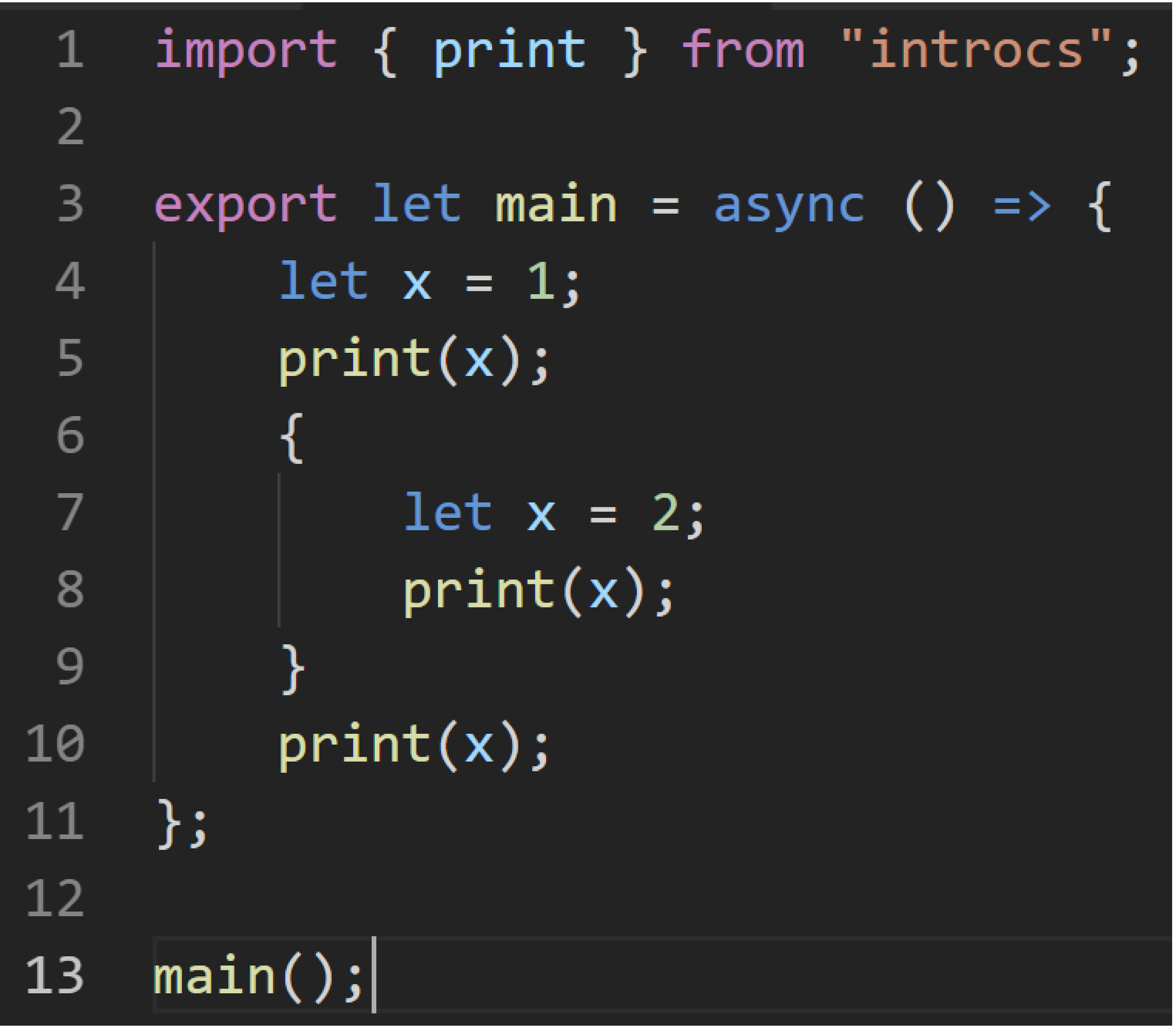
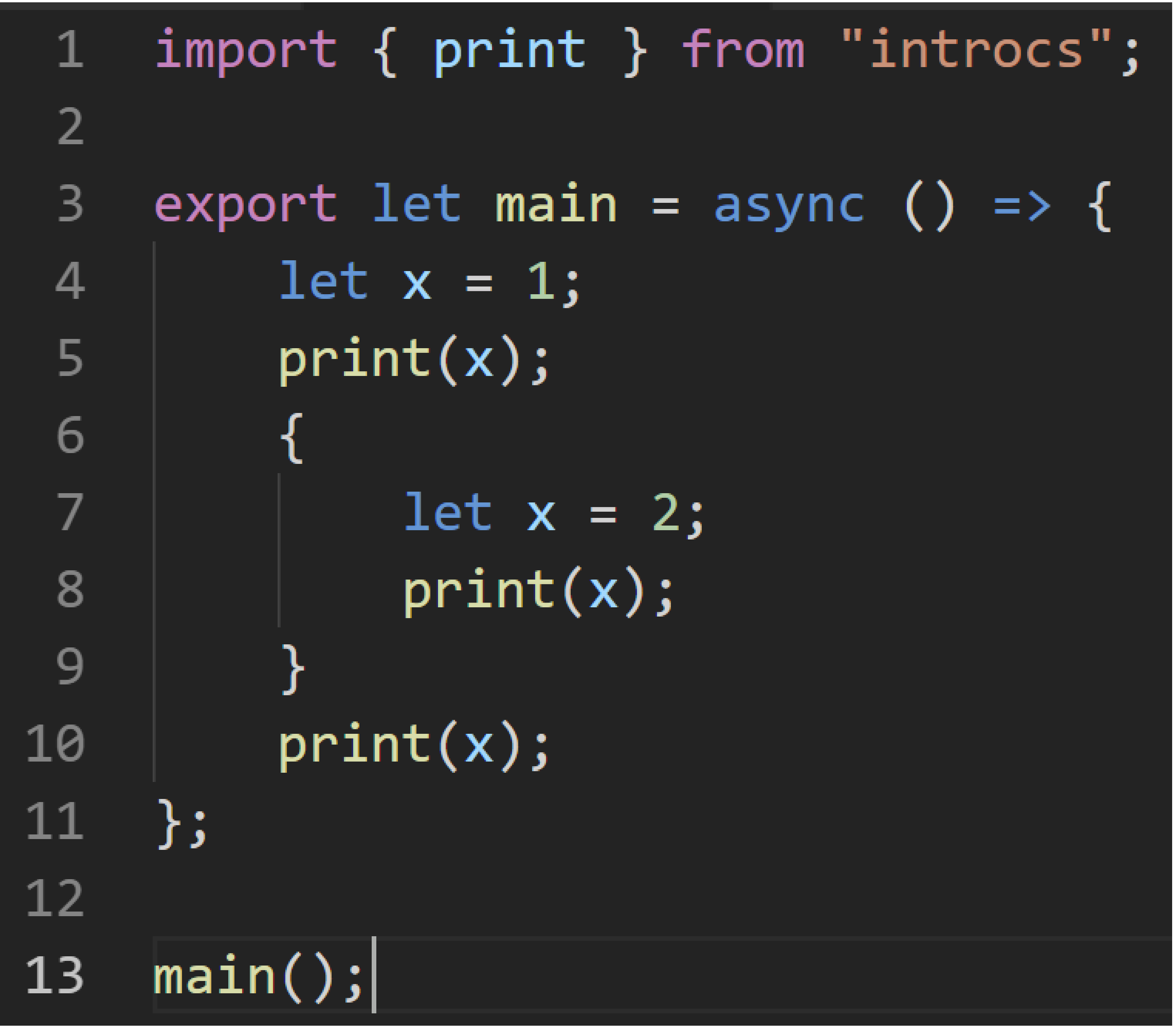
2D Array Applications and Scope Continued

Lecture 9 - Spring 2020

Challenge Question 1: What is the output?



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

export let main = async () => {
    let x = 1;
    print(x);
    {
        let x = 2;
        print(x);
    }
    print(x);
};

main();
```



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

export let main = async () => {
    let x = 1;
    print(x);
    {
        let x = 2;
        print(x);
    }
    print(x);
};

main();
```

Warning: Variable **Shadowing (1 / 2)**

- You cannot declare two variables with the same name in the same block.

```
{
   let x = 0;
   let x = 1; // ERROR! x declared prev in same block
}
```

- Why not?
  1. It helps you avoid accidents in longer blocks of code.
     - i.e. you forgot you declared a variable of the same name and used it for another purpose
  2. The *name* of this variable is already reserved in the current stack frame

Warning: Variable **Shadowing** (2 / 2)

- You *can* declare a variable of the same name in a nested, inner block. This is called "**variable shadowing**".
- The inner variable is a completely separate variable from the outer variable.
- When the processor returns to the outer block, **x** refers to the original **x** variable and its contents are unchanged.

```
let x = 0;
print(x); // Prints 0
{
  let x = 10;
  print(x); // Prints 10
}
print(x); // Prints 0
```

Variable shadowing is confusing and should be avoided by choosing meaningful variable names.

Conway's Game of Life

- A simple "simulation" involving a 2D grid of "**cells"**
    - First implemented in 1970 by John Conway

- A cell can either be **"alive"** (value is 1) or **"dead"** (value is 0)

- At each "step" of the simulation, 4 simple rules are applied to every cell to determine whether it is alive or dead at the next step
    - As these rules are applied, the outcome is assigned to a new 2D grid of cells not modifying the current step. So it's as if these rules are applied instantaneously.

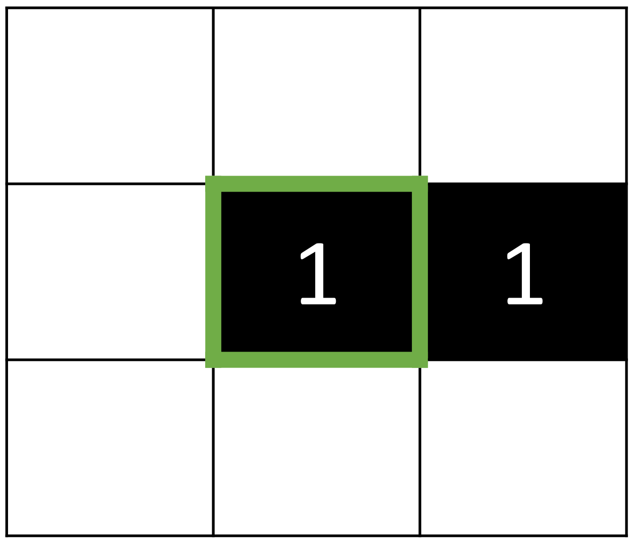
- Complex, emergent behaviors and systems arise from these simple rules.

Conway's Game of Life - Rules

- There are 4 rules, covered in the following 4 slides

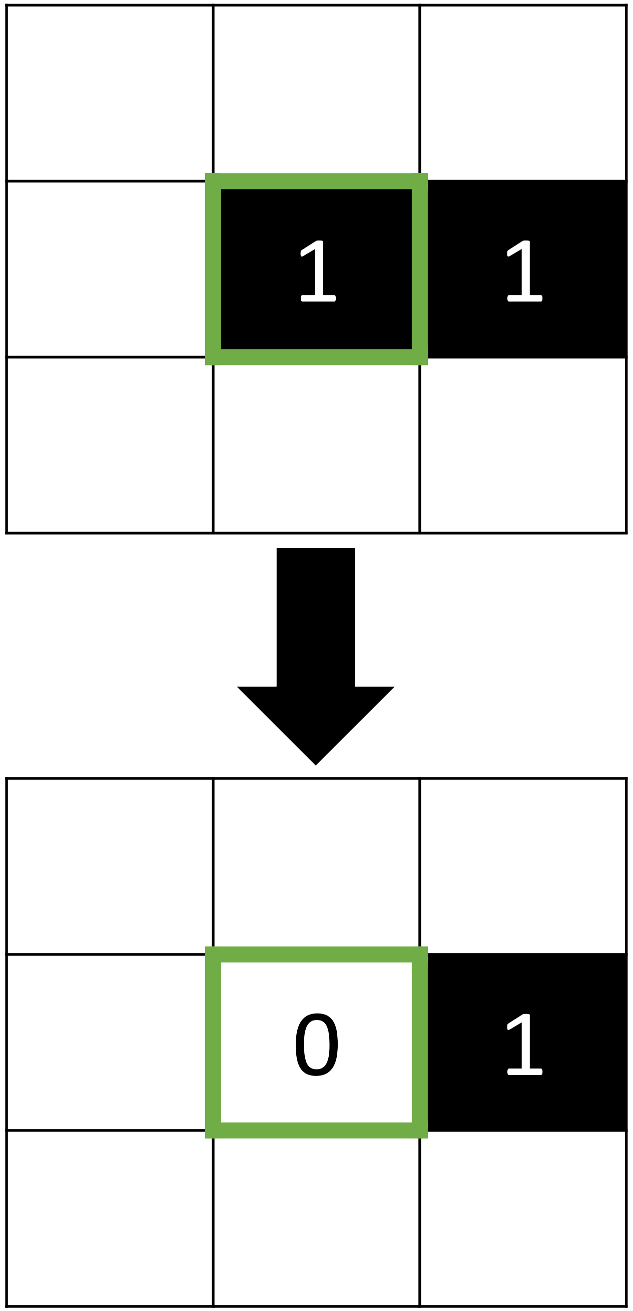
- Note that each example gives the current step and the next step for *only the cell outlined in green.*

- At each step, the same rules will also be applied to all surrounding cells, too, but we will not illustrate this in slides.



Conway's Game of Life - Rules

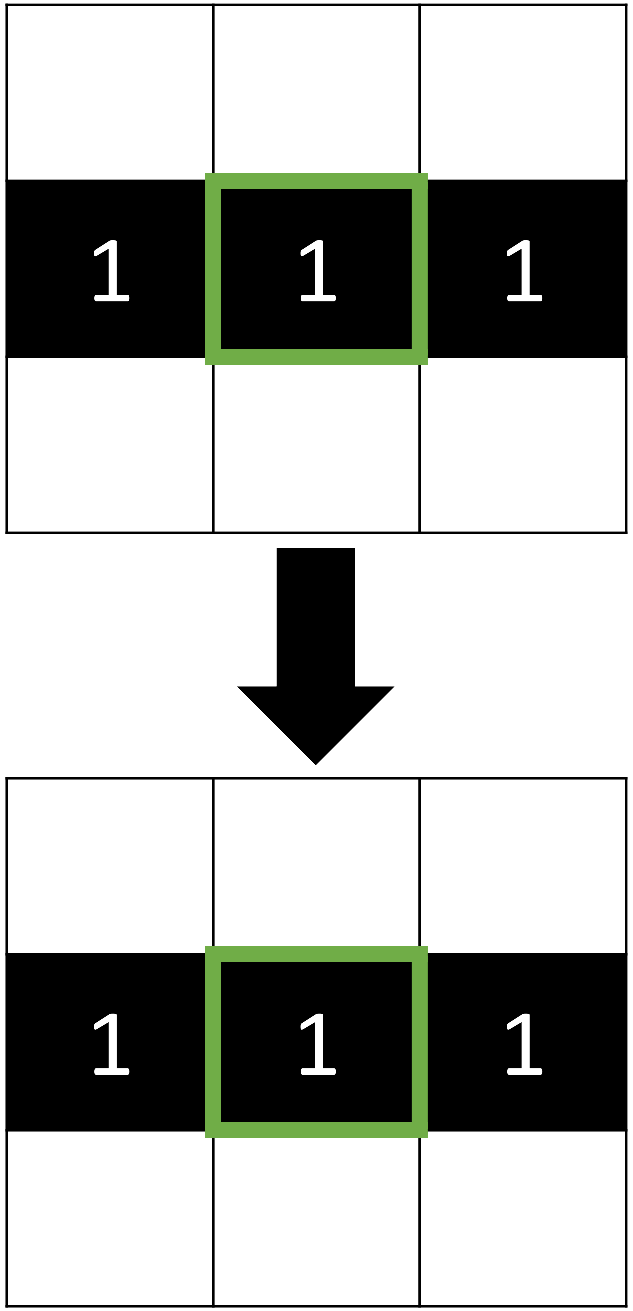
1. **Underpopulation: A live cell with fewer than 2 live neighbors dies.**



Conway's Game of Life - Rules

1. Underpopulation: A live cell with fewer than 2 live neighbors dies.

2. **Stasis: A live cell with 2 or 3 live neighbors survives.**

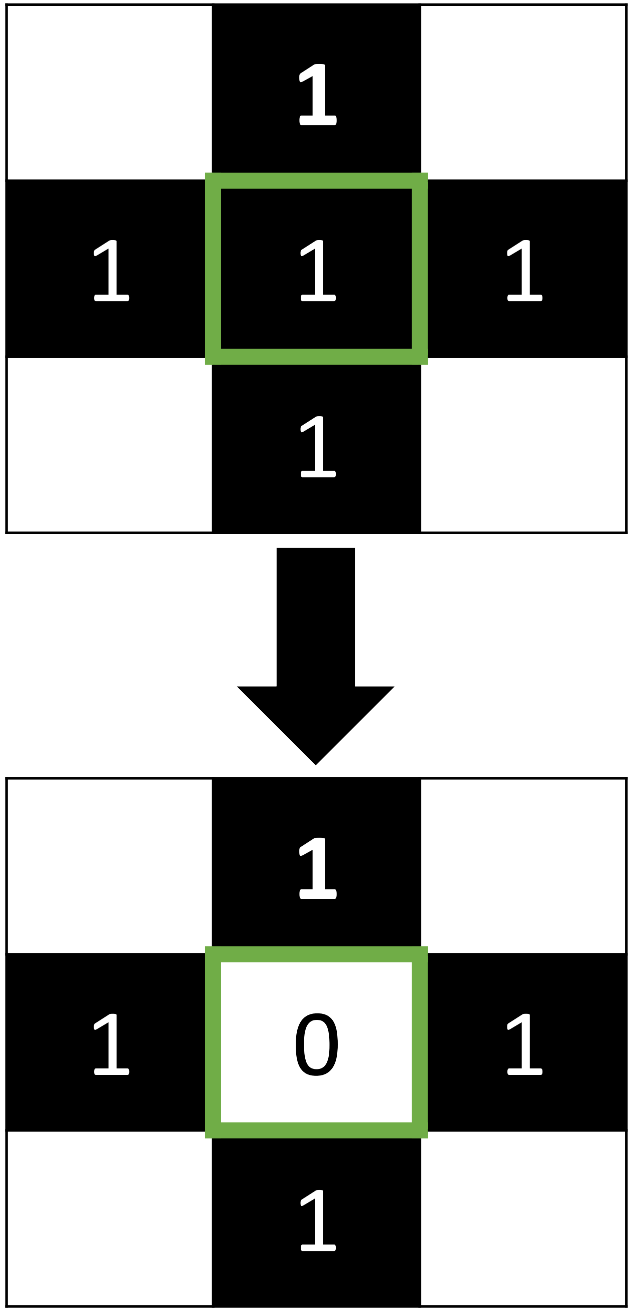


Conway's Game of Life - Rules

1. Underpopulation: A live cell with fewer than 2 live neighbors dies.

2. Stasis: A live cell with 2 or 3 live neighbors survives.

3. **Overpopulation: A live cell with more than 3 live neighbors dies.**



	1	
1	1	1
	1	

	1	
1	0	1
	1	

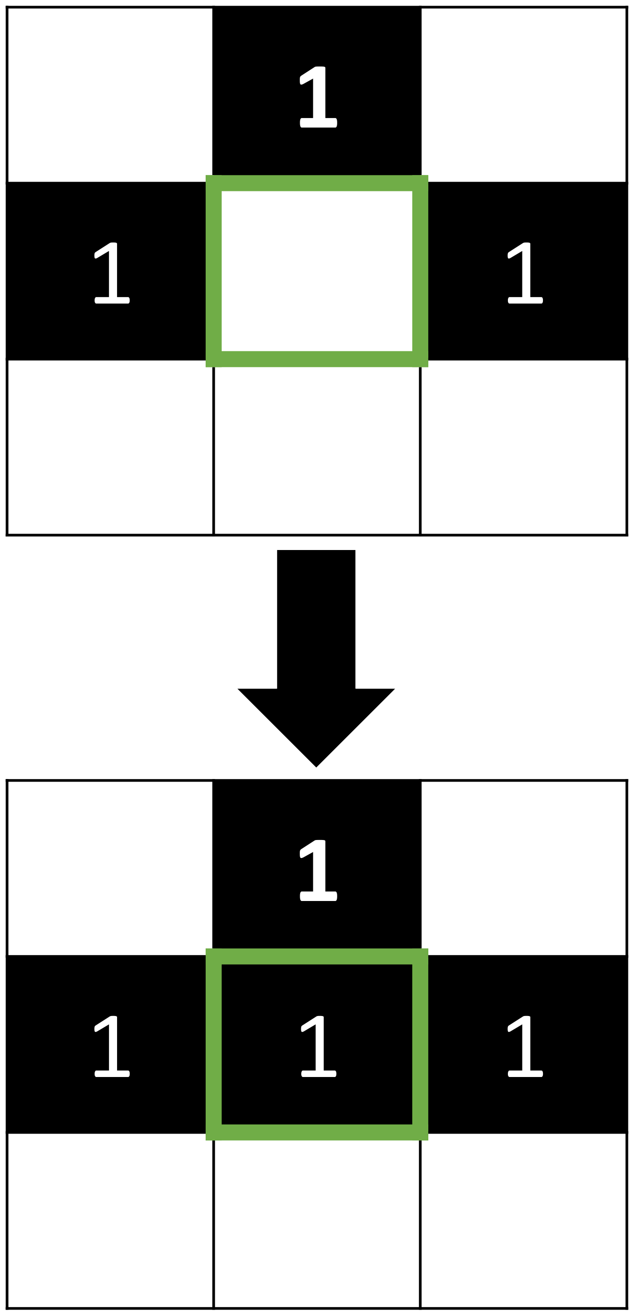
Conway's Game of Life - Rules

1. Underpopulation: A live cell with fewer than 2 live neighbors dies.

2. Stasis: A live cell with 2 or 3 live neighbors survives.

3. Overpopulation: A live cell with more than 3 live neighbors dies.

4. **Reproduction: Any dead cell with 3 live neighbors comes to life.**



Stencil Code Organization

- Today we will only focus on the *model* of Conway's Game of Life and write our code in gol-model.ts

- The "model" of a program refers to its essential data and logic

- The stencil code in today's lecture also contains the code for:
  1. The HTML document containing the user interface elements (game-of-life.html)
  2. The CSS style rules for the table of cells (styles.css)
  3. The visual representation of the grid (gol-view.ts)
  4. The event handling code for the buttons (gol-controller.ts)
  5. The main function that starts the program (game-of-life-script.ts)

- In COMP401 you'll learn about organizing your code using Model-View-Controller

Strategy

1. Write *function* to **"step"** through all cells and apply "rules" function to determine the next state of a single cell

2. Write function to **count the number of live neighbors** around a cell

3. Improve the **game logic** function to apply Game of Life rules

4. Improve a *function* to determine whether a cell is live or not
    - So that the game "wraps around" the edges

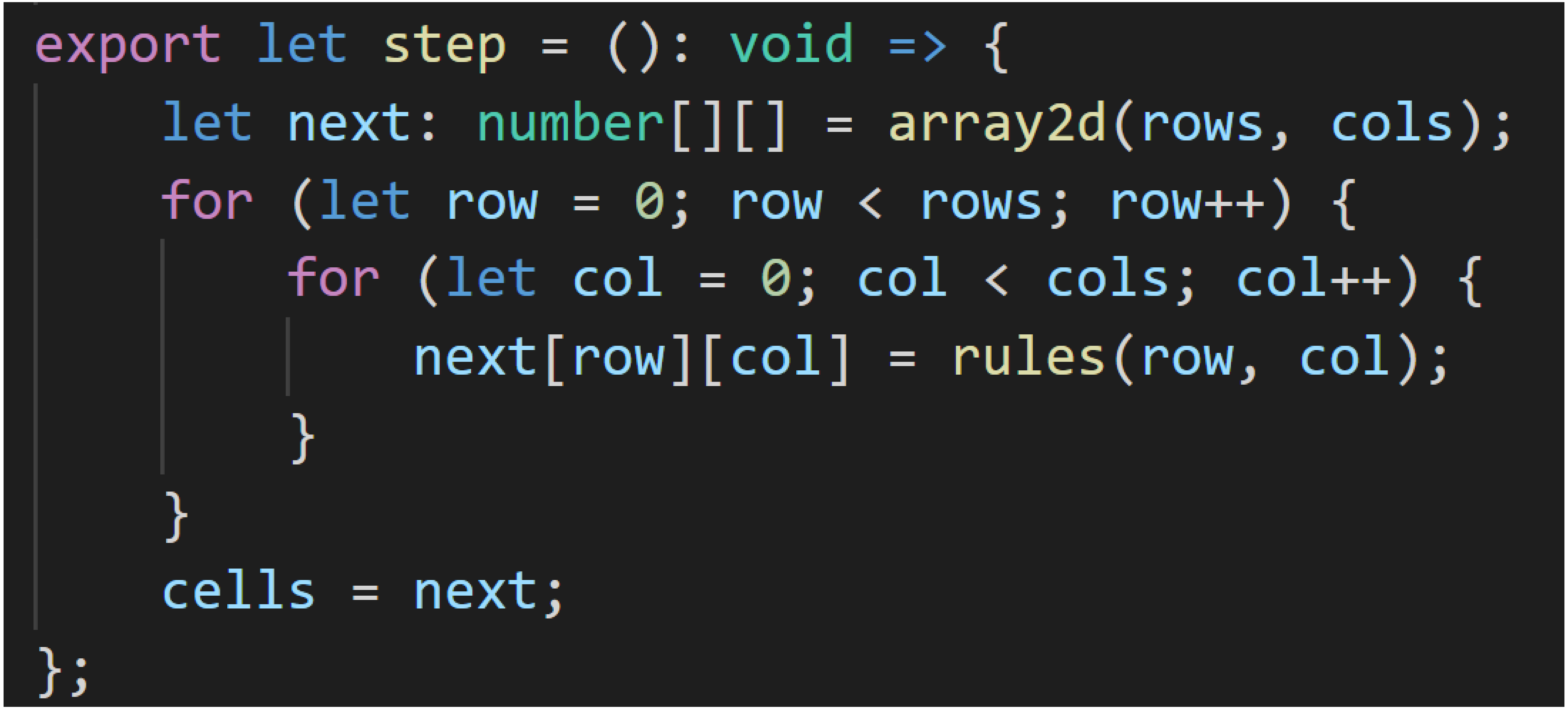
step

- Let's write a function that sets up an array to contain ***the next generation*** of cells.

- It will **loop through the current generation of cells** in a **nested loop** and **call the "rules" function** to determine the **next state of the cell**.

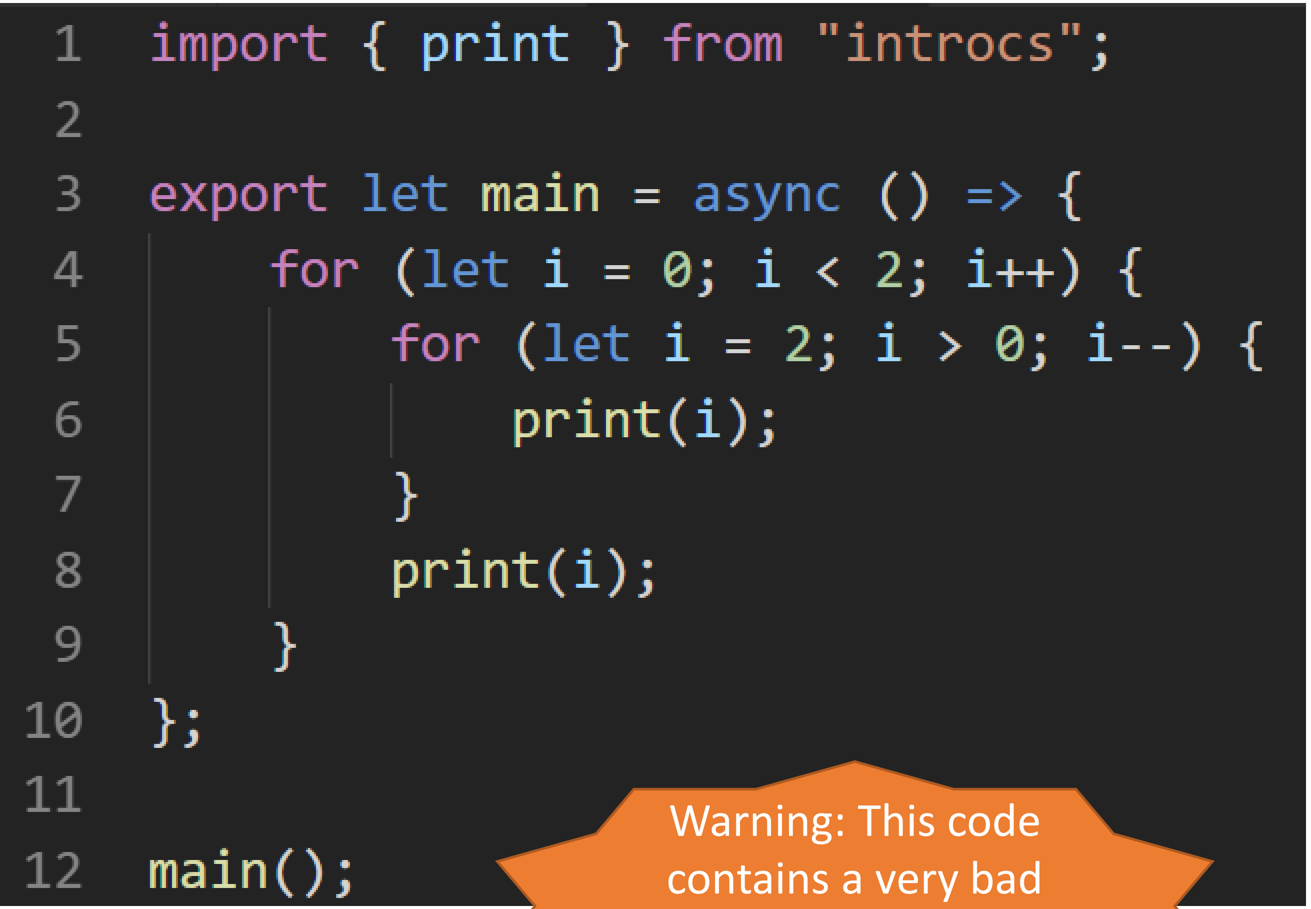
- The stencil code's controller is already calling the "step" function every time the step button is pressed.

step



```
export let step = (): void => {
    let next: number[][] = array2d(rows, cols);
    for (let row = 0; row < rows; row++) {
        for (let col = 0; col < cols; col++) {
            next[row][col] = rules(row, col);
        }
    }
    cells = next;
};
```

CQ2 – What is the output?

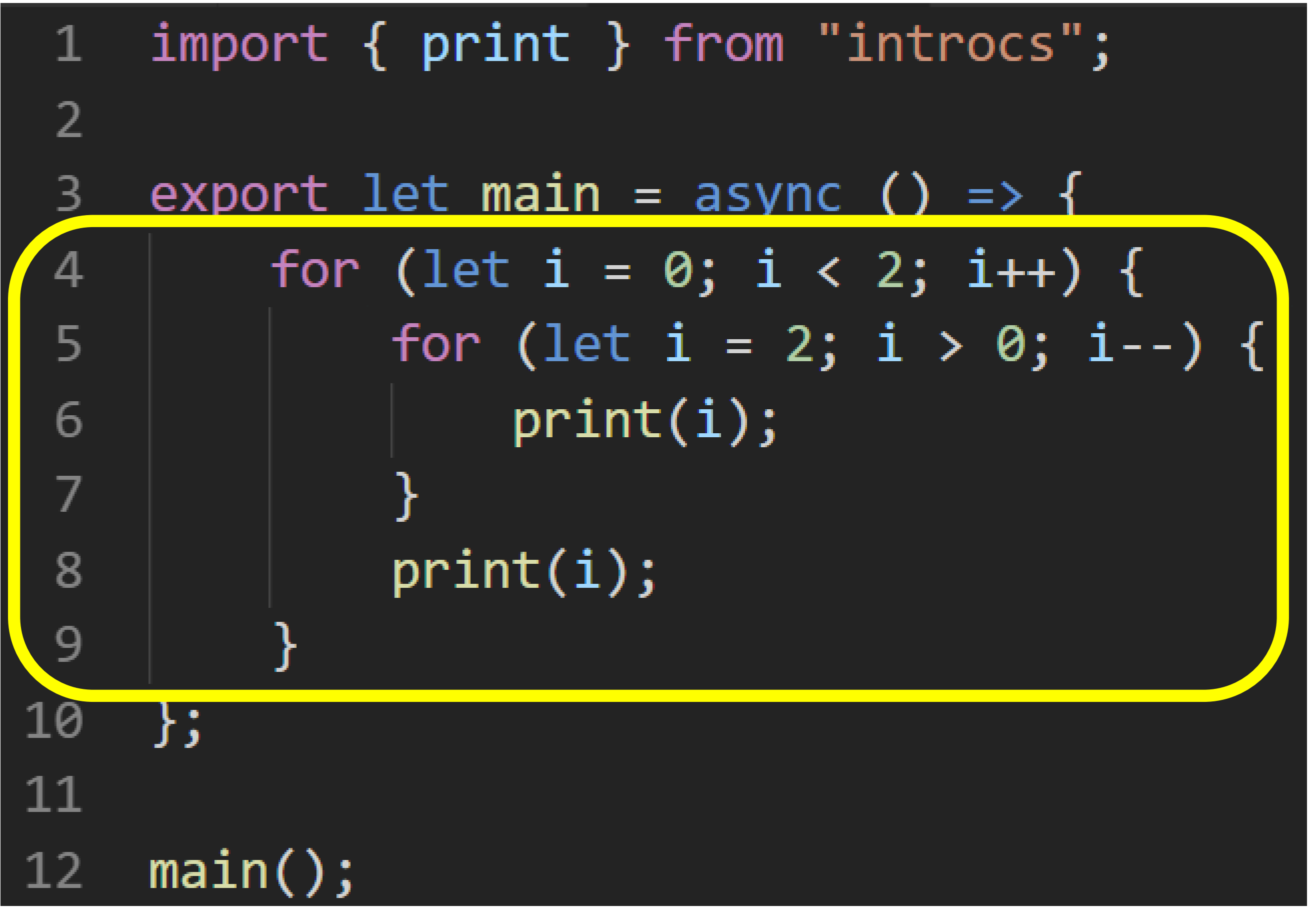


```
import { print } from "introcs";

export let main = async () => {
    for (let i = 0; i < 2; i++) {
        for (let i = 2; i > 0; i--) {
            print(i);
        }
        print(i);
    }
};

main();
```

Warning: This code contains a very bad practice! Used only for illustration.

Fast-Forward

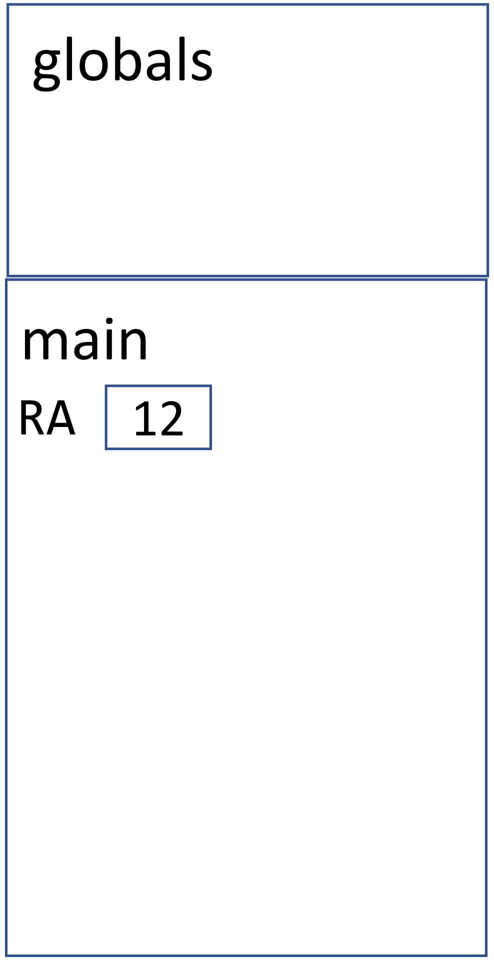
Imagine execution has reached the point of encountering this for loop.



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

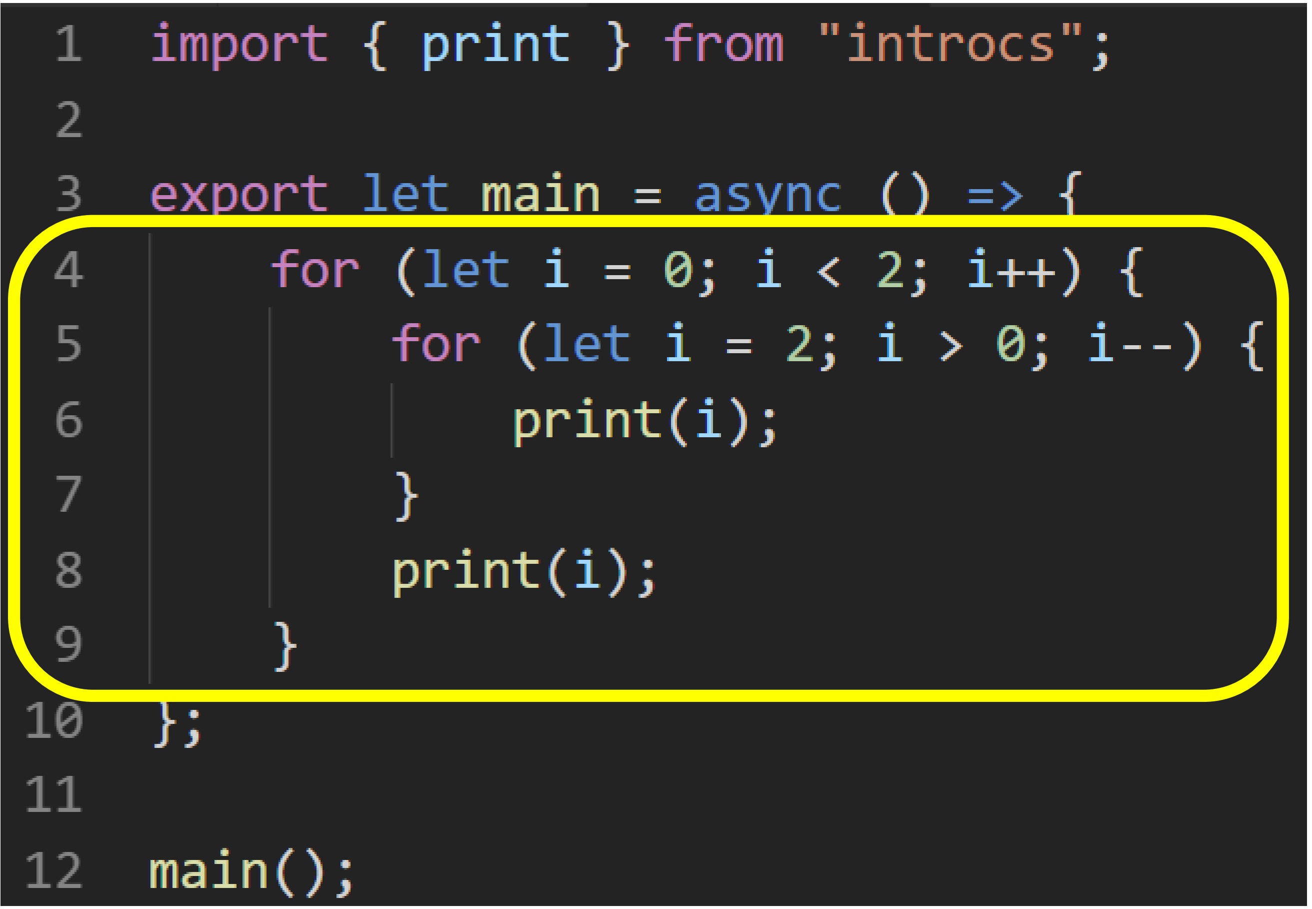
The Stack

The Heap

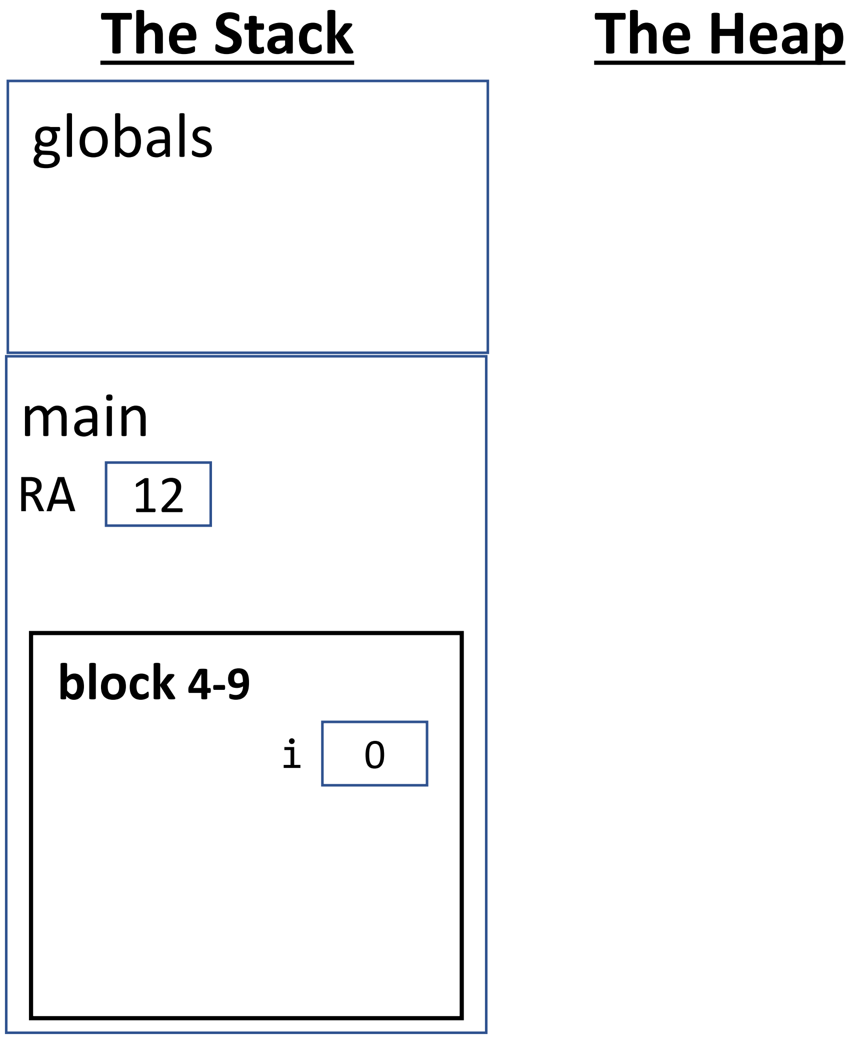


For Loop Block

A block is established to hold the counter variable of the *for* loop.



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```



The Stack

The Heap

globals

main

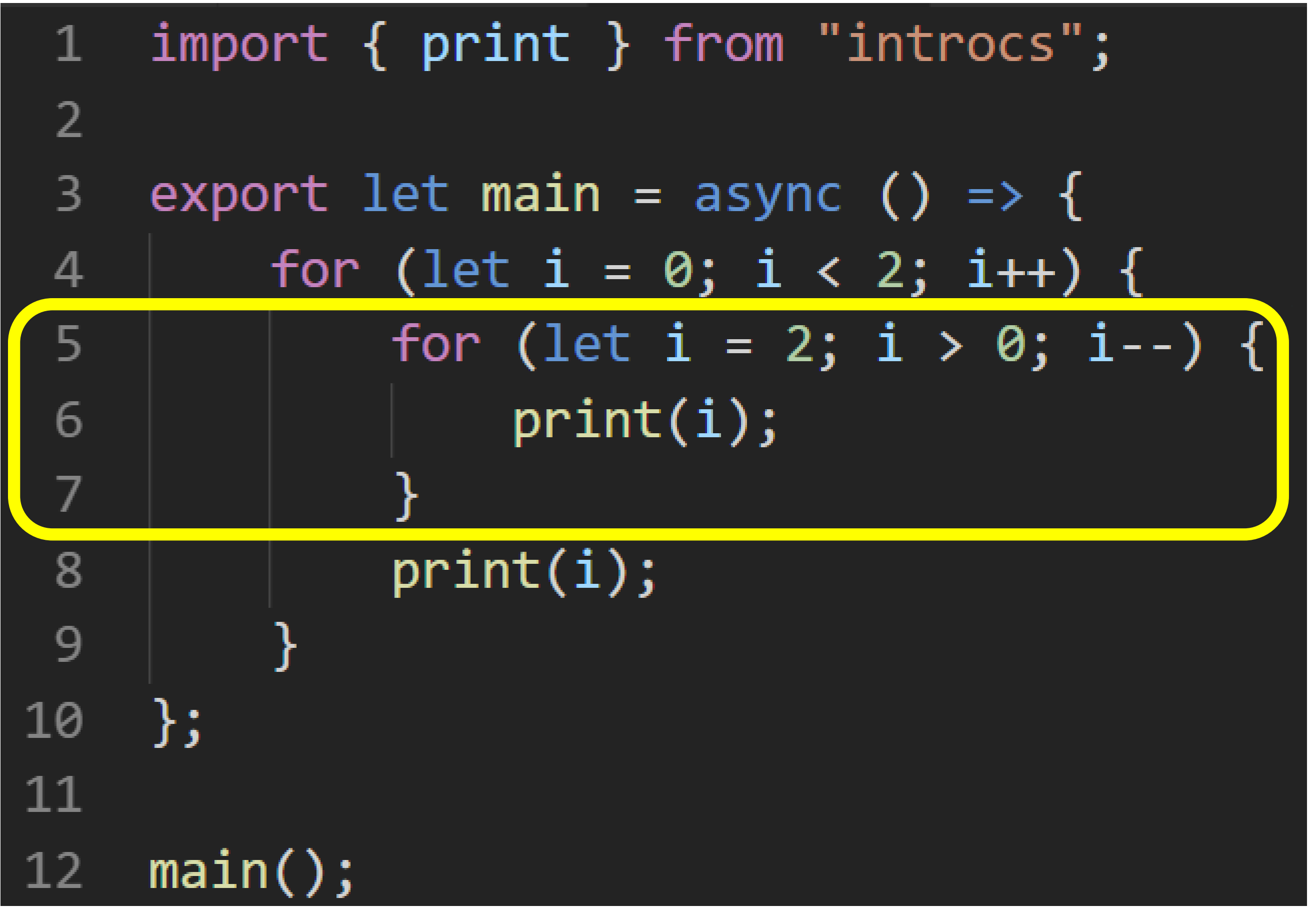
RA 12

block 4-9

i 0

For Loop Block

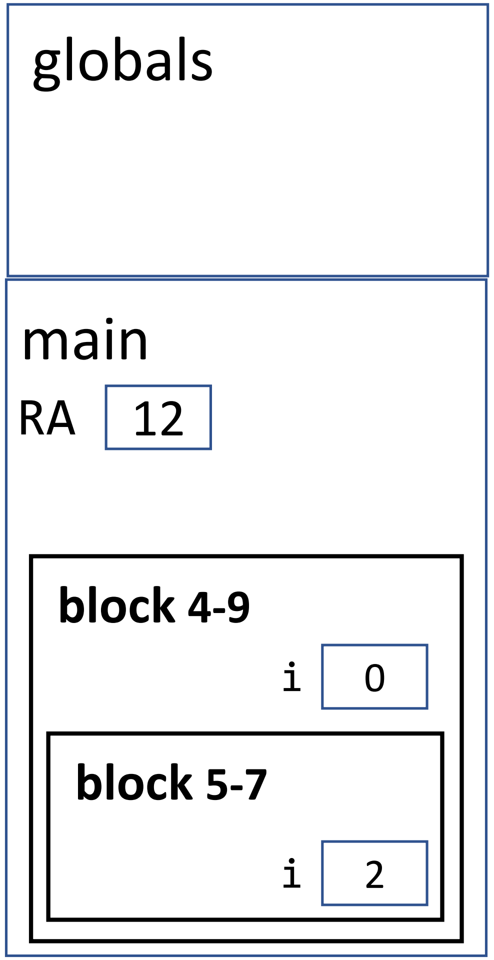
Another block is established inside the current block to hold the counter variable of the nested *for* loop. Notice since each loop has its own block they're separate.



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

The Stack

The Heap



globals

main

RA 12

block 4-9

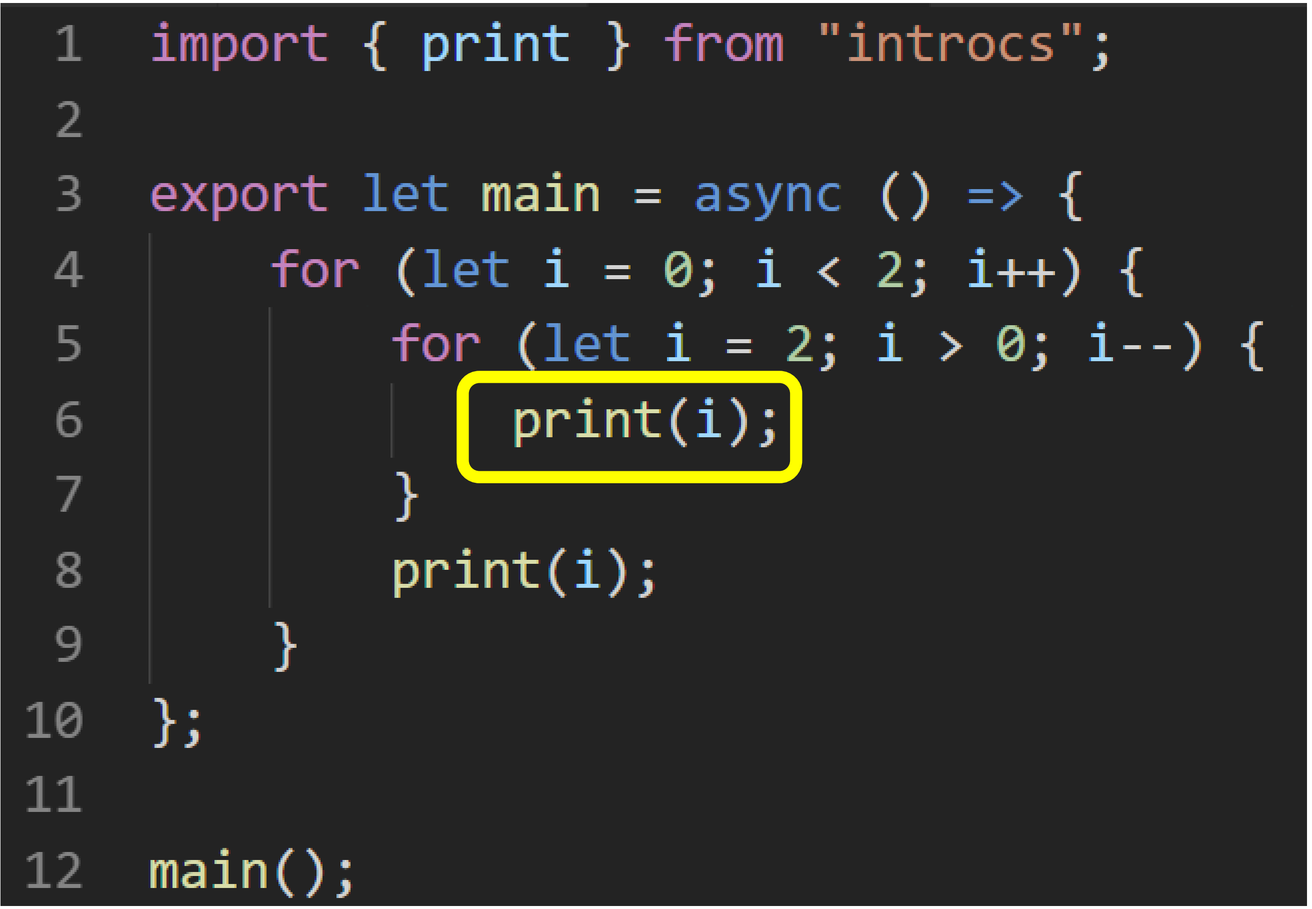
i 0

block 5-7

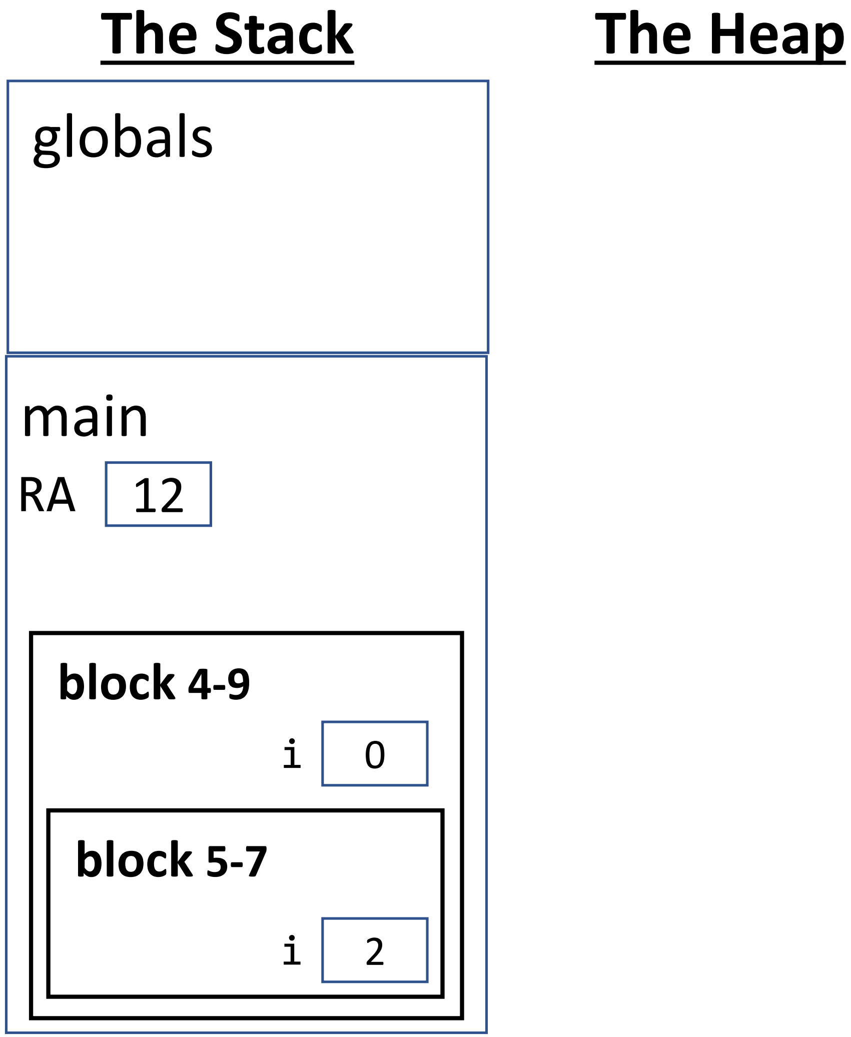
i 2

Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.

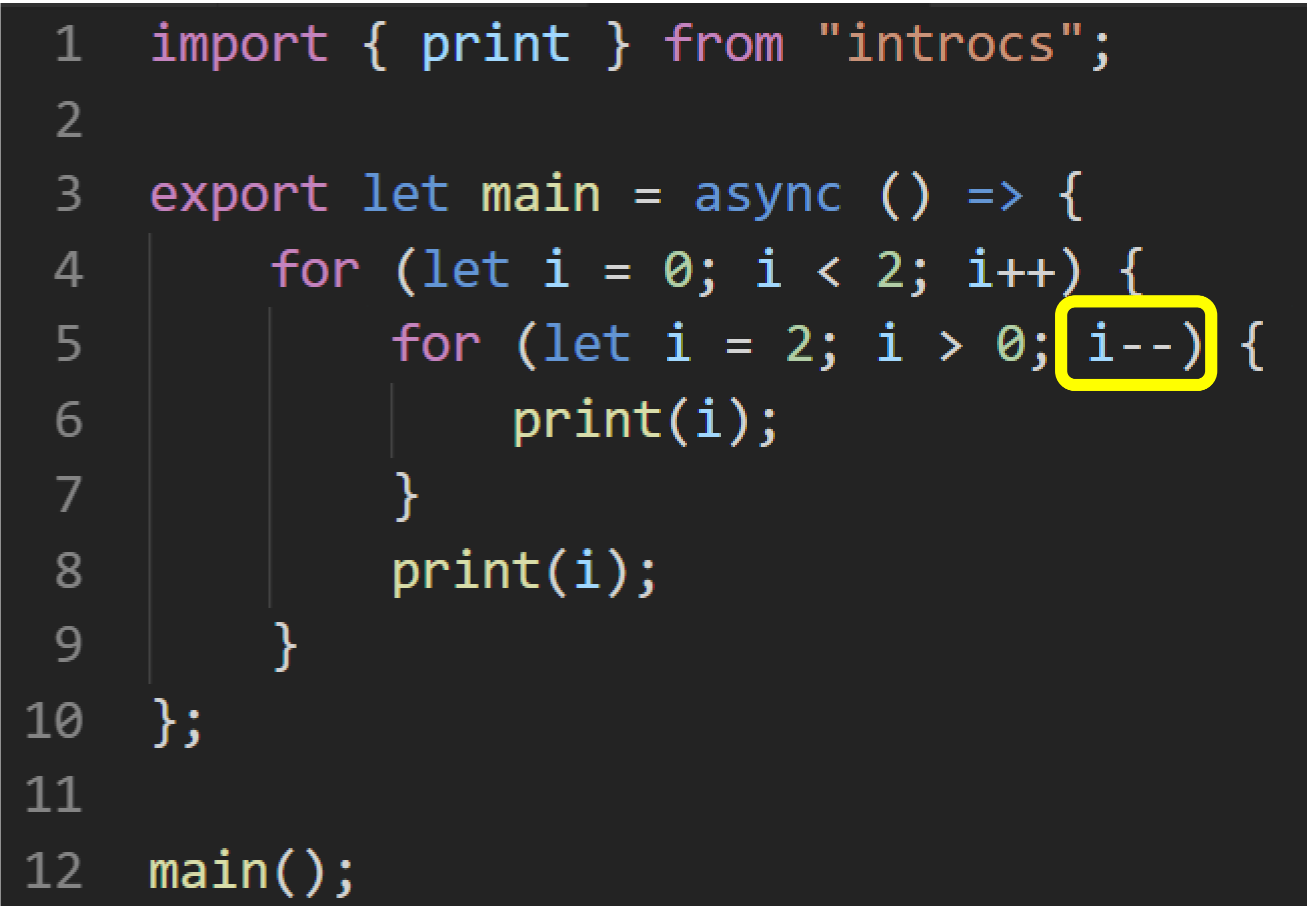


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

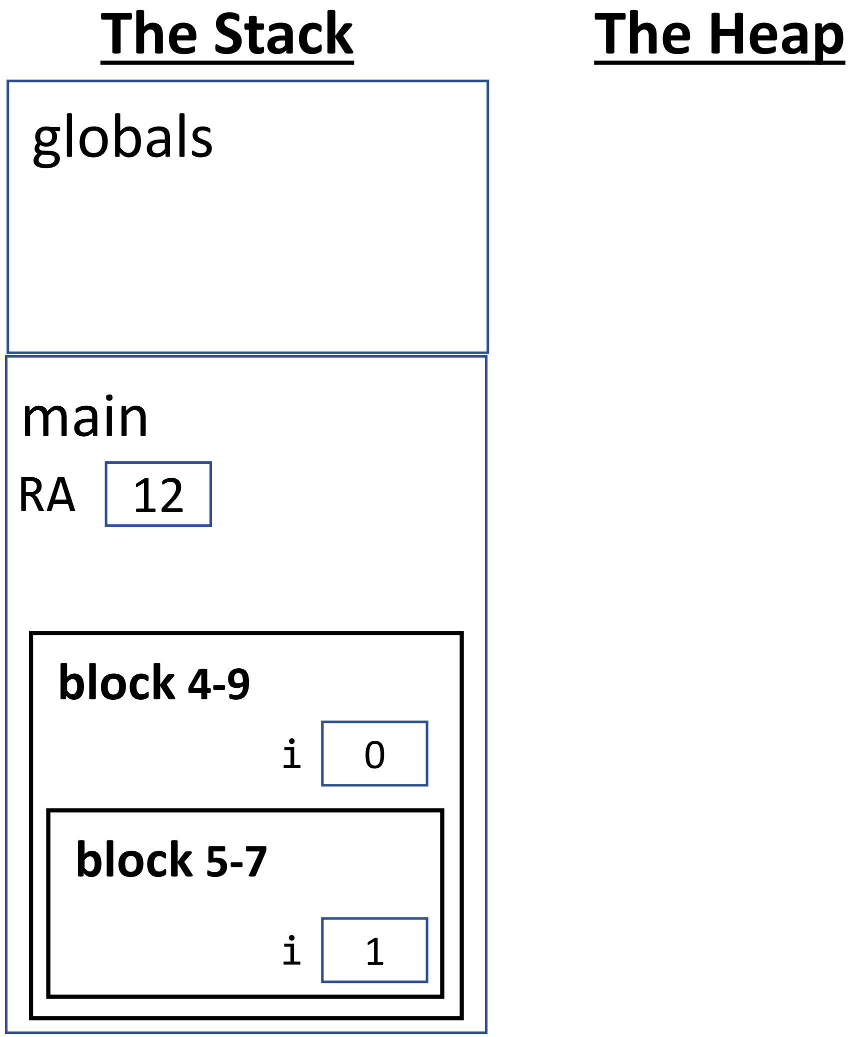


Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.



```
 1  import { print } from "introcs";
 2
 3  export let main = async () => {
 4      for (let i = 0; i < 2; i++) {
 5          for (let i = 2; i > 0; i--) {
 6              print(i);
 7          }
 8          print(i);
 9      }
10  };
11
12  main();
```



The Stack

The Heap

globals

main

RA 12

block 4-9

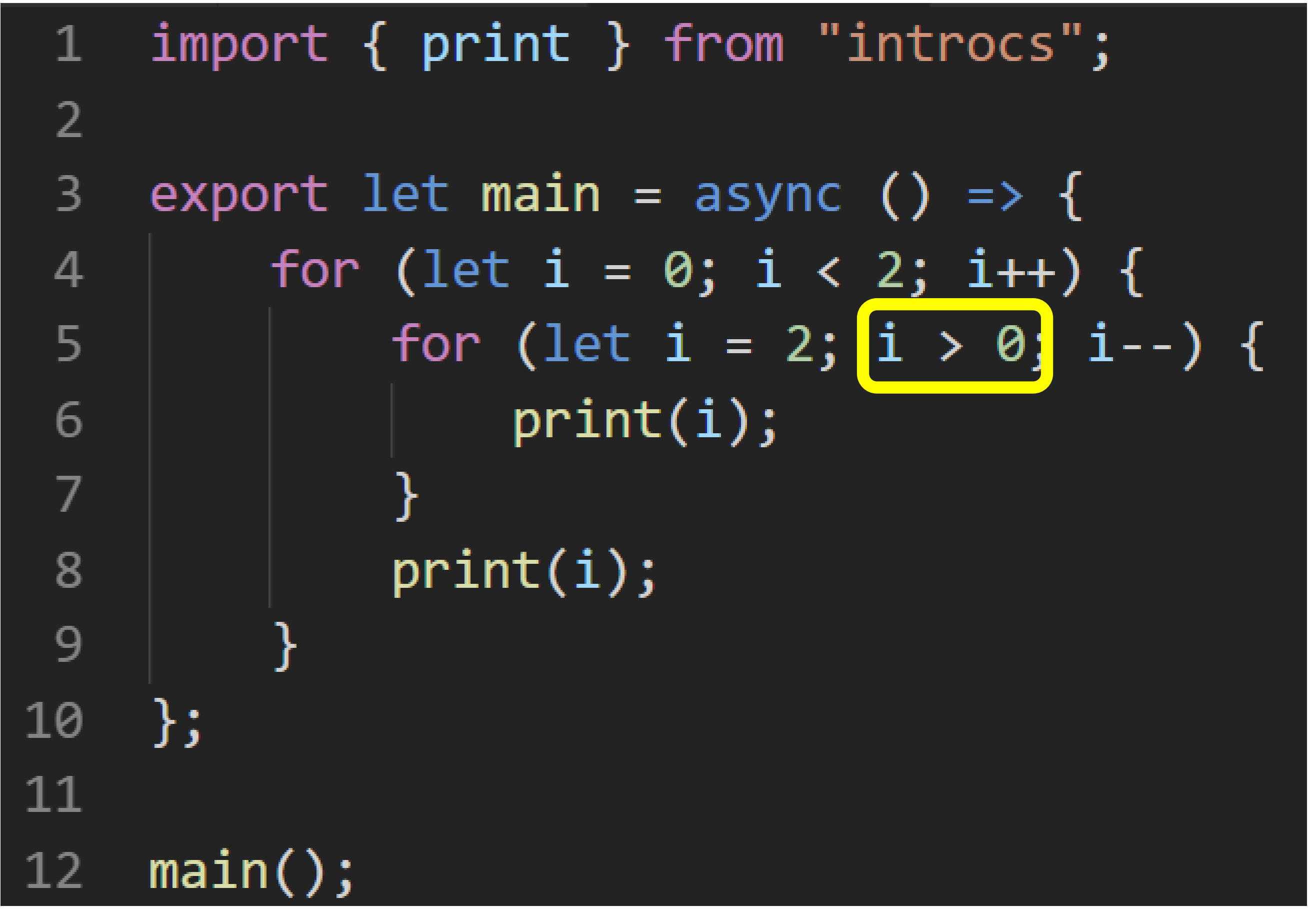
i 0

block 5-7

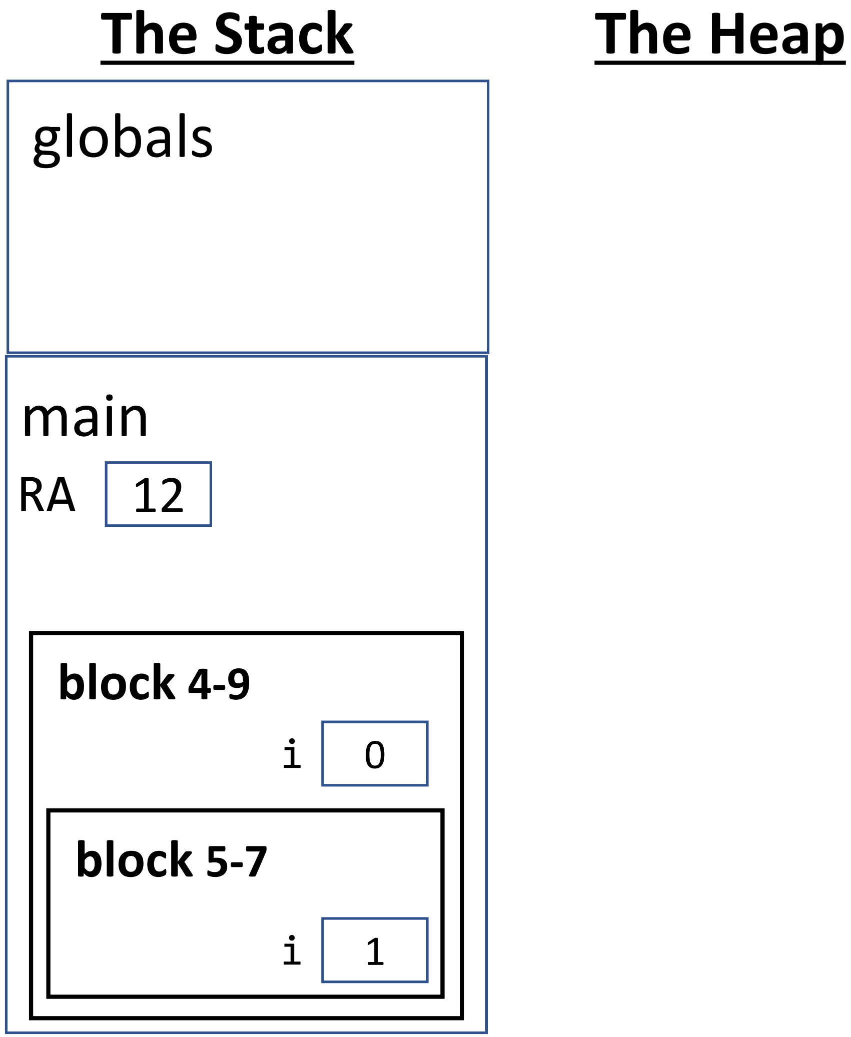
i 1

Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.

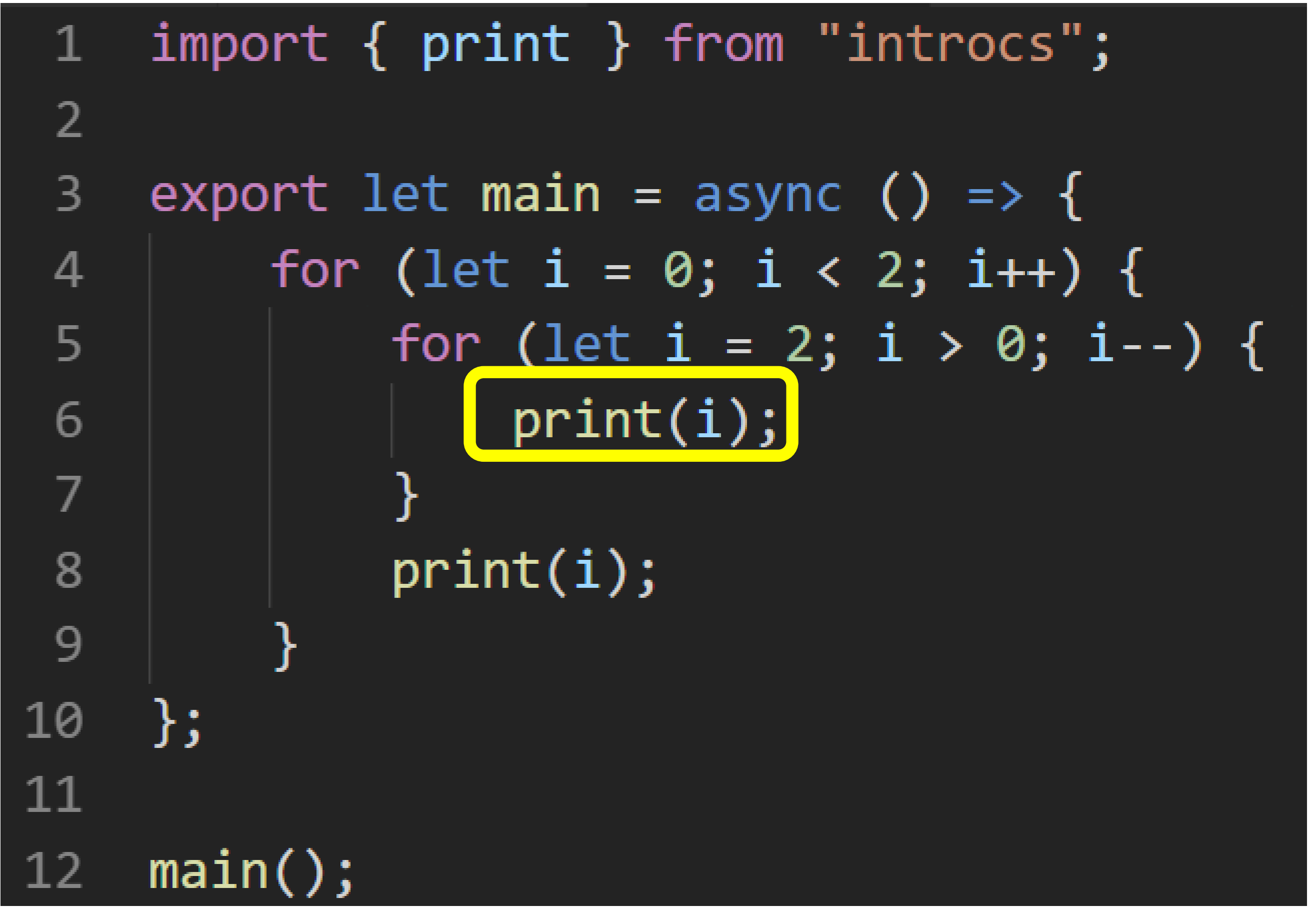


```
 1  import { print } from "introcs";
 2
 3  export let main = async () => {
 4      for (let i = 0; i < 2; i++) {
 5          for (let i = 2; i > 0; i--) {
 6              print(i);
 7          }
 8          print(i);
 9      }
10  };
11
12  main();
```

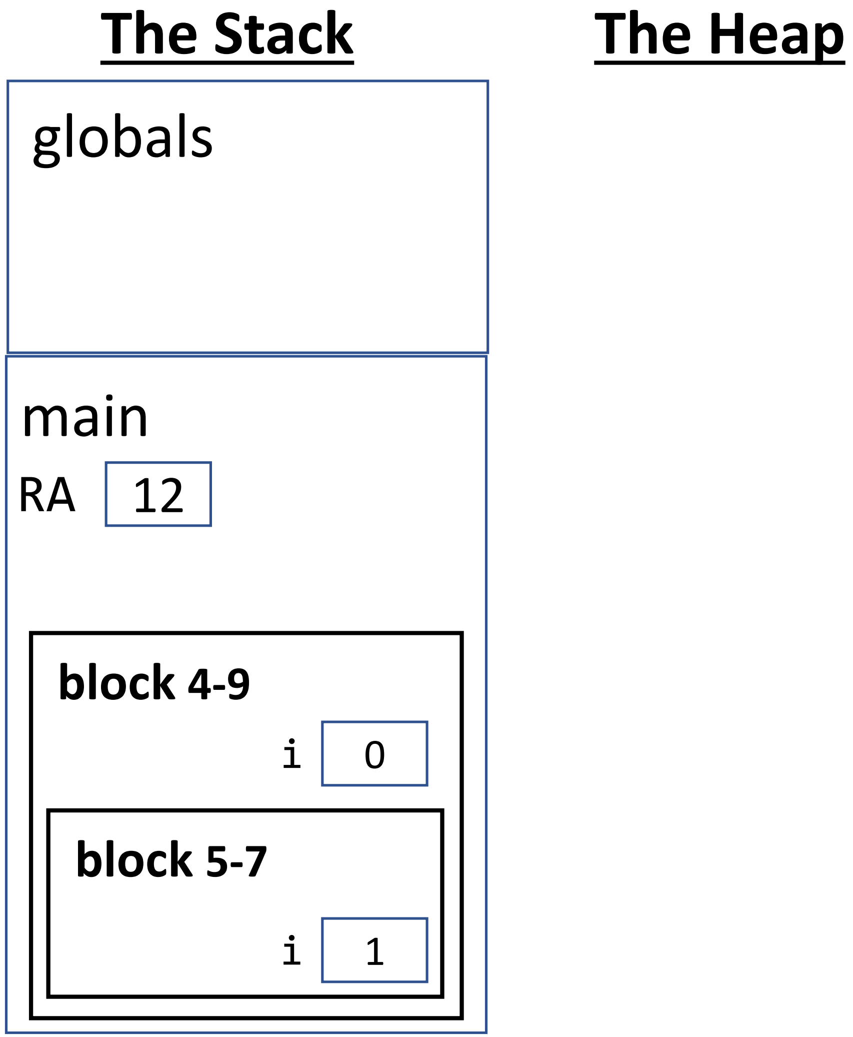


Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```



The Stack

The Heap

globals

main

RA 12

block 4-9

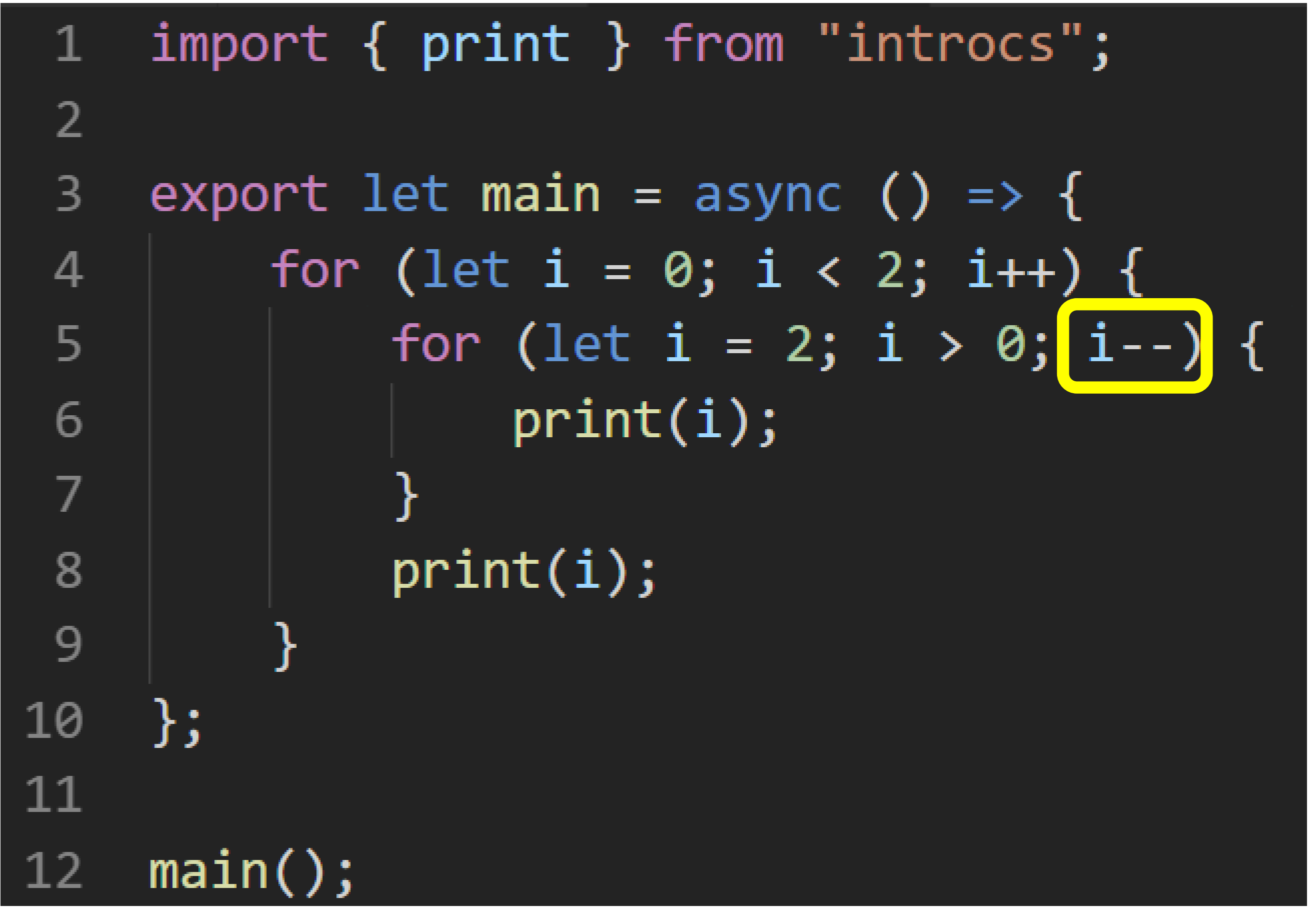
i 0

block 5-7

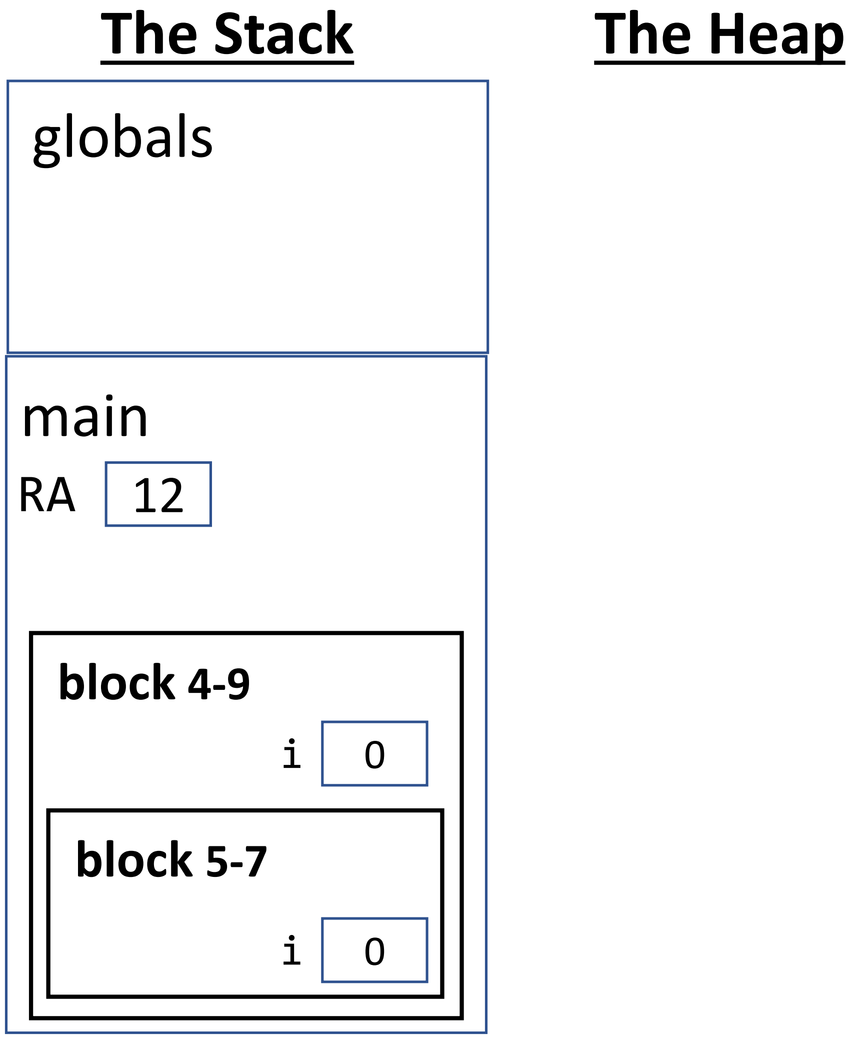
i 1

Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```



The Stack

The Heap

globals

main

RA 12

block 4-9

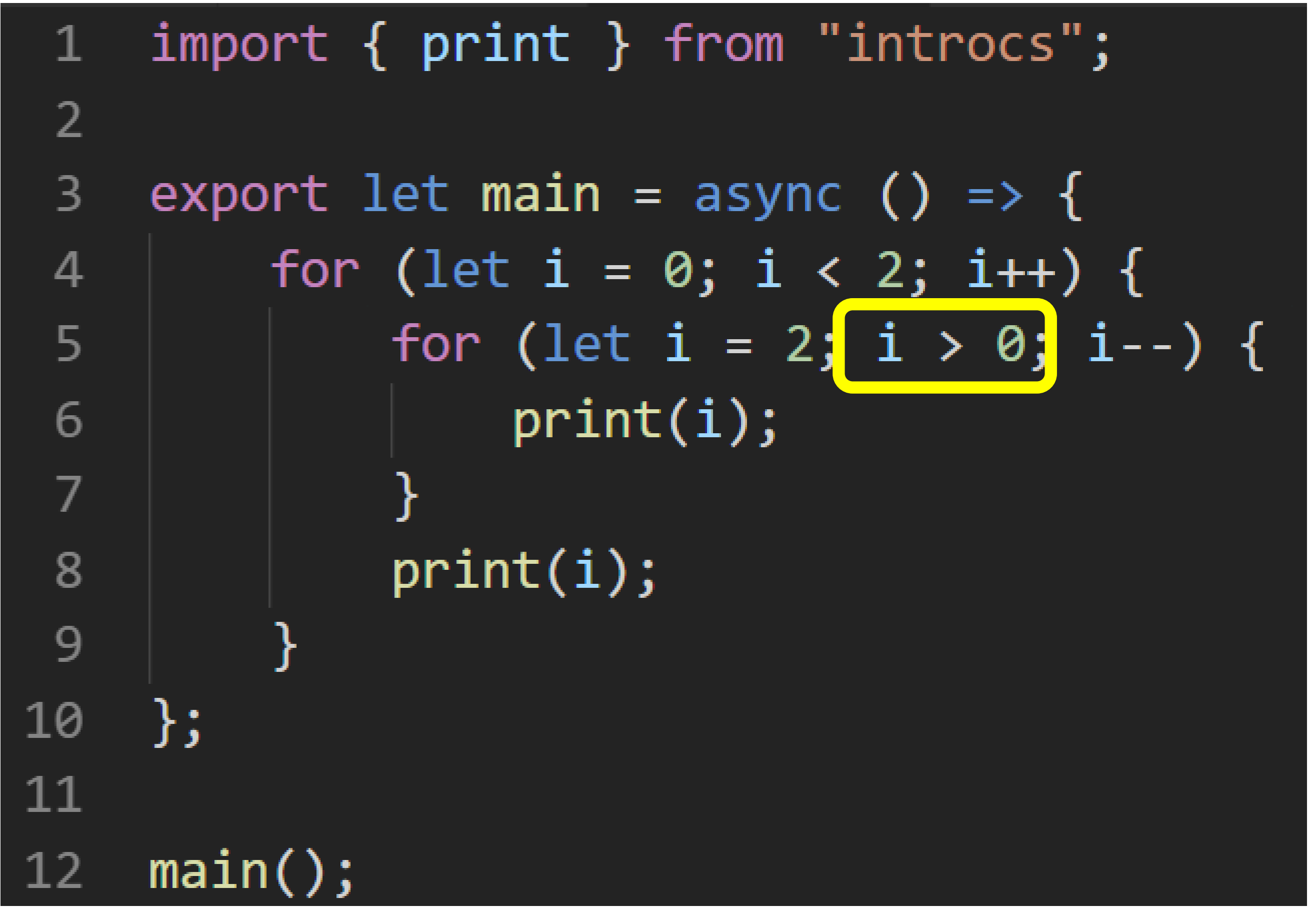
i 0

block 5-7

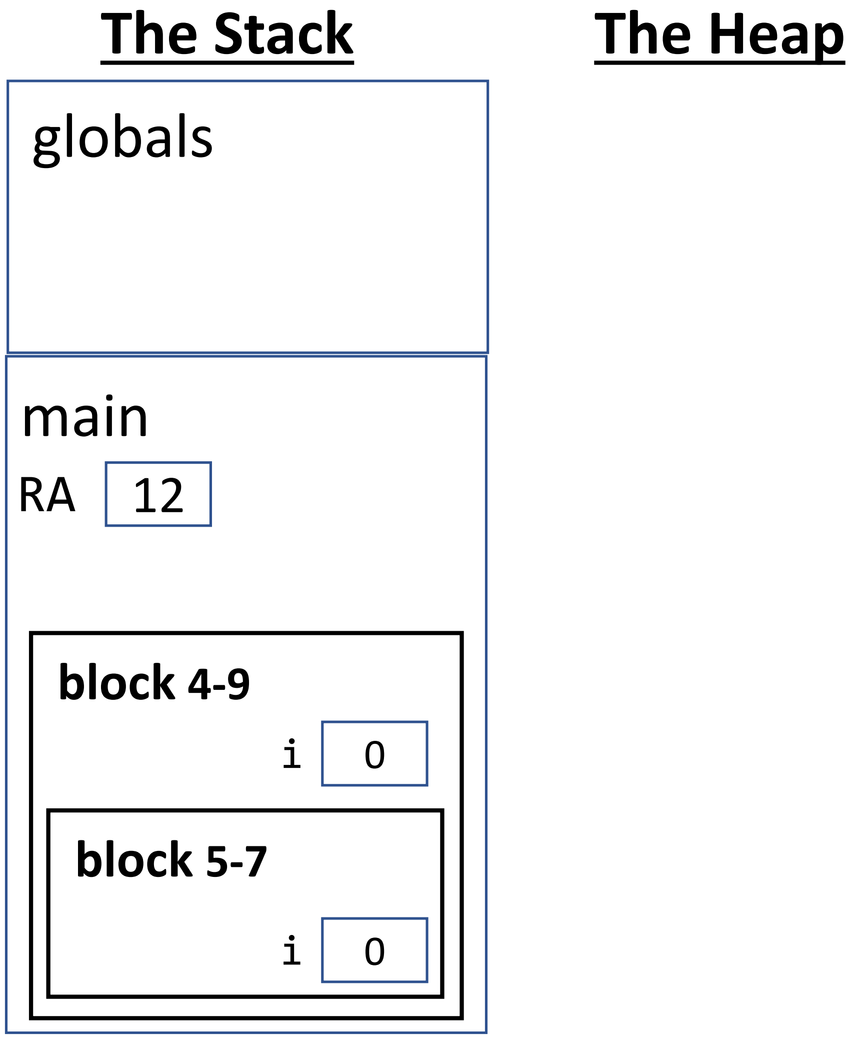
i 0

Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```



The Stack

The Heap

globals

main

RA 12

block 4-9

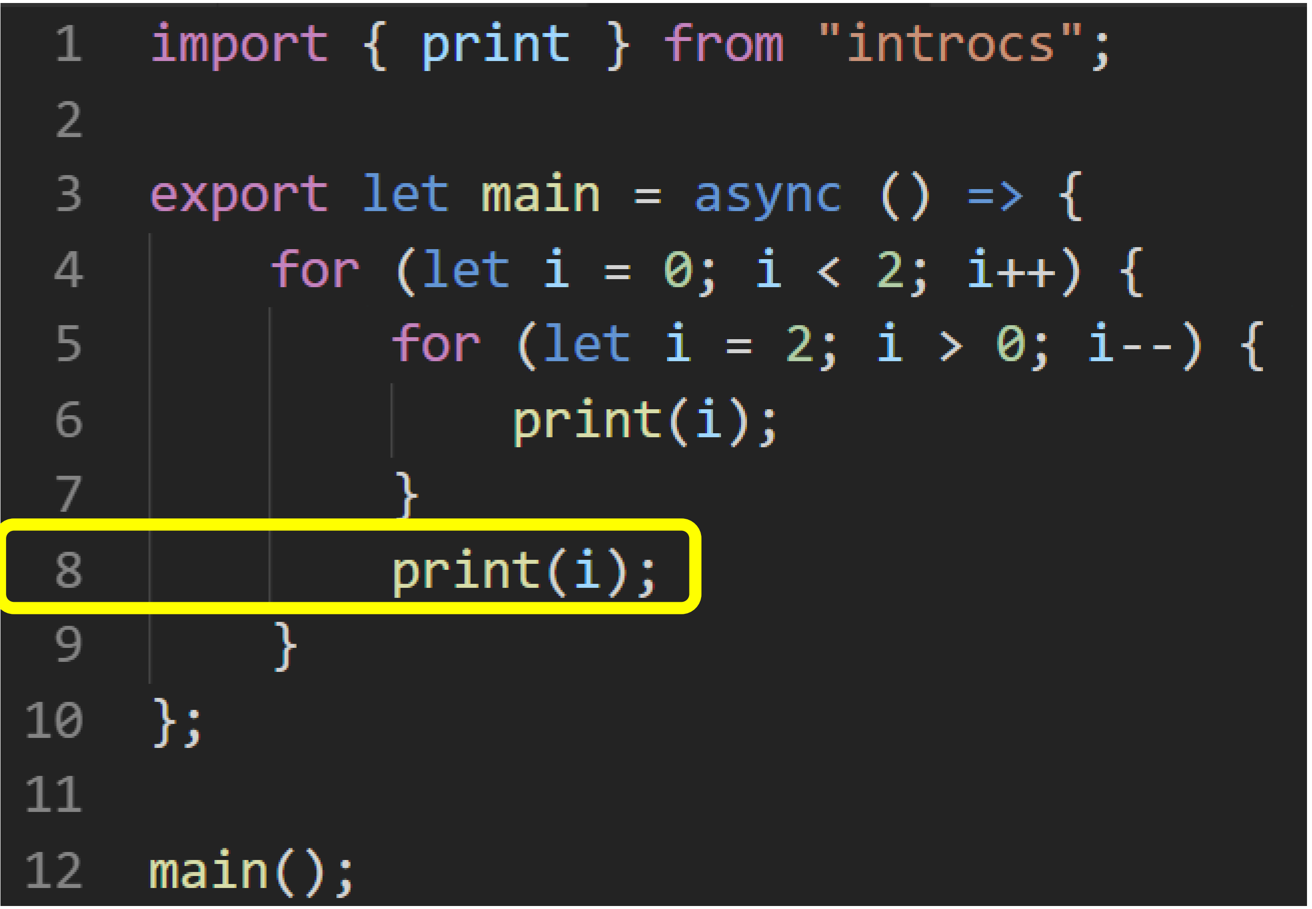
i 0

block 5-7

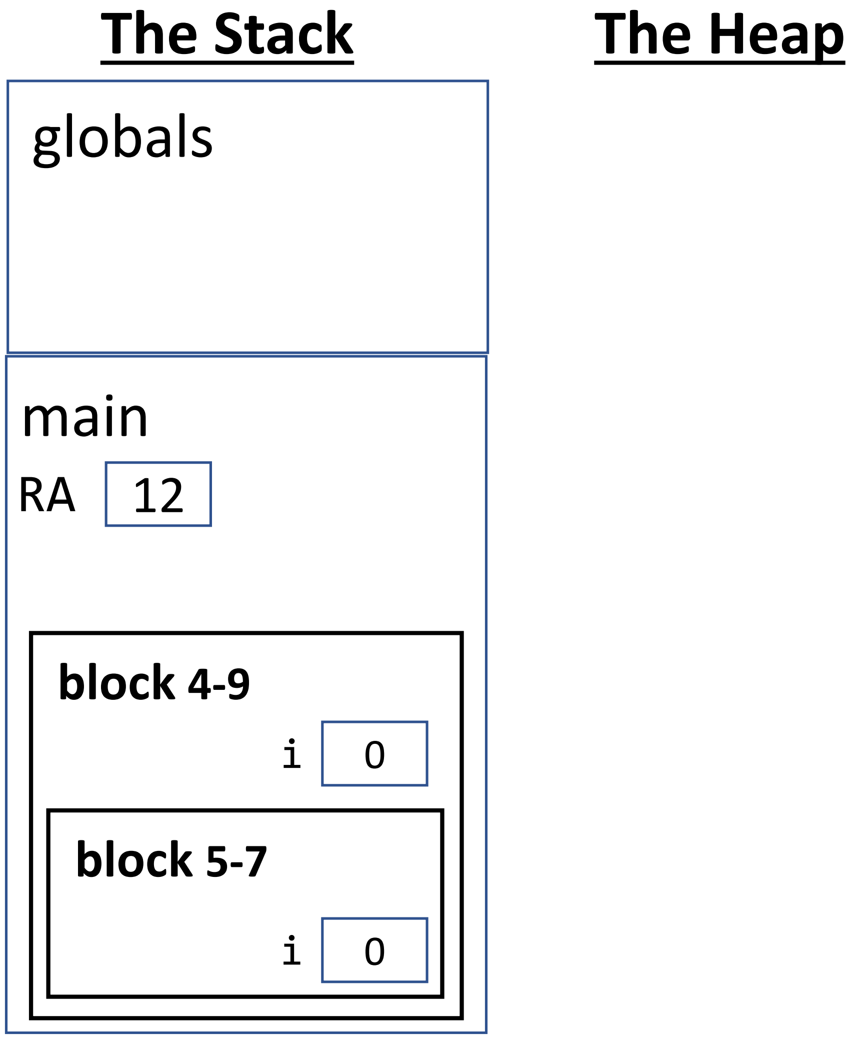
i 0

Name Resolution

Notice when line 8 is reached it is no longer in the block containing lines 5-7. At this point i refers to the one defined in the block surrounding lines 4-9.

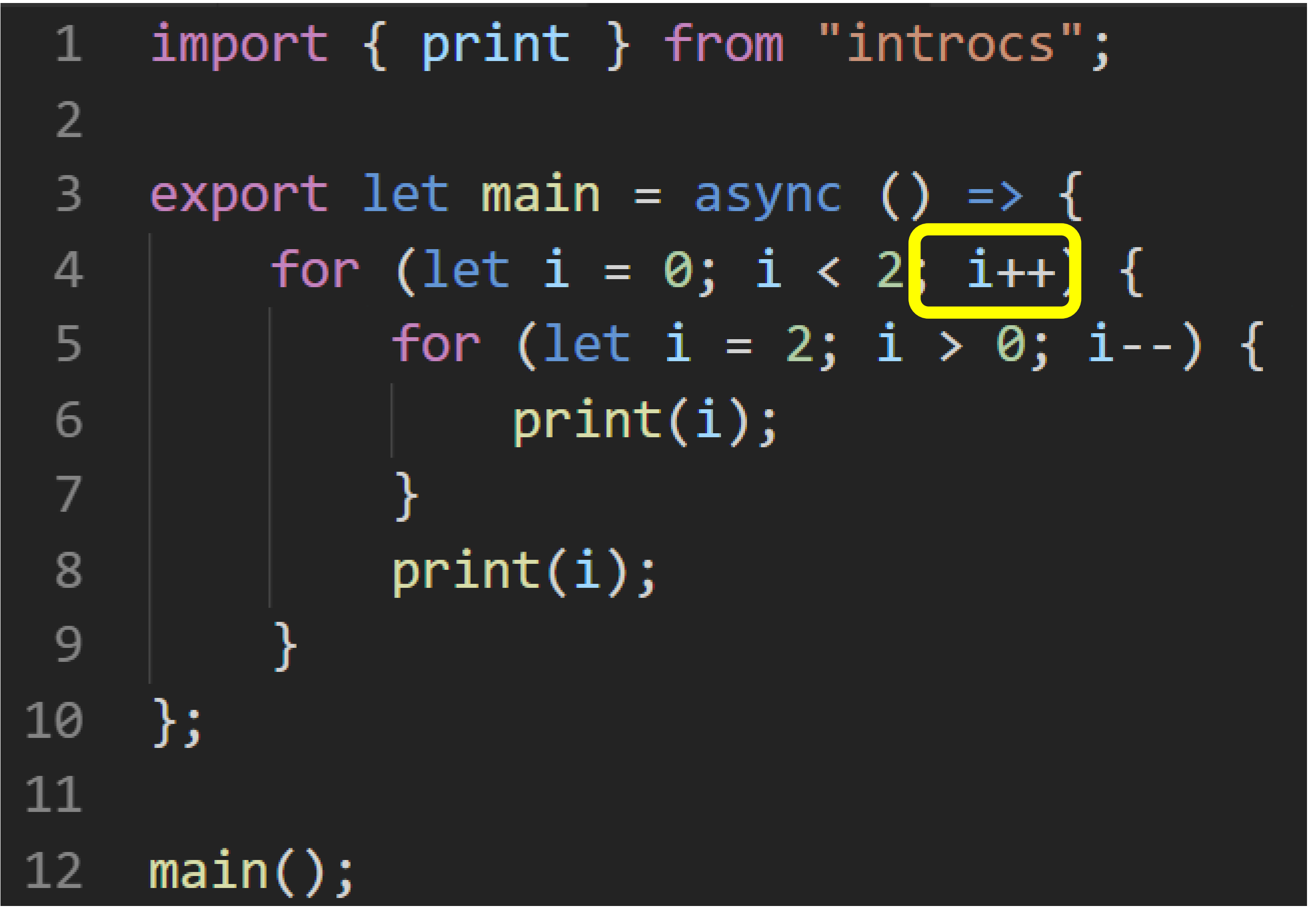


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

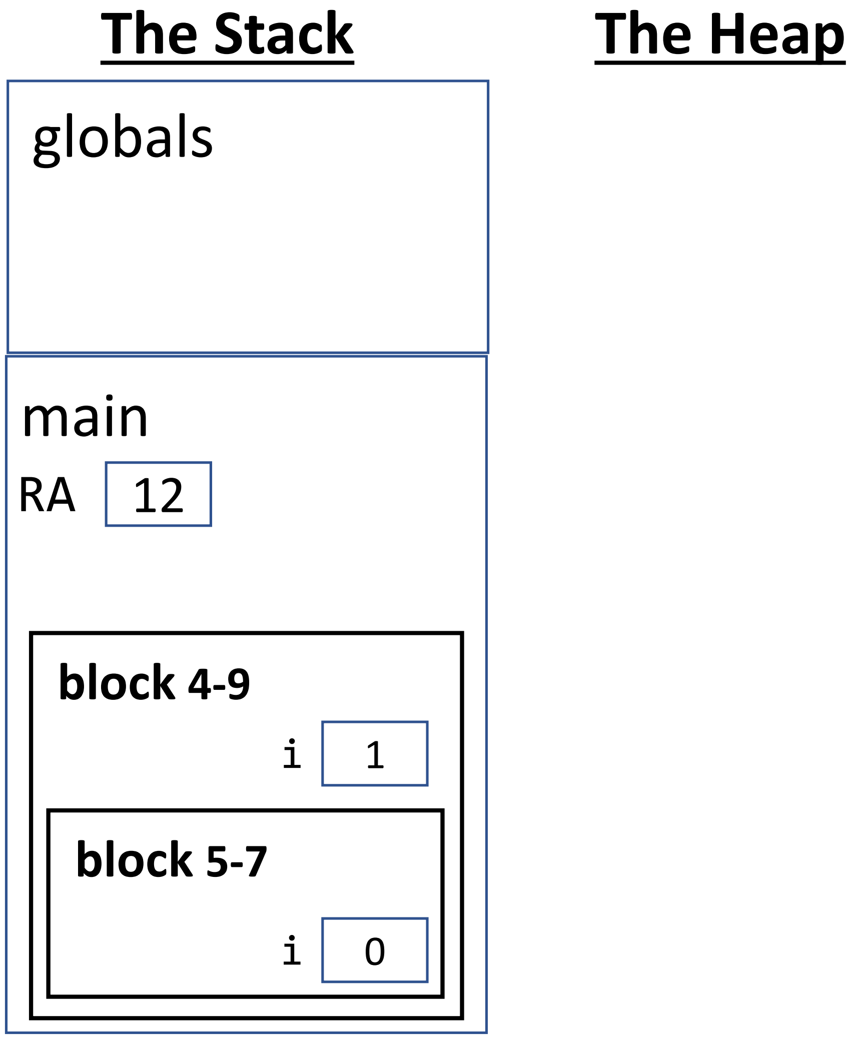


Name Resolution

Notice when line 8 is reached it is no longer in the block containing lines 5-7. At this point i refers to the one defined in the block surrounding lines 4-9.

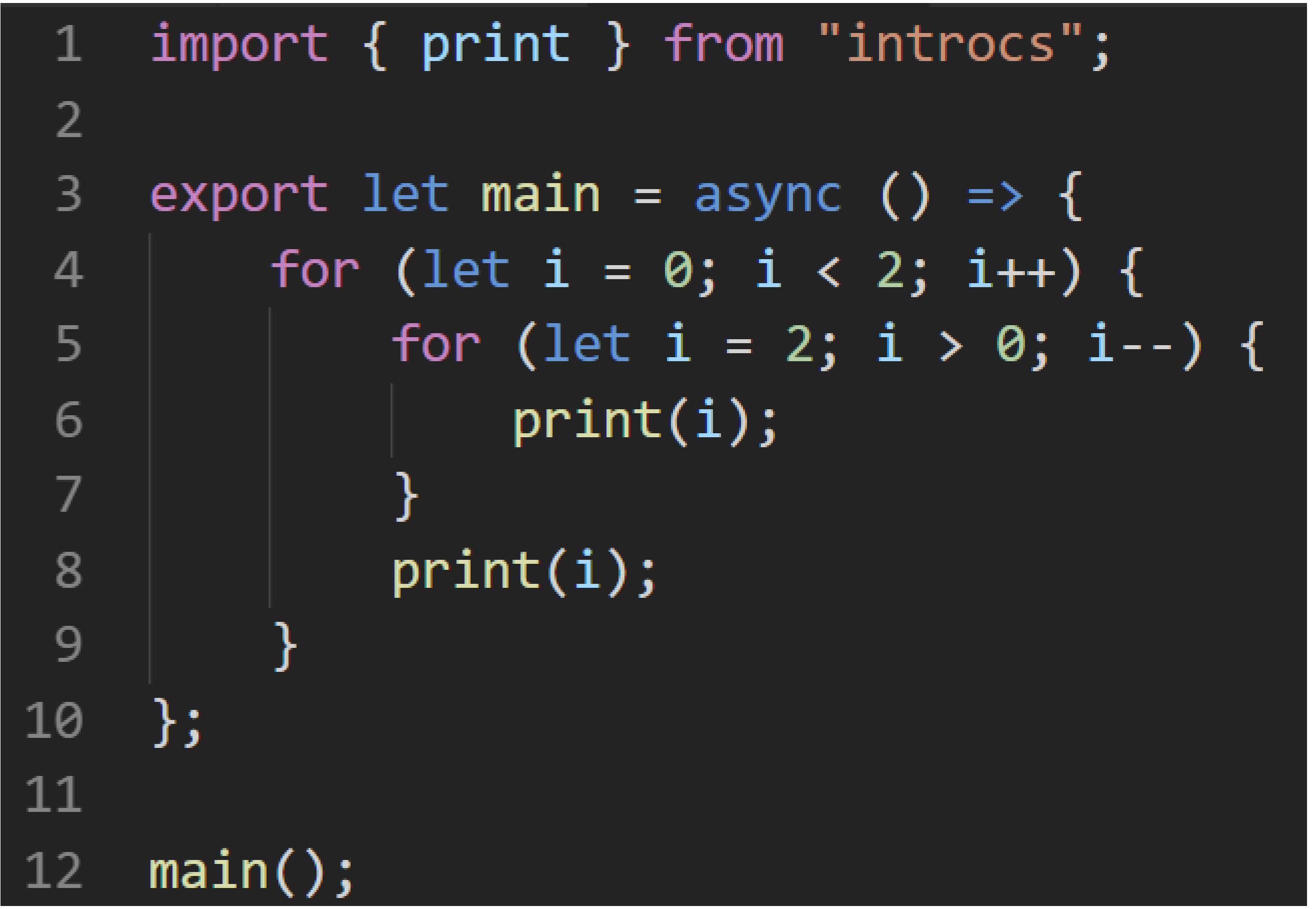


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

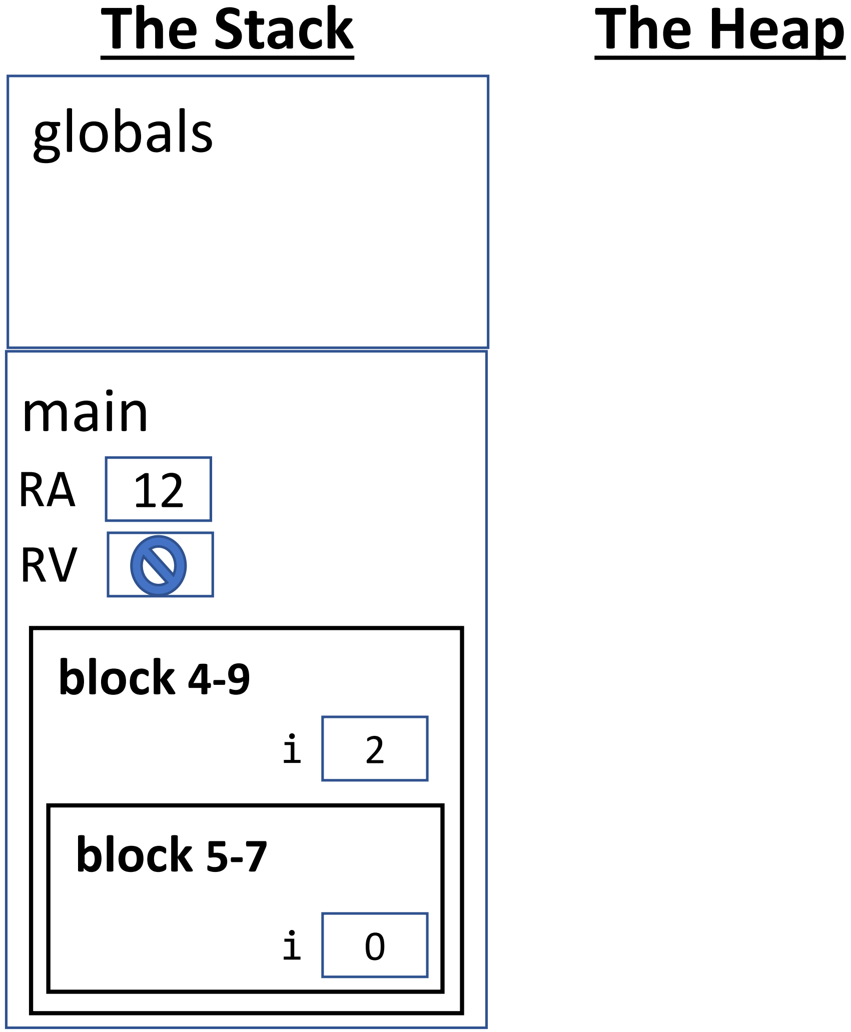


Fast-Forward

The final environment diagram would ultimately look like this. Reminder: Shadowing variable names in this way is bad practice! It is only shown here to illustrate the underlying rules at play.



```
1  import { print } from "introcs";
2
3  export let main = async () => {
4      for (let i = 0; i < 2; i++) {
5          for (let i = 2; i > 0; i--) {
6              print(i);
7          }
8          print(i);
9      }
10 };
11
12 main();
```



The Stack

The Heap

globals

main

RA 12

RV ⊘

block 4-9

i 2

block 5-7

i 0

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.

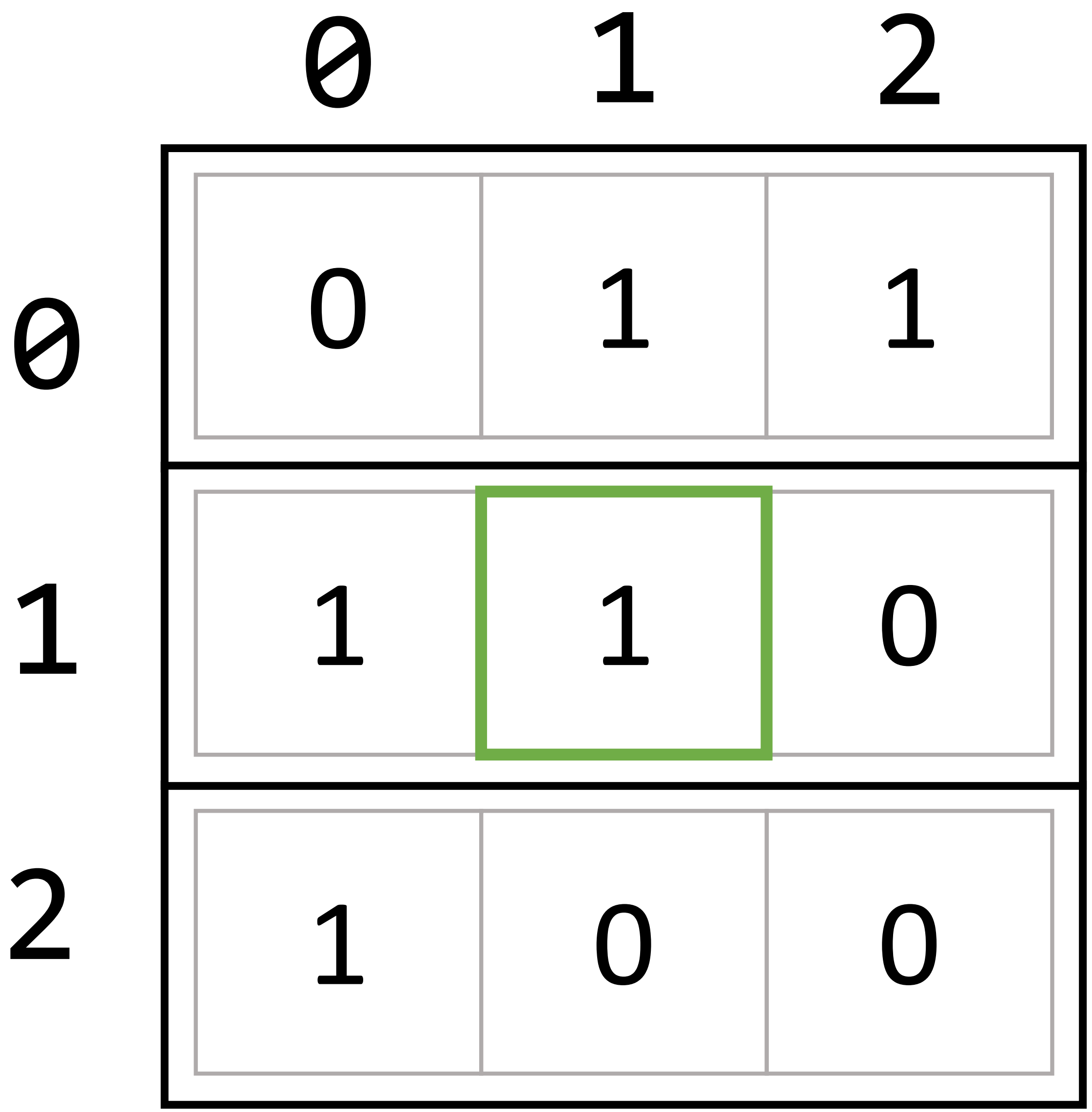
1. If currently inside of a block: check the block first.
2. If currently inside of a nested block: check the blocks from inside-to-out.
3. Check the current frame (the last one added without an RV).
4. Check the Globals frame.

countLiveNeighbors

- The rules of the game depend on how many live neighbors surround a given cell

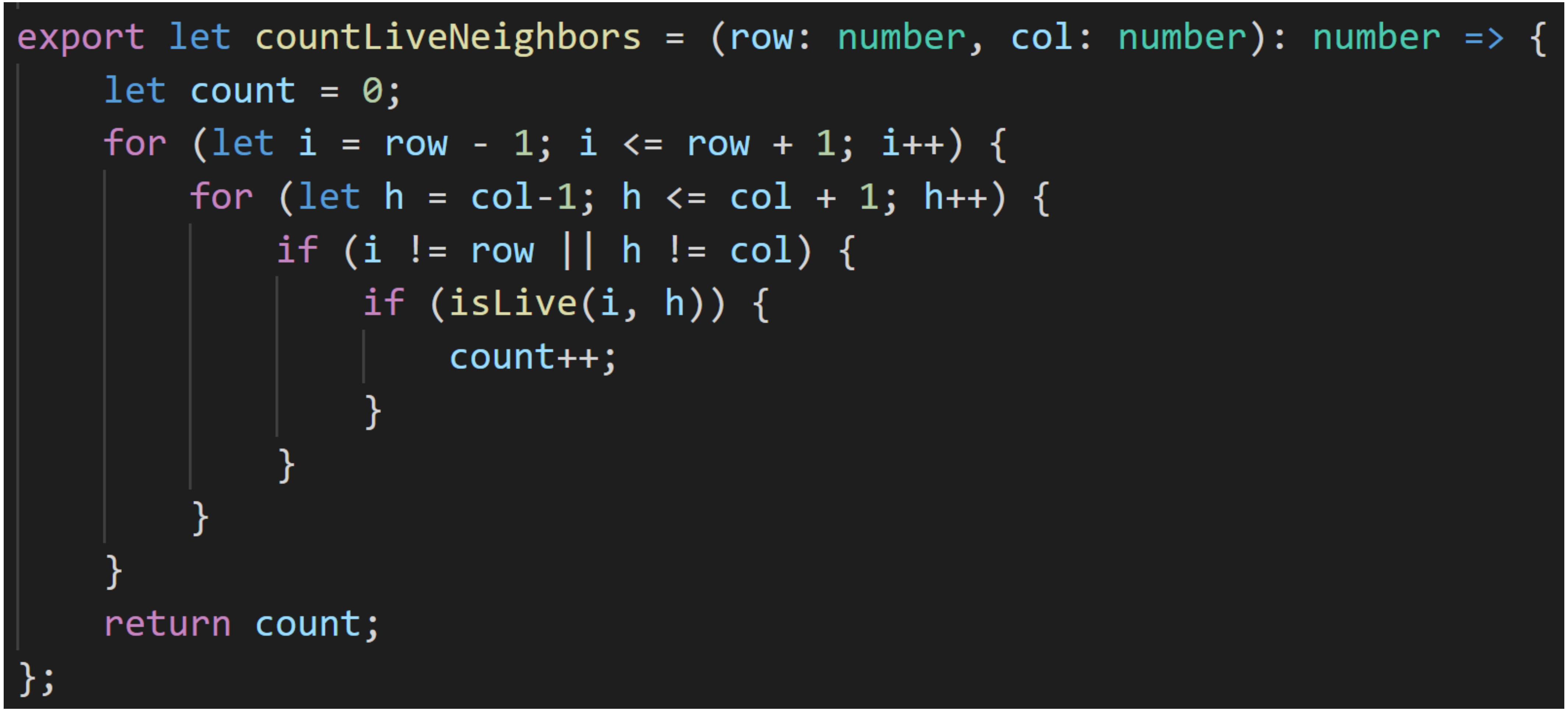
- Let's write a function that checks all surrounding cells and counts the number of 1s

- We'll use this when implementing rules



	0	1	2
0	0	1	1
1	1	1	0
2	1	0	0

countLiveNeighbors



```
export let countLiveNeighbors = (row: number, col: number): number => {
    let count = 0;
    for (let i = row - 1; i <= row + 1; i++) {
        for (let h = col-1; h <= col + 1; h++) {
            if (i != row || h != col) {
                if (isLive(i, h)) {
                    count++;
                }
            }
        }
    }
    return count;
};
```

Hands-on: **rules** method

- Given a row and column, apply the following rules.
  - Hint: make use of your this.**isLive** and this.**countLiveNeighbors** methods

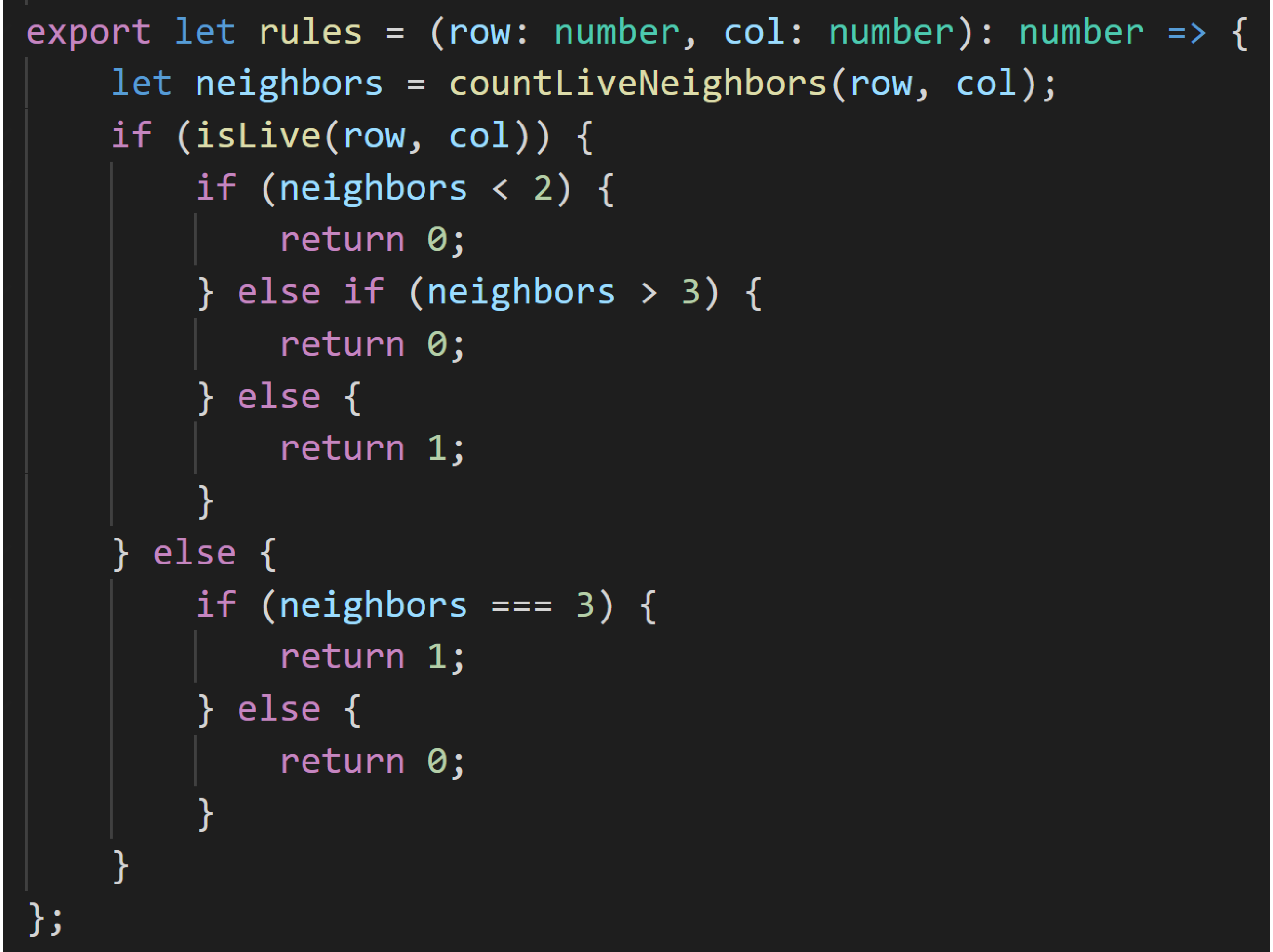
- If the cell is **alive**
  - Cell dies of underpopulation if live neighbors < 2
  - Cell survives if live neighbors is 2 or 3
  - Cell dies of overpopulation if live neighbors > 3

- If the cell is **dead**
  - Cell comes to life if live neighbors is 3
  - Otherwise cell remains dead

- Return 0 if cell rules result in a dead cell, 1 if cell rules result in a live cell

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

rules



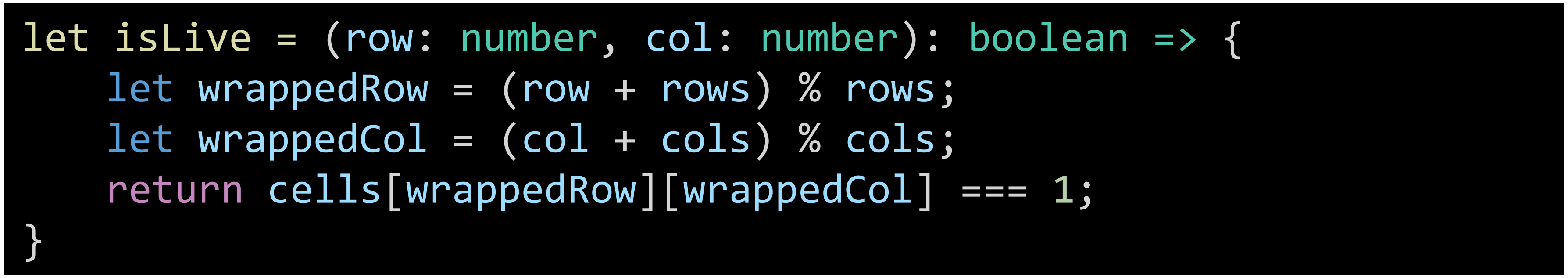
```
export let rules = (row: number, col: number): number => {
    let neighbors = countLiveNeighbors(row, col);
    if (isLive(row, col)) {
        if (neighbors < 2) {
            return 0;
        } else if (neighbors > 3) {
            return 0;
        } else {
            return 1;
        }
    } else {
        if (neighbors === 3) {
            return 1;
        } else {
            return 0;
        }
    }
};
```

isLive

- Let's improve the function that will test to see if a given cell is live
- This function will handle special edge cases:
  - It will "wrap around" a row/column if it is out of bounds (think: Pac-Man)
  - For example, if we ask whether the cell at row -1 and column 1 is alive we will actually test row 2 column 1.

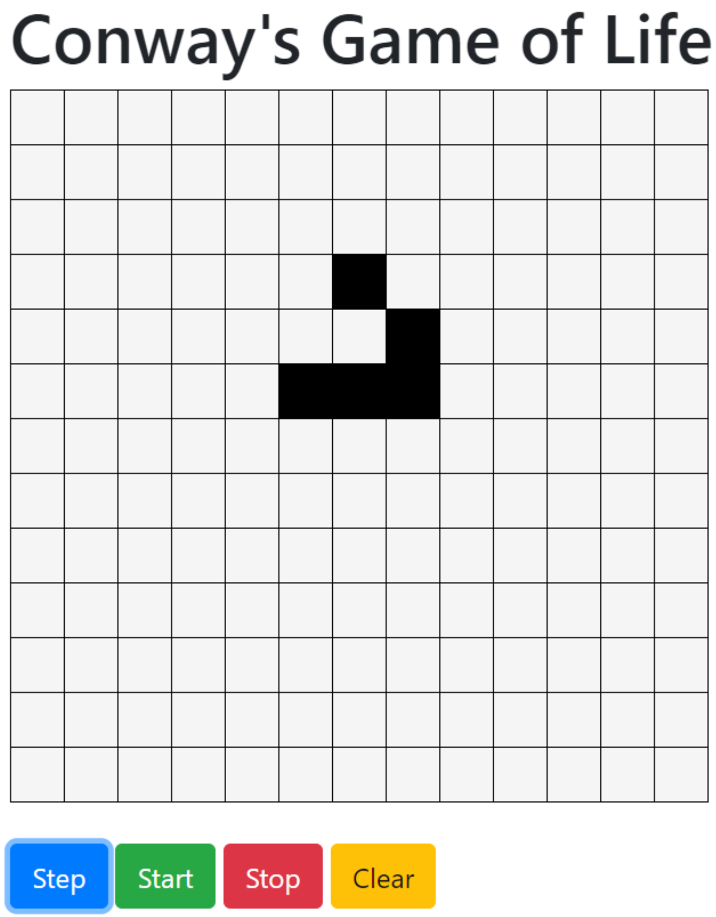
	0	1	2
0			
1			
2			

isLive



```
let isLive = (row: number, col: number): boolean => {
    let wrappedRow = (row + rows) % rows;
    let wrappedCol = (col + cols) % cols;
    return cells[wrappedRow][wrappedCol] === 1;
}
```

Emergent Behavior



Conway's Game of Life

- The shape to the left is called a Glider… try it out!
- Over the years many interesting, non-converging patterns have been found. Try searching the web for more.
- Simple example of how a few rules can lead to complex, emergent systems of behavior.