F20-21DV Data Visualisation and Analytics

Lab 3 – Interactions

Preamble

Again, you should remember the following as you are about to start this lab.

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

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

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

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

Make sure you read through this document in its entirety.

With Lab 1, we looked at the general idea when drawing visualisations (which elements to use, how to draw them, how to map them on the screen, etc.). In Lab 2, we explored how you can transform raw data so that it can fit with the shape our visualisations (or other analysis scripts) expect.

Now, we will practice with ways of updating our visualisations, notably as a result of users interacting with them:

1. Setting up event listeners on D3 selections.
2. Using event handler callbacks between visualisations' scopes.
3. Animating visualisation changes with transitions.
4. Implementing advanced interactions with D3 Behaviors.

By the end of the lab, you should be equipped with sufficient knowledge of the above to implement interactive visualisations that can enhance data exploration.

Tutorial 9 – D3 Selections and Events

Callbacks

Before we jump tackle events, we must cover a fundamental concept in JavaScript: Callback functions.

In JavaScript, functions or methods can take any type of arguments, including other functions. If one function is given as argument of another – often to execute a subroutine – it is referred to as a *Callback* (it's *called back* as part of the process). In the example below, both **one** and **two** are functions, but **two** takes a callback function as argument. In the final line, **one** is provided as this callback.

```
function one(){
    console.log('Callback speaking');
}

function two(callback){
    console.log('Before callback');
    callback();
    console.log('After callback');
}

two(one);
// Before callback
// Callback speaking
// After callback
```

Note that, when calling the function **two**, we are only giving the reference to function **one** – its name – not actively calling it. Then, as part of the code inside function **two**, a call to the callback function parameter is made.

In some instances, we can skip the sperate definition of the callback function, and simply provide it while calling the main function. Because they don't have names, we call these *Anonymous Callbacks*. It's what we have been using when computing shape attributes based on data bound to SVG elements.

```
function main(val, cb){
    console.log(`Result: ${cb(val)}`);
}

main(3, x=>x**2);
main(4, x=>x+3);
// Result: 9
// Result: 7
```

Events in D3

To manage interactions on web pages, JavaScript implements an *event*-driven system. We can think of events as messages sent throughout the application when something happens on the page:

- The mouse pointer moved.
- The mouse was clicked.
- A keyboard key was pressed.
- Etc.

To respond to these events, two elements are necessary:

- **Event listeners**, attached to DOM elements, are there to detect events when they occur.
- **Event handlers**, attached to event listeners, are callbacks registering what to do when an event is detected.

With D3 Selections, we can use the **`.on()`** method to attach event listeners and handlers to selections (which means the DOM element, and its associated datum).

The first parameter of this method is a string representing the type of event the selection will listen to: *click*, *touch*, *mouseover*, etc. There are many types of events, which you can check on the MDN website: https://developer.mozilla.org/en-US/docs/Web/Events For now, we will focus on:

- '*click*' – when the mouse's primary button is pressed (and released), while the mouse's pointer is over the element listening.
- '*mouseover*' – whenthe mouse's pointer is moved on the element listening.
- '*mouseout*' – when the mouse's pointer is moved off the element listening.

The second parameter of the **`.on()`** method is a callback function describing the event handler. This callback can take two arguments:

- The **event** object, containing information about the event itself, when and where it happened, what was the target, etc.
- The selection's **datum**.

Imagine this code:

```
let bars = svg.selectAll('rect')
    .data(dataset)
    .join('rect')
    .attr('fill', 'blue')
    .on('click', (event, datum)=>{
        console.log(datum);
        d3.select(event.target)
            .style('fill', 'red');
    })
```

By now you should be able to see that it selects SVG rectangle elements, binds an array of data elements (**`dataset`**) to them, makes sure one rectangle is created for each data element, and fills these rectangles in blue.

Then, we specify that, when one of these rectangles gets clicked on, a callback should be executed:

- The **`datum`** associated with the rectangle is logged in the web console.
- We select the rectangle clicked on only (**`event.target`**) and fill it in red – instead of blue.

Exercise – Highlight and Tooltips

In this exercise, we will add two simple interactions to our bar charts:

- First, we will implement a way to highlight bars when we pass our mouse pointer over them.
- Second, we will add a basic tooltip text that will show additional information about bars, again, when we pass our mouse pointer over them.

This exercise assume that you have the solution provided for the bar chart class we did in Lab 1.

Adding Highlights

In this class, locate the method that renders/updates bars:

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

When we render these bars, we can also programme interactions.

First, we must attach an event listener to our bars, with the **`.on()`** method. The type of event we want to listen to is '*mouseover*'.

```
#updateBars(){
    this.bars = this.bars.data(this.data, d=>d[0])
        ...
        .on('mouseover', ...);
}
```

Be careful to add this as part of the chain, i.e., the '**;**' needs to move places…

When attaching this event listener, we must also provide the event handler callback. As discussed above, this callback takes two arguments, the event object and the datum of the selection targeted by the event.

```
#updateBars(){
    this.bars = this.bars.data(this.data, d=>d[0])
        ...
        .on('mouseover', (event, datum)=>{

        });
}
```

For now, you should try to simply print the datum in the Web Console. Then, make sure it works on your test web page.

Our goal however is to visually highlight the bar we are interacting with. For instance, make it red, and add a thick outline to it.

As part of that callback function then, we will first select the target of the event:

```
#updateBars(){
    this.bars = this.bars.data(this.data, d=>d[0])
        ...
        .on('mouseover', (event, datum)=>{
            d3.select(event.target)
        });
}
```

From then, we can modify the selection as we see fit:

- Set its *fill* style to '*red*'.
- Set its *stroke* style to '*black*'.
- Set its *stroke-width* style to '*4px*'.

Try to do the above and check that it works on your test web page.

Because we are changing styles, we could instead interact with our CSS rules (it will make things easier when we try to remove highlights later). The code below will simply add a new CSS class to the one rectangle we are interacting with. It's not replacing the existing *bar* class, it's adding another one: *highlighted*. In the end, the rectangle we interact with will have two classes: *bar* and *highlighted*.

```
#updateBars(){
    this.bars = this.bars.data(this.data, d=>d[0])
        ...
        .on('mouseover', (event, datum)=>{
            d3.select(event.target)
                .classed('highlighted', true)
        });
}
```

Which means that we can now edit our CSS stylesheet, and add new rule for rectangles that are both *bar* and *highlighted*:

```
svg.barchart rect.bar{
    ...
}

svg.barchart rect.bar.highlighted{
    ...
}
```

Remember to check the CSS selector syntax if this gets confusing:
https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors

Removing Highlights

By now, if everything works correctly, you would have probably noticed something: yes you are highlighting bars, but the highlight doesn't seem to go away.

And that's normal: we haven't provided any instructions to do so. As far as the code is concerned, we are simply adding a new CSS class when the mouse moves over a rectangle.

We need a different type of event to trigger a response when the mouse moves out of a rectangle: '*mouseout*'.

We can attach multiple event listeners to selections, we just need to use another the **`.on()`** method:

```
#updateBars(){
    this.bars = this.bars.data(this.data, d=>d[0])
        ...
        .on('mouseover', (event, datum)=>{
            ...
        })
        .on('mouseout', (event, datum)=>{
            ...
        });
}
```

From then, to remove the visual highlights, we simply need to remove the *highlighted* class from the bar. How do you think you could do that?

Tooltips

Tooltips are a great way to allow users to query extra details about the visualisation they are currently looking at, without crowding the visualisation with too much text.

There are many JS libraries that let you program how and where to display these floating elements.

With SVG, we can use **`title`** elements, that add as children of other elements. All browsers will be able to display the text inside this title element as a floating tooltip when the element is hovered.

With out bar chart class, we could add these as part of the update method using the **`.append()`** function. This approach often seems to work, but, if you later decide to re-render your bar chart (for example with new data), these **`title`** elements will pile up.

Instead, we can:

- Select all title elements, children of our rectangle elements.
- Bind the rectangles' data to these titles.
- Join titles.
- Update the titles' texts.

This approach will ensure that old titles are removed or have their text updated and that new titles are added.

```
#updateBars(){
    this.bars = this.bars.data(this.data, d=>d[0])
        ...

    this.bars.selectAll('title')
```

```
        .data(d=>[d])
        .join('title')
        .text(d=>`${d[0]}: ${d[1]}`);
}
```

Note the way we bind data to these titles: we query the datum attached to rectangles, and return it as an array of 1 element: this lets us use the **`.join()`** method.

Tutorial 10 – Linked Interactions

With Tutorial 9 we looked at implementing events (setting up a listener and programming an handler callback) within one visualisation. In the lectures, however, we discussed interactions between visualisations: Linked Interactions.

Before we look at how to implement such things, we must discuss ***scopes***.

Scopes

In JavaScript (and other programming languages), we talk about scope when we want to describe the context within which variables or functions are visible. Here are the basic JS rules:

- Each function*, module, class, block of code (**`if`** or **`for`**) defines its own scope.
- A child (e.g., function inside a class) can see its parents' scope, all the way to the global scope.
- Siblings (e.g., two classes defined at the same level) cannot see each other's scopes. They know of each other's existence, but not the details of their implementation.
- Similarly, a parent cannot see inside their children's scope.

*This does not apply to arrow functions (**`() => {}`**) which inherit their parent's scope.

In practice it means is that:

- We have two classes, **`A`** and **`B`**.
- Class **`A`** has a method **`functionA`** that can invoke other methods from class **`A`**.
- Class **`B`** has a method **`functionB`** that can invoke other methods from class **`B`**.
- However, **`functionA`** cannot invoke **`functionB`** directly.

To bypass this limitation, we can use callbacks:

- We can instantiate an object **`objA`** of class **`A`** and an object **`objB`** of class **`B`**.
- If, part of **`objA`**'s **`functionA`** process is to instruct **`objB`**'s **`functionB`** method to execute, we can provide a reference to **`objB.functionB`** in the argument of **`objA.functionA`**'s call.

```
class A {
    //...
    functionA(callback){
        let x = 'Value from class A';
        callback(x);
    }
}

class B{
    //...
    functionB(value){
        console.log(`Class B processing value: ${value}`)
    }
}
```

```
let objA = new A(),
    objB = new B();

objA.functionA(objB.functionB);
// Class B processing value: Value from class A
```

Linked Interactions

What does that mean for our interactions?

Imagine the following scenario:

- A **`barchart`** object has public methods to:
  - Be given *references* to callback functions, that can be stored and bound to its selections' events.
  - Mark some of its bars as highlighted, given a list of keys.
- In our **`main.js`** script:
  - We instantiate to **`barchart`** objects, showing the same categories, but different values.
  - We implement a function that, given a datum, will query its key, and then instruct the two **`barchart`** objects to highlight bars with this key.
  - We then instruct each **`barchart`** object to register this function as their callback one the mouse hovers a bar.

We would have then created an event system where, if the user hovers over a bar in one bar chart, this bar **and** the corresponding bar in the other bar chart would be highlighted.

This pattern can be adapted to work on more advanced levels: for instance, if a bar gets clicked on, the data sent to a scatter plot gets filtered for just this category…

Exercise – Linked Selection and Filters

In this exercise, we will continue to tinker with our BarChart class, and aim to accomplish two things:

- Implement a linked highlight between two bar charts.
- Implement a selection and filter between two bar charts.

Changing the **`BarChart`** Class

The first thing we need to do is modify the **`BarChart`** class.

1 – We should add object attributes that will store callback references for us.

```
export class BarChart{
    // other attributes ...

    barClick = ()=>{};
    barHover = ()=>{};
    barOut = ()=>{};

    // constructor and methods...
```

```
}
```

Note that we initialise these callbacks with default behaviours, which are to do nothing. But this has the advantage to guarantee that these functions are defined.

2 – We should create a private method that will rebind these callbacks to events. Changing the function stored in **`barClick`** will not directly update the behaviour of the event handler, we therefore need to do this rebinding whenever we update **`barClick`** (or other event callbacks).

```
#updateEvents(){
    this.bars
        .on('mouseover', this.barHover)
        .on('mouseout', this.barOut)
        .on('click', (e,d)=>{
            console.log('bar clicked:', d);
            this.barClick(e,d);
        });
}
```

Note the two ways we can do this rebinding:

- We can simply put a reference to the callback (**`this.barHover`**).
- We can also call the callback in an anonymous function, that will also execute other statements. For example: print the datum when the bar is clicked, but also invoke the click callback (**`this.barClick(e,d)`**).

Because we have this private method now, we can remove the **`.on(…)`** methods from the **`#updateBars`** method. Instead we can call the **`#updateEvents`** method at the end.

```
#updateBars(){
    this.bars = //...

    this.#updateEvents();
}
```

3 – We should create several public methods to allow external scripts to provide a callback functions. We can put a default value (**`()=>{}`**), this will allow callbacks to be reset if no argument is provided when the method is invoked.

```
setBarClick(f = ()=>{}){
    // register new callback
    this.barClick = f;
    // rebind callback to event
    this.#updateEvents();
    // return this for chaining
    return this;
}
```

You should add other methods for the other callbacks (**`barHover`** and **`barOut`**).

4 – The final modification we should make, is to have a public method allowing external scripts to instruct the bar chart to highlight bars with specific keys.

```
highlightBars(keys = []){
    // reset highlight for all bars
    this.bars.classed('highlighted', false);
    // filter bars and set new highlights
    this.bars.filter(d=>keys.includes(d[0]))
        .classed('highlighted', true);
    // return this for chaining
    return this;
}
```

Again, notice that we give a default value to the array of keys (empty array). If this method gets called with no arguments, then all highlights are removed.

Configuring External Callbacks

The **`BarChart`** class now has all the necessary modifications that will allow us to configure an event cycle that will link interactions between charts.

We will work with the same dataset as Lab 2. We will set things up such that:

- One bar chart shows revenues per year (in ascending order).
- One bar chart shows number of releases per year (in ascending order).
- One bar chart shows number of releases per genre.

```
let movieDataFull = await d3.csv('data/movies_mock.csv', d=>{
    return {
        year: +d.release_year,
        revenues: parseFloat(d.revenues),
        genre: d.genre
    }
})

let bar1 = new BarChart('#bar1', 800, 500, [10,40,65,10]),
    bar2 = new BarChart('#bar2', 800, 500, [10,40,65,10]),
    bar3 = new BarChart('#bar3', 800, 500, [10,40,65,10]);

let sortYears = (a,b)=>a[0]-b[0];
let yearRevenues = d3.flatRollup(movieDataFull, v=>d3.sum(v,
d=>d.revenues), d=>d.year).sort(sortYears),
    yearCount = d3.flatRollup(movieDataFull, v=>v.length,
d=>d.year).sort(sortYears),
    genreCount = d3.flatRollup(movieDataFull, v=>v.length, d=>d.genre);

bar1.setLabels('Year', 'Total Revenues')
    .render(yearRevenues);
bar2.setLabels('Year', 'Total Number of Releases')
    .render(yearCount);
bar3.setLabels('Genre', 'Total Number of Releases')
    .render(genreCount);
```

Make sure to have the appropriate HTML elements set up too.

Linked Highlight

Let's first try to link the two bar charts that display years: when one of their bars gets moused over, we want to highlight that bar, and the bar in the other chart with the corresponding year.

We therefore need a callback that can:

- Retrieve the **`year`** value associated with the bar that triggers the event.
- Instruct **`bar1`** to highlight the corresponding bar.
- Instruct **`bar2`** to highlight the corresponding bar.

```
let highlightYear = (e,d)=>{
    let year = d[0];
    bar1.highlightBars([year]);
    bar2.highlightBars([year]);
}
```

Note that we use the D3 callback format for events: **`(event, datum) => {…}`**.

We also need a callback function that can undo the highlight.

```
let rmvHighlightYear = (e,d)=>{
    bar1.highlightBars();
    bar2.highlightBars();
}
```

All that is left is to link these callbacks to the two bar charts.

```
bar1.setBarHover(highlightYear).setBarOut(rmvHighlightYear);
bar2.setBarHover(highlightYear).setBarOut(rmvHighlightYear);
```

With this, each bar chart will, when the **`mouseover`** and **`mouseout`** events are triggered, call the functions defined in **`main.js`**, that both exist in a scope that see **`bar1`** and **`bar2`**.

Selection and Filter

With this interaction we want to be able to click on one of the bars showing the number of releases for a genre, to then filter the other two bar charts to only show data applicable to this selected genre.

We need a callback to:

- Retrieve the genre value associated with the bar that triggers the event.
- Filter our dataset with this genre value.
- Recompute the aggregations for revenues by year and releases by year with the filtered dataset.
- Rerender bar charts with the new data.

As a bonus, we could even redefine the axis legend for the updated bar charts.

```
let filterGenre = (e,d)=>{
    let genre = d[0];
    let fiteredData = movieDataFull.filter(d=>d.genre===genre),
        yearRevenuesFiltered = d3.flatRollup(fiteredData, v=>d3.sum(v,
d=>d.revenues), d=>d.year).sort(sortYears),
```

```
        yearCountFiltered = d3.flatRollup(fiteredData, v=>v.length,
d=>d.year).sort(sortYears);
    bar1.setLabels('Year', `Revenues ${genre}`)
        .render(yearRevenuesFiltered);
    bar2.setLabels('Year', `Number of Releases ${genre}`)
        .render(yearCountFiltered);
}
```

And then, we need to link this callback to the right bar chart.

```
bar3.setBarClick(filterGenre);
```

Summary

Linked Highlight and *Selection and Filter* are essential interaction paradigms. The patterns we implemented above allow you to set these interactions in a reusable manner by decoupling your charts and leaving the linkage setup to a controller script (**`main.js`** in this instance).

You can apply this methodology to other interactions.

But we missed one thing: there are no visual feedback for click on a bar in **`bar3`**. How would you implement this? (Hint: you cannot reuse the **`highlighted`** CSS class, this one is now used for hovering over a bar; but perhaps you can get inspiration from it.)

Tutorial 11 – D3 Transitions

D3 Transition is yet another module provided by D3: https://d3js.org/d3-transition

It implements *Transitions*. Like D3 Selections, D3 Transitions give you an interface to manipulate DOM element based on your data. Unlike D3 Selections, D3 Transitions let make these manipulations via **interpolated changes** (rather than instantaneous updates).

In short, transitions is what allow us to implemented animations in our charts.

Note that data binding and element creation is not supported by D3 Transitions, instead, you must use D3 Selections first, and then make these D3 Transitions.

A D3 Selection can be transformed into a D3 Transition by calling the **`.transition()`** method. From that point onward, any changes to the attributes (**`.attr()`**) or styles (**`.style()`**) will be interpolated, or changed over a short period of time.

```
let bars = svg.selectAll('rect')
    .data(barData)
    .join('rect')

// call .transition outside the assignment chain (bars = ...)
// otherwise bars will be a transition object, not a selection
bars.transition()
    .attr('width', 100)
    .attr('height', d=>d*100)
```

Colours, numbers, transforms, and some strings are automatically interpolated by D3. Otherwise, you can write your own interpolator with the **`.attrTween()`** method.

You can also control the timing of your transitions:

- **`.duration(val)`** sets the total duration of the transition (in milliseconds).
- **`.delay(val)`** sets a timer before the transition starts (in milliseconds). This is useful if you want to stagger transitions.
- **`.ease(function)`** sets an ease function to apply to the transition.

Exercise – Let's Make an Animated Bar Chart

Using transitions, we can now animate our bar charts, whenever we decide to update their data.

We only need to change the code inside the **`#updateBars()`** method. Let's divide our method chain: first we do the bind and join of rectangle elements, then we place and size them:

```
#updateBars(){
    // bind and join rectangles to data
    this.bars = this.bars
        .data(this.data, d=>d[0])
        .join('rect')
        .classed('bar', true);
    // placement and sizing
```

```
    this.bars.attr('x', d=>this.scaleX(d[0]))
        .attr('width', this.scaleX.bandwidth())
        .attr('y', d=>this.scaleY(d[1]))
        .attr('height', d=>this.scaleY(0)-this.scale(d[1]));

    // ...
}
```

With this new structure, we can quickly apply a transition to the placement and sizing of bars:

```
#updateBars(){
    // bind and join rectangles to data
    this.bars = this.bars
        .data(this.data, d=>d[0])
        .join('rect')
        .classed('bar', true);
    // animate placement and sizing
    this.bars.transition().duration(500)
        .attr('x', d=>this.scaleX(d[0]))
        .attr('width', this.scaleX.bandwidth())
        .attr('y', d=>this.scaleY(d[1]))
        .attr('height', d=>this.scaleY(0)-this.scale(d[1]));

    // ...
}
```

Look at the effects this has on your web applications. You should notice two things:

- When we click on a bar in the third bar chart, we can see that the other two bar chart nicely animated when they update their data.
- When we refresh the page, we can see how the bars are initially render: they appear to progressively grow and place themselves from the top left-hand corner. That's because they, in fact, are placed there initially.

To make a nice animation, where one can see bars rising from the bottom of the chart, we need to place them there. We could simply set the necessary attributes before we apply the transition, but that means we would see bars rising all the time, including when we simply want to see an existing bar updated. Instead, we need to only initialise new bars, not old ones. Making the distinction between the two means that we must breakdown the **`.join(…)`** method (see *Lab 1 – Tutorial 2* for more details).

```
#updateBars(){
    // bind and join rectangles to data
    this.bars = this.bars
        .data(this.data, d=>d[0])
        .join(
            // initial placement of new rectangles
            enter=>enter.append('rect')
                .attr('x', d=>this.scaleX(d[0]))
                .attr('width', this.scaleX.bandwidth())
```

```
                .attr('y', d=>this.scaleY(0)) // aligned at bottom
                .attr('height', 0), // no height
            // leave exisiting rectangles untouched
            update=>update,
            // animate old rectangles disappearing
            exit=>exit.transition().duration(300)
                .attr('y', d=>this.scaleY(0)) // aligned at bottom
                .attr('height', 0) // no height
                .remove() // destroy rectangle when finished
        )
        .classed('bar', true);
    // animate placement and sizing (enter + update only)
    this.bars.transition().duration(300)
        .attr('x', d=>this.scaleX(d[0]))
        .attr('width', this.scaleX.bandwidth())
        .attr('y', d=>this.scaleY(d[1]))
        .attr('height', d=>this.scaleY(0)-this.scale(d[1]));

    // ...
}
```

We should now have nice animations with our bar chart updates. You can apply the same methodology to other charts (some might require the use of **`.attrTween()`**).

Tutorial 12 – Advanced Behaviours

D3 Behaviours are a set of modules that implement common and complex interactions with visualisations. They provide:

- Custom series of events.
- Special event callbacks, with tailored event objects.
- Interaction specific settings.

Using these predefined complex behaviours also has the advantage of covering all sorts of inputs (mouse or touch), ensuring that a natural range of input is available, e.g., pinch to zoom.

There are three behaviours:

- Zoom, to implement navigation: https://d3js.org/d3-zoom
- Brush, to select items in batches: https://d3js.org/d3-brush
- Drag, to manipulate the position of items: https://d3js.org/d3-drag

We will focus on the first two in this tutorial.

Like other generator functions, a behaviour is first instantiated, with options set by chained methods. Then, we attach the behaviour to a selection (the one that will listen for events) by using the **`.call()`** method.

```
// General pattern
let behaviour = d3.behaviour() // d3.zoom() or d3.brush()
    .settingA(valueA)
    .settingB(valueB)
    .on('customEvent', callback);

selection.call(behaviour);
```

A side note:

> We have encountered the **`.call()`** method a few times now (the first was when we set axes). This method allows us to call a function on a selection exactly once. The function we pass must be able to take the *selection* as an argument. The D3 documentation gives a good explanation: https://d3js.org/d3-selection/control-flow#selection_call
>
> There is another similar method: **`.each()`**. Instead of running on the selection once, this method will be called on each item inside the selection, in order, and must take the *datum* and *index* as arguments. Again, the D3 documentation has a good explanation: https://d3js.org/d3-selection/control-flow#selection_each

Back to our behaviours.

D3 Zoom

D3 Zoom is all about zooming and panning, or more generally navigation. In terms of codes, it essentially helps us with ***translation*** and ***scale*** transformations. We can declare a new zoom behaviours by calling **`d3.zoom()`**.

It comes with many options to set. But you should remember the following:

- **`.extent([[x0,y0],[x1,y1]])`** lets us set the *viewport* of your navigation, i.e., the region within which the zoom behaviour will ensure the elements we eventually transform are visible. **`[x0,y0]`** and **`[x1,y1]`** refer to the top left-hand and bottom right-hand corners respectively. If you set these to your SVG's corners, then elements transformed by your behaviour will remain in the SVG's viewport.
- **`.translateExent([[x0,y0],[x1,y1]])`** allows us to define the region that can be navigated with the zoom behaviour. In conjunction with **`.extent()`** it sets the constraint of our panning interaction.
- **`.scaleExent([k0,k1])`** allows us to define the bounds of our scaling factor. In conjunction with **`.extent()`** it sets the constraint of our zooming interaction.

If you don't set any of the above, the navigation will be unconstrained.

Once we have setup our zoom behaviour, we can specify it's callbacks with the **`.on()`** method (like we did with selection events above). There are three custom events we can listen to and provided callbacks for:

- **`'start'`** when the mouse button is pressed down.
- **`'zoom'`** while the mouse is being dragged.
- **`'end'`** when the mouse button is released.

Each of the event handlers first parameter is an event object that comes with additional properties, notably a **`transform`** object with: the x and y panning translations, and the zooming scale factor. We can use this object directly to apply the transformation to elements of our charts:

```
zoomBehaviour.on('zoom', (event)=>{
    selection.attr('transform', event.transform);
});
```

A zoom behaviour also comes with several method to apply navigation programmatically: **`zoom.scaleTo()`**, **`zoom.scaleBy()`**, **`zoom.translateTo()`**, **`zoom.translateBy()`**, **`zoom.transform()`**.

D3 Brush

D3 Brush lets us implement a brush selection to our chart. It does two things for us: it returns bounds for the region that is being brushed and it draws a background rectangle making that region visible. We can declare a new brush behaviour with **`d3.brush()`**. Alternatively, we can use **`d3.brushX()`** or **`d3.brushY()`** if we want brushes that only horizontal or vertical (respectively).

Amongst the many options available, you should remember **`.extent([[x0,y0],[x1,y1]])`**. It lets us define the area within which the brush input can be used.

Like with D3 Zoom, there are three custom events that we can listen to: **`'start'`**, **`'brush'`** and **`'end'`**.

The event handler for these takes an event object as first parameter, with additional properties. The **`selection`** property is an array that contains the bound of the area being brushed (either the top left-hand and bottom right-hand corners, or simply the extremities of vertical/horizontal brushes).

Typically, we can use these values to filter our data, and update the visualisation (or other linked visualisations) accordingly. Note that the values inside this **`selection`** property are in pixels, so we sometime need to use the inverse of our scales to find the matching values in our data.

We can change the behaviour programmatically too, for example with the **`brush.move()`** and **`brush.clear()`** methods.

Exercise – D3 Behaviours Sandbox

Along with this lab sheet you should find a .zip file: *F20-21DV_lab3_behaviours.zip*.

It contains a project with sandbox scripts for the behaviours detailed above. Use these scripts to explore D3 behaviours, especially the extent options and event callbacks.