Constructors and Shapes

Lecture 18 - COMP110 - Spring 2018

After running **npm run pull** and **npm start**,
press the dark gray close button in the top-right corner of our development page
and then run **npm start** again!

Hack110

- https://www.youtube.com/watch?v=w8LY77pxceo
- Friday, April 30th at 7pm!
- Food, t-shirts, great talks by UTAs
- Please RSVP if you have not already: https://goo.gl/forms/f1KpdNWH8WtJl6Ie2

Time	Event
7:00 PM	Check In
7:15 PM	110, Now What? + Panel + Pizza
8:00 PM	Hacking Prep: Web Development 101
	Hacking Prep: Game Development 101
HACKING BEGINS	
10:00 PM	Intro to Java
10:30 PM	Intro to GitHub
	Blockchain 101
11:00 PM	How to Build a Chrome Extension
	Intro to iOS Development
12:00 AM	Escape Room!
	Super Smash Bros. Tournement
1:00 AM	Ice Cream Sundae Bar!
2:00 AM	Competitive Cup Stacking
5:30 AM	Sunrise Yoga!
6:30 AM	Breakfast and Demos! Get your "I Demoed" Stickers!

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());
```

Constructors

- An object's properties must be initialized before the object is usable
- A constructor allows us to both
  1. Specify unique initial values of properties upon construction
  2. Require certain properties are initialized
- A constructor is just a special function
  - Does *not* use the keyword "function"
  - Name is **`constructor`**
  - Special, self-referencing variable named **`this`**
  - No return type
- A class' constructor is called each time the **`new <Classname>`** expression is evaluated.

Before

```
let a: Point = new Point();
a.x = 10;
a.y = 0
```

Defining a constructor

```
class Point {

    x: number;
    y: number;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }

}
```

After

```
let a = new Point(10, 0);
```

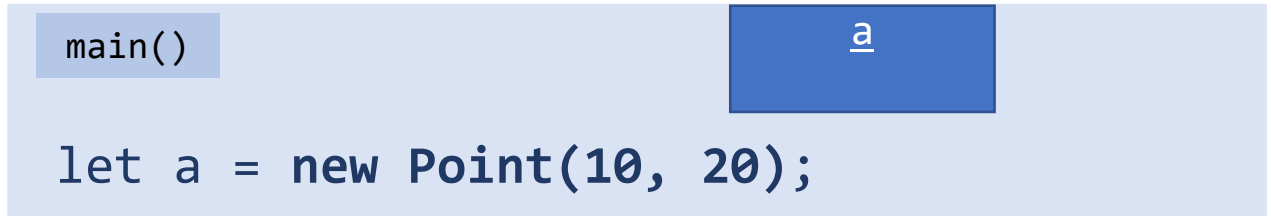
Follow-along: Constructors

- Let's open 00-constructor-app.ts
- We'll add the constructor from the previous slide.
- Then we'll need to update where we call the constructor from.

Tracing a Constructor Call (1 / 11)

When the syntax **`new <Classname>(…)`** is encountered, the processor will look to the class to see if it has a **constructor** defined. Like a function or method call, any arguments in parenthesis must match the parameters of the constructor.

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20);
    print(a);

};
```



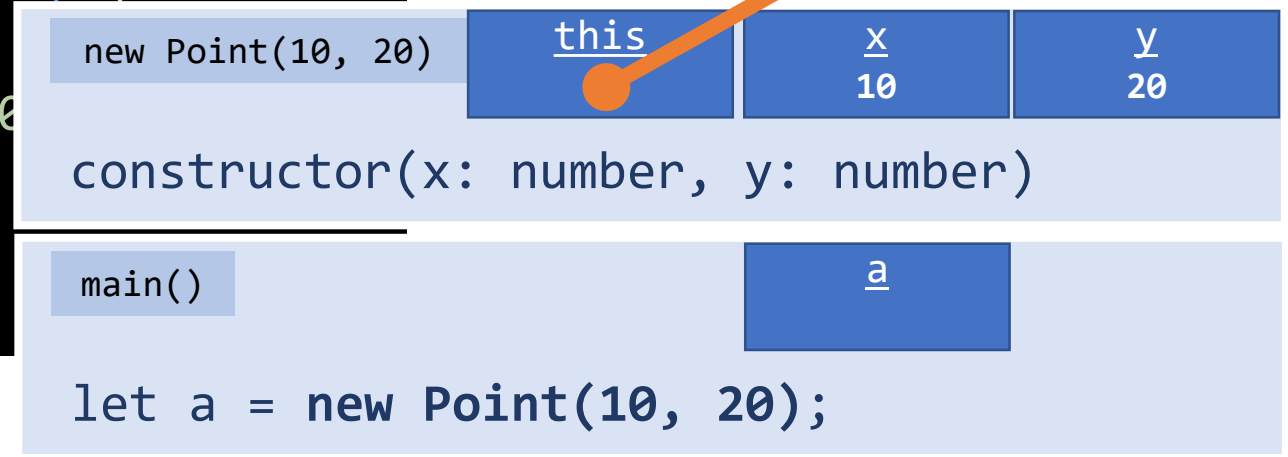
The Stack

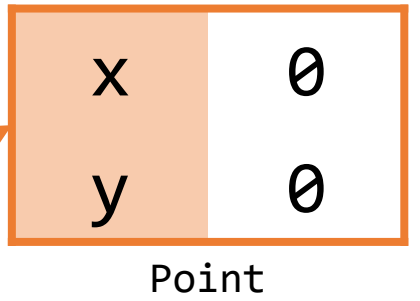
The Heap

Tracing a Constructor Call (2 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20
    print(a);

};
```

A bookmark will be dropped and arguments will be assigned to parameters.
Additionally, the **this** variable will be assigned reference the object being constructed.



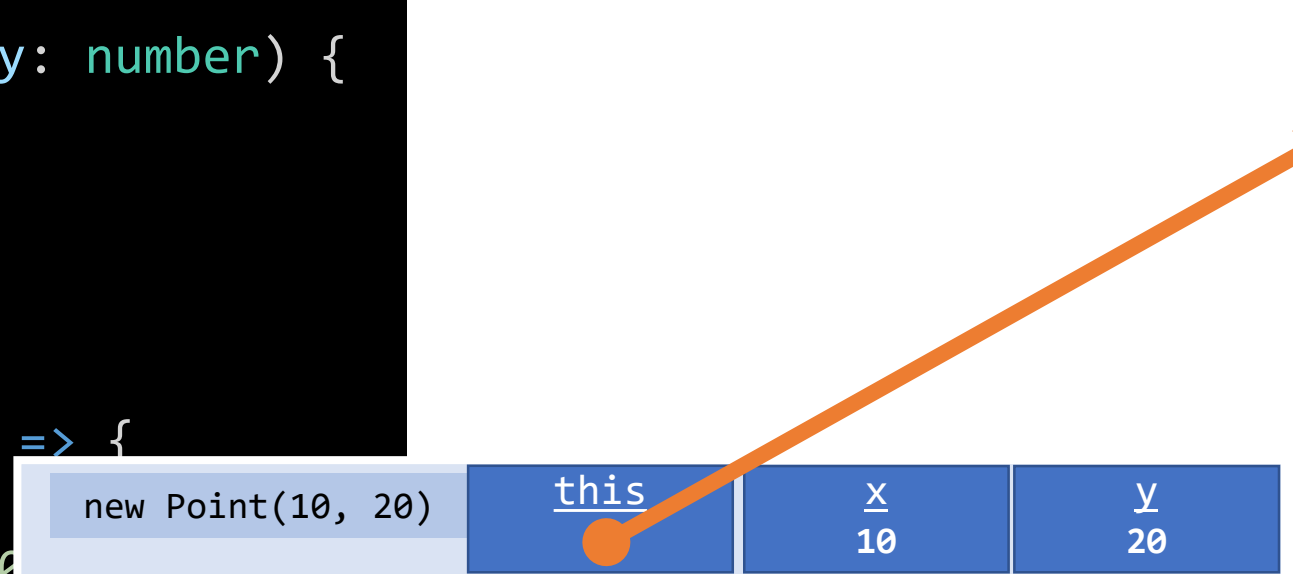


The Heap

Tracing a Constructor Call (3 / 24)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20);
    print(a);

};
```

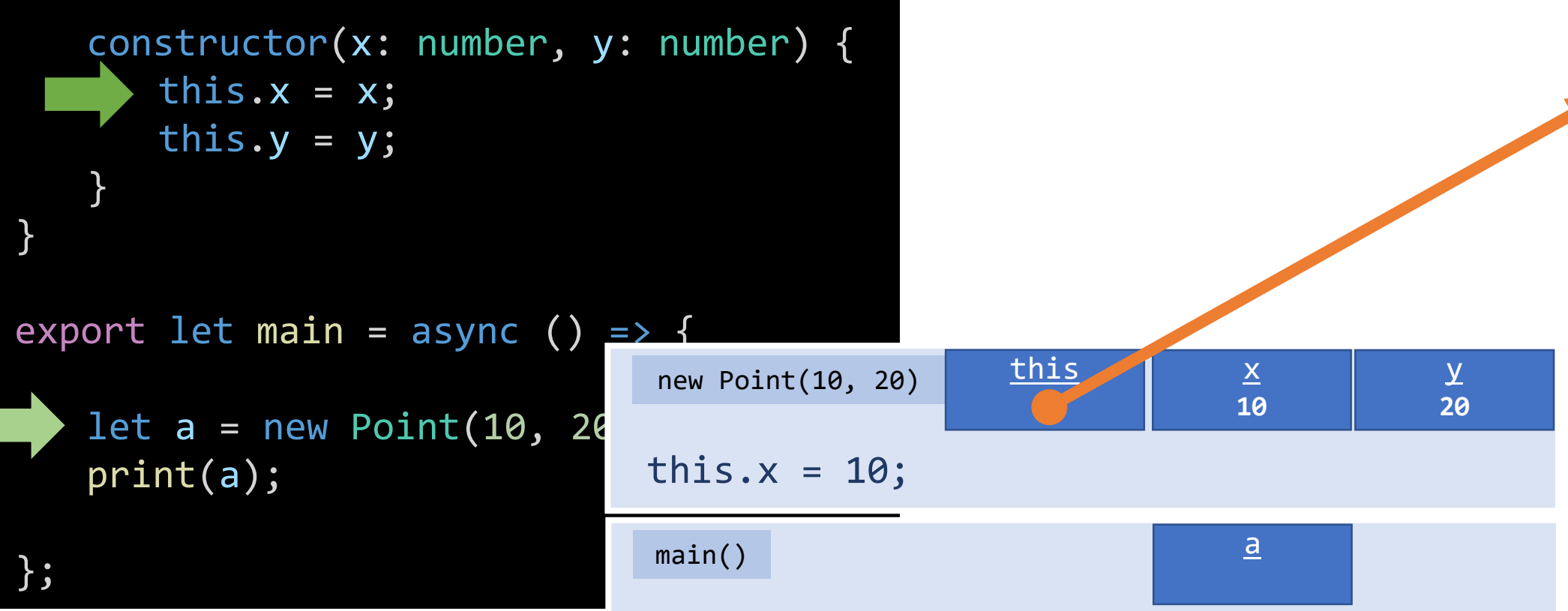
Then, the processor will enter the constructor.



Tracing a Constructor Call (4 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20);
    print(a);

};
```

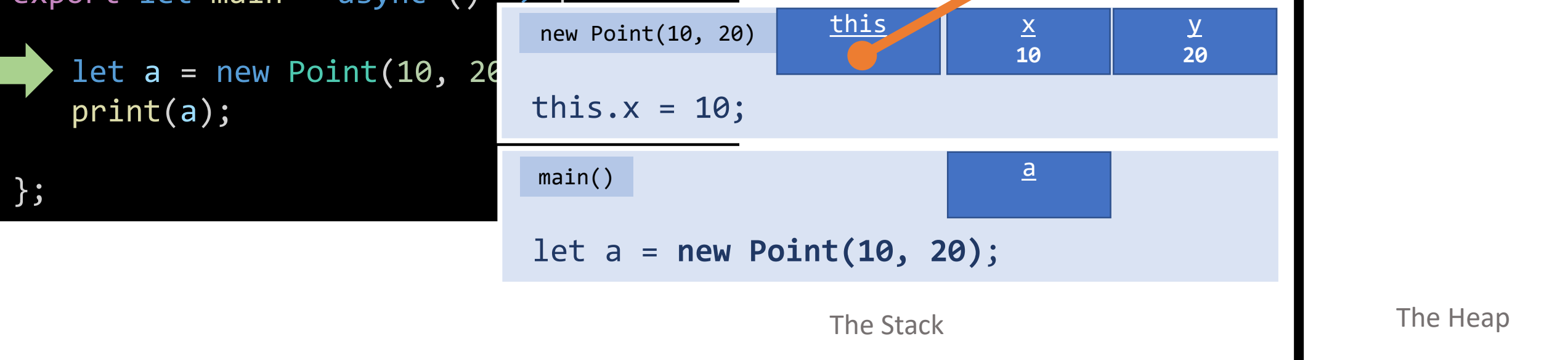
The assignment statement's right hand side will be evaluated first.



Tracing a Constructor Call (5 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20);
    print(a);

};
```

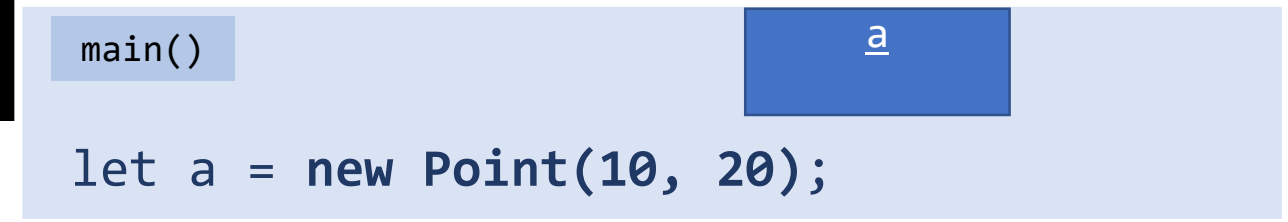
Then the value will be assigned to the x property of the object being constructed.

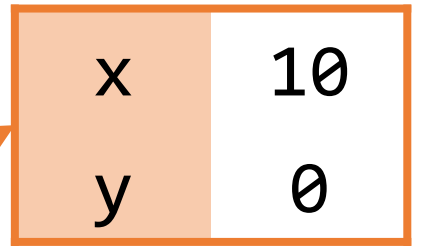


Tracing a Constructor Call (6 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20
    print(a);

};
```

The same process continues for the y property initialization.





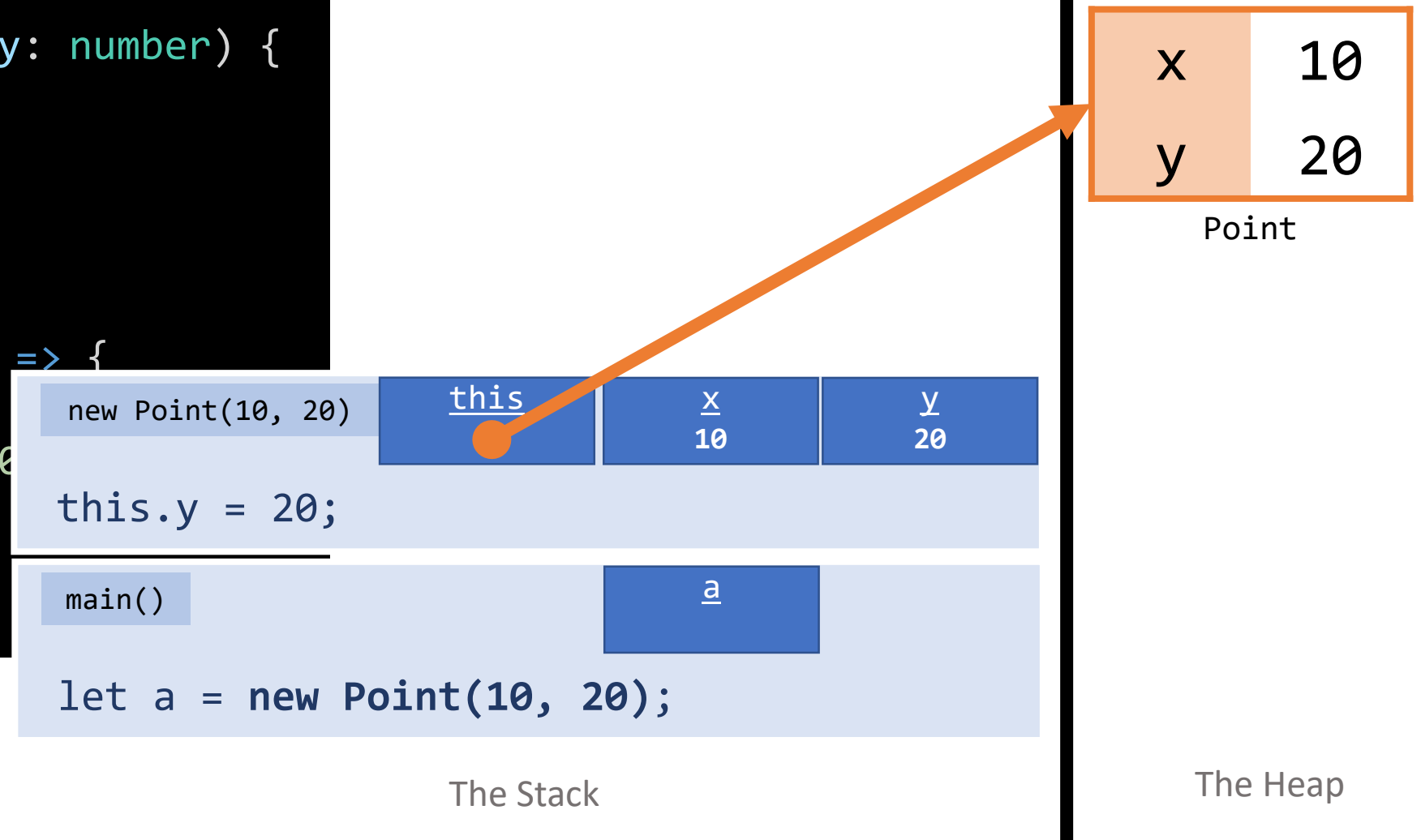
The Stack

The Heap

Tracing a Constructor Call (7 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20);
    print(a);

};
```

The same process continues for the y property initialization.

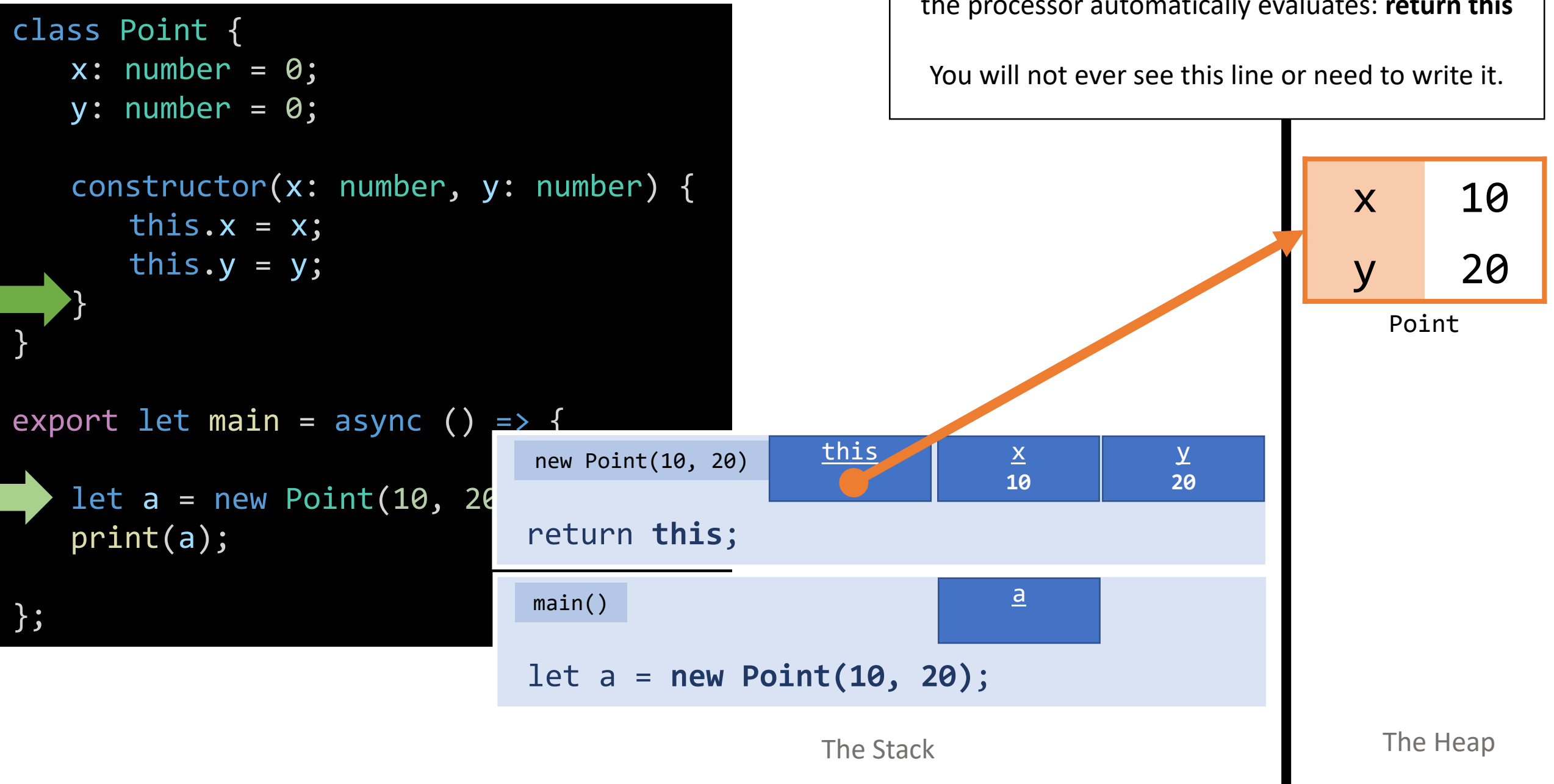


Tracing a Constructor Call (8 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20
    print(a);

};
```

When the end of the constructor is reached it is as if the processor automatically evaluates: **return this**

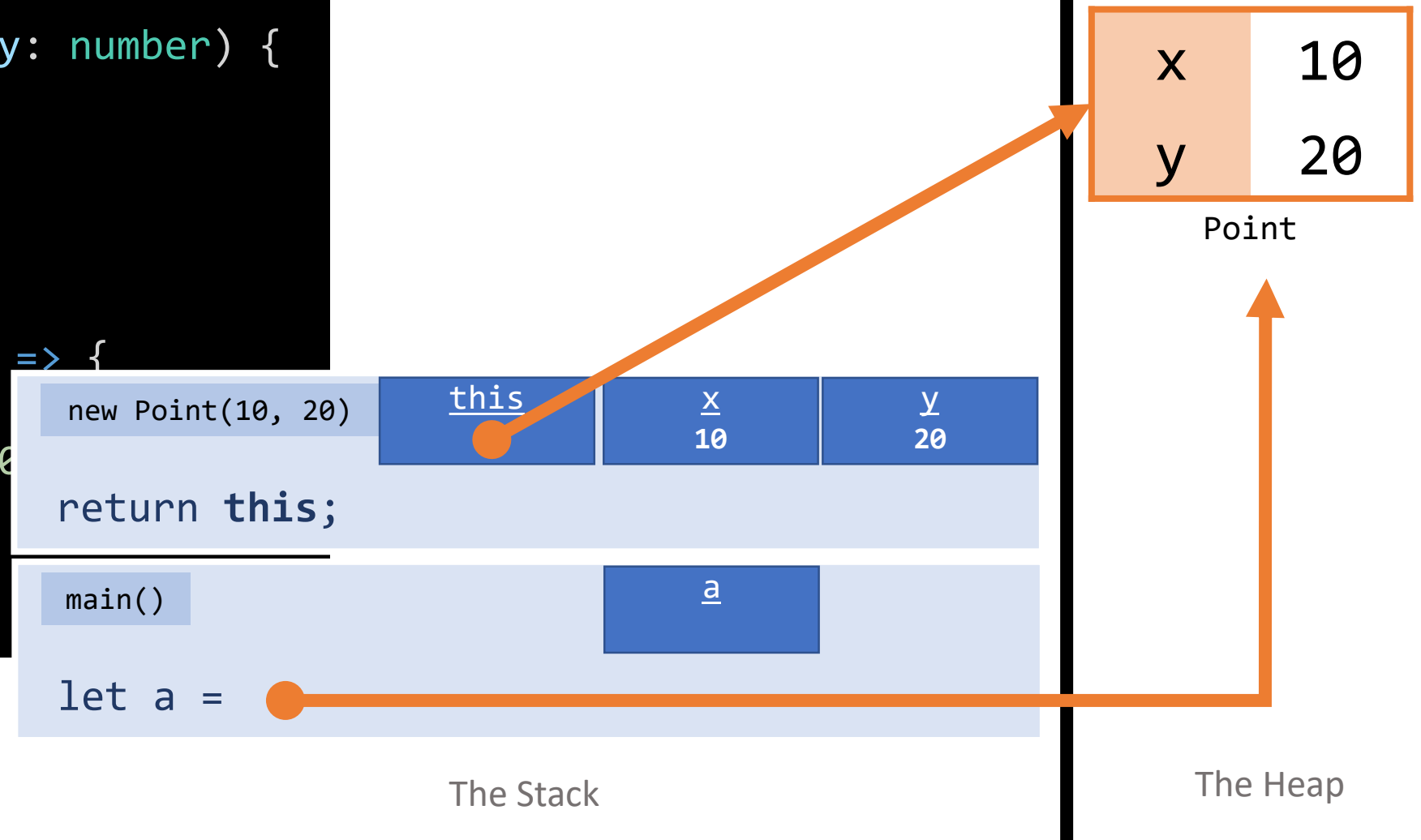
You will not ever see this line or need to write it.



Tracing a Constructor Call (9 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20
    print(a);

};
```

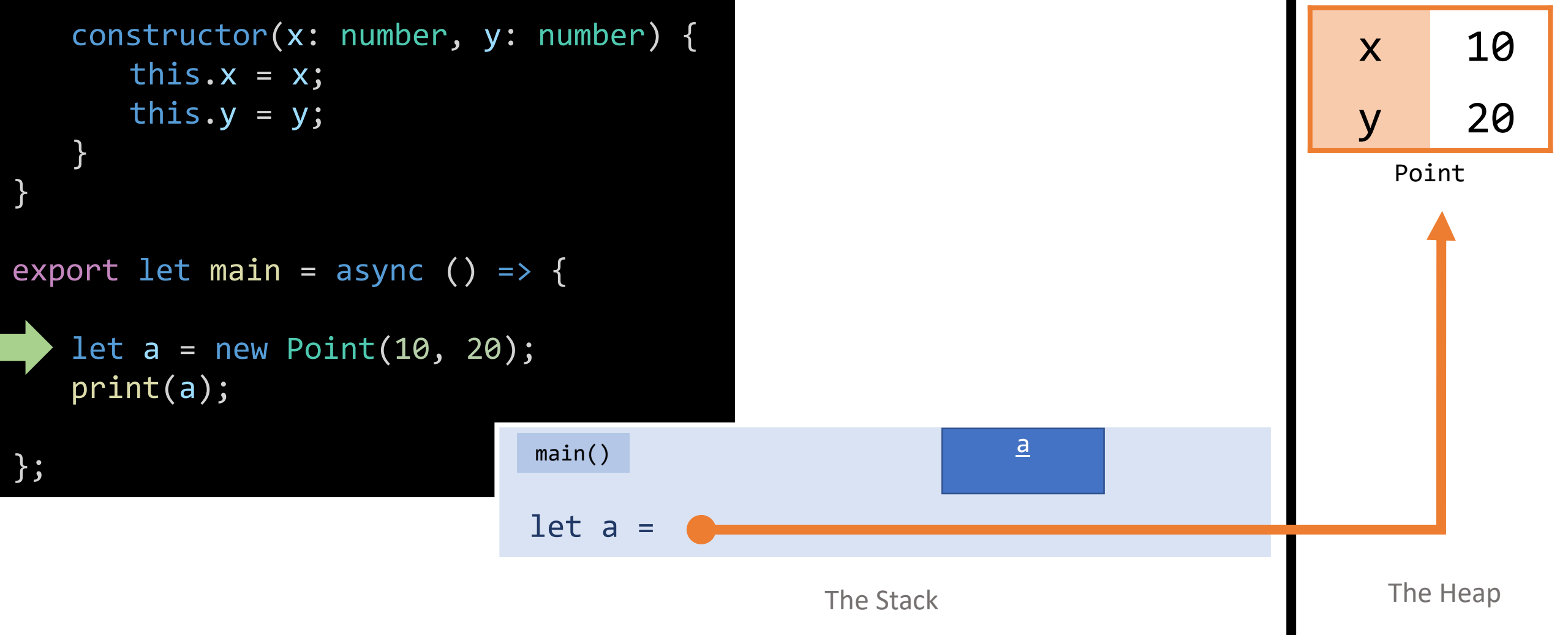
This implicit return statement will cause where the constructor was called:
`new Point(10, 20)`
to evaluate to the **this** reference.



Tracing a Constructor Call (10 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20);
    print(a);

};
```

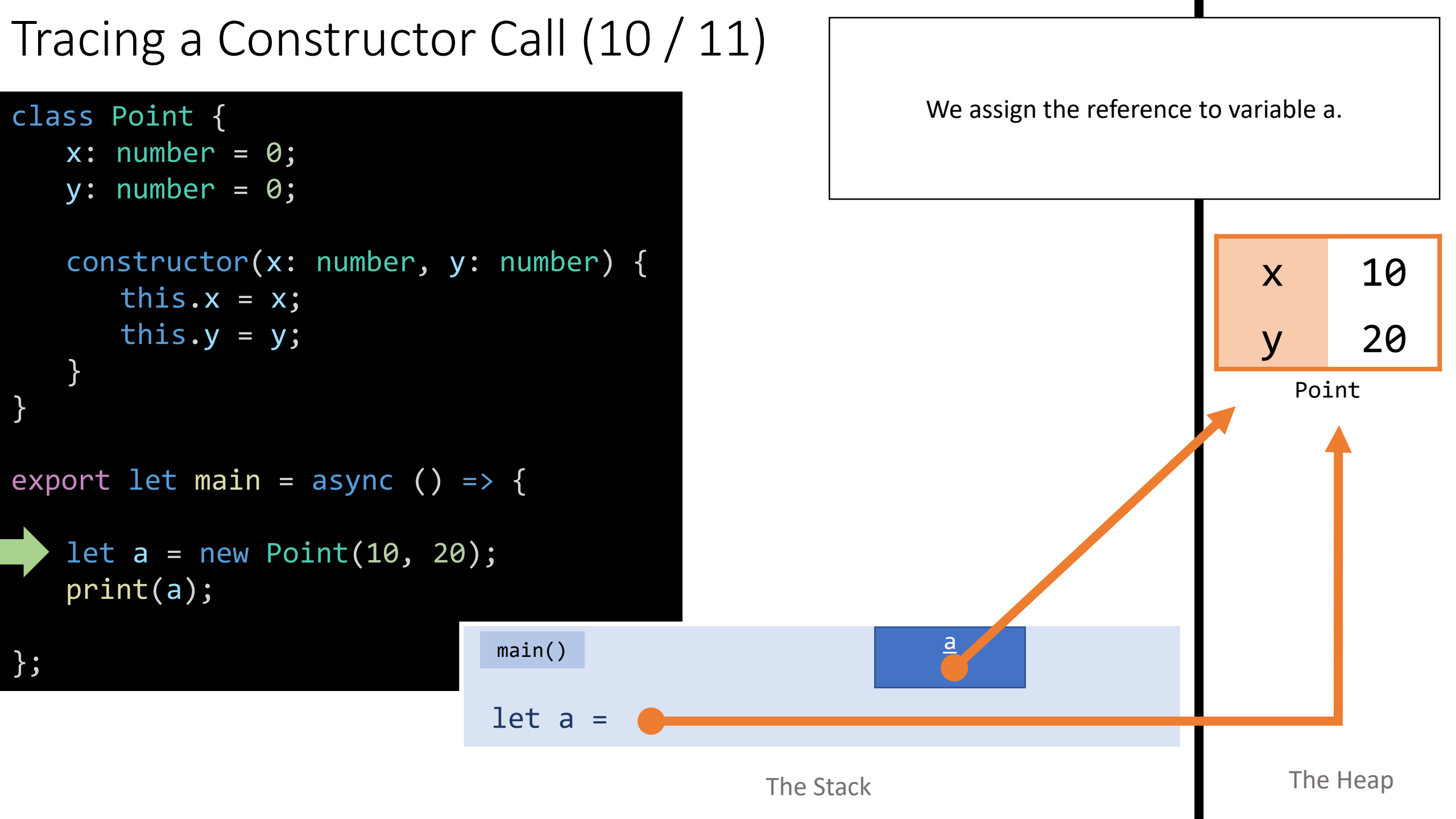
Finally, we jump back to the main function and return to our bookmark.



Tracing a Constructor Call (10 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20);
    print(a);

};
```

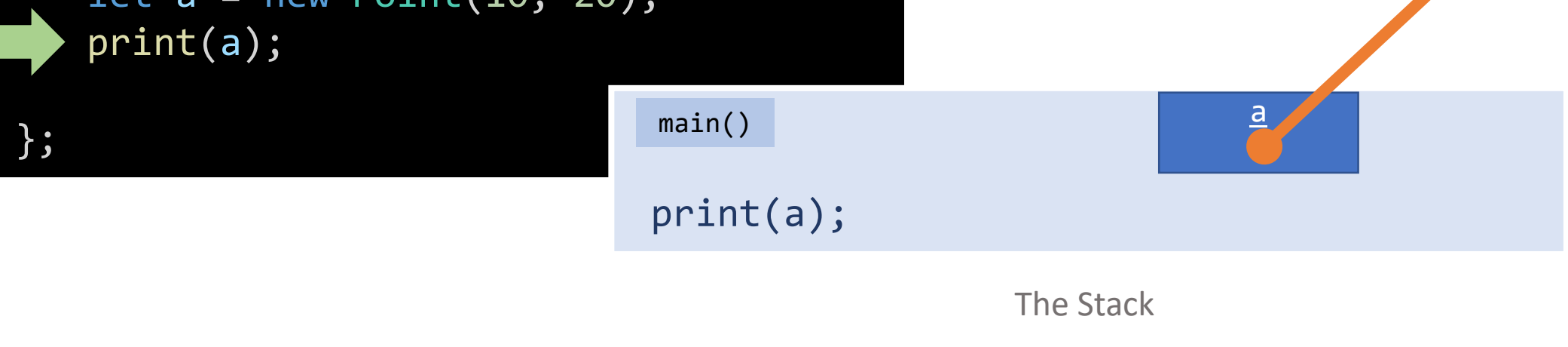
We assign the reference to variable a.



Tracing a Constructor Call (11 / 11)

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

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

export let main = async () => {

    let a = new Point(10, 20);
    print(a);

};
```

And our construction process is completed.
Next, we would print **a**.



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 Rectangle

```
Rectangle's Constructor's parameters:
(width: number, height: number)
```

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

1. Declare a variable named **r** of type Rectangle.

2. Initialize **r** by constructing a new Rectangle using the constructor above. Try giving it a `width` of 100 and `height` of 10.

3. "Render" it to our HTML by calling: **`svgTag.render(r);`**

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 r = new Rectangle(100, 10);
    artboard.render(r);

}
```

Scaling & Dimensions

- The `introcs` graphics library is automatically scaling and centering our graphics to fit our browser's size.
  - This is usually *really* convenient, but today we want to get a sense positioning.

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

Follow-along: Axes Part I

- Let's add two lines, one for each axis. The constructor for a Line is:

  `(x1: number, y1: number, x2: number, y2: number)`

  - Where (x1, y1) are the point of one end of the line and (x2, y2) are the other point.

```
let xAxis = new Line(0, 0, 100, 0);
let yAxis = new Line(0, 0, 0, 100);
artboard.render(xAxis);
artboard.render(yAxis);
```

- When we run this, we have an issue! Only the last shape rendered is displayed. How can we get multiple shapes to be rendered at once?

Introducing: **Group**

- Group is an SVG container element that "groups together" any number of shapes.

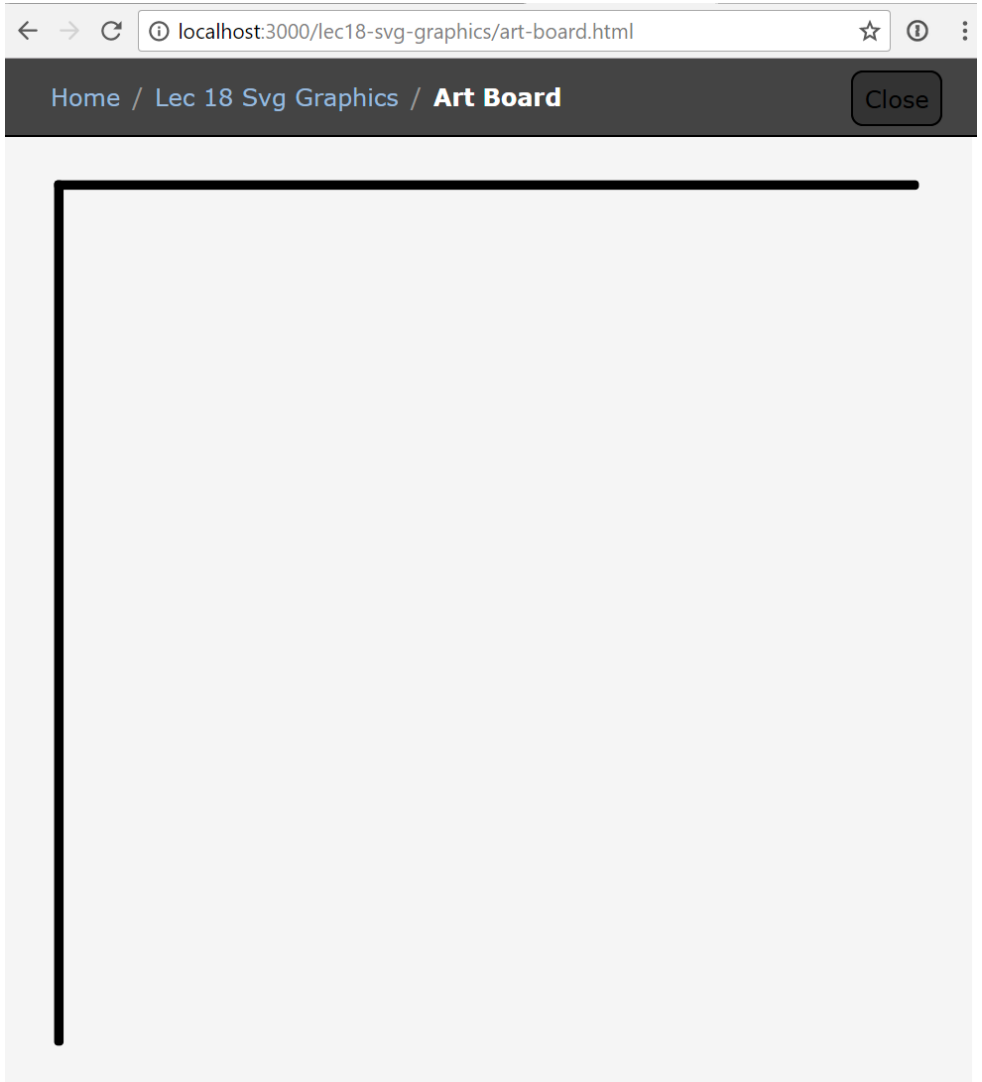
- A Group of shapes can be added to another Group of shapes forming a hierarchy or tree

- Elements added to a Group are called its "children"
  - The intuition here is based on thinking of this as a tree with parent and child elements

- Let's add a **Group** and add our **xAxis** and **yAxis** rectangles to it.

```
let g = new Group();

let xAxis = new Line(0, 0, 100, 0);
g.add(xAxis);

let yAxis = new Line(0, 0, 0, 100);
g.add(yAxis);

artboard.render(g);
```

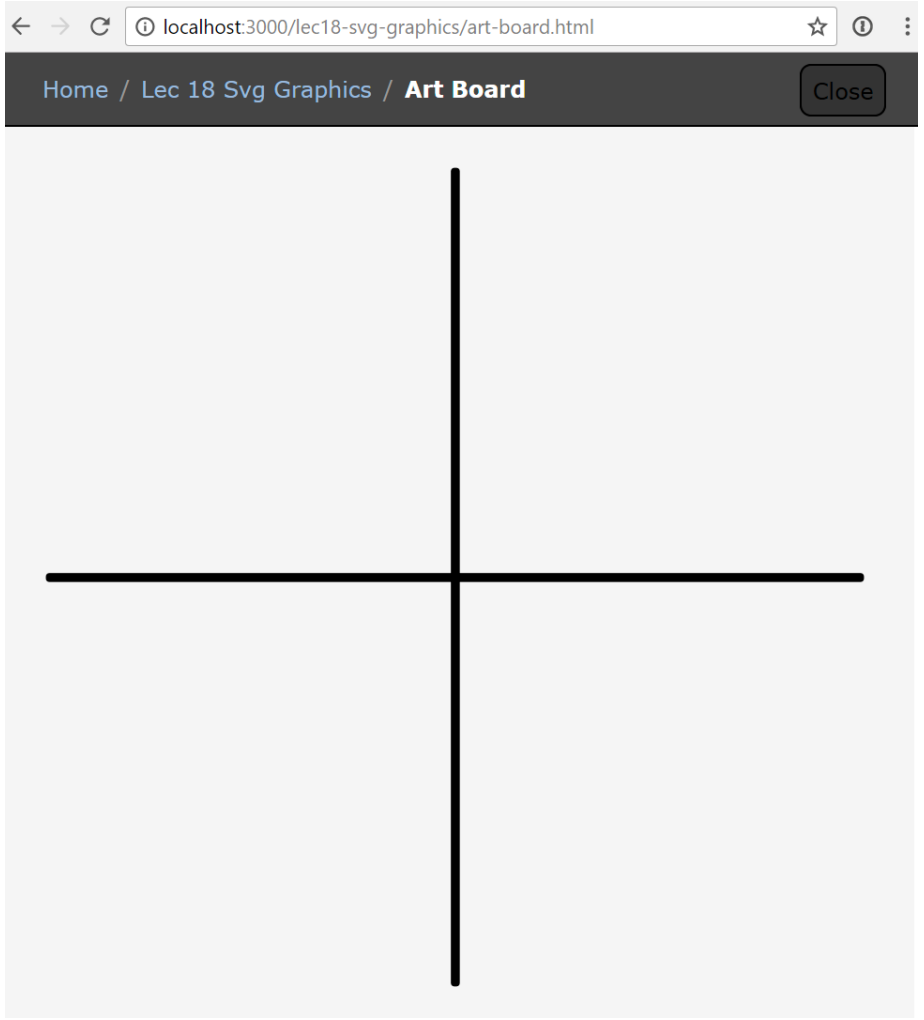
We've got a problem. This doesn't look axes...



- ... ***and*** *the y-axis is flipped compared to how we draw it in Math!*
  - This is a common convention in digital graphics. The positive y-axis typically points down the screen.
- The Rectangle constructor initializes its x0, y0 values to be 0, 0...
- ...unless we override them by specifying the optional parameters.
- Lets change the top-left corner of the xAxis to be at (-100, 0) and the top-left corner of the yAxis to be at (0, -50) to form a + sign

```
let xAxis = new Line(-100, 0, 100, 0);
g.add(xAxis);

let yAxis = new Line(0, -100, 0, 100);
g.add(yAxis);
```

Ah, that's better.



- Axes seem like a concept that would be useful to reuse.

- We've already setup the scaffolding of this class in Axes.ts, so we'll move the code for making these shapes into it. Let's try using them:

```
let axes = new Axes();
axes.height = 100;
axes.width = 100;
g.add(axes.getShapes());
```

```
function main(): void {

    let group: Group = new Group();
    let axes: Axes = new Axes(200, 200);
    group.add(axes.getShapes());


    svgTag.render(group);

}
```

```
export class Axes {
    width: number;
    height: number;

    constructor(width: number, height: number) {
        this.width = width;
        this.height = height;
    }

    getShapes(): Group {
        let group: Group = new Group();

        let xAxis: Rectangle = new Rectangle(this.width, 0.1, -this.width / 2, 0);
        group.add(xAxis);
        let yAxis: Rectangle = new Rectangle(0.1, this.height, 0, -this.height / 2);
        group.add(yAxis);

        return group;
    }
}
```

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 the parameter values width and height to the Axes object's properties (look back at 00-constructor-app.ts as a reminder)

- This will cause an error in your app-script.ts where the new Axes() object was created. Go ahead and add arguments to this constructor so that the initial width and height are 100.

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

```
export class Axes {
    width: number;
    height: number;

    // TODO: Declare constructor
    constructor(width: number, height: number) {
        this.width = width;
        this.height = height;
    }
    //…
```

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

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.

Hands-on #2) 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. Declare a counting variable **`i`** and initialize it to 0
3. Write a while-loop using **`i`** as its counter that will loop **`count`** times… ***don't forget to increment i inside of the loop***!
4. Inside of the while-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 to come up with an expression by multiplying **`i`** by some amount to space the circles out
5. Add the circle to the group named **`g`**.
6. Check-in when you have at least 1 circle visible on your graphics.

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

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 + 1) / (count)`**
  - Why (i + count)? Because we are starting from i/count % and the final loop iteration to be 100%

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

    let c = new Circle(5, i * 20 - 90, 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)

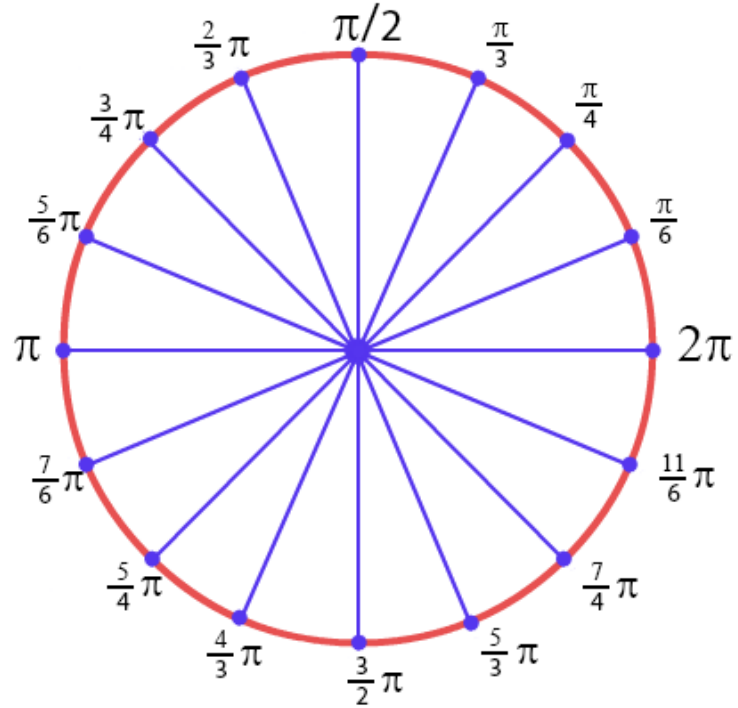
- We want to pick numbers between 1.0 and 0.0 evenly as we construct each circle.

- 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.

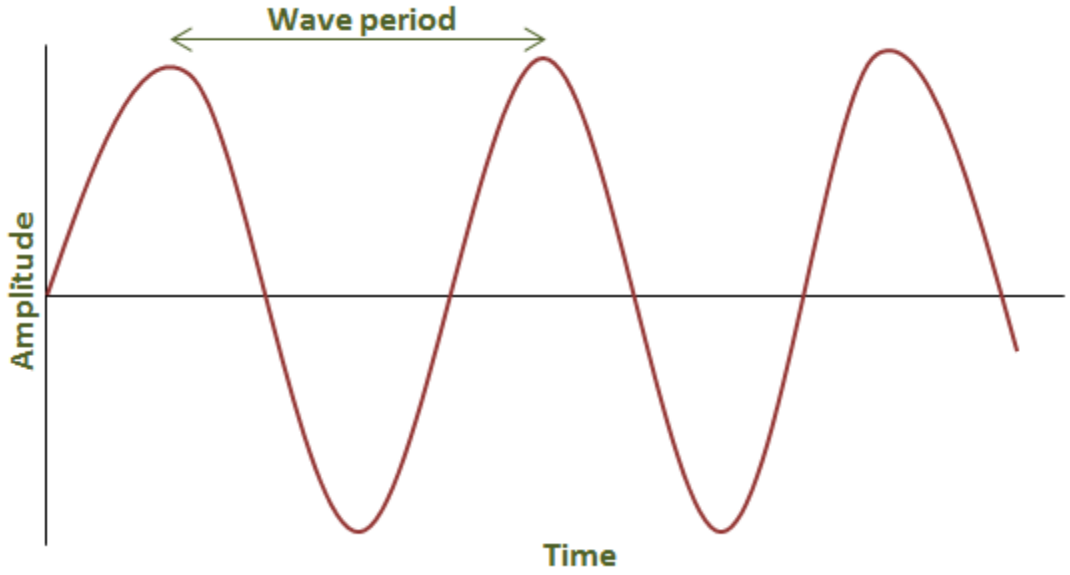


```
let artboard = new SVG("artboard");
const RADIANS = Math.PI * 2;
```

```
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;

let count = 10;
let i = 0;
while (i < count) {
    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);
```

Reference Documentation for IntroCS Graphics

- Want to explore shapes and graphics more?

- You can find additional 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

- There will be additional shapes and concepts added to the semester as we move forward (i.e. Lines, Images, Ellipses, etc)