Code Review

Lecture 21

HACK110 Please RSVP: **bit.ly/hack110-f17**

- The COMP110 Hackathon is **NEXT FRIDAY!!! 11/17**
  - Choose your own challenge final project for COMP110
  - Please RSVP (link above) if you are definitely or maybe coming so we can plan food!

- Starts at 7pm with bootcamp lessons on:
  - What comes next in computer science?
  - More content on HTML & CSS
  - Game Development 101

- Additional activities include
  - Tech talk: TypeScript vs. Java (used in COMP401), Mobile App Dev 101
  - Make Your Own CPU Hat, Early Morning Yoga
  - Many more fun activities to break up the hacking

- HACK110 is optional and designed for you who are continuing to COMP401

PS05 - Emoji

- Due Sunday at 11:59pm
- Practice defining classes
- Practice using library documentation:
  - http://comp110.com/introcs-graphics/

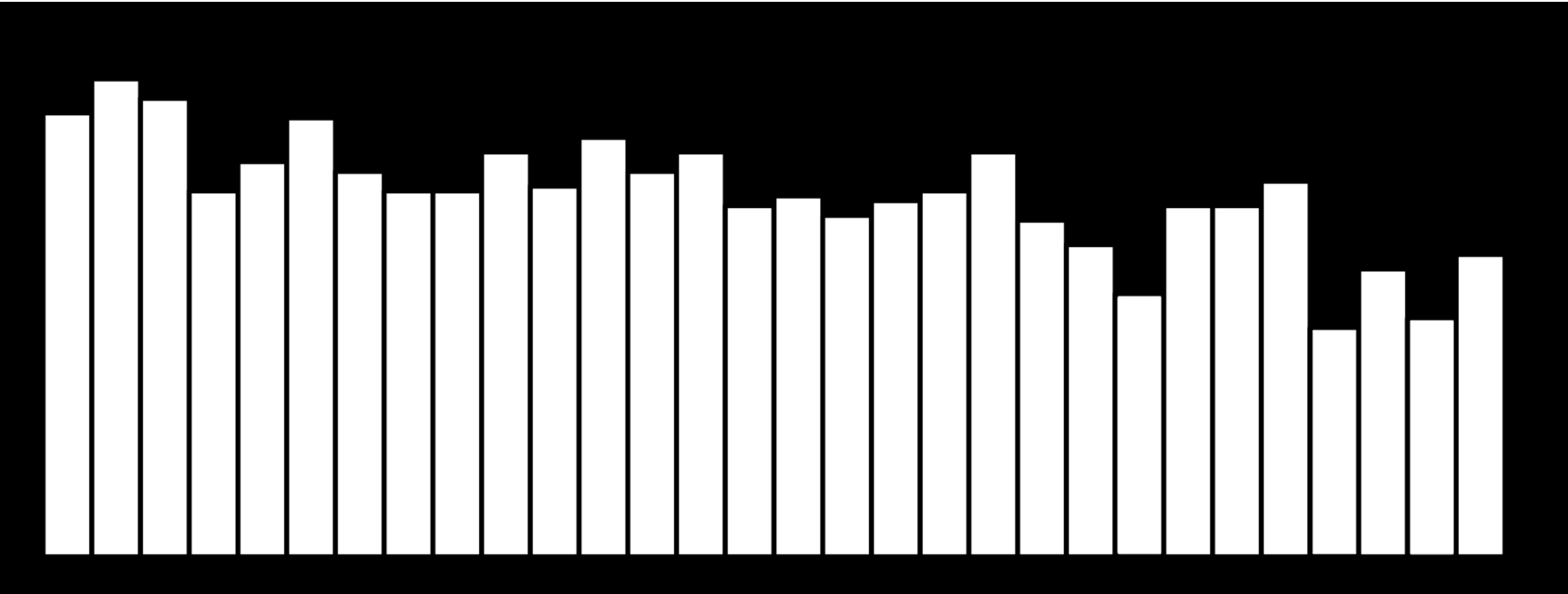
Midterm 1

- Functional Programming
  - Functional Interfaces
  - Higher-order Functions
    - Predicate<T> / Filter
    - Transform<T, U> / Map
    - Reducer<T, U> / Reduce
- Object-oriented Programming
  - Properties
  - Constructor / **new** Keyword
  - Methods
  - **`this`** Keyword
- Recursion
  - Base Case vs. Recursive Case
  - Stack Overflow
- Values vs. References
  - Value types
    - number, string, boolean
  - Reference types
    - arrays
    - objects
    - self-referencing types
      - (i.e. Linked Lists)
- Looping
  - **while** vs. **for**
- Modulus

Review Sessions

- Regular:
  - Weds 11/8 at 5pm in SN014
  - Th/Fr from 2-6pm in FB008
- In-class:
  - Today: Code-focused Review
  - Thursday: Question-based Review
- Midterm:
  - Monday at 5pm – 7pm in Hamilton 100
  - First 60 minutes lecture, Q&A after

Generating a Simple Data Visualization using Object-oriented + Functional Techniques



Hands-on:
How do we "model" a Bar in a Bar chart?

- Many possible ways! To get practice with simple OOP, we'll setup a class.

1. Above the main function in 00-visualizing-data-script.ts, declare a class named **Bar**
2. Give it a property named **amount** of type number.
3. Add a **single-parameter constructor** that initializes the amount property.
4. Add a method named **shape**
    1. Parameters: None
    2. Return type: Rectangle
    3. Logic: Return a **new Rectangle** object with width 10 and a height of the Bar's amount property. How do you reference a property of the same object in a method? (this)

- Check-in when you believe you have it, or are close!

```
class Bar {
    amount: number;

    constructor(amount: number) {
        this.amount = amount;
    }

    toRectangle(): Rectangle {
        return new Rectangle(10, this.amount);
    }
}
```

This Exercises Data Processing Pipeline

- Let's plot the high temperature of WeatherRow rows that have rain.

- We'll:

1. Filter down to an array of Weather Row objects that have rain
2. Map the array of Weather Row objects to Bars
3. Map the array of Bars to Rectangles (translated to the correct x)
4. Reduce the array of Rectangles into a single Group
5. Add the Group to our scene

Hands-on: 1. Filter Rows with Rain

- Declare a regular function that satisfies the **Predicate<WeatherRow>** functional interface (below). Name the function **hasRain**.

```
interface Predicate<T> {
  (item: T): boolean;
}
```

- The function should return true when its WeatherRow's precipitation property is greater than 0.
- In the plotData function: declare a variable named rain and assign it the result of filtering the data parameter with the hasRain predicate. What should the rain variable's type be?
- Print the length of the rain variable. Test with Fall 2016 Data. It should print 30.
- Check-in when your predicate is working!

```
function hasRain(row: WeatherRow): boolean {
        return row.precipitation > 0;
}
```

```
function plotData(data: WeatherRow[]): void {
    let scene: Group = initScene();
    let rain: WeatherRow[] = data.filter(hasRain);
    print(rain.length);
```

Hands-on: 2. Map WeatherRows to Bars

1. Declare a function named **toTempHighBar** that satisfies the functional interface Transform<WeatherRow, Bar> (below)

```
interface Transform<T, U> {
    (item: T): U;
}
```

2. The function should return a new Bar object whose amount is initialized to be the WeatherRow's tempHigh property's value.

3. In the plotData function, setup a variable named **bars**. Assign it the result of calling the **map** method on your **rain** variable and using the **toTempHighBar** as your transform function.

   What must bars' type be?

4. Check-in on PollEv.com

```
function toTempHighBar(row: WeatherRow): Bar {
    return new Bar(row.tempHigh);
}
```

Follow along: 3. Map Bars to Rectangles (1 / 2)

1. It turns out the transform functional interface specifies an optional 2nd parameter of type number. We've ignored it for simplicity.

   It will give you the *index* of the item in an array. (You won't need to know this for the midterm.)

```
interface Transform<T, U> {
    (item: T, index?: number): U;
}
```

2. If we write a Transform with a second parameter, we'll know which element # in the array we're working with.

3. We can use this to shift each **Rectangle** to the right! We will do the shifting using a "transform" which you'll need to use in Emoji.

Follow along: 3. Map Bars to Rectangles (2 / 2)

```
function toRectangle(bar: Bar, index: number): Rectangle {
    let rect: Rectangle = bar.shape();
    rect.transform = rect.transform.translate(10 * index, 0);
    return rect;
}
```

1. All SVGElements (including Groups! important for Emoji) have a property named *transform*

2. The transform property allows you to translate (think "move"), scale, and rotate any SVGElement (and all of its children in the case of Group).

3. The above example is asking a Bar for its shape and then translating the bar by some amount along the X axis.
    1. Could we have done this just using Bar's x property? Yes! We're doing it this way here to give you an example of using the transform property you can reference in Emoji.

```
let rects: Rectangle[] = bars.map(toRectangle);
```

Hands-on 4: Reducing into a Group

1. Now that we have an array of Rectangles, lets reduce them into a single Group. Declare a Reducer<Rectangle, Group> function named **intoGroup.** Remember: order of T/U matters!

```
interface Reducer<T, U> {
  (memo: U, item: T): U;
}
```

2. The function should **add** the **item** Rectangle to the **memo Group**. It should then return the **memo** Group.

3. In **plotData**, declare a variable named **chart**. Initialize it with the return value of calling the **reduce** method on **rects**. The first argument should be the **intoGroup** reducer function, the second argument is the starting memo value… a **new Group()**.

4. Add **chart** to **scene** and try it again! Check-in when you have an (inverted) Chart showing!

```
function intoGroup(memo: Group, rect: Rectangle): Group {
    memo.add(rect);
    return memo;
}
```

```
let rain: WeatherRow[] = data.filter(hasRain);
let bars: Bar[] = rain.map(toTempHighBar);
let rects: Rectangle[] = bars.map(toRectangle);
let chart: Group = rects.reduce(intoGroup, new Group());
scene.add(chart);
```

Our bars are upside down! Let's fix…

- Remember, the y-axis in digital graphics is inverted.
- When we generated a Rectangle in Bar, it's top-left corner is at (0, 0)
- Let's update Bar so that the top-left corner is at (0, -height)

```
shape(): Rectangle {
    return new Rectangle(10, this.amount, 0, -this.amount);
}
```

Method Call Chaining

- When you call a method and its return type is an array or object, you can continue calling methods on the returned value by chaining method calls together.

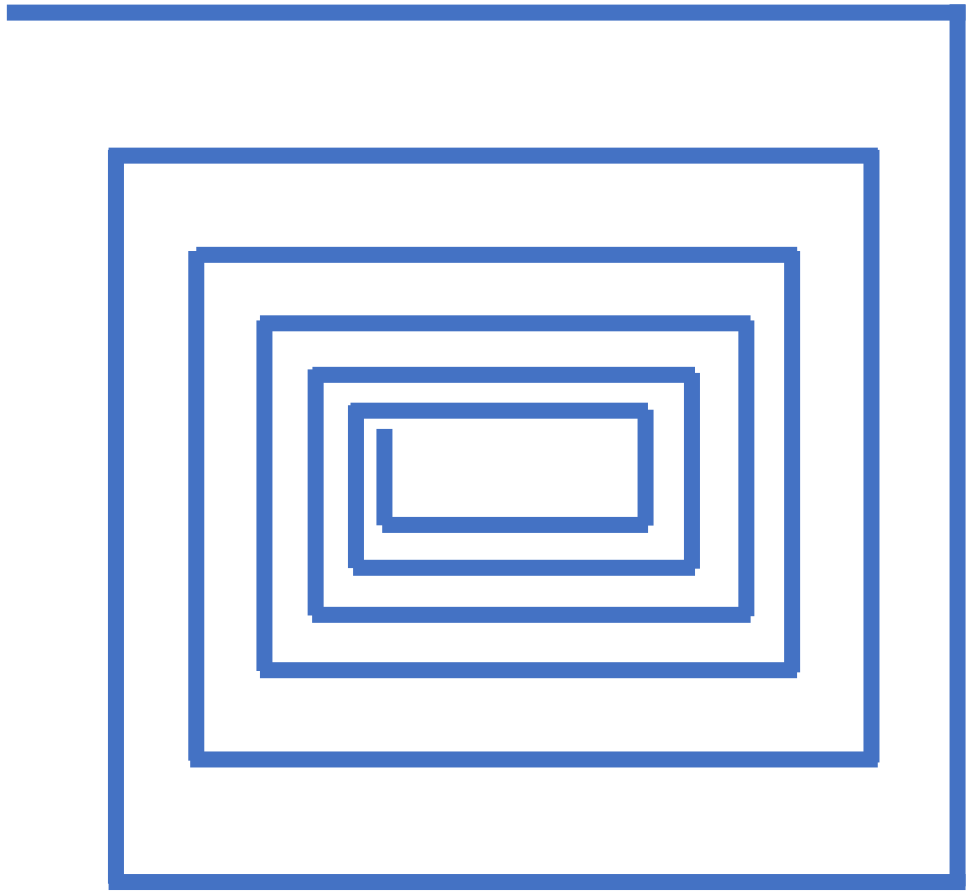
- For example, we wrote this:

```
let rain: WeatherRow[] = data.filter(hasRain);
let bars: Bar[] = rain.map(toTempHighBar);
let rects: Rectangle[] = bars.map(toRectangle);
let chart: Group = rects.reduce(intoGroup,
                                new Group());
scene.add(chart);
```

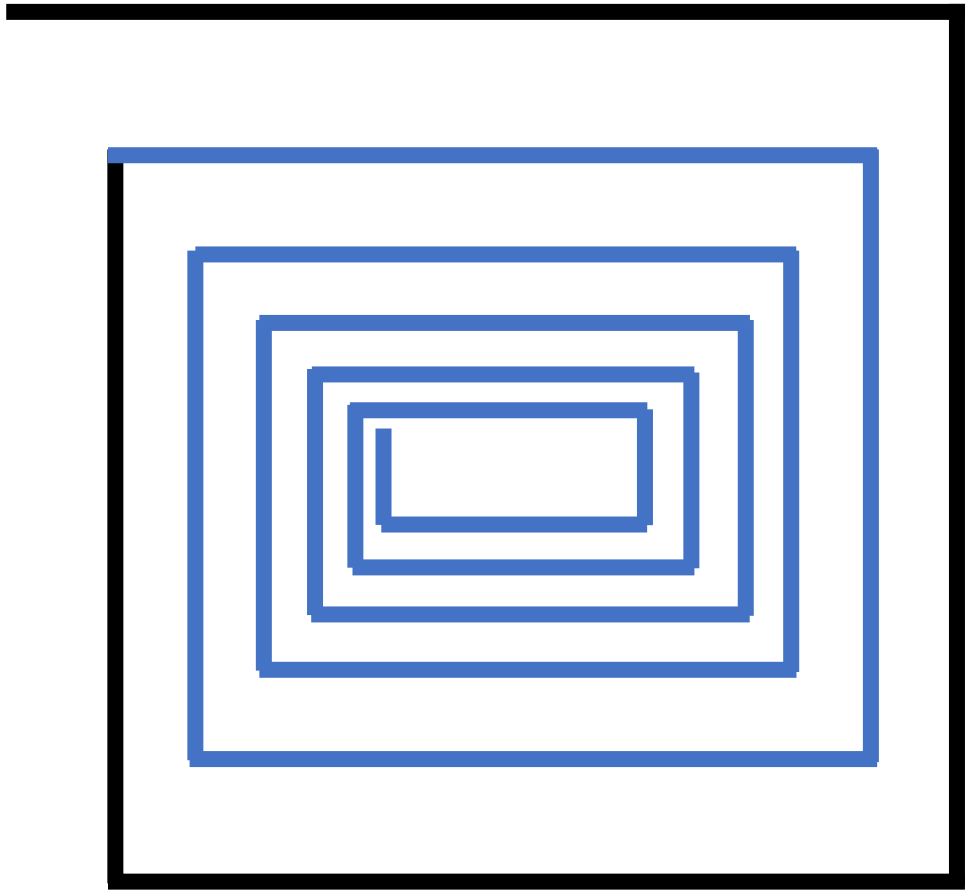
- But we could have written this:

```
scene.add(
    data
    .filter(hasRain)
    .map(toTempHighBar)
    .map(toRectangle)
    .reduce(intoGroup, new Group())
);
```

Changing Gears: Drawing a Spiral Recursively



We'll take a Path that slightly constricts on itself...

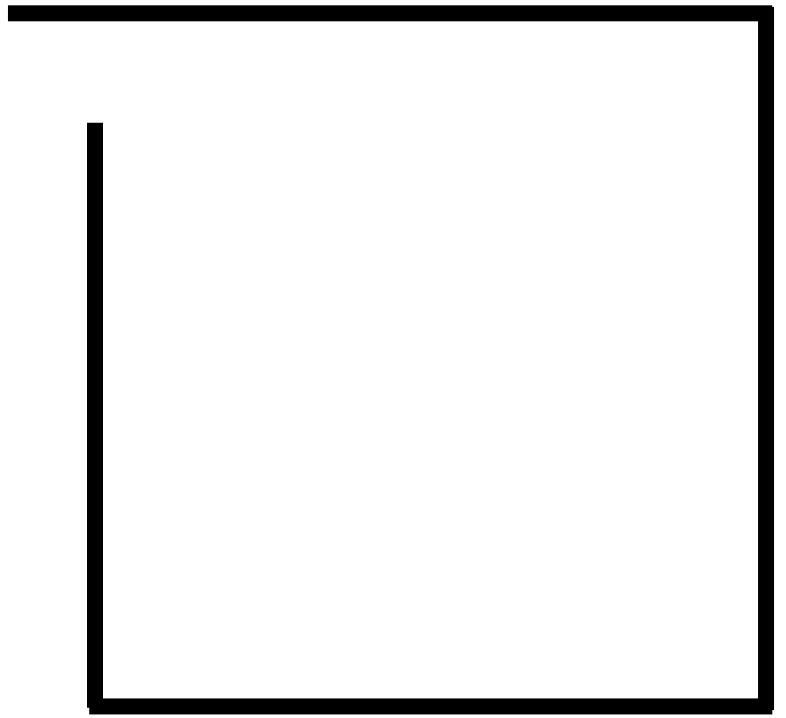


...and continue drawing it recursively!

Setting up our recursive **spiral** function in 01-recursive-drawing-script.ts

```
function spiral(startX: number, startY: number, width: number): Group
```

- Each step of the spiral will be provided a starting X, starting Y, and width value.
- Each call to this function will be responsible for drawing one step as a Path (shown to the right).
- When the width gets too small, we will return just a single step (BASE CASE)
- When the width is greater than the base case, we will recur with a startX, startY representing the ending point and a width that is SCALE % smaller.



```
const NARROW: number = 0.1;
const SCALE: number = 1 - (2 * NARROW);

function spiral(startX: number, startY: number, width: number): Group {
    let group: Group = new Group();

    // TODO

    return group;
}
```

Follow-along: Reconfiguring our Grid and calling our spiral function.

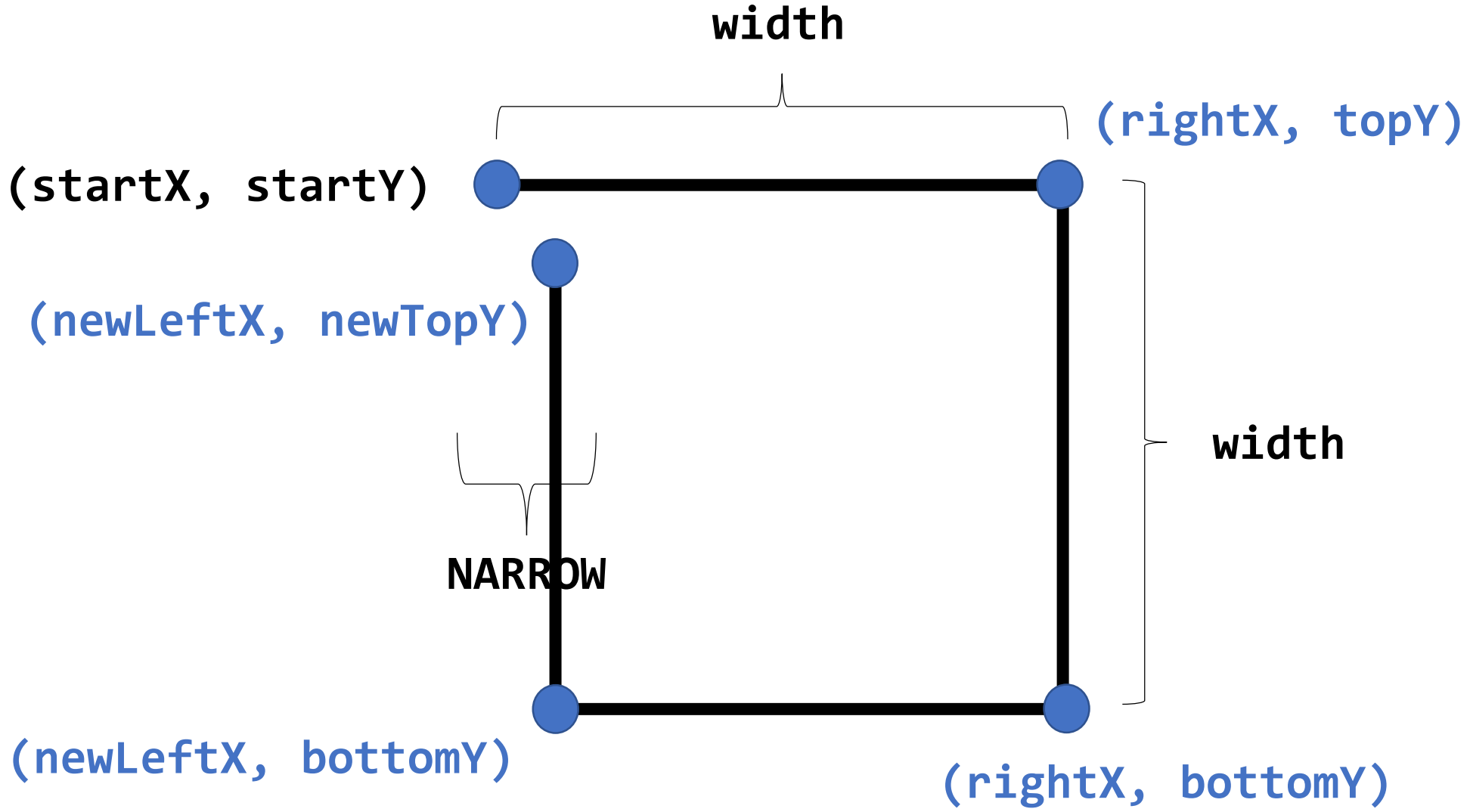
- In **initScene**, change **autoScale** to **true**

- In main, we'll change our Grid's setup to be default. We'll also add the Group returned to us by our spiral function.

```
let grid: Grid = new Grid(0, 0, 100, 100);
scene.add(grid.shapes());
scene.add(spiral(10, 10, 80));
```

Step 1: Draw our single step

We *have* the parameters startX, startY, width, and the constant NARROW (a percentage). We'll need to calculate the variables in blue from the variables in black.

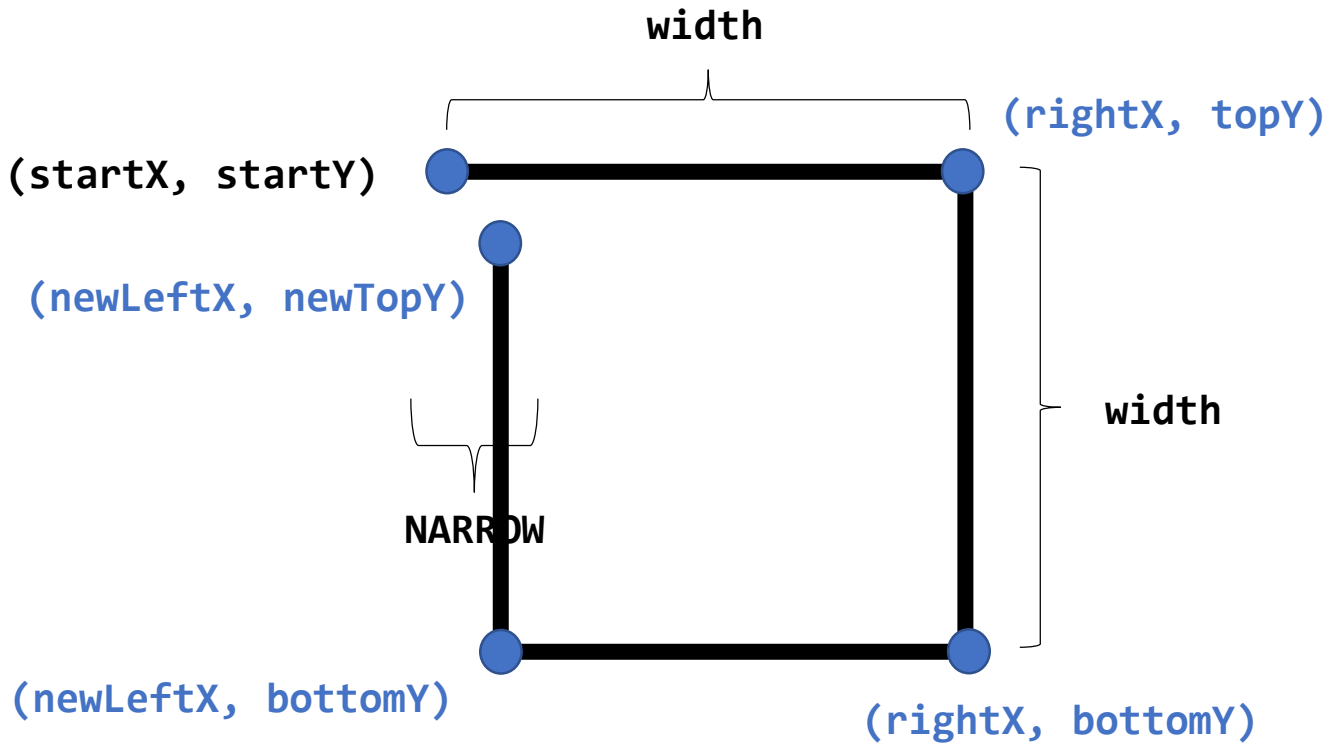


Hands-on: Declare and Initialize Variables

- Declare & initialize variables:
  - rightX
  - topY
  - bottomY
  - newLeftX
  - newTopY

- For newLeftX and newLeftY, you can assume the "margin" we're moving in is **width * NARROW**

- Check-in on PollEv.com



```
let rightX: number = startX + width;
let topY: number = startY;
let bottomY: number = startY + width;
let newLeftX: number = startX + (width * NARROW);
let newTopY: number = startY + (width * NARROW);
```

Hands-on: Read the Docs & Make a Path

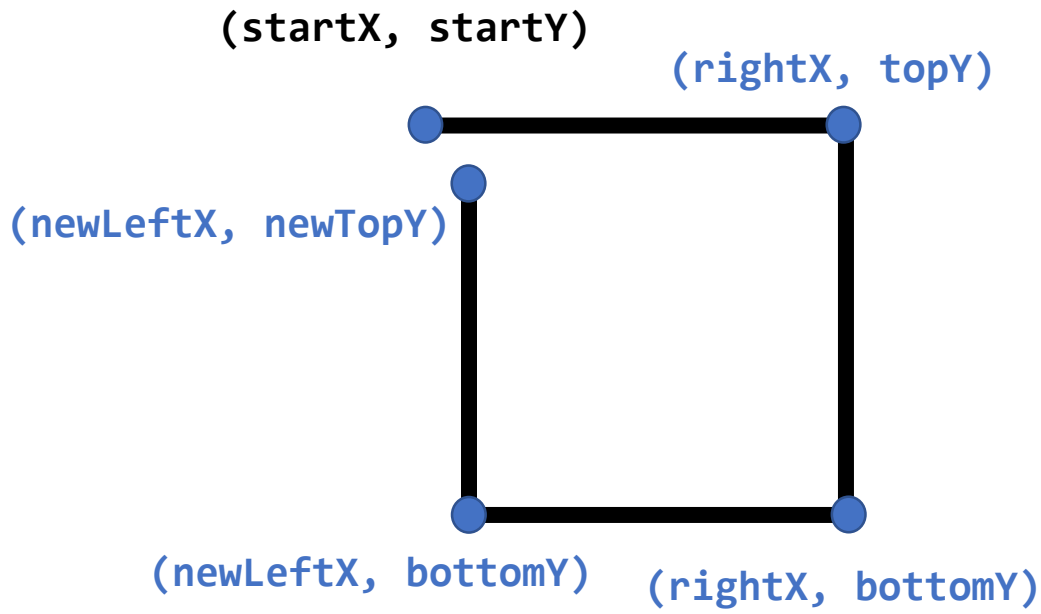
1. Open: **comp110.com/introcs-graphics**

2. Click on the **Path** class. Find its constructor's documentation.

3. Declare a variable named **path** of type **Path** and initialize it using **startX** and **startY**.

4. Find the documentation for the **lineTo** method. Call the lineTo method 4 times on **path** to connect the four points of the step (right).

5. Add path to group and check-in on PollEv



You could have written this...

```
let path: Path = new Path(startX, startY);
path.lineTo(rightX, topY);
path.lineTo(rightX, bottomY);
path.lineTo(newLeftX, bottomY);
path.lineTo(newLeftX, newTopY);
group.add(path);
```

OR something that looks like this....

```
group.add(
    new Path(startX, startY)
    .lineTo(rightX, topY)
    .lineTo(rightX, bottomY)
    .lineTo(newLeftX, bottomY)
    .lineTo(newLeftX, newTopY)
);
```

The right example is called method call chaining. Because lineTo returns a reference to the same Path you called lineTo on, you can continue calling lineTo (or other path segment methods) on it in a chained fashion.

Hands-on: Recursion

- Now make the **spiral** function recursive.

- BASE CASE: width is less than 3
  - Return group immediately.

- RECURSIVE CASE:
  - Call the **spiral** function again and **add the Group it returns to group**.
  - The width argument will be **width * SCALE**
  - What should the startX and startY values be? (Hint: the ending point of the path you just drew!)

- Check-in when you have a recursive spiral being drawn. Done? Try tinkering with the NARROW and SCALE constants.

```
if (width < 3) {
    return group;
} else {
    group.add(spiral(newLeftX, newTopY, width * SCALE));
    return group;
}
```

- Isn't having these two return statements redundant?

- Why not simply test if width >= 3 and make that the recursive case and not specify the base case?
  - This would be a perfectly reasonable code design decision to make once you're comfortable with recursion.
  - In COMP110, while we're still getting comfortable with recursion, you'll see us tend to always explicitly handle the base case first and the recursive case separately.