Interactive Data Visualisation

Data-driven design and interaction

Dr Ruairi O’Reilly

To be covered:

- Enabling interactivity for graphical elements
- Working with color effectively
- Loading traditional HTML for use as pop-ups
- Loading external SVG icons into charts

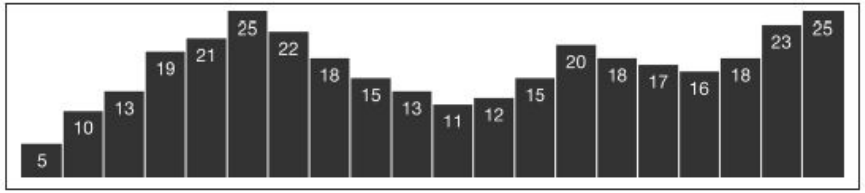
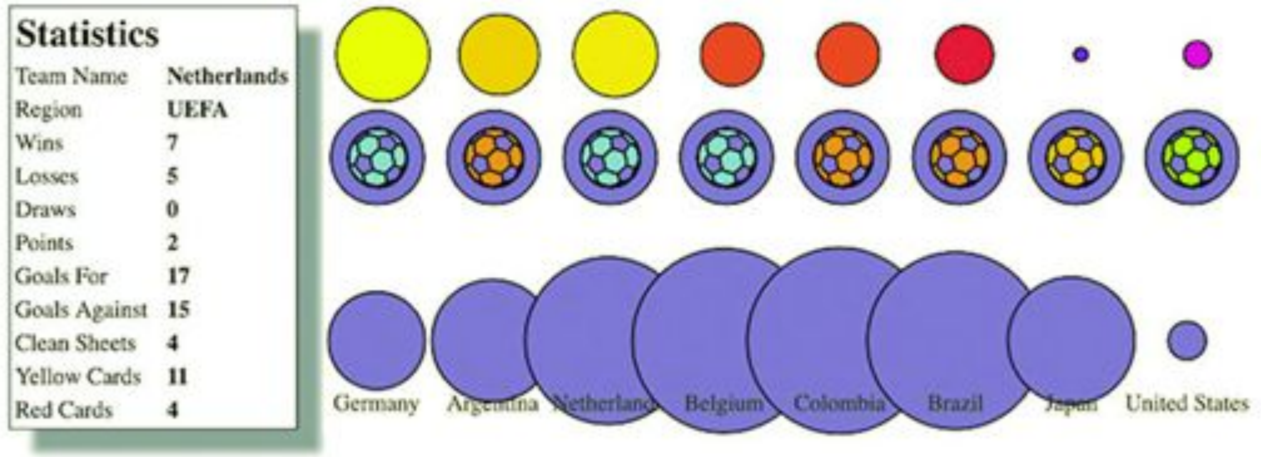


Figure 1-1. Data values mapped to visuals

Design from a broad perspective

- not only on graphical design but on interaction design, project architecture, and the integration of pre generated content.
- Highlight the connections between D3 and other methods of development, (compatible libraries typically used alongside D3, integrating HTML and SVG resources created using other tools)
- Focus on how to use particular D3 functionality to follow the best practices established by design professionals to create a simple data visualization
- Example - 2014 World Cup

Figure 3.1. This chapter covers loading HTML from an external file and updating it (section 3.3.2), as well as loading external images for icons (section 3.3.1), animating transitions (section 3.2.2), and working with color (section 3.2.4).



Statistics

Team Name	**Netherlands**
Region	**UEFA**
Wins	**7**
Losses	**5**
Draws	**0**
Points	**2**
Goals For	**17**
Goals Against	**15**
Clean Sheets	**4**
Yellow Cards	**11**
Red Cards	**4**

Project architecture

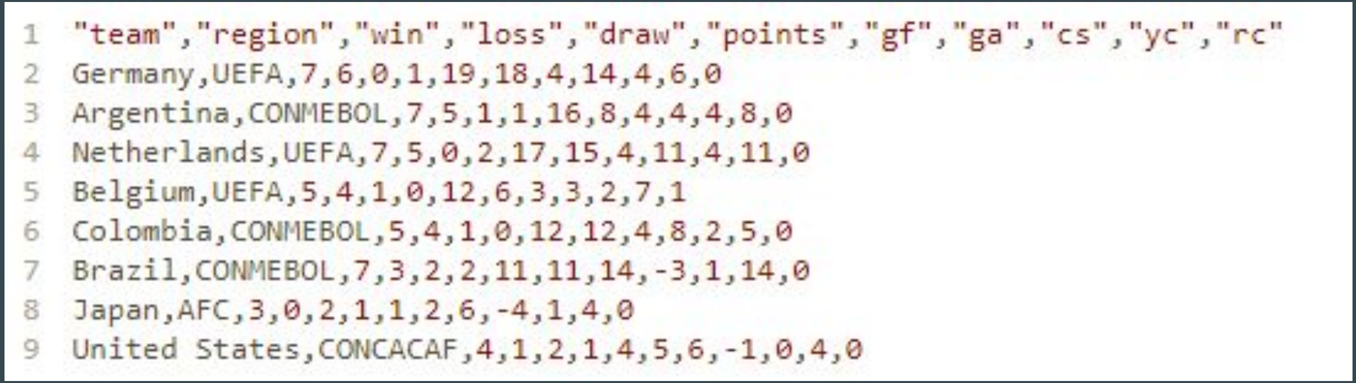
Single web page with an interesting visualization

vrs

An application providing multiple points of interaction and different states

Data

Two forms: either dynamically delivered via server/API or in static files. If you're pulling data dynamically from a server or API, it's possible that you'll have static files as well. A good example of this is building maps, where the base data layer (such as a map of countries) is from a static file and the dynamic data layer (such as the places where tweets are made) comes from a server.



```
"team","region","win","loss","draw","points","gf","ga","cs","yc","rc"
Germany,UEFA,7,6,0,1,19,18,4,14,4,6,0
Argentina,CONMEBOL,7,5,1,1,16,8,4,4,4,8,0
Netherlands,UEFA,7,5,0,2,17,15,4,11,4,11,0
Belgium,UEFA,5,4,1,0,12,6,3,3,2,7,1
Colombia,CONMEBOL,5,4,1,0,12,12,4,8,2,5,0
Brazil,CONMEBOL,7,3,2,2,11,11,14,-3,1,14,0
Japan,AFC,3,0,2,1,1,2,6,-4,1,4,0
United States,CONCACAF,4,1,2,1,4,5,6,-1,0,4,0
```

Resources

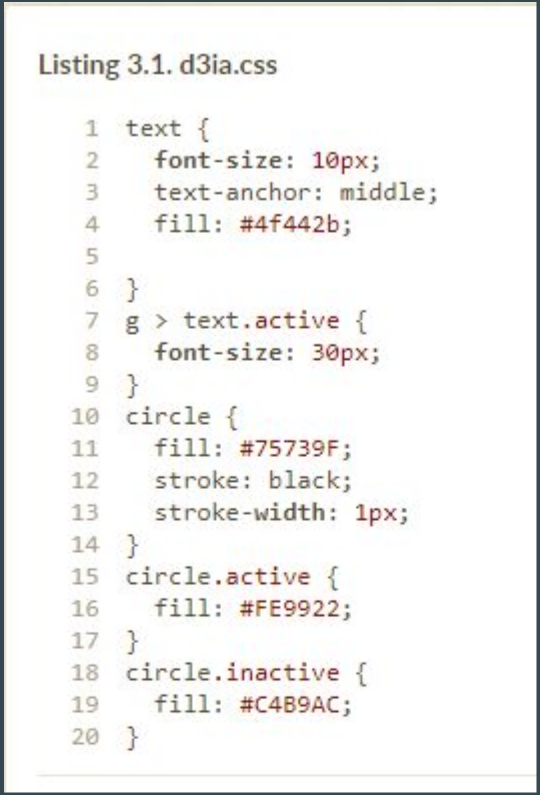
Pregenerated content, like hand-drawn SVG and HTML components, comes as an external file that you'll need to know how to load.

In more advanced projects, this can take the form of templates or resources that are imported or rolled up into your project using more sophisticated methods involving a build process. (e.g. integrating Webpack)

Images - PNG vrs SVG, SVG images as static images and not as code that you want to manipulate in D3

Style sheets

Our style sheet shown in listing 3.1 has classes for the different states of the SVG elements we're dealing with, including SVG text elements. Remember that regular text in CSS uses color while SVG <text> uses fill to set its color.



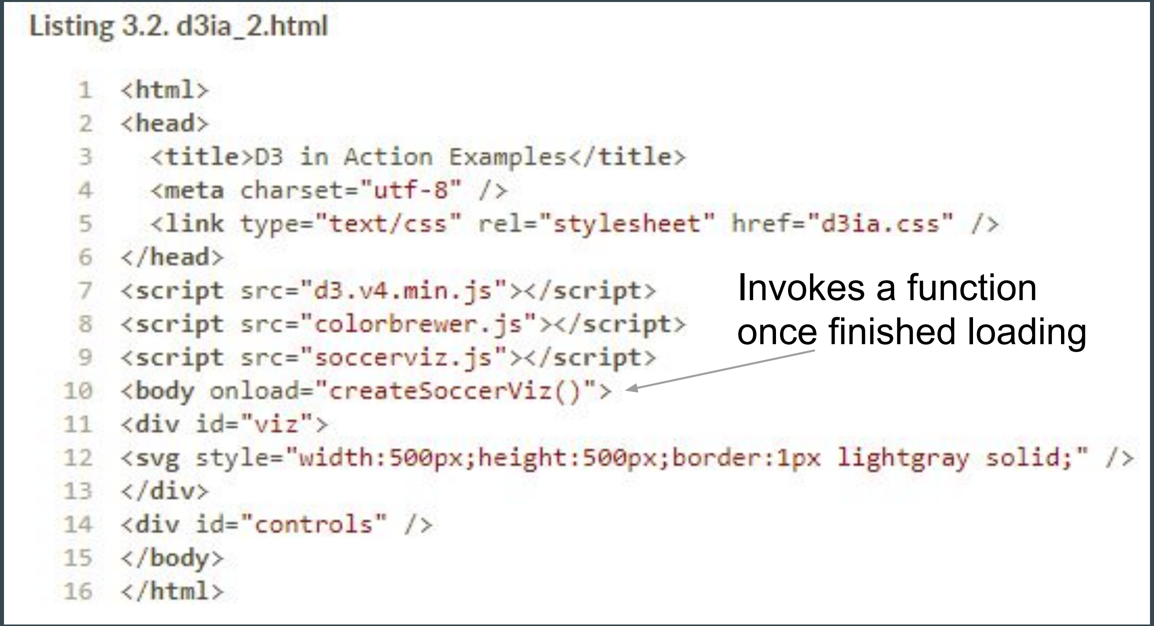
Listing 3.1. d3ia.css

```css
text {
  font-size: 10px;
  text-anchor: middle;
  fill: #4f442b;

}
g > text.active {
  font-size: 30px;
}
circle {
  fill: #75739F;
  stroke: black;
  stroke-width: 1px;
}
circle.active {
  fill: #FE9922;
}
circle.inactive {
  fill: #C4B9AC;
}
```

External libraries - for our World Cup example

We'll use two more .js files besides d3.min.js, which is the minified D3 library. The first is soccerviz.js, where we'll put the functions we'll build and use. The second is colorbrewer.js, which provides a set of predefined color palettes that we'll find useful.

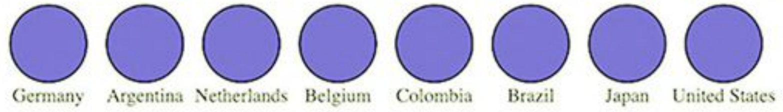

Listing 3.2. d3ia_2.html

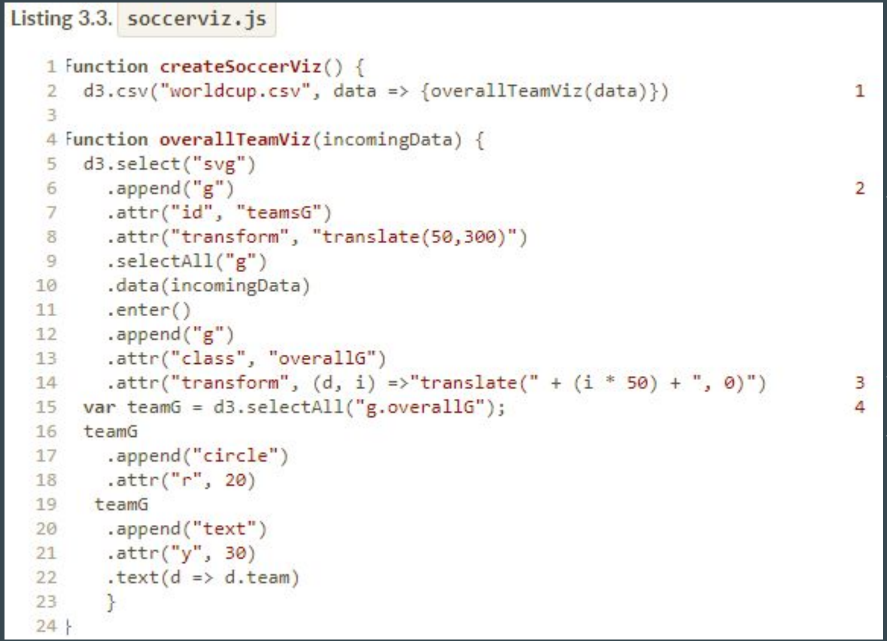
```html
<html>
<head>
  <title>D3 in Action Examples</title>
  <meta charset="utf-8" />
  <link type="text/css" rel="stylesheet" href="d3ia.css" />
</head>
<script src="d3.v4.min.js"></script>
<script src="colorbrewer.js"></script>
<script src="soccerviz.js"></script>
<body onload="createSoccerViz()">
<div id="viz">
<svg style="width:500px;height:500px;border:1px lightgray solid;" />
</div>
<div id="controls" />
</body>
</html>
```

Invokes a function once finished loading

createSoccerViz() - loads the data and binds it to create a labeled circle for each team.

Figure 3.2. Circles and labels created from a CSV representing 2014 World Cup statistics.





Listing 3.3. `soccerviz.js`

```
function createSoccerViz() {
  d3.csv("worldcup.csv", data => {overallTeamViz(data)})                1

function overallTeamViz(incomingData) {
  d3.select("svg")
    .append("g")                                                        2
    .attr("id", "teamsG")
    .attr("transform", "translate(50,300)")
    .selectAll("g")
    .data(incomingData)
    .enter()
    .append("g")
    .attr("class", "overallG")
    .attr("transform", (d, i) =>"translate(" + (i * 50) + ", 0)")       3
  var teamG = d3.selectAll("g.overallG");                               4
  teamG
    .append("circle")
    .attr("r", 20)
   teamG
    .append("text")
    .attr("y", 30)
    .text(d => d.team)
    }
}
```

Loads the data and runs createSoccerViz with the loaded data

Appends a <g> to the <svg> canvas to move it and center its contents more easily

Creates a <g> for each team to add labels or other elements as we get more ambitious

Assigns the selection to a variable to refer to it without typing out d3.selectAll() every time

Interactive style and DOM

Creating interactive data visualization is necessary for your users to deal with large and complex datasets. And the key to building interactivity into your D3 projects is the use of events, which define behaviors based on user activity. After you learn how to make your elements interactive, you'll need to understand D3 transitions, which allow you to animate the change from one color or size to another. With that in place, you'll turn to learning how to make changes to an element's position in the DOM so that you can draw your graphics properly. Finally, we'll look more closely at color, which you'll use often in response to user interaction.

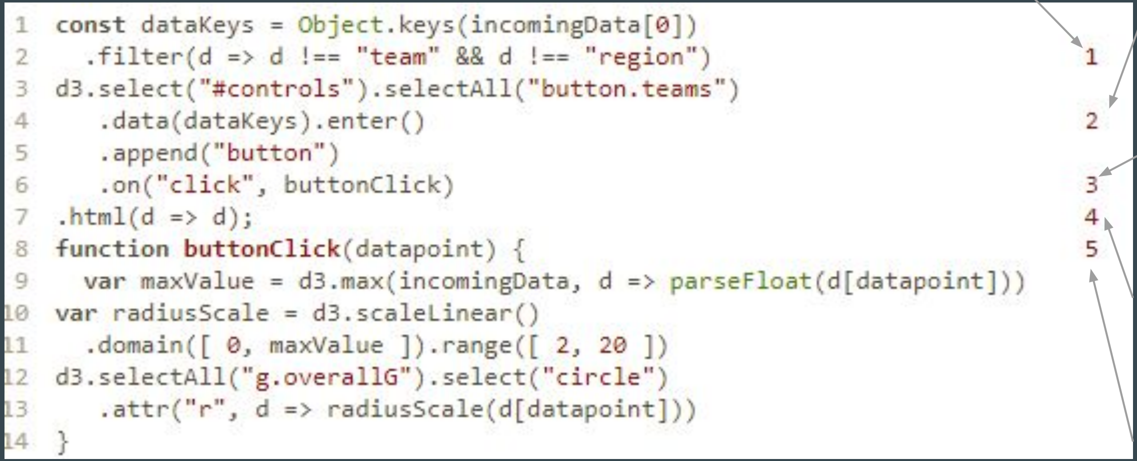
Events

- Add buttons that change the appearance of our graphics to correspond with different data.
- Static approach vs Dynamic
- Dynamic: added benefit of scaling to the data, so that if we add more attributes to our dataset, this function automatically creates the necessary buttons.

Events

Notice how we're using Object.keys on the first element in the data, so if you had elements with different kinds of keys, you would have to iterate through the whole array:

d,i



```
const dataKeys = Object.keys(incomingData[0])
  .filter(d => d !== "team" && d !== "region")
d3.select("#controls").selectAll("button.teams")
    .data(dataKeys).enter()
    .append("button")
    .on("click", buttonClick)
.html(d => d);
function buttonClick(datapoint) {
  var maxValue = d3.max(incomingData, d => parseFloat(d[datapoint]))
var radiusScale = d3.scaleLinear()
  .domain([ 0, maxValue ]).range([ 2, 20 ])
d3.selectAll("g.overallG").select("circle")
    .attr("r", d => radiusScale(d[datapoint]))
}
```

Builds buttons based on the data that's numerical—we want all attributes except the team and region attributes, which store strings

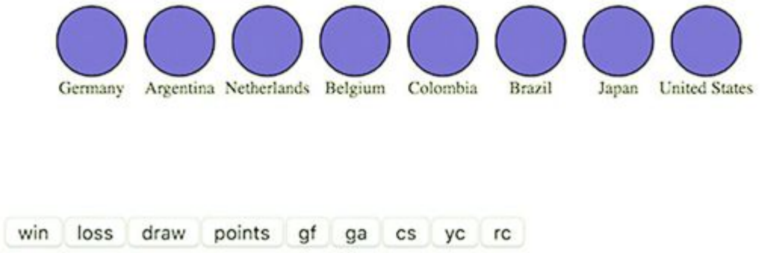
Registers an onclick behavior for each button, with a wrapper that gives access to the data that was bound to it when it was created

dataKeys consists of an array of attribute names, so the d corresponds to one of those names and makes a good button title

The function each button is calling on click, with the bound data sent automatically as the first argument

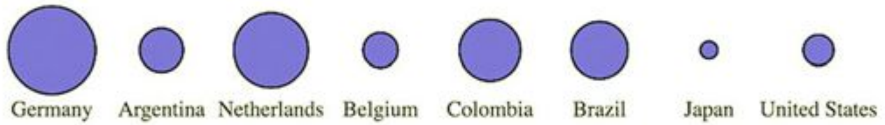
Filtered the array to remove attributes???

Figure 3.3. Buttons for each numerical attribute are appended to the `controls` div behind the `viz` div. When a button is clicked, the code runs `buttonClick`.

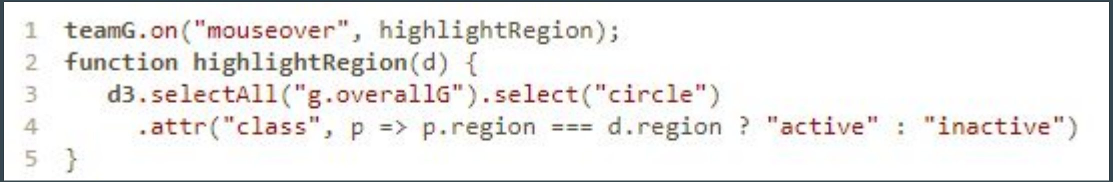


The .on function is a wrapper for the traditional HTML mouse events, and accepts "click", "mouseover", "mouseout", and so on. We can also access those same events using .attr, for example, using .attr("onclick", "console.log('click')"), but notice that we're passing a string in the same way we'd use traditional HTML. There's a D3-specific reason to use the .on function: it sends the bound data to the function automatically and in the same format as the anonymous inline functions we've been using to set style and attribute. We can create buttons based on the attributes of the data and dynamically measure the data based on the attribute bound to the button. Then we can resize the circles representing each team to reflect the teams with the highest and lowest values in each category

Figure 3.4. Our initial `buttonClick` function resizes the circles based on the numerical value of the associated attribute. The radius of each circle reflects the number of goals scored against each team, kept in the `ga` attribute of each datapoint.



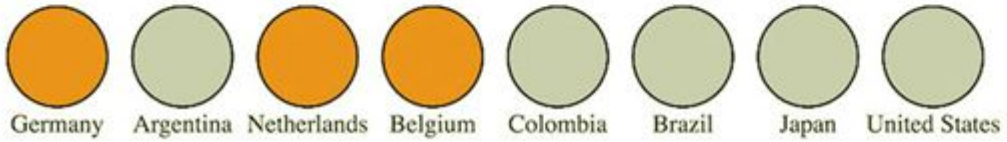
We can use .on() to tie events to any object, so let's add interactivity to the circles by having them indicate whether teams are in the same FIFA region:



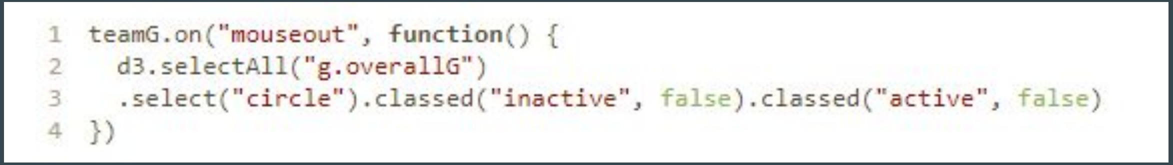
```
teamG.on("mouseover", highlightRegion);
function highlightRegion(d) {
   d3.selectAll("g.overallG").select("circle")
      .attr("class", p => p.region === d.region ? "active" : "inactive")
}
```

Here you see what some people call an ifsie, an inline if statement that compares the region of each element in the selection to the region of the element that you moused over, and if true returns "active" and if false returns "inactive" as the class of the circle

Figure 3.5. The effect of our initial `highlightRegion` selects elements with the same region attribute and colors them orange, while coloring gray those that aren't in the same region.



Restoring the circles to their initial color on mouseout is simple enough that the function can be declared inline with the .on function using a selection's built-in classed method, which allows you to selectively turn on or off classes of an element by setting it to true or false:

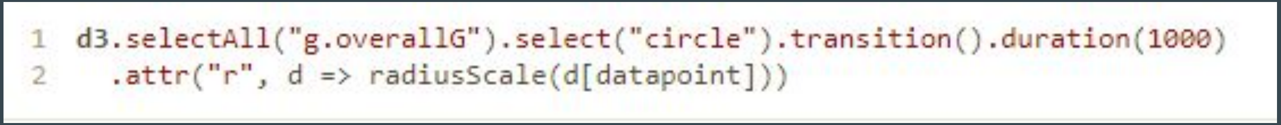


```
teamG.on("mouseout", function() {
  d3.selectAll("g.overallG")
  .select("circle").classed("inactive", false).classed("active", false)
})
```

Graphical transitions

One of the challenges of highly interactive, graphics-rich web pages is to ensure that the experience of graphical change isn't jarring. The instantaneous change in size or color that we've implemented doesn't only look clumsy, it can prevent a reader from understanding the information we're trying to relay.

Transitions are defined for a selection and can be set to occur after a certain delay using delay() or to occur over a set period of time using duration(). We can easily implement a transition in our buttonClick function:



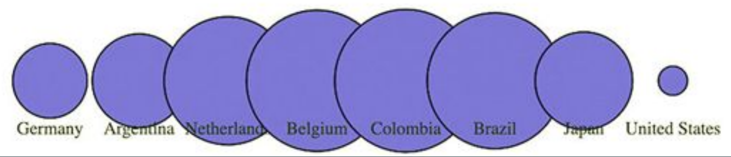
```
d3.selectAll("g.overallG").select("circle").transition().duration(1000)
  .attr("r", d => radiusScale(d[datapoint]))
```

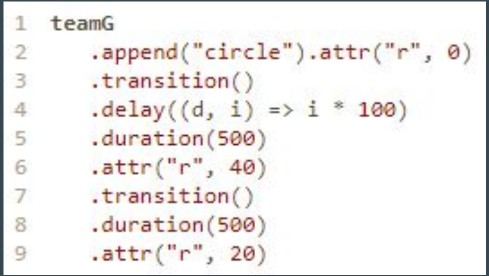
Now when we click our buttons, the sizes of the circles change, and the change is also animated. This isn't only for show. We're encoding new data in the size of the circle, indicating the change between two datapoints using animation. When there was no animation, the reader had to remember if there was a difference between the ranking in draws and wins for Germany. Now the reader has an animated indication that shows Germany visibly shrink or grow to indicate the difference between these two datapoints.

The use of transitions also allows us to delay the change through the .delay() function. Like the .duration() function, .delay() is set with the wait in milliseconds before implementing the change. Slight delays in the firing of an event from an interaction can be useful to improve the legibility of information visualization, allowing users a moment to reorient themselves to shift from interaction to reading. But long delays will usually be misinterpreted as poor web performance.

Why else would you delay the firing of an animation? Delays can also draw attention to visual elements when they first appear. By making the elements pulse when they arrive onscreen, you let users know that these are dynamic objects and tempt users to click or otherwise interact with them. Delays, like duration, can be dynamically set based on the bound data for each element. You can use delays with another feature: transition chaining. This sets multiple transitions one after another, and each is activated after the last transition has finished. If we amend the code in overallTeamViz() that first appends the <circle> elements to our <g> elements, we can see transitions of the kind that produce the screenshot in figure 3.6.

Figure 3.6. A screenshot of your data visualization in the middle of its initial drawing, showing the individual circles growing to an exaggerated size and then shrinking to their final size in the order in which they appear in the bound dataset.





```
teamG
    .append("circle").attr("r", 0)
    .transition()
    .delay((d, i) => i * 100)
    .duration(500)
    .attr("r", 40)
    .transition()
    .duration(500)
    .attr("r", 20)
```

This causes a pulse because it uses transition chaining to set one transition, followed by a second after the completion of the first. You start by drawing the circles with a radius of 0, so they're invisible. Each element has a delay set to its array position i times 0.1 seconds (100 ms), after which the transition causes the circle to grow to a radius of 40 px. After each circle grows to that size, a second transition shrinks the circles to 20 px. The effect, which isn't easy to present with a screenshot, causes the circles to pulse sequentially.

DOM manipulation

Getting access to the actual DOM element in the selection can be accomplished in one of two ways:

- Using this in the inline functions (cannot be used with arrow functions)
- Using the .node() function

Inline functions always have access to the DOM element along with the datapoint and array position of that datapoint in the bound data. The DOM element, in this case, is represented by the this context within the scope of the function. That context isn't available in arrow functions. We can see it in action using the .each() function of a selection, which performs the same code for each element in a selection. We'll make a selection of one of our circles and then use .each() to send d, i, and this to the console to see what each corresponds to

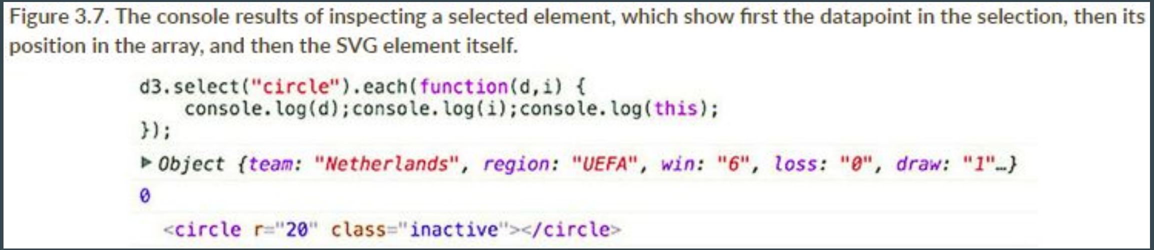
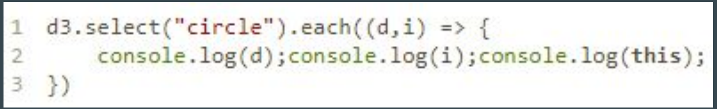


Figure 3.7. The console results of inspecting a selected element, which show first the datapoint in the selection, then its position in the array, and then the SVG element itself.

```
d3.select("circle").each(function(d,i) {
    console.log(d);console.log(i);console.log(this);
});
▸Object {team: "Netherlands", region: "UEFA", win: "6", loss: "0", draw: "1"…}
0
  <circle r="20" class="inactive"></circle>
```



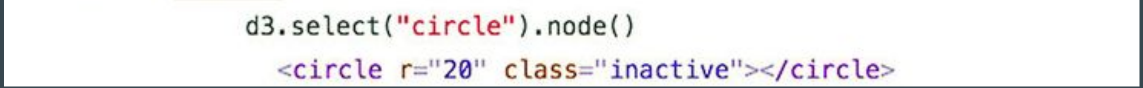
```
d3.select("circle").each((d,i) => {
    console.log(d);console.log(i);console.log(this);
})
```

Unpacking this a bit, we can see the first thing echoed, d, is the data bound to the circle, which is a JSON object representing the Netherlands team. The second thing echoed, i, is the array index position of that object in the selection, which in this case is 0 and means that incomingData[0] is the Netherlands JSON object. The last thing echoed to the console, this, is the <circle> DOM element itself. We can also access this DOM element using the .node() function of a selection:

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

Getting to the DOM element, as shown in figure 3.8, lets you take advantage of built-in JavaScript functionality to do things like measure the length of a <path> element or clone an element. One of the most useful built-in functions of nodes when working with SVG is the ability to re-append a child element. Remember that SVG has no Z-levels, which means that the drawing order of elements is determined by their DOM order. Drawing order is important because you don't want the graphical objects you interact with to look like they're behind the objects that you don't interact with. To see what this means, let's first adjust our highlighting function so that it increases the size of the label when we mouse over each element, as in figure 3.8.

Figure 3.8. The results of running the `node` function of a selection in the console, which is the DOM element itself—in this case, an SVG `<circle>` element.



```
d3.select("circle").node()
  <circle r="20" class="inactive"></circle>
```

```
teamG.on("mouseover", highlightRegion)
function highlightRegion(d,i) {
  d3.select(this).select("text").classed("active", true).attr("y", 10)
  d3.selectAll("g.overallG").select("circle").each(function (p) {
   p.region == d.region ?
     d3.select(this).classed("active",true) :
     d3.select(this).classed("inactive",true)
  })                                                              1
}
```

By turning on "active" class for the <g> that we hover over, we take advantage of the "g > text.active" rule in CSS that makes any text elements in that <g> increase their font size

Because we're doing a bit more, we should change the mouseout event to point to a function, which we'll call unHighlight:

```
teamG.on("mouseout", unHighlight)
function unHighlight() {
  d3.selectAll("g.overallG").select("circle").attr("class", "")
  d3.selectAll("g.overallG").select("text")
  .classed("active", false).attr("y", 30)
}
```

As shown in figure 3.9, Netherlands was appended to the DOM before Belgium. As a result, when we increase the size of the graphics associated with Netherlands, those graphics remain behind any graphics for Belgium, creating a visual artifact that looks unfinished and distracting. We can rectify this by re-appending the node to the parent <g> during that same highlighting event, which results in the label being displayed above the other elements, as shown in figure 3.10.

Figure 3.9. The `<text>` element "Netherlands" is drawn at the same DOM level as the parent `<g>`, which, in this case, is behind the element to its right.

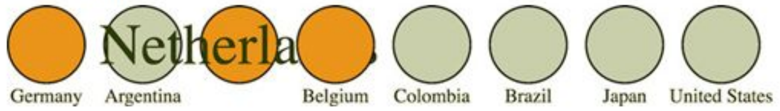
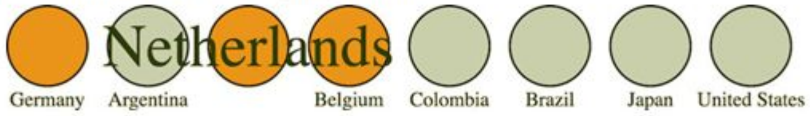
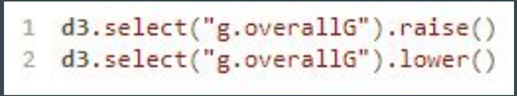


Figure 3.10. Re-appending the `<g>` element for Germany to the `<svg>` element moves it to the end of that DOM region and therefore it's drawn above the other `<g>` elements.



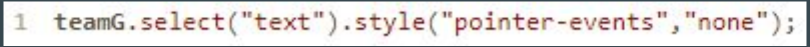
```
function highlightRegion (d) {
 d3.select(this).select("text").classed("active", true).attr("y", 10);
    d3.selectAll("g.overallG").select("circle")
       .each(function (p) {
         p.region == d.region ?
              d3.select(this).classed("active", true) :
              d3.select(this).classed("inactive", true);
       });
 this.parentElement.appendChild(this);
}
```

New in D3v4 are several helper functions to let you bump elements up and down in the DOM: selection.raise and selection.lower. These functions will move your selected element to the end of the list of its siblings in the DOM or move them to the beginning, respectively. This will have the effect of moving them forward or backward onscreen (above or below overlapping sibling elements):



```
d3.select("g.overallG").raise()
d3.select("g.overallG").lower()
```

You'll see in this example that the mouseout event becomes less intuitive because the event is attached to the <g> element, which includes not only the circle but the text as well. As a result, mousing over the circle or the text fires the event. When you increase the size of the text, and it overlaps a neighboring circle, it doesn't trigger a mouseout event. We'll get into event propagation later, but one thing we can do to easily disable mouse events on elements is set the style property "pointer-events" of those elements to "none" inline or in your CSS:



```
teamG.select("text").style("pointer-events","none");
```

That’ll do folks....

Using color wisely

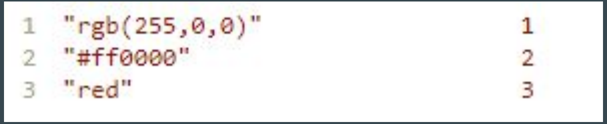
Color seems like a small and dull subject, but when you're representing data with graphics, color selection is of primary importance. There's good research on the use of color in cognitive science and design, but that's an entire library. Here, we'll deal with a few fundamental issues: mixing colors in color ramps, using discrete colors for categorical data, and designing for accessibility factors related to colorblindness. We're going to see a few strategies that will help you deal with deploying color wisely later.

INFOVIZ TERM: COLOR THEORY

Artists, scholars, and psychologists have thought critically about the use of color for centuries. Among them, Josef Albers—who has influenced modern information visualization leaders like Edward Tufte—noted that in the visual realm, one plus one can equal three. The study of color, referred to as color theory, has proved that placing certain colors and shapes next to each other has optical consequences, resulting in simultaneous and successive contrast as well as accidental color.

It's worth studying the properties of color—hue, value, intensity, and temperature—to ensure the most harmonious color relationships in your work. Leonardo da Vinci organized colors into psychological primaries, the colors the eye sees unmixed, but the modern exploration of color theory, as with many other phenomena in physics, can be attributed to Newton. Newton observed the separation of sunlight into bands of color via a prism in 1666 and called it a color spectrum. Newton also devised a color circle of seven hues, a precursor to the many future charts of color that would organize colors and their relationships. About a century later, J. C. Le Blon identified the primary colors as red, yellow, and blue, and their mixes as the secondaries. The work of other more modern color theoreticians such as Josef Albers, who emphasized the effects of color juxtaposition, influences the standards for presentation in print and on the web.

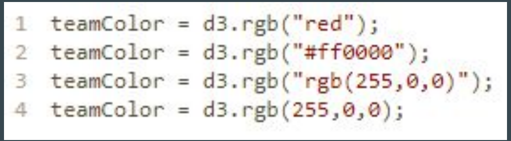
Color is typically represented on the web in red, green, and blue coordinates, or RGB, using one of three formats: hex, RGB, or CSS color name. The first two represent the same information, the level of red, green, and blue in the color, but do so with either hexadecimal or comma-delimited decimal notation. CSS color names use vernacular names for its 140 colors (you can read all about them here). Red, for instance, can be represented like this:



```
"rgb(255,0,0)"
"#ff0000"
"red"
```

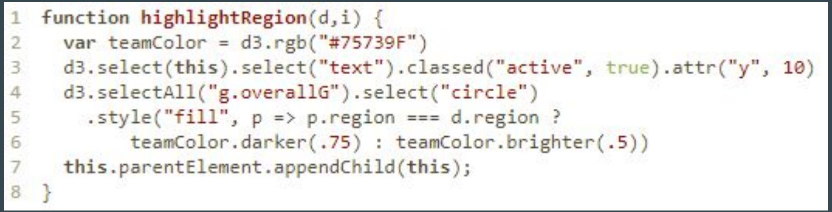
RGB, or red-green-blue, encoded color
Hex, or hexadecimal, formatted
CSS3 web color name

D3 has a few helper functions for working with colors. The first is d3.rgb(), which allows us to create a more feature-rich color object suitable for data visualization. To use d3.rgb(), we need to give it the red, green, and blue values of our color:



```
teamColor = d3.rgb("red");
teamColor = d3.rgb("#ff0000");
teamColor = d3.rgb("rgb(255,0,0)");
teamColor = d3.rgb(255,0,0);
```

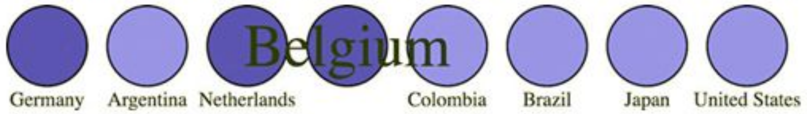
These color objects have two useful methods: .darker() and .brighter(). They do exactly what you'd expect: return a color that's darker or brighter than the color you started with. In our case, we can replace the gray and red that we've been using to highlight similar teams with darker and brighter versions of pink, the color we started with:



```
function highlightRegion(d,i) {
  var teamColor = d3.rgb("#75739F")
  d3.select(this).select("text").classed("active", true).attr("y", 10)
  d3.selectAll("g.overallG").select("circle")
    .style("fill", p => p.region === d.region ?
        teamColor.darker(.75) : teamColor.brighter(.5))
  this.parentElement.appendChild(this);
}
```

Notice that you can set the intensity for how much brighter or darker you want the color to be. Our new version (shown in figure 3.11) now maintains the palette during highlighting, with darker colors coming to the foreground and lighter colors receding. Unfortunately, you lose the ability to style with CSS because you're back to using inline styles. As a rule, you should use CSS whenever you can, but if you want access to things like dynamic colors and transparency using D3 functions, then you'll need to use inline styling.

Figure 3.11. Using the darker and brighter functions of a `d3.rgb` object in the highlighting function produces a darker version of the set color for teams from the same region and lighter colors for teams from different regions.



D3 allows you to represent colors in different color spaces, using d3.hsl, d3.lab, d3.cubehelix, d3.hcl and other non-core color libraries such as d3.hcg. in other ways with various benefits, but we'll only deal with HSL, which stands for hue, saturation, and lightness. The corresponding d3.hsl() allows you to create HSL color objects in the same way that you would with d3.rgb(). The reason you may want to use HSL is to avoid the muddying of colors that can happen when you build color ramps and mix colors using D3 functions that are going to use RGB by default.

Color mixing

We previously mapped a color ramp to numerical data to generate a spectrum of color representing our datapoints. The interpolated values for colors created by these ramps can be quite poor. As a result, a ramp that includes yellow can end up interpolating values that are muddy and hard to distinguish. You may think this isn’t important, but when you’re using a color ramp to indicate a value and your color ramp doesn’t interpolate the color in a way that your reader expects, then you can end up showing wrong information to your users.

Though we should avoid color ramps, we are forced to use them, whether because of expediency or requirement from users, and so you need to know how to deploy them with the least amount of damage to your visualization. You would be amazed at how quickly someone can lose confidence in your data visualization when the colors aren’t mapping the way they expect.

The way you encode color (RGB/Hex, HSL, HCL, LAB) doesn’t matter when it comes to a single color—you can get different codes for the same display color. It matters when you try to come up with the colors in between. When you’re doing that, different interpolation methods will result in different in-between colors, and if you need to use a ramp, you can be prepared to use the right interpolation method. Let’s add a color ramp to our button-Click function and use the color ramp to show the same information we did with the radius.

```
var ybRamp = d3.scaleLinear()
  .domain([0,maxValue]).range(["blue", "yellow"])

d3.selectAll("g.overallG").select("circle")
   .attr("r", d => radiusScale(d[datapoint]))
   .style("fill", d => ybRamp(d[datapoint]))
```

This is the same kind of color ramp we built previously using the maxValue we calculated for our circle radius scale

You'd be forgiven if you expected the colors in figure 3.12 to range from yellow to green to blue. The problem is that the default interpolator in the scale we used is mixing the red, green, and blue channels numerically. We can change the interpolator in the scale by designating one specifically, for instance, using the HSL representation of color (figure 3.13) that we looked at earlier.

Figure 3.12. Color mixing between yellow and blue in the RGB (red-green-blue) scale results in muddy, grayish colors displayed for the values between yellow and blue.

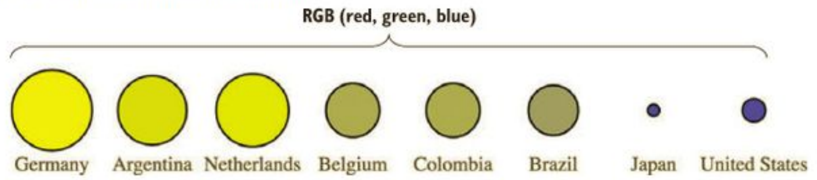
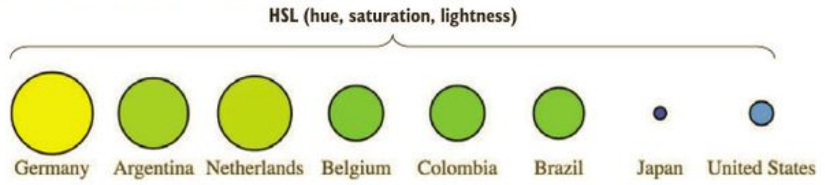


Figure 3.13. Interpolation of yellow to blue based on hue, saturation, and lightness (HSL) results in a different set of intermediary colors from the same two starting values.

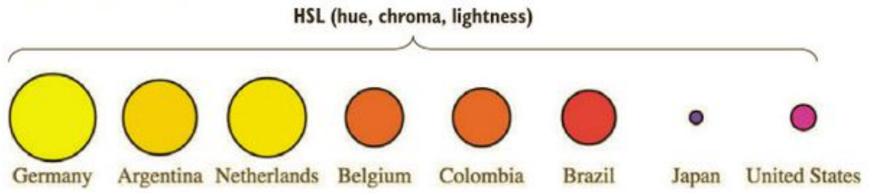


Setting the interpolation method for a scale is necessary when we don't want it to use its default behavior, such as when we want to create a color scale with a method other than interpolating the RGB values

```
var ybRamp = d3.scaleLinear()
  .interpolate(d3.interpolateHsl)
  .domain([0,maxValue]).range(["yellow", "blue"]);
```

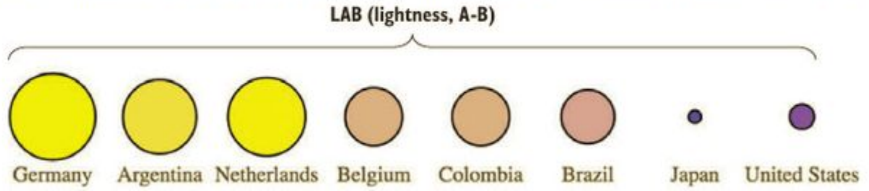
D3 supports two other color interpolators, HCL (figure 3.14) and LAB (figure 3.15), which each deal in a different manner with the question of what colors are between blue and yellow. First, the HCL ramp and then the LAB ramp:

Figure 3.14. Interpolation of color based on hue, chroma, and luminosity (HCL) provides a different set of intermediary colors between yellow and blue.



```
var ybRamp = d3.scaleLinear()
  .interpolate(d3.interpolateHcl)
  .domain([0,maxValue]).range(["yellow", "blue"]);
```

Figure 3.15. Interpolation of color based on lightness and color-opponent space (known as LAB; L stands for lightness and A-B stands for the color-opponent space) provides yet another set of intermediary colors between yellow and blue.

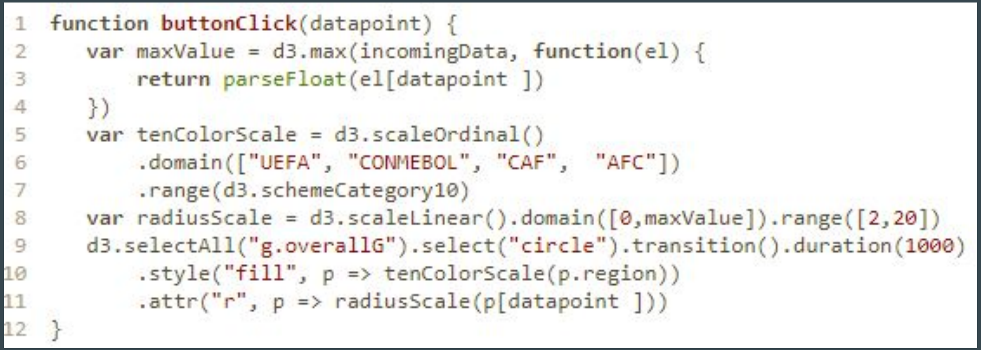


```
var ybRamp = d3.scaleLinear()
  .interpolate(d3.interpolateLab)
  .domain([0,maxValue]).range(["yellow", "blue"]);
```

As a general rule, you'll find that the colors interpolated in RGB tend toward muddy and gray, unless you break the color ramp into multiple stops. You can experiment with different color ramps or stick to ramps that emphasize hue or saturation (by using HSL). Or you can rely on experts by using the built-in D3 functions for color ramps that are proven to be easier for a reader to distinguish, which we'll look at now.

Discrete colors

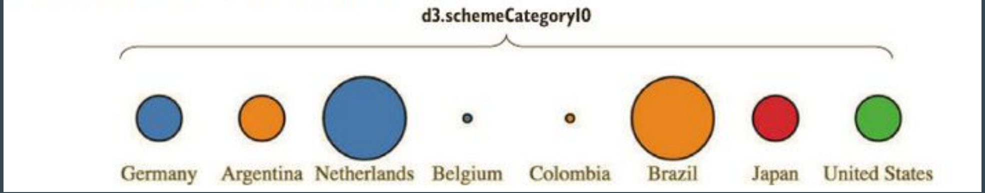
Oftentimes, we use color ramps to try to map colors to categorical elements. It's better to use the discrete color scales available in D3 for this purpose. The popularity of these scales is the reason why so many D3 examples have the same palette. D3 includes four collections of discrete color categories: d3.schemeCategory10, d3.schemeCategory20, d3.schemeCategory20b, and d3.schemeCategory20c. These are arrays of colors meant to be passed to d3.scaleOrdinal, which can be used to map categorical values to particular colors. In our case, we want to distinguish the various regions in our dataset, which consists of the top eight FIFA teams from the 2010 World Cup, representing four global regions. We want to represent these as different colors, and to do so, we need to create a scale with those values in an array:



```
function buttonClick(datapoint) {
   var maxValue = d3.max(incomingData, function(el) {
       return parseFloat(el[datapoint ])
   })
   var tenColorScale = d3.scaleOrdinal()
       .domain(["UEFA", "CONMEBOL", "CAF",  "AFC"])
       .range(d3.schemeCategory10)
   var radiusScale = d3.scaleLinear().domain([0,maxValue]).range([2,20])
   d3.selectAll("g.overallG").select("circle").transition().duration(1000)
       .style("fill", p => tenColorScale(p.region))
       .attr("r", p => radiusScale(p[datapoint ]))
}
```

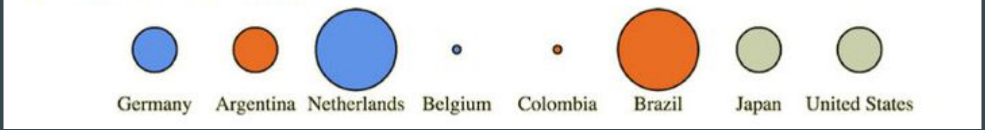
The application of this scale is visible when we click one of our buttons, which now resizes the circles as it always has, but also applies one of these distinct colors to each team (figure 3.16).

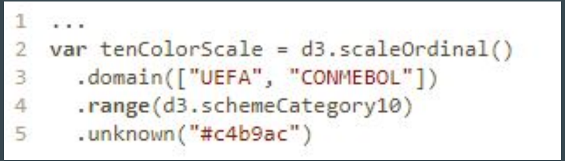
Figure 3.16. Application of the `schemeCategory10` to an ordinal scale in D3 assigns distinct colors to each class applied, in this case, the four regions in your dataset.



A useful feature of scaleOrdinal for situations like these is its .unknown method, which allows you to return a value when passed a value that doesn't exist within the domain, so that you can color using an "unknown" color like we see in figure 3.17 with gray. You don't always need to use gray as your unknown color.

Figure 3.17. Utilizing the `.unknown()` method of an ordinal scale to serve back values for data that doesn't have a corresponding entry in the scale's domain

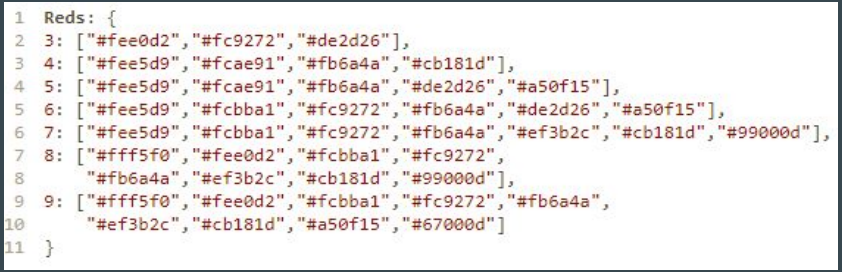




```
...
var tenColorScale = d3.scaleOrdinal()
  .domain(["UEFA", "CONMEBOL"])
  .range(d3.schemeCategory10)
  .unknown("#c4b9ac")
```

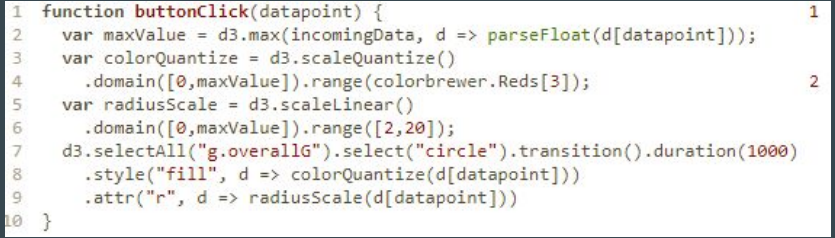
Color ramps for numerical data

Another option is to use color schemes based on the work of Cynthia Brewer, who has led the way in defining effective color use in cartography. Helpfully, d3js.org provides colorbrewer.js and colorbrewer.css for this purpose. Each array in colorbrewer.js corresponds to one of Brewer's color schemes, designed for a set number of colors. For instance, the reds scale looks like this:



```
Reds: {
3: ["#fee0d2","#fc9272","#de2d26"],
4: ["#fee5d9","#fcae91","#fb6a4a","#cb181d"],
5: ["#fee5d9","#fcae91","#fb6a4a","#de2d26","#a50f15"],
6: ["#fee5d9","#fcbba1","#fc9272","#fb6a4a","#de2d26","#a50f15"],
7: ["#fee5d9","#fcbba1","#fc9272","#fb6a4a","#ef3b2c","#cb181d","#99000d"],
8: ["#fff5f0","#fee0d2","#fcbba1","#fc9272",
    "#fb6a4a","#ef3b2c","#cb181d","#99000d"],
9: ["#fff5f0","#fee0d2","#fcbba1","#fc9272","#fb6a4a",
    "#ef3b2c","#cb181d","#a50f15","#67000d"]
}
```

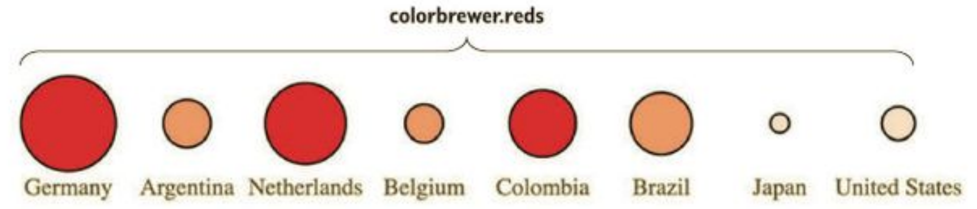
This provides high-legibility, discrete colors in the red spectrum for our elements. Again, we'll color your circles by region, but this time, we'll color them by their magnitude using our buttonClick function. We need to use the quantize scale that you saw earlier in chapter 2, because the colorbrewer scales, despite being discrete scales, are designed for quantitative data that has been separated into categories. In other words, they're built for numerical data, but numerical data that has been sorted into ranges, such as when you break down all the ages of adults in a census into categories of 18–35, 36–50, 51–65, and 65+:



```
function buttonClick(datapoint) {                                         1
  var maxValue = d3.max(incomingData, d => parseFloat(d[datapoint]));
  var colorQuantize = d3.scaleQuantize()
    .domain([0,maxValue]).range(colorbrewer.Reds[3]);                     2
  var radiusScale = d3.scaleLinear()
    .domain([0,maxValue]).range([2,20]);
  d3.selectAll("g.overallG").select("circle").transition().duration(1000)
    .style("fill", d => colorQuantize(d[datapoint]))
    .attr("r", d => radiusScale(d[datapoint]))
}
```

One of the conveniences of using colorbrewer.js dynamically paired to a quantizing scale is that if we adjust the number of colors—for instance, from colorbrewer.Reds[3] (shown in figure 3.18) to colorbrewer.Reds[5]—the range of numerical data is represented with five colors instead of three.

Figure 3.18. Automatic quantizing linked with the ColorBrewer 3-red scale produces distinct visual categories in the red family.

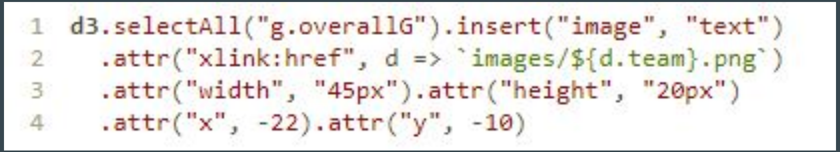


Color is important, and it can behave strangely on the web. Colorblindness, for instance, is a key accessibility issue that most of the colorbrewer scales address. But even though color use and deployment is complex, smart people have been thinking about color for a while, and D3 takes advantage of that. I've given you several of the tools you need to be successful with color, but the most important key to success with color is to not simply ignore it or pretend it to be something that's either too hard or already solved.

Pregenerated content

It’s neither fun nor smart to create all your HTML elements using D3 syntax with nested selections and appending. More importantly, there’s an entire ecosystem of tools out there for creating HTML, SVG, and static images that you’d be foolish to ignore because you’re using D3 for your general DOM manipulation and information visualization. Fortunately, it’s straightforward and easy to load externally generated resources—like images, HTML fragments, and pregenerated SVG—and tie them into your graphical elements.

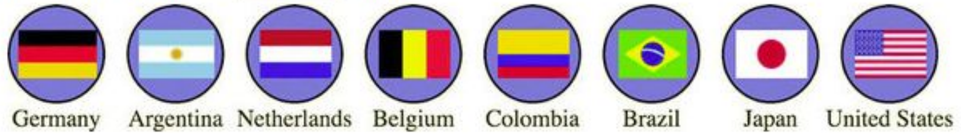
Images - You’ll find that adding images to your data visualizations can vastly improve them. In SVG, the image element is <image>, and its source is defined using the xlink:href attribute if it’s located in your directory structure. We have files in our images directory that are PNGs of the respective flags of each national team. To add them to our data visualization, select the <g> elements that have the team data already bound to them and add an SVG image:



```
d3.selectAll("g.overallG").insert("image", "text")
  .attr("xlink:href", d => `images/${d.team}.png`)
  .attr("width", "45px").attr("height", "20px")
  .attr("x", -22).attr("y", -10)
```

To make the images show up successfully, use insert() instead of append() because that gives you the capacity to tell D3 to insert the images before the text elements. This keeps the labels from being drawn behind the newly added images. Because each image name is the same as the team name of each data point, we can use an inline function to point to that value, combined with strings for the directory and file extension. We also need to define the height and width of the images because SVG images, by default, have no setting for height and width and won't display until these are set. We also need to manually center SVG images—here the x and y attributes are set to a negative value of one-half the respective height and width, which centers the images in their respective circles, as shown in figure 3.19.

Figure 3.19. Our graphical representations of each team now include a small PNG national flag, downloaded from Wikipedia and loaded using an SVG `<image>` element.



You can tie image resizing to the button events, but raster images don't resize particularly well, and so you'll want to use them at fixed sizes.

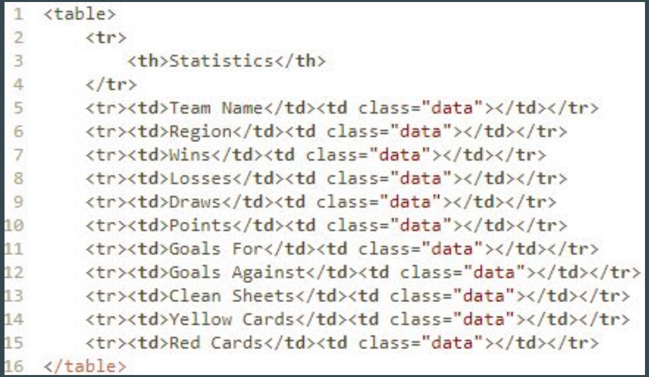
INFOVIZ TERM: CHARTJUNK

Now that you're learning how to add images and icons to everything, let's remember that because you can do something doesn't mean you should. When building information visualization, the key aesthetic principle is to avoid cluttering your charts and interfaces with distracting and useless "chartjunk," such as unnecessary icons, decoration, or skeuomorphic paneling. Remember, simplicity is force.

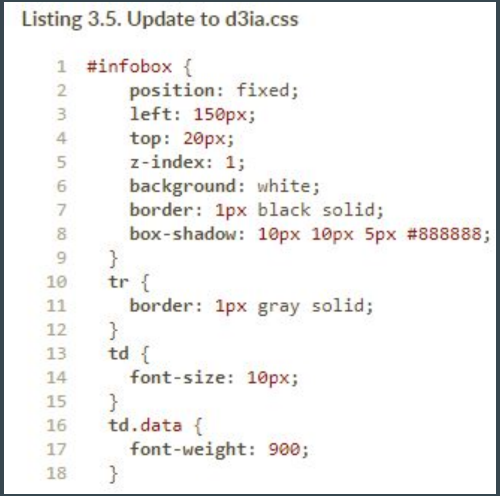
The term chartjunk comes from Tufte, and in general refers to the kind of generic and useless clip art that typifies PowerPoint presentations. Although icons and images are useful and powerful in many situations, and thus shouldn't be avoided only to maintain an austere appearance, you should always make sure that your graphical representations of data are as uncluttered as you can make them.

HTML fragments

We’ve created traditional DOM elements in this chapter using D3 data-binding for our buttons. If you want to, you can use the D3 pattern of selecting and appending to create complex HTML objects, such as forms and tables, on the fly. You’ll likely be working with designers and other developers who want to use those tools and require that those HTML components be included in your application. This isn’t a common practice, because most HTML generation is going to be handled by other templating libraries or frameworks, but you can use D3 to import and add them. For instance, let’s build a dialog box into which we can put the numbers associated with the teams. Say we want to see the stats on our teams—one of the best ways to do this is to have a dialog box that pops up as you click each team. We can write only the HTML we need for the table itself in a separate file, as shown in the following listing.



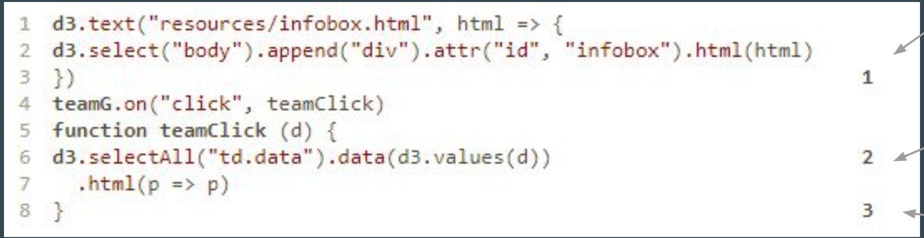
```
<table>
    <tr>
        <th>Statistics</th>
    </tr>
    <tr><td>Team Name</td><td class="data"></td></tr>
    <tr><td>Region</td><td class="data"></td></tr>
    <tr><td>Wins</td><td class="data"></td></tr>
    <tr><td>Losses</td><td class="data"></td></tr>
    <tr><td>Draws</td><td class="data"></td></tr>
    <tr><td>Points</td><td class="data"></td></tr>
    <tr><td>Goals For</td><td class="data"></td></tr>
    <tr><td>Goals Against</td><td class="data"></td></tr>
    <tr><td>Clean Sheets</td><td class="data"></td></tr>
    <tr><td>Yellow Cards</td><td class="data"></td></tr>
    <tr><td>Red Cards</td><td class="data"></td></tr>
</table>
```


Listing 3.5. Update to d3ia.css

```css
#infobox {
    position: fixed;
    left: 150px;
    top: 20px;
    z-index: 1;
    background: white;
    border: 1px black solid;
    box-shadow: 10px 10px 5px #888888;
  }
  tr {
    border: 1px gray solid;
  }
  td {
    font-size: 10px;
  }
  td.data {
    font-weight: 900;
  }
```

And now we'll add CSS rules for the table and the div that we want to put it in. As you see in the following listing, we can use the position and z-index CSS styles because this is a traditional DOM element.

Now that we have the table, all we need to do is add a click listener and associated function to populate this dialog, as well as a function to create a div with ID "infobox" into which we add the loaded HTML code using the .html() function. To do this we use d3.text to load the raw text of the file:



```javascript
d3.text("resources/infobox.html", html => {
d3.select("body").append("div").attr("id", "infobox").html(html)
})                                                                      1
teamG.on("click", teamClick)
function teamClick (d) {
d3.selectAll("td.data").data(d3.values(d))                              2
  .html(p => p)
}                                                                       3
```

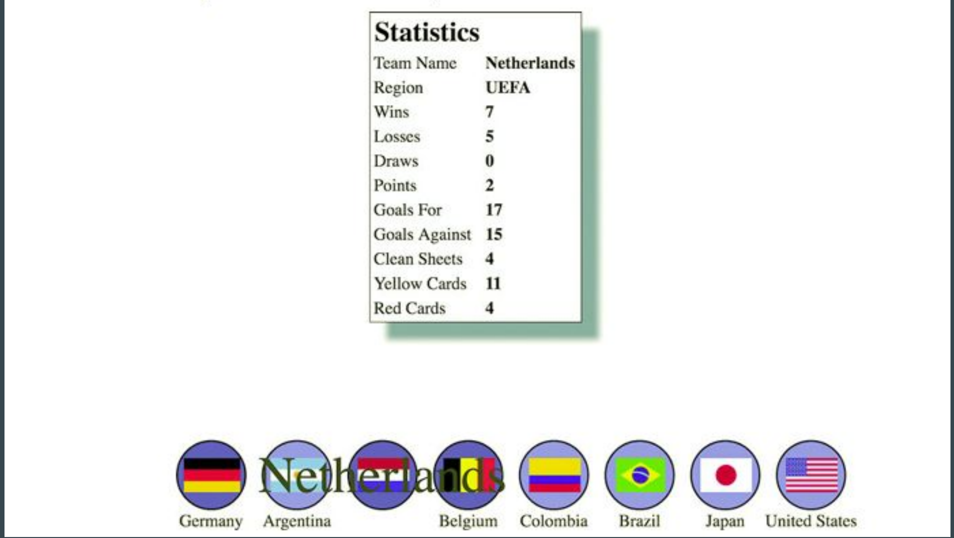
1 Creates a new div with an id corresponding to one in our CSS, and populates it with HTML content from infobox.html

2 You could also simply use Object.values if you're developing for browsers that support this functionality

3 Selects and updates the td.data elements with the values of the team clicked

The results are immediately apparent when you reload the page. A div with the defined table in infobox.html is created, and when you click it, it populates the div with values from the data bound to the element you click

Figure 3.20. The infobox is styled based on the defined style in CSS. It's created by loading the HTML data from infobox.html and adding it to the content of a newly created div.



Statistics

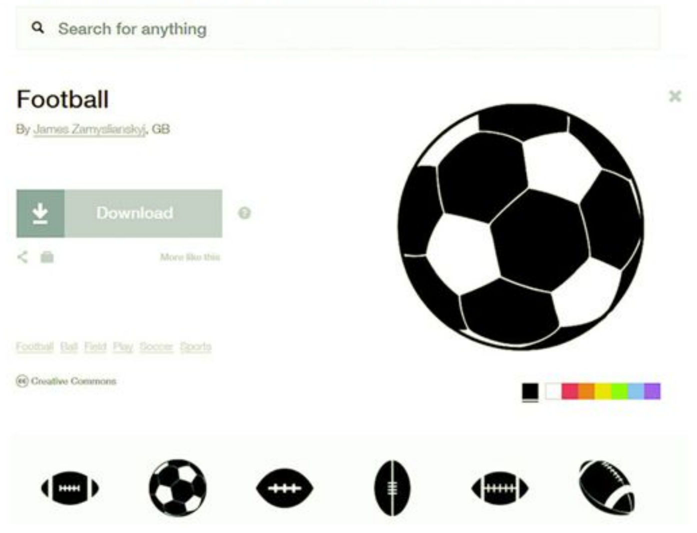
Team Name	**Netherlands**
Region	**UEFA**
Wins	**7**
Losses	**5**
Draws	**0**
Points	**2**
Goals For	**17**
Goals Against	**15**
Clean Sheets	**4**
Yellow Cards	**11**
Red Cards	**4**

We used d3.text() in this case because when working with HTML, it can be more convenoient to load the raw HTML code like this and drop it into the .html() function of a selected element that you've created. If you use d3.html(),you get HTML nodes that allow you to do more sophisticated manipulation, which you'll see now as we work with pregenerated SVG.

Pregenerated SVG

SVG has been around for a while, and there are, not surprisingly, robust tools for drawing SVG, such as Adobe Illustrator and the open source tool Inkscape. You might want pregenerated SVG for icons, interface elements, and other components of your work. If you're interested in icons, The Noun Project (http://thenounproject.com) has an extensive repository of SVG icons, including the football (soccer ball) in figure 3.21.

Figure 3.21. An icon for a soccer ball created by James Zamyslianskyj and available at http://thenounproject.com/term/football/1907/ from The Noun Project

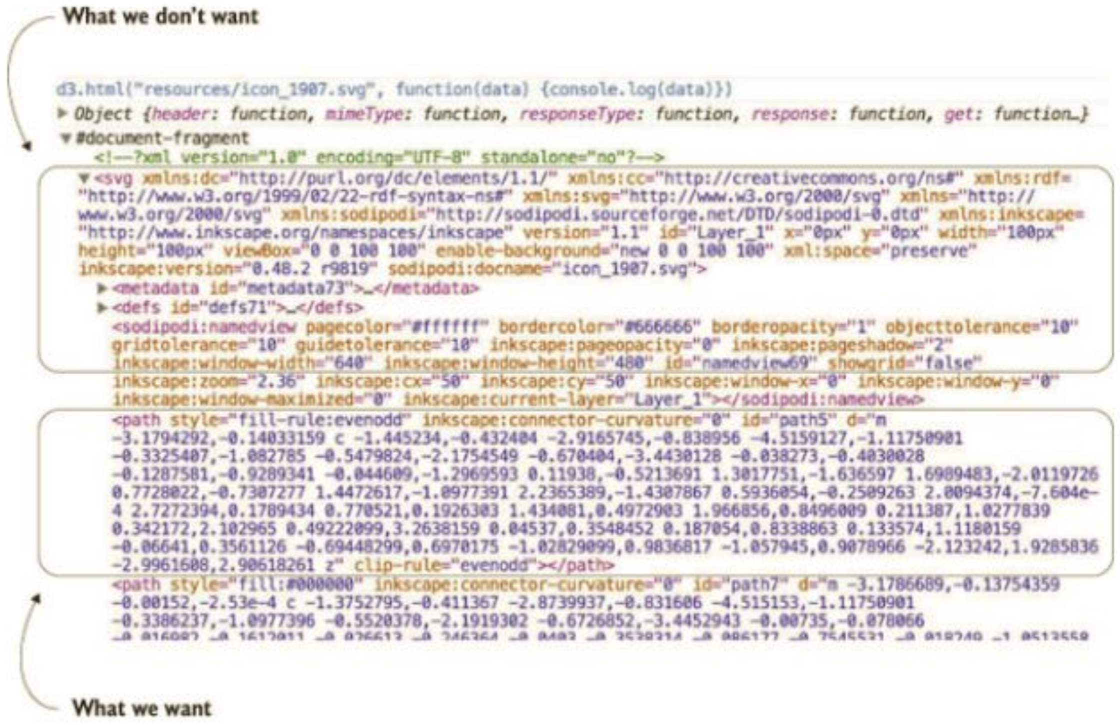


When you download an icon from The Noun Project, you get it in two forms: SVG and PNG. You've already learned how to reference images, and you can do the same with SVG by pointing the xlink:href attribute of an <image> element at an SVG file. But loading SVG directly into the DOM gives you the capacity to manipulate it like any SVG elements that you create in the browser with D3.

Let’s say we decide to replace our boring circles with balls, and we don’t want them to be static images because we want the ability to modify their color and shape like other SVG. In that case, we’ll need to find a suitable ball icon and download it. For downloads from The Noun Project, this means we’ll need to go through the hassle of creating an account, and we’ll need to properly attribute the creator of the icon or pay a fee to use the icon without attribution (not a hassle, but rather The Right Thing To Do™). Regardless of where we get our icon, we might need to modify it before using it in our data visualization. In the case of the soccer ball icon in this example, we need to make it smaller and center the icon on the 0,0 point of the canvas. This kind of preparation is going to be different for every icon, depending on how it was originally drawn and saved.

With the table in the dialog box we used earlier, we assumed that we pulled in all the code found in infobox.html, and so we could bring it in using d3.text() and drop the raw HTML as text into the .html() function of a selection. But in the case of SVG, especially SVG that you’ve downloaded, you often want to ignore the verbose settings in the document, which will include its own <svg> canvas as well as any <g> elements that have been not-so-helpfully added. You probably want to deal only with the graphical elements. With our soccer ball, we want to get only the <path> elements. If we load the file using d3.html(),the results are DOM nodes loaded into a document fragment that we can access and move around using D3 selection syntax. Using d3.html() is the same as using any of the other loading functions, where you designate the file to be loaded and the callback. You can see the results of this command in figure 3.22.

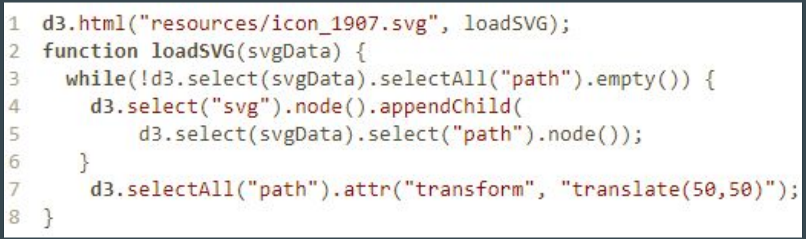
Figure 3.22. An SVG loaded using `d3.html()` that was created in Inkscape. It consists not only of the graphical `<path>` elements that make up the SVG but also much data that's often extraneous.



```
d3.html("resources/icon_1907.svg", function(data) {console.log(data)})
▶ Object {header: function, mimeType: function, responseType: function, response: function, get: function…}
▼ #document-fragment
    <!--?xml version="1.0" encoding="UTF-8" standalone="no"?-->
  ▼ <svg xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:cc="http://creativecommons.org/ns#" xmlns:rdf=
  "http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:svg="http://www.w3.org/2000/svg" xmlns="http://
  www.w3.org/2000/svg" xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd" xmlns:inkscape=
  "http://www.inkscape.org/namespaces/inkscape" version="1.1" id="Layer_1" x="0px" y="0px" width="100px"
  height="100px" viewBox="0 0 100 100" enable-background="new 0 0 100 100" xml:space="preserve"
  inkscape:version="0.48.2 r9819" sodipodi:docname="icon_1907.svg">
    ▶ <metadata id="metadata73">…</metadata>
    ▶ <defs id="defs71">…</defs>
      <sodipodi:namedview pagecolor="#ffffff" bordercolor="#666666" borderopacity="1" objecttolerance="10"
      gridtolerance="10" guidetolerance="10" inkscape:pageopacity="0" inkscape:pageshadow="2"
      inkscape:window-width="640" inkscape:window-height="480" id="namedview69" showgrid="false"
      inkscape:zoom="2.36" inkscape:cx="50" inkscape:cy="50" inkscape:window-x="0" inkscape:window-y="0"
      inkscape:window-maximized="0" inkscape:current-layer="Layer_1"></sodipodi:namedview>
      <path style="fill-rule:evenodd" inkscape:connector-curvature="0" id="path5" d="m
      -3.1794292,-0.14033159 c -1.445234,-0.432404 -2.9165745,-0.838956 -4.5159127,-1.11750901
      -0.3325407,-1.082785 -0.5479824,-2.1754549 -0.670404,-3.4430128 -0.038273,-0.4030028
      -0.1287581,-0.9289341 -0.044609,-1.2969593 0.11938,-0.5213691 1.3017751,-1.636597 1.6989483,-2.0119726
      0.7728022,-0.7307277 1.4472617,-1.0977391 2.2365389,-1.4307867 0.5936054,-0.2509263 2.0094374,-7.604e-
      4 2.7272394,0.1789434 0.770521,0.1926303 1.434081,0.4972903 1.966856,0.8496009 0.211387,1.0277839
      0.342172,2.102965 0.49222099,3.2638159 0.04537,0.3548452 0.187054,0.8338863 0.133574,1.1180159
      -0.06641,0.3561126 -0.69448299,0.6970175 -1.02829099,0.9836817 -1.057945,0.9078966 -2.123242,1.9285836
      -2.9961608,2.90618261 z" clip-rule="evenodd"></path>
      <path style="fill:#000000" inkscape:connector-curvature="0" id="path7" d="m -3.1786689,-0.13754359
      -0.00152,-2.53e-4 c -1.3752795,-0.411367 -2.8739937,-0.831606 -4.515153,-1.11750901
      -0.3386237,-1.0977396 -0.5520378,-2.1919302 -0.6726852,-3.4452943 -0.00735,-0.078066
      -0.016087,-0.1612011 -0.026613,-0.246364 -0.0483,-0.3530314 -0.006177,-0.7545531 -0.018249,-1.0513558
```

```
d3.html("resources/icon_1907.svg", data => {console.log(data)});
```

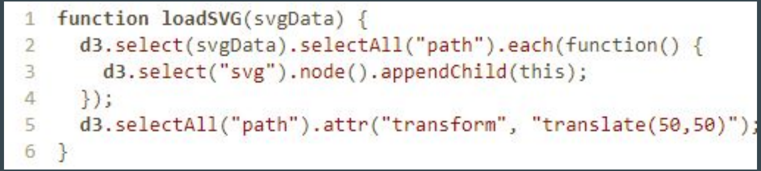
After we load the SVG into the fragment, we can loop through the fragment to get all the paths easily using the .empty() function of a selection. The .empty() function checks to see if a selection still has any elements inside it and eventually fires true after we've moved the paths out of the fragment into our main SVG. By including .empty() in a while statement, we can move all the path elements out of the document fragment and load them directly onto the SVG canvas:



```
d3.html("resources/icon_1907.svg", loadSVG);
function loadSVG(svgData) {
  while(!d3.select(svgData).selectAll("path").empty()) {
    d3.select("svg").node().appendChild(
        d3.select(svgData).select("path").node());
   }
    d3.selectAll("path").attr("transform", "translate(50,50)");
}
```

Notice how we've added a transform attribute to offset the paths so that they won't be clipped in the top-right corner. Instead, you clearly see a soccer ball in the top corner of your <svg> canvas. Document fragments aren't a normal part of your DOM, so you don't have to worry about accidentally selecting the <svg> canvas in the document fragment, or any other elements.

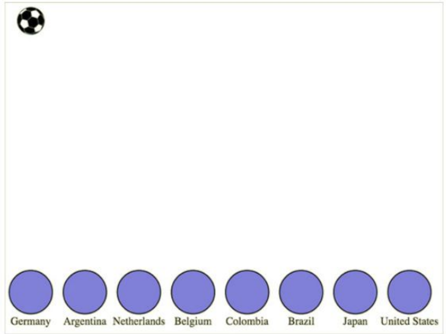
A while loop like this is sometimes necessary, but typically the best and most efficient method is to use .each() with your selection. Remember, .each() runs the same code on every element of a selection. In this case, we want to select our <svg> canvas and append the path to that canvas:



```
function loadSVG(svgData) {
  d3.select(svgData).selectAll("path").each(function() {
    d3.select("svg").node().appendChild(this);
  });
  d3.selectAll("path").attr("transform", "translate(50,50)");
}
```

We end up with a soccer ball floating in the top-left corner of our canvas, as shown in figure 3.23.

Figure 3.23. A hand-drawn soccer ball icon is loaded onto the `<svg>` canvas, along with the other SVG and HTML elements we created in our code.



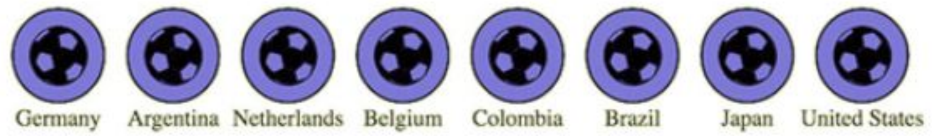
Note that we can't use arrow functions here because we need to have access to this context within the selection that corresponds to the DOM node

Loading elements from external data sources like this is useful if you want to move individual nodes out of your loaded document fragment, but if you want to bind the externally loaded SVG elements to data, it's an added step that you can skip. We can't set the .html() of a <g> element to the text of our incoming elements like we did with the <div> when we populated it with the contents of infobox.html. That's because SVG doesn't have a corresponding property to innerHTML, and therefore the .html() function on a selection of SVG elements has no effect. Instead, we have to clone the paths and append them to each <g> element representing our teams:

```
d3.html("resources/icon_1907.svg", loadSVG);
function loadSVG(svgData) {
  d3.selectAll("g").each(function() {
  var gParent = this;
  d3.select(svgData).selectAll("path").each(function() {
      gParent.appendChild(this.cloneNode(true))
  });
});
};
```

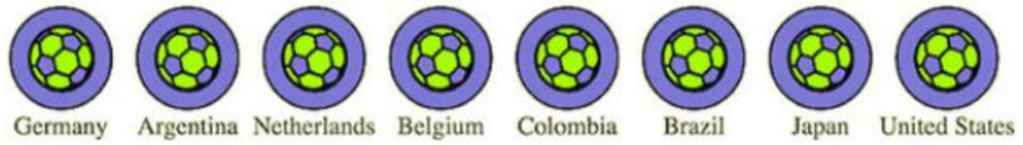
It may seem backwards to select each <g> and then select each loaded <path>, until you think about how .cloneNode() and .appendChild() work. We need to take each <g> element and go through the <path>-cloning process for every path in the loaded icon, which means we use nested .each() statements (one for each <g> element in our DOM and one for each <path> element in the icon). By setting gParent to the actual <g> node (the this variable), we can then append a cloned version of each path in order. The results are soccer balls for each team, as shown in figure 3.24.

Figure 3.24. Each <g> element has its own set of paths cloned as child nodes, resulting in soccer ball icons overlaid on each element.



We can easily do the same thing using the <image> syntax from the first example in this section, but with our SVG elements individually added to each. And now we can style them in the same way as any path element. We could use the national colors for each ball, but we'll settle for making them green, with the results shown in figure 3.25.

Figure 3.25. Football icons with a fill and stroke set by D3

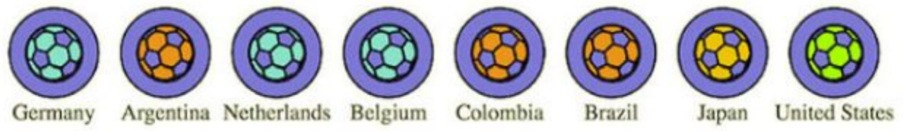


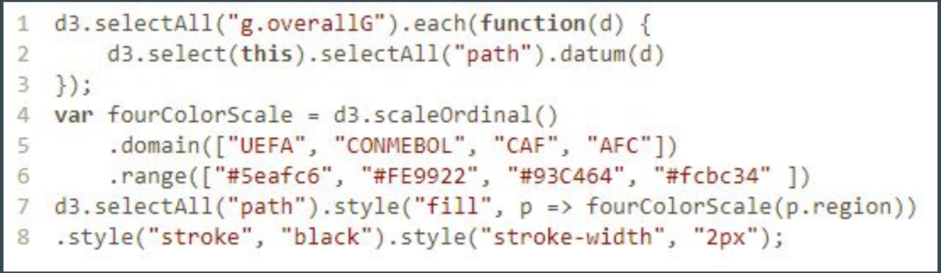
```
d3.selectAll("path").style("fill", "#93C464")
  .style("stroke", "black").style("stroke-width", "1px");
```

One drawback to this method is that the paths can't take advantage of the D3 .insert() method's ability to place the elements behind the labels or other visual elements. To get around this, we'll need to either append icons to <g> elements that have been placed in the proper order, or use the selection.lower() and selection.raise() functions to move the paths around the DOM, as described earlier in this chapter.

The other drawback is that because these paths were added using cloneNode and not selection#append syntax, they have no data bound to them. We looked at rebinding data back in chapter 1. If we select the <g> elements and then select the <path> element, this will rebind data. But we have numerous <path> elements under each <g> element, and selectAll doesn't rebind data. As a result, we have to take a more involved approach to bind the data from the parent <g> elements to the child <path> elements that have been loaded in this manner. The first thing we do is select all the <g> elements and then use .each() to select all the path elements under each <g>. Then, we separately bind the data from the <g> to each <path> using .datum(). What's .datum()? Well, datum is the singular of data, so a piece of data is a datum. The datum function is what you use when you're binding just one piece of data to an element. It's the equivalent of wrapping your variable in an array and binding it to .data(). After we perform this action, we can dust off our old scaleOrdinal with a new set of colors and apply it to our new <path> elements. We can run this code in the console to see the effects, which should look like figure 3.26.

Figure 3.26. The paths now have the data from their parent element bound to them and respond accordingly when a discrete color scale based on region is applied.





```
d3.selectAll("g.overallG").each(function(d) {
    d3.select(this).selectAll("path").datum(d)
});
var fourColorScale = d3.scaleOrdinal()
    .domain(["UEFA", "CONMEBOL", "CAF", "AFC"])
    .range(["#5eafc6", "#FE9922", "#93C464", "#fcbc34" ])
d3.selectAll("path").style("fill", p => fourColorScale(p.region))
.style("stroke", "black").style("stroke-width", "2px");
```

Now you have data-driven icons. Use them wisely.

References

All content is taken verbatim from Chapter 3 of D3.js in Action 2nd Edition