Environment Diagrams Pt 2

Lecture 10

Announcements

- Worksheet 2 – Classes/Objects – Due Tonight!

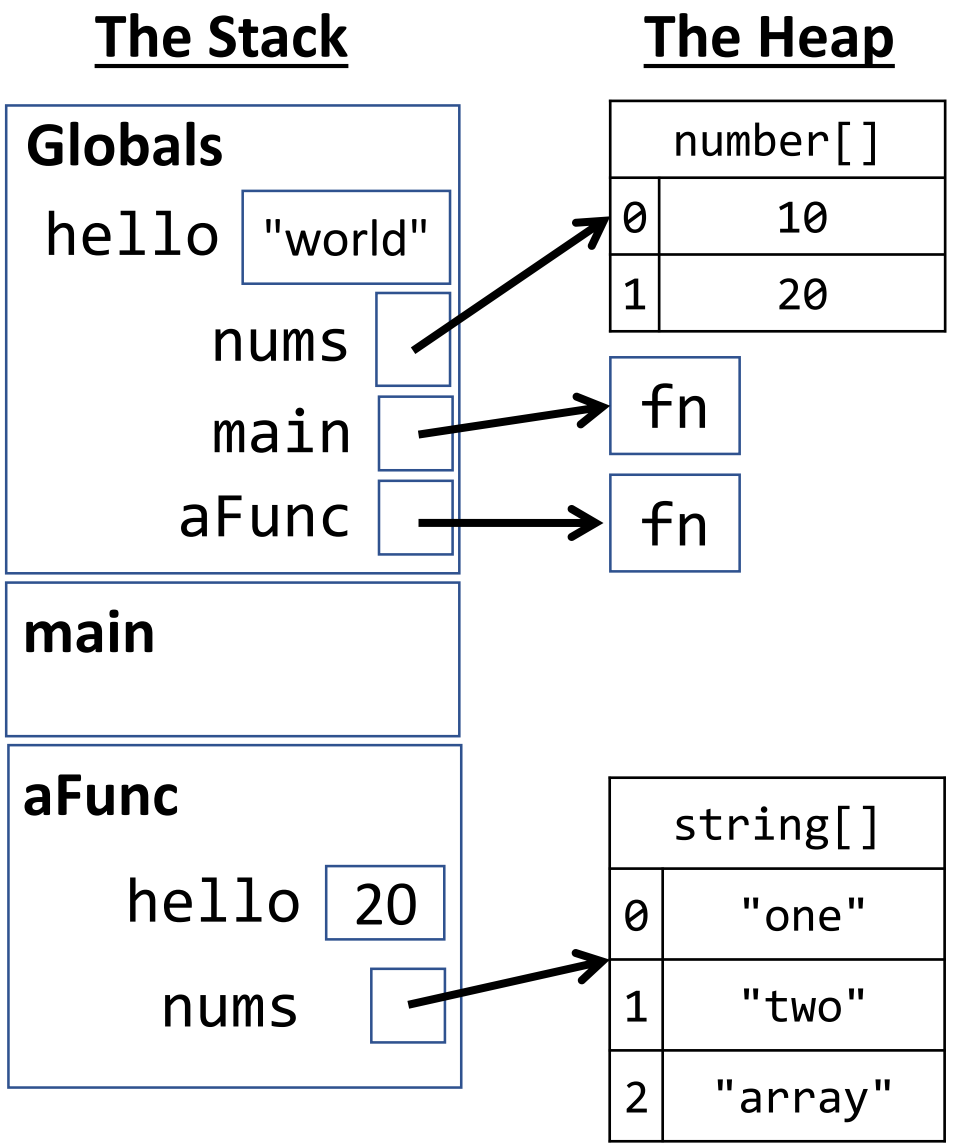
- Worksheet 3 – Environment Diagrams – Due Sunday at 11:59pm
  - We'll do one problem together in class!
  - **Work in PENCIL! Don't scratch things out in pen, it gets to be chaotic.**

- Quiz 3 – Classes/Objects and Environment Diagrams – Next Tuesday

- Problem Set 3 – Who Stole the Paper – Due *next* Wednesday
  - Turn in early for EC and enjoy Fall Break!

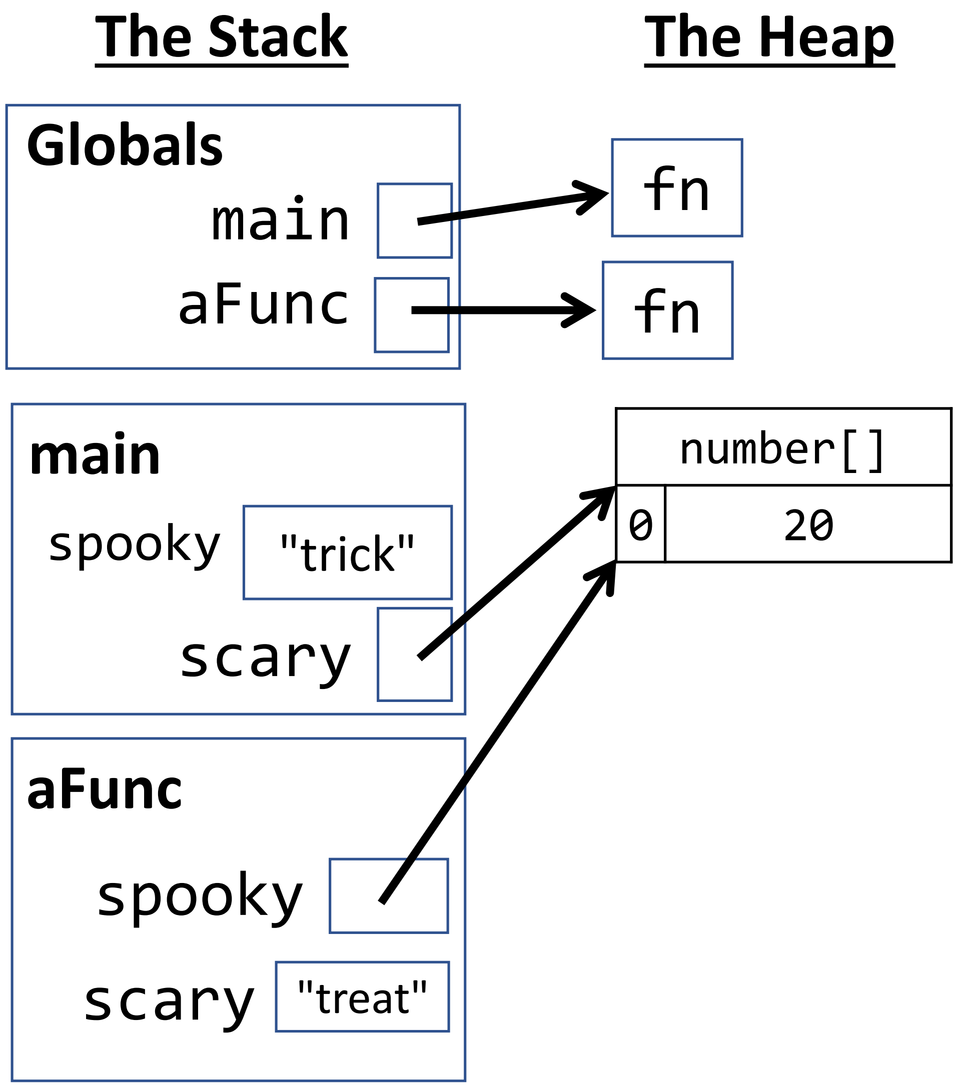
```
import { print } from "introcs";

let hello = "hello";
let nums = [10];

export let main = async () => {
    hello = "world";
    nums[1] = 20;
    aFunc();
};

let aFunc = (): void => {
    let hello = 20;
    let nums = ["one", "two"];
    print("Break!");
};

main();
```

Challenge 0: Given the following code and environment diagram, respond to the questions on PollEv.



Challenge 1: Given the following code and environment diagram, respond to the questions on PollEv.

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

export let main = async () => {
    let spooky = "trick";
    let scary = [10];
    aFunc(scary, spooky);
};

let aFunc = (spooky: number[], scary: string):void=>{
    spooky[0] = 20;
    scary = "treat";
    print("Break!");
};

main();
```



Name Resolution

How does the program interpreter know what value is bound to a name, technically called an *identifier*, from any frame in a program?

1. It looks for the name in the specified frame.

2. If it isn't there, it looks in the globals frame*.

When the program interpreter is interpreting a specific line of code, it will first look in the *current function call's* frame on the call stack.

* When functions are defined *within* other functions this rule becomes more nuanced. For now, though, this simplified rule serves us well.

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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 0;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

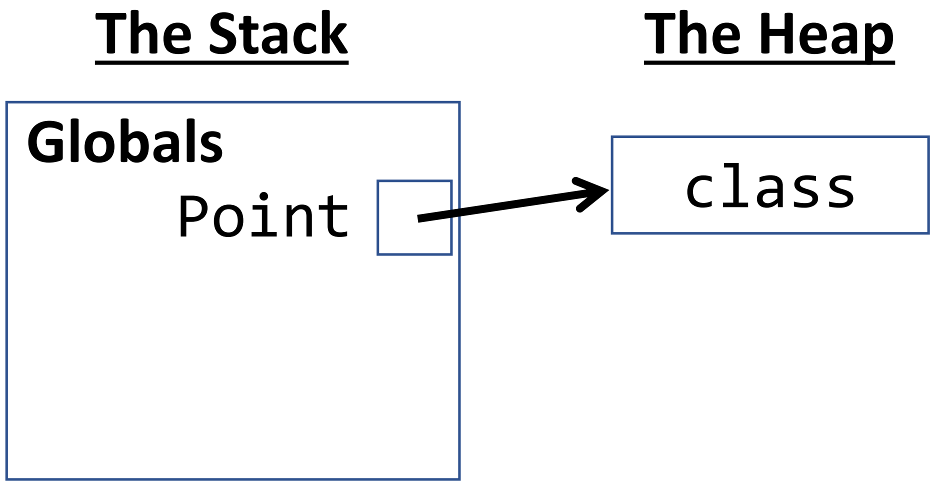
Challenge #3 - What is printed in this example?

Case: Objects

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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Class Definition

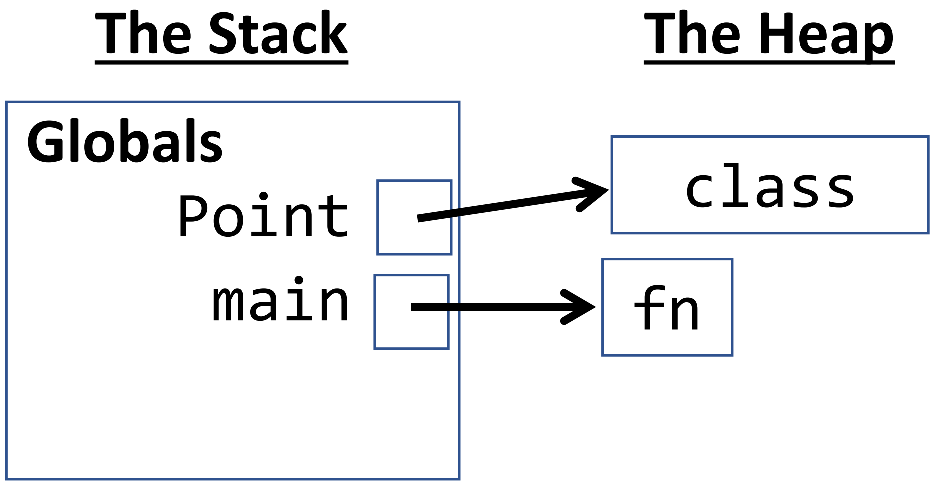
When a class definition is encountered, add its name to the current frame on the stack, draw an arrow to a "class box" on the Heap. This is where the definition of the class is stored, though we will not try to represent it more specifically.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Variable Declaration - Function Definition

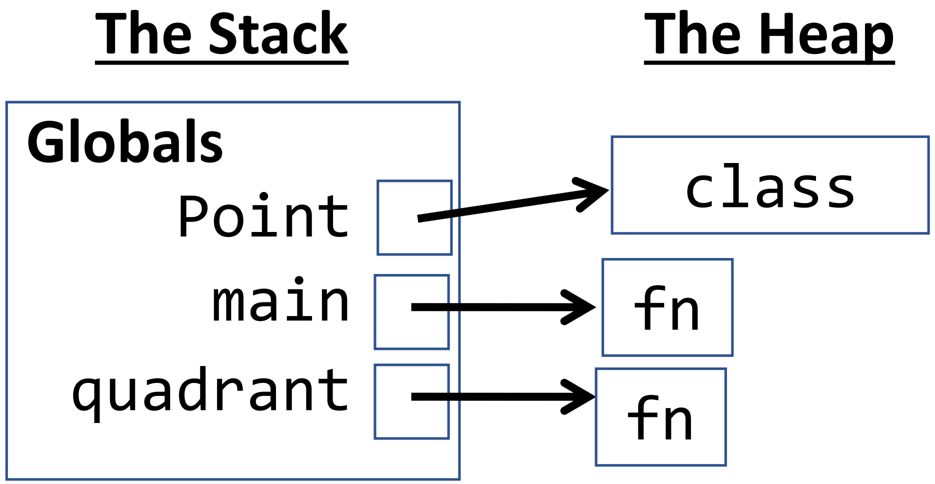
When a variable defining a function is evaluated, add its name to the current frame on the stack, draw an arrow to a "function box" on the Heap. This is where the definition of the function is stored.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Variable Declaration - Function Definition

When a variable defining a function is evaluated, add its name to the current frame on the stack, draw an arrow to a "function box" on the Heap. This is where the definition of the function is stored.

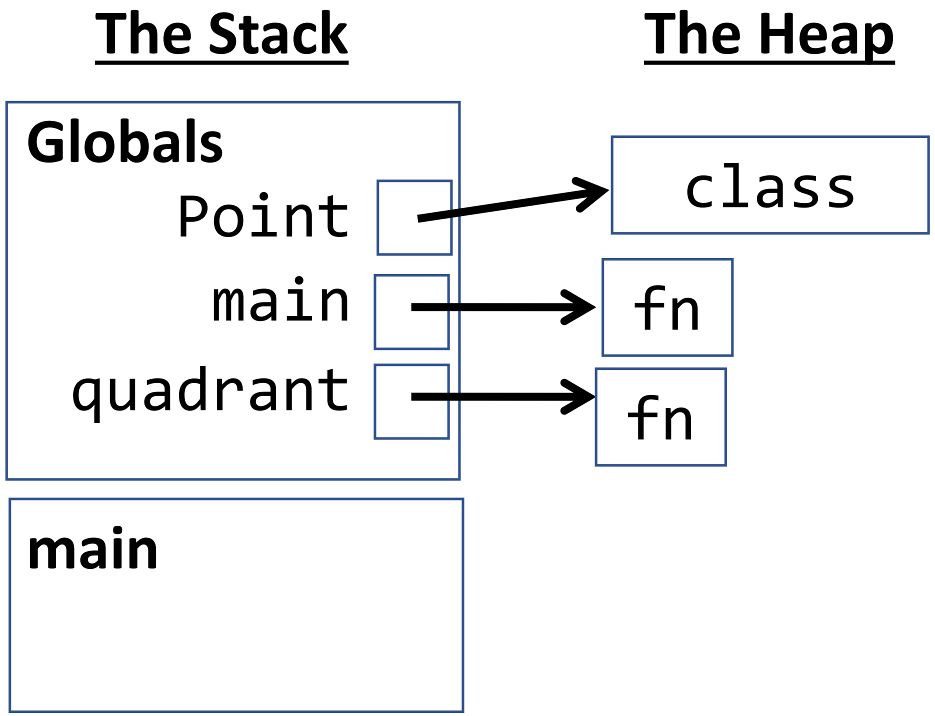


```
class Point {
    x: number = 0;
    y: number = 0;
}

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Function Call

When a function call is evaluated, evaluate its arguments first. In this case, there are no arguments.

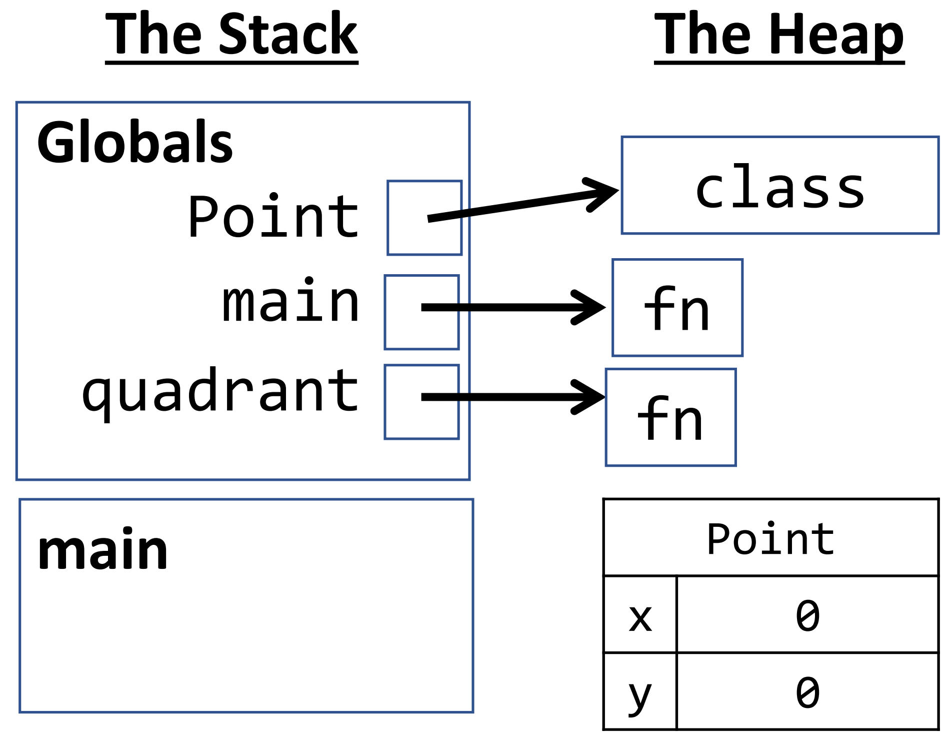
Then, add a frame onto the (function) call stack with the function's name.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Object Construction

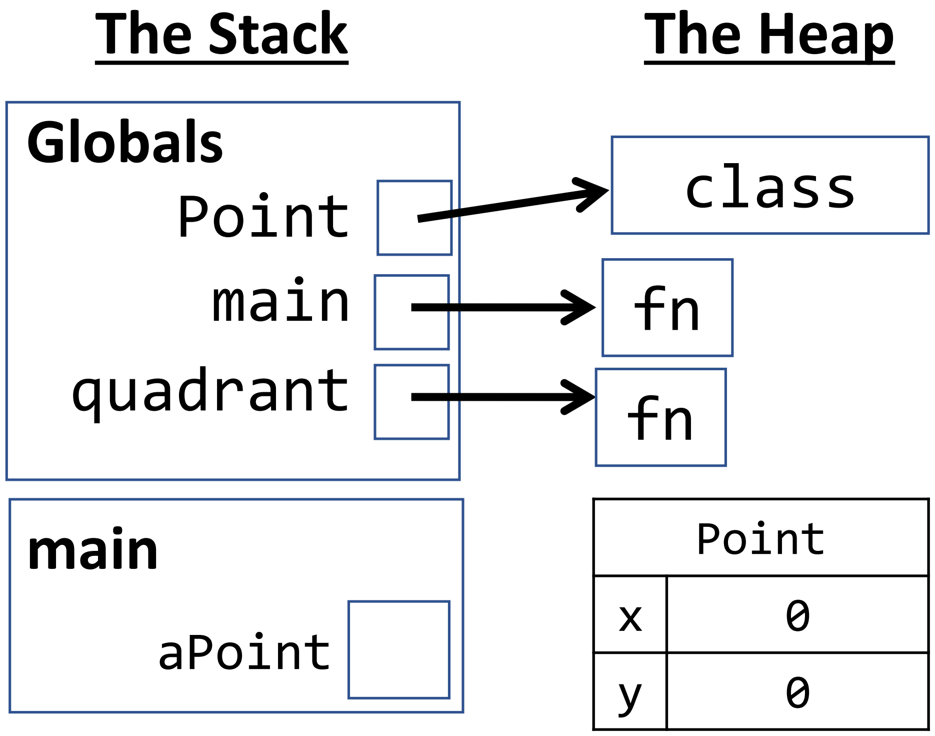
When a new object is constructed, which happens whenever you see the new keyword followed by a class name, establish it on the Heap. Assign its properties their default values.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Variable Declaration

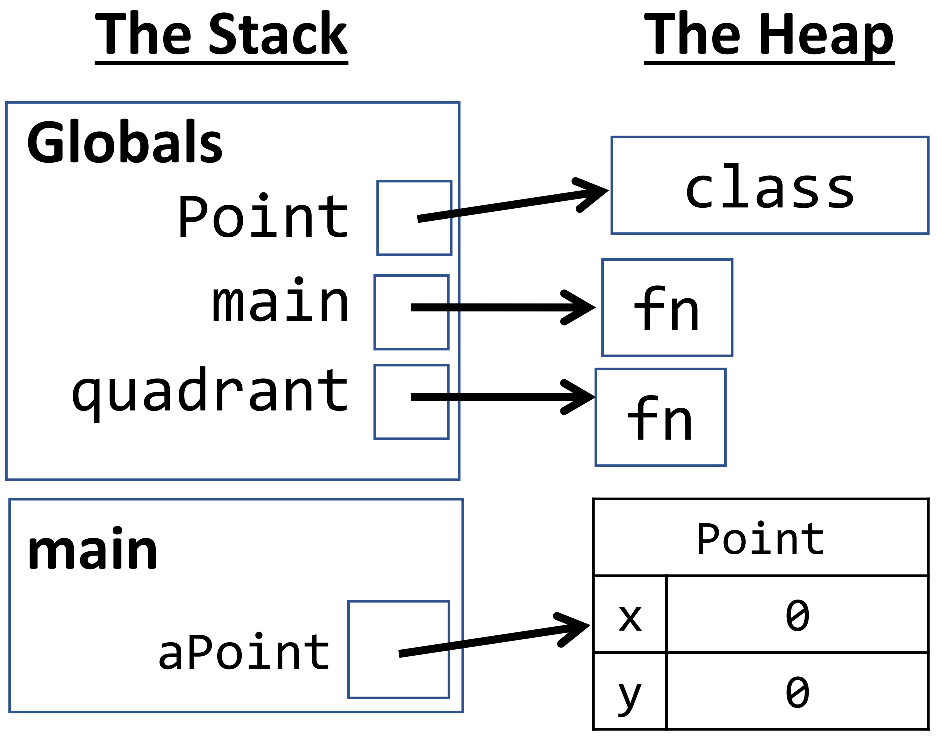
When a variable is declared, add its name and some space for its value to the current frame on the call stack.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Variable Assignment - Object

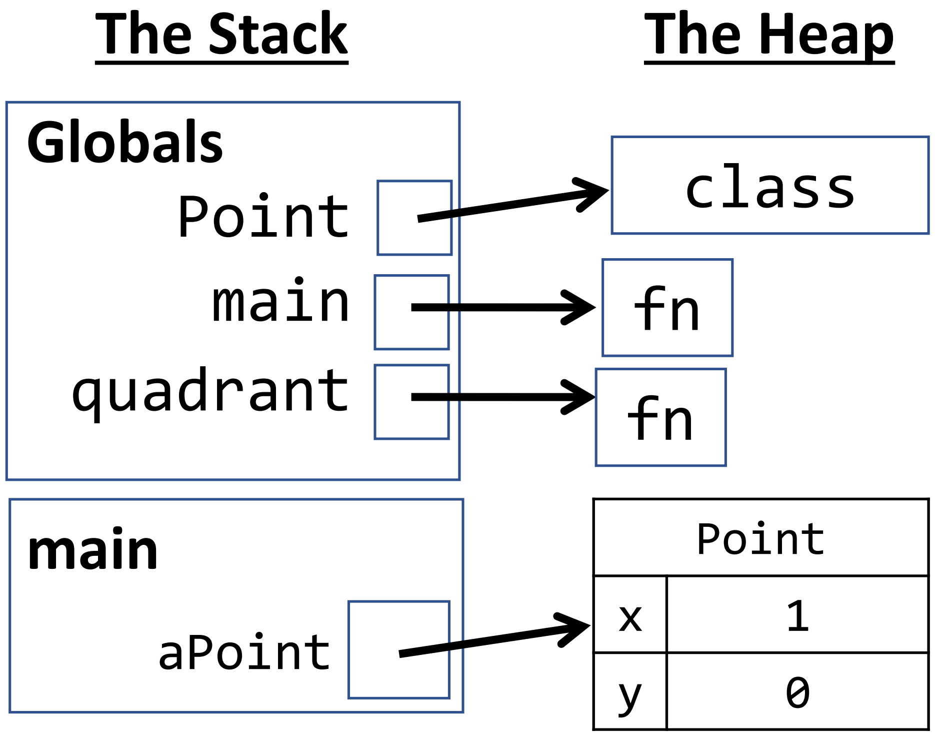
When an object is assigned to a variable, the variable's value is a reference (or **pointer** arrow) to the object. We would say, "aPoint now refers to a Point object on the Heap."



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Property Assignment

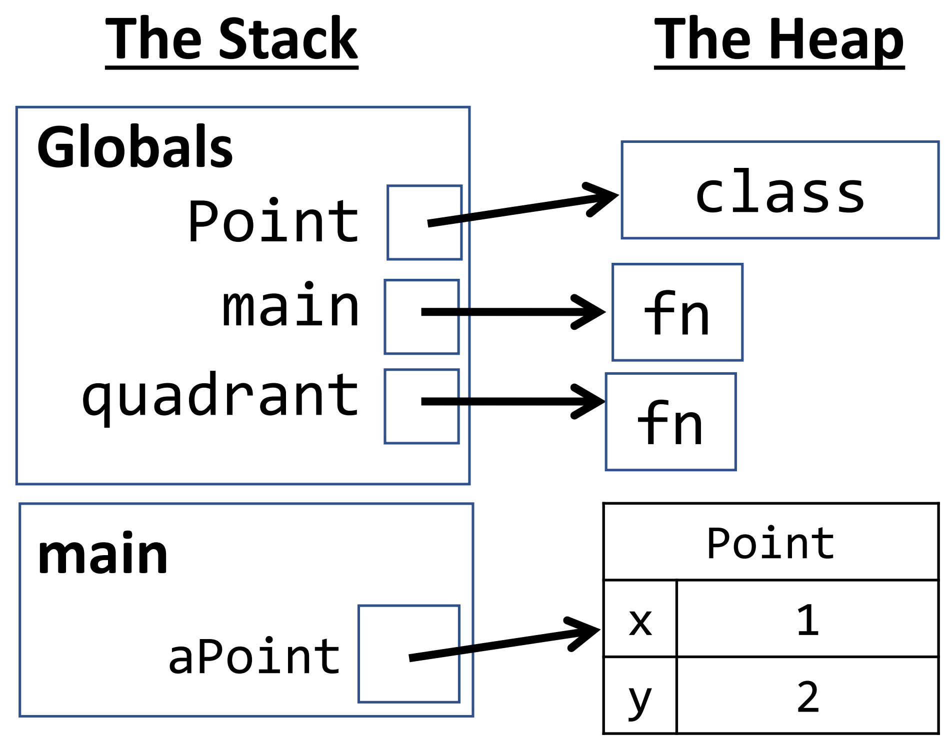
Use name resolution to identify where **aPoint** is. Then, the dot operator follows the arrow and looks up the property x. Once it is resolved where **aPoint.x** is, update the value.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Property Assignment

Use name resolution to identify where **aPoint** is. Then, the dot operator follows the arrow and looks up the property x. Once it is resolved where **aPoint.y** is, update the value.

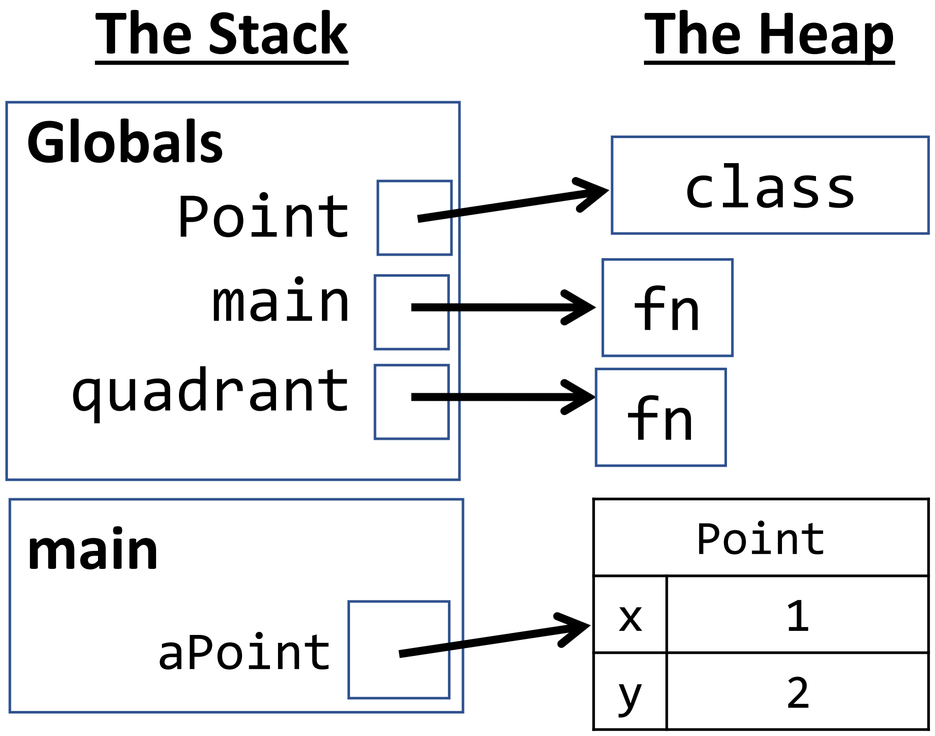


```
class Point {
    x: number = 0;
    y: number = 0;
}

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Variable Access

Use name resolution to identify what **aPoint's** value is.

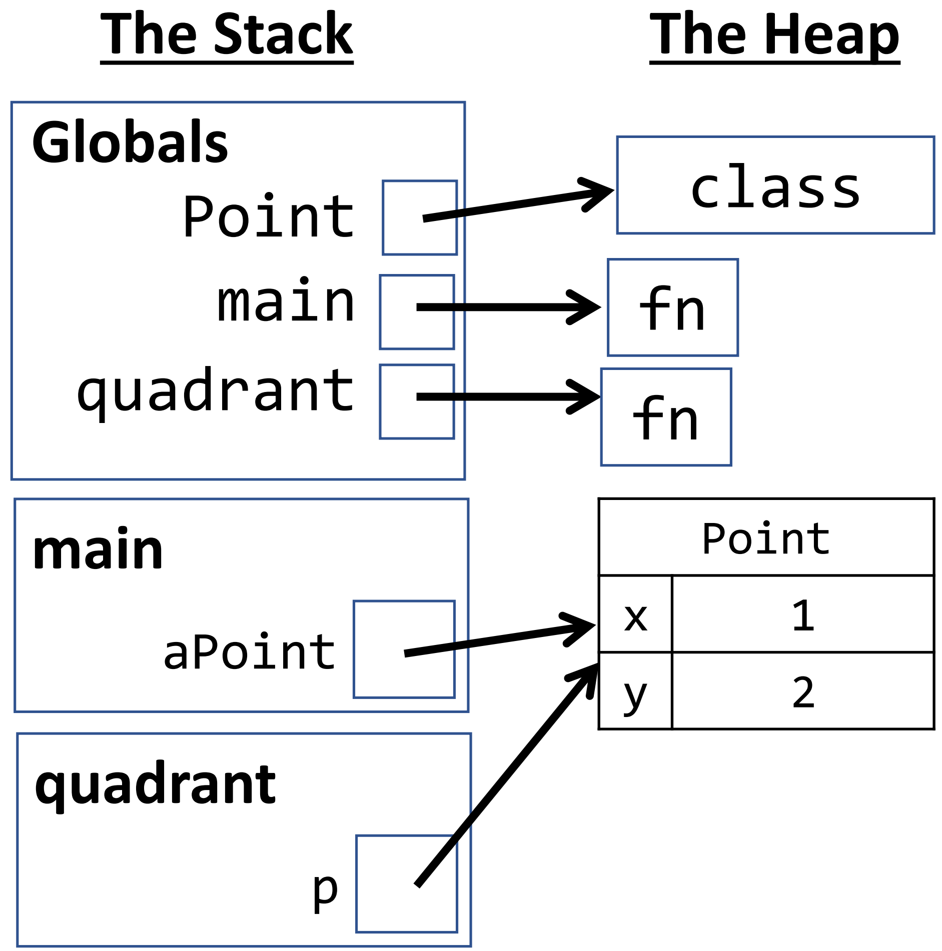
Notice it's value is a pointer to an object on the heap! This pointer will be assigned to quadrant's **p** parameter.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Function Call

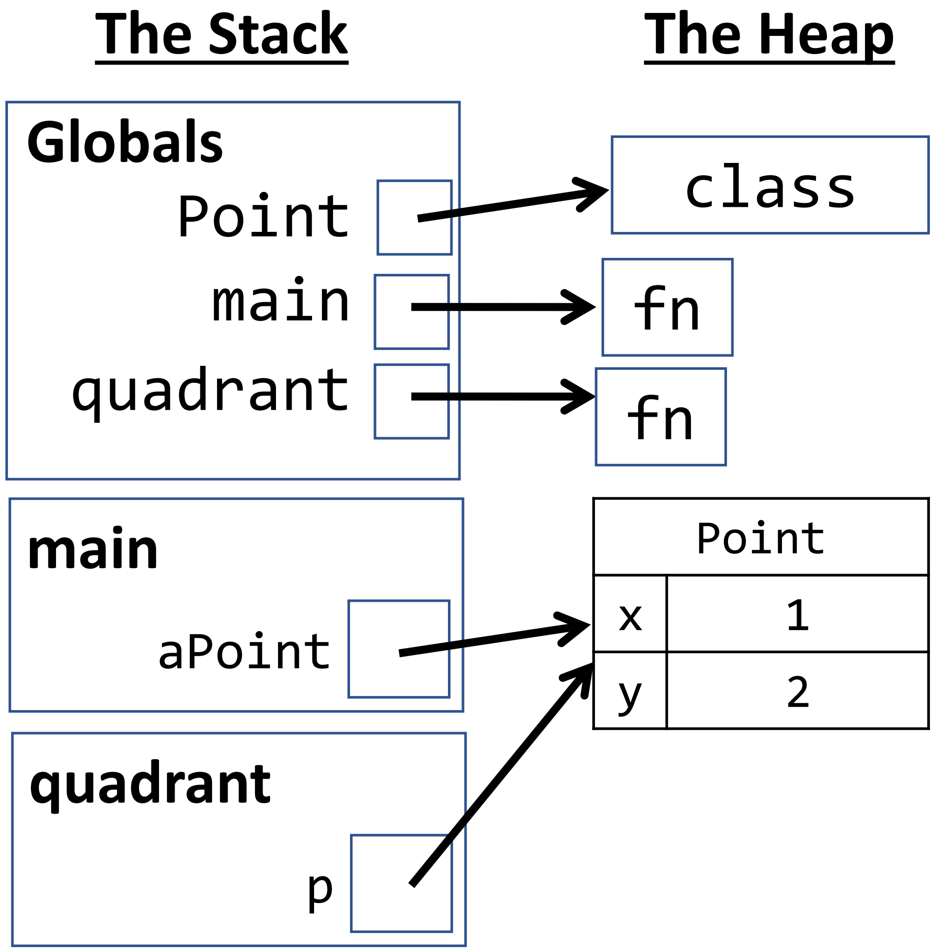
Is quadrant a function we know? Yes, it is defined in globals. So we setup a frame on the call stack for it. It has a single parameter. Before jumping *to* the function body, parameter names are added to the frame and corresponding argument values are bound.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Object Parameters are Assigned References

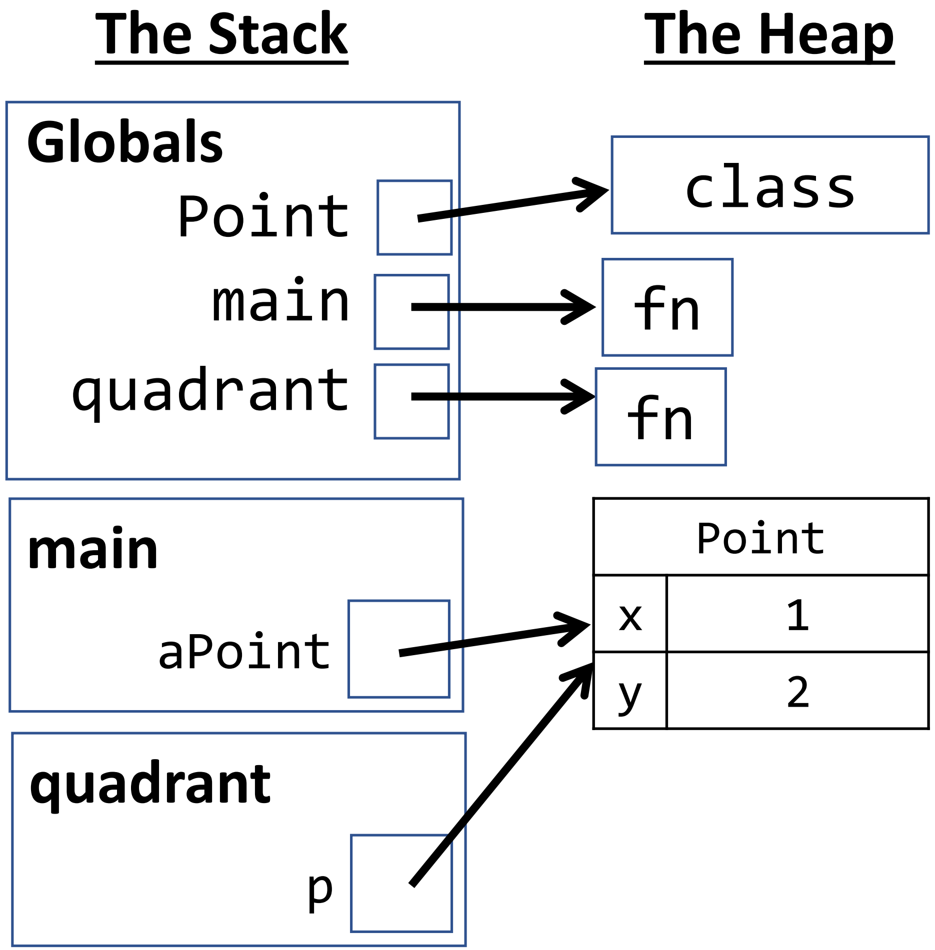
It is worth emphasizing an object parameter, like p, is assigned a reference to whatever value it is passed. This is true of array-type parameters as well. Contrast this with primitive value parameters whose values are *copied.*



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Property Access

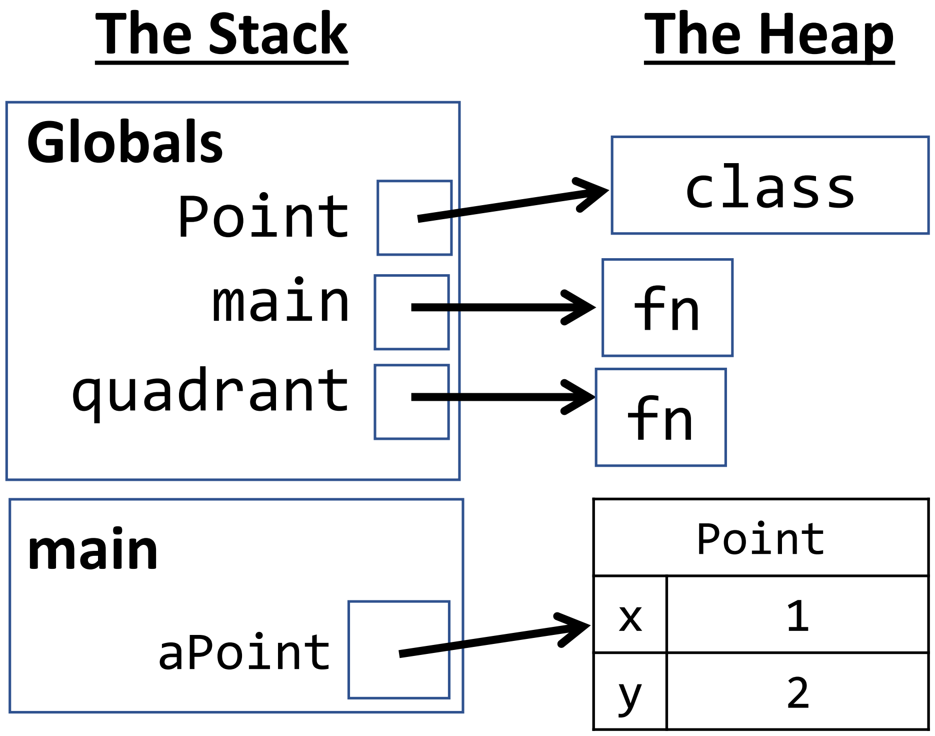
Use name resolution and follow references via dot operator to identify the values of **`p.x`** and **`p.y`**. Both are greater than zero, so this boolean expression evaluates to **`true`**.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Return Statement

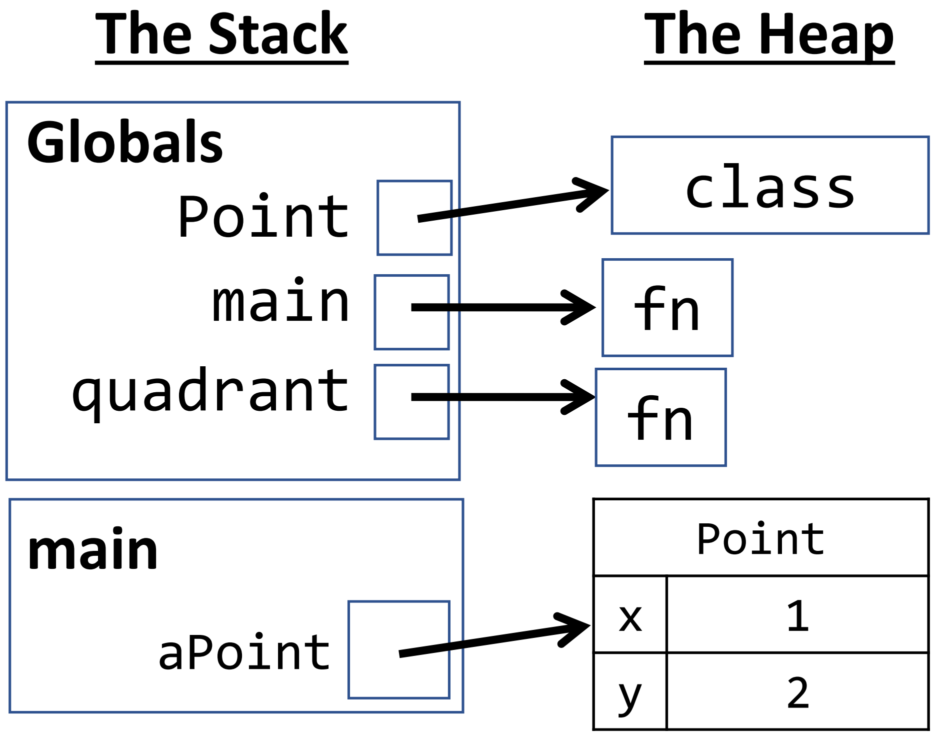
When a return statement is encountered, the return value is substituted for where the function call originated. The current frame on the stack is cleared.



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

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Calls to Library Functions

We will not attempt to evaluate calls to library functions in our environment diagrams. Technically, the same process happens: a new frame is added, arguments are passed and assigned to parameters, and the interpreter jumps into the function called.

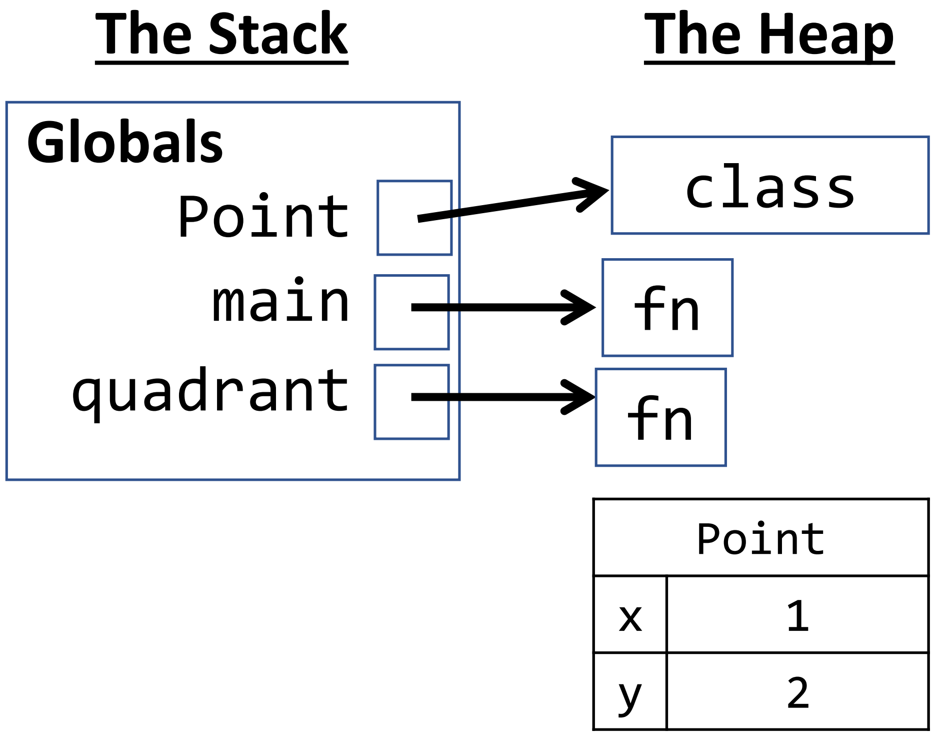


```
class Point {
    x: number = 0;
    y: number = 0;
}

export let main = async () => {
    let aPoint = new Point();
    aPoint.x = 1;
    aPoint.y = 2;
    print(quadrant(aPoint));
};

let quadrant = (p: Point): number => {
    if (p.x > 0 && p.y > 0) {
        return 1;
    } else if (p.x < 0 && p.y > 0) {
        return 2;
    } else if (p.x < 0 && p.y < 0) {
        return 3;
    } else if (p.x > 0 && p.y < 0) {
        return 4;
    } else {
        return 0;
    }
};

main();
```

Void Function Returns

When a void function ends, which we consider main to be a special void function, its frame is cleared from the stack.

Execution will return to where the function call originated.



Problem #2 on the Worksheet

- We'll spend 5 minutes working through it individually.
- Then, we'll pair up with neighbors.
- Finally, we'll work and talk through it together.

Challenge Question #3: What is printed?

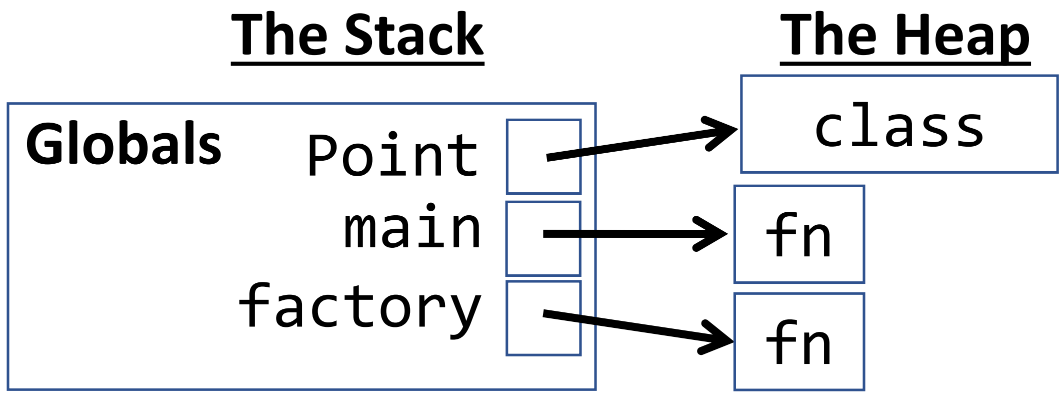
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```

Case: Array of Objects

Let's Fast-Forward to Here

Assume the globals frame is now populated and we've reached the call to **`main`**.

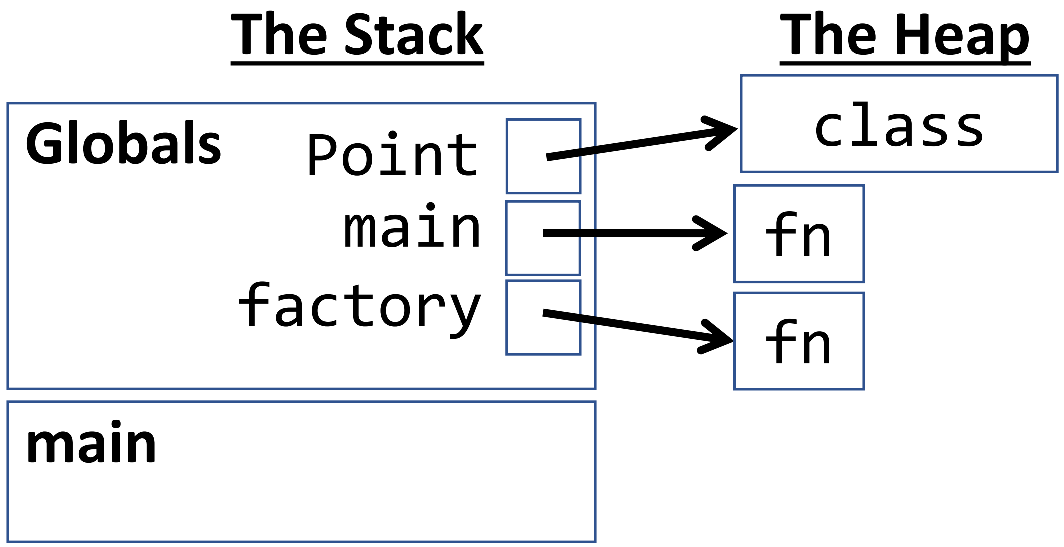
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



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

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```

Function Call

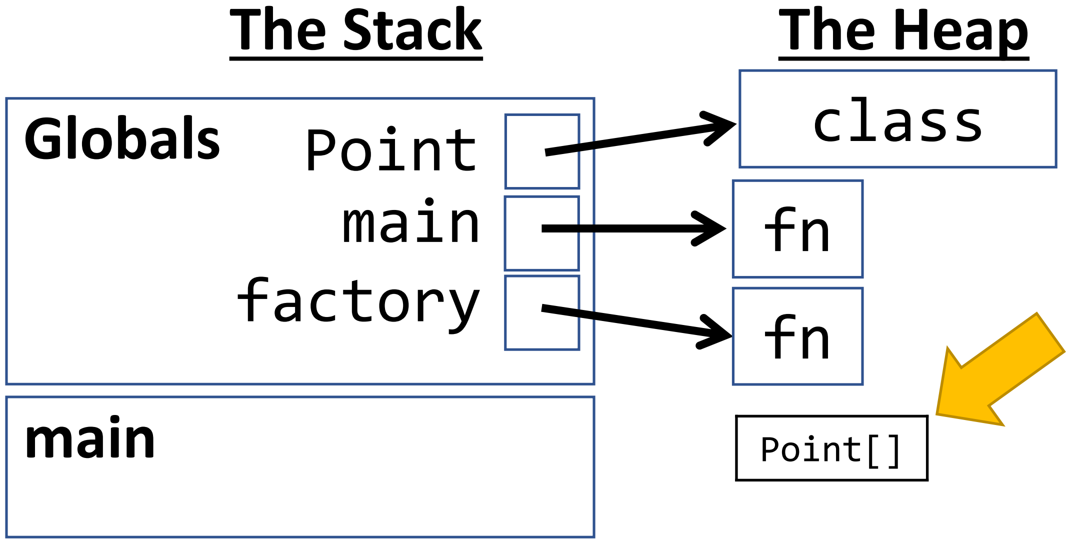
Add a frame to the stack with the function's name in it. No parameters means we can jump straight in!



Array Literal

Construct a corresponding array on the dynamic memory heap. The literal is an empty array, so add no elements to it at initialization.

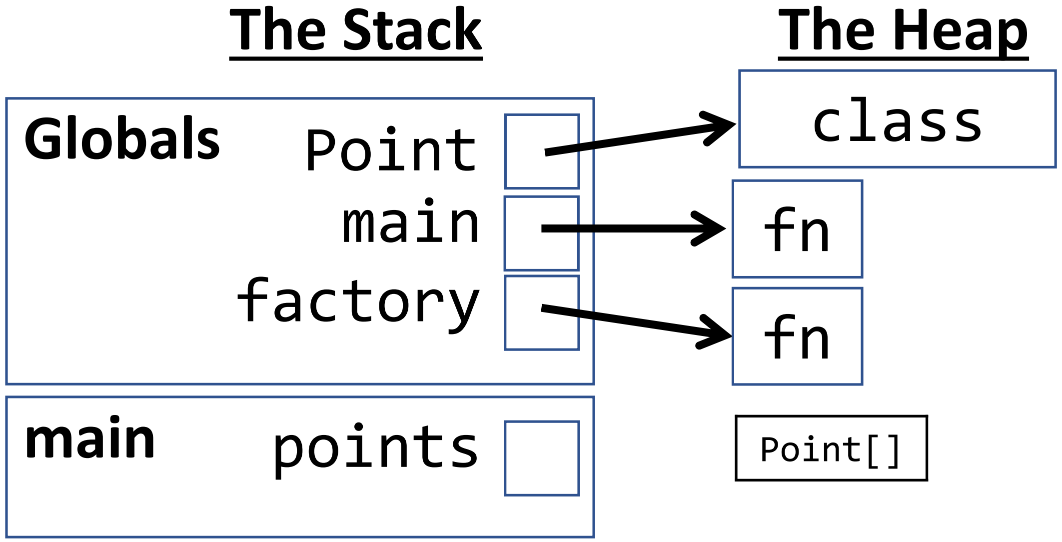
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Variable Declaration

Add the variable's name to the current frame of the call stack.

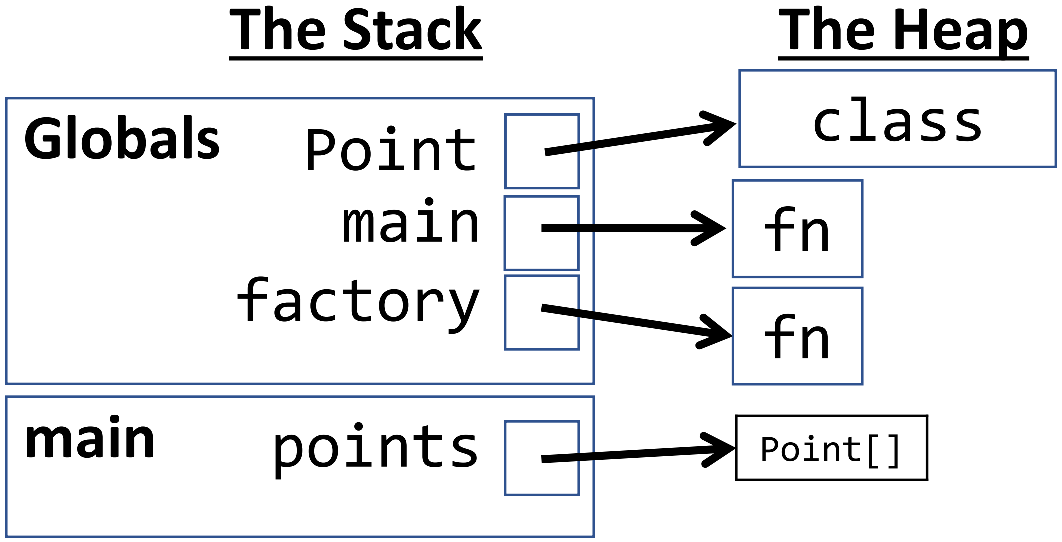
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Variable Assignment - Array

Assign a reference from the variable to the array on the heap.

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

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```

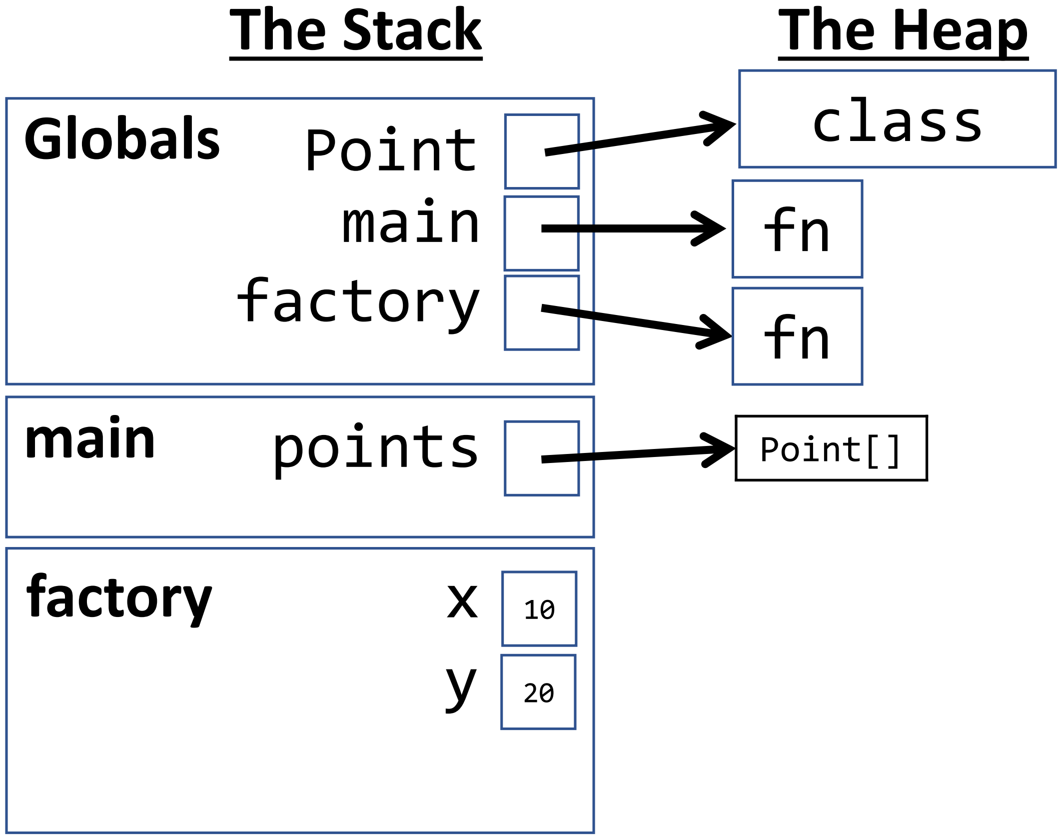


```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```

Function Call

Establish a new frame on the stack for the function call. Write in the function's name.

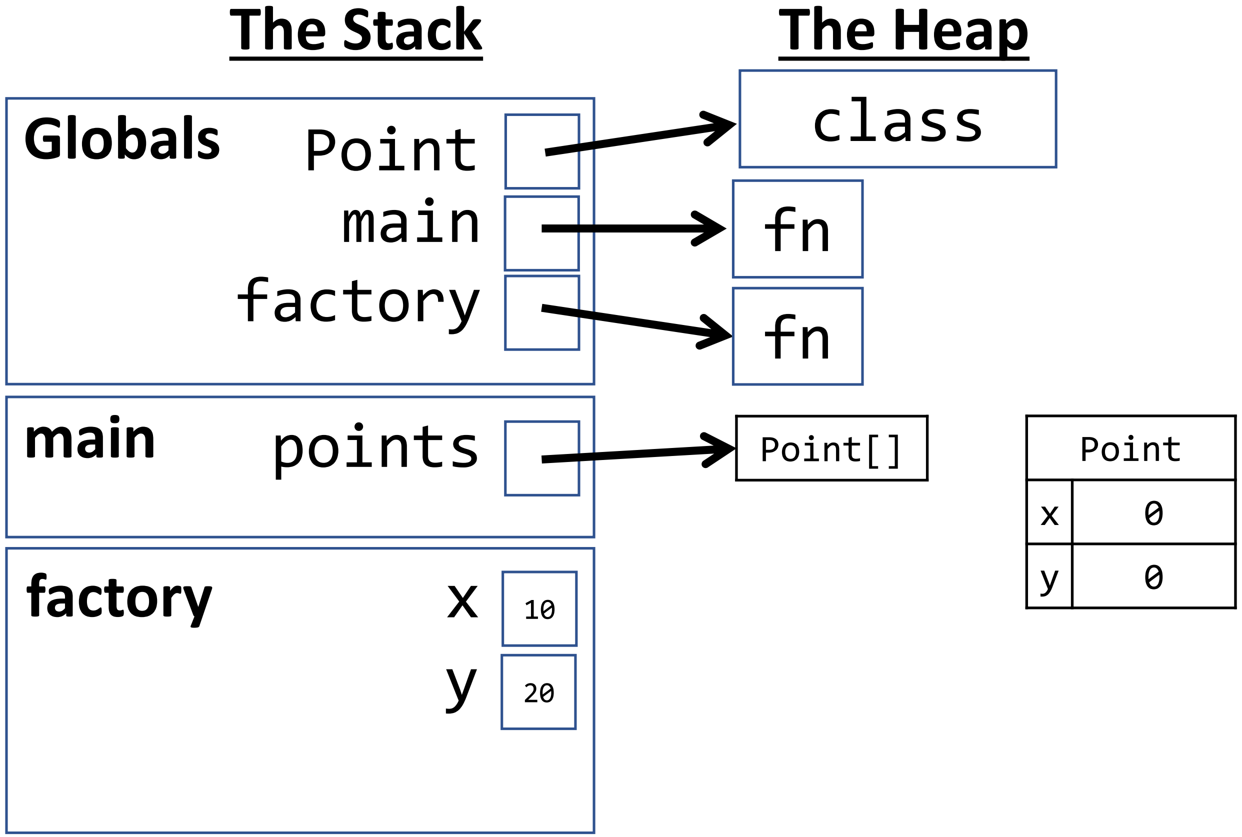
For each parameter, assign it its corresponding argument value.



Object Construction

When a new object is constructed, which happens whenever you see the new keyword followed by a class name, establish it on the Heap. Assign its properties their default values.

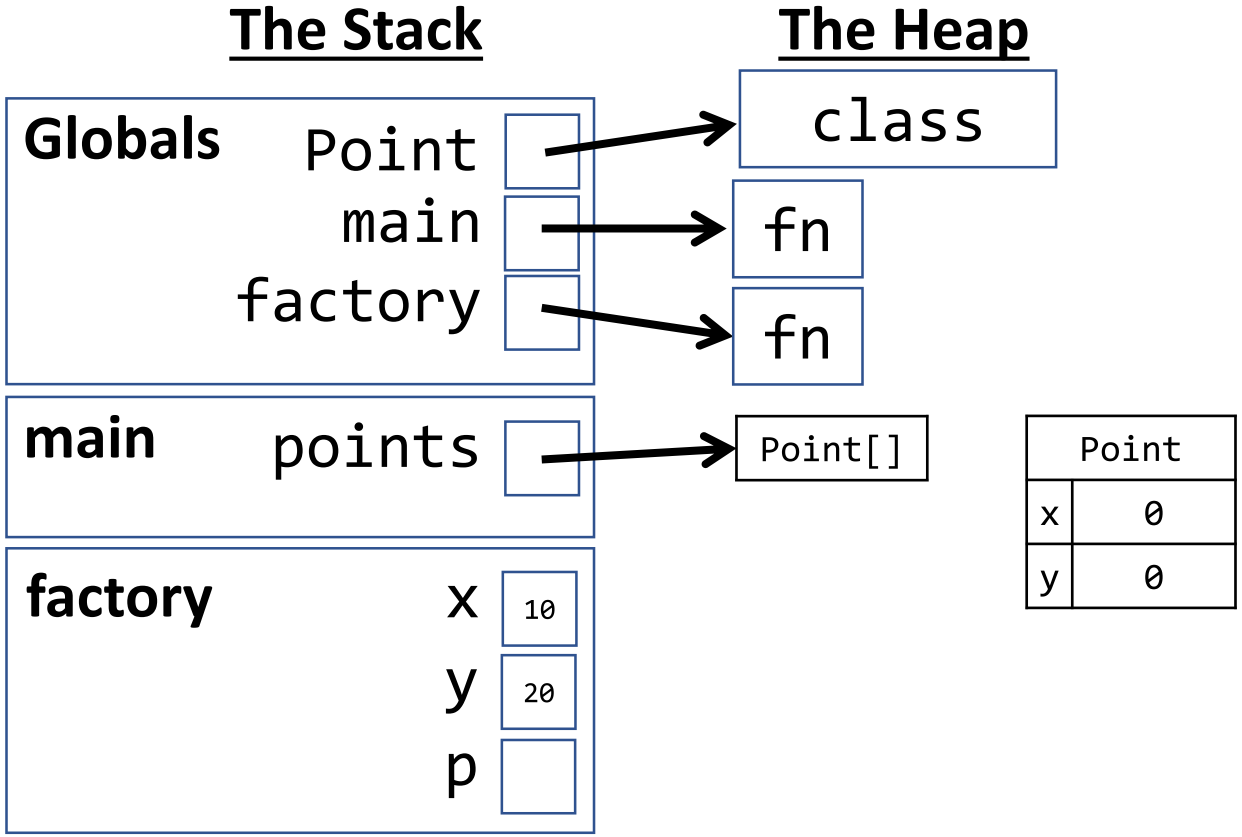
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Variable Declaration

Add the variable's name to the current frame of the call stack.

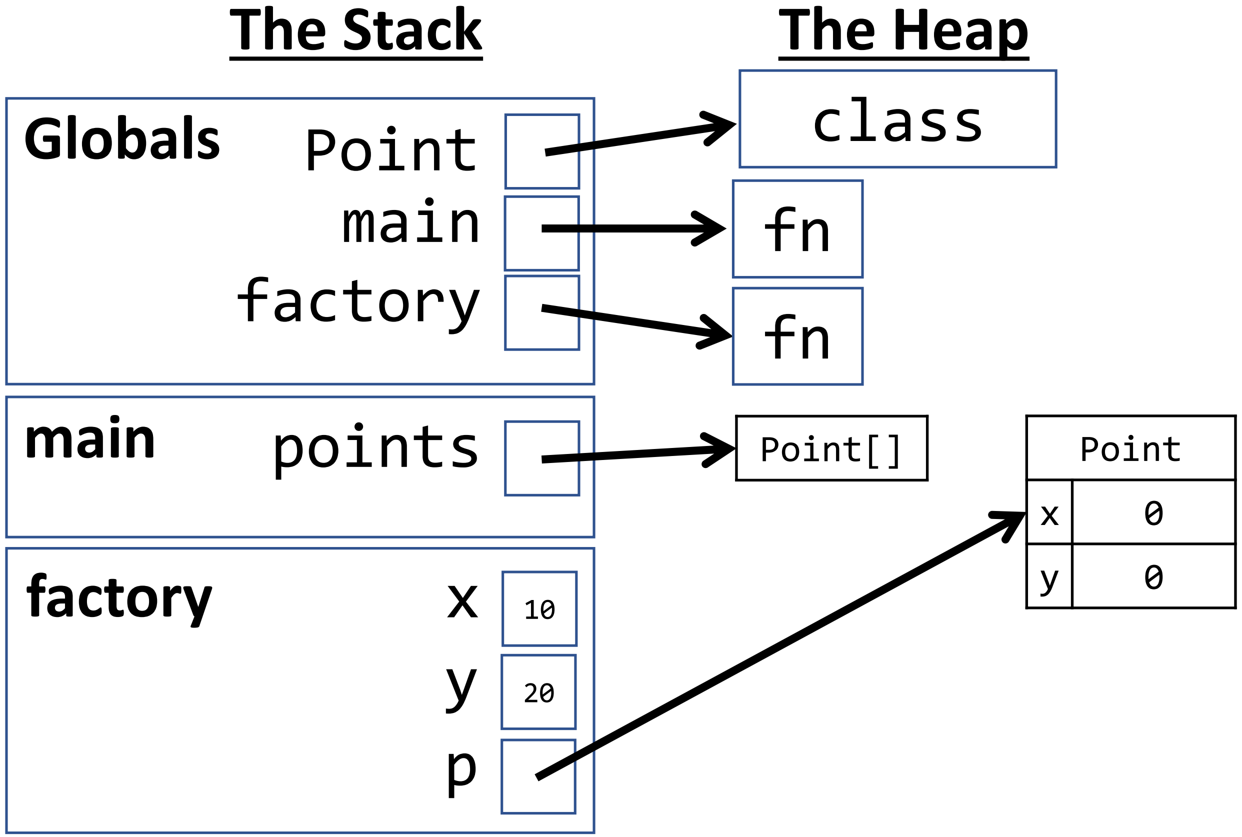
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Variable Assignment - Object

When an object is assigned to a variable, the variable's value is a reference (or **pointer** arrow) to the object. We would say, "aPoint now refers to a Point object on the Heap."

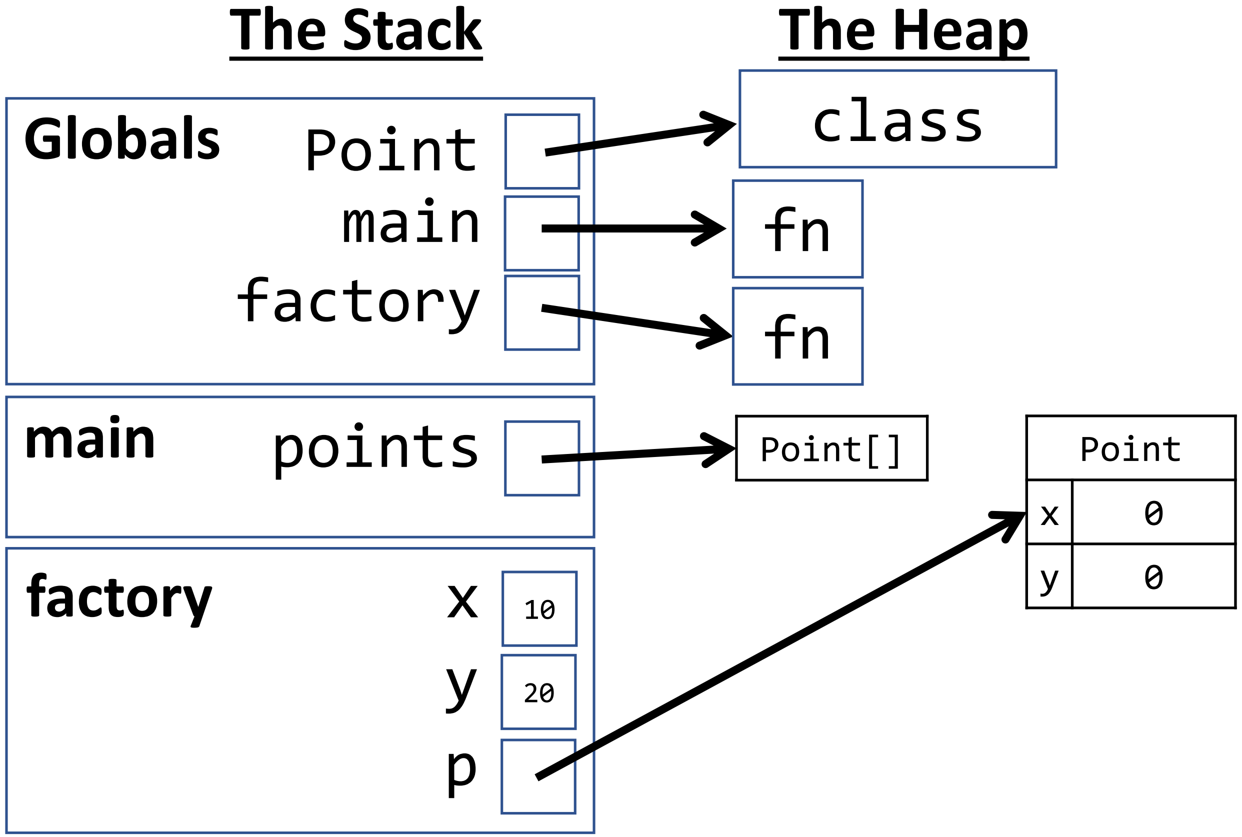
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Variable Access

Use name resolution to find the value x is bound to.

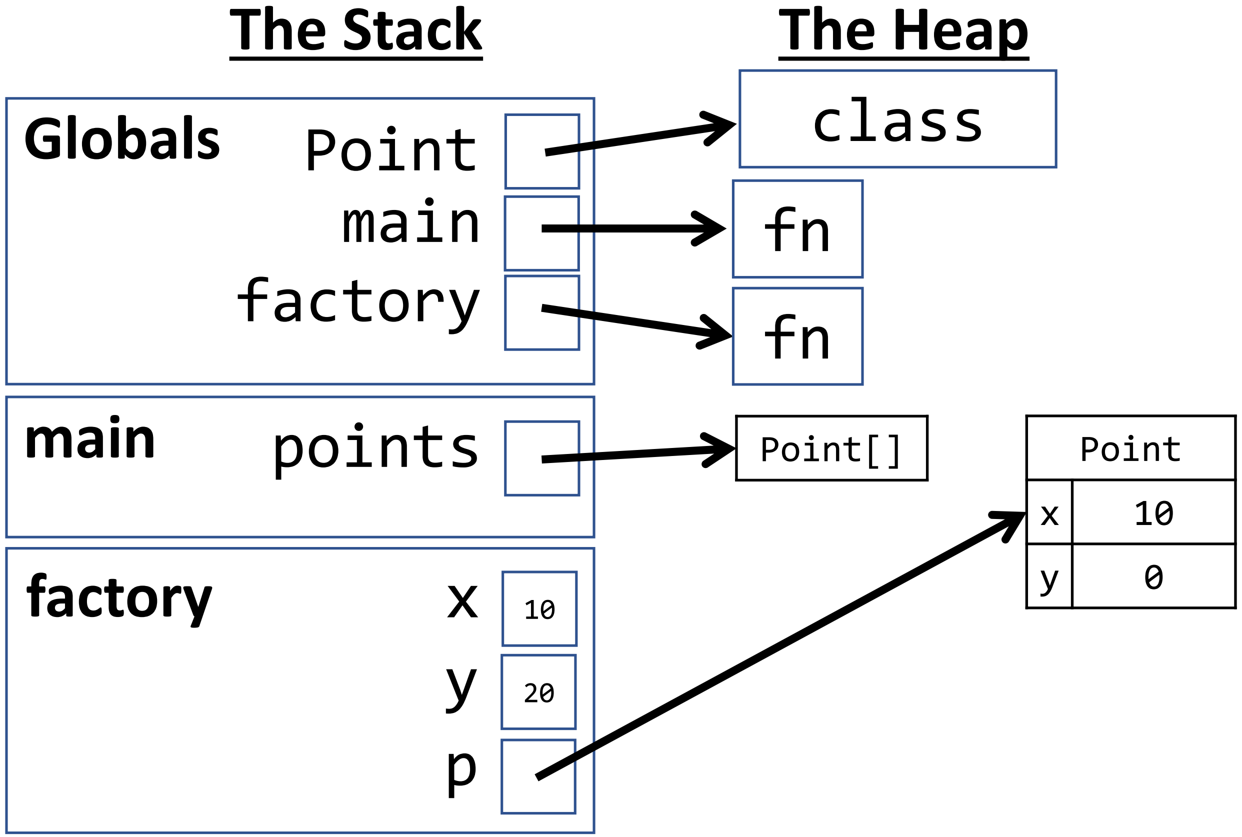
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Property Assignment

Use name resolution to find the p.x refers to. Then update it to the assigned value.

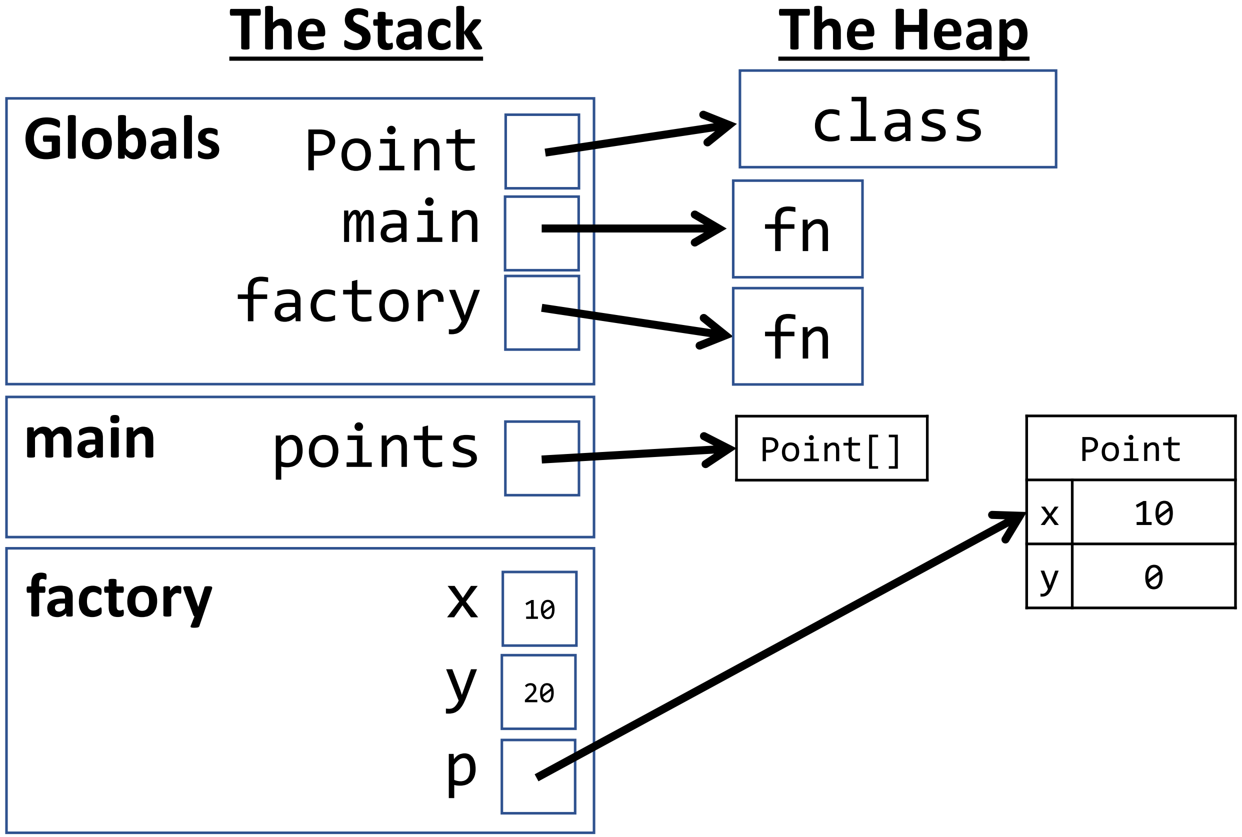
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Variable Access

Use name resolution to find the value y is bound to.

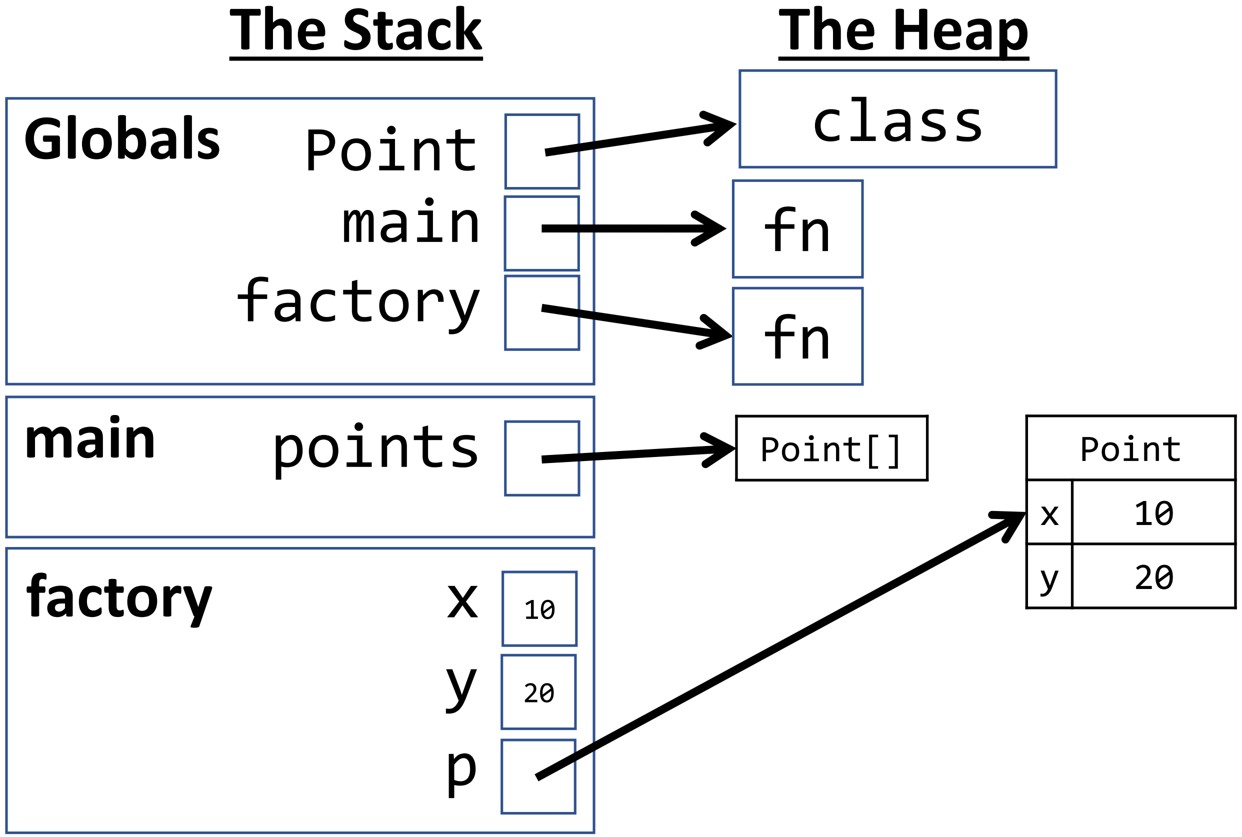
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Property Assignment

Use name resolution to find the p.y refers to. Then update it to the assigned value.

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

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



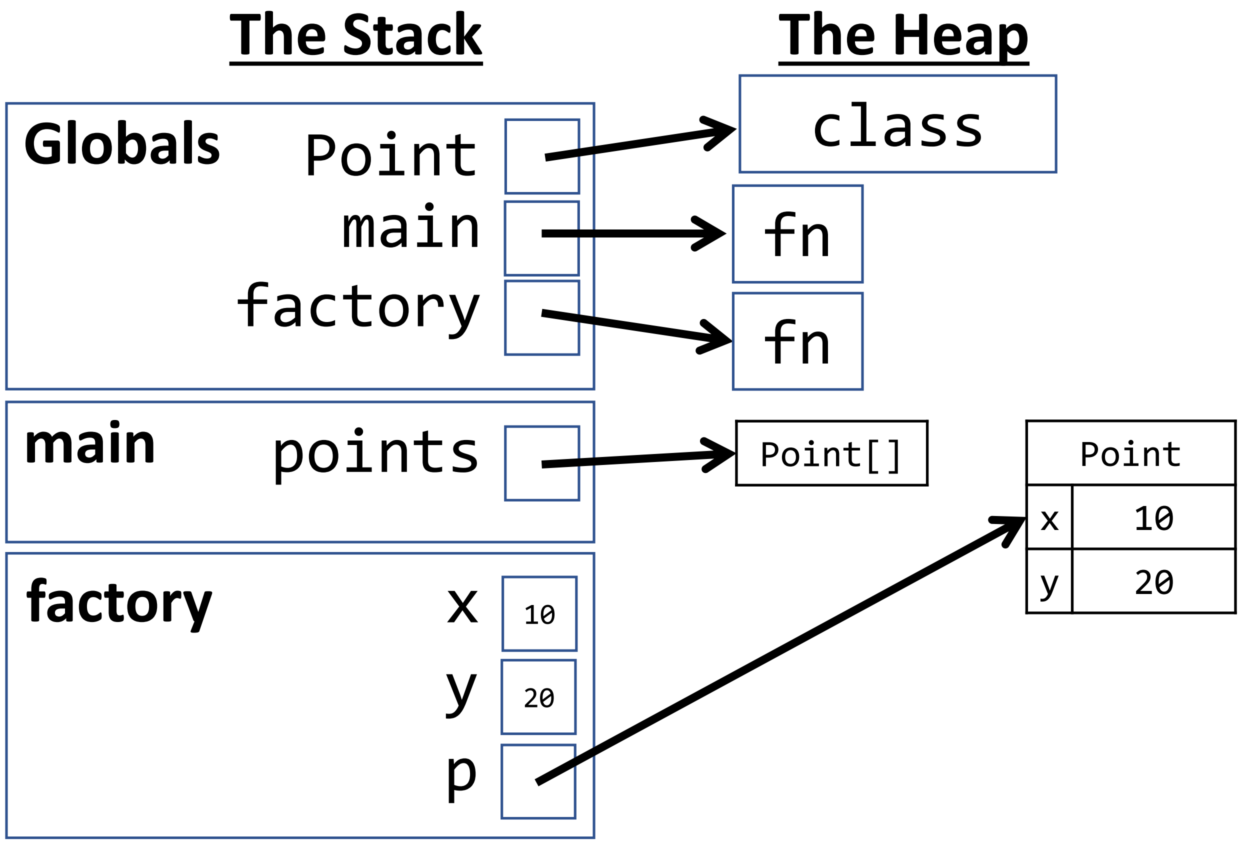
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```

Variable Access

Use name resolution to find the value p is bound to.

NOTICE!!! In this case, **p** is bound to a **reference / pointer arrow**.

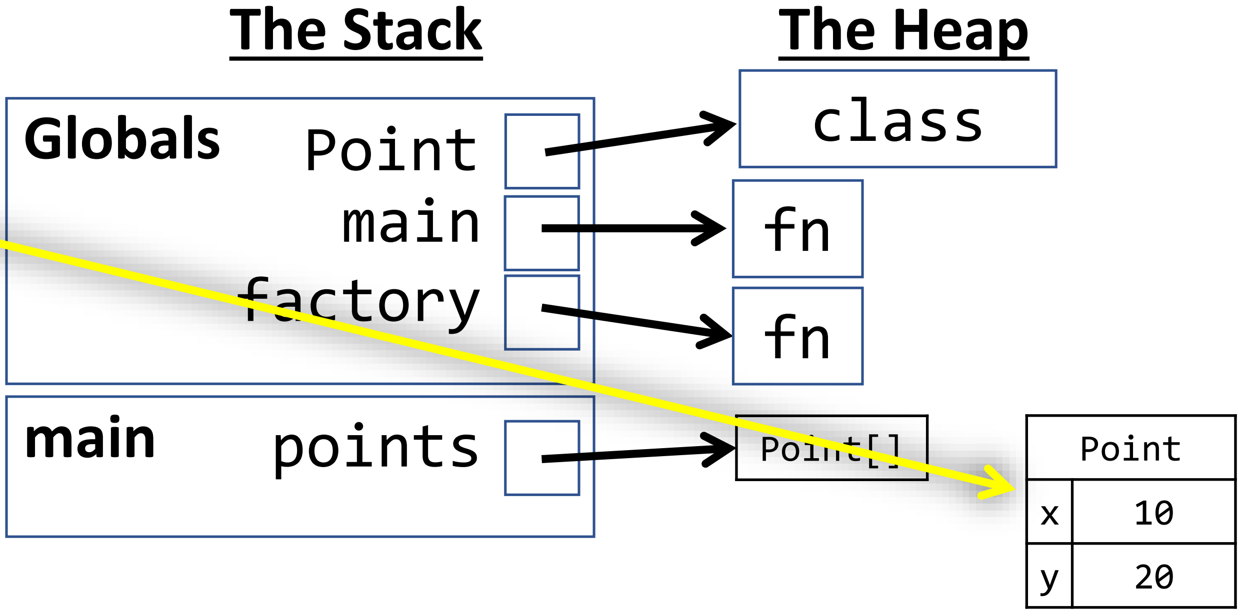
This reference is what will be returned!



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

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```

Return Statement

The return statement will return the reference **p** held back to where the call originated. This slide draws that arrow for illustration purposes. The frame for the *factory* call is cleared.

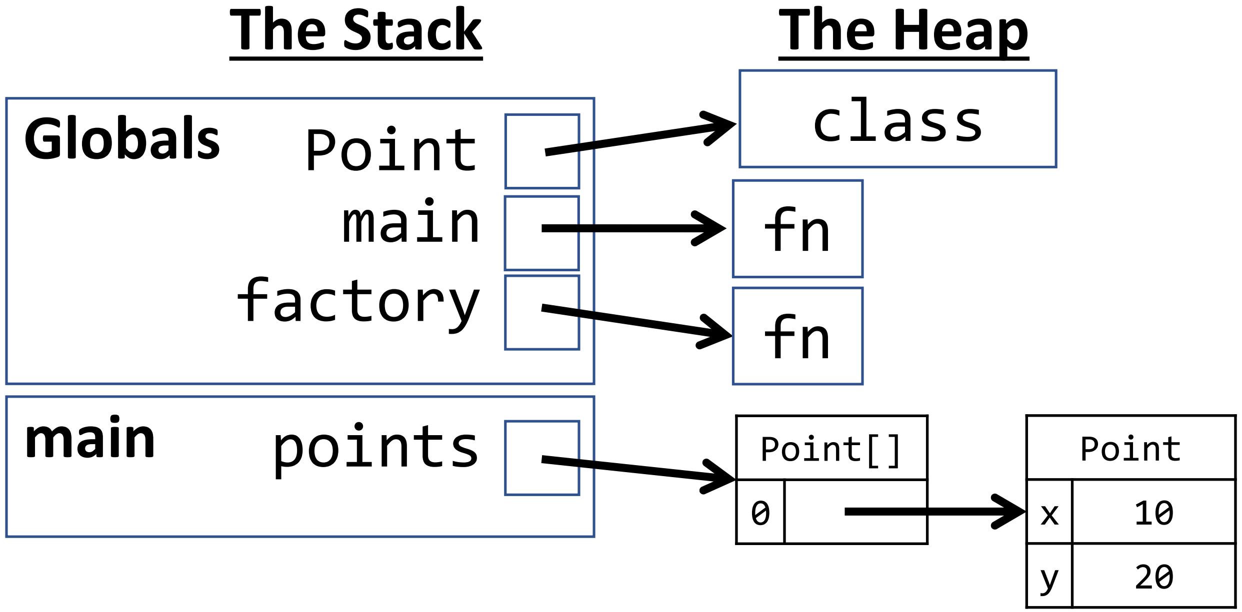


Array Element Assignment

The returned value (a reference to the newly constructed Point object in the heap) is assigned to the array.

Notice! A value on the heap (for example, an array element *or* an object property) *can* refer to other values on the heap.

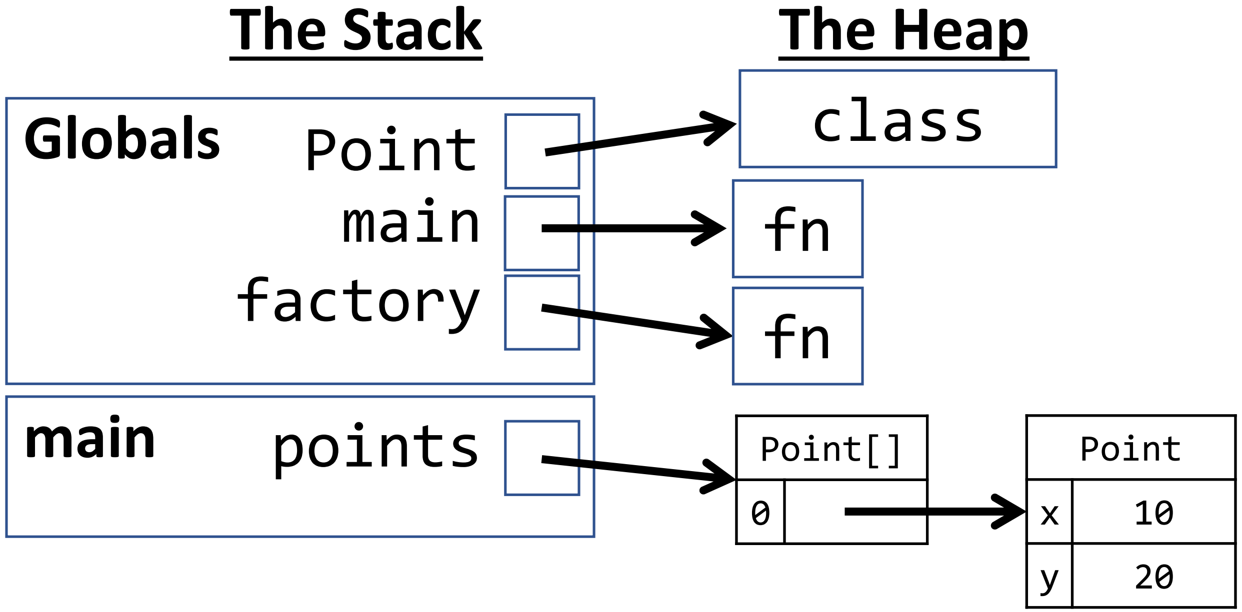
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Challenge for You

See if you can complete the environment diagram after this line evaluates...

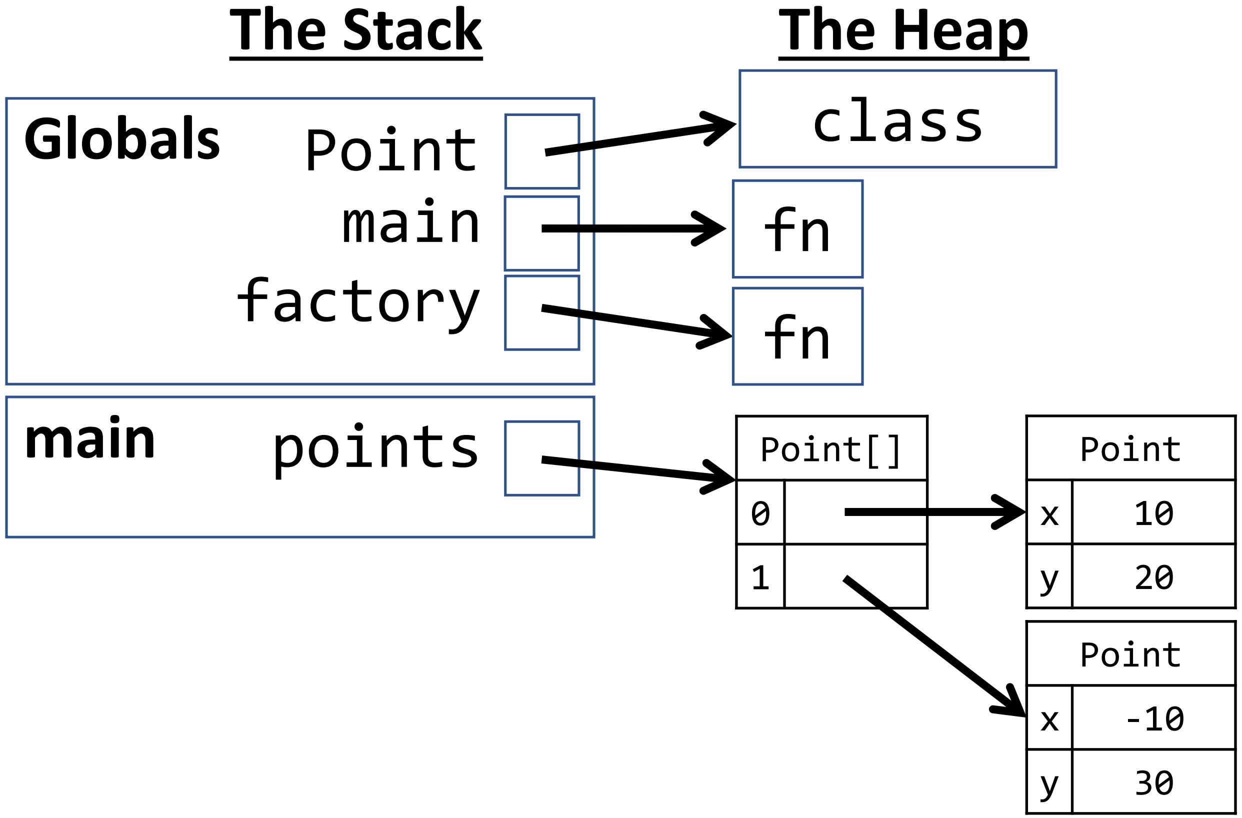
```
import { print } from "introcs";

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



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

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```

Solution

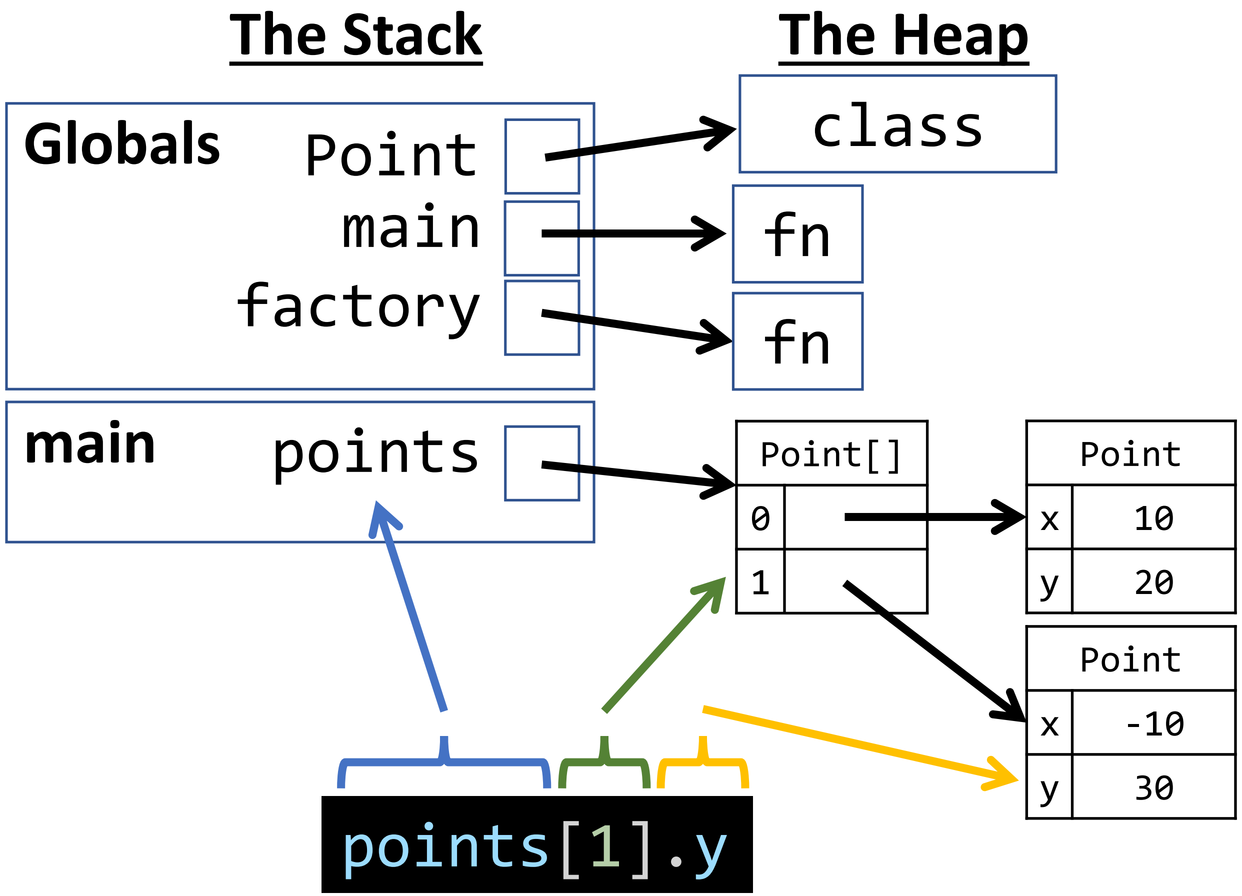
If you follow the same set of steps as the previous call to the factory function, you should end up in a state that looks like this.



Name Resolution, Array Indexing, and Property Access

Notice to determine the value that will be printed when this line evaluates, we must start with points and work our way through two references!

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

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

export let main = async () => {
    let points: Point[] = [];
    points[0] = factory(10, 20);
    points[1] = factory(-10, 30);
    print(points[1].y);
};

let factory = (x: number, y: number): Point => {
    let p = new Point();
    p.x = x;
    p.y = y;
    return p;
};

main();
```



Environment Diagramming Rules

Starting Point: add globals frame to your stack and establish an empty heap.

Variable Declaration: add name to current stack frame.

Name Resolution: look for a variable or function name in current stack frame. If not found, look in globals frame.

Array Literal: Add an indexed array with any default elements to the heap.

new Object: Add an object with default property values to the heap.

Variable Assignment: first, resolve the variable's frame location using *name resolution*. Then,

- **Function:** add a **'fn'** value to the heap, assign pointer to it in stack frame.
- **Primitive:** assign value in stack frame.
- **Object/Array:** assign pointer from the stack frame to the value in the heap.
- **Array Element:** find array in heap, update element value.
- **Object Property:** find object in heap, update property value.

Function Call: establish new frame on stack labeled as function's name.

- **Parameters:** Add names to frame, assign arguments using variable assignment rules.

Function Return: erase frame from your stack. Send return value back to caller.