Working with Classes

Lecture 13

1. Given the class Point, what is the output when the code to the right runs?

```
class Point {
    x: number = 0;
    y: number = 0;

    quadrant(): number {
        if (this.x >= 0) {
            if (this.y >= 0) {
                return 1;
            } else {
                return 4;
            }
        } else {
            if (this.y >= 0) {
                return 2;
            } else {
                return 3;
            }
        }
    }
}
```

```
let a = new Point();
a.x = 10;
a.y = -10;

let b = a;
b.x = -10;
b.y = 10;

print(a.quadrant());
```

```
3  class Dog {
4      age: number;
5
6      constructor(age: number) {
7          print("7");
8          this.age = age;
9      }
10
11     speak(): void {
12         print("woof " + this.age);
13     }
14 }
15
16 export let main = async () => {
17     print("17");
18     let fido = new Dog(1);
19     fido.speak();
20     print("20");
21 };
22
23 main();
```

2. Draw an environment diagram of the code left. Check-in with the printed output.

TypeScript and HTML/CSS

- So far, we've made "apps" by creating TypeScript files named "-app.ts"
  - In these apps we have been able to print values and prompt for inputs
- Today we'll see a single-page web app
  - It will be made up of HTML, TypeScript, and CSS
- HTML - (Hyper-Text Markup Language)
  - The primary "document" being loaded into the web browser
  - Our TypeScript code is included by the HTML
- TypeScript
  - The "code" you'll write!
  - Can access HTML and update / change it to present graphics, data, and forms to the user.
- CSS - (Cascading Style Sheets) - The "style" or "theme" of page.

Today's HTML (1/4)

- We will not spend much time on, nor test on, HTML & CSS in COMP110

- However, to give you a rough sense of how these three pieces fit together, it's worth taking a quick peep...

```
<html>
    <head>
        <title>Intro to CS - Graphics</title>
        <link rel="stylesheet"
              type="text/css"
              href="./style.css">
    </head>
    <body>
        <svg id="artboard"></svg>
        <div id="console"></div>
        <script src="./app-script.ts"></script>
    </body>
</html>
```

Today's HTML (2/4)

- HTML documents are made up of "tags" describing Elements
- Typically these tags are paired:
  - Open tag: <html>
  - Close tag: </html>
- Tags can be nested inside of one another and must be properly nested (like curly brace blocks in TypeScript)
- The **head** tag is short for "header" and contains meta information about a web page
- The **body** tag is what gets displayed in the web browser.

```
<html>
    <head>
        <title>Intro to CS - Graphics</title>
        <link rel="stylesheet"
              type="text/css"
              href="./style.css">
    </head>
    <body>
        <svg id="artboard"></svg>
        <div id="console"></div>
        <script src="./app-script.ts"></script>
    </body>
</html>
```

Today's HTML (3/4)

- The "style" of our web page is established by linking to the "./style.css" file in our project folder using this `<link>` tag.

- The TypeScript "script" or "code" of our web page is established by linking to the "./app-script.ts" file in our project folder using the `<script>` tag.
  - The "-script.ts" suffix is important here!
  - Note: Typically, you'd have to compile a TypeScript file to a JavaScript file on your own before being able to include it in this way. Our programming environment is doing this automatically for you behind the scenes.

```
<html>
    <head>
        <title>Intro to CS - Graphics</title>
        <link rel="stylesheet"
              type="text/css"
              href="./style.css">
    </head>
    <body>
        <svg id="artboard"></svg>
        <div id="console"></div>
        <script src="./app-script.ts"></script>
    </body>
</html>
```

Today's HTML (4/4)

- The **<svg>** tag stands for "scalable vector graphics" and is where our shapes will be displayed when we render them.
  - Notice: it's **id** is the string "artboard" - we will use this **id** in our code to be able to interact with this element.

- We also have a ** ** or "division" tag which is where our printed output will be displayed if we use the **`print()`** function.

```
<html>
    <head>
        <title>Intro to CS - Graphics</title>
        <link rel="stylesheet"
              type="text/css"
              href="./style.css">
    </head>
    <body>
        <svg id="artboard"></svg>

        <div id="console"></div>
        <script src="./app-script.ts"></script>
    </body>
</html>
```

Graphics

- We'll use the **introcs** library to work with SVG graphics
  - We'll import SVG, Shape, Color, etc. classes from "introcs/graphics"

- In **`app-script.ts`**, let's draw our first Shape!

- First, we must establish a connection between an SVG Object in our TypeScript code and the SVG tag in our HTML...
  - We'll setup a global variable to do this outside of the main function...
  - The "artboard" string is referring to the `id` of the SVG tag in the HTML.

```
let artboard = new SVG("artboard");
```

Hands-on #1) Draw a Circle

```
Circle's Constructor's parameters:
(radius: number, cx: number, cy: number)
```

- Inside the main function of *app-script.ts:*

1. Declare a variable named **c** of type Circle.

2. Initialize **c** by constructing a new Circle using the constructor above. Try giving it a `radius` of 50, a cx of 50, and a cy of 50.

3. Add it to the shapes Group variable: **`shapes.add(c);`**

4. Save and refresh your web browser. (Unfortunately, with these HTML-based applications we have to save and then refresh the page ourselves.)

5. Check-in on PollEv.com/comp110

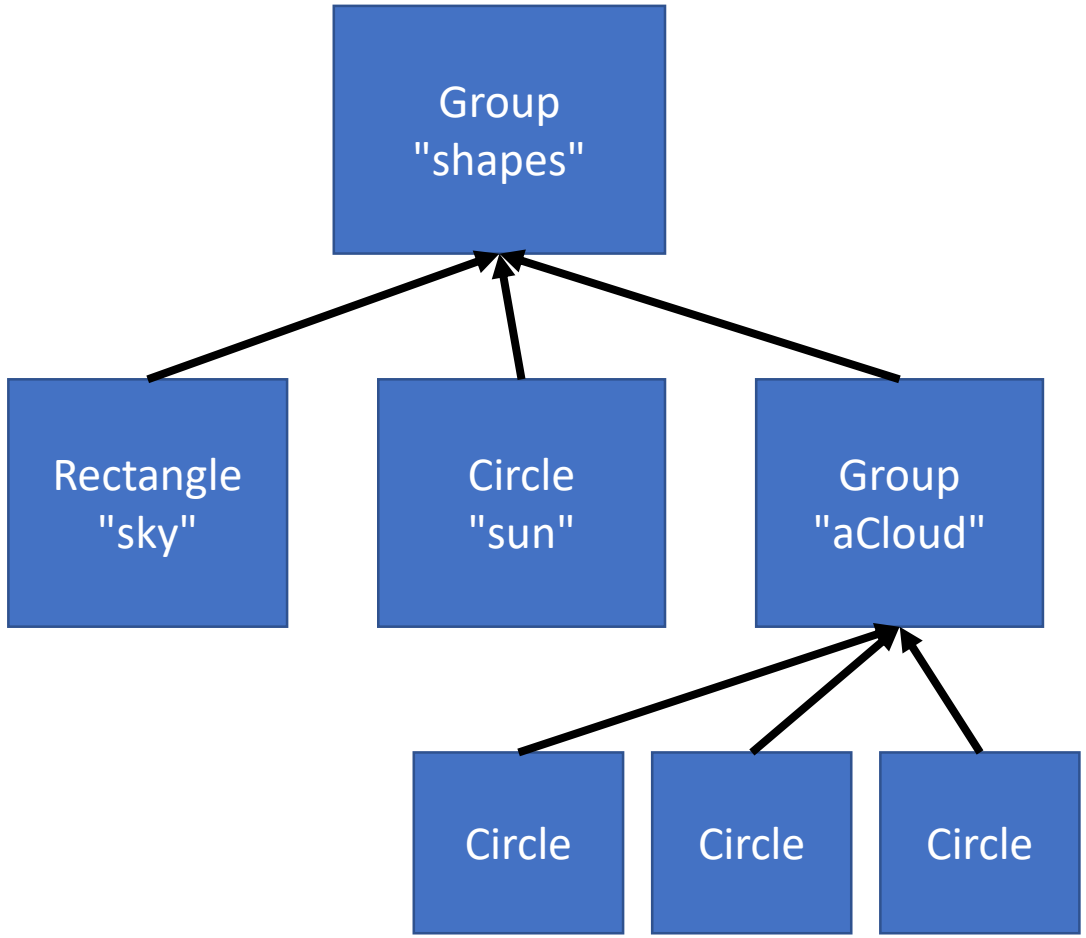
```
export let main = async () => {

    let shapes = new Group();
    let c = new Circle(50, 50, 50);
    shapes.add(c);
    artboard.render(shapes);

}
```

Nested Groups of Shapes

- A Group is a special kind of "Shape" object that can hold references to other Shapes
- You can add one Group of shapes to another Group
- This makes it possible to "compose" more complex scenes by breaking each drawing task into smaller functions



Scaling & Dimensions

- In computer graphics, the y-axis tends to be inverted. So larger y numbers move graphics *down* on the screen.

- The `introcs` graphics library can automatically scaling and centering our graphics to fit our browser's size. To enable: `artboard.autoScale = true;`

- To get a better sense of how positioning works, let's add some axes to our drawing.

- We've setup the basis of an Axes class that can be used to demonstrate how we can use a class and to get more experience with constructors and methods.

Hands-on: Adding a constructor to the Axes class

- In Axes.ts, **define a constructor with 2 parameters**:
  1. width which is a number
  2. height which is a number

- Assign ***both*** parameter values **width** and **height** to the Axes object's respective properties... for example: **`this.height = height;`**

- In **app-script.ts**, in main before you setup the Rectangle, declare a variable named **axes** and **assign it a new Axes object constructed with arguments 100, 100**.

- Then, call its **getShapes** method and add the resulting Group to the shapes Group in main: **shapes.add(axes.getShapes());**

- Check-in once you've made these changes (save every file!) and refreshed to see you have a Y-Axis showing up!

```
// TODO: Declare a constructor with width and
height parameters
constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
}
```

```
let axes = new Axes(100, 100);
shapes.add(axes.getShapes());
```

Hands-on: Adding an xAxis

- Under the TODO in Axes.ts' getShapes method...
- Declare a variable named xAxis and assign it a new Line object. The Line class' constructor has 4 parameters:
  1. x0 - the x component of the starting point
  2. y0 - the y component of the starting poing
  3. x1 - the x component of the ending point
  4. y1 - the y component of the ending point
- Choose arguments such that the xAxis is centered about 0, 0 and has a total width of this.width (hint: see how the yAxis was constructed)
- Add the xAxis line to the Group of shapes named g
- Check-in when you have both Axes showing!

```
let yAxis = new Line(0, -this.height / 2, 0, this.height / 2);
let xAxis = new Line(-this.width / 2, 0, this.width / 2, 0);
let g = new Group();
g.add(yAxis);
g.add(xAxis);
return g;
```

Reference Documentation for IntroCS Graphics

- The graphics/shape library reference documentation here:
  - http://comp110.com/introcs-graphics/

- Getting practice reading through documentation like this is valuable
  - When building projects in later classes and industry, this is how you'll learn the capabilities of a library

- You'll notice each class and object has many more properties, methods, and constructor variations than we're covering in lecture!

Hands-on) Draw 10 Circles

```
Circle's Constructor's parameters:
(radius: number, cx: number, cy: number)
```

The cx and cy parameters correspond to the center point of the circle.

In the main function, after you've added the axes to the group...

1. Declare a variable named **`count`** and initialize it to **`10`**
2. Write a for-loop using **`i`** as its counter that will loop **`count`** times
3. Inside of the loop, declare a variable named **`c`**, of type **Circle**
   - Initialize it to a new Circle object using the constructor above
   - radius: 5
   - cy: 0
   - cx: Try multiplying **`i`** by some amount to space the circles out
4. Add the circle to the group named **shapes**.
5. Check-in when you have at least 1 circle visible in your scene.

```
let count = 10;
for (let i = 0; i < count; i++) {
    let c = new Circle(5, i * 10 - 45, 0);
    g.add(c);
}
```

Painting by Numbers

- What we're going to do from here forward is a bit more free flowing
- We're going to explore working with Shapes and Colors using some math
- This will give us practice making use of constructors, loops, and "composing" object graphs
- There is nothing conceptually new to commit to memory or study from this point forward, this is just us making some art together as a class.

Let's add Color

- Digital colors are often specified as a combination of red, green, blue (RGB)

- In our graphics library, we will assign *percentage amounts* of RGB using decimal values ranging from 0.0 to 1.0 (0% to 100% respectively)
  - The color white is Red: 1.0, Green: 1.0, Blue: 1.0
  - The color black is Red: 0.0, Green: 0.0, Blue: 0.0
  - Carolina Blue is: Red: 0.2929, Green: 0.6094, Blue: 0.8242

Follow-along: Adding Color

```
let count = 10;
let i = 0;
while (i < count) {
    let c = new Circle(5, i * 10 - 45, 0);

    let red = 0.2929;
    let green = 0.6094;
    let blue = 0.8242;
    c.fill = new Color(red, green, blue);

    g.add(c);
    i++;
}
```

Generating colors programmatically (1/2)

- How could we make our circles *increasingly blue* starting from black?

- Start: Color(0.0, 0.0, 0.0)
- End: Color(0.0, 0.0, 1.0)

- We want to pick numbers between 0.0 and 1.0 evenly as we construct each circle.
  - The technical term for this is *linear interpolation*

- We have some variables that will help us: **i** and **count**

- "Percent of Iteration Completed" Formula: **`i / (count - 1)`**
  - Why count - 1? Because we are starting from 0% and want the final loop iteration to be 100%

```
    let percent = i / (count - 1);

    let c = new Circle(5, i * 10 - 45, 0);

    let red = 0;
    let green = 0;
    let blue = percent;
```

Generating colors programmatically (1/2)

- How could we make our circles *decreasingly green* ending with black?

- Start: Color(0.0, 1.0, 0.0)
- End: Color(0.0, 0.0, 0.0)

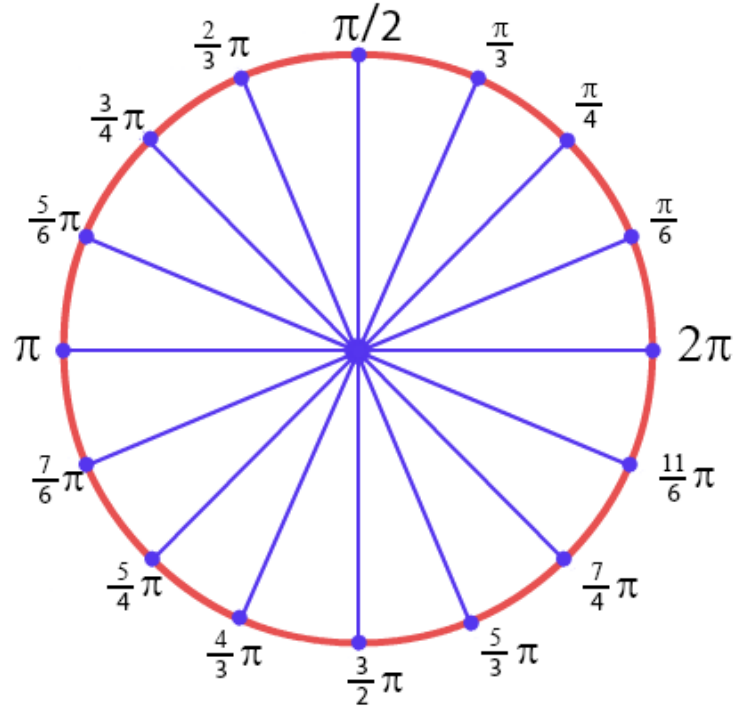
- We want to pick numbers between 1.0 and 0.0 evenly as we construct each circle.

- Formula: **`1.0 - percent`**
  - We can invert our percentage

```
let red: number = 0;
let green: number = 1 - percent;
let blue: number = percent;
c.fill = new Color(red, green, blue);
```

How would we make a wave?

- We have to go back to our old friend **sine**.
- Let's make a sine wave!
- The Math.sin function expects that we're working in *radians*
- First, let's setup a constant to represent total radians in a circle (2*pi).
- Then, we'll calculate the center-Y of each circle as a percentage of total radians.

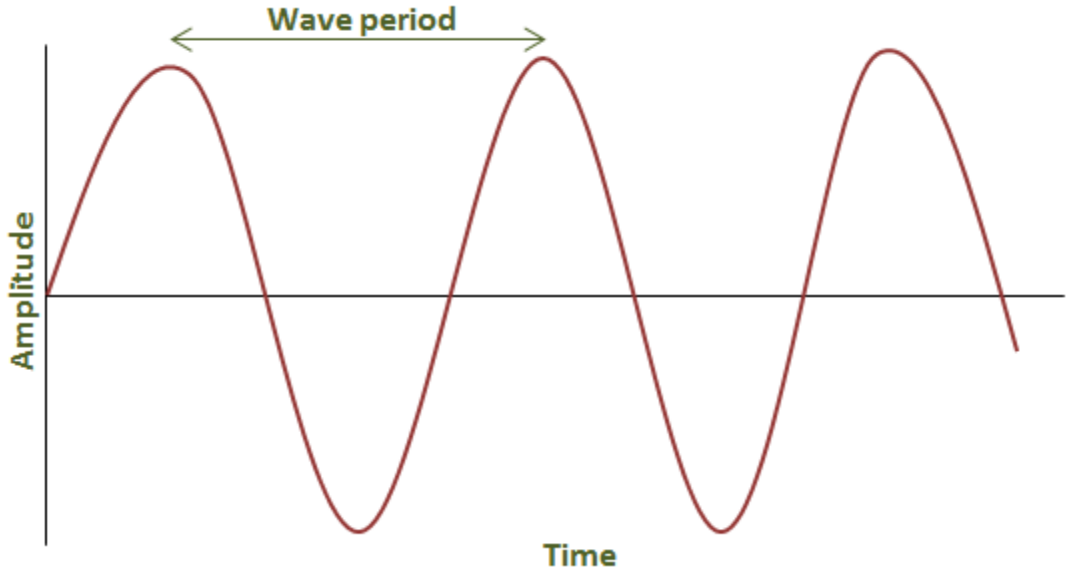


```
let artboard = new SVG("artboard");
const RADIANS = Math.PI * 2;
```

```
let angle = percent * RADIANS;
let cy = Math.sin(angle) * 20;
let c = new Circle(5, i * 10 - 45, cy);
```

How would get the sine wave to flow over a period of time?

- Currently every circle is placed statically at its sine position
- As time elapses through some **period** (of time) we want to offset its y-value by the percentage of the **period** that has completed
- We can get the current time using the built-in Date.now() method. This will give us the number of milliseconds passed since 1/1/1970.
  - How do we convert ms to s?
  - Once we have the current time in seconds, how do we get a period that will cycle through from 0-period as time passes? Hint: modulo!
  - How do we calculate the percentage of the period that has passed?



```
let time = Date.now() / MS_IN_S;
let periodPercent = time % PERIOD / PERIOD;

for (let i = 0; i < count; i++) {
    let percent = (i + 1) / count;
    let angle = percent * RADIANS;
    let angleT = periodPercent * RADIANS;
    let cy = Math.sin(angle + angleT) * 20;

    let c = new Circle(5, i * 10 - 45, cy);
```

How do we get the sine wave to animate?

- We need the main function to be called repeatedly over time…
- Luckily, there is a built-in function that can do this for us!

```
interface VoidFunction {
    (): void;
}
```

`setInterval(f: VoidFunction, milliseconds: number): number`

- main does not return anything so it satisfies the VoidFunction functional-interface
- So, we can call `setInterval` and passing a reference to the main function. Let's try an interval of 20ms.

```
setInterval(main, 30);
```