Intro to D3.js: Making a Chart

What is D3?

D3 stands for Data Driven Documents. It is a JavaScript library developed by Mike Bostock that is designed to manipulate documents based on data, allowing the user to create rich and dynamic visualizations using web standards of HTML, SVG, CSS, and JavaScript. The library is well developed, and very popular among Data Analysts and Data Visualization Specialists.

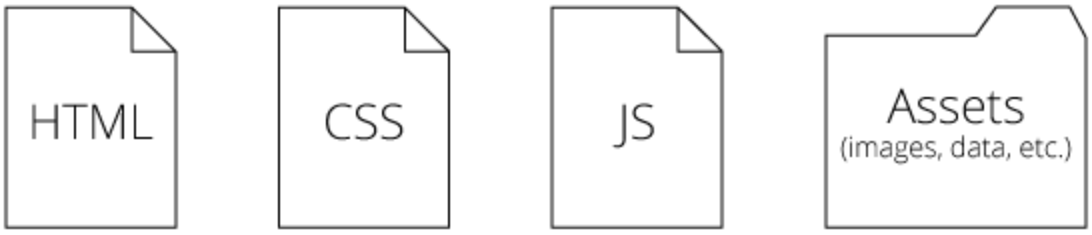
It is perhaps easier to describe D3 by detailing what it is not. You've probably come across standard plotting libraries; ggplot2 or shiny for R, Excel's built-in plotting functions, matplotlib in Python... these are libraries that make it easy to do *particular kinds of plotting*. They will have built-in functions to make producing certain kind of graphics very simple. Generating a complete bar chart might require only one line of code, or the click of one button.

This is not the case in D3. D3 is output-neutral; this means that every graphic requires quite a bit of explicit coding to give direction to our web browser. On the one hand, this means that producing even simple graphics will at first seem very laborious. On the other hand, because D3 is not prescriptive, you can quite literally do anything with it. Where dedicated plotting and mapping libraries break down quite quickly when you want to do something other than what they're intended to do, D3 will have no complaint when you try to push at the edges.

Think of it as a very fancy tool that you have to learn to operate, telling it what to draw, what data to base the drawing on, and which document to manipulate.

Web Page Documents and Assets

To describe how D3 works, first consider what we mean by the term **Documents**. A webpage is a collection of documents sitting on your web server. The web server has an address on the World Wide Web that other computers can navigate to. When a visitor navigates to your address, their browser will request one or more documents. These documents describe to their browser what to display, how to display it, where to get data from that aids in the display, and when to execute events such as user clicks. A basic webpage could contain a series of text documents that contain code written to structure the webpage (HTML), style that page (CSS), and add dynamic elements (JS). Webpages will often have many assets, including images, data, and other supplemental material that makes the page display and function properly.



How D3 Works

When working with D3, you are creating and manipulating elements within web documents. For example, the D3 JavaScript library will find a div element in your page, bind a dataset to it, then set attributes of that element according to values in the dataset. It might dynamically update the element based on, for example, a user click or a change in the dataset.

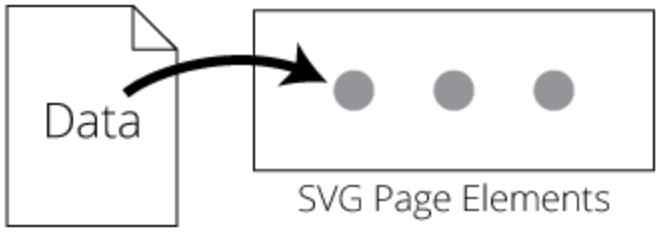
D3 is, indeed, data-driven; this means that changes on your web page are generated by data that you are expected to 'bind'. D3 can work with just about any element found on your webpage and modify it by binding data to that element and setting attributes of that element accordingly. The data drives the document. For example, D3 can be used to generate an HTML table from an array of numbers, or you can use the same data to create an interactive bar chart with those same numbers. Events you connect to these elements allow for interaction with the data[1].

Just what does this mean? Take a look at a sample dataset. D3 allows these data to be attached to elements in our document.

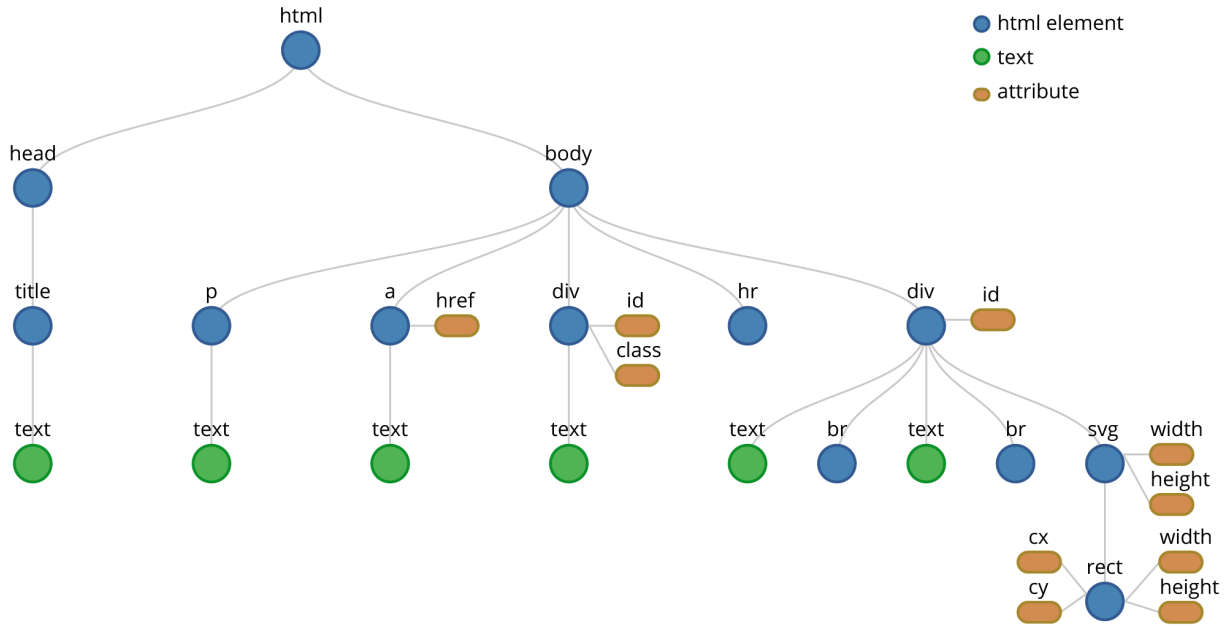
City	# of Rats
Brookline	40
Boston	90
Cambridge	30
Somerville	60

The Key to D3 is Understanding SVGs

The primary element you will find yourself working with is the SVG, or the Scalable Vector Graphic. SVG is an XML-based, web-friendly vector image format. It also provides support for animation and interactivity. SVG is unique in that all of the behaviors and components of the SVG images can be accessed from JavaScript and CSS just like any other element in a webpage. We use D3 we bind data to SVG elements.



SVG's are HTML elements that can be given attributes like anything else in the DOM.



About SVG: SVG stands for "Scalable Vector Graphic" and you can use SVGs to create graphics (shapes, lines, etc.) on websites. D3 visualizations are based on SVGs. For example, in D3 bar charts are based on a series of rectangles, scatter plots include a series of circles, line

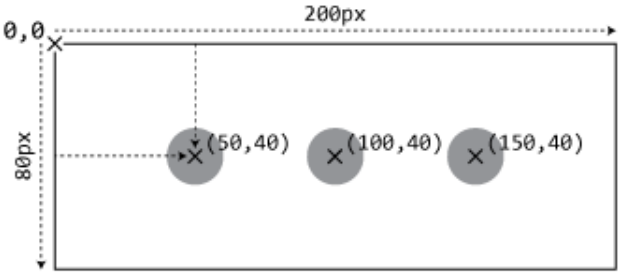
charts consist of a series of lines, etc. When you make a visualization in D3, you'll be combining SVG shapes to make a compelling visual.

SVG Coordinate system: SVGs are based on a grid with X and Y coordinates. You assign X and Y values to determine the position of the shape, measured in pixels. Unlike the mathematical grid we're used to, with SVGs the starting point (0, 0) is at the top left. When you create SVGs on a "canvas", you always set the starting point for the shape based on coordinates. If you want a circle at the top left of the canvas, you set X = 0 and Y = 0. If you want a circle inset by 100 pixels both vertically and horizontally, you set X = 100 and Y = 100.

Working with Scalable Vector Graphics (SVG)

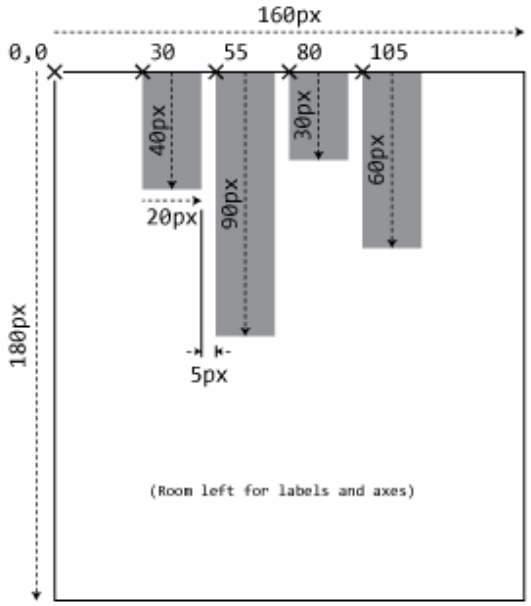
As already mentioned SVGs are page elements that can sit right within the body of your page and can be manipulated like any other element in the Document Object Model. The following shows the layout of three circles in the DOM, and how they will appear in our browser.

```
<svg width="200" height="80">
  <circle cx="50" cy="40" r="10"></circle>
  <circle cx="100" cy="40" r="10"></circle>
  <circle cx="150" cy="40" r="10"></circle>
</svg>
```



Circles with SVG: diagram based on Three Little Circles, by Mike Bostock.

```
<svg width="160" height="180">
      <rect x="30" width="20" height="40"></rect>
      <rect x="55" width="20" height="90"></rect>
      <rect x="80" width="20" height="30"></rect>
      <rect x="105" width="20" height="60"></rect>
</svg>
```



Rectangles with SVG.

Note that 0,0 is in the upper left corner of the SVG element, and all child elements are located in relation to this parent element. *For further reading and an excellent tutorial on SVG and D3, see Three Little Circles, by Mike Bostock, for a fantastic tutorial on SVG creation and manipulation* [2].

For more on SVG, its capabilities and available elements, check out the documentation.

A note about comments

Using comments allows you to document your code. They are very useful when sharing code, or even when going back to your own code after a while. It is good practice to comment your code. We can have single or multi-line comments.

Single-line comments start with two forward slash characters (//) and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code.

Multiline comments can be helpful to write out a header providing a longer description of a program. These are opened using /* and closed using */.

Let's Get Started!

In this exercise, you will create a chart that will help explain what we mean by driving a document with data.

Connect a Simple Dataset into SVG Elements Using D3

As mentioned, D3 operates with a goal of binding data to elements on our page. D3 will allow us easy access to these elements, we can use D3 to give the height to the element.

Step 1: Open d3example.html in your text editor and web browser

Step 2: Add D3 library to the d3example.html file using the following code inside your <head> tags:

```
<script src="https://d3js.org/d3.v5.min.js"></script>
```

To begin, we will manually code our data. For illustration, let's say we have a dataset with the following values (the same as the data in the table above).

exData = [40, 90, 30, 60]

Step 3: Add the following code between your <script> tags at the bottom of the page. :

exData = [40, 90, 30, 60]

We want a bar chart, with the height of each bar as the respective value in the dataset. (i.e. The first bar has a height of 40 pixels, etc.). To do so, we must bind data to an SVG object using select statements, as well as the data and attr methods.

Working with Selections

A D3 selection is an array of previously defined elements and follows the same guidelines as CSS. The selection process lets D3 select elements from the document so that operators can be to the elements, telling them to do stuff. D3 has select() and selectAll() methods to find single or multiple DOM elements, respectively.

```
{
d3.selectAll( "circle" ); // select all SVG circle elements
d3.select( "#boston" ); // select an element where id='boston'
d3.selectAll( ".bar" ); // select all elements with the class 'bar'
```

}

Since selectAll("circle") finds multiple elements, everything in the chain following this will be happening to each of those elements. Using this will iterate through each of the elements on the page that we are binding data to.

SelectAll Iteration

The selectAll method iterates through the data values one by one. Going one by one, data values are returned to the DOM. This ultimately allows for functions and operations to be performed on each data value. The order, unless specified in another way, will be from top to bottom down you page.

For more on working with selections and the options available, check out the documentation.

Additional recommended reading on selections from Mike Bostock's How Selections Work.

To make a bar chart, we must also use the data and attr methods.

The data() method

The data() method is the very soul of D3. With it, an array of data is bound to page elements.

The attr() method

The attr() method allows us to set attributes of the page elements. In this example, we set the height, width, color, and x/y location attributes.

Anonymous Functions

In our code we set height to an anonymous function. These can be a bit confusing, so let's dig into what this means.

Step 4: Add the following code between your <script> tags.

```
var svg = d3.select("div#main")
        .append("svg")
        .attr("width", width)
        .attr("height", height);
 svg.selectAll("rect")
        .data(exData)
```

```
        .enter()
        .append("rect")
        .attr( "height", d => d )
```

The argument **d** that is being passed to the function represents our dataset. The anonymous function has a parameter for the data values you just bound to your page elements in the **data** statement. This is built into D3. The name of this variable (**d**) is arbitrary, but d is usually used as it represents a data value.

The data you bind to your page elements is an object in itself. In this example, d is our object that can be operated on locally within this function. If our data object has properties, you can refer to these properties in this step. For example, if our dataset is a JSON, with two properties, number of rats (*number*) and city (*city*), we can reference it as we would in any other JavaScript object. i.e. d.number.

Step 5: Add the following code between your <script> tags (add the portion highlighted in yellow):

```
var width = 350;
var height = 175;
var barWidth = 50;

var svg = d3.select("div#main")
        .append("svg")
        .attr("width", width)
        .attr("height", height);
 svg.selectAll("rect")
        .data(exData)
        .enter()
        .append("rect")
        .attr( "height", d => d )
        .attr( "x", (d,i) => i*barWidth + 50 )
        .attr( "y", 0 )
        .attr( "width", barWidth )
        .attr( "fill", "red");
```

Your results should look as follows:

Things just got a little weird. Did we just select a bunch of rectangle elements that don't exist? Well, yes... kinda. With D3, you always have to first select what you are going to be operating on, even if it doesn't yet exist. This is a bit abstract, but hang with me, the next steps will explain this more.

What we did is select a bunch of rectangles that are not there, so we get an empty selection. The next few lines of code in our block above create these elements by binding data, using the enter(), and appending a new element.

1. data() - We bind the data to our empty selection using the data() method, it will return the four data values in our **dataset**.
2. enter() - When we load data, it will iterate through the dataset and apply all methods that follow to each of the values of our dataset. The enter() method creates placeholders for each data element for which no corresponding DOM element was found. Because it iterates, it will create four placeholders.
3. append() - Finally, the append("rect") method will insert a rectangle into each of the placeholders that do not have a "rect" element, which is all of them.
4. attr() - Iteratively sets attributes, such as (x,y) location, width, and height for each of the rectangle elements. Right now, these are all in the same location **(0,0)** and have the same width **(20)** and height **(100)**. We need to use functions to make this work properly, and will detail that next.

Save and refresh your document. You should see this... pretty boring, but if you see this it is working! You have four rectangles, but they are all in the same location.

Working with the Enter and Exit methods

The enter() and exit() methods deal with new elements and unused elements, respectively, based on incoming data. It's worth taking a minute to fully grasp these!

The enter() method tells D3 what to do when there are more elements in a data array than there are elements in the selection. So if you have 12 rows in an input dataset but only 8 rect elements when the enter() method is invoked, D3 will go to what follows the enter() method to determine what to do. Usually, the enter function will be used to create (append()) new elements to a given visualization.

The exit() method tells D3 what to do when there are more elements in a data array than there are elements in a selection. Usually, this will be used to clear elements that are no longer needed. You can think of these like the 'go' and 'stop' methods, where enter() is the former and exit() is the latter.

Styling the rect Elements

Finally, we properly size and arrange the rect elements. To do this, we can modify the height attribute and x and y attributes for each of our rectangle elements. The attributes can read functions that allow us to dynamically change attributes based on the data value of the current iteration.

So, let's update the attributes to properly display the “rect” elements by changing the x attribute and height attribute. This will look familiar! It is exactly how we assigned a height in the previous example.

Step 6: Change the code to look like the following:

```
var width = 150;
var height = 175;

var svg = d3.select("div#main")
        .append("svg")
        .attr("width", width)
        .attr("height", height);
 svg.selectAll("rect")
        .data(exData)
        .enter()
```

```
.append("rect")
.attr( "height", d => d )
.attr( "x", (d,i) => i*25 + 30 )
.attr( "y", 0 )
.attr( "width", 20 )
.attr( "fill", "steelblue");
```

Your results should look as follows:

Setting the x attribute

You notice here a second argument is provided to our anonymous function (**i**). This represents the index location of the data value referred to in the **d** argument. You are looping through the index as you add the attributes, so i in this case has 4 index values.

Here we space the bars horizontally using the second i argument. The i argument is the index of each bar in the selection as the code iterates through. For our four rectangles, **i** here will be 0, 1, 2, and 3, giving us **x** positions of **0, 25, 50, and 75**.

Setting the height attribute

The d argument is the data value for each piece of data. The function will return the value for each rectangle based on the data value, setting the height attribute equal to the data value for each respective rectangle.

Setting the fill attribute

Let's color our chart while we are at it. Set the fill attribute to the color of your choice. I used steelblue. This attribute will take web standard colors. Look up some Hex codes and pick your favorite color, or just use steelblue.

In Class Example

Modify the previous code and use different height and width values for your chart. For example, you can make your bars wider, or multiply the heights by a constant.

Design the Chart

As is, this chart is not very useful. In fact, it is just four rectangles. We need to add some context, re-justify the bars, and perhaps add some axes. We can do this right in our script by adjusting the attributes and properties of the SVG elements. In the following steps, we'll add those axes and label.

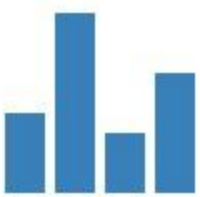
Bottom-justify the Bars

The chart is confusing. The higher the number the farther down the page the bar extends. We can change this quite easily by adjusting the value of the y attribute. See the adjustment to our code below.

```
var width = 150;
var height = 175;

exData = [ 40, 90, 30, 60 ]

          var svg = d3.select("div#main")
         .append("svg")
         .attr("width", width)
         .attr("height", height);
 svg.selectAll("rect")
         .data(exData)
         .enter()
         .append("rect")
         .attr( "height", d => d ) // Set height of rectangle to data value
```

```
    .attr( "x", (d,i) => i*25 + 30 )
    .attr( "y", d => height-d )  // Set y coordinate for each bar to height minus the data value
    .attr( "width", 20 )
    .attr( "fill", "steelblue");
```

Your results should look as follows:

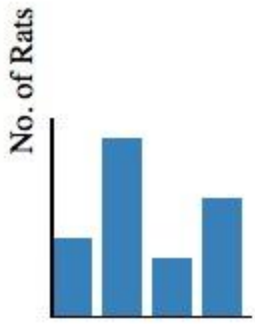


In Class Example

Modify the previous code and use a different color for the bars.

Add Labels

This will provide a bit more context, and add axes to the map along the left side and bottom. When finished, our chart will look like this. Let's take a look.



Add the X and Y axes

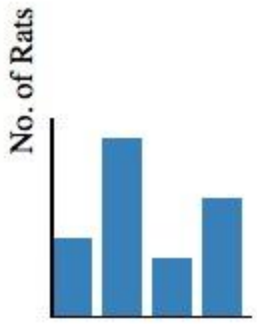
A simple method of doing this is to add SVG line elements to our script. We can manually set the X and Y coordinates for the start and end of the line, along with the stroke and width.

Add a Label

There are ways when working with numbers to automate this, but for now, let's create one label by adding text elements to our svg element. We can also use some CSS in style tags to modify your font.

```
var width = 150;
var height = 175;


exData = [ 40, 90, 30, 60 ]

          var svg = d3.select("div#main")
         .append("svg")
         .attr("width", width)
         .attr("height", height);
 svg.selectAll("rect")
         .data(exData)
         .enter()
         .append("rect")
         .attr( "height", d => d ) // Set height of rectangle to data value

         .attr( "x", (d,i) => i*25 + 30 )
         .attr( "y", d => height-d )   // Set y coordinate for each bar to height minus the data value
         .attr( "width", 20 )
         .attr( "fill", "steelblue");

// Create y-axis
svg.append("line")
          .attr("x1", 30)
          .attr("y1", 75)
          .attr("x2", 30)
          .attr("y2", 175)
          .attr("stroke-width", 2)
          .attr("stroke", "black");
```

```
// Create x-axis
svg.append("line")
        .attr("x1", 30)
        .attr("y1", 175)
        .attr("x2", 130)
        .attr("y2", 175)
        .attr("stroke-width", 2)
        .attr("stroke", "black");

 // Add a Label
 // y-axis label
 svg.append("text")
        .attr("class", "y label")
        .attr("text-anchor", "end")
        .text("No. of Rats")
        .attr("transform", "translate(20, 20) rotate(-90)");
```



What happens if our array changes size?

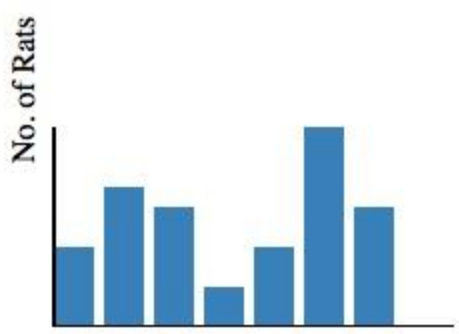
Data can change. What if your number of data values changes? Or the value of the maximum data value changes? There are two different methods for this. One is to manually change your scales / axis, etc., and the other is to use the D3 Domain and range methods (which we'll learn later). We will use a combo of the two in our example.

Let's add a few more elements to our data array and update some other parts of our code - here is an example of manually making updates:

```
var width = 450;
var height = 175;

exData = [ 40, 70, 60, 20, 40, 100, 60 ];

var svg = d3.select("div#main")
        .append("svg")
        .attr("width", width)
        .attr("height", height);
 svg.selectAll("rect")
        .data(exData)
        .enter()
        .append("rect")
        .attr( "height", d => d )
        .attr( "x", (d,i) => i*25 + 30 )
        .attr( "y", d => height-d )
        .attr( "width", 20 )
        .attr( "fill", "steelblue");

// Create y-axis
svg.append("line")
        .attr("x1", 30)
        .attr("y1", 75)
        .attr("x2", 30)
        .attr("y2", 175)
        .attr("stroke-width", 2)
        .attr("stroke", "black");

// Create x-axis
svg.append("line")
        .attr("x1", 30)
        .attr("y1", 175)
        .attr("x2", 230)
        .attr("y2", 175)
        .attr("stroke-width", 2)
        .attr("stroke", "black");

  // Add a Label
  // y-axis label
```

```
svg.append("text")
        .attr("class", "y label")
        .attr("text-anchor", "end")
        .text("No. of Rats")
        .attr("transform", "translate(20, 20) rotate(-90)");
```

Your results should look as follows:



Add Simple Hovering

In the last step in this exercise, we are going to label the bars of our chart and show the number of rats. Let's do this by adding a hover tooltip.

The hover tooltip is created in two steps.

1. Using D3, create a DIV element for the tooltip and set it to be hidden
2. Use mouse event listeners (mouseover and mouseout) to listen for hover and set properties of the tooltip DIV element

Create the Tooltip div element

Add the following code to your <body> element (you can put this above the div with id="main"):

```
<div id="tooltip" class="hidden">

    <p><span id="value">100</span></p>

</div>
```

Next, add CSS styles to add and style the tooltip:

```
<style>
    #tooltip.hidden {
        display: none;
    }

    #tooltip {
        position: absolute;
        width: 100px;
        height: 65px;
        padding: 5px;
        background-color: white;
        border: solid gray 1px;
        z-index:10;
    }
</style>
```

About adding Event Listeners for mouse action to the rectangle generating method

Locate the method in which we create the rectangles based on the data. Here we need to add methods that listen for mouse events and modify the tooltip div based on the mouse event.

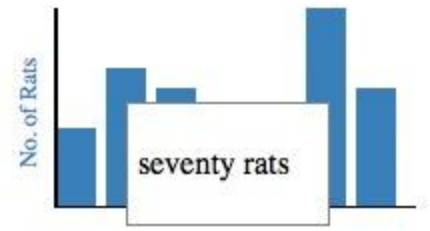
1. Set the tooltip to display on **mouseover**,
2. Set it move with the mouse on **mousemove**,

3. And, hide it on **mouseout**.

Next, modify your code to include event listeners, and add to your dataset to include label text:

```
var width = 450;
var height = 175;

exData = [ [40,"forty rats"], [70,"seventy rats"], [60,"sixty rats"], [20,"twenty rats"], [40, "forty rats"], [100,"one-hundred rats"],[60,"sixty rats"] ];


var svg = d3.select("div#main")
        .append("svg")
        .attr("width", width)
        .attr("height", height);
 svg.selectAll("rect")
        .data(exData)
        .enter()
        .append("rect")
        .attr( "height", d => d[0] )
        .attr( "x", (d,i) => i*25 + 30 )
        .attr( "y", d => height-d[0] )
        .attr( "width", 20 )
        .attr( "fill", "steelblue")
        .on("mouseover", function(d) {
                                        d3.select("#tooltip")
                                        .style("left", d3.event.pageX + "px")
                                        .style("top", d3.event.pageY + "px")
                                        .select("#value")
                                        .html("<p>" + String(d[1]) +"</p>")
                                d3.select("#tooltip").classed("hidden", false);
                                })

                                .on("mouseout", function() {
                                        d3.select("#tooltip").classed("hidden", true);
                                });
```

Your chart should look similar to the following:



There we have it!

Exhausted? You should be, we just covered a lot! These are some basics of D3, displaying how to work with a simple array dataset, bind that dataset to an SVG, and create SVG elements that are encoded with our data. We have merely scratched the tip of the iceberg, but we'll spend next week building on what we talked about today.