# 2-1, Lesson 2 - Data

## 2.1 Magical Data

D3 starts with data. We have already seen how the library can use a single data value—a datum—to draw and manipulate graphics. This is just the tip of the iceberg. We will now dive into the much more powerful capabilities of D3 to make use of *data*, plural—any number of data values strung together in a JavaScript array.

In addition to the textbook chapters cited above, a useful reference for this activity is the blog post **D3 Selection Mysteries...Solved? (https://northlandia.wordpress.com/2014/11/01/d3-selection-mysteries-solved/)** by Carl Sack (2014).

Beyond a single datum, D3 rigidly requires any set of data to be formatted as an array. *You cannot feed an object to a D3 selection*. But as long as it's formatted as an array, D3 can utilize virtually any *type* of data. Consider, for instance, the following arrays (Example 2.1):

**Example 2.1**: Various types of JavaScript arrays

JavaScript

```
 var numbersArray = [1, 2, 3];

var stringsArray = ["one", "two", "three"];

var colorsArray = ["#F00", "#0F0", "#00F"];

var objectsArray = [
    {
        city: 'Madison',
        population: 233209
    },
    {
        city: 'Milwaukee',
        population: 594833
    },
    {
        city: 'Green Bay',
        population: 104057
    }
];

var arraysArray = [
    ['Madison', 23209],
    ['Milwaukee', 593833],
    ['Green Bay', 104057]
];
```

D3 will work with any of the data variables above, *as long as the outer data structure is an array*. The array can contain anything—or a mix of things—but is hashable to D3 as long as it's an array.

> **Rule!** All data passed to the `.data()` operator must be formatted as an array.

Let's think about how this might relate to mapping for a moment (after all, this is a cartography course). In the Leaflet Lab, you used geographic data in GeoJSON format to place proportional symbols on your map. Of course, a GeoJSON starts with an outer object, not an array. But what's the main thing *inside* the GeoJSON object? A `"features"` array (Example 2.2 line 3). Just to refresh your memory of what this looks like:

**Example 2.2**: the start and end of *MegaCities.geojson*

JSON

```
{
    "type":"FeatureCollection",
    "features":[
        {
            "type":"Feature",
            "properties":{
                "City":"Tokyo",
                ...
    ]
}
```

We will find that D3 allows us to map GeoJSON geometries by making use of the `"features"` array. For now, let's return to the basics.

---

## 2.2 Joining Data

In Lesson 1, we looked at how to bind a single datum to a selection for styling a markup element. But what if you want to create a *set* of new elements and style them in a way that corresponds to a bunch of different data values? Say, for instance, we want to create a bubble chart, with several circles styled according to a dataset. The bit of magic D3 uses to do this is called a **join (http://bost.ocks.org/mike/join/)** because it joins an array of data to an array of markup elements in the DOM.

The first thing we need to create a join is a data array, such as those presented in Example 2.1. To our existing script from Lesson 1, let's add a simple array of numbers (Example 2.3):

**Example 2.3**: a data array in *main.js*

JavaScript

```
//below Example 1.9
 var dataArray = [10, 20, 30, 40, 50];
```

Now we need to start a new block that will create our circles. Since we are creating these circles after the inner rectangle, they will appear on top of the rectangle on the browser page. To create multiple, different circles at once, we can't use the `d3.select()` method we used for the rectangle, as this type of selection can only make use of a single datum. Instead, to begin the block, we use a "magical" sequence of three methods: `d3.selectAll()`, `.data()`, and `.enter()` (Example 2.4):

**Example 2.4**: the magic trio in *main.js*

JavaScript

```
//Example 2.3 line 1
var dataArray = [10, 20, 30, 40, 50];

var circles = container.selectAll(".circles") //but wait--there are no circles yet!
    .data(dataArray) //here we feed in an array
    .enter() //one of the great mysteries of the universe
```

Recall from Lesson 1 that `d3.selectAll()` selects all matching elements in the DOM. However, in the case above (Example 2.4 line 4), there are not yet any existing elements with the class name `"circles"` in the DOM. Feeding a parameter that doesn't return anything to `d3.selectAll()` creates an **empty selection**. The parameter `".circles"` is merely a placeholder; in fact, any string that doesn't match existing elements would do, although it is convention to use the class name of the *future* elements you will create. If you were to feed it a selector referencing already-existing elements, `d3.selectAll()` would create a selection out of all of the elements matching the selector. But since we want to create new elements based on our data array, we need to create an empty selection by using a selector it won't find a match for.

> **Rule!** Always pass the block's name as a class selector to the `.selectAll()` method when creating an empty selection.

The next step is to feed in our data array as the parameter of the `.data()` operator (Example 2.5 line 5). The third method, `.enter()` (line 6), is where the magic really happens. It takes no parameters. Its function is to join the data to the selection, creating an array of placeholders for one markup element per data value in the array. The rest of the block then functions like a loop through the data array: each operator will be applied once for each value in the data array. We will first make use of this to create a circle for every array value (Example 2.5):

**Example 2.5**: adding circles to match the data in *main.js*

JavaScript

```
//Example 2.4 line 1
var dataArray = [10, 20, 30, 40, 50];
```

```
    var circles = container.selectAll(".circles") //but wait--there are no circles yet!
        .data(dataArray) //here we feed in an array
        .enter() //one of the great mysteries of the universe
        .append("circle") //add a circle for each datum
        .attr("class", "circles") //apply a class name to all circles
```

If we now view the markup using the developer tools HTML tab, we can see our newly-created `<circle>` elements (Figure 2.1):

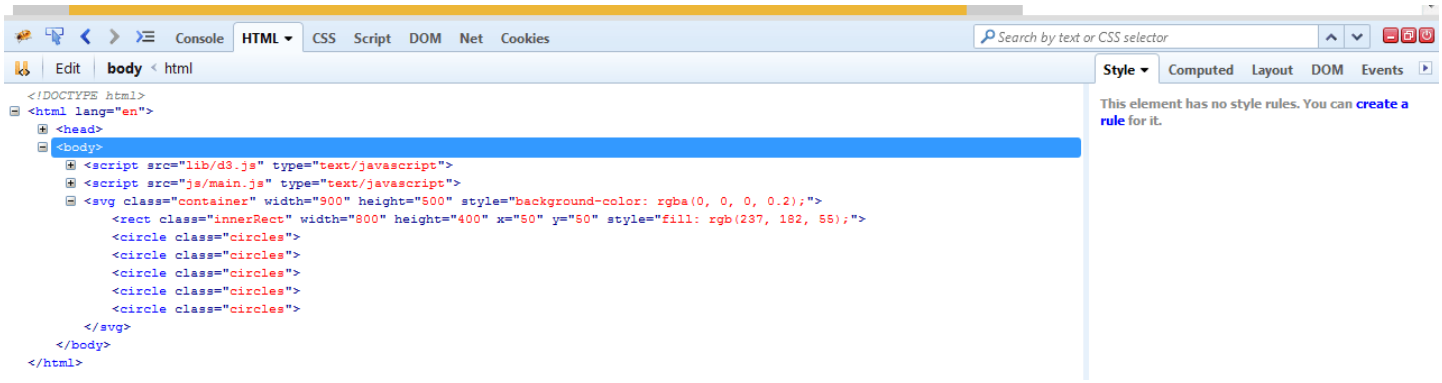


*Figure 2.1: the circles exist*

Notice there are five circles, one for each datum. Using `.append()` after a data join will always create the same number of new elements as data values in the dataset. The circles aren't yet visible on the page because, as you may recall from previous Module, they require `r`, `cx`, and `cy` attributes. We can use our joined data to position the circles (Example 2.6):

**Example 2.6**: using the joined data in *main.js*

JavaScript

```
    //Example 2.5 line 1
    var dataArray = [10, 20, 30, 40, 50];

    var circles = container.selectAll(".circles") //but wait--there are no circles yet!
        .data(dataArray) //here we feed in an array
        .enter() //one of the great mysteries of the universe
        .append("circle") //add a circle for each datum
        .attr("class", "circles") //apply a class name to all circles
        .attr("r", function(d, i){ //circle radius
            console.log("d:", d, "i:", i); //let's take a look at d and i
            return d;
        })
        .attr("cx", function(d, i){ //x coordinate
            return 70 + (i * 180);
        })
        .attr("cy", function(d){ //y coordinate
            return 450 - (d * 5);
        });
```

In a block with a data join, the anonymous functions that return a second parameter to operators (such as on lines 9, 13, and 16 of Example 2.6) can make use of each datum (`d` in the example) as well as the

index of the datum in the data array (`i`). The `console.log()` statement on line 10 prints each parameter to the console to make them easier to understand. Again, picture the entire block after `.enter()` as a loop—one in which you have immediate access to each array value and the array index within every anonymous function fed to an operator. In Example 2.6, we use each datum to set the radius and center y (vertical) coordinate of the circle, and the index to set the center x (horizontal) coordinate. Figure 2.2 shows the result along with the data and index values in the Firebug Console.

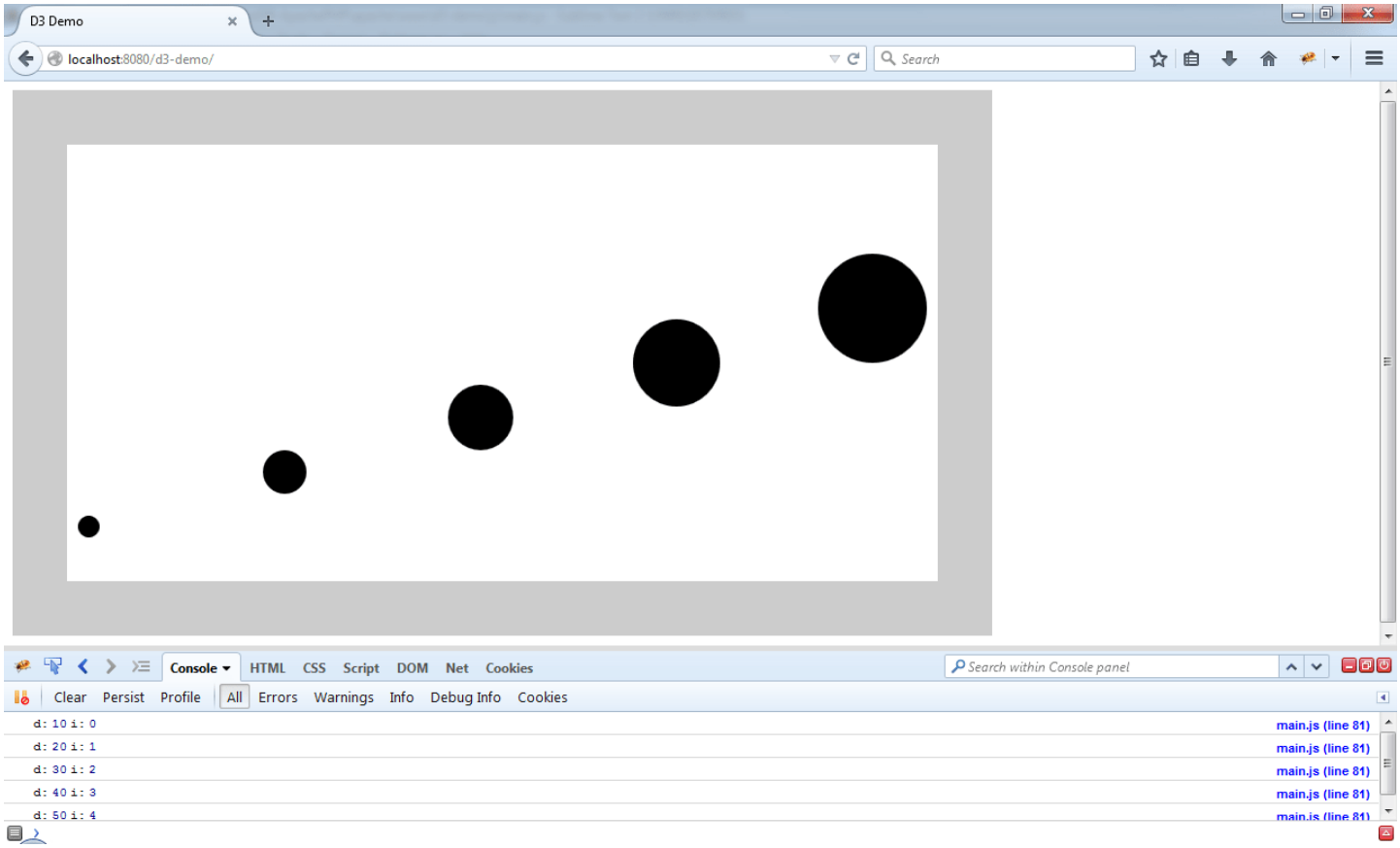


*Figure 2.2: data-driven circles*

## 2.3 Complex Data

So far, we have relied upon a very simple array to build our example graphic. What if we want something a little more complex and meaningful—say, a graph of city populations. We might start with our more complicated data array from way back in Pre-Module 1-2 (Example 2.7):

**Example 2.7**: city populations data array from Pre-Module 1-2

JavaScript

```
var cityPop = [
    {
        city: 'Madison',
        population: 233209
    },
```

```
    {
        city: 'Milwaukee',
        population: 594833
    },
    {
        city: 'Green Bay',
        population: 104057
    },
    {
        city: 'Superior',
        population: 27244
    }
];
```

We could make a bubble chart out of this data by combining it with our `circles` block. We need to make a few modifications to the block: derive the circle radii from the populations as areas and derive the center y coordinates from the populations times a scale factor (Example 2.8):

**Example 2.8**: using the `cityPop` array to create circles in *main.js*

JavaScript

```
    var cityPop = [
        {
            city: 'Madison',
            population: 233209
        },
        {
            city: 'Milwaukee',
            population: 594833
        },
        {
            city: 'Green Bay',
            population: 104057
        },
        {
            city: 'Superior',
            population: 27244
        }
    ];

    //Example 2.6 line 3
    var circles = container.selectAll(".circles") //create an empty selection
        .data(cityPop) //here we feed in an array
        .enter() //one of the great mysteries of the universe
        .append("circle") //inspect the HTML--holy crap, there's some circles there
        .attr("class", "circles")
        .attr("id", function(d){
            return d.city;
        })
        .attr("r", function(d){
            //calculate the radius based on population value as circle area
            var area = d.population * 0.01;
            return Math.sqrt(area/Math.PI);
        })
```

```
        .attr("cx", function(d, i){
            //use the index to place each circle horizontally
            return 90 + (i * 180);
        })
        .attr("cy", function(d){
            //subtract value from 450 to "grow" circles up from the bottom instead of down from the top of th
e SVG
            return 450 - (d.population * 0.0005);
        });
```

Note that in Example 2.8, `d` still holds each of our array values, only now each value is an object with two properties (`city` and `population`). Thus, on on line 27, we use `d.city` to assign each city name as the circle `id`, and on lines 31 and 40, we use `d.population` to access the `population` value of each object. Figure 2.3 shows our city circles on the page and in the HTML:

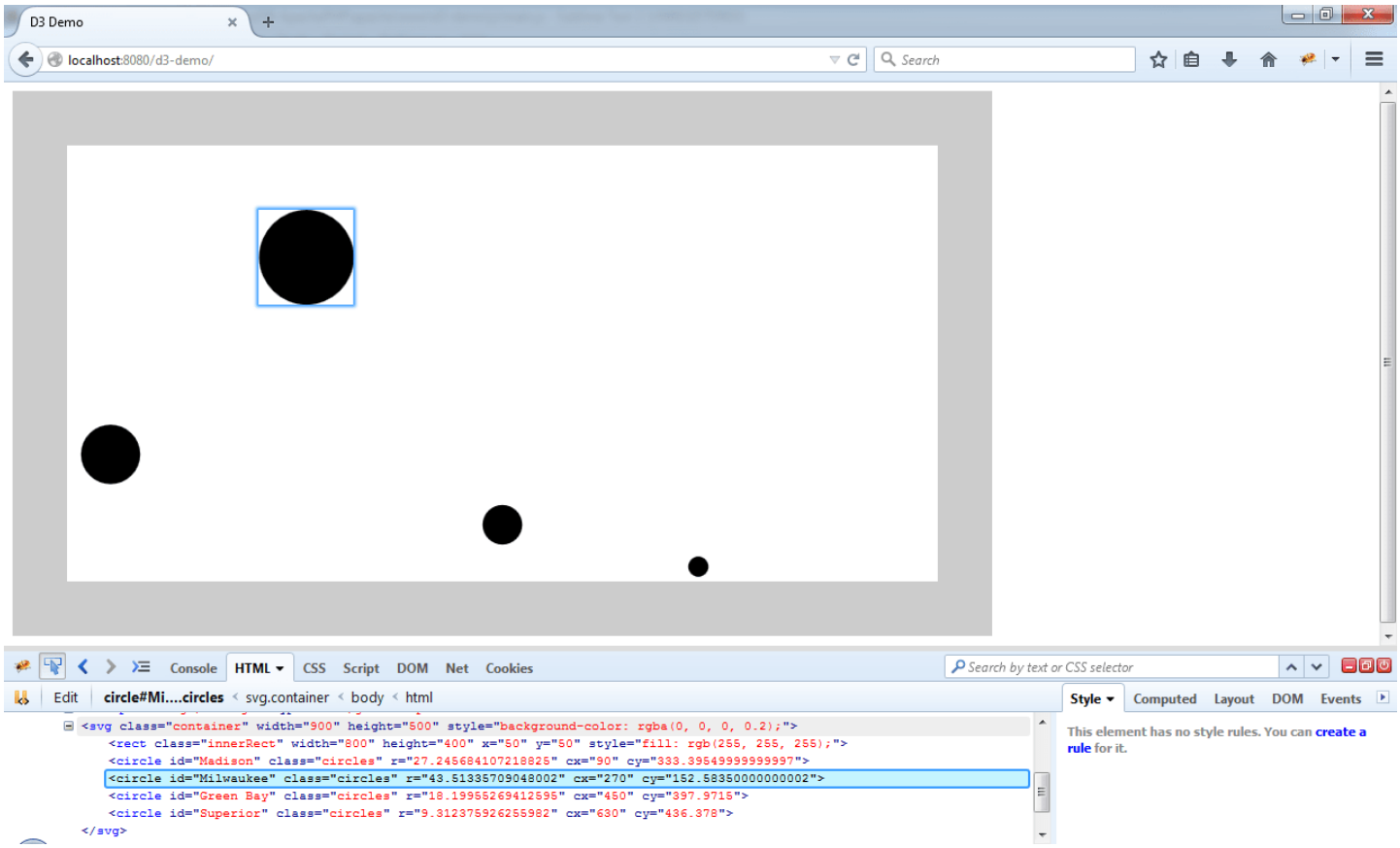


*Figure 2.3: A bubble chart of city populations*

---

> **Practice!** Find population data for at least four cities of your choice (*other than* those used in the example), and create an array containing an object for each city similar to the `cityPop` array in Example 2.7. Using properly formatted D3 code blocks, create a bubble chart with circles named according to city name and sized according to city population.

---

> **Self-Check:**

**1. What JavaScript data structure(s) can be used as the outer-level container of data passed to the `.data()` operator?**

**a**. Array
**b**. Object
**c**. String
**d**. Number
**e**. All of the above

**2. True/False:** For an empty selection with joined data, the number of new elements created by `.append()` always equals the number of values in the joined dataset.