2D Arrays & Scope

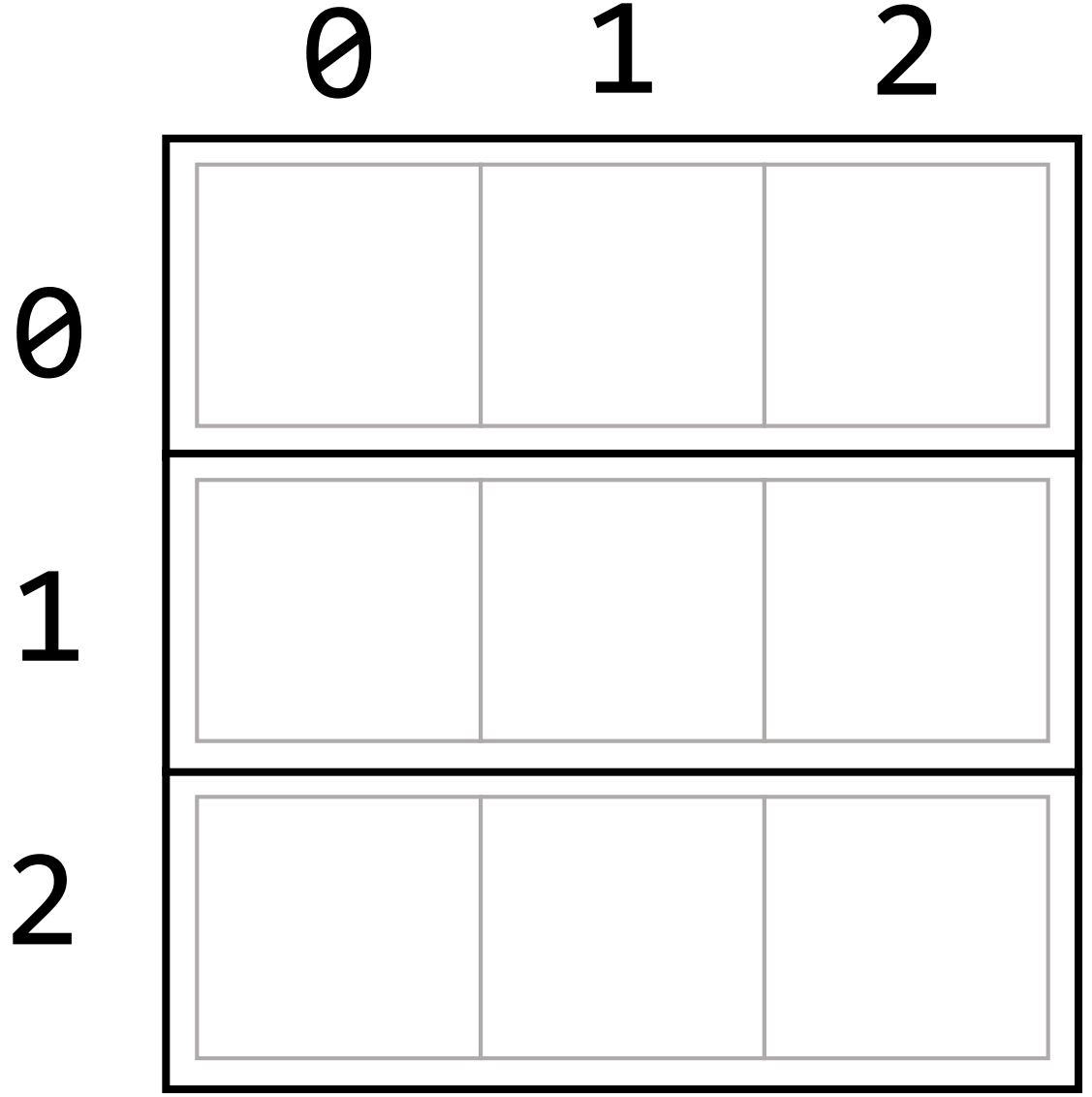
Lecture 8

CQ0. What is the output?

```
1   import { print } from "introcs";
2
3   export let main = async() => {
4
5       let letters: string[][] = [
6           ["a", "b", "c"],
7           ["d", "e", "f"],
8           ["g", "h", "i"],
9       ];
10      print(letters[2][1]);
11
12  };
13
14  main();
```

2D Arrays

- Easy to think of as a 2D grid
- Useful for storing data naturally represented in a table or grid
- For example: Picture Data
- Really an “array of arrays”



An “Array of Arrays”…

- The way we declare an array of some type is:

  `<type>[]`

- For example, if we want an **“array of numbers”**
  - Each element’s **type** is **`number`**
  - So we’d declare:

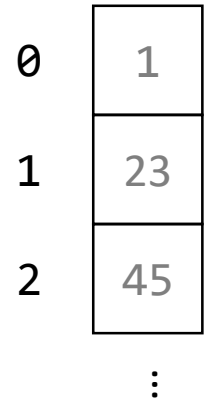
  `number[]`

- What if we wanted an **“array of number arrays”**
  - Each element’s **type** is **`number[]`**
  - So we’d declare the type to be:

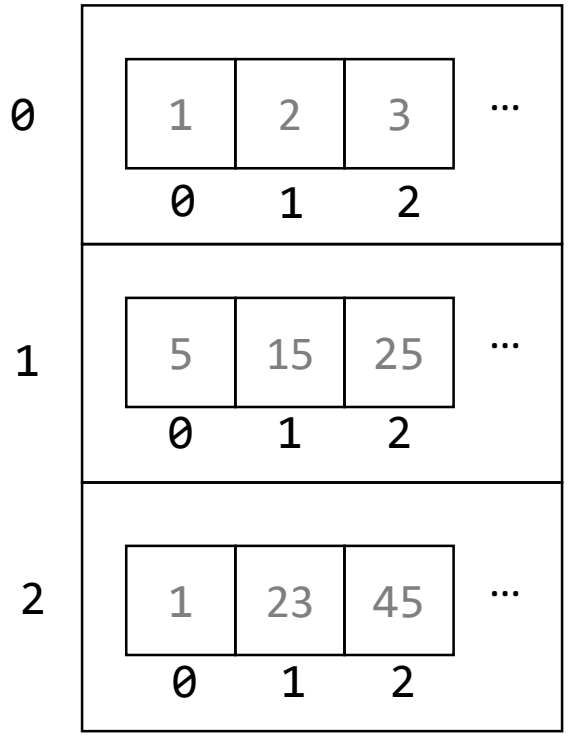
  `number[][]`

- You *can* keep doing this with arrays that are 3D, 4D, and so on…

“**array** of **numbers**”



“**array** of **number arrays**”



2D Array Operations – Variable Declaration

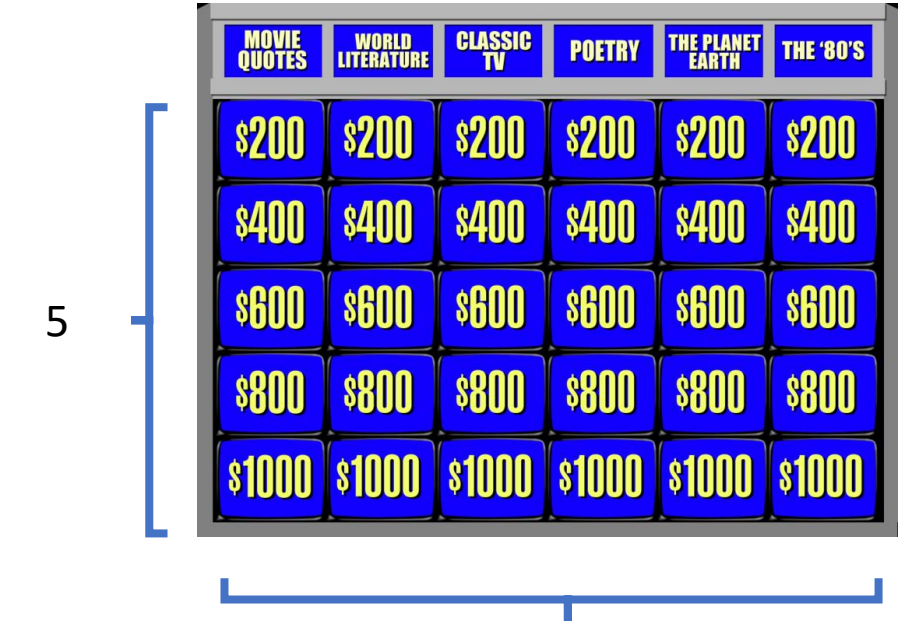
`let <name>: <type>[][]`

- For example:

```
let jeopardyBoard: number[][];
```

2D Array Operations – Initialization
Row-major Literals

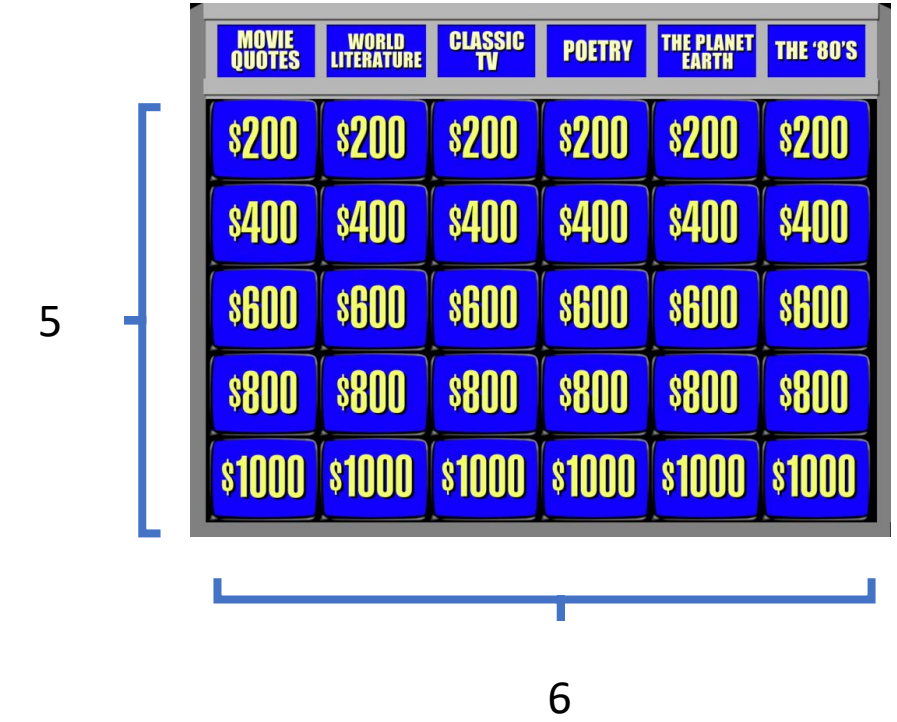
```
let jeopardyBoard: number[][] = [
    [  200,  200,  200,  200,  200,  200 ],
    [  400,  400,  400,  400,  400,  400 ],
    [  600,  600,  600,  600,  600,  600 ],
    [  800,  800,  800,  800,  800,  800 ],
    [ 1000, 1000, 1000, 1000, 1000, 1000 ]
];
```



- Arrays can be logically organized as "column-major" or "row-major" where "major" refers to the outer most array. In this example, we're initializing row-major.
- Row-major order means we're accessing elements via a first index of row and a second index of column. For example **`jeopardyBoard[0][2]`** is **`200`**.

2D Array Operations – Initialization – Column-major Literals

```
let jeopardyBoard: number[][] = [
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ]
];
```



- In this example, we're initializing column-major instead.
- Column-major order means we're accessing elements via a first index of column and a second index of row. For example **`jeopardyBoard[0][2]`** is **`600`**.

Row-major vs. Column-major Layouts

- **In COMP110, we'll use row-major based 2D arrays**

- For most sizes of problems you'll encounter *it doesn't matter,* as long as you are consistent and document your decisions.

- In upper-level classes you'll learn optimal layouts depend on both:
  1. How your programming language organizes 2D arrays in memory
  2. How your algorithms tend to iteratively access the arrays

2D Array Operations – Initializing with Code

- Let's implement the **2d-arrays-helpers.ts** function with the following signature:

```
array2d(rows: number, cols: number): number[][]
```

- This function is given the # of rows and # of columns and its purpose is to initialize a 2D array of numbers where each element's initial value is 0.

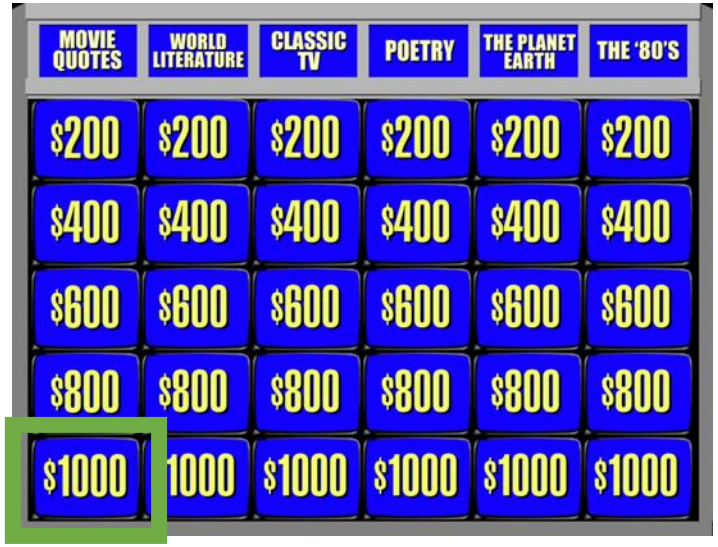
```
export let array2d = <T> (rows: number, cols: number): number[][] => {
    let a: number[][] = [];

    for (let row = 0; row < rows; row++) {
        // Initialize the next row as an empty array
        a[row] = [];
        for (let col = 0; col < cols; col++) {
            a[row][col] = 0;
        }
    }

    return a;
};
```

2D Array Operations – Assigning to Inner Array

<name>[index$_r$][index$_c$] = <value>;

- For example:

```
jeopardyBoard[4][0] = 1000;
```

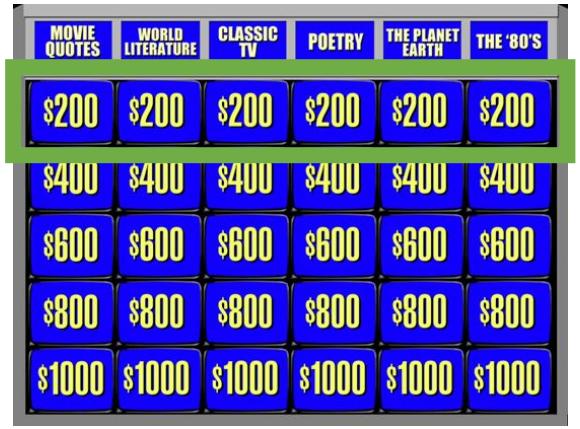


2D Array Operations – Assigning to Outer Array

<name>[index$_r$] = <array of correct type>;

- For example:

```
jeopardyBoard[0] = [200, 200, 200, 200, 200, 200, 200];
```

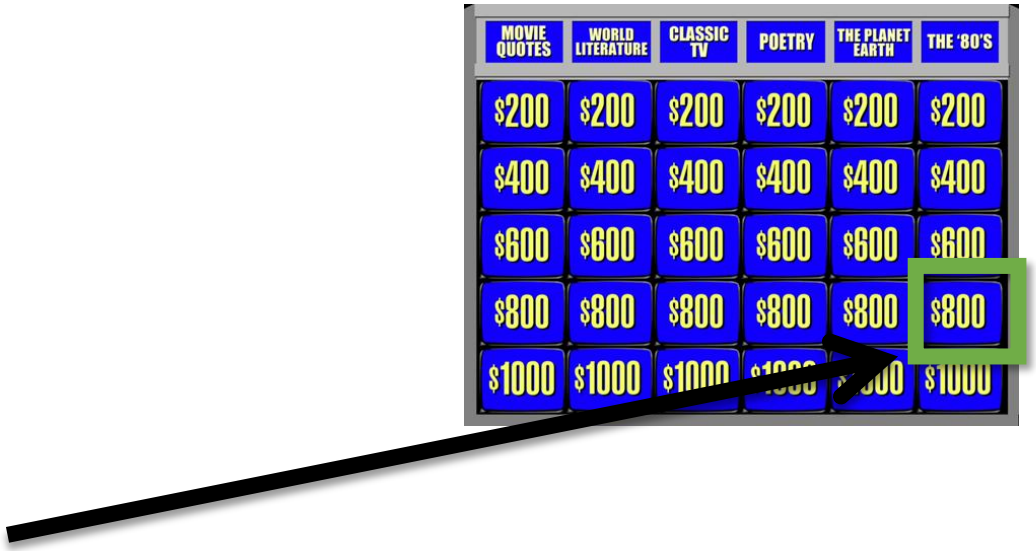


2D Array Ops – Accessing Element of Inner Array

<name>[index$_r$][index$_c$]

- For example:

```
print(jeopardyBoard[3][5]);
```

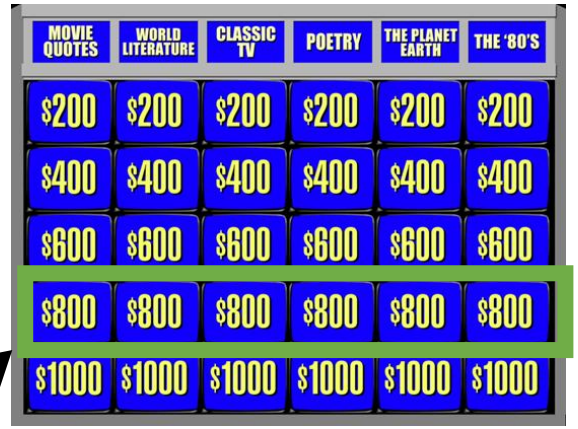


2D Array Ops – Accessing Element of Outer Array

<name>[index$_r$]

- For example:

```
let eightHundreds: number[] = jeopardyBoard[3];
```



2D Array Ops – Get # Elements of Outer Array

<name>.length

- For example:

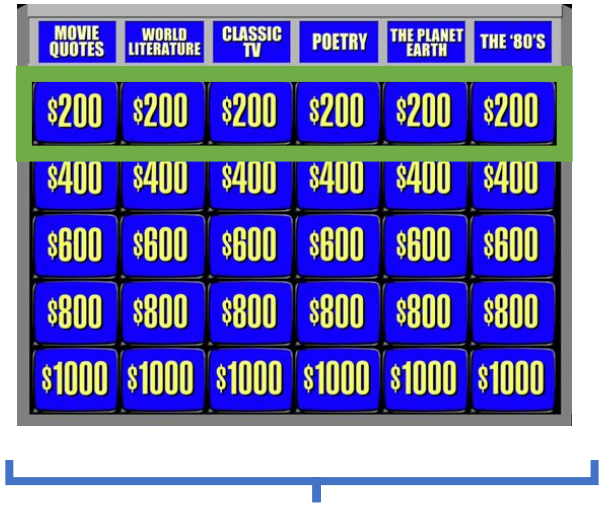
```
print(jeopardyBoard.length);
```

2D Array Ops – Get # Elements of Inner Array

<name>[index$_r$].length

- For example:

```
print(jeopardyBoard[0].length);
```



6

- *Usually* 2D arrays are perfectly rectangular in lengths. *However*, there is no guarantee each inner array has the same # of elements as one another.

```
import { print } from "introcs";
import { array2d } from "./2d-arrays-helpers";

// Declare
let a: number[][];

// Initialize using Literals
a = [
      [1, 2],
      [3, 4],
      [5, 6]
];

// Initialize using array2d helper function
a = array2d(12, 10, 0);

// Assigning to an element
a[3][5] = 1;

// Read the # of rows
print(a.length);

// Read the # of cols
print(a[0].length);

// Read from an element
print(a[1][0]);

// Read from a top-level element
print(a[3]);

// Inspect complete array in developers' tools console
console.log(a);
```

Hands-on: Multiplication Table

- Change examples to multiply-app.ts and 2d-array-helpers.ts
- Implement the function **multiply2d** in 2d-array-helpers.ts so that when called from main with 10 rows and 12 columns the printed output is as shown to the right.
  - Hint: refer to the array2d function for guidance.
- The first row and column should not be all 0s!
- Check-in on PollEv.com/compunc

index	0	1	2	3	4	5	6	7	8	9	10	11	length
0	1	2	3	4	5	6	7	8	9	10	11	12	12
1	2	4	6	8	10	12	14	16	18	20	22	24	12
2	3	6	9	12	15	18	21	24	27	30	33	36	12
3	4	8	12	16	20	24	28	32	36	40	44	48	12
4	5	10	15	20	25	30	35	40	45	50	55	60	12
5	6	12	18	24	30	36	42	48	54	60	66	72	12
6	7	14	21	28	35	42	49	56	63	70	77	84	12
7	8	16	24	32	40	48	56	64	72	80	88	96	12
8	9	18	27	36	45	54	63	72	81	90	99	108	12
9	10	20	30	40	50	60	70	80	90	100	110	120	12

number[][]

```
export let multiply2d = <T> (rows: number, cols: number): number[][] => {
    let a: number[][] = [];

    for (let row = 0; row < rows; row++) {
        // Initialize the next row as an empty array
        a[row] = [];
        for (let col = 0; col < cols; col++) {
            a[row][col] = (row + 1) * (col + 1);
        }
    }

    return a;
};
```

Challenge Question #1 - What is printed?

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

let x: number;

export let main = async () => {
   x = 0;
   f();
   print(x);
};

let f = (): void => {
   x += 1;
};

main();
```

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

let x: number;

export let main = async () => {
   x = 0;
   f();
   print(x);
};

let f = (): void => {
   x += 1;
};

main();
```

Notes on the Globals Frame

- The final segment of memory to add to our environment diagrams is the Globals frame.
- Global variables (introduced today) are bound in the Globals frame.
- The Globals frame is established before any function call frames on the call stack and behaves like any other frame when variable declarations are reached.

Globals - Starting Point

When a program* is loaded by an interpreter, it begins with an empty** stack and heap. The top-most frame of our stack is called the **Globals frame**.

```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```

The Stack

The Heap

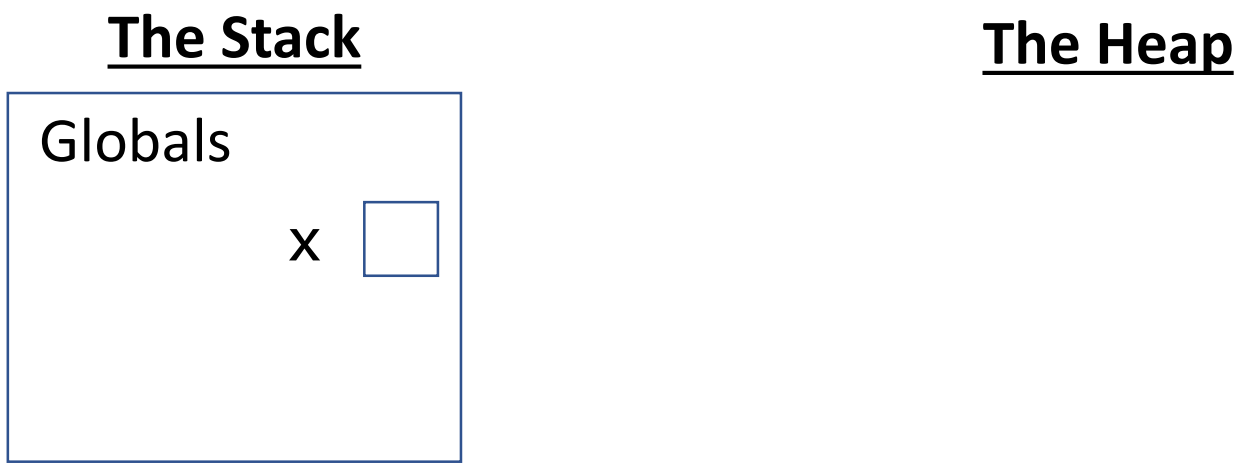
* The `import { print }` statement is hidden in this example for illustration, but it should be there.

** This is a simplification for modeling purposes. In reality there are a lot of commonly used names established globally by the programming language (like Math)

Variable Declaration

When a **variable** is declared, add its name to the current frame on the stack.
Since **x** is declared in the global frame it's called a **global variable**.

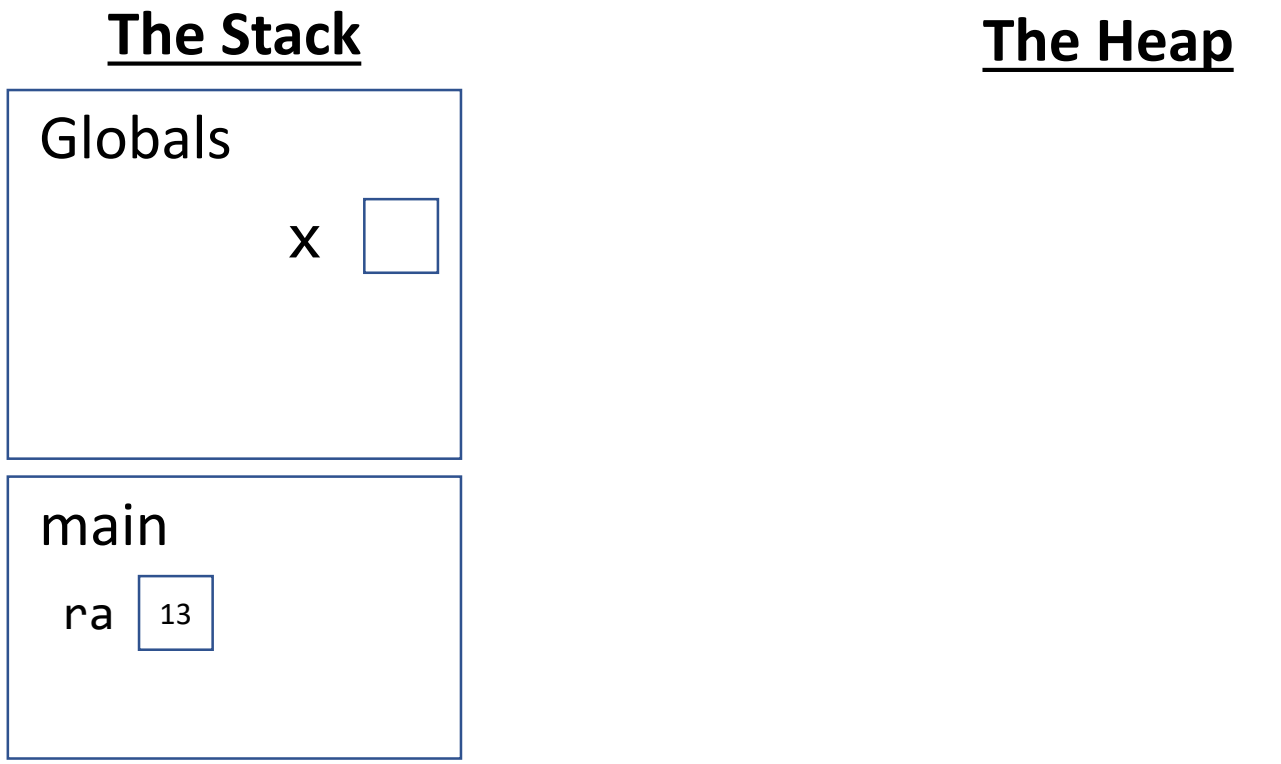
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Function Call

When a function call is encountered, a new **frame** is added to your stack. Label it with the function name. When it has parameters, you'll need to add them, too.

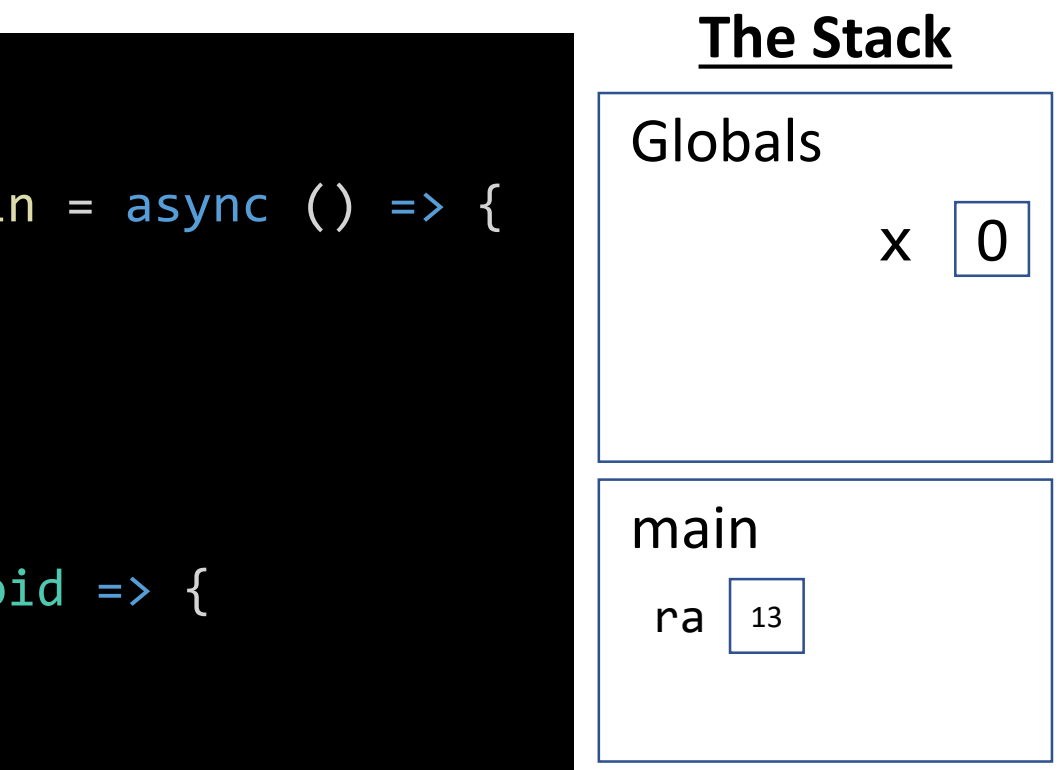
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Name Resolution - Variable Assignment

How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals. Primitive? Assign value in stack.

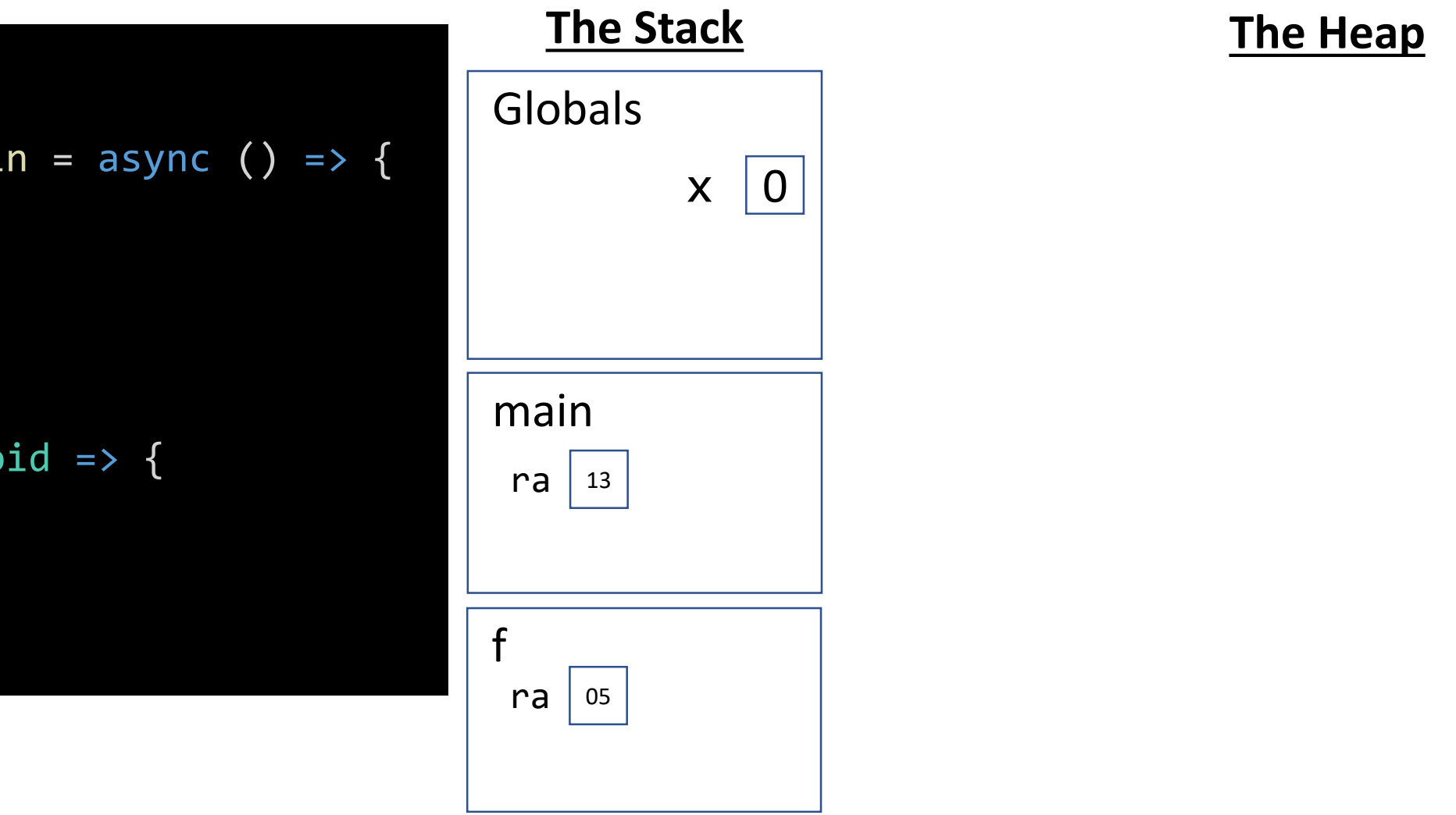
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Function Call

When a valid function call is encountered, a new **frame** is added to your stack. Label it with the name, add return address line, and establish parameters.

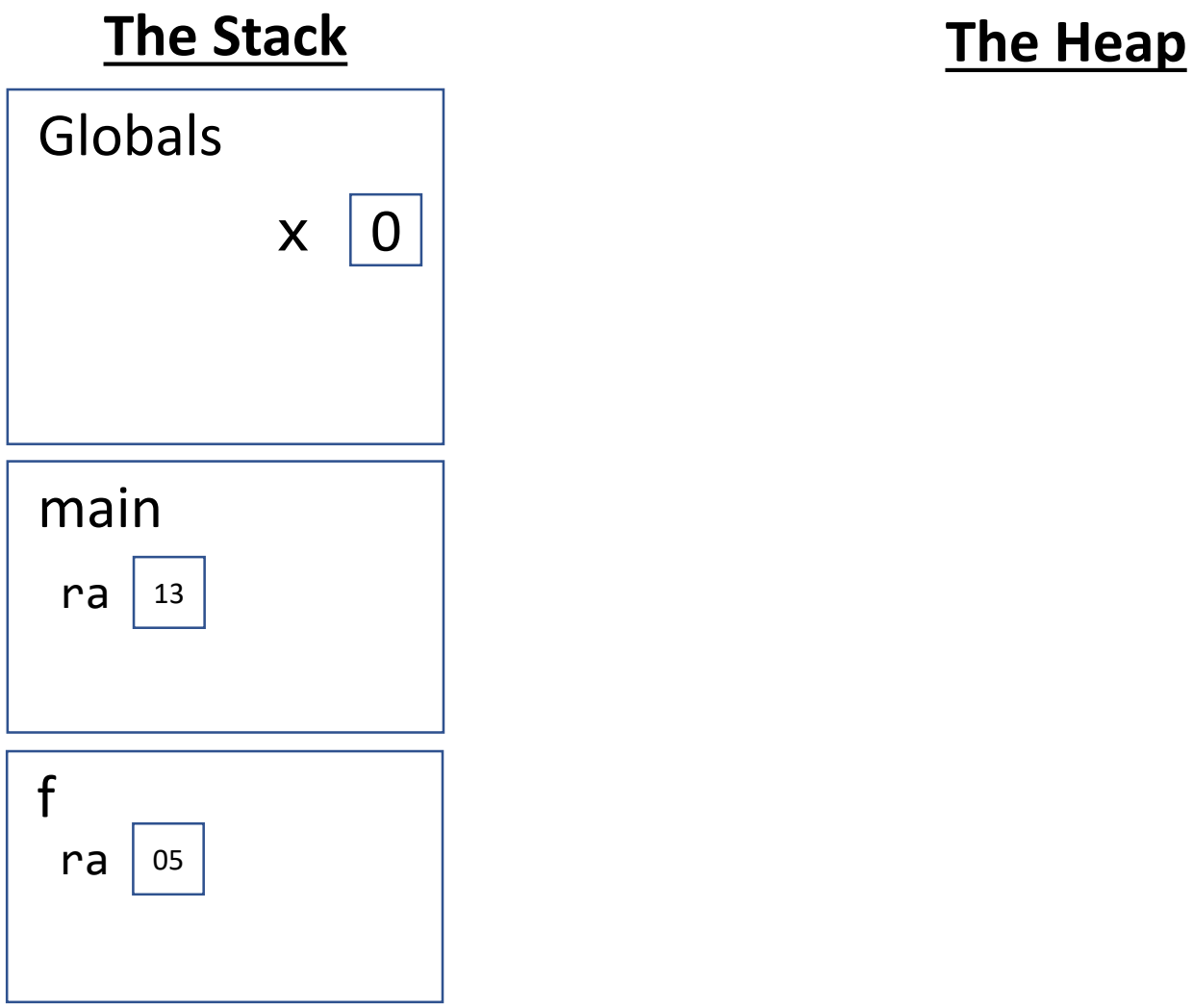
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Name Resolution - Variable Access

How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals frame. In this case, it's in globals!

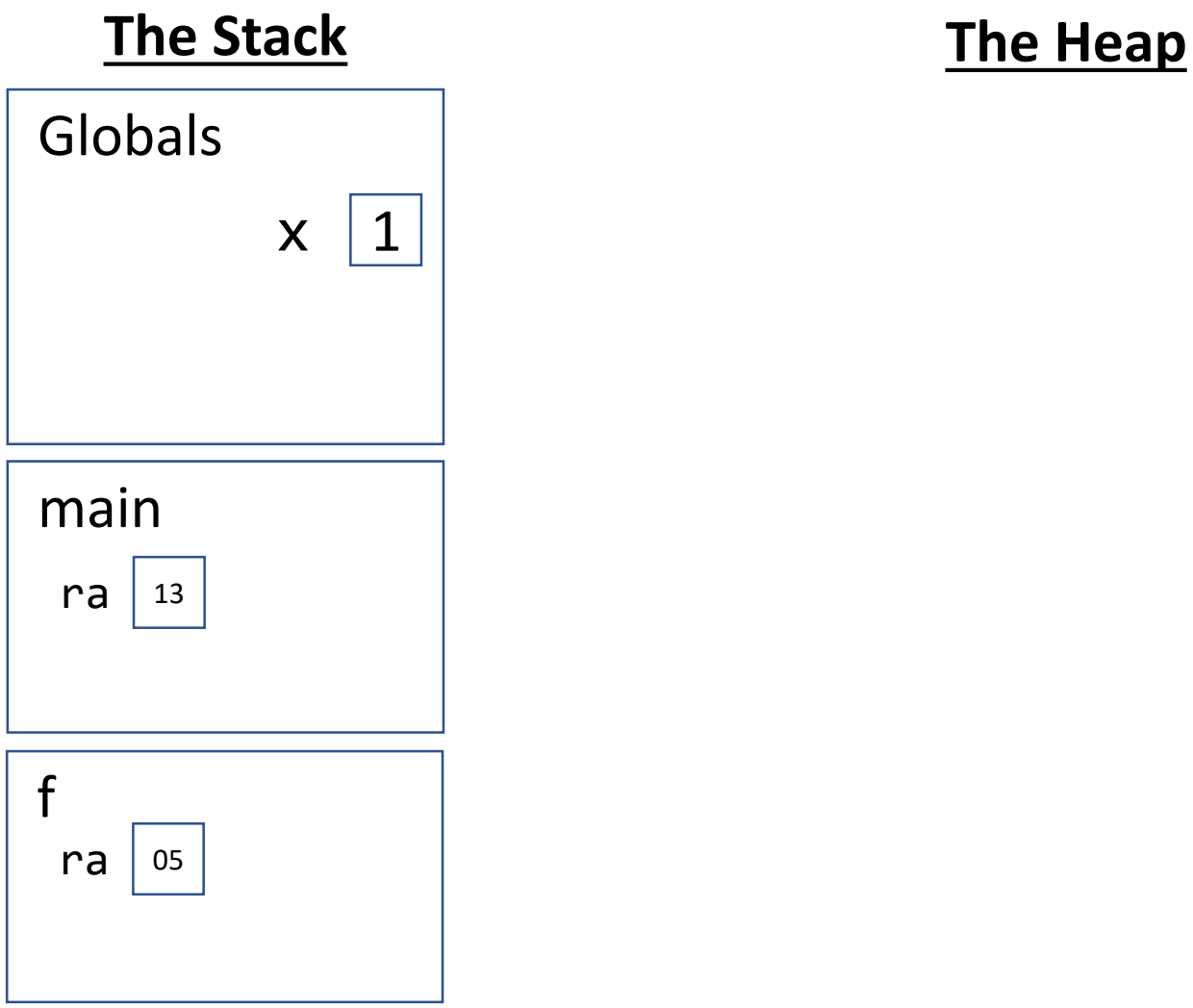
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Name Resolution - Variable Assignment

When a **primitive variable** is assigned a value, first resolve its frame location by name, then update its value.

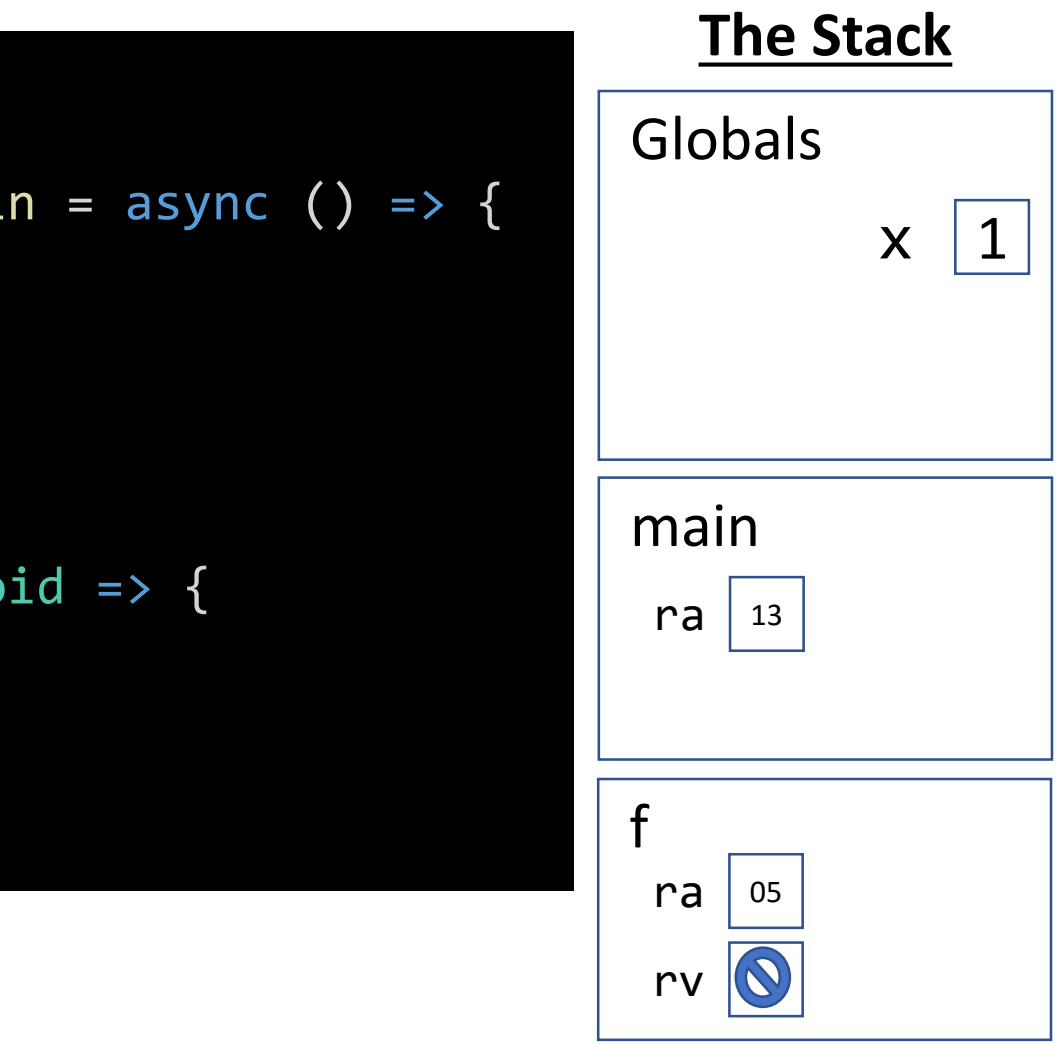
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Function Return - `void` Functions

When a **void function** completes, it returns nothing and control jumps back to the return address.

```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```

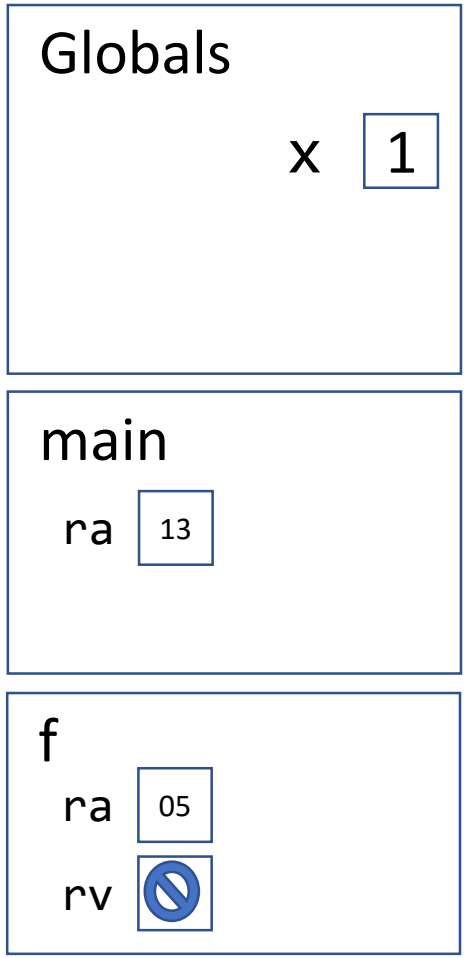


Name Resolution - Variable Access

How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals frame. **In this case, it's in globals**!

```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```

The Stack

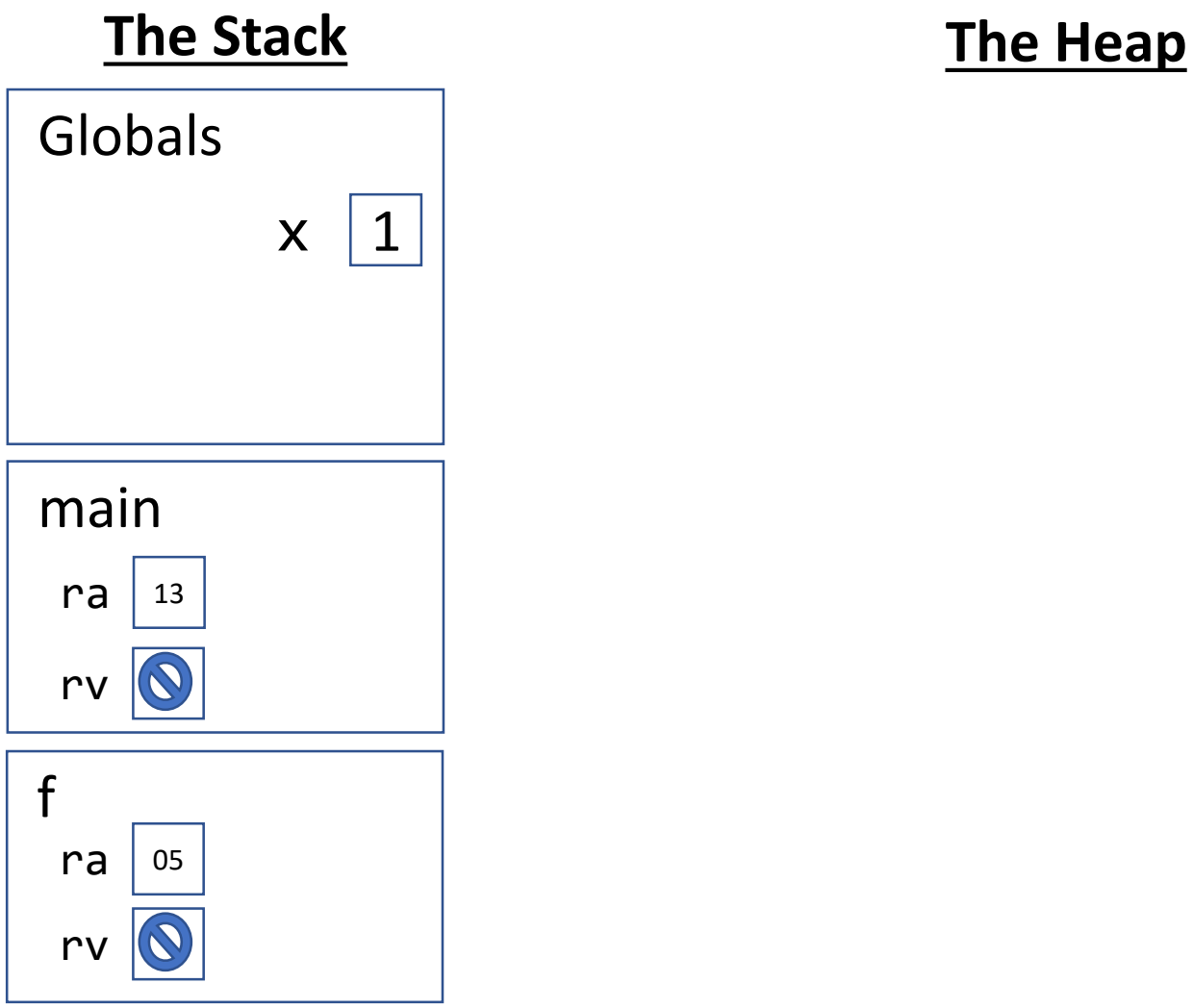


The Heap

Function Return - `void` Functions

When a **void function** completes, it returns nothing and control jumps back to the return address.

```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Variable **Scope**

- The **scope** of a variable refers to *where* its name is *usable* in code

- **Local Variable Scope**
  - Defined within some function body or other block.
  - Useable after initialization and until the *block* it is defined inside closes.

- **Global Variable Scope**
  - Defined outside of any function body or block.
  - Usable after initialization from *anywhere* in the program.

- When in doubt: *prefer local variables*.
  - Since global variables can be changed from anywhere using them is tough to debug.
  - Using global variables in smaller programs (like the next PS) can be OK, but as the size of your programs grow and your capabilities grow you will learn ways to avoid relying upon them and thus ways to write more robust, easier to debug programs.

CQ #2: What is the output?

```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       let cost = 0;
5       {
6           let costPerBook = 100;
7           let numberOfBooks = 3;
8           cost = costPerBook * numberOfBooks;
9       }
10
11      print(costPerBook);
12      print(cost);
13  };
14
15  main();
```

- Let's use the program to the left to illustrate how block scopes are represented in environment diagrams.

Block Statements (1 / 2)

- A **block** is a special kind of statement that groups multiple, related statements.

- Blocks are **enclosing curly braces** that "contain" its statements.

```
{
   // This is a block statement
   print("Statement one");
   print("Statement two");
}
```

- Blocks do not end with a semicolon after the closing curly brace. The closing curly signals the end of the block.

- The *then-block* and *else-block* of *if* statements, as well as the *repeat-block* of loops, are all *just blocks* like the one shown above.

Block Statements (2 / 2)

- Anywhere you can write a statement, you can also write a block statement.
  - Thus, you can nest inner blocks inside of outer blocks.

```
{
    print("Statement one");
    {
        print("Statement two");
    }
    print("Statement three");
}
```

- **Important** formatting rule: **each statement inside of a block is indented one additional level**!

Variable's **Block Scope** Rule

- A variable is only accessible after it is declared in the same block or in an outer, containing block.

```
{
   let x = 0;
   print(x); // OK! x declared in same block
   {
      print(x); // OK! x declared in outer block
   }
}

print(x); // ERROR! x declared in different block
```

Blocks in Environment Diagrams

```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```

- Let's use the program to the left to illustrate how block scopes are represented in environment diagrams.

Blocks in Environment Diagrams

```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```

Imports

We will not represent imported names (like functions) in our diagrams. Technically, these names would be entered into the globals frame and bound to their definitions.

```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```

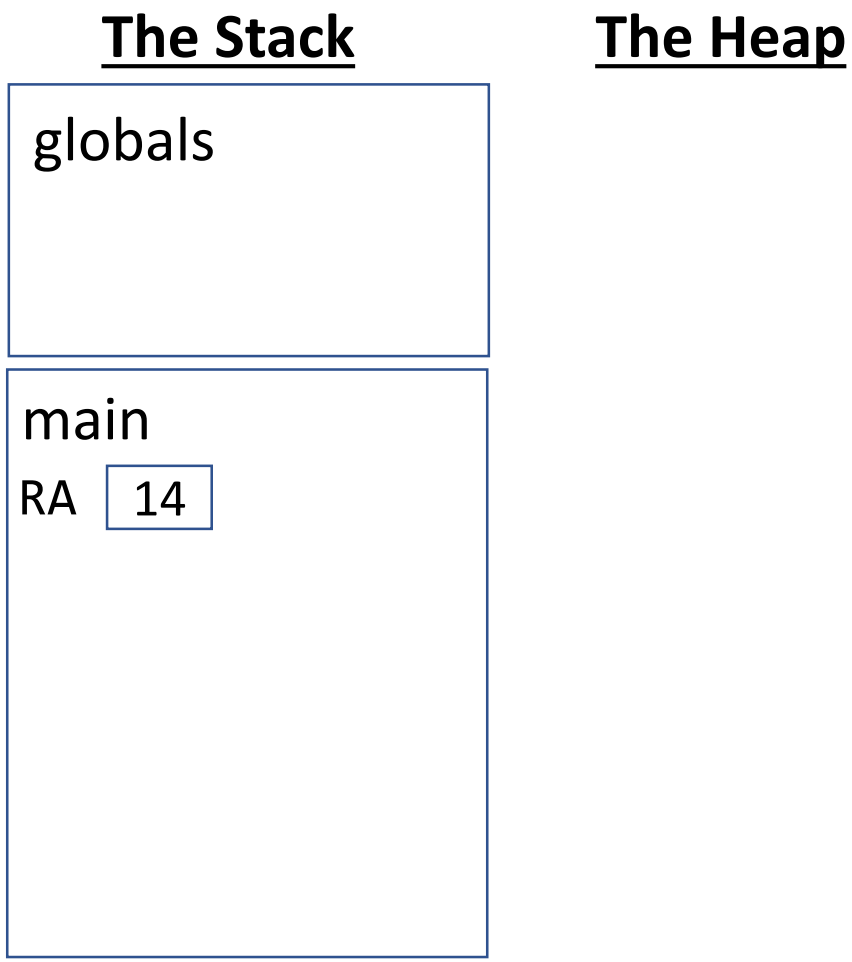
The Stack

globals

The Heap

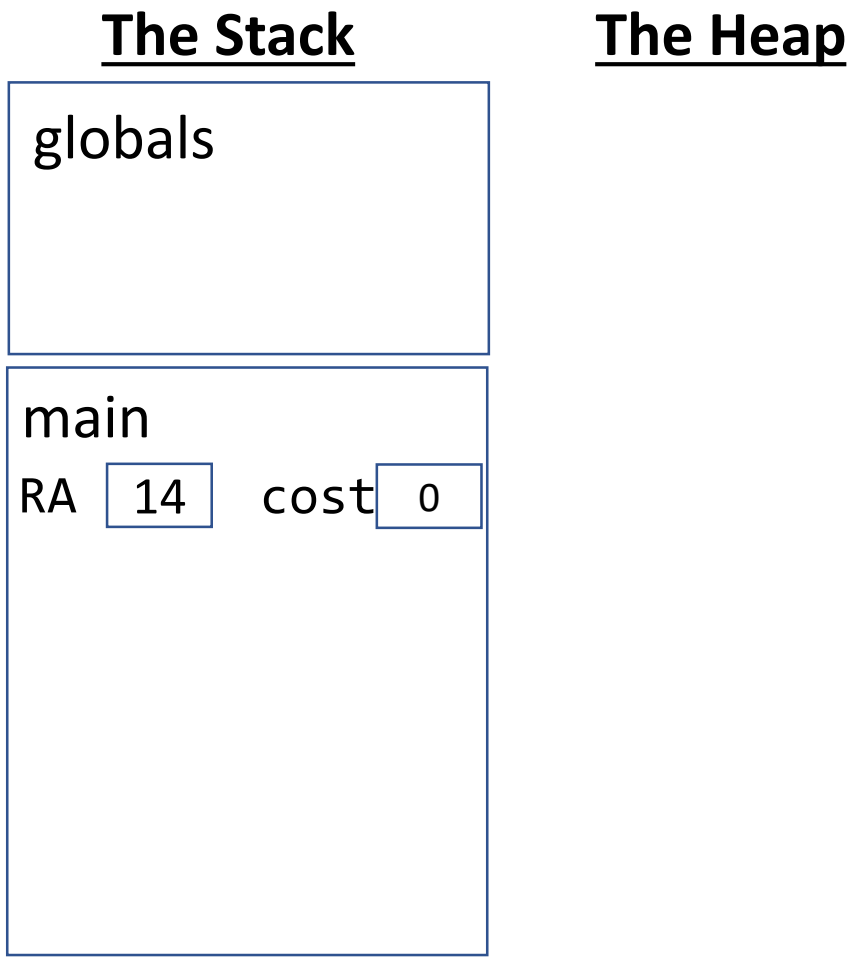
Function Call

```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```



Variable Declaration and Initialization

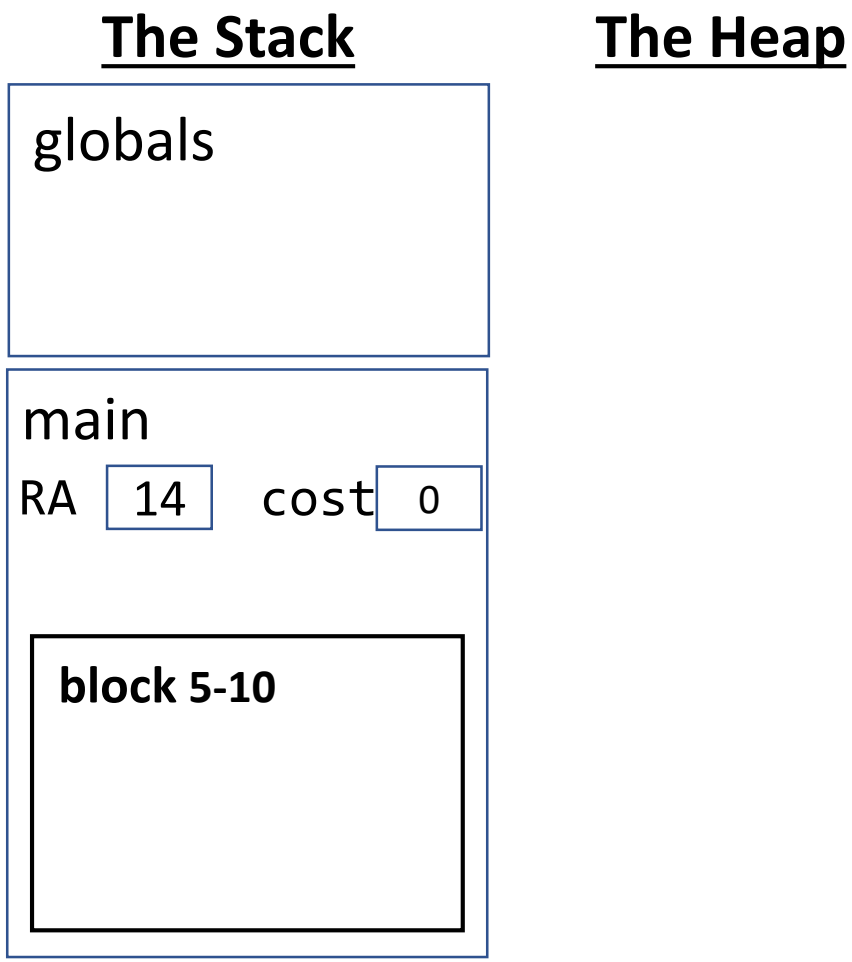
```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```



Block

When a block ***with variables declared inside of it*** is encountered, add a block entry with the start/end lines onto the current frame of the stack.

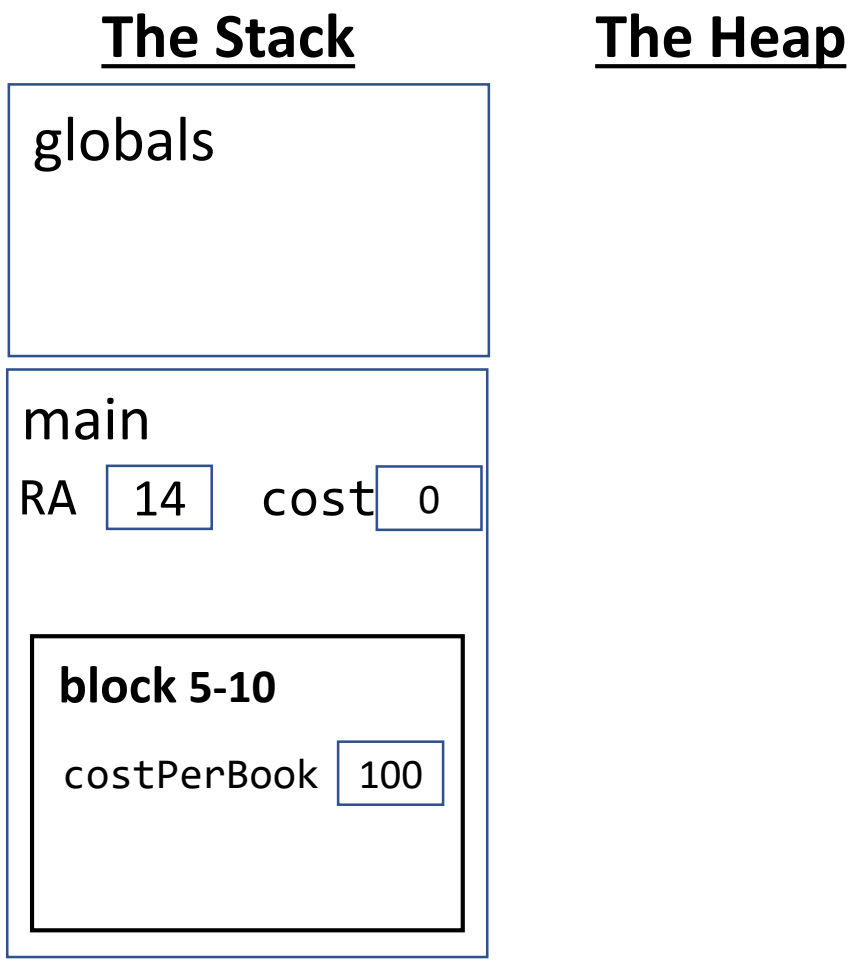
```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```



Variable Declaration

Notice the declared variable is established inside of the block's scope.

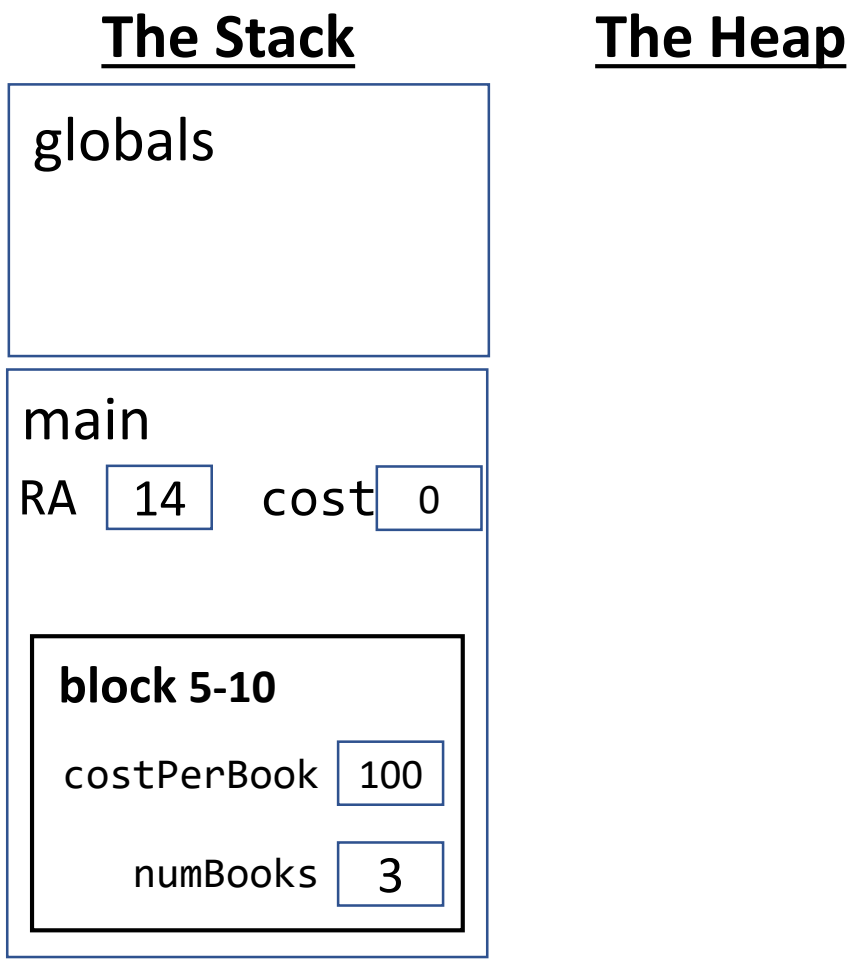
```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```



Variable Declaration

Notice the declared variable is established inside of the block's scope.

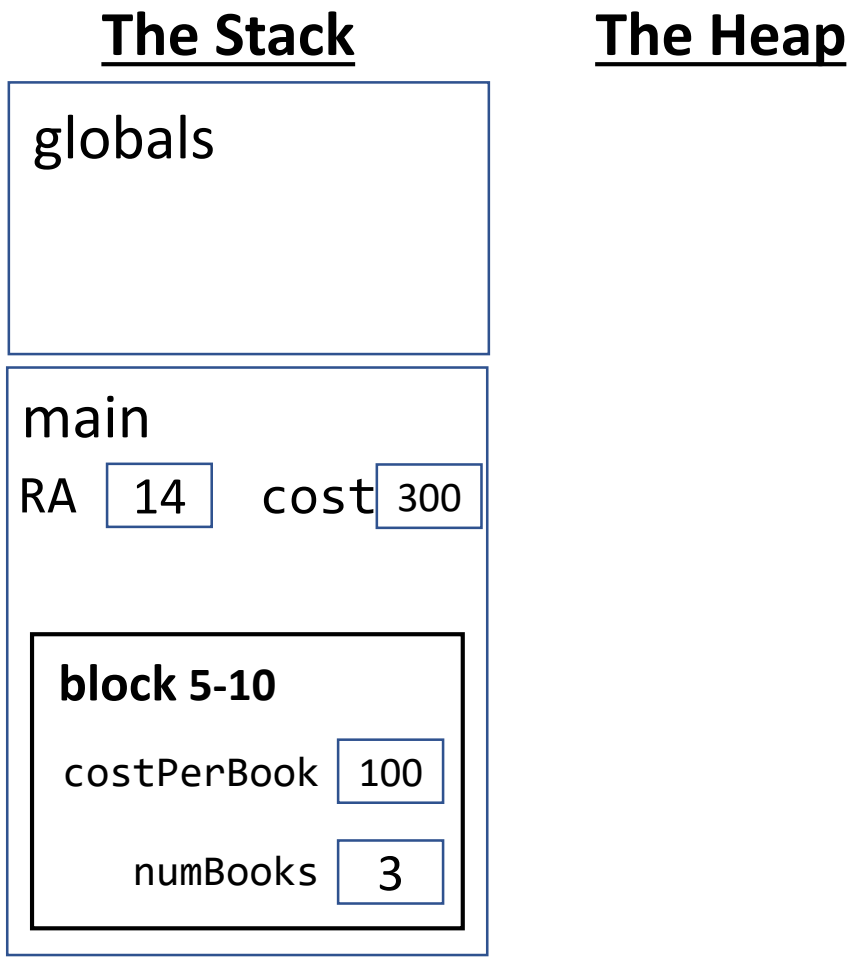
```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```



Name Resolution

Notice the cost variable is declared outside the block. To find a name inside of a block, work your way out toward the surrounding frame, then check globals.

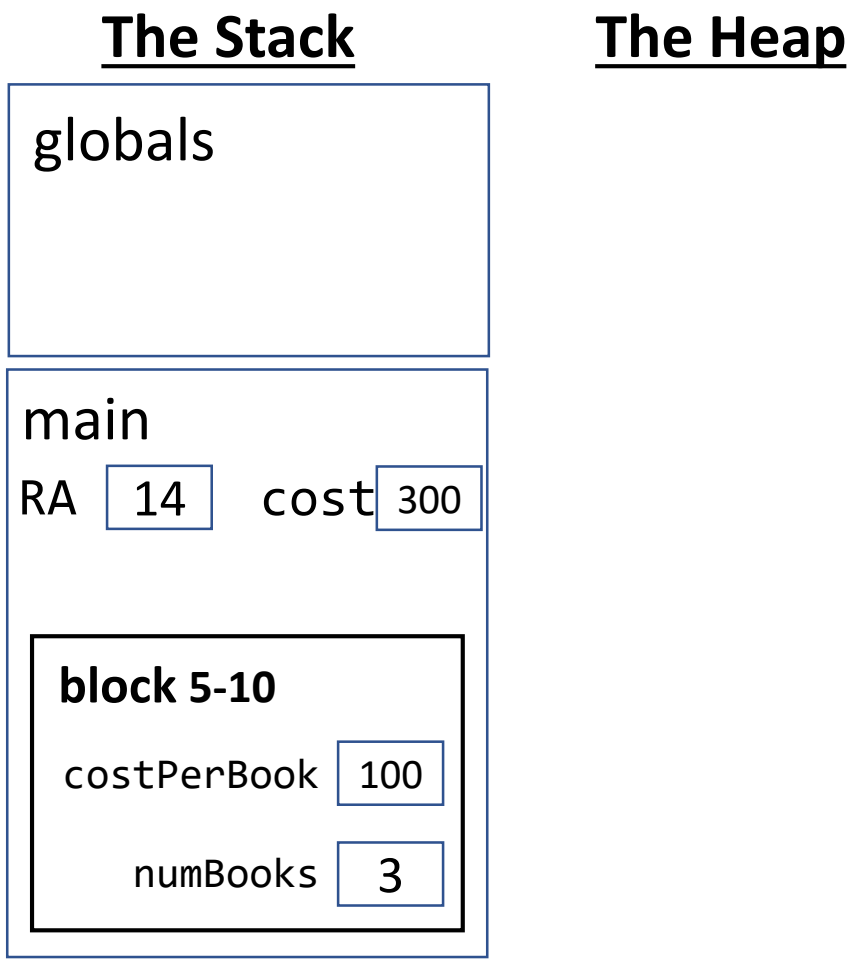
```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```



Name Resolution

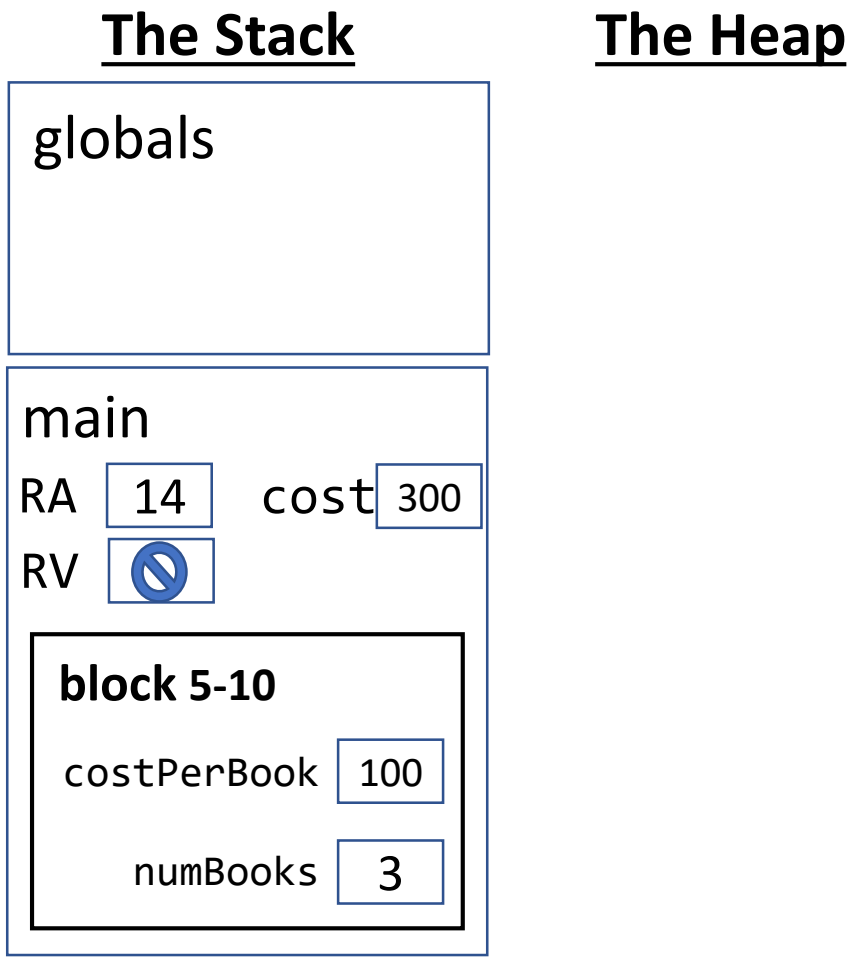
At this point in the program, only the cost variable is accessible in the main frame. Any attempt to print costPerBook or numBooks would error.

```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```



Return from **`main`** Function

```
1  import { print, promptNumber } from "introcs";
2
3  export let main = async () => {
4      let cost = 0;
5      {
6          // This is a block
7          let costPerBook = 100;
8          let numBooks = 3;
9          cost = costPerBook * numBooks;
10     }
11     print(cost);
12 };
13
14 main();
```



Name Resolution Rules

To find the space in memory *any* ***name*** (technically called an ***identifier***) is bound to in your program, follow these steps. The first rule to match wins.

1. If currently inside of a block: check the block first.

2. If currently inside of a nested block: check the blocks from inside-to-out.

3. Check the current frame (the last one added without an RV).

4. Check the Globals frame.

Variable Scoping Intuition

- Why have these specific rules?

- Block statements are like building blocks
  - Programs are constructed through many block statements

- Declaring a variable *in a block* prevents unrelated blocks from interference
  - Needing to worry about the existence of unrelated variables
  - Accidentally changing and mucking up another block's variables

- The rules help you avoid accidental logical errors