

# Working with data: **d3.min()** & *Array.sort()*

Class #7

rev. 2022.10.31.a

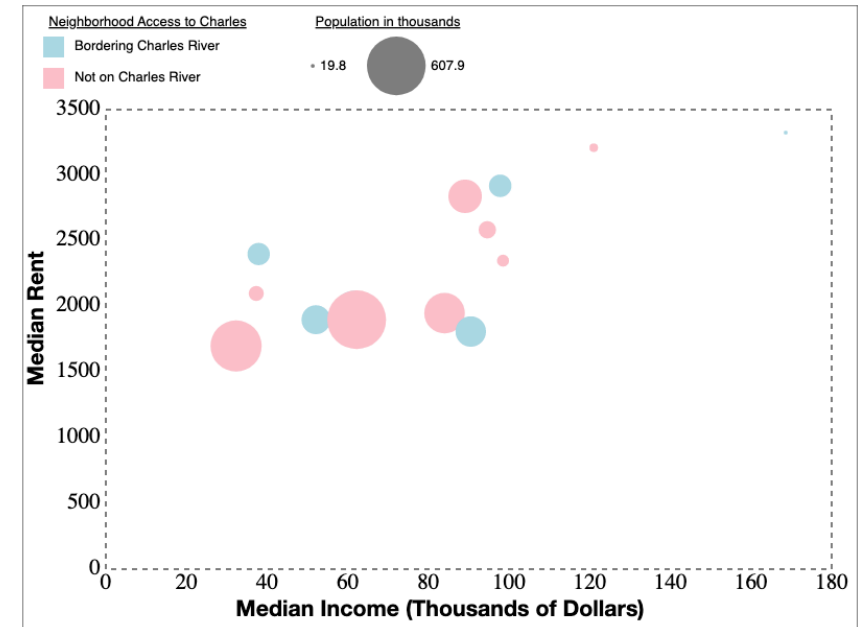
# Don't panic! (and don't cheat!)

*\* That is, unless we think you're **cheating** and have to give you an **exam** so you can **prove** that you know the material! **Please** don't make us resort to this. It isn't fun for anyone!*

- The key to learning D3.js is to learn the **patterns** of coding with it.
  - You have already learned several patterns for things such as scaling and positioning your data on the screen. You will add to this list of patterns each class session.
  - Occasionally we will teach you something that **expands on** or **replace** an earlier pattern with something a bit more complex, usually to give you more control over what your code is doing.
    - This is to help you **build good coding habits** without overwhelming you at first. Just be sure that you **understand why we are making each change**, even if the code seems a bit intimidating at first!
      - You need to **understand** enough about the code does so that you can modify it for your own use, but **you don't have to memorize** how any of this works!\*
- Just **don't copy code** (even from class demos!) **without understanding how it works!**
  - In this class, **you will always have to explain your code** in comments, presentations, and essays, so be sure that you understand the code you're using!
    - **Academic Integrity** also requires that you **credit all code sources**, so if you use even part of an example from anywhere or anyone, **you must document that as well!**

# Today's Project

- Unzip **scatterplots\_part2\_starter.zip**
- Rename the folder **w7\_class\_scatterplot**
  - Always immediately name your projects meaningfully so that you can easily find them later!
- Open the renamed project in VSCode and preview using **Live Server**
- Scan through the source code in **main.js**
  - The only elements that may be new to you are **.classed()** and **.style()**, which we'll cover today
  - Some of the code may **appear** more complicated than what you've done so far, but it's **entirely composed** of things that you've **learned already**!
- We'll review key concepts from last week as we explore this project



# freedom from the tyranny of the semicolon!

## *(in a responsible manner)*

- ▶ You may notice that today's project has far fewer semicolons than past ones
- ▶ In modern JavaScript, semicolons to end lines\* are usually optional:
  - ▶ Modern JavaScript **parsers** (the computer code that actually interprets your JavaScript) figure out where the "end of lines" are based on **context**
  - ▶ If you write entirely well-formed JavaScript, you don't need semicolons
  - ▶ **Benefit:** Writing without semicolons means that you don't have to remember to remove them if you decide to add to a line of code
- ▶ However, semicolons may still be useful to you:
  - ▶ Help to remind you where you intended to end your JavaScript line
  - ▶ Helps parser to report correct line number for many "unexpected end of line" errors
  - ▶ Helps VSCode auto-formatter to understand your formatting style

\* They are still required other places, such as between the 3 parts of a **for** loop declaration!

# Margins: making better use of our space

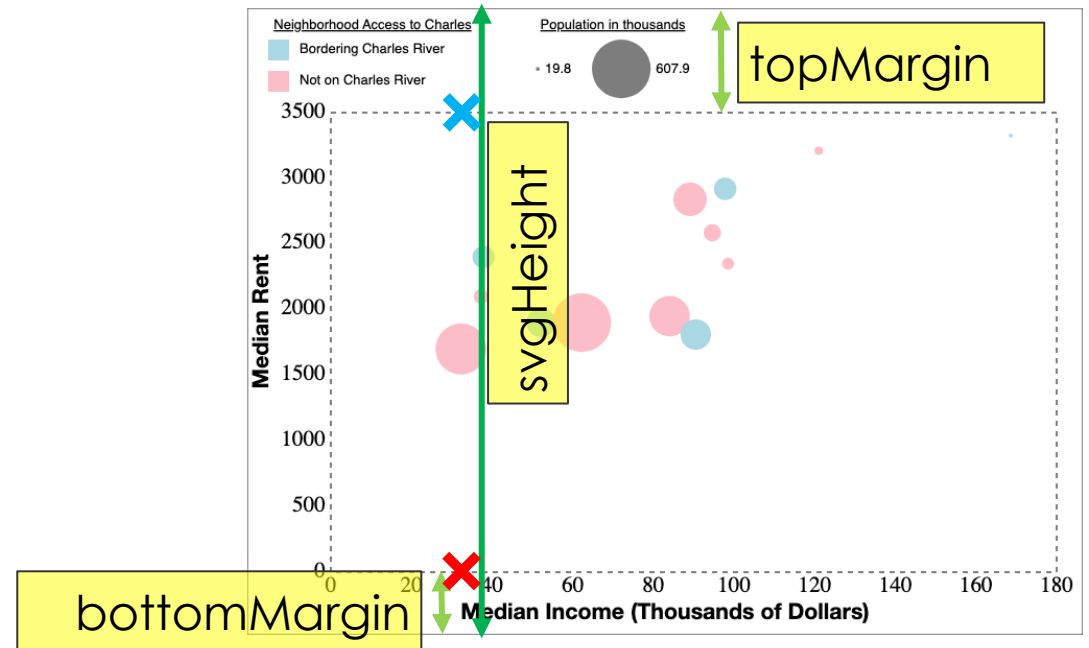
- Four margin variables this time
- Only difference is now you need to think about which margin(s) affect each calculation

- Example:

```
let svgHeight = 600  
let topMargin = 100  
let bottomMargin = 60
```

```
let yScale = d3.scaleLinear()  
  .domain([0, 3500])  
  .range([svgHeight - bottomMargin, topMargin])
```

- In the illustration:
  - Color-coded **x** for each **range** parameter
  - Labeled arrows show part of SVG represented by each variable



# Code folding: hide your **dataset**

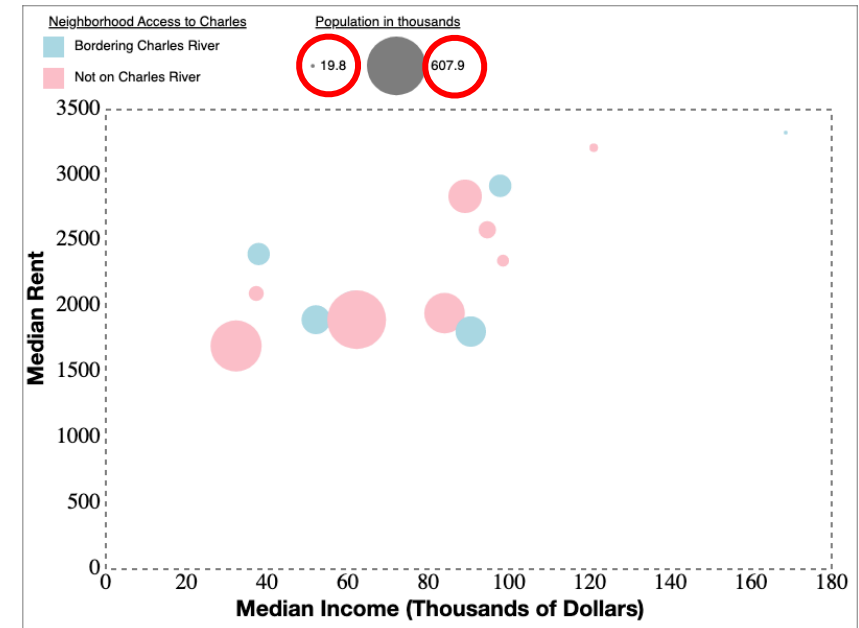
- Click the turned-down arrow next to any **code block** to hide the code
- Arrow turns **sideways**, just like a "reveal triangle" in your computer's file system
- **Line numbering doesn't change**, so you see that there's hidden code where numbers skip!
  - *TIP: If you ever think you've "lost" some code, check that you didn't accidentally hide it!*
- By folding the code at **let dataset:**
  - We still can see where the variable is declared
  - We can see a lot more of our actual code
  - We can always "unfold" the code if we need to check something in our dataset

```
38  /* Define data */
39  ▾ let dataset = [{
40      area: "Back Bay",
41      rent: 2920,
42      income: 97.8,
43      population: 218.8,
44      charles: true
45  },
46  ▾ {
47      area: "Charlestown",
48      rent: 2585,
49      income: 94.6,
50      population: 164.4,
```

```
• 38  /* Define data */
39  > let dataset = [{...
137  }];
138
```

# Using d3 to find minimums and maximums

- Key shows the circle sizes for smallest and largest populations in current data
- What if we added more data?
  - We've at least created **configuration variables** so that we'd only have to change these numbers in one place (hopefully)!
  - We still have to go through our whole dataset to figure out lowest and highest values, though!
- D3 provides several functions to automate this process: **d3.min()**, **d3.max()**, and **d3.extent()**
  - *There are also variations on these functions if you need to find out **what position** (i.e., **index**) in your list holds the minimum or maximum value:*  
**d3.minIndex()** and **d3.maxIndex()**



# Using `d3.min()` to set **populationMin**

- Format:

```
d3.min(data, accessorFunction);
```

- Example:

```
d3.min(dataset, function(value) {return value.population;});
```

- First parameter is your **data**
  - Often the same as what you provide to **.data** when drawing
- Second parameter is an **accessor function**
  - Same form as accessor functions when drawing: automatically takes each **value** (and optionally **index**) from the list as **parameters**, then **returns** a value based on that information
  - Here, we just grab the **population** property from each item in our dataset
- TRY IT: Enter the above example code in place of the number **19.8** in order to set **populationMin**. You should see no change to your visualization (but also no errors!)



# Practice: `d3.max()`

- Format:

```
d3.max(data, accessorFunction);
```

- Works identically to `d3.min()`, but finds the maximum value from the dataset
- Write code similar to the previous example that will:
  - get the **maximum value** of **population** found in our **dataset**
  - set variable **populationMax** to this value (instead of to 607.9)
- As before, you should see no change to your visualization (but also no errors!)

# Where else can we use this information?

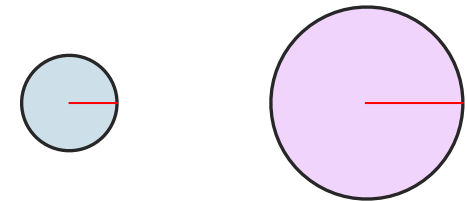
- Scaling function **rScale** currently allows for a population as low as **0** (unlikely!) or as high as **650** (which isn't a bad guess, but we have a more accurate number now!)
- Try modifying the definition for **rScale** so that its **domain** is instead determined by **populationMin** and **populationMax**
- If you succeed in doing this, you won't get any errors, but your **key** for the neighborhood with a population of only **19.8** should become almost unreadably small!
  - Any ideas on how to adjust the **domain** values to produce reasonably sized circles?
  - What **other values** can we adjust in our scale setup to affect circle size?
- After you're done experimenting, set the **domain** back to **[0,650]**
  - Also reset the **range** to **[1,30]** if you changed it
  - Next, we'll look at a different approach to improving our circle scaling!

# d3.scaleSqrt(): proportional circle growth

- Doubling a circle's radius quadruples its area
- Area =  $\pi * \text{radius} * \text{radius}$
- To follow the [rule of proportional ink](#), each circle should take up an overall amount of space that is proportional to the value it represents
- This is a simple change – just one word!

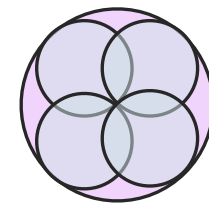
```
let rScale = d3.scaleLinearscaleSqrt()
```

- Look at the circles in the key and in the visualization. What's changed?
  - Undo and redo this code change to compare!
- When comparing multiple circles:
  - **scaleLinear** exaggerates value differences
  - **scaleSqrt** accurately represents value differences



$$r = 0.5$$
$$A = 0.25\pi$$

$$r = 1.0$$
$$A = 1\pi$$

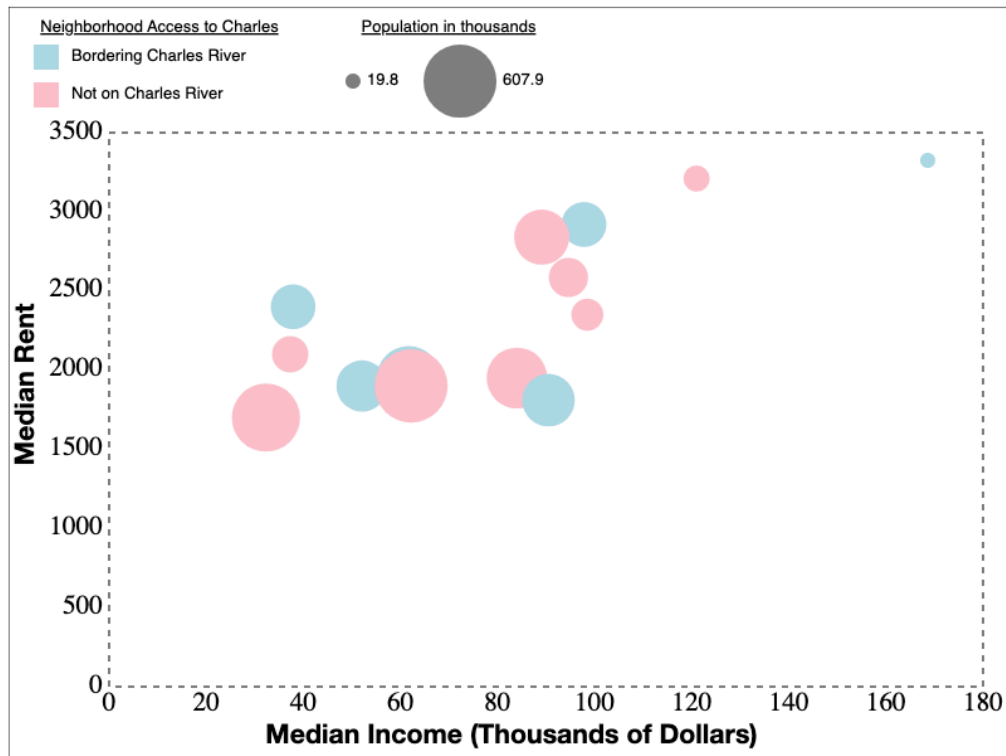


(if cut apart, the overlapping bits of the 4 blue circles would fit in the rest of the purple circle)

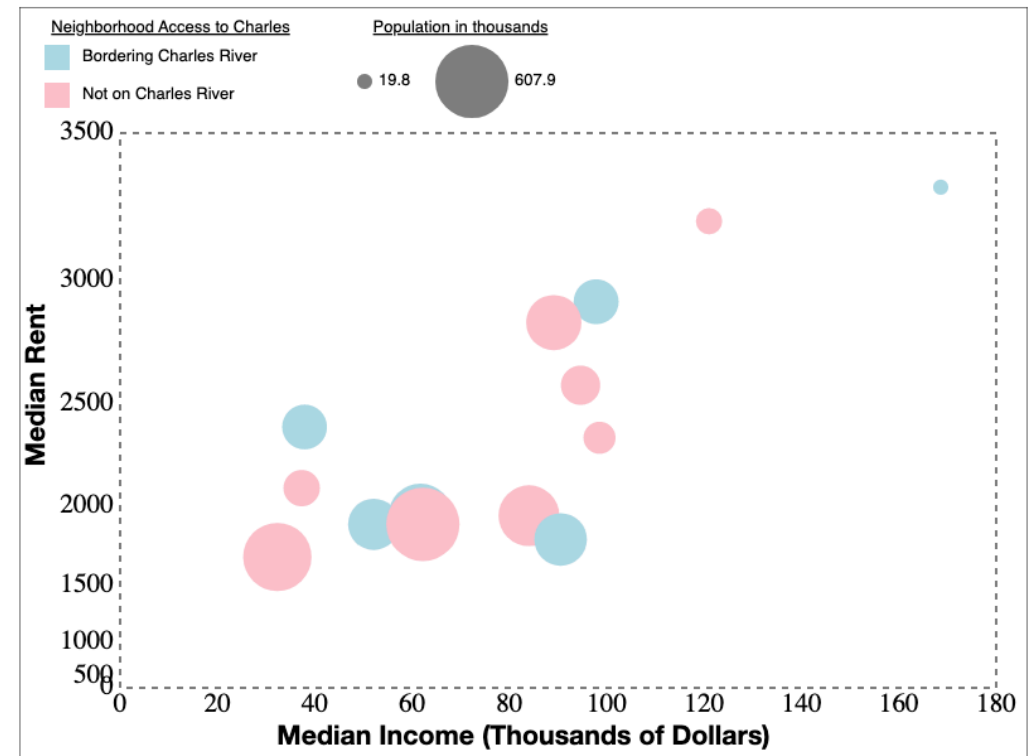
# An experiment with yScale

Change the scaling function type for your **yScale**. Note that **scalePow()** requires an additional configuration method: **.exponent()**. Can you explain what it does? Experiment and observe!

`yScale = d3.scaleLinear()`

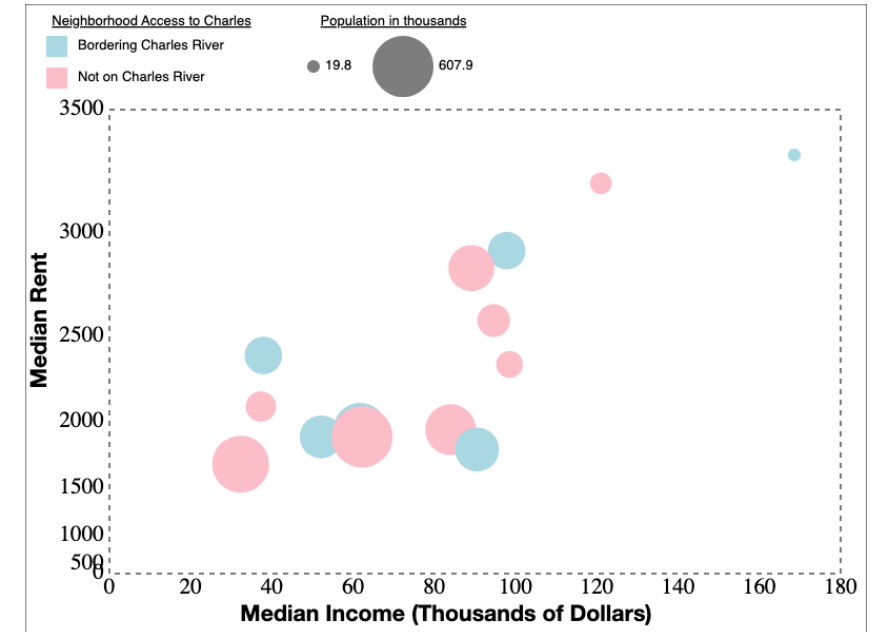


`yScale = d3.scalePow().exponent(2)`



# scalePow(): an exponential scale

- Like **scaleSqrt()**, **scalePow()** adjusts the relationship between the **domain** and the **range**
- Domain values are raised to the exponent given in the **.exponent()** configuration method
  - `let yScale = d3.scalePow().exponent(2)`
  - "raised to the power of 2" is often referred to as "squared" and is the most common exponent used
  - Now our visualization's Y axis has each regularly spaced number (the **domain**: 500, 1000, 1500, etc.) positioned **twice as far away** as the previous one
    - Compare the difference in positions for 500 and 1000 to the difference in positions for 1000 and 1500
- **scaleSqrt()** is actually just a special case of **scalePow()**. Just change **one value** and **scalePow()** will perform identically to **scaleSqrt()**



**KEY POINT:** **scalePow()** is useful when you have a lot of data clustered at the **high end** of an axis and need to **spread** it out. Just be careful that you aren't **squishing** the **low end** of your axis too much (as we do here!)

# scale practice: draw the X axis line

- Change your **yScale** back to a **scaleLinear()** function. Be sure to remove **.exponent()**
- **Comment out** the code that draws the dashed margin border  
(near top of **main.js** under */\* Draw margin border \*/* comment)
- Under the comment */\* TO DO: Draw Axis Lines & remove margin border \*/* write code that:
  - draws a **line** marking the border of the X axis
    - horizontally extends between **median income = 0** and **median income = 180**
    - is positioned vertically where **median rent = 0**
- Template for code to draw a black line:

```
svg.append("line")  
  .attr("x1", xValue1)  
  .attr("y1", yValue1)  
  .attr("x2", xValue2)  
  .attr("y2", yValue2)  
  .attr("stroke", "black")
```

**MAIN TASK:** At a minimum, use the **functions** that we've been working with to convert the above **data** to **positions** for the two points that define this line.

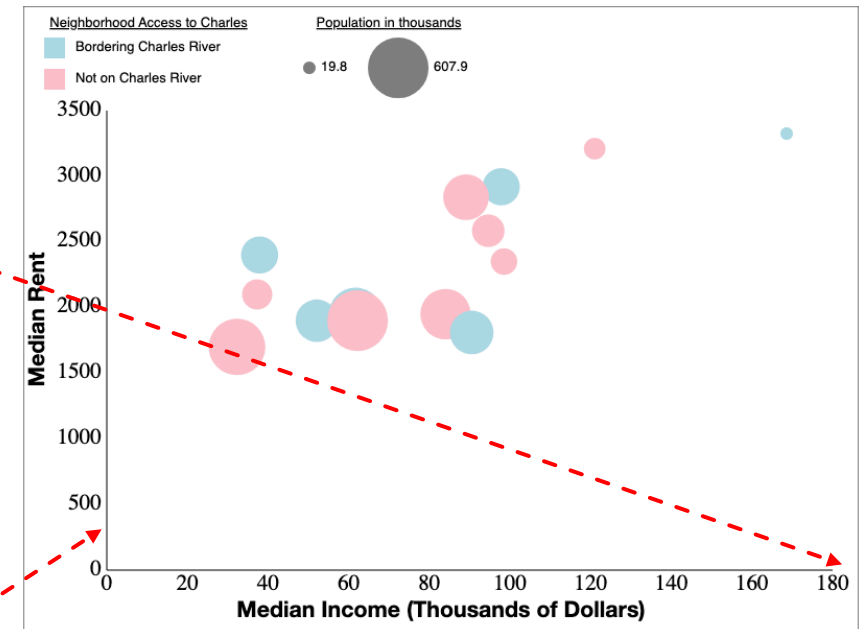
**APPLY MORE OF WHAT YOU KNOW:** What might we add to the project so that we don't have to search through our code for the values we've chosen to define the ends of our axes?

# solution & scale practice: draw the axis lines

- Here is the code to draw our x axis line:

```
svg.append("line")  
  .attr("x1", xScale(0))  
  .attr("y1", yScale(0))  
  .attr("x2", xScale(180))  
  .attr("y2", yScale(0))  
  .attr("stroke", "black")
```

- We use our scale functions same as on our data to ensure the positioning matches
  - We could define some additional **configuration variables** to establish the low and high values for each scale— either just by setting the values for later use, or by using **d3.min()** and/or **d3.max()**, as we did for **populationMin** and **populationMax**
- Try this exercise again to draw a **y axis line**, as shown here.



# solution: draw the y axis line

- Here is the code to draw our y axis line:

```
svg.append("line")  
    .attr("x1", xScale(0))  
    .attr("y1", yScale(0))  
    .attr("x2", xScale(0))  
    .attr("y2", yScale(3500))  
    .attr("stroke", "black")
```

- Note that we did not have to refer to our **margin** variables at all!
  - Our **scale functions** for **positioning** each take into account the margins already
  - By doing this careful setup, we know we can position anything relative to our data
- Now, we only need to refer to our **margin** variables when drawing elements **outside of** the visualization data area or **unrelated to** the data's positioning!

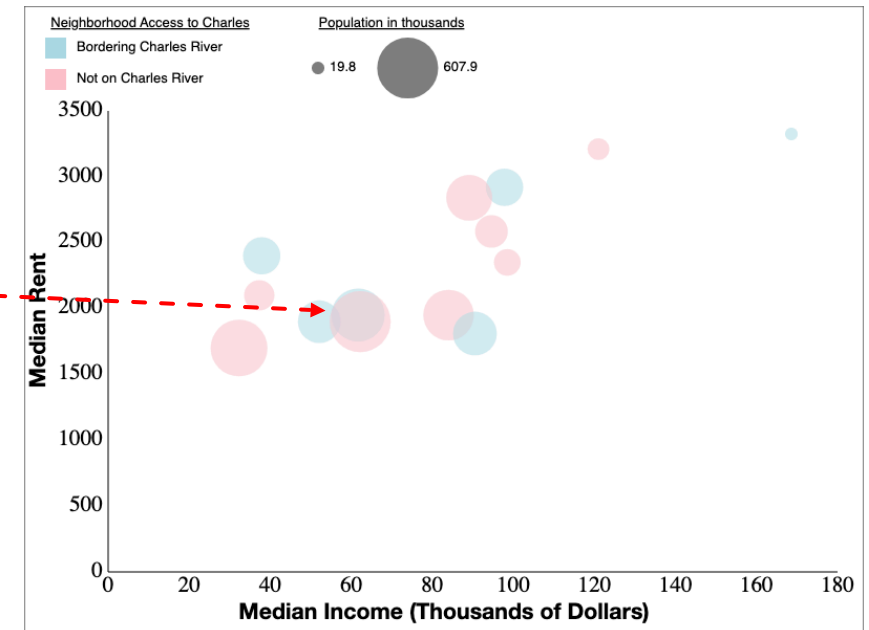


# checking for hidden data

- Our scatterplot looks pretty good, but are we seeing everything that we're supposed to see?
- Adjust the **circles** by temporarily adding one more attribute:

```
.attr("opacity", 0.5)
```

- You should now see that a blue circle is almost completely hidden behind a pink one!!
- Later, you will learn additional ways to distribute your data to avoid such overlaps
- Even now, you should only use the opacity trick as a short-term solution because:
  - it produces colors that don't quite match your key
  - any overlaps produce the illusion of additional colors that don't appear in your key at all



# stating our problem

*\* there are ways to change the stacking order, but this is the default behavior that you've observed since Week 1*

Let's **decompose** the problem by starting with the most **general problem** (what we **see** is happening) and then breaking it down into **smaller** "technical problems" based on what we know how to do... and what we don't yet know how to do!

- **PROBLEM:** Some data is hidden by other data being drawn on top of it
  - SOLUTION: Figure out how to ensure that bigger circles are drawn before smaller ones
- **WHAT WE KNOW:**
  - When shapes overlap on an SVG, the shape drawn **later** will appear **in front of** the shape drawn earlier, as if stacked\*
  - The **population** property of our data determines the **radius** of each circle
- **WHAT WE NEED TO LEARN:**
  - How to re-organize our data from largest population to smallest
    - IN OTHER WORDS: How do we **sort** our data, which is currently an **Array** of **Objects**

The result is a list of things that we can **research**, with specific **terms** that we can search on!

# "Rubber Ducking"

- See <https://youtu.be/WahYlxs9Ns0>
- Updated "things to do before asking for help"\*:
  - Have you checked the **JavaScript Console**?
  - Have you used the **Chrome Debugger**?
  - Have you used the **DOM Inspector**?  
(*Chrome Elements panel*)
  - Have you **talked about it** with your rubber duck?  
(*or a non-coder friend, a pet, or a stuffed animal*)
- To support this:
  - Start your projects (*and ask for help*) **early**!
  - **Experiment** (*and back up your work*) constantly!
  - Practice practice practice (**challenge** yourself)!

\* The instructional team is **here to help** you, so please **don't be afraid to ask us!** If you **try these four things first**, you'll be better prepared to **help us to help you!**



# Sorting with `Array.sort()`

- `.sort()` method can be called on any Array
- Reorganizes Array data based on a **comparison function** that you specify
  - Data sorted "in place", so Array is permanently changed to the new order!
    - Later in your term project, to get the effects that you want, you may need to sort your data multiple times or sort multiple copies of your data different ways
  - Function compares **two elements of data** at a time
    - If function returns a **negative number or 0**, then the current order of compared data is maintained
    - If function returns a **positive number**, then the two compared data elements **swap positions**

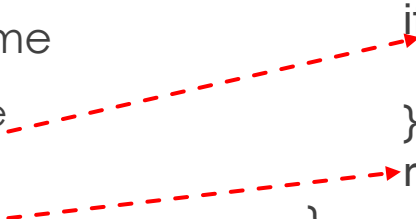
NOTE: `.sort()` has a default behaviour that does not require a comparison function, but it is fairly useless for our purposes, since it can't sort Arrays of Objects— so for this project, always plan to write a comparison function!

- **comparison function** is the **parameter** for `.sort()`
- `.sort()`, is **called** on the Array to be sorted:

`arrayName.sort(comparisonFunc)`

- Comparison function template:

```
function (a,b) {  
    if (a <= b) {  
        return -1;  
    }  
    return 1;  
}
```



You can also reverse the conditions and check **if (a > b)** and **return 1** first, if that makes more logical sense!

# Sorting practice #1: Numbers

- Save your work on our current project, just in case!
- Open a new VSCode window (so we can have both projects open)
- Unzip and open project **sort\_functions\_starter.zip**. Test the project.
  - Currently the "before" and "after" output are identical.
    - We will change this, one dataset at a time!
- Under the code comment */\*\*\*\* Write your sort functions here \*\*\*\*/*
  - Write code to **sort** Array **data1** from **lowest** to **highest** number
  - Then, **modify** this code to sort the data from **highest** to **lowest** number
- *TIP: If you keep the project running, then each time you save your work, you can see the updated output in Chrome!*

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

# Solution: Sorting Numbers

## Sort ascending:

```
data1.sort(function (a, b) {  
  if (a <= b) {  
    return -1  
  }  
  return 1  
})
```

## Sort descending:

```
data1.sort(function (a, b) {  
  if (a > b) {  
    return -1  
  }  
  return 1  
})
```

or

```
data1.sort(function (a, b) {  
  if (a <= b) {  
    return 1  
  }  
  return -1  
})
```

- To sort Numbers in ascending order, we just follow the model code
- To sort Numbers in descending order, we can either:
  - Reverse our conditional logic or
  - Swap the two return values
  - In either case, the result is "swap the two values if **a** is **smaller**"; it's just a matter of which phrasing makes more logical sense to you!

FOR LATER EXPERIMENTATION AND RESEARCH: Try sorting **data1** without providing a comparison function (no parameters to **sort()**). What happens to the numerical order? [This MDN page](#) will help you to understand!

# Sorting Practice #2: Alphabetizing

- Beneath your sort for **data1**:
  - Try sorting **data2** alphabetically, A to Z
    - Your first attempt likely won't produce the result that you want!
      - This is because computers sort uppercase letters before lowercase letters
        - This is because of ASCII (the American Standard Code for Information Exchange), which remains a subset of the more modern UTF-8 (which accommodates worldwide alphabets and symbols)
  - How can we tell JavaScript to compare these Strings without considering case?
    - *REMINDER: You have learned String methods that enable you to convert between cases. You can review that information [here](#) if needed.*
- When sorting **Strings**, "less than" means "this item appears earlier in the current character set than the item it's being compared to"
- These statements are all **true**
  - "a" < "b"
  - "z" > "y"
  - "A" < "a"

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

# Solution: Alphabetizing

## Returns "A, C, b, d" (wrong):

```
data2.sort(function (a, b) {  
  if (a <= b) {  
    return -1  
  }  
  return 1  
})
```

## Returns "A, b, C, d" (right):

```
data2.sort(function (a, b) {  
  if (a.toUpperCase() <= b.toUpperCase()) {  
    return -1  
  }  
  return 1  
})
```

- **.toUpperCase()** only changes the case of the data for purposes of the comparison.
  - When the data is output, we see the values in their originally written cases.
- Sometimes you want to transform your data to make it easier to read for the user; other times, you just want to transform it to make it easier to read for the computer
- Challenge Yourself: After class, try to sort data2 in reverse order, Z to A
  - Extra challenge: Try writing a loop to permanently change all values in Array **data2** to uppercase



# Sorting Practice #3: Object Properties

- Beneath your code to sort **data2**:
  - Write code to sort **data3** by age
  - Remember:
    - the values **a** and **b** in the model function are just automatic parameters that could have any name (just like **value** and **index** in your **d3 accessor functions**)
    - the values passed to the function are two sequential items from your Array, which are in this case Objects
    - You know how to get the property of an Object using dot syntax. It's the same idea as when you write **value.age** in a d3 accessor function!
- Instead of looking back at earlier slides, try to use your earlier **sort** functions as models for this one!

## FOR THE FUTURE:

- This is the sort that you'll most likely do at some point in your term project!
- Your term project may need to sort on multiple items (for example, alphabetizing names for people of the same age)
- This can be done by sorting parts of the data multiple times and/or by writing more complex conditional logic for your sort

# Solution: Sorting by Object Properties

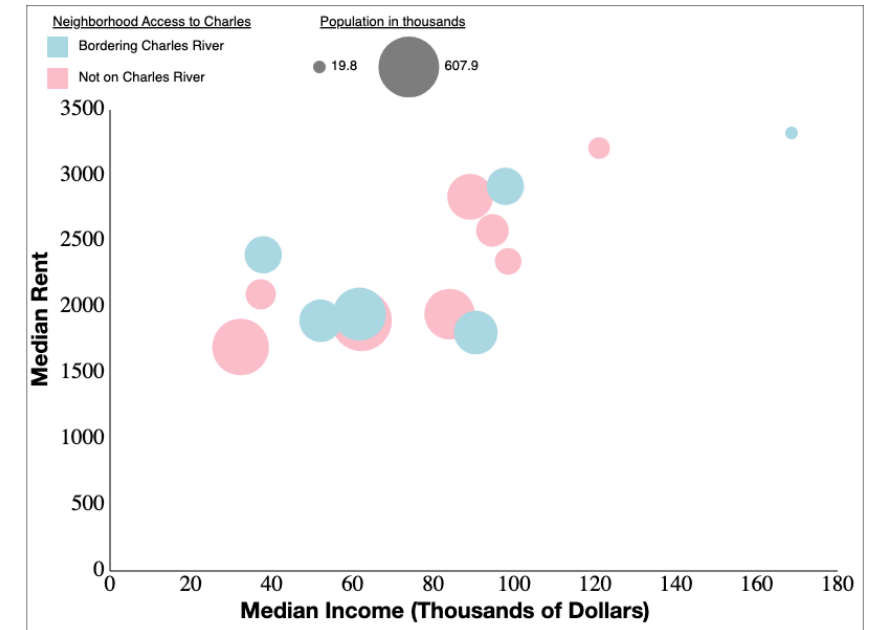
## + Additional Practice Problems

```
data3.sort(function (a,b) {  
  if (a.age <= b.age) {  
    return -1  
  }  
  return 1  
})
```

- Additional practice problems for later review:
  - Sort **data3** alphabetically, A to Z
  - Sort **data3** by weight, highest to lowest
- Advanced challenges:
  - Add more data to the **data3**, but make all new names start with the same letter. Then, sort **data3** first alphabetically, then by lowest to highest weight (so within your new data, weights are in ascending order, but alphabetizing is maintained)
  - Add last names to the **name** property in **data3**. Then, sort the data alphabetically by last name, using a String method to temporarily separate the last names from the first!

# Applying `sort()` to our project

- Return to our scatterplots project in VSCode
- Under the code comment `/** ** PREPROCESS DATA ** */`
  - Write a `sort()` on `dataset` so that it's sorted by **population** in **descending order**
  - If you do this correctly, the blue circle should come out of hiding!
- Advanced questions to consider now:
  - Now that our data is sorted by population, could we set our **populationMin** and **populationMax** a simpler way?
  - Since `sort()` only swaps data when its **comparison function** returns a **positive number**, can you think of a simple math expression that could replace the conditional logic in our function?



# Solutions: sorting by population

## conditional logic solution

```
dataset.sort(function (a,b){  
    if(b.population > a.population){  
        return 1  
    }  
    return -1  
})
```

## mathematical solution

```
dataset.sort(function (a,b){  
    return (b.population - a.population)  
})
```

## getting our minimum and maximum from the order of our data

*(after moving the configuration variables declarations to occur after our sort is completed!)*

```
let populationMin = dataset[0]  
let populationMax = dataset[dataset.length-1]
```

- The first solution applies what you've practiced today and is perfectly good
- The "mathematical solution" shows why **.sort()** was written to look at positive vs negative numbers:
  - If b is **greater than** a, the subtraction result is positive; no swap will occur
  - If b is **less than** a, the subtraction result is **negative**; a swap will occur
- After sorting, the **minimum** and **maximum** values for data we sorted on will be at start and end of **dataset**
  - Using **d3.min()** and **d3.max()** is probably better than sorting your data a bunch of times if you're just looking for minimum and maximum values, but now you know it's an option!

# .style()

- CSS styling can be applied to SVG elements
  - CSS stylesheet styles can be applied to SVG elements as well
    - Many (but not all!) CSS properties are the same for HTML and SVG elements
- **.style()** is a method for adding style attributes to your SVG elements
  - It is the same basic format as **.attr()**, and **.attr()** can usually be used in its place
  - **.style()** can only take a **value** (or a variable containing a value) as its second parameter
    - it can't take an **accessor function** like **.attr()** can
- In this code, we have used **.style()** to point out elements that could perhaps be moved to a **class** in our **style.css** sheet:

```
let lowKeyLabel = svg.append("text")
    .text(populationMin)
    .attr("x", Number(lowKey.attr("cx")) + Number(lowKey.attr("r")) + 5)
    .attr("y", 30 + rScale(populationMax))
    .style("alignment-baseline", "middle")
    .style("font-family", "sans-serif")
    .style("font-size", "13px")
```

# .classed() to apply CSS classes

- In **style.css**, add this new style at the bottom:

```
.keyLabel {  
    alignment-baseline: middle;  
    font-family: sans-serif;  
    font-size: 13px;  
}
```

NOTE: You will get a squiggly underline beneath **alignment-baseline** because VSCode's CSS checker only knows about style properties for HTML, and this property has been [deprecated](#) for use in CSS3.

- In **lowKeyLabel**, replace the three **.style()** lines with this line:  

```
.classed("keyLabel", true)
```
- The first parameter of **classed** is the name of the CSS class to add or remove
- The second parameter says whether the class is added or removed from the **selection**
  - **true** means **add** the class from this selection's list of classes
  - **false** means **remove** the class this selection's list of classes
- Test this on one label. You can then apply it to the others later!

# Why use **.classed()** instead of **.attr("class")**

- **.attr("class")** sets the entire class attribute -- i.e., in the DOM, it's replacing any current value for **class="someValue"** with the new value(s) provided with **className**
  - You can use **.attr("class", "className")** if your SVG object definitely only needs to have one class attached to it, or if you can list all of the classnames in **className** (just as you'd do when styling HTML with CSS, for example: `class = ".red .bigger"`)
- **.classed()** adds or removes values from the class attribute instead of replacing values that exist there. For example:
  - **circles.classed("red", true)** means you'd see `class = ".red"` when checking the DOM for your circles
  - Adding **circles.classed("bigger", true)** means you'd now see `class = ".red .bigger"` when checking the DOM for your circles
  - Adding **circles.classed("red", false)** means you'd only see `class = ".bigger"` when checking the DOM for your circles
  - Finally, adding **circles.classed("red")** would not change anything. Similar to **.attr()**, sending **.classed** only one parameter **asks** about the value sent instead of **setting** it

# Relative positioning revisited

- With **Live Server** running, watch the **Population in thousands** key at the top of the visualization as you change the variable values below
  - Save each time to update the Live Server preview in Chrome!
- First, try changing **populationMax** to **300**
  - Easiest way: type **300 //** after **let populationMax =**
    - This will comment out the **d3.max()** portion of the line so you can easily restore it later
- Try changing **populationMax** to another number
  - Pick something big enough that you'll notice a **change**!
- Do the same thing with **populationMin**
  - Try a few numbers **small** and **big** and pick one that is **bigger than** your current **populationMax**
- What elements of the key change neatly? What elements do not?
  - Why do you think the programmer thought this was an OK design decision?



# Planning ahead for a larger circle

- Here is the code that draws the circle representing the **smallest population** in the key:

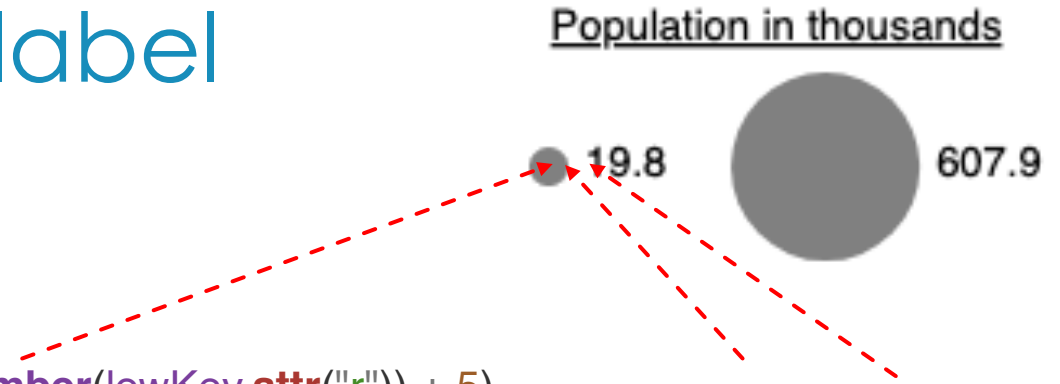
```
let lowKey = svg.append("circle")
                  .attr("fill", "grey")
                  .attr("cx", 280)
                  .attr("cy", 30 + rScale(populationMax))
                  .attr("r", rScale(populationMin))
```

- We want to have the circles in our key vertically aligned (i.e., **cy** attributes the same)
  - We know that our **populationMax** circle will take up more space, so we set both of our key circles based on the most space needed
    - It's a reasonable assumption that **populationMax** will be greater than **populationMin** (given the names of the variables). We could write more complicated code to say "position relative to whichever variable value is greater", but it's ok to **make reasonable assumptions in your code**
- We still do a bit of guess-work in setting "cx" and the offset of 30 for "cy", but we're already writing more flexible code here. *This is **very similar** to what you did on Project 1 in this class, but now you're **planning ahead** for accommodating things you'll draw **later**!*

# Lowest population key label

- Here is the first part of the key label:

```
let lowKeyLabel = svg.append("text")
    .text(populationMin)
    .attr("x", Number(lowKey.attr("cx")) + Number(lowKey.attr("r")) + 5)
```



- To figure out where to horizontally position our text, we ask our first key circle about its position and size, then add a few pixels of additional space
  - This ensures that the text will always be the same distance (5 pixels) from the circle's edge
- Reminders:
  - **.attr(attributeName)** **gets** an attribute's current value instead of setting it  
*If you provide a value as a second parameter to **.attr()**, then it **sets** the named attribute*
  - Document Object **attributes** are always sent as Strings, so if we didn't explicitly convert our two **.attr()** results here to Numbers, the values would be concatenated, giving us a value of approximately **2806.1** (280\_6.0614379227798345\_5 strung together)

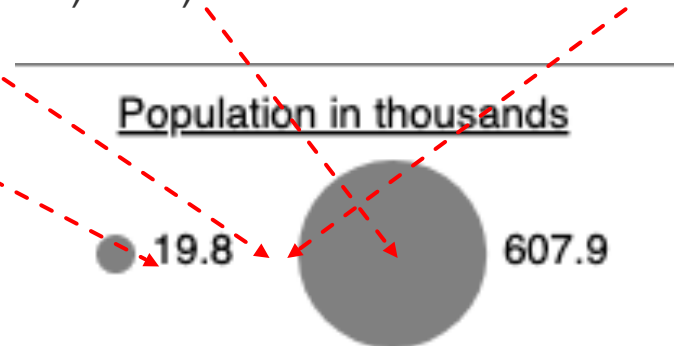
# Highest population key circle

This code positions the **second key circle** relative to what we've already drawn in our key:

```
let highKey = svg.append("circle")  
  .attr("fill", "grey")  
  .attr("cx", Number(lowKeyLabel.attr("x")) + 25 + rScale(populationMax) + 20)  
  .attr("r", rScale(populationMax))
```

► The "cx" line:

- starts us at the left edge of **lowKeyLabel**
- adds **25** as a guess for how long the text is
- adds an additional **20** to create space between the text and the big circle
- adds the **same value as the radius of the circle we're drawing** because circles are drawn from their centers out
- *NOTE: We could have added **25** and **20** to make **45**, but since those numbers represent 2 different parts of our calculation, we decided to keep them separate. This may make it easier to identify **patterns** in our numbers later and to replace some with **variables** or **functions** that can do some of the work for us!*



# How wide is **lowKeyLabel**, actually?

- As you expand your **d3** vocabulary, you'll find more and more places where you can replace figuring out values through trial-and-error with calculated values
- SVG text elements **don't** have a "width" attribute that we can easily query
- However, **d3** provides us a way of asking for this information:

*selection*.**node().getComputedTextLength()**

- Remember, when you **append** or **join** using **d3**, you're effectively creating a new **selection**, which is what's actually being stored in any variable you assign it to. Thus:
  - **svg** holds the **selection** of the SVG that we **appended** to div "#drawing"
  - **lowKeyLabel** holds the **selection** of the text element that we added to our SVG
- **.node()** tells d3 that we want the actual item represented by the selection.
  - **getComputedTextLength()** is a method of the SVG text item, not the d3 selection, so without **.node()**, you'd be told that **getComputedTextLength()** is "not a function". Why? The d3 selection doesn't know about it!

# Using `getComputedTextLength()`

## ► Try it:

- Change the **25** in our calculation for the "cx" of **highKey** so that instead that number is calculated using `.getComputedTextLength()` on the **node()** associated with **lowKeyLabel**

## ► `getComputedTextLength` Limitations:

- Can only be used on an SVG "text" element that already has text  
*(otherwise how would it know how long the text is going to be?)*
- Can only be used on an element that has been added to the DOM  
*(It is possible to create elements for later use and not immediately put them into the DOM so that they remain inactive, but not if you want to use **getComputedTextLength**)*
- As a result, this means you can't ask a "text" element its own width while it is in the process of being created, but you can after some key attributes are set.  
*(So that thing we told you before about being able to apply **.attr()** methods in any order? This represents an exception to that rule!)*
- **Handy fact for the future:** **node()** technically gets the first non-null element from a **selection**. **nodes()** will return an array of all non-null elements in a **selection**. How might this be useful?

# Discussion: making more key circles & labels

- For **bubble plots**, we asked you to produce a key of three circles: one for your **smallest** data, one for your **largest** data, and one for a "middle value" (such as a **mean** or **median**)
  - *DESIGN NOTE: If you clutter your key with a scaled circle for every value of your data, it will become overwhelming. Generally 3 to 5 examples is fine. Sometimes even 2 is enough!*
- Based on what we just discussed about relative positioning text and circles:
  - Could we create and position our "middle value" key (both circle and label) based on the information that we have for the other two circles?
  - Could we write a **loop** to draw as many key circles and text labels as we want?
  - To do this, what **assumptions** would we need to make about our data, if any?
  - How would we go about doing these things?
- We won't actually code this today. Our goals are to get you to think about:
  - how to design this (*hint: drawing it out may help!*) and
  - how to build new things using the JavaScript and d3 that you already know!

# Why grey circles? Why squares?

- Why make the circles in our key grey instead of pink or blue?
  - We want the viewer to understand that the size of the circle is what's important in the population key.
  - If these circles were colored the same as our visualized data, people might think that the color is also a factor here!
- Why are the color keys squares instead of circles?
  - If we had multiple shapes in our visualization that meant different things, we'd want the viewer to understand that the color is what's important.
    - This one is less important right now because we're only using one shape in our visualization. It is fine if you continue to use circles for your color keys, for this week.
  - Squares are generally a good shape for this because people are used to thinking of them for "color swatches".

