uses: `d3.timeParse()` and JavaScript `Array.foreach()`)

12.6 Additional Resources

Multiple Time Series in D3 by Eric Boxer (EDAV 2018)

Chapter 13

Layouts

IDVW2, Chapter 13 Layouts (`d3.stack()` only, pp. 264-270)

13.1 Lecture slides

layouts.pdf

Chapter 14

Debugging Tips

- Make extensive use of Elements to see what's being added to the DOM.
- Make extensive use of Console to check the values of variables and/or test code.
- Pay attention to errors in the Console.
- Use `console.log()` esp. in functions
- Post Minimal Working Examples on Piazza. See:
  "How to create a Minimal, Complete, and Verifiable example"
  "So you've been asked to make a reprex"
  (But don't worry if it's not perfect, we're not going to judge.)
- Use a text editor that helps you identify unmatched () {} [].

Chapter 15

Your solutions here

Add your code and submit a PR.

Web tech: shapes

D3 in the Console: green circles

D3 in the Console: blue circles

D3 in the Console: data bind

Update, Enter, and Exit: horizontal bar chart

Update, Enter, and Exit: merge

Chapter 16

More chapters coming soon

via GIPHY

16.1 Maintainer links

Edit `_bookdown.yml`

Chapter 17

Appendix: advanced CSS

Work-in-progress

17.1 Buttons

17.1.1 `.active`

This is a *class* not a pseudoclass, indicates which button(s) are pressed.

17.1.2 `:active`

A pseudoclass. Clicking is the main way to trigger an `:active` state.

17.1.3 `:focus`

A pseudoclass. Mainly used for tabbing. Clicking *may* focus a button but not always. A box shadow such as `box-shadow: 0 0 0 3px lightskyblue;` is a good choice

17.1.4 `:hover`

As expected, can have separate hover behavior for `.active` buttons.

17.1.5 Example

See the Pen Button .active vs :active by Joyce Robbins (https://codepen.io/jtr13) on CodePen.