Intro to D3

As we saw in the exercises for the previous week, you can manipulate the DOM - adding graphical primitives to an svg region on a webpage- manually or using javascript.
Here we are going to introduce D3, a javascript library that was designed to manipulate the DOM based on data. It is one of the most popular projects on github. The key insight of D3 is to allow users to express visual encodings (how data is related to graphical elements) in a completely novel form, and was designed with the DOM in mind.
To quote its creator:
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, Mike Bostock*

D3 is generally used with svg, but can also be used with html directly. At core- you connect data to elements in the DOM. D3 is also not a single framework, but many modules that you can use for varying purposes. For example, last week we used the d3 csv loaded, without using D3 in the rest of the project.
NOTE: D3 version 7 is the most recent version, and what we will use in the class. D3 v4 was a big update from v3, but you will find many examples online in v3. Be aware of this so you avoid looking at examples written for v3.

How to include d3 in your projects

Here is an example d3 project starter directory:
https://drive.google.com/file/d/1cAOKGT3t4Mir39cT_yirajxEXsv78mhB/view?usp=drive_link

In our coding template directory, we put the script for d3 in a library folder. You can also link to an external d3 library. Putting it in the folder allows you to control which version of d3 you are using.

```
project/
     index.html
     css/
          style.css
     js/
          d3.min.js
```

```
          main.js
```

Then you just need to include it in html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>D3 Project</title>
    <link rel="stylesheet" href="css/style.css">
</head>
<body>

     <script src="js/d3.min.js"></script>
     <script src="js/main.js"></script>
</body>
</html>
```

Simplest d3 example- method chaining

Previously, when we wanted to add an element to the DOM for each element in the array, we looped through the array, and then added and styled the element. In D3, we don't use a loop. Let's see how it works:

You can use d3 to add elements to the DOM. Here is a simple example:
`d3.select("body").append("p").text("Hello World!");`

This one-line operation works due to **'method chaining"**. In method chaining

- Functions are chained together with periods.
- The output from one function flows into the input of the next.

Alternatively, we could have written the same thing as above, but on multiple lines:

```
const body = d3.select('body');

const p = body.append('p');

p.text('Hello World!');
```

What are the steps in this code?

D3 Select:

With **[d3.select]()** you can grab page elements with CSS selectors. This then returns a reference to the first element in the DOM that matches the selector.

You can select more than one reference using **d3.selectAll().** This will return more than one element.   But in the example above, we select the body, and then pass it to the next function in the chain (append).

Later we will see examples with more

D3 Append:

Append adds a child to the selection, placing it last if there are other children. In the example above, this means that an empty paragraph is added to the body of the text, the last child if there are other children.
Finally, we can use the text operation to add text between the tags of our appended paragraph.
Typically we write this operation in this way, with each chain on its own line:

```
d3.select('body')
      .append('p')
      .text('Hello World!');
```

Select all

In the last example, we added one paragraph. What if we wanted to add a paragraph for every data element in an array?

```
const iceCreamFlavors = ['vanilla', 'chocolate', 'strawberry',
'cookies and cream', 'cookie dough'];

const p = d3.select('body').selectAll('p')
      .data(iceCreamFlavors)
      .enter()
      .append('p')
      .text('Ice Cream Flavor');
```

Result:

Ice Cream Flavor

Ice Cream Flavor

Ice Cream Flavor

Ice Cream Flavor

Ice Cream Flavor

Let's step through it:

1. `.select('body')` This returns a reference to the body of the page
2. `.selectAll('p')` This selection represents the elements we want to create (paragraphs)
3. `.data(iceCreamFlavor)` Loads the dataset (array of strings) into the chain. The contents of data is not important- it could be objects, integers. This is where each element in the array is assigned to one of the items selected above ('p').
   The `data()` operator returns **three virtual selections - you can use any of these in the next step of the chain:**
   - **Enter** has a new placeholder for any missing elements (for instance, in the above example where there are no paragraphs). this is typically what you use when creating a visualization.
   - **Update** contains existing elements bound to the data (for instance, if you were adding new ice cream flavors)
   - **Exit** contains existing elements that are not bound to data anymore and should be removed (if the array now had fewer flavors)
4. In the above situation, there are no <p> elements within the body, so the enter selection wlll have placeholders for each element in the array. For now, we will focus on the enter selection, and will discuss update and exit in later tutorials.
5. `.enter()` Placeholders are returned here for each element in the array, for the selected element (p).

6. `.append('p')` This is where the new paragraph element is appended to the placeholder, for each element in the dataset.
7. `.text('Ice Cream Flavor')` For each of the returned placeholders, add some text.

This doesn't seem very useful yet. We have only created elements to match the number of entries in the array. But the content is not connected to the data itself. That comes next.

```
const iceCreamFlavors = ['vanilla', 'chocolate', 'strawberry',
'cookies and cream', 'cookie dough'];

const p = d3.select('body').selectAll('p')
    .data(iceCreamFlavors)
    .enter()
    .append('p')
    .text(d => d);
```

What happened here? This is where understanding inline functions, anonymous functions and arrow functions is important. The last line `.text(d => d)` can be read as - take each data element in the array as input to a function, and return that input. You could also have written this one of two ways:

```
// preferred option: ES6 arrow function syntax
.text(d => d);

// Alternative: Traditional function syntax
.text( function(d) { return d; } );
```

In case that is confusing, go back and review arrow functions in the previous tutorial on Javascript part 2.

Why ‘d’

And here's the output

vanilla

chocolate

strawberry

cookies and cream

cookie dough

In this case, we wanted to directly use the data elements in the array themselves, but suppose the array held objects, and we wanted to use a field. We can do that too:

```
.text(d => d.firstName);
```

The above examples passed elements in the array. You can also pass the index of the data elements in the array. For instance:

```
.text((d, index) => {
     console.log(index); // Debug variable
     return `element: ${d} at position: ${index}`;
});
```

The functions used here can be regular functions, with if/else statements and other operations.

Setting properties:

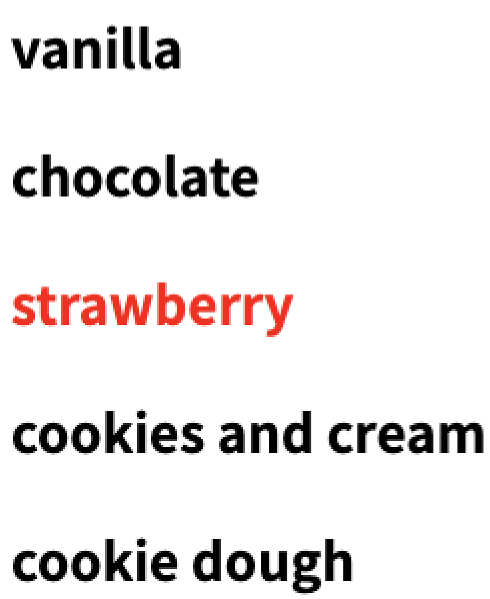
Once the elements are bound to the data, you can set other properties, like HTML attributes or CSS properties.

```
const iceCreamFlavors = ['vanilla', 'chocolate', 'strawberry', 
'cookies and cream', 'cookie dough'];



// Append paragraphs and highlight one element
```

```
let p = d3.select('body').selectAll('p')
    .data(iceCreamFlavors)
    .enter()
  .append('p') //now we have a paragraph for each element in the array
    .text(d => d) //text is set to the element
    .attr('class', 'custom-paragraph') //a class label allows you to
define styles in css, or select these elements later
    .style('font-weight', 'bold')
    .style('color', d => {
      if(d == 'strawberry')
        return 'red';
      else
        return 'black';
    });
```



SVG

You can append graphical elements to the svg in the same way.

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

// Add <svg> element (drawing space)
```

```
const svg = d3.select('body').append('svg')
    .attr('width', 300)
    .attr('height', 50);

// Add rectangle
svg.selectAll('rect')
    .data(numericData)
    .enter()
  .append('rect')
    .attr('fill', 'red')
    .attr('width', 50)
    .attr('height', (d) => d*10)
    .attr('y', 0)
    .attr('x', (d, index) => index * 60);
```

- It is crucial to set the SVG coordinates of visual elements. If we don't set the `x` and `y` values, all the rectangles will be drawn at the same position at (0, 0). We use the index to set the position of each bar.
- We also use the data value to set the height of the bars.

Practice:

1. **Create a new D3 project or clone the [d3-starter-template-master-1.zip]()**
2. **[Download d3-starter-template-master-1.zip]()**
3. **Append a new SVG element to your HTML document with D3** (width: 500px, height: 500px)

4. Draw circles with D3
Append a new **SVG circle** for every object in the following array:

```
const sandwiches = [
      { name: "Ham", price: 7.95, size: "large" },
```

```
      { name: "Turkey", price: 8.95, size: "large" },
      { name: "Veggie", price: 6.50, size: "small" },
      { name: "Tune", price: 6.50, size: "small" },
      { name: "Roast Beef", price: 7.95, size: "large" },
      { name: "Special", price: 12.50, size: "small" }
     ];
```

5. Define dynamic properties

- Set the x/y coordinates and make sure that the circles don't overlap each other
- Radius: *large sandwiches* should be twice as big as small ones
- Colors: use two different circle colours. One colour (`fill`) for cheap products < 7.00 USD and one for more expensive products
- Add a border to every circle (SVG property: `stroke`)

Get creative!

Loading external data

Most of the time, we will need to load an external data file. D3 provides functions to help us do that- for csv, json and other files:

`d3.csv`

`d3.json`

These functions load the data **asynchronously**. Behind the scenes, this asynchronous function call uses Promises, which you may be familiar with in other web development projects. It will work to load your data, and wait for it to be completed. Then it will run the callback function you apply using 'then'.

We need the data to be loaded asynchronously, because then the page is still active and not waiting on the data to load.

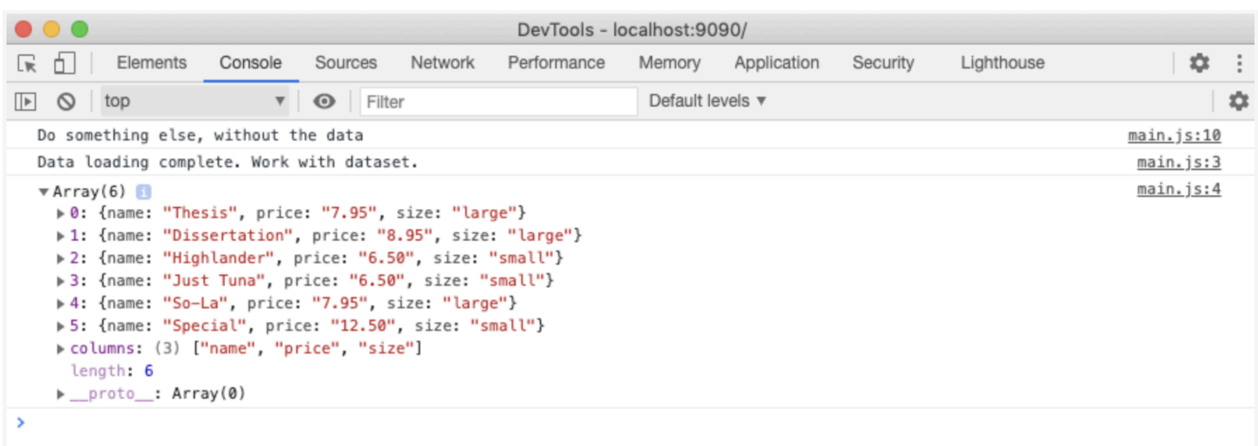
Here's an example:

```
d3.csv('data/sandwiches.csv')
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('Error loading the data');
  });
```

Once the data loads, then() is executed, and the callback function will be invoked. NOTE: Any operations that depend on the data need to be in the callback function, which is what is called when the data loads. For instance, the order of this code may be different than what you expect.

```
d3.csv('data/sandwiches.csv')
  .then(data => {
      console.log('Data loading complete. Work with dataset.');
    console.log(data);
  })
  .catch(error => {
    console.error('Error loading the data');
  });

console.log('Do something else, without the data');
```



Data processing:

Before drawing anything, you may need to process that data.

To convert Strings to int's, you can use parseInt or this simple approach:

```
d.age = +d.age;
```

Credits:

https://github.com/UBC-InfoVis/2021-436V-tutorials/tree/master/1_D3_Tutorial

Links to an external site.

https://www.dataviscourse.net/tutorials/lectures/lecture-d3/