Lab 3: Intro to D3 Pre Lab

Wijaya, Mario edited this page 6 hours ago · 1 revision

Learning Objectives

After completing this lab you will be able to:

- Load data with D3 (includes learning about asynchronous methods)
- Method chaining with JavaScript
- Basics of D3:
  - Create a D3 selection
  - Append elements with D3
  - Data-bindings with D3 to create new HTML elements based on data
  - Set HTML style and attributes with D3

Prerequisites

- Update your *starter code repository* using one of the following methods:
  i. From the GitHub Desktop app click `Sync` on the top right
  ii. Open a command line prompt. Navigate to the repository directory, for example `cd ~\Development\CS-4460\labs` and run command `git pull`.
- You have **read Chapters 5 and 6** in D3 - Interactive Data Visualization for the Web by Scott Murray

Additional Reading

- Selections in d3 – the long story by Jerome Cukier
- D3.js Introduction
- Let's Make a Bar Chart by Mike Bostock (creator of D3)
- DOM Manipulation and D3 Fundamentals by A. Lex of U. of Utah
- Dashing D3.js
- Practical applications of a d3js selection

Data-Driven Documents (D3)

D3.js (Document-Driven-Data) is a powerful JavaScript library for manipulating documents based on data.

> "D3 allows you to bind arbitrary data to a Document Object Model (DOM), and then apply data-driven transformations to the document. For example, you can use D3 to generate an HTML table from an array of numbers. Or, use the same data to create an interactive SVG bar chart with smooth transitions and interaction." (D3.js, Mike Bostock)

A summary of D3's features and key aspects by Scott Murray:

- **Loading** data into the browser`s memory
- **Binding** data to elements within the document and creating new elements as needed

Pages 6

Lab 0: HTML & CSS
Lab 1: Javascript 101
Lab 2: SVG
Lab 3: Intro to D3 (Pre Lab)
Lab 3: Intro to D3
Lab 4: D3 Chart Types & Scales
Lab 5: D3 Selections & Grouping
Lab 6: D3 Enter, Update & Exit
Lab 7: Interaction & Transition 1
Lab 8: Interaction & Transition 2
Lab 9: D3 Layouts
Lab 10: D3 Maps

- **Transforming** those elements by interpreting each element's bound datum and setting its visual properties accordingly
- **Transitioning** elements between states in response to user input

We will introduce all these concepts in the following weeks.

D3 Version 4.x

We will use the new version (4.x) of D3 in this class. D3 version 4.x was previously released in 2016. It has some additional features compared to previous versions and it is modular. This means that it is now built on many small libraries, instead of one large bundle. This allows developers to pick only the parts they are interested in and most importantly, it allows independent release cycles. Thus, the driving forces behind D3 can now improve the micro libraries independently.

Normally, you will access a bundled version of the entire D3 library (https://d3js.org/d3.v4.min.js) - *so you don't need to worry about the micro libraries.*

Important: Scott Murray's book Interactive Data Visualization for the Web was written for D3 version 3.x. You will notice that some of the version 3.x code looks different from the version 4.x code. Most notably, the namespace for version 4.x follows a `camelCase`, whereas 3.x uses `.dot.notation` - this is really the only major difference.

D3 Integration

This is a brief overview of how to set up a basic D3 project. This should not be completely new but it might help with future projects or homework.

Before working with D3 you have to include the D3 library first. We recommend the minified version which has a smaller file size and faster loading time. D3 hosts the latest minified version at: https://d3js.org/d3.v4.min.js

You will include a `<script>` element in your html to include the D3 library:

```
        ...
                </div>
        </body>
        <script src="https://d3js.org/d3.v4.min.js"></script>
        <script src="./my_javascript.js"></script>
</html>
```

What does this do? Including a library with a `<script>` element loads the library as a global namespace. For D3 the global namespace is: `d3`.

`d3` - References the D3 object, so we can access its methods and properties by starting a statement with `d3`.

> You should keep your own JavaScript code separated from the JavaScript libraries you are using. In the future you might have more than one library and don't want to change your code every time you update one of them (e.g. new release), so make sure you encapsulate your own code into separate files (libraries).

Make sure to include other libraries before using them in your code (order of script tags).

Loading Data Files

Instead of typing the data in a local variable (like we did in earlier labs), we can load data asynchronously from external files. The D3 built-in methods make it easy to load JSON, CSV and other data file types. The use of the right file format depends on the data - JSON should be used for hierarchical data and CSV is usually a proper way to store tabular data.

By calling D3 methods like `d3.csv()`, `d3.json()`, `d3.tsv()` etc. we can load external data resources in the browser:

```
d3.csv('./baseball_hr_leaders_2017.csv', function(error, dataset) {
        // Here you have access to the dataset variable (if there was no error)
});
```

These functions (asynchronous requests) take two arguments: a string representing the path of the file, and an anonymous function, to be used as a callback function.

Callback Functions and Asynchronous Execution

Why do we need an asynchronous execution? → The page should be visible while data is loading and scripts that do not depend on the data should run immediately, while scripts that do depend on the data should only run once the data has been loaded!

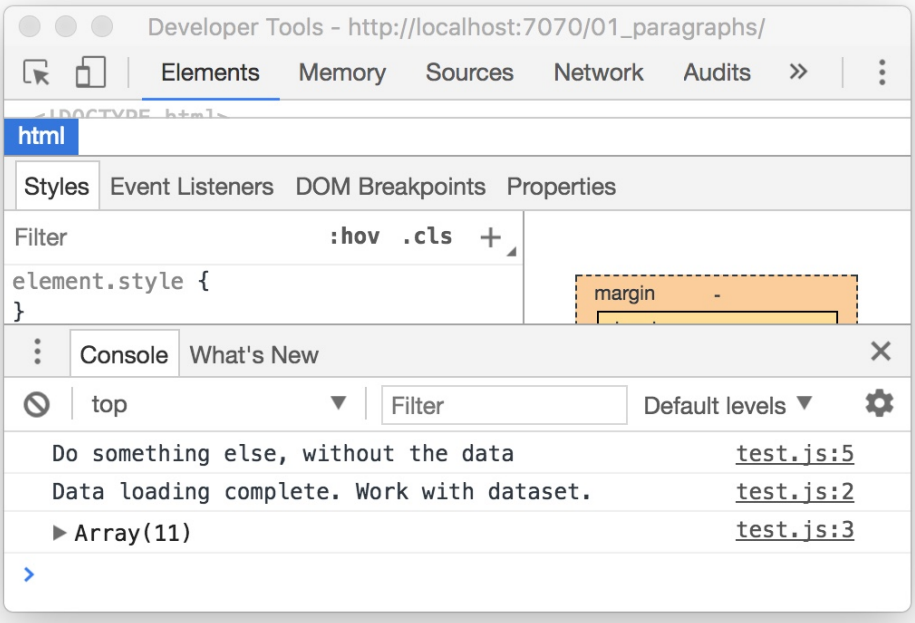
A callback function is a function that is passed to another function. It can be anonymous or named. That means, we don't call the anonymous function directly and it is also not getting executed immediately. The callback function is invoked after an event (such as the data loading) and it is "called back" once its parent function is complete.

In our data loading problem loading a file from the disk or an external server takes a while. Hence, we are using an asynchronous execution: We don't have to wait and stall, instead we can proceed with further tasks that do not rely on the dataset. After receiving a notification that the data loading process is complete, the callback function is executed.

Code that depends on the dataset should generally exist only in the callback function! (You can still structure your code in separate functions, however, if these functions depend on the dataset, they should only be called inside the callback function).

```
d3.csv('./baseball_hr_leaders_2017.csv', function(error, dataset) {
        console.log("Data loading complete. Work with dataset.");
        console.log(dataset);
});
console.log("Do something else, without the data");
```

The result below shows that the execution order is different from what you might have expected:



The callback function - the inner function of `d3.csv()` - is called only after the dataset is loaded completely to browser memory. In the meantime other scripts are executed.

JavaScript Method Chaining

Method or function chaining is a common technique in JavaScript, especially when working with D3. It can be used to simplify code in scenarios that involve calling multiple methods on the same object consecutively. The functions are "chained" together with periods.

```
d3.selectAll('circle').attr('r', 10).style('fill', '#777');
```

Note: Chaining is only possible when: the return type of a method is the original caller. For example `.attr()` returns the original selection object `d3.selectAll('circle')`.

Alternative code without method chaining:

```
var circles = d3.selectAll('circle');
circles.attr('r', 10);
circles.style('fill', '#777');
```

The main drawback here is that you have to instantiate a new variable and use 2 extra lines of code. The example code from here on out will use method chaining extensively.

D3 Selections

The selection is your entry point for finding, creating, and removing DOM elements. You also use selections to apply styles and attributes that dictate how those elements appear on the page and respond to events.

`d3.select()` selects the first element that matches a selector. Selectors can specify tags (`p` in our example below), classes, and IDs, all through the same interface:

```
d3.select('p')
        .style('font-size', '0.9em')
        .text('First Paragraph');
```

Notice, however, that as mentioned previously, only the first element that matches is selected. Of course, it is more practical to select all elements of a certain type, which we can achieve with `d3.selectAll()`:

```
d3.selectAll('p')
        .style('font-size', '0.9em');
```

This example illustrates **the declarative approach of D3:** we don't have to iterate over a list of elements and apply the style. Instead we select a set of elements through rules and declare properties.

Once you have a selection, you can bulk-modify it's content, not only in terms of style, but we can modify arbitrary attributes using `selection.attr(name[, value])`, the textual content of the elements with `selection.text([value])`, etc. We can also append elements to a selection.

Append Elements

With a D3 selection, we can now `append()` elements to the selection. You should already have some experience with this from the last lab. For example, calling `svg.append('g')` creates a new `<g>` element and adds it as a child of the `svg` selection. Note that `append()` adds child

elements at the end of the child list, while `insert()` adds them to the beginning of the child list of an element.

```
var group = svg.append('g');
group.append('circle');
group.append('text');
```

Finally, note that D3 returns a javascript object when you call `d3.append()` or `d3.insert()`. This javascript object is another D3 selection. Appending elements that have just been added is how you create nested structures. For example, svg grouping shown above would create this svg:

```
<g>
        <circle/>
        <text></text>
</g>
```

Binding Data

What is binding, and why would I want to do it to my data?

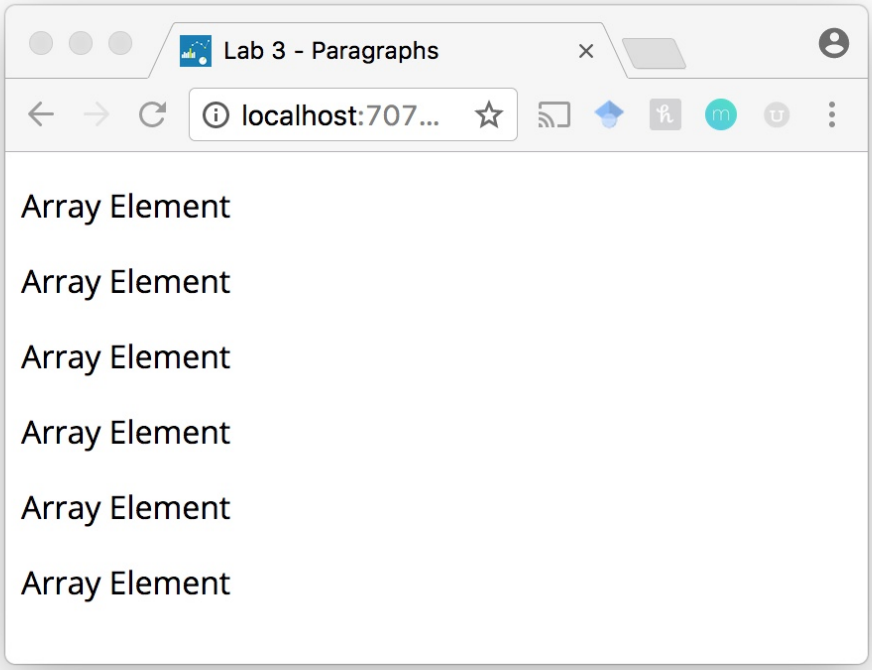
Data visualization is a process of mapping data to visuals. Data in, visual properties out. Maybe bigger numbers make taller bars, or special categories trigger brighter colors. The mapping rules are up to you. With D3, we bind our data input values to elements in the DOM. Binding is like “attaching” or associating data to specific elements, so that later you can reference those values to apply mapping rules.

We use D3’s `selection.data()` method to bind data to DOM elements. Binding data requires two things:

- The data (an array of objects, strings, numbers, etc.)
- A selection of DOM elements - the elements your data will be associated with

```
var states = ["Connecticut", "Maine", "Massachusetts", "New Hampshire", "Rhode Islan

var p = d3.select("body").selectAll("p")
        .data(states)
        .enter()
        .append("p")
        .text("Array Element");
```



Here's what's happening:

```
d3.select("body")
```

Finds the body in the DOM and hands off a reference to the next step in the chain.

```
.selectAll("p")
```

Selects all paragraphs in the DOM. Because none exist yet, this returns an empty selection. Think of this empty selection as representing the paragraphs that will soon exist.

```
.data(dataset)
```

Counts and parses our data values. There are six values in our array called `states`, so everything past this point is executed six times, once for each value.

```
.enter()
```

To create new, data-bound elements, you must use `enter()`. This method looks at the current DOM selection, and then at the data being handed to it. If there are more data values than corresponding DOM elements, then `enter()` creates a new placeholder element on which you can work your magic. It then hands off a reference to this new placeholder to the next step in the chain.

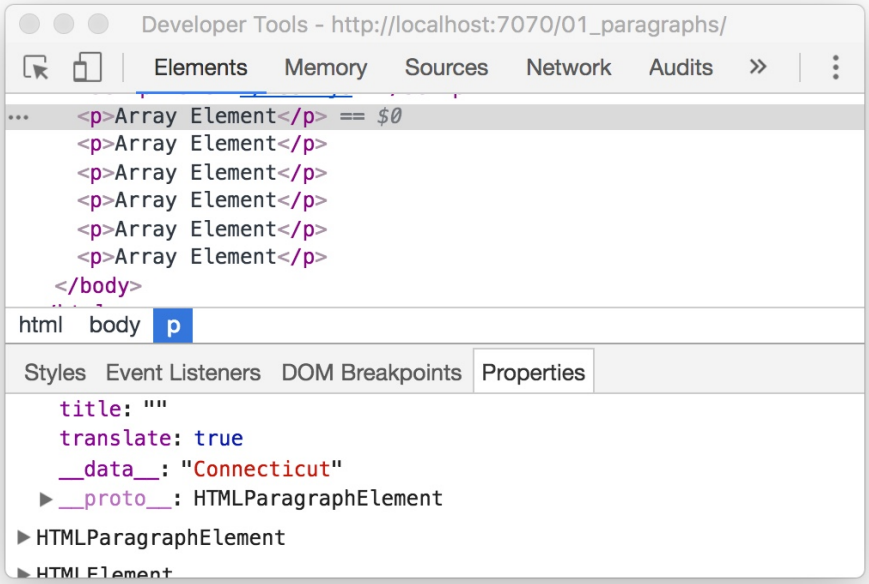
```
.append("p")
```

Takes the empty placeholder selection created by `enter()` and appends a `p` element into the DOM. Hooray! Then it hands off a reference to the element it just created to the next step in the chain.

```
.text("Array Element")
```

Takes the reference to the newly created p and inserts a text value.

If we inspect the `p` elements that we added to the `body` with the Web Inspector, we will see that in the "Properties" tab for each of these `p` elements that there is a `__data__` property associated with each.



When D3 binds data to an element, that data doesn't exist in the DOM, but it does exist in memory as a `__data__` attribute of that element. And the console is where you can go to confirm whether or not your data was bound as expected.

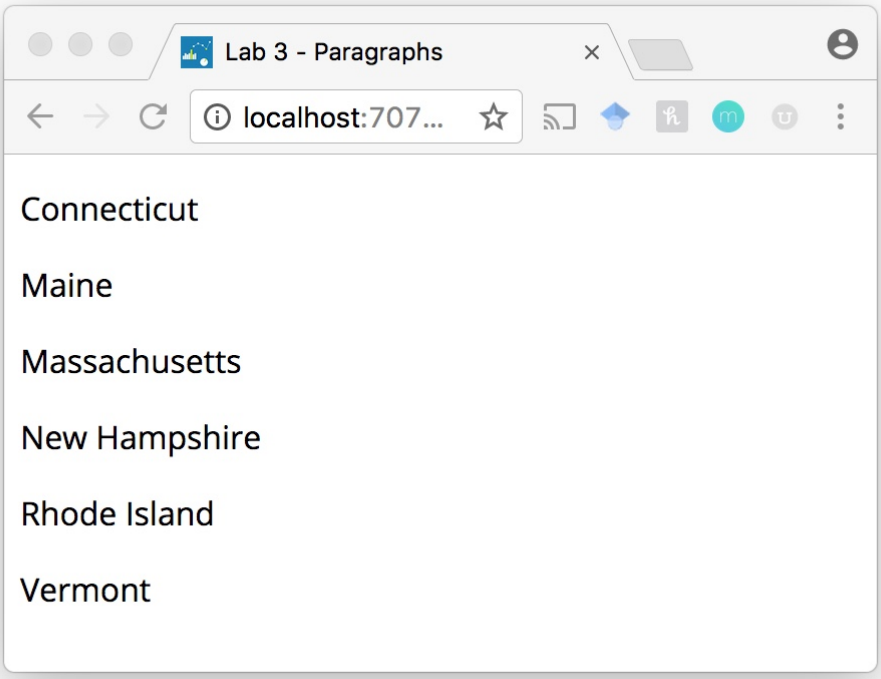
We can see that the data has been loaded into the page and is bound to our newly created elements in the DOM. Now we are going to use each string bound to the `p` elements by setting the text contents of each `p` to the state name:

```
var p = d3.select("body").selectAll("p")
        .data(states)
        .enter()
        .append("p")
        .text("Array Element");
```

Let's change the last line to:

```
        .text(function(d, i) { return d; });
```

Now we will see the following on our page:



What happened here? We used an anonymous function that is called by D3 for each element in the selection. The anonymous function takes two inputs `function(d, i)`.

- `d` = the data element bound (in this case each state, e.g. "Connecticut", "Maine", etc.)
- `i` = the index for that bound array element (e.g. 0, 1, etc.)

We will use anonymous functions very often in D3 to access individual values and to create interactive properties.

In our case we are using the function to access individual values of the loaded array. That is one feature of D3: It can pass array/data elements and corresponding data indices to an anonymous function (which is called for each array element individually).

Generally in D3 documentation and tutorials, you'll see the parameter `d` used for the current data element and `i` (or index) used for the index of the current data element. The index is passed in as the second element to the function calls and is optional.

Note, an anonymous function is still a regular function, so it doesn't have to be a simple return statement. We can use if-statements, forloops, and we can also access the index of the current element in our selection.

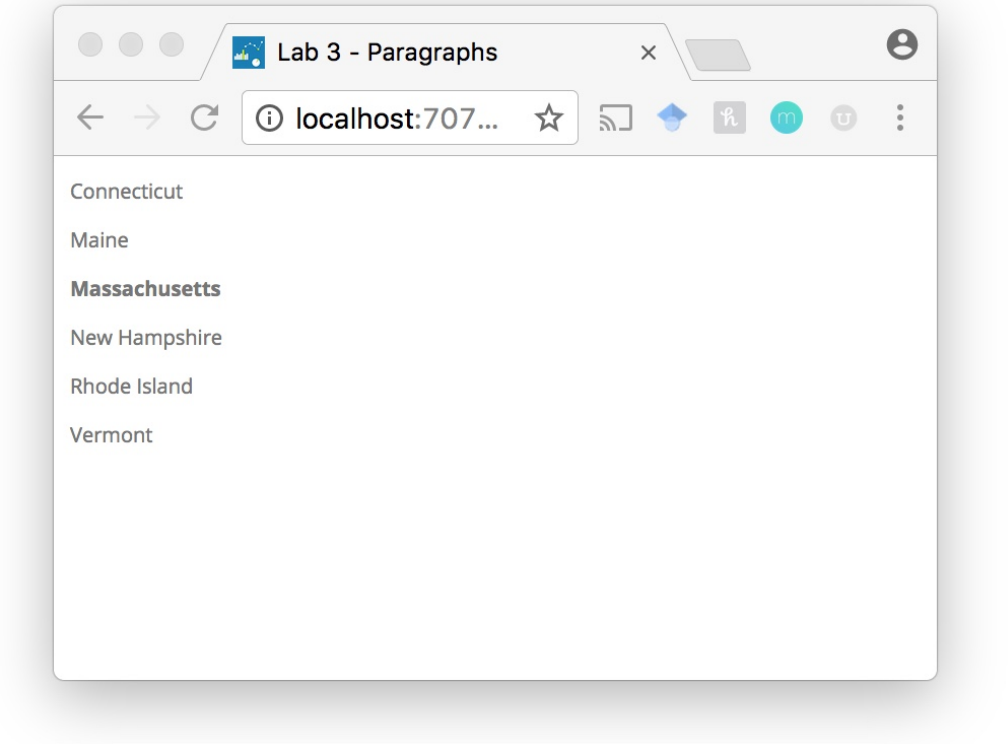
HTML Attributes and CSS Properties

Styles, attributes, and other properties can be specified as functions of data in D3, not just simple constants. As already mentioned earlier, we can get and set different properties and styles - not only the textual content. This becomes very important when working with SVG elements.

For example, in the previous example we can emphasize "Massachusetts" with a bold font-weight. We can also style all of the `p` elements:

```
var p = d3.select('body').selectAll('.state-name')
        .data(states)
        .enter()
        .append('p')
```

```
        .attr('class', 'state-name')
        .text(function(d, i) { return d; })
        .style('color', '#777')
        .style('font-size', '10px')
        .style('font-weight', function(d) {
                return d == 'Massachusetts' ? 'bold' : 'normal';
        });
```



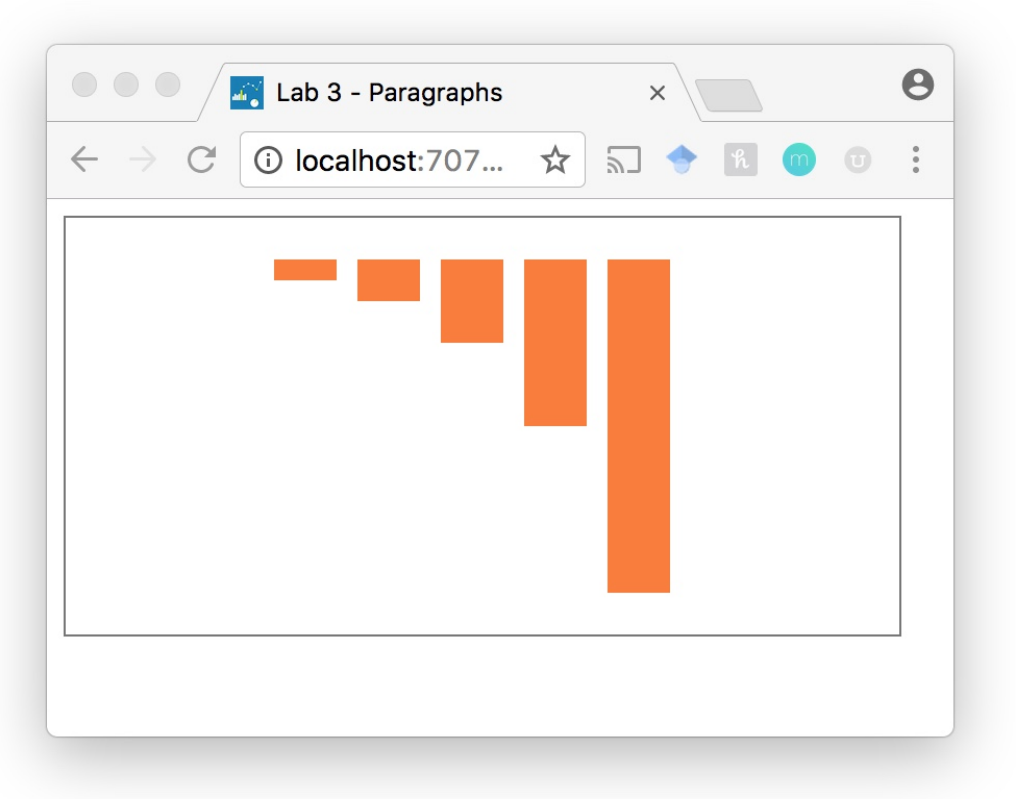
- We use D3 to set the paragraph content, the HTML class, the `color` and as the last property, the `font-weight` which depends on the individual array value.
- If you want to assign specific styles to the whole selection (e.g. font-color: blue), we recommend you define an HTML class ("state-name" in our example) and add these rules in an external stylesheet. That will make your code concise and reusable.

In the following example we use numeric data to create five rectangles. We then style them by data:

```
var numericData = [1, 2, 4, 8, 16];

var svg = d3.select('svg');

// Add rectangles
svg.selectAll('rect')
        .data(numericData)
        .enter()
        .append('rect')
        .attr('fill', '#f77e46')
        .attr('width', 30)
        .attr('height', function(d){
        return 160 * d / 16;
        })
        .attr('y', 20)
        .attr('x', function(d, i) {
        return (i * 40) + 100;
        });
```



- We have appended SVG elements to the DOM tree in our second example.
- It is crucial to set the SVG coordinates. If we don't set the x and y values, all the rectangles will be drawn on the same position at (0, 0). By using the index - of the current element in the selection - we can create a dynamic x property and shift every newly created rectangle `40px` to the right (plus an initial `100px` to center them).
- We also set the `height` attribute of each rectangle based on the bound data. Remember from our previous lab that the y-coordinate system in SVG starts at the top (0px) and goes down. This is why our bar chart is aligned at the top. We would need to set the y-position of the rectangles based on data to align them at the bottom (hint for upcoming homework).

Home