Lab 5: D3 Selections & Grouping PreLab

Wijaya, Mario edited this page 3 days ago · 6 revisions

Learning Objectives

After completing this lab you will be able to:

- Use d3-nest to reformat tabular data
- Create groupings with d3 selections
- Use SVG transforms along with `<g>` elements to create a visualization structure
- Understand the concept of d3 selections and data joins

Prerequisites

- Update your *starter code repository* using one of the following methods:
  i. From the GitHub Desktop app click `Sync` on the top right
  ii. Open a command line prompt. Navigate to the repository directory, for example `cd ~\Development\CS4460-Spring2018\Labs` and run command `git pull`.
- You have **read How Selections Work** by Mike Bostock

Additional Reading

- Nested Selections by Mike Bostock
- Manipulating data like a boss with d3 by Jerome Cukier
- D3, Conceptually. Lesson 2: Charts by Mikey Levine
- D3, Conceptually. Lesson 3: (Moderately) Advanced Data by Mikey Levine
- Advanced D3: More on selections and data, scales, axis by A. Lex of U. of Utah

SVG Refresher

The following section is a brief refresher on **SVG groups and transforms**. If you feel comfortable with these concepts, feel free to skip to the next section.

Grouping

The 'g' in `<g>` stands for 'group'. The group element is used for logically grouping together sets of related graphical elements. The `<g>` element groups all of its descendants into one group. Any styles you apply to the `<g>` element will also be applied to all of its descendants. This makes it easy to add styles, transformations, interactivity, and even animations to entire groups of objects.

We have already used groups for axes, but no we will start to use them to arrange and manage the visual marks of our visualizations.

Transforms

The transform attribute is used to specify one or more transformations on an element. It takes a `<transform-list>` as a value which is defined as a list of transform definitions, which are applied in the order provided. The individual transform definitions are separated by whitespace and/or a comma. An example of applying a transformation to an element may look like the following:

```
<g transform="translate(20, 20) rotate(40) translate(10)"></g>
```


Clone this wiki locally

`https://github.gatech.edu`

This will transform the group:

- 20 pixels in the x-direction and 20 pixels in the y-direction
- rotate the group 40 degrees clockwise
- 10 pixels along the 40 degree line

Translate

To translate an SVG element, you can use the `translate()` function. The syntax for the translation function is:

```
translate(<tx>, [<ty>])
```

The `translate()` function takes one or two values which specify the horizontal and vertical translation values, respectively. `tx` represents the translation value along the x-axis; `ty` represents the translation value along the y-axis.

D3 Nest

Creating visualizations requires you to work with tabular data a lot. Sometimes you will need to aggregate or re-configure the data based on nominal, ordinal or even quantitative data attributes for visualization. `d3.nest()` helps with this.

What does d3-nest do? d3-nest turns a flat array of objects, which thanks to `d3.csv()` is a very easily available format, in an array of arrays with the hierarchy you need.

With the `.key()` method, we are indicating what we will be using to create the hierarchy. In the example below we want to group the data by sector. Here, we just have one level of grouping, but we could have several by chaining several `.key()` methods.

The last part of the statement, `.entries()`, says that we want to receive a key-value array from the nesting operation. `.object()` on the other hand would return a JavaScript object map.

In the example below we want to compute the total stock price. It would be nice if we could characterize some aggregate information for our nested data? This is a job for the `rollup` function. Rollup is the aggregating function for d3-nest.

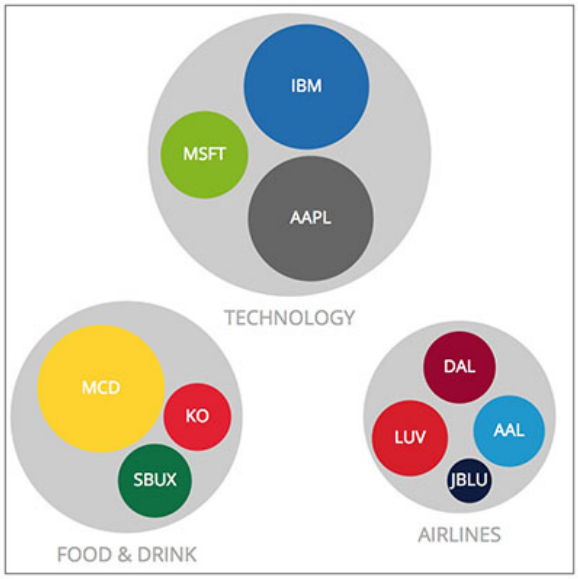
An Example

This week we will be working with stock prices of publicly traded companies. We have a list of companies, their current stock price, their stock ticker name, and the sector of each company. Here is a snippet of the data:

company	price	sector	color
MSFT	77.74	Technology	`#85b623`
IBM	159.48	Technology	`#236cb0`
SBUX	55.24	Food & Drink	`#0e7042`
DAL	52.88	Airlines	`#980732`

This data has already been stored in a variable called `stockData1` in the starter code provided for this example.

Our goal is to create a set visualization of this dataset using nested circles:



This type of enclosure hierarchy diagram is called circle-packing. They are similar to Euler Diagrams, however circle packing is not suited for set unions like an Euler.

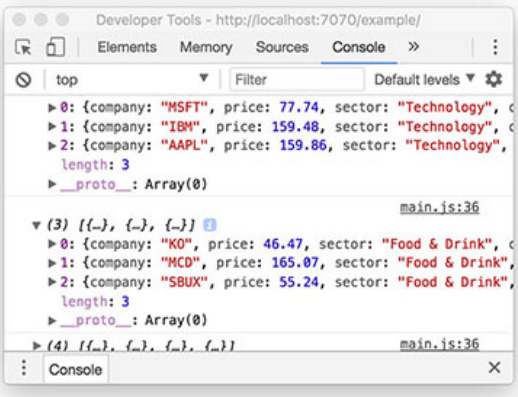
To create this type of visualization we need to mold and re-configure our data into a form that matches our desired visualization. In this case we need a hierarchical structure with the following traits:

1. A list of top-level objects for each `sector` category
2. Low-level objects for each `company` that includes the `price` and `color`
3. Top-level objects contain a list of low-level objects

Enter `d3.nest`. We will use d3 nest to get the data into the above form.

```
var nested = d3.nest()
    .key(function(c) {
        // Returns 'Technology', 'Food & Drink', or 'Airlines'
        return c.sector;
    })
    .rollup(function(leaves) {
        console.log(leaves);
    })
    .entries(stockData1); // entries returns a key-value array
```

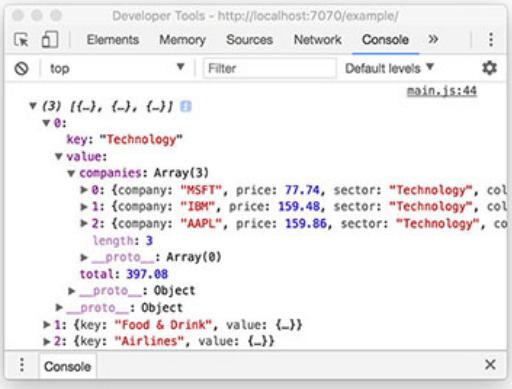
Logging the input to the `rollup` anonymous function, we can see how the data has been nested for each `sector` key. The leaves are a list of companies with that same `sector` attribute (every sector gets its own set of leaves):



Now by returning a value for the `rollup` function we can create a hierarchy structure. The anonymous function you pass to `rollup` can return an object, a number, a boolean, anything really. In this case we want to create a new object:

```
    .rollup(function(leaves) {
        // Compute the total price for the companies in this sector
        var totalPrice = d3.sum(leaves, function(c){
            return c.price;
        });
        // Return a new object for each sector
        return {total: totalPrice, companies: leaves};
    })
    .entries(stockData1);
```

This gives us the following results, which we can use for our circle diagram:



Part 2: Grouping Elements by Data

Note: You should comment out the Part 1 code (start and end specified in the starter code) for this part.

In many cases we want to apply data not directly to low-level SVG elements, but instead use a hierarchy of elements. For our company circle diagram, for example, we are going to need to layout the sector circles and the company circles. There are two approaches to doing this:

1. Laying out the sector circles first, and then the company circle elements independently so that they appear on top of each other. **OR**
2. Using a group element to group the sectors with the companies that belong to that sector.

The latter is the better approach, as we can use groupings to position elements relative to their related elements in the hierarchy. This makes a big difference for hierarchy diagrams, where the elements can be `translated` together and positioned relative to a parent group.

We could have used the `var nested` for this part as well, but for convenience, we have added `pos` arrays for each element's *relative* position in the canvas. The dataset now looks like this:

```
var stockData = [
    {key: 'TECHNOLOGY', pos: [200, 105], value: {
        total: 397.08,  companies: [
            {company: "MSFT", price: 77.74, pos: [-60, 0], color: '#85b623'},
            {company: "IBM", price: 159.48, pos: [12, -48], color: '#236cb0'},
            {company: "AAPL", price: 159.86, pos: [15, 45], color: '#666666'}
        ]}
    },
    {key: 'FOOD & DRINK', pos: [85, 290], value: {
```

```
            total: 266.78, companies: [
                {company: "KO", price: 46.47, pos: [50, 0], color: '#e32232'},
                {company: "MCD", price: 165.07, pos: [-18, -20], color: '#fed430'},
                {company: "SBUX", price: 55.24, pos: [20, 45], color: '#0e7042'}
            ]}
        },
        {key: 'AIRLINES', pos: [320, 290], value: {
            total: 183.51, companies: [
                {company: "DAL", price: 52.88, pos: [0, -35], color: '#980732'},
                {company: "AAL", price: 51.95, pos: [35, 10], color: '#1f98ce'},
                {company: "JBLU", price: 20.08, pos: [7, 45], color: '#101e40'},
                {company: "LUV", price: 58.60, pos: [-35, 15], color: '#d81f2a'}
            ]}
        }
    ];
```

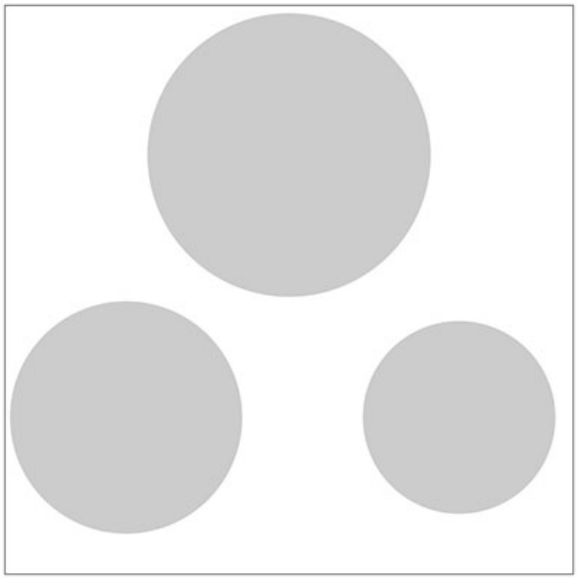
Now for our d3 code, first we will add the `<g>` element for each sector and `translate` them:

```
var sectorG = svg.selectAll('.sector')
    .data(stockData)
    .enter()
    .append('g')
    .attr('class', 'sector')
    .attr('transform', function(d) {
        return 'translate('+d.pos+')';
    });
```

Notice, that we declare a variable for the `sectorG` d3-selection. All of these `g` elements are data-bound to the high-level `sector` objects. We can now add a `circle` element to each of the 3 sectors:

```
sectorG.append('circle')
    .attr('r', function(d) {
        return rSectorScale(d.value.total);
    }).style('fill', '#ccc');
```

From this we now have 3 circles positioned on the canvas:



Notice that we don't have to set the `cx` or `cy` position of the circle because it is handled by the parent group's translation.

All of this should look familiar so far. However, now we are going to add in a completely new concept - **nesting elements with data**. In the following code we will make a new selection for `.company` and make a data binding for each sector's list of companies:

```
var companyG = sectorG.selectAll('.company')
    .data(function(d) {
        return d.value.companies;
    })
    .enter()
    .append('g')
    .attr('class', 'company')
    .attr('transform', function(d) {
        return 'translate('+d.pos+')';
    });
```

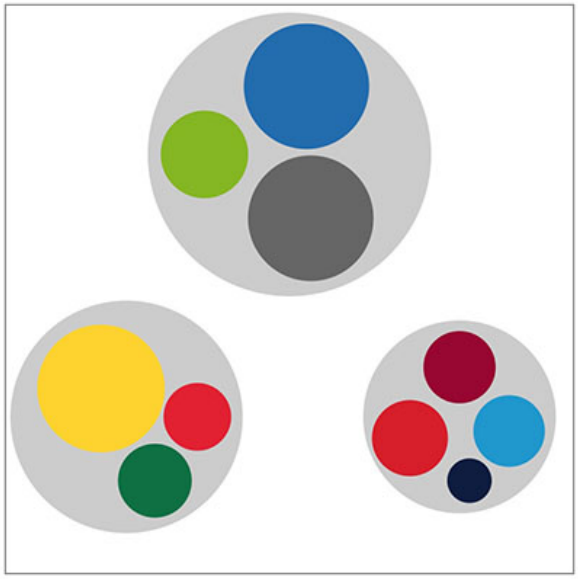
Notice that the data method can either take an array or a function that returns an array. Arrays are often used with flat selections, since flat selections only have one group, while nested selections typically require a function.

Nesting selections has another subtle yet critical side-effect: it sets the parent node for each group. The parent node is a hidden property on selections that determines where to append entering elements. In our case the 3 `.sector` groups are the parent nodes. We will explain parent nodes in more depth later on.

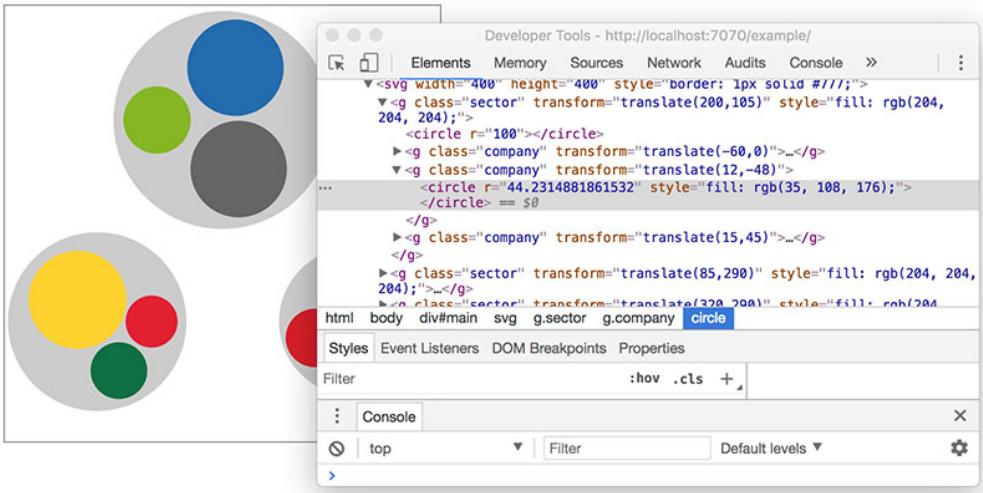
If we append to our new `companyG` selection, the elements will be added to each of the 10 `.company` groups:

```
companyG.append('circle')
    .attr('r', function(d) {
        return rScale(d.price);
    })
    .style('fill', function(d) {
        return d.color;
    });
```

This is because the previous `.append('g').attr('class', 'company')` changes the parent node for that selection. Anything appended after will be appended once for each element in the new selection. Here's the result:



And if we inspect the element we can see the hierarchy that we have created for this circle diagram:

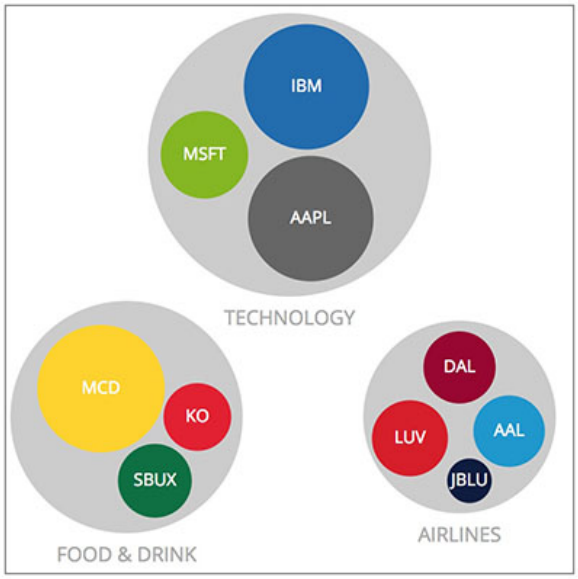


Notice that with grouping, all of the `translate` definitions are relative. The inner `.company` groups are positioned relative to the center of the parent circle. This makes positioning a lot easier! For example when we add text, we don't need to do much to center the text in legible positions:

```
sectorG.append('text')
    .text(function(d) {
        return d.key;
    })
    .attr('y', function(d) {
        return radiusScale(d.value.total) + 16;
    });

companyG.append('text')
    .text(function(d) {
        return d.company;
    });
```

And with the added text we get our final product:



Home