Enter and exit

In the Data joins section we show how to join an array of data to a D3 selection.

To recap, given some DOM elements:

```
<div id="content">
  <div></div>
  <div></div>
  <div></div>
</div>
```

and some data:

```
var myData = [ 10, 40, 20 ];
```

we join the array to the `div` elements using:

```
d3.select('#content')
  .selectAll('div')
  .data(myData);
```

In this example `myData` is the same length as the selection.

However, what happens if the array has more (or less) elements than the selection?

- if the array is **longer** than the selection there's a **shortfall** of DOM elements and we need to **add** elements
- if the array is **shorter** than the selection there's a **surplus** of DOM elements and we need to **remove** elements

Fortunately D3 can help in adding and removing DOM elements using two functions `.enter` and `.exit`.

.enter

`.enter` identifies any DOM elements that need to be added when the joined array is longer than the selection. It's defined on an **update selection** (the selection returned by `.data`):

```
d3.select('#content')
  .selectAll('div')
  .data(myData)
  .enter();
```

`.enter` returns an **enter selection** which basically represents the elements that need to be added. It's usually followed by `.append` which adds elements to the DOM:

```
d3.select('#content')
  .selectAll('div')
  .data(myData)
  .enter()
  .append('div');
```

Let's look at an example. Suppose we have the following `div` elements:

```
<div id="content">
  <div></div>
  <div></div>
  <div></div>
</div>
```

and this data:

```
var myData = ['A', 'B', 'C', 'D', 'E'];
```

we use `.enter` and `.append` to add `div` elements for D and E:

```
  d3.select('#content')
    .selectAll('div')
    .data(myData)
    .enter()
    .append('div');
```

Add elements using .enter and .append

View source | Edit in GistRun

Note that we can join an array to an empty selection which is a very common pattern in the examples on the D3 website.

Add elements using .enter and .append

View source | Edit in GistRun

.exit

`.exit` returns an **exit selection** which consists of the elements that need to be removed from the DOM. It's usually followed by `.remove`:

```
d3.select('#content')
  .selectAll('div')
  .data(myData)
  .exit()
  .remove();
```

Let's repeat the example above, but using `.exit`. Starting with elements:

```
<div id="content">
  <div></div>
  <div></div>
  <div></div>
</div>
```

and data (notice that it's shorter than the selection):

```
var myData = ['A'];
```

we use `.exit` and `.remove` to remove the surplus elements:

```
d3.select('#content')
  .selectAll('div')
  .data(myData)
```

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

Remove elements using .exit and .remove

View source | Edit in GistRun

Putting it all together

So far in this section we've not concerned ourselves with modifying elements using functions such as `.style`, `.attr` and `.classed`.

D3 allows us to be specific about which elements are modified when new elements are entering. We can modify:

- the existing elements
- the entering elements
- both existing and entering elements

(Most of the time the last option is sufficient, but sometimes we might want to style entering elements differently.)

The existing elements are represented by the **update selection**. This is the selection returned by `.data` and is assigned to `u` in this example:

```
var myData = ['A', 'B', 'C', 'D', 'E'];

var u = d3.select('#content')
  .selectAll('div')
  .data(myData);

u.enter()
  .append('div');

u.text(function(d) {
  return d;
});
```

Update existing elements

View source | Edit in GistRun

When the button is clicked, new elements are added, but because `.text` is only called on the update selection, it's only the existing elements that are modified. (Note that if the button is clicked a second time, all the elements are modified. This is because the selection will contain all 5 `div` elements. Don't worry about this too much if you're new here!)

The entering elements are represented by the **enter selection**. This is the selection returned by `.enter`. We can modify the enter selection using:

```
var myData = ['A', 'B', 'C', 'D', 'E'];

var u = d3.select('#content')
  .selectAll('div')
  .data(myData);

u.enter()
  .append('div')
  .text(function(d) {
    return d;
  });
```

Update entering elements

View source | Edit in GistRun

When the button is clicked, new elements are added and their text content is updated. Only the entering elements have their text updated because we call `.text` on the enter selection.

If we want to modify the existing **and** entering elements we could call `.text` on the update and enter selections.

However D3 has a function `.merge` which can merge selections together. This means we can do the following:

```
var myData = ['A', 'B', 'C', 'D', 'E'];

var u = d3.select('#content')
  .selectAll('div')
  .data(myData);

u.enter()
  .append('div')
  .merge(u)
  .text(function(d) {
    return d;
  });
```

Update all

View source | Edit in GistRun

> This is a big departure from v3. The entering elements were implicitly included in the update selection so there was no need for `.merge`.

General update pattern

A common pattern (proposed by D3's creator Mike Bostock) is to encapsulate the above behaviour of adding, removing and updating DOM elements in a single function:

```
function update(data) {
  var u = d3.select('#content')
    .selectAll('div')
    .data(data);

  u.enter()
    .append('div')
    .merge(u)
    .text(function(d) {
      return d;
    });

  u.exit().remove();
}
```

View source | Edit in GistRun

Typically the update function is called whenever the data changes.

Here's another example where we colour entering elements orange:

```
function update(data) {
  var u = d3.select('#content')
    .selectAll('div')
    .data(data);

  u.enter()
    .append('div')
    .classed('new', true)
    .text(function(d) {
      return d;
    });

  u.text(function(d) {
      return d;
    })
    .classed('new', false);

  u.exit().remove();
}
```

View source | Edit in GistRun

Data join key function

When we do a data join D3 binds the first array element to the first element in the selection, the second array element to the second element in the selection and so on.

However, if the order of array elements changes (such as during element sorting, insertion or removal), the array elements might get joined **to different DOM elements**.

We can solve this problem by providing `.data` with a **key function**. This function should return a unique id value for each array element, allowing D3 to make sure each array element stays joined to the same DOM element.

Let's look at an example, first using a key function, and then without.

We start with an array `['Z']` and each time the button is clicked a new letter is added at the start of the array.

Because of the key function **each letter will stay bound to the same DOM element** meaning that when a new letter is inserted each existing letter transitions into a new position:

Insert element

View source | Edit in GistRun

Without a key function the DOM elements' text is updated (rather than position) meaning we lose a meaningful transition effect:

Insert element

View source | Edit in GistRun

There's many instances when key functions are not required but if there's any chance that your data elements can change position (e.g. through insertion or sorting) and you're using transitions then you should probably use them.