d3js v4 discussion and examples - part 1

Steven Brawer
4/2017

introduction

In this document, I try to illustrate how the d3 chain of selections work. The discussion includes the following functions:

selectAll
select
data
enter
exit
remove
append
text
attr
style
filter

d3js can be used in two very different manners. The less useful and less interesting one is to create and style DOM elements in a way similar to (but with a very different syntax from), for example, raw javascript, jquery or React. I will call this mode of working **html-driven**, as it explicitly creates DOM elements with an html-like approach.

However, this is not where the power lies. Rather, d3.js can be used to **create DOM elements based on data**. Generally this is data one wants to visualize, and d3,js has become the de-facto web visualization standard. I will refer to this as the **data-driven** approach. It is very different conceptually from the html-driven approach, though basically it uses the exact same programming syntax and data structures.

I give examples of both the html-driven and data-driven approaches.

d3.js is a complex and sophisticated system. There is a huge complexity involving nested looops, considerable logic branching and functional methodologies, all of which are completely under the covers and which are very difficult to document in words. As part of this discussion, I will present code snippets from the d3.js system. In many cases, a small amount of code can be worth a thousand words.

This is a rather technical document. The essence of the data-driven approach is in the very last section of this document, which is unfortunate. I think this document is likely to be confusing on a first and even second reading, and getting to the end will not be a trivial undertaking.

html-driven approach

I will start off with a simple example and then explain roughly what is happening under the covers. We start off with an explicit html document which is basically

```
<html>
......
        <body>
                <script src="d3..js"></script>
                <script>
                .....
                </script>
        </body>
</html>
```

where d3.js is the downloaded file for v 4.7.4, obtained from https://d3js.org.

Ignoring the *<script>* and *<html>* and other elements, for simplicity we write the initial html document as

```
<body></body>
```

For the first example, we will create, using d3.js, the following DOM:

```
<body>
        <p>Hello, World</p>
</body>
```

LIST 1

Here is the d3 program to create List 1 (this and all programs go in the *<script>* section):

```
let bd1 = d3.select("body");
let app = bd1.append("p");
let h = p.text("Hello, world.");
```

LIST 2

This can be written in the simpler manner:

```
 d3.select("body").append("p").text("Hello, world.");
```

LIST 3

The form List 3 uses **chaining**. The form List 2 is pedagogicaally useful.

select()

The variable *d3* is the top-level object in d3.js, and *select()* is one of its methods. The method is:

```
var select = function(selector) {
  return typeof selector === "string"
    ? new Selection([[document.querySelector(selector)]], [document.documentElement])
    : new Selection([[selector]], root);
};
```

LIST 4

The fundamental data structure for this document is the **Selection** class:

```
function Selection(groups, parents) {
  this._groups = groups;
  this._parents = parents;
}
```

LIST 5

So we see that the *select()* function creates an *Selection* object with the schematic form:

```
bd1 = {
        _groups: [ [.....] ];
        _parents: [ .....];
      }
```

LIST 6

where *bd1* is defined in List 2. Note that List 4 creates *_groups* as an array of just one array. In general, as will be seen, *_groups* will be an array of multiple arrays or array-like objects (for example, *NodeList* objects). Each sub-array may have multiple elements. The arrays could also be empty or array elements may be undefined. *_parents* is a single array generally of *HTMLElement* objects.

In our case, when we log *bd1* to the console, we get:

```
log(bd1):
_groups: [  [body ] ]
_parents: [ html ]
```

LIST 7

In List 7, *body* represents an *HTMLElement*. The expressions

```
bd1._groups[0][0] instanceof  HTMLElement
bd1._groups[0][0] instanceof  Element
bd1._groups[0][0] instanceof  Node
```

are all true (*Node* is the highest level in the hierarchy, *HTMLElement* the lowest). Then *html* is also an *HTMLElement*, the element returned by *document.documentElement* in List 4.

The important point is that *bd1* is a *Selection* object. The *Selection* object also has a number of methods, which are not apparent in List 5. They are assigned via the *prototype* in the following way (here are just a few):

```
Selection.prototype = selection.prototype = {
 constructor: Selection,
 select: selection_select,
 selectAll: selection_selectAll,
 filter: selection_filter,
 data: selection_data,
 enter: selection_enter,
....
 append: selection_append,
....

etc etc
```

LIST 8

The constructor is List 5. We see that, for example, *select* is different from List 4. Here, in the context of a *Selection* object (as opposed to *d3*), it is an alias for the function *selection_select*. So *filter* is an alias for *selection_filter* and so forth.

Each one of the functions List 8 returns a *Selection* object. This is what allows the chaining List 3. For example, in List 2, the variable *bd1* represents a *Selection* object. Then *bd1.append()* invokes the function *selection_append()* (see List 8) on the *Selection* object *bd1*, which returns a *Selection* object, so the variable *app* represents a *Selection* object. Then *app.text()* invokes the function *selection_text(),* returning a *Selection* object, so the variable *h* refers to a *Selection* object, and so forth. During the progression of the chain, the *Selection* object gains and loses properties and the contents of the properties change. Sometimes a new *Selection* object is created and returned by a function.

append()

The next statement in List 2 is

```
let app = bd1.append("p");
```

This actually creates the *p* element. The *append* function (alias for the function *selection_append()*) creates a new element every single element in *_groups[i][j]* for all relevant *i,j* - in this case as a child of the *body* element. The *append()* function ultimately executes

```
let Elem = app._groups[0][0];  // <body>
let child = document.createElement("p");   //<p>
Elem.appendChild(child);
```

In this way, the *p* element becomes a child of *body*. In addition, **the *append()* function creates and returns a new *Selection* object,** where now *_groups* has the *p* element rather than the *body* element. Therefore, after executing the *append()* function, the expression

```
app === bd1
```

is false.

We have

```
log(app):
_groups: [ [ p ] ]
_parents: [ html ]
```

LIST 9

Note that *body* is not explicitly visible at the level of List 9. The *p* element parent attribute is *body*, and the *body* element appears when the *html* element is expanded.

text()

The text() function is an alias for selection_text.

At this stage, we can't see the *p* element in the browser because it doesn't have any *textContent* (you can see it in the web inspector). By executing

```
let h = text("Hello world.")
```

we assign the string *Hello world* to the *textContent* attribute of the *p* element, and the string *Hello world* is now visible in the browser. The *text()* function assigns values to the *textContent* attribute of all elements *_groups[i][j]* for all relevant *i,j*.

After executing the *text()* function, we have:

```
log(h):
_groups: [ [ <p>Hello world</p> ] ]
_parents: [ html ]
```

You can access the contents of *_groups* by doing

```
let groupSub = h._groups[0][0];
```

which returns

```
<p >Hello, world.</p>
```

and you can then style the element, for example

```
groupSub.style.color = "red";
```

which will make the string *Hello world.* in the browser red. But we can also do this directly in d3js, as will be seen below.

Finally, the *text()* function returns the same *Selection* object that is its context, so that

```
h === appp
```

is true.

text() and html() may destroy elements

Consider a small change in List 2 as follows:

```
let bd1 = d3.select("body");
let app = bd1.append("p");
let h = app.text("Hello, world.");
let hbd = bd1.text("body text");
```

LIST 10

We have merely added a text child to *body after* creating the *p* element. It's a little weird, of course, but still legal. What List 10 causes to be printed to the browser is

```
body text
```

There is no *Hello world*. If you examine elements in the browser, you'll find that the *p* element no longer exists (and in fact, neither do the <*script*> elements). The reason is that the statements

```
node.innerHTML = .....
node.textContent = ....
```

remove all descendents of *node*. This is documented behavior of the javascript assignments and is not special to d3js. The *text()* method invokes *element.textContent* on the elements in *_groups* (in this case, the *body* node). There is also an *html()* function which invokes *innerHTML* on the elements in *_groups*.

style and attributes

Very briefly, we can set attributes and styles as in the following example, using List 2 (not List 10).

```
let p=d3.select("body").append("p").text("Hello world");
p.attr("class","myClass").style("color","red").style("border","3px solid green");
```

p._groups[0][0] returns

```
[ [<p class="myClass" style="color: red; border: 3px solid green;">Hello world</p> ] ]
```

nesting-1

With d3js, we can create mnested elements. Consider

```
let bd = d3.select("body");
let dv = bd.append("div");
let p = dv.append("p");
```

LIST 11

This creates the following:

```
<body>
        <div>
                <p>
        <div>
<body>
```

LIST 12

Because *append()* creates new *Selection* objects, the three variables (*bd,dv,p*) in List 11 all point to different *Selection* instances.

problem - logging may be asynchronous

One thing to watch out for is that logging to the console seems to be asynchronous, so that if you log *bd* after the first statement of List 11, for example, the actual output may not happen until after the final statement of List 11, in which case the log happens when *bd* contains the children which are created after the logging statement. The problem is that the log may just store a pointer to the object being logged (which is OK for a string or number) and, since in general HTMLElements are live, the deeper elements may evolve before the actual output occurs. See the following for one discussion:

http://stackoverflow.com/questions/23392111/console-log-async-or-sync

In order to get the logs at the appropriate times, you have to stringify the logs (or save strings/numbers) at the time you want them. However, stringify doesn't automatically stringify deep structures of objects. A simple approach is to put a program, such as List 2 or List 11, in a function, and use a *return* statement right after the log of interest.

nesting - 2

We now briefly consider creating multiple children under a given node using d3js.

Consider the program below.

```
let bd = d3.select("body");
let dv1 = bd.append("div");
let dv2 = bd.append("div");

dv1.text("dv1 text");
dv2.text("dv2.text");
```

LIST 13

Examining the DOM in the developer tools, we see that this creates

```
<body>
        <div>div1 text</div>
        <div>div2 text</div>
</body>
```

and writes

div1 text
div2 text

to the browser window.

Notice the difference between List 11 and List 13. In List 13, *divs* are created at the same level, as children of *body*, while in List 11, the *p* is created as a child of *div* and *div* is a child of *body*. Note in List 13 how *bd* is used twice.

We now add some *p* nodes with text.

```
let bd = d3.select("body");
let dv1 = bd.append("div");
let dv2 = bd.append("div");
dv1.text("dv1 text");
dv2.text("dv2.text");

let p1a = dv1.append("p");
let p1b = dv1.append("p");
let p2a = dv2.append("p");
let p2b = dv2.append("p");
p1a.text("p1a");
p1b.text("p1b");
p2a.text("p2a");
p2b.text("p2b");
```

LIST 14

This creates

```
<body>
	<div>
		"div text"
		<p>p1a</p>
		<p>p1b</p>
	</div>
	<div>
		"div2 text"
		<p>p2a</p>
		<p>p2b</p>
	</div>
</body>
```

And writes the following to the browser window:

div1 text
p1a
p1b
div2 text
p2a
p2b

Note in List 14 that we had to assign the text to the *div* elements prior to appending the *p* elements. If we had appended *p* elements first, then assigning text to *div* would have removed the *p* elements.

data-driven approach

introduction - one data array

Based on the previous discussion, it is clear that the HTML-driven methodologycan create any DOM structure we desire. But what would be the point? There are many other ways of using not dissimilar methodologies, such as raw javascript, React and jquery.

The power of d3js is that one can create elements and nested structures wholesale based on the structure and content of data rather than by specifying each element individually.

Things get a lot trickier now. Here is a very common template for creating HTML elements using data. First I presentthe entire chain of functions, then will break it up to show what is happening.

```
let dataArr = ["a","b","c","d","e"];
let x = d3.select("body").selectAll("foo")
            .data(dataArr).enter().append("p")
              .text(function(d){return d;});
```

LIST 15

This creates 5 p-elements as children of *body* gives each a *textContent* of a,b,... successively, and therefore displays in the browser:

a
b
c
d
e

It creates

```
<body>
        <p>a</p>
        <p>b</p>
        <p>c</p>
        <p>d</p>
        <p>e</p\>
</body>
```

LIST 16

In List 15 we have only one *append("p")* function, yet we get five *p* elements, each with different *textContent*. The reason for this is essentially in the action of the *data()* function.

Here is the first step of List 15, which should be familiar from previous sections.

```
let bd = d3.select("body") :

log(bd):
_groups: [ [ body ] ]
_parents: [ html ]
```

LIST 17

where *body* and *html* represent *HTMLElement* instances.

The second step of List 15 is:

```
let foo = d3.select("body").selectAll("foo");
```

The resulting new Selection object is

```
log(foo):
_groups: [  NodeList[]  ]
_parents: [ body ]
```

LIST 18

In this context, *selectAll* is an alias for *selection_selectAll.* This function has a number of important uses (we will see it in all its glory later), but in the context of List 17 its function is purely technical. That is, there is no significant conceptual meaning here. *selectAll* merely serves to transmute the *Selection* object List 17 to a new *Selection* object List 18. In the present context there is no other effect.

In addition, this is a very common use case. Common enough that semantically it might be more suggestive of functionality to have a new function named, say, *rearrange()*, which takes no arguments, and which acts in the context of List 17 to return a new *Selection* object such as the one below.

```
log(foo):
_groups: [  []  ]
_parents: [ body ]
```

LIST 19

List 19 could replace List 18 in the chain of *Selection* objects of List 15, and the remaining program would function identically. In List 19, *_groups* is an array with one member and that member, when coerced to *Boolean*, is false. This is the case with List 18, where the member is an empty *NodeList*. *_groups* could be anything so long as the aforementioned properties hold. This property is needed when the *data()* function executes, as we shall see in some detail later.

Therefore, List 15 could be rewritten using *rearrange()* instead of *selectAll()* as follows:

```
let x = d3.select("body").rearrange().data(dataArr)........
```

I think this is semantically reasonable. (Anyhow, it is pedagogically useful.) The *rearrange()* function can only follow a *d3.select* function which references a single existing *HTMLElement*. That element will be the parent of objects created later. Here is a program to create the *rearrange()* function, make it an alias of *selection_rearrange()*, and to use it.

```
let selection_rearrange = function(){
        return new this.constructor([ [] ], [ this._groups[0][0] ]);
}
// create rearrange()
d3.selectAll("dummy").__proto__.rearrange = selection_rearrange;

// use rearrange()
let bd = d3.select("body");
let foo = bd.rearrange();
console.log(foo);
..... etc
```

LIST 20

The *log(foo)* from List 20 is exactly List 19. For reference, the *d3.selectAll()* function in List 20 is very different from the *Selection.selection_selectAll* function used in List 15. For reference, it is

```
var selectAll = function(selector) {
  return typeof selector === "string"
     ? new Selection([document.querySelectorAll(selector)], [document.documentElement])
     : new Selection([selector == null ? [] : selector], root);
};
```

B2

where the *Selection* class is List 5.

The rest of List 15 proceeds unchanged. Note that *Selection.selection_selectAll* or *Selection.selection_rearrange* returns a new *Selection*. Therefore

```
bd === foo
```

is false in both cases.

The third step of List 15 gets to the crux of the matter.

```
let dta = foo.data(dataArr);
```

This is where the data-driven part comes in. We will examine the *data()* function later. For now we note that *dta* is a *Selection* object with four properties, each of which is an array.

```
log(dta):
_groups: [[., ., ., ., .]]
_exit: [ [] ]
_enter:[ [ EnterNode, EnterNode, EnterNode, EnterNode, EnterNode] ](__data__=a,b,c,d,e)
_parents: [ body ]
```

LIST 21

In List 21, the notation [., ., ., ., .] means an array of length 5 where each member is *undefined*. The notation [[]] means an array of length 1 with a single member array of length 0.

The *EnterNode* "class" is:

```
function EnterNode(parent, datum) {
  this._next = null;
  this._parent = parent;
  this.__data__ = datum;
  ....
}
```

LIST 22

The first *EnterNode* object of *_enter* in List 21 has *__data__ = "a",* the second has *__data__ = "b"* and so forth. Obviously these values come from the array *dataArr* in List 15. We will see later in detail how this happens. In addition, each *EnterNode* has *_parent* equal to the *body HTMLElement*.

So in this particularly simple example, the data has created *_enter as an* array with a sub-array with 5 *EnterNode* objects, each of which stores a data element from the *data* array. The *data* array is no longer needed after this.

Note that *dta* is a new *Selection* object which has acquired two new properties, *_enter* and *_exit*. We have

```
foo === dta
```

is false.

For the fourth step in List 15 we have the *enter()* function.

```
let ent = dta.enter();
```

```
log(ent):
_groups: [ [ EnterNode, EnterNode, EnterNode, EnterNode, EnterNode] ]
_parents: [ body ]
```

LIST 23

The array of *EnterNode* objects in List 23 is exactly the same as in *_enter* of List 21. The purpose of *enter()* in this case is simply to set *_groups* in the new *Selection* equal to *_enter* from the old *Selection*. The *Selection* returned as *ent* has only the *_groups* and *_parents* properties.

The function *enter()* returns a new *Selection* object, so

```
ent === dta
```

is false. The code for the *enter()* function (alias for *Selection.selection_enter()*) is simple enough and is:

```
var selection_enter = function() {
  return new Selection(this._enter || this._groups.map(sparse), this._parents);
};
```

LIST 24

where "this" is the *Selection* object *dta*, which is the context of the *enter()* function. Only the left side of || is of interest in this document.

In the fifth step of List 15 we create elements in the DOM.

```
let app = ent.append("p");
```

As noted previously, the *append()* function uses the contents of *_groups* to create DOM elements. The important point here (generalized later) is: **append() will create an *HTMLElement* for each EnterNode object in the *_groups* subarray.** In this case, *append()* recognizes *EnterNode* objects, and for each one it creates a *p* element. The parent of the *p* element is the value of the *_parent* property of the *EnterNode*. In addition, the *p* element will have its *__data__* attribute set equal to the *__data__* value of the *EnterNode* object (see List 22 and List 23).

In addition, as previously, *append("p")* further replaces the *EnterNode* objects with the *HTMLElements*, each created from them. Any text and styling further along the chain will affect the HTMLElements in *_groups*.

We find

```
log(app):
groups: [  [<p  __data__= "a"></p>,  <p  __data__ = "b"></p>,  .....]  ]
parents: [        <body>
                         <p __data__="a"></p>
                         <p  __data__ = "b"></p>                    ...
                         ...
                  </body>
         ]
```

The *p* elements do not yet have a *textContent,* so the browser is still blank.

text(function(){.......})

The sixth and final step of List 15 is

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

which causes browser output. The *text()* function iterates through all elements of *_groups[i][j]*, for all relevant *i,j* and for each *p* element, calls the anonymous function which is the argument to *text(),* passing as *d* the value of the __data__ attribute of the element. The value returned by the anonymous function is the value that is assigned to the *textContent* of the *p* element.

The DOM is now given by List 16.

what just happened?

Let's review the previous discussion. We started with the *body* element. Then, from an array of five data elements, we created five *EnterNode* objects in a *Selection*, and these were converted to *p* elements by *append()* and then given *textContents* by *text()*. Schematically

Setup ---------->	data ------------->	enter -------------->	append ---------------->	text
_group	_enter	__group	_group	_group
[[]]	[[EN, EN,...]]	[[EN, EN,...]]	[[p, p,...]]	[[a .....]]

where *EN* is an *EnterNode* object.

The data() and bindInxdex() functions

The *data()* function in List 15 is an alias for the function *Selection.selection_data()*. This method takes one or two arguments. In List 15 we pass one argument - an array of strings. Without going into detail at this point (we will later), the *data()* function in this particular case invokes a function *bindIndex()* for each element of *_groups[0]*. The logic in this function is critical. This function, rewritten somewhat, is shown below. It does not return anything, but modifies some of its arguments.

```
function bindIndex(parent, group, enter, update, exit, data) {
  for (let i = 0; i < data.length; ++i) {
   if (group[i]) {
     group[i].__data__ = data[i];
     update[i] = group[i];
   } else {
     enter[i] = new EnterNode(parent, data[i]);
   }
  }

 // Put any non-null nodes that don't fit into exit.
 for (let i=data.length; i < group.length; ++i) {
   if (group[i]) {
     exit[i] = group[i];
   }
  }
}
```

LIST 25

As seen in List 20, the *data()* function executes in the context of the *Selection* object *foo*, List 19. The arguments to *bindIndex()* are:

```
parent:        foo._parents[0]  = body   (an HTMLElement)
group:         foo._groups[0]  = []  (an array of length 0)
enter:         [.,.,.,.,.]  (new array length 5)
update:        [.,.,.,.,.]  (new array length 5)
exit:          [.,..,.,.,.]  (new array length 5)
data:          dataArr   = ["a","b","c","d","e"]
```

LIST 26

In the first for-loop of List 25, the loop variable *i* takes successive values 0,1,2,3,4. Then *group[i]* is undefined for each *i*. Therefore, at the end of the first for loop, *enter* is an array of 5 *EnterNodes* (see List 22), with *__data__ = "a","b",....* successively and *_parent = body* in each (see List 21).

The second for-loop does not execute at all because *data.length* > *group.length* (5 > 0).

On return from *bindIndex()*, the arguments *group, exit, parent, update* are unchanged (see List 26), while *enter[]* is as in List 21.

Back to the *data()* function. We skip some parts, and the code of interest here is at the end.

```
update = new Selection(update, parents);
 update._enter = enter;
 update._exit = exit;
 return update;
```

LIST 27

The *Selection* object returned from *selection_data()* is given in List 21. The property *_groups* in the new *Selection* is given by the *update* array, as is clear from List 27.

a variation

Referring to List 15, we note that, if *selectAll("foo")* were absent from List 15 (alternatively, the *rearrange()* function, List 20, were absent), then, from List 26, the *group* argument passed to *bindIndex()* would be
[body]. In that case, in *bindIndex(), group[0]* would be defined in the first for-loop, *i* = 0, and therefore *enter[0]* would be undefined while *update[0]* would contain the *body* element. Once this result passes through List 27, the *Selection* object returned from *data()* is

```
log(.):
_groups: [[ ,. ,. ,. ,. body ]]
_exit: [ [.] ]
_enter:[ [ ., EnterNode, EnterNode, EnterNode, EnterNode] ](__data__= b,c,d,e)
_parents: [ html ]
```

When we look at the *selection_data()* function later, the logic behind this result will become clear.

Then the browser would display

b
c
d
e

The logic behind this should now be clear.

another variation

Another variation is to leave off of List 15 both *selectAll("foo")* and also the *enter()* function, so we would have the program

```
let dataArr = ["a","b","c","d","e"];
let x = d3.select("body") .data(dataArr).append("p").text(function(d){return d;});
```

At the end, there is only one *p* element as a child of *body*. The browser only displays *"a"*.

The exit() function

The next example demonstrates the *exit()* function. It also allows us to examine the *bindIndex()* function with different function arguments.

To begin, set up an **explicit html** as follows:

```
<body>
        <p>1</p>
        <p>2</p>
        <p>3</p>
</body>
```

LIST 28

Consider now the following d3 code.

```
let dataArr = ["a","b"];
let ps = d3.selectAll("p");
let dta = ps.data(dataArr);
let ex = dta.exit();
let app = ex.append("div");
let txt = app.text("stuff" );
```

LIST 29

We are now selecting pre-existing *p* elements, and we have *exit()* following the *data()* function rather than *enter()*. Note also that the *dataArr* array has only two elements, while (see below) initially *_groups[0]* has three elements.

The browser displays:

1
2
3
stuff

The new DOM hierarchy is given in List 34 later on.

First step in List 29. The *d3.selectAll("p")* function is given in B2. It should be clear how this leads to

```
log(ps):
_groups: [ NodeList[  <p>1</p> , <p>2</p> , <p>3</p> ] ]
_parents: [ html ]
```

LIST 30

Second step in List 29. The *selection_data()* function invokes *bindIndex()*. The arguments to the *bindIndex()* function are:

```
parent:        ps._parents[0]  = html   (an HTMLElement)
group:         ps._groups[0]  = NodeList[  <p>1</p> , <p>2</p> , <p>3</p> ]  (an array-like
                                                  object of length 3)
enter:         [.,.,]  (new array length 2)
update:        [.,.,]  (new array length 2)
exit:          [.,.,]  (new array length 2)
data:          dataArr (["a","b"] )
```

Note that on return from *bindIndex()*, the *exit* array is of length 3. The result of *selection_data()* (computed by List 27) is

```
log (dta):
_enter: [ ., ., ]
_exit: [., .,  <p>3</p> ]
_groups: [ [<p>1</p>, <p>2</p> ] ]
_parents: [ html ]
```

LIST 31

One very important aspect of List 31 is the following: **The parent of each *p* element is *body***. This happened at the first step, because the *p* elements are pre-existing and children of *body*. The *_parents* property is not used. Note that there are no *EnterNodes* here.

Third step in List 29. Now comes the *exit()* function. Here it is:

```
var selection_exit = function() {
  return new Selection(this._exit || this._groups.map(sparse), this._parents);
};
```

LIST 32

In other words, *exit()* creates a new *Selection* object, *ex* (see below), where *_groups* in *ex* equals *_exit* from *dta*, List 31. So *exit()* is analogous to *enter()*, but it copies *_exit* rather than *_enter* into *_groups*.

```
log(ex):
_groups: [ [ <p>3</p> ] ]
_parents: [ html ]
```

LIST 33

Again, the parent of the *p* element in List 33 is *body*.

Fourth step of List 29. The *append()* step then appends a *div* element under the *p* element in *_groups*. This is weird, but legal. After applying *text()*, we have the following element structure:

```
<body>
        <p>1</p>
        <p>2</p>
        <p>
                "3"
                <div>stuff</div>
        </p>
</body>
```

LIST 34

The two *p* elements, with *textContent* 1 and 2 are still in the DOM hierarchy. They have never been removed. The chain of *d3* statements manipulates *Selection* objects (without modifying the DOM in this example) and in the end adds a *div* element to a *p* element.

To remove the other two *p* elements (if that's what we want to do), we need a *remove()* function. To remove an element, we must remove it from its parent element. One cannot remove an element directly.

The remove() function

Continuing with the previous example, the *remove()* function removes all elements in *_groups*. For each element in *_groups*, it gets the parent, then removes the particular child of that parent. In order to apply this to eliminate the first two *p* elements, it must be applied before the *exit()* function (because *exit()* returns a new *Selection* Object with a completely new *_groups* property, and the two elements of interest are not there).

Therefore, in order to remove the first two *p* elements, List 29 now becomes (written as a single chain)

```
let dataArr = ["a","b"];
let ps = d3.selectAll("p").data(dataArr).remove().exit().append("div").text("stuff" );
```

LIST 35

Note the presence of *remove()*. The *remove()* function does NOT return a new *Selection*. Therefore, the *Selection* after the *remove()* function is unchanged.

```
log(.)
_groups: [ [<p>1</p>, <p>2</p>] ]
_enter: [ [.,.,] ]
_exit: [ [<p>3</p> ] ]
_parents: [ html ]
```

LIST 36

The *remove()* function only removes elements from the DOM. It does not modify the *Selection* object which is its context. So the *p* elements that were removed from the DOM are still in the *Selection* List 36. However, the *exit()* function does remove these elements from *_groups*. After *exit()*, the new *Selection* object is

```
log(.)
_groups: [ [., ., <p>3</p> ] ]
parents: [ html ]
```

At the end of List 35, the browser displays

```
3
stuff
```

where *div* is still a child of *p*.

summary of bindIndex() function List 25.

In the *bindIndex* function, there is an interplay between the *group* and *data* arrays passed in as arguments. (One also has to understand the *selection_data()* function which invokes *bindIndex*, which is covered later.)

Here are the basics:

- ***enter[i] = new EnterNode* object ONLY when *group[i]* is undefined and *i* is between 0 and *data.length-1***. This is probably the most common case, and can lead to a new *HTMLElement* for each *data* array member.
- ***exit[i] = group[i]* only for *i* >= *data.length* and *group[i] defined*.**
- ***update[i]=group[i]* only if *group[i]* is defined and *i* < *data.length*.**

I'm not sure if it is easier to remember the summary or just look at the code List 25 when necessary.

bindIndex() creates a 1-1 relationship between *group* and *data*. (The following references the arguments to *bindIndex()*). That is, actions affecting *group[i]* involve *data[i]* - ie, both arrays are accessed with the same array index. Note that the order of elements of *group* (stemming from *_groups*) is important here. This is one reason why the function is called *bindIndex* - it involves indexes. (There is a way of changing this relationship, involving a function called *bindKey*. This is not covered in this document.)

bindIndex() is said to **bind** data to elements. This terminology (I think) references the fact that the *__data__* property of an element is assigned its value from (or, in general, based on) the appropriate *data* array member. So the data is **bound to** the element through the *__data__* property of the element.

two data arrays

Now we get a bit more complicated. We consider two data arrays. The full program is:

```
let dataArr1 = ["a","b","c","d","e"];
let dataArr2 = [6,7,8,9,10,11];

let ent= d3.select("body").selectAll("foo").data(dataArr1).enter();
let txt = ent.data(dataArr2).enter().append("p").text(function(d){return d;});
```

LIST 37

This prints the following in the browser:

11

How can this come about? We add data and print less. Well, this is sort of complicated.

As discussed in the previous section, the *Selection* object *ent* (defined in List 37 above) is the same as List 23, repeated below.

```
log(ent):
_groups: [ [ EnterNode, EnterNode, EnterNode, EnterNode, EnterNode] ]
_parents: [ body ]
```

LIST 38

where the *EnterNodes* have a,b,... successively as the values of the *__data__* property. Continuing with List 37,

```
let dta = ent.data(dataArr2);
```

Once again the *bindIndex()* function List 25 is invoked. This time, the data argument is *dataArr2*, length 6. The *group* argument to *bindIndex* is the subarray of *_groups,* List 38, an array with 5 *EnterNodes*. The *update, enter, exit* arrays are empty upon entrance to this function.

In the first for-loop, the variable *i* = 0,1,2,3,4,5. For the first 5 values of these, *group[i]* is an existing *EnterNode*, hence the if-statement is true. Besides transferring the value of *__data__*, we have *update[i] = group[i], i = 0,1,2,3,4*. On th*e other hand, for i = 5, group[i] is undefined, and therefore enter[5] is a new EnterNode with* __data__ = 11. This is where the 11 comes from that is printed in the browser.

The second for-loop of List 25 does not execute because *data.length* > *group.length* (6 vs 5).

On return from *bindIndex()*, then, the *update* and *enter* arrays are not empty, the *exit* array is empty, the *group* and *data* arrays are unchanged.

As before, still in the *data()* function, after *bindIndex()*, the code List 27 is invoked. This returns the following *Selection* object:

```
log(dta):
_groups: [ [EnterNode, EnterNode, EnterNode, EnterNode, EnterNode] ] (data 6,7,8,9,10)
_exit: [ [., ., ., ., .] ]   (empty)
_enter: [ [., ., ., ., ., EnterNode(__data__ = 11) ]
_parents: [ body ]
```

LIST 39

where the dots indicate undefined array elements.

The next step in List 37 is

```
let ent1 = dta.enter();
```

the *enter()* function List 24 creates a new Selection in which *_groups* equals what was in *_enter*. Therefore, after the *enter()* function List 24 we have the *Selection* object:

```
log(ent1):
_groups: [ [., ., ., ., ., EnterNode(__data__ = 11, _parent=body)] ]
_parents: [ body ]
```

The next step in List 37 is

```
let app = ent1.append("p");
```

This creates a new *Selection* object, in which each *EnterNode* in *_groups* is converted to a *p HTMLElement*. The *p* element is created as a child of *body*. The *Selection* object referenced by *app* is:

```
log(app):
_groups: [ [., ., ., ., ., p]
_parents: [ body ]
```

where as usual *p* represents an *HTMLElement* object. This element is given a *textContent* value by the *text()* function. Note that the *p* element has *__data__* value 11, and this is why only 11 is output to the browser.

a variation

Suppose instead of List 37 we had the slightly different second line

```
let dataArr1 = ["a","b","c","d","e"];
let dataArr2 = [6,7,8,9,10,11];

let ent= d3.select("body").selectAll("foo").data(dataArr1).enter();
let txt = ent.data(dataArr2).append("p").text(function(d){return d;});
```

LIST 40

Note that in the second line *we removed the enter() statement.* That statement copied the single *EnterNode* from *_enter* into *_groups*. In this case, however, the copy is not done, and what is printed in the browser is the contents of *_groups*, given in List 39, so the browser now displays

6
7
8
9
10

grouping and nesting

Now we get much more complicated. Grouping is crucial to many data applications. In html, one usually groups based on *div* - the example shown here. In SVG applications, one creates <*g*> elements for grouping. This section illustrates the basics of data-driven grouping.

We are going to use one data array to create 5 *div* elements as children of *body*, and then another data array to create, for EACH *div*, 3 child *p* elements. So, two data arrays in all, one going after the other, but behavior very different from the previous section. The end result is the following DOM hierarchy

```
<body>
        <div>
                <p>1</p>
                <p>2</p>
                <p>3</p>
        </div>
        <div>
                <p>1</p>
                <p>2</p>
                <p>3</p>
        </div>
        ......  etc etc for 3 more divs
</body>
```

LIST 41

The following is displayed in the browser:

1
2
3
1
2
3
1
2
3
.... etc 5 times

for 5 groups of 1 2 3.

We start off with only

```
<body></body>
```

as the explicit HTML. Here is the deceptively simple program.

```
let dataArr = ["a","b","c","d","e"];
let moreData = [1,2,3];

let bd = d3.select("body").selectAll("foo").data(dataArr).enter().append("div");
let top = bd.selectAll("bar");
let dta = top.data(moreData);
let ent = dta.enter();
let app = ent.append("p")
let txt = app.text(function(d,i){return d;} );
```

LIST 42

We now walk through List 42.

The *Selection* object *bd* in List 42 should at this point be well-understood. It is

```
log(bd):
_groups: [ [div div div div div ];
_parents: [ body ]
```

LIST 43

Each *div* is an HTMLElement, and the *divs* have *__data__* = a, b, c... successively. Each *div* has as its *parent* attribute the *body* element. We see that *_groups[0].length* = 1.

The next statement

```
let top = bd.selectAll("bar")
```

looks familiar but, in the context of List 43, is not. The *selectAll* function is an alias for the function *selection_select("bar")*. The *Selection* object which is the context of *selection_selectAll("bar")* is List 43, very different from the one forming the context of *selection_selectAll("foo")* in List 42. Now the details of the *selection_selectAll()* function are very important - in this context, we **cannot** replace this function with *rearrange(),* List 20.

The *selection_selectAll()* function, somewhat rewritten, is given below.

```
var selection_selectAll = function(select) {
  if (typeof select !== "function") select = selectorAll(select);

 let groups = this._groups;
 let subgroups = [],  parents = [];

  for (let j = 0; j < groups.length; ++j) {
    let subgroup = groups[j];
    for (let i = 0; i < subgroup.length; ++i) {
        if (subgroup[i]) {
          subgroups.push(select.call(subgroup[i], subgroup[i].__data__, i, subgroup));
          parents.push(subgroup[i]);
       }
     }
  }

  return new Selection(subgroups, parents);
};
```

LIST 44

On entrance to this function, *select = "bar"*. After the first statement, the result of the *selector()* function, we have:

```
select = function(){
          return this.querySelectorAll("bar");
}
```

LIST 45

The value of *this* will be determined when this function is called. The context of the *selection_selectAll()* method is the *Selection* object List 43. Therefore, in List 44,

```
this._groups = [ [div div div div div ] ];
```

In the first for-loop, *j* = 0 is the only value of *j*, since *groups.length = 1*. In the second for-loop, we see from the above that *groups[0].length = 5*. The variable *subgroup* is

```
subgroup = [div div div div div ];
```

Now consider the statement

```
select.call(subgroup[i], subgroup[i].__data__, i, subgroup))
```

which appears in the ith iteration of the second for-loop of *selection_selectAll(),* List 44. The variable *subgroup[i]* is a *div* node. The function List 45 is invoked, with *this* equal to *subgroup[i]*. The other variables in the call argument list are not used, so the function invocation is

```
<div>.querySeletorAll("bar");
```

where *<div>* is the *HTMLElement subgroup[i].*

The *select.call(...)* expression returns an empty *NodeList*, since the <*div*> element does not have any children. The variable *top*, List 42, references what is returned from *selection_selectAll()* and is

```
log(top):
_groups: [ NodeList[], NodeList[], NodeList[],. NodeList[], NodeList[] ]
_parents: [ div div div div div ]
```

LIST 46

_groups is length 5. (Each *NodeList* is empty, so coercing *_groups[i]* to *Boolean* gives false). This is the first time we have encountered a situation where _groups[0].length > 1. (Contrast it with List 43.) *This structure is the basis of grouping*.

We now go to the *data()* function, from List 42.

```
let dta = top.data(moreData);
```

The *data()* function is an alias for the *selection_data()* function. Here is the relevant part of the *selection_data* function, rewritten and abbreviated. The actual function is more complicated, but this is sufficient for our purposes.

```
.var selection_data = function(data) {
 let groups = this._groups;
 let parents = this._parents;
 let enter = new Array(groups.length),
     exit = new Array(groups.length),
     update = new Array(groups.length);
 for (let j = 0; j < groups.length; ++j) {
      let enter[j] = new Array(data.length),
      let update[j] = new Array(data.length),
      let exit[j] = new Array(groups[j].length);

      bindIndex(parents[j], groups[j], enter[j], update[j], exit[j], data);
  }
 update = new Selection(update, parents);
 update._enter = enter;
 update._exit = exit;
 return update;
};
```

LIST 47

This creates new arrays *enter*, *exit*, *update*, whose structures are based on the *_groups* structure and the *moreData* array. Initially, all array elements are *undefined*. The *enter* array is of length 5, and each element is an array of length 3 (*data.length*). The *update* array is similar. For each *j*, the variable *groups[j]* is an empty *NodeList*, which has length 0. Thus for each *j*, *exit[j]* is an array of length 0. So for given *j*, on entry to *bindKey,* List 25, we have the arguments:

parent: div (an HTMLElement)
group: NodeList[], empty NodeList
enter: empty array of length 3
update: empty array of length 3
exit: empty array of length 0.
data: [1,2,3]

In List 25, since *group.length* is 0, the three iterations of the first for-loop create three *EnterNode* objects as the three members of the *enter* array. That's all that happens. Very simple!

Returning to *selection_data*, List 47, at the end of the for-loop the *enter* array is:

```
enter = [ [EnterNode(1,div("a")), EnterNode(2,div("a"),EnterNode(3,div("a") ] ,
          [EnterNode(1,div("b"), EnterNode(2,div("b")),EnterNode(3,div("b")) ],
           [...], [...],[...] ]
```

LIST 48

That is, *enter* is an array of length 5, each element being an array of 3 *EnterNode* objects. Each *EnterNode* has *_parent* = the appropriate *div*, and *__data__* equal to 1,2,3 successively.

The grouping structure is exactly here. **The grouping structure List 48 mimics the desired DOM structure List 41.** The remainder of the program List 42 implements this structure in the DOM.

So we see that the two data arrays produce a grouping structure which mimics the DOM hierarchy we wish to create. This is the essence of the data-driven approach.

The *update* array has only undefined array elements, so the *Selection* object returned from *selection_data* is:

```
log(dta):
_groups: [ [., ., .], [., ., .],........]
_enter: [ [EnterNode(1,div("a")), EnterNode(2,div("a"),EnterNode(3,div("a") ] ,
          [EnterNode(1,div("b"), EnterNode(2,div("b")),EnterNode(3,div("b")) ],
           [...], [...],[...] ]
_exit: [ [], [], [], [], [] ]
_parents: [div div div div div]
```

LIST 49

Finally we want to turn this into DOM elements. The next step in List 42 is

```
let ent = dta.enter();
```

The ent Selection object has *_groups* given by *_enter* of List 49 in the usual way.

Then

```
let app = ent.append("p")
```

will create one *p* element for each *EnterNode* object in *_groups*, and the *text()* function then creates a *textContent* for each *p*.