Week 7

JOINING + TRANSITIONS

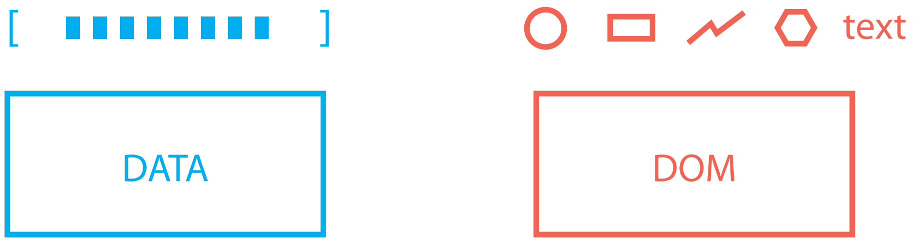
Review of Concepts: DOM and Selection

Where Are We Now?

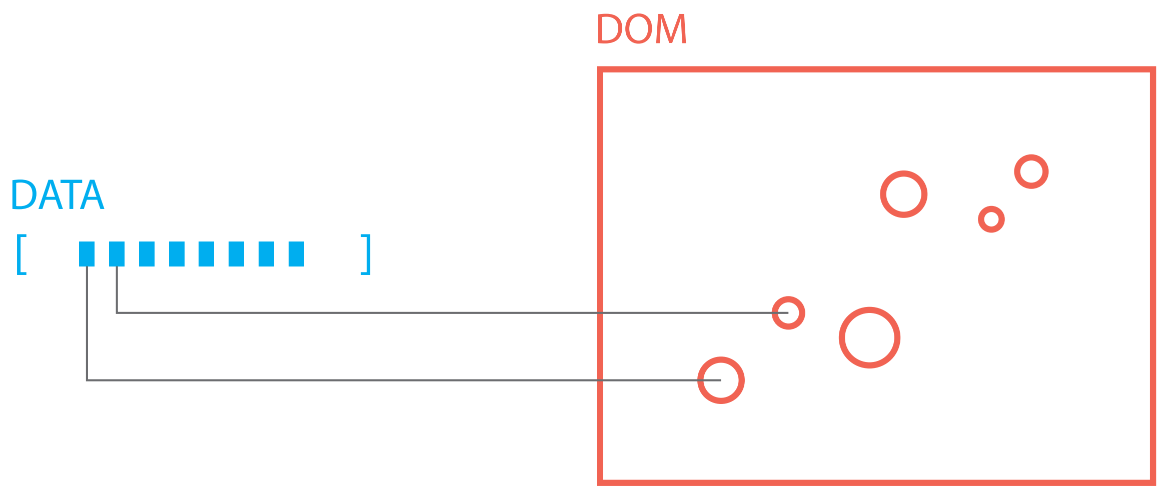
	Week 5	Week 6	This Week
Concepts	• d3 selections; • Using `.append()` to add elements; • Using `.attr()` to set attributes	• Understanding CSV as a data format; • Importing and parsing data; • Why and how of mining data; • Scales, domain, range • Joining data to DOM elements using `selection.`	A systematic overview of how joining works
Implementation		• `d3.csv()` • `d3.max(), d3.min()` • `d3.scale.linear()` • `d3.svg.axis()`	

What is “Join”, and Why?

Essentially, to visualize data is to express data attributes with visual properties



True for scatterplot...



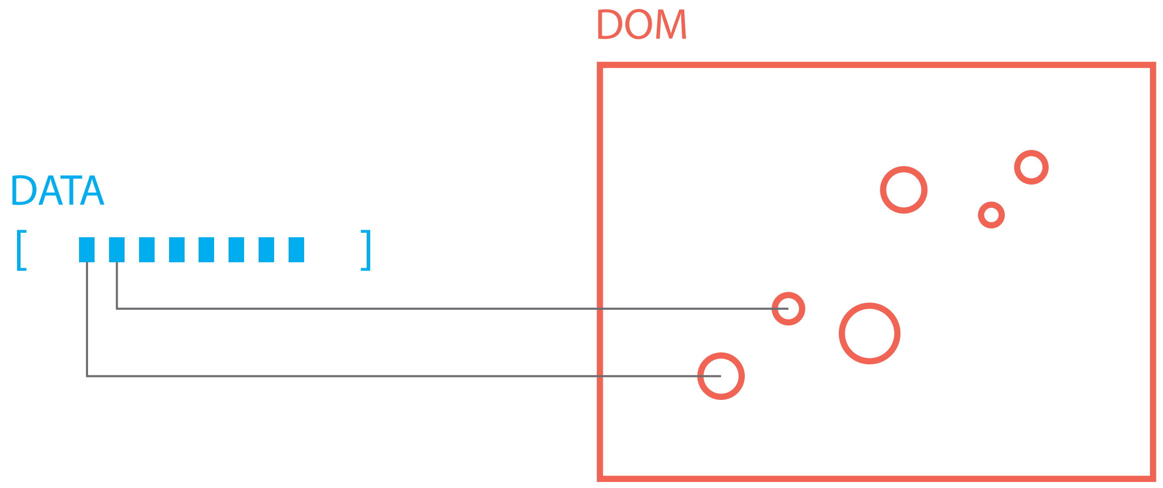
<circle>

x-y position

size

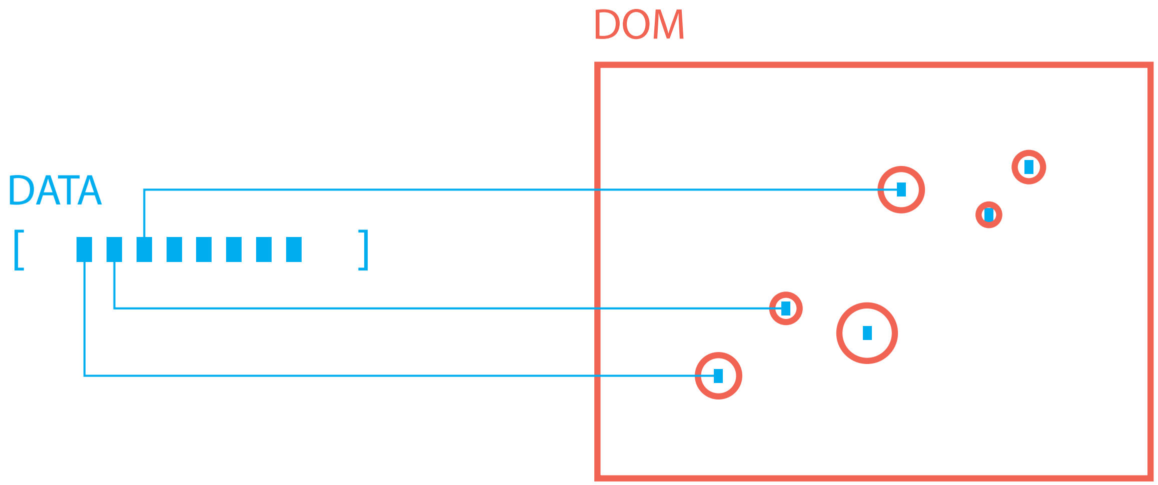
color

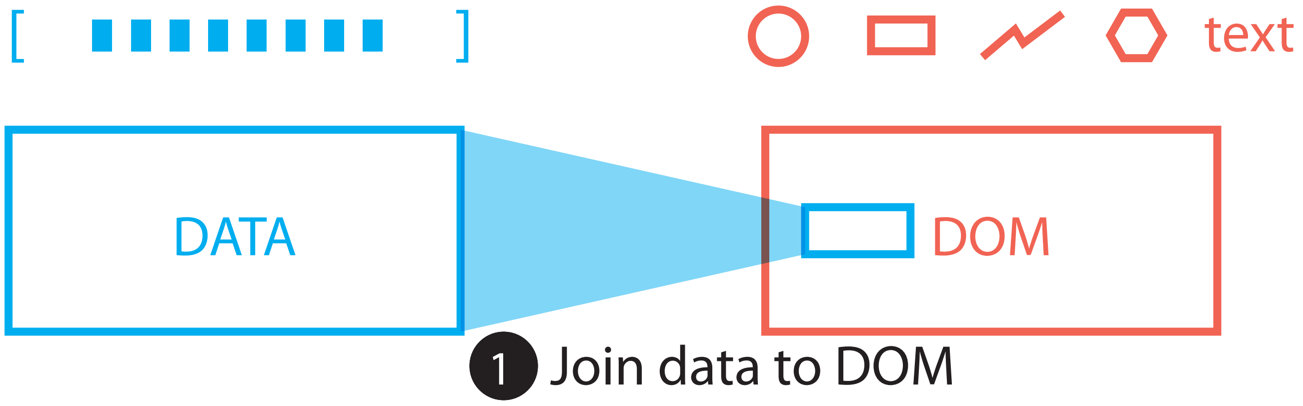
...also true for choropleth

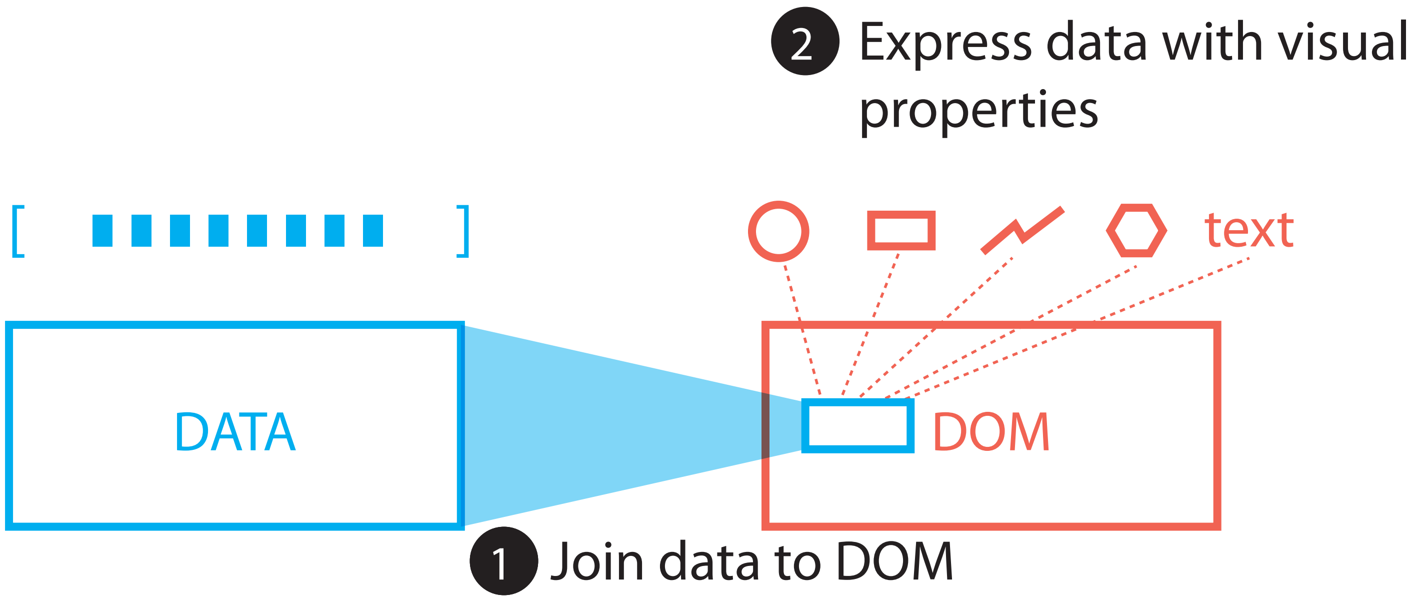


<path>
position
outline shape
color

“Joining”: Binding Data to DOM




[ ]
text
DATA
DOM
1 Join data to DOM


2 Express data with visual properties
text
DATA
DOM
1 Join data to DOM

How d3 Implements Joining

This "pattern" of code usage allows us to join data with DOM

`.selectAll() - .data() - .enter() - .append()`

```
    var dataArray = [23,22,1,0,...];
    var circles = svg.selectAll('circle')
        .data(dataArray)
        .enter()
        .append('circle')
        .attr(...)
        ...
```

JOINING, REVISTED

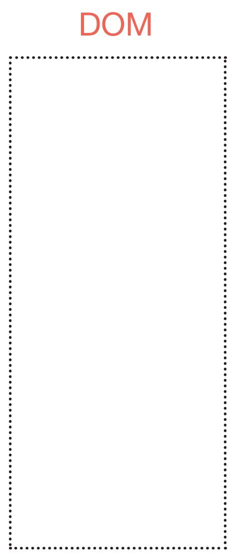
The `.data()` call computes a **join** between DOM elements and data elements. That is to say, it forces a one-to-one match between DOM and data.

```
var dataArray = [23,22,1,0,...];
var circles = svg.selectAll(‘circle’)
    .data(dataArray)
    .enter()
    .append(‘circle’)
    .attr(...)
    ...
```

How This Works, Visually

`.selectAll() - .data() - .enter() - .append()`

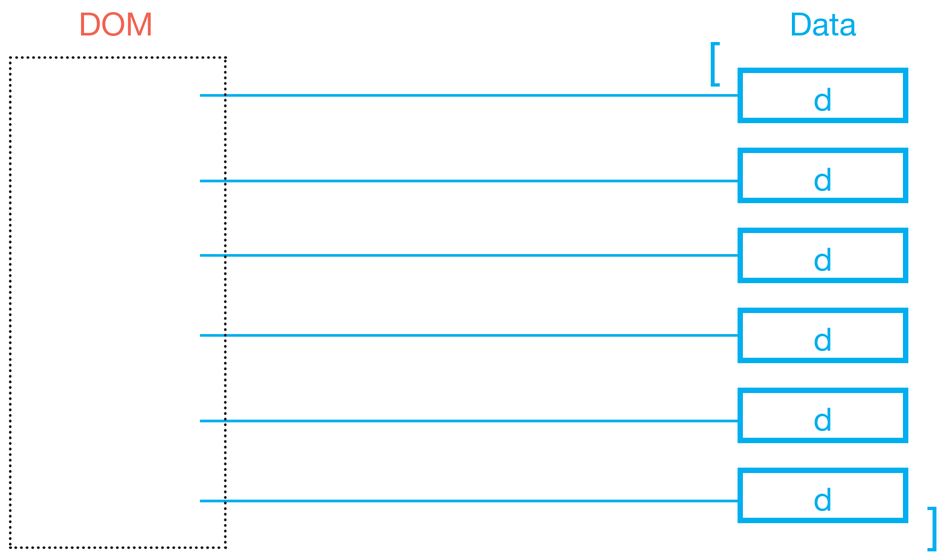
This tries to select all DOM elements that fit the criteria. Often, this results in an empty selection.



How This Works, Visually

```
.selectAll() - .data() - .enter() - .append()
```

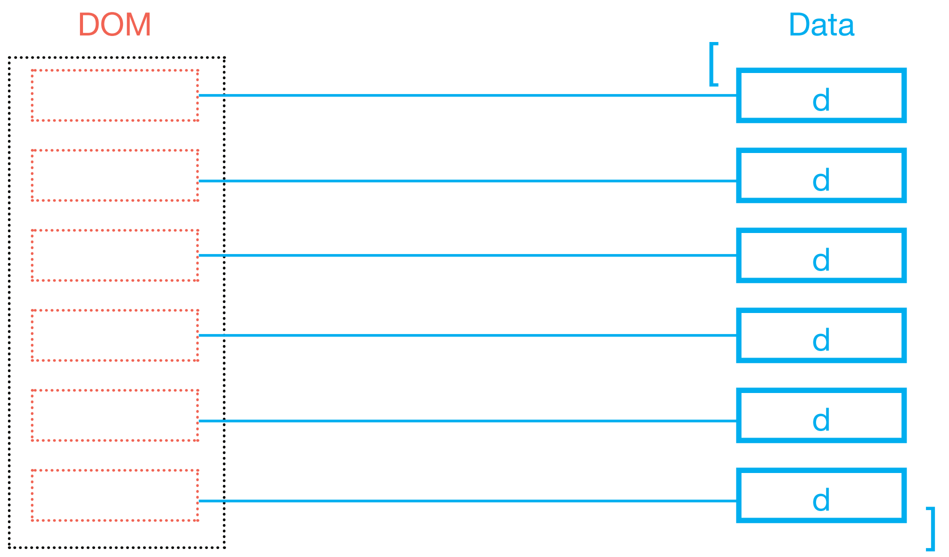
This tries to bind each element in the data array to each DOM element in the selection



How This Works, Visually

`.selectAll() - .data() - .enter() - .append()`

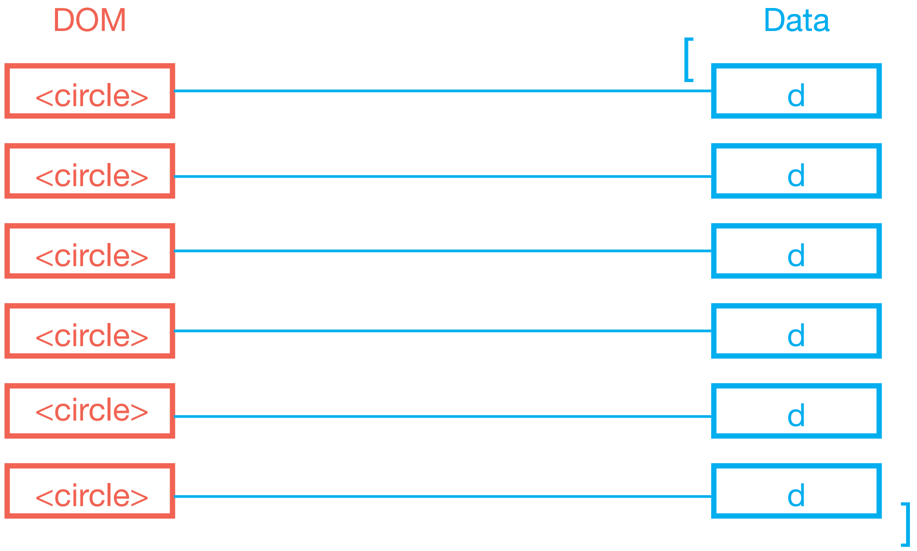
For each mismatch, create an empty placeholder



How This Works, Visually

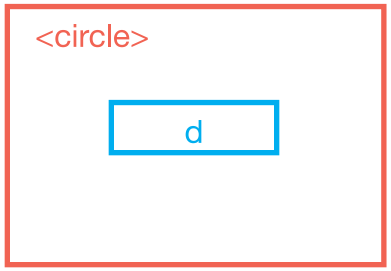
`.selectAll() - .data() - .enter() - .append()`

For each empty placeholder, append some DOM element (could be anything!)



After the Join...

After a DOM element has been joined to a data element, the relationship becomes **one-to-one**. You can imagine that the DOM element in a way contains the data element (and you can actually check this via console).



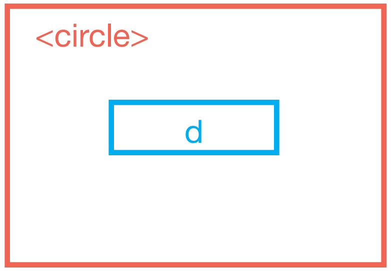
```
var circles =
svg.selectAll(‘circle’)
    .data(dataArray)
    .enter()
    .append(‘circle’)
    ...
    .attr(‘r’,function(d,i){
        return d;
    });
```

After the Join...Using "Accessor" Functions to Access the Data

Accessor functions allow you to access the data element bound to individual DOM elements from the selection. They are usually of the form:

```
...
    .attr('r', function(d,i){...};)
...
```

where argument `d` represents the data element (either a value or an object), and `i` represents the index of the DOM element within the overall selection.



ACCESSOR FUNCTIONS

Let’s look at this example:

```
var data = [{x:10,y:20},{x:40,y:50}];
var circles = svg.selectAll(‘circle’)
    .data(data)
    .enter()
    .append(‘circle’)
    .attr(‘cx’,function(d){ return d.x;})
    .attr(‘cy’,function(d){ return d.y;})
    .attr(‘r’,20);
```

ACCESSOR FUNCTIONS

Let’s look at this example:

```
var data = [{x:10,y:20},{x:40,y:50}];
var circles = svg.selectAll(‘circle’)
    .data(data)
    .enter()
    .append(‘circle’)
    .attr(‘cx’,function(d){ return d.x;})
    .attr(‘cy’,function(d){ return d.y;})
    .attr(‘r’,20);
var anotherCircle = svg.append(‘circle’)
    .attr(‘cx’,function(d){ return d.x; })
    ... //would this work?
```

PRACTICE JOINING

Let’s revisit the scatterplot and practice joining.

Aside: DOM Interaction

One way we can check to see the one-to-one relationship between data and DOM elements is to use interaction.

d3 implements interaction like so:

```
selection.on(eventType, callback);
```

Where the callback is a function that has argument `d` that represents the data bound to the selection.

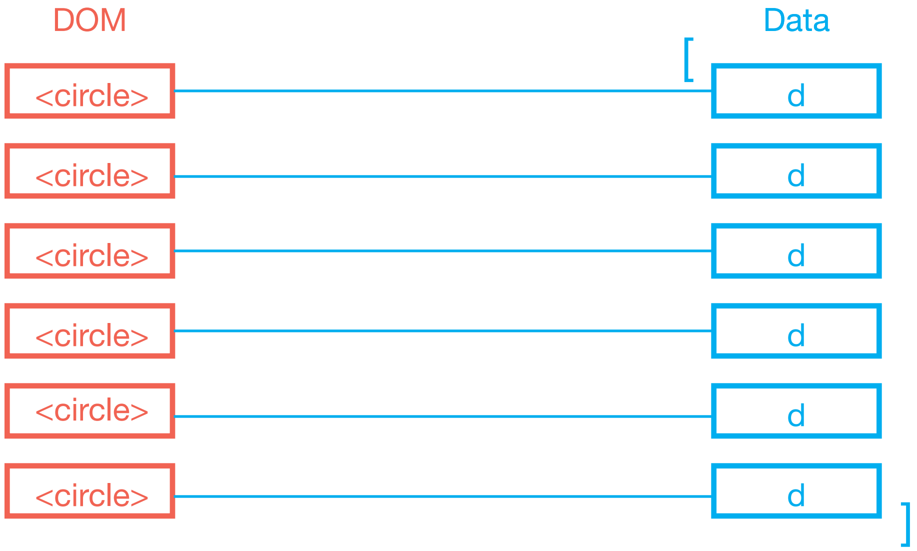
Aside: DOM Interaction

What does this block do?

```
selection.on(‘click’, function(d,i){
    console.log(d);
});
```

Enter, Exit, Update

Joining data to DOM using `.data()` makes no assumption about the number of either.



Enter, Exit, Update

Theoretically, a `.data()` call can result in three situations.

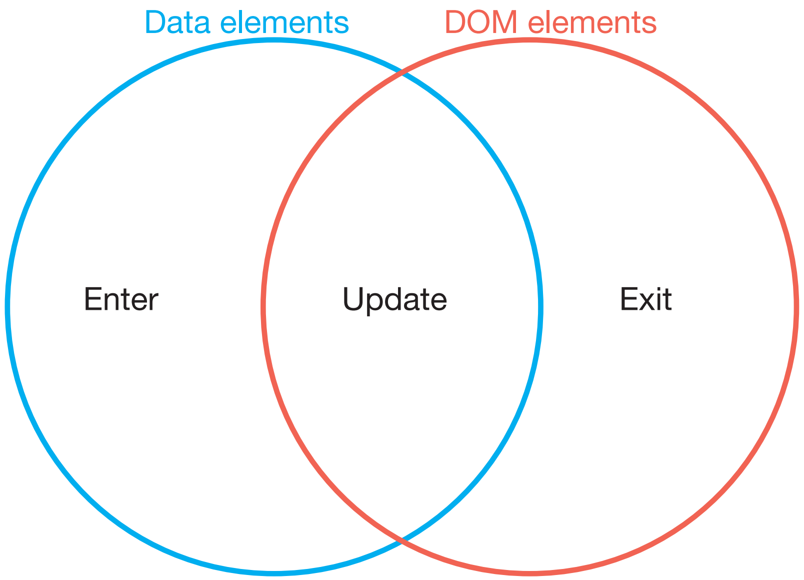
# of DOM elements < # of data elements
We need to create additional DOM elements using .enter()

# of DOM = # of data elements
No addition or removal; we simply re-join the data

# of DOM > # of data elements
Too many DOM elements; we need to remove some

Enter, Exit, Update

A join using `.data()` actually produces **three different selections**: the **enter** selection, the **exit** selection, and the **update** selection.



Enter, Exit, Update

The `.data()` call returns the update selection; the enter and exit selections “hang off” of the update selection, and can be accessed via the `.enter()` and `.exit()` call.

```
var dataArray = [23,22,1,0,...];
var circles = svg.selectAll(‘circle’)
   .data(dataArray); //UPDATE
var circlesEnter = circles.enter() //ENTER
   .append(‘circle’)...
var circlesExit = circles.exit().remove(); //EXIT
```

Enter, Exit, Update

This paradigm is general and flexible. Every time this block of code is run, we re-establish the one-to-one correspondence between data and DOM elements--hugely useful for dynamic visualizations where data changes all the time!

```
var dataArray = [23,22,1,0,...];
var circles = svg.selectAll(‘circle’)
   .data(dataArray);
var circlesEnter = circles.enter()
   .append(‘circle’)...
var circlesExit = circles.exit().remove();
```

Enter, Exit, Update

It’s also great if we want to target specific operations (like .attr() and .style() to specific states):

```
var dataArray = [23,22,1,0,...];
var circles = svg.selectAll(‘circle’)
    .data(dataArray);
var circlesEnter = circles.enter()
    .append(‘circle’)
    .attr(‘r’,0)
    .transition().attr(‘r’,function(d){return 
d;});
var circlesExit = circles.exit().remove();
```

Enter, Exit, Update

This block of code also showcases two other important concepts: `.transition()` and **accessor functions**.

```
...
var circlesEnter = circles.enter()
    .append(‘circle’)
    .attr(‘r’,0)
    .transition().attr(‘r’,function(d){return
d;});
...
```

PRACTICE ENTER / EXIT / UPDATE

Let's open exercise 7.0.W

OBJECT CONSTANCY

Transitions can support the practical implementation of an important principle of visualization design: **object constancy**.

Object constancy is the idea that visual elements should consistently represent a data point.

Object constancy makes it easier to follow changes in data: instead of scanning through the whole visualization every time, our eyes can detect movement of existing elements much more efficiently.

Let’s look at some examples.

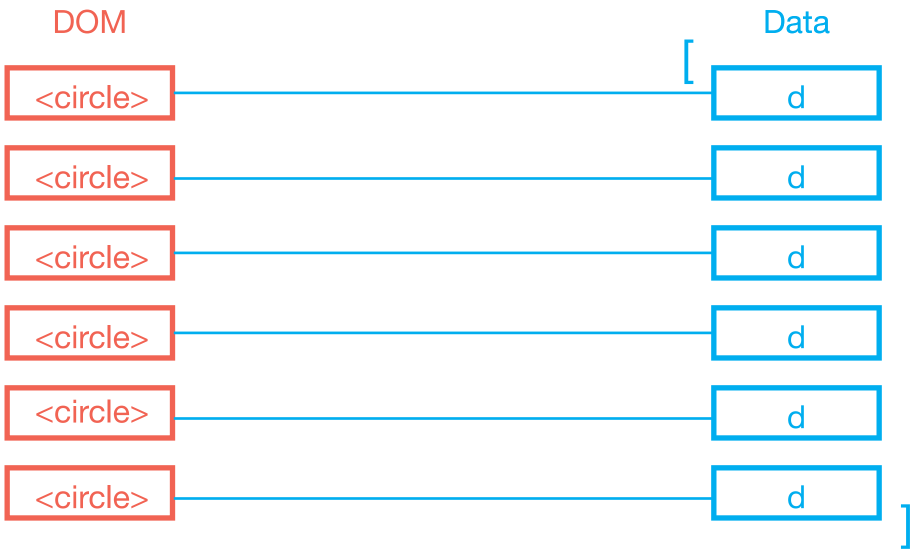
OBJECT CONSTANCY

D3 provides an easy way to maintain object constancy. When binding data to DOM elements, we can specify a **key function** as a second argument, which looks like this:

```
var countries = svg.selectAll(‘.country’)
    .data(countryData, function(d){
        return d.countryName;
    })
    .enter()
    .append(‘g’)
    .attr(‘class’, ‘country)
    ...
```

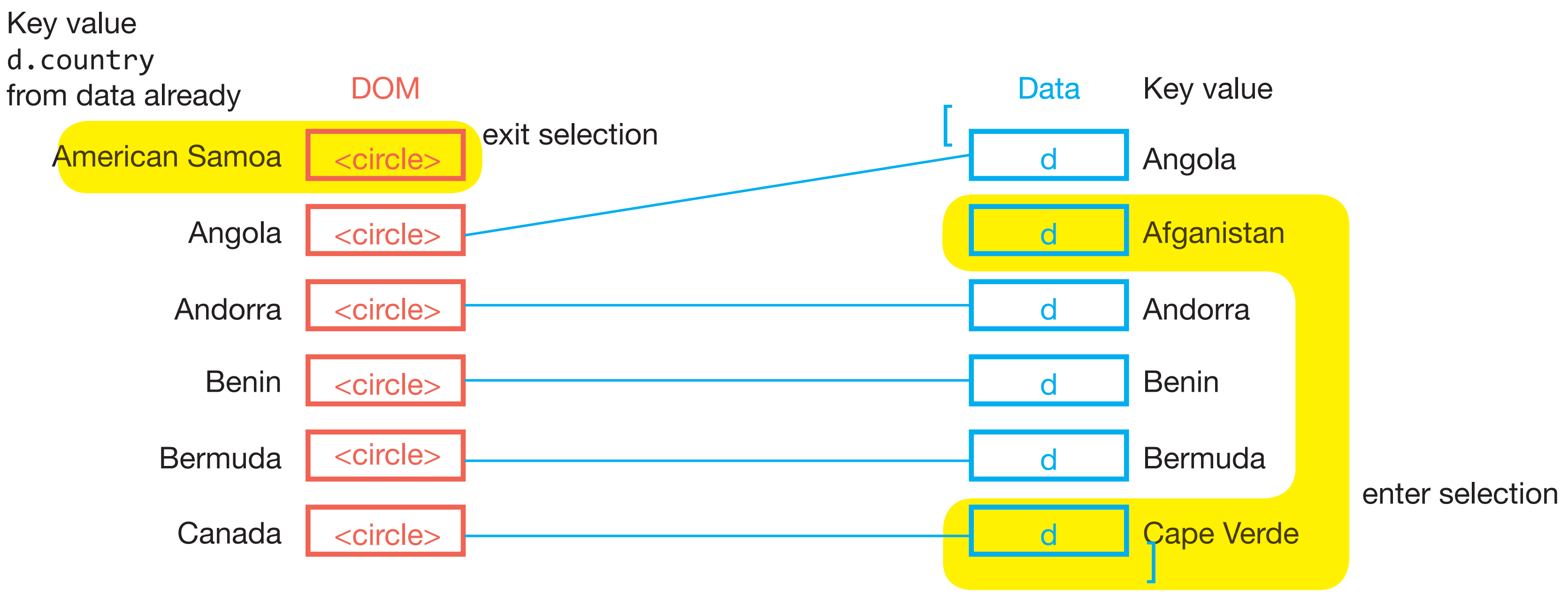
OBJECT CONSTANCY

Without a key function, the join operation via `.data()` tries to match DOM with data elements one for one, with the matching order being purely indexical:

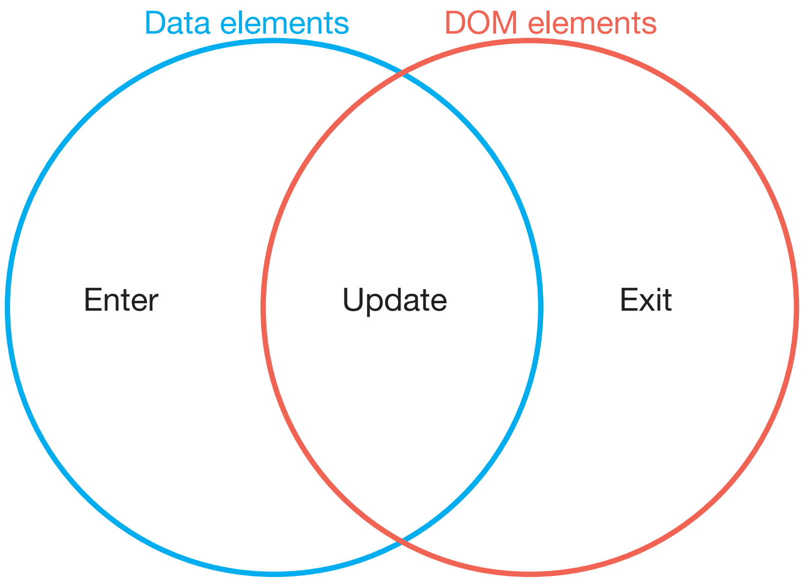


OBJECT CONSTANCY

With a key function, data elements are matched to DOM elements based on a key value.



...AND THIS IS THE FULL ENTER/EXIT/UPDATE PATTERN



With key functions and object constancy, it’s entirely possible to have an enter, exit, and update set all at the same time.

A General Paradigm for Dealing with All Three

```
//update set
var update = selection.data( someArr );

//enter set
var enterSet = update.enter()...

//exit set
var exitSet = update.exit()...

//update + enter
update
    .merge(enterSet)
```

PRACTICE OBJECT CONSTANCY

Let’s continue with 7.0 to practice object constancy.

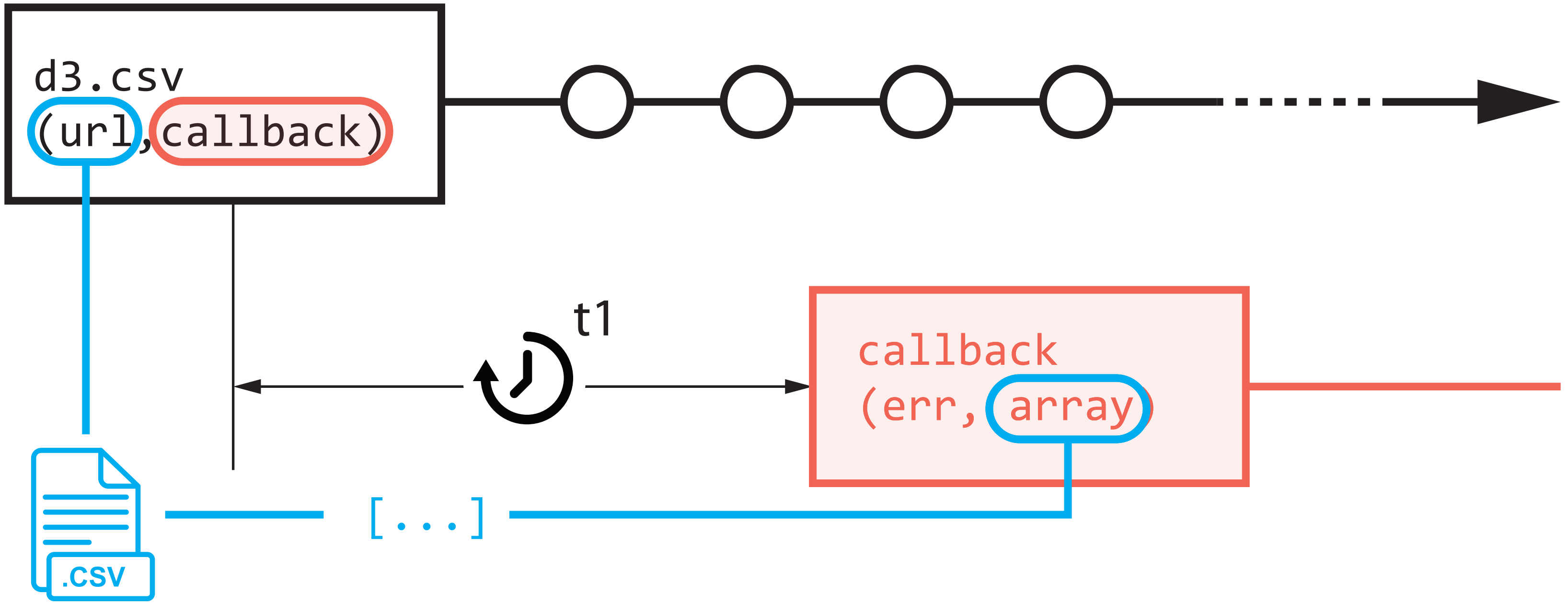
Working on a Practical Example for the Enter/Exit/ Update Pattern

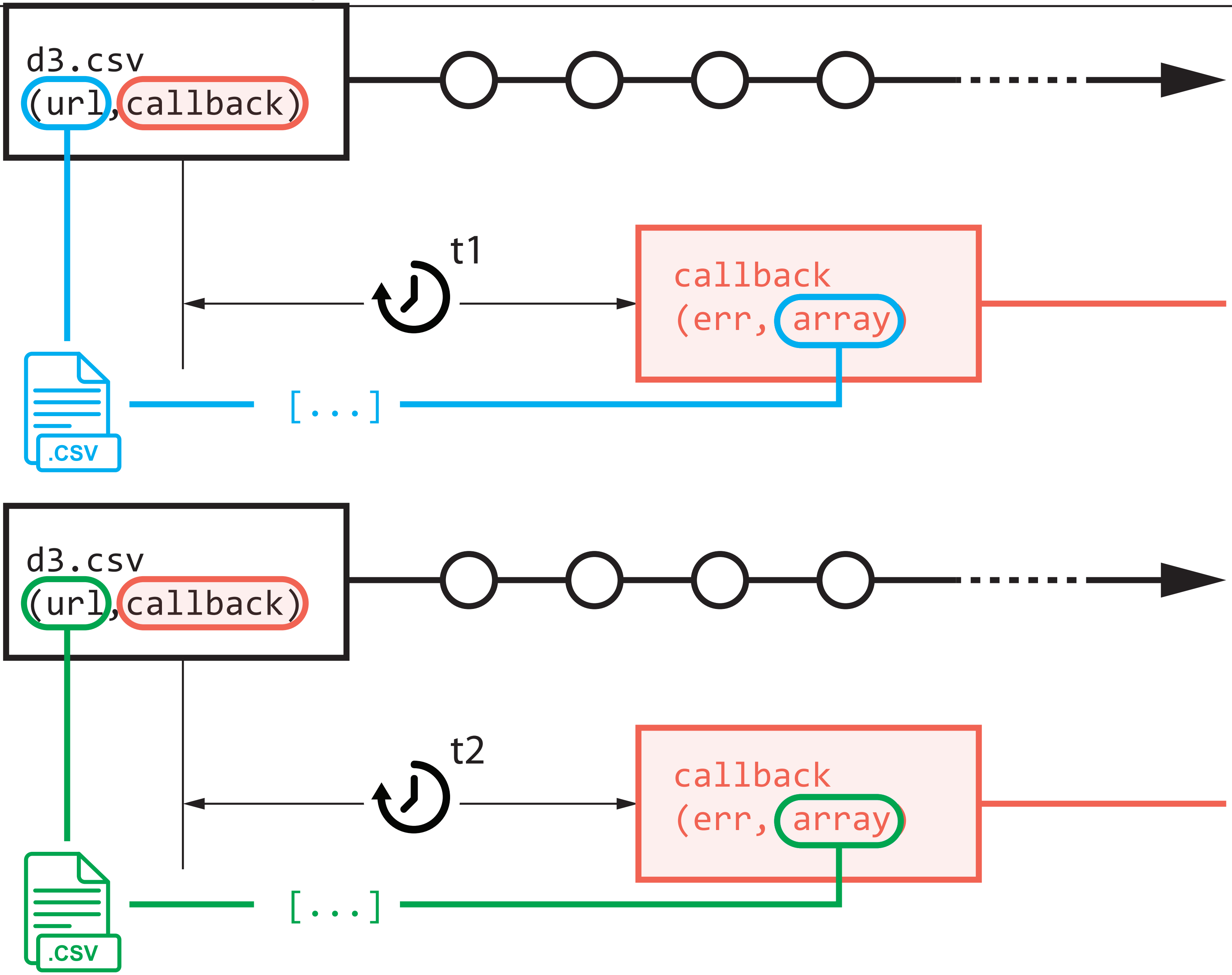
Acquire
Parse
Filter
Mine
Represent
~~Refine~~
Interact

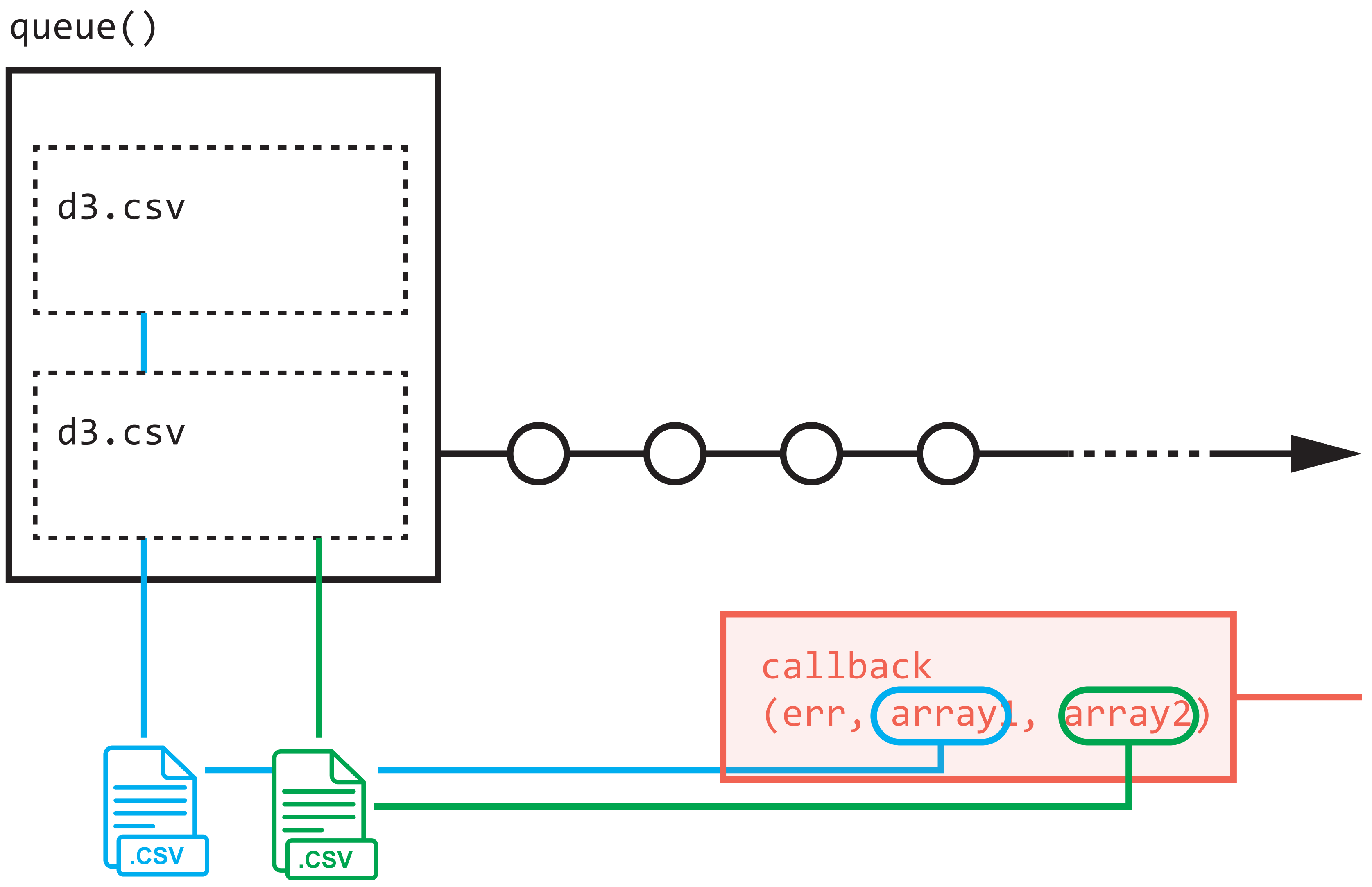
Let's look at some interaction strategies...

Using `queue()` to Import More than One Datasets

```
console.log(“Start”);
d3.csv(“dataset1.csv”,parse,function(err,rows){
      console.log(“Loaded dataset 1”);
   });
d3.csv(“dataset2.csv”,parse,function(err,rows){
      console.log(“Loaded dataset 2”);
   });
console.log(“Finish”);
```


d3.csv
(url, callback)
t1
callback
(err, array)
[...]
.CSV


d3.csv
(url, callback)
t1
callback
(err, array)
.CSV
[...]
d3.csv
(url, callback)
t2
callback
(err, array)
.CSV
[...]


queue()
d3.csv
d3.csv
callback
(err, array1, array2)
.CSV
.CSV

Using `queue()` to Import More than One Datasets

```
d3.csv(“dataset1.csv”,parse,function(err,rows){
        console.log(“Loaded dataset 1”);
    });
d3.csv(“dataset2.csv”,parse,function(err,rows){
        console.log(“Loaded dataset 2”);
    });
```

```
queue()
    .defer(d3.csv, “dataset1.csv”, parse)
    .defer(d3.csv, “dataset2.csv”, parse)
    .ready(function(err, rows1, rows2){
        ...
    });
```

Please review after class:

```
queue()

selection.on( )
```