April 26, 2013 / Mike Bostock

How Selections Work

Any sufficiently advanced technology is indistinguishable from magic. –Arthur C. Clarke

In the past I have presented simplified descriptions of D3's selections, providing only enough detail to get started. This article takes a more comprehensive approach; rather than saying how to *use* selections, I will explain how selections are *implemented*. This may take longer to read, but it should dispel any magic and help you master data-driven documents.

日本語

The structure of this article may at first seem arbitrary. It describes the internal workings of selections rather than the design motivations, so you may wonder why such mechanisms are needed. This is simply because it is easier to lay all the pieces out first before explaining how everything works in tandem. By the time you read to the end, the intent of the design as well as its function should be clear.

D3 is a visualization library, so this article incorporates visual explanations to accompany the text. In subsequent diagrams, the left side of the diagram will show the structure of selections, while the right side will show the structure of data:

Rounded rectangles such as thing indicate JavaScript objects of various types, ranging from literal objects (`{foo: 16}`), primitive values (`"hello"`), arrays of numbers (`[1, 2, 3]`) to DOM elements. Certain special object types are colored, including selection, array, and element. References from one object to another are indicated with connecting lines (———). For example, an array containing the number 42 looks like:

```
var array = [42];
```

Wherever possible, the code that generates the given selection appears immediately above the diagram. Opening your browser's JavaScript console and creating selections interactively is a great way to test your understanding of the text!

Let's begin.

A Subclass of Array

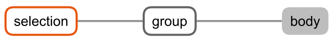
You were probably told that selections are arrays of DOM elements. False. For one, selections are a *subclass* of array; this subclass provides methods to manipulate selected elements, such as setting attributes and styles. Selections inherit native array methods as well, such as array.forEach and array.map. However, you won't often use native methods as D3 provides convenient alternatives, such as selection.each. (A few native methods are overridden to adapt their behavior to selections, namely selection.filter and selection.sort.)

JavaScript doesn't yet support array subclasses directly, so arrays are subclassed through prototype chain injection.

Grouping Elements

Another reason selections aren't literally arrays of elements is that they are *arrays of arrays* of elements: a selection is an array of groups, and each group is an array of elements. For example, d3.select returns a selection with one group containing the selected element:

```
var selection = d3.select("body");
```

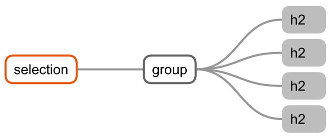


In the JavaScript console, try running this command and inspecting the group as `selection[0]` and the node as `selection[0][0]`. While accessing a node directly is supported by D3's API, for reasons that will soon be apparent it is more common to use selection.node.

In Chrome, you can open the console with the keyboard shortcut ⌥⌘J.

Likewise, d3.selectAll returns a selection with one group and any number of elements:

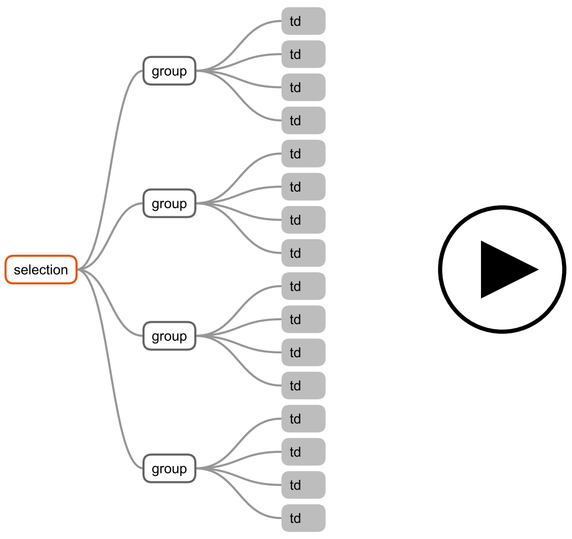
```
d3.selectAll("h2");
```



While the selection is a subclass of array, groups are plain arrays.

Selections returned by d3.select and d3.selectAll have exactly one group. The only way for you to obtain a selection with multiple groups is selection.selectAll. For example, if you select all table rows and then select the rows' cells, you'll get a group of sibling cells for each row:

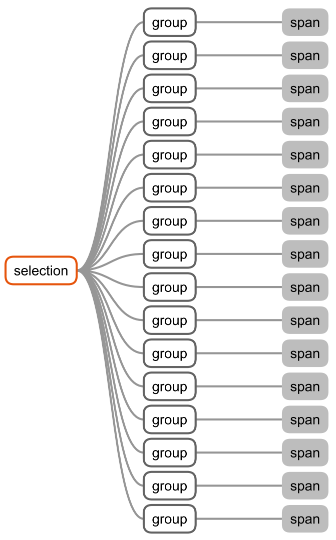
```
d3.selectAll("tr").selectAll("td");
```

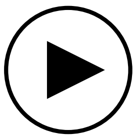


Although this selection's groups all have four elements, in general a selection's groups can have different numbers of elements. Some groups might even be empty!

With selectAll, **every element in the old selection becomes a group in the new selection**; each group contains an old element's matching descendant elements. So, if each table cell contained a span element, and you called selectAll a third time, you'd get a selection with sixteen groups:

```
d3.selectAll("tr").selectAll("td").selectAll("span");
```





Try inspecting a selection in the JavaScript console to find the groups' parent nodes. For more on this topic, read my previous tutorial on nested selections.

Each group has a `parentNode` property which stores the shared parent of all the group's elements. The parent node is set when the group is created. Thus, if you call `d3.selectAll("tr") .selectAll("td")`, the returned selection contains groups of td elements, whose parents are tr elements. For selections returned by d3.select and d3.selectAll, the parent element is the document element.

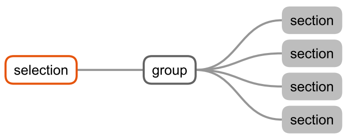
Most of the time, you can safely ignore that selections are grouped. When you use a function to define a selection.attr or selection.style, the function is called for each element; the main difference with grouping is that the second argument to your function (`i`) is the within-group index rather than the within-selection index.

Non-Grouping Operations

Only selectAll has special behavior regarding grouping; select preserves the existing grouping. The select method differs because there is exactly one element in the new selection for each element in the old selection. Thus, select also propagates data from parent to child, whereas selectAll does not (hence the need for a data-join)!

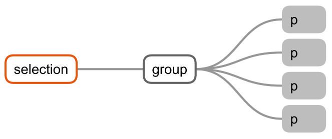
The append and insert methods are wrappers on top of select, so they also preserve grouping and propagate data. For example, given a document with four sections:

```
d3.selectAll("section");
```



If you append a paragraph element to each section, the new selection likewise has a single group with four elements:

```
d3.selectAll("section").append("p");
```



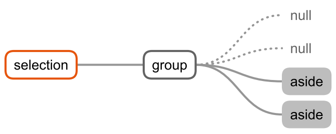
Note that the `parentNode` for this selection is still the document element because selection.selectAll has not been called to regroup the selection.

Null Elements

Groups can contain nulls to indicate missing elements. Nulls are ignored for most operations; for example, D3 skips null elements when applying styles and attributes.

Null elements can occur when selection.select cannot find a matching element for the given selector. The select method must preserve the grouping structure, so it fills the missing slots with null. For example, if only the last two sections have asides:

```
d3.selectAll("section").select("aside");
```



Here, selection.node would return `selection[0][2]`, because this aside is the first non-null element.

As with grouping, you can usually ignore null elements, but note their use in preserving the grouped structure of a selection and its within-group index.

Bound to Data

Perhaps surprisingly, data is *not* a property of the selection, but a property of its elements. This means that when you bind data to a selection, the data is stored in the DOM rather than in the selection: data is assigned to the `__data__` property of each element. If an element lacks this property, the associated datum is undefined. Data is therefore persistent while selections can be considered transient: you can reselect elements from the DOM and they will retain whatever data was previously bound to them.

Data is bound to elements one of several ways:

- Joined to groups of elements via selection.data.
- Assigned to individual elements via selection.datum.
- Inherited from a parent via append, insert, or select.

To verify that data is a property of elements, see selection.datum's implementation.

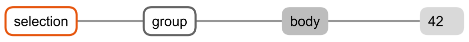
While there is no reason to set the `__data__` property directly when you can use selection.datum, doing so illustrates how data binding is implemented:

```
document.body.__data__ = 42;
```

You might also find this knowledge useful when inspecting selections in your browser's developer tools; `$0.__data__` shows the data bound to the inspected element.

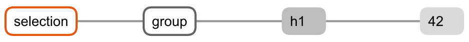
The D3-idiomatic equivalent is to select the body and call datum:

```
d3.select("body").datum(42);
```



If we now append an element to the body, the child automatically inherits data from the parent:

```
d3.select("body").datum(42).append("h1");
```



And that brings us to the last method of binding data: the mysterious join! But before we can achieve enlightenment, we must answer a more existential question.

What is Data?

Data in D3 can be any array of values. For example, an array of numbers:

```
var numbers = [4, 5, 18, 23, 42];
```

Or an array of objects:

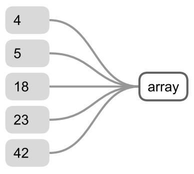
```
var letters = [
  {name: "A", frequency: .08167},
  {name: "B", frequency: .01492},
  {name: "C", frequency: .02780},
  {name: "D", frequency: .04253},
  {name: "E", frequency: .12702}
];
```

One of Scott Murray's many tutorials covers common types of data in JavaScript.

Even an array of arrays:

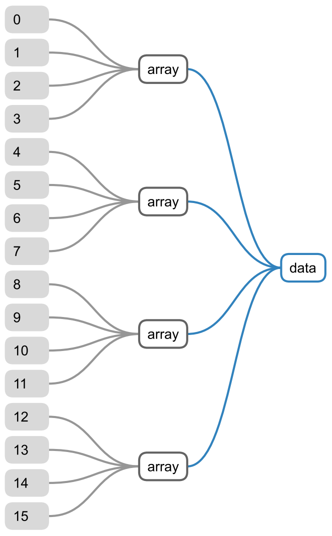
```
var matrix = [
  [ 0,  1,  2,  3],
  [ 4,  5,  6,  7],
  [ 8,  9, 10, 11],
  [12, 13, 14, 15]
];
```

We can mirror the visual representation of selections to represent data. Here's a plain array of five numbers:



Just as selection.style takes either a constant string to define a uniform style property (*e.g.*, `"red"`) for every selected element, or a function to compute a dynamic style per-element (`function(d) { return d.color; }`), selection.data can accept either a constant value or a function.

However, unlike the other selection methods, **selection.data defines data per-group rather than per-element**: data is expressed as an array of values for the group, or a function that returns such an array. Thus, a grouped selection has correspondingly grouped data!



Since there are four groups in the selection, this data function is invoked four times and returns four arrays (one per invocation). Each returned array here also happens to contain four values, but returned arrays may vary in length depending on data.

The blue lines in the diagram indicate that the data function *returns* the linked array. Your data function is passed the datum of the group's `parentNode` (`d`) and the group's index (`i`), and returns whatever array of data you want to join to that group. Thus, data is typically expressed as a function of parent data, facilitating the creation of hierarchical DOM elements from hierarchical data.

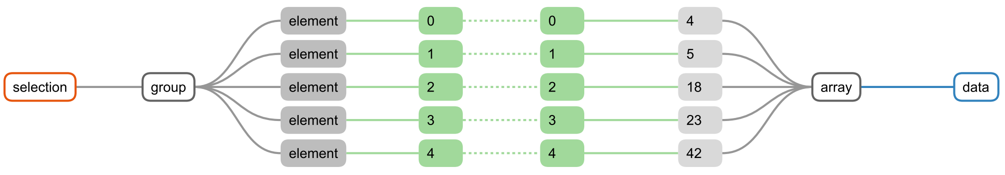
For selections with only a single group, you can pass the corresponding single array to selection.data directly; you only need a function when binding different data to different groups.

The Key to Enlightenment

To join data to elements, we must know which datum should be assigned to which element. This is done by pairing keys. A *key* is simply an identifying string, such as a name; when the key for a datum and an element are equal, the datum is assigned to that element.

The simplest method of assigning keys is by index: the first datum and the first element have the key "0", the second datum and element have the key "1", and so on. Joining an array of numbers to a matching array of paragraph elements therefore looks like this, with keys shown in green:

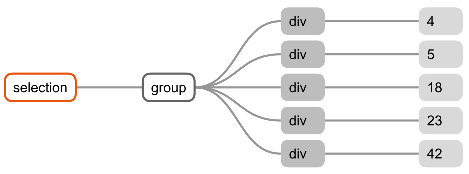
```
var numbers = [4, 5, 18, 23, 42];
```



The resulting selection now has elements bound to data:

```
d3.selectAll("div").data(numbers);
```

Tom MacWright's "fun, difficult" introduction to D3 explains data joins by way of a simple reimplementation.

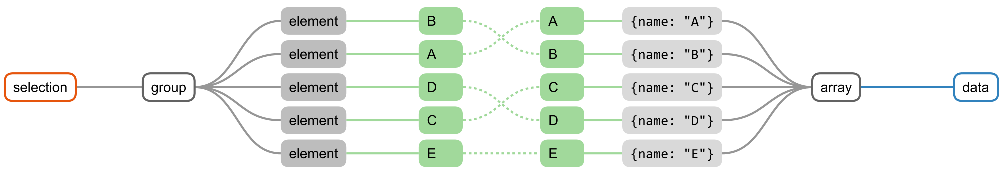


Joining by index is convenient if your data and elements are in the same order. However, when orders differ, joining by index is insufficient! In this case, you can specify a key function as the second argument to selection.data. The key function returns the key for a given datum or element. For example, if your data is an array of objects, each with a name property, your key function can return the associated name:

```
var letters = [
  {name: "A", frequency: .08167},
  {name: "B", frequency: .01492},
  {name: "C", frequency: .02780},
  {name: "D", frequency: .04253},
  {name: "E", frequency: .12702}
];

function name(d) {
  return d.name;
}
```

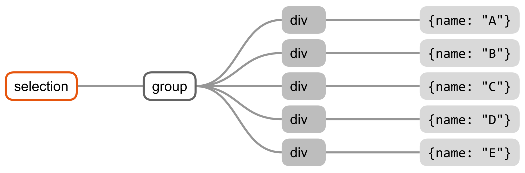
The key function is called for each old element and each new data: ten times in this example. The previously-bound data is used to compute old keys, while the new data is used to compute new keys.



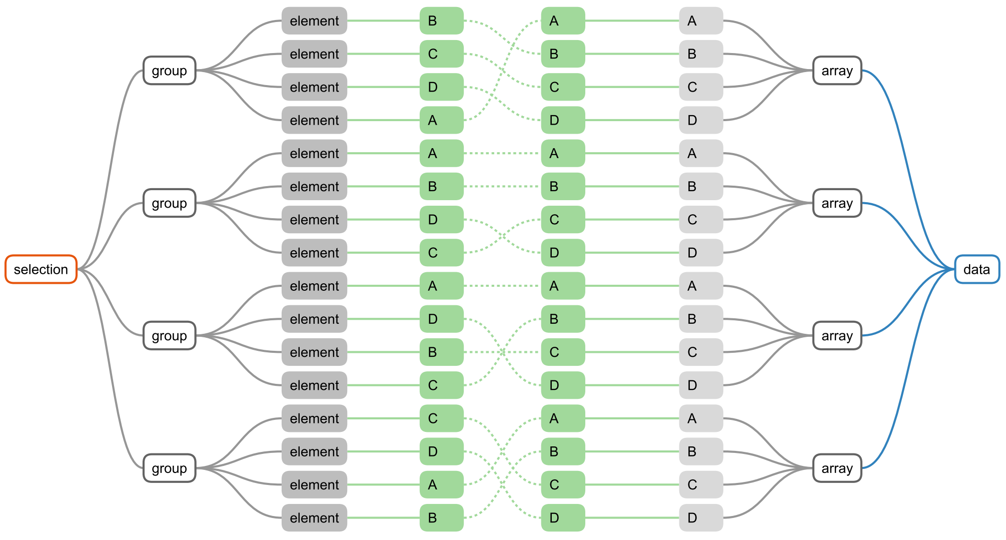
Again, the selected elements are now bound to data. The elements have also been reordered within the selection to match the data:

```
d3.selectAll("div").data(letters, name);
```

Although the selection now matches the data, the elements are *not* automatically reordered in the DOM. For that you must call selection.order or selection.sort.



This process can be quite complicated for large grouped selections, but is simplified somewhat because **each group is joined independently**. Thus, you only need to worry about unique keys within a group, not across the entire selection.



The above examples assume an exact 1:1 match between data and elements. But what happens when there's no matching element for a given datum, or no matching datum for a given element?

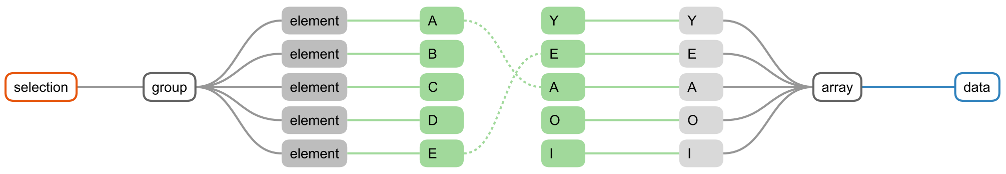
You can read more about key functions in my previous tutorial on object constancy.

Enter, Update and Exit

When joining elements to data by key, there are three possible logical outcomes:

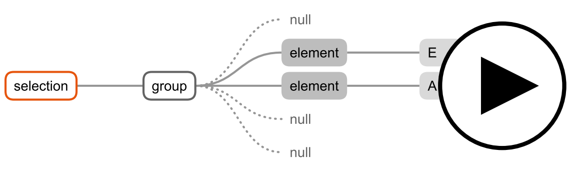
- *Update* - There was a matching element for a given datum.
- *Enter* - There was no matching element for a given datum.
- *Exit* - There was no matching datum for a given element.

These are the three selections returned by selection.data, selection.enter and selection.exit, respectively. To illustrate, imagine you had a bar chart of the first five letters of the alphabet (ABCDE), and you want to transition to your five favorite vowels (YEAOI). You can use a key function to maintain association of letters to bars across the transition, resulting in the following data join:



Two of the previously-displayed letters (A and E) are vowels. These bars are therefore placed in the **update** selection, in order of the new data:

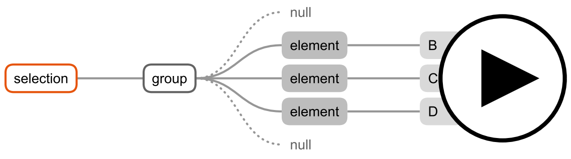
```
var div = d3.selectAll("div").data(vowels, name);
```



The enter and update selections match the order of the new dataset. New data replaces old data in the update selection.

The other three displayed letters (B, C and D) are consonants, and thus have no corresponding data in the new dataset. These elements are therefore placed in the **exit** selection. Note that the exit selection preserves the order of the original selection, which is sometimes useful when animating prior to removal:

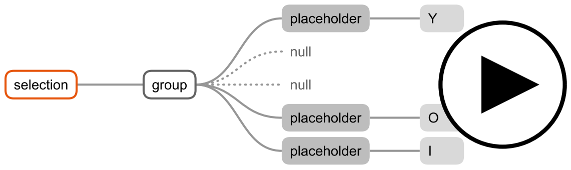
```
div.exit();
```



The exit selection preserves the order, indexes and data of the old selection.

Lastly, three of the vowels (Y, O and I) were not previously displayed, and thus have no corresponding element. These form the **enter** selection:

```
div.enter();
```



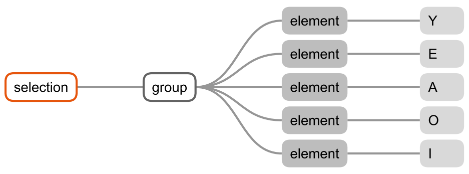
The placeholders in an enter selection are typically transient; the enter selection is replaced with a normal selection of elements when you call enter.append or enter.insert.

While update and exit are normal selections, enter is a subclass of selection. This is necessary because it represents elements that *do not yet exist*. An enter selection contains placeholders rather than DOM elements; these placeholders are simply objects with a `__data__` property. The implementation of enter.select is then specialized such that nodes are inserted into the group's parent, replacing the placeholder. This is why it is critical to call selection.selectAll prior to a data join: it establishes the parent node for entering elements.

Merging Enter & Update

The general update pattern with a data join appends entering elements and removes exiting elements, while modifying dynamic attributes, styles and other properties of updating elements. Often, there's overlap between properties of updating and entering elements.

To reduce duplicate code, enter.append has a convenient side-effect: it replaces null elements in the update selection with the newly-created elements from the enter selection. Thus, after enter.append, the update selection is modified to contain both entering and updating elements. The update selection subsequently contains all currently-displayed elements:



With the selection once again consistent with the document, the life-cycle of the data join is complete.

Acknowledgements

Thank you to Anna Powell-Smith, Scott Murray, Nelson Minar, Tom Carden, Shan Carter, Jason Davies, Tom MacWright and John Firebaugh for reviewing and providing feedback to improve this article.

Further Reading

If you found this article informative, if you found parts unclear or confusing, or if you have followup questions or feedback, please let me know via Twitter or Hacker News. To continue learning about selections, reading D3's source is a rigorous way to test your understanding. And here are several excellent talks and tutorials by others:

- Binding Data by Scott Murray
- Journey to the Source by Anna Powell-Smith
- A Fun, Difficult Introduction to D3 by Tom MacWright

April 26, 2013 / Mike Bostock