D3 for R Users

Joyce Robbins

2019-11-19

Contents

Welcome

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*–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. (To add to the complication, the current version of D3 is v5. However, since differences between v4 and v5 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.

In terms of content, we will use certain ES6 conventions not covered in *IDVW2* that make coding easier (and more like R!). 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.

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.)

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.

1.1 Get ready

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

2. Download a copy of `shapes.html`: open the following page and then click *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. In Chrome, click *View, Developer, Developer Tools,* then the Elements tab.

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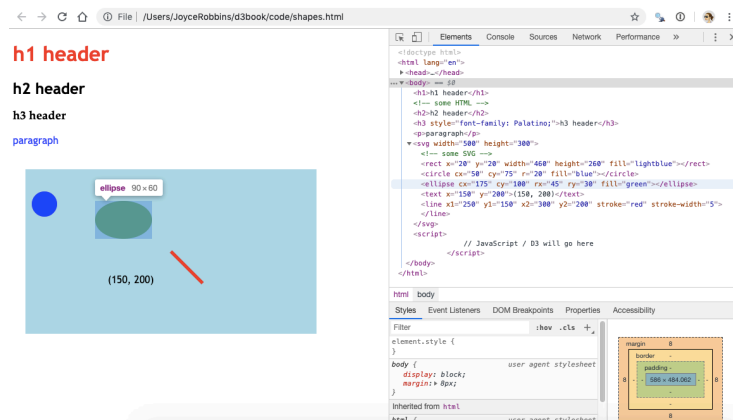
3. Now try the reverse: right click on elements on the web page, choose "Inspect" and see what is highlighted in the Elements pane. Get comfortable with the connection between the code on the right and the rendered elements on the left.

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   Transitions

1. Try these:

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

d3.select("ellipse").transition().duration(2000).attr("transform", "translate (40(

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   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?

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

Chapter 2

Web tech

Read: Chapter 3 "Technology Fundamentals" (pp. 17-62)

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.v5.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>
```

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:

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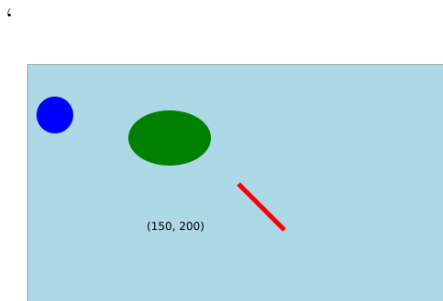
There are very few SVG tags that you'll need to know, and once we get going with D3, you will not have to code any SVG manually. It is worth doing a little to become familiar with the format and in particular to get used to the new location of the origin.

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.

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.
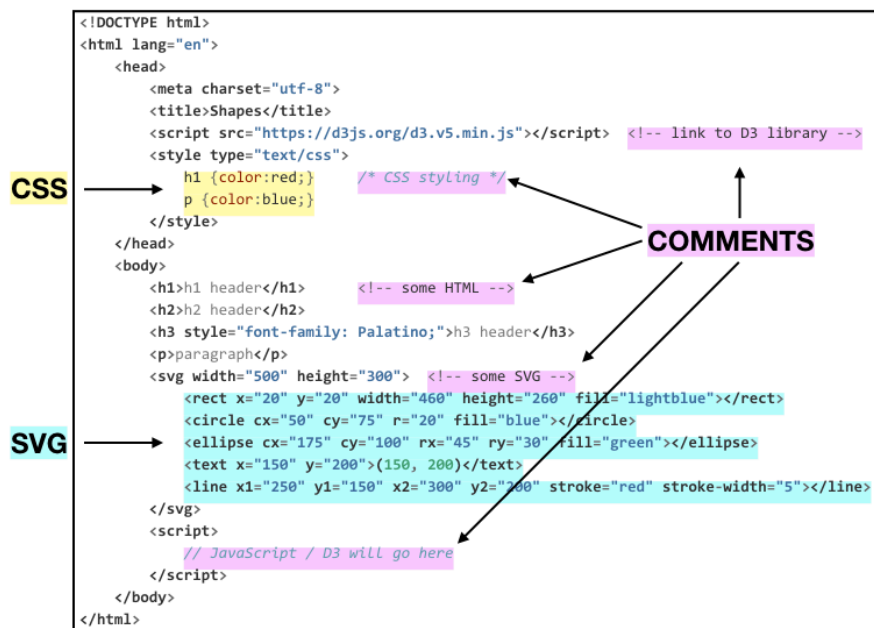
Of note:

- 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 fresh copy of `shapes.html`. (Reminder: open the following page and then click *File, Save Page As...*: `shapes.html`). Open the file in a text editor of your choice on one half of your screen. If you don't want to think about it, just use RStudio since it's already installed and provides syntax highlighting for `.html`. On the other half of your screen open the same file in Chrome. Developer Tools should not be open; we will not be using the Console. As you make changes to the `.html` file, save the file and then refresh the browser to see the effects. Keyboard shortcuts to save and refresh are helpful here.

Your screen should look like this:
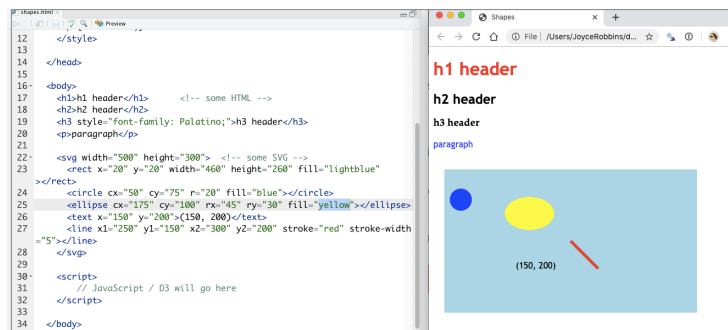
1. Add an additional circle to the svg.

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

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>
```

Web scrapping is fun.

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

```
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>
```

I want to move left.

Hover to execute this code:

```
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:

```
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!*

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 storying 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("r", 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

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.

Open Developer Tools and copy and paste the code below in the Console. (Click above to close the table of contents on the left so you'll have more screen space.)

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!

More examples from Mike Bostock (creator of D3):

General Update Pattern, I

General Update Pattern, II

General Update Pattern, III

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

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>EDAV5_1</title>
    <script src="https://d3js.org/d3.v5.min.js"></script>
  </head>

  <body>

    <script id="s1">

// Create svg and initial bars
```

```javascript
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

```javascript
var array_dataset = [[100, 200, 40], [300, 150, 20]];

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

5.2   Arrays of objects

```javascript
var object_dataset = [ {
    cx: 100,
    cy: 200,
    fill: `red`
  },
```

```
  {
    cx: 300,
    cy: 100,
    fill: `blue`
  }];

d3.select("svg")
  .selectAll("circle")
  .data(object_dataset)
  .enter()
  .append("circle")
  .attr("cx", d => d.cx)
  .attr("cy", d => d.cy)
  .attr("r", "50")
  .attr("fill", d => d.fill);
```

5.3   .map()

Take *one data value*, return one value.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/
Global_Objects/Math#Methods

```
Math.sqrt(3);

var x = [3, 5, 7];

Math.sqrt(x); // oops, it's not R

Math;
```

map() to the rescue

Use map to operate on each array element separately.

Analogy:

R list:lapply() :: JavaScript array:.map()

(What is the tidyverse equivalent of lapply()? map()!!!!!!)

Do something to every element of a simple array:

```
x.map(Math.sqrt);

x.map(d => d*3);
```

```
x.map(function(d) {return d*3;});

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

My nested array language:

[*element1, element2*]

element1 = [*item1, item2, item3*]

element2 = [*item1, item2, item3*]

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

```
array_dataset.map(d => Math.sqrt(d[0]))
```

Sum up all items in each element of the array:

```
array_dataset.map(d => d[0] + d[1] + d[2]);
```

Created a nested array out of a simple array:

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

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)}));

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

5.4   D3 statistics

Take *arrays* –> return single values (like R!)

```
var x = [12, 34, 1, 43, 90, 72];

d3.mean(x);

d3.sum(x);

d3.min(x);

d3.max(x);
```

Take an array, return two values:

```
d3.extent(x);  \\ Think "range"
```

Take a single value, return an array:

```
d3.range(5); \\ Think: count from 0 to n-1
```

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

5.5   D3 + .map()

Helpful when dealing with arrays of arrays, or arrays of objects

```
var dataset = [[100, 200, 40], [300, 150, 20]];

d3.sum(dataset.map(d => d[0]));

dataset.map(d => d3.sum(d));

d3.range(5).map(d => [d - 1, d, d + 1]);
```

5.6   Exercise: : math

Use the following to find the sample standard deviation of 3, 5, 7, 8 and 9.

```
d3.sum()
d3.mean()

Math.sqrt()
Math.pow()

.map()
```

Chapter 6

Scales and axes

6.1 Lecture slides

Scales

Practice creating an ordinal scale in the 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.

6.2 Examples

6.2.1 d3.scaleBand()

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

6.2.2 `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

6.3 Axes

(to be added)

Chapter 7

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 8

More chapters coming soon

via GIPHY

8.1   Maintainer links

Edit `bookdown.yml`