SVG groups, D3 axis generators, class selections

Class #8 – Fall 2022

rev. 2022.11.05.a

Setup

- Download, open, and test **scatterplots_part3_starter.zip**
- This project picks up where last week left off, with a few changes:
  - A few code modifications to emphasize things that you've learned over the past few weeks.
  - Some new code comments to help you analyze code and experiment later
    - Study this code later as you review past class PDFs!
  - A bunch of new **/* TO DO */** items in the comments for us to finish off today!

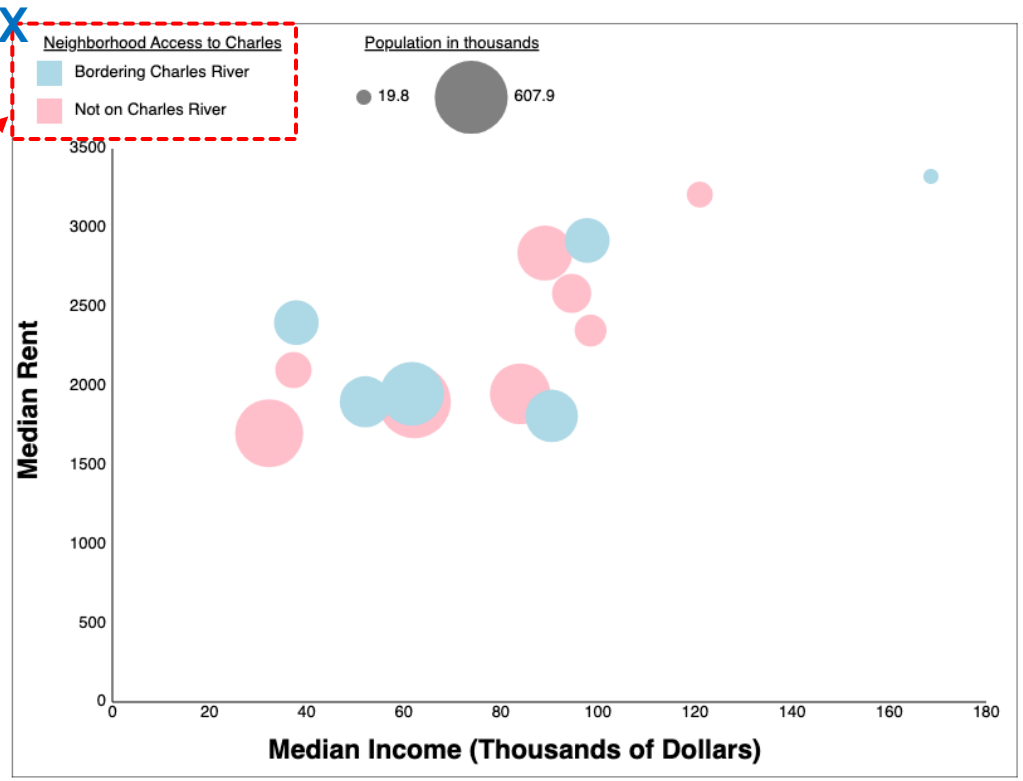
- Today may be the end of our scatterplot explorations, but it's just the beginning of learning how to visualize your data!
  - Everything that you've learned in the past two weeks and everything that you'll learn today can be applied to any visualization, not just scatterplots!
  - Your job over the next few weeks will be to practice these techniques and to learn to generalize these solutions for other design challenges that you may encounter!

SVG Groups

Look at the code under the comment:
`/* MANUALLY ADD MAP LEGEND ELEMENTS */`

- We laid out our color key using specific coordinates on our SVG, which is a fine way to start, but not very flexible
- The left side of our visualization is a bit crowded, but there's space on the right!
- Wouldn't it be nice if we could move **all 5 elements** of this key without having to recalculate the coordinates?
- We can, if we add these elements to a **group** instead of directly to the **svg**

*A caveat: To make this exercise simpler, we drew these elements so that the group's **origin** matches the SVG **origin**. If we had drawn this key elsewhere on the SVG, we'd still need to do some math to make the group work as we want (very much like what you did with your project1 drawings)*



Setting up the group

- Just below our **TO DO** comment, add this line of code:

```
let colorKey = svg.append("g")
```

- Then, for the 5 objects that form our color key, change the target of each **append** from **svg** to **colorKey**
  - *TIP: all 5 are directly beneath the line we just wrote!*
- These 5 objects were previously not stored in **variables**, so we didn't have a way to "talk to" them after they were created. We still can't easily refer to an individual one, but we can refer to the **group** of them as **colorKey**

About Groups

"g" is an SVG object type like **"circle"** or **"rect"**, but it works a bit differently:

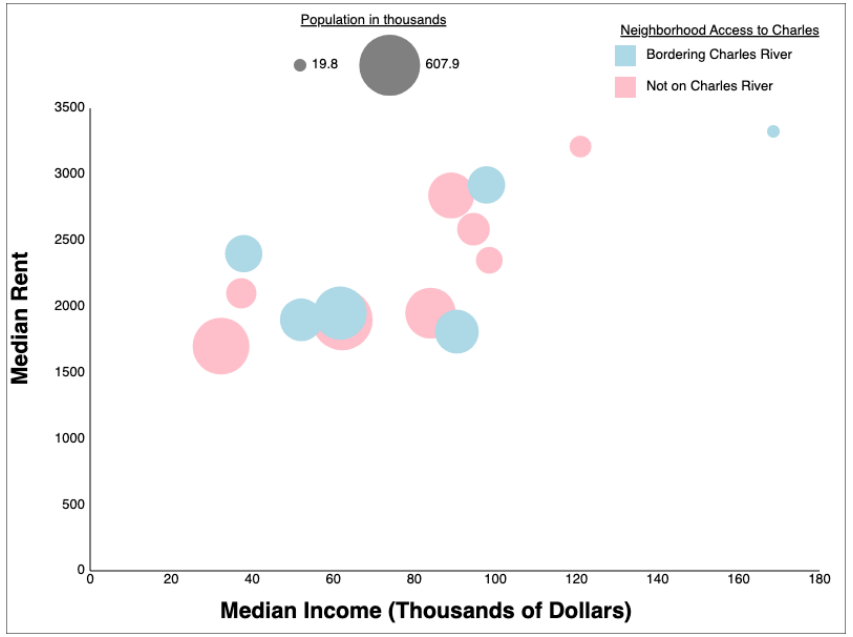
- Unlike an SVG shape, the **group** is designed to contain other SVG objects
- A **group** has no **size** or **position** properties
  - Its dimensions are determined its contents and can't be read using **.width** and **.height**
  - Its upper-left corner is always **0,0** and it can't be repositioned using **.x** and **.y** (or **.cx** and **.cy**)
- Learning different **methods** to manipulate **"g"** objects may seem a pain, but hopefully you'll see that the benefits outweigh the difficulties *(and if not, you can go back to using math)*!

That **transform** attribute again!

After the code that appends our 5 objects to **colorKey**, add this line to see our group immediately relocate:

```
colorKey.attr("transform", "translate(560,10)")
```

- What do you think would happen if we moved this line to **before** we added our objects to the group? Try it!
  - A group can be transformed without anything in it. Anything added will take on the transformations.
  - We will shortly do something that requires the group's content, so move the line back to its original spot!
- The **"translate"** directive enables us to **move** an SVG element by a specified number
  - Here we say "Move this group 560 pixels **to the right** and 10 pixels **down** from where it started"
  - This is **relative movement.** For example, if this group was originally positioned at **100, 10** on **our** svg, the above **translate** would place it at **660, 20**



CSS / D3 update: Using **.style** and new transform function properties (OPTIONAL)

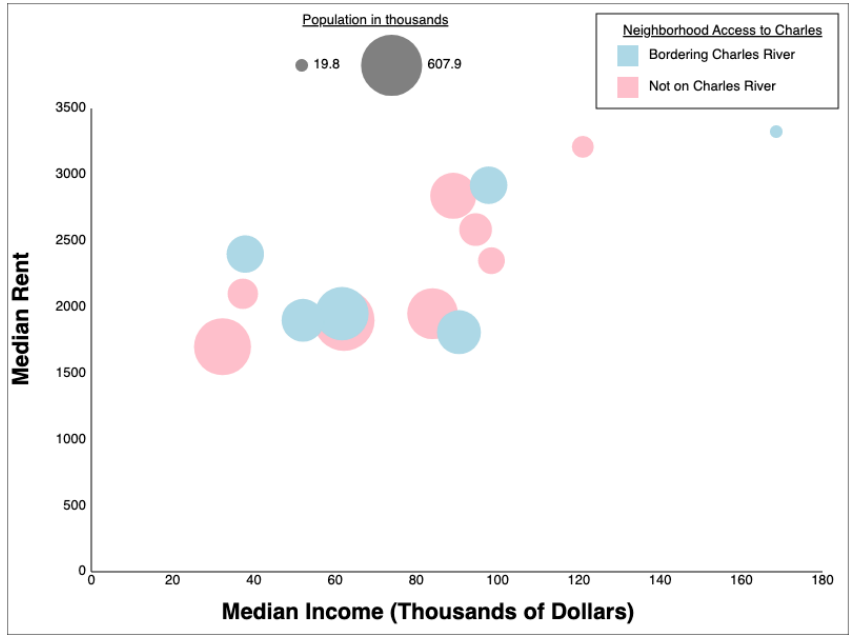
- With D3 v7 (the version we're using for the term project and in today's class), you can take advantage of the upgrades to CSS transforms, as documented here: https://developer.mozilla.org/en-US/docs/Web/CSS/transform-function
  - These new transform "functions" enable you to break down complex transforms into smaller steps
  - You still have to string the commands together, but they might make more sense the new way
- These transforms are based in **CSS** instead of **SVG**, so we use **.style()** to apply them instead of **.attr()**
  - Before: `colorKey.attr("transform", "translate(560,10)")`
  - After: `colorKey.style("transform", "translateX(560px) translateY(10px)")`
- The new system also allows you to use any CSS units instead of just pixels.
  - Example: `colorKey.style("transform", "translateX(50%) translateY(50%)")`
  - With this code, your key moves halfway across the canvas and halfway down, without you having to reference your SVG's **height** or **width** at all! Might be a handy way to center stuff?

Putting a box around our color key

- Add this code wherever you think appropriate:

```
colorKey.append("rect")
        .attr("width", 230)
        .attr("height", 90)
```

- Then, continue this chain to set more attributes so that the **rect** is:
  - unfilled, so we can see what's behind it
  - surrounded by a black 1-pixel border
  - positioned so that its **upper left corner** is the same as the group's **upper left corner**
    - *HINT: What is the **origin** of a **rect**? That is, where is it drawn from? What about the **origin** of a **g**, based on what you've observed so far?*



Solution & a "What if?" for later

- Here is the code:

```
colorKey.append("rect")
    .attr("width", 230)
    .attr("height", 90)
    .attr("fill", "none")
    .attr("stroke", "black")
    .attr("x", 0)
    .attr("y", 0)
```

- This could go anywhere after we created our **group**, but let's put it above our **transform** code
  - You might later want to **transform** the **colorKey** location based on the size of its contents, so putting the transform last ensures that everything is in place!

We can remove this to-do from our code comments! Yay!

Experiment for later

- What if we wanted to make the outline box a little bigger on all sides? For the left and top, are we stuck because we've hit **0,0**? Experiment and find out!
  - Try increasing the rectangle's **width** and **height** by 10 pixels each.
  - Then try to adjust the rectangle's **position** by 5 pixels so that the key's content is centered in the rectangle again.
    - This means moving to the **left** and **up** from 0,0, which means **subtraction**!

Working with built-in axes in d3.js

- Comment out all of the code under **`/* x axis line */`** and **`/* x axis values */`**
  REMINDER: highlight the code and press command-/ to quickly comment/uncomment
- Add the following code in its place:

```
let xAxis = svg.append("g")
          .classed("key", true)
          .attr("transform", `translate(0, ${svgHeight - bottomMargin})`)
          .call(d3.axisBottom()
                .scale(xScale))
```

- Things to watch out for:
  - "template string" format with **backticks** so that we can insert some JavaScript in our String
  - a secondary **chain** of methods within the new method **.call()** – just watch out for where you put your parentheses!
- We'll explore all of this over the next few slides!

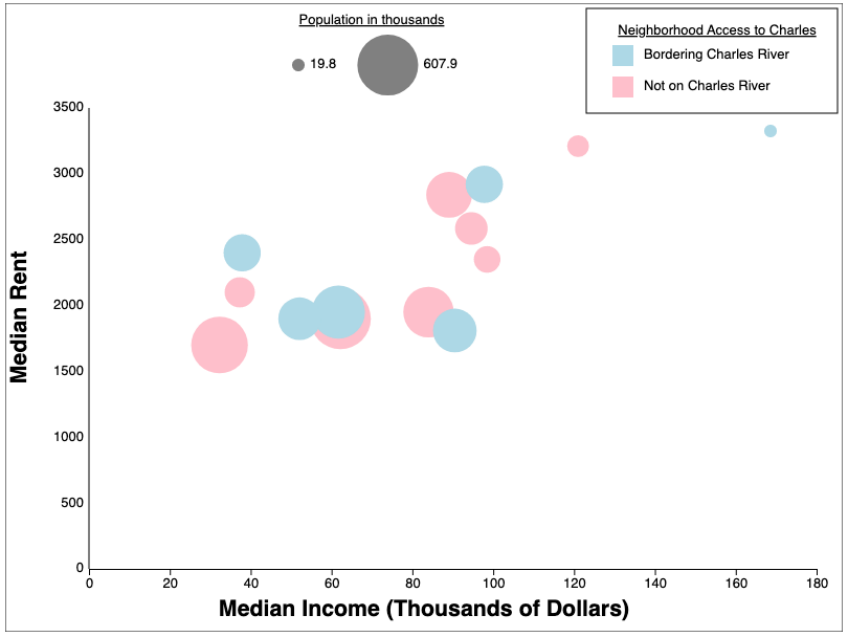
*REMINDER: **backtick** is the character to the left of the number 1 on US keyboards. It is below the tilde (~) character.*

D3-generated axes: the group

- D3 can generate a properly labeled axis based on any **scale function** that you've generated
  - uses the **domain** information from your **scale function** to set up the **value labels**
  - uses the **range** information to **position** the labels and ticks on the generated **axis**
- The d3-generated axis must be placed into an SVG **"g"** (group), or things get messy. How?
  - Try commenting out **.append("g")** so that your code begins like this:

```
let xAxis = svg//.append("g")
                .classed("key", true)
```

    - The visualization will update to look very wrong, but you shouldn't see any errors on the console!



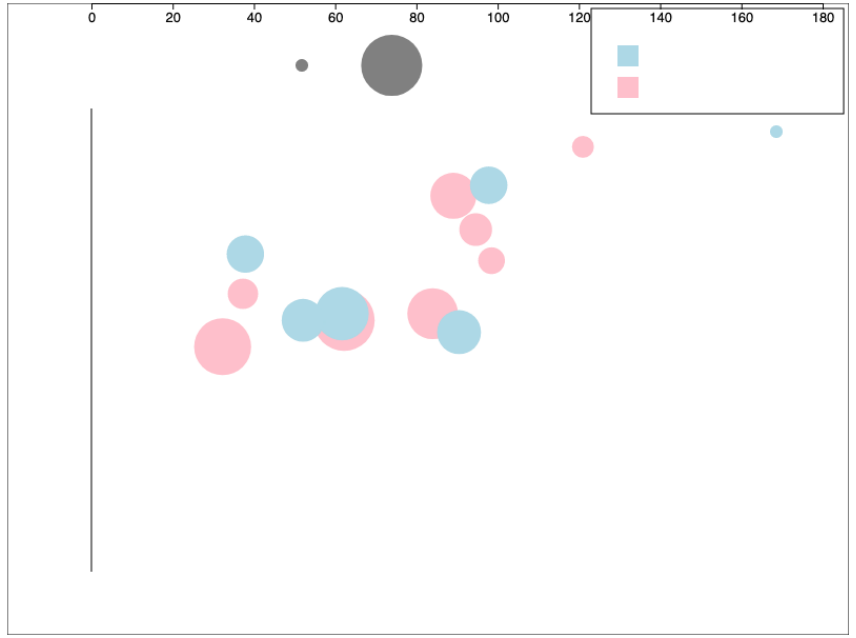
Your scatterplot should now have a slightly fancier X axis with little tick marks on it!

Experiment to understand

- To get a bit more clarity about what's happening, comment out the **.attr("transform")** line.
- Your code should end up looking like this:

```
let xAxis = svg//.append("g")
            .classed("key", true)
            //.attr("transform", `translate(0,
${svgHeight - bottomMargin})`)
            .call(d3.axisBottom()
                    .scale(xScale))
```

- SVG should now be at top of page, with **x** axis drawn at the top of the SVG. Consider:
  - What effect was the **transform** attribute having on our **svg** before we disabled it?
  - Why is the **x** axis still correctly positioned **horizontally**, but not **vertically**?

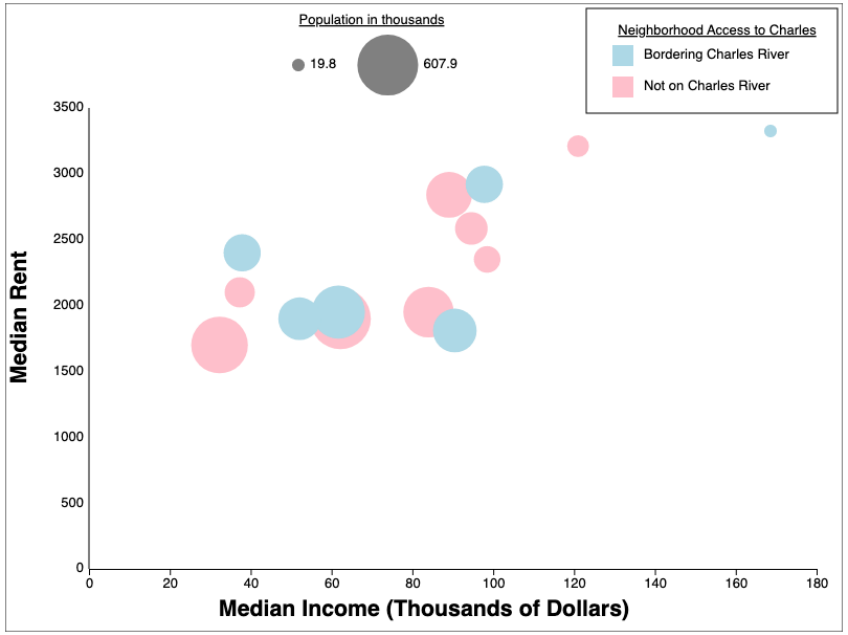


Moving the X axis into place

- **Uncomment** both lines that you commented out
- Look at the **transform** line:

```
.attr("transform",
    `translate(0, ${svgHeight - bottomMargin})`)
```

  - This says "move (**translate**) the selection (our **"g"** that will contain the axis) **0** pixels to the **left** and **svgHeight – bottomMargin** pixels **down**"
    - Just like we did with our **color key** earlier!
  - The generated axis uses our **xScale** function, so all horizontal positioning for the scale is handled
  - Despite the name **d3.scaleBottom**(), d3 doesn't actually know where you want this scale, so it plops it at the default Y position
    - **0,0** is always the default for SVG positioning



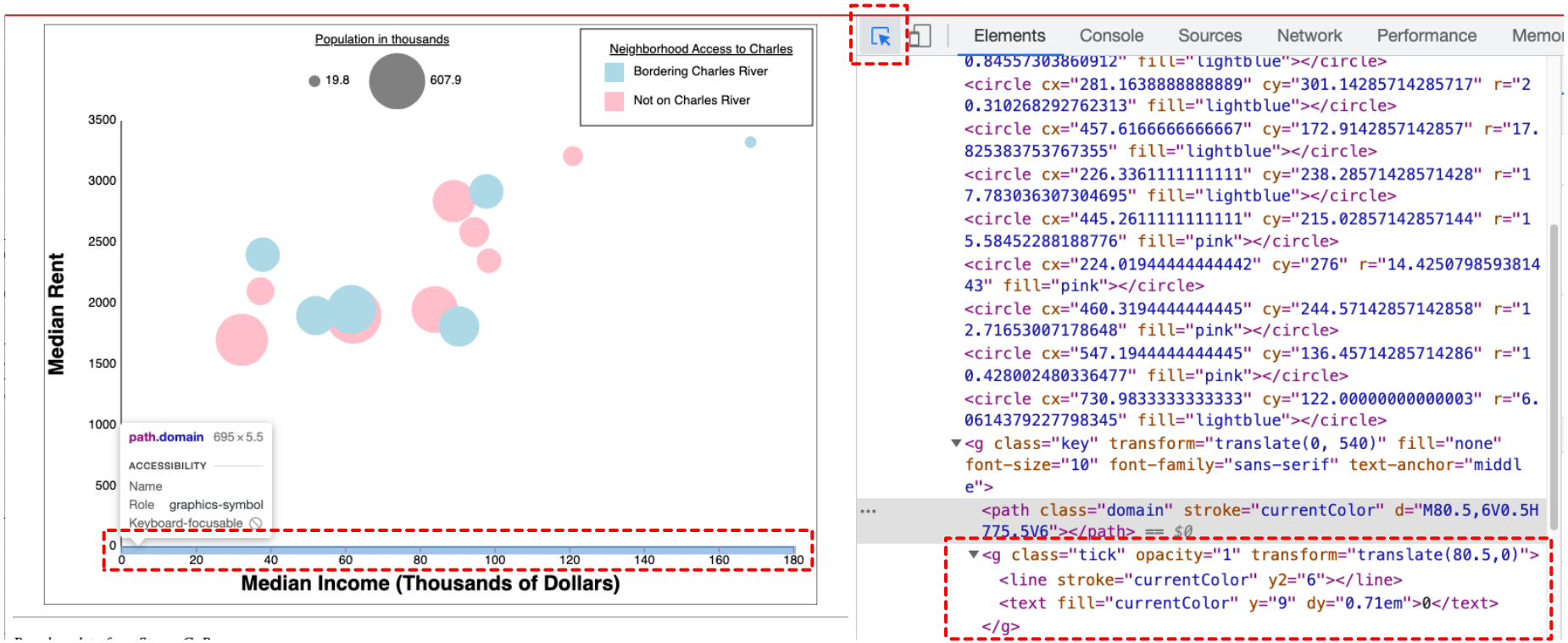
REVIEW: "string template" with backticks

```
let xAxis = svg.append("g")
    .classed("key", true)
    .attr("transform",
          `translate(0,
${svgHeight - bottomMargin})`)
    .call(d3.axisBottom()
        .scale(xScale));
```

- When you use **backticks** to define a **String**, you tell JavaScript to look for the pattern **${}** inside the String
- Anything in the String that's inside of **${}** will be **interpreted as code.**
  - **${** "pauses" String interpretation and goes back to JavaScript interpretation until a matching **}** is reached
    - You can even use curly braces *within* this code, as long as you keep them *balanced* so the *last* one closes the **${}**
- With old String concatenation, this code would be written:
  "translate(0," + (svgHeight - bottomMargin) + ")"

TIP: You can use backticks to specify that something is a String in any place where you create a String in JavaScript (instead of using single or double quotes). This gives your Strings extra flexibility, since you can then use both regular quote marks and apostrophes within the String! And what else are you using backtick for, anyway? ☺

Exploring the axis



- Use the **DOM element picker** to select something on your new **x axis**
- Open the **reveal triangles** until you see the group information for **class = "tick"**

Styling our **tick** elements

- D3 adds some **grouping classes*** to the items that it creates:
  - **domain** for the **line** that runs across the axis
  - **tick** for the **group** of each **tick line** and **value label**
- These aren't terribly well documented, but we need to know they're there. Why?
  - If we created CSS classes named **domain** or **tick** for another purpose, those styles would be applied to these items, too!
  - We can use this knowledge to our benefit, too!
- This is another example of why you should always explore what your code is producing, rather than just accepting the results!

Let's use our knowledge of CSS to affect how D3 draws the axes!

- In **style.css**, add this class:

```
.tick {
    stroke: red;
}
```

- The ticks and value labels turn **red**!
- This is because **tick** class is applied to the **group** of these objects.
- But what if we just want to change the tick **lines**, not the labels?

CSS specificity to the rescue!

- We want to style just our tick lines, not the labels.
- Looking at our DOM element, we can see that each tick **group** contains a **line** and a **text** item:

```
<g class="tick" opacity="1" transform="translate(80.5,0)">
    <line stroke="currentColor" y2="6"></line>
    <text fill="currentColor" y="9" dy="0.71em">0</text>
</g>
```

- In our CSS, we can specify that only any **line** elements affected by **class "tick"** should be turned **red** with this small addition:

```
.tick line {
    stroke: red;
}
```

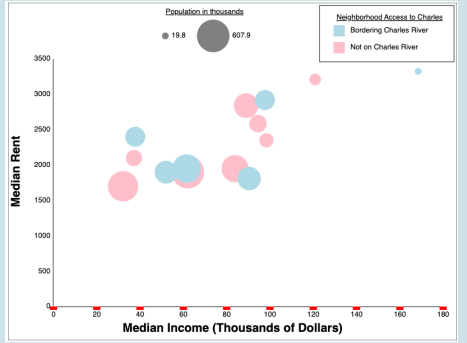
- REMINDER: **.name** is a class selector, **#name** is an id selector, and just **name** is an element selector

Experiment with units on your own!

- Try adding this to your **.tick line** CSS style:

```
stroke-width: 1em;
```

  - The **1** by itself generally means pixels, though modern CSS always prefers that units are specified.
  - As soon as you type the number **1**, VSCode should suggest a bunch of different units to complete it. Try a few!



d3 selections: review

REVIEW ANSWER: the "g" created here is the selection stored in **xAxis**

```
let xAxis = svg
              .append("g")
              .classed("key", true)
              .attr("transform",
                  `translate(0,
${height - margin.bottom})`)
              .call(d3.axisBottom()
              .scale(xScale))
```

TECH NOTE: ***.append()*** *and* ***.join()*** *are also* ***methods*** *of every* ***d3 selection****. Remember, variable* ***svg*** *holds a* ***selection*** *of something that you created earlier (even though you might just think of it as "my SVG variable), so its* ***methods*** *(the things it knows how to do) are also methods of every* ***d3 selection.***

- The d3 **selection** is whatever item (or set of items) d3 is currently operating on
  - When you ask d3 to create a new item (via **.append()** or **.join()**), that item becomes the new **selection**
  - Here, our variable **svg** holds an initial **selection**: the SVG canvas we created
    - Then we **append** our new **group**, so the **group** becomes the **selection**
- **.classed()** and **.attr()** are **methods** (functions) that are a part of every d3 **selection**
  - In other words, they're things that every selection knows how to do. Here, they affect our **group** element because it is the current **selection** at that point.
- REVIEW QUESTION: What **selection** ultimately gets stored in **xAxis?**

The problem **call()** solves

```
let xAxis = svg
              .append("g")
              .classed("key", true)
              .attr("transform",
                  `translate(0, ${height -
margin.bottom})`)
              .call(d3.axisBottom()
              .scale(xScale))
```

The **"g"** object does not know what **axisBottom()** is, but the **d3** global object* does!

Here, we pass our **current selection** (the **"g"** object) to the **d3.axisBottom() method** so that it knows what to work with.

- **d3.axisBottom()** creates our x axis elements for us, but it needs to know where to put these elements
- Normally, we just call a **method** (function) on the **d3 selection** that we want to work with
  - **`svg.append("g")`**
  - **`circles.attr("r", 10)`**
- The **problem**: it's the **d3** object that knows how to make an **axisBottom**, and not the current selection!
- The **solution**: the **.call()** method passes the current selection to another function that is not a method of the selection!

**d3 global object is the container that holds all of d3's code*

.scale() configures **d3.axisBottom()**

```
let xAxis = svg
            .append("g")
            .classed("key", true)
            .attr("transform",
            `translate(0, ${height -
                    margin.bottom})`)
            .call(d3.axisBottom()
                .scale(xScale))
```

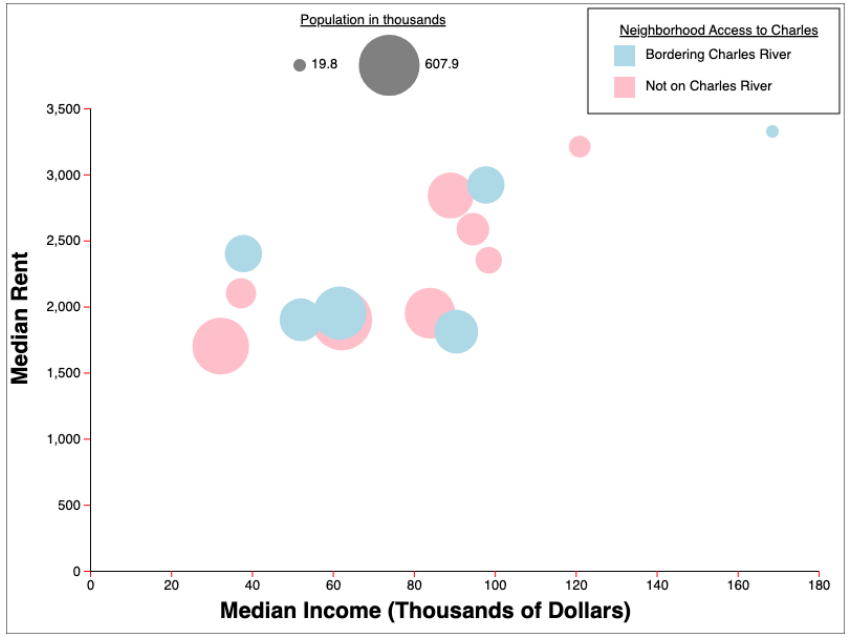
- Note the parenthesis placement!
  - **.scale(xScale)** is **chained** to **d3.axisBottom()**
  - **.call()** calls **d3.axisBottom()**
  - **.scale(xScale)** is called on the object that is returned from **d3.axisBottom()**
    - The returned object is the visual scale

- **d3.axisBottom()** generates the visual axis, including its ticks and labels
- **.scale()** configures the output from **d3.axisBottom()**
  - the same way that **.domain()** and **.range()** configure the output from **d3.scaleLinear()**
  - uses the **.domain** and **range()** from whatever **scale function** it's passed to determine tick & label placement
- We pass the **name** of our **scale function** as the **parameter** to **scale()**.
  - We don't write **xScale()** because we are talking about the function itself, not calling it
    - **xScale** without the invoking parentheses is a **variable** holding the function's code – i.e., the code that **d3.scaleLinear()** generated for us!
  - We say "use this function to get the information you need" and **.scale()** does the rest!

Practice: Create the Y Axis!

- First, comment out the code under `/* y axis line */` and `/* y axis values */`

- To create the Y axis, follow the same model as for the X axis, with the following changes:
  - Call **d3.axisLeft()** instead of **d3.axisBottom()**
  - Tell **d3.axisLeft()** to use your **yScale** function to determine tick values and positioning
  - The result should be translated **horizontally** so that it's at the **leftMargin** position
    - You won't need to translate the axis **vertically** since its origin will be determined by **yScale**
  - Store the results in variable **yAxis**
- Your result should look like this image



Solution: Creating the Y Axis

- The code is the same as **xAxis**, with the changes highlighted below.

```
let yAxis = svg.append("g")
    .classed("key", true)
    .attr("transform", `translate(${leftMargin}, 0)`)
    .call(d3.axisLeft()
            .scale(yScale))
```

- Remember: the goal here is to learn this **pattern** for creating something in D3.
- You don't have to sweat all the details as long as you learn what to change in order to customize the axis as you want!

*POST-CLASS EXPERIMENTS: Try using **.axisRight()** and **.axisTop()** in place of these two axes. Try using this technique to replace our **for()** loop-based axes in other projects (in particular, the one using **d3.scalePoint()** from lab). Try changing the type of scale and its **domain** and **range** values. Play more with the CSS classes. Investigate the resulting SVG objects further in the Elements panel!*

*For additional guidance, check out: **https://d3-graph-gallery.com/graph/custom_axis.html***

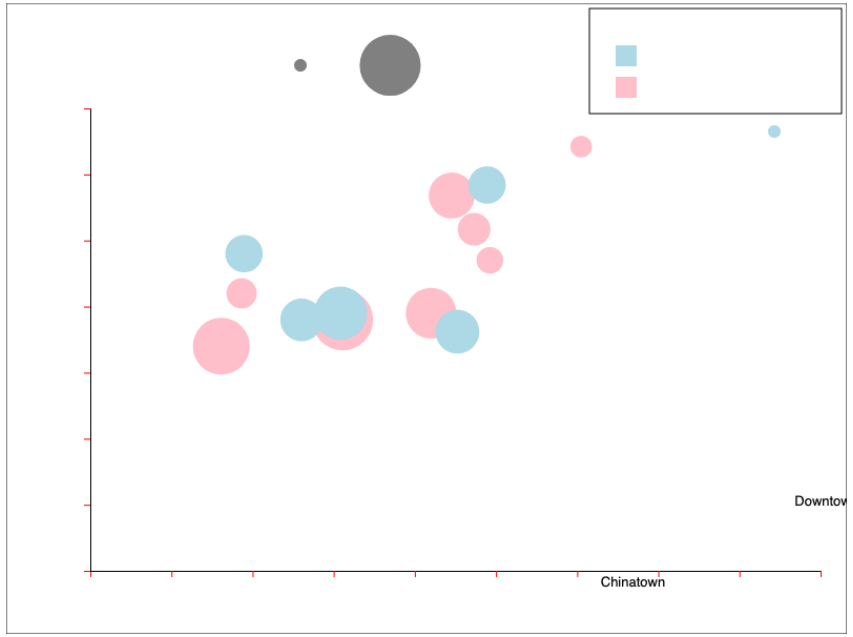
Text labels for data points, or a big mess?

- Uncomment the commented-out code at the very end of **main.js**
  - *Leave the 2 English comments alone, of course!*
- Look at this code. Similar to the code to generate our circles, it:
  - Uses **.join()** to generate one **text** per datum*
  - Positions the text based on **income** and **rent** (our X and Y axis values)
- It also uses **.text()** to set the text to the **area** property (neighborhood names) of each datum
- So what went wrong?
  - How might we investigate?
  - What does **.join()** actually do, again?

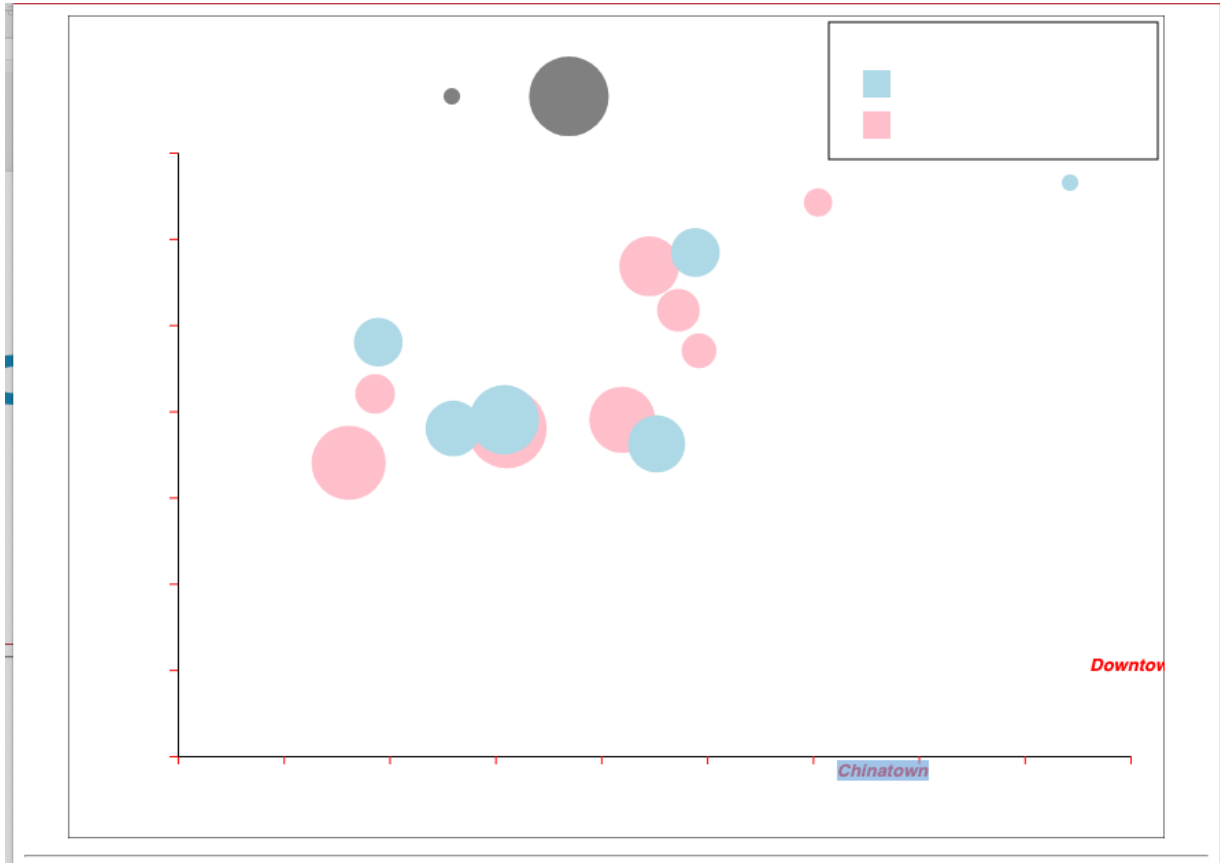
***"datum"** is the term for "one item of data", meaning one full item from your **dataset** list – i.e., a single "observation" containing all properties of your data. D3 also uses **.datum()** to access single items of data, so we will start using this term more in place of "one element of data" or "one data item"*

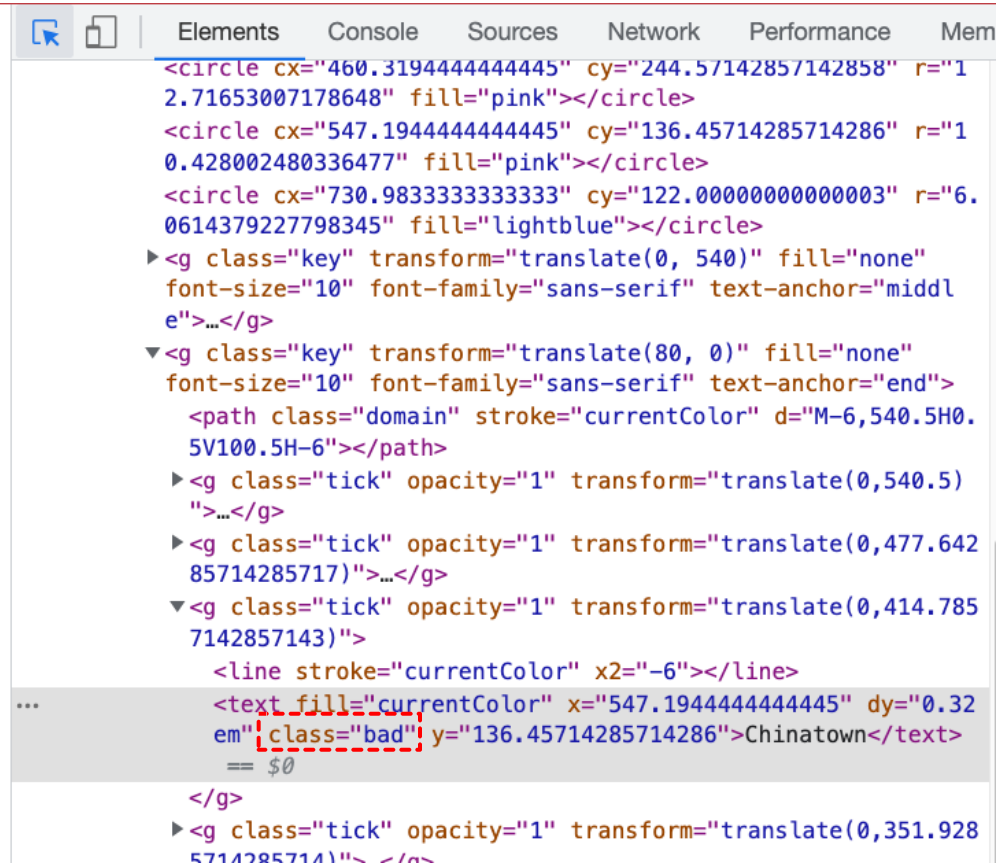
What happened?!

- We **should** investigate this problem using the JavaScript Debugger to single-step through the program.
  - In the interest of time, we won't, but you should **make a habit of doing this**
- You might also **comment out the code** that you just wrote (well, uncommented!) and **observe** exactly what changes
- You might use the **Elements** inspector to see if the missing **text items** even still exist in our SVG (maybe they're invisible or off-screen?)
- To get a **visual** hint, try changing the **class** of this selection from **key** to **bad**, which is a CSS style we set up for you. Then, use the **Elements** pane to see what's going on with our axis value labels!



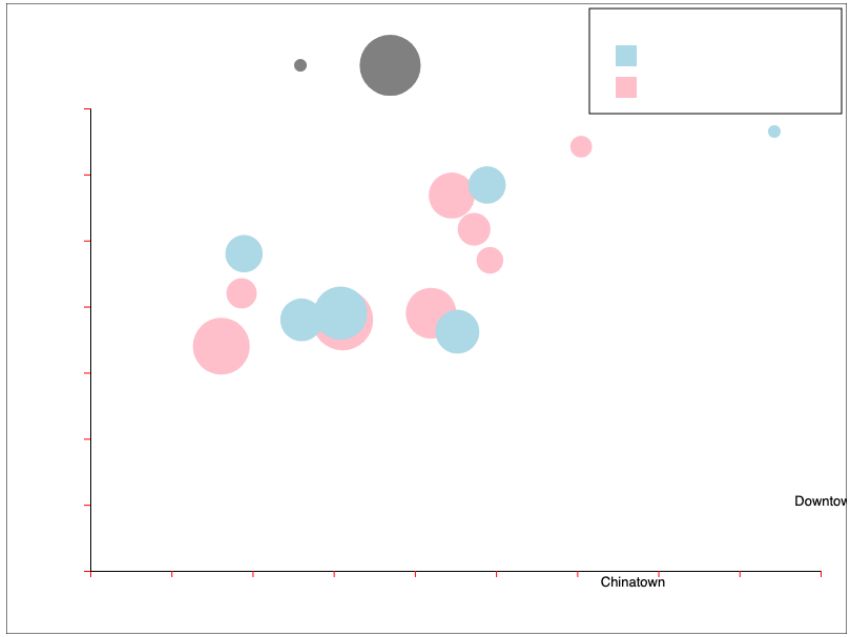
.classed() adds classes, but somehow class "bad" has replaced class "key"? Sort of...





What actually happened

- **.data().join()** tells D3 "make sure there is **exactly one** of the specified **object type** in this **selection** for **each** piece of data"
  - RECALL: When we add data, we get more circles; when we remove data, those circles go away
- When we say **.join("text")**, D3 counts **every "text" item in the SVG** in this decision to add or remove elements!
  - REMINDER: Before our new labelling code runs, we have already created a bunch of "text" objects in our visualization (for our keys and our axes)
- We now see only the **first few** pieces of text that our program created, which were part of our X axis but have been moved by this code's **.attr("x")** and **.attr("y")**
- To **solve** this problem, we need to tell D3 "**only** pay attention to a **specific set** of 'text' objects when syncing them to this data"



Keeping things **class**-y: a **super important** pattern to learn!

Make the changes **highlighted in yellow:**

```
let pointLabels = svg
            .selectAll("text.label")
            .data(dataset)
            .join("text")
            .classed("label", true)
```

- We are now telling D3 to only **select text** items of class **label** to **join** to the **dataset**.
- We give the text items in this **selection** the class **label** so that they're part of the count for any future **join**s on this **selection**

If you test the project now, everything should look good...other than the addition of some mediocre-looking data labels!

KEY POINT: We can use a CSS class to establish a "group" of DOM elements

- This is unrelated to making an SVG "g" object, which is an actual group!
- As with any CSS styling class, "group" formed by our class **.label** could even contain a variety of different DOM element types.
- To keep things simple, create one class for each representation of your data (for example, the circles and the labels in this project)

Adding **class** to your d3 programming pattern

```
let pointLabels = svg
            .selectAll("text.label")
            .data(dataset)
            .join("text")
            .classed("label", true)
```

This pattern is super important!

Starting today, in your projects, use this technique to make different groups of any shape that you draw to represent your data, even if you only use that shape in one place in your visualization. **It is better to over-use this technique than to forget about using it!**

(TIP: You will not lose points for using this pattern where it's not needed, as long as you use it right!)

- **.label** is just a standard CSS class selector: "select all text of class **label**".
  - It could be called anything!
- Since we specify a type of DOM object to check and a class, the number of data items will be synced only to the count of items of this type and in this class
- **.classed("labels", true)** adds this CSS class to our newly created text so that it's counted as part of the group
  - If we updated our data and asked D3 to re-run this code to update our chart, it would need to know what **text.labels** had already been added!

Assign a unique class to every set of shapes in your visualization!

Template:

```
selectionName.selectAll("elementName.className")
    .data(datasetName)
    .join("elementName")
    .classed("className", true)
```

Example:

```
let circles = svg.selectAll("circle.population")
    .data(dataset)
    .join("circle")
    .classed("population", true)
```

LEARN MORE ABOUT IT**! There are lots of other ways to make selections and sub-selections! Be sure to check out https://devdocs.io/d3~7/d3-selection for more information and **cite that page if you use any of the tecniques shown!

Important Notes:

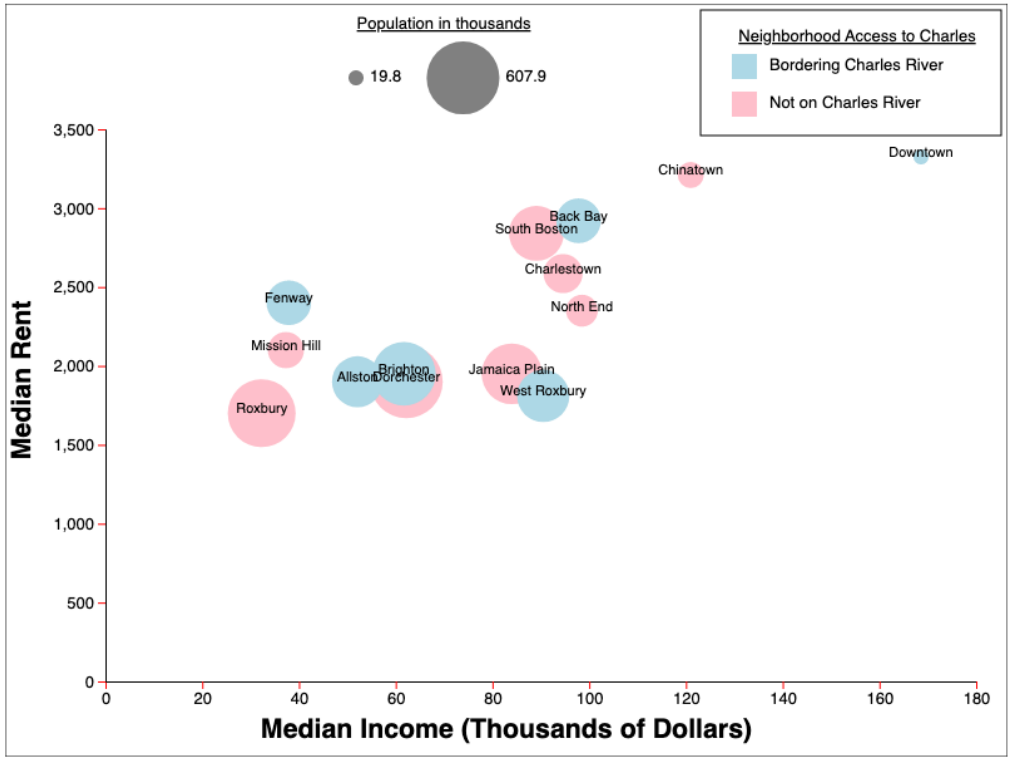
- **selectionName** can be any D3 selection. Generally you've referred to your drawing canvas, but you can refer to any selection that you've stored.
  - For example, after the code on the left executes, variable **labels** will contain the selection of all **text.labels**, so you could select from a subset of those via **labels.selectAll()**
- It is **optional** to store the result of a **D3 selection** in a **variable**, but for this class, it is good practice to do so. *A well-named variable gives you another way of clearly indicating what your code does!*
- The period preceding **className** indicates that it is a CSS class. It is not the same as the period between an object and a property.
  - For example, you could write **svg.selectAll("text .labels .reds")** to select **text** elements that have both classes: **reds** and **labels**

Improving the label style

- So far, we are just using this CSS class for grouping, so it doesn't even need a definition in our **style.css**
- However, we can use the class to improve the style, too!
- In **style.css**, add:

```
.label {
    font-family: sans-serif;
    font-size: 11px;
    text-anchor: middle;
}
```

- You should see some immediate improvement in the labels: they're centered and in matching font!



OPTIONAL: CSS vs JavaScript for centering

We know we can center text using CSS, but if you want to focus on using JavaScript wherever possible, there's an alternative! Try this:

- In **style.css**, comment out **text-anchor:middle;** in your **.label** class
  - CSS comments must be **/* */**, not **//**, but the **command-/** shortcut will do this for you in CSS files!
  - We have to do this because we will shift our text from the default text-anchor position (**left**)
- Change the code that sets our label text's **"x"** attribute to this:

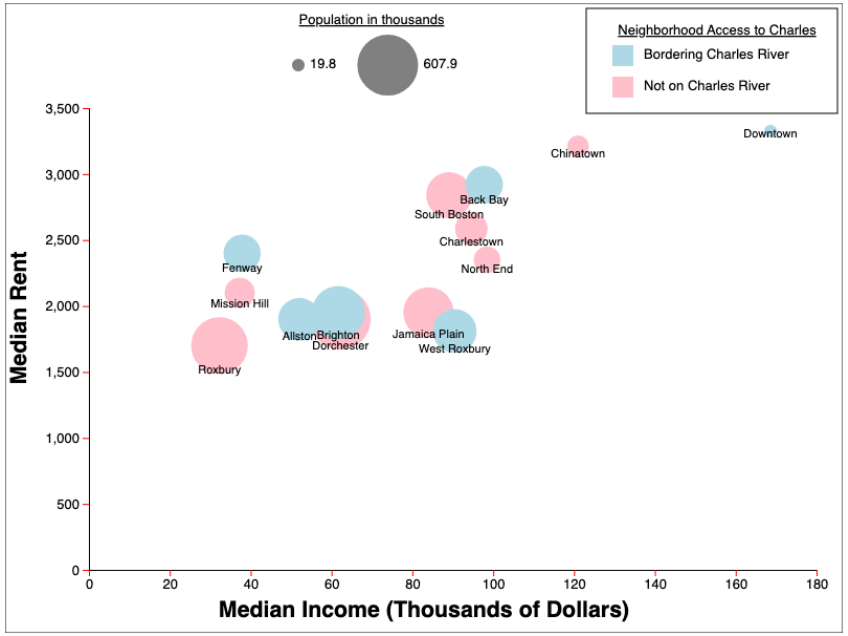
```
.attr("x", function (value) {
    return xScale(value.income) - this.getComputedTextLength() / 2;
})
```

  - Remember, **.getComputedTextLength()** tells us the width of the text that **has already been placed** in the "text" object; we can't run this code before **.text()**
  - This line of code says "figure out how wide the text is, then shift the text to the left (*subtraction*!) by half the text's width (*division*!)".
    - This puts the **center** of the text in the same position as **xScale(value.income)**

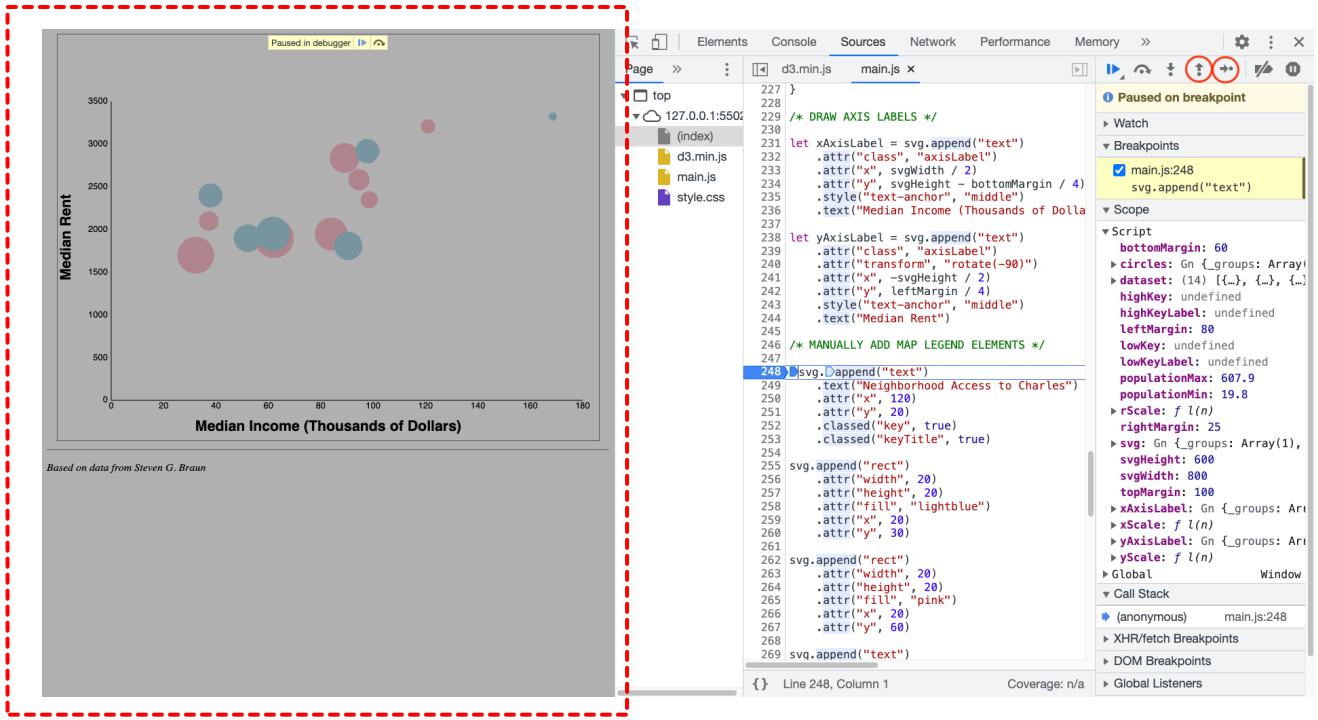
Labels below the data

You might like the labels centered on the circles, but if you had smaller data points, the labels might obscure them!

- The **accessor functions** that d3 uses to set object **attributes** can perform any calculations you want; d3 only cares about the **return** value
- When you access **value**, you're getting **all** of the data and picking what you need.
  - So far, we've been using just **one** property of data to determine a given **attribute**
- Try to modify the **accessor function** for **.attr("y")** to say "place the label at the bottom of the circle"
  - HINT: What data did we use to determine our circle sizes, and how did we use it? Could we use that same data and calculation to help? Refer back to the **circles** code for for some ideas!



debugger "two-step" to "slow motion" your d3



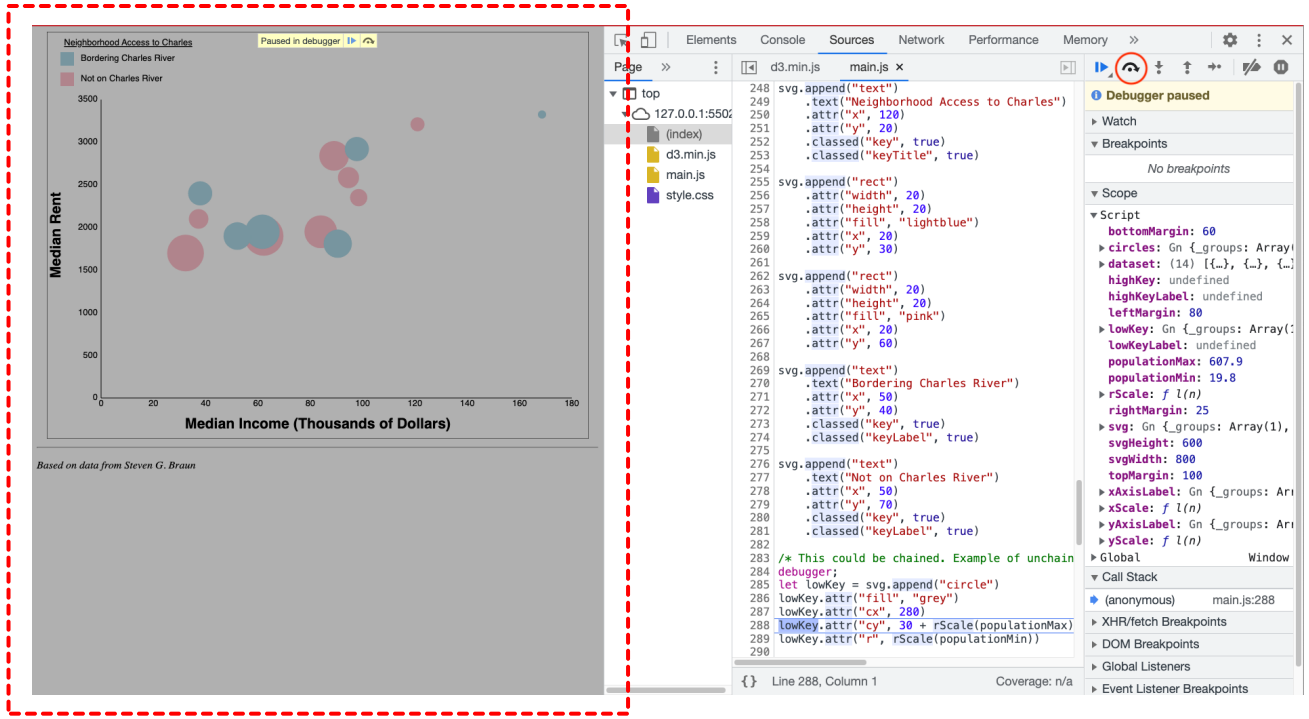
The debugger is a great way to "step through" what's happening in your visualization! Watch how your visualization changes with each step!

PROBLEM: "chained" commands all execute as a single step in the debugger if you use **step over**

SOLUTION:

1. Set a **breakpoint** or use **debugger;** keyword in your code
2. Make sure that you can see your visualization while you do this!
3. Use the **step** button in the Debugger to see the effect of each **.attr()**, **.classed()**, **.style()**, etc.
4. Use the **step out** button to get out of the **d3.min.js** code and back to your code
5. Click back and forth between these buttons to see your visualization gradually build up!

slo-mo alternative: "unchain" your code



"chained" methods operate on the object returned from the previous method. With D3, this generally means the same thing that was just operated on.

You can instead store your **selection** (generally the first line of any D3 operation) in a **variable** and then call your D3 methods on the variable on every line.

An example of this can be seen in the code that sets up the **lowKey** circle in today's project.

This method of single-stepping your project takes a bit more work to set up, but then you can just use the **step over** button to watch each step.

PRACTICE: Group & Move Population Key

- This week, try to turn the population key for this project into a moveable SVG group
  - The group will contain the 2 gray circles, their labels, and the title
- Remember, you want the origin of the group to be 0,0, so objects should be drawn relatively close to the "edge" of the group
  - Currently, **lowKey** is drawn 280 pixels from the left; that number probably should be smaller!
  - **lowKey** is drawn 30 pixels from the top of the SVG, in addition to the shift down to accommodate the size of the bigger circle; we still want to accommodate the bigger circle and we also need to accommodate the title for this key, so maybe 30 pixels is OK here?
- **lowKeyLabel, highKey,** and **highKeyLabel** are already drawn relative to **lowKey**, so they should all just work once you get **lowKey** adjusted
- The **Population in thousands** key title is currently drawn at a fixed position on the SVG
  - **Easy:** Adjust the numbers to reposition it relative to **lowKey**
  - **More Practical:** Change the numbers to calculations so that the key title is positioned **relative to** the other objects in the new SVG group
- Test your setup by moving this new group using **.attr("transform", "translate()")**