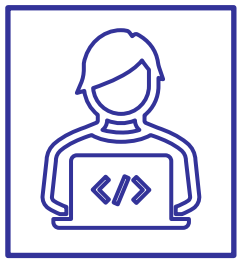
D3: Working with Data

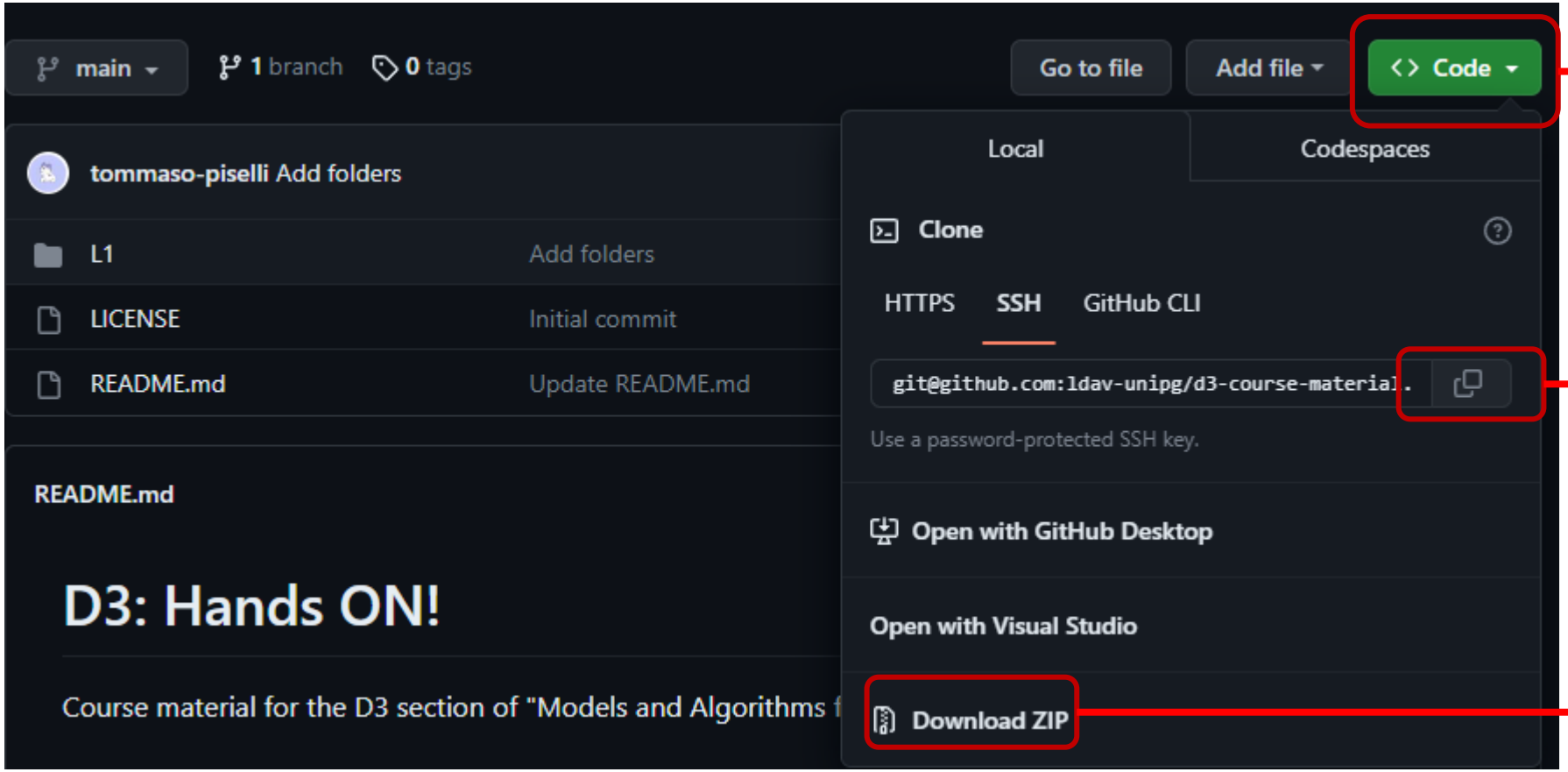
Tommaso Piselli

Recall: 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.
- if you see a **blue coder icon** in the current slide (like the one in the bottom left corner), then it means that in that slide we are producing some code (and you are highly invited to try it and code 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.



Recall: How to access the code files

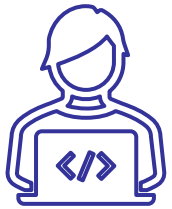


1. Click the Code button

2.a. Copy the url and then, from your terminal, launch a `git clone <url>`

OR

2.b. Download the files



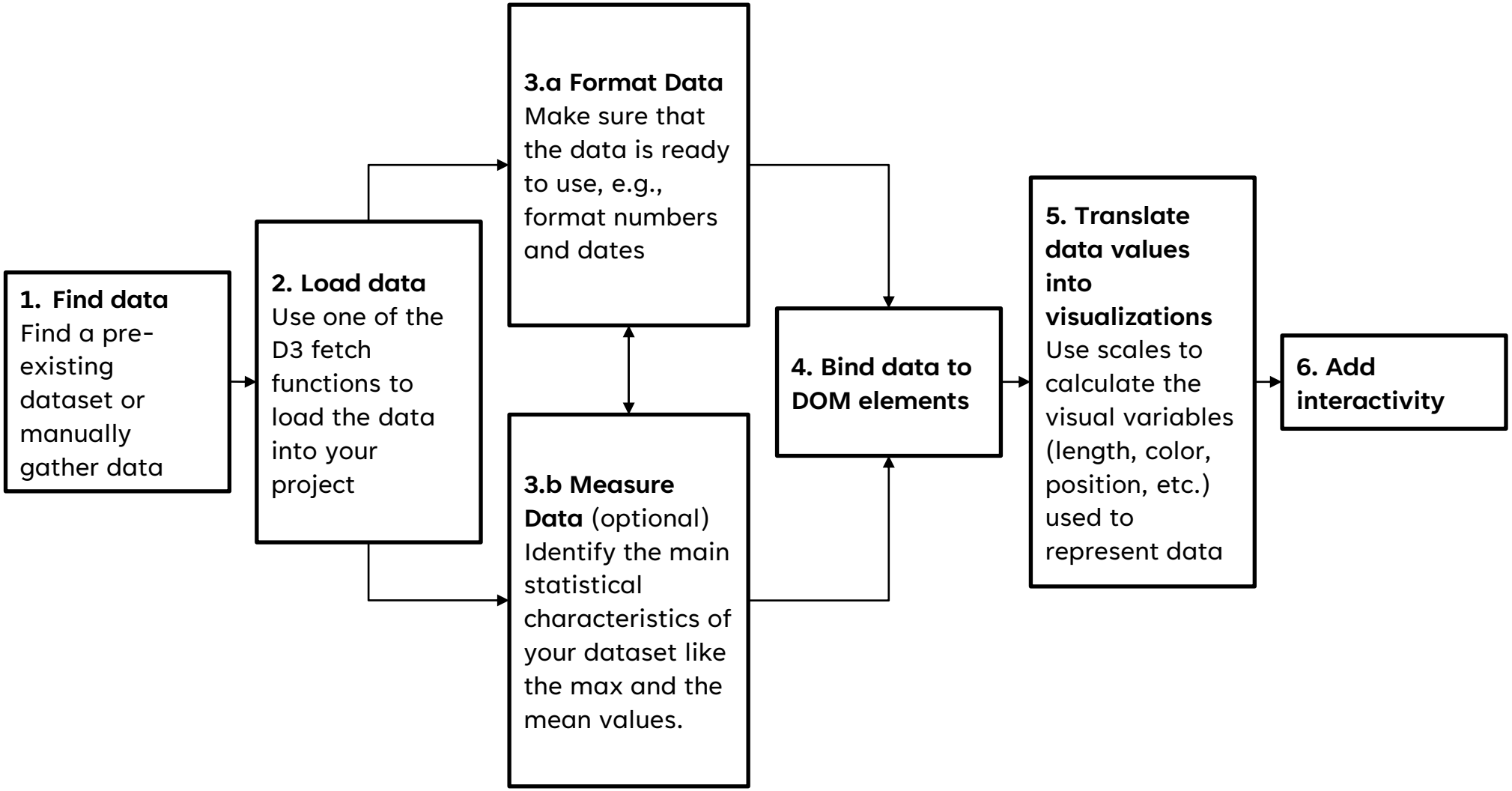
Recall: Course Material

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

Other Material:

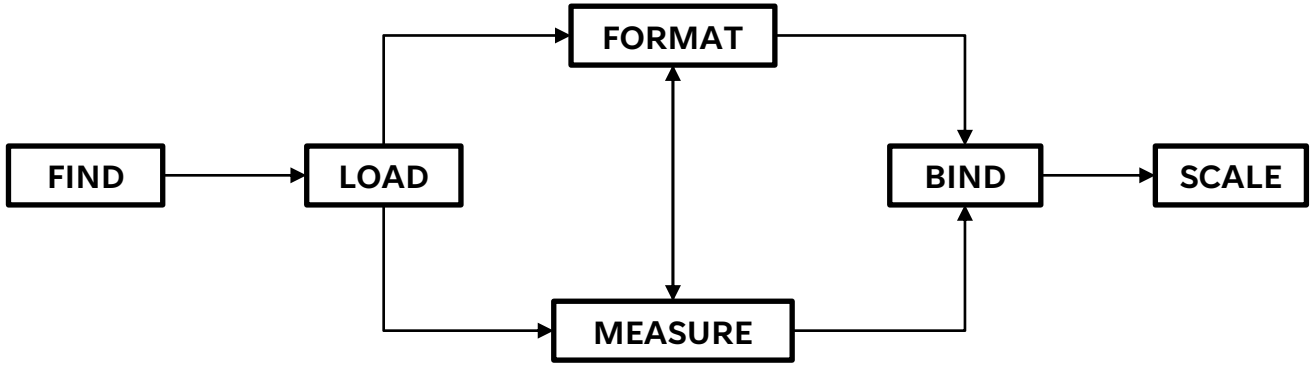
- MDN for anything related the Web
- “Programmazione per Internet e Web” material
- All the slides are based on “D3.js in Action, Third Edition” book

Lesson Topic



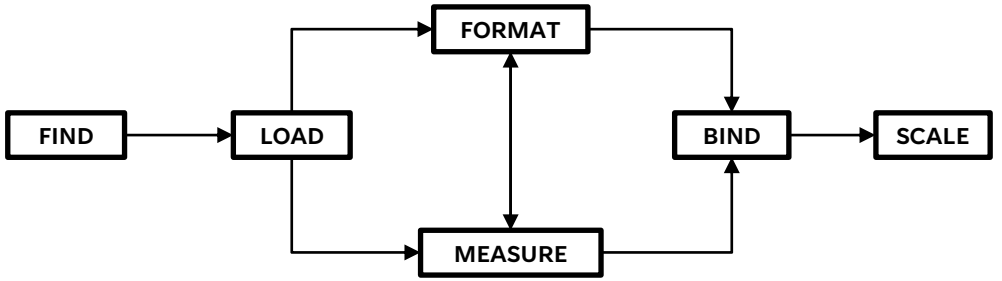
From «D3.js in Action» (third edition)

Lesson Topic



Lesson Topic

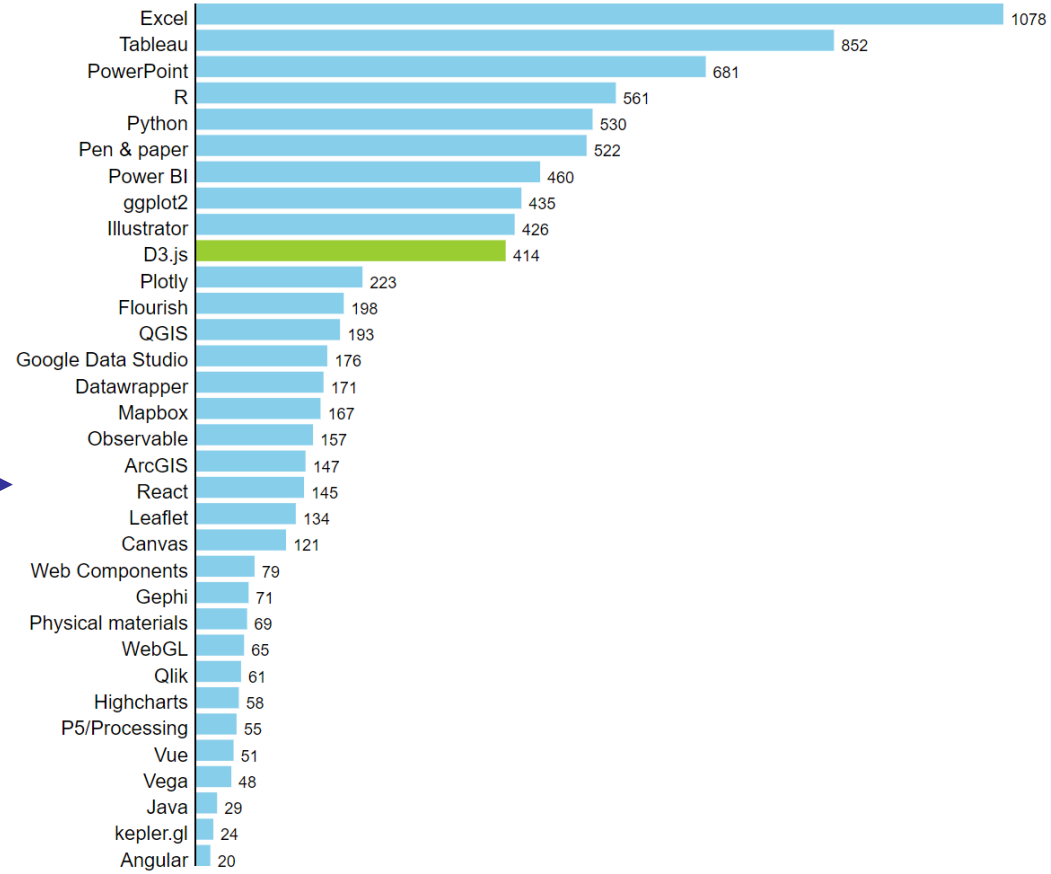
1. In the first half, we will discuss a **data workflow** that can be applied to D3 projects (and, in general, to any data visualization project).
   - **Finding (and cleaning) Data.**
   - **Load Data to a D3 project.**
   - **Formatting the Dataset.**
   - **Measures.**
   - **Binding Data to a Visualization.**
   - **Scale the Data to fit the screen.**



2. In the second half of the lesson we will draw a more complex visualization using **svg paths** and some D3 functions.

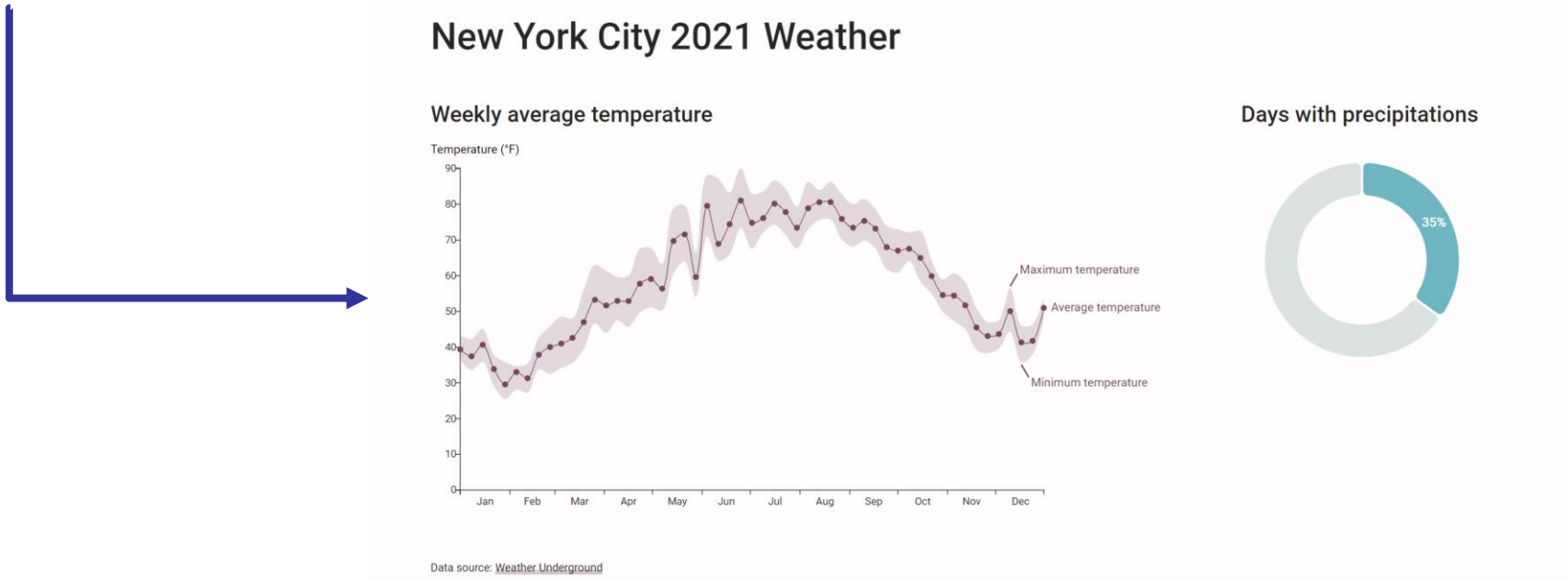
Lesson Topic

- As we present these topics, we will also produce a couple of visualizations.
- This one for the first half



Lesson Topic

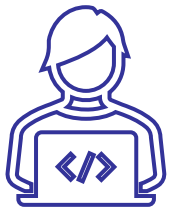
- As we present these topics, we will also produce a couple of visualizations.
- This one for the first half
- And this for the second half



D3: Data Handling

For a written guide of this tutorial, please refer to the **README.md** file in the **L3** folder. You can open the file either in VSCode or Github.

This tutorial was taken from Chapter 3 of “D3.js in Action, Third Edition”



Finding Data

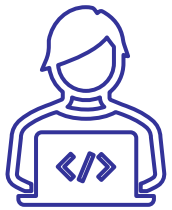
Some useful links for open datasets.

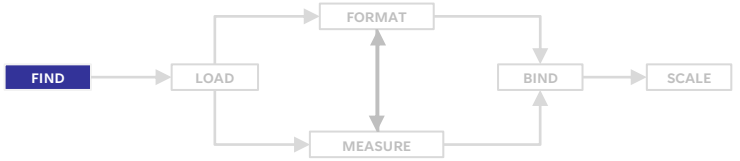
Mostly non-relational:

- https://www.cs.ubc.ca/group/infovis/resources.shtml#data-repos
- https://data.world/datasets/open-data
- https://www.kaggle.com/datasets
- https://www.tableau.com/learn/articles/free-public-data-sets

Relational:

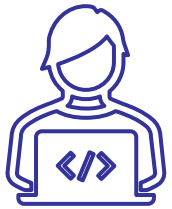
- https://visdunneright.github.io/gd_benchmark_sets/
- Graph Drawing contest (*creative topic* of the various editions):
  - https://mozart.diei.unipg.it/gdcontest/history/

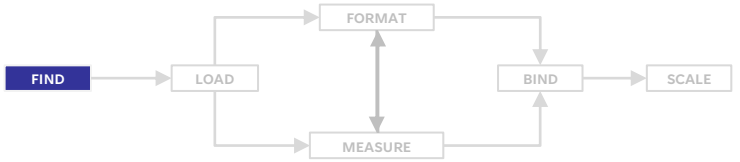




Finding Data

- In data visualization, there are two main types of data:
  - **Quantitative**: numerical information (e.g., temperature)
  - **Qualitative**: textual information (e.g., names)
- Using D3, we will work with **different formats** of datasets.
- The most common ones are:
  - Tabular dataset => csv files
  - JavaScript Objects => JSON files





Finding Data

Data Types

Quantitative

Numerical information:
Temperature, speed, …

Discrete

Integers that cannot be subdivided.
School grades

Continuous

Values that can be divided into smaller units.
Amount of rain drop

Qualitative

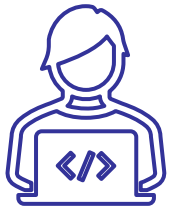
Non-numerical information:
Text, images, …

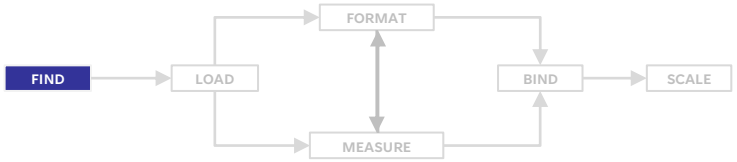
Nominal

Values that don't have a specific order.
Gender

Ordinal

Values that can be organized in order of magnitudes.
T-shirt sizes: S, M, L

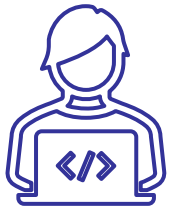


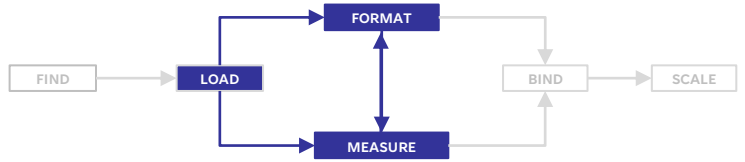


Preparing Data

Once we have the dataset ready and clean, we:

- **Load** it into D3,
- **Format** everything,
- **Measure** different aspects of the data.

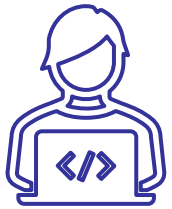


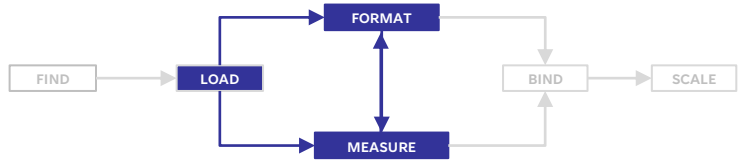


Preparing Data

Once we have the dataset ready and clean, we:

- **Load** it into D3,
- **Format** everything,
- **Measure** different aspects of the data.

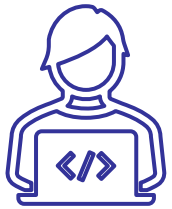


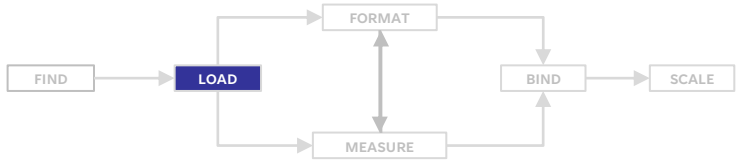


Loading Data

- D3 has different functions to load data into the project, depending on their type:
  - `d3.csv(filePath)`
  - `d3.json(filePath)`
  - and many more (see, for reference, the d3-fetch docs)
- As it loads, D3 transforms the dataset into an array of objects.

```
d3.csv("./data/data.csv", (d) => {console.log(d);});
```

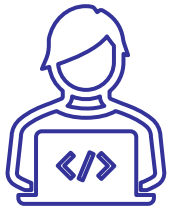


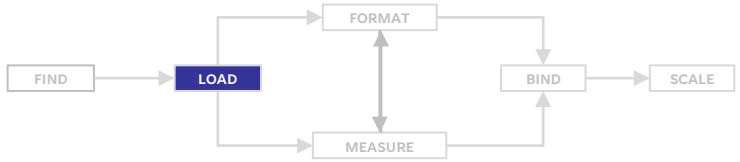


Loading Data

- Loading is an **asynchronous process**.
  - This means that the browser continues to read and execute the script while the data is being loaded.
  - To wait for the data to be fully available we can use the `then()` method for JS Promises.
  - The callback function of `then()` gives us access to the entire dataset once it is loaded.

```
d3.csv("./data/data.csv", (d) => {
// ...
}).then((data) => {
console.log(data);
});
```

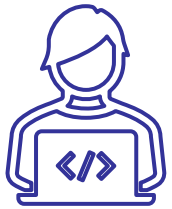


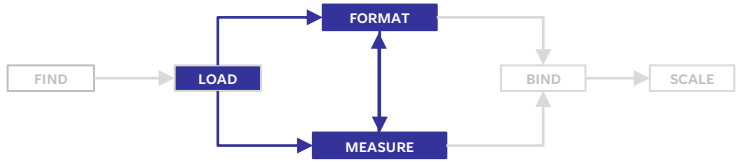


Preparing Data

Once we have the dataset ready and clean, we:

- **Load** it into D3,
- **Format** everything,
- **Measure** different aspects of the data.





Formatting Data

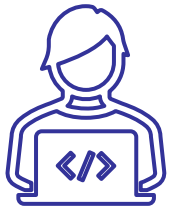
- The callback function of `d3.csv()` is called the **row conversion** function:
  - It gives access to data **row by row**.
- So, the previous example has this output in the console:

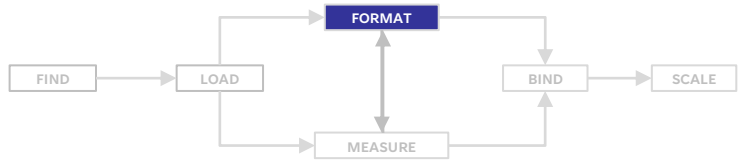
```
Live reload enabled.
▼ Object i
    count: "147"
    technology: "ArcGIS"
  ▶ [[Prototype]]: Object
▼ Object i
    count: "414"
    technology: "D3.js"
  ▶ [[Prototype]]: Object
▼ Object i
    count: "20"
    technology: "Angular"
  ▶ [[Prototype]]: Object
▼ Object i
    count: "171"
    technology: "Datawrapper"
  ▶ [[Prototype]]: Object
▶ Object
▶ Object
▶ Object
▶ Object
▶ Object
```

```
d3.csv("./data/data.csv", (d) => {console.log(d);});
```

Where the `data.csv` file has these rows

```
technology,count
ArcGIS,147
D3.js,414
Angular,20
Datawrapper,171
Excel,1078
Flourish,198
ggplot2,435
Gephi,71
Google Data Studio,176
```





Formatting Data

- The callback function of `d3.csv()` is called the **row conversion** function:
  - It gives access to data **row by row**.
- So, the previous example has this output in the console:

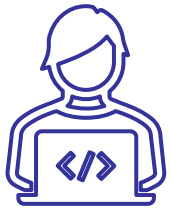
```
Live reload enabled.
▼ Object i
    count: "147"
    technology: "ArcGIS"
  ▶ [[Prototype]]: Object
▼ Object i
    count: "414"
    technology: "D3.js"
  ▶ [[Prototype]]: Object
▼ Object i
    count: "20"
    technology: "Angular"
  ▶ [[Prototype]]: Object
▼ Object i
    count: "171"
    technology: "Datawrapper"
  ▶ [[Prototype]]: Object
▶ Object
▶ Object
▶ Object
▶ Object
▶ Object
```

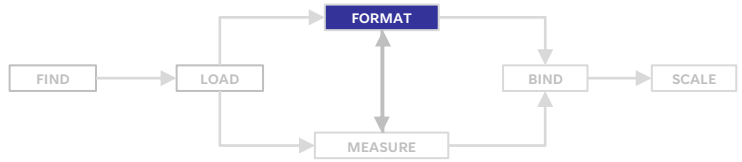
```
d3.csv("./data/data.csv", (d) => {console.log(d);});
```

Where the `data.csv` file has these rows

```
technology,count
ArcGIS,147
D3.js,414
Angular,20
Datawrapper,171
Excel,1078
Flourish,198
ggplot2,435
Gephi,71
Google Data Studio,176
```

Note: values from `count` are parsed as strings instead of numbers





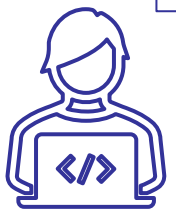
Formatting Data

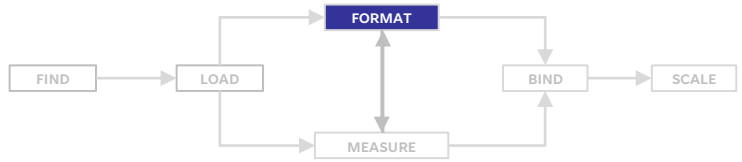
- Since the callback function gives us the data one row at a time, it is a great place to start doing some formatting.
- In this way, data will be ready to use for any visualization.

```
d3.csv("./data/data.csv", (d) => {
console.log(`technology: ${d.technology}, count:
${+d.count}`);
});
```

- Here, we use a key-value formatting for the data.
- This is very handy if you need to clean up your dataset from useless information stored in some columns.
- However, if you need them all, you can wait to load the whole dataset before doing the formatting.

```
technology: ArcGIS, count: 147
technology: D3.js, count: 414
technology: Angular, count: 20
technology: Datawrapper, count: 171
technology: Excel, count: 1078
technology: Flourish, count: 198
technology: ggplot2, count: 435
technology: Gephi, count: 71
technology: Google Data Studio, count: 176
technology: Highcharts, count: 58
technology: Illustrator, count: 426
```

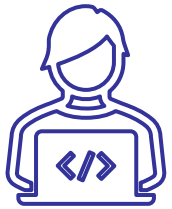


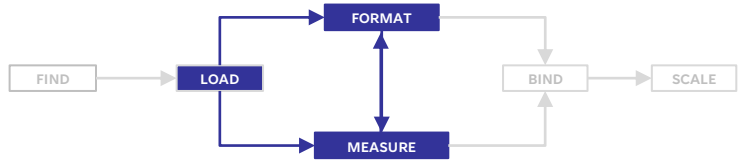


Preparing Data

Once we have the dataset ready and clean, we:

- **Load** it into D3,
- **Format** everything,
- **Measure** different aspects of the data.

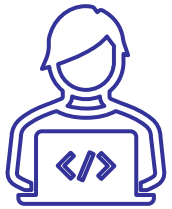


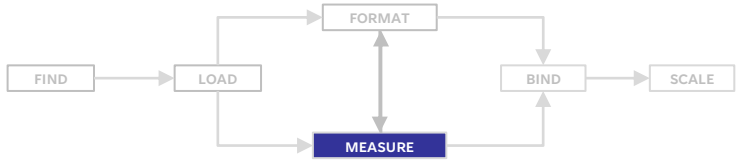


Measuring Data

- Now, we can finally see the JS Promise in Action.
- Once the data is loaded, the **Promise is fulfilled**.
- The dataset is available in the callback of the `then()` method.
- Instead of just logging our formatted rows, we first **create** a JS object, storing the key-value pairs.
- Then, if we want to log the complete dataset of our example, we can write:

```
d3.csv("./data/data.csv", (d) => {
return { technology: d.technology, count: +d.count };
}).then((data) => { console.log(data);});
```

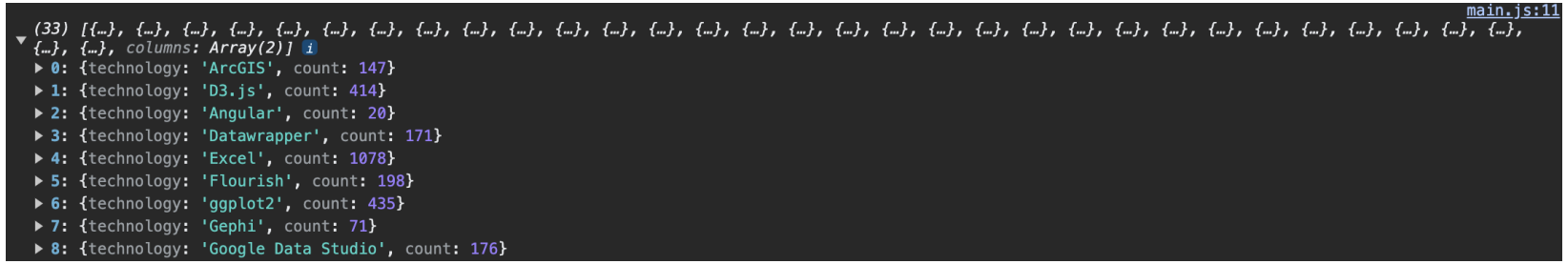


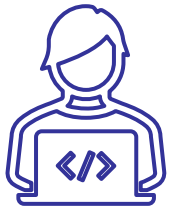


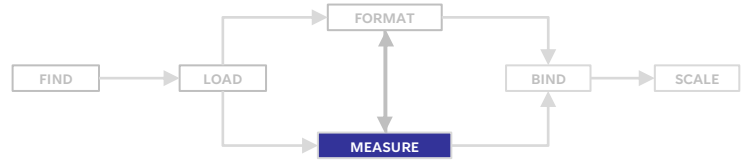
Measuring Data

```
d3.csv("./data/data.csv", (d) => {
return { technology: d.technology, count: +d.count };
}).then((data) => { console.log(data);});
```

Our data are now converted into an **array of JS Objects**!





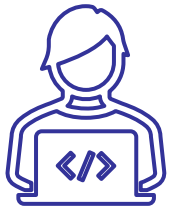


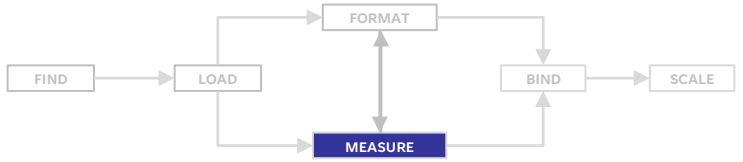
Measuring Data

- We can achieve the same result, but with an **automatic conversion**.
- The function `d3.autoType` detects data types and converts them into the corresponding JS type.
  - However, we know that the dynamic duo **JS-type** is not famous to be a clear and systematic concept.
  - If you want to be sure to have the correct conversion, continue to do it with the manual operation (or use TypeScript).

```
d3.csv("./data/data.csv", d3.autoType};
}).then((data) => { console.log(data);});
```

```
main.js:11
(33) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, columns: Array(2)]
  0: {technology: 'ArcGIS', count: 147}
  1: {technology: 'D3.js', count: 414}
  2: {technology: 'Angular', count: 20}
  3: {technology: 'Datawrapper', count: 171}
  4: {technology: 'Excel', count: 1078}
  5: {technology: 'Flourish', count: 198}
  6: {technology: 'ggplot2', count: 435}
  7: {technology: 'Gephi', count: 71}
  8: {technology: 'Google Data Studio', count: 176}
```

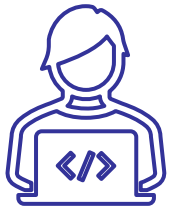


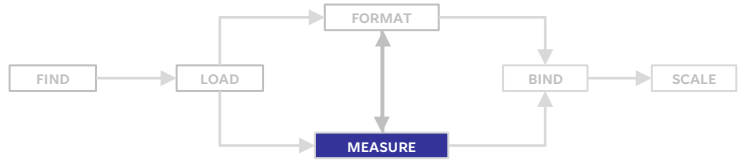


Measuring Data

- We can start to explore and measure the properties of this dataset.
- The most common methods are:
  - `data.length`: number of entries in the dataset
  - `d3.max(iterable element, accessor function):` max element in the iterable element
  - `d3.min(iterable element, accessor function):` min element in the iterable element.
  - `d3.extent(iterable element, accessor function):` returns an array containing the min and the max.

```
d3.max(data, d => d.count); // 1078
d3.min(data, d => d.count); // 20
d3.extent(data, d => d.count) // [20, 1078]
```





Measuring Data

- To enhance the readability of the dataset, we can **sort** our data.
- To do so, we use the JS `sort()` method.
- It takes a **compare function** as an argument.
- Here we are saying that, if the count of `b` is greater than the count of `a`, `b` should appear before `a` in the dataset (**descending sort**).

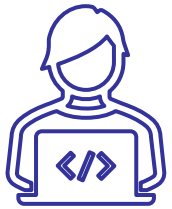
```
data.sort((a, b) => b.count - a.count);
```

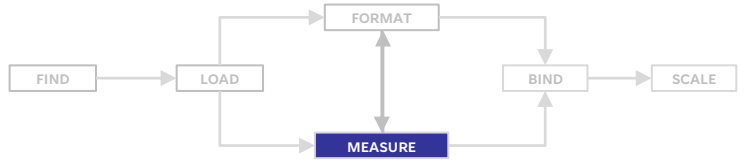
BEFORE

```
(33) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, columns: Array(2)]
  0: {technology: 'ArcGIS', count: 147}
  1: {technology: 'D3.js', count: 414}
  2: {technology: 'Angular', count: 20}
  3: {technology: 'Datawrapper', count: 171}
  4: {technology: 'Excel', count: 1078}
  5: {technology: 'Flourish', count: 198}
  6: {technology: 'ggplot2', count: 435}
  7: {technology: 'Gephi', count: 71}
  8: {technology: 'Google Data Studio', count: 176}
```

AFTER

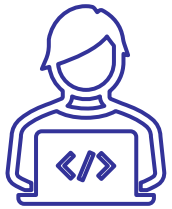
```
(33) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, columns: Array(2)]
  0: {technology: 'Excel', count: 1078}
  1: {technology: 'Tableau', count: 852}
  2: {technology: 'PowerPoint', count: 681}
  3: {technology: 'R', count: 561}
  4: {technology: 'Python', count: 530}
  5: {technology: 'Pen & paper', count: 522}
  6: {technology: 'Power BI', count: 460}
  7: {technology: 'ggplot2', count: 435}
  8: {technology: 'Illustrator', count: 426}
```

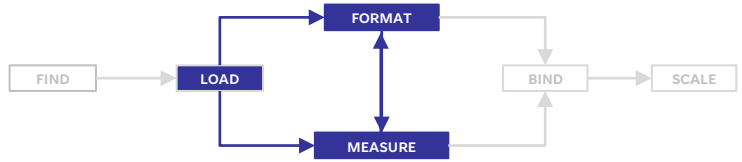




Preparing Data: Overview

1) **Loading** data with a fetch function
2) **Format** the data (row conversion)
3) Chain `then()` to access the entire dataset when is completely loaded.
    a) (Optional) **Refactor** the data.
    b) (Optional) **Measure** the data.
4) **Pass** the data to another function that will **handle** the visualization.



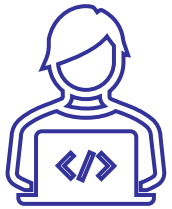


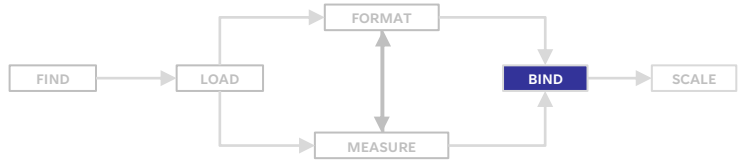
Binding data to DOM elements

- **Data-Binding** is one of the most useful features of D3.
- We can couple single pieces of the dataset to DOM elements.
  - Examples:
    - bind the length of a rectangle in a barchart to a number in our dataset;
    - bind the degree of a pie chart to a percentage from the data.
- The **pattern** used to bind data is the following one:

```
selectAll("selector").data(myData).join("element")
```

- **Note**: you can find, for older versions of D3, the pattern `enter().append("element")` in place of `join("element")`.

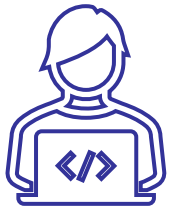


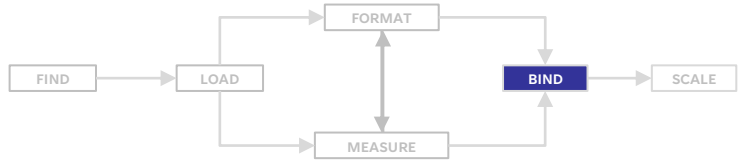


Binding data to DOM elements

- So, the data binding pattern **generates** as many SVG elements as there are data points.
- Once the elements are generated, we then access their attributes through **inline functions**.
- Data bound to an element is also **passed** to its children.

```
selectAll("selector").data(myData).join("element")
```

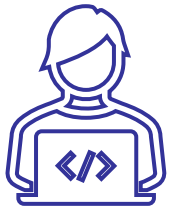


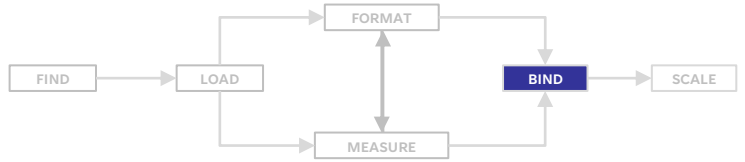


Binding data to DOM elements

```
selectAll("rect").data(data).join("rect")
```

- For example, we want to add one rectangle for each data point in our dataset.
- The first instruction `selectAll("rect")` will select elements that are not yet present in our DOM.
- This is called **empty selection**.

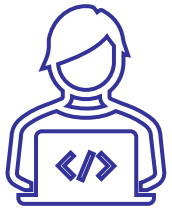


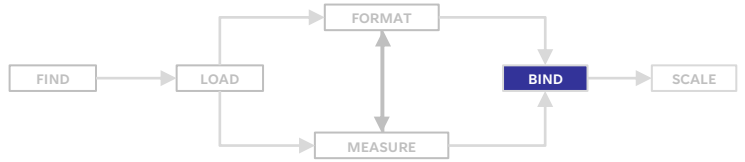


Binding data to DOM elements

```
selectAll("rect").data(data).join("rect")
```

- To tell the DOM how many rectangles it needs to place, we chain the `data()` method.
- The parameter of this method is our dataset.

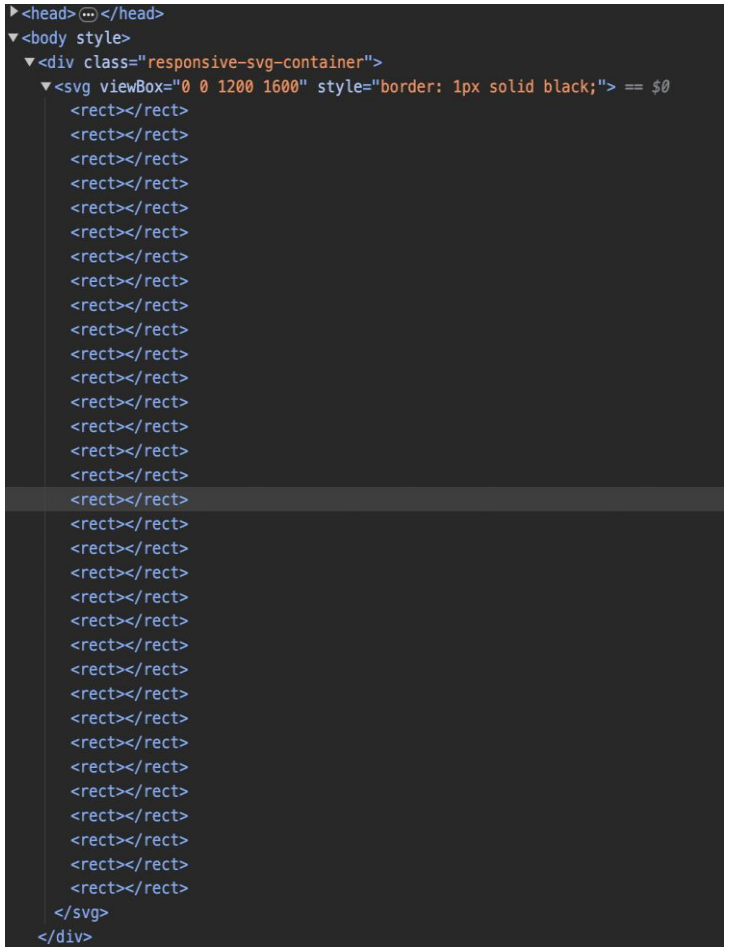


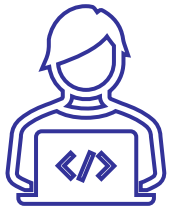


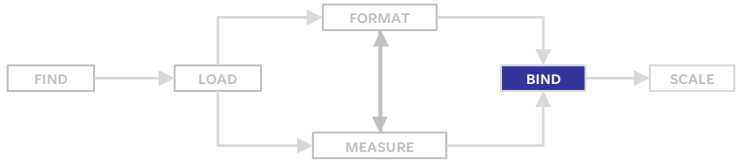
Binding data to DOM elements

```
selectAll("rect").data(data).join("rect")
```

- Finally, the rectangles can enter the DOM with the `join()` method.
- If we look in the HTML page, we can see the following
- So, in D3, the **selection** becomes the combination of the DOM elements and the data together.



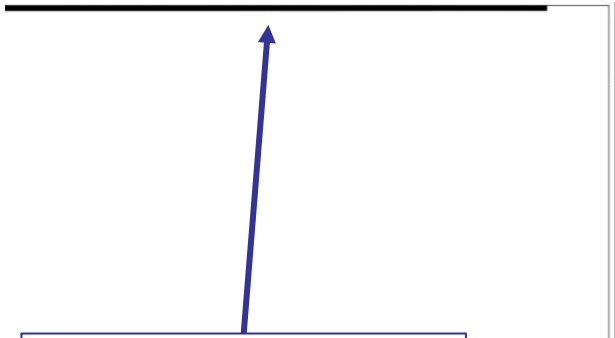




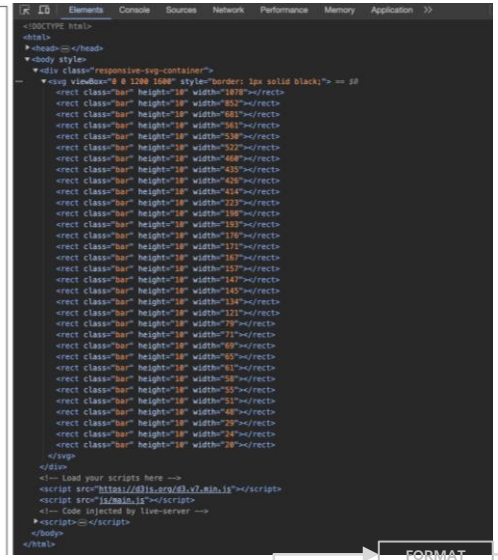
Setting attributes dynamically

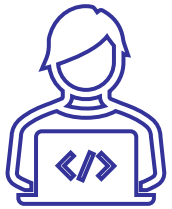
- Now, we can **access** the created elements in the DOM to **dynamically set** their attributes.
- In the following code, we set the same `class` for all the elements, the `height` equals to a constant meanwhile the `width` is read from the `count` value of each data point.

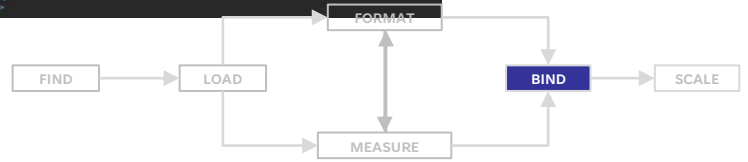
```
const barHeight = 10;
d3.select("svg")
.selectAll("rect")
.data(data)
.join("rect")
.attr("class", "bar")
.attr("height", barHeight)
.attr("width", (d) => d.count);
```



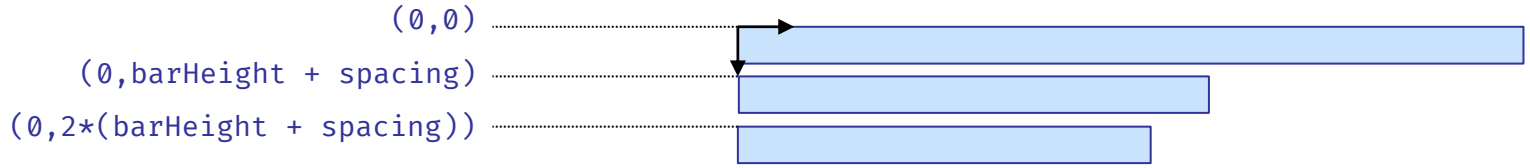
Here, the rectangles are one on top of the other: we need to move them!





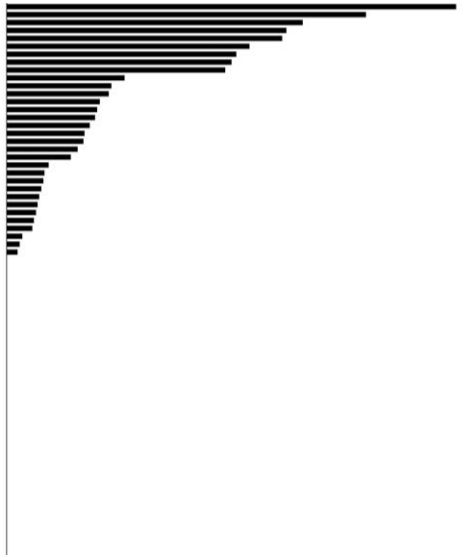


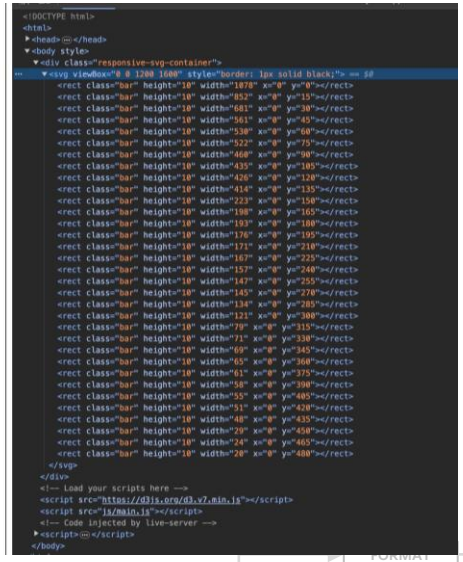
Setting attributes dynamically

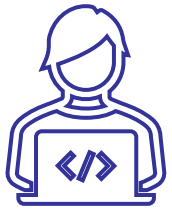


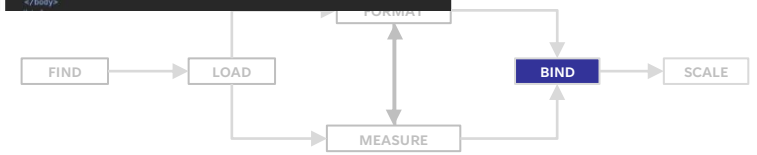
For now, we are doing this operation manually by setting x and y ourselves.

```
const barHeight = 10;
d3.select("svg")
.selectAll("rect")
.data(data)
.join("rect")
.attr("class", "bar")
.attr("height", barHeight)
.attr("width", (d) => d.count)
.attr("x", 0)
.attr("y", (d, i) => i *
(barHeight + 5));
```



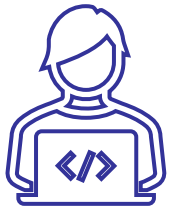


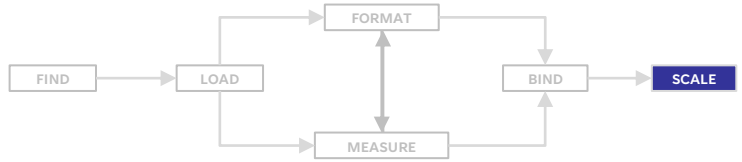




Adapting Data for the Screen

- In D3, the translation from data to screen is handled with **scales**.
- **Scales** map a dimension of abstract data to a visual representation.
  - Usually, the **output** of a scale is the position, the size, or the color of the element.
- More formally, in D3 we have:
  - `domain()`: all the possible values of the **input**;
  - `range()`: spectrum of all the **output** values.
- D3 has multiple **scale functions**. We will use:
  - **Linear Scale**: for quantitative data (continuous domain/continuous range)
  - **Band Scale**: for categorical/ordinal data (discrete domain/continuous range)
  - If you want to learn more about other scales: d3-scale





Adapting Data for the Screen

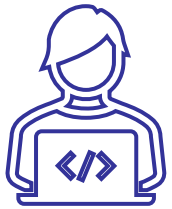
- For example:

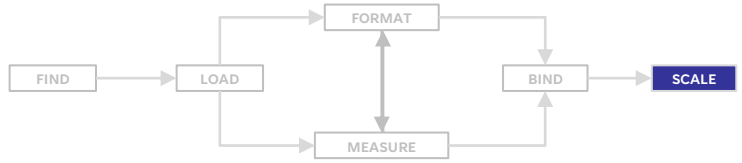
```
const myScale = d3.scaleLinear().domain([0, 100]).range([0, 1000]);
```

- Here, if we want to try the scale:

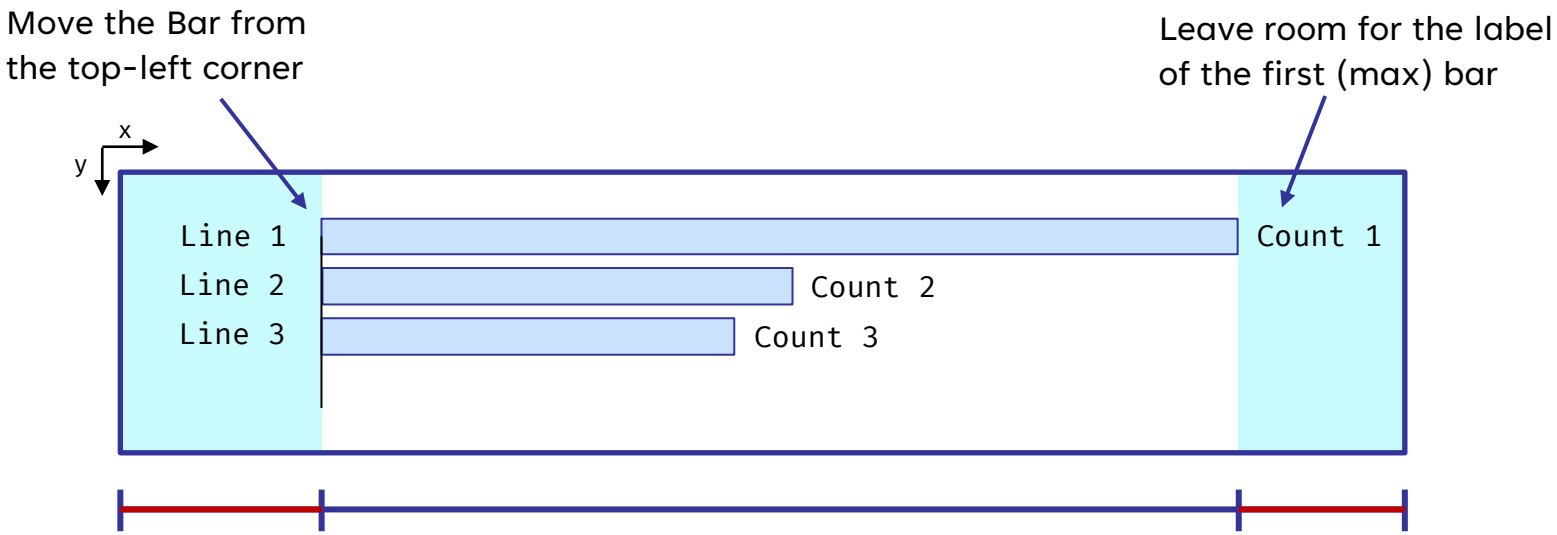
```
myScale(50); // 500
```

- The scale is linear, meaning that every 1 unit in the domain is mapped to its value multiplied by 10.
- How to use scales in our example?
  - We create two scales: one for fitting the data on the x-axis and one for the y-axis.



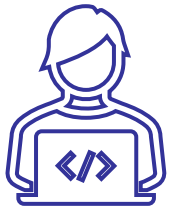


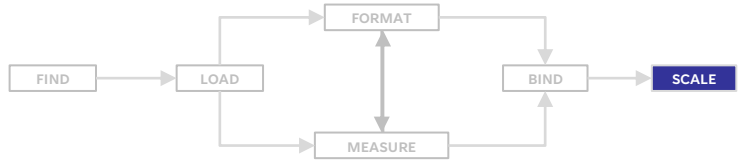
Adapting Data for the Screen




```
const xScale = d3.linearScale().domain([0, Count1]).range([0, available_range]);
```

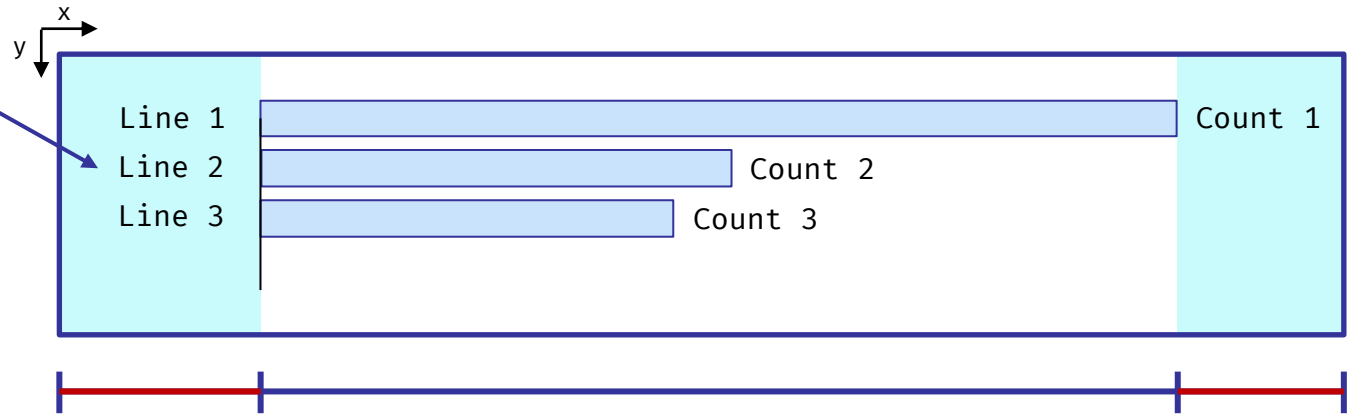
```
...
attr("width", d => xScale(d.count));
...
```





Adapting Data for the Screen

We don't need to position bars by hand like we did before



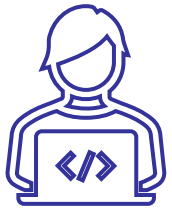
Available Range: `available_range`

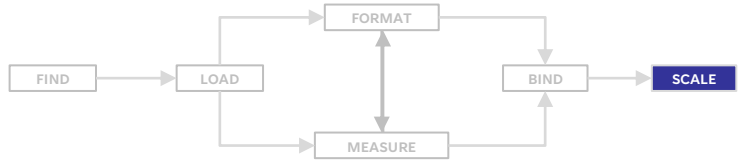
```
const yScale = d3.scaleBand().domain(data.map(d => d.technology)).range([0, svgHeight]);
```

```
...
attr("y", d => yScale(d.technology));
...
```

```
...
attr("height", d => yScale.bandwidth());
...
```

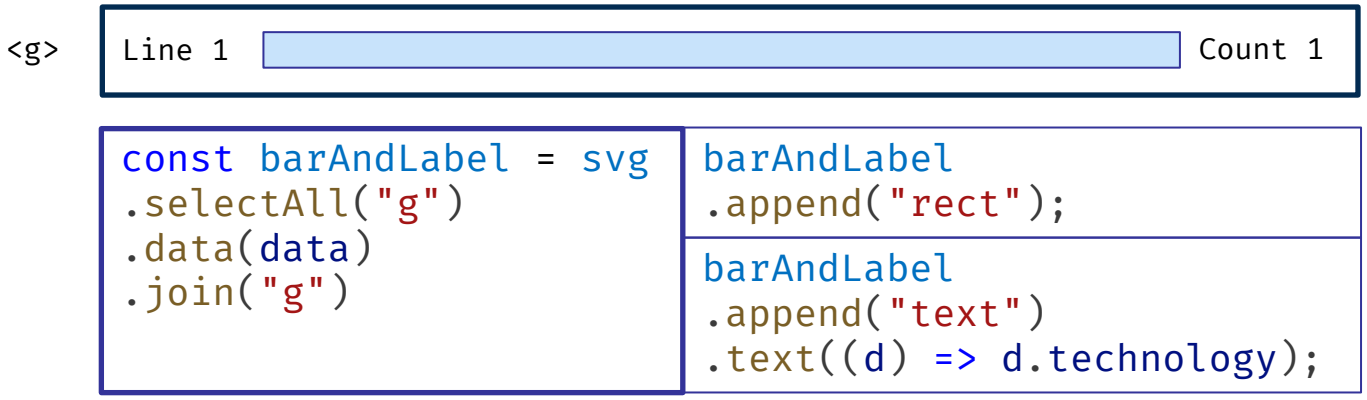
To compute the thickness of the bar, we can use the `bandwidth` function of the `scaleBand`

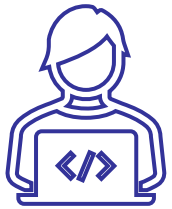


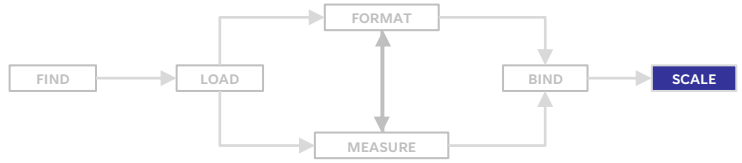


Adding labels to a chart

- In SVG-based images, we add labels with SVG **text elements**.
- Our strategy is the following:
  - We need to add two labels to our bars, one for the technology name and one for the count number.
  - We then **group bars** and labels together into a HTML <g> element so that we can translate everything in one go.

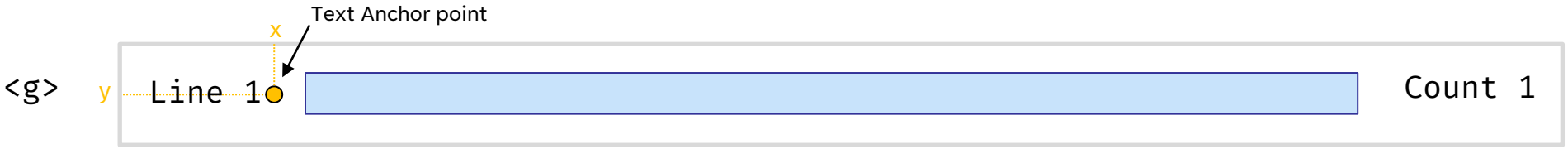






Adding labels to a chart

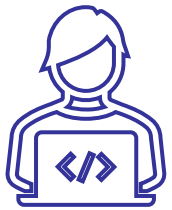
- To compute the exact position of our labels inside the group, we can easily set the x and y attribute.

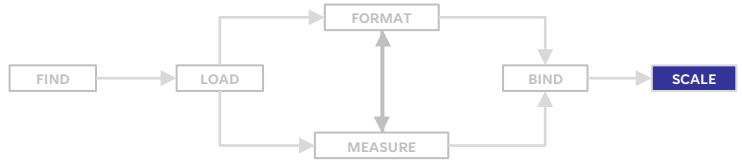



```
const barAndLabel = svg
.selectAll("g")
.data(data)
.join("g");
```

```
barAndLabel
.append("text")
.text ("Line 1")
.attr("x", x)
.attr("y", y)
.attr("text-anchor", "end")
.attr("font-family", "sans-serif");
```

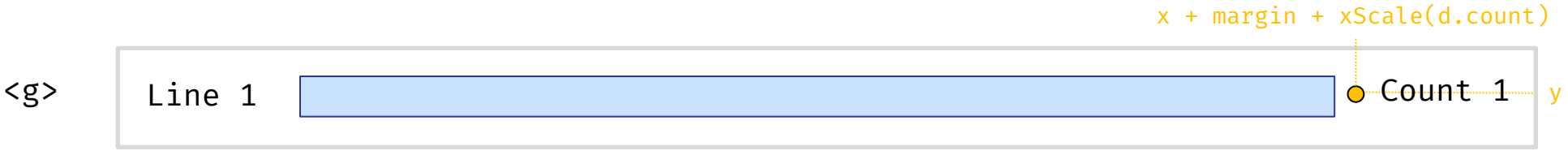
If we want that our text is aligned with the start of the bar, we can set the `text-anchor` to `end`.





Adding labels to a chart

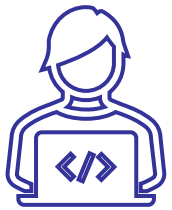
- To compute the exact position of our labels inside the group, we can easily set the x and y attribute.

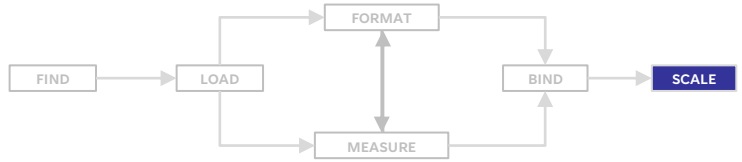



```
const barAndLabel = svg
.selectAll("g")
.data(data)
.join("g");
```

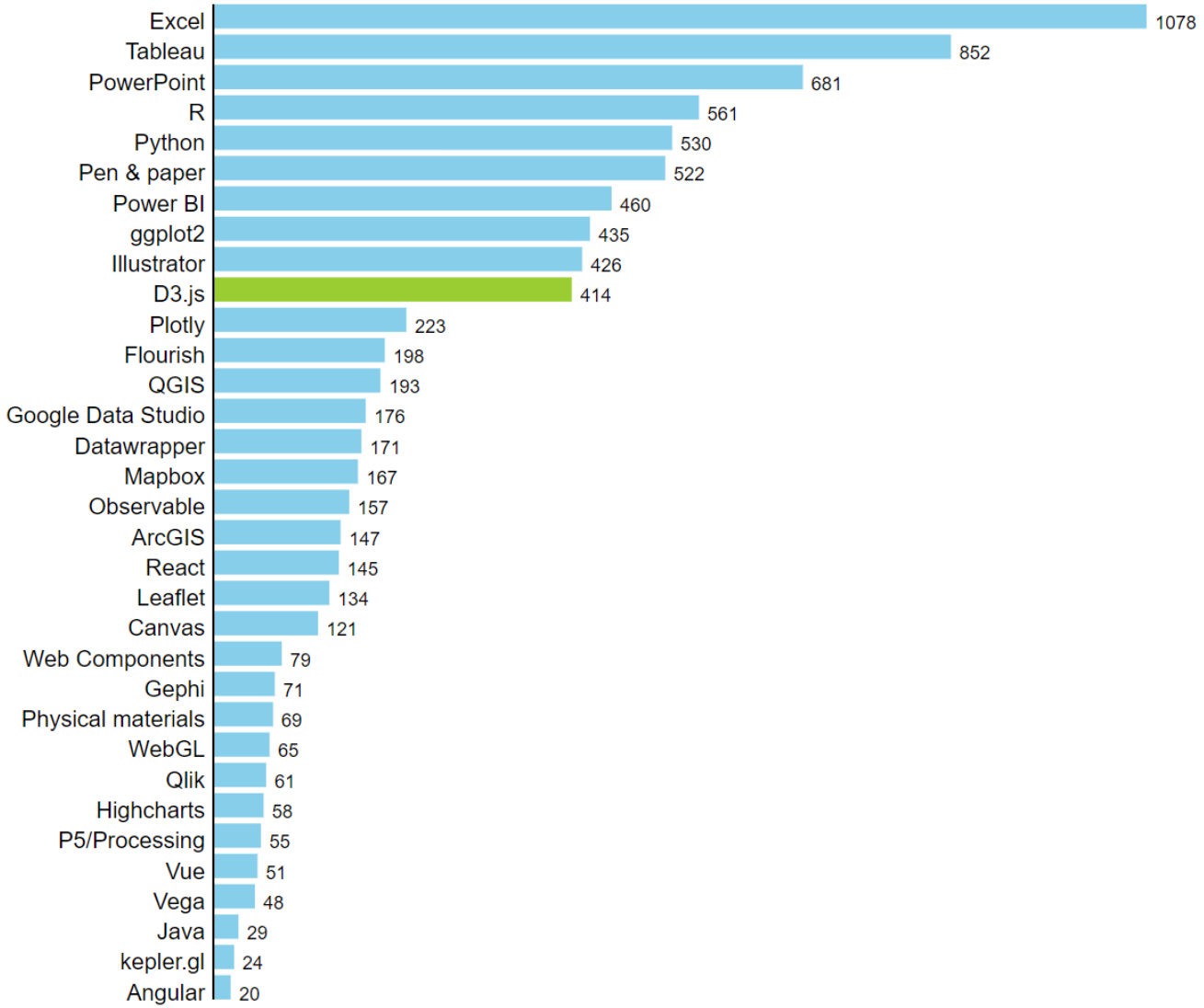
```
barAndLabel
.append("text")
.text ("Count 1")
.attr("x", x + xScale(d.count) + margin)
.attr("y", y)
.attr("font-family", "sans-serif");
```

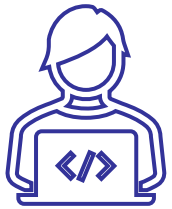
Here x is the starting point of the previous label.





Visualization

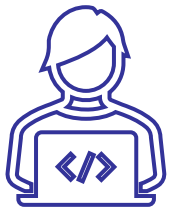




D3: Drawing Functions

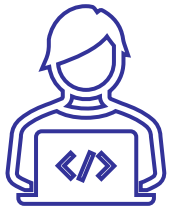
For a written guide of this tutorial, please refer to the **README.md** file in the **L3** folder. You can open the file either in VSCode or Github.

This tutorial was taken from Chapter 4 of “D3.js in Action, Third Edition”



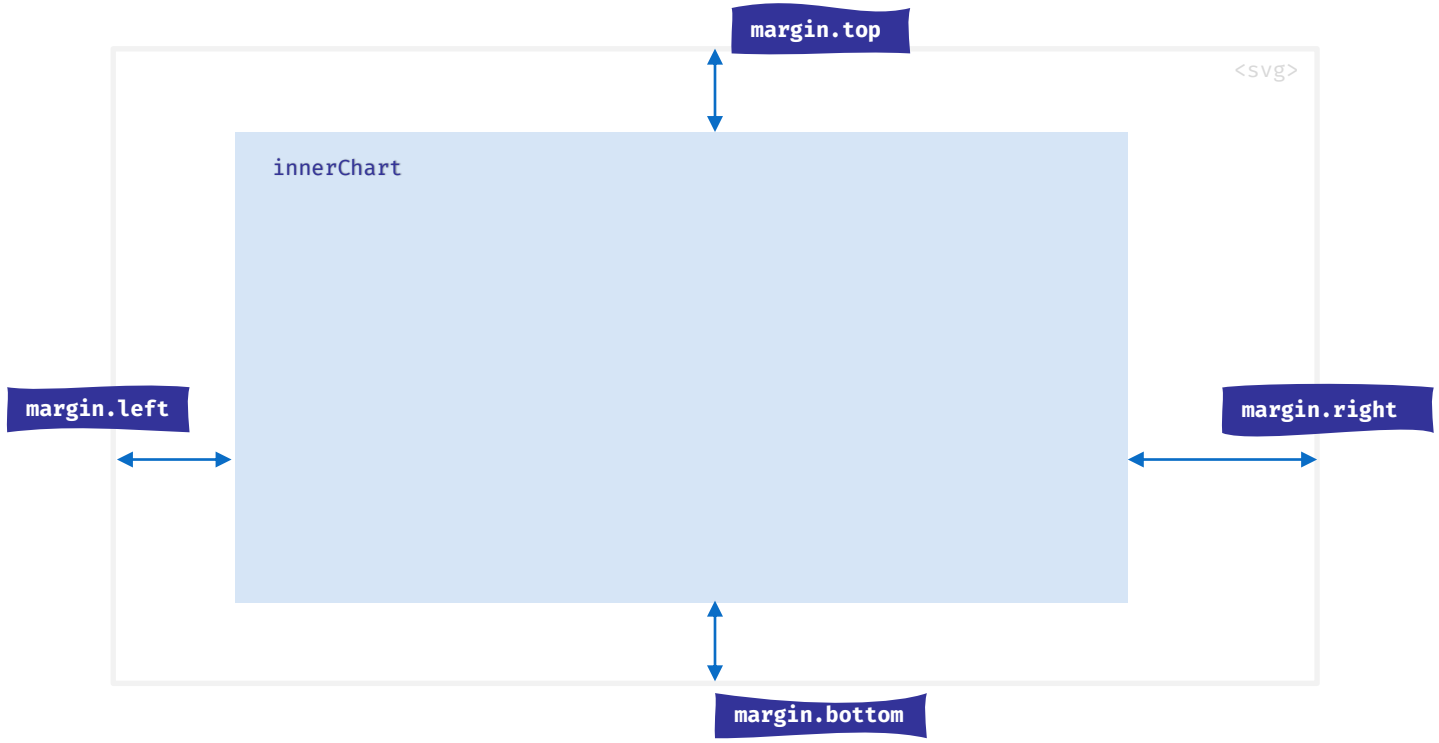
D3 and paths

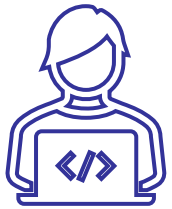
- In the last tutorial, we built a bar chart using only SVG primitives.
- However, to create more complex visualizations, we will generally use **SVG paths**.
- The shape of an SVG path is determined by its `d` attribute:
  - As we already saw in the previous lesson, the `d` attribute of a path can become very long, very fast.
- In this second tutorial, we will learn how to use **D3 shape generators** to automatically draw paths.



D3 Margin Convention

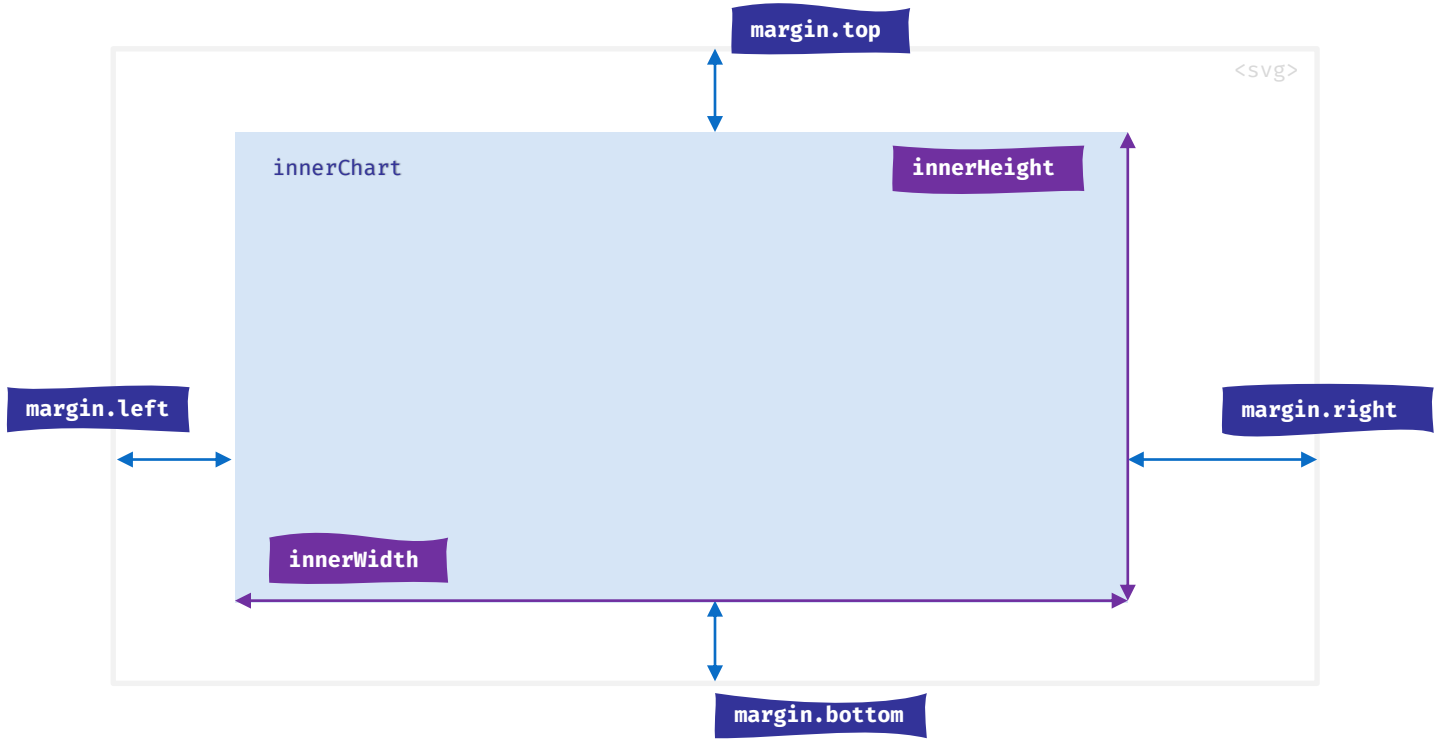
- Before introducing the shape generators, we will briefly talk about the **Margin Convention** and **Axes Generator**.
- D3 **Margin Convention** is a summary of best practices, aimed at reserving space around a chart for axes, labels, and legend in a systematic and reusable way.
- This convention use a `top`, `right`, `bottom`, and `left` margin.
- By defining these margins, we know the remaining area for the chart, usually called the **Inner Chart**.

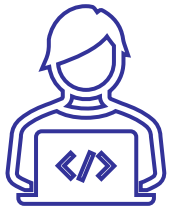




D3 Margin Convention

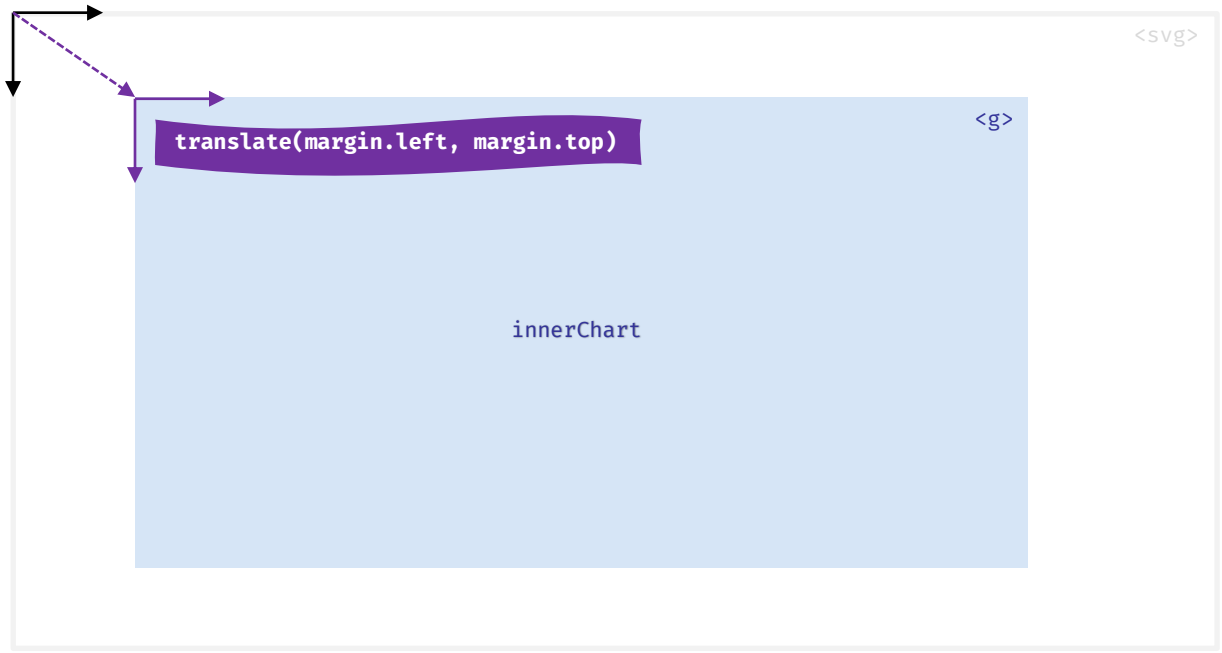
- If we know the size of the `svg` container and the **margins**, we can compute the `innerWidth` and `innerHeight` of the chart.
- By making these two variables proportional to the margins and the svg container size, we ensure that they will **automatically adjust** if we need to change these margins.
  - In fact, for now we are just guessing how much space we need to reserve.

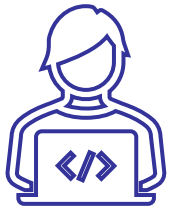




D3 Margin Convention

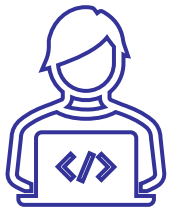
- A useful strategy is to **wrap** the elements constituting the chart itself into an **SVG group** and position this group inside the SVG container based on the margins.
- This creates a new origin for the chart elements and facilitates their implementation.





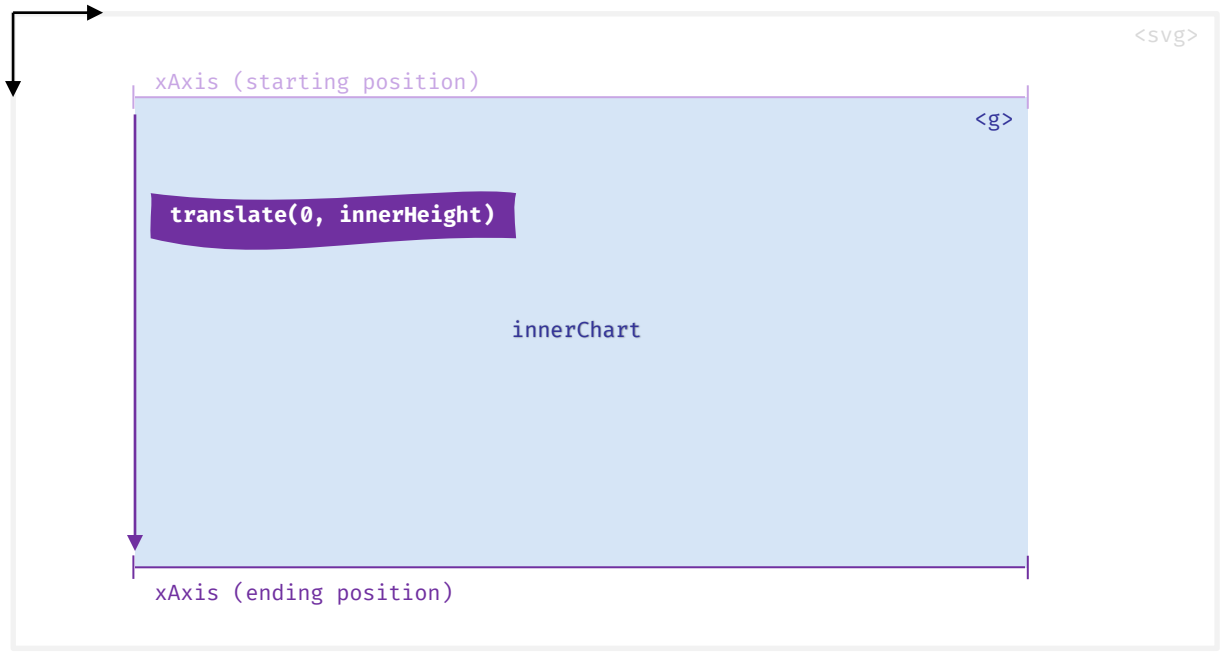
D3 Generating Axes

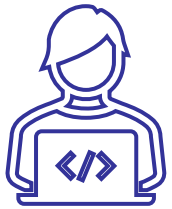
- **Axes** are references for the reader to better understand the numbers and values represented in the chart.
- D3 has four axis generators:
  - `axisTop()`, `axisRight()`, `axisBottom()`, and `axisLeft()`, that create the components of top, right, bottom, and left axis, respectively.
- These axis generators take a scale as an **input** and return the SVG elements composing an axis as an **output** (a line along the axis and multiple sets of tick and label).
- We append an axis to a chart by chaining the `call()` method to a selection and passing the axis as an argument.



D3 Generating Axis

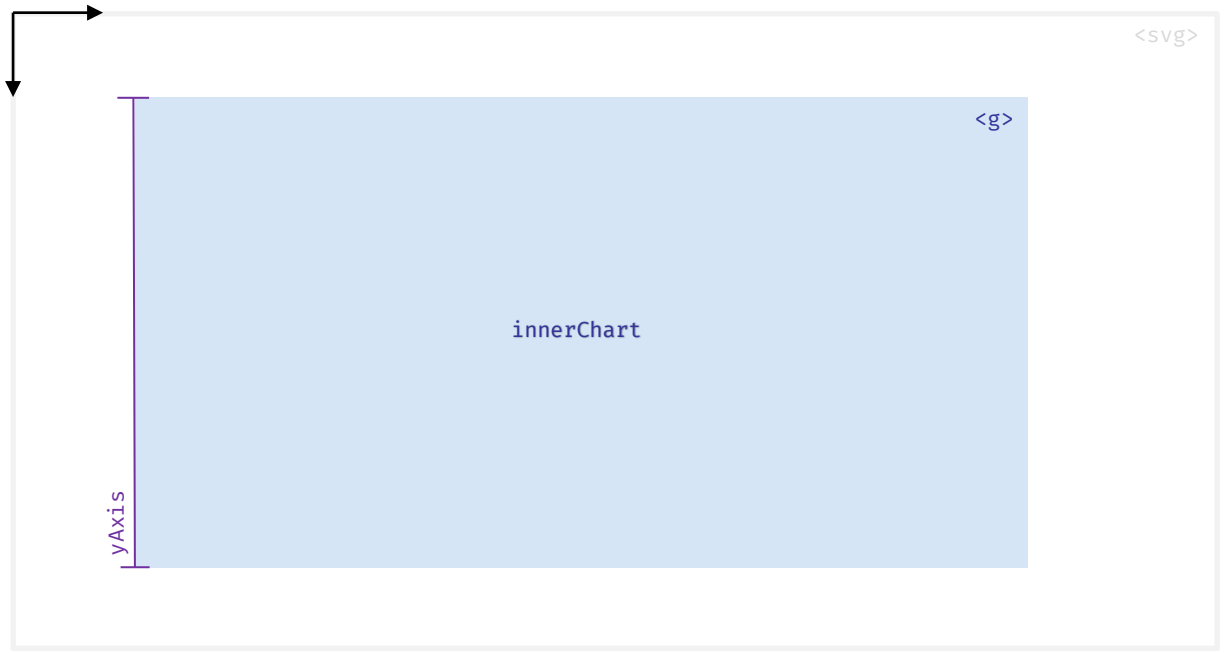
- We usually use the `axis.bottom()` and `axis.left()` for the x-axis and y-axis, respectively.
- By default, D3 axes are displayed at the top-left corner of the selection.
  - **Note for the yScale**: the starting point of the range is the `maxValue`, and the last value is the `minValue`.
  - If we wrap an axis and its labels into a group, we can easily translate them in their specific position.

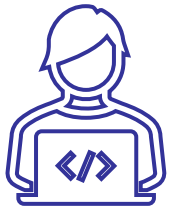




D3 Generating Axis

- We usually use the `axis.bottom()` and `axis.left()` for the x-axis and y-axis, respectively.
- By default, D3 axes are displayed at the top-left corner of the selection.
  - **Note for the yScale**: the starting point of the range is the `maxValue`, and the last value is the `minValue`.
  - If we wrap an axis and its labels into a group, we can easily translate them in their specific position.

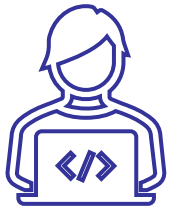




Drawing a Line Chart

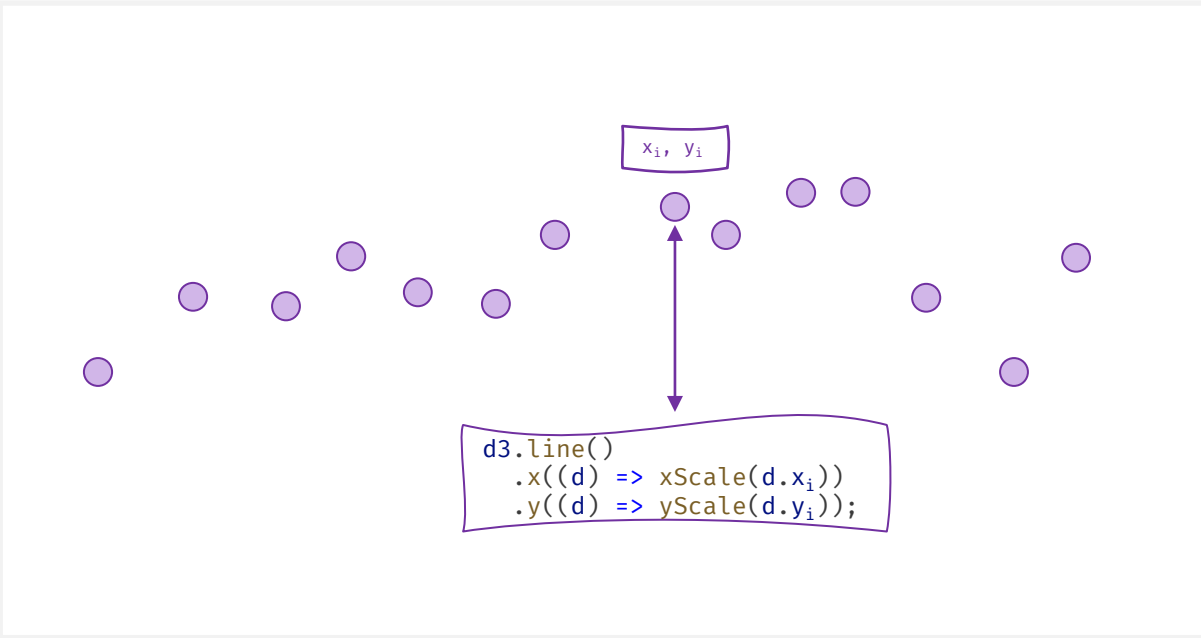
- **Line charts** are lines (or curves) connecting (or interpolating) data points.
  - They are usually used to show the **evolution** of a phenomenon over time.
- To draw a line chart, we first initialize a **line generator** with the method `d3.line()`.
  - This will compute the d attribute of the path for us.
- The line generator has two accessor functions, `x()` and `y()`, which compute each data point's horizontal and vertical position.

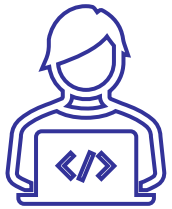
```
const lineGenerator = d3.line()
    .x((d) => xScale(d.x))
    .y((d) => yScale(d.y));
```



Drawing a Line Chart

```
const lineGenerator = d3.line()
    .x((d) => xScale(d.x))
    .y((d) => yScale(d.y));
```





Drawing a Line Chart

- We can turn a line chart into a curve with the `curve()` accessor function.
- D3 offers multiple curve interpolation functions, which affect data representation and must be selected carefully.

```
const curveGenerator = d3
    .line()
    .x((d) => xScale(d.x))
    .y((d) => yScale(d.y))
    .curve(d3.curveCatmullRom);
```

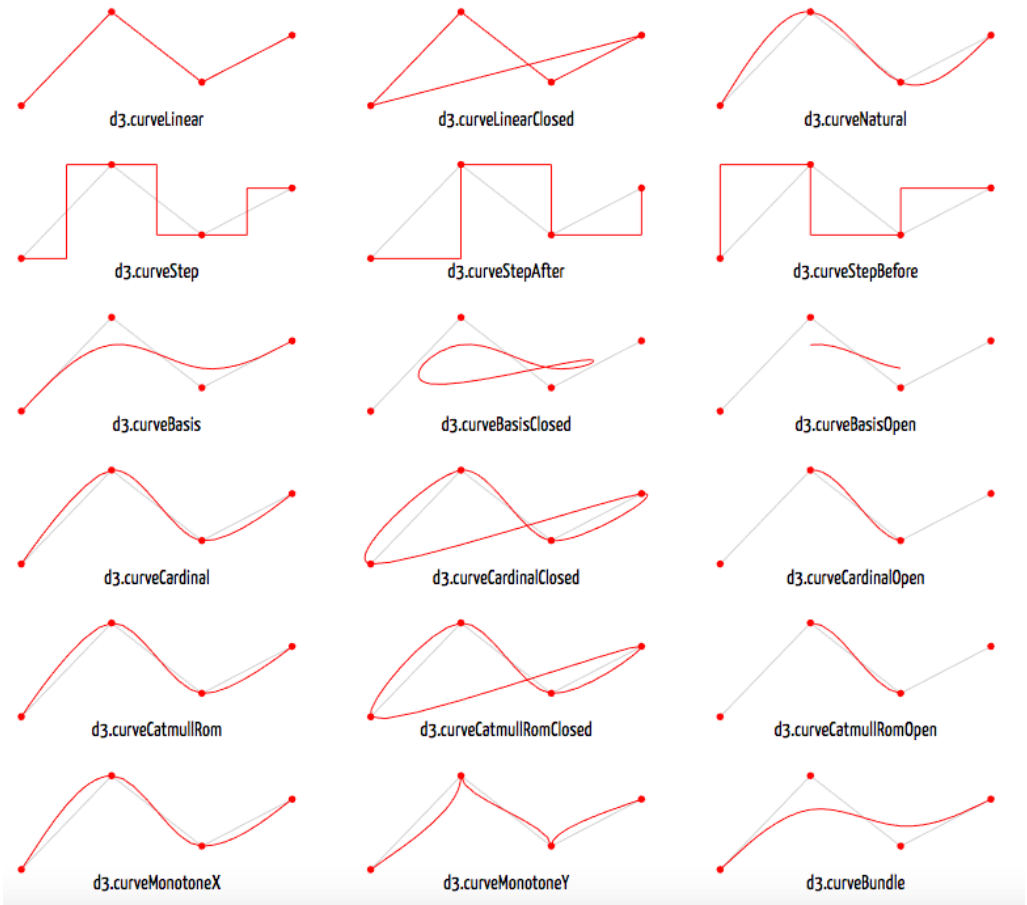
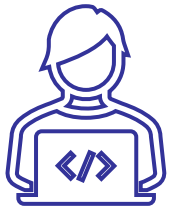


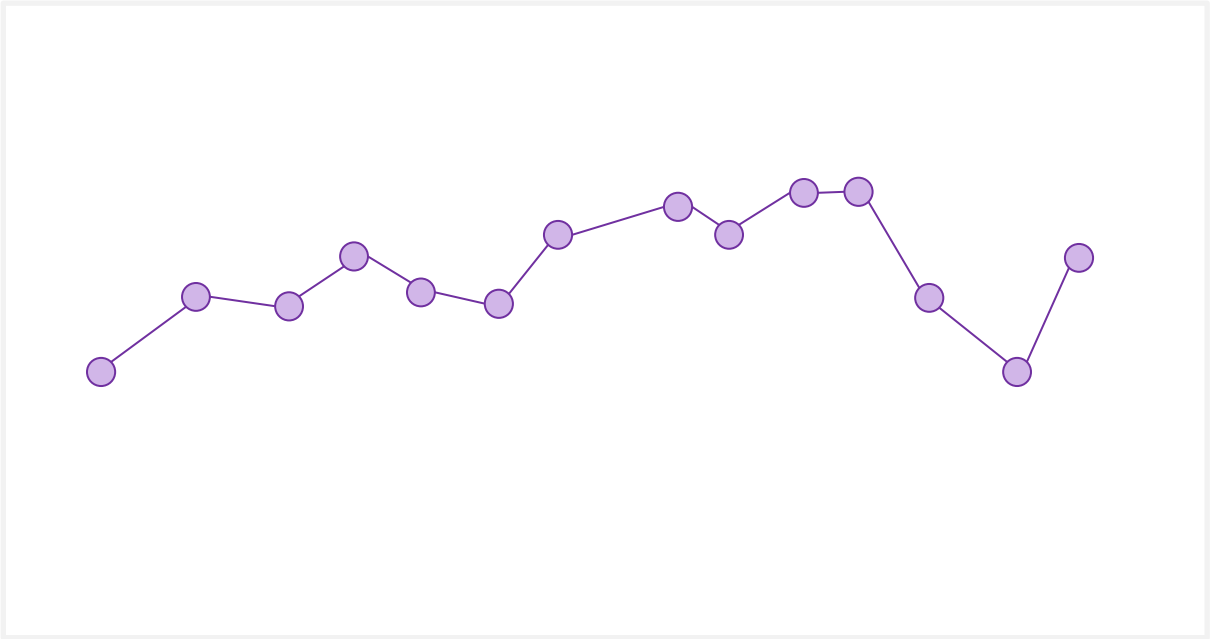
Image from: Learn D3.js

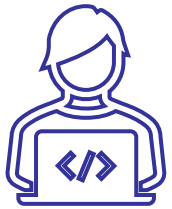


Drawing a Line Chart

- To make a line chart appear on the screen, we **append a path** element to a selection and set its d attribute by calling the line generator and passing the dataset as an attribute.

```
innerChart
    .append("path")
    .attr("d", lineGenerator(data))
    .attr("fill", "none")
    .attr("stroke", purple);
```

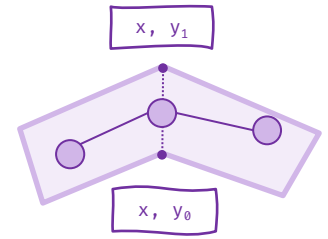


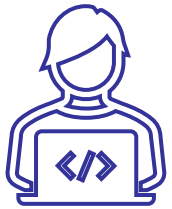


Drawing the Area

- An **area** is a region between two boundaries, and drawing an area with D3 is like drawing a line.
- To draw an area, we first declare an **area generator** with the method `d3.area()`.
  - This method requires at least three accessor functions to calculate the position of each data point along the edges of the area.

```
d3.area()
  .x((d) => xScale(d.xValue))
  .y0((d) => yScale(d.y0Value))
  .y1((d) => yScale(d.y1Value));
```

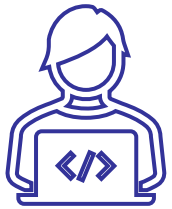




Drawing the Area

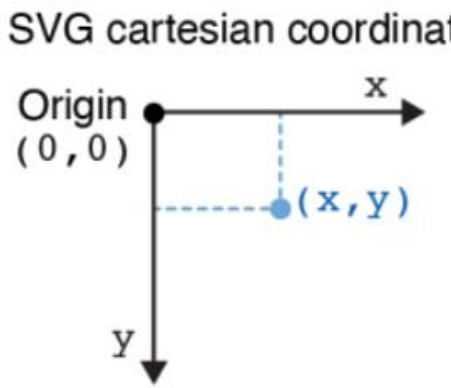
- All the things we saw for lines are **still valid** for areas. In particular:
  - D3 provides interpolation functions that can be applied with the `curve()` accessor function.
  - To make an area appear on the screen, we **append a path** element to a selection and set its `d` attribute by calling the area generator and passing the dataset as an attribute.

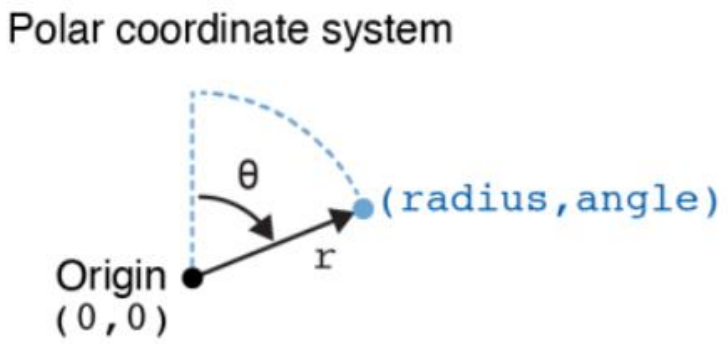
```
innerChart
  .append("path")
    .attr("d", areaGenerator(data))
    .attr("fill", aubergine)
    .attr("fill-opacity", 0.2)
```



Drawing Arcs

- **Arcs** are a common shape in pie charts.
- Like lines and areas, arcs are drawn with **SVG paths**.
- It is easier to work with a **polar coordinates** system when working with arcs:





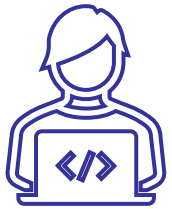
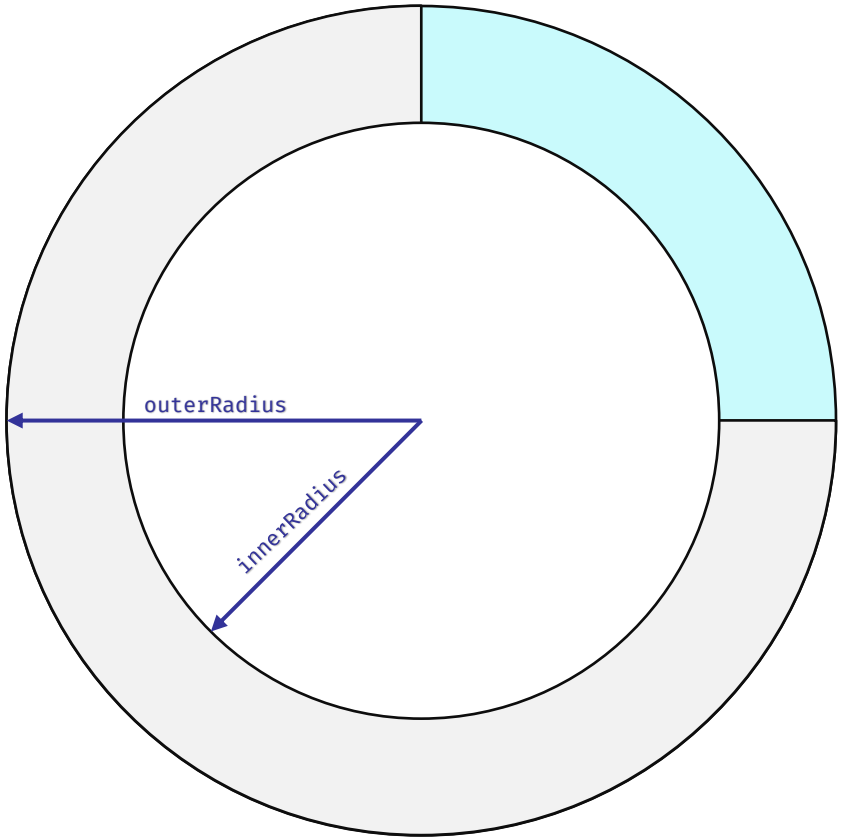


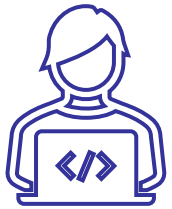
Image from: Chapter 4, “D3.js in Action”

Drawing Arcs

- We can use the `d3.arc()` function to compute the chart.
- It requires two main accessor function:
  - `innerRadius(radius1)`
  - `outerRadius(radius2)`
- Then, one can also add, by chaining:
  - `padAngle(radians):` to add padding
  - `cornerRadius(pixels):` to smooth the corner

```
const arcGenerator = d3.arc()
    .innerRadius(radius1)
    .outerRadius(radius2)
    .padAngle(radians)
    .cornerRadius(pixels);
```



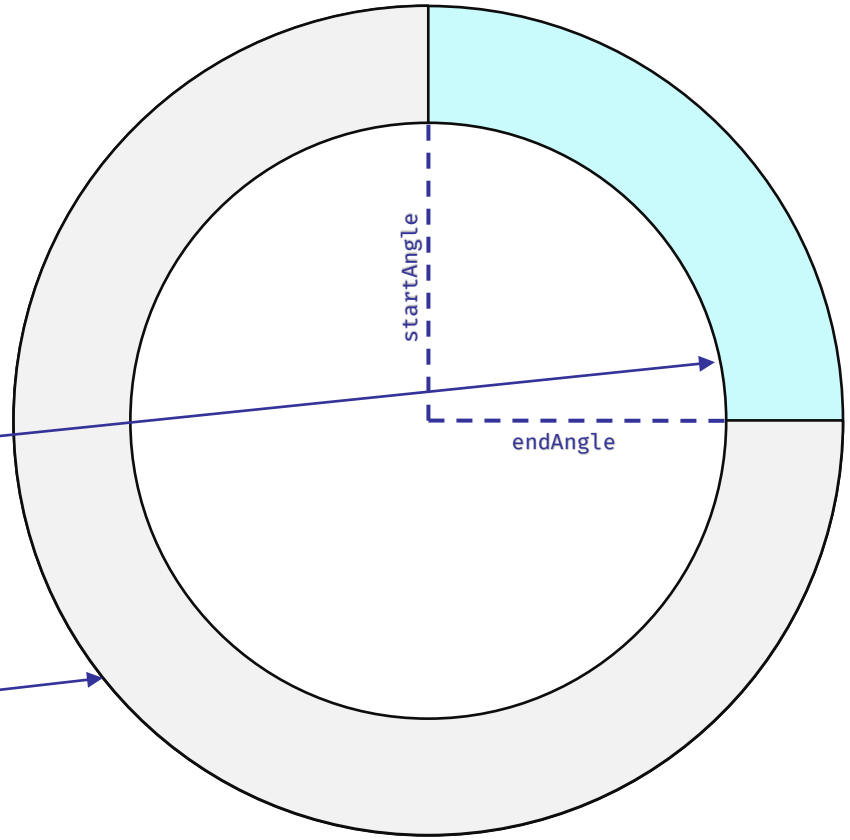


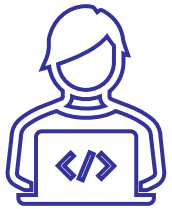
Drawing Arcs

- We can manually compute the start and end angle for our chart.
  - These values can then be passed to the generator.

```
innerChart.append("path").attr("d", () => {
  return arcGenerator({
    startAngle: 0,
    endAngle: endAngle,
  });
});
```

```
innerChart.append("path").attr("d", () => {
  return arcGenerator({
    startAngle: endAngle,
    endAngle: 2*Math.PI,
  });
});
```

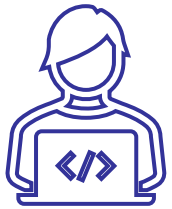




D3: Improving Readability

For a written guide of this tutorial, please refer to the **README.md** file in the **L3** folder. You can open the file either in VSCode or Github.

This tutorial was taken from Chapter 4 and Chapter 7 of “D3.js in Action, Third Edition”



Adding Labels

- **Labels** are particularly useful to help readers understand our data visualizations.
- In D3, labels are simply **text elements** that we need to position within the SVG container.
  - The position of SVG text is controlled by its `x` and `y` attributes.
- The `y` attribute sets the position of the text's baseline, which by default is positioned at its bottom.
- We shift the baseline of a SVG text with the attribute `dominant-baseline`.
- The value `middle` moves the baseline to the vertical middle of the text, while the value `hanging`, shifts the baseline to the top.

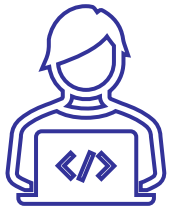
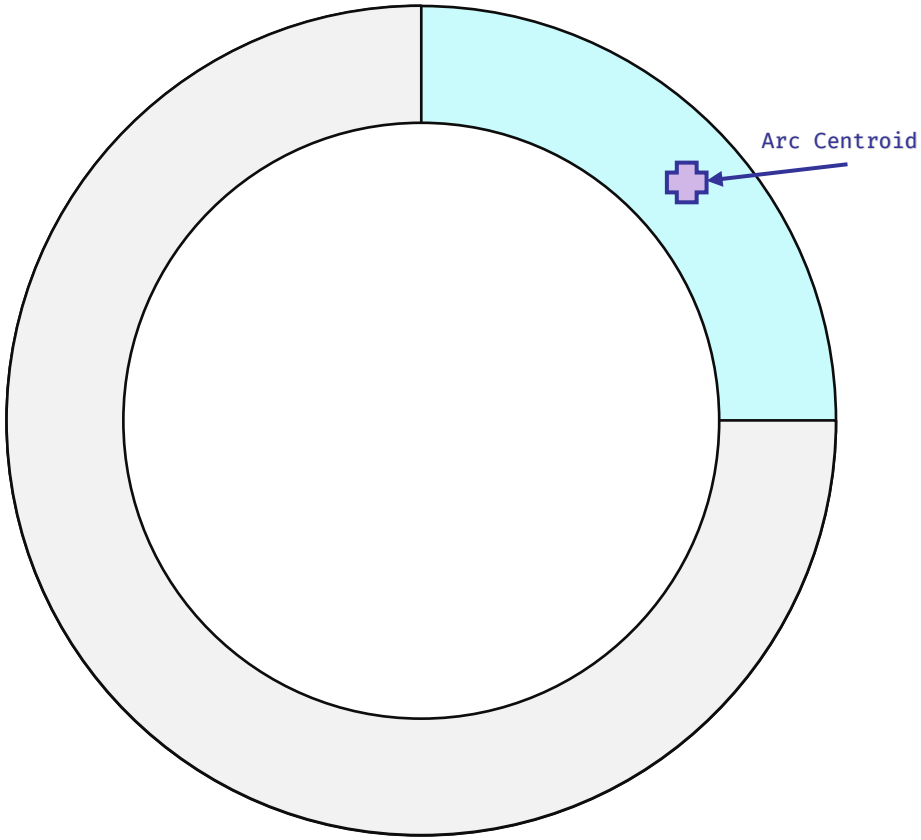


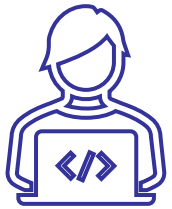
Image from: Chapter 4, "D3.js in Action"

Calculating the centroid of an Arc

- The **center of mass** of an arc can be calculated with the `centroid()` method.
- Chained to an arc generator, this accessor function returns an array containing the horizontal and vertical position of the center of mass.

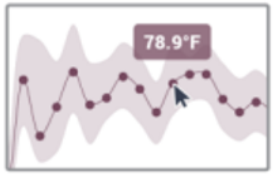
```
const centroid = arcGenerator
    .startAngle(0)
    .endAngle(angleDaysWithPrecipitations_rad)
    .centroid();
```



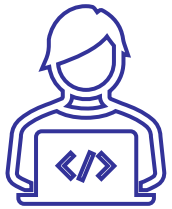


Add Tooltips

- **Tooltips** are the simplest interaction that one can add to a visualization.
- They enhance the readability of the chart by showing some information of the data without over-pollute the visualization.
- Tooltips are shown on **mouseover**.

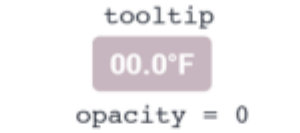


- When building tooltips, it is best to avoid **obstructing** the view of the adjacent markers.
  - One trick is to make the tooltip's background **semi-transparent**.



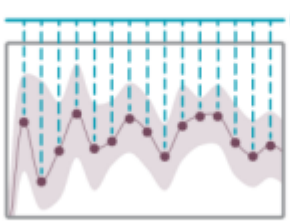
Add Tooltips

1. Build the tooltip with SVG elements and append it to the inner chart. Set its opacity to zero.



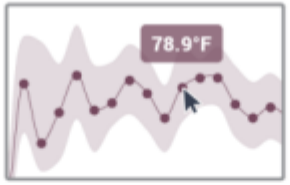

```
<g>
  <rect />
  <text></text>
</g>
```

2. Attach two event listeners to each circle: one for the "mouseenter" event and one for "mouseleave".




```
.on("mouseenter", (e, d) => {
  // Do something
})
.on("mouseleave", (e, d) => {
  // Do something
});
```

3. When the mouse is positioned over a circle, populate the tooltip with the average temperature data attached to the circle element. Then translate the tooltip over the circle, and set its opacity to 100%.

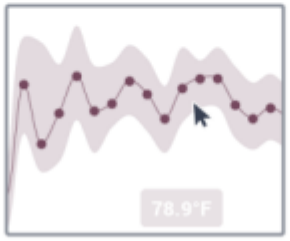



```
.on("mouseenter", (e, d) => {

  text => data
  position => above circle
  opacity => 100%

})
```

4. When the mouse leaves the circle, change the tooltip's opacity back to zero and move it away from the chart.




```
.on("mouseleave", (e, d) => {

  opacity => 0
  position => away from chart

})
```

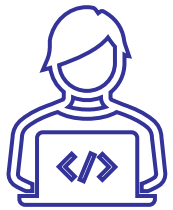


Image from: Chapter 7, “D3.js in Action”