Tutorial 1
Intro to D3

DATA VISUALISATION

Alfonso Bueno Orovio
DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF OXFORD | HILARY TERM 2023

Tutorial 1: Intro to D3

In this tutorial you will learn how to create your first D3 implementation. We will show you how to draw basic SVG shapes and how to bind it to data values. We will also reiterate advanced JS concepts: *method chaining*, *anonymous functions*, *asynchronous execution*, and *callbacks*.

1. D3 Project

D3 (Document-Driven-Data) is a powerful JS 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, Mike Bostock*

In this course, we will use the official D3 bundle that contains all the default modules to make things easier. Once you are more experienced, you don't need to always load the entire library and you can include only the parts that you actually use. We will use D3 version 7.

When looking up code examples online, be aware that many examples still use D3 version 3, which was a lot less modular than newer versions. The differences between version 4 and later versions are mostly minor, simplifying some of the concepts and introducing new features. You can look up the differences between major releases in the D3 docs.

D3 integration

Our D3 projects will have the structure of an HTML document (`index.html`) similar to the template code snippet shown below. In here, the `head` section includes the title of the project (to be shown in the tab of your web browser), a web reference to the D3 library, and a reference to an optional CSS file (`styles.css`) for the formatting of styles. The `body` section includes a `svg` object that will be our canvas for adding D3 objects, and a reference to one or several JS files (`index.js`) implementing the desired functionality. Defining these JS files as `type="module"` will simplify dividing your code into separate files, as we will see in future tutorials.

```
<!DOCTYPE html>
<html>
  <head>
    <title>D3 project</title>
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <svg width="960" height="500"></svg>
    <script src="index.js" type="module"></script>
  </body>
</html>
```

You should keep your own JS code separated from the JS libraries that 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, so make sure you encapsulate your own code into separate files (libraries). Make also sure to include libraries before using them in your code (order of *script* tags).

2. A brief overview of SVG

We will use D3 to bind data values to visual marks and channels on a web page. D3's default rendering platform is SVG (Scalable Vector Graphics) that we will use throughout this course. Here are some key facts about SVG:

- SVG is defined using markup code similar to HTML.
- SVG elements don't lose any quality when they are resized.
- SVG elements can be included *directly* within any HTML document or *dynamically* inserted into the DOM (Document Object Model) with JS.
- Before you can draw SVG elements, you have to add an `<svg>` element with a specific `width` and `height` to your HTML document, for example: `<svg width="960" height="500"></svg>`.
- The SVG coordinate system places the origin (0,0) in the top-left corner of the svg element. The X coordinate increases from left to right, while the Y coordinate grows from top to bottom.
- SVG has no layering concept. The order in which elements are coded determines their depth order.

The example below (see `'ex1.1_Basic_SVG\'`) shows how to include directly some SVG primitives within an HTML document. We will learn how to bind elements dynamically to our data using D3 in the next section.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Basic SVG</title>
  </head>
  <body>
    <svg width="400" height="50">
      <!-- Rectangle: x,y: coordinates of upper-left corner -->
      <rect x="0" y="0" width="50" height="50" fill="blue" />
      <!-- Circle: cx,cy: centre coordinates; r: radius -->
      <circle cx="85" cy="25" r="25" fill="green" />
      <!-- Ellipse: rx,ry: separate radius values -->
      <ellipse cx="145" cy="25" rx="15" ry="25" fill="purple" />
      <!-- Line: x1,y1 & x2,y2: coordinates of the ends of the line -->
      <line x1="185" y1="5" x2="230" y2="40" stroke="gray" stroke-width="5" />
      <!-- Text: x: position of left edge; y: vertical position of baseline -->
      <text x="260" y="25" fill="red">SVG Text</text>
    </svg>
  </body>
</html>
```

Output:

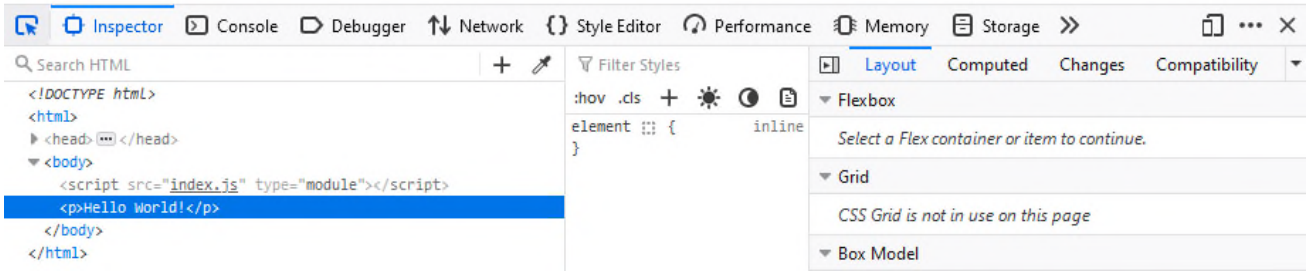
3. Adding DOM elements with D3

Before continuing, please explore the contents of the folder `'ex1.2_Hello_World'`. Its HTML document `index.html` includes the JS script `index.js`, which consists of a single line of JS code:

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

where we are using D3 to add a paragraph with the text "Hello World!" to a basic web page. All functions are in the d3 namespace, so we can access them by starting our statements with: `d3.`. If you open the Web Developer Tools of your browser after loading the page, you can verify this successfully creates *dynamically* the HTML paragraph `<p>Hello World!</p>` within the served document.

Hello World!



Before going into further details, we want to introduce the JS concept of *method chaining*. Method or function chaining is a common technique in JS and particularly useful 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.
- The output type of one method has to match the input type expected by the next method in the chain.

The alternative code of the example above without method chaining would be:

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

D3 Select

The D3 `select()` method uses CSS selectors as input to grab page elements. It will return a reference to the first element in the DOM that matches the selector. In our example we have used `d3.select('body')` to select the first DOM element that matches our CSS selector, body (as there is only one such element). Once an element is selected –and handed off to the next method in the chain– you can apply **operators**. These D3 operators allow you to get and set **properties**, **styles**, and **content** (and will again return the current selection).

Alternatively, if you need to select more than one element, use `selectAll()`, as we will see in later examples.

D3 Append

After selecting a specific element, we can apply an operator, such as `append('p')`. The `append()` operator adds a new element as the last child of the current selection. We specified 'p' as the input argument (as paragraphs are indicated by in HTML), so an empty paragraph is added to the end of the HTML body. The new paragraph is automatically selected for further operations. At the end, we use the `text()` operator to insert a string between the opening and closing tags of the current selection (`<p></p>`).

In summary, all methods together:

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

Your D3 statements can be much longer, so you should always put each operator on its own indented line (standard D3 notation).

4. Binding data to visual elements

Similar to our last example, in the folder `'ex1.3_Binding_Data'` we keep using HTML tags, but this time we append a new circle for each value in a given array, also used to set dynamically the radius of each circle:

```
const radii = [15, 25, 25, 10, 15];

d3.select('svg')
  .selectAll('circle')
  .data(radii)
  .enter()
  .append('circle')
    .attr('cx', (d,i) => 25 + i*60)
    .attr('cy', 25)
    .attr('r', d => d)
    .attr('fill','red');
```

Output:

1. `select('svg')`: Reference to the target container (now our SVG canvas in `index.html`, instead of body).
2. `selectAll('p')`: Selection representing the elements (circles) that we want to create. As no circles exist initially in our canvas, it creates an empty selection where we will add the new elements, one per each value in the array `radii`.
3. `data(radii)`: Loads the dataset `radii` (array of integers). The data could be also strings, objects, or other arrays. Each item of the array is assigned to one element of the current selection.
   Instead of returning just the regular selection, the `data()` operator returns three virtual selections:
   - **Enter** contains a new placeholder for any missing elements
   - **Update** contains existing elements bound to the data
   - **Exit** contains existing elements that are not bound to data anymore and should be removed

   There were no circle elements on the page, so the **enter** selection will contain placeholders for all elements in the array. In this and the following examples, we will concentrate only on the *enter* selection. You will learn more about the *enter-update-exit* sequence later when we will create interactive visualizations.
4. `enter()`: Creates new data-bound elements/placeholders.
5. `append('circle')`: Takes the empty placeholder selection and appends a circle to the DOM for each element.
6. `attr()`: Sets different attributes (centre, radius, colour) for each of the created circles.

Dynamic properties

If you want access to the corresponding values from the dataset, it is common to use *anonymous functions*. We are doing this when setting the radius of each circle (d is commonly used to refer to a single data value):

```
// Our preferred option: ES6 arrow function syntax
.attr('r', d => d)

// Alternative: Traditional function syntax
.attr('r', function(d) { return d; } )
```

In comparison, an ordinary JS function looks like the following code below. It has a function name, an input and an output variable. If the function name is missing, then it is called an *anonymous function*.

```
function doSomething(d) {
    return d;
}
```

Anonymous functions are still regular functions, so they don't have to be a simple return statement. We can use if-statements, for-loops, console message, or we can also access the index of the current element in our selection. The latter is done in D3 by exploiting anonymous functions with two parameters `(d,i)`, as used in our code to set the x coordinate for the centre of each circle (shifting each one by 60 pixels to the right). The index `i` is passed as a second parameter in function calls and is optional.

This and the following examples in this tutorial are not intended to be a best practice example of how to work with D3 scales. They are designed to help you to get a better understanding of different basic concepts in D3 (especially selections). In later tutorials, you will learn how to create real scales for different data types, you will work with more flexible size measurements and you will learn how to use D3 axes in your visualisations.

HTML attributes and CSS properties

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.

- In our previous example, we have used D3 to set positions of the centres, sizes and colour of the circles. By using the Properties Inspector of your browser's Web Developers Tools, you can check that the SVG canvas now contains the definition of the different circles with the appropriate HTML attributes.
- If you want to assign specific styles to the whole selection (e.g., same colour for all circles, as in this case), it is recommended to add these rules to an external CSS file. Using classes and CSS styles will make your code more concise and reusable. We will explore this in our next example.

5. Loading external data and asynchronous execution

Instead of typing the data in a local variable, which is only convenient for very small datasets, we can load data *asynchronously* from external files. The D3 built-in methods make it easy to load JSON, CSV, and other files.

We will exemplify this (see folder `'ex1.4_Loading_External_Data'`) by extending our previous example to load the items of a menu, adapt the size of the circles to provide a prize indication (items cheaper than £10 to be displayed in a smaller size), and to highlight those suitable for vegetarians (in green). The file `menu.csv` includes the relevant information on the different items. This looks like:

```
item,prize,vegetarian
Soup of the day,6.95,yes
Roast chicken,12.95,no
Caesar salad,8.95,no
Fried augerbines,11.95,yes
Crispy duck,14.95,no
Expresso,2.49,yes
English tea,2.49,yes
```

By calling D3 methods, such as `d3.csv()`, `d3.json()`, or `d3.tsv()`, we can load external data resources in the browser. These functions take the file path as an argument and load the data asynchronously. Once the data is loaded and the associated Promise is resolved[1], you can work with the data:

[1] A Promise is an object representing the eventual completion or failure of an asynchronous operation.

```
d3.csv('menu.csv')
  .then(data => {                          // data loading
    data.forEach(d => {
      d.prize = +d.prize;                  // data parsing
    });
    render(data);                          // data rendering (calls 'render')
  })
  .catch(error => {
    console.error('Error loading the data');
  });
```

1. Reading files from disk or external servers can take a while. Here, `d3.csv()` uses asynchronous execution: we don't have to wait and stall, instead we could proceed with further tasks that do not rely on the dataset. After receiving a notification that the data loading is complete, the callback function `then()` is executed.
2. Each value of a CSV file is stored as a string. Numerical values need to be converted to numbers to avoid unexpected behaviour. For data parsing, we recommend iterating over all the elements in the dataset as shown in the code snippet. Putting a "+" in front of a variable converts that variable to a number[2].
3. **Code that depends on the dataset should generally exist only in the `then()` callback function.** You could structure your code in separate functions; however, if these functions depend on the dataset, they should only be called inside the callback function. Here, after loading the data, we invoke our function `render()`:

```
const render = data => {
  d3.select('svg')
    .selectAll('circle')                   // select all existing circles (none)
    .data(data).enter()                    // create data join
    .append('circle')                      // append one circle per element
      .attr('cx', (d,i) => 25 + i*60)
      .attr('cy', 25)
      .attr('r', d => (d.prize < 10) ? 15 : 25)[3]
      .attr('fill', d => (d.vegetarian == 'yes') ? 'green' : 'yellow');
};
```

Output:

Note that all our circles also display some extra properties (a black border -stroke- of 1 px width) we haven't encoded directly. As these apply to all circles, they have been set by using the external CSS file `styles.css`:

```
circle {
  stroke: black;
  stroke-width: 1px;
}
```

Any property or style specified in an external CSS file will override any properties or styles that you may have previously hardcoded in JS.

[2] You could also use `parseInt()` or `parseFloat()`.
[3] The conditional JS operator (`condition ? exprIfTrue : exprIfFalse`) is equivalent to an if-else statement.