Interactive Web Programming

1st semester of 2021

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

Heavily based on **Victoria Kirst** slides

Schedule

Today:

- Interactions with D3
  - Listening for click events
  - Unidirectional data flow
  - Selecting marks by clicking or hovering
  - Capturing mouse motion to render text

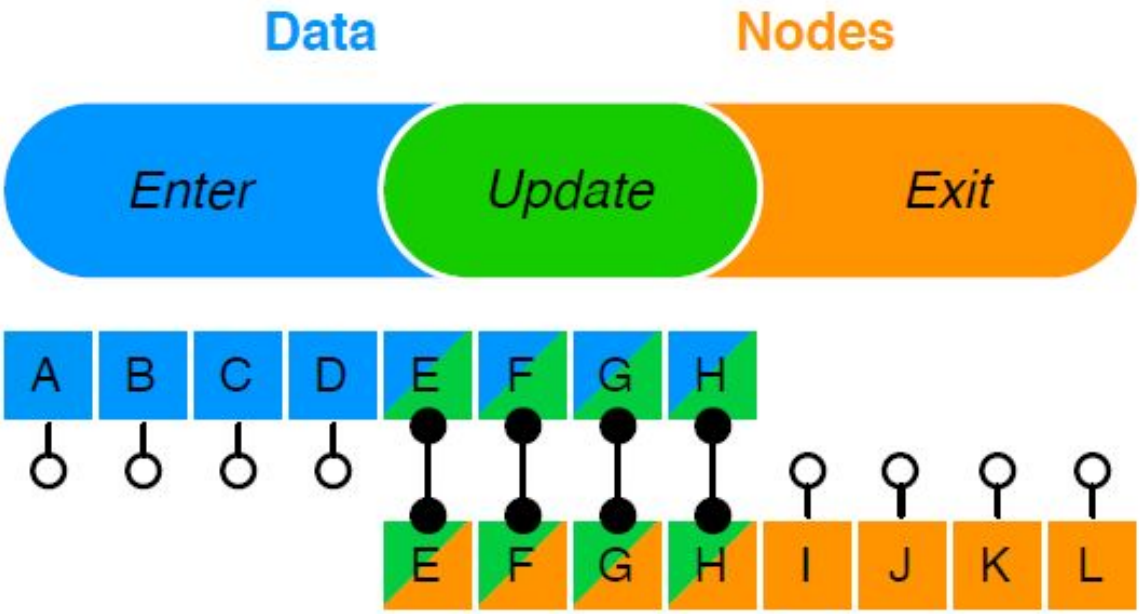
Credits:

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

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**

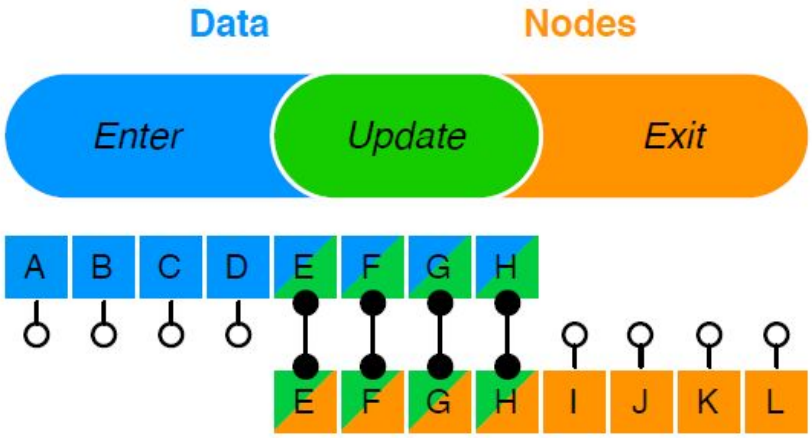
- Say you have a bowl with 5 apples.

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?

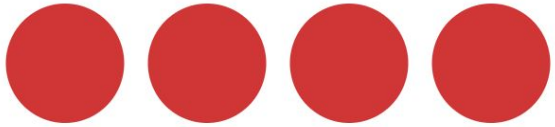
```
22  fruits.pop();
```

- 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?

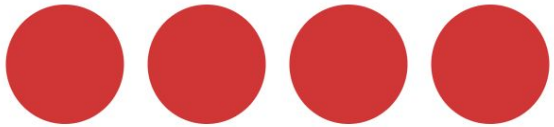
```
35 setTimeout(() => {
36   fruits[2].type = 'lemon';
37   render(fruits);
38 }, 2000);
```

- 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**!!!

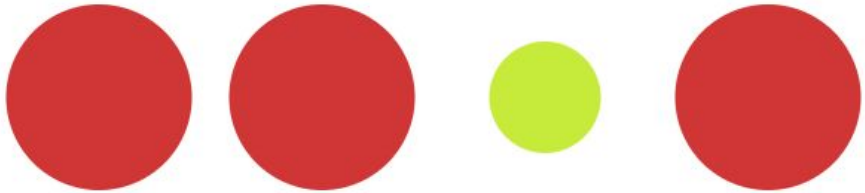


[Codepen]

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.

Interaction using D3.js

Listening for click events

- Say we want to be able to “select” a fruit, e.g., adding a border. **The first step is to listen for click events**.

Listening for click events

- Say we want to be able to “select” a fruit, e.g., adding a border. **The first step is to listen for click events**.
- To add a “**click**” event listener like we used to do with “**addEventListener**” we can use D3’s method “**on**”.
  - `selection.on(‘click’, callbackFunction)`

Listening for click events

- Say we want to be able to “select” a fruit, e.g., adding a border. **The first step is to listen for click events**.
- To add a “**click**” event listener like we used to do with “**addEventListener**” we can use D3’s method “**on**”.
  - `selection.on(‘click’, callbackFunction)`
- It can be added in the “merge” selection:

```
circles
  .enter().append('circle')
    .attr('cx', xPosition)
    .attr('cy', dim.height/2)
    .attr('r', 0)
  .merge(circles)
    .attr('fill', d => colorScale(d.type))
    .on('click', () => {
      console.log('clicked');
    })
  .transition().duration(1000)
```

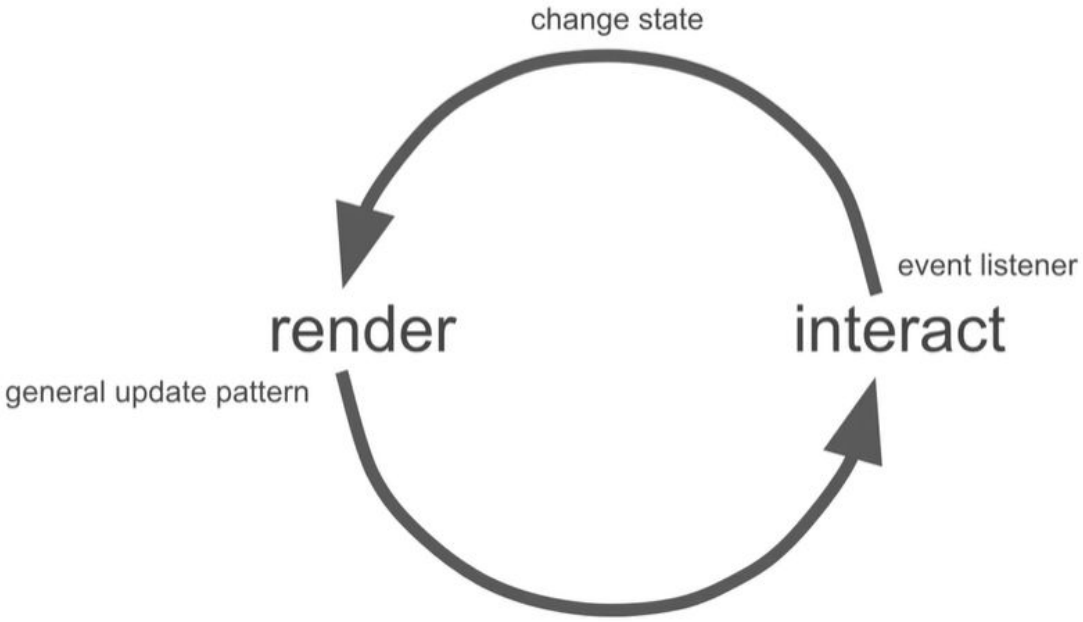
Console

```
"clicked"
```

[Codepen]

Unidirectional data flow

- After rendering, the user can interact with what's on the screen, then the system re-renders everything.
- After listening to the event, we need to change the state.
- Then we re-render everything with the general update pattern.



Unidirectional data flow

- Our fruits have IDs and we need a state variable to store the selected fruit:

```
let fruits = [
  {type: 'apple', id: 0},
  {type: 'apple', id: 1},
  {type: 'apple', id: 2},
  {type: 'apple', id: 3},
  {type: 'apple', id: 4}
];
let selectedFruit = null;
```

Unidirectional data flow

- Our fruits have IDs and we need a state variable to store the selected fruit:

```
let fruits = [
    {type: 'apple', id: 0},
    {type: 'apple', id: 1},
    {type: 'apple', id: 2},
    {type: 'apple', id: 3},
    {type: 'apple', id: 4}
];
let selectedFruit = null;
```

- The setTimeout implements the change state/render pattern. We can do something similar to select a fruit:

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

```
const selectFruit = fruitID => {
    selectedFruit = fruitID;
    render(fruits, dim.height);
};
```

[Codepen]

Selecting a mark by clicking

- The function of the event listener has two parameters:
  - The event itself
  - The datum associated with the event

```
.on('click', (event, datum) => {
  console.log(event);
  console.log(datum);
})
```

Selecting a mark by clicking

- The function of the event listener has two parameters:
  - The event itself
  - The datum associated with the event

```
.on('click', (event, datum) => {
  console.log(event);
  console.log(datum);
})
```

- Now we can call “selectFruit” passing the clicked fruit ID:

```
.on('click', (e, d) => selectFruit(d.id))
```

Codepen

Console

```
// [object MouseEvent]
{
  "isTrusted": true
}
```

```
// [object Object]
{
  "type": "apple",
  "id": 1
}
```

Highlighting selected mark

- Now we can set the “stroke” attribute to show the selection.

```
38      .merge(circles)
39        .attr('fill', d => colorScale(d.type))
40        .attr('stroke-width', 5)
41        .attr('stroke', d =>
42          d.id === selectedFruit ? 'black' : 'none')
43        .on('click', (e, d) => selectFruit(d.id))
```

Codepen

Selecting a mark by hovering

- Say we want to select a mark when we hover it with the mouse and deselect when the pointer goes out.

Selecting a mark by hovering

- Say we want to select a mark when we hover it with the mouse and deselect when the pointer goes out.

- Instead of listening to “click” event you can listen to the “mouseover” event:

```
.on('mouseover', (e, d) => selectFruit(d.id))
```

Codepen

Selecting a mark by hovering

- Say we want to select a mark when we hover it with the mouse and deselect when the pointer goes out.

- Instead of listening to “click” event you can listen to the “mouseover” event:

```
.on('mouseover', (e, d) => selectFruit(d.id))
```

Codepen

- The selection is still active after we move the mouse out of the object.

Selecting a mark by hovering

- Say we want to select a mark when we hover it with the mouse and deselect when the pointer goes out.

- Instead of listening to “click” event you can listen to the “mouseover” event:

```
.on('mouseover', (e, d) => selectFruit(d.id))
```

Codepen

- The selection is still active after we move the mouse out of the object.

- We just need to implement the “mouseout” event

```
.on('mouseout', () => selectFruit(null))
```

Codepen

Let’s create a tooltip

Creating the toolbox

- First thing is to create the toolbox composed by a group with
  - A rect box
  - A text element

```
const tooltip = svg.append('g')
  .attr('transform', 'translate(50,50)');
```

Creating the toolbox

- First thing is to create the toolbox composed by a group with
  - A rect box
  - A text element

```
const tooltip = svg.append('g')
  .attr('transform', 'translate(50,50)');
```

```
const tooltipBox = tooltip.append('rect')
  .attr('fill', 'white')
  .attr('stroke', 'black')
  .attr('stroke-width', 2)
  .attr('width', 100)
  .attr('height', 40);
```

```
const tooltipText = tooltip.append('text')
  .text('apple')
  .attr('x', 50)
  .attr('y', 25)
  .attr('class', 'tooltip-text')
  .attr('text-anchor', 'middle')
  .attr('font-size', '1.5em');
```

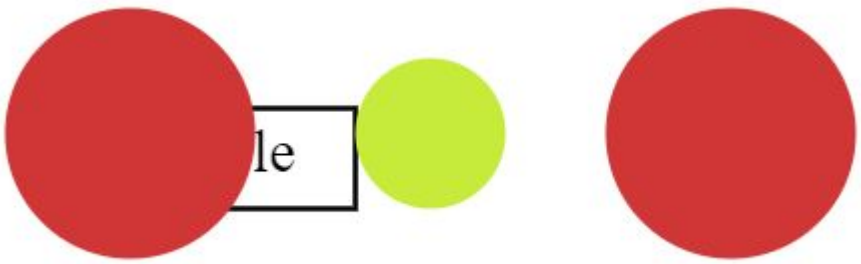
Creating the toolbox

- First thing is to create the toolbox composed by a group with
  - A rect box
  - A text element

```
const tooltip = svg.append('g')
  .attr('transform', 'translate(50,50)');
```

```
const tooltipBox = tooltip.append('rect')
  .attr('fill', 'white')
  .attr('stroke', 'black')
  .attr('stroke-width', 2)
  .attr('width', 100)
  .attr('height', 40);
```

```
const tooltipText = tooltip.append('text')
  .text('apple')
  .attr('x', 50)
  .attr('y', 25)
  .attr('class', 'tooltip-text')
  .attr('text-anchor', 'middle')
  .attr('font-size', '1.5em');
```

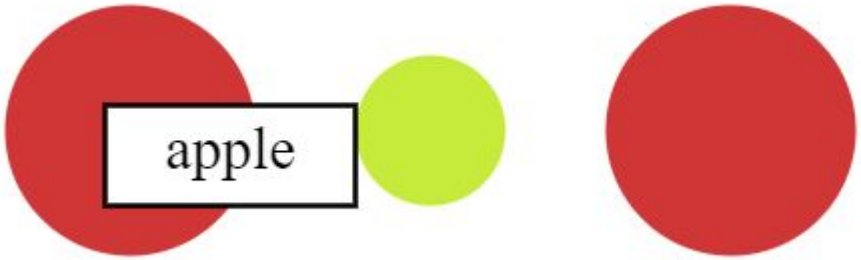


[Codepen]

Moving the box “up”

- D3 provides a great function to “raise” the objects.
  - We can add it at the end of our “render” function

```
tooltip.raise();
```

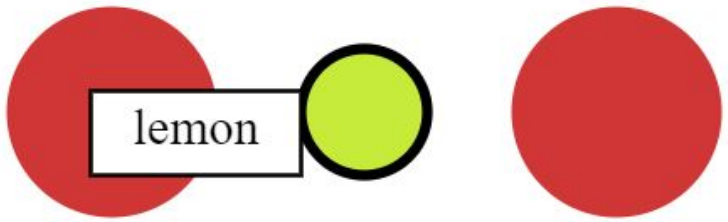


Codepen

Changing the name dynamically

- Now let's change the "selectFruit" function to accept the whole fruit object and change the "tooltipText"

```
const selectFruit = fruit => {
  if (fruit !== null) {
    selectedFruit = fruit.id;
    tooltipText.text(fruit.type);
  } else {
    selectedFruit = null;
  }
  render(fruits, dim.height);
};
```



[Codepen]

Show/hide the tooltip

- We can set the initial state of the whole tooltip's opacity to zero (hiding it)

```
const tooltip = svg.append('g')
  .attr('transform', 'translate(50,50)')
  .attr('opacity', '0');
```

Show/hide the tooltip

- We can set the initial state of the whole tooltip's opacity to zero (hiding it)

```
const tooltip = svg.append('g')
  .attr('transform', 'translate(50,50)')
  .attr('opacity', '0');
```

- Then we tweak this opacity accordingly in our "selectFruit" function.

```
const selectFruit = fruit => {
  if (fruit !== null) {
    selectedFruit = fruit.id;
    tooltipText.text(fruit.type);
    tooltip.attr('opacity', '1');
  } else {
    selectedFruit = null;
    tooltip.attr('opacity', '0');
  }
  render(fruits, dim.height);
};
```

[Codepen]

Changing the position dynamically

- We can access the mouse position with pageX and pageY and use that to change the tooltip's position.
- If we want to change with the movement, we can use the "mousemove" event:

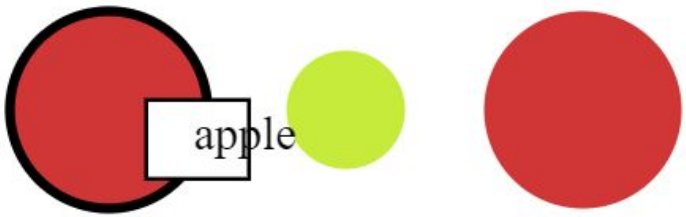
```
.on('mousemove', e => {
  tooltip
    .attr('transform', `translate(${e.pageX},${e.pageY})`)
})
```

Codepen

Changing the size dynamically

- We can change the tooltip size according to the text size using the getBBox function:

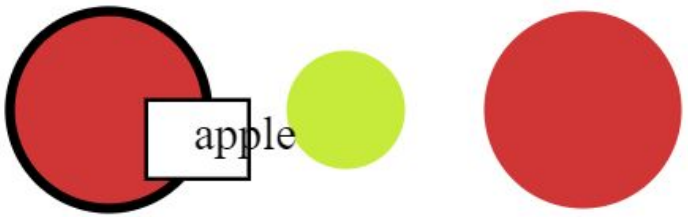
```
const tooltipSize = tooltipText.node().getBBox();
const tooltipWidth = tooltipSize.width;
tooltipBox.attr('width', tooltipWidth);
```



Changing the size dynamically

- We can change the tooltip size according to the text size using the getBBox function:

```
const tooltipSize = tooltipText.node().getBBox();
const tooltipWidth = tooltipSize.width;
tooltipBox.attr('width', tooltipWidth);
```



- We also need to reposition the text in the new box

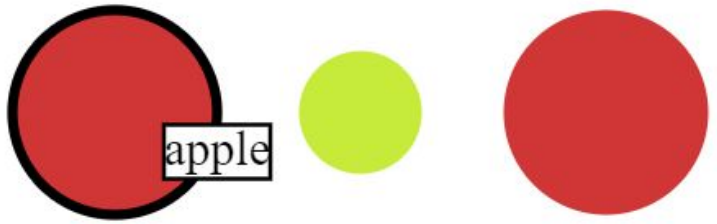
```
const tooltipSize = tooltipText.node().getBBox();
const tooltipWidth = tooltipSize.width;
tooltipBox.attr('width', tooltipWidth);
tooltipText.attr('x', tooltipWidth/2);
```

Codepen

Changing the size dynamically

- If we do the same thing with the height:

```
const tooltipSize = tooltipText.node().getBBox();
const tooltipWidth = tooltipSize.width;
const tooltipHeight = tooltipSize.height;
tooltipBox
  .attr('width', tooltipWidth)
  .attr('height', tooltipHeight);
tooltipText
  .attr('x', tooltipWidth/2)
  .attr('y', tooltipHeight/2 + tooltipSize.height/4);
```

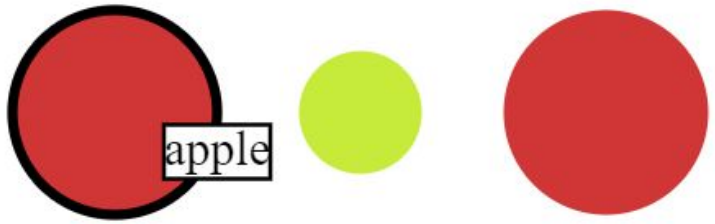


[Codepen]

Changing the size dynamically

- If we do the same thing with the height:

```
const tooltipSize = tooltipText.node().getBBox();
const tooltipWidth = tooltipSize.width;
const tooltipHeight = tooltipSize.height;
tooltipBox
  .attr('width', tooltipWidth)
  .attr('height', tooltipHeight);
tooltipText
  .attr('x', tooltipWidth/2)
  .attr('y', tooltipHeight/2 + tooltipSize.height/4);
```



[Codepen]

- To create more space, we can increase the tooltip size:

```
const tooltipWidth = tooltipSize.width + 20;
const tooltipHeight = tooltipSize.height + 10;
```