F20-21DV Data Visualisation and Analytics

Lab 1 – Introduction to D3

Preamble

These labs are guided self-study instructions on the practical aspects of the Data Visualisation and Analytics course. While you will work on these during your timetabled lab sessions, where you will get the help of lab helpers, **you must also practise between classes**. There are 4 labs in total.

Each lab is a mixture of:

- **Tutorials**, introductions and guidelines to D3 (and JavaScript) concepts; and
- **Exercises**, guided instructions on how to apply these concepts, often with a goal.

Tutorials may seem abstract until you get to practise their content in exercises. This is normal. Hence you might want to read them *at least* twice: before you get to the related exercises, to get some context; and after the exercises, to make sure you understand what you did.

We won't be able to cover everything in detail. However, each section provides links to relevant documentation or examples for you to go further. You are strongly encouraged to read these to (a) get an idea of the scope of what you can do, and (b) to inspire your creativity for the coursework and help you demonstrate your mastery of the topic.

It can be tempting to simply skip most of the text and jump to code snippets and copy paste them in your projects and expect great results …

1. Not reading the tutorials and text accompanying exercises means you don't get the context of these code snippets and skip the part where you learn things.
2. Most of the code snippets are succinct and focus on one thing. You are required to understand where it fits in the overall project, and sometimes, fill in the blanks that were left out purposefully.

Make sure you read through this document in its entirety.

In this first lab, we will focus on two points:

1. (Re)Introduce you to **web programming** while setting up your development environment for this course; and
2. Introduce the core concepts to D3 which can help you build your first charts:
   a. **Selections**
   b. **Scales** (and axes)
   c. **Shapes**

Tutorial 1 – Web Languages

In this course, we will develop interactive data visualisations for the web. As such, it is important that you have a basic understanding of what is meant by web development.

There are two sides to web development:

- The *back-end*, or **server-side**, to develop APIs for HTTP requests; and
- The *front-end*, or **client-side**, that focuses on what on the website content (responses to HTTP requests) that your browser loads and displays.

We will exclusively work on the second side in this course. While there are many (***many***) frameworks and language supersets for front-end programming, there are fundamentally four languages that you need to know. On top of these, we will use one library to link our web pages and graphics with datasets.

If you already have experience in web programming, you can skip this first section (or have a quick read to refresh your memory).

If you have little or no experience in web programming, you will find an extra Lab sheet on Canvas with links to online tutorials.

HTML, CSS and JavaScript

These three languages are the basis of all websites.

Hyper-Text Markup Language, or HTML for short, is your web page. With a hierarchy of nested ***tags***, an HTML file defines your page's structure and content:

- There is the ***head***, which defines the page metadata:
  - The ***title*** of the page
  - The dependencies needed
- There is the ***body*** of the page, which contains:
  - A ***header***, with:
    - A ***level 1 heading***
  - A ***main*** section, with:
    - A ***level 2 heading***
    - A ***section***, with:
      - A ***paragraph*** with some text
      - Another ***paragraph*** with more text
      - A ***link***
    - …
  - A ***footer***, with:
    - A ***paragraph***

Etc.

This structure constitutes the **Document Object Model**, or DOM, of the page.

Cascading Style Sheets, or CSS for short, is for styling the element of your web page. You do so by first writing CSS **selectors** that will query a set of elements from the page, then you can write rules for these elements: what font to use, which background colour, how are their content positioned, etc. While initially very static, CSS has evolved to let you apply dynamic and responsive styles as well.

Finally, ***JavaScript***, or JS for short, is where you can program any logic behind your web page: store data, write objects and functions, and make the web page dynamic by accessing and mutating its elements (or **nodes**). This last part has been the main purpose of JavaScript since its inception, and as the capabilities of browsers evolved over the years, many JS frameworks have been created to support modern web application development.

In this course, we will be using Vanilla JavaScript, i.e., JavaScript with no framework or language superset, only native functionalities. You should also note that, despite what the name may suggest, JavaScript as nothing to do with Java, at least nothing more than a marketing stunt to promote it...

This video by Fireship provides a good overview of the history of JavaScript, up until 2019: https://www.youtube.com/watch?v=Sh6lK57Cuk4

At the end of this lab, you will find an appendix that summarises the main features of these languages. Make sure to refer to it as you develop your project.

SVG

While HTML deals with text-based content (including user input elements), ***Scalable Vector Graphics***, or SVG for short, allows us to describe the structure and content of graphical elements: polygons, lines, circles, rectangles, etc. Essentially what we need for visualisations.

Importantly, SVG is written in a Markup Language format, just like HTML, which means that we can:

- embed it in our web pages;
- use CSS selectors to style its elements; and
- use JavaScript to access and modify it.

You can check a comprehensive documentation for SVG on the MDN website: https://developer.mozilla.org/en-US/docs/Web/SVG

SVG Elements

All SVG graphics are defined with the following elements:

- **`svg`** is always the top-level element for the graphic. It has width ad height attributes that define the viewport: child elements drawn outside this viewport will not be visible on the page!
- **`g`** is an aggregator element (like div in HTML), it stands for group. It's useful when applying transformations (e.g., translation, scaling, rotation) to a set of elements.
- Basic shapes:
  - **`circle`**, with attributes such as **`cx`**, **`cy`** and **`r`** (centre x, centre y and radius).
  - **`rect`**, with **`x`**, **`y`**, **`width`** and **`height`**. x and y define the top left-hand corner of the rectangle.
  - **`line`**, with **`x1`**, **`y1`**, **`x2`** and **`y2`** (the two extremities of the line segment).
- Advanced shapes:
  - **`polyline`**, a series of straight line defined by a **`points`** attribute.
  - **`polygon`**, a closed shape drawn from connected straight lines defined by a **`points`** attribute.
  - **`path`**, a complex element that can define any shape from a draw (**`d`**) string.

- **`text`** lets us add floating lines of text in the SVG.

Note that the SVG coordinate system places the top left-hand corner as the origin. The X axis goes horizontally from left to right, and the Y axis goes vertically from top to bottom.

D3

D3 will be the main focus of this course (on the practical side of things). It's a library specifically designed for building interactive data visualisation on webpages.

It was created in 2011 by Mike Bostock (in collaboration with Jeffrey Heer and Vadim Ogievetsky) from the Stanford Visualization Group, and published in an article of one of the top data visualisation journals:

> Bostock, M., Ogievetsky, V. and Heer, J., 2011. D³ data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12), pp.2301-2309.

Since then, it has gained immense popularity in the data visualisation community and gone through 192 iterations. As of November 2023, it reached version 7.8.5 and is still an active project on GitHub.

Its success and popularity are based on three factors:

1. It came at the right place at the right time. In 2011, web browsers were getting more and more powerful, allowing significant processing to take place directly in the browser and the emergence of more advanced web applications. Simultaneously, they started integrating more web development and debugging tools.
2. From its foundation, D3 has been a framework-agnostic project, working in pure JavaScript <u>and</u> updating itself to align with the latest language specifications.
3. It's free and open source.

But what does it do anyway? Here is a quote from the 2011 paper:

> *"Rather than hide the underlying scenegraph within a toolkit-specific abstraction, **D3 enables direct inspection and manipulation of a native representation**: the standard document object model (DOM). With D3, designers selectively **bind input data to arbitrary document elements**, applying dynamic transforms to both **generate and modify content**."*

Translation: D3 is all about directly manipulating the DOM of webpages based on data. It produces **<u>D</u>**ata-**<u>D</u>**riven **<u>D</u>**ocuments. Importantly, it's a low-level declarative language, meaning that there are no pre-built visualisations. Instead, you can build your own, highly customised, interactive data visualisations by instructing the program to modify the DOM according to the data.

As of version 4 (around the time that JavaScript introduced modules with **`import`** and **`export`** statements), D3 comes as a collection of independent modules:

- For manipulating the DOM;
- For loading and transforming data;
- For creating complex visualisation components; and
- For implementing advanced interactions and animations.

You can find the D3 documentation on their website: https://d3js.org/what-is-d3

You can also find many examples on the Observable community: https://observablehq.com/@d3. Some words of caution:

- The examples shown on Observable are for the Observable platform, which does not behave like normal web pages. The use of D3 functions can still be adapted though.
- There are many examples of D3 online, but not all of them target the version we are working with (version 7). As with any online resources, don't simply copy/paste things and expect them to work with no side effects. **Be critical and smart**.

Exercise – Project Setup

In this "code-along" exercise, we will go through setting up a basic project structure that you can then reuse for other labs and your coursework. We will use this opportunity to also revisit some important web programming features that you will need for the course.

Development Environment

Installations

We will start by creating the basic development environment for your labs and coursework. All you really need are three things:

- An up-to-date web browser. Hopefully you have one of these already. Chrome and Firefox are good. Safari can be hard for debugging JavaScript.
- A code editor. We recommend you use VSCode.
- A program for launching local servers (more on that later).

First, it is recommended that you download and install VSCode. This is a lightweight editor that comes with plenty of functionalities and extensions for web development (amongst other things, there are also official extensions for Python and Jupyter notebooks for example). You can download it from the website: https://code.visualstudio.com/Download. Note that this is **not** Microsoft's Visual Studio IDE! Visual Studio and VSCode are two separate things.

In your system, create a project root directory, that you can name **`F20-21DV_code`** for instance. Open this directory in VSCode:

- Launch VSCode;
- Click on File > Open Folder;
- Selecting your project directory in the prompt.

Then, open the Extension panel (by clicking on the icon on the left, or using *Ctrl+Shift+X*) and search for the **Live Server** extension by Ritwick Dey (it should be the most popular option) and install it. You should then see a **Go Live** option at the bottom right of the editor. Clicking on it should open your browser, at the URL: *127.0.0.1:5500*.

This is called a **local HTTP Server**, which mimics a real system that serves files over an HTTP protocol. Check this page for more details on why this is important when building and testing websites.

Alternatives

As an alternative, if you have another preferred development environment, feel free to use it. Make sure that it can provide a local server. If not, you can use the following python command in a terminal:

```
$ python -m http.server
```

Of course, this requires you to have python installed. Your test website will then be visible at the URL `localhost:8000`. You can use other local servers such as those offered by Node.js or PHP.

Note: Lab helpers might not be able to offer support with these alternatives.

First Steps

Next, we will create the application. Under the project root file, create the following:

- an **`index.html`** file, this would be the entry point of our application;
- a **`scripts`** directory, where we will add all the JavaScript files of our application, we can start with a **`main.js`** file; and
- a **`styles`** directory, where we will add all the CSS files of our application, we can start with a **`main.css`** file.

You should now have the following structure:

- F20-21DV_code
    - index.html
    - scripts
        - main.js
    - styles
        - main.css

Testing the Environment

Let's add a few things to our HTML file to check everything works. A basic HTML document structure is as follows:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=1920, initial-scale=1.0">
    <title>Page Title</title>

    <!-- Libraries and stylesheets imports -->
</head>
<body>
    <!-- Page body -->
</body>
</html>
```

This defines the DOM of the HTML page, i.e., the hierarchy of elements on the page. Let's add a few elements to the **`body`**:

- a heading (**`h1`**) with any content;
- a **`div`** element, containing 3 paragraphs (**`p`**), each with the content you wish to add.

With these elements added, the DOM now has an **`h1`** element, with a **`div`** sibling which has three **`p`** children.

Now, check your website in the browser. You should see the text that you have just added to the page. If not, make sure you reload the page with a clear cache (*Shift+F5* or *Ctrl+Shift+R*).

Next, open the **Web Console** by either right-clicking on the page and selecting *inspect* or typing *Ctrl+Shift+I*. Explore the web console to see what options it offers.

Styling the Page

Now, we will work with CSS to add styles to our HTML elements.

The first thing to do is load our stylesheet **`main.css`** onto the HTML page ("…" refers to other parts of the code left out):

```
...
<head>
    ...

    <!-- Libraries and stylesheets imports -->
    <link rel="stylesheet" href="styles/main.css">
</head>
...
```

CSS Selectors

In CSS, we can add style rules to specific elements by first *selecting* them and then writing the rules that apply to them. For example, if we add the following to **`styles/main.css`**, every **`h1`** elements will be displayed in green:

```
h1{
    color: green;
}
```

With *classes* and *ids*, we can refine our element selection. Let's add a class to one of the paragraphs and an id to another:

```
<p>Paragraph 1</p>
<p class="myClass">Paragraph 2</p>
<p id="myId">Paragraph 3</p>
```

Note that ids must be unique, only one element can have a specific id. Classes however can be shared by several elements.

In the CSS file, that means we can add several specific rules for paragraphs, using **`.`** to indicate a class and **`#`** to indicate an id:

```
p{
    color: tomato;
}
p.myClass{
    color: teal;
}
p#myId{
    color: goldenrod;
}
```

Now, while the default for paragraphs is to be in red, specific rules apply to the paragraphs with the class **`myClass`** and to the one paragraph with id **`myId`**.

You can read this page for more details about CSS Selectors.

CSS Variables

Sometimes we might need to reuse several CSS values for different elements of our DOM. For this, we can use *variables*. A variable can be defined either in the scope of a specific DOM element or globally as part of the **`:root`** element.

Let's add a file in our styles directory dedicated to storing these global variables, **`vars.css`** for example. In it, we will select the **`:root`** element and declare colour variables within it:

```
:root{
    --main-color: #222222;
    --accent-color: #003C71;
}
```

We then need to *import* this file into our **`main.css`** stylesheet to be able to use these variables:

```
@import 'vars.css';

h1{
    color: var(--main-color);
}

p{
    color: var(--main-color);
}
p.myClass{
    color: var(--accent-color);
}
```

Variables don't just apply to colours, but any CSS properties. For example, fonts are another typical example of properties you might need to reuse several times. In the example below, fonts are imported from Google Fonts and then declared as variables:

```
@import
url('https://fonts.googleapis.com/css2?family=Open+Sans:ital,wght@0,400;0,700;1,400;1,700&display=swap');
@import
url('https://fonts.googleapis.com/css2?family=DM+Serif+Display:ital@0;1&display=swap');

:root{
    ...
    --font-text: 'Open Sans', sans-serif;
    --font-title: 'DM Serif Display', serif;
}
```

We can then use those where we want in **`main.css`**:

```
...
h1{
    color: var(--main-color);
    font-family: var(--font-title);
}

p{
    color: var(--main-color);
    font-family: var(--font-text);
}
...
```

By opening the *Web Console* to the *Elements* tab, you can select elements of the DOM, inspect the styles applied to those and even preview style changes.

Using *imports* can be quite advantageous. For example, you might have several stylesheets, each dedicated to one aspect of your page (text, layout, input, etc.) or later, one per visualisation. You can then centralise them all into the **`main.css`** stylesheet. Note that, just like when loading several stylesheets into an HTML document, the order in which you import several stylesheets matters.

JavaScript

Now let's dive into the core programming language of this module: JavaScript. First, let’s load **`main.js`** into the HTML page:

```
<body>
    ...

    <script type="module" src="scripts/main.js"></script>
</body>
```

The **`type="module"`** will become important later when we deal with JavaScript modules. Note that we are also loading this script at the end of our DOM, again this is something that will be important later, when part of the script will require some DOM elements to be created before it starts. Inside `main.js` we will add the following sample code:

```
'use strict'; // to prevent us from overwriting important variables

const c = 'constant'; // a constant value, assignment to c is no longer
allowed

let v = 'variable'; // a primitive variable

let a = [1, 2, 3, false]; // an array

let o = { // an object
    'key1': 1,
    'key2': 'something'
};

console.log(c);
console.log(v);
console.log(a);
console.log(o);
console.log(o['key1']);
console.log(o.key2);
```

Open the *Web Console* to the *Console* tab to inspect the log operations.

Functions

There are several ways to declare functions in JavaScript:

```
function one(a, b){
    return a+b;
}
let two = function(a, b){
    return a+b;
}
let three = (a,b)=>a+b;
```

The last option is called an *arrow function*, which acts similarly to a lambda expression. One key difference with normal functions is that arrow functions don't have a scope of their own.

Note that in JavaScript, functions are variables too. As such, writing **`o['key3'] = two;`** is valid, and means that **`o.key3`** now points to function **`two`** (check the Web Console to verify this!).

To call a function, one simply uses it and gives it parameters: **`one(4,5)`**. Note the difference between using a function reference **`myFunc`** and calling a function **`myFunc()`**.

You can also specify default values for the parameters, for example, **`function one(a, b=6){…}`**.

Modules

Modules are a way for us to handle complexity in JavaScript applications. Put simply, a module is a JavaScript file that defines variables accessible to other modules.

For example, let's create a file in the **`scripts`** folder for Mathematical functions: **`math.js`**. This file will have two functions:

```
export function GCD(a, b){
    if(b===0) return a;
    return GCD(b, a%b);
}

export function factorial(n){
    if(n===0) return 1;
    return n * factorial(n-1);
}
```

By prepending the keyword **`export`** in front of both of these functions, we are making this file a module. In turn, in **`main.js`**, we can import these functions like so:

```
import {GCD} from './math.js';

console.log(GCD(84, 52));
```

We may even rename these imports:

```
import {factorial as myFactorial} from './math.js';

console.log(myFactorial(5));
```

Classes

JavaScript implements Object-Oriented features, which means that we can define classes. Create a new file in the **`scripts`** folder, called **`Book.js`**. In it we will create a class representing a book:

- A book has a title and author.
- A book has a private status for being *onLoan* (boolean flag).
- A book has methods to loan it (mark the *onLoan* status to *true*), return it (mark the *onLoan* status to *false*) and check if it's available (*onLoan* is *false*).

Use the *Web Language Guide* appendix below to help you write this class or check the MDN Website.

When your class is ready, prepend the keywords **`export default`** in front of **`class`** to turn it into a module. **`default`** means that the class is the only thing exported, and as such, we can import it as follow in **`main.js`**:

```
import Book from './Book.js';
```

You can now instantiate several books (with the **`new`** keyword), save them in a variable, and print them to the Web Console for inspection.

As your project progresses, you might find useful to bundle the various visualisation you create into classes. That way, you will be able to reuse them multiple times, without having to copy/paste and maintain several versions.

Tutorial 2 – D3 Selections

With the principles of web development in out of the way, we can now focus on the main challenge for this course: making data visualisations with D3.

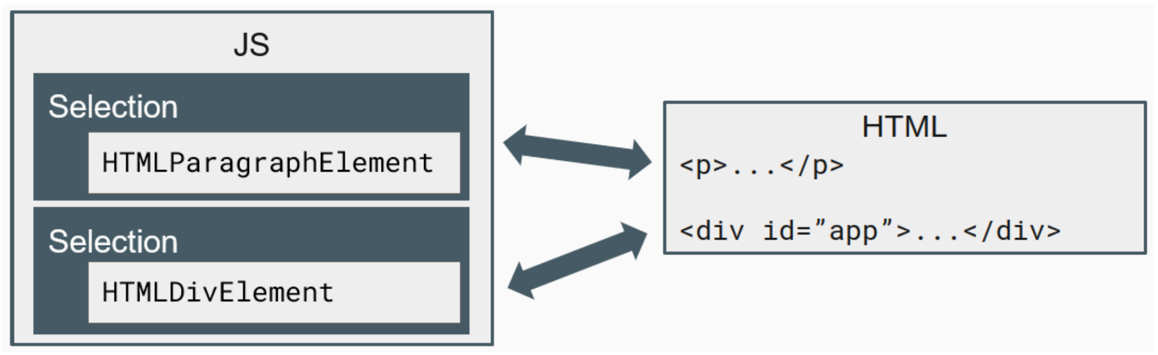
This tutorial may appear abstract until you practice it in the exercise. That's fine: have a first read, do the exercise and come back to make sure you have understood what happens in your code.

As we saw in Tutorial 1, the structure of a web page is defined in what is called the Document Object Model (DOM), as a collection of nested elements:

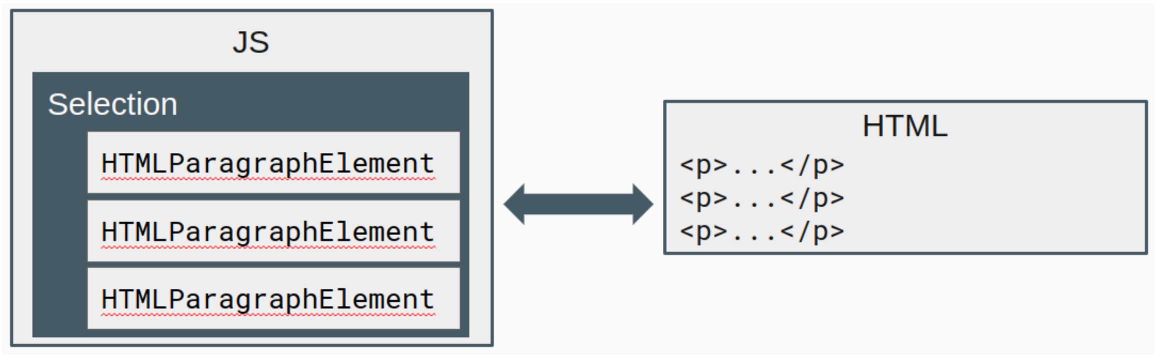
- In HTML, they are described as **tags**: `<p>…</p>` or `<div id="myId">…</div>`;
- In CSS, we can access them with **selectors**: `div#myId{…}`;
- In JavaScript, they are stored as **node** Objects, with dedicated types: `HTMLParagraphElement` or `HTMLDivElement`.

Selections

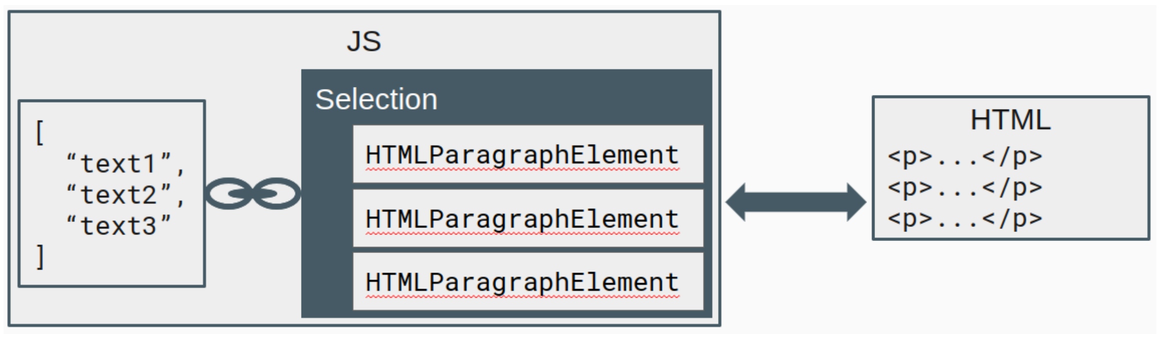
D3 provides a new type of Object, `Selection`, that acts as a wrapper around DOM nodes in JavaScript:



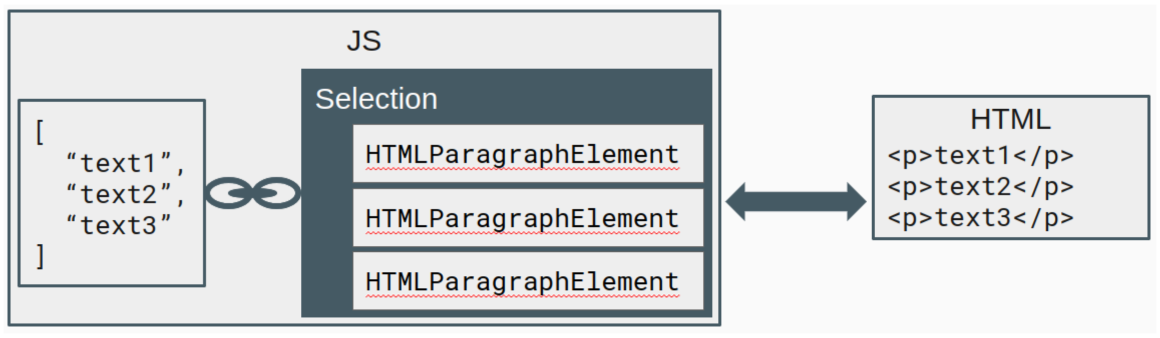
More interestingly, one Selection can contain several nodes of the same level and type (i.e. sibling nodes):



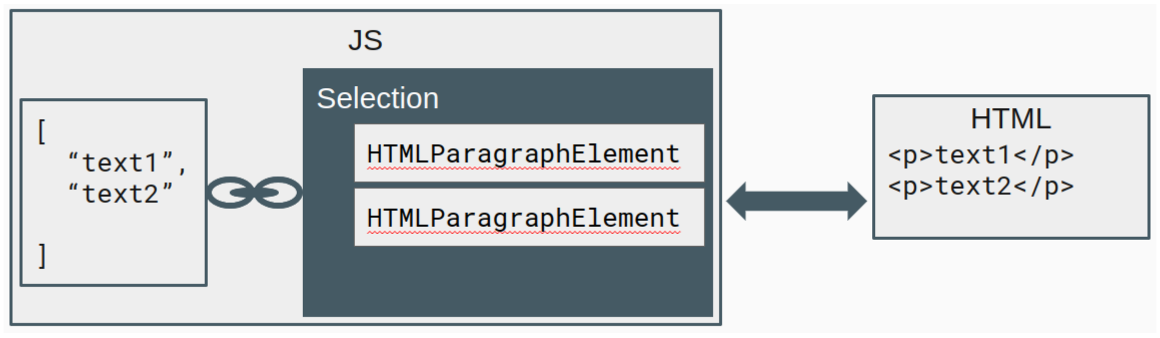
One important feature of Selections is their ability to bind data to DOM elements:

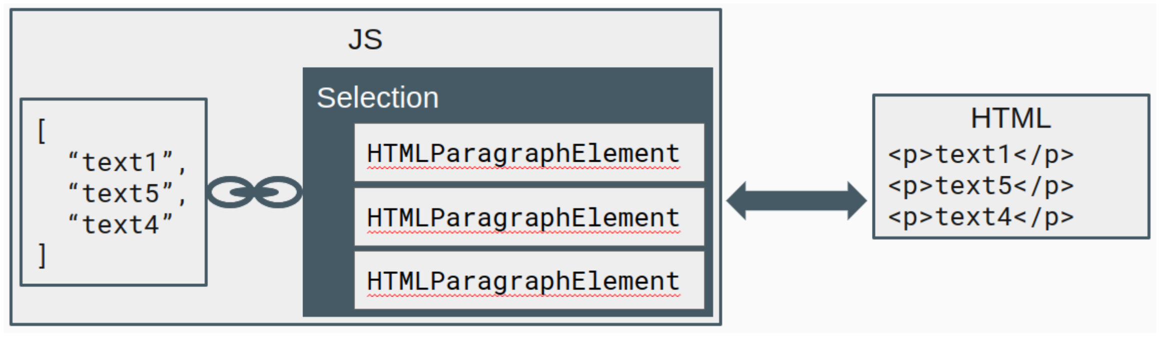


Which means we can then modify these DOM elements based on the data that has been bound to them:



This includes dynamically adding, removing and updating the DOM as the data bound to it changes:





Creating and Using D3 Selections

We can start working with D3 selections by first instantiating one:

- **`d3.select(selector)`** will return a selection with the first existing matching element.

- **`d3.selectAll(`*`selector`*`)`** will return a selection with all existing matching elements.

Note that in both cases, we use CSS selectors to match element, as they provide a concise way to find such elements.

If nothing matches, an *empty* selection is created. The selection is not null and will contain methods that we can use, but at this stage, it doesn't point to any DOM node.

Then, we can chain selections to access descendants of the DOM node, for example, **`mySelection.selectAll(`*`childSelector`*`)`**.

The command **`mySelection.append(`*`tag`*`)`** lets you add a direct child to each element of **`mySelection`**, and returns these children as a new selection.

Here is an example of the type of code you can write in your JavaScript files:

```javascript
// make app the top-level selection
let app = d3.select('div#app');

// add a section to app
let sec1 = app.append('div');

// create empty selection for paragraphs
let pars = sec1.selectAll('p');
```

With a selection now instantiated, we can bind data to it: **`mySelection.data(myData, key)`** is the method used for binding the **`myData`** to elements of **`mySelection`**. The **`key`** parameter can be added to provide an accessor used for uniquely matching data items to DOM elements. By default, this key function uses data indices, but accessors can be used to declare how to reach parts of the data items and use these parts for matching, e.g., **`d=>d.k`** means "use the attribute k of each data item (d)".

Once DOM nodes are bound to data entries, we have four cases:

		Data entry present?	
		Yes	*No*
DOM node exists?	*Yes*	We need to ***update*** the DOM node	We need to ***remove*** the DOM node
	No	We need to ***create*** a new DOM node	*We don't need to do anything*

The function **`mySelection.join(...)`** gives us access to these three (temporary) sub-selections:

- The ***enter*** selection, for elements to add
- The ***update*** selection, for elements to change
- The ***exit*** selection, for elements to remove

Inside the function's parameter, we can provide a function for each of these sub-selections, allowing us to specify how we want to see them being created, updated or removed. Then, the function returns the full updated selection, corresponding to a merge of the enter and update sub-selections.

For example, the following code:

1. Binds the data to paragraphs, including a key function that uses the attribute **`k`** of each data item
2. Joins the enter selection by appending new paragraphs elements
3. Joins the update selection by doing nothing
4. Joins the exit selection by removing relevant paragraphs elements
5. Finally, it changes the text value of the new selection of paragraphs (enter+update) to the attribute **`v`** of their associated data entry.

```
let myData = [{k:1,v:'text1'},
              {k:2,v:'text2'},
              {k:3,v:'text3'}];

pars.data(myData, d=>d.k)
    .join(
        enter=>enter.append('p'),
        update=>update,
        exit=>exit.remove()
    )
    .text(d=>d.v);
```

This behaviour of adding missing elements and removing extra ones is a very common thing to do. As such, the join method has a shorthand, **`mySelection.join(`*`tag`*`)`**:

```
let myData = [{k:1,v:'text1'},
              {k:2,v:'text2'},
              {k:3,v:'text3'}];

pars.data(myData, d=>d.k)
    .join('p')
    .text(d=>d.v);
```

Beyond this, as we saw with the **`.text()`** method, D3 selections provide many useful methods to update the DOM elements with functions that will query the data. Most notably, the **`.attr()`** method which lets you edit attributes of the DOM element, and the **`.style()`** method which allows you to change its CSS styles.

As you progress, you should check the D3 documentation for more details: https://d3js.org/d3-selection.

Exercise – Let's Make a Bar Chart

In this exercise, we will get started on building data visualisations. "*Let's Make a Bar Chart*" has been a famous first tutorial to learn D3 since the library's first versions, written by Mike Bostock himself first, and adapted and updated by many others.

Bar charts are simple to understand, which should make it easy for you to understand what we will be doing. Despite this, building a bar chart lets us explore many core principles of D3!

The first step is to add the D3 library to your application. In the **`head`** of your **`index.html`** file, add this line:

```
<head>
    ...
    <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
```

This loads a minified script from D3's website directly, corresponding to the library's latest version. Because it loads the script live via your internet connection, this might not be an ideal solution going forward. Instead, what you can do is:

- Download the **`d3.v7.min.js`** file, (using the URL in the code example);
- Save it in your project directory under a descriptive folder, e.g., **`lib`**;
- Replace the **`src`** attribute in the code above to use your local file, e.g., **`"lib/d3.v7.min.js"`**.

To verify that D3 is correctly loaded, add the following code to your **`index.html body`**, make sure that your local server is running, and inspect the page's console output in a browser.

```
<body>
    ...
    <script>
        console.log('d3 version: ', d3.version);
    </script>
</body>
```

Note, that in the code above, we are not loading a separate script. Instead, we are writing inline JavaScript, inside a **`script`** tag. For clarity, I would suggest you prefer writing JavaScript in separate **`.js`** file, here we just need to do a quick check.

Building the App Skeleton

Before we can add any kind of visualisation to our web page, we must make the necessary preparations. Essentially, we need to ask ourselves: *where will the visualisations go?*

We could simply append them to the **`body`** element of the DOM. However, to help us with our dashboard design later, it is best to create one dedicated **`div`** element for each visualisation.

For now, we should add one **div** element for a bar chart (you can remove the **h1** and **p** tags from the previous exercise). To identify it, we should also give it an **id**. As you progress through these labs and the coursework, make sure to add a **div** for each visualisation you want to see in your dashboard.

```
...
<body>

    <div id="bar1"></div>

    <script type="module" src="scripts/main.js"></script>

</body>
```

Why are we loading our main script last?

We briefly touched on that before. JavaScript is a *just-in-time* compiled language: as soon as it gets loaded, it gets executed. Since we will be selecting elements of our page in the **main** script, it's best to ensure they exist first. It's the same reason why we import all third-party libraries before loading our own scripts.

Now let's turn to **scripts/main.js**, we will start with a clean state, erase everything from the previous exercise, except for the first line:

```
'use strict';
```

Our first step is to make a top-level D3 selection, from which we will select and/or append child selections for our bar chart components. To make a selection, we will use the **d3.select()** method, passing the appropriate selector as a parameter:

```
let barContainer = d3.select('div#bar1');
```

We can then use **barContainer** as an access point to add an **svg** child element. This can be done with the **.append()** selection method, which also returns the new child selection:

```
let barSvg = barContainer.append('svg');
```

Check the Web Console's Elements tab to see the created **svg** element.

We can now use this selection to manipulate the DOM and add attributes to our **svg**, namely, a **width**, **height** and **class**. This is done using the **.attr()** selection method. This method returns the current selection (unless it's used to get the values of attributes). This means we can chain methods in our code:

```
let barSvg = barContainer.append('svg')
    .attr('width', 800)
    .attr('height', 500)
    .attr('class', 'barchart');
```

Check the Web Console to inspect the **`svg`** and its updated attributes.

Note that the **`.attr()`** method completely overrides the attribute value. Hence, beware of accidentally removing classes when you intend to merely add one. To prevent that, you should use the **`.classed()`** method, which lets you toggle classes:

```
let barSvg = barContainer.append('svg')
    .attr('width', 800)
    .attr('height', 500)
    .classed('barchart', true);
```

We have now set up the necessary elements to get a bar chart going. You can repeat this process for most other charts.

Binding Data and Joining Elements

With the **`svg`** elements and selections in place, we can start drawing charts.

First, let's have some data. We will be using citie's population and area data taken from Wikipedia:

```
let cities = [
    {city: 'Edinburgh', pop: 506000, area: 119, alt: 47},
    {city: 'Dubai', pop: 3604000, area: 1610, alt: 5},
    {city: 'Putrajaya', pop: 109000, area: 49, alt: 38},
    {city: 'Qingdao', pop: 10071000, area: 11228, alt: 25},
    {city: 'Lagos', pop: 8048000, area: 1171, alt: 41},
    {city: 'Ottawa', pop: 1017000, area: 2790, alt: 70}
]
```

Each entry is a record with 4 attributes: **`city`** is the name of the city, **`pop`** is the approximate population size, **`area`** is the area in square kilometres, and **`alt`** is the average elevation in metres.

We will draw a bar chart that displays the area per city. Hence, we need a bar for each city, with its height proportional to the city's area.

SVG allows the following shapes:

- **`rect`** - Rectangles with one origin and two dimensions
- **`circle`** - Circles with one origin and a radius
- **`ellipse`** - Ellipses with one origin and two radii
- **`line`** - Straight lines with two extremity points
- **`polyline`** - Compound straight lines with several points
- **`polygon`** - Similar to **`polyline`** but also ensure the last point connects to the first

- **`path`** - A complex element to let you draw any shape

Looking at this list, **`rect`** is the obvious candidate for us to draw bars. We will first create a group element (**`g`**) that will contain bars, and then make a selection of all rectangles in that group:

```
let barGroup = barSvg.append('g');
let bars = barGroup.selectAll('rect');
```

At this stage though, the selection is *empty*. There are no **`rect`** elements in the DOM yet, and we haven't created any. What we need to do is **bind** our data set to this selection, using the **`.data()`** method, so that we can create and manipulate the **`rect`** element based on the data values:

```
let bars = barGroup.selectAll('rect')
    .data(cities, d=>d.city);
```

Note the **`d=>d.city`** key function, providing an accessor to uniquely identify data entries in the selection.

D3 uses **accessor functions** a lot. An accessor function is typically used as parameter of other functions. It's a callback that should be used to help the method access particular values in data entries. Typically, D3 accessor functions have two parameters: the datum and its index **`(d,i)=>{...}`**, but here we only need the datum.

Notice also how we are calling **`.data()`** directly after **`.select()`**. In most cases, when using one of D3's methods, the method will return the object it was called from, allowing us to chain methods.

With our selection now bound to data, D3 would have internally created *enter*, *update* and *exit* sub-selections to help us add new **`rect`** elements needed, change old ones and remove old unnecessary ones. We can access those with the **`.join()`** method:

```
let bars = barGroup.selectAll('rect')
    .data(cities, d=>d.city)
    .join(
        enter => ... ,
        update => ... ,
        exit => ...
    );
```

Accessing each sub-selection individually lets us customise how we want them to behave and be displayed upon creation or removal, for example with animations. For now, however, we will be using the default behaviour: simply add new elements and remove old unnecessary ones. For this we can use a shortcut:

```
let bars = barGroup.selectAll('rect')
```

```
    .data(cities, d=>d.city)
    .join('rect');
```

This is equivalent to:

```
let bars = barGroup.selectAll('rect')
    .data(cities, d=>d.city)
    .join(
        enter => enter.append('rect') , // create new elements
        update => update ,              // don't do anything
        exit => exit.remove()           // remove elements
);
```

Inspect the Web Console (Elements tab) to see how it has affected the page.

With the rectangles created, our next step is to set their attributes in accordance with the data. Four attributes matter for rectangles:

- **`x`**: the rectangle's origin (top left-hand corner) horizontal coordinate
- **`y`**: the rectangle's origin vertical coordinate
- **`width`**: the rectangle width
- **`height`**: the rectangle height

It's important to note that, in SVG, **vertical axes go from top to bottom**.

We can set these attributes using the **`.attr()`** method again. To give them a value based on the data, we should use an accessor function. For example:

```
let bars = barGroup.selectAll('rect')
    .data(cities, d=>d.city)
    .join('rect')
    .attr('x', (d,i)=>i*40+5)
    .attr('height', d=>d.alt*10);
```

In the example above, bars will be positioned horizontally based on their index, every 40 pixels, with an offset of 5 pixels. They will also have their height based on their value of **`alt`** (factored to have it fit inside the **`svg`**).

Add the other missing attributes:

- The **`width`** should be 40 pixels (or any interval set with **`x`**). When providing constants, you don't need to write an accessor, the value on its own works just as well.
- The **`y`** coordinate should be the total SVG height minus the bar's height (to have it aligned at the bottom of the chart)

Remember to check the Web Console and inspect elements in case of problems.

Inspect the bar associated with the city Ottawa, is there any problems with it? What could be the reason?

Styling

There are two ways we can style visualisation elements:

- In CSS stylesheets
- By editing their **`style`** attribute

Let's first look at editing the CSS stylesheets. We have to make sure we can match the right elements with CSS selectors. We will do so by adding classes to the bars:

```
let bars = barGroup.selectAll('rect.bar')
    .data(cities, d=>d.city)
    .join('rect')
    .classed('bar', true)
    ...
```

Note that we also updated the **`.selectAll()`** method to reflect that change. Now we are only concerned with rectangle elements with the class **`bar`**.

Let's then move to our **`styles/`** folder. To make things clearer, we will have a dedicated stylesheet for bar chart visualisations, hence, we need to add an import statement in **`main.css`**:

```
@import 'vars.css';
@import 'barchart.css';
```

We can then create a new stylesheet **`barchart.css`**. In it, we will first add a rule for our top-level **`svg`** element:

```
svg.barchart {
    border: solid 1px #333;
}
```

This should add a border around the chart, making it easier to locate on the page.

Tip: In the Web Console, when inspecting elements, you can check all the styles applied to them. You can even quickly add/remove styles in order to see how they would work on the page.

Then we can add a rule for the bars themselves:

```
svg.barchart rect.bar{
    fill: #3F94D3;
    stroke: #003C71;
    stroke-width: 2px;
}
```

Check how this has changed the bar chart, on the page and in the Web Console.

Using this approach is great to make global default style settings.

The other alternative is to change these styles with the D3 selections. This lets us use a more case-specific approach, where styles can be driven by the data.

Note that both techniques can be used simultaneously. Keeping the stylesheet, we can for example add a rule to make the bars red if the **`pop`** value is below a certain threshold using the **`.style()`** method:

```
let bars = barGroup.selectAll('rect.bar')
    .data(cities, d=>d.city)
    .join('rect')
    .classed('bar', true)
    ...
    .style('fill', d=>d.pop<1000000?'#ba4a53':null)
    .style('stroke', d=>d.pop<1000000?'#381619':null)
```

The JavaScript **ternary operator** takes the format: **`condition ? ifTrue : ifFalse`**

Note that some SVG elements can have some of their characteristics defined by attributes and styles. But, as anything with CSS styles, there is an order of priority:

1. (top priority) local element **`style`** attribute
2. last-defined global style in CSS stylesheet or **`<style>`** tag
3. (least priority) element attribute

Going Further

Making Reusable Charts

So far you should have been working in the **`main.js`** file mostly, simply adding statements to build the bar chart. This is fine when you get started and want to develop your charts.

However, as you progress, you might be interested in wrapping this code into reusable bits, such that you can then instantiate several charts of the same type, without duplicating code (and introducing errors and bugs...).

There are two options in JavaScript:

- Historically, the way to go about it was to create a **factory function**: a function that will take on several parameters for your chart. Inside this function’s scope, you can declare any variable and functions needed to create the chart. Importantly, you also declare and return an object that will contain “*public*” attributes and methods.
- Since ES 2015, JavaScript supports **classes**. You can check the guide in Appendix A to see how to use classes in JavaScript.

Because they have several efficiency advantages over factory functions, I’ll recommend you use classes.

Here is an example with bar charts, note that we are abstracting the dataset structure to an array of key/value pairs, in order to make this class reusable:

```
// in file BarChart.js
export default class BarChart{

    // Attributes (you can make those private too)
    width; height; // size
    svg; chart; bars; // selections
    data; // internal data

    // Constructor
    constructor(container, width, height){
        this.width = width;
        this.height = height;

        this.svg = d3.select(container).append('svg')
            .classed('barchart', true)
            .attr('width', width).attr('height', height);
        this.chart = this.svg.append('g');
        this.bars = this.chart.selectAll('rect.bar');
    }

    // Private methods
    // data is in the format [[key,value],...]
    #updateBars(){
        this.bars = this.bars
            .data(this.data, d=>d[0])
            .join('rect')
            .classed('bar', true)
            .attr('x', (d,i) => i*40+5)
            .attr('y', d => this.height - d[1]*10)
            .attr('width', 40)
            .attr('height', d => d[1]*10);
    }

    // Public API

    // The dataset parameter needs to be in a generic format,
    // so that it works for all future data
    // here we assume a [[k,v], ...] format for efficiency
    render(dataset){
        this.data = dataset;
        this.#updateBars();
        return this; // to allow chaining
    }
}
```

Note that this code should go in its own **`BarChart.js`** file. The **`export default`** keywords are used to turn this class in a module. Which mean we can use it other parts of the application. With the Bar Chart code written in a dedicated class, we can now replace the content of our **`main.js`** file:

```
// in file main.js
import BarChart from './BarChart.js';

let cities = [ ... ]

let bar1 = new BarChart('div#bar1', 800, 500);

// this line transforms the cities dataset in the generic format
// that BarChart expects: [[k, v], ...]
// we will explain it further in the next lab
let citiesElevation = cities.map(d=>[d.city, d.alt]);

bar1.render(citiesElevation);
```

The **`‘./’`** in front of **`‘Barchart.js’`** is extremely important here. You have to provide a relative path to ensure your application will find the script you are talking about. (Non-relative paths are historically used for *npm* packages, and this was kept to allow backward compatibility – if you wonder why JavaScript does things in a weird way sometimes, just assume that this is a backward compatibility issue).

Making Other Charts

This all process can be applied to make other types of charts too. You just need to look for the SVG shape that best suits your problem apply the General Update Pattern:

1. Select the elements.
2. Bind data to elements.
3. Join elements.
4. Update the elements’ attributes with data values.

For instance, to make a Bubble Chart (with circles):

1. Select your circles.
2. Bind data to circles.
3. Join circles.
4. Update the circles’ **`cx`**, **`cy`** and **`r`** attributes (centre x, centre y, radius).

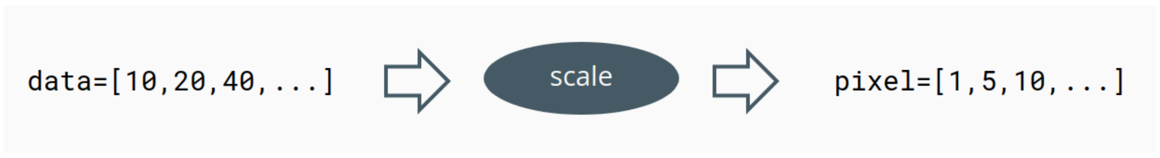
Tutorial 3 – D3 Scales and Axes

As you would have realised with the previous exercise, it's difficult to make data values fit in limited pixel spaces. Rather than trying to figure out the right factor every time, we can use **scales** to help us automatically do the computation.

Again, this tutorial might seem abstract until you start experimenting with scales in your application, but make sure to revisit this section if things are a bit confusing.

Scales

Scales are essentially functions that can translate data values from one range to another. We typically use them to translate values from a dataset into pixel values for our screens, but they have many more applications (e.g., matching categories to colours).



While we could write our own scales (and in some rare cases, you might need to), the D3 scale module already provides a set of scales that will suit most our needs. We can cover them all in detail, but as always, you should check the documentation to learn more: https://d3js.org/d3-scale.

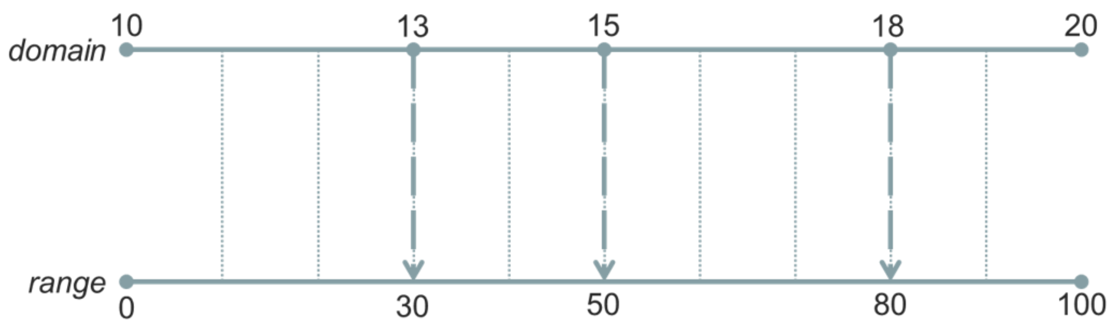
Types of Scales in D3

In D3's language, a scale will map an input ***domain*** to an output ***range***. There are many scales available, but we can put them in 4 categories.

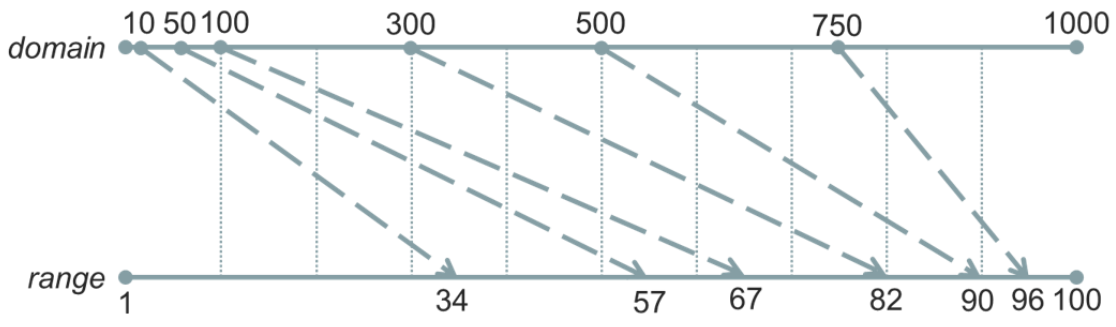
		Range	
		Continuous	*Discrete*
Domain	*Continuous*	Linear, Log, Symlog, Power / Square root, Identity, Radial, Time (UTC) Sequential and Diverging (using colour interpolators)	Quantize, Quantile, Threshold
	Discrete	Band, Point	Ordinal

Going **from continuous domain to continuous range** simply means to project values within a new set of minimum and maximum values. However, each type of scale will have different growth ratios.

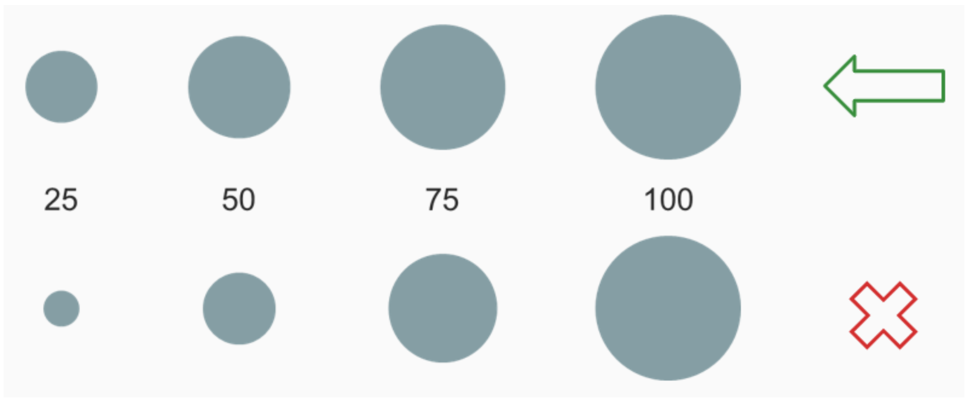
A linear scale, for example, will ensure that differences rate remain the same between domain and range.



A logarithmic scale will instead aim to preserve the differences in orders of magnitude. For example, with a logarithmic scale of base 10, the perceptual difference between 1 and 10 will be the same as between 10 and 100. This is useful when you have few large values that would otherwise “squish” the many small value.



A square root scale can be useful when you are trying to map a single value meant for establishing an area, typically the radius of a circle. Using a linear scale for this purpose introduces a visual bias since the difference in area would be the square of the difference in radius: if your radius doubles, your area quadruples. A square root scale prevents this. In the image below, the top row uses a square root scale, and the bottom row uses a linear scale.

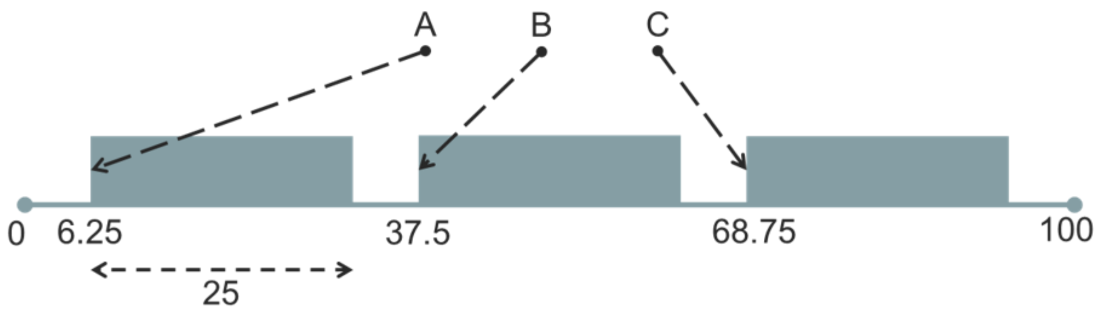


Time scales are specifically meant to parse temporal values. Sequential and diverging scales are typically meant to map continuous values to continuous colour scales, using colour interpolators.

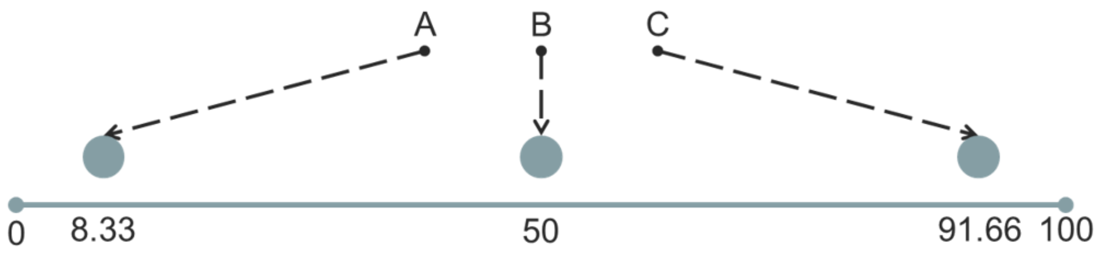
Going **from continuous domain to discrete range** means separating your values into categorical bins. This is used to discretely allocate colours scales.

Going **from discrete domain to continuous range** often means calculating the position and space occupied by a category. Typically, these scales are used to figure out the position, in pixels, for marks representing categories. There are two types: band and point.

Band scales will equally divide the space available into the required number of categories. You can then use the scale to find the starting point of a category, and the size of *bands*.



Point scales will only make sure that each category is equally spread, without giving them a size.



These two scales have many options for the padding applied between and outside of the categories.

Finally, going **from discrete domain to discrete range** simply means matching one set of categories to another. This is typically used to assigned colours to categories, but you can also use it to match shapes to categories too. There is only the ordinal scale that fits in this category.

Using D3 Scales

What D3 actually provides are constructor methods for scales. With each scale instantiated you can then adjust several parameters depending on their type.

Generally speaking, you can create a scale with the statement:

```
let scale = d3.scaleType(domain, range);
```

Note that **`scaleType`** needs to be replaced with the actual scale type you intend to use, e.g., **`scaleLinear`** or **`scaleOrdinal`**. You can omit the domain from this statement, or omit both domain and range (default values will be applied).

Later on, you can modify the domain and range (e.g., if you need to scale new data, or resize your chart):

```
scale.range(newRange);
scale.domain(newDomain);
```

With continuous values, **`domain`** or **`range`** are an array of two elements: the minimum and maximum values. With discrete values, you need to supply an array of all possible values.

You should familiarise yourself with the D3 Scale documentation to check the various options available for the type of scale you wish to use.

Once your scale has been defined, you can use it by passing a value from the domain, so that it returns the equivalent value in the range. Some scale types allow the operation to be reversed too.

```
let rangeValue = scale(domainValue);
```

Some Useful Array Functions

We will have a look at these in more detail in the next lab. For constructing scales, however, you might find these array operations useful.

- **`array.map(accessor)`** - maps the values of **`array`** according to the accessor function and returns the new array;
- **`d3.min(array, accessor)`** - returns the minimum value in **`array`**. If an accessor is given it will perform a map first.
- **`d3.max(array, accessor)`** - returns the maximum value in **`array`**. If an accessor is given it will perform a map first.
- **`d3.extent(array, accessor)`** - returns the extent of values (in the **`[min,max]`** format) in **`array`**. If an accessor is given it will perform a map first.

```
let data = [['a',12],['b',18],['c',25],['d',3]];

data.map(d=>d[0]);          // ['a','b','c','d']
d3.min(data, d=>d[1]);      // 3
d3.max(data, d=>d[1]);      // 25
d3.extent(data, d=>d[1]); // [3,25]
```

Axes

D3 Axis is a module that works hand-in-hand with D3 Scale. It essentially allows you render the scales created by the D3 Scale module.

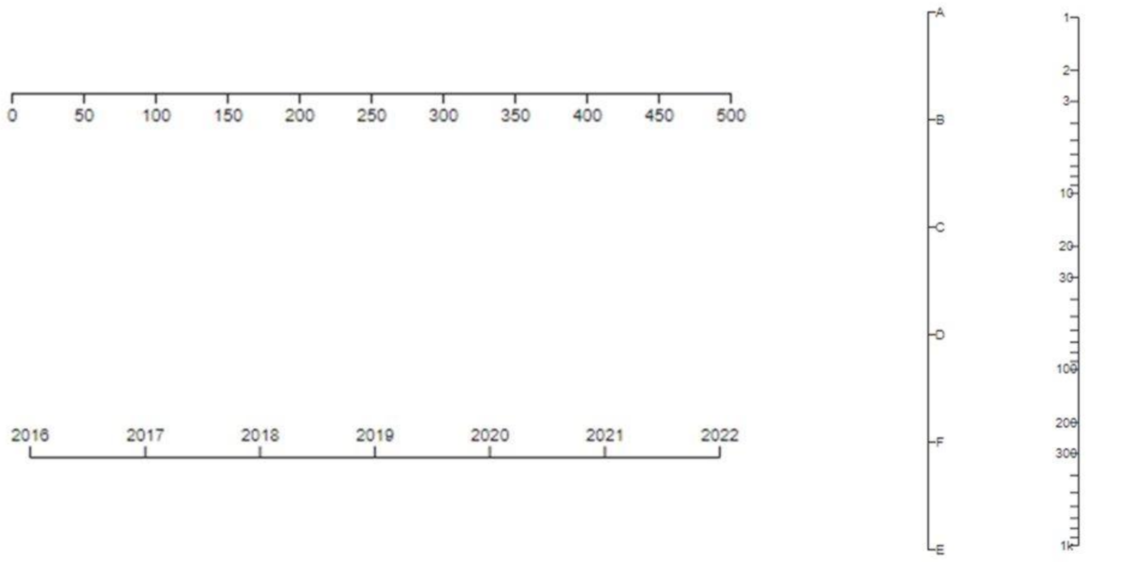
You typically first need to create an axis generator, providing a scale you want to visualise. Then, to render the axis, you need to call the axis generator on a D3 selection. The rest of the axis elements will be automatically appended. See the examples below.

```
let s1 = d3.scaleLinear([0,500],[0,400]);
let axis = d3.axisBottom(s1);
mySelection.call(axis);
```

```
let s2 = d3.scalePoint(['A','B','C','D','F','E'],[0,400]);
let axis = d3.axisRight(s2);
mySelection.call(axis);
```

```
let s3 = d3.scaleLog([1,1000], [0,400]);
let axis = d3.axisLeft(s3);
mySelection.call(axis);
```

```
let dates = [new Date(2016, 0, 1), new Date(2022, 0, 1)];
let s4 = d3.scaleTime(dates, [0,400]);
let axis = d3.axisTop(s4);
mySelection.call(axis);
```



You can typically customise the number and format of your axis’ tick marks. Check the [D3 Axis documentation]() to learn more.

Exercise – Scalable Charts

In this exercise, we will continue developing our bar chart visualisation. This time we will focus on making the bar’s height and width scale with the SVG’s own dimensions.

Updating the Constructor

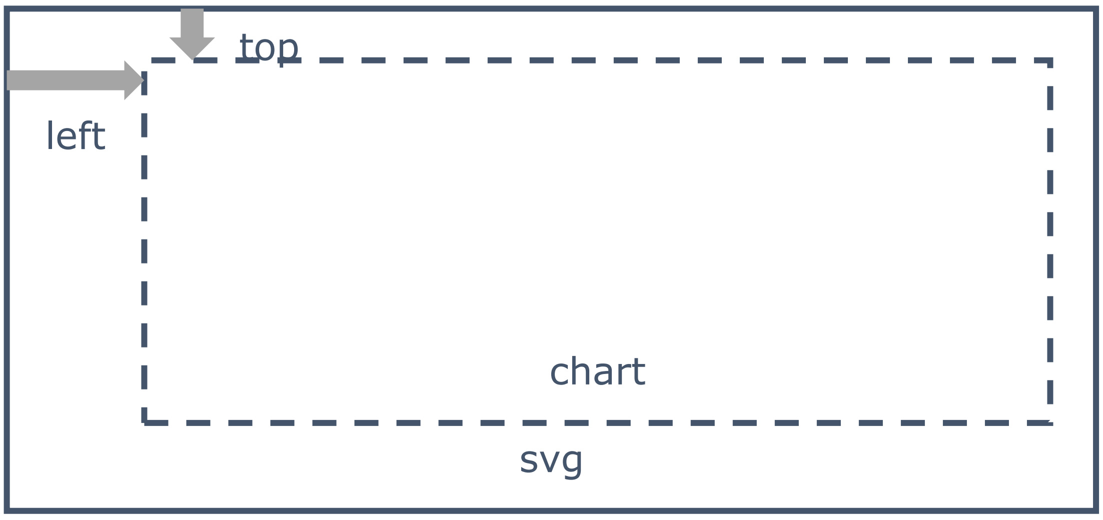
We are going to first change our bar chart constructor to include margins (so that we can fit axes later).

```
/*
- container: DOM selector
- width: visualisation width
- height: visualisation height
- margin: chart area margins [top, bottom, left, right]
*/
constructor(container, width, height, margin){

    this.width = width;
    this.height = height;
    this.margin = margin;

    this.svg = d3.select(container)
        .append('svg')
        .attr('width', this.width)
        .attr('height', this.height)
        .classed('barchart', true);

    this.chart = this.svg.append('g')
        .attr('transform',
              `translate(${this.margin[2]},${this.margin[0]})`);
    this.bars = this.chart.selectAll('rect.bar');
}
```

Make sure to add a **`margin`** attribute to your class too, and add it in the list of parameters where you instantiate your bar chart in main.js.

```
let bar1 = new BarChart('div#bar1', 700, 500, [10,40,45,20]);
```

Note that we have added an attribute to our **`chart`** selection: **`transform`**. The **`transform`** attribute allows you to apply common transformations to your selection (translation, scaling, rotation, skew). Here we translate the whole **`chart`** group to a new origin, corresponding the left margin horizontally and top margin vertically. It means that every child of this group will be drawn in relation to that new origin.



We used the JavaScript String Templates, or Template Literals. It allows you to invoke variables in a string. The variables will be evaluated at run time and added in place in the string. We define them using backticks ` ` and add variables with **`${ }`**.

Scales

Our task is now to implement scales in the bar chart. We need two scales: one to position and size the bars horizontally, on the x axis, and one to position and size them vertically, on the y axis.

First let's add properties to the class for these two scales, for example, **`scaleX`** and **`scaleY`**. Then, like the private **`#updateBars()`** method, we will add a private **`#updateScales()`** method.

The aim is to call this method at the start of the rendering process to create/update the scales before using them when drawing bars. As such we must call it inside the **`render`** method. Make sure to do so **after** saving the dataset, as we will need to access the dataset when defining our scales:

```
render(dataset){
    this.data = dataset;
    this.#updateScales();
    this.#updateBars();
    return this;
}
```

Defining Ranges

First, we will define all our scales' ranges, i.e., the bounds within which we can draw marks in our view:

- the horizontal space available in our view
- the vertical space available in our view

The horizontal space should go from the left margin to the full width minus the right margin. However, because we will draw bars from the origin of the chart area and because we have already translated this area, we don't need to worry about the margins anymore. The horizontal space, therefore, goes from `0` to the chart area width (full width minus margins).

```
#updateScales(){
    let chartWidth = this.width-this.margin[2]-this.margin[3];
    let rangeX = [0, chartWidth];
}
```

We can apply a similar logic with the vertical space. However, remember that the vertical axis in SVG goes from top to bottom. Hence, if we want to draw bars from bottom to top, we should invert the bounds of your range. This way, the lower bounds of our domain will match with the bottom of the chart area, while the upper bounds of the domain will match with the top of the chart area.

```
#updateScales(){
    let chartWidth = this.width-this.margin[2]-this.margin[3],
        chartHeight = this.height-this.margin[0]-this.margin[1];

    let rangeX = [0, chartWidth],
        rangeY = [chartHeight, 0];
}
```

Defining Domains

Then, we will query the data to get the domains of our scales, I.e., the bounds matching the chart area bounds in the data world.

With our horizontal axis (x), we want to evenly spread our *discrete* set of categories (or keys) in our data. Each category should have a width. The ideal scale for this is a band scale, which requires the domain to be an array of all possible values. We can get this using the array map function, which mutates an array with an accessor on a one-to-one basis.

```
#updateScales(){
    ...
    let domainX = this.data.map(d=>d[0]);
}
```

Obviously, you should adapt the accessor function to something that matches the data format you have set for bar charts.

With the vertical axis (y), we want to fit the bounds of our *continuous* domain within the view. But beware of simply capturing the extent of the values you have in your data. With this bar chart, we want to see the **full** quantity of population or area. Hence, our data domain actually goes from `0` to the maximum value, otherwise we start introducing biases in our representations. There can be two exceptions to that logic:

- Some values are negative – in which case you might choose the full extent; or
- All values are large and with low variance – in which case you would want to *zoom* in to see differences.

To find the maximum value in an array, we can use the **`d3.max`** functions, which can take an optional accessor as parameter to map values first.

```
#updateScales(){
    ...
    let domainX = this.data.map(d=>d[0]),
        domainY = [0, d3.max(this.data, d=>d[1])];
}
```

Instantiating Scales

With the domains and ranges established, we can now instantiate our scales. We will use two types:

- A band scale for positioning (and sizing) bars horizontally; and
- A linear scale for sizing (and positioning) bars vertically.

Remember to check the D3 Scale documentation to get a full description of the scales provided by D3.

To instantiate a scale, we typically just need to call the appropriate scale generator, passing the domain and range as arguments. We can also use the **.domain()** and **.range()** methods to update them after the scale has been instantiated.

```
#updateScales(){
    let chartWidth = this.width-this.margin[2]-this.margin[3],
        chartHeight = this.height-this.margin[0]-this.margin[1];
    let rangeX = [0, chartWidth],
        rangeY = [chartHeight, 0];
    let domainX = this.data.map(d=>d[0]),
        domainY = [0, d3.max(this.data, d=>d[1])];
    this.scaleX = d3.scaleBand(domainX, rangeX).padding(0.2);
    this.scaleY = d3.scaleLinear(domainY, rangeY);
}
```

Note that we are saving these scales as class attributes to reuse them in other methods. Make sure to have these attributes declared! Note that we are giving a padding to our band scale in order to separate the bars a bit.

Using Scales

With the scales now instantiated, we can change the code rendering bars to make use of them. We typically use a scale by calling it and passing a value from the domain, it will then return the associated value in the range.

Let’s start with the horizontal positioning of bars. Using the band scale by passing the category (or city) value will return the start of the band on the horizontal axis. Which exactly what we need.

```
.attr('x', d=>this.scaleX(d[0]))
```

As it calculated how to spread bars horizontally, the band scale would have also calculated the width they can occupy. We can query this value using the **.bandwidth()** function and assign it to the **width** attribute of bars.

```
.attr('width', this.scaleX.bandwidth())
```

Then we can position the bars vertically, the top of the bar (`y`) corresponding to the value we want to show. Again, the linear scale has been created just for that purpose.

```
.attr('y', d=>this.scaleY(d[1]))
```

Finally, we need to give our bar a height. Using the same trick we used before, we can calculate it by looking up the maximum height possible (bottom of our chart area) and then subtracting the space above the bar (which corresponds to the y coordinate). When these sort of geometry gets complicated, I recommend you use a piece of paper where you can scribble things and see what's happening.

```
.attr('height', d=>this.scaleY(0)-this.scaleY(d[1]))
```

Check the changes on the web page. You can now change the width and height of your bar chart (in **`main.js`**) to see how the scales adapt your bars to it. Do you think there could be a more appropriate scale for the vertical axis? Check the D3 Scale documentation and adapt your code as you see fit.

Axes

We have to inform users that our charts are using scales. The best way to do this is to render axes next to the chart area (which is why we needed margins).

The first thing to do is create **`g`** elements in which the axes will be rendered. In the bar chart properties and constructor add the necessary code for this (as we did for the chart area).

```
constructor(container, width, height, margin){
    ...
    this.chart = this.svg.append('g')
        .attr('transform',
        `translate(${this.margin[2]},${this.margin[0]})`);
    this.bars = this.chart.selectAll('rect.bar');

    this.axisX = this.svg.append('g')
        .attr('transform',
        `translate(${this.margin[2]},${this.height-this.margin[1]})`);
    this.axisY = this.svg.append('g')
        .attr('transform',
        `translate(${this.margin[2]},${this.margin[0]})`);
}
```

Note that we are placing the x-axis at the bottom of the visualisation.

All that we have to do then, is create axis generators with the scales and pass these generators to the appropriate **`g`** elements. Create a private method for this, and call it in the render function, after updating the scales.

```
#updateAxes(){
    let axisGenX = d3.axisBottom(this.scaleX),
        axisGenY = d3.axisLeft(this.scaleY);
    this.axisX.call(axisGenX);
    this.axisY.call(axisGenY);
}
```

Wrapping Up

Your bar chart class should now look something like this.

```
export default class BarChart{

  width; height; margin;
  svg; chart; bars; axisX; axisY; labelX; labelY;
  scaleX; scaleY;
  data;

  constructor(container, width, height, margin){
    this.width = width;
    this.height = height;
    this.margin = margin;

    this.svg = d3.select(container).append('svg')
      .classed('barchart', true)
      .attr('width', this.width).attr('height', this.height);

    this.chart = this.svg.append('g')
      .attr('transform',
      `translate(${this.margin[2]},${this.margin[0]})`);
    this.bars = this.chart.selectAll('rect.bar');

    this.axisX = this.svg.append('g')
      .attr('transform',
      `translate(${this.margin[2]},${this.height-this.margin[1]})`);
    this.axisY = this.svg.append('g')
      .attr('transform',
      `translate(${this.margin[2]},${this.margin[0]})`);

    this.labelX = this.svg.append('text')
      .attr('transform', `translate(${this.width/2},${this.height})`)
      .style('text-anchor', 'middle').attr('dy',-5);
    this.labelY = this.svg.append('text')
      .attr('transform', `translate(0,${this.margin[0]})rotate(-90)`)
      .style('text-anchor', 'end').attr('dy',15);
  }
  #updateScales(){
    let chartWidth = this.width-this.margin[2]-this.margin[3],
        chartHeight = this.height-this.margin[0]-this.margin[1];
    let rangeX = [0, chartWidth],
        rangeY = [chartHeight, 0];
    let domainX = this.data.map(d=>d[0]),
        domainY = [0, d3.max(this.data, d=>d[1])];
    this.scaleX = d3.scaleBand(domainX, rangeX).padding(0.2);
    this.scaleY = d3.scaleLinear(domainY, rangeY);
  }

  #updateAxes(){
    let axisGenX = d3.axisBottom(this.scaleX),
        axisGenY = d3.axisLeft(this.scaleY);
    this.axisX.call(axisGenX);
    this.axisY.call(axisGenY);
  }

  #updateBars(){
    this.bars = this.bars
      .data(this.data, d=>d[0])
      .join('rect')
      .classed('bar', true)
      .attr('x', d=>this.scaleX(d[0]))
      .attr('height', d=>this.scaleY(0)-this.scaleY(d[1]))
      .attr('width', this.scaleX.bandwidth())
      .attr('y', d=>this.scaleY(d[1]));
  }

  render(dataset){
    this.data = dataset;
    this.#updateScales();
    this.#updateBars();
    this.#updateAxes();
    return this;
  }

  setLabels(labelX='categories', labelY='values'){
    this.labelX.text(labelX);
    this.labelY.text(labelY);
    return this;
  }
}
```

Your code might look different. If it works that's great, if not you can compare it with the code above.

You might also notice that I have added something new to this class. Hopefully, by now, you should be able to decipher this code, understand what it does, and make use of it in the **`main.js`** file.

Exercise – Let's Make a Scatter Plot

At the end of the *Let's Make a Bar Chart* exercise, I've mentioned that you could reuse the same process to build a bubble chart.

Using your knowledge of D3 selections, scales and axes, create a **`ScatterPlot`** class. Then, use your class to plot the cities' population against their area.

You can repurpose most of the code shown above. Just make sure to adapt it to show circles and not rectangles.

Can you find a way to add a colour scale too? For example one colour per city name? (You will need to re-think your data abstraction).

If you have implemented a Bubble Chart class, you could also update it to use scales. You will need a third scale for setting the bubble radii.

Ask a lab helper for help if you are not sure how to proceed.

Tutorial 4 – D3 Shapes

So far, we have looked at the creation of simple SVG shapes:

- Rectangles that need two coordinates and two size values; and
- Circles that need two coordinates and one radius value.

Lines, polylines and polygons are simple too, as they just require you to provide a list of points. You just need to calculate the coordinates of such points.

But what about complex shapes: polynomial curves, arcs, glyphs and symbols, etc.? SVG provides a special element for those: **`path`**. A path only needs one attribute called **`d`** (for draw). The value of **`d`** is essentially a string containing a list of commands:

- Move to these coordinates;
- Draw a string line;
- Draw a Bézier curve;
- Draw an elliptical arc.

Paths are extremely powerful, with just a set of few commands, we can draw just about anything. But we need to calculate the parameters for all these commands. From experience, this tends to be very complicated …

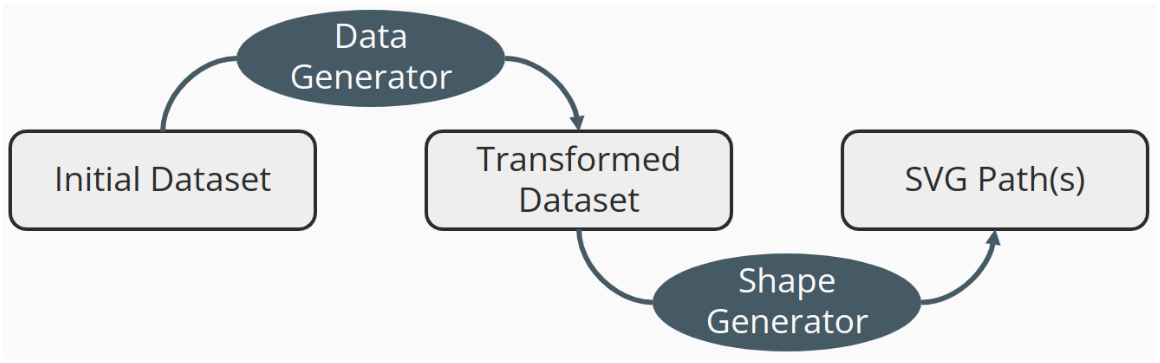
Fortunately, D3 comes with the Shape module, which contains objects and functions that can help us draw these complex shapes.

Data Generators

Sometimes, the data we have at hand is not enough to draw the marks we need to draw. For example, how can we draw a pie chart from the following data? Where are the angles we need?

```
[{k:“cat”,v:86}, {k:“dog”,v:75}, {k:“fish”,v:20}, {k:“hamster”,v:34}];
```

Data generators are functions that will take an original dataset and create a new transformed dataset that contains values that corresponds better to what we want to visualise. This data can then be passed into shape generators to get SVG paths.



The **pie generator** for instance, calculates the angles necessary to draw a pie chart or a donut chart. You can instantiate one by calling the **`d3.pie()`** function, as always, you should check the documentation to learn about all the options available. With pie generator, the dataset above becomes:

```
[{data:{...},index:0,value:86,startAngle:0,endAngle:2.5,padAngle:0},
 {data:{...},index:1,value:75,startAngle:2.5,endAngle:4.7,padAngle:0},
 {data:{...},index:3,value:20,startAngle:5.7,endAngle:6.2,padAngle:0},
 {data:{...},index:2,value:34,startAngle:4.7,endAngle:5.7,padAngle:0}]
```

Note that the original data gets copied in the **`data`** attribute.

The **stack generator** will calculate the cumulative values of layers, which can be useful for stacked displays. You instantiate one with the **`d3.stack()`** function (documentation). With it, this dataset:

```
[{dog:45,cat:32}, {dog:48,cat:50}, {dog:42,cat:48}]
```

Can become:

```
[[[0,45], [0,48], [0,42], key:'dog', index:0],
[[45,77], [48,98], [42,90], key:'cat', index:1]]
```

Shape Generators

Shape generators are functions that actually do the job of transforming dataset into SVG paths. Note that the data you provide doesn't always need to have been transformed with a data generator.

A **line generator** will draw the SVG path that connects a series of points. You can instantiate one with **`d3.line()`** (documentation). The main options you should pay attention to concern the accessors for values of x and y, and the *curve interpolator* you might want to use. With it:

```
[{x:0,y:32},{x:2,y:50},{x:4,y:48},{x:6,y:36},{x:8,y:24}]
```

Becomes:

```
'M0,32L2,50L4,48L6,36L8,24'
```

An **arc generator** will draw SVG paths representing arcs, typically for pie and donut charts. You can instantiate one with **`d3.arc()`** (documentation). The default options integrate directly with the results of **`d3.pie()`**. For example:

```
{data:{...},index:0,value:86,startAngle:0,endAngle:2.5,padAngle:0}
```

Becomes:

```
'M2.4492935982947065e-15,-
40A40,40,0,0,1,23.511410091698927,32.3606797749979L11.755705045849464,16.18
033988749895A20,20,0,0,0,1.2246467991473533e-15,-20Z'
```

However, you can still use options to access and transform the angle values, as well as setting the inner and outer radii of your pie chart or donut chart.

There are other shape generators available:

- Areas is similar to lines, but fills the surface below the curve;
- Links draws Bézier curves connecting a source point ot a target point; and
- Symbols provides a pre-set list of shapes that you can use for categorising dots in a scatter plot.

Curve Interpolators

The line and area generators often use curve interpolators to help drawing the actual path a line should take, especially between two consecutive points.

You can create your own, but D3 provides several presets:

- d3.curveBasis & d3.curveBundle - B-Spline interpolation
- d3.curveBumpX/Y - pair-wise Bézier curves
- d3.curveCardinal - cardinal spline interpolation
- d3.curveNatural - cubic spline interpolation
- d3.curveStep - Horizontal and vertical line interpolation
- d3.curveLinear - pair-wise straight lines

Putting it Together

With data and shape generators we can now draw stacked bar charts, area charts, line charts and pie/donut charts and many more.

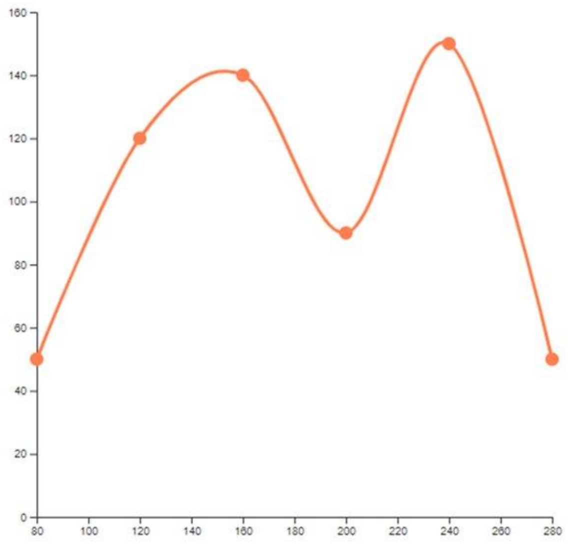
For example, **to draw a line chart**, we can use the following code:

```
// data
let data = [[80, 50],[120, 120], [160, 140],
            [200, 90], [240, 150],[280, 50]];

// scales
let xScale = d3.scaleLinear(...),
    yScale = d3.scaleLinear(...);

// creating the line generator
let lineGen = d3.line()
    .curve(d3.curveCardinal)
    .x(d=>xScale(d[0]))
    .y(d=>yScale(d[1]));

// creating a path, joining datum
// drawing with the line generator
let line = chart.append('path')
    .datum(data)
    .attr('fill', 'none')
    .attr('stroke', 'coral')
    .attr('stroke-width', 3)
    .attr('d', lineGen);

// draw the dots
// draw the axes
```

You should note the following:

- This code assumes that you have a **`chart`** selection.
- The use of the **`d3.curveCardinal`** interpolator, and x and y options in the line generator definition.
- We are drawing **one** line (one mark) therefore we are binding **one** data item (the list of points composing the line) using the **`.datum()`** method (singular of data).
- Because we are drawing one line, we are not making use of the full General Update Pattern. To draw multiple lines, or allow this code to be reusable, you would need to adapt it.

With this code, we get this result (with dots and axes added):



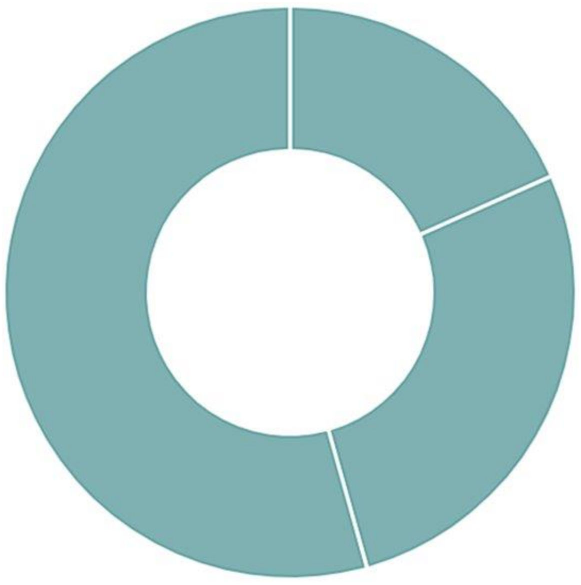
To draw a donut chart, we can use the pie data and arc shape generators:

```
// getting the transformed data for a pie chart
let data = [['a',2], ['b',3], ['c',6]];
let pieGen = d3.pie()
    .padAngle(0.02)
    .sort(null)
    .value(d=>d[1]);
let pieData = pieGen(data);

// creating an arc generator
let arcGen = d3.arc()
    .innerRadius(width/4)
    .outerRadius(width/2-5);

 // drawing the arcs
let arcs = chart.selectAll('path')
    .data(pieData, d=>d.data[0])
    .join('path')
    .attr('fill', 'cadetblue').attr('fill-opacity', 0.8)
    .attr('stroke', 'cadetblue').attr('stroke-width', 2)
    .attr('d', arcGen);
```

Again, there are a few things to note:

- The code assumes that there is a **`chart`** selection and a **`width`** variable.
- The **`.sort(null)`** option for the pie generator forces it to not reorder the data items. Otherwise, it would sort them in descending order of values. The **`.padAngle()`** option lets us put a bit of space between sequential angles.
- Instead of binding the original **`data`** to the selection of **`paths`**, we bind the transformed version, **`piedata`**.
- Because **`d3.pie()`** and **`d3.arc()`** integrate with each other well, we don't need to provide a lot of options for the arc generator, only the radius values for the donut chart.

With this code we get the following chart:



Exercise – Let's Make a Donut Chart

Using the code example as reference, and by researching the documentation, create a donut chart showing the proportions of population each city (using the dataset example from previous exercises).

You can start by working in **`main.js`**. Then, you can attempt to make a separate **`DonutChart`** class.

Hints:

- You don't need have scales.
- You should translate your **`chart`** selection to the middle of the **`svg`**, i.e., **`width/2`** and **`height/2`**.
- When establishing your donut's radius values, you can use the JavaScript **`Math.min`** function to find which of the chart width or height is smallest (and should be used as baseline).
- To add labels, you can
    - create a selection of texts (**`chart.selectAll('text')`**);
    - bind the transformed pie data to that selection;
    - join texts;
    - translate the text using the centroid function from the arc generator: **``.attr('transform',d=>`translate(${arcGen.centroid(d)})`);``**
    - Finally, edit the text value with the **`.text()`** function.

Exercise – Let's Make a Line Chart

Using the example code above as reference, and by researching the documentation, create a line chart showing the historical population of Edinburgh.

```
let historicPop = [
    [2000,451000],[2001,454000],[2002,457000],[2003,460000],
    [2004,463000],[2005,466000],[2006,468000],[2007,471000],
    [2008,474000],[2009,477000],[2010,480000],[2011,483000],
    [2012,489000],[2013,495000],[2014,501000],[2015,507000],
    [2016,513000],[2017,519000],[2018,525000],[2019,531000],
    [2020,537000],[2021,543000],[2022,548000],[2023,554000],
]
```

You can start by working in **`main.js`**. Then, you can attempt to make a separate **`LineChart`** class (you will have to rethink the D3 Selection methods used).

Ask a lab helper for help if you get stuck!

Appendix A – Web Languages Guide

HTML

A full list of HTML tags can be found here:

https://developer.mozilla.org/en-US/docs/Web/HTML/Element

Basic File Structure

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document Title</title>
    <link rel="stylesheet" href="path/to/stylesheet.css">
    <script src="path/to/script.js"></script>
</head>
<body>
    <div>
        This is a section.
        <h1>This is a heading</h1>
        <p>This is a paragraph.</p>
        <!-- This is a comment -->
    </div>
</body>
</html>
```

Input Elements

Texts and Search

```
<input type="text" name="text" id="textId" placeholder="type here">
<input type="search" name="search" id="searchId" placeholder="search here">
<textarea name="area" id="areaId" cols="30" rows="10" placeholder="type
here"></textarea>
```

Buttons

```
<input type="button" value="Button Title">
<button>Button <b>title</b></button>
```

Range Sliders and Numbers

```
<input type="range" name="range" id="rangeId" min="2" max="6" step="0.5">
<input type="number" name="number" id="numberId" min="2" max="6" step="0.5">
```

Radio and Checkboxes

```
<input type="radio" name="radio" id="radioId1" value="1">
<label for="radioId1">1</label>
<input type="radio" name="radio" id="radioId2" value="2">
```

```
<label for="radioId2">2</label>
<input type="checkbox" name="check" id="checkId1">
<label for="checkId1">1</label>
<input type="checkbox" name="check" id="checkId2">
<label for="checkId2">2</label>
```

Dropdown Selection

```
<select name="select" id="selectId">
    <option selected value="Option1">Option 1</option>
    <option value="Option2">Option 2</option>
    <optgroup>
        <option value="Option3">Option 3</option>
        <option value="Option4">Option 4</option>
    </optgroup>
</select>
```

Date and Time

```
<input type="date" name="date" id="dateId">
<input type="time" name="time" id="timeId">
```

CSS

Selectors

```
div#myid { } /* div with id myId*/
div.myClass { } /* div with class myClass*/

div > p { }  /* p direct child of div*/
div p { }  /* p descendant of div */

div, h1, h2 { } /* div, h1 and h2 */
```

Layouts

Flexbox

A nice guide to flexbox can be found here:

https://css-tricks.com/snippets/css/a-guide-to-flexbox/

Grid

From the same website, here is a guide to grids:

https://css-tricks.com/snippets/css/complete-guide-grid/

Variables and Functions

You can find a complete guide here:

https://css-tricks.com/complete-guide-to-css-functions/

Variables

```
div{ /* applies inside div only */
    --my-color: #e5233f;
}

:root{ /* global declaration: applies to whole DOM */
    --my-global-color: #e5233f;
}

div > p.special {
    color: var(--my-color);
}

h2.special {
    color: var(--my-global-color);
}
```

Calc Function

```
div {
    width: calc(100% - 50px);
    font-size: calc(1.3em * 1.4);
}
```

Comparison functions

```
div {
    /* preferably 200px, but definitely between 50px and 40% */
    width: clamp(50px, 40%, 200px);
    font-size: max(1.3em, 12px); /* min also works */
}
```

JavaScript

Variables

Declaration

```
// variable declaration
let variable;
// constant declaration / immutable value
const constant;
```

Primitive Types

```
// string
let str = 'value';
// number (integer and float)
let num = 1.2;
// boolean
```

```
let bool = true;

// dynamic typing
let a; // type undefined
a = 'string'; // type string
a = 2.2; // type number
a = false; // type boolean
a = null; // type null
```

Data Structures

Arrays

```
// declaration
let arr = [1, 'string', false];
let arr2 = new Array();
// accessing/setting values
arr[1];
arr[6] = 2.5;
```

You can find a full list of array methods here:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Objects

```
// declaration
let obj = {'key1': 2,
           'key2': 'str'};
let obj2 = new Object();
// accessing/setting values
obj.key1;
obj['key2'] = 'value';
```

Maps

```
// declaration
let m = new Map();
// accessing / setting values
m.set('key', 'value');
m.get('key');
```

You can find a full list of map methods here:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Sets

```
// declaration
```

```
let s = new Set();
// adding values
s.add(2);
```

You can find a full list of set methods here:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Functions

```
// declaration
function myFunction(a, b){
    let c;
    if(a < b){
        c = b - a;
    } else {
        c = a - b;
    }
    return c;
}
// or
let myFunc = function(){
    // ...
}

// call
myFunction(3,4);

// using arrow functions (compact, but no scope) with ternary operator
let myFunc = (a,b)=>a < b ? b-a : a-b;

// using default parameters
function func(a=3,b=true){
    //...
}
```

Classes

```
// declaration
class MyClass extends SuperClass{
    publicField ;
    #privateField ;
    static field = 'value';

    constructor(a, b){
        this.publicField = a;
        this.#privateField = b;
    }
    publicMethod(){
        return this.#privateField;
    }
    #privateMethod(){
        console.log(this.publicField);
    }
    static staticMethod(){
        return MyClass.field;
    }
}
// instantiation
let myObj = new MyClass(3,6);
```

Modules

Individual Named Exports

```
export function sum(a,b){  }
export function prod(a,b){  }
export let pi = 3.1415
```

List (Re)Named Exports

```
export {sum, prod, pi as approxPi}
```

(Re)Named Imports

```
import {sum as mySum, approxPi} from 'myMath.js';
```

Full Module Import

```
import * as myMath from 'myMath.js';
```

Default Export/Import

```
export default class MyClass{  }
import ClassA from 'myClass.js'
```

Loading Modules in HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document Title</title>
    <!-- using JS script that imports modules -->
    <script type="module" src="myModule.js"></script>
</head>
<body>
    <!-- ... -->
</body>
</html>
```