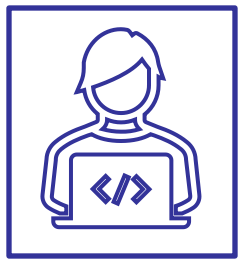
D3 Introduction

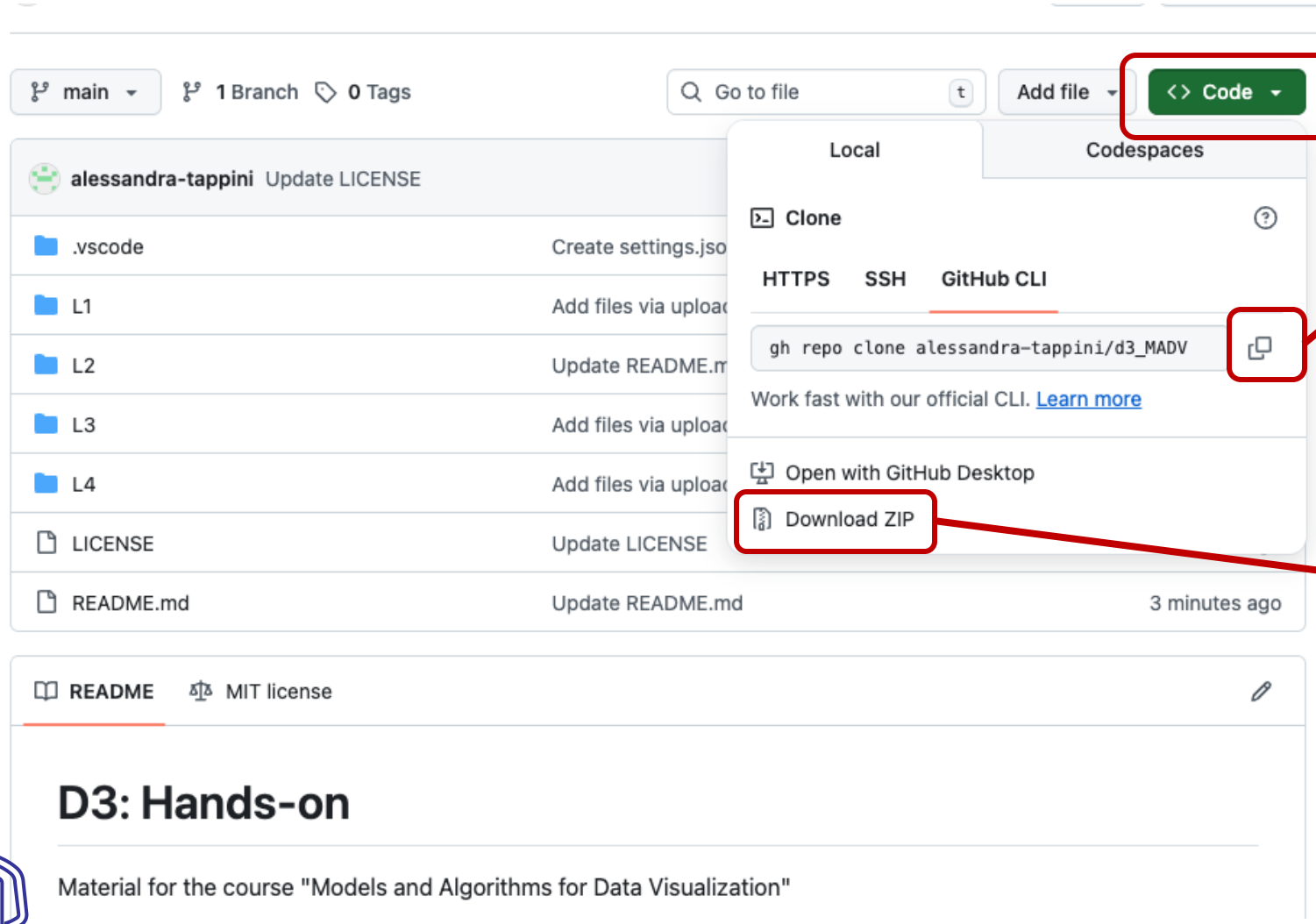
Tommaso Piselli

How these lectures work

- This series of lessons is meant to be interactive.
- The practical part will be accompanied by summary slides to resume the main concepts that we study.
- A **blue coder icon** in the current slide (like the one in the bottom left corner) means that in that slide we are producing some code (and you are highly invited to try it and code by yourself).
- For these lessons there is a **Github** repo (this) that you can clone (or download) with all the initial projects to follow the lessons.
  - This repo includes folders for each lesson.
  - You will find some starting code and the complete one.
  - More about this later.



How to access the code files



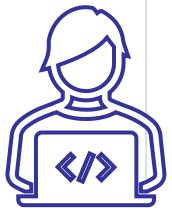
1. Click the Code button

2.a. Copy the url and then, from your terminal, launch

```
git clone <url>
```

OR

2.b. Download the files



Course Material

- **Slides** for Theory
- **Github repo** for starting code, written guides and practical exercises

Other Material:

- MDN for anything related the Web
- “Programmazione per Internet e Web” material
- All the slides are based on “D3.js in Action, Third Edition” book

What is D3?

- D3.js stands for **Data-Driven Documents**
- It is an open-source Javascript Library created in 2011 by Mike Bostock to generate dynamic and interactive data visualizations for the web.

When do we use D3.js?

- The number of tools to generate data-bound graphics has exploded in the last decade:
  - Excel
  - Power BI (Microsoft)
  - Ggplot2 (R)
  - Matplotlib (Python)
  - Tableau
  - Flourish
  - ...
  - And, of course, D3.js

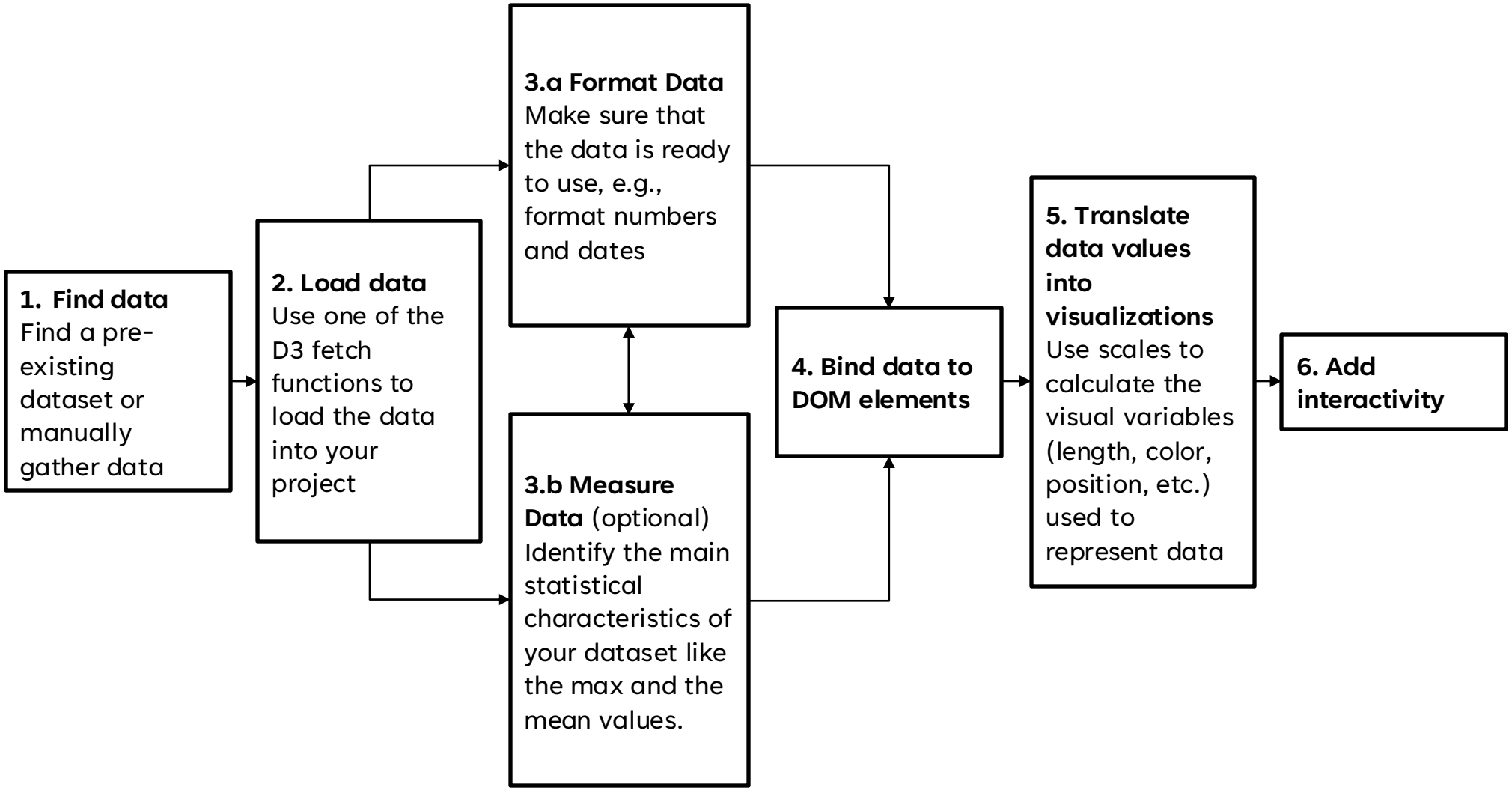
When do we use D3.js?

- With D3.js one can build visualizations ranging from a **very simple** design with no interactions to a **very complex** one in both aspects (see, for example, this).
- However, the power of D3.js shines in the **second scenario:**
  - It is generally not recommended to build simple visualizations with D3
  - Every project takes some time, just for the set up and the binding (you will soon experience this during these lectures)
  - So, you are free to use the most popular and easy tools such as Excel or Power Bi

How D3.js works

- D3.js can (among other things):
  - **Bind data to DOM elements**
    - When the data change, the elements of the DOM and their properties change
    - That is why its name is “data-driven documents”
  - **Handle interactions and animations**
    - Decide what happens when the user interacts with the graphics
    - Decide the timings of the changes
  - **Generate SVG on-the-fly**

How D3.js works



From «D3.js in Action» (third edition)

What you need to know to understand D3

- D3 is part of an **ecosystem** of technologies:
  - It is built within the DOM and leverages the power of HTML
  - Generate visualizations with SVG graphics and Canvas
  - It is a JavaScript library, so it combines many JS native functions with custom methods to access and manipulate data
  - It can be installed in different ways in your project (see later), but it usually comes as a module. It can then be easily integrated with NodeJS or other JS frameworks (React, Svelte, Vue, Angular, ...)

Raster vs Vector Graphics

Raster graphics: Based on **pixels** arranged on a grid.

- A pixel represents the smallest unit of a video image that has specific RGBA values.

Vector graphics: Based on **geometric primitives** such as points, lines, curves, and polygons.

- Based on mathematical expressions.
- More abstract level.

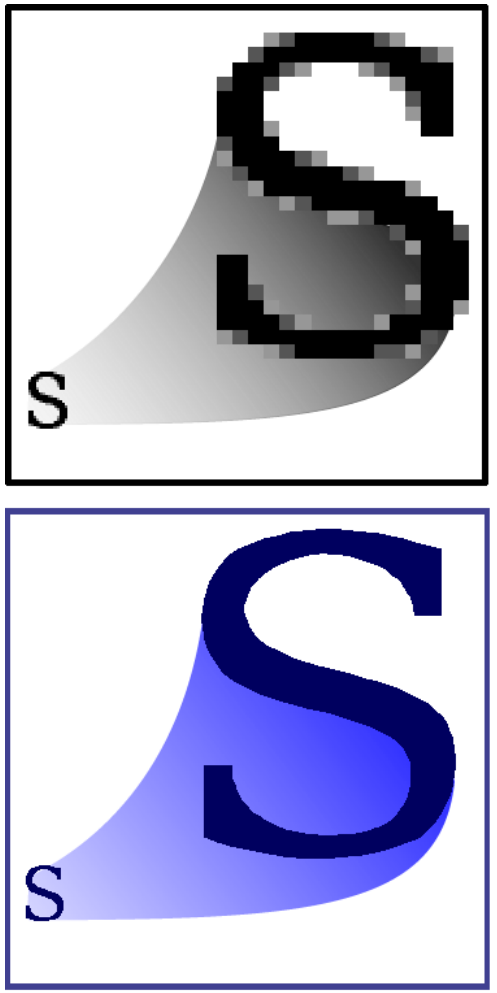
Raster vs Vector Graphics

Raster

+ Granular control down to individual pixels
+ Ideal for color images (photographs)
- Increasing the resolution or depth of the color affects the size
- Problems with scaling

Vectorial

+ Resolution Independence
+ Resize with little to no loss
- Not suitable for extremely complex images
- Time-consuming and talented



[Image from Wikipedia.org]

Scalable Vector Graphics

- While raster graphics can be realized by using the canvas element of HTML 5, Scalable Vector Graphics (**SVG**) is a family of specifications for creating 2D vector graphics.
- SVG images and their behaviors are defined in XML text files.
  - The XML text can be included in an HTML document.
  - The DOM includes XML as part of the DOM specification, we can use the DOM Tree to access and update the structure, content and style of SVG Images.

Scalable Vector Graphics

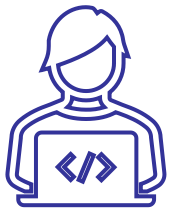
- When creating data visualizations with D3, we usually **inject** SVG shapes into the DOM and **modify** their attributes to **generate** the visual elements that compose the visualization.
- Understanding how SVG works, the main SVG shapes, and their presentational attributes are essential to most D3 projects.

- We will now cover the SVG shapes that are used in every D3 project.

SVG tutorial

For a written guide of this tutorial, please refer to the **README.md** file in the **L1** folder. You can open the file either in VSCode or Github.

This tutorial was taken from Chapter 1 of “D3.js in Action, Third Edition”

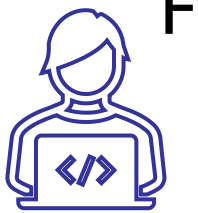


Prerequisites

- During these lectures, I will use the following tools to write and test the code. If you want my same setup, please use:
  - **Required**
    - Visual Studio Code
    - Live Server Extension for VSC
    - Prettier
    - Google Chrome
  - **Optional**
    - VSC font: Fira Code + ligatures

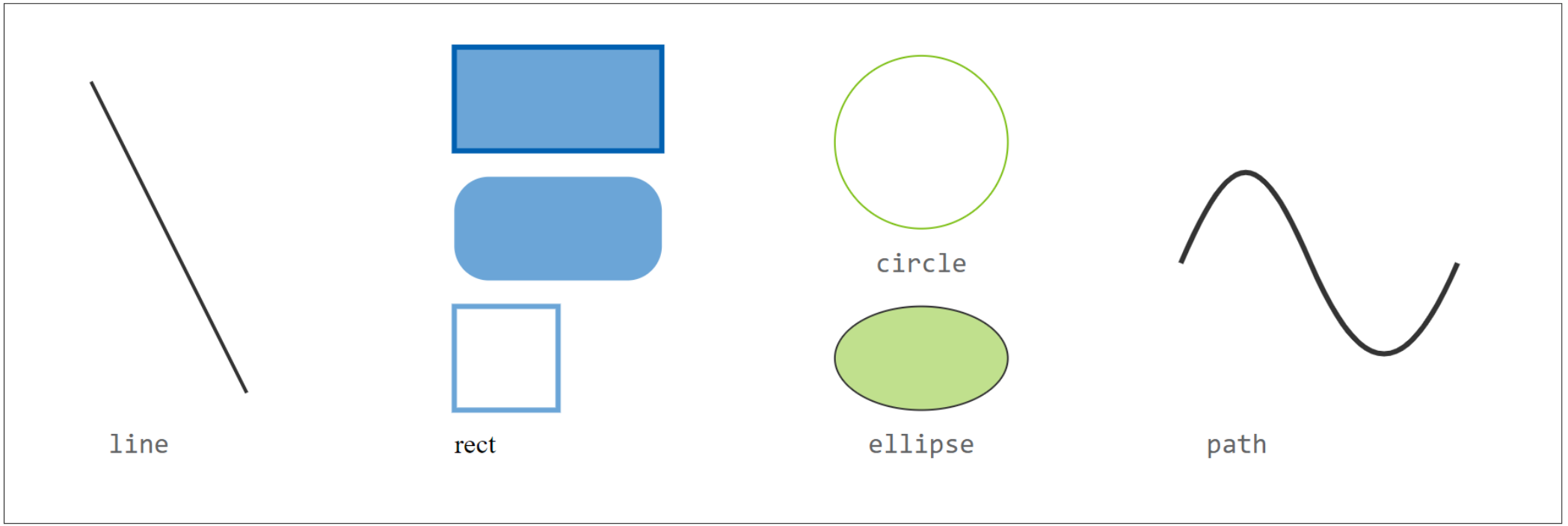
These two in particular

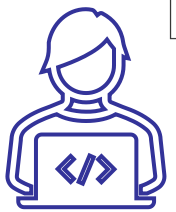
For reference: D3.js



Scalable Vector Graphics Tutorial

- We are going to do something very similar to what you see here:

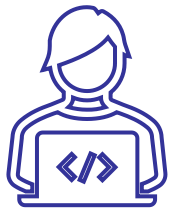




Scalable Vector Graphics Tutorial

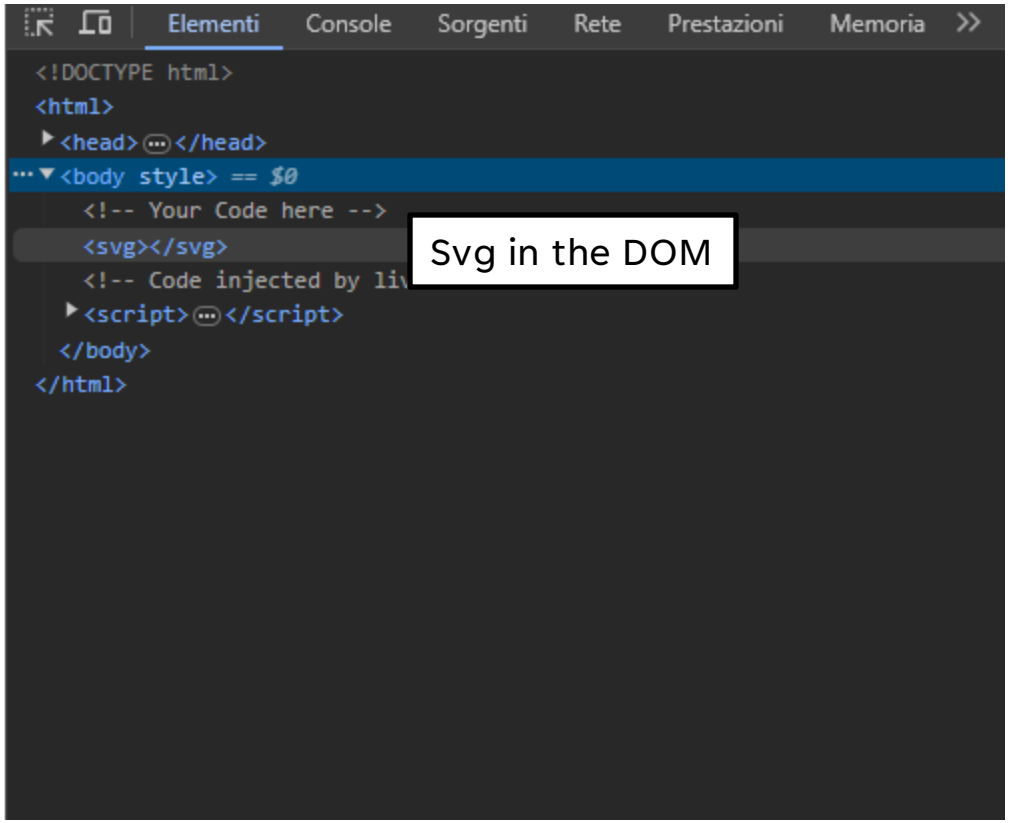
What you will learn from this tutorial:

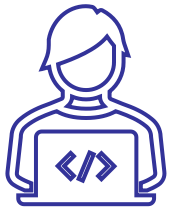
- Initialize a **responsive <svg> container**
- **Drawing**:
  - Lines
  - Rectangles
  - Circles and Ellipses
  - Paths
- **Adding Text** to an SVG
- **Grouping** elements



Responsive SVG

- In the environment SVG graphics, the `<svg>` container is where we «draw».
- If you are following the lesson or the written tutorial, you have now something like this in the browser (F12 or rightClick>Inspect):

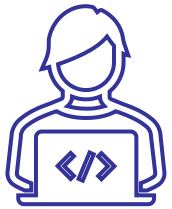




Responsive SVG

- To change the standard `weight` and `height` of the `<svg>` container, we can use **attributes**:
  - In HTML, attributes are used to provide additional information about elements.

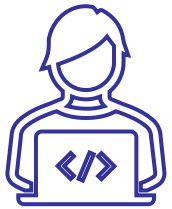
- In this case, if we simply change these two parameters by setting some values for the pixels, we will have a not so well desired behaviour:
  - If we resize the browser window, the container will shrink accordingly!
  - [Try it!]



Responsive SVG

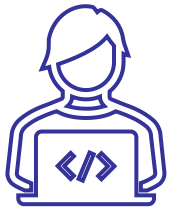
- To make **responsive SVG elements** we can use the attribute **`viewBox`**.
- It consists of a list of values where:
  - The first two numbers specify the origin of the viewBox (*x*- and *y*-coordinates*)*
    - *We usually set them to 0 0*
  - The last two numbers are the *width* and the *height*.

- The `viewBox` keeps the ratio between *width* and *height* constant. So, if the browser window shrinks, the box shrinks accordingly.



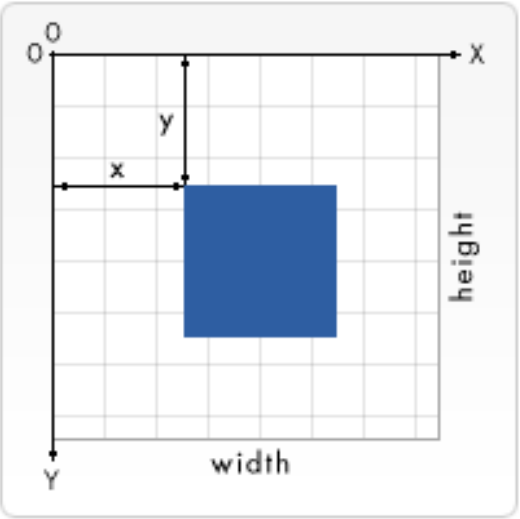
Responsive SVG

- By now, the container of our `<svg>` was the `<body>` element.
- However, if the browser's viewport becomes very large, the `<svg>` becomes very large as well.
- So, we wrap a `<div>` element around the svg and we set the **`max-width`** and **`width`** attributes of the `<div>`.
  - In this way, the svg element is contained by fixed values.



SVG coordinates

- The SVG coordinates system is like the cartesian system, but the y-axis goes from top to bottom.
- The origin of the system is in the top-left corner.
- Remember this when you need to position your shapes in the svg!



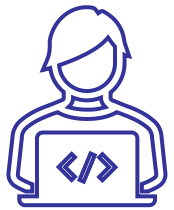
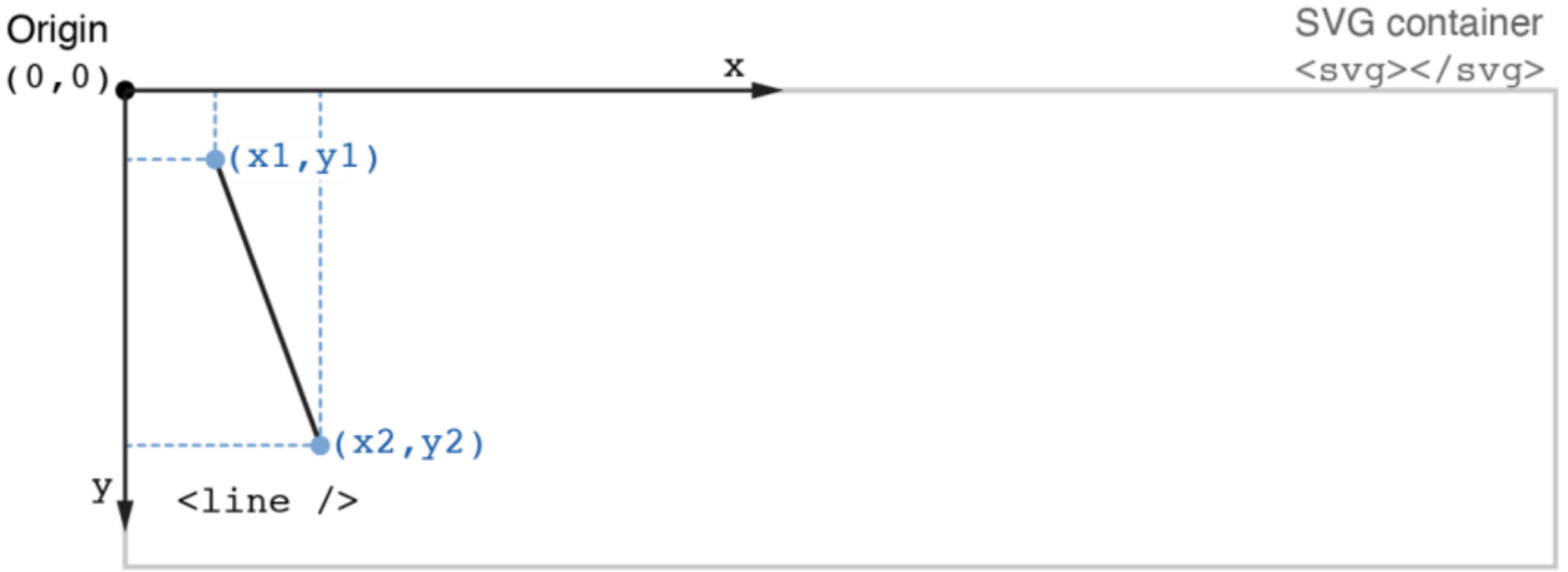


Figure from: https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Positions

<line/> element

- The **<line/>** element is the simplest svg shape.
- It takes four values:
  - x1 and y1 for the position of the first point
  - x2 and y2 for the position of the second point
- The stroke attribute is required to give a thickness to the line



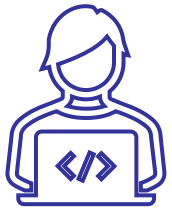
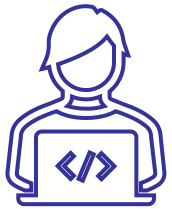


Figure from: D3.js in Action, Third Edition - p. 20

`<rect/>` element

- The **`<rect/>`** element requires four attributes to be visible.
  - `x` and `y` are the position of the top-left corner
  - `width` and `height` are the dimensions of the shape
- Then one can play with these other attributes:
  - `fill` and `fill-opacity` to set the color and the alpha for the inner part
  - `stroke`, `stroke-width` and `stroke-opacity` to set, respectively, the color, the thickness and the opacity of the border
- Bonus: if you want your rectangle to have rounded corners, just add the two attributes `rx` and `ry`.



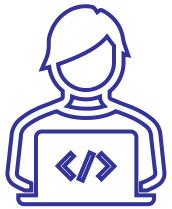
<rect/> element

```
fill="#81c21c"
```

```
fill="#81c21c"
stroke="#000"
```

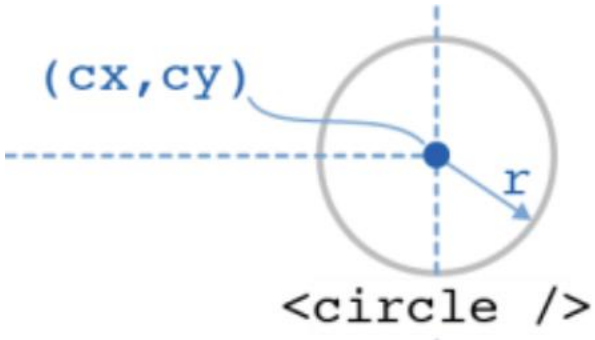
```
fill="none"
stroke="#0060b1"
stroke-width="3"
```

```
fill="#0060b1"
fill-opacity="0.3"
stroke="#0060b1"
stroke-width="3"
stroke-opacity="0.6"
```



<circle/> element

- The **<circle/>** element requires three attributes.
  - x and y are the coordinates of the position of the center
  - r is the size of the radius
- The previously introduced attributes can still be used to alter the color of the inner part or the border of the circle.



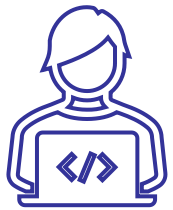
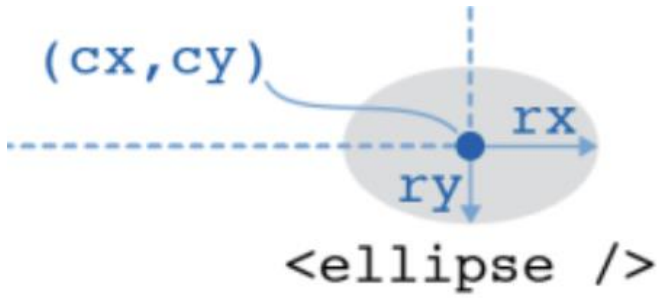


Figure from: D3.js in Action, Third Edition - p. 24

<ellipse/> element

- The **<ellipse/>** element requires one more attribute than the <circle/>.
  - x and y are still the coordinates of the position of the center
  - rx is the size of the horizontal radius and ry is the size of the vertical radius
- The previously introduced attributes can still be used to alter the color of the inner part or the border of the circle.



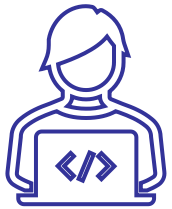
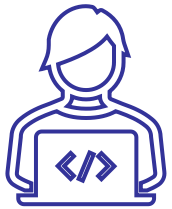


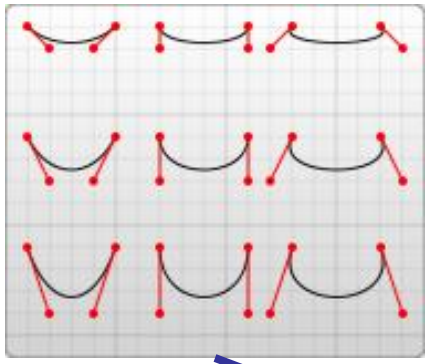
Figure from: D3.js in Action, Third Edition - p. 24

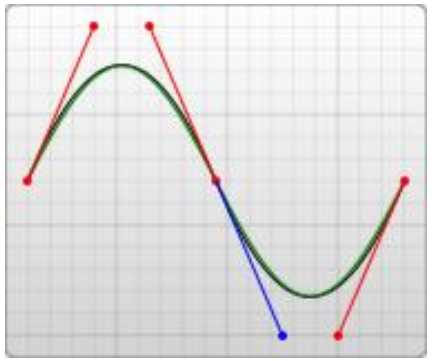
<path/> element

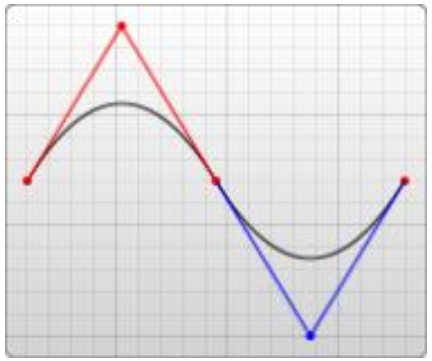
- The **<path/>** element is (by far) the most flexible svg element.
- In D3, it is extensively used to draw any complex shape that cannot be represented by the previous primitives.
- To draw a <path/> we give instructions to the browser by setting its d attribute (which stands for *"draw")*
- The d attribute contains a list of commands to specify the starting and the ending point, the curves to be used, the colors, and so on.
  - The exhaustive comprehension of the <path/> element is behind the scope of this course.
  - If you want to learn more about it, please refer to the MDN documentation at this link

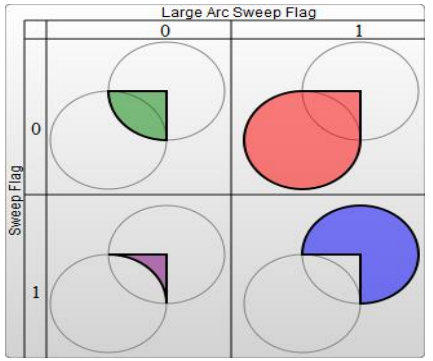


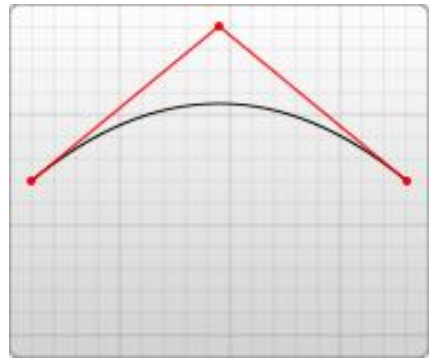
<path/> element



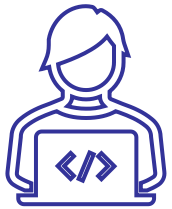








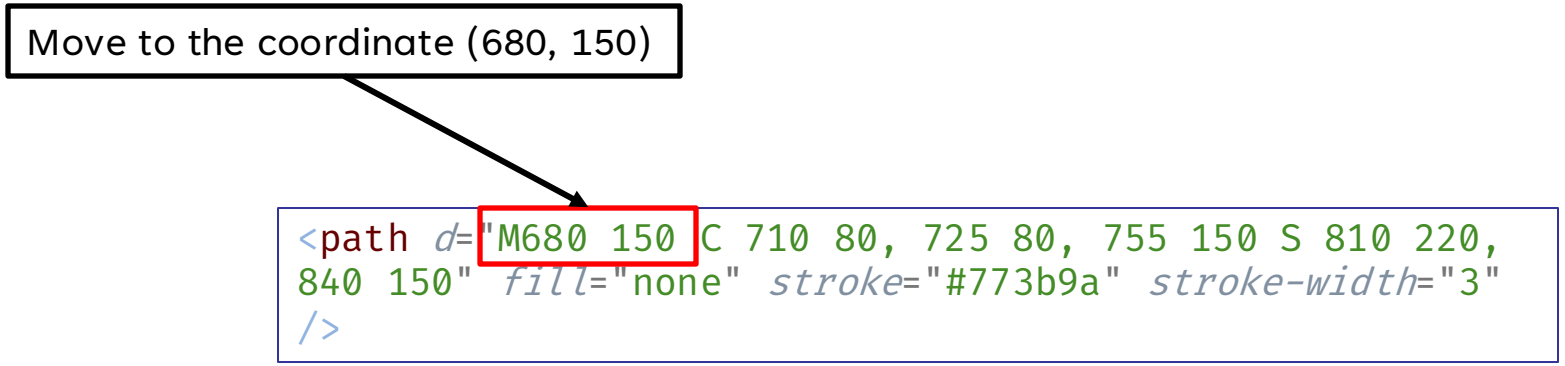
```svg
<svg width="190" height="160" xmlns="http://www.w3.org/2000/svg">
<path d="M 10 10 C 20 20, 40 20, 50 10" stroke="black" fill="transparent"/>
<path d="M 70 10 C 70 20, 110 20, 110 10" stroke="black" fill="transparent"/>
<path d="M 130 10 C 120 20, 180 20, 170 10" stroke="black" fill="transparent"/>
<path d="M 10 60 C 20 80, 40 80, 50 60" stroke="black" fill="transparent"/>
<path d="M 70 60 C 70 80, 110 80, 110 60" stroke="black" fill="transparent"/>
<path d="M 130 60 C 120 80, 180 80, 170 60" stroke="black" fill="transparent"/>
<path d="M 10 110 C 20 140, 40 140, 50 110" stroke="black" fill="transparent"/>
<path d="M 70 110 C 70 140, 110 140, 110 110" stroke="black" fill="transparent"/>
<path d="M 130 110 C 120 140, 180 140, 170 110" stroke="black" fill="transparent"/>
</svg>
```

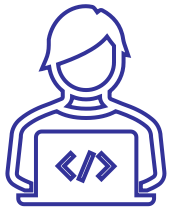


Figures from: MDN-Paths

<path/> element

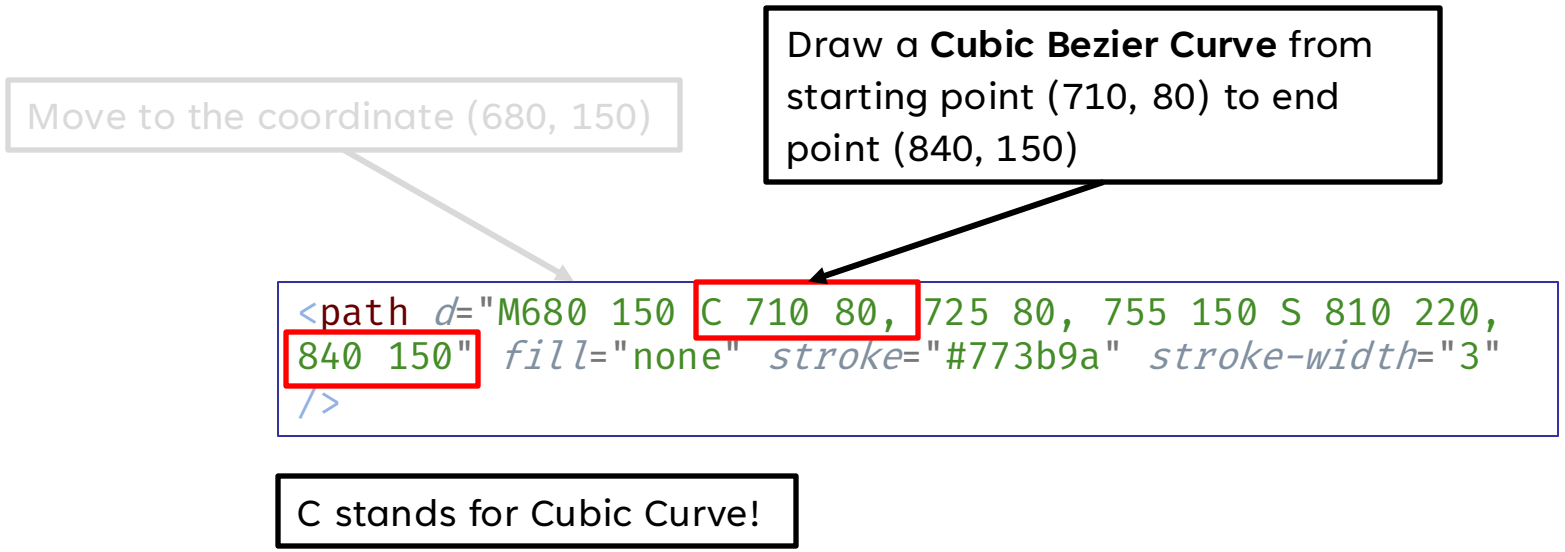
- In the written guide, we are going to do the following example:

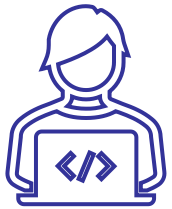




<path/> element

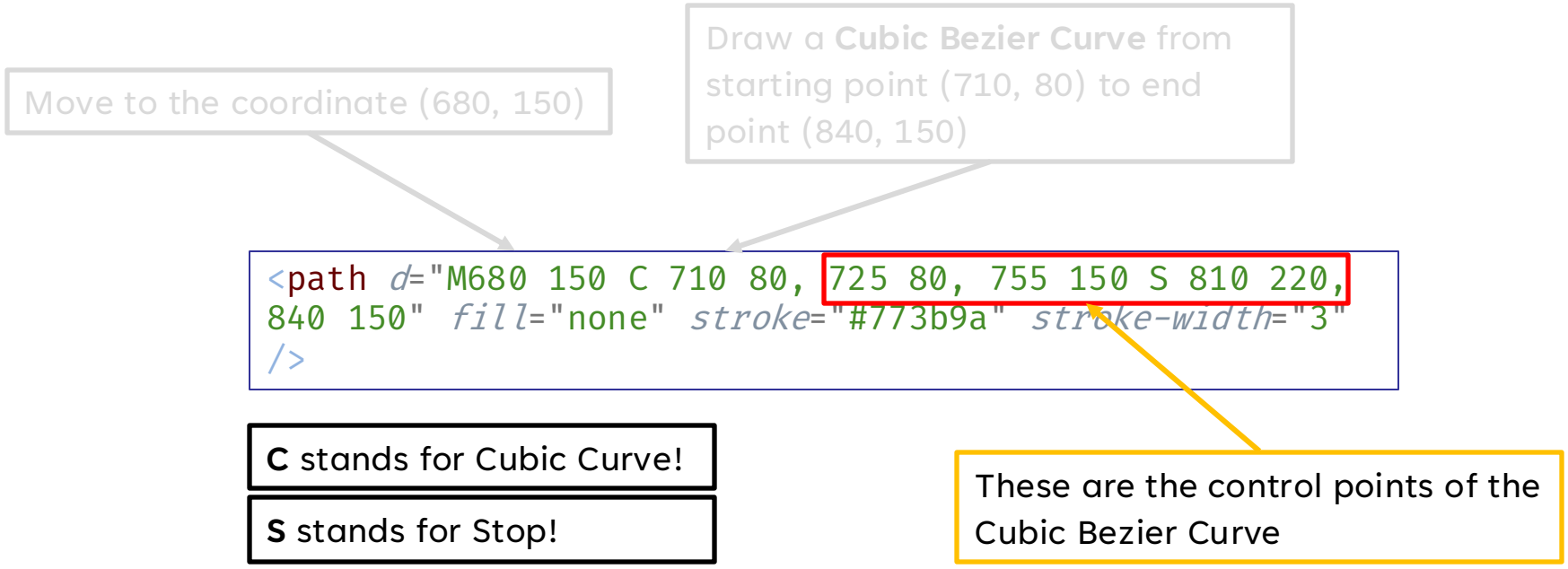
- In the written guide, we are going to do the following example:

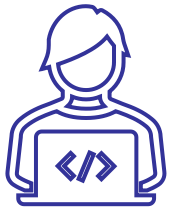




<path/> element

- In the written guide, we are going to do the following example:





<path/> element

Move to the coordinate (680, 150)

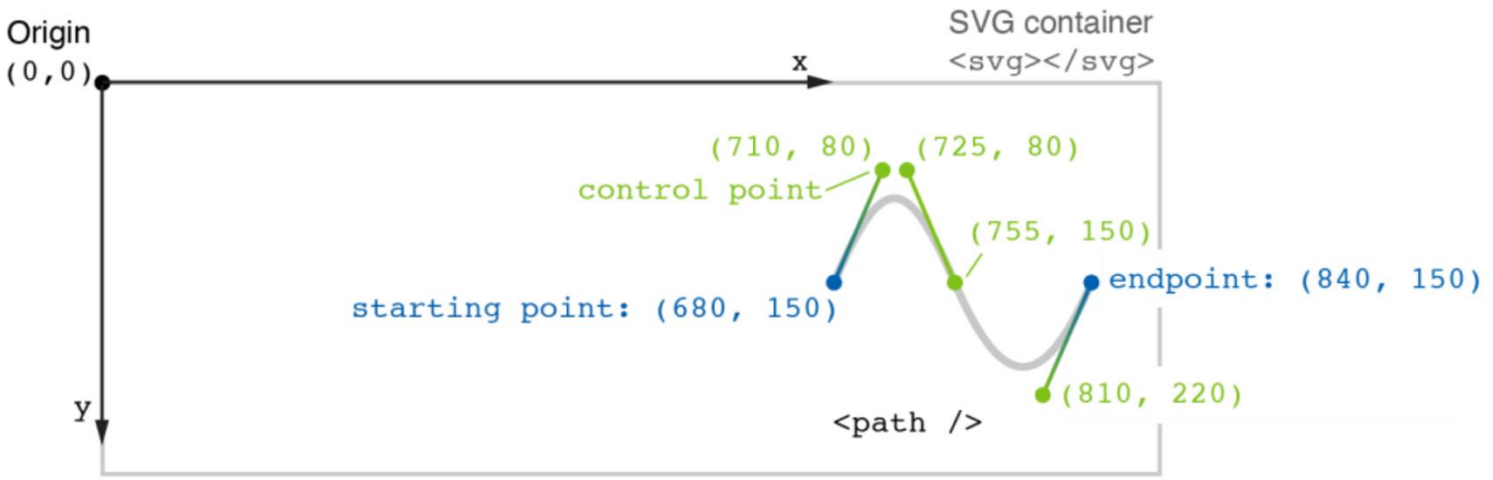
Draw a **Cubic Bezier Curve** from starting point (710, 80) to end point (840, 150)

```
<path d="M680 150 C 710 80, 725 80, 755 150 S 810 220, 840 150"
fill="none" stroke="#773b9a" stroke-width="3" />
```

C stands for Cubic Curve!

S stands for Stop!

These are the control points of the Cubic Bezier Curve



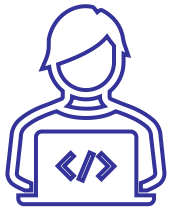
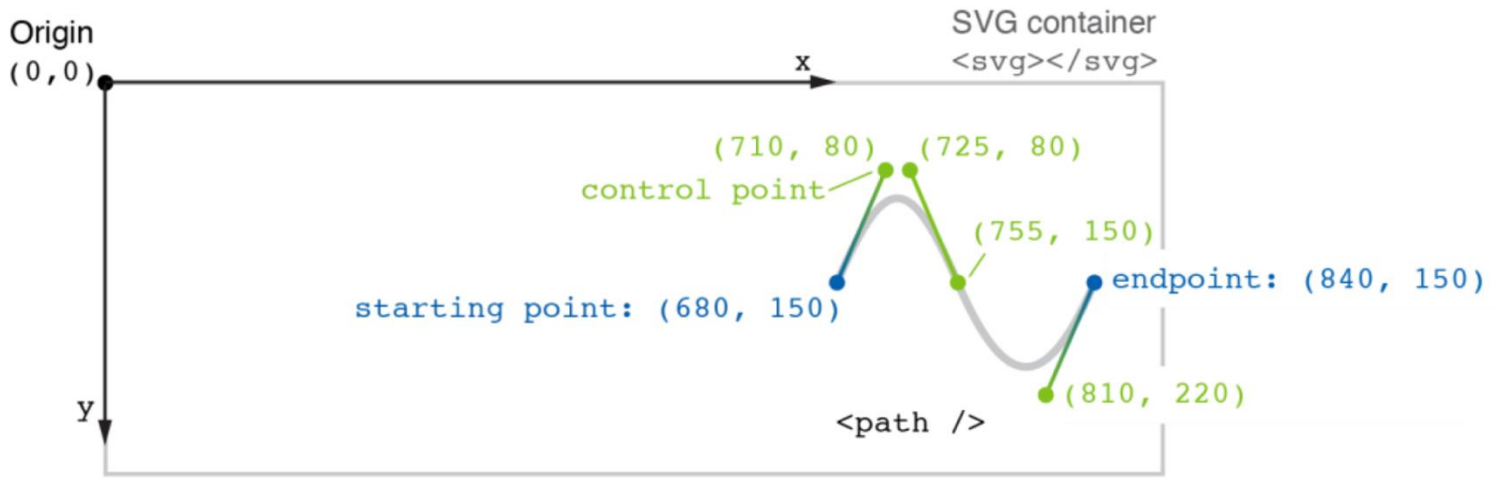


Figure from: D3.js in Action, Third Edition - p. 25

<path/> element

fill="none" is required not to draw the filling of the path (try to set it to a color and check what happens)

```
<path d="M680 150 C 710 80, 725 80, 755 150 S 810 220, 840 150"
fill="none" stroke="#773b9a" stroke-width="3" />
```



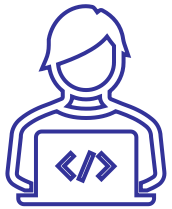
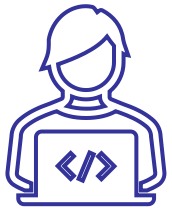


Figure from: D3.js in Action, Third Edition - p. 25

More on the `<path/>` element

- If you want to draw the shape of simple paths, then you can do it as we just showed.
- However, for more complex shapes, this can become very tedious.
- Fortunately, **D3** offers a wide variety of predefined functions to draw complex shapes.
- In the following lecture, we will see some of them:
  - `d3.line()` to draw lines by specifying the points of the path
  - `d3.area()` to draw the area given a specific data points



CSS

- **CSS** stands for **Cascading Style Sheets** and is the language that describes how DOM elements are displayed on the screen and how they look like.
- In **D3 projects**, we generally apply CSS styles using inline-styles or via an external stylesheet.
- Remember that inline-styles take precedence over the ones applied from an external stylesheet.
- More on CSS later, when we introduce D3 selections.

JavaScript

- D3 is a JS library
  - It adds new methods to the core features of native JS

To understand D3, you need to know two key JavaScript concepts:

- **Method Chaining**
- **Object Manipulation**

JavaScript: Method Chaining

- **Method Chaining** is a JS technique to keep the code concise, readable, and clean.
- It consists of calling a method after another in a long **chain** (whence the name):
  - Every output of a previous method is taken as input of the following one.
- Each call is separated by a dot
- The methods are executed in the order in which they are chained.

```
d3.selectAll("div").append("p").attr("class", "p-class").text("Hello").append("span").text("Hi").style("font-weight", "600");
```

JavaScript: Method Chaining

- In D3, it is common to break lines for readability's sake.
  - This is where the auto-formatter **Prettier** comes in handy!
- We pass from this:

```
d3.selectAll("div").append("p").attr("class", "p-class").text("Hello").append("span").text("Hi").style("font-weight", "600");
```

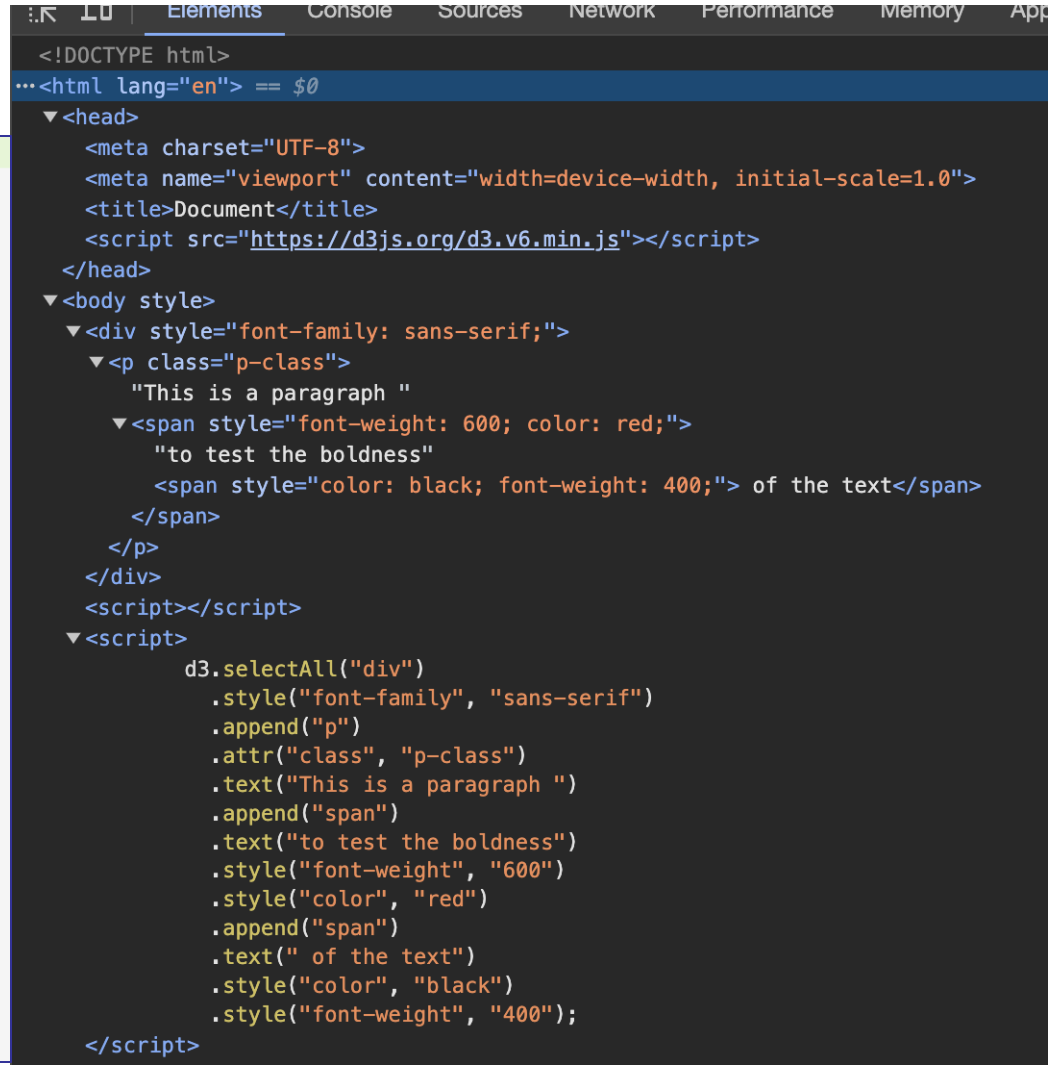
- To this:

```
d3.selectAll("div")
.append("p")
.attr("class", "p-class")
.text("Hello")
.append("span")
.text("Hi")
.style("font-weight", "600");
```

JavaScript: Method Chaining example

This is a paragraph **to test the boldness** of the text **output**

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>Document</title>
7      <script src="https://d3js.org/d3.v6.min.js"></script>
8    </head>
9    <body>
10     <div></div>
11     <script></script>
12     <script>
13       d3.selectAll("div")
14         .style("font-family", "sans-serif")
15         .append("p")
16         .attr("class", "p-class")
17         .text("This is a paragraph ")
18         .append("span")
19         .text("to test the boldness")
20         .style("font-weight", "600")
21         .style("color", "red")
22         .append("span")
23         .text(" of the text")
24         .style("color", "black")
25         .style("font-weight", "400");
26     </script>
27   </body>
28 </html>
29
```



code vs **DevTools HTML**: Can you spot the differences?

JavaScript: Array Manipulation

- Understanding how to access **data** and manipulate it for a **data** visualization tool as D3 is crucial.
- We start by first talking about JS **arrays**.

```
const arrayOfNumbers = [1, 2, 3, 4, 5];
const arrayOfStrings = ["one", "two", "three", "four"];
```

- If we want a specific entry in the array:

```
arrayOfNumbers[0]                    // 1
arrayOfStrings[3]                    // "four"
```

JavaScript: Array Manipulation

- Each array has a property called `length` which is the number of objects inside of it.
- Remember that arrays in JS are **zero-indexed**, meaning the index starts from `0`.

```
arrayOfNumbers.length; // 5
arrayOfNumbers[arrayOfNumbers.length]; // undefined
arrayOfNumbers[arrayOfNumbers.length - 1]; // 5
```

- However, most of the real-world datasets are not a simple collection of numbers and strings (unfortunately).

JavaScript: Object Manipulation

- Imagine that we have a set of records like this one:

Name	Surname	PhD
Giuseppe	Liotta	True ☺
Alessandra	Tappini	True ☺
Tommaso	Piselli	False ☹

- Then each data point has this form:

```
const row1 = {
name: "Giuseppe",
surname: "Liotta",
PhD: True
}
```

```
const row2 = {
name: "Alessandra",
surname: "Tappini",
PhD: True
}
```

```
const row3 = {
name: "Tommaso",
surname: "Piselli",
PhD: False
}
```

JavaScript: Object Manipulation

- We can access each property of the object with the **dot notation** or with the **bracket notation**.

```
const row3 = {
name: "Tommaso",
surname: "Piselli",
PhD: False
}
```

```
row3.name; // "Tommaso"
row3["name"]; // "Tommaso"
row3.PhD; // False
```

- In D3, we will usually load the dataset in a variable called `data`

```
const data = [
{ name: "Giuseppe", surname: "Liotta", PhD: True },
{ name: "Alessandra", surname: "Tappini", PhD: True },
{ name: "Tommaso", surname: "Piselli", PhD: False },
];
```

JavaScript: Object Manipulation

- We can then iterate through each element of the dataset, usually called **datum**, with a loop.
- One commonly used iterator in JS is the `forEach` loop.

```
data.forEach((d) => {console.log(d.name);}); // Giuseppe, Alessandra, Tommaso
```

- Another very handy iterator is `map`.
  - This is used to quickly create array from data.

```
data.map(d => d.surname); // ["Liotta", "Tappini", "Piselli"]
```

- If we want to search for a data in the dataset, we can use `find`.

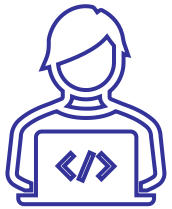
```
data.find(d=>d.name==="Giuseppe"); // {name: "Giuseppe", surname: "Liotta", PhD: True}
```

- If we want to isolate an attribute from the dataset, we can use `filter`

```
data.filter(d => d.PhD); // [{name: "Giuseppe", surname: "Liotta", PhD: True},
{name: "Alessandra", surname: "Tappini", PhD: True}]
```

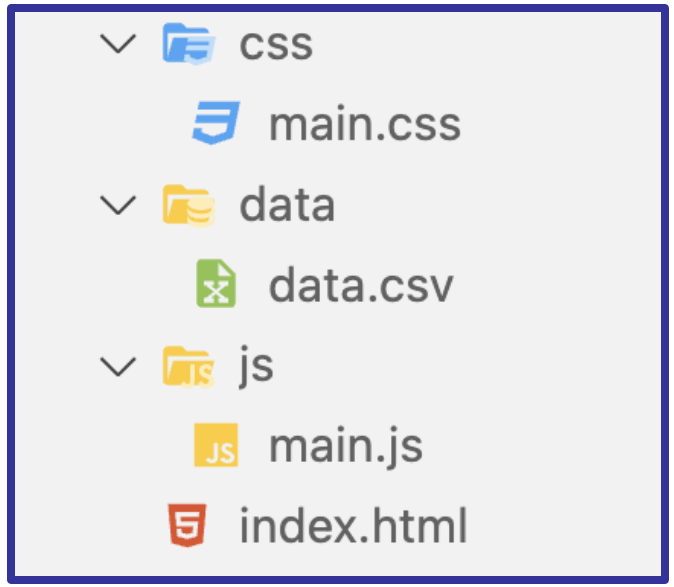
Manipulating the DOM

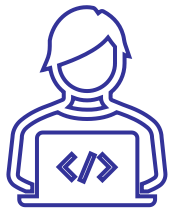
D3 basics



D3: First steps

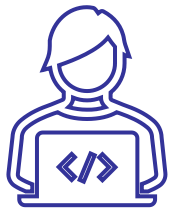
- As we already mentioned several times, D3 is a data visualization library of JavaScript, so the best way to understand it is by making a data visualization project.
- In your local repo, go to `L1>2_d3_basics>start`, you will have a folder structured as the picture on the right.





D3: First steps

- There are two methods to load D3 into your project.
- **First Method:** add a script tag to the `index.html` file that links the latest version of D3.
  - **Very simple**
  - **Requires Internet**
- **Second Method:** load D3 as a Node module using NPM
- Third Method (FORBIDDEN): download the zip and put it in the same directory of your project.



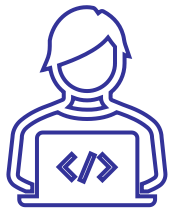
Selecting elements

- With D3, we manipulate the DOM by **selecting** elements.
- D3 has two different selections:
  - `d3.select()` takes a selector as a parameter and returns the **first** element that matches the selection.
  - `d3.selectAll()` takes a selector as a parameter and returns **all** the elements that match the selection.

```
d3.select("selector");      // standard Selector
d3.select(".class");        // class Selector
d3.select("#id");           // id Selector
```

```
d3.selectAll("h1, .intro");
```

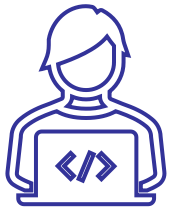
- Like in CSS, we can group multiple selectors separated by a comma.



Storing selected elements

- It is a common practice in D3 to store selected elements into **JavaScript variables**.
- In this way, we don't have to reselect them when we need them.

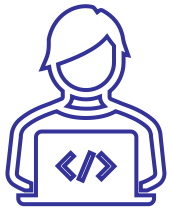
```
const svg = d3.select('svg');
```



Adding elements to selections

- After we have selected an element, we can **append** new elements to it.
- The `append()` method adds a new element as the **last child of the selection**.
  - It takes the type, the name, or the tag of the element as a parameter.

```
selection.append("type");
```



Adding elements to selections

```
selection.append("type");
```

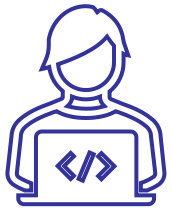
- If we want to add a `rect` to our `svg`, we can simply write:

```
const svg = d3.select('svg');
svg.append('rect');

d3.select('svg').append('rect'); // with chaining
```

- If we want to add a `<p/>` to every `<div/>`:

```
d3.selectAll('div').append('p');
```



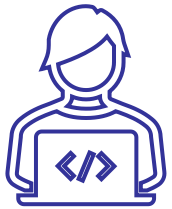
Setting and modifying attributes

- **Attributes** can be set and modified with the D3 method `.attr()`

```
selection.attr("name of attribute", "value of attribute");
```

- For example, if we want to set the class of every `<p/>` element in our DOM to `paragraph`, we can write:

```
d3.selectAll('p').attr('class', 'paragraph');
```



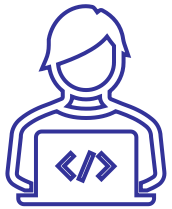
Setting and modifying styles

- **Styles** can be set and modified with the D3 method `.style()`

```
selection.style("name of style property", "value");
```

- For example, if we want to set a border for our `<svg/>` element, we can write:

```
d3.select("svg").style("border", "2px solid black");
```



Bibliography

- All the slides are based on “D3.js in Action, Third Edition” book
- MDN for anything related with the Web
- Slides by E. Di Giacomo and F. Montecchiani