Object-Oriented Programming

Methods and Constructors

Lecture 13

Go ahead and open VSCode – In the terminal:

npm run pull

npm start

Object-oriented Programming

- So far we've used objects as compound data types
  - i.e. to model a row of data in a spreadsheet

- We've written functions, *separate from classes*, that operate on objects

- The only thing we've been able to *do* with an "object" is access and assign values to its properties

- Object-oriented programming allows us to give objects *capabilities*
  - We'll do this with two special kinds of functions: methods and constructors

Review of **Classes** and **Objects**

- A class defines a new **Data Type**
  - The class definition specifies properties

- Instances of a class are called **objects**
  - To create an object you must use the **new** keyword: `new <Classname>()`

- *Every object of a class* has the **same properties** but has **its own values**

- Objects are reference-types
  - variables do not hold objects, but rather *references to objects*

Follow-along: Simple Method App

- Let's implement and call the sayHello method example from previous slides in 00-simple-method-app.ts

```typescript
class Person {
    // ... properties elided...

    sayHello(): void {
        print("Hello, world");
    }
}
```

```typescript
let aPerson = new Person();
aPerson.sayHello();
```

Introducing: Methods

- A **method** is a special function defined in a class.
  - Everything you know about a function's parameters, return types, and evaluation rules are the same with methods.
  - *Syntactically*, you'll notice there *are* some minor differences. No let keyword, no assignment operator, and no arrow.

- Once defined, you can call a method <u>**on**</u> any object of that class using the dot operator.
  - Just like how properties were accessed except followed by parenthesis and any necessary arguments

```
class ClassName {

    // Properties Elided

    <methodName>(<parameters>): <returnType> {
        <method body>;
    }


}
```

```
let a = new ClassName();
a.methodName();
```

Functions vs. Methods

1. Let's define a *silly* **function.**

```
let sayHello = (): void => {
    print("Hello, world");
};
```

2. Once defined, we can then call it.

```
sayHello();
```

3. Now, let's define that same function as a **method** *of the Person class*.

```
class Person {
    // ... properties elided...

    sayHello(): void {
        print("Hello, world");
    }
}
```

4. Once defined, we can call the method on any Person object:

```
let a = new Person();
a.sayHello();
```

Hands-on: Practice with the **this** keyword

- In 01-this-keyword-app.ts...

1. At TODO #1, define the **toString method** to the right.

2. In the `main` function, at TODO's #2 , call the **toString** method on **Points a** and **b** respectively.

```
class Point {

    // ... Properties Elided ...

    toString(): string {
        return this.x + ", " + this.y;
    }

}
```

Check-in on PollEv.com/compunc when complete.

Method's Special Feature:
Methods can *refer* to the object the method was called on.

Consider this plain **function.**
Notice that its parameter **p** is
a reference to a Point object.

```
let toString = (p: Point): string => {
    return p.x + ", " + p.y;
};
```

To call it, we would pass a
reference to a Point object as an
argument.

```
let a = new Point();
print(toString(a));
```

It turns out we *can* write a method
that does the same thing and it can
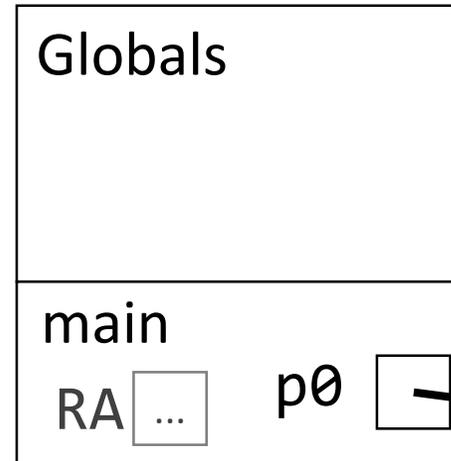be called like the example to the
right.

```
let a = new Point();
print(a.toString());
```
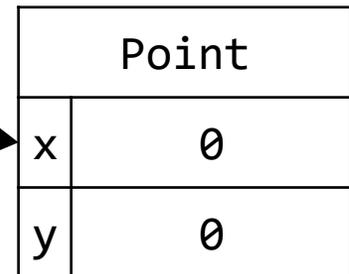
How does this magic work???

Suppose the processor *just* completed this line…

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   toString(): string {
06     /** Elided */
07   }
08 }
09
10 export let main = async () => {
➡  let p0 = new Point();
12   print(p0.toString());
13 };
```
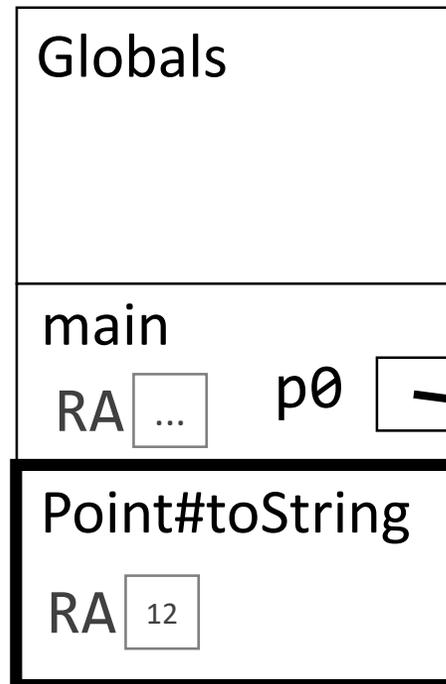
The Stack

Globals

main
RA […]   p0 [ ]

The Heap

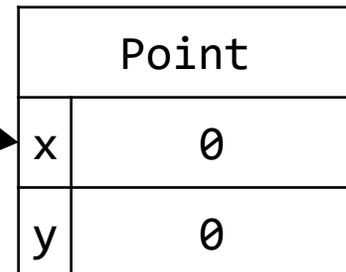| Point	
x	0
y	0

How is this *method call* processed? First, a frame is added...

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   toString(): string {
06     /** Elided */
07   }
08 }
09
10 export let main = async () => {
11   let p0 = new Point();
12   ▶nt(p0.toString());
13 };
```

The Stack

Globals

main
RA ...

p0

Point#toString
RA 12

The Heap

Point	
x	0
y	0

What's up with this pound sign? It's conventional across many programming languages to identify a method by **ClassName#method**.

THEN, a reference named **this** is established TO the object the method was called on.... and *this* is *all the magic* of a **method call**.

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   toString(): string {
06     /** Elided */
07   }
08 }
09
10 export let main = async () => {
11   let p0 = new Point();
12     nt(p0.toString());
13 };
```
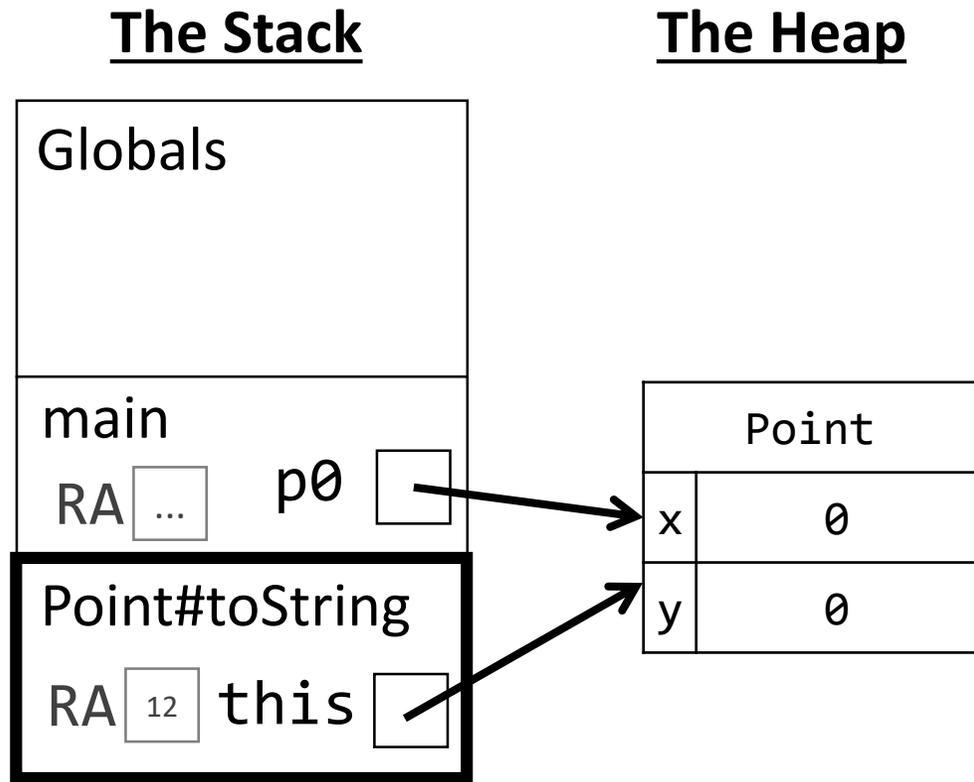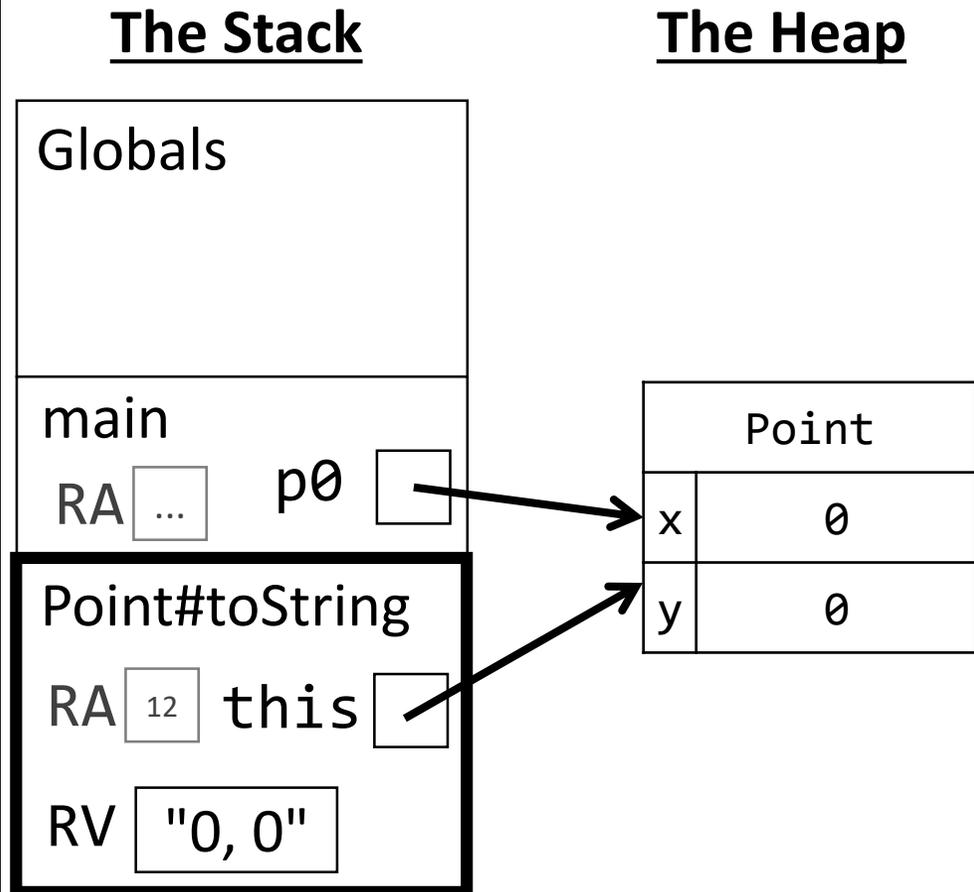
The Stack

The Heap

Globals

main

RA [...]   p0 [ ]

Point#toString

RA [12] this [ ]

Point

| x | 0 |
| y | 0 |

The processor is performing this step magically behind the scenes.

When name resolution occurs inside of a method, the special variable *this* always refers to the object the method was called on.

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   toString(): string {
06     return this.x + ", " + this.y;
07   }
08 }
09
10 export let main = async () => {
11   let p0 = new Point();
12   print(p0.toString());
13 };
```

The Stack

The Heap

Globals

main

RA ...   p0 ▢

Point#toString

RA 12 this ▢

RV "0, 0"

Point

x   0

y   0

Method's Special Feature:
Methods can refer to the object the method was called on.

When a method is called, inside of the function, a special "variable" is initialized named **this**

The **this** keyword *refers to* the object the method was called upon.

```
class Point {

    // ... Properties Elided ...

    toString(): string {
        return this.x + ", " + this.y;
    }


}
```

```
let a = new Point();
a.x = 110;
a.y = 110;
print(a.toString());
```

When the above code jumps to *toString*, **this** will refer to the same Point object **a** refers to.

```
let b = new Point();
b.x = 401;
b.y = 401;
print(b.toString());
```

When the above code jumps to *toString*, **this** will refer to the same Point object **b** refers to.

Hands-on: Practice with the **this** keyword

- In 02-stateful-object-app.ts, let's make it easy to move a Point relative to its current position.

1. At #1, increase the **x** property of the object **translate** is called on by **dx.** Then, increase the **y** property of the object **translate** is called on by **dy**.
   - Hint: reassign `this.x` and `this.y` by adding `dx, dy respectively.`

2. Call `translate` on **Point a** in the `main` function using any values you'd like at each of the TODOS # 2 and #3.

3. Once you've tested that it works, check-in on PollEv.com/compunc

```typescript
translate(dx: number, dy: number): void {
    this.x += dx;
    this.y += dy;
}
```

Follow-Along: Distance Method

- Let's add a method to compute the distance between two points.

- We'll specify the 2nd point as a parameter named *other*.

- We'll also make use of the special Math function:
  - Math.sqrt(x) computes square root

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
class Point {
    // … elided …
    distanceTo(other: Point): number {
        let xDelta2 = (other.x - this.x) ** 2;
        let yDelta2 = (other.y - this.y) ** 2;
        return Math.sqrt(xDelta2 + yDelta2);
    }
}
```

```
// Calling the distanceTo method
print(a.distanceTo(b));
```

Why have both functions and methods?

- Different schools of thought in *functional programming-style (FP)* versus *object-oriented programming-style (OOP).*
  - *Both are equally **capable**, but some problems are better suited for one style vs. other.*

- FP tends to shine with *data processing* problems
  - Data analysis programs like processing *stats* and are natural fits

- OOP is great for stateful systems like *user interfaces, simulations, graphics*

- Methods allow objects to have "built-in" functionality
  - You don't need to import extra functions to work with an object, they are bundled.
  - As programs grow in size, methods and OOP have some extra capabilities to help teams of programmers avoid accidental errors. You'll see this in 401!

Method Call Tracing Steps

When a method call is encountered on an object,

1. The processor will determine the class of the object and then confirm it:
   1. Has the method being called defined in it.
   2. The method call's arguments agree with the method's parameters.

2. Next it will initialize the RA, parameters, *and* the **this** keyword
   - The **this** keyword is assigned a reference to the object the method is called on

3. Finally, when the method completes, processor returns back to the RA.

Constructors

- An object's properties must be initialized before the object is usable

- A constructor allows you to
  1. Specify initial values of properties upon construction of an object
  2. Require certain properties be specified

- A constructor is just a special method
  - Name is **constructor**
  - Also has a variable named **this**
  - Return type is an object of its class

- A class' constructor is called each time the **new <Classname>** expression is evaluated.

Before

```
let a = 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;
    }

}
```
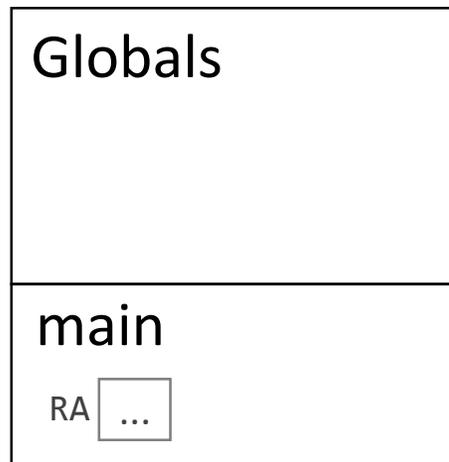
After

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

Tracing a **constructor**. Suppose we're about to construct!

```
01  class Point {
02    x: number = 0;
03    y: number = 0;
04
05    constructor(x: number, y: number) {
06        this.x = x;
07        this.y = y;
08    }
09  }
10
11  export let main = async () => {
12    let p0 = new Point(10, 12);
13  };
```

The Stack

The Heap

Globals

main

RA ...

When the frame is established, a new Point object is referred to by **this**. Arguments are assigned to parameters in the constructor's frame.

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   constructor(x: number, y: number) {
06     this.x = x;
07     this.y = y;
08   }
09 }
10
11 export let main = async () => {
12   let p0 = new Point(10, 12);
13 };
```

The Stack

The Heap

Globals

main

RA ...

Point#constructor

RA 12   this

x 10

y 12

Point

| x | 0 |
| y | 0 |
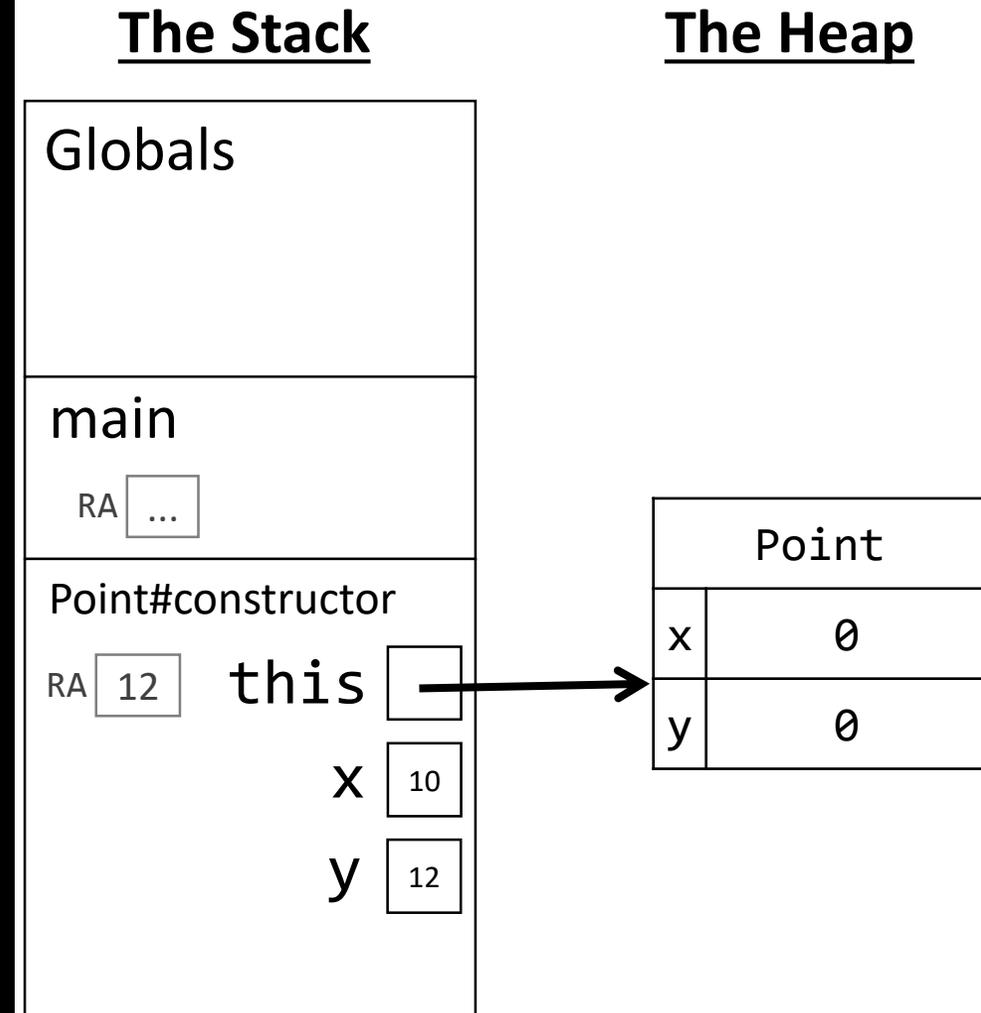
Notice the default property values are initialized just before entering the constructor.
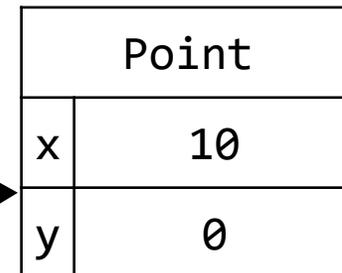
Using name resolution, the value of **x** from the constructor's frame is assigned to **this.x**, which is the new Point object's **x** property.

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   constructor(x: number, y: number) {
06     this.x = x;
07     this.y = y;
08   }
09 }
10
11 export let main = async () => {
12   let p0 = new Point(10, 12);
13 };
```
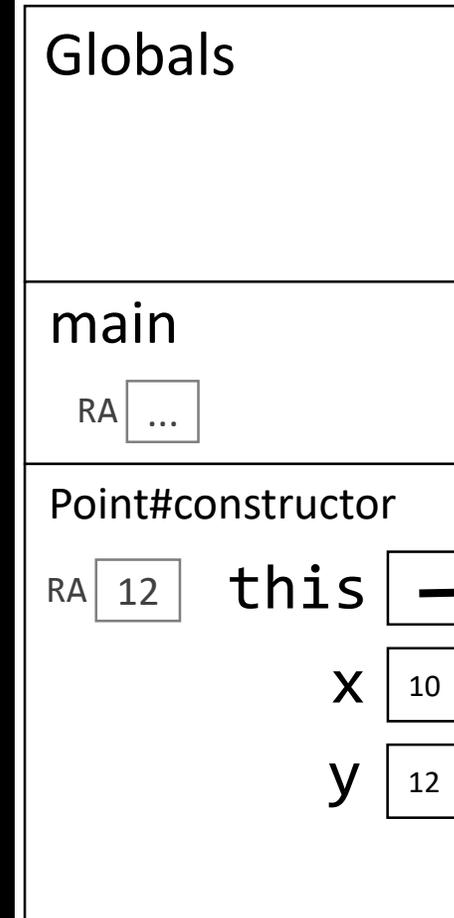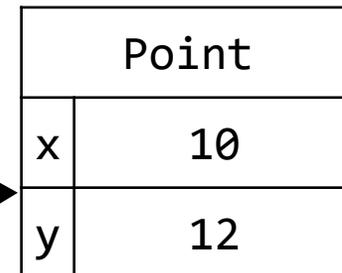
The Stack

Globals

main

RA ...

Point#constructor

RA 12   this ☐

X 10

y 12
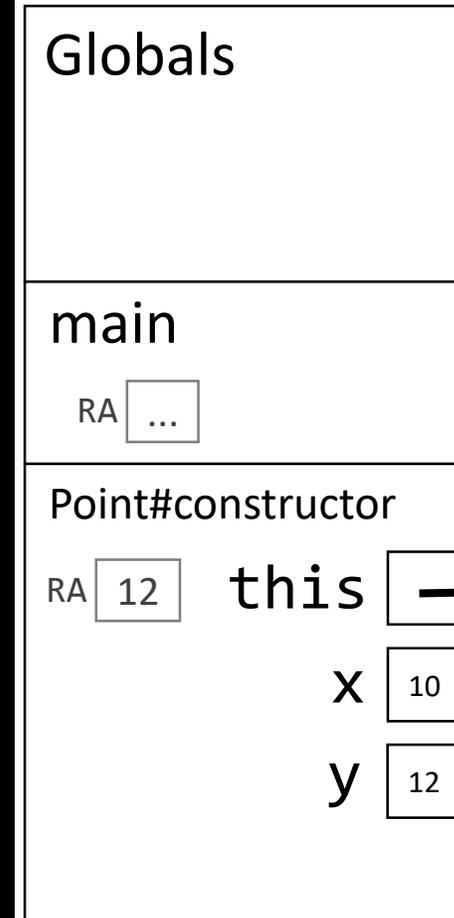
The Heap

Point	
x	10
y	0

Using name resolution, the value of **y** from the constructor's frame is assigned to **this.y**, which is the new Point object's **y** property.

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   constructor(x: number, y: number) {
06     this.x = x;
0      this.y = y;
08   }
09 }
10
11 export let main = async () => {
12   let p0 = new Point(10, 12);
13 };
```
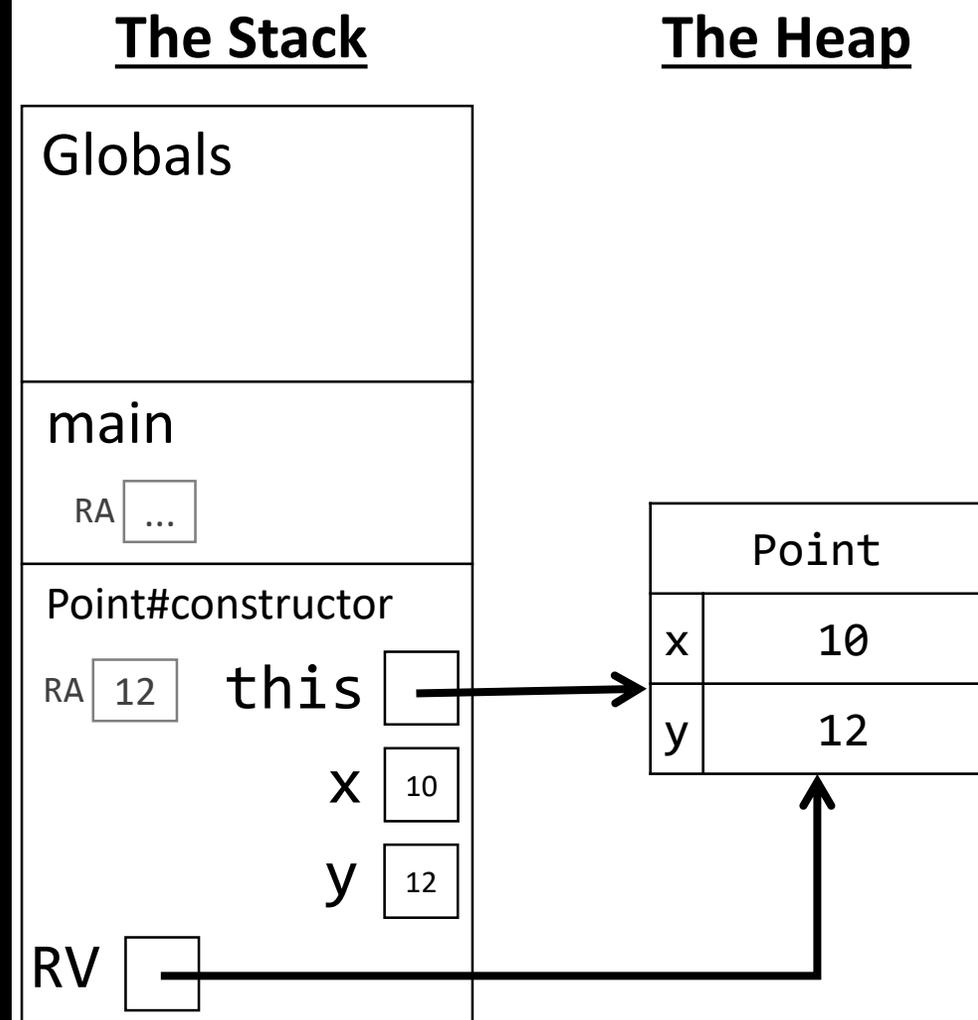
The Stack

Globals

main

RA ...

Point#constructor

RA 12   this

x 10

y 12

The Heap

Point

x	10
y	12

The **return value of a constructor** is implicitly the same reference as **this**.

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   constructor(x: number, y: number) {
06     this.x = x;
07     this.y = y;
08   }
09 }
10
11 export let main = async () => {
12   let p0 = new Point(10, 12);
13 };
```

The **return value of the constructor** is assigned to **p0** in **main**.

```
01 class Point {
02   x: number = 0;
03   y: number = 0;
04
05   constructor(x: number, y: number) {
06     this.x = x;
07     this.y = y;
08   }
09 }
10
11 export let main = async () => {
12   let p0 = new Point(10, 12);
13 };
```