Values, References and Methods

Lecture 17

Announcements

- Review Session – Tomorrow at 5pm in SN014
- Worksheet Out – Due Tuesday 4/3
- No tutoring this week due to holiday Friday. Come to office hours for conceptual help anytime, though!

CS Diversity Panel: Tomorrow at 6pm in SN011

- Interested in continuing in CS?
- Ask questions and hear experiences from women and people of color in tech.
- On the panel: undergraduates, graduates, industry, and the CS director of undergraduate studies.




1. What is printed when the user responds to the prompt with "B"?

```
export let main = async () => {
   intro();
   let c = await promptString("?");
   outro(c);
};

let intro = (): void => {
   print("A");
};

let outro = (s: string): void => {
   print(s);
   print("D");
};
```

Functions that return **`void`**

- Are there times when you need a function that *returns nothing?*
  - Yes! You've used a void function frequently this semester: **`print`**

- There are three common reasons to write a **`void`** function:
  1. To give a command to an external system that has no logical return value (i.e. printing something to the screen)
  2. To *mutate* an object, an array, or a global variable
  3. To reuse of a common series of steps that do not result in a return value

- What is a *mutation?* Mutating an:
  - Array means changing element(s) value(s)
  - Object means changing property values
  - Global variable means reassigning a new value

Pure Functions – "That function is *so pure*."

- A **pure function** is one whose return value depends *only* on its parameters.
  - Until now, all of the functions you've written except for `main` were pure.

- **Impure functions** depend upon or mutate external *state,* as well.
  - (i.e. user input, global variables, or, as we'll see later today: object properties)
  - `void` functions are, by nature, impure. Functions that return values *can* be impure, too.

- Pure functions are *far* easier to use, write, and reason about than impure.
  - All of their logic is self-contained!
  - Pure functions reduce the sources of surprising bugs to its parameter values' edge cases.

- Impure functions have important uses, though, and can't be avoided at times.
  - Pro-tip: You'll write better programs if your programs do as little as possible in impure functions and as much as possible in pure functions.

What are **b**'s elements after this code runs?

```
let a = [10, 20, 30];
let b = a;
a[0] = 20;
b[2] = 20;
print(b);
```

Value Types vs. Reference Types

- Primitive types (number, string, boolean) are **value types**
  - Variables hold *copies* of actual values.
  - Assigning one variable to another ***copies the value***.
  - Changing a copied variable's value does not impact original or vice-versa.

- Composite types (classes, arrays) are **reference types**
  - Variables hold *references* to actual values.
  - Assigning one variable to another ***copies the reference***. Both variables now refer to the same value in memory.
  - Modifying a referenced value will impact all references to it.

Value Types Visualized

```
let a = "hello";
let b = a;
a = "world";
```

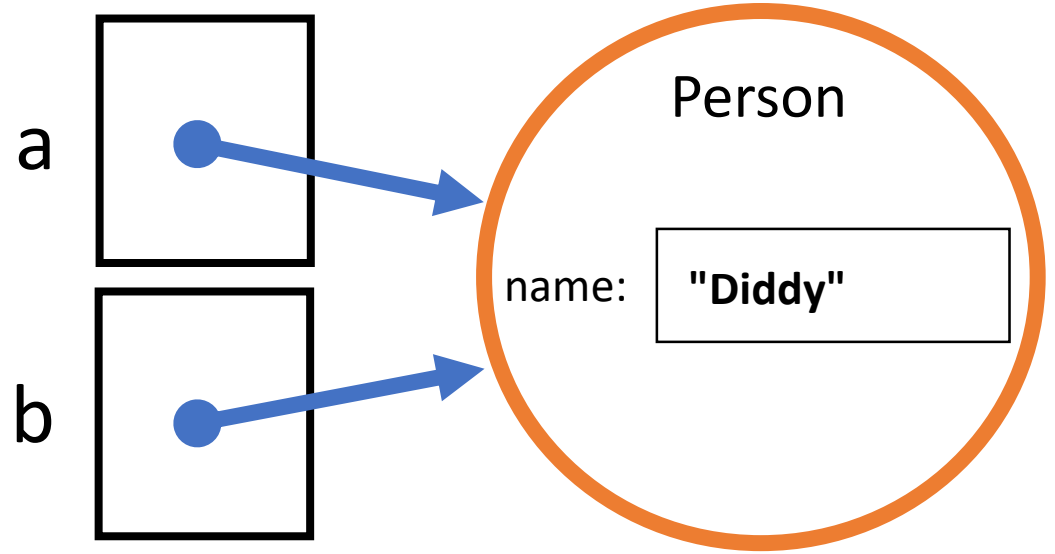
Notice: **b** was assigned **a**. This copied the string "hello" to **b**.

The variable **a** was then assigned the string "world".

This had no impact on **b** because string variables are value types.

Reference Types Visualized - Objects

```
let a = new Person();
a.name = "Sean Combs";
let b = a;
a.name = "Diddy";
```

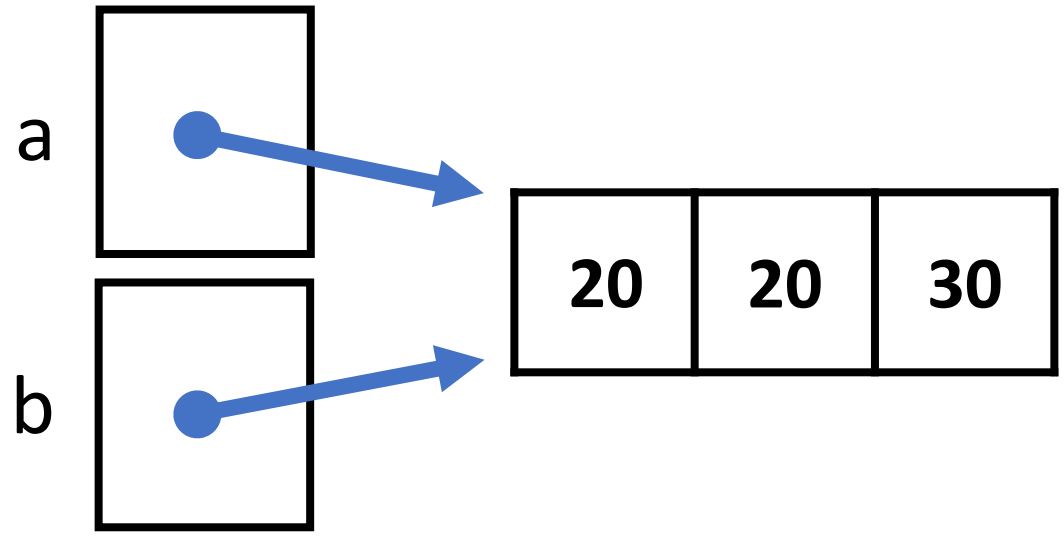


Notice: When the **a** variable is assigned a **new Person** object, it is assigned reference to the Person object.

When **b** is assigned **a**'s value, the **reference** is copied, ***not the object***.
Variables **a** and **b** now *refer* to the same object.

Reference Types Visualized - Arrays

```
let a = [10, 20, 30];
let b = a;
a[0] = 20;
```



Notice: When the **a** variable is assigned a new array, it *refers* to the array.

When **b** is assigned **a**'s value, the **reference** is copied, ***not the array***.
Variables **a** and **b** now *refer* to the same array.

Hands-on: Calling a **`void`** function with a reference type parameter and mutating it

- Open 00-references-app.ts

1. In main, declare a variable named numbers and initialize it with an array of a few numbers

2. Append a value to your array by calling the `append` function, passing it your array as the first argument and any valid second argument

3. Print the variable you declared in 1

- Was the number appended? If so, check-in. Talk through how and why this works with your neighbor.

When do multiple variables commonly *refer* to the same object or array?

- When passing an object or array as a parameter!

```
import { print } from "introcs";

export let main = async () => {
   let numbers = [1, 2];
   append(numbers, 3);
   print(numbers);
};

let append = <T> (a: T[], item: T): void => {
   a[a.length] = item;
};

main();
```

Changing Gears:
Object-oriented Programming

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***

Introducing: **Methods**

- A **method** is just a function defined in a class.
  - Everything you know about a function's parameters, return types, and evaluation rules are the same with methods.
  - *Syntactically*, the differences are we do not write 'let' and we do not need a semi-colon after the closing curly brace
- Once defined, you can call a method on any object of that class using the dot operator.
  - Just like how properties were accessed except followed by parenthesis and any necessary arguments

```
class Point {

    // Properties Elided

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

}
```

```
let a = new Point();
print(a.methodName());
```

Functions vs. **Methods**

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

```
let sayHello = () => {
    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* `Point` *class*.

```
class Point {
    // ... properties elided...

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

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

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

Follow-along: Simple Method App

- Let's implement and call the sayHello method example from previous slides in 01-simple-method-app.ts

```
class Point {
    // ... properties elided...

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

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

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 can this magic work???

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: Point = 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.

Follow-along: Practice with the **this** keyword

- In 02-this-keyword-app.ts...

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

2. In the `main` function, at TODO's #2 and #3, call the **`toString`** method on **`Points a`** and **`b`** respectively.

```
class Point {

    // ... Properties Elided ...

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

}
```

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

- In 02-this-keyword-app.ts, let's make it easy to shift a Point's **x** property.

1. At TODO#4, define a method named **`shiftX,`** that has a single number parameter named **amount** and a **`void`** return type.

2. Increase the **`x`** property of the object **shiftX** is called on by **amount**.
   - Hint: reassign **`this.x`**

3. Call **`shiftX`** on **`Points a`** and **`b`** in the `main` function using any values you'd like, after their properties were assigned.

4. Once you've tested that it works, check-in on PollEv.com/comp110

```
shiftX = (amount: number): void => {
    this.x = this.x + amount;
}
```

```
b.shiftX(10);
```

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 some special Math functions:
  - Math.pow(x, n) raises **x** to the **n**th
  - Math.sqrt(x) computes square root

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

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

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

Tracing a Method Call (1 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

The program begins in the **`main`** function.

The **stack** is where a function's variables are stored.
The **heap** is where objects and arrays are stored.

The Stack

main()

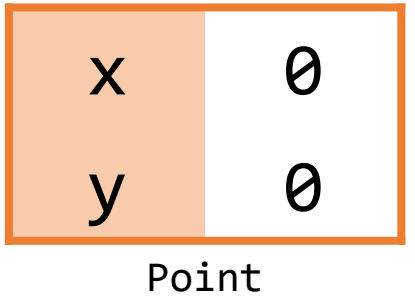
```
let a = new Point();
```

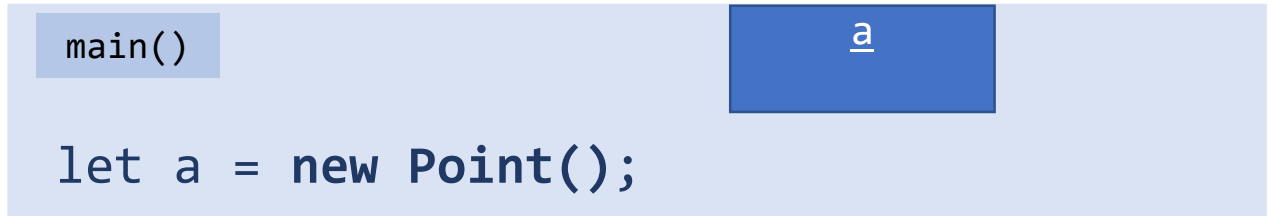
The Heap

Tracing a Method Call (2 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

When a new object is constructed, space is allocated for it in the heap and its properties are initialized.





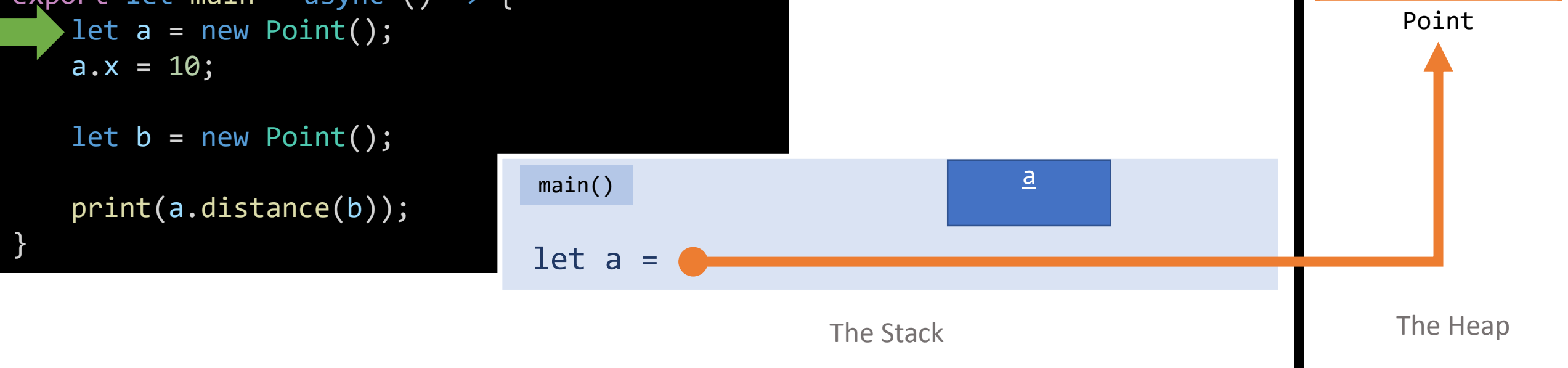
The Stack

The Heap

Tracing a Method Call (3 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

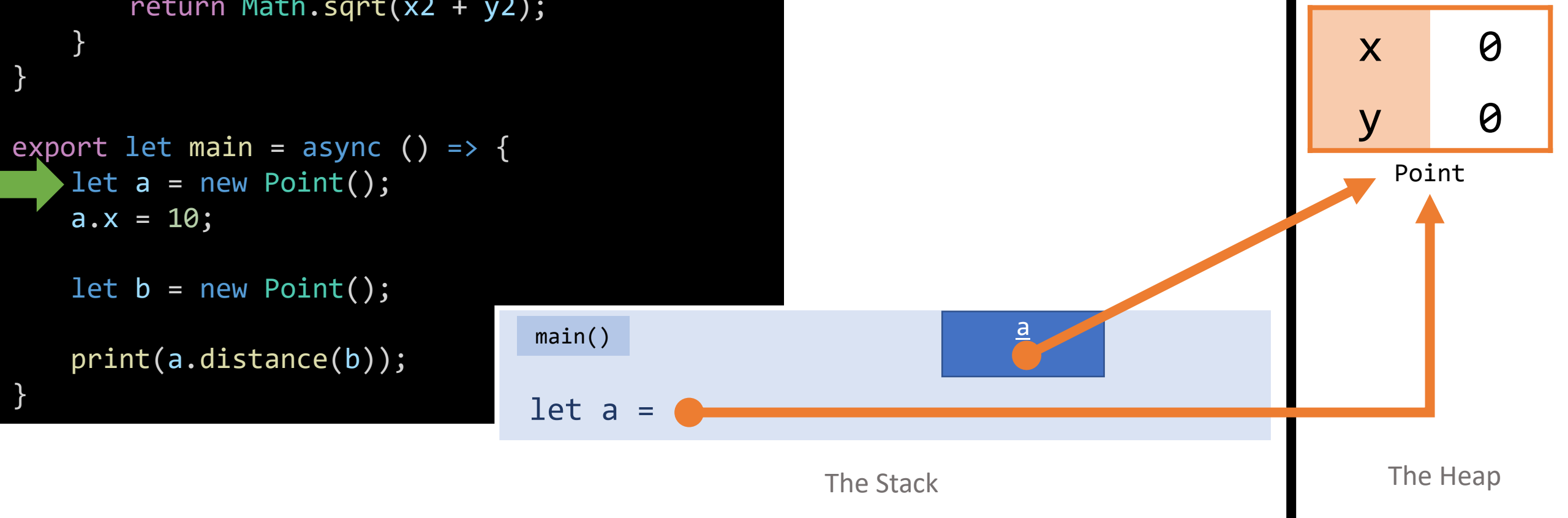
After it is initialized, **`new <Class>`** evaluates to a *reference* to the object constructed.



Tracing a Method Call (4 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

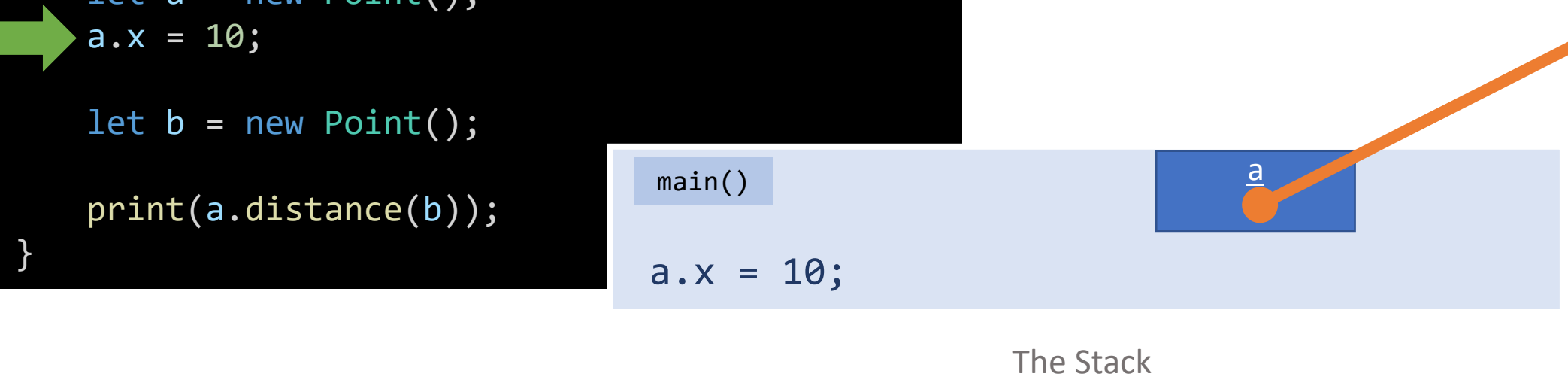
The reference is then assigned to the variable **a**.



Tracing a Method Call (5 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

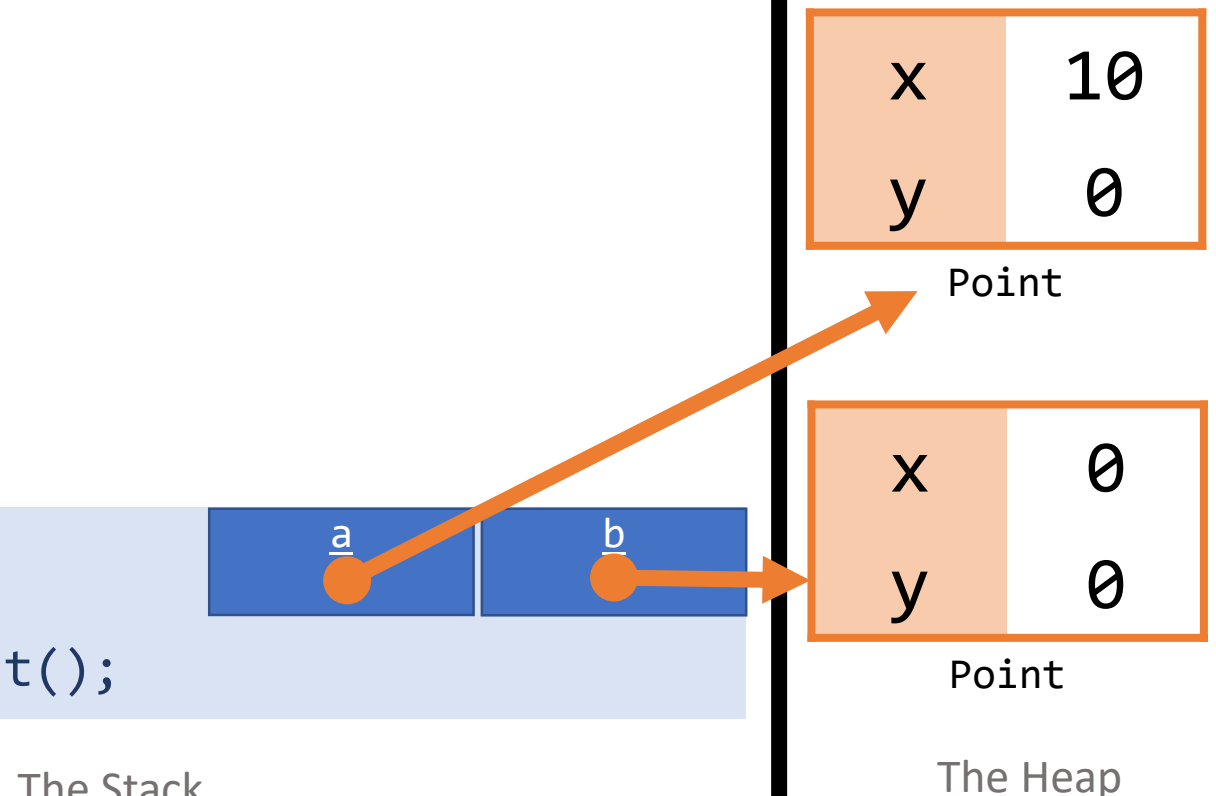
The **x** property of **a** is assigned 10.



Tracing a Method Call (6 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

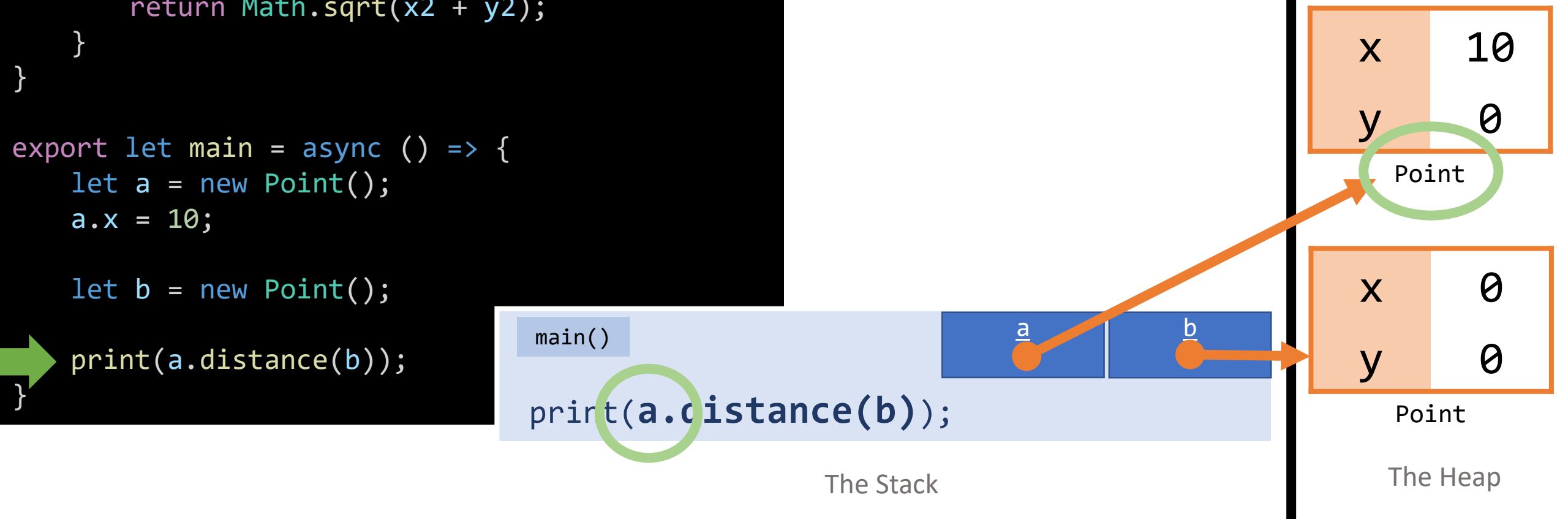
The same series of steps that happened when initializing variable **a** to be a reference to a new Point would also apply here to initialize **b**. We are skipping those steps for brevity.



Tracing a Method Call (7 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

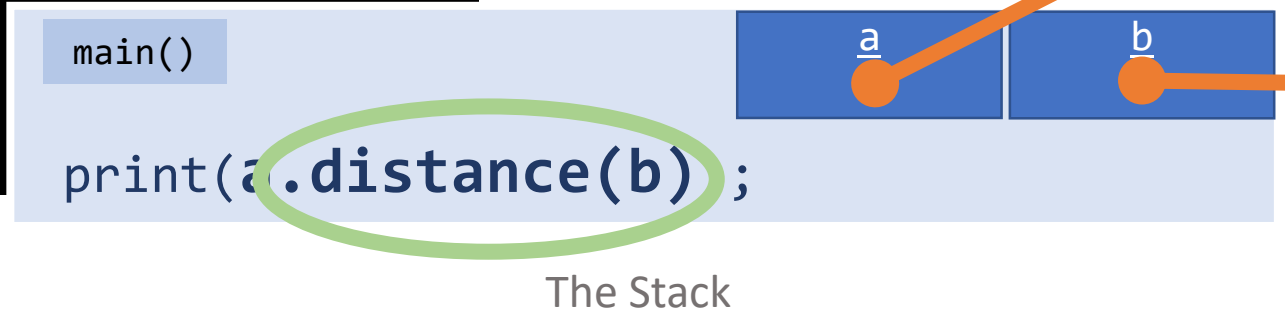
We've reached a method call! An important series of steps follows. First, the processor needs to find the class of the object the method was called on. What is **a**'s class?

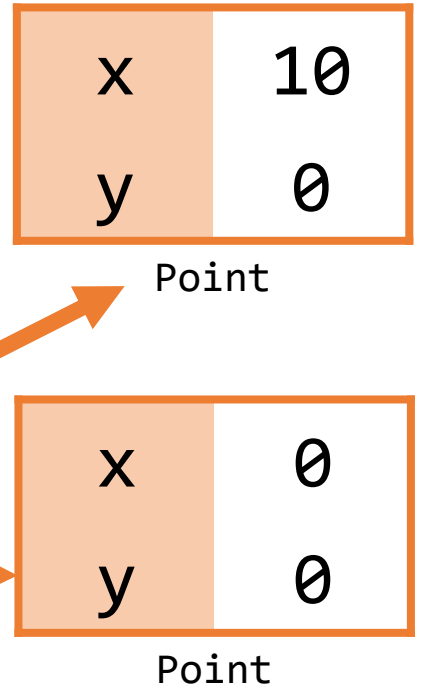


Tracing a Method Call (8 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

Next, the processor will check to be sure this class a definition for the method being called. Like with functions, arguments and parameters must match.

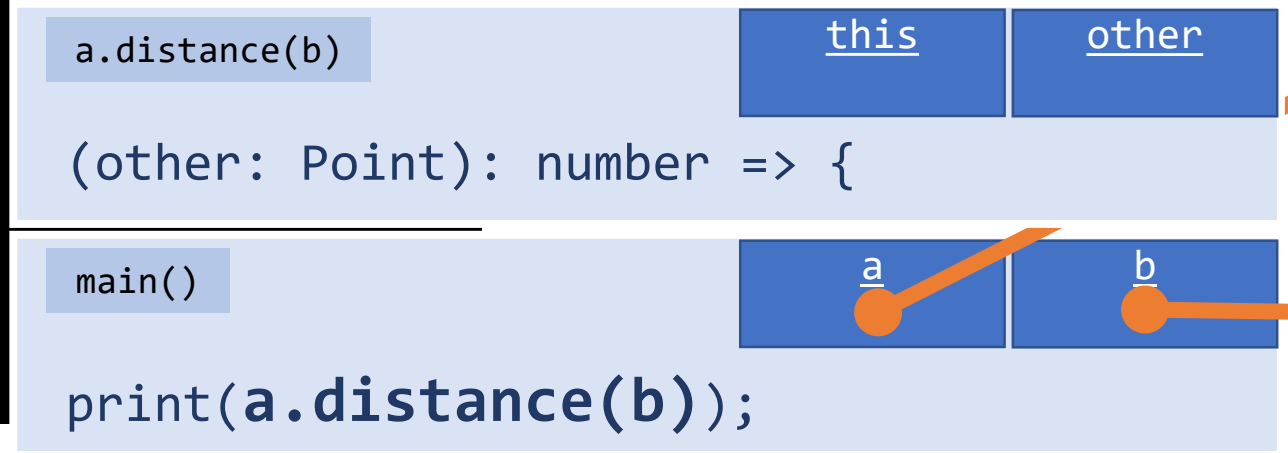


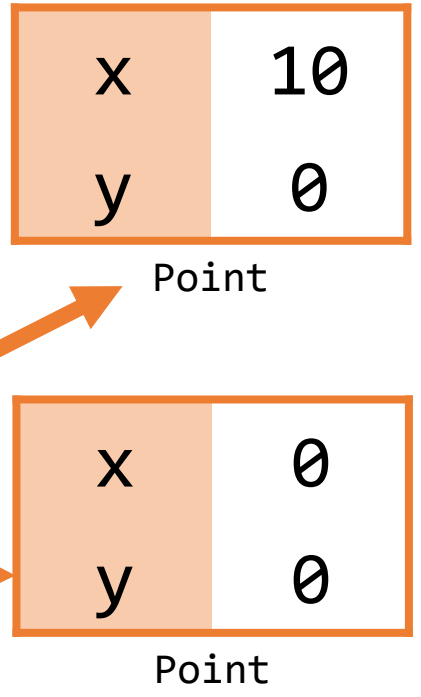


Tracing a Method Call (9 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

Then, the processor will prepare for the jump to the method by setting up a frame for the distance method's variables.



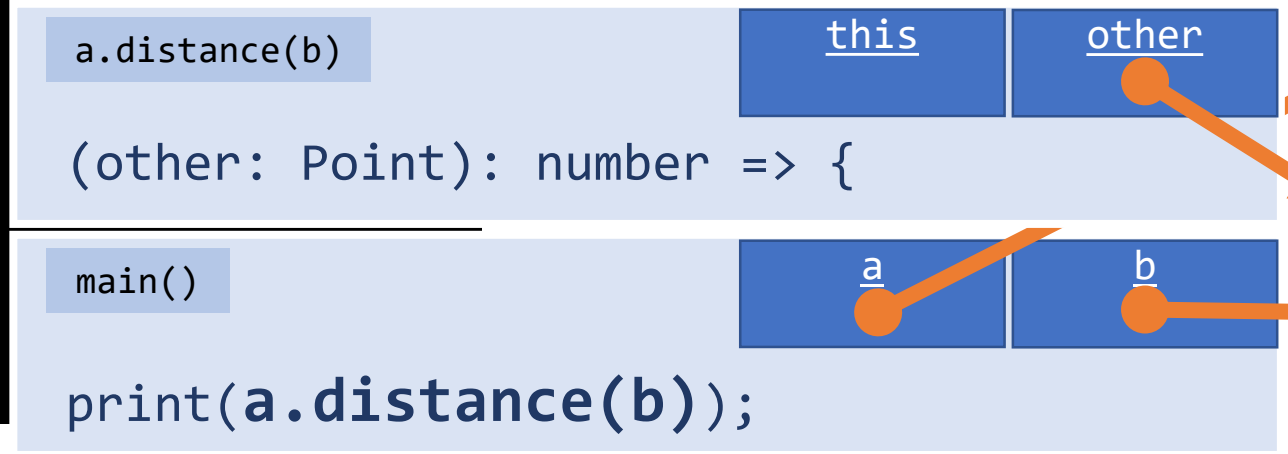


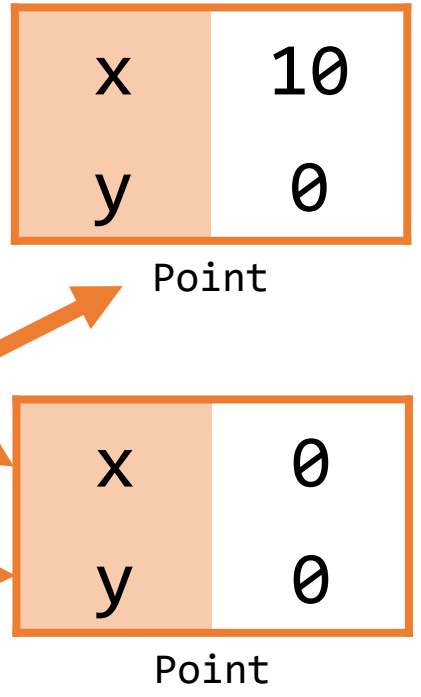
Tracing a Method Call (10 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

b was the argument passed to the **other** parameter.

What is **b**? A reference! So we copy the reference.

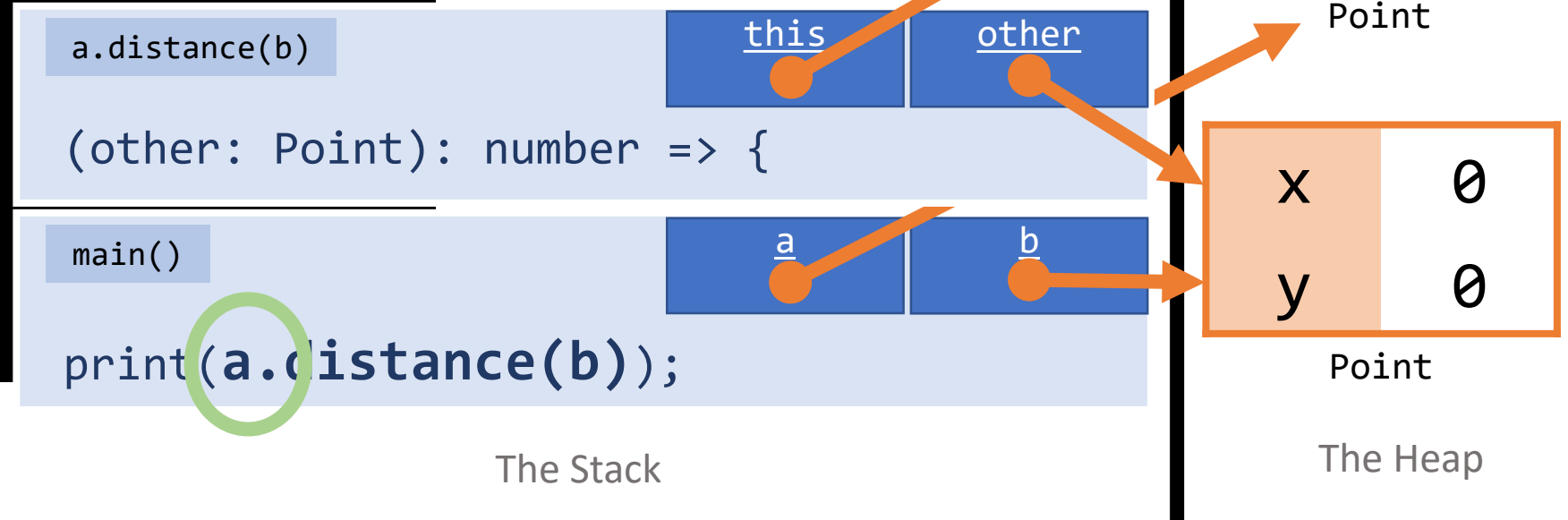




Tracing a Method Call (11 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

Method calls have one more **important** step vs. function calls… the processor will assign **this** a reference to the object the method was called on.
What object was distance called on? It was called on **a**.

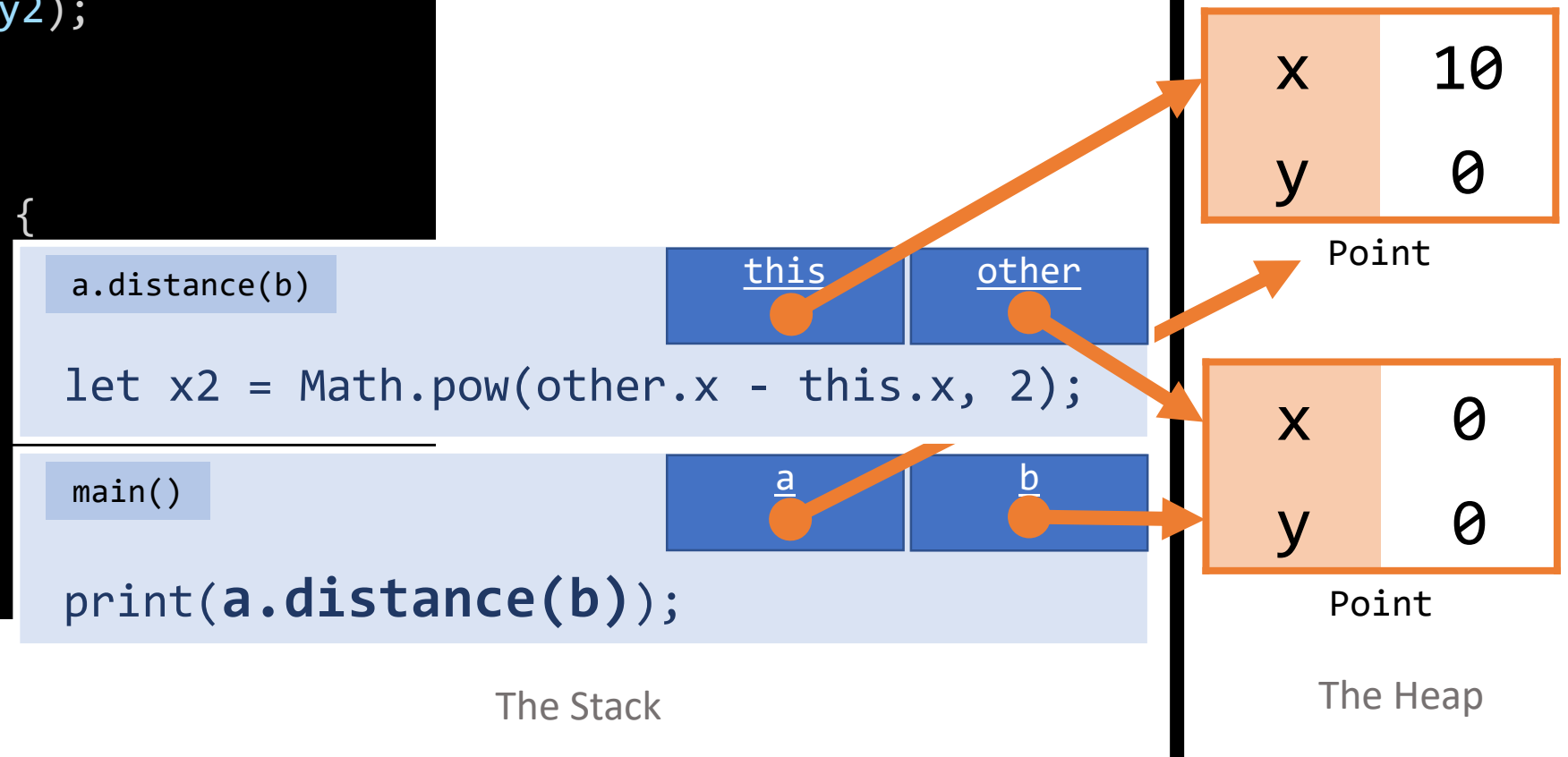


Tracing a Method Call (12 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

Now we're ready to jump into the distance method!

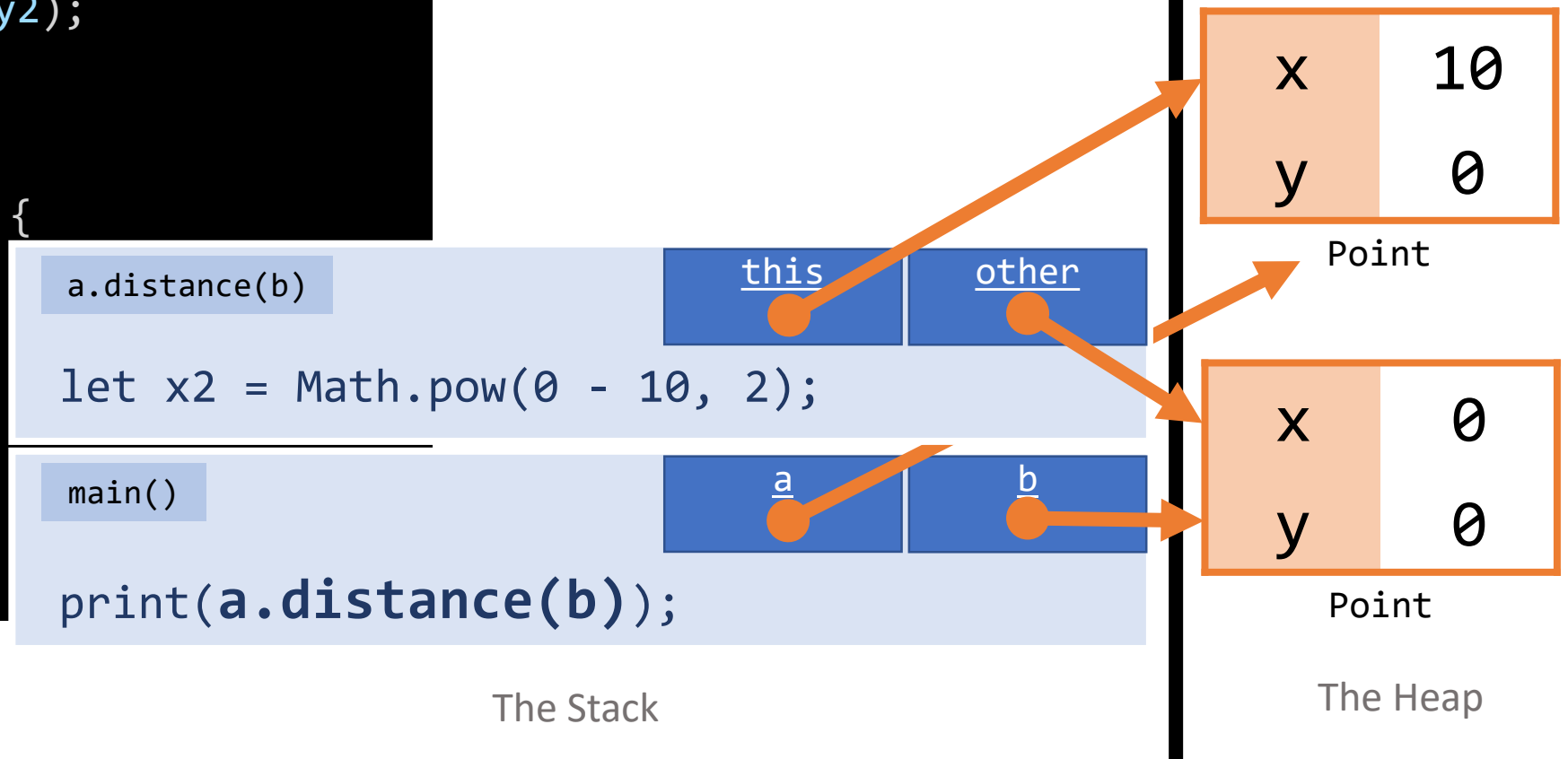
So we evaluate the first line…



Tracing a Method Call (13 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

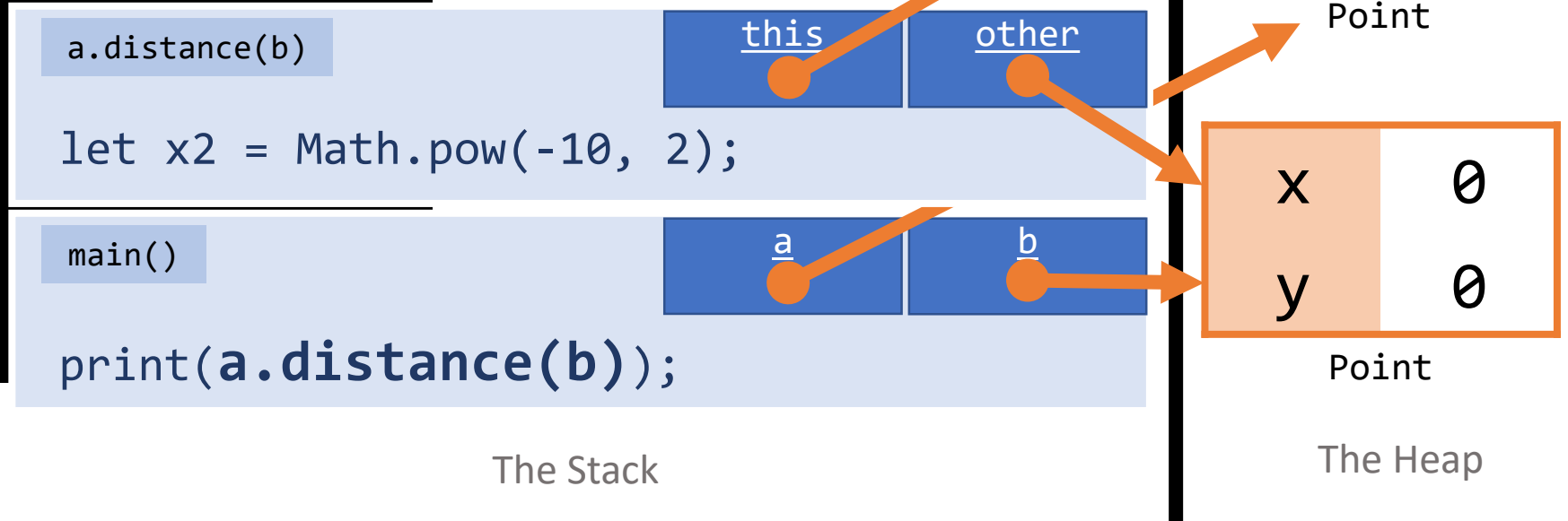
We substitute this.x with 10 and other.x with 0.



Tracing a Method Call (14 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

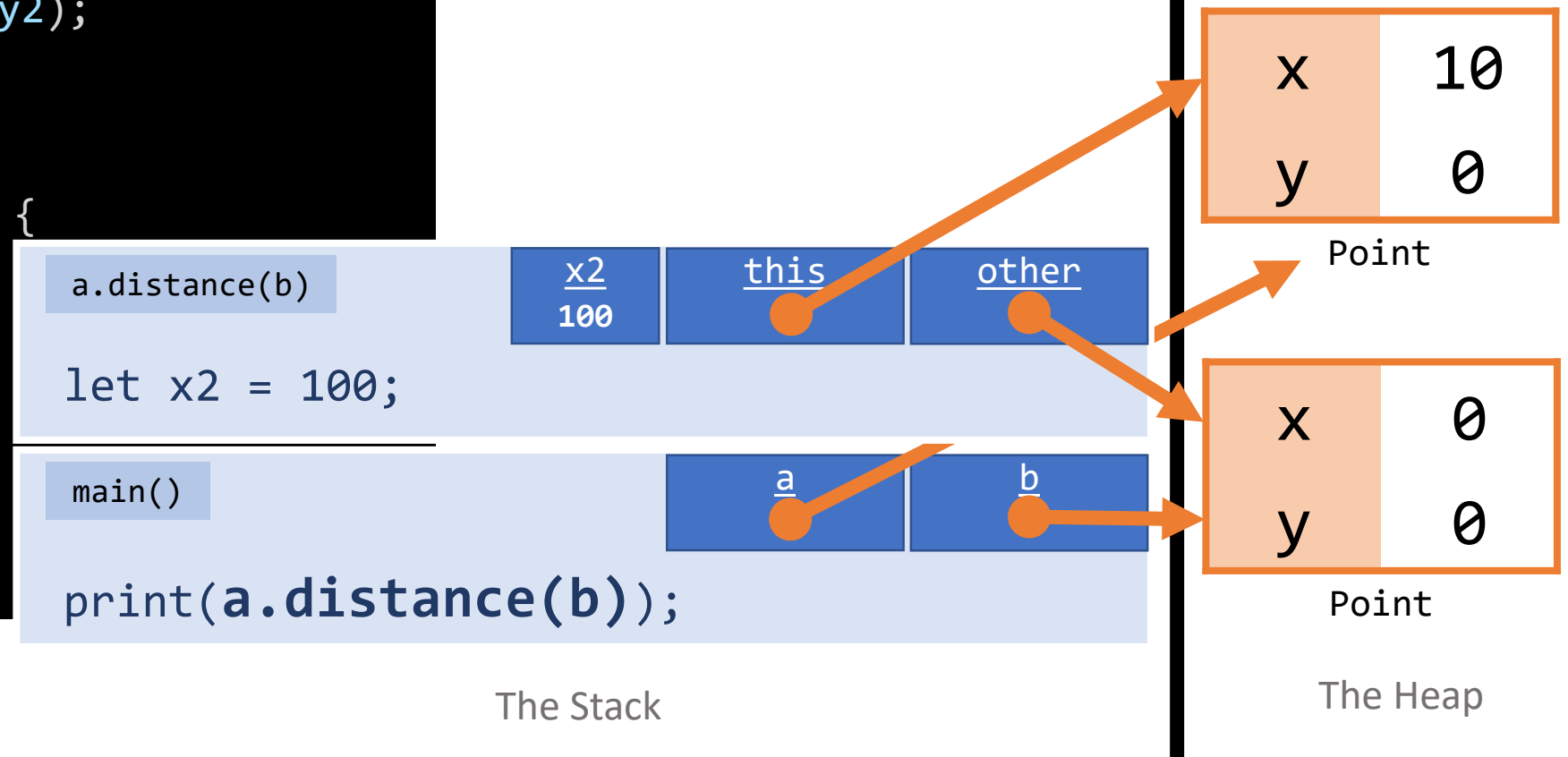
Then we call Math's pow (raise to the power of) function. We'll skip over those details and just square -10.



Tracing a Method Call (15 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

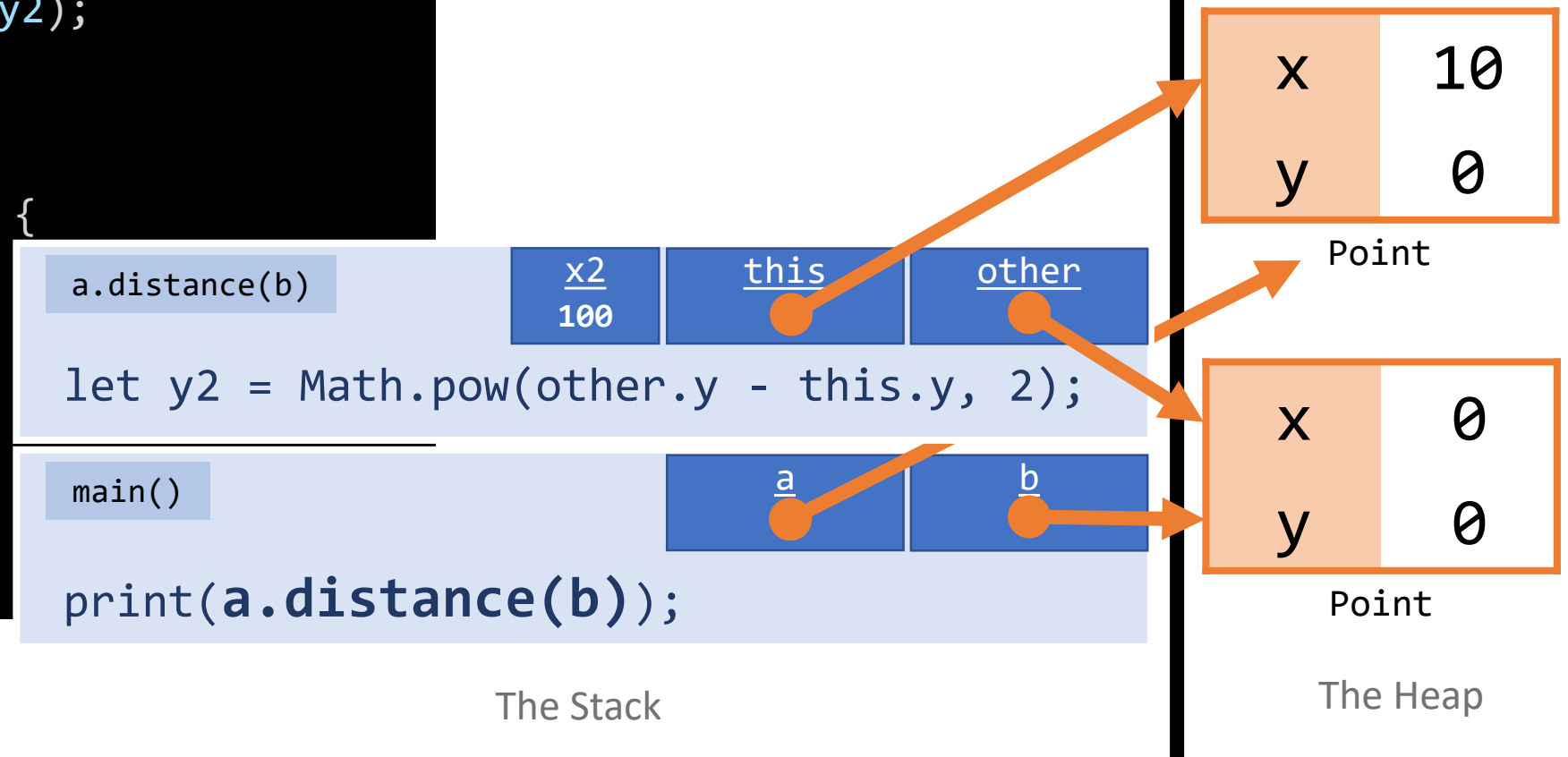
Finally, we initialize the variable x2 to 100.



Tracing a Method Call (16 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

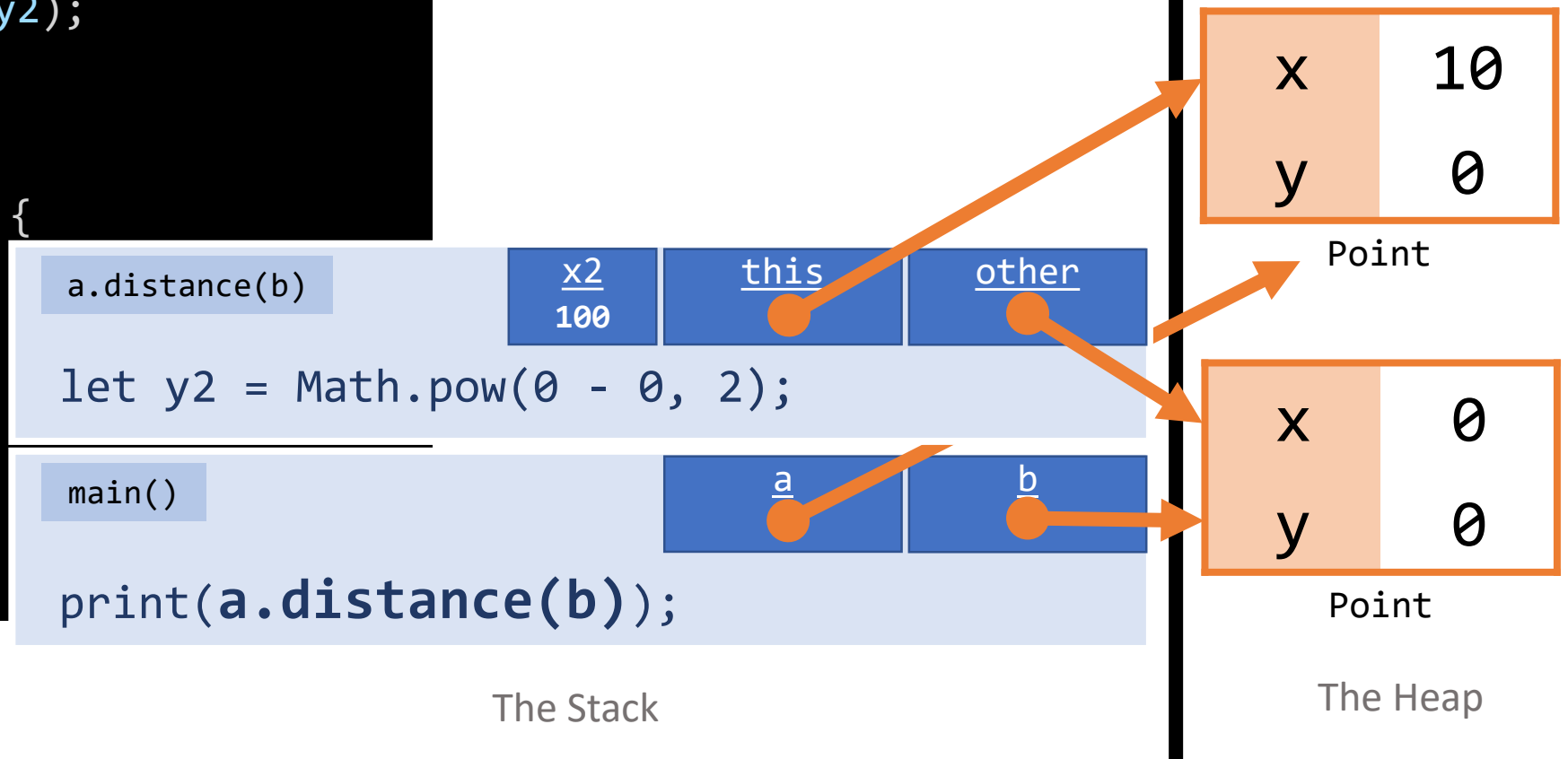
Next, we do the same process for the y properties.



Tracing a Method Call (17 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

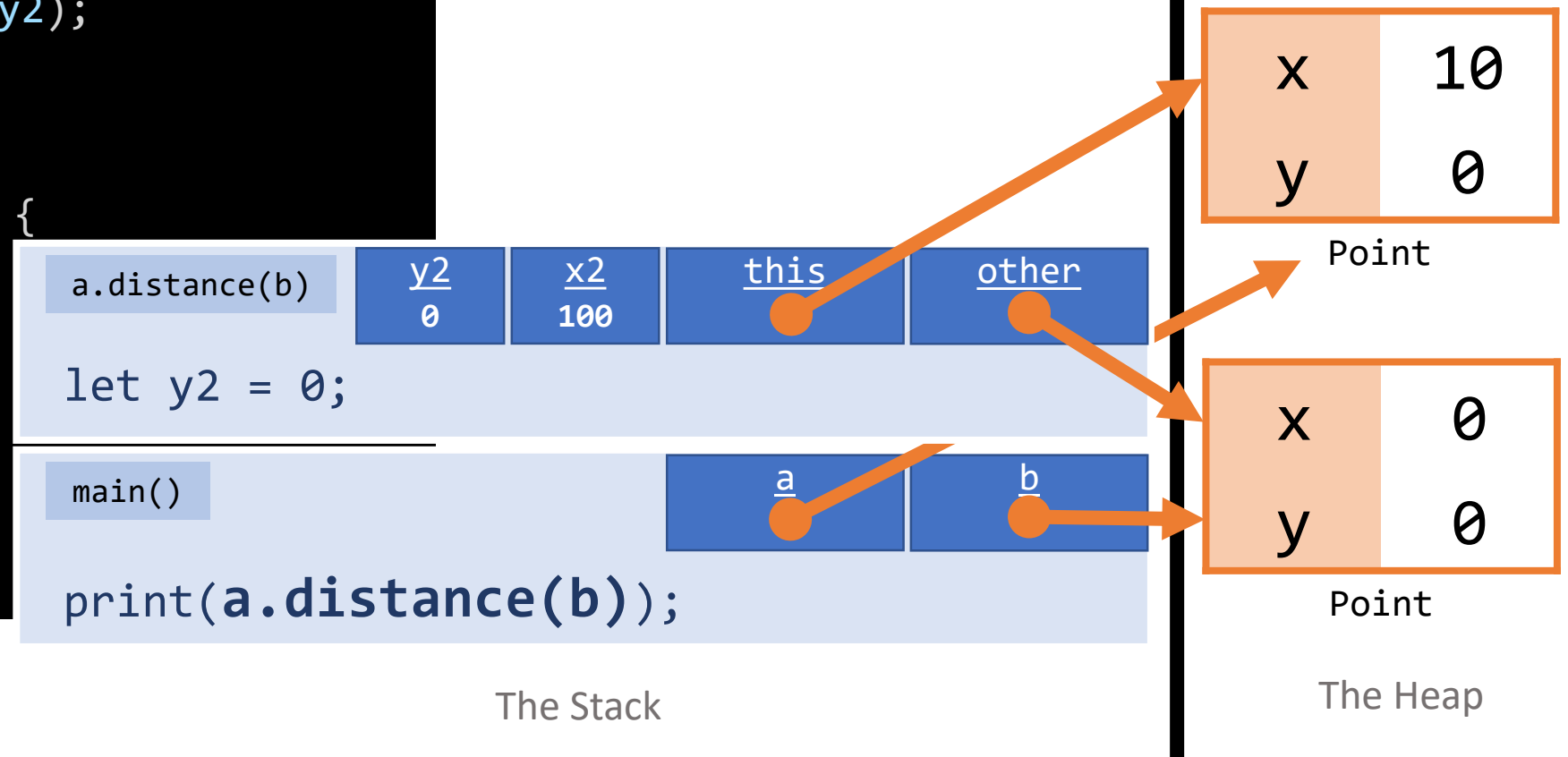
Next, we do the same process for the y properties.



Tracing a Method Call (18 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

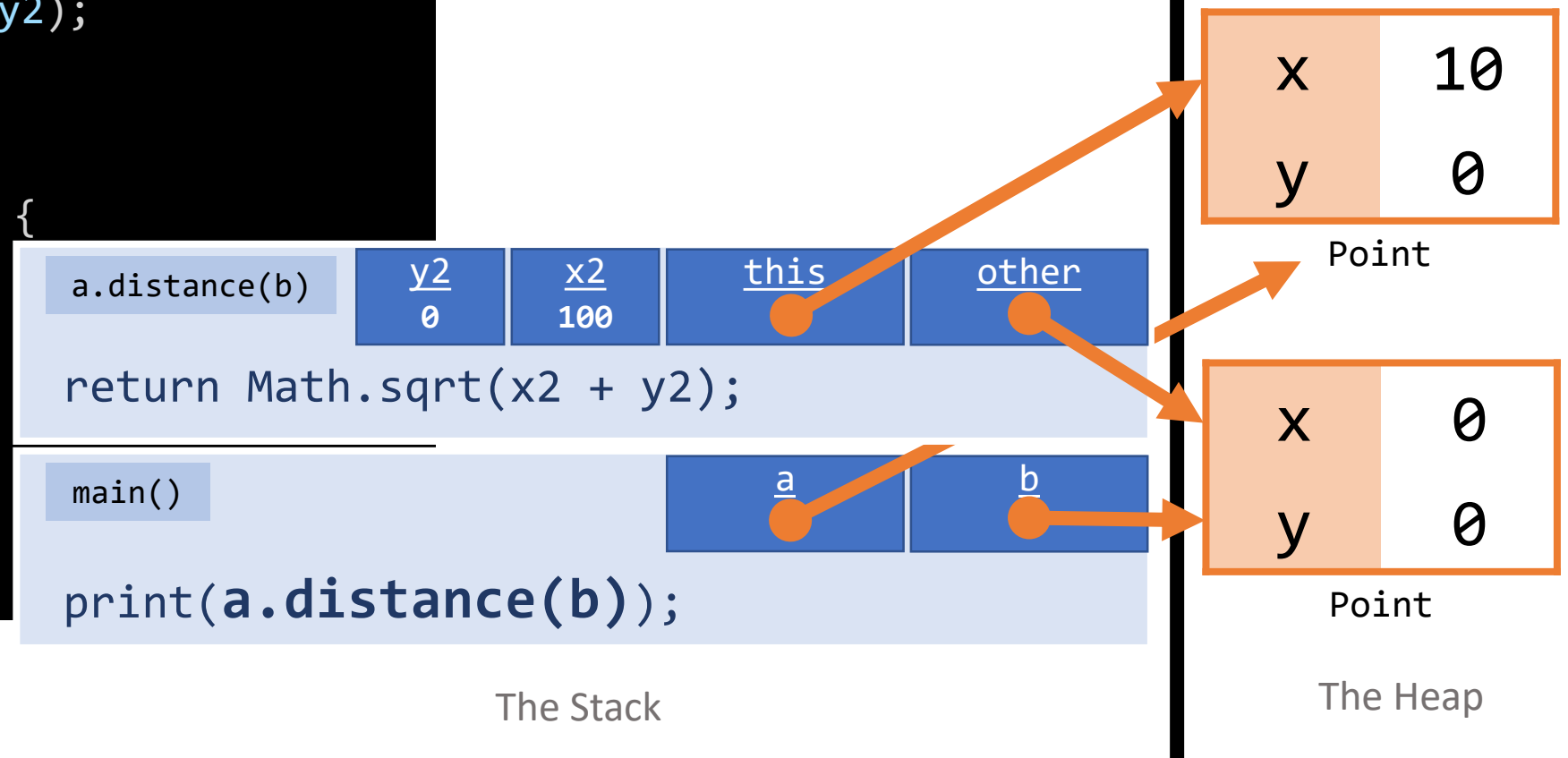
The variable y2 is initialized to 0.



Tracing a Method Call (19 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

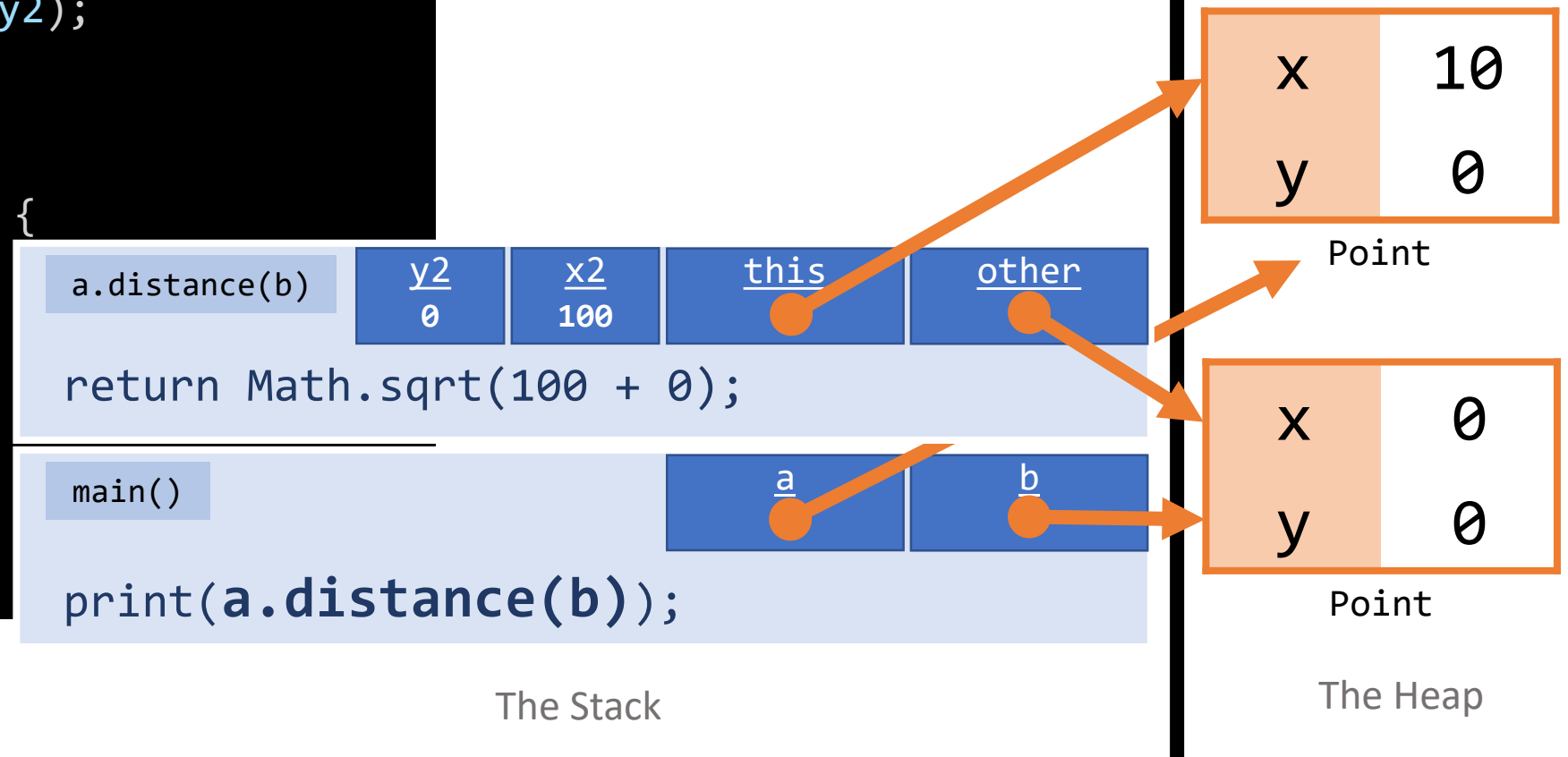
Finally we hit the method's return statement and must evaluate its expression.



Tracing a Method Call (20 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

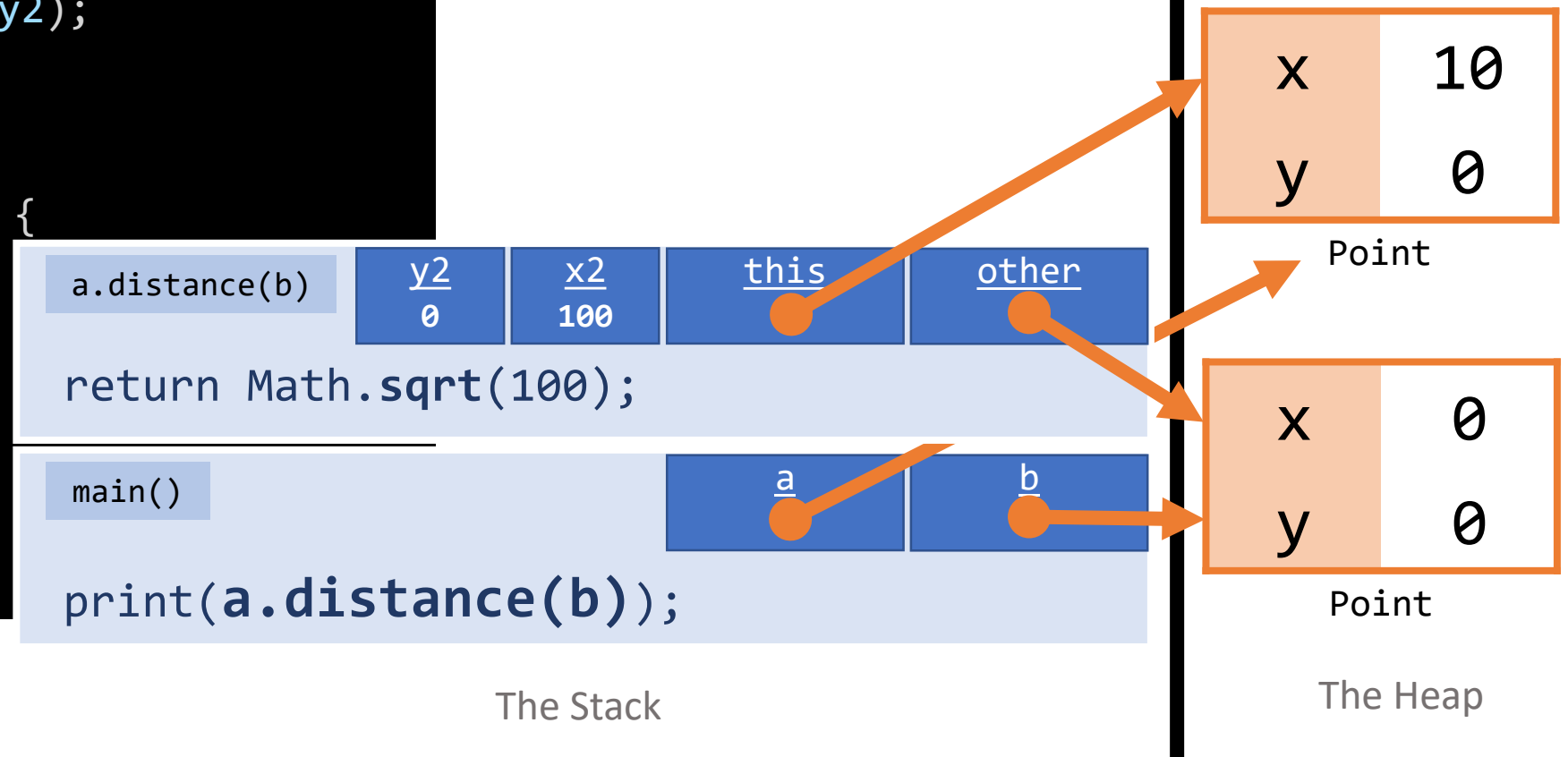
First we substitute.



Tracing a Method Call (21 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

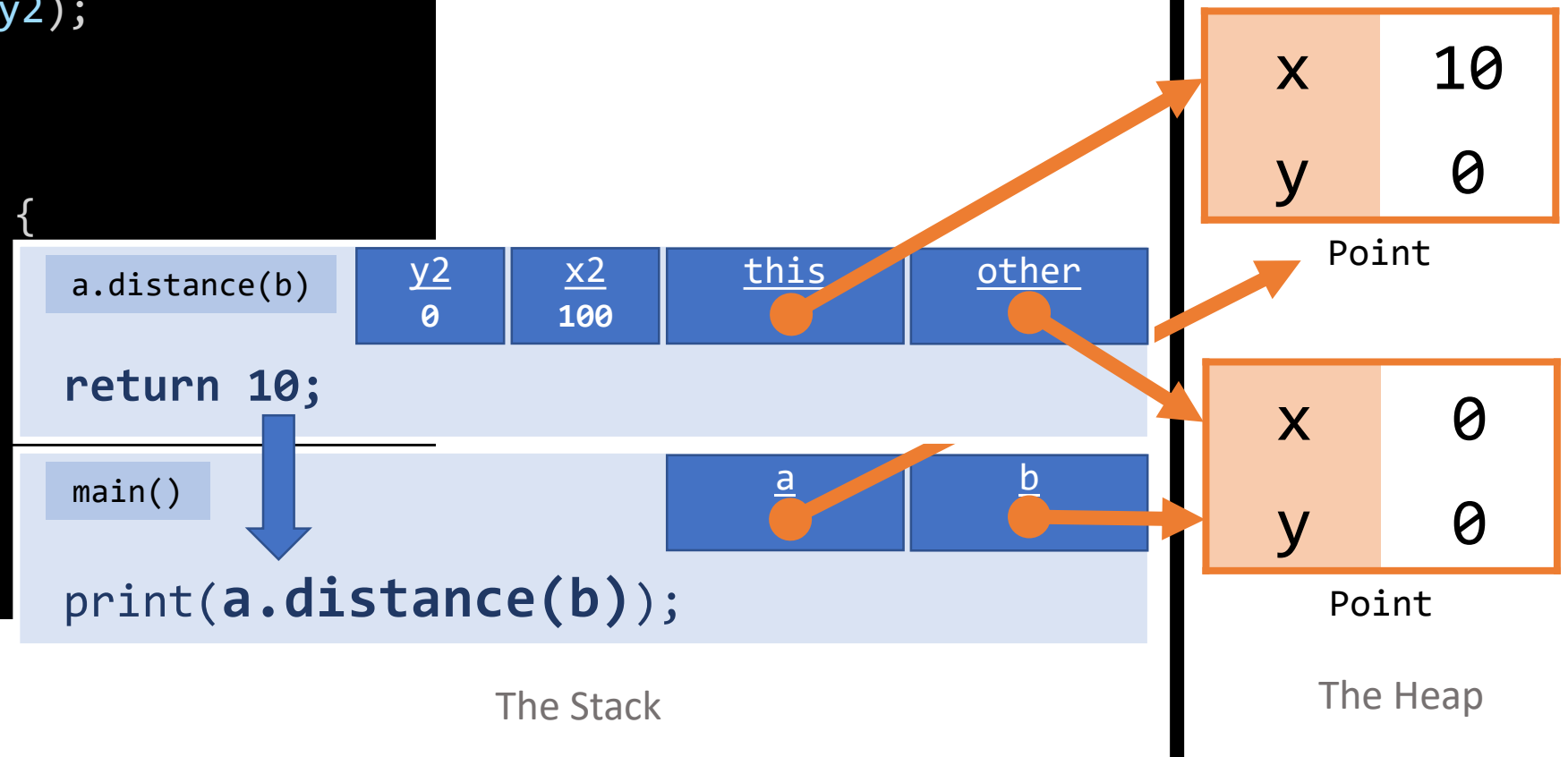
Then we call out to Math's square root function.
What is the square root of 100? 10!



Tracing a Method Call (22 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

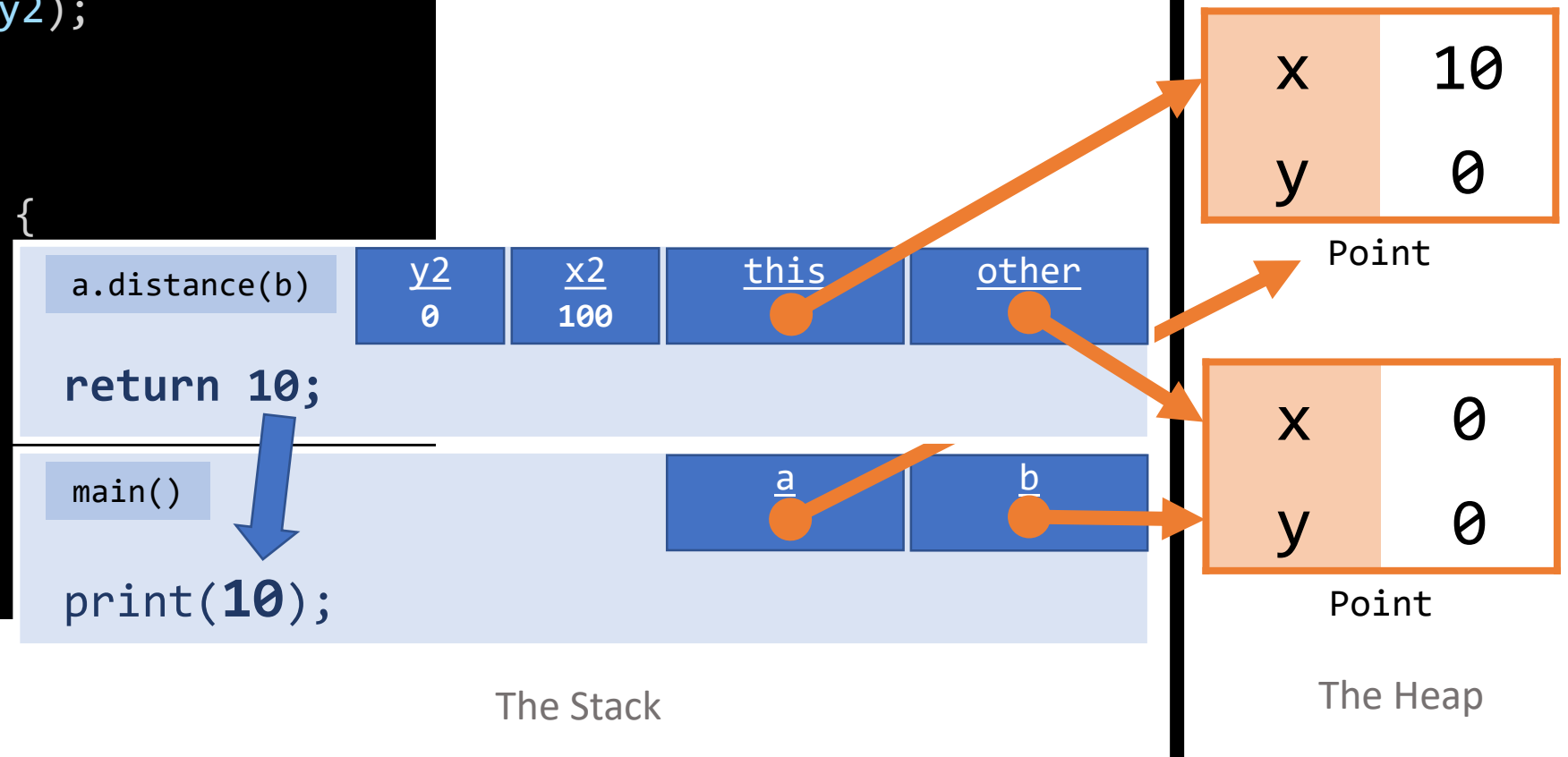
Now we're ready to return 10 back to where this method was called. This return value will substitute the entire method call expression.



Tracing a Method Call (23 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

Now we're ready to return 10 back to where this method was called. This return value will substitute the entire method call expression.

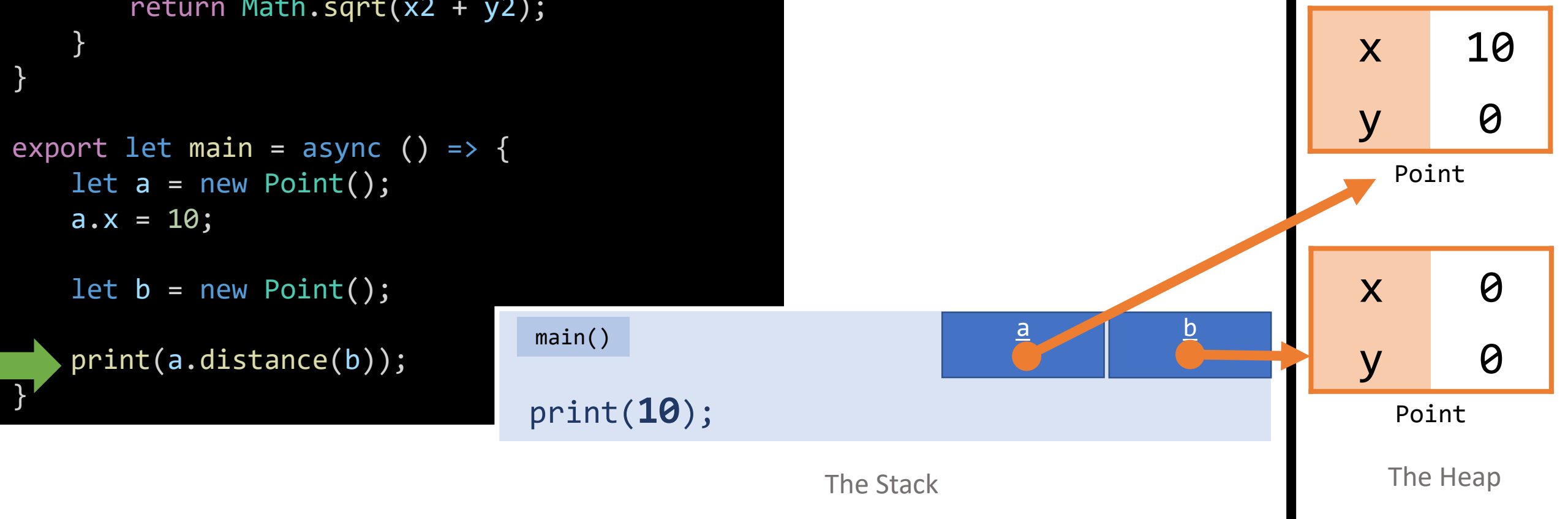


Tracing a Method Call (24 / 24)

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

    distance = (other: Point): number => {
        let x2 = Math.pow(other.x - this.x, 2);
        let y2 = Math.pow(other.y - this.y, 2);
        return Math.sqrt(x2 + y2);
    }
}

export let main = async () => {
    let a = new Point();
    a.x = 10;

    let b = new Point();

    print(a.distance(b));
}
```

Finally, we clear all of our intermediary work in the distance method and return back to the main function.

The value 10 is printed to the screen.



Method Call Steps

When a method call is encountered on an object,

1. The processor will determine what class of object it is.

2. It will then go look to confirm the class:
    1. Has the method being called defined in it.
    2. The method call's arguments are in agreement with the method's parameters.

3. Next it will initialize the method's parameters *and* the **`this`** keyword
    1. Arguments will be assigned to parameters, just like a function call
    2. The **`this`** keyword is assigned a reference to the object the method is called on

4. A bookmark is dropped at the method call and processor jumps into the method.

5. Finally, when the method completes, processor returns back to the bookmark.

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 *graphics, user interfaces, simulations, systems*
- Methods allow us to build and package functionality *into* objects.
  - 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!

3. Arrays have a built in **`filter`** method.
It works just like our **`List filter`** function.
What are **`ys`** elements after this code runs?

```
let xs = [1, 2, 3, 4];
let ys = xs.filter( (x) => !(x >= 3) );
print(ys);
```

4. What is the output?

```
let xs = [1, 2, 3, 4];

let ys = xs.filter((x) => x % 2 === 1)
           .map((x) => x * 2)
           .reduce((memo, x) => memo + x);

print(ys);
```

Array's **filter** Method

- Every array of type **T[]** has a **filter** method.

- The **filter** method has a single parameter: a **Predicate<T>** of the same type **T**

- For example:

```
let a = [-1, 0, 1, 2];
let b = a.filter((x) => x > 0);
print(b); // Prints: 1, 2
```

- Calling the **filter** method on array **a** will return a new array of type **T**.
  The filter method tests all elements in the original array *using the Predicate<T>*.
  Elements that return true will be copied to the returned array.

Array's **map** Method

- Every array of type **T[]** has a **map** method.

- The **map** method has a single parameter: a **Transform<T, U>** of the same type **T**
  - The **map** method will return an array of type **U[]**

- For example:

```
let a = ["one", "two", "three"];
let b = a.map((s) => s.length);
print(b); // Prints: 3, 3, 5
```

- Calling the **map** method on array **a** will return a new array of type **U[]**.
  The map method transforms all elements in the original array *using the Transform<T,U>*.
  All transformed elements are copied to the returned array in the same order.

Array's **reduce** Method

- Every array of type **T[]** has a **reduce** method.

- The **reduce** method has two parameters:
  1. a **Reducer<T, U>** of the same type **T**
  2. An initial **memo** ("memory" accumulator) value of type **U**

- For example:

```
let a = ["one", "two", "three"];
let b = a.reduce((memo, x) => memo + x, 0);
print(b); // Prints: 11
```

- Calling the **reduce** method on array **a** will return a single value of type **U**. Starting with the initial **memo** parameter, it will call the reducer with memo and each element in **a** successively replacing memo's value with the reducer's returned value. The final **memo** value is returned.

Chaining Method Calls (1 / 5)

- Arrays methods **filter** and **map** each returns another array… and with method call chaining, we can directly call another method on the array it returns! Let's walk through how this gets evaluated…

```
let numbers = [10, 21, 30];

let result = numbers.filter((x) => x % 2 === 0)
                    .map((x) => x * 2)
                    .reduce((memo, x) => memo + x);
```

Chaining Method Calls (2 / 5)

- At runtime, first, the processor will evaluate numbers to resolve to the array [11, 22, 31]. Then the *filter* method will be called on this array.

```
let numbers = [10, 21, 30];

let result = [10, 21, 30].filter((x) => x % 2 === 0)
                         .map((x) => x * 2)
                         .reduce((memo, x) => memo + x);
```

Chaining Method Calls (3 / 5)

- The *filter* method will return the array [10, 30], because it is applying a Predicate that returns true when a number is even, and this value replaces the method call. Next, the map method will be called on [10,30]

```
let numbers = [10, 21, 30];

let result = [10, 30].map((x) => x * 2)
                     .reduce((memo, x) => memo + x);
```

Chaining Method Calls (4 / 5)

- The return value of the map method was [20, 60] because its Transform doubled each of the elements. Finally, the processor needs to evaluate this reduce method call which is summing the elements.

```
let numbers = [10, 21, 30];

let result = [20, 60].reduce((memo, x) => memo + x);
```

Chaining Method Calls (5 / 5)

- And so a final value of 80 is assigned to result.

```
let numbers = [10, 21, 30];

let result = 80;
```

Follow-along:
Array's filter/map/reduce Methods and Chaining