CS4460 / **Spring23-Labs-PUBLIC** Public

Code Issues Pull requests Actions Projects **Wiki** Security Insights

Lab 4: D3 Selections and Grouping

Jump to bottom

Alex Endert edited this page on Feb 20 · 1 revision

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

- Download the corresponding lab from the code repo (either using git or downloading the folder from the code of this repo (in the Code tab above))
- You have **read How Selections Work** by Mike Bostock

Recommended Reading

- Nested Selections by Mike Bostock
- Manipulating data like a boss with d3 by Jerome Cukier

Additional Reading

- 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

What to submit

1. You should have completed Activity 1, Activity 2, and Activity 3 (in each respective subfolder).
2. Rename your `lab4` folder to `LastName_FirstName_lab4`
3. Zip up `LastName_FirstName_lab4` as `LastName_FirstName_lab4.zip` and submit it to Canvas.

Grading

Your assignment will be graded on the following requirements:

- Functionality of Activity 1, 2, and 3 completed

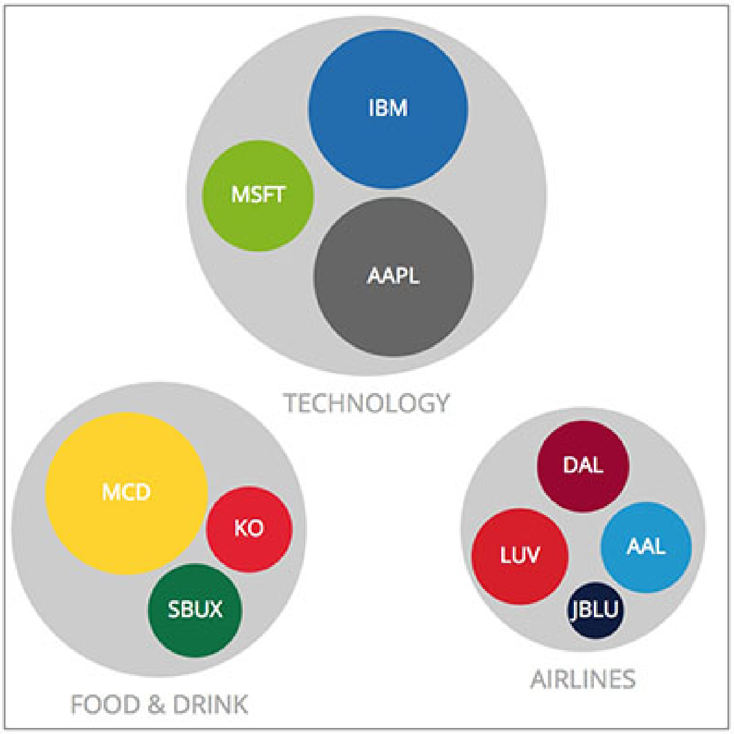
Tutorial 1: D3 Nest

This week we will be working with stock prices of publicly traded companies. You can find the materials for tutorials 1 - 3 in `/lab4/tutorials` .

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`

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 the `d3.group`, `d3.rollup` and `d3.index` functions. We will use d3 rollup to get the data into the above form.

Overview

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.rollup()` helps with this.

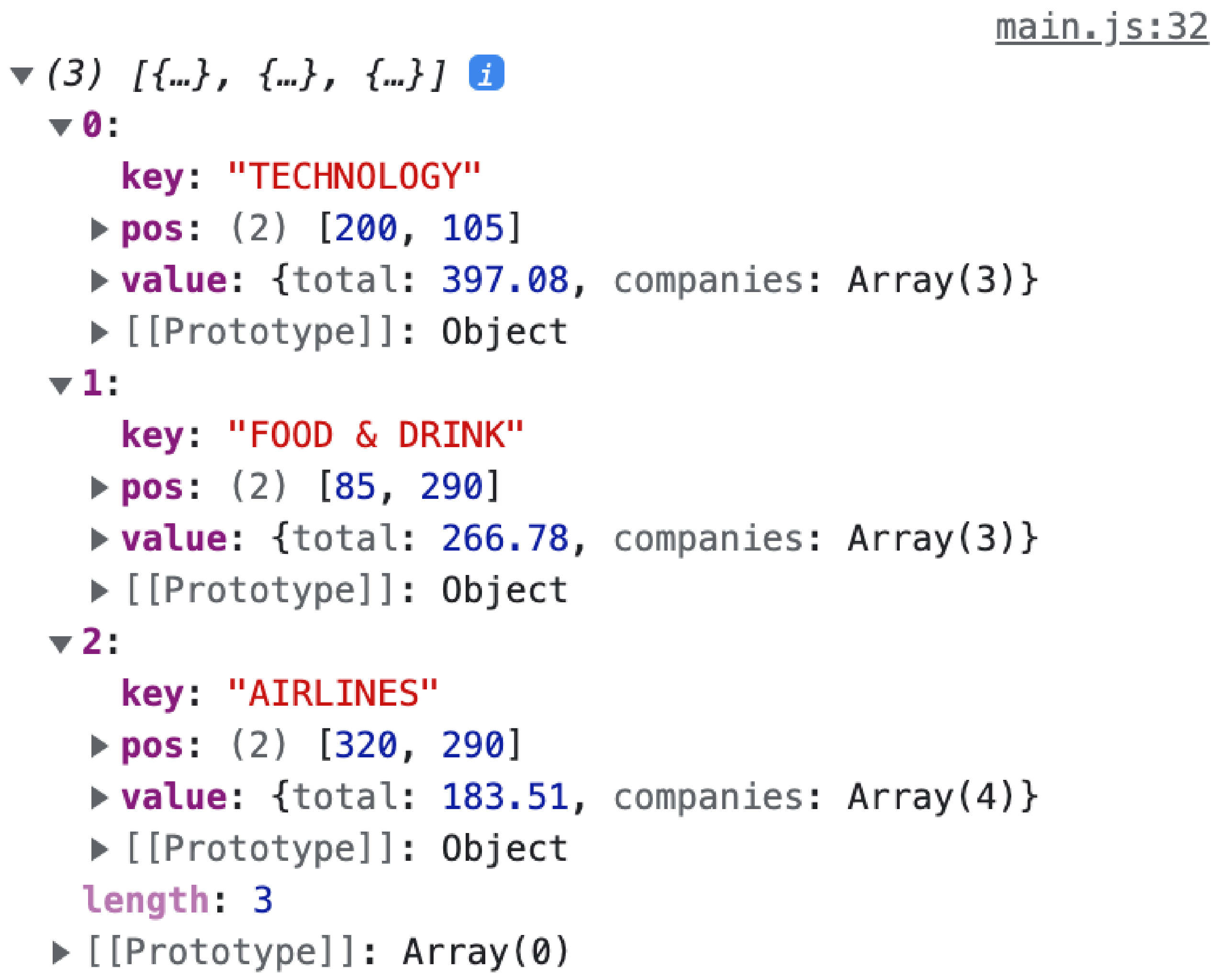
What does d3-rollup do? d3-rollup turns a flat array of objects, which thanks to `d3.csv()` is a very easily available format, into an array of arrays with the hierarchy you need.

If the above text still leaves you puzzled, we strongly recommend reading this excellent tutorial on Observable.

OK, let's take a look at some code using `d3.rollup`.

```
var nested = d3.rollup(stockData,
    v => { return { values: { companies: v, total: d3.sum(v, d => d.price) } }; },
    d => { return { key: d.sector } },
>);
```

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:



```
                                                        main.js:32
▼ (3) [{…}, {…}, {…}] i
  ▼ 0:
      key: "TECHNOLOGY"
    ▶ pos: (2) [200, 105]
    ▶ value: {total: 397.08, companies: Array(3)}
    ▶ [[Prototype]]: Object
  ▼ 1:
      key: "FOOD & DRINK"
    ▶ pos: (2) [85, 290]
    ▶ value: {total: 266.78, companies: Array(3)}
    ▶ [[Prototype]]: Object
  ▼ 2:
      key: "AIRLINES"
    ▶ pos: (2) [320, 290]
    ▶ value: {total: 183.51, companies: Array(4)}
    ▶ [[Prototype]]: Object
    length: 3
  ▶ [[Prototype]]: Array(0)
```

We can use this result in our circle diagram.

We are now going to use `d3.group()` in the following activity to re-configure our dataset.

Activity 1: Nesting a Dataset

> Reminder: Start an http server for this lab's directory (e.g. cd `/lab4/activities`). From command line call `python -m SimpleHTTPServer 8080` (for Python 2) or `python -m http.server 8080` (for Python 3).

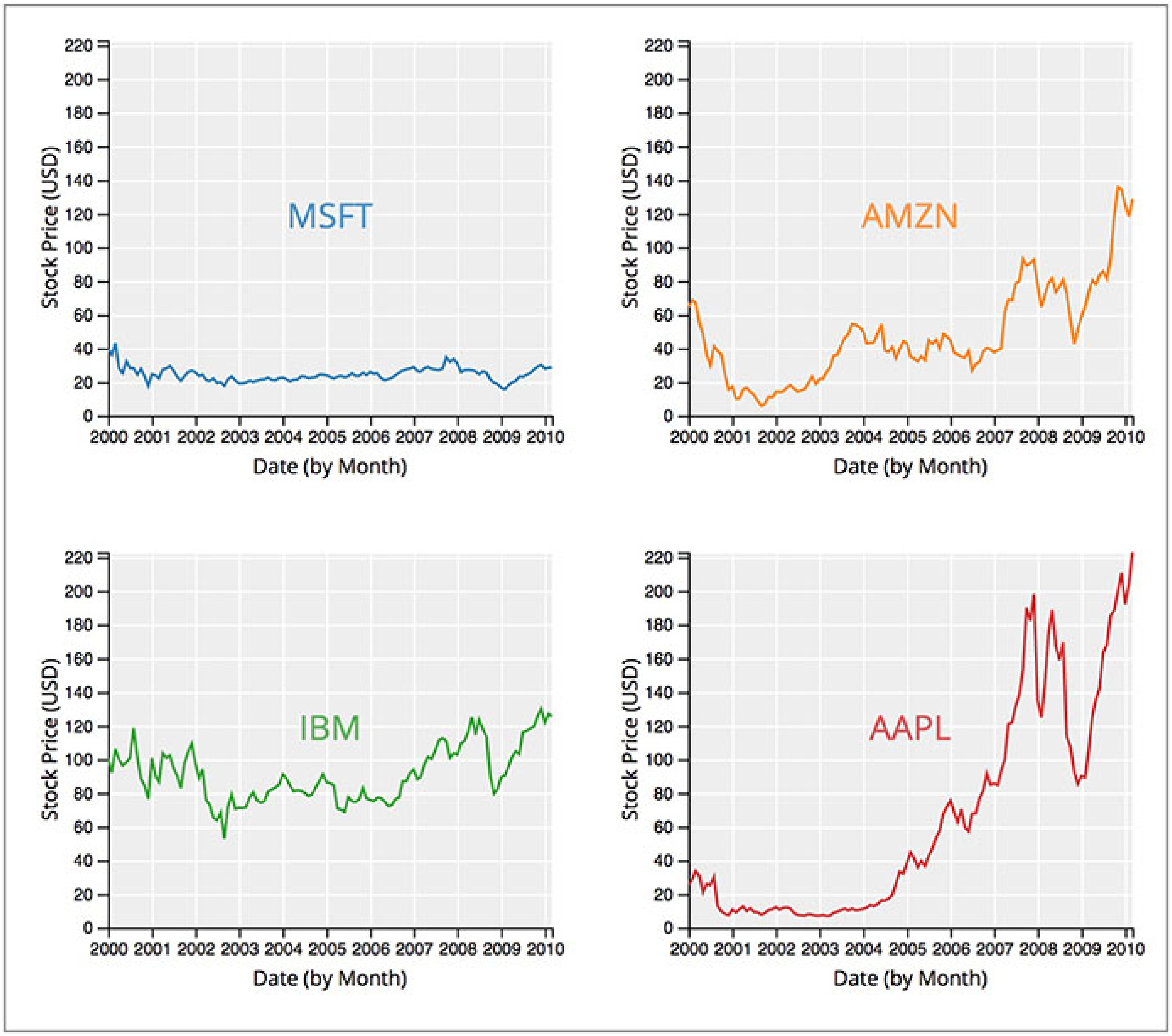
For activities 1 - 3, you will be working with the same HTML/CSS/JavaScript code. All of it can be found in `/lab4/activities` . At the end of this activity, rename the `activities` folder as `activity_1` . Duplicate this folder and name it as `activity_2` and start there for the next activity.

You will be working with the `stock_prices.csv` dataset. The dataset includes 492 rows. Each row corresponds to the closing stock price at the end of each month for a company. Here is a snippet of the data table:

company	date	price
MSFT	Jan 2000	39.81

company	date	price
MSFT	Feb 2000	36.35
AMZN	Jun 2006	38.68
AAPL	Jul 2009	163.39

You will be working toward creating the following trellis line chart today:



For this activity you will use the D3 you have learned so far - e.g. d3 data-binding, axes, scales, and appending new elements. You are also going to get familiar with creating grouped SVG elements with nested data. First though, you will need to learn how to nest your dataset and to structure your code to layout trellis subplots in a grid.

1. How to structure your d3 code

We have already added structure to your starter code in `stock_prices.js` in the folder trellis_lines. The variable declarations at the top help to compute pixel-space values for laying out a 2x2 trellis plot grid.

Take a second now to follow along in the comments of the code so that you understand the purpose of the variables.

You can then see them in action as we add 4 rectangles that will be the backgrounds for your trellis plots.

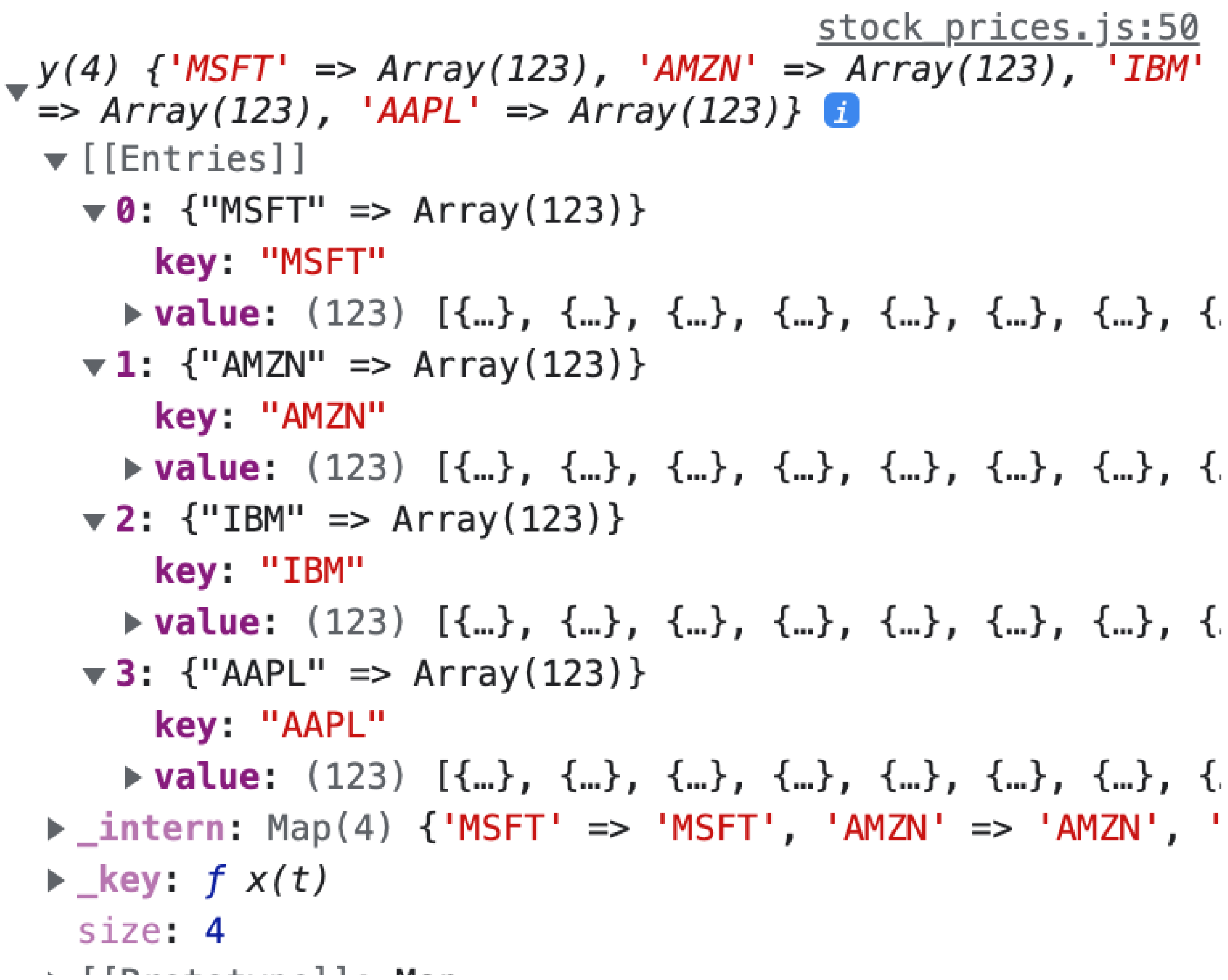
2. Parse the dates

With our dataset loaded, we will need to convert the `string` date attributes into actual JavaScript `Date` objects. `Date` objects refer to a specific date and time, they can be used by D3 to create a `d3.scaleTime()`. So in order to parse the dates you need to loop through the entire dataset, and parse each date string with the `parseDate` function that we have already added for you, and then declare the `Date` object. **Hint:** for each data entry `d` of the dataset, declare `d.date = parseDate(d.date)`.

3. Nest the loaded dataset

Now we need to nest the dataset to prepare it for visualizing. Our trellis plot varies on the `company` data attribute, so we want our data to follow suit. Let's call a `d3.group()` function where we return the `company` property for each data object.

`console.log()` your nesting result, and you should get this:



```
                                                       stock_prices.js:50
▼ y(4) {'MSFT' => Array(123), 'AMZN' => Array(123), 'IBM'
  => Array(123), 'AAPL' => Array(123)}
  ▼ [[Entries]]
    ▼ 0: {"MSFT" => Array(123)}
        key: "MSFT"
      ▶ value: (123) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {
    ▼ 1: {"AMZN" => Array(123)}
        key: "AMZN"
      ▶ value: (123) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {
    ▼ 2: {"IBM" => Array(123)}
        key: "IBM"
      ▶ value: (123) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {
    ▼ 3: {"AAPL" => Array(123)}
        key: "AAPL"
      ▶ value: (123) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {
  ▶ _intern: Map(4) {'MSFT' => 'MSFT', 'AMZN' => 'AMZN', '
  ▶ _key: f x(t)
    size: 4
```

Take a screenshot of your browser that shows this information. Save it to your folder and include it in the submission. Please also include the code that allowed you to log the information to the console.

At this point, rename the `activities` folder as `activity_1`. Duplicate this folder and name it as `activity_2`. You will start there for the next activity.

Refresher: SVG Groups and Transforms

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 now 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>
```

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.

Tutorial 2: Grouping Elements by Data

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, in the `example` folder, 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.
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 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'}
        ]}
    }
];
```

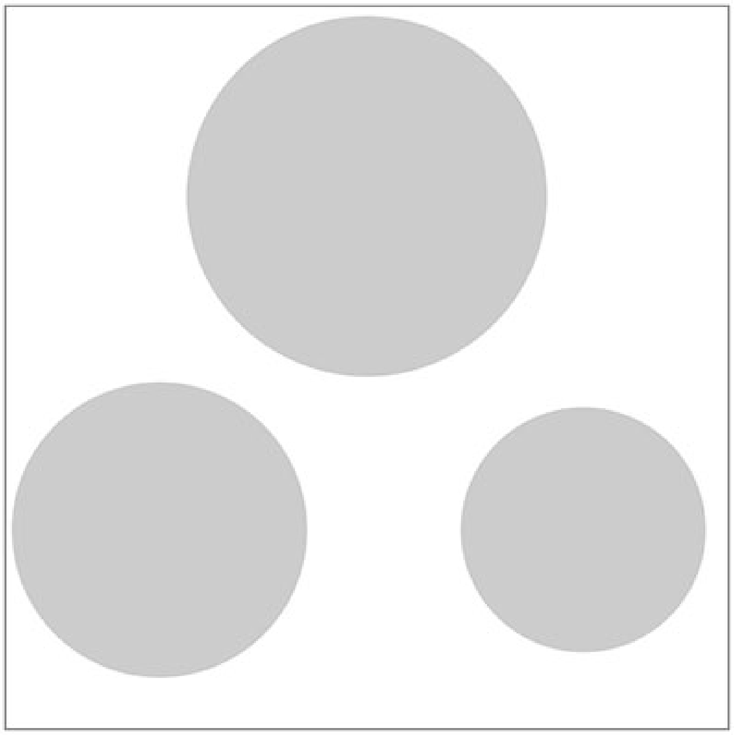
Now for our d3 code, first we will add the `<g>` element for each sector and `translate` them:

```
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 radiusScale(d.value.total);
    }).style('fill', '#ccc');
```

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



Notice that, similar to what you did in Lab 3, 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**. Next, 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.

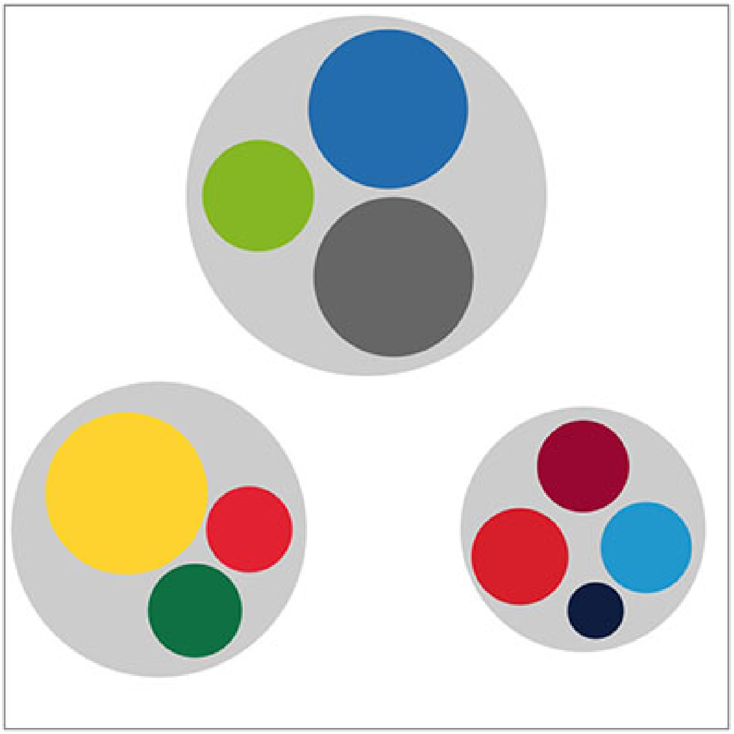
Nesting selections have another subtle yet critical side-effect: it sets the parent node for each group. The parent node is a hidden property on selections that determines where to append entering elements. In our case, the 3 `.sector` groups are the parent nodes. We will explain parent nodes in more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

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

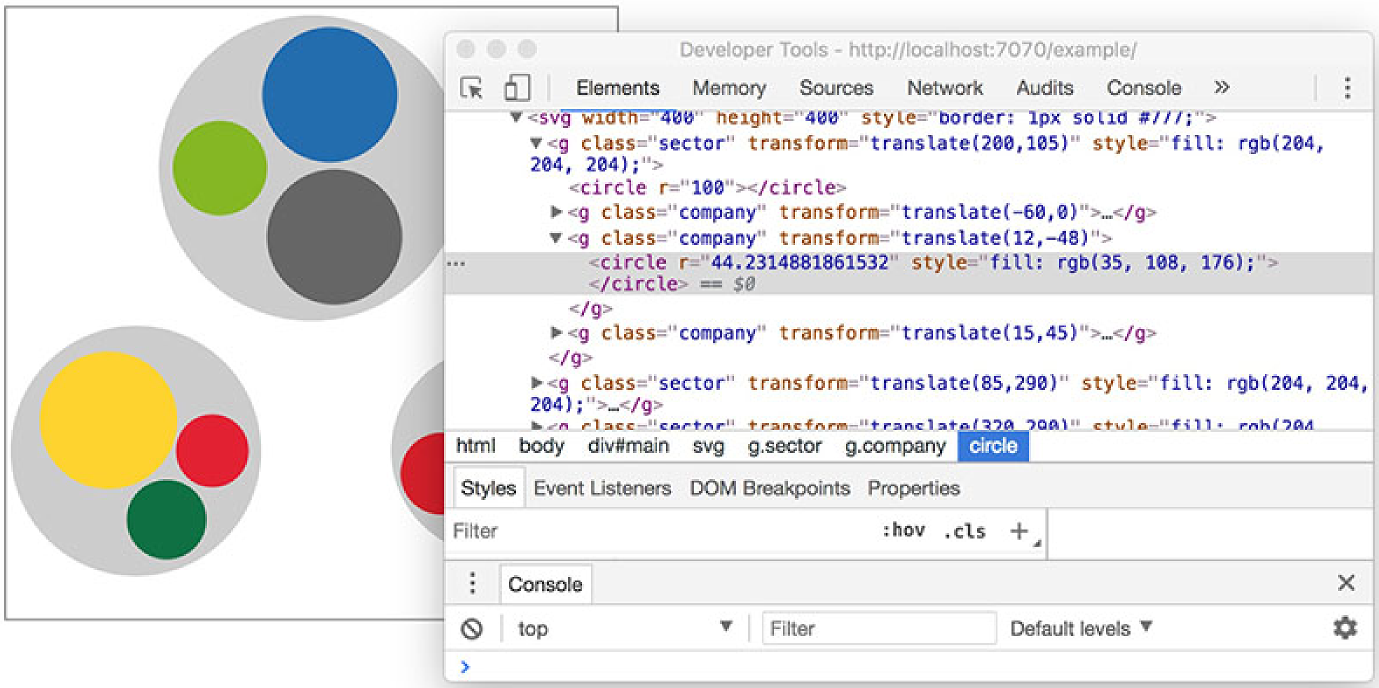
```
companyG.append('circle')
    .attr('r', function(d) {
        return rScale(d.price);
    })
    .style('fill', function(d) {
        return d.color;
    });
```

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 is what the result should look like:



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


Developer Tools - http://localhost:7070/example/
Elements
Memory
Sources
Network
Audits
Console
»

```
▼<svg width="400" height="400" style="border: 1px solid #777;">
  ▼<g class="sector" transform="translate(200,105)" style="fill: rgb(204,
  204, 204);">
      <circle r="100"></circle>
    ▶<g class="company" transform="translate(-60,0)">…</g>
    ▼<g class="company" transform="translate(12,-48)">
        <circle r="44.2314881861532" style="fill: rgb(35, 108, 176);">
        </circle> == $0
      </g>
    ▶<g class="company" transform="translate(15,45)">…</g>
    </g>
  ▶<g class="sector" transform="translate(85,290)" style="fill: rgb(204, 204,
  204);">…</g>
```

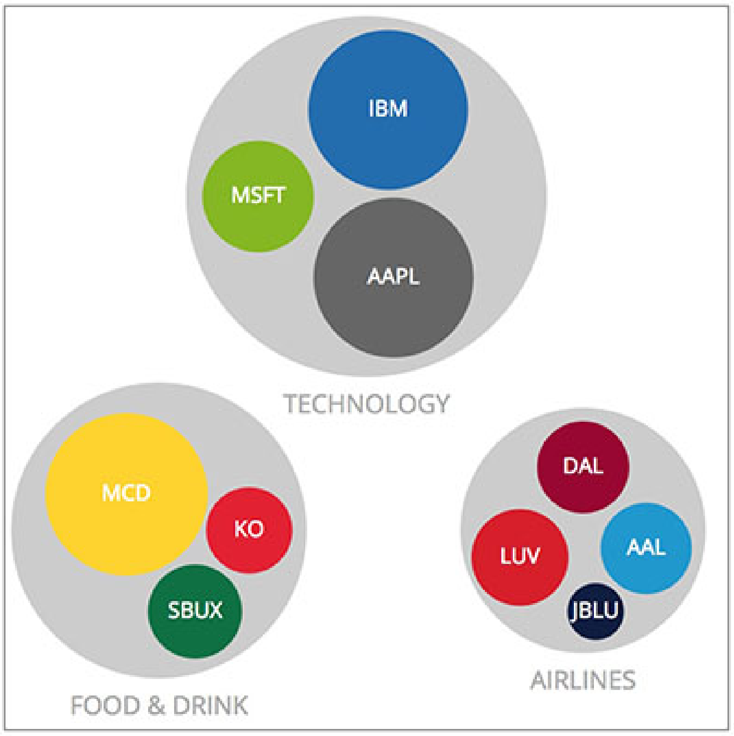
html body div#main svg g.sector g.company circle
Styles Event Listeners DOM Breakpoints Properties
Filter :hov .cls +
Console
top
Filter
Default levels

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, our final visualization should look like this:



We will now use nesting visual elements with data to create our trellis line chart of the stock data.

Activity 2: Grouping a Trellis Line Chart

> Reminder: Start from the `activity_2` folder that you duplicated from Activity 1

For this activity, you will create a Trellis Line Chart.

It is important when learning something new that you work incrementally. Start with creating one element, check that it appears in the canvas (remember DOM inspection is the best way to debug). If the element wasn't correctly added you only have that one block of code to debug. If you add a lot of code at once, it's much harder to find what went wrong.

The steps below are broken up in a logical way to allow you to do this. However, not all of them have specific endpoints that you need to turn in. Only the final visualization is what is graded.

1. Append trellis groupings

First we want to append a group for each trellis subplot. You should have already created a nested dataset earlier in the lab. The nested dataset should be a list of 4 objects for each company, each object should have a `values` list of the monthly stock prices for that company. Please use that data to create 4 `.trellis` groups. **Hint:** similar to the way how you append a `<g>` element for each data case in Lab 3 Activity 3.

Then, we will want to position each of these groups in front of the rectangle backgrounds. To do this, you will use the same `.attr('transform', function(d,i) {...` callback function used when creating the background rectangles.

> This code uses the indices of the 4 appended groups to position them. The math `(i % 2)` and `Math.floor(i/2)` takes the indices `[0,1,2,3]` and converts them to position indices `[[0,0], [1,0], [0,1], [1,1]]`. We can then multiply those indices by a constant factor `width + padding` to get the pixel space position.

Check to make sure your groups were translated correctly using DOM inspection.

2. Create scales for our line charts

Next we are going to append a `path` element for each trellis subplot. But before we do that - we need to create scales to convert the stock prices and dates into trellis pixel space. We have already provided the data `domain` s for both the `price` and `date` attributes.

Hints:

- You need to create a `xScale` with `d3.scaleTime()` (for `date`) and a `yScale` with `d3.scaleLinear()` (for `price`).
- The range for the `xScale` and the the `yScale` would be `[0, trellisWidth]` and `[trellisHeight, 0]` .

Notice, that the `range` for the `yScale` has been flipped. This is because we want our line charts to grow upward. Also notice, that we only need to use 0 and the `trellisWidth` or `trellisHeight` for the range. We don't want to hard-code any padding in our scale, because the 4 groups that we just added will take care of positioning in the overall canvas space.

Next, define a line interpolator for the line chart with `d3.line()` . `d3.line()` is a function that generates a `d` path attribute for x and y data callbacks. **Hint:** you will use the `xScale` and the `yScale` to set the path attributes x and y.

We will use this line interpolator function in the following section to define the x and y points of the lines.

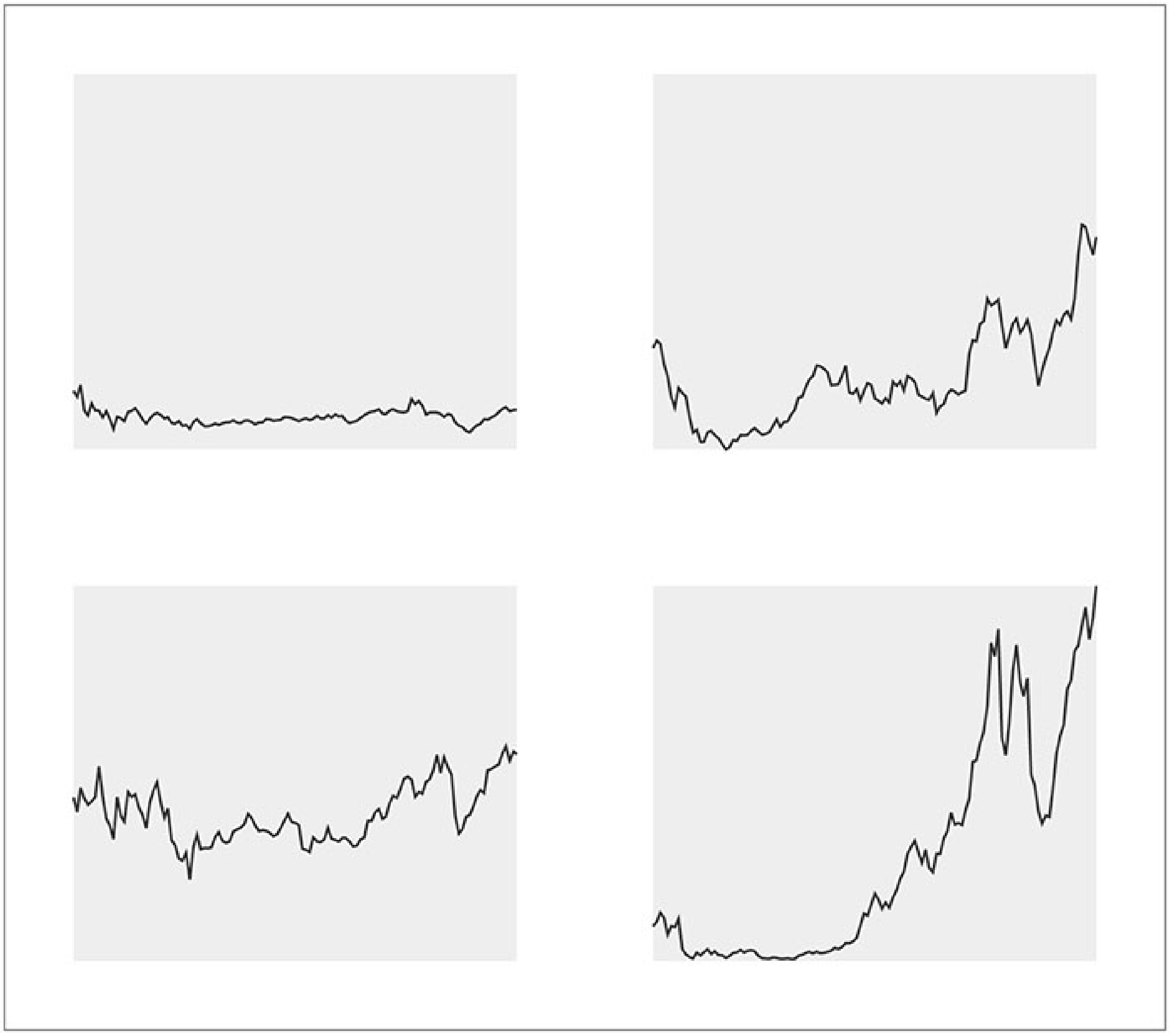
3. Create the line chart

Finally we can add our lines! To do this we will need to nest the `values` array for each company into a new `path` element. The `path` element will also be appended to its respective parent group element.

Hints:

- So for each group that you append to the svg in step 1, you need to append a `path` element to it. Apply the class `line-plot` to the new `path` and select all the `path` s by this class name instead of `path` for data binding.
- The data you want to bind to the new `path` element should be a new array that contains the array of `values` . This is because we only want to append one `path` element. So we want to return an array with one array element in it.
- You are then going to use all of the `values` to populate the position of the `path` . Use `.attr('d', lineInterpolate)` to handle all of this. You should already create the `lineInterpolate` function with `d3.line()` . The `lineInterpolate` will specify the x and y locations for the path based on the data bound to it, and the rules we defined when we declared `lineInterpolate` .
- Style the `path` to make its `stroke` as `#333` .

Here is what the result should look like at this point:



4. Create axes for each subplot

At this point, it's a good idea to add appropriate axes to your visualization. **Hint:** you will want to continue using your group selection created in step 1 to add new elements, so that any elements that you append to this selection will be added to all 4 of your subplots.

Hints:

- Define a new `xAxis` with `d3.axisBottom(xScale)` and a new `yAxis` with `d3.axisLeft(yScale)`.
- Append the `xAxis` to the group selection. Apply the classes `x axis` to it and translate the `xAxis` by `trellisHeight` in the y-axis direction.
- Append the `yAxis` to the group selection. Apply the classes `y axis` to it.

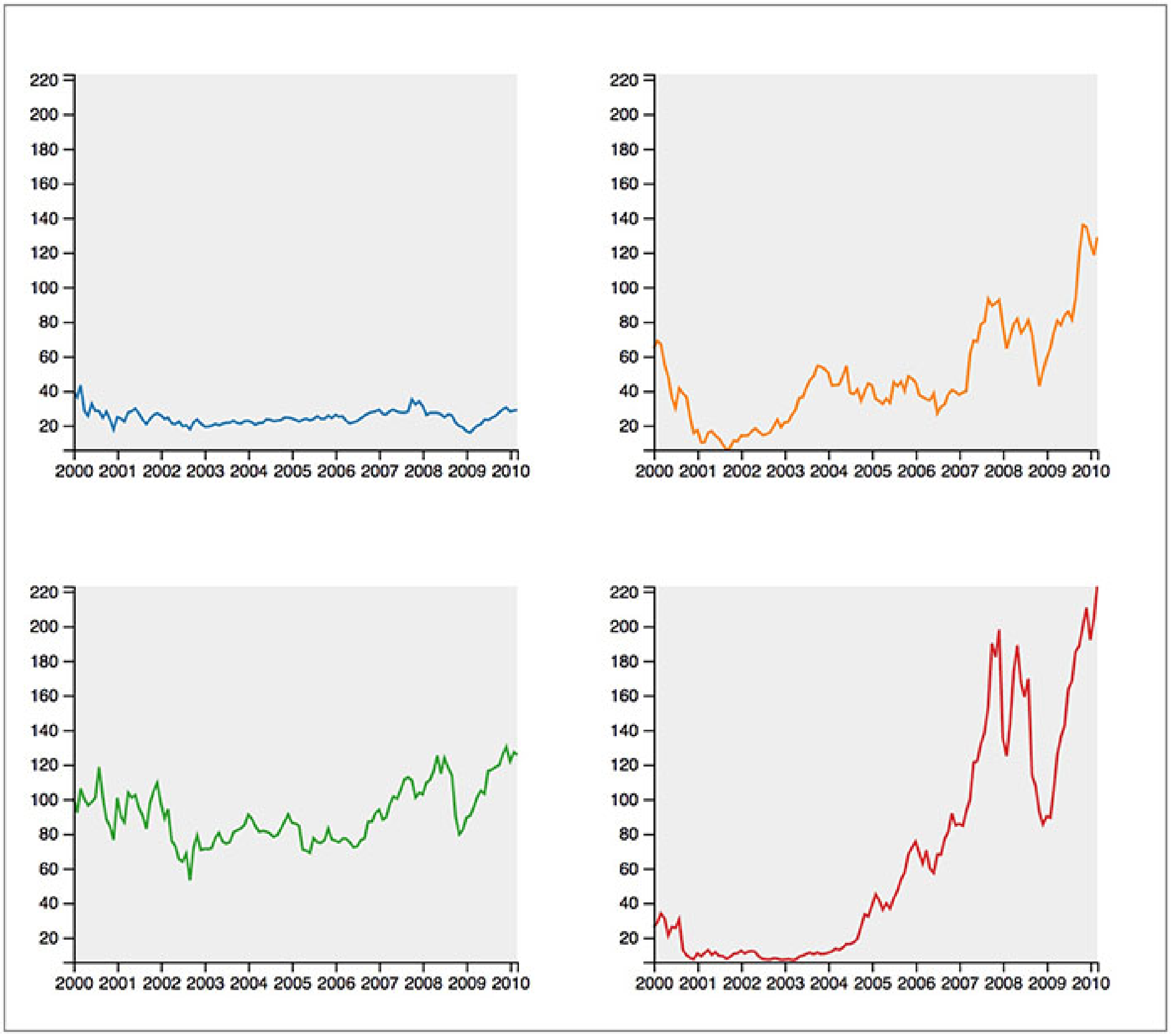
> Notice that the `translate` positions only need to be relative to the trellis pixel space! Because of relative positioning inside of a group we only need to position the axes at the left and bottom of each of the trellis subplots.

5. Add color

Finally, add some color to your chart. Use the `d3.scaleOrdinal` and the `d3.schemeCategory10` to create an automated categorical color mapping and name it as `colorScale`. D3 has a handful of pre-defined color mappings we can use. All we need to do is define an input `domain`. Which we do by using the array `map` function to generate a list of the `string` company names, the result looks like this `['MSFT', 'AAPL', 'IBM', 'AMZN']`. It is recommended that you look up examples of using these D3 functions.

We can now use our newly defined `colorScale` on our line charts to color each line. We do this by modifying the style we applied for each line in step 3.

Your result should look like this:



At this point, duplicate this folder and name it as `activity_3` . You will start there for the next activity.

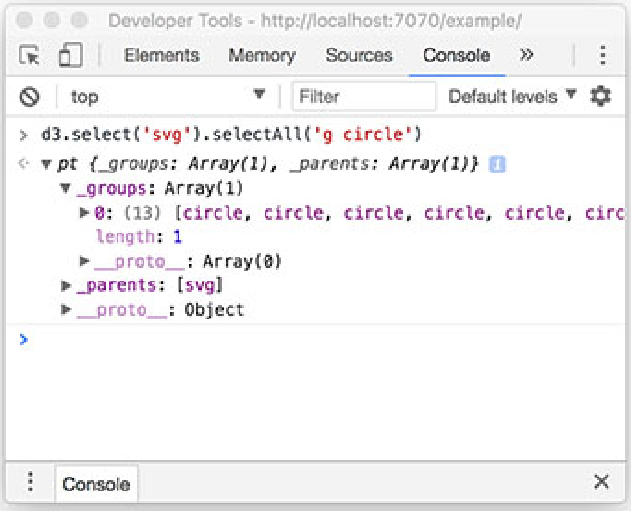
Tutorial 3: D3 Selections

We are going to revisit D3 selections. Before, we explained D3 Selections as a way to select objects on the canvas, but there is clearly more going on here than meets the eye. So let's take a look at the data structure that the D3 selection uses. For this section, we are going to continue to use our example circle diagram from before.

First off, we are going to open the Web Console and run the following command:

```
d3.select('svg').selectAll('g circle')
```

The console will return the D3 Selection that we just made. We can now see the structure of a D3 Selection:



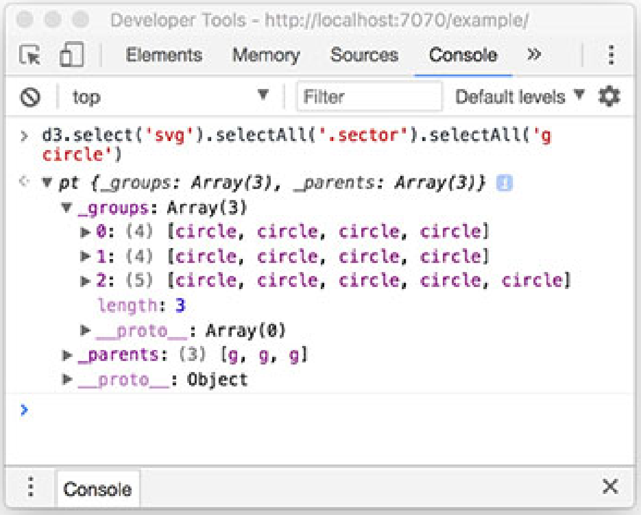
```
> d3.select('svg').selectAll('g circle')
< ▼ pt {_groups: Array(1), _parents: Array(1)}
   ▼ _groups: Array(1)
     ▶ 0: (13) [circle, circle, circle, circle, circle, circ
       length: 1
     ▶ __proto__: Array(0)
   ▶ _parents: [svg]
   ▶ __proto__: Object
>
```

A Selection object contains (among other things) a `_groups` array, which contains a `NodeList`. In the `NodeList` we can see all the matching objects (the `circle` elements). A selection is an array of groups, and each group is an array of elements.

You can also see that the Selection object has other fields, e.g., a list of `_parents`. Because we used `d3.select('svg')` we defined the single `_parents` element for this selection. When we use `.enter().append()` the new elements are appended to the `_parents`.

In the above example we selected all of the `circle`s in the diagram. What if we only want to select the `circle`s that represent the companies. We could do `d3.selectAll('.company')` because we already classed them, but say we want a reference to the parent `sector` group. We could do the following:

```
d3.select('svg').selectAll('.sector').selectAll('g circle')
```



```
> d3.select('svg').selectAll('.sector').selectAll('g
  circle')
< ▼ pt {_groups: Array(3), _parents: Array(3)}
   ▼ _groups: Array(3)
     ▶ 0: (4) [circle, circle, circle, circle]
     ▶ 1: (4) [circle, circle, circle, circle]
     ▶ 2: (5) [circle, circle, circle, circle, circle]
       length: 3
     ▶ __proto__: Array(0)
   ▶ _parents: (3) [g, g, g]
   ▶ __proto__: Object
>
```

From this example you can see that we know have 3 groups of `circle` elements selected in the `_groups` array, and we no longer have one element in the `_parents` array. With this type of selection we now have access to the `_parent` elements. This is important when doing grouping or nesting that you append to the correct `_parent` elements.

In the next lab, we will continue to work with D3 selections as we cover the *Enter, Update, Exit Pattern* that will allow you to *add, update, and remove* elements from the canvas.

Activity 3: Finishing Touches

> Reminder: Start from the `activity_3` folder that you duplicated from Activity 2

In this final activity, we will add grid lines to the chart, a text label for each trellis subplot, and label the axes. Again, let's do this incrementally:

1. Append company labels

We first want to append a text element for each trellis plot. Remember that we have a group selection (created in Activity 2 step 1) that we can use to append an element to each subplot. So let's use that to add a company label:

Hints:

- Apply the class `company-label` to the text label appended.
- Translate the text label to make it display in the center of each subplot.
- Color the text label with the same color as the line chart.

> Note that this appended `text` element still has a reference to the company data object. Please use the `.text(function(d) { ...` command to set the content.

2. Label the axes

Next we are going to label the axes for each chart. We can do this similar to how we added the company label.

Hints:

- Apply the classes `x axis-label` to the text label appended near the x-axis and translate it to make it display in the center of the x-axis but below the x-axis by 34. Set the content as `Date (by Month)` .
- Apply the classes `y axis-label` to the text label appended near the y-axis and translate it to make it display in the center of the y-axis but left to the y-axis by 30. Set the content as `Stock Price (USD)` .

> Remember that when we append the text labels to the group selection, the coordinate space is relative to each of the trellis subplots. So we only need to position the text relative to that space.

3. Add grids

Adding grids in D3, is a bit of a hack. Even Mike Bostock uses this method though, so who are we to protest. To add an axis we are going to create modified axes that have the following properties:

- Long ticks to extend across the chart
- No text on each tick
- No domain path

To achieve all of this we only need to set the `tickSize` and `tickFormat` to special values:

Hints:

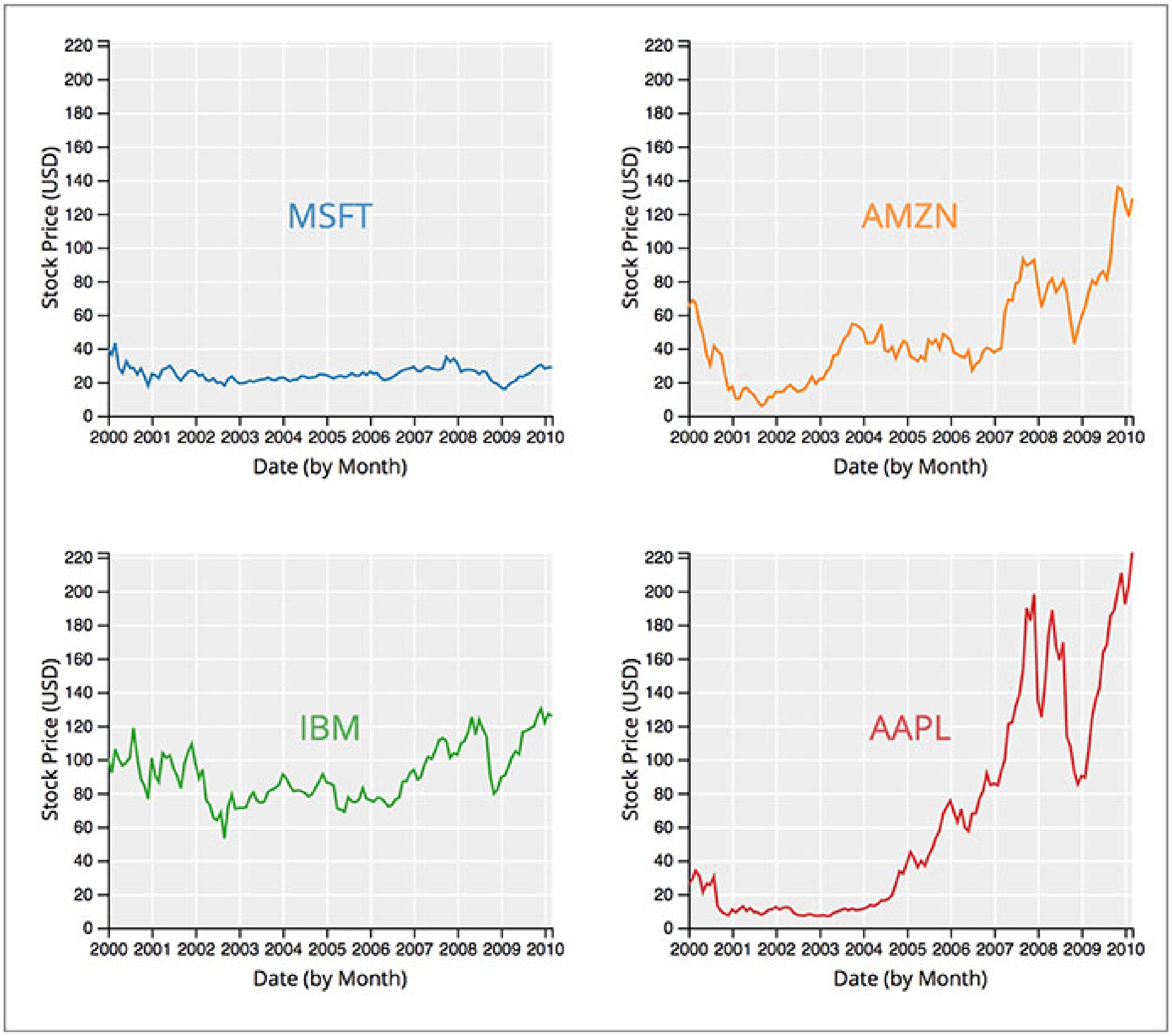
- Create the `xGrid` and the `yGrid` using the following code.
- Append both as special axes to the group selection and apply the class `x grid` to the `xGrid` and the class `y grid` to the `yGrid` .

Sample code:

```
var xGrid = d3.axisTop(xScale)
    .tickSize(-trellisHeight, 0, 0)
    .tickFormat('');

var yGrid = d3.axisLeft(yScale)
    .tickSize(-trellisWidth, 0, 0)
    .tickFormat('')
```

And with that final addition we have created our trellis line chart which should look like this:



At this point, you should have three subfolders, `activity_1` , `activity_2` , and `activity_3` , in your `lab4` folder.

This lab was based on the following material:

- Nested Selections by Mike Bostock
- Manipulating data like a boss with d3 by Jerome Cukier
- Advanced D3: More on selections and data, scales, axis by A. Lex of U. of Utah

Pages 11

Find a page…

Home

Lab 1: Intro to HTML, CSS, and SVG

▸ Lab 7 D: Interactive Visual Comparison

Clone this wiki locally

```
https://github.gatech.edu/CS4460/Spring23-Labs-PUBLIC.wiki.git
```