Interactive Web Programming

1st semester of 2021

Murilo Camargos
(**murilo.filho@fgv.br**)

Heavily based on **Victoria Kirst** slides

Schedule

Today:

- The general update pattern of D3
  - Another view on enter, update and exit
  - Animated transitions
  - Object constancy
  - Nested and single elements

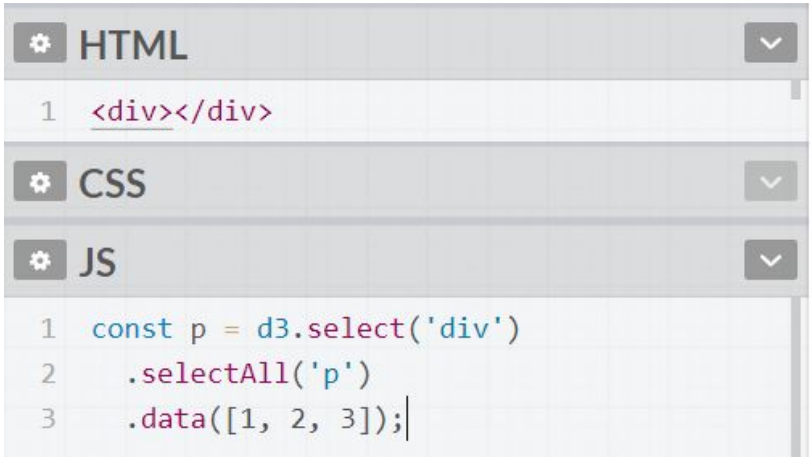
Credits:

- https://www.youtube.com/watch?v=_8V5o2UHG0E
- Intro to Data Viz at Ohio State University
  - http://web.cse.ohio-state.edu/~shen.94/5544/

Data binding with D3

D3 Data Bind

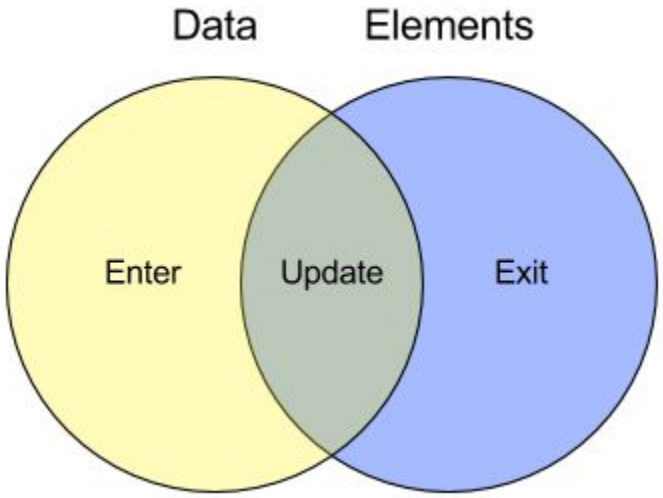
- We can Bind data to a D3 selection by calling the method “data” (even if the DOM elements does not exist yet):



- When data is bound to a selection, each element in the data array is paired with the corresponding node in the selection.

D3 Data Join

- D3 uses a declarative style of programming to bind data to DOM elements and returns three virtual selections:
  - **Enter:** New data, no selection elements
  - **Update:** Data item mapped to existing selection element
  - **Exit:** Selection elements with no data item mapped to them



https://www.amphinicy.com/blog/post/manipulating-svg-using-d3js-library

D3 Data Join: enter

- Since we don’t have any “p” inside our “div”, the join must happen at the “enter” stage:

HTML

```
1  <div></div>
```

CSS

JS

```
1  const p = d3.select('div')
2    .selectAll('p')
3    .data([1, 2, 3])
4    .enter().append('p')
5      .text(d => `Paragraph ${d}`);
```

Paragraph 1

Paragraph 2

Paragraph 3

[Codepen]

D3 Data Join: update

- What if we had one “p” inside our “div”?

HTML

```html
<h1>Example 0</h1>
<div id="ex0">
  <p></p>
</div>

<h1>Example 1</h1>
<div id="ex1">
  <p></p>
</div>

<h1>Example 2</h1>
<div id="ex2">
  <p></p>
</div>
```

JS

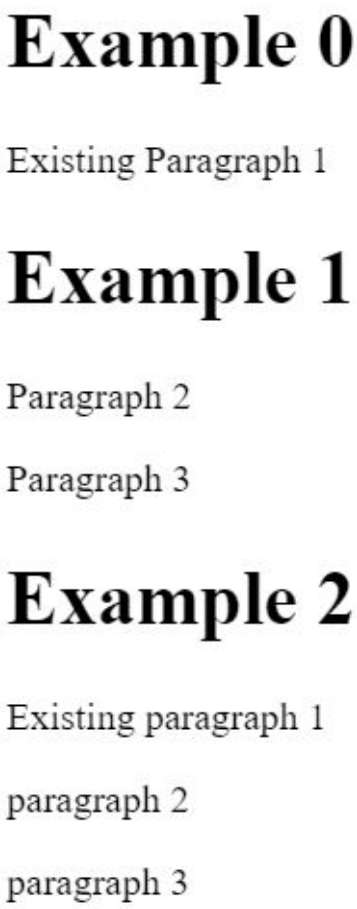
```js
const ex0 = d3.select('#ex0')
  .selectAll('p')
  .data([1, 2, 3])
    .text(d => `Existing Paragraph ${d}`);

const ex1 = d3.select('#ex1')
  .selectAll('p')
  .data([1, 2, 3])
  .enter().append('p')
    .text(d => `Paragraph ${d}`);

const ex2 = d3.select('#ex2')
  .selectAll('p')
  .data([1, 2, 3])
    .text(d => `Existing paragraph ${d}`);

ex2.enter().append('p')
  .text(d => `paragraph ${d}`);
```

[Codepen]



D3 Data Join: exit

- What if we had more than three “p”s inside our “div”?

HTML

```
<div>
  <p></p>
  <p></p>
  <p></p>
</div>
```

JS

```
const div = d3.select('div')
  .selectAll('p')
  .data([1, 2, 3])
    .text(d => `Paragraph ${d}`);
```

Paragraph 1

Paragraph 2

Paragraph 3

```
<div>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
  <p>Paragraph 3</p>
  <p></p>
</div>
```

JS

```
const div = d3.select('div')
  .selectAll('p')
  .data([1, 2, 3])
    .text(d => `Paragraph ${d}`);

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

Paragraph 1

Paragraph 2

Paragraph 3

```
<div>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
  <p>Paragraph 3</p>
</div>
```

Codepen

Making a bar chart with D3

CSV

CSV: **C**omma **S**eparated **V**alue

- Allows data to be save in tabular format
- Each row is a record
- Each record has one or more fields separated by commas

E.g.:
https://raw.githubusercontent.com/murilocamargos/iwp/main/pages/countries/countries.csv

```
country,population
China,1407692960
India,1376238018
United States,331449281
Indonesia,271350000
Pakistan,225200000
Brazil,213057783
Nigeria,211401000
Bangladesh,170566060
Russia,146171015
Mexico,126014024
```

CSV

We want to load this CSV file in our JS code using D3.

- Use the **d3.csv** method.
- It returns a **Promise** (like fetch)!
- It takes care of data structure processing and returns a list of records with **named fields**.

JS

```
1  d3.csv('https://raw.githubusercontent.com/murilocamargos/iwp/main/pages/countries/countries.csv').then(data => console.log(data))
```

Codepen

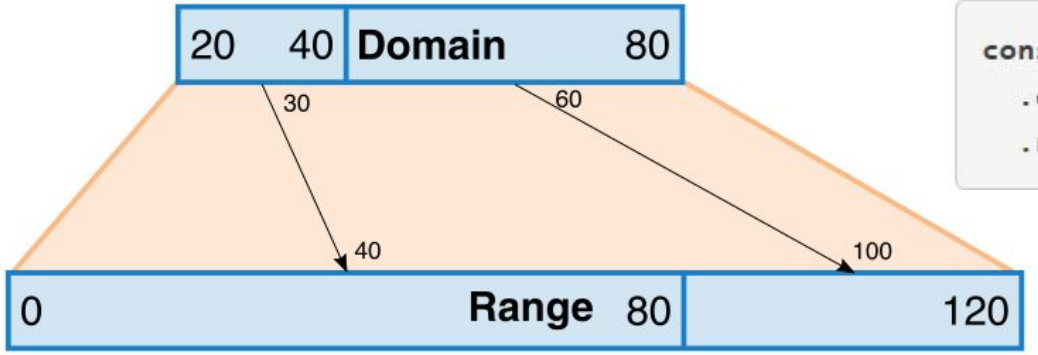
Console

```
// [object Array] (10)
▾ [// [object Object]
▾ {
    "country": "China",
    "population": "1407692960"
  },// [object Object]
▾ {
```

D3 Linear scales

- We can map our values in pixels using D3 scales.
  - “Scales are functions that map from an input domain to an output range.” - **Mike Bostock**

- The linear scale is useful with quantitative attributes.

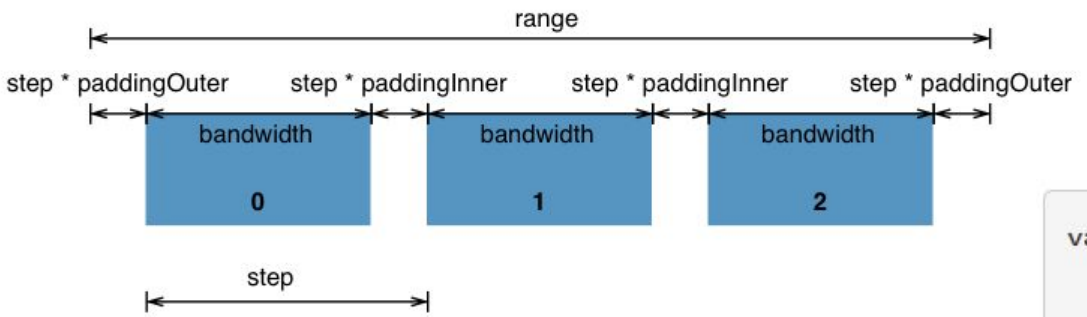



```
const myScale = d3.scaleLinear()
  .domain([20, 80])
  .range([0, 120]);
```

https://github.com/d3/d3-scale/blob/master/README.md
https://www.d3indepth.com/scales/

D3 Band scales

- What about the vertical positioning?
- Band scales are great when the domain is ordinal but the range is continuous and numeric.



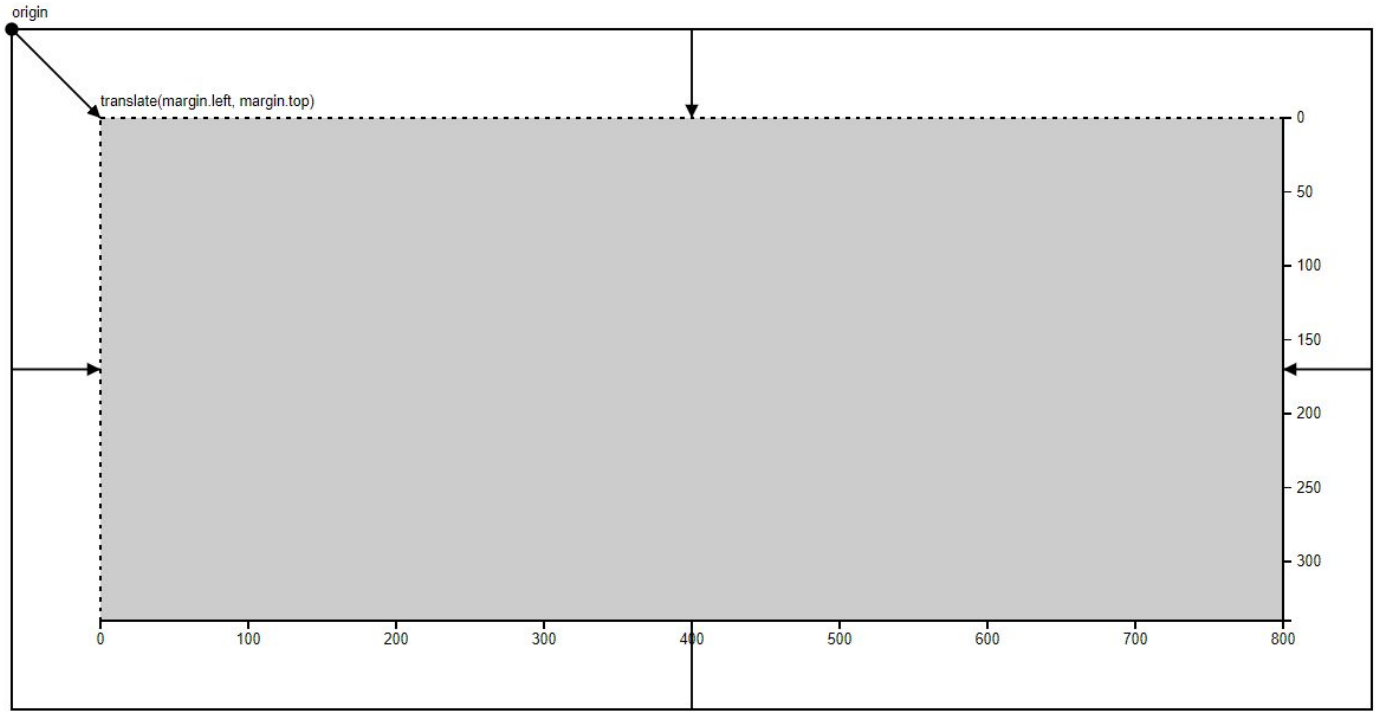

```
var bandScale = d3.scaleBand()
  .domain(['Mon', 'Tue', 'Wed', 'Thu', 'Fri'])
  .range([0, 200]);

bandScale('Mon'); // returns 0
bandScale('Tue'); // returns 40
bandScale('Fri'); // returns 160
```

https://github.com/d3/d3-scale/blob/master/README.md
https://www.d3indepth.com/scales/

D3 Axis

- We need a place to put the axis, i.e., we need space around our bars.
- The margin convention



https://bl.ocks.org/mbostock/3019563

D3 Axis

- You have axisLeft, axisRight, axisBottom and axisTop

- The pattern is to create the axis and pass the scale

```
const yAxis = d3.axisLeft(yScale);
```

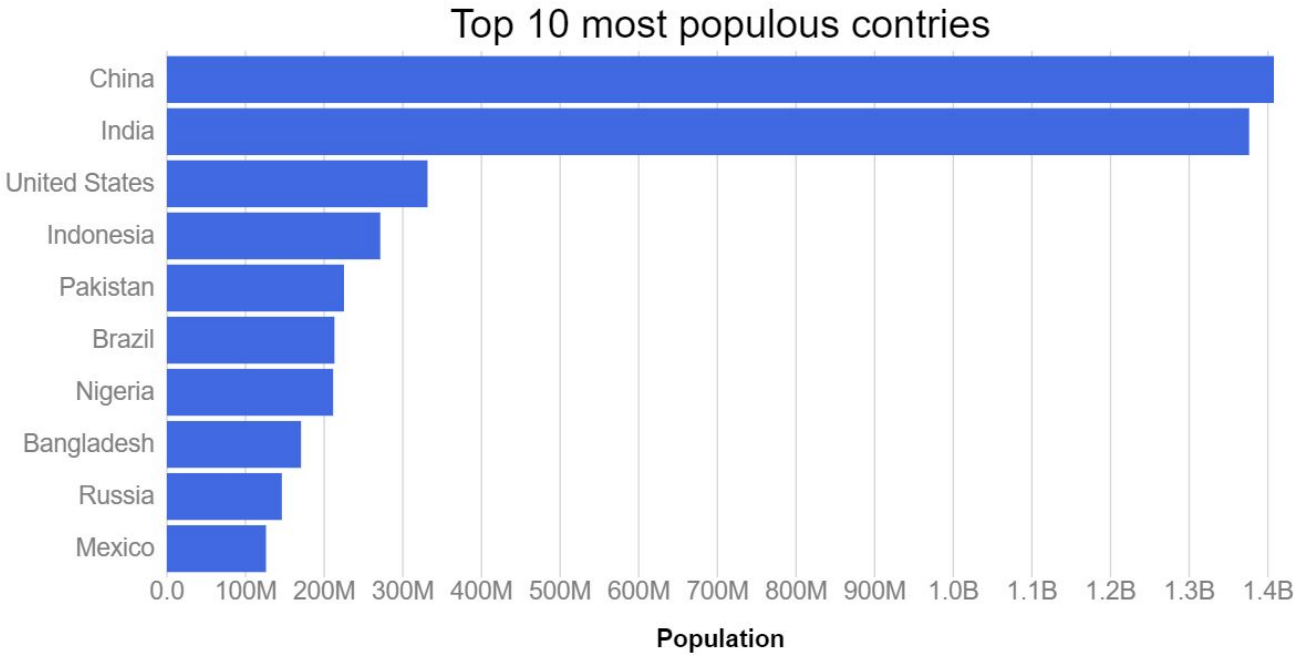
- Then, we create a new group inside the previous one to append the created axis to it:

```
g.append('g').call(yAxis);
```

https://github.com/d3/d3-axis

D3 Axis (customizing)

- Change the number format
- Using CSS
- Removing specific lines
- Add titles and axis labels
- Change tick size

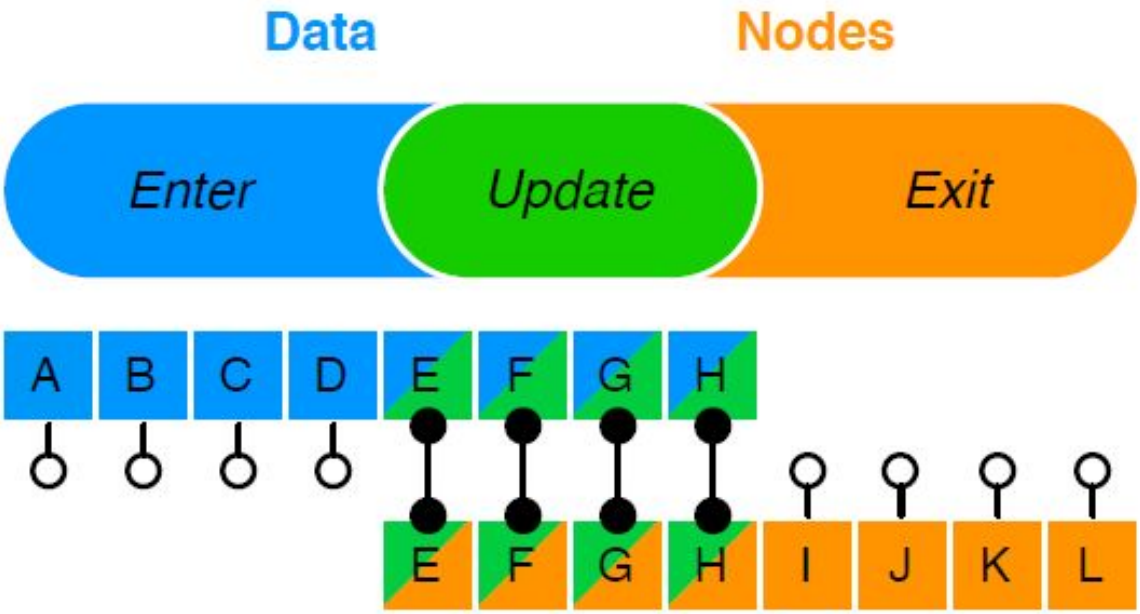


[Codepen]

The general **update** pattern

D3 Data Join (another view)

- You create a **data join** when you call **selectAll** before **data**.
  - You need to select all existing nodes
  - Then pair them with your data array



http://vis.stanford.edu/files/2011-D3-InfoVis.pdf

An example: a bowl of fruits

- Say you have a bowl with 5 apples.

JS

```
const dim = {height: 100, width: 600};

const svg = d3.select('svg')
  .attr('width', dim.width)
  .attr('height', dim.height);

const fruits = [
  {type: 'apple'},
  {type: 'apple'},
  {type: 'apple'},
  {type: 'apple'},
  {type: 'apple'}
];
```

Bowl of fruits: **enter**

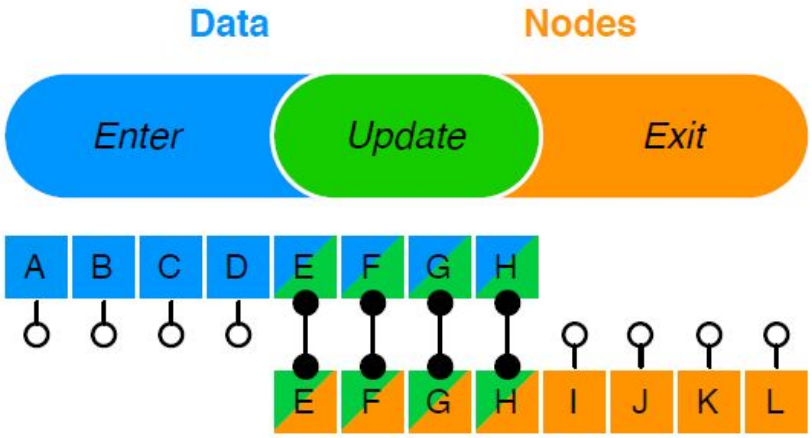
- Say you have a bowl with 5 apples.

JS

```
1  const dim = {height: 100, width: 600};
2
3  const svg = d3.select('svg')
4    .attr('width', dim.width)
5    .attr('height', dim.height);
6
7  const fruits = [
8    {type: 'apple'},
9    {type: 'apple'},
10   {type: 'apple'},
11   {type: 'apple'},
12   {type: 'apple'}
13 ];
```

Create a **Data Join**

```
15 svg.selectAll('circle').data(fruits)
```



Bowl of fruits: **enter**

- Say you have a bowl with 5 apples.

JS

```
1  const dim = {height: 100, width: 600};
2
3  const svg = d3.select('svg')
4    .attr('width', dim.width)
5    .attr('height', dim.height);
6
7  const fruits = [
8    {type: 'apple'},
9    {type: 'apple'},
10   {type: 'apple'},
11   {type: 'apple'},
12   {type: 'apple'}
13 ];
```

Specify **region** and make **transformations**

```
svg.selectAll('circle').data(fruits)
  .enter().append('circle')
    .attr('cx', (d,i) => i*120 + 60)
    .attr('cy', dim.height/2)
    .attr('fill', '#d13636')
    .attr('r', 50);
```

Codepen

Bowl of fruits: **exit**

- What if you eat an apple?

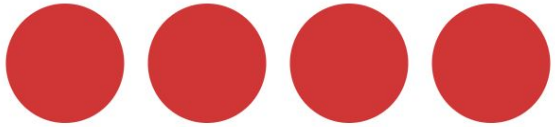
```
22  fruits.pop();
```

- We need to create the data join **again** and handle the **exit** section.

```
24  svg.selectAll('circle').data(fruits)
25    .exit().attr('fill', 'black');
```

- We generally want to remove the node:

```
22  fruits.pop();
23
24  svg.selectAll('circle').data(fruits)
25    .exit().remove();
```



[Codepen]

Bowl of fruits: **exit**

- Having two chunks of code works, but a more general pattern is to combine them into one **render** function.

JS

```
7  const fruits = [
8    {type: 'apple'},
9    {type: 'apple'},
10   {type: 'apple'},
11   {type: 'apple'},
12   {type: 'apple'}
13 ];
14
15 svg.selectAll('circle').data(fruits)
16   .enter().append('circle')
17     .attr('cx', (d,i) => i*120 + 60)
18     .attr('cy', dim.height/2)
19     .attr('fill', '#d13636')
20     .attr('r', 50);
21
22 fruits.pop();
23
24 svg.selectAll('circle').data(fruits)
25   .exit().remove();
```

Bowl of fruits: **exit**

- Having two chunks of code works, but a more general pattern is to combine them into one **render** function.

JS

```
const fruits = [
  {type: 'apple'},
  {type: 'apple'},
  {type: 'apple'},
  {type: 'apple'},
  {type: 'apple'}
];

svg.selectAll('circle').data(fruits)
  .enter().append('circle')
    .attr('cx', (d,i) => i*120 + 60)
    .attr('cy', dim.height/2)
    .attr('fill', '#d13636')
    .attr('r', 50);

fruits.pop();

svg.selectAll('circle').data(fruits)
  .exit().remove();
```

```
const render = (fruits) => {
  svg.selectAll('circle').data(fruits)
    .enter().append('circle')
      .attr('cx', (d,i) => i*120 + 60)
      .attr('cy', dim.height/2)
      .attr('fill', '#d13636')
      .attr('r', 50);
  svg.selectAll('circle').data(fruits)
    .exit().remove();
}
```

```
render(fruits);
fruits.pop();
render(fruits);
```

[Codepen]

Bowl of fruits: **exit**

- We can also clean a little bit the render function

```
const render = (fruits) => {
  svg.selectAll('circle').data(fruits)
    .enter().append('circle')
      .attr('cx', (d,i) => i*120 + 60)
      .attr('cy', dim.height/2)
      .attr('fill', '#d13636')
      .attr('r', 50);
  svg.selectAll('circle').data(fruits)
    .exit().remove();
}
```

Bowl of fruits: **exit**

- We can also clean a little bit the render function

```
const render = (fruits) => {
  svg.selectAll('circle').data(fruits)
    .enter().append('circle')
      .attr('cx', (d,i) => i*120 + 60)
      .attr('cy', dim.height/2)
      .attr('fill', '#d13636')
      .attr('r', 50);
  svg.selectAll('circle').data(fruits)
    .exit().remove();
}
```

```
const render = (fruits) => {
  const circles = svg.selectAll('circle')
    .data(fruits);
  circles
    .enter().append('circle')
      .attr('cx', (d,i) => i*120 + 60)
      .attr('cy', dim.height/2)
      .attr('fill', '#d13636')
      .attr('r', 50);
  circles
    .exit().remove();
}
```

[Codepen]

Bowl of fruits: **exit**

- We can also clean a little bit the render function
- And add a time dimension to our change

```
const render = (fruits) => {
  svg.selectAll('circle').data(fruits)
    .enter().append('circle')
      .attr('cx', (d,i) => i*120 + 60)
      .attr('cy', dim.height/2)
      .attr('fill', '#d13636')
      .attr('r', 50);
  svg.selectAll('circle').data(fruits)
    .exit().remove();
}
```

```
const render = (fruits) => {
  const circles = svg.selectAll('circle')
    .data(fruits);
  circles
    .enter().append('circle')
      .attr('cx', (d,i) => i*120 + 60)
      .attr('cy', dim.height/2)
      .attr('fill', '#d13636')
      .attr('r', 50);
  circles
    .exit().remove();
}
```

[Codepen]

```
render(fruits);
setTimeout(() => {
  fruits.pop();
  render(fruits);
}, 1000);
```

Bowl of fruits: **update**

- What if you replace an apple with a lemon?

```
35 setTimeout(() => {
36   fruits[2].type = 'lemon';
37   render(fruits);
38 }, 2000);
```

Bowl of fruits: **update**

- What if you replace an apple with a lemon?

```
35 setTimeout(() => {
36   fruits[2].type = 'lemon';
37   render(fruits);
38 }, 2000);
```

- Both the color and the size of lemons are different than apples. We can use ordinal scales to handle that:

```
7  const colorScale = d3.scaleOrdinal()
8    .domain(['apple', 'lemon'])
9    .range(['#d13636', '#c7eb3b']);
10
11 const sizeScale = d3.scaleOrdinal()
12   .domain(['apple', 'lemon'])
13   .range([50, 30]);
```

```
circles
  .enter().append('circle')
    .attr('cx', (d,i) => i*120 + 60)
    .attr('cy', dim.height/2)
    .attr('fill', d => colorScale(d.type))
    .attr('r', d => sizeScale(d.type));
```

Bowl of fruits: **update**

- What if you replace an apple with a lemon?

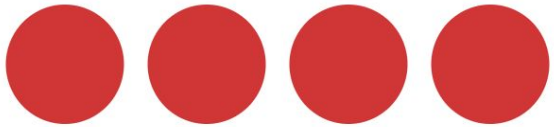
```
35 setTimeout(() => {
36   fruits[2].type = 'lemon';
37   render(fruits);
38 }, 2000);
```

- Both the color and the size of lemons are different than apples. We can use ordinal scales to handle that:

```
7  const colorScale = d3.scaleOrdinal()
8    .domain(['apple', 'lemon'])
9    .range(['#d13636', '#c7eb3b']);
10
11 const sizeScale = d3.scaleOrdinal()
12   .domain(['apple', 'lemon'])
13   .range([50, 30]);
```

```
circles
  .enter().append('circle')
    .attr('cx', (d,i) => i*120 + 60)
    .attr('cy', dim.height/2)
    .attr('fill', d => colorScale(d.type))
    .attr('r', d => sizeScale(d.type));
```

- But still, **nothing different happens**!!!

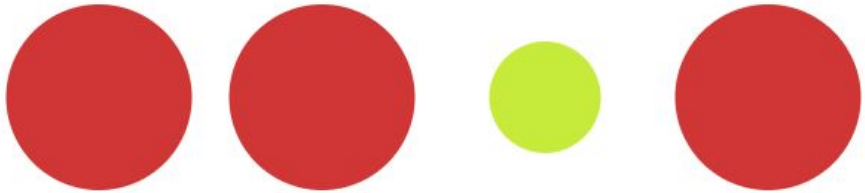


[Codepen]

Bowl of fruits: **update**

- We are only handling the **enter** and **exit** selections.
- To change our nodes according to data changes we need to implement the **update** selection.
- Enter and exit selections are explicit, update is the default.

```
const render = (fruits) => {
  const circles = svg.selectAll('circle')
    .data(fruits);
  circles
    .enter().append('circle')
      .attr('cx', (d,i) => i*120 + 60)
      .attr('cy', dim.height/2)
      .attr('fill', d => colorScale(d.type))
      .attr('r', d => sizeScale(d.type));
  circles
    .attr('fill', d => colorScale(d.type))
    .attr('r', d => sizeScale(d.type));
  circles
    .exit().remove();
}
```



[Codepen]

Bowl of fruits: **merge**

- We can use merge to clear the duplicated code we have for enter and update selections:

```
const render = (fruits) => {
  const circles = svg.selectAll('circle')
    .data(fruits);
  circles
    .enter().append('circle')
      .attr('cx', (d,i) => i*120 + 60)
      .attr('cy', dim.height/2)
      .attr('fill', d => colorScale(d.type))
      .attr('r', d => sizeScale(d.type));
  circles
    .attr('fill', d => colorScale(d.type))
    .attr('r', d => sizeScale(d.type));
  circles
    .exit().remove();
}
```

```
15 const render = (fruits) => {
16   const circles = svg.selectAll('circle')
17     .data(fruits);
18   circles
19     .enter().append('circle')
20       .attr('cx', (d,i) => i*120 + 60)
21       .attr('cy', dim.height/2)
22     .merge(circles)
23       .attr('fill', d => colorScale(d.type))
24       .attr('r', d => sizeScale(d.type));
25   circles
26     .exit().remove();
27 }
```

[Codepen]

Bowl of fruits: **merge**

- We can use merge to clear the duplicated code we have for enter and update selections:

```
15 const render = (fruits) => {
16   const circles = svg.selectAll('circle')
17     .data(fruits);
18   circles
19     .enter().append('circle')
20       .attr('cx', (d,i) => i*120 + 60)
21       .attr('cy', dim.height/2)
22     .merge(circles)
23       .attr('fill', d => colorScale(d.type))
24       .attr('r', d => sizeScale(d.type));
25   circles
26     .exit().remove();
27 }
```

- At line 21 you have the **enter** selection.
- At line 22 you **merge** this selection with the **update** selection.

- This is a complete update pattern that handles enter, update and exit.

Codepen

Bowl of fruits

- To wrap it up, let's remove the second apple, filtering it out:

```
49 ▾ setTimeout(() => {
50     fruits = fruits.filter((elem, i) => i != 1);
51     render(fruits, dim.height);
52   }, 3000);
```

[Codepen]

Animated transitions

Bowl of fruits: **transitions**

- We can change the fruits' radius in a smooth way:

```
15 const render = (fruits, height) => {
16   const circles = svg.selectAll('circle')
17     .data(fruits);
18   circles
19     .enter().append('circle')
20       .attr('cx', (d,i) => i*120 + 60)
21       .attr('cy', dim.height/2)
22       .attr('r', 0)
23     .merge(circles)
24       .attr('fill', d => colorScale(d.type))
25     .transition().duration(1000)
26       .attr('r', d => sizeScale(d.type));
27   circles
28     .exit().transition().duration(1000)
29       .attr('r', 0)
30     .remove();
31 }
```

Codepen

Bowl of fruits: **transitions**

- We can change the fruits' radius in a smooth way:

```
15 const render = (fruits, height) => {
16   const circles = svg.selectAll('circle')
17     .data(fruits);
18   circles
19     .enter().append('circle')
20       .attr('cx', (d,i) => i*120 + 60)
21       .attr('cy', dim.height/2)
22       .attr('r', 0)
23     .merge(circles)
24       .attr('fill', d => colorScale(d.type))
25     .transition().duration(1000)
26       .attr('r', d => sizeScale(d.type));
27   circles
28     .exit().transition().duration(1000)
29       .attr('r', 0)
30     .remove();
31 }
```

- Notice how the lemon does not move as it should.

- D3 needs to know how to map data and DOM elements
  - By default it uses the index.

Codepen

Bowl of fruits: **object constancy**

- We can change the default and use specific IDs for each object

```
15 const render = (fruits, height) => {
16   const circles = svg.selectAll('circle')
17     .data(fruits, d => d.id);
```

```
33 let fruits = [
34   {type: 'apple', id: 0},
35   {type: 'apple', id: 1},
36   {type: 'apple', id: 2},
37   {type: 'apple', id: 3},
38   {type: 'apple', id: 4}
39 ];
```

[Codepen]

Bowl of fruits: **object constancy**

- We need to update the cx when data changes as well:

```
15 const render = (fruits, height) => {
16   const circles = svg.selectAll('circle')
17     .data(fruits, d => d.id);
```

```
circles
  .enter().append('circle')
    .attr('cx', (d,i) => i*120 + 60)
    .attr('cy', dim.height/2)
    .attr('r', 0)
  .merge(circles)
    .attr('fill', d => colorScale(d.type))
  .transition().duration(1000)
    .attr('cx', (d,i) => i*120 + 60)
    .attr('r', d => sizeScale(d.type));
```

[Codepen]

You'll need to repeat the logic but you can create a function.

Nested elements

Bowl of fruits: adding names

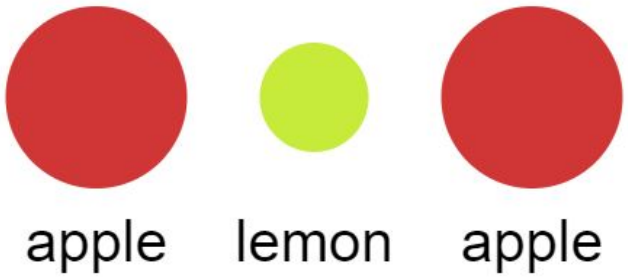
- Let's add the names of each fruit below the circle:

```
28    const text = svg.selectAll('text')
29      .data(fruits);
30    text
31      .enter().append('text')
32        .attr('x', (d,i) => i*120 + 60)
33        .attr('y', dim.height/2 + 90)
34      .merge(text)
35        .text(d => d.type);
36    text
37      .exit().remove();
```

CSS

```
1  text {
2    font-size: 2em;
3    font-family: sans-serif;
4    text-anchor: middle;
5  }
```

We just copied the logic for the circles.



Codepen

Bowl of fruits: adding groups

- We can create the groups with the **g** tag:
  - Now we only have **transform** to move the groups

```
16    const groups = svg.selectAll('g')
17      .data(fruits);
18    groups.enter().append('g')
19      .merge(groups)
20        .attr('transform', (d,i) => {
21          `translate(${i*120 + 60}, ${height/2})`
22        });
23    groups
24      .exit().remove();
```

Codepen

Bowl of fruits: nesting elements

- We can use the **groups** selection to add a circle to each group:

```
26    const circles = groups.select('circle');
```

- For the enter selection, we need access to the **groups** enter selection

```
16    const groups = svg.selectAll('g')
17      .data(fruits);
18    const groupsEnter = groups.enter().append('g');
19    groupsEnter.merge(groups)
20      .attr('transform', (d,i) => {
21        `translate(${i*120 + 60}, ${height/2})`
22      });
23    groups
24      .exit().remove();
```

Codepen

Bowl of fruits: nesting elements

- Now we can append the circle for each group in the groups selection:

```
27    groupsEnter.append('circle')
28      .merge(circles)
29        .attr('fill', d => colorScale(d.type))
30        .attr('r', d => sizeScale(d.type));
```

- Notice we don’t need cx and cy anymore!

Codepen

Bowl of fruits: nesting elements

- We can do the same thing with the text nodes:

```
31    groupsEnter.append('text')
32      .merge(groups.select('text'))
33        .attr('y', 90)
34        .text(d => d.type);
```

- We still need the **y** offset but we can get rid of the **x** position.

```
▼<svg width="600" height="300">
  ▼<g transform="translate(60, 150)">
      <circle fill="#d13636" r="50"></circle>
      <text y="90">apple</text>
    </g>
  ▼<g transform="translate(180, 150)">
      <circle fill="#c7eb3b" r="30"></circle>
      <text y="90">lemon</text>
    </g>
  ▼<g transform="translate(300, 150)">
      <circle fill="#d13636" r="50"></circle>
      <text y="90">apple</text>
    </g>
</svg>
```

Codepen

Single elements

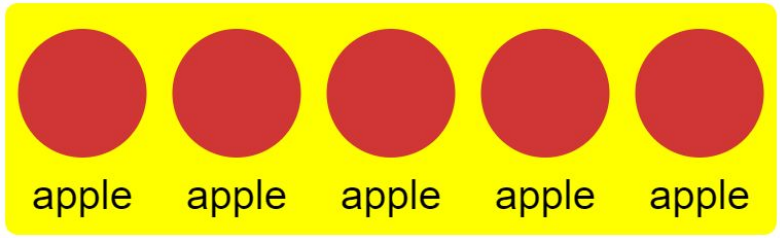
A special case

Bowl of fruits: single element

- Suppose you want to render a single element: the fruit bowl

```
15 const render = (fruits, height) => {
16   const bowl = svg.append('rect')
17     .attr('y', height/2-70)
18     .attr('width', 600)
19     .attr('height', 180)
20     .attr('rx', 10)
21     .attr('fill', 'yellow');
```

First update

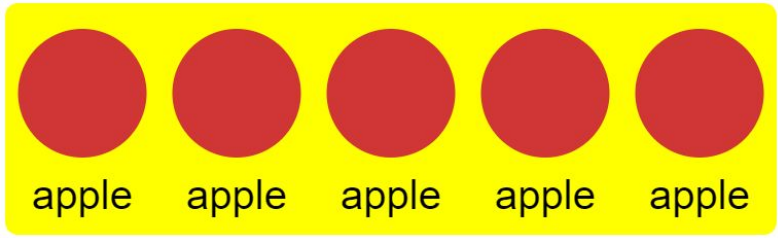


Bowl of fruits: single element

- Suppose you want to render a single element: the fruit bowl

```
15 const render = (fruits, height) => {
16   const bowl = svg.append('rect')
17     .attr('y', height/2-70)
18     .attr('width', 600)
19     .attr('height', 180)
20     .attr('rx', 10)
21     .attr('fill', 'yellow');
```

First update



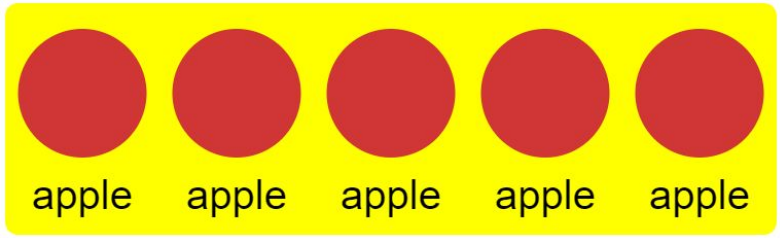
Next update

Bowl of fruits: single element

- Suppose you want to render a single element: the fruit bowl

```
15 const render = (fruits, height) => {
16   const bowl = svg.append('rect')
17     .attr('y', height/2-70)
18     .attr('width', 600)
19     .attr('height', 180)
20     .attr('rx', 10)
21     .attr('fill', 'yellow');
```

First update



Next update

The result:

```
▼<svg width="600" height="300">
    <rect y="80" width="600" height="180" rx="10" fill="yellow"></rect>
  ▶<g transform="translate(60, 150)">…</g>
  ▶<g transform="translate(180, 150)">…</g>
  ▶<g transform="translate(300, 150)">…</g>
    <rect y="80" width="600" height="180" rx="10" fill="yellow"></rect>
    <rect y="80" width="600" height="180" rx="10" fill="yellow"></rect>
    <rect y="80" width="600" height="180" rx="10" fill="yellow"></rect>
  </svg>
```

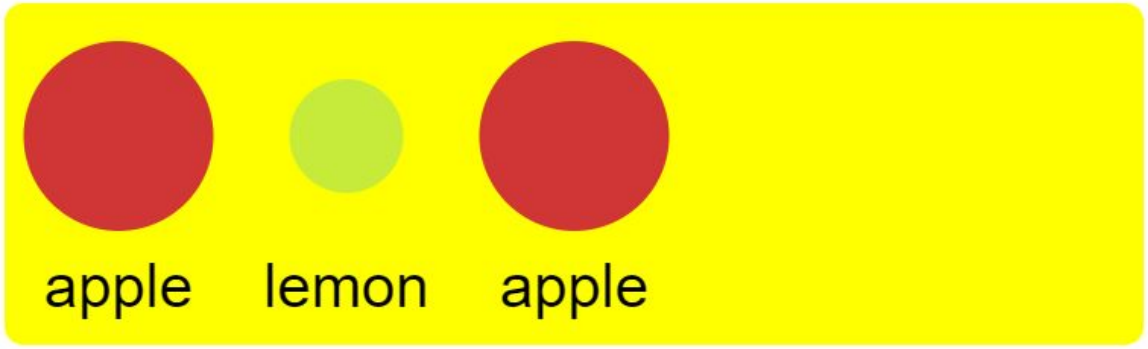
Codepen

Bowl of fruits: single element

- You can use a single element in the general update pattern:

```
15 const render = (fruits, height) => {
16   const bowl = svg.selectAll('rect').data([null])
17     .enter().append('rect')
18       .attr('y', height/2-70)
19       .attr('width', 600)
20       .attr('height', 180)
21       .attr('rx', 10)
22       .attr('fill', 'yellow');
```

The result:



Codepen