Environment Diagrams

The State of a Program: Lecture 10

Warm-up Questions

- A Blueprint : A Constructed House :: A Class : An Object
- Suppose you have a variable named p that is an object of type Person with a property named age. How would you access its age property? **`p.age`**
- Once a class is defined, you can use its name as a... type!
- Suppose you have a class named Person. How would you declare an empty array of Person objects? **`let people: Person[] = [];`**
- The bundling of related values is an important benefit of composite data types because it is more semantic and more convenient.
- Suppose you have an array of Person objects named people. How would you access the age property of the first element in the array? **`people[0].age`**

Announcements

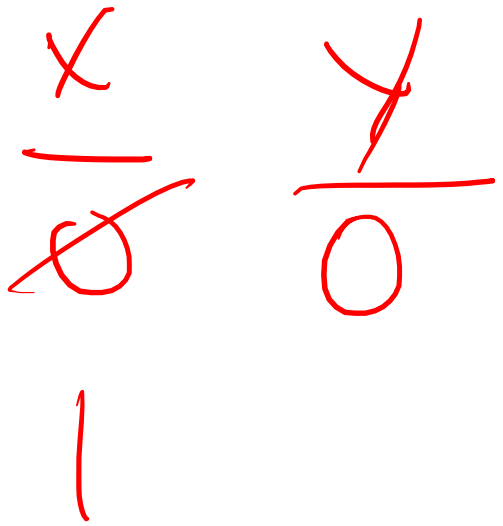
- Worksheet on classes/objects
  - Posted Tuesday afternoon
  - Due Tuesday 10/9

Challenge Question #0 - What is printed?

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



Challenge Question #1 - What is printed?

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

export let main = async () => {
    let x: number[] = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```

Environment Diagram

- A visual tool to represent the **complete state** in memory, or environment, of a running program at any given point in its execution.

- Why learn how to diagram these in COMP110?
  Three upcoming very important, but often perplexing, ideas will be much easier to understand with these diagrams:

  1. Values vs. References
  2. Recursion
  3. Higher-order Functions

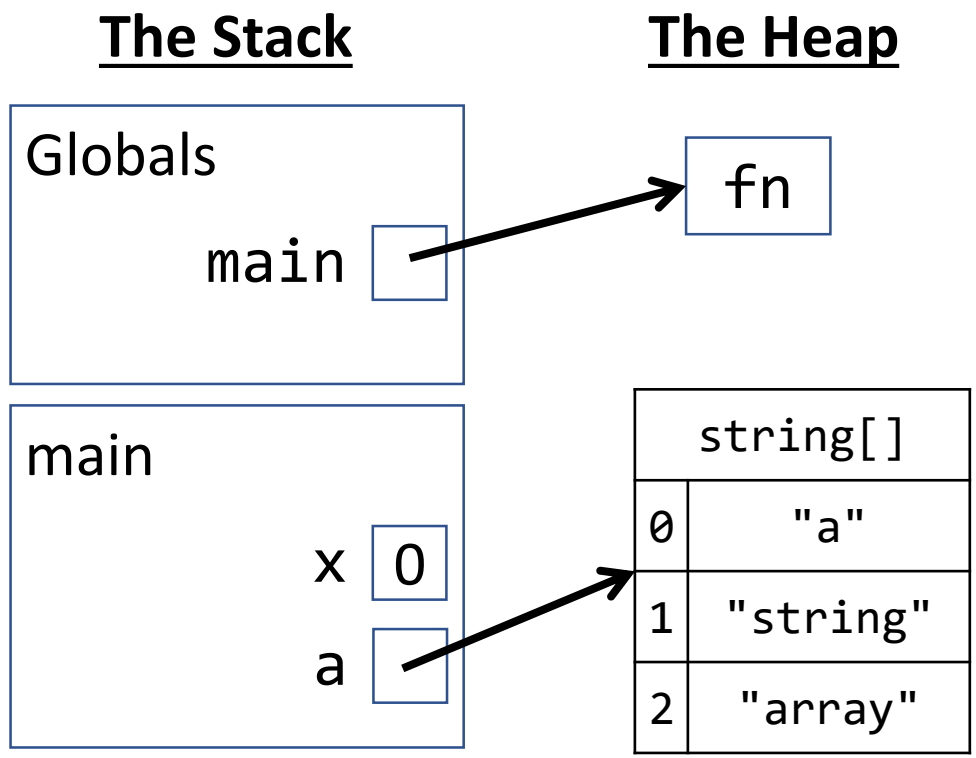
Environment Diagram

- There are two areas of an environment diagram:

1. Call Stack (or **"The Stack"**)
   - Every stack has a **Globals Frame**
   - When a function is called, a new **Frame** is added
   - Every frame has:
     - The name of its function definition
     - A list of **variable names** and boxes holding their **bound values**
     - Primitive variable values are stored in stack frames
2. Dynamic Memory Heap (or **"The Heap"**)
   - All *function, array,* and *object values* are located here

- This is *effectively* the model of how state in your programs is managed by the processor.

Example:



Stack vs. Heap

- You can think of the call **stack** as the program's "working space"
  - The current function being evaluated is always working in the most recent frame added to the call stack

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.

Case: Local Primitive Variables

Globals - Starting Point

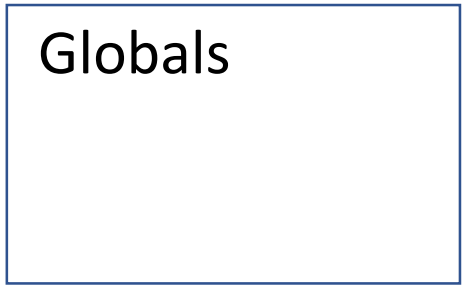
When a program is loaded by an interpreter, it begins with an empty* stack and heap. The top-most frame of our stack is called the **Globals frame**.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```

The Stack



The Heap

* This is a lie. The interpreter has established built-in functions, variables, and classes in its own stack and heap space *before* our program loads. Not our concern.

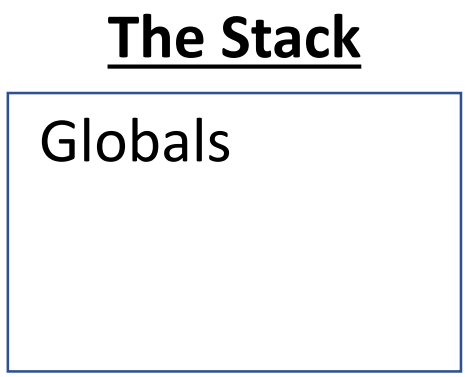
Globals - Imports

When a function, class, or value is imported from another file, *technically* it is entered into the Globals frame. However, **we will skip this step in our diagrams**.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



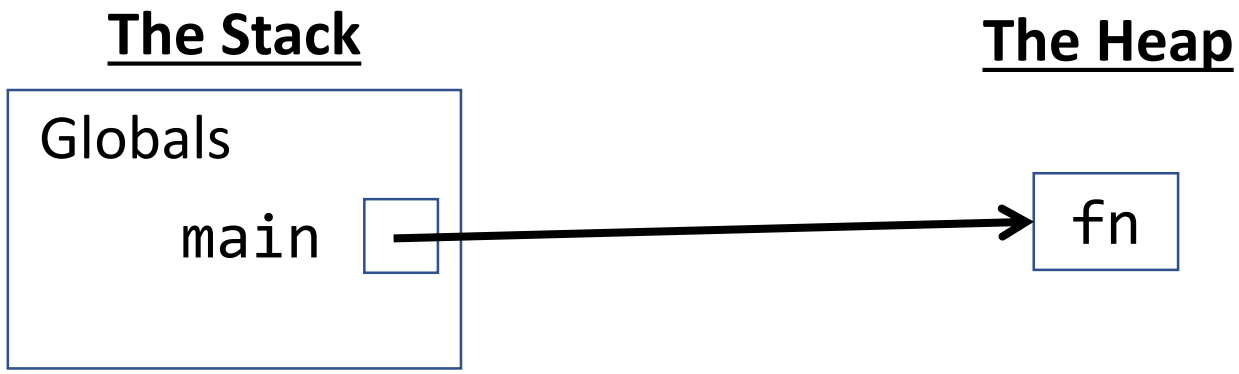
Variable Declaration - Function (1 / 2)

When a function definition is encountered, you will add its name to the current stack frame connected to a shorthand 'fn' symbol on the heap via pointer.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



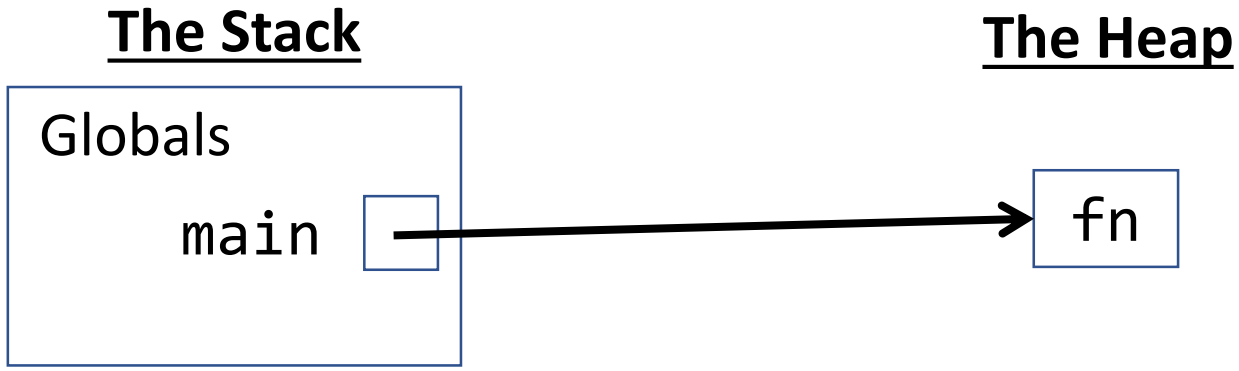
Variable Declaration - Function (2 / 2)

Wait! *A function definition is actually variable definition? Its code is stored in memory with other variables?* Yes*!!! We will explore these ideas soon.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* This is true in programming languages with *first-class functions.* Most modern programming languages feature first-class functions or "functions as values".

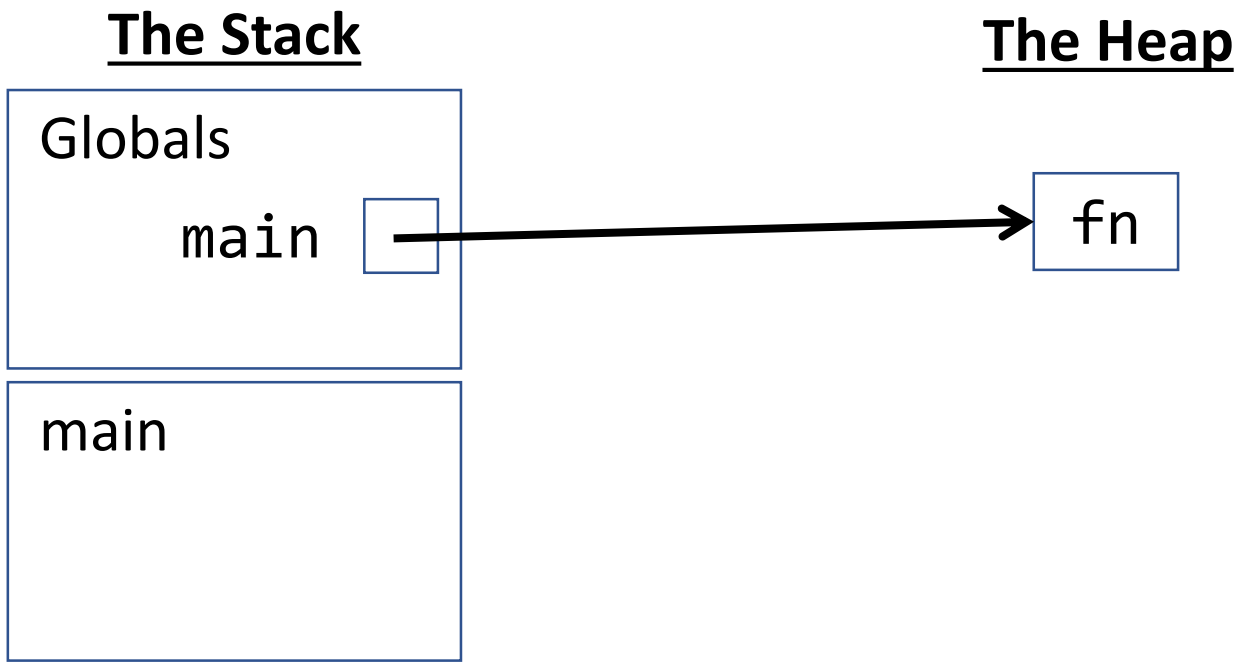
Function Call

When a function call* is encountered, a new **frame** is added to your stack. Label it with the function name.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* The last line of your programs all along has been a simple function call to the **`main`** function! *This* is why **`main`** is our programs' starting point.

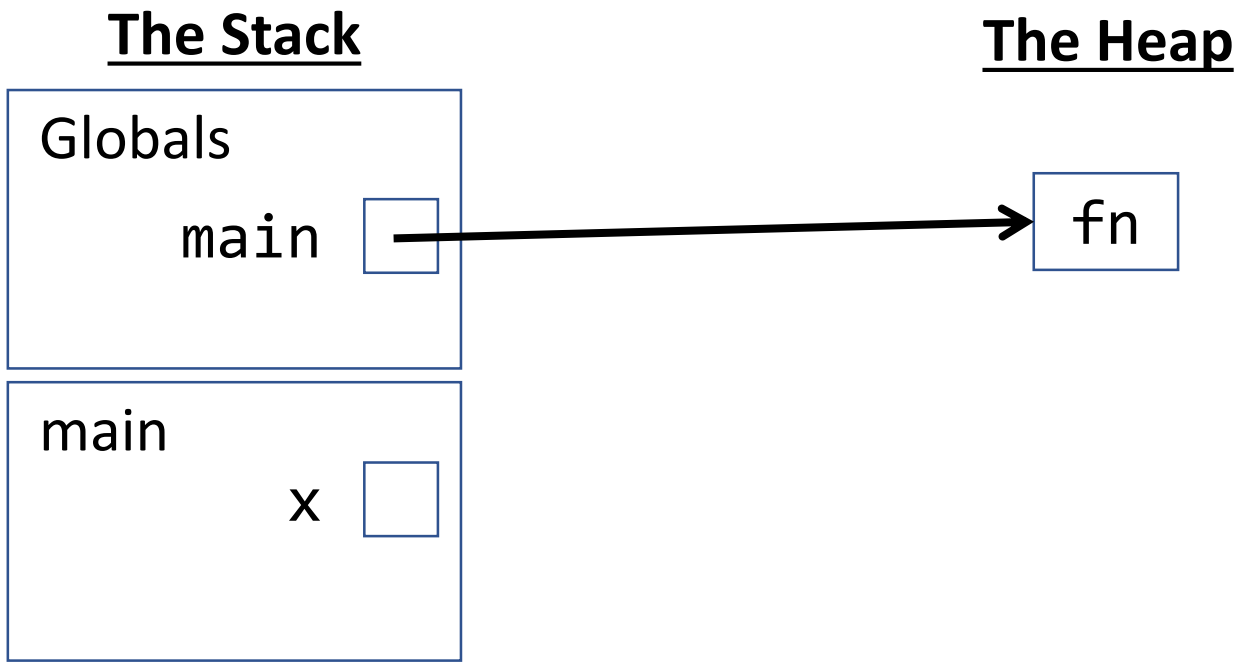
Variable Declaration

When a **variable** is declared, add its name to the current frame on the stack.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* The last line of your programs all along has been a simple function call to the **`main`** function! *This* is why **`main`** is our programs' starting point.

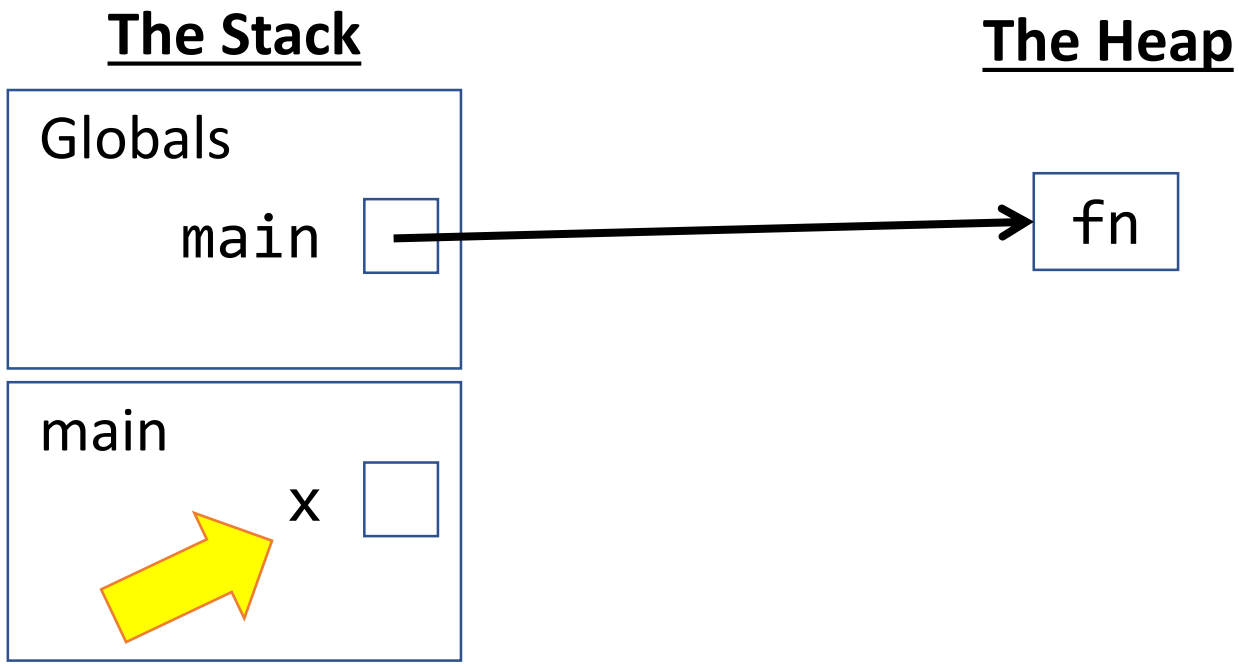
Name Resolution - Variable Assignment

When a **variable** is assigned to, find its name in the current frame on the stack. If it is not there, then look in the globals frame.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* The last line of your programs all along has been a simple function call to the **`main`** function! *This* is why **`main`** is our programs' starting point.

Variable Assignment - Primitives (number, boolean, string)

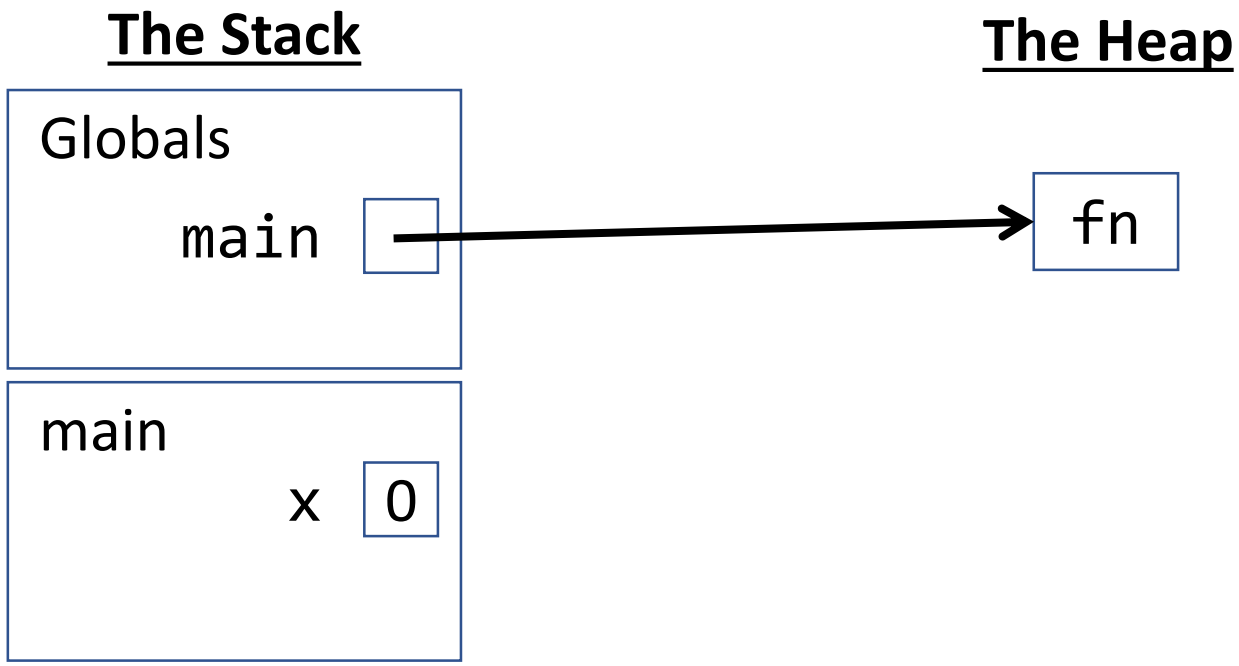
When a **primitive variable** is assigned a value, update its value in its stack frame.

Primitive variables' values are stored on the stack.*

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* The last line of your programs all along has been a simple function call to the **`main`** function! *This* is why **`main`** is our programs' starting point.

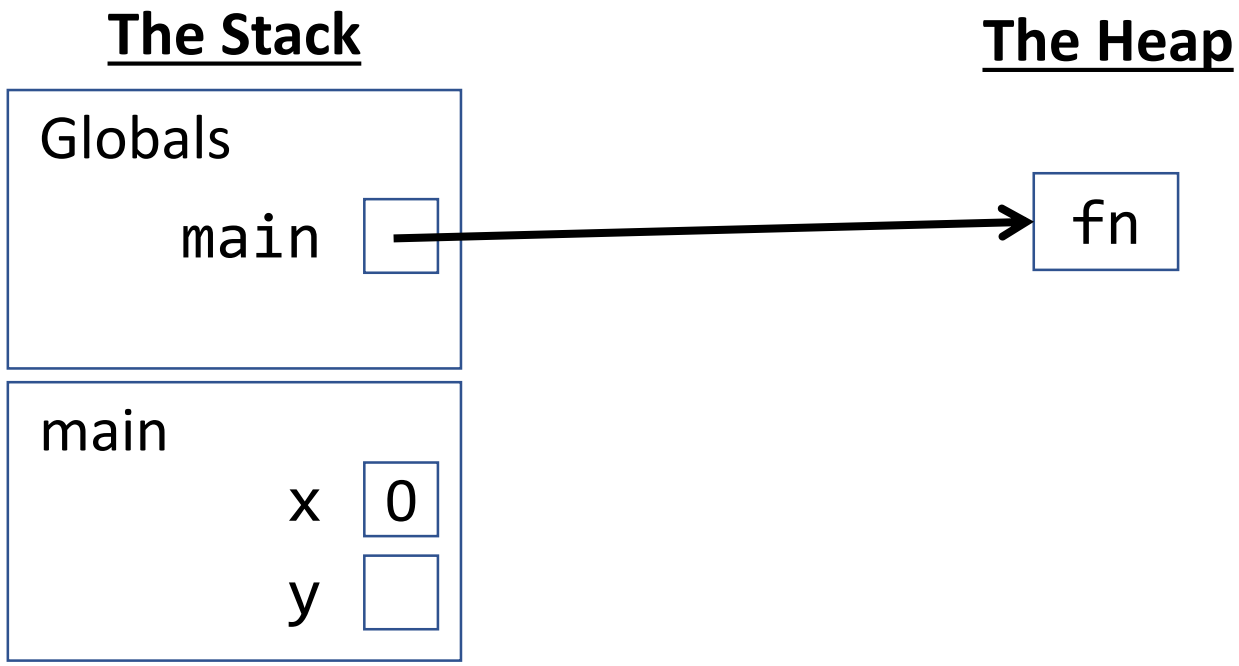
Variable Declaration

When a **variable** is declared, add its name to the current frame on the stack.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* The last line of your programs all along has been a simple function call to the **`main`** function! *This* is why **`main`** is our programs' starting point.

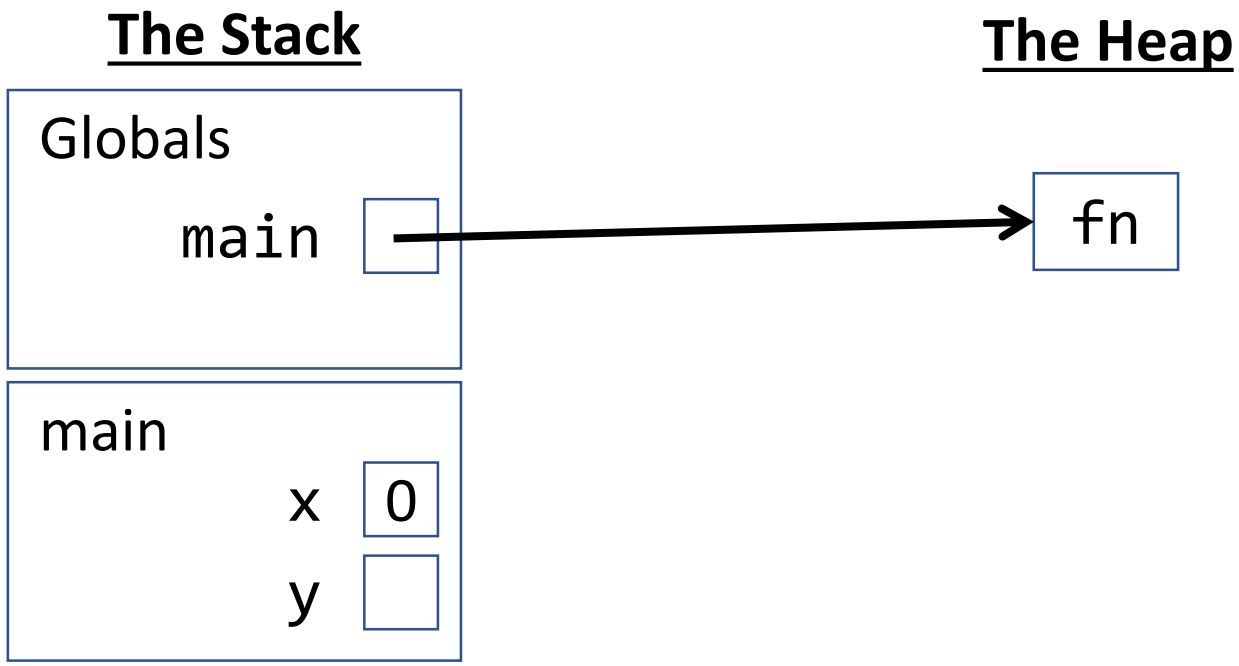
Name Resolution - Variable Access

When a **variable** is accessed, find its name in the current frame on the stack. If it is not there, then look in the globals frame. Substitute access with value*.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* In this case, x is substituted in the expression y = x, with the value 0, so it simplifies to y = 0.

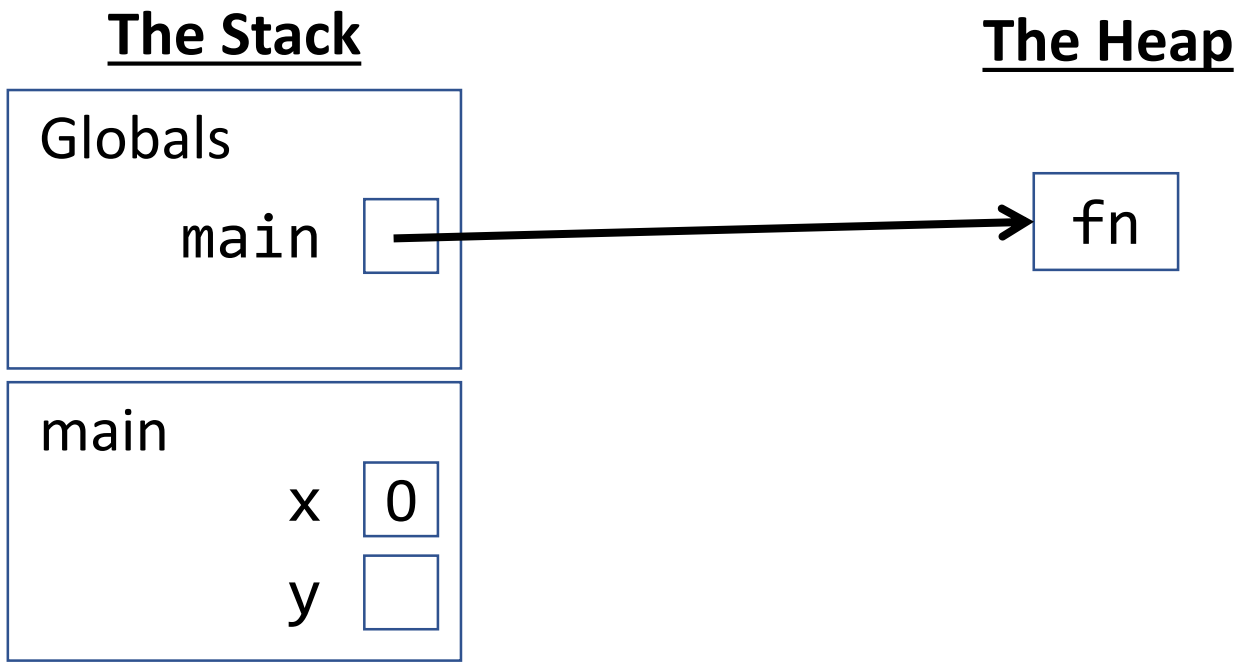
Name Resolution - Variable Assignment

When a **variable** is assigned to, find its name in the current frame on the stack. If it is not there, then look in the globals frame.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



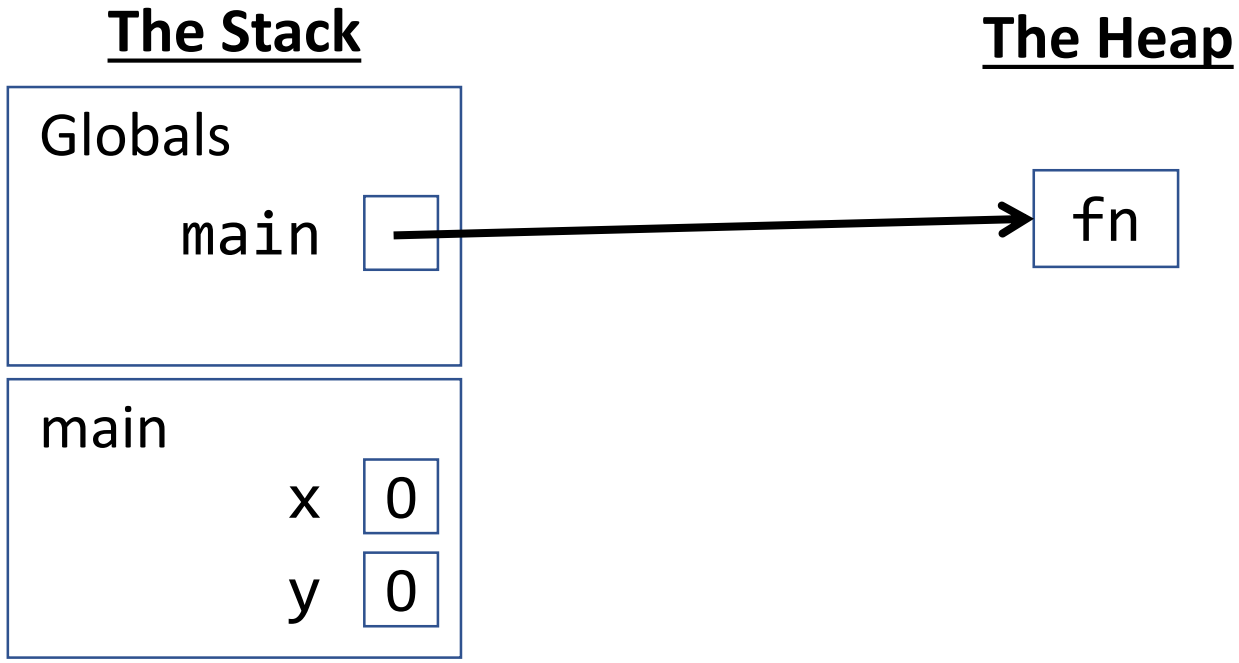
Variable Assignment - Primitives (number, boolean, string)

When a **primitive variable** is assigned a value, update its value in its stack frame.
Primitive variables' values are stored on the stack *and* **each holds its own value*.**

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* Further: Never write another variable's name as a value. Always copy the value! Never draw an arrow from one primitive variable's box to another's.

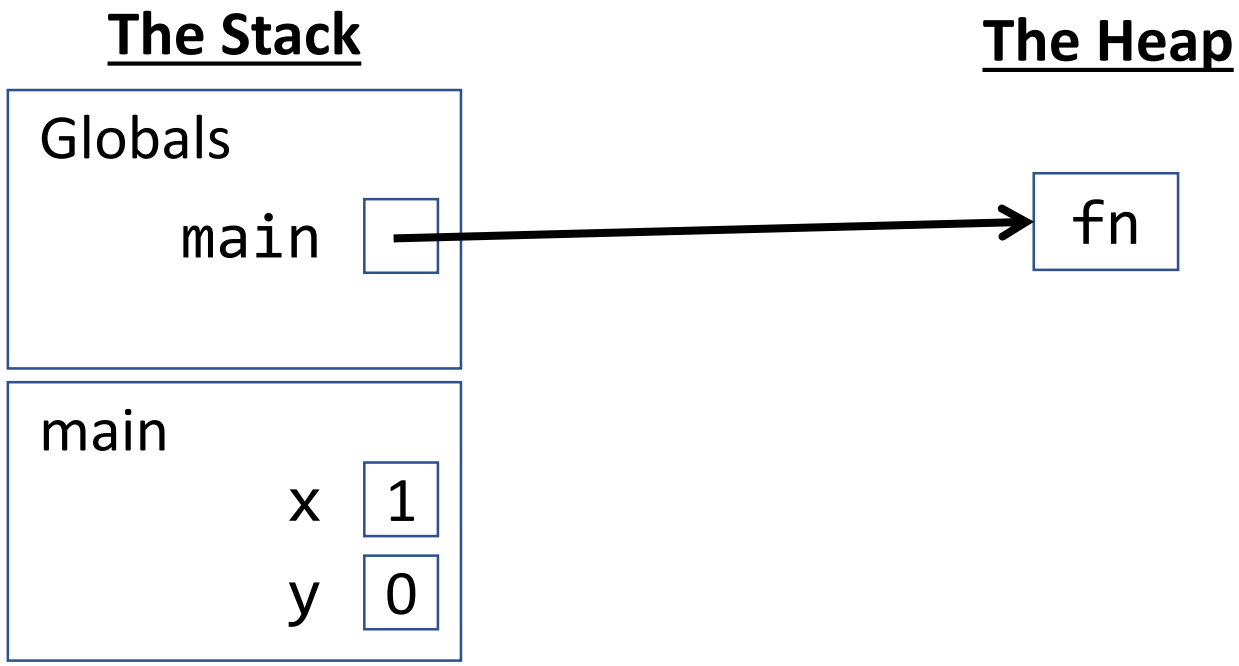
Name Resolution - Variable Assignment

When a **variable** is assigned to, find its name in the current frame on the stack. If it is not there, then look in the globals frame.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



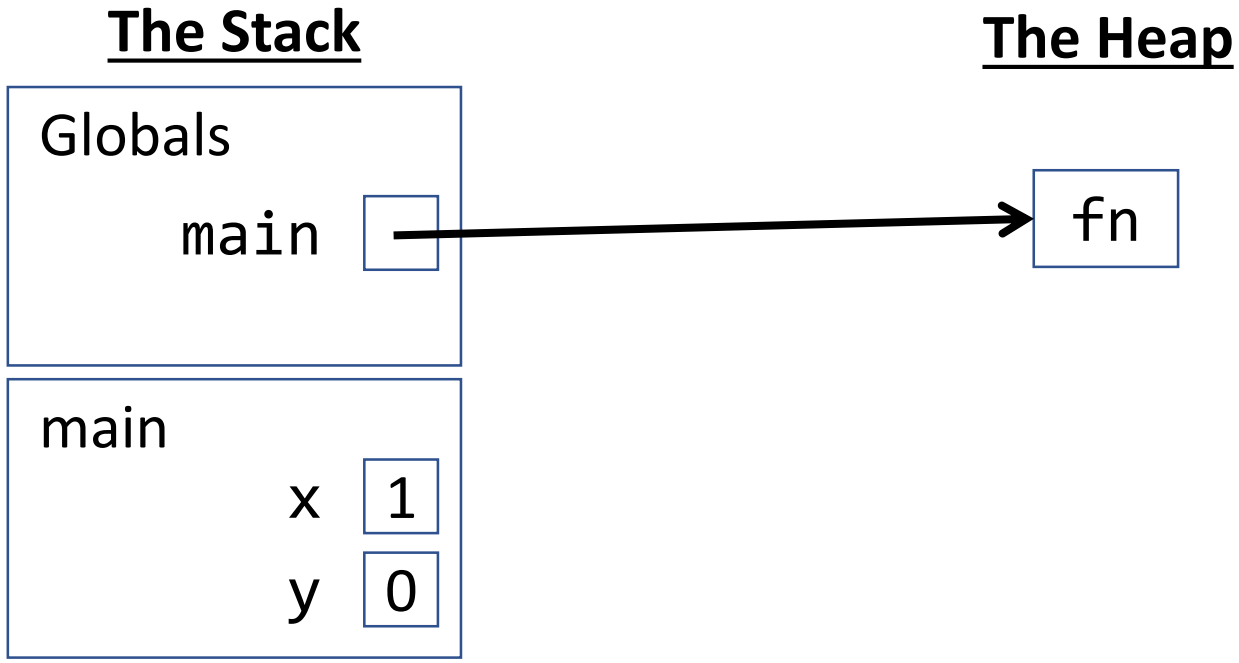
Name Resolution - Variable Access

When a **variable** is accessed, find its name in the current frame on the stack. If it is not there, then look in the globals frame. Substitute access with value*.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* In this case, `x` is substituted with the value `1` and `y` with the value `0`. The printed output would be "x: 1, y: 0".

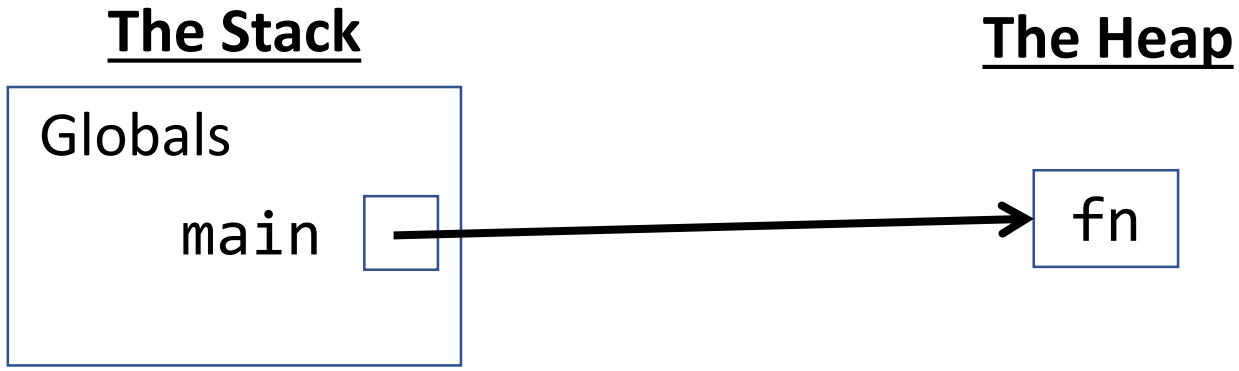
Function Return - `void` Functions

When a **void function** completes*, remove its frame from the stack and return to where the function call was invoked.

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

export let main = async () => {
    let x: number;
    x = 0;
    let y: number;
    y = x;
    x++;
    print("x:" + x + ", y:" + y);
};

main();
```



* `main`'s `async` flag makes it a special kind of function that allows us to `await` prompts from within it. We can think of it as a `void` function, though! If you were to add a `print("Goodbye");` statement after the call to `main();`, you would see the interpreter will continue on in our code past this call.

Case: Local Reference (Array) Variables

Globals - Starting Point

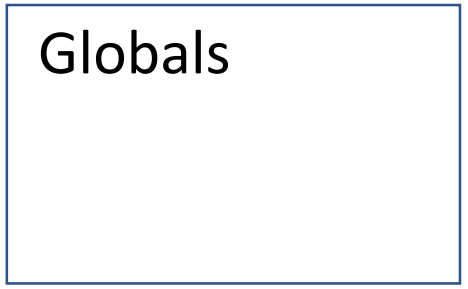
When a program is loaded by an interpreter, it begins with an empty stack and heap. The top-most frame of our stack is called the **Globals frame**.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```

The Stack



The Heap

Globals - Imports

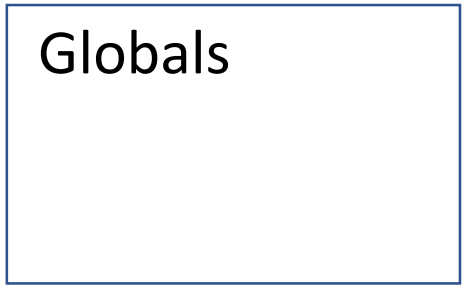
When a function, class, or value is imported from another file, *technically* it is entered into the Globals frame. However, **we will skip this step in our diagrams**.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```

The Stack



The Heap

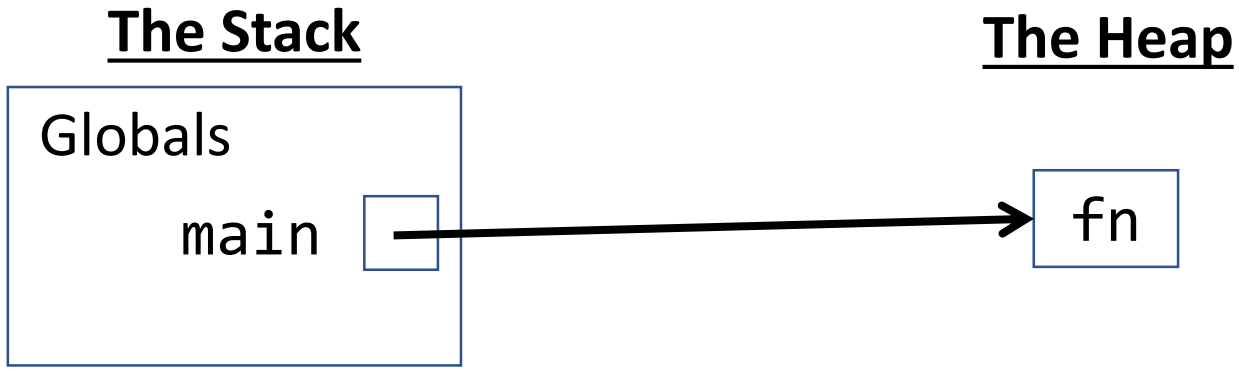
Variable Declaration - Function

When a function definition is encountered, you will add its name to the current stack frame connected to a shorthand 'fn' symbol on the heap via pointer.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



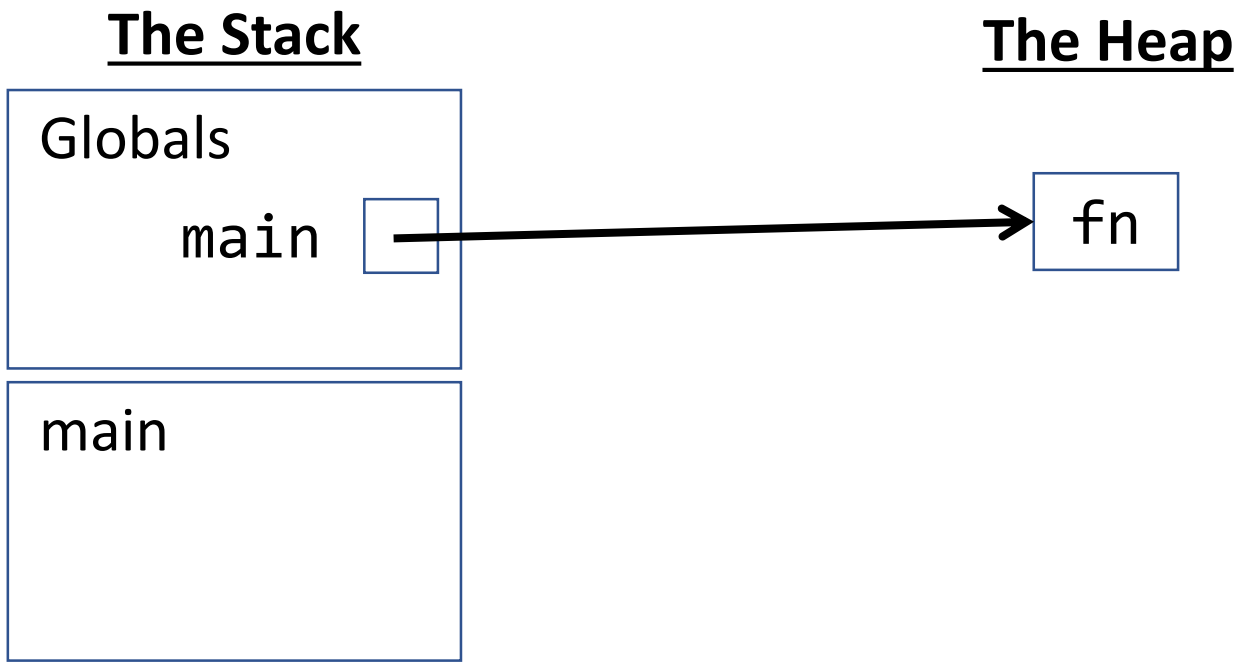
Function Call

When a function call is encountered, a new **frame** is added to your stack. Label it with the function name.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



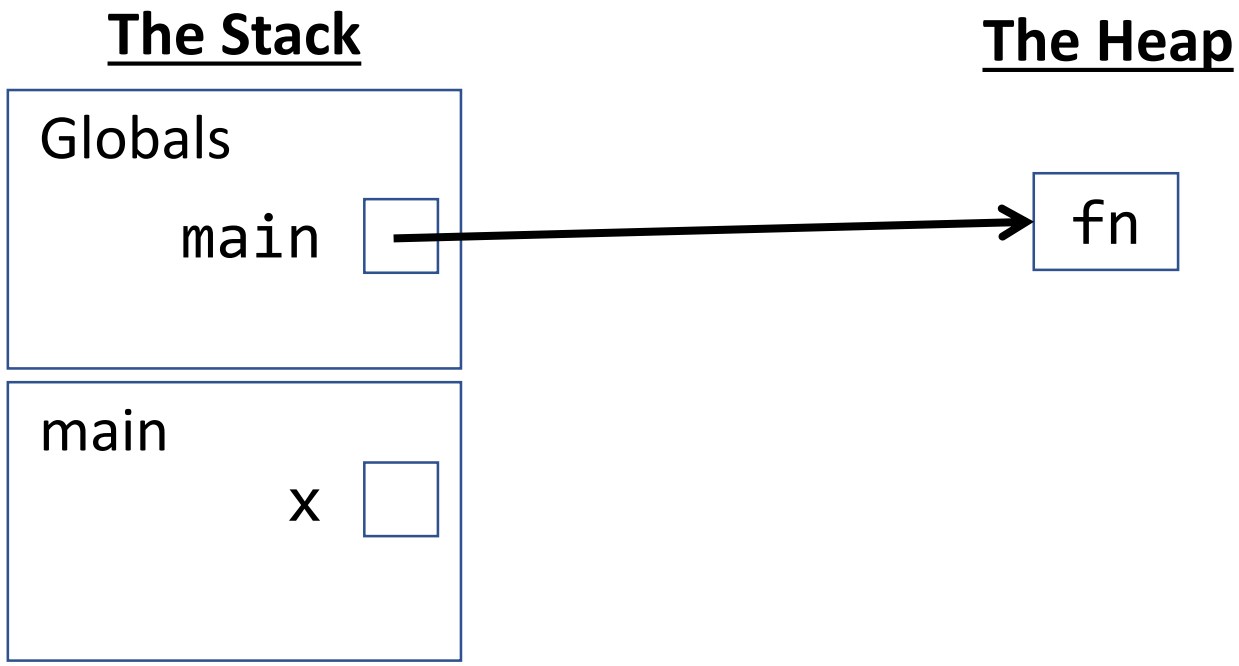
Variable Declaration

When a **variable** is declared, add its name to the current frame on the stack.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



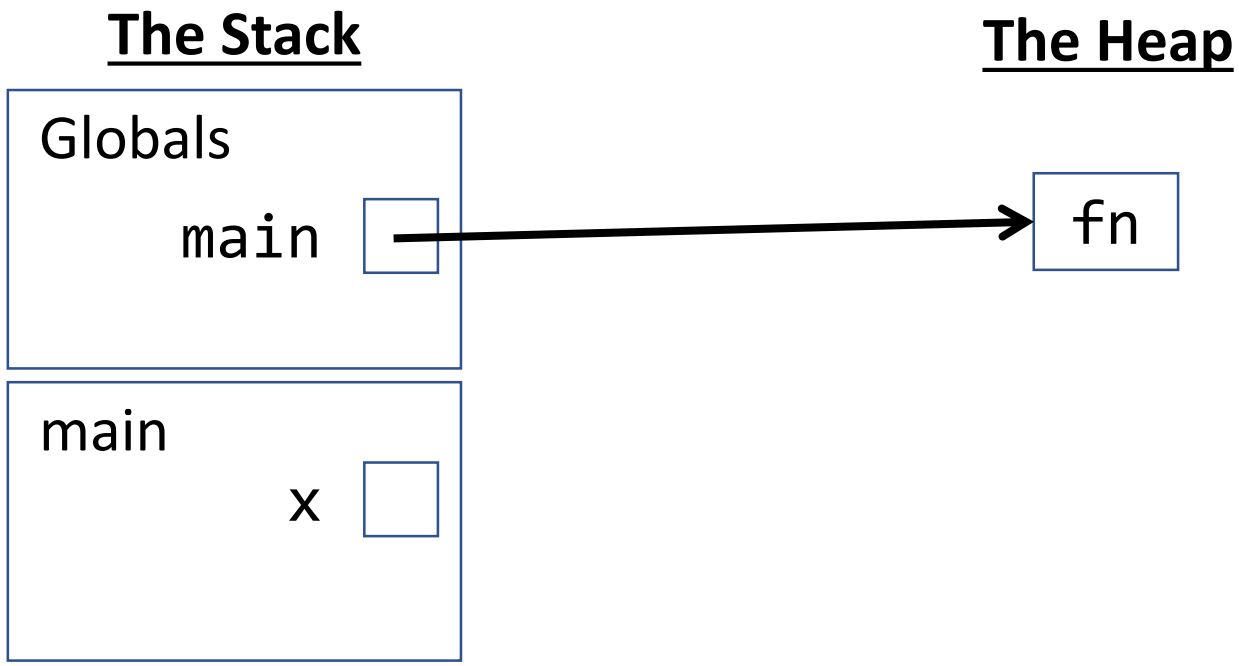
Name Resolution - Variable Assignment

When a **variable** is assigned to, find its name in the current frame on the stack. If it is not there, then look in the globals frame.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



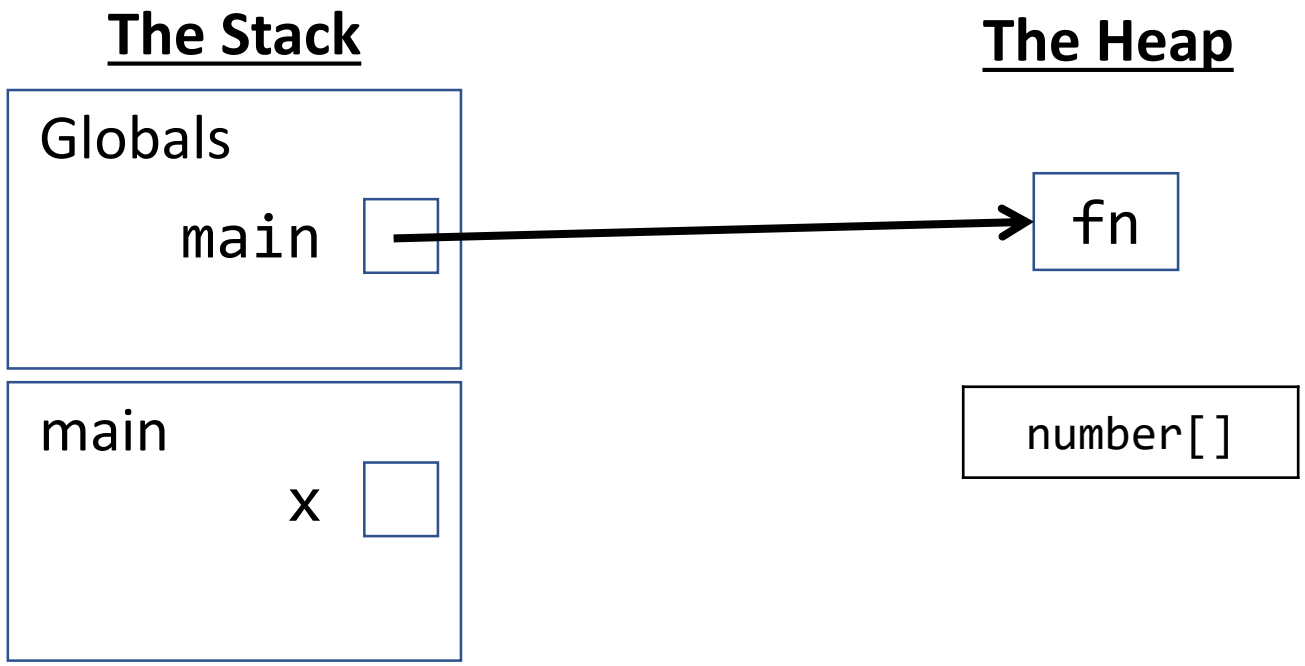
Array Literal

When an Array Literal is encountered, **establish it in The Heap**.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



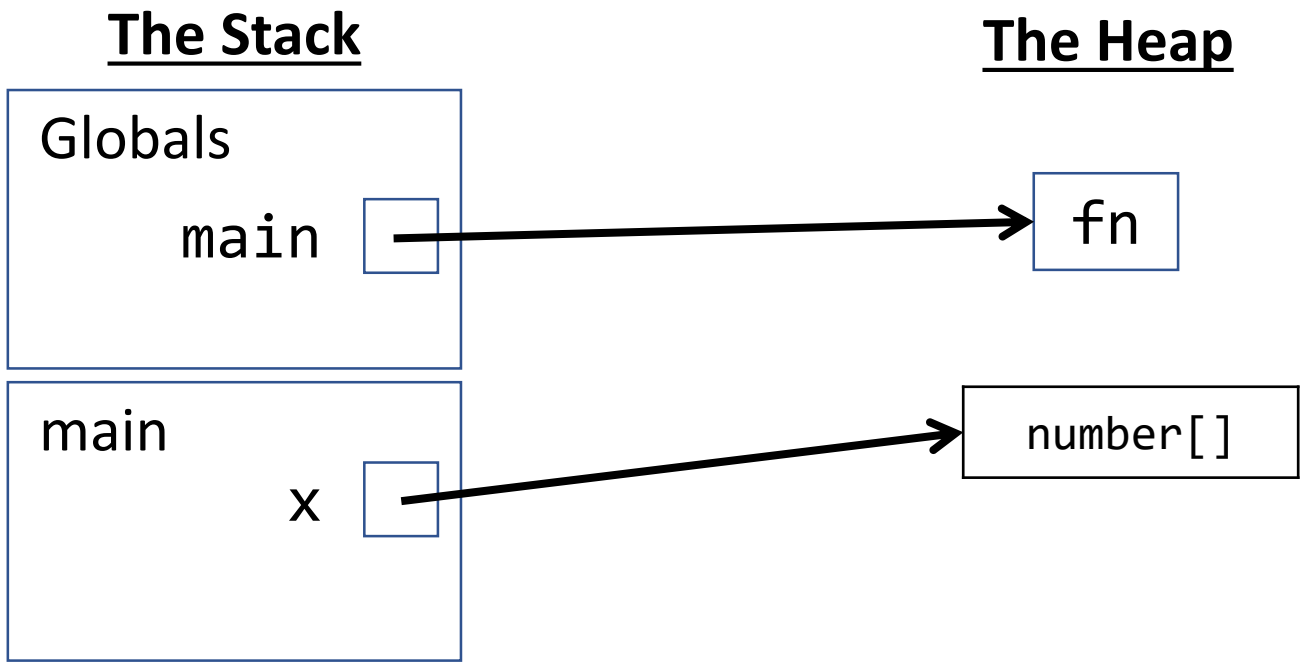
Variable Assignment - References (Arrays, Objects)

When a value stored on the heap is assigned to a **reference variable** on the stack, draw a **pointer* arrow** to it. ***All array values are stored on the heap.***

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



* If you are a computer science concentrator, starting to call these arrows by the name **pointers** now will serve you well in the future.

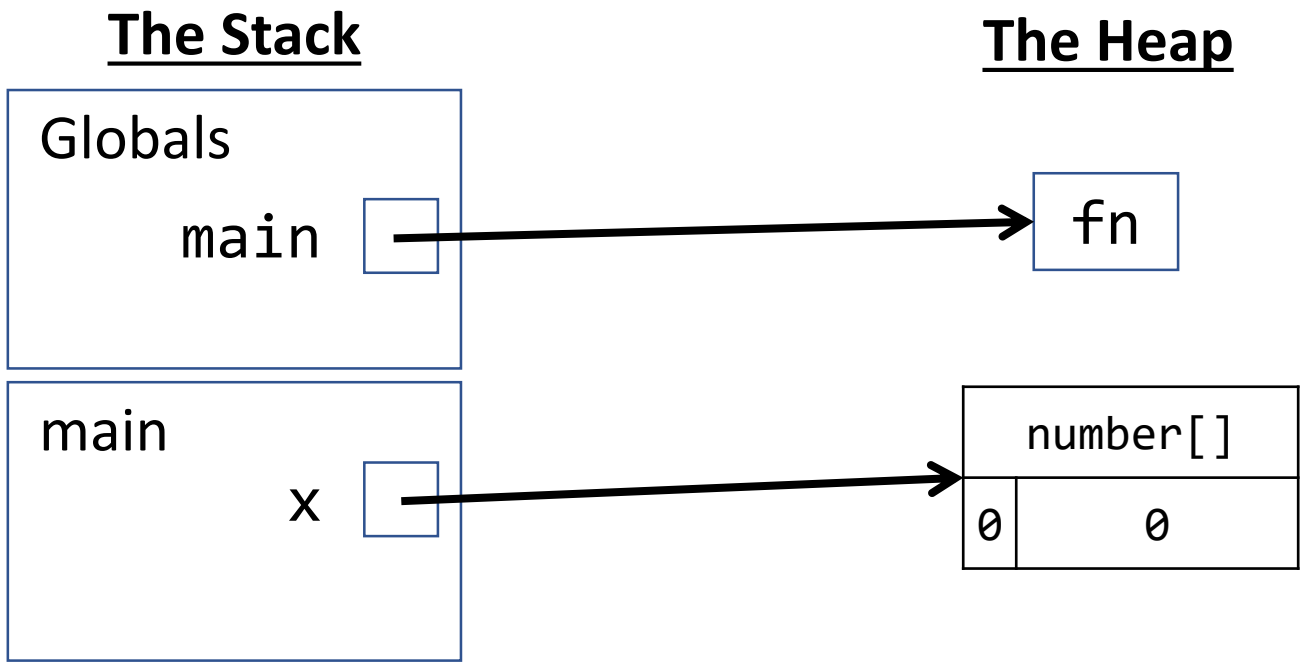
Array Element Assignment

When assigning to an element of an array, use name resolution to find the correct variable on the stack, then follow the pointer to its heap value to assign.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



* If you are a computer science concentrator, starting to call these arrows by the name **pointers** now will serve you well in the future.

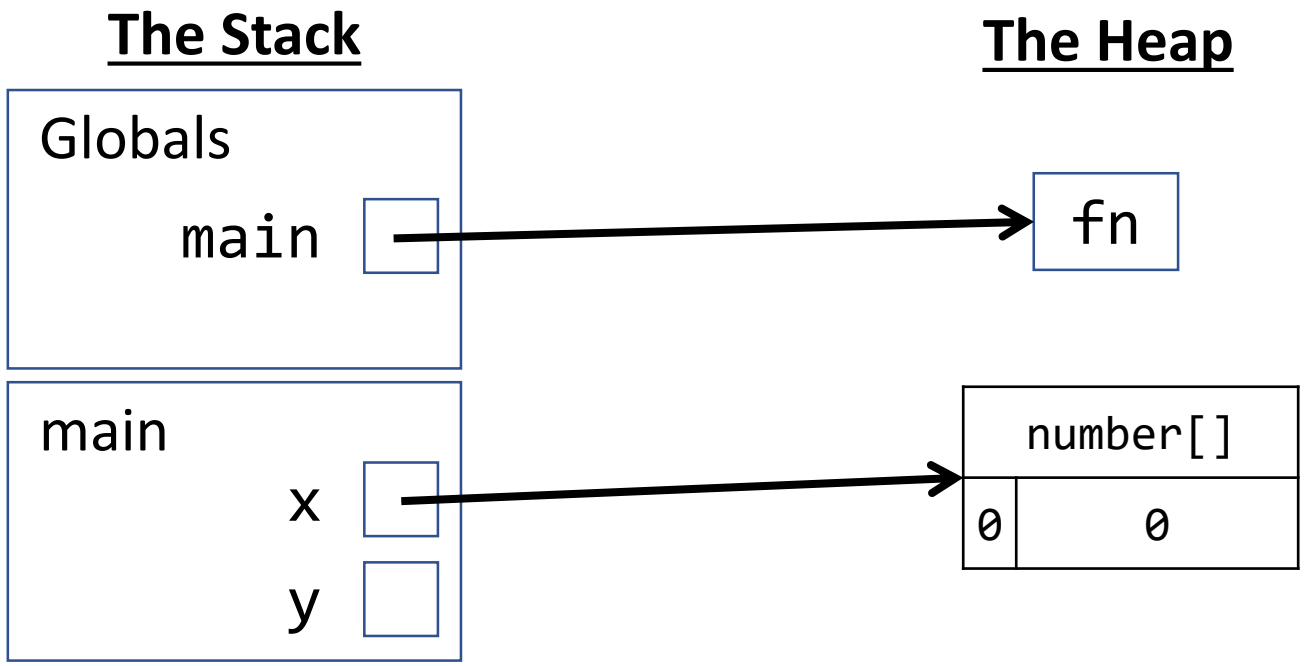
Variable Declaration

When a **variable** is declared, add its name to the current frame on the stack.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



* If you are a computer science concentrator, starting to call these arrows by the name **pointers** now will serve you well in the future.

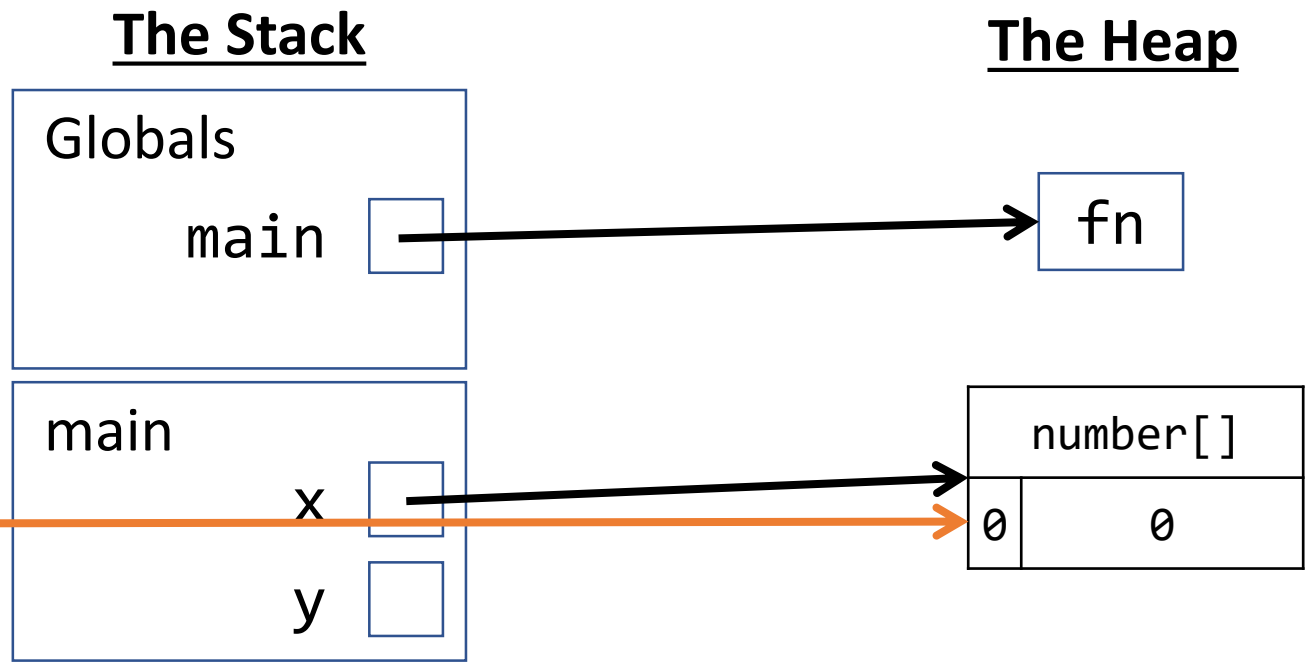
Name Resolution - Reference Variable Access

When a **reference variable** is accessed, find its name in the current frame on the stack. If it is not there, then look in globals. Substitute access with pointer*.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



* If you are a computer science concentrator, starting to call these arrows by the name **pointers** now will serve you well in the future.

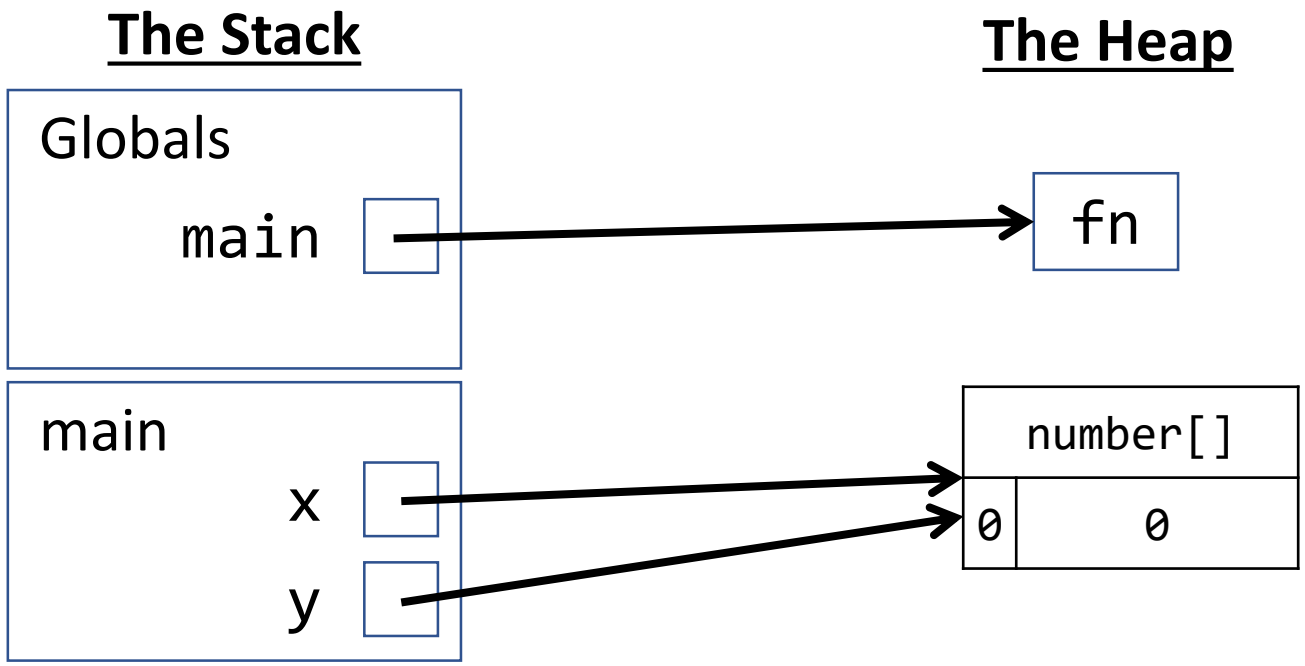
Variable Assignment - References (1 / 2)

When a value stored on the heap is assigned to a **reference variable** on the stack, draw a **pointer arrow *to the value on the heap***.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



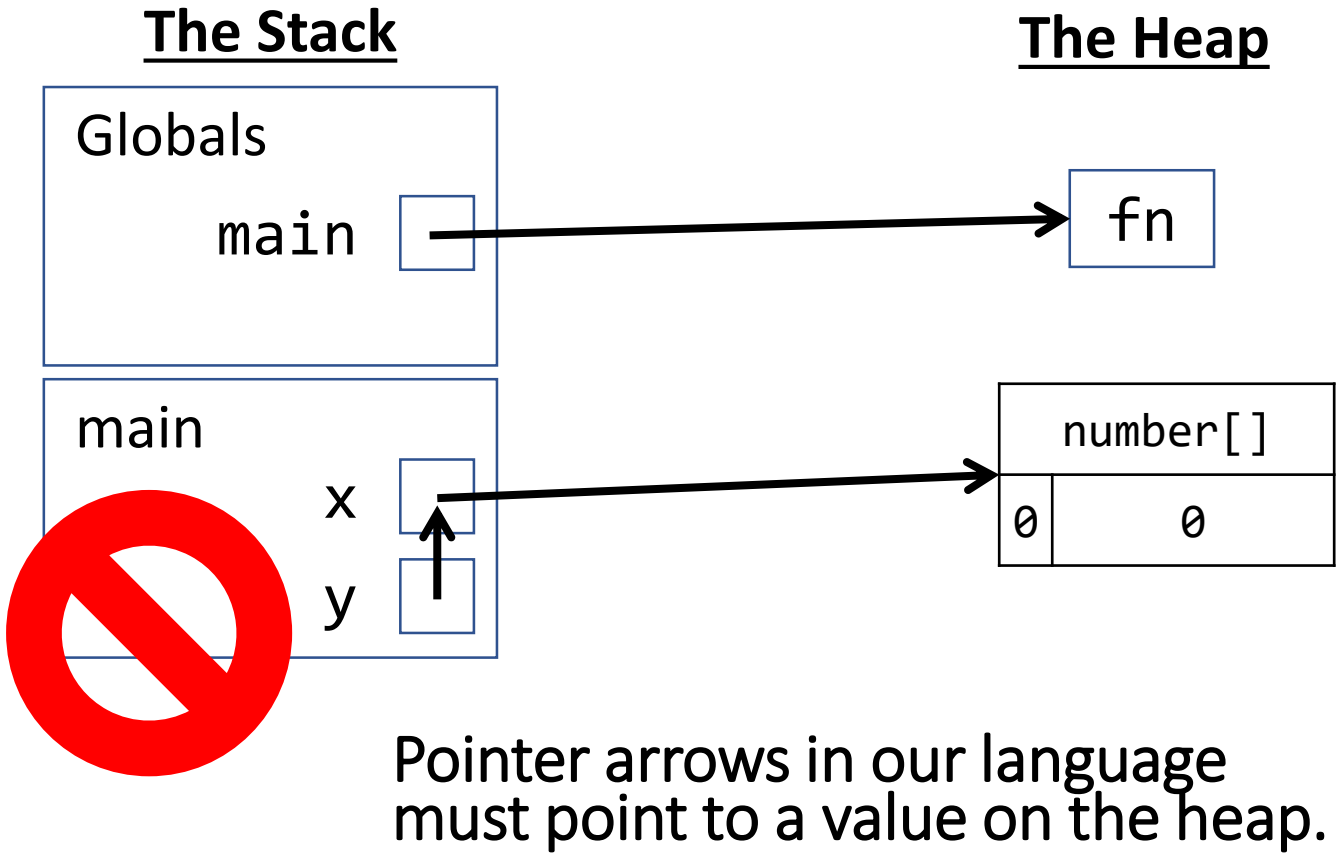
WARNING: Variable Assignment - References (2 / 2)

NEVER DO THIS!!!! You are not saying "y refers to x". You are saying "y refers to the same array on the heap that x also refers to," or, "y is assigned the same pointer value as x."

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



* This is true in Java, C#, and most "managed" languages as well. In other languages, like C and C++, it is possible to work with pointers to values on the stack.

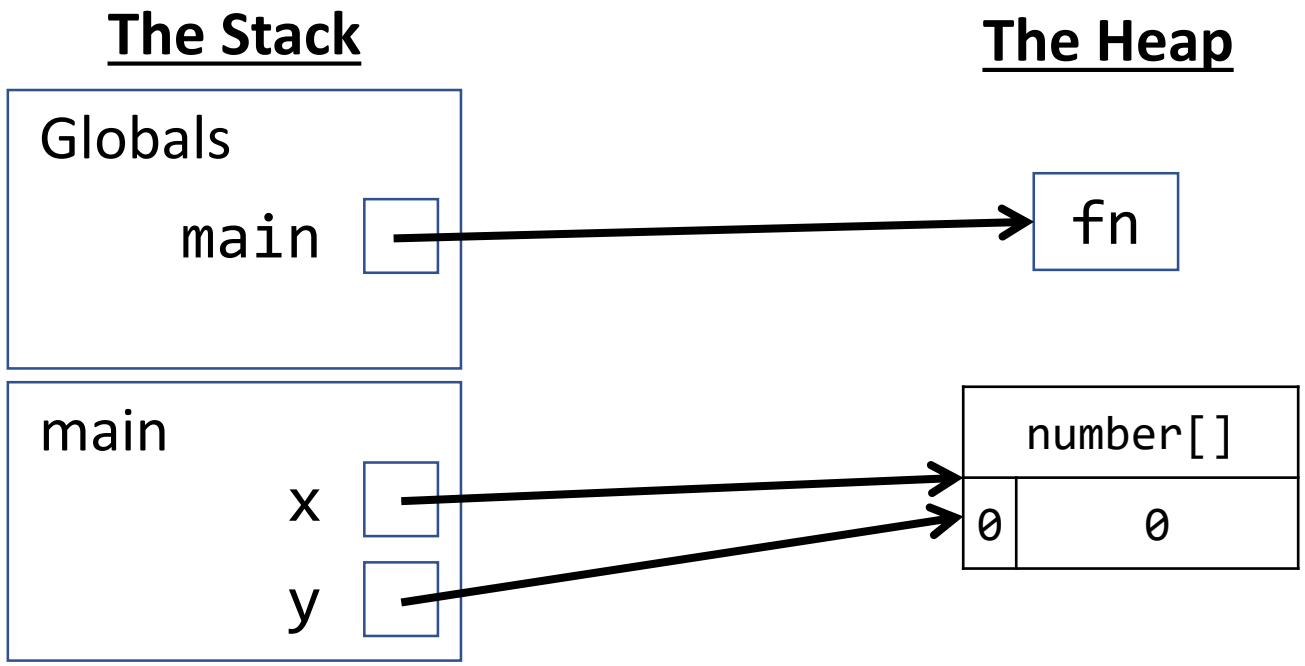
Array Index Acccess

When accessing to an element of an array by index, use name resolution to find the correct variable on the stack, follow the pointer, and lookup element at index.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



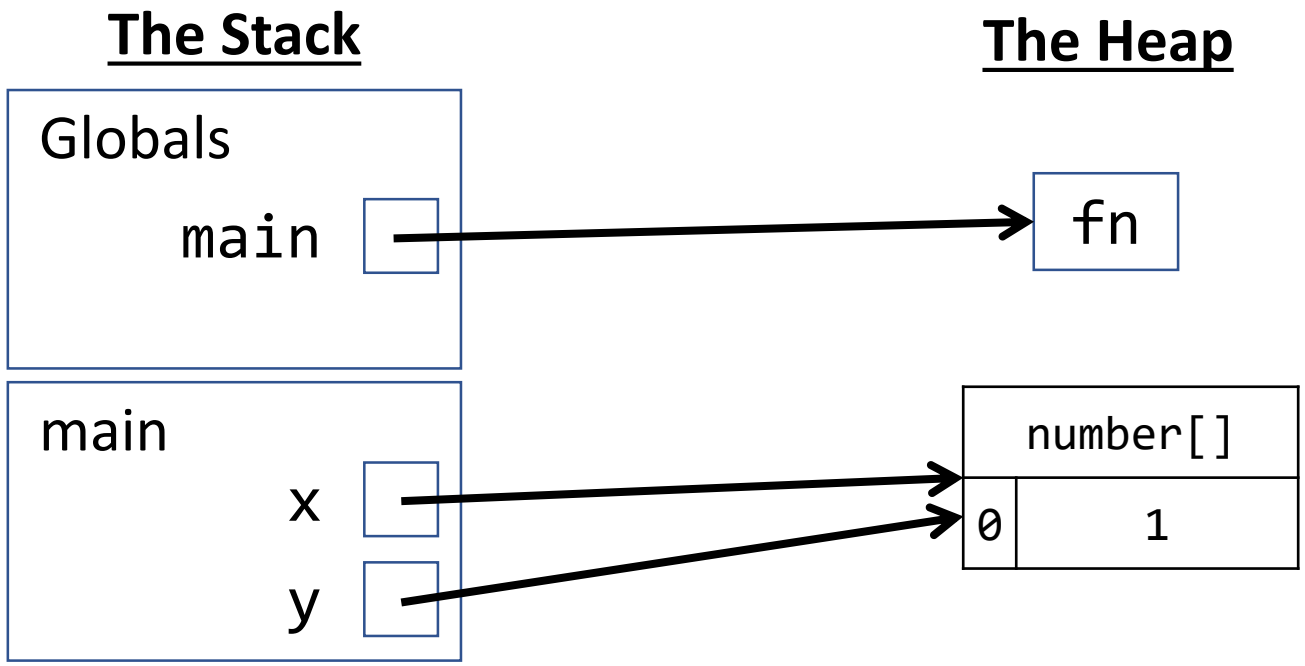
Array Element Assignment

When assigning to an element of an array, use name resolution to find the correct variable on the stack, then follow the pointer to its heap value to assign.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



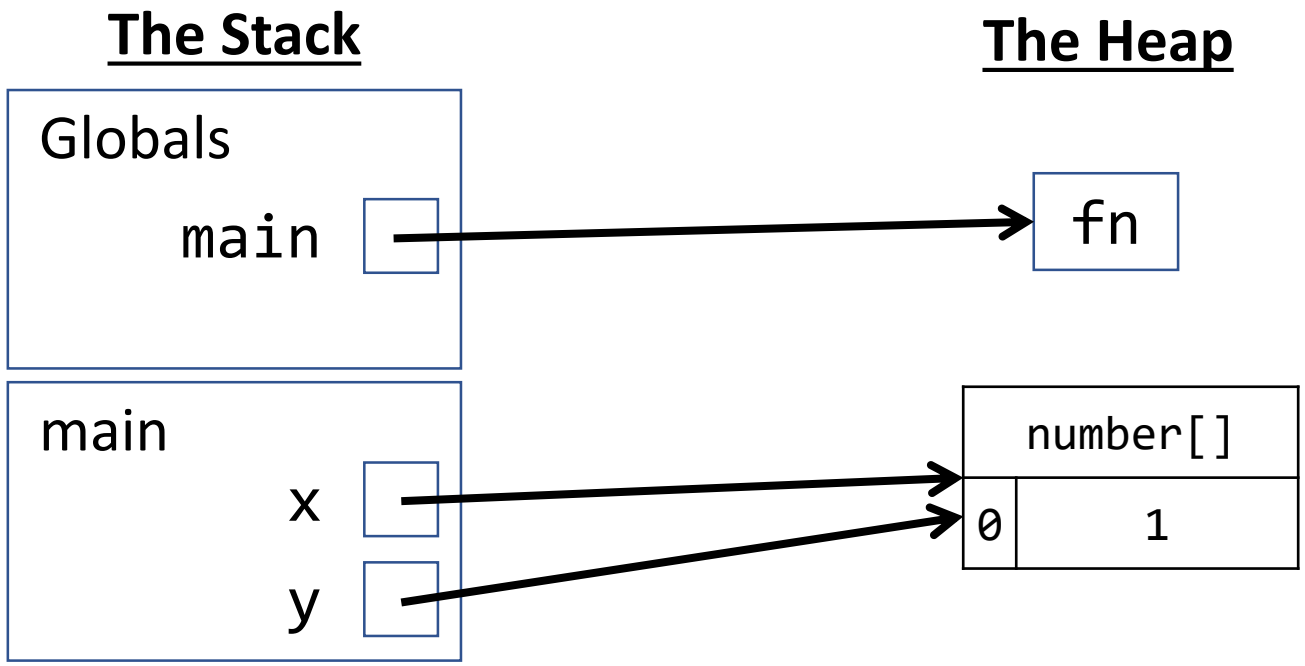
Array Index Acccess

When accessing to an element of an array by index, use name resolution to find the correct variable on the stack, follow the pointer, and lookup element at index.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



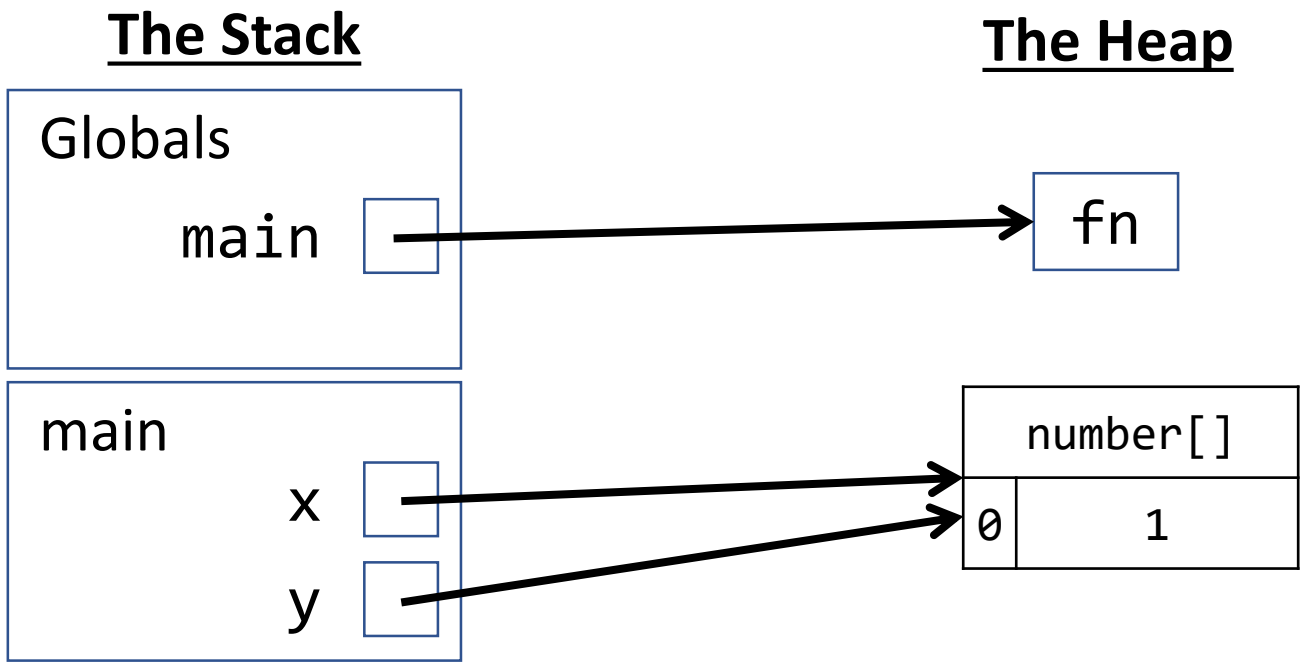
Array Index Acccess

When accessing to an element of an array by index, use name resolution to find the correct variable on the stack, follow the pointer, and lookup element at index.

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



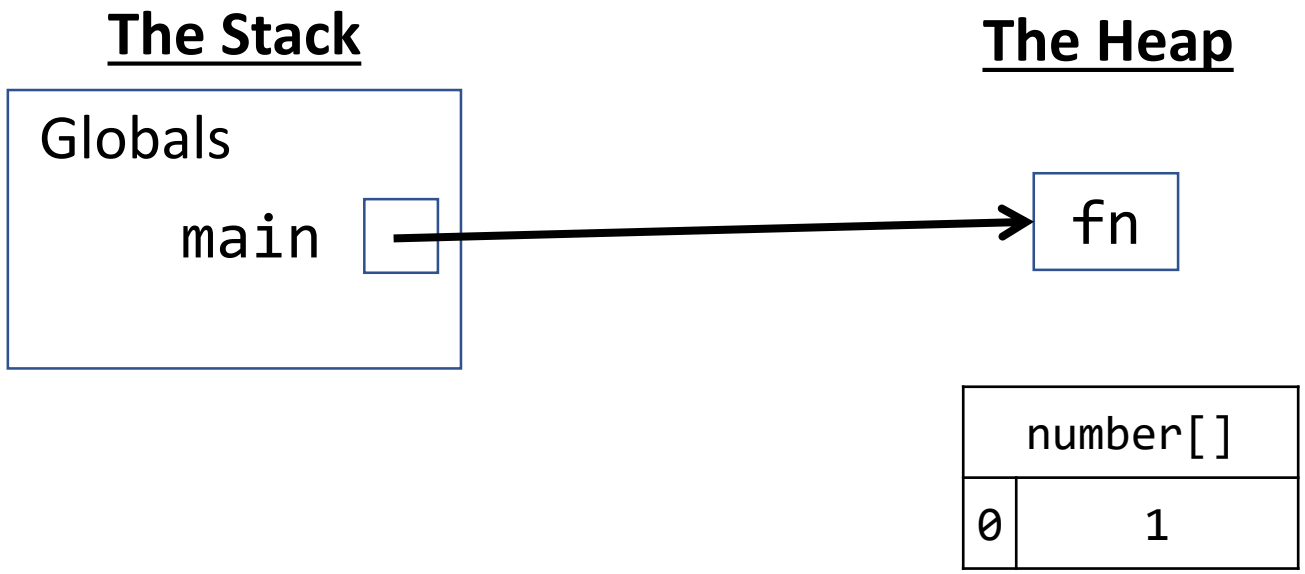
Function Return - `void` Functions

When a **void function** completes, remove its frame from the stack and return to where the function call was invoked... *but what about that array on the heap?*

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

export let main = async () => {
    let x: number[];
    x = [];
    x[0] = 0;
    let y: number[];
    y = x;
    x[0] = x[0] + 1;
    print("x[0]:" + x[0]);
    print("y[0]:" + y[0]);
};

main();
```



When there is a value on the heap and no path of pointers from the stack to that value exists, then at some point later a garbage collector will automatically remove it from the heap for you*.

* This is true in Java, C#, and most "managed" languages as well. In other languages, like C and C++, you must free/delete values from the heap yourself.

Case Study: Global Variables

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**.

```
let x: number;

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

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

main();
```

The Stack

The Heap

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

Variable Declaration

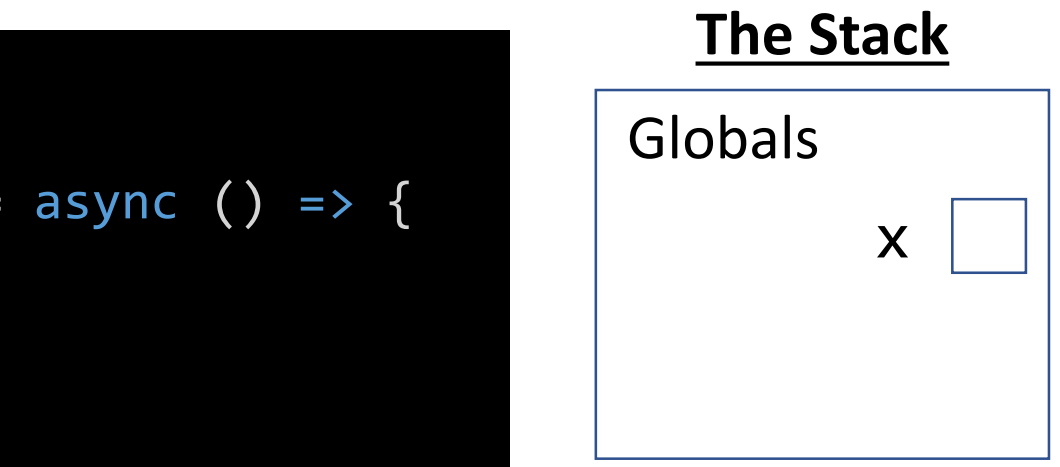
When a **variable** is declared, add its name to the current frame on the stack.
Since **x** is declared in the global frame it's called a **global variable**.

```
let x: number;

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

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

main();
```



Variable Declarations - Function

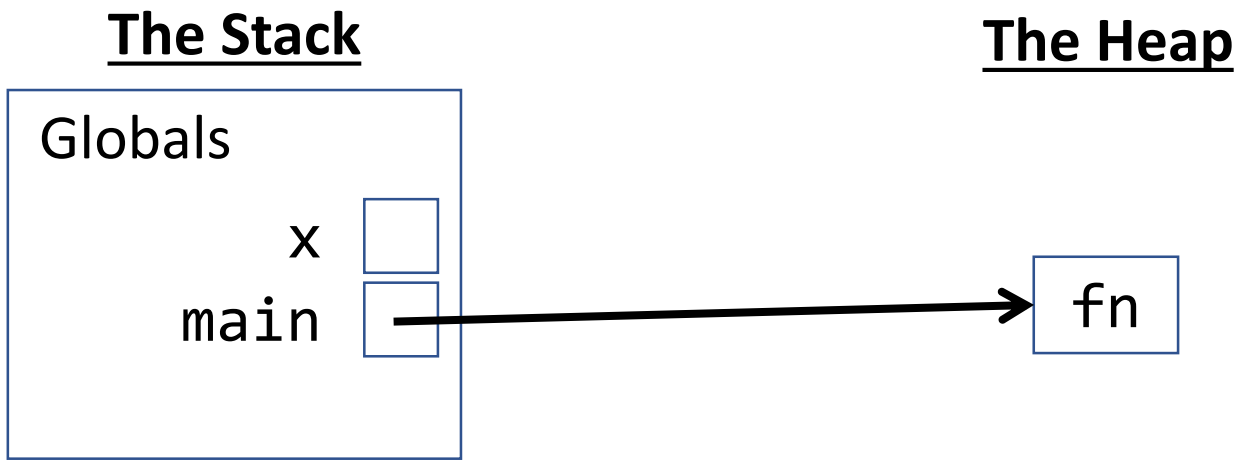
When a function definition is encountered, you will add its name to the current stack frame connected to a shorthand 'fn' symbol on the heap via pointer.

```
let x: number;

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

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

main();
```



Variable Declarations - Function

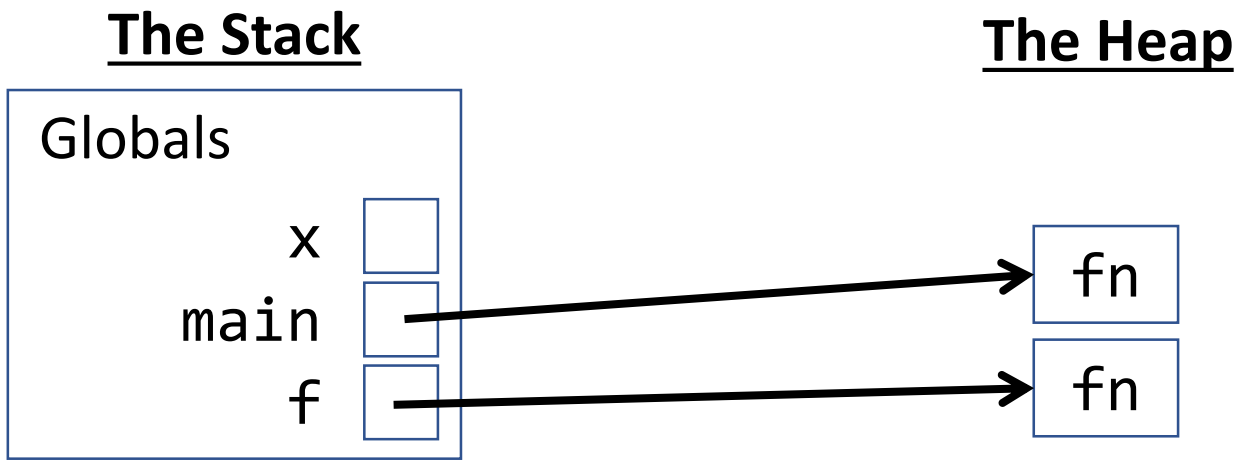
When a function definition is encountered, you will add its name to the current stack frame connected to a shorthand 'fn' symbol on the heap via pointer.

```
let x: number;

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

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

main();
```



Function Call

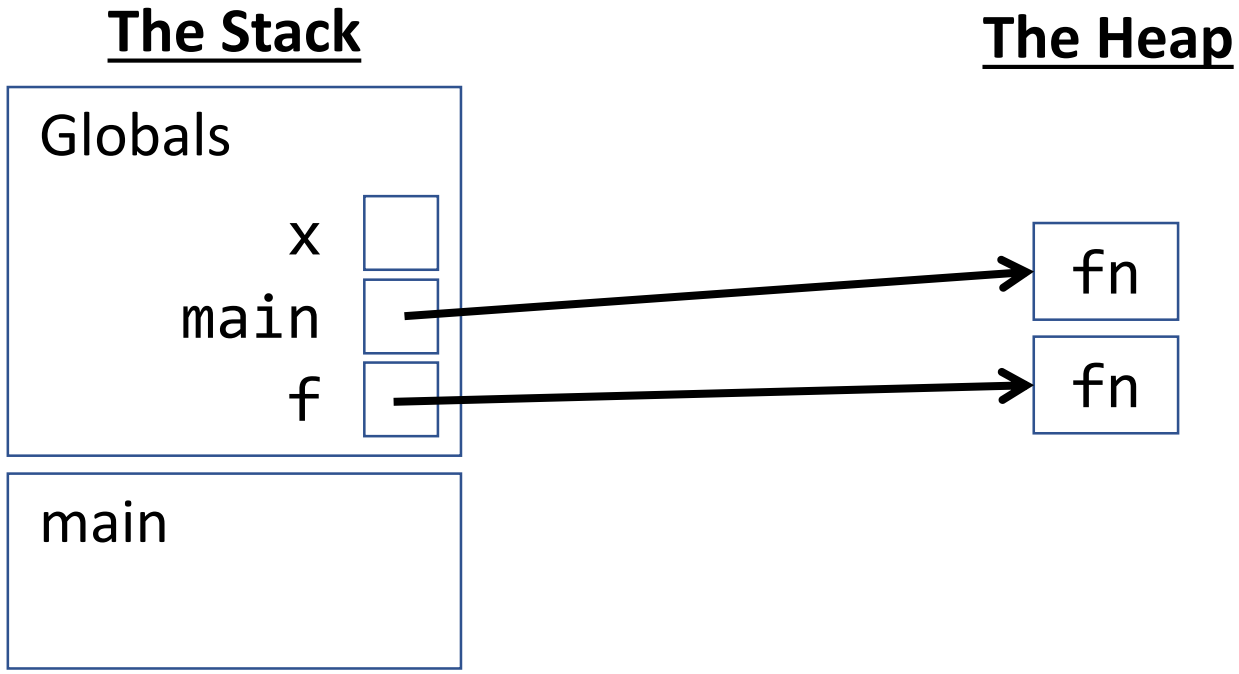
When a function call is encountered, a new **frame** is added to your stack. Label it with the function name. When it has parameters, you'll need to add them, too.

```
let x: number;

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

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

main();
```



Name Resolution - Variable Assignment

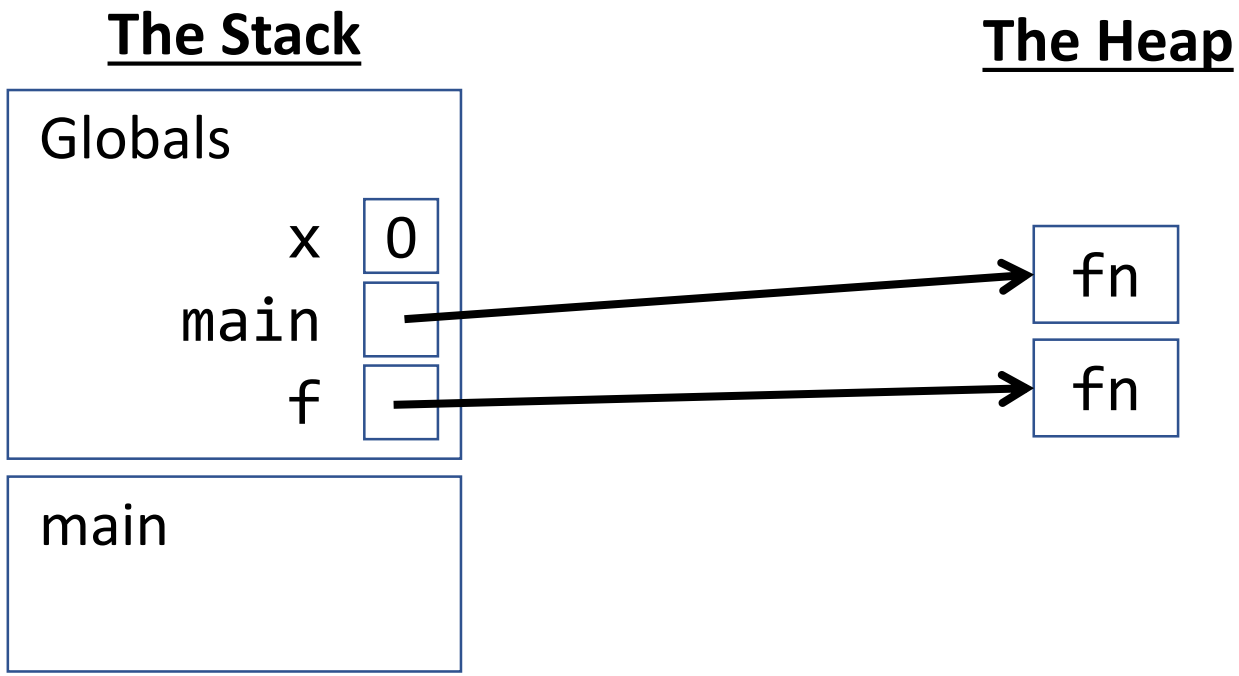
How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals. Primitive? Assign value in stack.

```
let x: number;

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

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

main();
```



Function Call

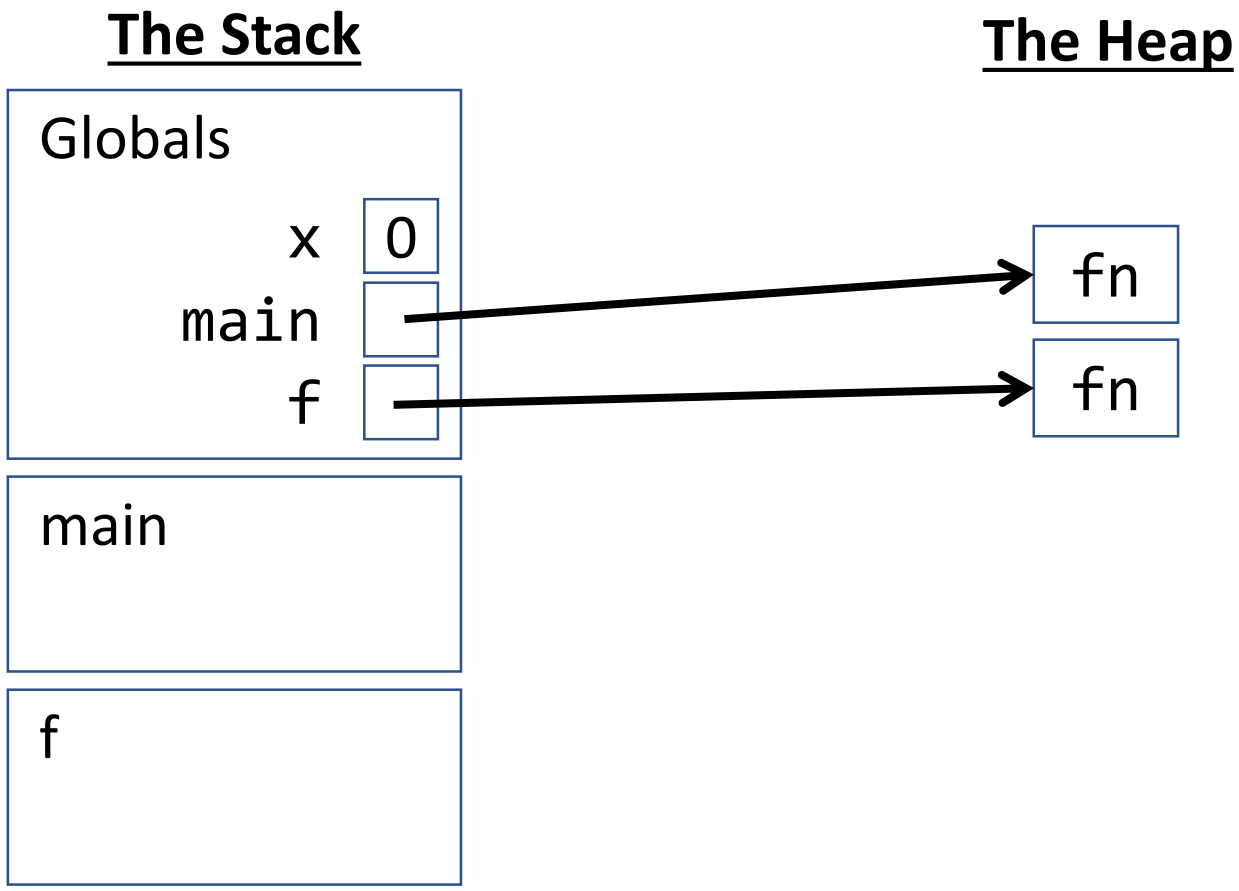
When a function call is encountered, a new **frame** is added to your stack. Label it with the function name. When it has parameters, you'll need to add them, too.

```
let x: number;

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

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

main();
```



Name Resolution - Variable Access

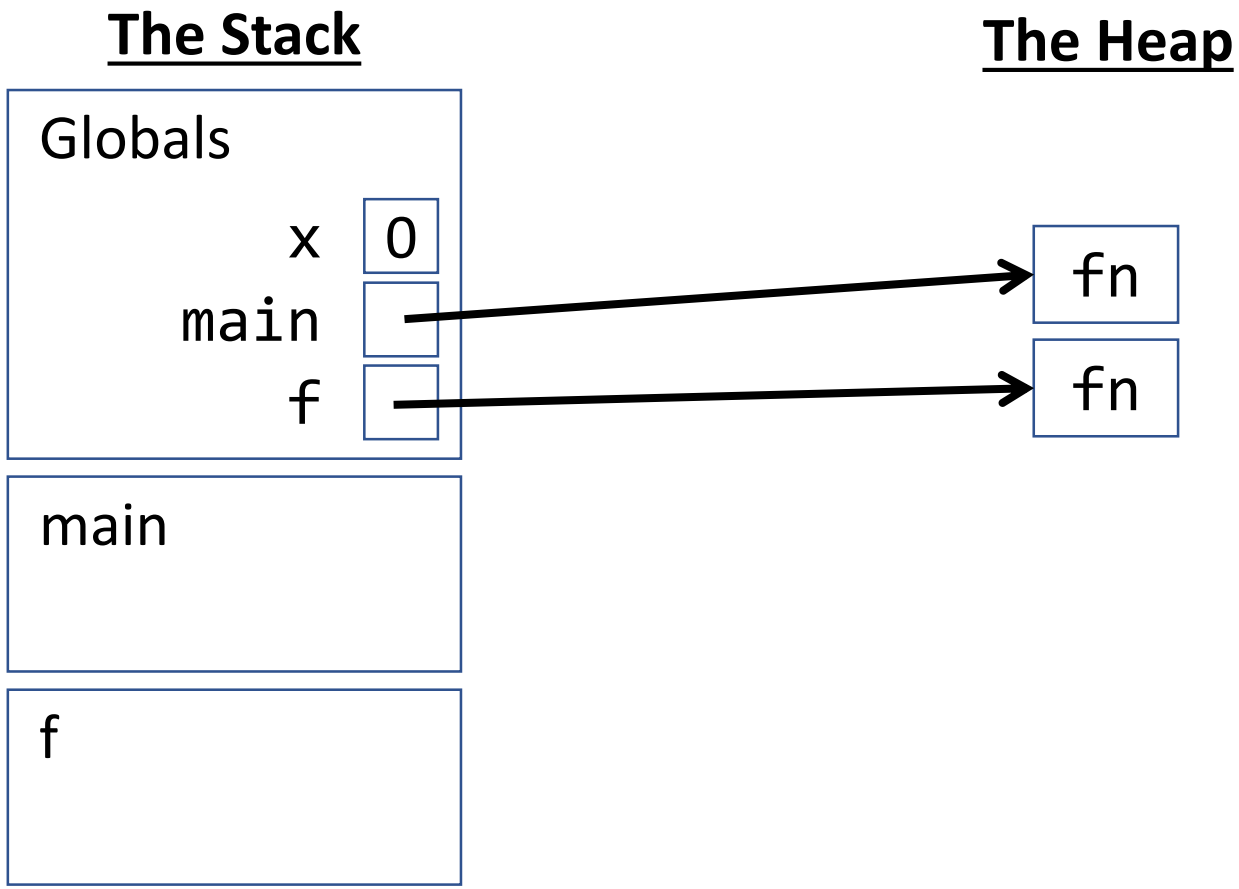
How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals frame. In this case, it's in globals!

```
let x: number;

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

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

main();
```



Name Resolution - Variable Assignment

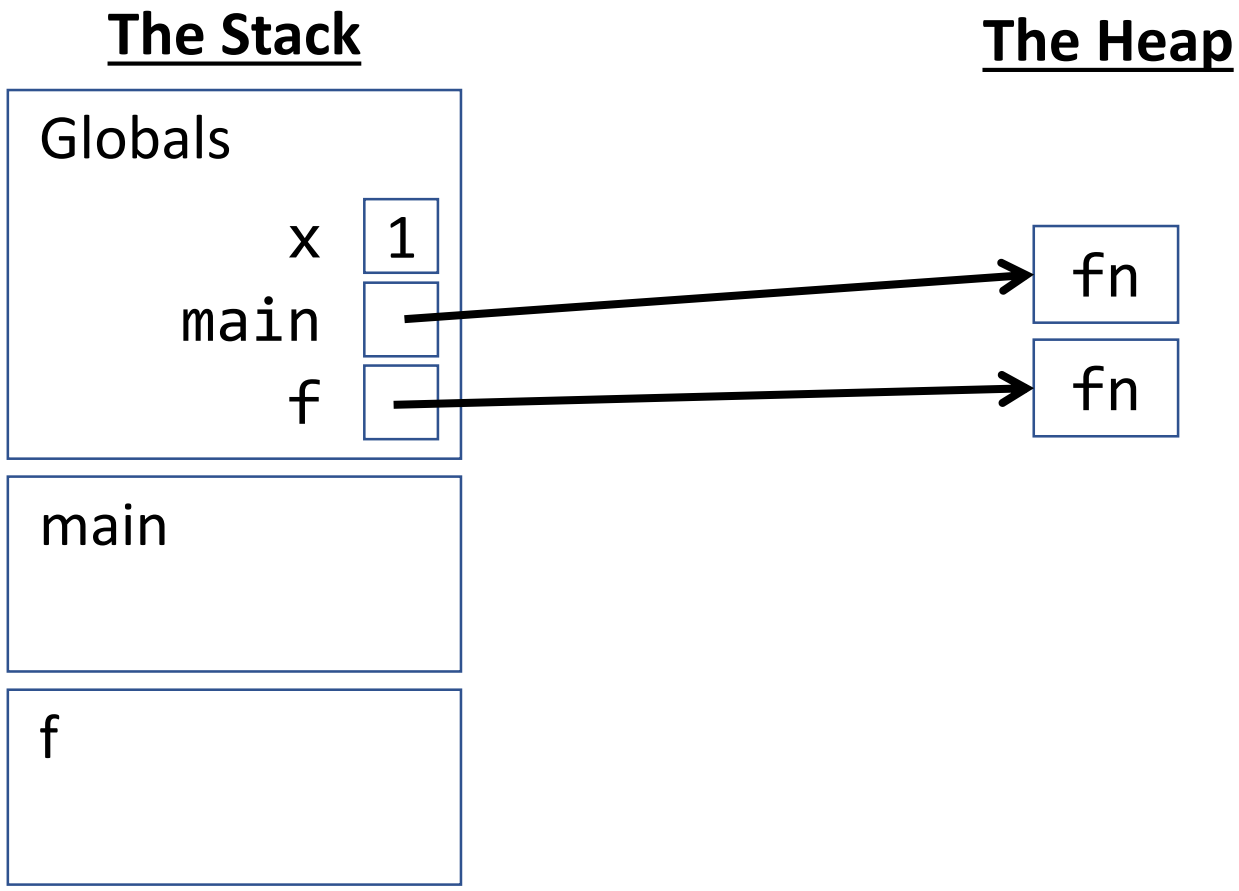
When a **primitive variable** is assigned a value, first resolve its frame location by name, then update its value.

```
let x: number;

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

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

main();
```



Function Return - `void` Functions

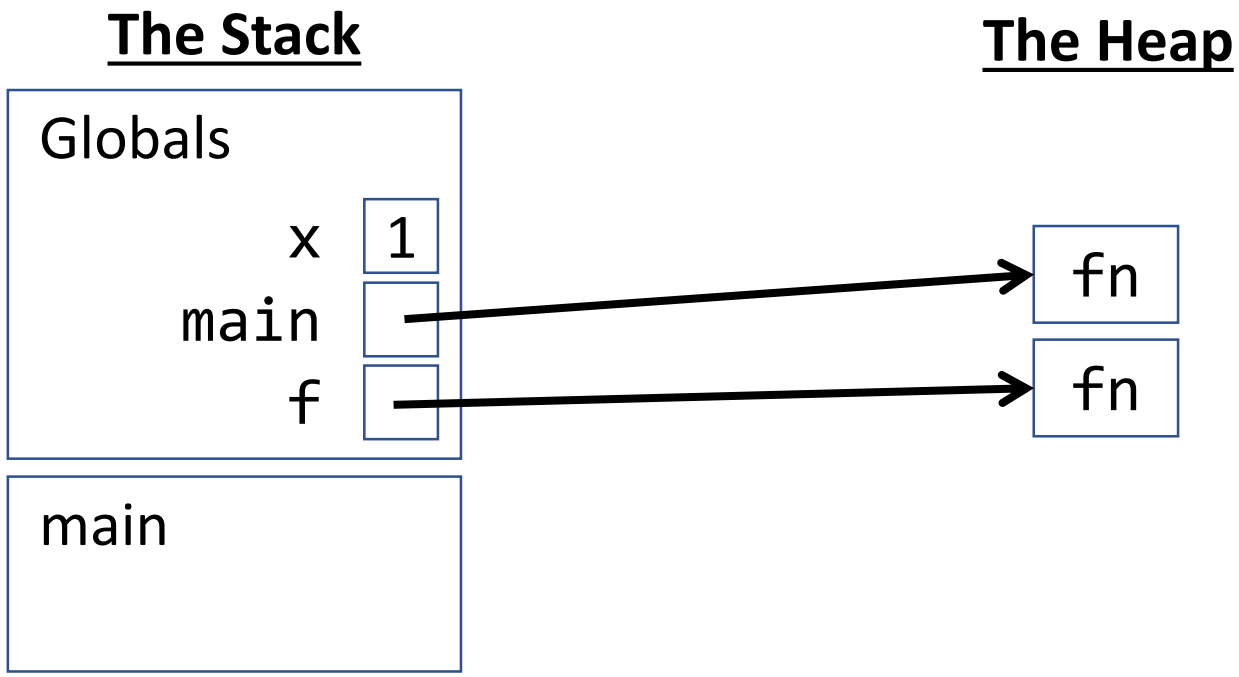
When a **void function** completes, remove its frame from the stack and return to where the function call was invoked.

```
let x: number;

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

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

main();
```



Name Resolution - Variable Access

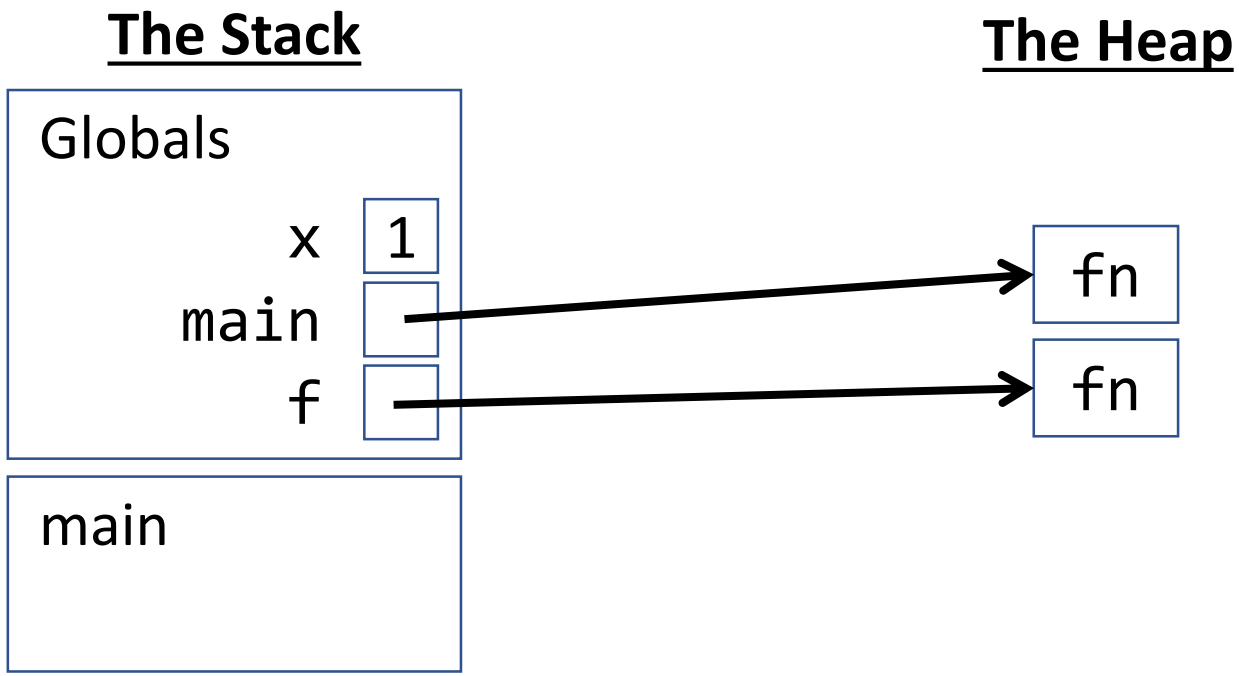
How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals frame. In this case, it's in globals!

```
let x: number;

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

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

main();
```



Function Return - `void` Functions

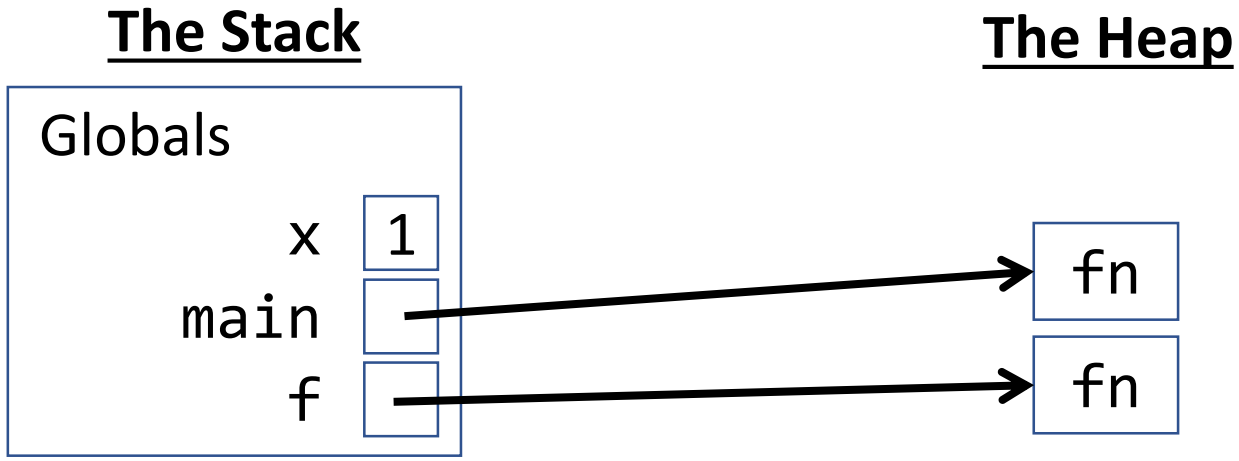
When a **void function** completes, remove its frame from the stack and return to where the function call was invoked.

```
let x: number;

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

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

main();
```



Case: Primitive Parameters

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**.

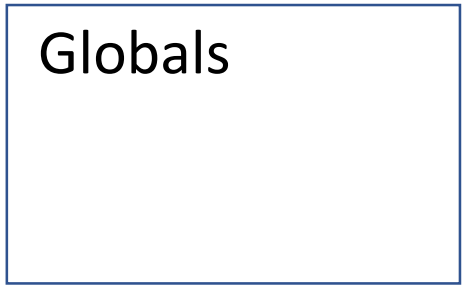
```
import { print } from "introcs";

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

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

main();
```

The Stack



The Heap

* This is a lie. The interpreter has established built-in functions, variables, and classes in its own stack and heap space *before* our program loads. Not our concern.

Globals - Imports

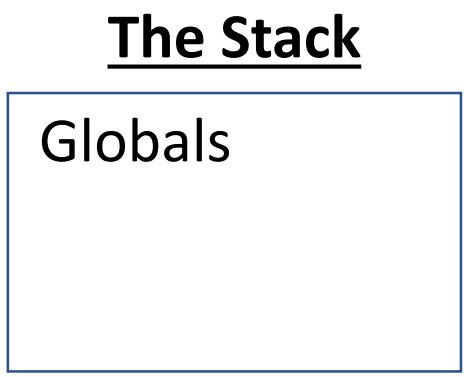
When a function, class, or value is imported from another file, *technically* it is entered into the Globals frame. However, **we will skip this step in our diagrams**.

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

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

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

main();
```



Variable Declaration - Function (1 / 2)

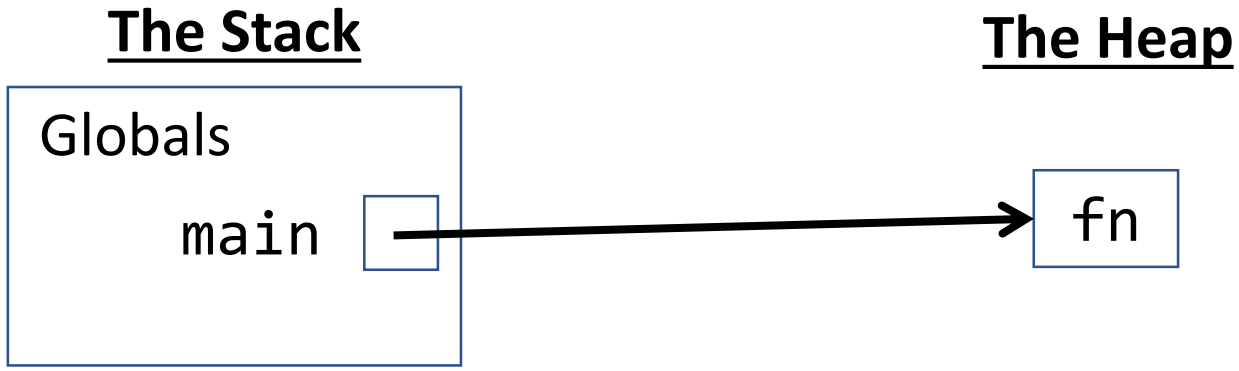
When a function definition is encountered, you will add its name to the current stack frame connected to a shorthand 'fn' symbol on the heap via pointer.

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

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

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

main();
```



Variable Declaration - Function (2 / 2)

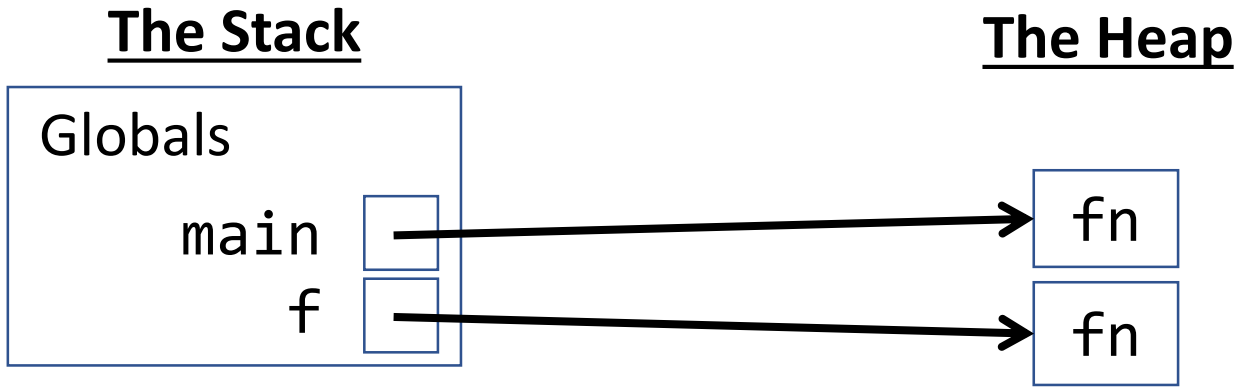
Wait! *A function definition is actually variable definition? Its code is stored in memory with other variables?* Yes*!!! We will explore these ideas soon.

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

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

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

main();
```



* This is true in programming languages with *first-class functions.* Most modern programming languages feature first-class functions or "functions as values".

Function Call

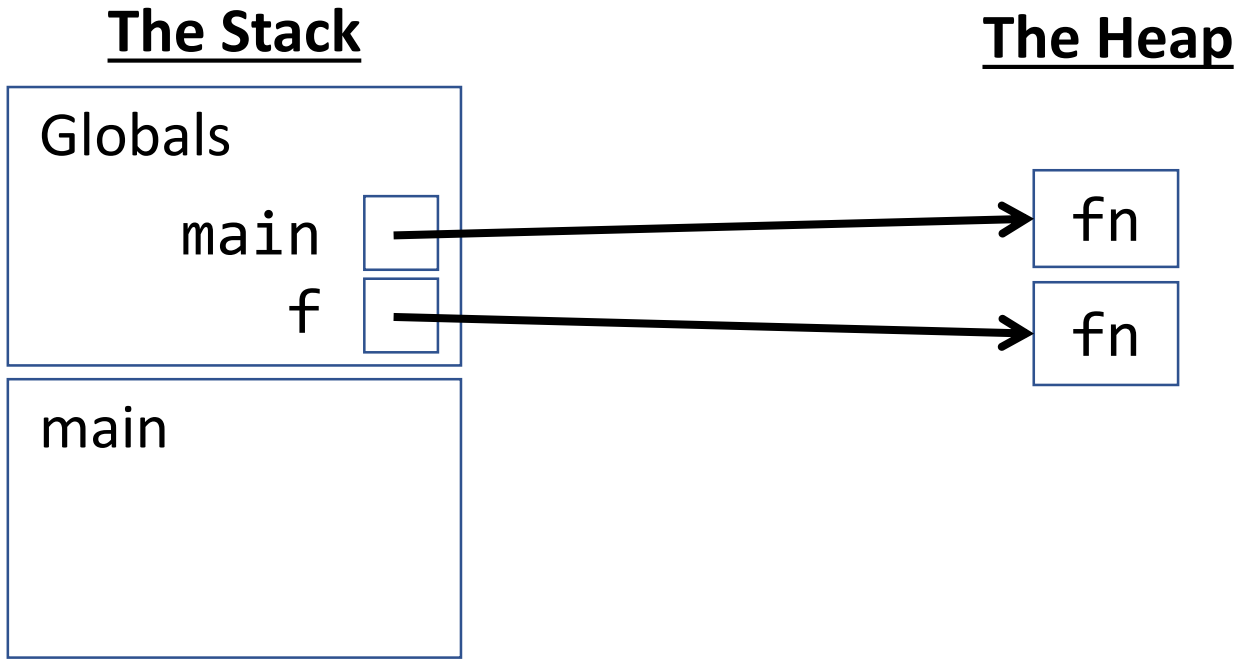
When a function call* is encountered, a new **frame** is added to your stack. Label it with the function name. When it has parameters, you'll need to add them, too.

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

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

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

main();
```



* The last line of your programs all along has been a simple function call to the **`main`** function! *This* is why **`main`** is our programs' starting point.

Variable Declaration

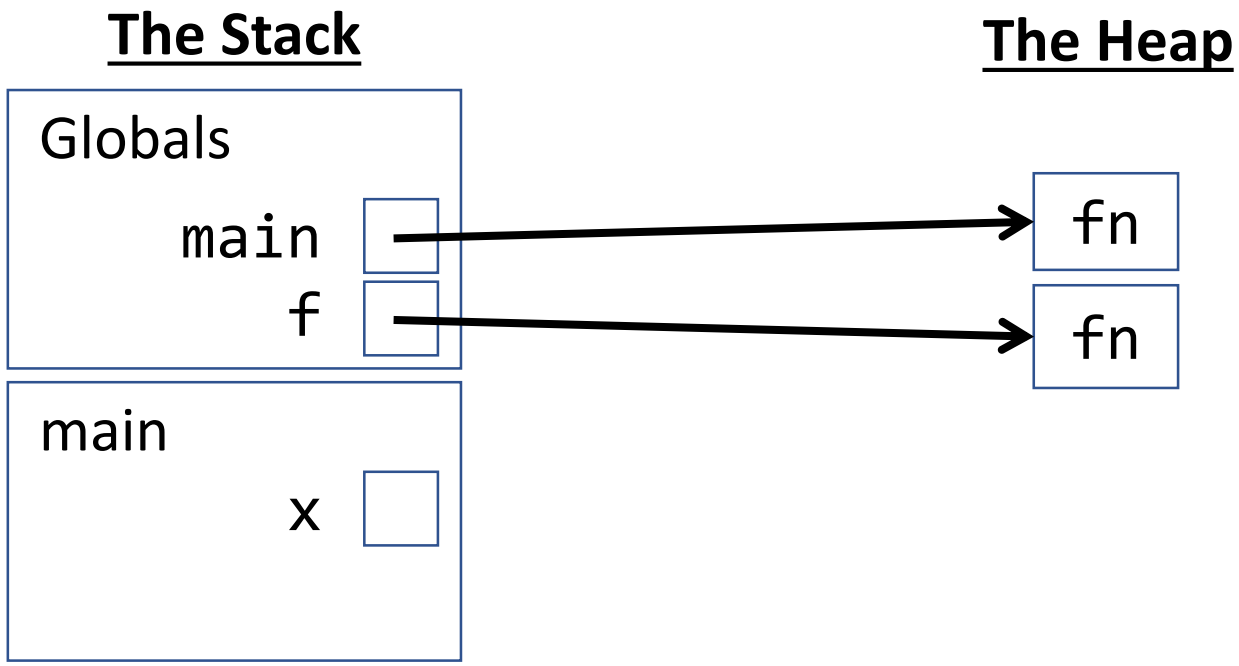
When a **variable** is declared, add its name to the current function's frame.

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

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

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

main();
```



Variable Assignment - Primitives (number, boolean, string)

When a **primitive variable** is assigned a value, update its value in its stack frame.

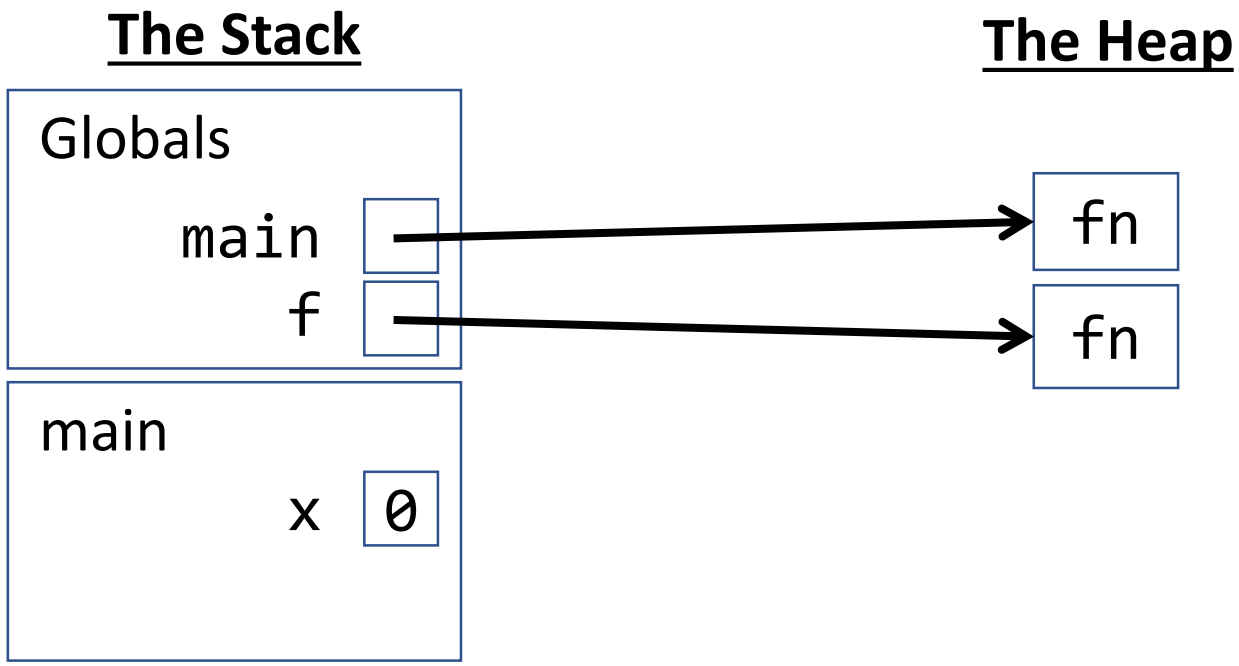
Primitive variables' values are stored on the stack.*

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

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

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

main();
```



* Array and object variables' values, you will soon learn, *are stored on the heap*.

Name Resolution - Variable Access

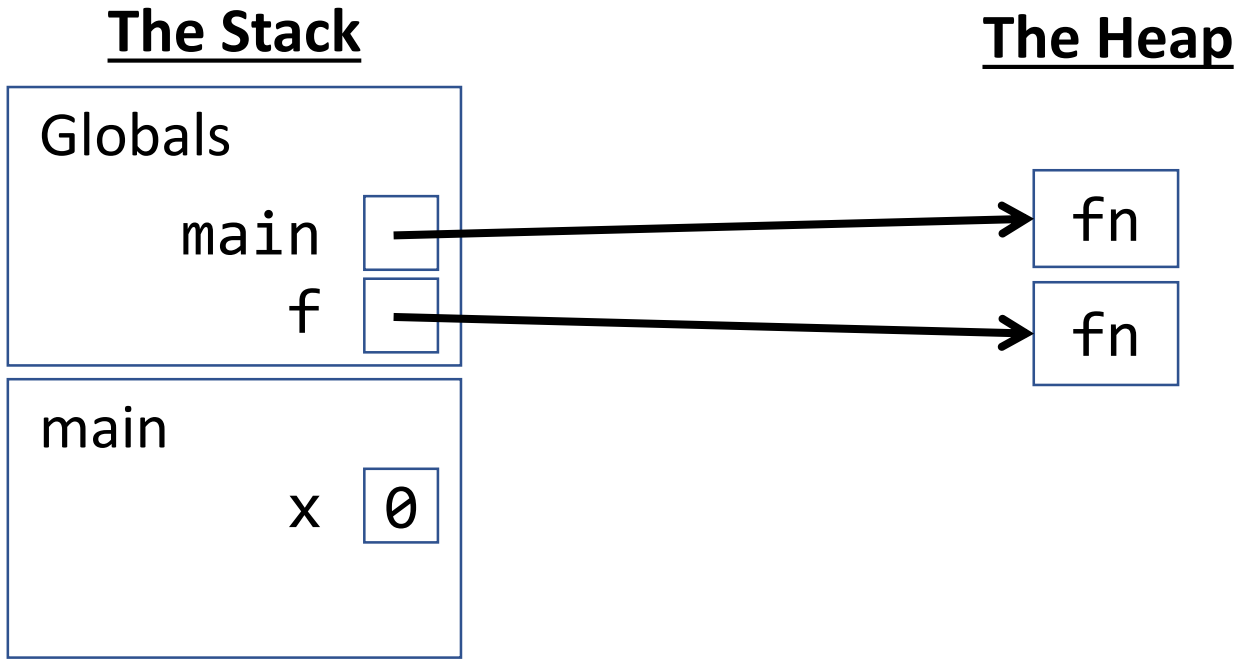
How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals frame. In this case, it's a number!

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

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

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

main();
```



* It's a matter of coincidence that **f**'s parameter's name is also **x**. Remember, parameter names are only known and scoped inside of a function.

Name Resolution - Function Call

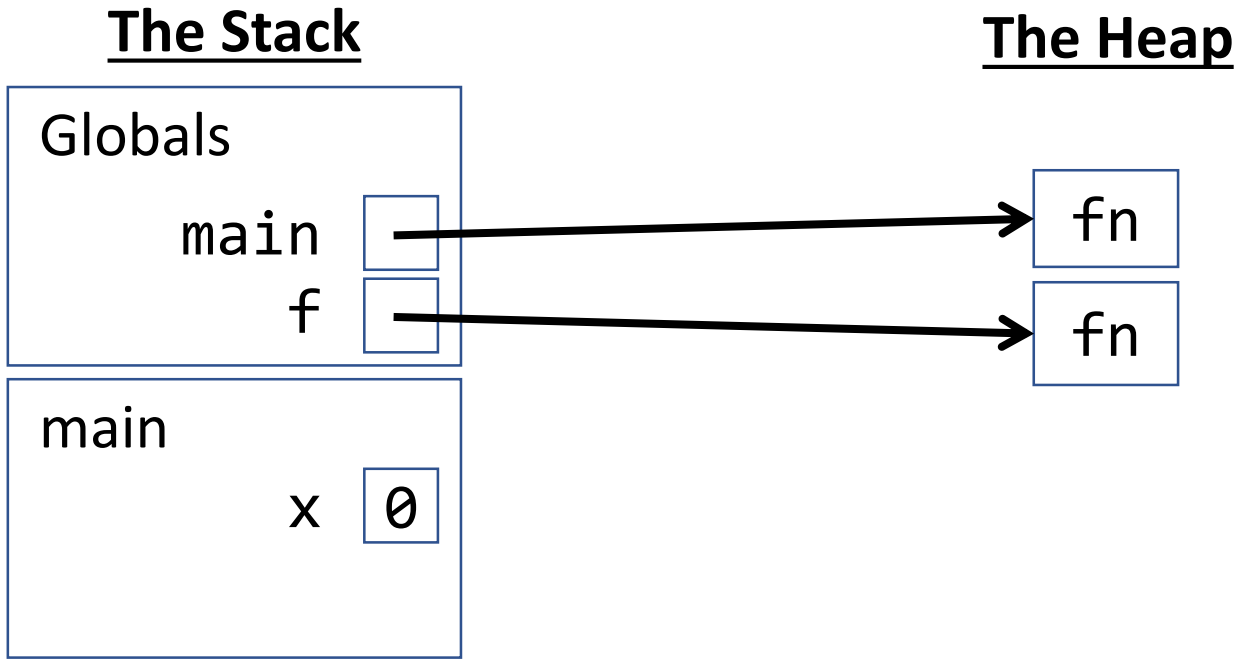
How do you know what the name **f** is? First look for a name in the current stack frame. If not there, then look in the globals frame. In this case, it's a function!

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

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

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

main();
```



* It's a matter of coincidence that **f**'s parameter's name is also **x**. Remember, parameter names are only known and scoped inside of a function.

Function Call *with Arguments*

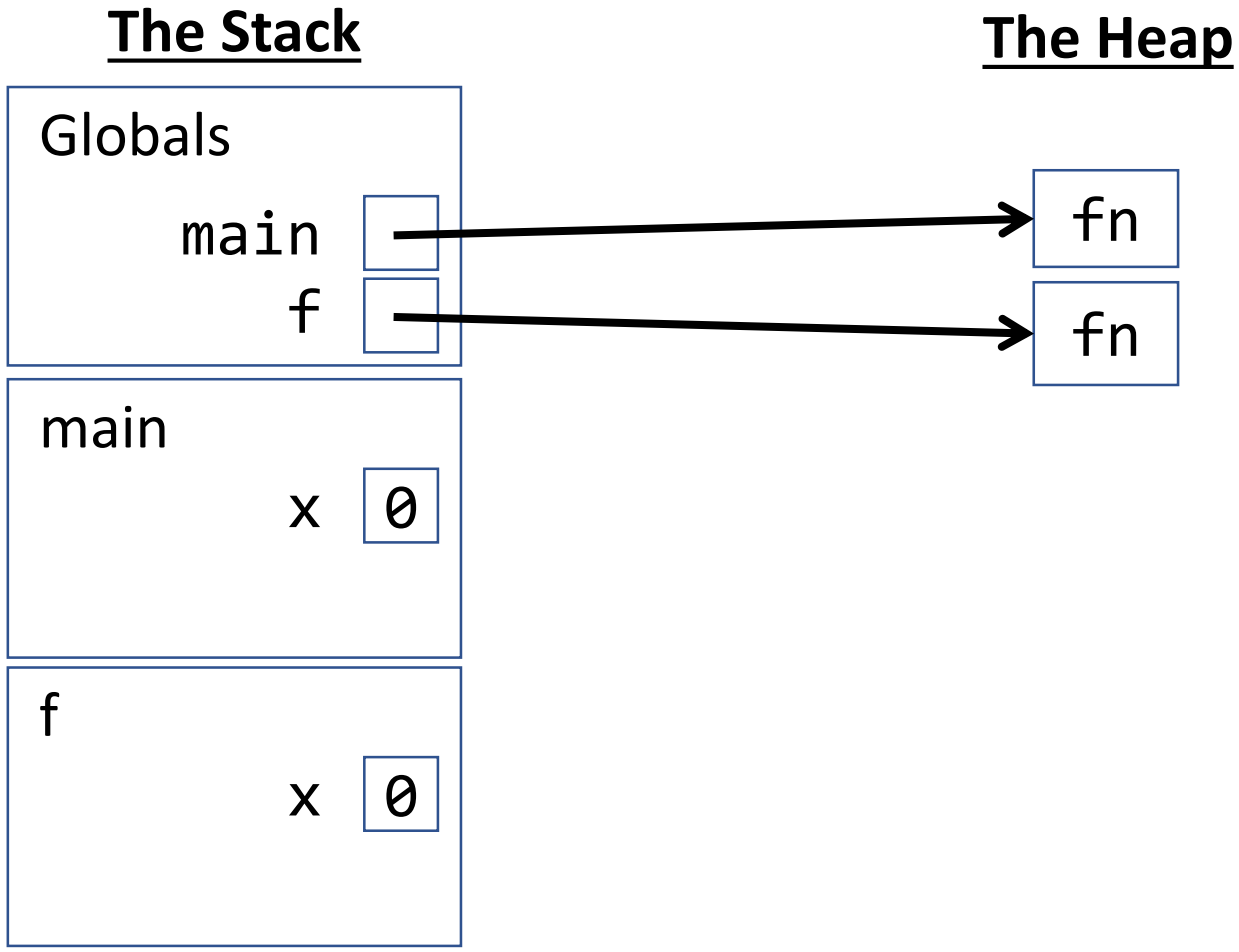
When a call to a function with parameters is made, its frame is added to the stack and **each parameter is added to the frame with its argument's value***.

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

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

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

main();
```



* It's a matter of coincidence that **f**'s parameter's name is also **x**. Remember, parameter names are only known and scoped inside of a function.

Name Resolution - Variable Access

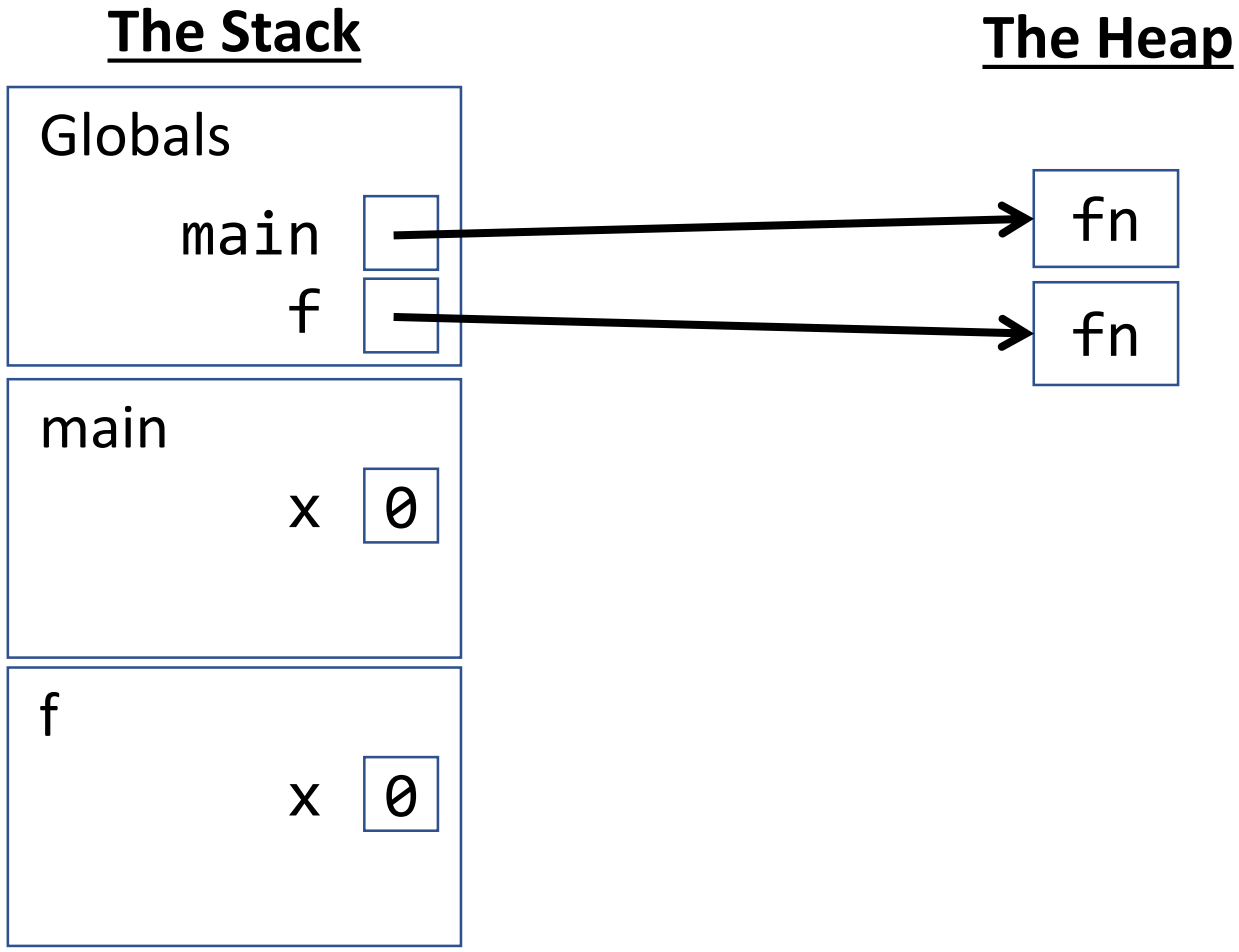
How do you know which **x**? First look for a name in the current stack frame. If not there, then look in the globals frame. In this case, it's the **x** in **f**'s frame.*

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

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

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

main();
```



Variable Assignment - Primitives (number, boolean, string)

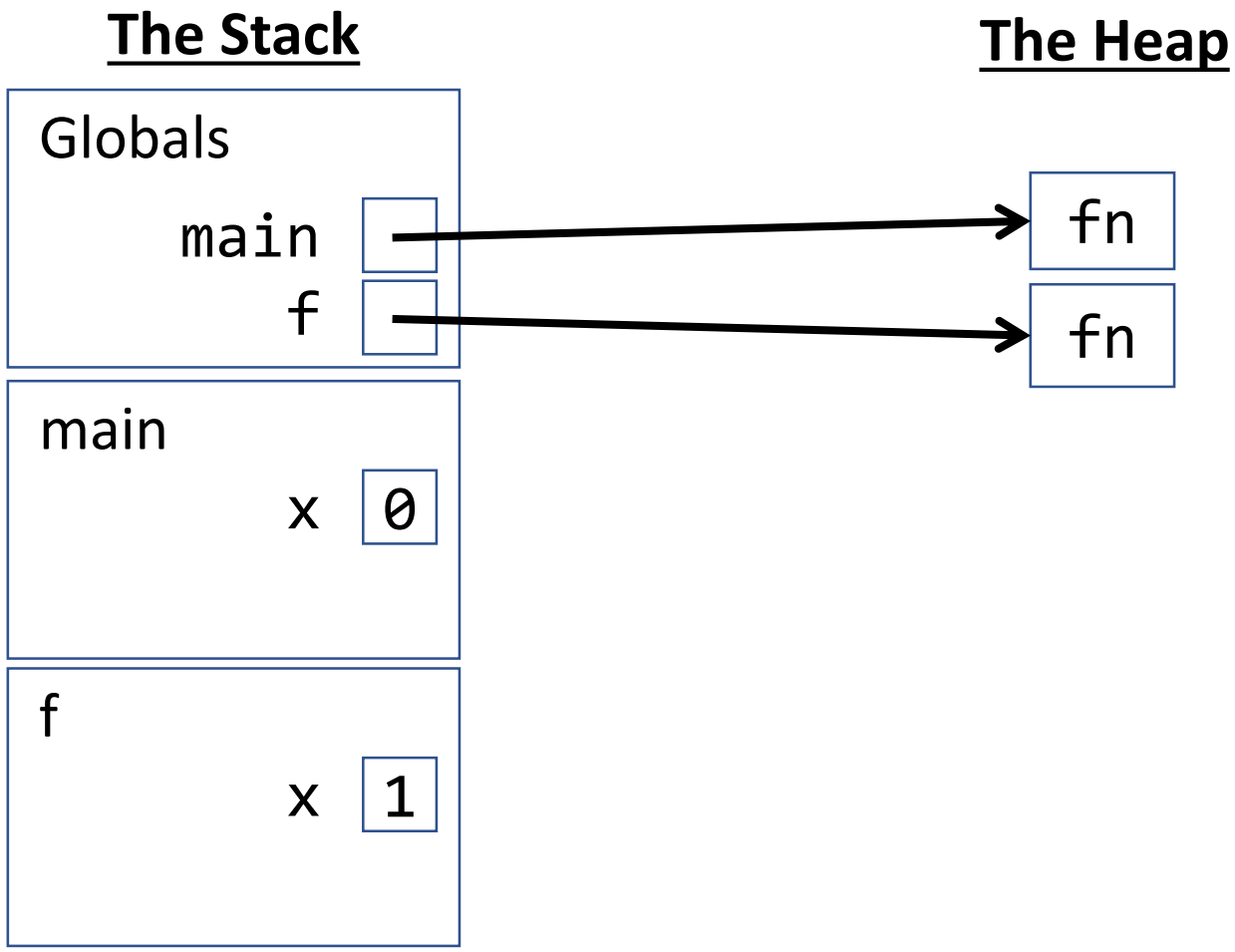
When a **primitive variable** is assigned a value, first resolve its location by name, then update its value.*

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

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

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

main();
```



* Notice you *did not change **x**'s value in the main function's frame.*

Function Return - `void` Functions

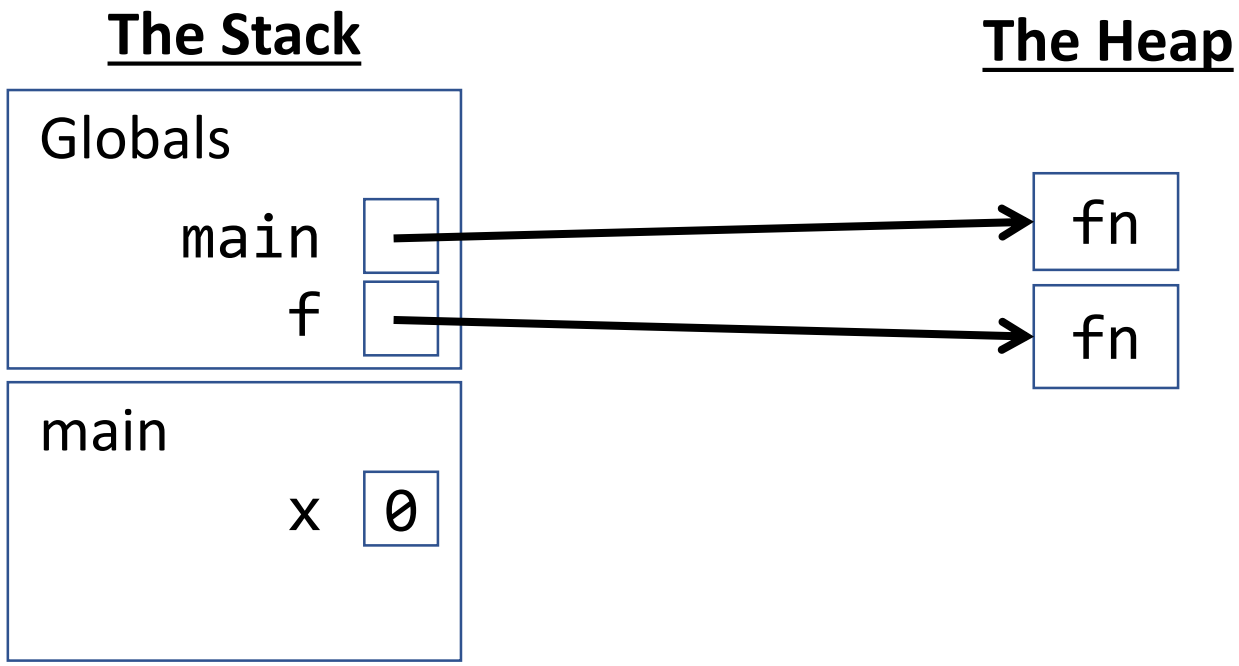
When a **void function** completes, remove its frame from the stack and return to where the function call was invoked.

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

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

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

main();
```



Name Resolution - Variable Access

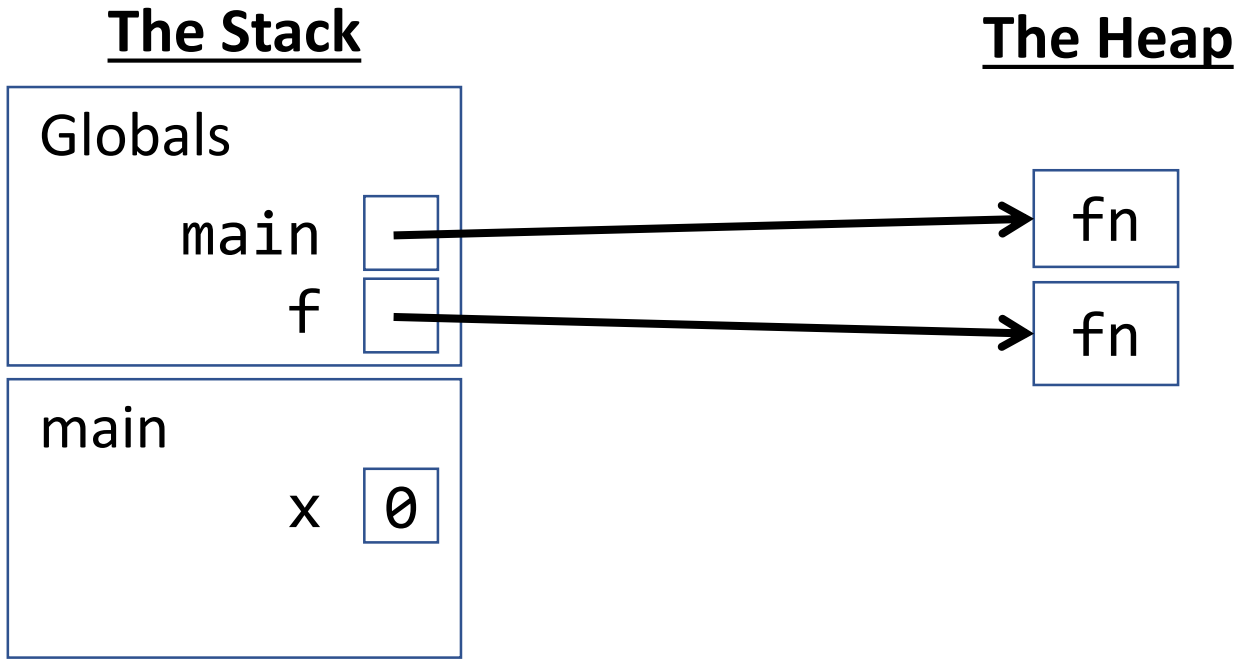
First look for a name in the current stack frame. If not there, then look in the globals frame. In this case, it's the **`x`** in **`main`**'s frame.

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

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

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

main();
```



Function Return - `void` Functions

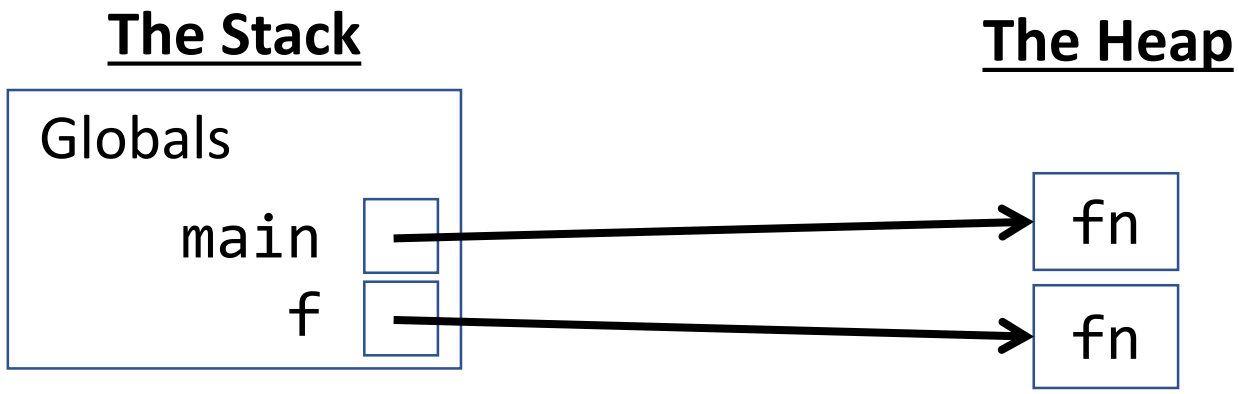
When a **void function** completes*, remove its frame from the stack and return to where the function call was invoked.

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

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

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

main();
```



* `main`'s `async` flag makes it a special kind of function that allows us to `await` prompts from within it. We can think of it as a `void` function, though! If you were to add a `print("Goodbye");` statement after the call to `main();`, you would see the interpreter will continue on in our code past this call.

Case: Reference Parameters

Globals - Starting Point

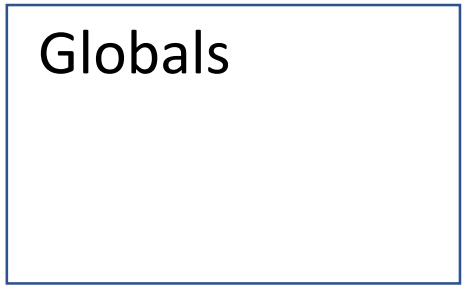
When a program is loaded by an interpreter, it begins with an empty stack and heap. The top-most frame of our stack is called the **Globals frame**.

```
export let main = async () => {
    let a: number[] = [];
    a[0] = 0;
    f(a);
    print(a[0]);
};

let f = (a: number[]): void => {
    a[0] = a[0] + 1;
};

main();
```

The Stack



The Heap