Tutorial 3
Data Joins

DATA VISUALISATION

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

Tutorial 3: Data Joins

In this tutorial, we introduce the *enter-update-exit* pattern, which is a key concept to make D3 visualizations fully interactive.

1. Enter, Update, Exit

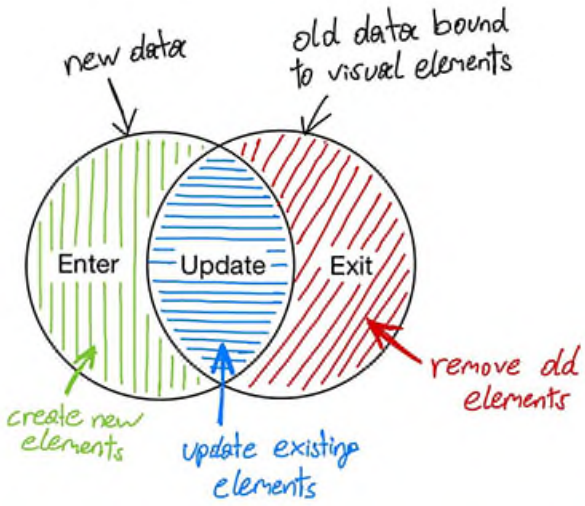
By now you have learned how to load external data and how to map it to visual elements, for example, to create a bar chart. But very often we have to deal with changing data or a continuous data stream rather than a static CSV file. Dynamic data often requires more sophisticated user interfaces that allow users to interact with the data (e.g., filter and sort).

Instead of removing and redrawing visualisations each time new data arrives, we want to update only those affected components to improve loading times and create smooth transitions. We will accomplish this by using D3's enter-update-exit pattern.

The general update pattern

A data-join is followed by operations on the three virtual selections: **enter**, **update** and **exit**. This means that we are merging new data with existing elements. In the merging process we have to consider:

- Enter: What happens to new data values without existing, associated DOM elements?
- Update: What happens to existing elements which have changed?
- Exit: What happens to existing DOM elements which are not associated with data anymore?



To take care of the enter-update-exit (general update) pattern, we have to change the sequence of our D3 code a little bit. Instead of chaining everything together, some code snippets must be separated.

We select our SVG element and bind some data to SVG circles:

```
const svg = d3.select('svg');
const width  = +svg.attr('width');
const height = +svg.attr('height');

let circles = svg.selectAll('circle')
  .data([5, 10, 15]);
```

The length of the dataset is 3 and we select all SVG circles in the document. That means, if there are 3 or more existing circles, the enter selection is empty, otherwise it contains placeholders for the missing elements.

The page is initially empty because we have not appended any circles yet. We can access the *enter selection* and append a new circle for each placeholder with the following statement:

```
circles
  .enter().append('circle')
    .attr('r', d => d)
    .attr('cx', (d,index) => (index * 80) + 50)
    .attr('cy', height/2);
```

(You might have noticed that we've actually already used this pattern multiple times.)

But often you want the exact opposite operation and remove existing elements. In this case, you have to use the `exit` selection. `exit` contains the leftover elements for which there is no corresponding data anymore.

Imagine we call the drawing function again with new data:

```
circles = svg.selectAll('circle')
  .data([20, 30]);
```

The new dataset contains 2 elements but on the visualisation there are currently 3 circles. We can access the *exit selection* and remove the element that has no data-binding anymore:

```
circles.exit.remove();
```

There is still one problem left: *dynamic properties*. We are using a data-dependent radius and the values in the new dataset have changed. For this reason, we have to **update** the dynamic properties (previously set in the *enter selection*) every time we update the data. To do this we use the *merge* function to apply changes to the *enter and update* selection. Putting everything together[1] (see `ex3.1_general_update_pattern/`):

```
const svg = d3.select('svg');
const width  = +svg.attr('width');
const height = +svg.attr('height');

// Call rendering function with 2 datasets sequentially
// the second with a delay of 1 second since loading the web
drawCircles(svg, [5, 10, 15]);
setTimeout(() => {             // Implement delay[2]
  drawCircles(svg, [20, 25]);
}, 1000);

function drawCircles(parent, data) {
  // Data-join (circles now contains the update selection)
  const circles = parent.selectAll('circle')
    .data(data);

  // Enter (initialise the newly added elements)
  const circlesEnter = circles.enter().append('circle');
```

[1] The general update pattern has been coded into a function that takes the state you want to represent in the DOM, and has the side-effect of mutating the DOM so it matches with its input. The structure use here is similar to the one used in REACT components (another powerful visualisation library), with 2 arguments: `parent` (the parent element we want to modify), and `props` (an object with varying elements, although for simplicity here we are only passing our data).

[2] The HTML-built function `setTimeout` takes 2 arguments: first, a function (here with no arguments and only invoking `drawCircles`), and second, the desired delay (in milliseconds).

```
  // Enter and Update (set the dynamic properties of the elements)
  circlesEnter.merge(circles)
    .attr('cx', (d,i) => (i * 80) + 50)
    .attr('cy', height/2)
    .attr('r', d => d);

  // Exit
  circles.exit().remove();
}
```

Animated transitions

Our visualisation is now capable of updating and removing our DOM elements based on the current state of our data, but these changes happen rather abruptly. Imagine we want our circles to pop in when created, and to smoothly adap when their radius changes. We can achieve this easily by adding a D3 `transition()` to the merge section of the general update pattern when setting the dynamic properties of our elements (see folder `ex3.2_animated_transitions/`), with the desired duration specified in milliseconds:

```
  // Enter and Update (set the dynamic properties of the elements)
  circlesEnter.merge(circles)
    .attr('cx', (d,i) => (i * 80) + 50)
    .attr('cy', height/2)
    .transition().duration(500)
      .attr('r', d => d);
```

We can apply the same concept to smoothly shrink to zero the radius of any deleted circle, before removing it from our data join in the exit selection:

```
  // Exit
  circles.exit()
    .transition().duration(500)
      .attr('r', 0)
    .remove();
```

The key function

There is still a small issue with our code. Consider the following case: our initial data is still `[5, 10, 15]`, and we now want to delete the first circle. One might think it would be sufficient to call our function with new data `[10, 15]`. However, if we use the code above to do so (please check!), you will see that while the final circle smoothly disappears, the other two also smoothly change their radii to the new data, which is not the desired behaviour. This brings the concept of **object constancy**: the DOM elements should hold a constant correlation to their corresponding data. Therefore, the D3 data join needs to somehow know how they map up.

For the sake of clarity and simplicity, we have not mentioned an important detail in our previous examples: the *key* function. `selectAll("circle")` selects all circle-elements, and if we chain it with `.data(data)` we are joining the given data with the selected circles. The default key function applies and the keys are assigned by index. In our example it will use the first three circles that it finds. The first datum (first item in our array) and the first circle have the key "0", the second datum and circle have the key "1", and so on. If we start the pipeline again with the slightly different array, the index will be used again as the default key to match the new data to the actual circles. There are three circles on the webpage and two items in the new dataset. Therefore, the last circle will be removed and the other two circles will be bound to the new data.

What has been describe above is the simplest method of joining data, and it is often sufficient. However, when object constancy is required, joining by index is insufficient. In this case, you can specify a key function as the second argument (callback function). The key function returns the key for a given datum or element:

```
  const circles = parent.selectAll('circle')
    .data(data, d => d); // could be also d => d.customer_id if d is an object
```

The key function thus allows us to map the data value directly instead of the default by-index behaviour:

As intended, by calling now our function with the new data `[10, 15]`, the first element now disappears, while the remaining ones no longer change in size (see `ex3.3_key_function\`). The disappearance of the first circle is however a bit abrupt. We may want to include the x-position of the circles to the transition in the merge selection (so the remaining ones are smoothly displaced to the left). We may also want to set their initial positions in the enter selection, to avoid the first set of data to enter from the left of the screen. Our final function implementing the general update pattern[3] and animated transitions would look like:

```
function drawCircles(parent, data) {
  // Data-join (circles now contains the update selection)
  const circles = parent.selectAll('circle')
    .data(data, d => d); // could be also d => d.customer_id if d is an object

  // Enter (initialise the newly added elements)
  const circlesEnter = circles.enter().append('circle')
    .attr('cx', (d,i) => (i * 80) + 50);

  // Enter and Update (set the dynamic properties of the elements)
  circlesEnter.merge(circles)
    .attr('cy', height/2)
    .transition().duration(500)
      .attr('cx', (d,i) => (i * 80) + 50)
      .attr('r', d => d);

  // Exit
  circles.exit()
    .transition().duration(500)
      .attr('r', 0)
    .remove();
}
```

[3] Some of the cases considered in this example (such as displacements after deleting elements) are rather borderline, and you often won't need to implement such complicated update patterns.

2. Extensions to the general update pattern

Nested elements

For this part of the tutorial, we'll be using the materials included in `ex3.4_nested_elements/`. Please take a moment to familiarise yourself with the contents of `index.js` and `showFruits_v1.js`. As a main change compared to our previous examples, you will see that most of the functionality to handle our visualisation has been moved to an external module (`showFruits_v1.js`), from which we import the function `showFruits` in `index.js`. Note also the definition of the auxiliary function `updateVis` in `index.js`:

```
const updateVis = () => {
  showFruits(svg, {
    fruits,
    height: +svg.attr('height')
  })
};
```

This is basically a handler to invoke `showFruits`, passing as first argument the selection of our parent element to modify (our SVG), and as second argument `props`, an object containing the current state of all properties required by the function (here, our data in the variable `fruits`, and the SVG's `height`). In this way, upon any changes in our state, we only need to call `updateVis()`: the component becomes responsible for rendering itself based on `props`, making the DOM to match up with any state changes.

Going back to our example, it follows closely the principles seen so far in this tutorial, but with a slightly more complex datatype (`fruits`, including fields for type, amount, and colour) and using now both circles and text labels to visualise the data. For this, `showFruits` implements two separate data joins (one for circles, one for labels), using in both cases the fruit type as key function. While this achieves the desired functionality, there is an awful amount of repetition between both data joins, especially in terms of positioning the elements.

We can address this by defining a single data join, containing a separate SVG group for each fruit and use its general update pattern to control all the positioning (see `showFruits_v2.js`)[4]:

```
  const groups = parent.selectAll('g')
    .data(fruits, d => d.type);
  const groupsEnter = groups.enter().append('g')
    .attr('transform', (d,i) => `translate(${(i * 100) + 80},${height/2})`);
  groupsEnter.merge(groups)
    .transition().duration(1000)
      .attr('transform', (d,i) => `translate(${(i * 100) + 80},${height/2})`);
  groups.exit().remove();
```

Now, instead of creating new data joins for circles and text elements, we can simply leverage the one created for the groups. As the main group elements have already been created in their `enter` selection, we only need to update (`merge`) the rest of dynamic properties, as shown below for the circles and done similarly for the text labels (see `showFruits_v2.js`). This exemplifies in brief how we perform data joins on nested elements.

```
  const circle = groups.select('circle');
  groupsEnter.append('circle')
    .merge(circle)
      .transition().duration(1000)
        .attr('fill', d => colourScale(d.colour))
        .attr('r', d => d.amount);
```

[4] You will need to change the module that is loaded in `index.js`.

Singular elements

To finalise this tutorial, situations may arise where we need to manage a single element together with a data join. In our previous example, imagine we want to have a background behind our fruits and labels. If we added the following piece of code to our function `showFruits`:

```
const background = parent.append('rect')
  .attr('width', 800)
  .attr('height', 200)
  .attr('y', 160)
  .attr('rx', 80);  // roundness of corners
```

the problem we'd be creating is that a new rectangle is generated every time the function is invoked (which, by order of definition, would overlap all our circles and text elements).

To overcome this, we can create a separate data join for this singular element so only one instance of rectangle is created upon initialisation. There is no real data bound to this element, all we need to pass to the `.data()` method is an array with a single element, which can perfectly be `[null]`.[5] Our modified code for handing the singular element inside of `showFruits` (see `ex3.5_singular_elements/`) therefore becomes:

```
const background = parent.selectAll('rect')
  .data([null])
  .enter().append('rect')
    .attr('width', 800)
    .attr('height', 200)
    .attr('y', 160)
    .attr('rx', 80);
```

Although not explicitly illustrated here, if the properties of this singular element also depend on the data (e.g., its size, colour…), a `merge` section could also be added to its data join as seen earlier.

[5] This is done, for example, in the definition of D3 axes.