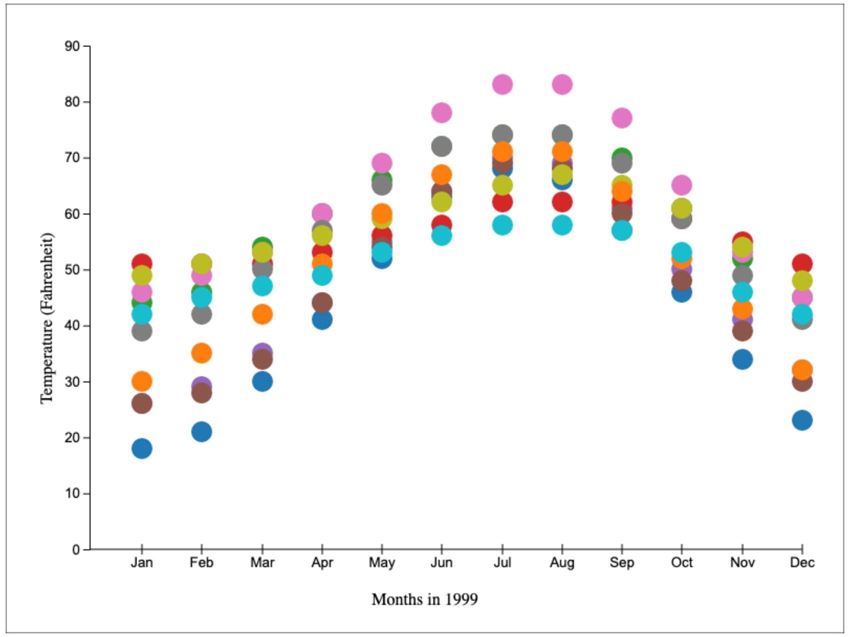
d3 concepts for heatmap-style visualizations

Class #9 – Fall 2022

rev. 2022.11.13.a

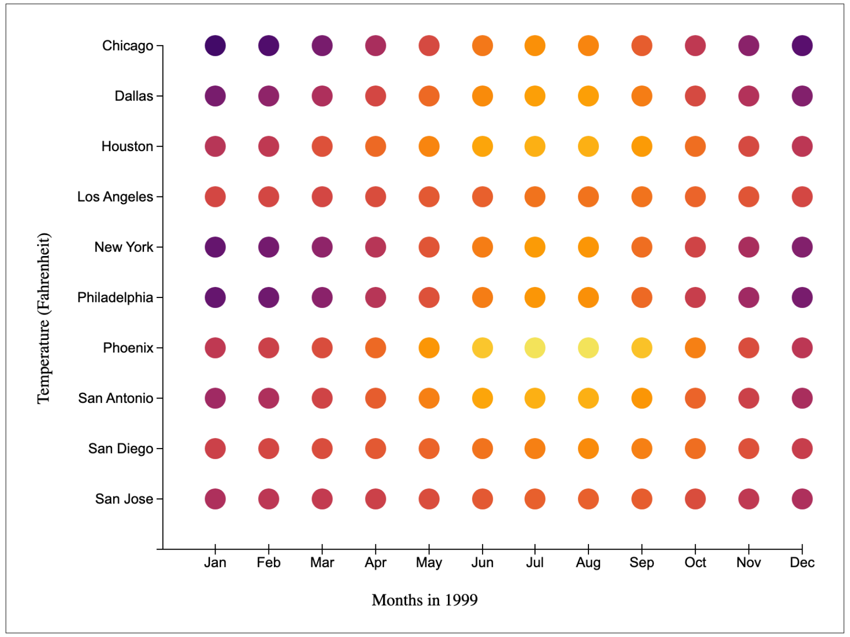
Our starting point & what's missing

- Open and test the project contained in **1_scalepoint_months_low_temps.zip**
- Visualization similar to comparison of 4 chicken feed diets from lab:
  - Discrete data (months) on X axis
  - Continuous data (temperatures) on Y axis
- Information for 10 cities shown
- What's missing?
- Do you think having this many colors is OK for this particular visualization? Why or why not?
- How could we improve this visualization's clarity?



Initial goal: reorganize the visualization

- Version we currently have is probably best if you're looking for trends in temperatures
- If our goal is instead to compare temperatures across the US, it might be more useful if we could see all of the cities and all of the temperatures at the same time
- This means we will have **discrete data** in both the X and Y axis, while our **continuous data** will be based on color or scale (color shown here)
  - X: months in 1999
  - Y: cities where temperatures measured
- Do you think this new version that we're planning would be as good for the given hypothesis?



REVIEW: d3.scalePoint()

- Provides a scale for **discrete** values
- Domain is an Array of values that will be used on this scale
  - Example: **.domain(["one", "two", "three])**
  - *NOTE: Unlike **scaleLinear()** and other continuous scales, **scalePoint()** only works with the values in the Array. It does not interpolate additional values.*
- Range is Array containing at least min and max values to be output using the scale
  - For our X axis scale, we use [**leftMargin, svgWidth – rightMargin]**
  - *REMINDER: d3 scales generally just output numbers! While they're often designed to be used for things like an X or Y axis, that isn't their only use!*

REVIEW QUESTIONS:

- What values are in the **domain** for **xScale**?
- How did we obtain those values?

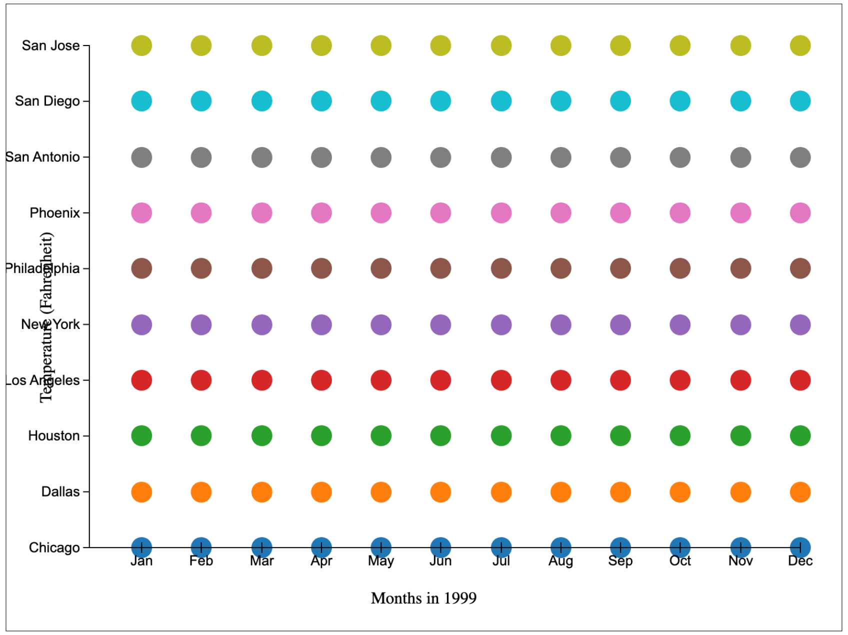
PRACTICE: **yScale** using **scalePoint**

Modify the **yScale** definition so that it:

- Uses **d3.scalePoint()**
- Uses the city names as the **domain**
  - *TIP:* You <u>could</u> copy all of the city names manually from the data, but it's much faster to use almost the same code that set up Array **xAxisValues** to create something to set up this Array
    - Maybe call it **yAxisValues**
    - What data are we gathering here? It's just one small change from what we do for the X axis!

Modify code that sets **"cy"** attribute for **temperatures** (our circles) so it:

- applies **yScale** to the correct data



PART 1: Setting up Array **yAxisValues**

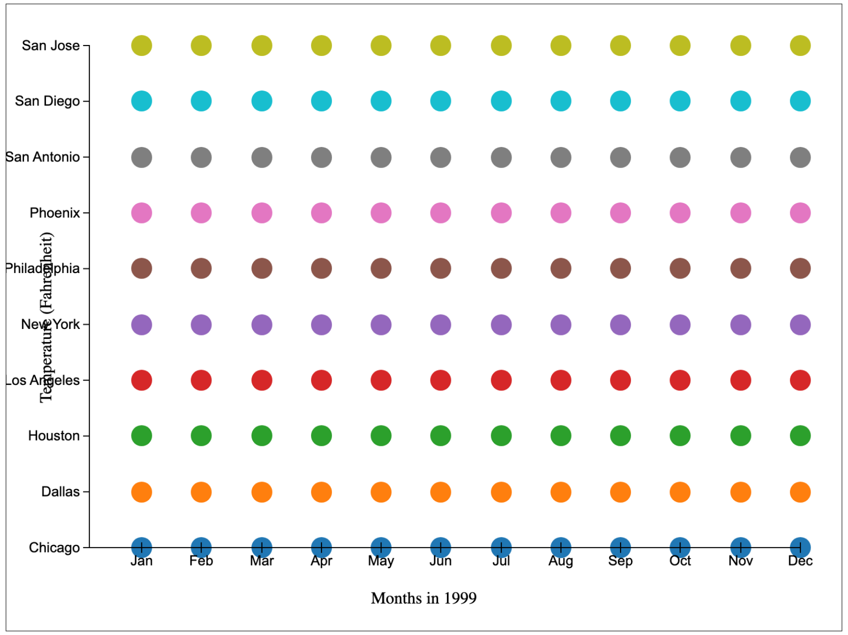
*We could also define this Array directly within the **.domain()** method for **yScale**, but having the separate variable makes our code easier to read— and means we can re-use the information later!*

Method 1 (the long, error-prone way):

```
let yAxisValues = ["Chicago", "Dallas",
"Houston", "Los Angeles", "New York",
"Philadelphia", "Phoenix", "San Antonio", "San
Jose"];
```

Method 2 (having JavaScript do the work for us):

```
let yAxisValues = Array.from(
    new Set(dataset.map(function (value) {
        return value.city
    }))
```



PART 2: Set up and use the **yScale** function

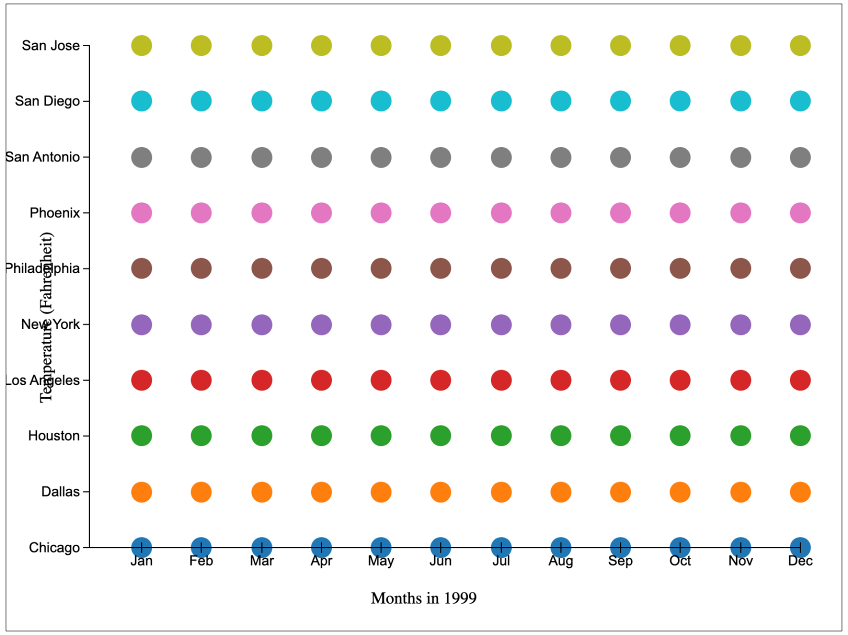
For both methods:

```
let yScale = d3.scalePoint()
.domain(yAxisValues)
.range([svgHeight - bottomMargin, topMargin])
```

*To make this easier to edit, we broke apart the chain for **temperatures** so you only need to find this line of code:*

```
temperatures.attr("cy", function (value) {
    return Math.round(yScale(value.city))
})
```

*Use of **Math.round()** here is optional, but recommended. It ensures that your graphics are "pixel perfect", giving you easy-to-read attributes like **cx="24"** instead of long funky values like **cx="24.33333379"***

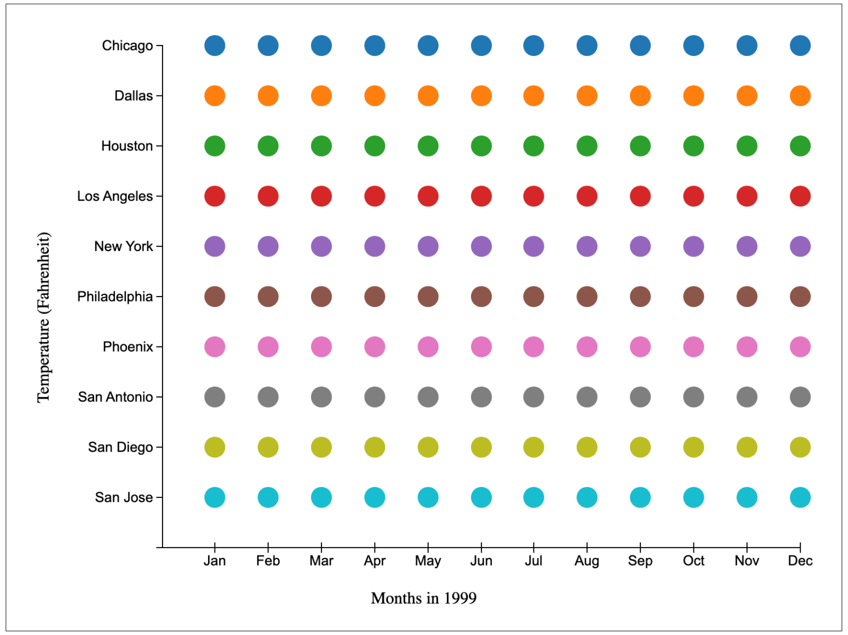


PRACTICE: Improve fit to canvas

This new design doesn't really use our SVG space very well!

Spend a few minutes trying to improve the visualization in these ways, in whatever order makes sense to you *(TIP: pick a problem you can solve first!)*

- The **left margin** feels very cramped right now; **make more room** for the text
- The **y axis label** is being overlapped by some of the city names; **move it out** so it has some space
- The data for the bottom city is on the X axis line, which looks bad. **Shift the Y axis values up** so that they don't start on the X axis line
- The yAxis data is not in alphabetical order or is in reverse alphabetical order. Fix this by modifying **yScale** and/or by using **Array.sort()** on our **yAxisValues** variable.



SOLUTION: Improve fit to canvas

Left margin really cramped:

```
let leftMargin = 150
```

Title being overlapped:

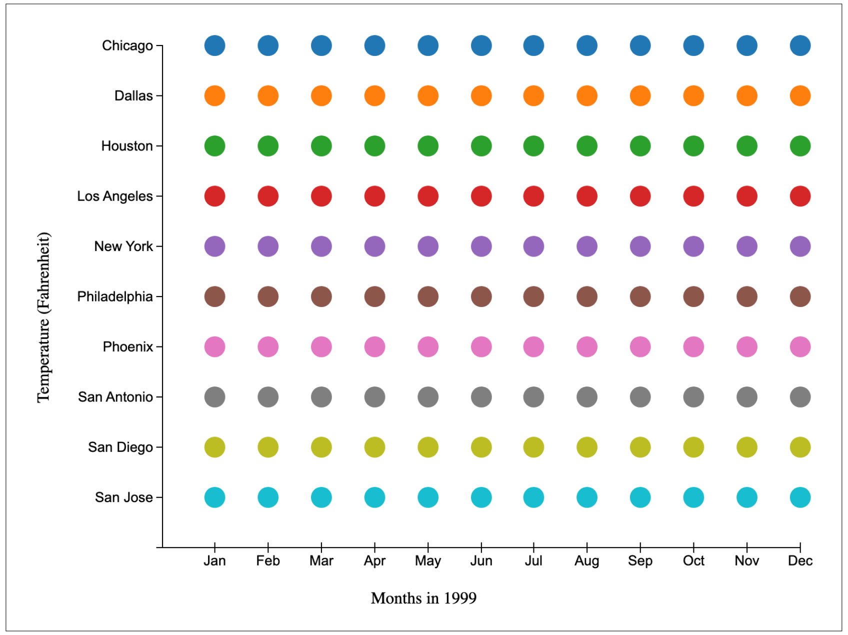
```
yAxisLabel.attr("y", leftMargin / 4)
```

Chicago data is on 0 line:

Use one of these two methods to add an empty string as "padding" to the start or end of Array:

- `yAxisValues.push("")`
- `yAxisValues.unshift("")`

No data will ever be "scaled" to this additional position because all of our data has city names!



SOLUTION: Re-order the labels

Alphabetize using **sort**:

```
yAxisValues.sort(function (a, b) {
    if (a.toLowerCase() < b.toLowerCase()) {
        return 1
    }
    return -1
})
```

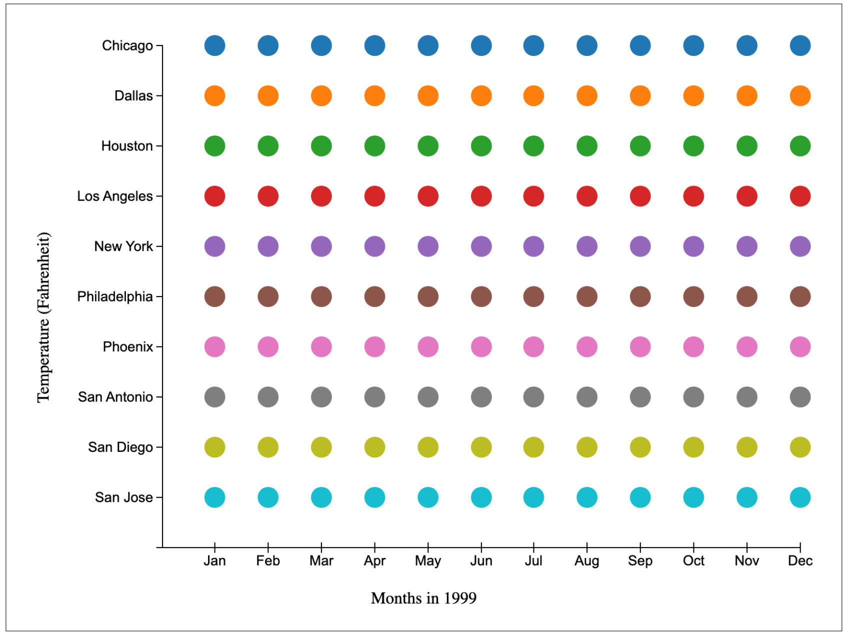
If you manually sorted within Array, your city names might be in reverse alphabetical order.

This is because **yScale** puts lowest values at the bottom (as we want for most graphs).

This is a rare case where you might not want your yScale to be flipped and might write instead:

```
.range([topMargin,svgHeight - bottomMargin])
```

Consider this on a case-by-case basis in your own work: What visually makes sense for your project?



REVIEW: Current (bad) color coding for cities

Here is the current code:

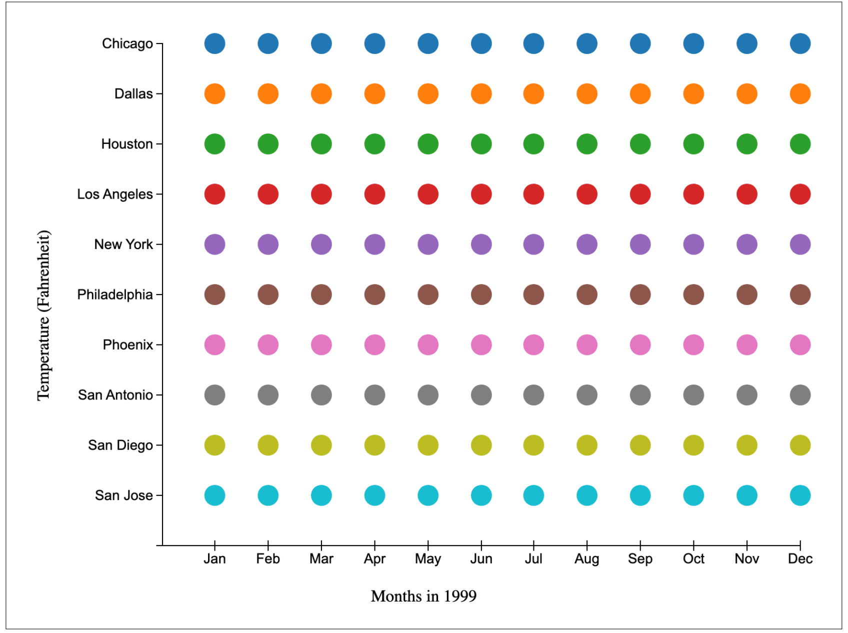
```
let colorScale =
d3.scaleOrdinal(d3.schemeCategory10)
.domain(["Chicago", "Dallas", "Houston", "Los
Angeles", "New York", "Philadelphia", "Phoenix",
"San Antonio", "San Diego", "San Jose"])
```

One new concept here:

- Passing a d3 **color scheme** as a parameter to **scaleOrdinal** causes the colors in that scale to be set as the **range.**
  - See https://github.com/d3/d3-scale-chromatic/blob/main/README.md for more color schemes.

The problem: Our color coding is now redundant with your Y axis information. It is fairly useless!

This common mistake is why we ask you not to use the same property of your data to control two attributes in your visualizations!*



** There are exceptions to this prohibition, as sometimes it can be useful to reinforce the contrast in data using multiple properties, but we will advise you if we think this is appropriate. For now, please use only one attribute (color, scale, x position, y position, shape) to represent each property of your data!*

scaleSequential for specialized scaling

- **d3.scaleSequential** creates a function that maps a continuous domain to a continuous range
  - Similar to **d3.scaleLinear** and friends, but uses an **interpolator function** to do something more complicated
  - **.interpolator(*functionName*)** is required to configure
    - just like **.exponent()** is required when using **d3.scalePow**
- Often used for color scales. In fact, there is a **d3 interpolator function generator** to do that.
  - Example shown here, but we will actually use one of the color palettes already set up in D3!

EXAMPLE:

Here, **d3.interpolate** generates a **function** that is then used by **scaleSequential**, which generates a **function** to **map** the numbers **0-99** to a **gradient** going between **purple** and **orange**

```
let colorScale = d3.scaleSequential()
        .domain([0,99])
        .interpolator(
            d3.interpolate("purple",
                "orange"))
```

For more information, see:
http://using-d3js.com/04_05_sequential_scales.html

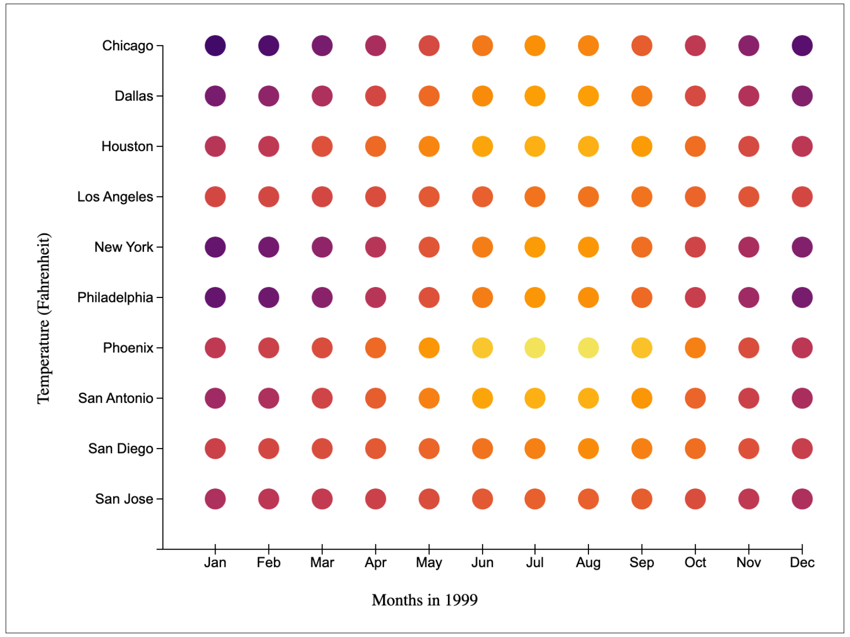
Color scale for temperatures

- Let's apply this concept to our visualization!
  - **Domain** is whatever range of numbers you want interpolated across the colors, as usual
  - For **interpolator** we'll use the built-in **d3.interpolateInferno**
    - Note that we only type in the name of the function! We are pointing d3 at a function, not calling it!
- Change the **colorScale** code as follows:

```
let colorScale = d3.scaleSequential()
    .interpolator(d3.interpolateInferno)
    .domain([0, maxTemp])
```

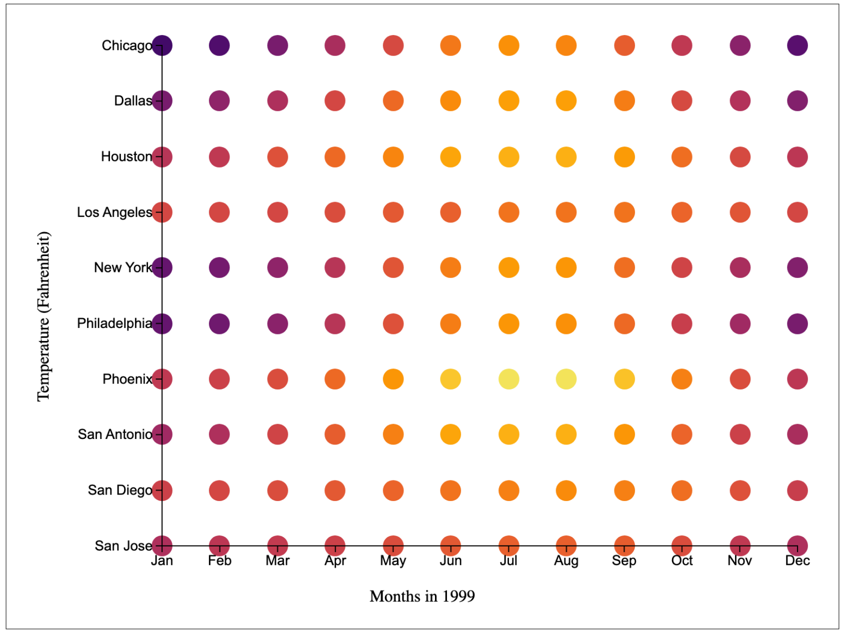
- And to apply this scale to our **low temperatures:**

```
temperatures.attr("fill", function (value) {
    return colorScale(value.temps[0])
})
```



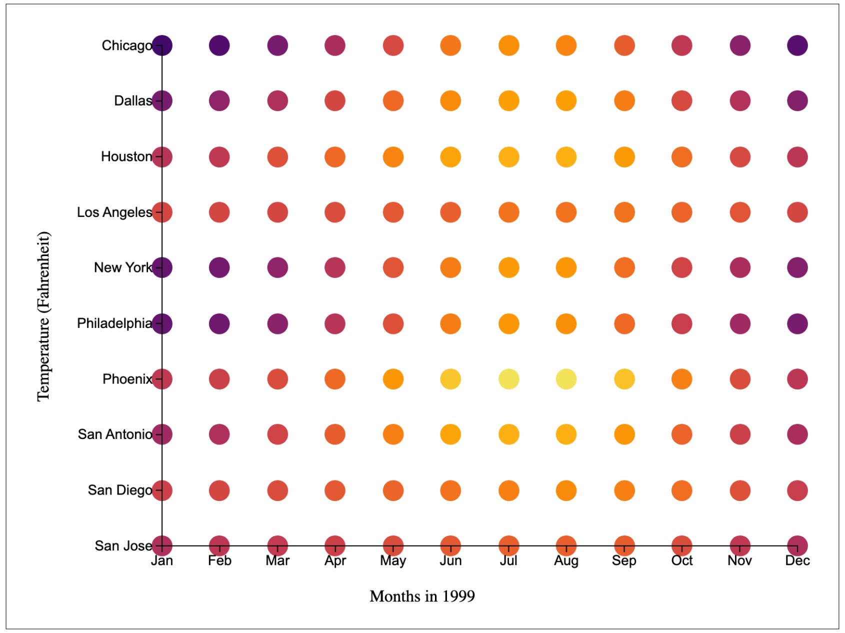
Part 2: Converting to a heatmap

- Load and test the project found in **2_scalepoint_months_cities_low_temps.zip**
- Almost the same as what you just accomplished
- Seemingly a step backwards, visually!
  - X and Y axis are back to having data plotted on their origin lines
  - Always avoid this / fix this on your projects!
- Note **code comments** throughout:
  - **CHANGE** comments point out where we've modified something for you, to make this exercise easier
  - **TO DO** comments that outline what you'll be doing in this exercise (and again, something you should be doing in your own projects as notes to yourself!)

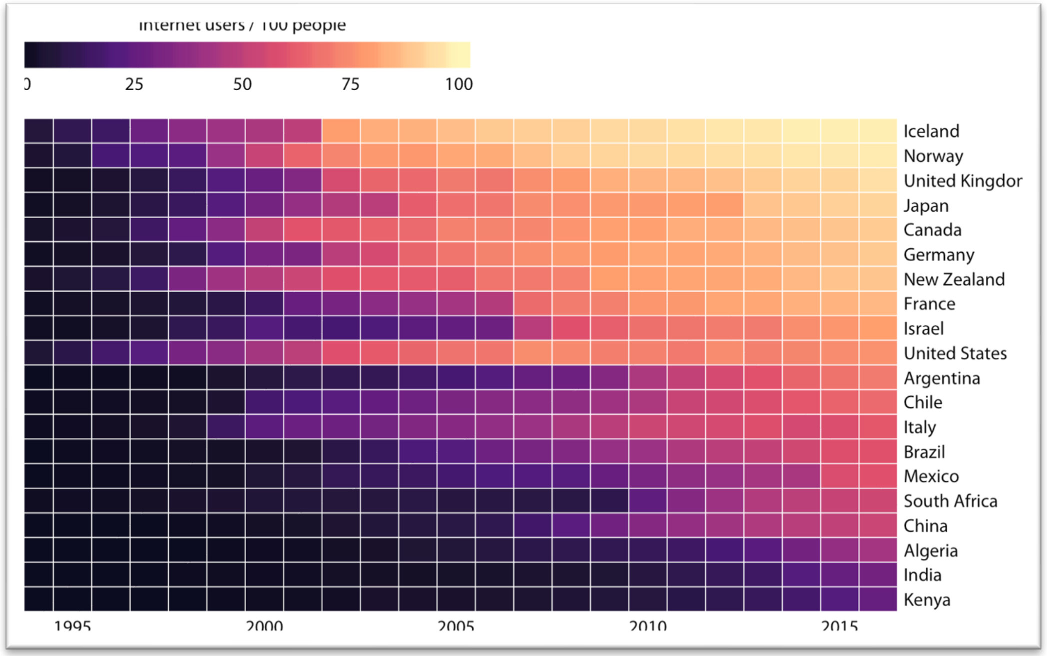


One big change: X axis generation method

- Previously we used a **for()** loop to generate the X axis. This time we use **d3.axisBottom()**
  - **for()** loop can give you more control over little details in your axis (if you write additional conditional logic, for example)
  - **d3.axis** methods take care of ensuring your axis is in sync with your scale. You can still customize the axis look, but the values are tied to the scale
- REVIEW:
  - How could we pad X and Y axis values so that our dots aren't on either axis line?
  - NOTE: We won't actually add this padding now. We'll approach the problem differently in a bit!



heatmap: discrete or "binned" data on XY axes, discrete or continuous data for color coding

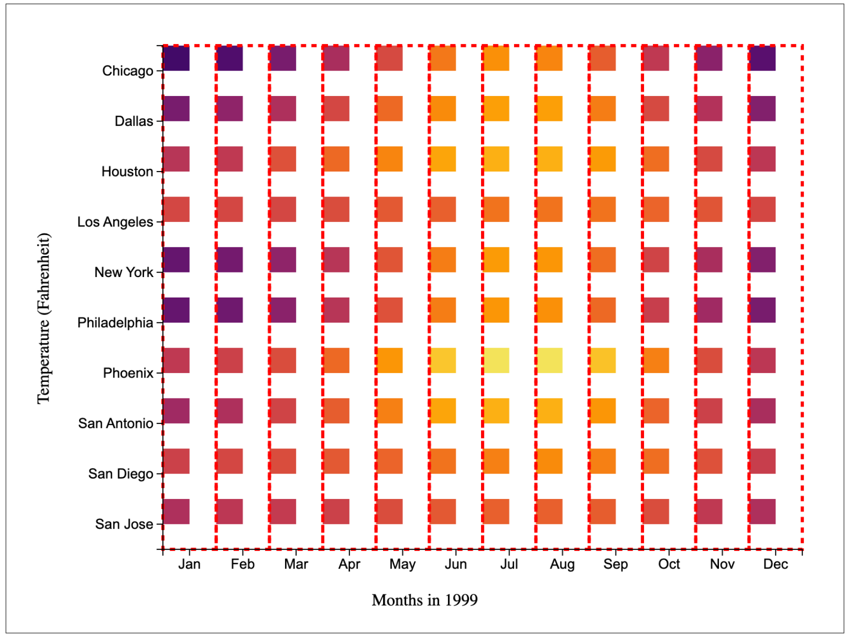


Percent of citizens using the Internet in various countries 1992-2018

- This heatmap shows 3 data elements:
  - Percentage of people in the country with Internet access
  - Various countries
  - Time
- Once you understand the colour scale, you can easily see that every country is getting more Internet access as time goes by.
- If you look closer, you can see interesting data trends such as:
  - Which countries got the earliest access to the Internet
  - Which countries now have the most prevalent Internet access

Purpose of **d3.scaleBand()**

- Subdivides axis into spans that are filled in to represent each value
- Value is "middle of" scale instead of a single point
  - Think of it as a "band" spanning out both directions from the scaled point
  - it's a similar idea to **scalePoint()**, but using the area between the gridlines to represent the point
  - In this image, the red dashed lines each show a "band" on the X axis, with each month corresponding to a "band"
- New few slides cover basic usage
- For more information, see: https://www.d3indepth.com/scales/#scaleband



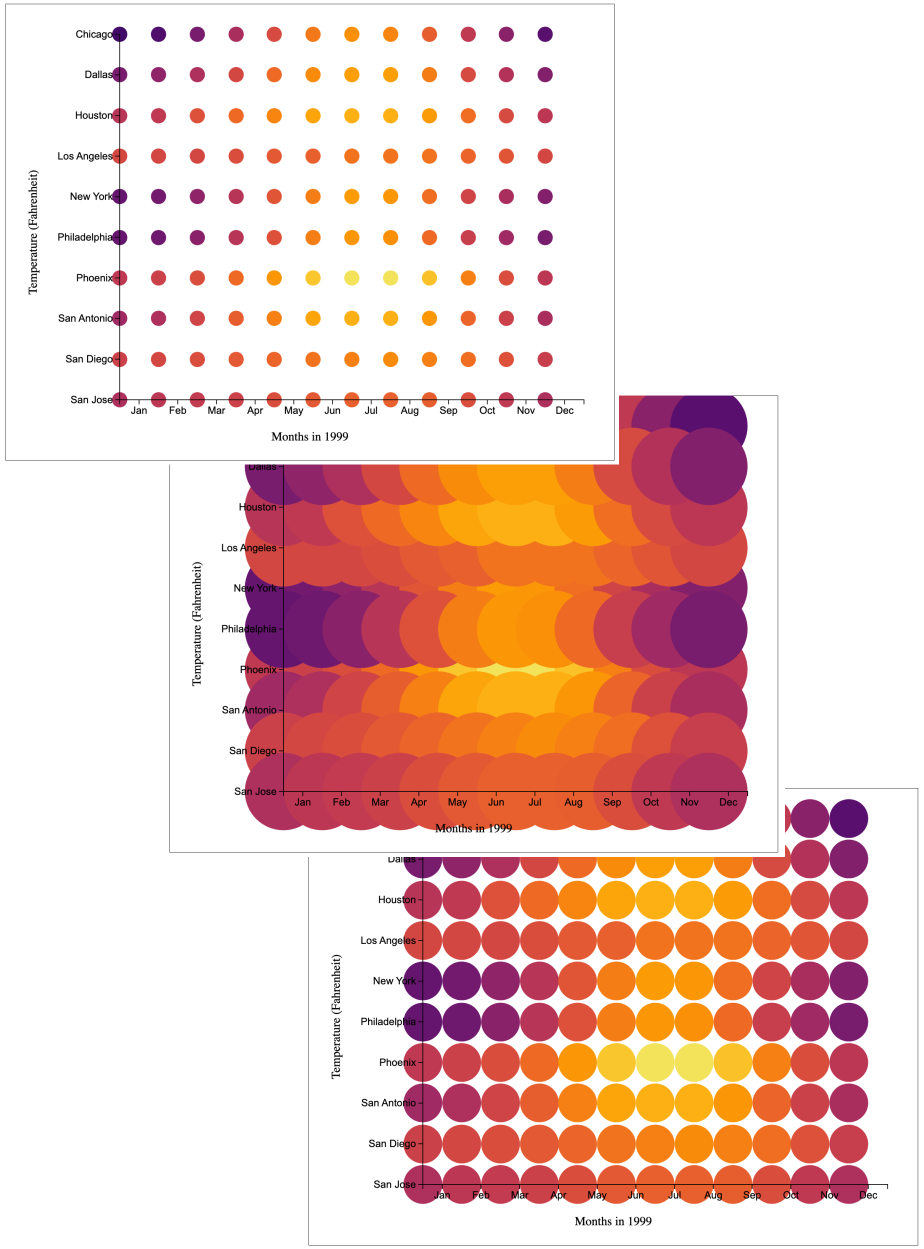
Converting our xScale

First, just change **scalePoint()** to **scaleBand()** and see what happens!

- Now our dots are offset from our labels.
  - This is a common sign of mixing up **scalePoint** and **scaleBand**, so watch out for this in your own work
- **scaleBand()** expects you to use the space between bands, which accessed by the scale's **bandwidth()** function

Next, set the **"r"** attribute of our **temperature** circles to **xScale.bandwidth()**

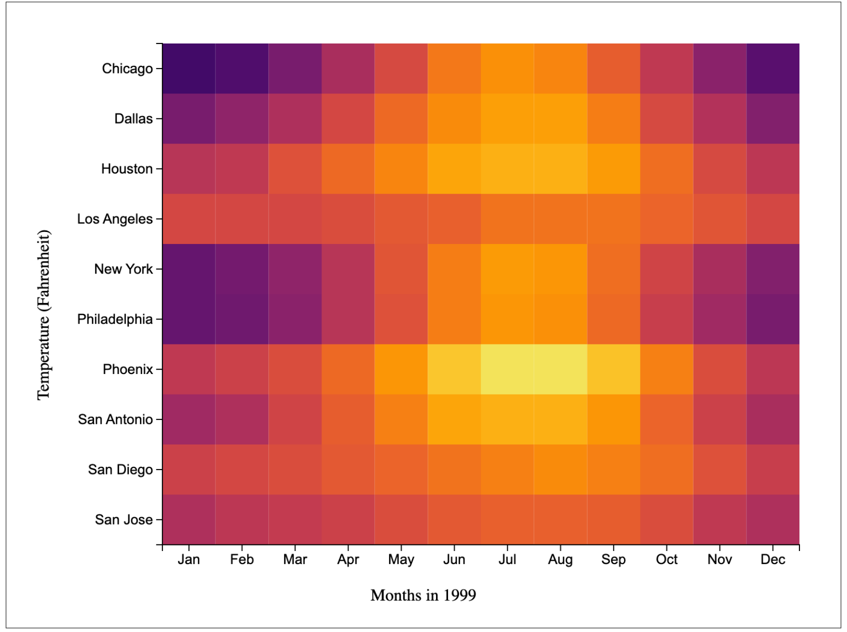
- Now our circles are clearly too big!
- What simple math to fix?
  - HINT: **bandwidth()** assumes rectangles, not circles!



Change to rectangles

Adjust the code for **temperatures** so that it uses rectangles instead of circles

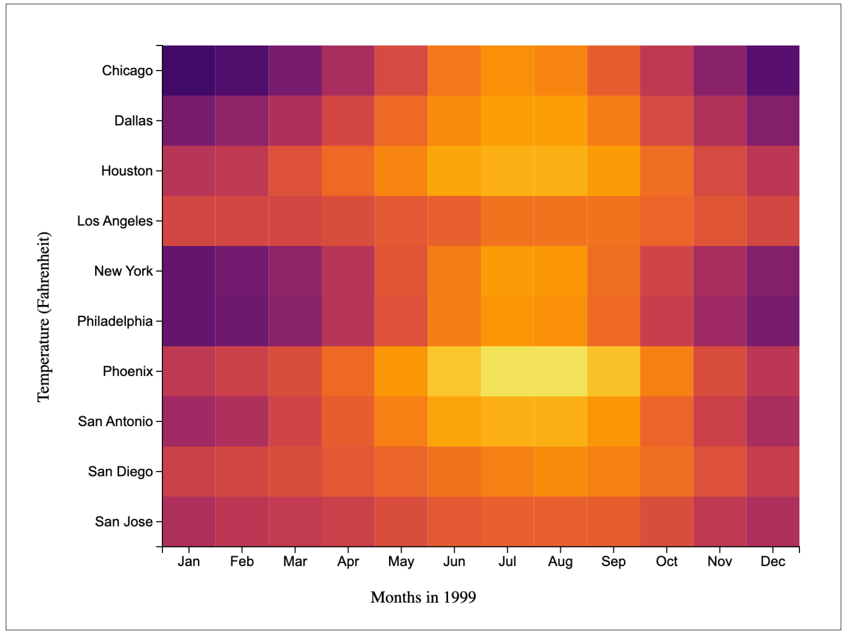
- Change **circle** to **rect** wherever needed
- **"r"** becomes 2 attributes: **"width"** and **"height"**
  - Use **xScale.bandwidth()** for "width"
  - Change your **yScale** function generator to **d3.scaleBand()** then apply its **bandwidth()** method to the **"height"** of the **temperatures** rectangles
- **"cx", "cy" position attributes** become **"x", "y"**
  - For now don't worry about centers versus corners. Sometimes you just need to experiment and then adjust later!



Wait, did something just work?! (the code)

```
let yScale = d3.scaleBand()
             // rest of yScale remains same


let temperatures = svg.selectAll("rect.temp")
             .data(dataset)
             .join("rect")
             .classed("temp", true)
             .attr("width", xScale.bandwidth())
             .attr("height", yScale.bandwidth())
             .attr("x", function (value) {
                 return xScale(value.month)
             })
             .attr("y", function (value) {
                 return yScale(value.city)
             })
```

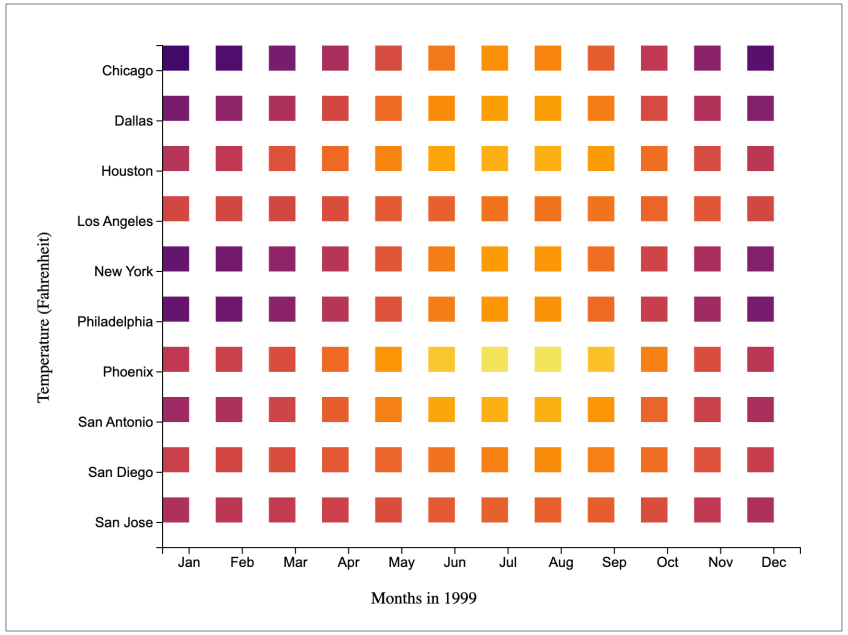


Wait, did something just work?! (testing)

Could it be that our rectangles are somehow being drawn from the center, like our circles?

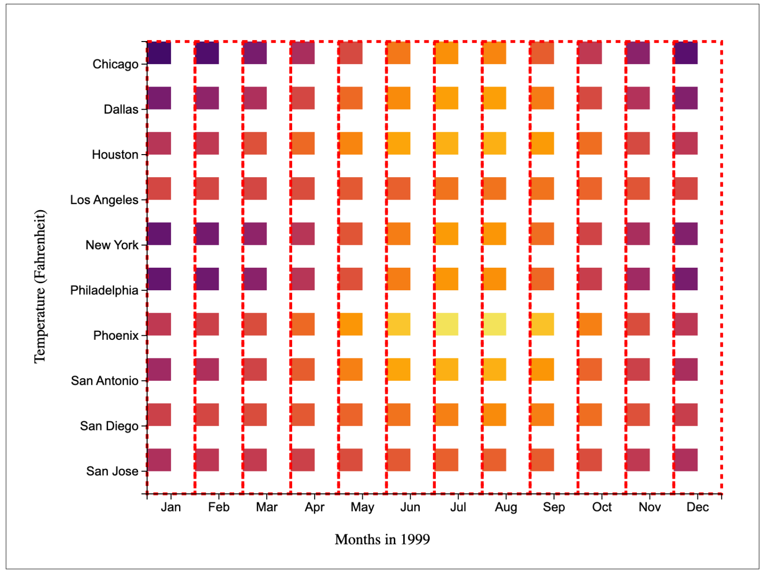
How could we test this?

- Try making rectangles half as wide and tall and see where they're currently drawn from!
  - (You should end up with the result shown here)
- Looks like they're NOT being drawn from the center but from the upper left corner of the "box" that d3 has defined for each scale space
  - This "box" is the intersection of the horizontal and vertical bandwidth

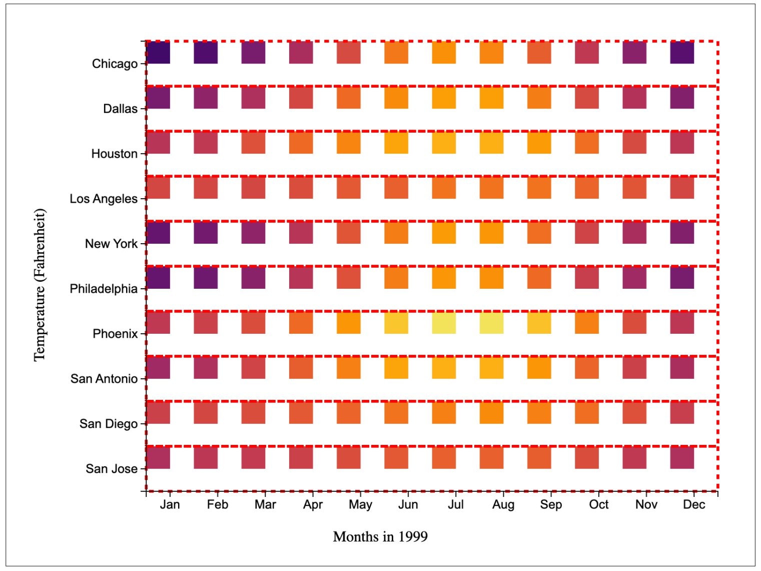


A visual explanation of bandwidths

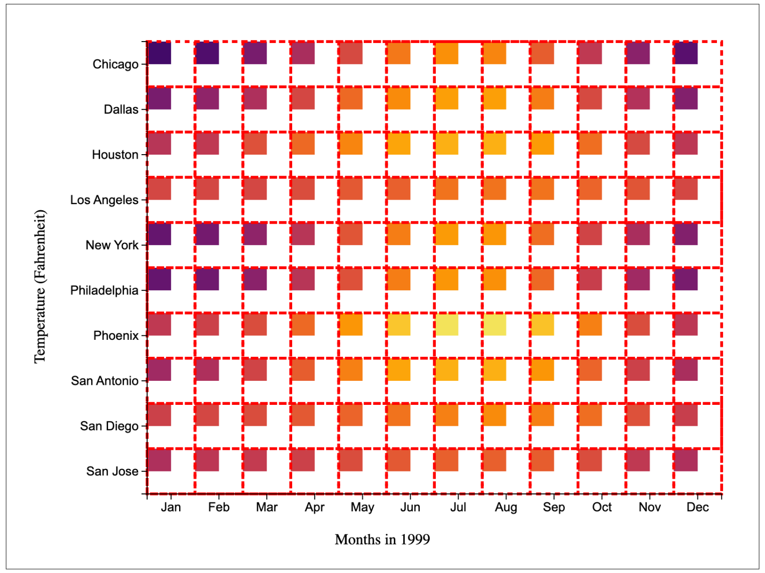
x axis bandwidths



y axis bandwidths



bandwidth intersections



Refinements

- First, make **width** and **height** use the full bandwidth values again (undo our experiment)
- Axis ticks visually indicate a value is at a particular **point**
  - Our axis labels now each refer to a **span** of on that axis (meaning a range of values, not to be confused with an HTML span!) a whole area, so our ticks are not giving the right visual message!
- This is a shockingly simple change, shown here for the X axis. Do the same for the Y axis.

```
.call(d3.axisBottom()
    .scale(xScale).tickSize(0))
```

- **Your challenge:** Remove the domain lines that run along the X and Y axis.
  - They're barely visible, so uncomment **.domain** style in CSS if you want to see these lines clearly as you try to remove them!
  - D3 has a *selection*.**remove()** command to get rid of these lines, but **how should we specify this selection** from the items generated by d3.axis?
    - Use the Elements Inspector to check out those lines if you don't remember!

Another shockingly simple refinement!

- To get rid of the domain lines, we just add this code <u>after we have drawn our axes:</u>

```
svg.selectAll(".domain").remove()
```

  - REMINDER: Code order matters. If we put the line of code before we have drawn our .domain lines, then it will remove nothing and the lines will be added after.
- The **.domain** in the CSS was a hint!

REVIEW:

- What would we call **.selectAll().remove()** on if we wanted to remove only the X axis domain line but not the Y axis one?
- If we only wanted to remove one element of class **.domain**, what selection method would we use instead of **.selectAll()**

Giving our visualization a little breathing room

- Rounded rects for **temperatures**

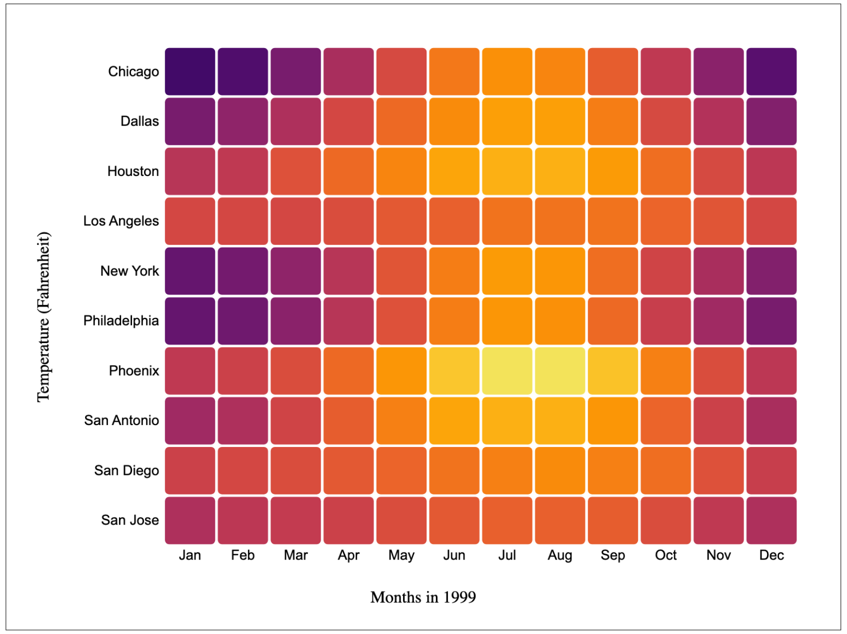
```
.attr("rx", 4)
.attr("ry", 4)
```

  - Attributes **"rx"** and **"ry"** provide rounded rectangle corners. Try playing with these values, including setting one to 0 and the other to a bigger number!

- Add a new configuration method to your **xScale** and **yScale** definitions:

```
.padding(.05)
```

  - This configuration method is only usable when setting up **scaleBand** functions.
    - There are other ways of doing padding for other d3 elements. You can research these!
  - Method parameter is a value between 0 and 1, expressing a **percentage** of the width of the band to be used for whitespace (.05 = 5%)



Figuring out our temperature ranges

- **d3.max** lets us easily access a value of a property to determine which one is biggest, but what if that value is a complex data type such as an Array?
- We have to figure out maximum value of an Array of values for each city and each month's temp, so **d3.max** won't do the job on its own here.
- If we only had low and high values for temps we could say something like "only look at low temperatures **temps[0]** for low and only look at high temperatures **temps[1]** for high"
  - In fact, this is what our earlier temperature dataset had, so you can play with these values.
- For this phase of the project, we added a bunch more temperature measurements and put them in random order!
- In fact, our lazy temperature measurers were very inconsistent in how many temperature samples they took in each city each month.
- This means we need to figure out our maximum temp based on an unknown number of items in each list!

Array.**reduce()**: One number, from many

Template:

```
Array.reduce (function (result, value) {
    // operation to do on value and result
    return result
}, initialValue)
```

- Note the additional element here: an **initial value** is set after the **helper function** declaration. This value sets variable *result*, which is a very important step!

Example *(try at console without let!)*

- In this example, each **iteration** through the **reduce** function simply adds the next **value** from the Array **numbers** to the current **result** (which is initialized to 0)

```
let numbers = [1, 2, 3, 4]

let total = numbers.reduce(function (result, value) {
    return result + value
}, 0)
```

Conditional logic to find biggest number

If you were manually going through a list of numbers, how would you find the biggest one?

Write down the first number	**initial value of result**
Compare it to each number that follows	**value sent from reduce**
If the number being compared is greater, write it down as the new biggest number	**value replaces result**
Otherwise, keep using the number we already noted as biggest	**result remains same**

At the JavaScript console, try coding this algorithm as conditional logic using **if** and **return**

- Start with `numbers = [1, 2, 3, 4]`
- Use this template to write your function to find the biggest number!

```
numbers.reduce(function (result, value) {
    // your code here
}, 0)
```

- You should see result 4 on the Console if you get this right

Solution: Conditionally find maximum

- Here is one version of the solution. You could do this a number of ways!

```
numbers.reduce (function (result, value) {
   if (value > result) return value // value becomes new result
    return result // result remains the same
},0)
```

- Note that we said **return value** above. This is because whatever is **return**ed from the evaluator function becomes the new **first parameter** in the function's next pass.
  - In other words, **return value** puts the value from **value** into the **result**
  - We could also write **if (value > result) result = value** to make clearer that we are changing **result**
  - The **return result** line would then always do the return, whether its value was changed or not!

JavaScript's **Math.max()**

- Much simpler than **d3.max**
- Just returns the biggest number from the parameters it receives:
  - **Math.max (3,2)** returns **3**
  - **Math.max (4,2,1,3)** returns **4**
- Doesn't work on an Array, has to be numbers sent as individual **parameters**
- However, we can still use this with our Array, since with *Array*.**reduce()** *we are* looking at one **value** at a time and we have a place to store our **result**
- CHALLENGE: At the Console, try to rewrite your **reduce()** function so that it uses **Math.max()** in place of the conditional logic
  - Think about how to rephrase this in English: "the **result** is the bigger of the two **values** I am comparing"
- You should again get the answer **4**

Solution: Math.max()for maximum in Array

- Here, we are simply saying "**return** whichever number is higher: the current **result** or the new **value**"
  - Again, whatever is returned from this evaluation function ends up as the new **result** parameter value for the next iteration!

```
numbers.reduce (function (result, value) {
    return Math.max(result, value)
},0)
```

TIP: Neither way is "better"; you should code in the way that makes the most sense to you!

getting maxTemp

- Now we can get the biggest number from each of the 120 **temps** lists in our data
- We can then pass those numbers to d3.max to compare
- Remember: a d3 **accessor function** can do anything with the **value** it receives, and one **property** of our data objects is our **temps** arrays!
- The framework will look like this. Enter this code and try to fill in the missing line!

```
let maxTemp = d3.max (dataset, function(value) {
    let tempsList = value.temps
    let biggestInList = tempsList.reduce(function (result, value) {
        // code to find biggest temp in current list
    },0)
    return biggestInList
})
```

SOLUTION: getting **maxTemp**

```
let maxTemp = d3.max (dataset, function(value) {
    let tempsList = value.temps
    let biggestInList =
    tempsList.reduce(function (result, value) {
        return Math.max(result, value)
    },0)
    return biggestInList
})
```

Here is an abbreviated version of the code that doesn't use extra variables:

```
let maxTemp = d3.max (dataset, function(value) {
    return value.temps.reduce(
        function (result, value) {
            return Math.max(result, value)
        },0)
    })
```

- The first solution given here is the "fill in the blank" solution from the code we provided on the last slide.
  - You could **console.log()** the values **tempsList** and **biggestInList** if you needed to more closely examine what's going on.
- The second solution does exactly the same thing in the same way, but you need to understand what each **return** is sending, and where that value goes
- For this class, **write your code so you understand it**.
  - For some people this means more variables; for others it means more code comments
  - *Being 100% clear about what your code is doing is the most important thing!*

Practice: **minTemp**

- Create a variable **minTemp**
- Use **d3.min** and **Math.min** along with **Array.reduce** to set this value
  - **HINT: They work identically to the .max versions but return smaller value**
- Check the value of **minTemp**. It should not be 0, as our temperatures never get that low!
  - Where is the 0 coming from?
  - What can we change the 0 to in order to ensure our numbers are all lower?
- Change our **colorScale** function to use **minTemp** instead of 0

minTemp and Infinity

- Here is the solution:

```
let minTemp = d3.min(dataset, function (value) {
    return value.temps.reduce(function (result, value) {
        return Math.min(result, value)
    }, Infinity)
})
```

- **Infinity** is a built-in number that ensures all values that you test against will be smaller than this initial value.
- You can also use **–Infinity** (negative infinity) to ensure all values that you test against will be bigger than this initial value!
    - Why didn't we need to do this for our **maxTemp?**

Average temperatures

- Currently we are still showing the <u>first</u> temperature from each **temps** list
- Since we just figured out the actual <u>low</u> and <u>high</u> temperature, we should be able to use a similar technique to figure out the **average** temperature for each **temps** list
- Still using your same **colorScale()**, try making the "**fill"** attribute reflect the **average** temp for each datum
  - This means averaging the values in each **temps** list
  - **Reminder:** Average is **total** of all values in list **divided** by **number** of values in list
- You just learned how to total the values with **Array.reduce**, but you can use another method (like a loop) if you want.
- *HINT: How do we check how many items are in a list (specifically an Array)?*

SOLUTION: Average Temperatures

- Here is the solution using **reduce**:

```
.attr("fill", function (value) {
    return colorScale(value.temps.reduce(function (result, value) {
        return result + value
    },0)/value.temps.length)
})
```

Reminder: Average is **total of all values** in list divided by **number of values in list**

- To get **the total of all of the values**, we write the same **evaluation function** that we wrote to practice adding up numbers!
- Then after getting the **result** from **reduce**, we divide the final answer by **value.temps.length** (i.e., the **number of values** in our Array)