Function Calls, Environment Diagrams, and Expressions

Lecture 03 - Spring 2020

Go ahead and have out your one page of notes for the warm-up questions and open up pollev.com/compunc in preparation.

Tutoring! Tues, Weds, Thu 5pm-7pm

- **What**: Conceptual Help and Practice in Small Groups
  - Working on problem sets is prohibited in tutoring
  - For Problem Set help come to in regular Office Hours (open 11a-6p most days)

- **Where**: Fred Brooks Building 007

- **When**: Tuesdays, Wednesdays, Thursdays from 5pm to 7pm

- **Who**: Tutoring led by senior COMP110 UTAs

Announcements

- WS0 – Release tonight and Due Sunday 1/26
    - It will require concepts from videos 9-11 and what we discuss today.
    - If this is your first time uploading a scanned document to Gradescope through an app on your phone – you should assume the deadline is the day before earlier to give yourself time to figure it out.

Challenge Question 1 - What is the printed output the user of this program will see when it runs?

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

export let main = async () => {
    f(2);

    let x = f(4);
    print("x: " + x);
};

let f = (n: number): number => {
    print(n);
    return n + 1;
};

main();
```

The **`return`** Statement vs. **"Printing"**

- **The `return` statement** is ***for your computer*** to send a result back to the function call's bookmark *within your program.*
  - A bookmark is dropped when you *call* a function with a return type.
    When that function's body reaches a *return statement,* the returned value *replaces* the function call and the program continues on.

- **Printing *is for humans* to see**. To share some data with the user of the program you must *output* it in some way.

- If you have a function f that returns some value, you can print the value it returns by:
  - 1. Printing its return value directly **`print(f())`**, or
  - 2. By storing its return value in a variable and later printing the variable.

Tracing Programs by Hand

- Understanding how a program will evaluate depends on systematically keeping track of many related things.

- As your program is evaluated, there are many moving parts:
  1. The current line of code, or expression within a line, it will process next
  2. The trail of function call bookmarks that led to the current line
  3. The values of all variables and a map of variable "names" to the location of their values

- As a human this quickly becomes more information than you can maintain in your head.
  - Good news: Environment diagrams will help us keep track of these things.

Environment Diagrams

- A program's runtime *environment* is the mapping of *names* in your program to their *locations* in memory.
- A program's *state* is made up of the *values stored* in those locations.
- You can use *environment diagrams* to visually keep track of both the *environment* and its *state*.
- Additionally, *environment diagrams* will help you keep track of how function calls are processed.
- In the 2018-2019 academic year we began teaching Environment Diagrams
  - On the final exam, students who used environment diagrams to trace code were over 50% less likely to make errors than students who did not.

Environment Diagram

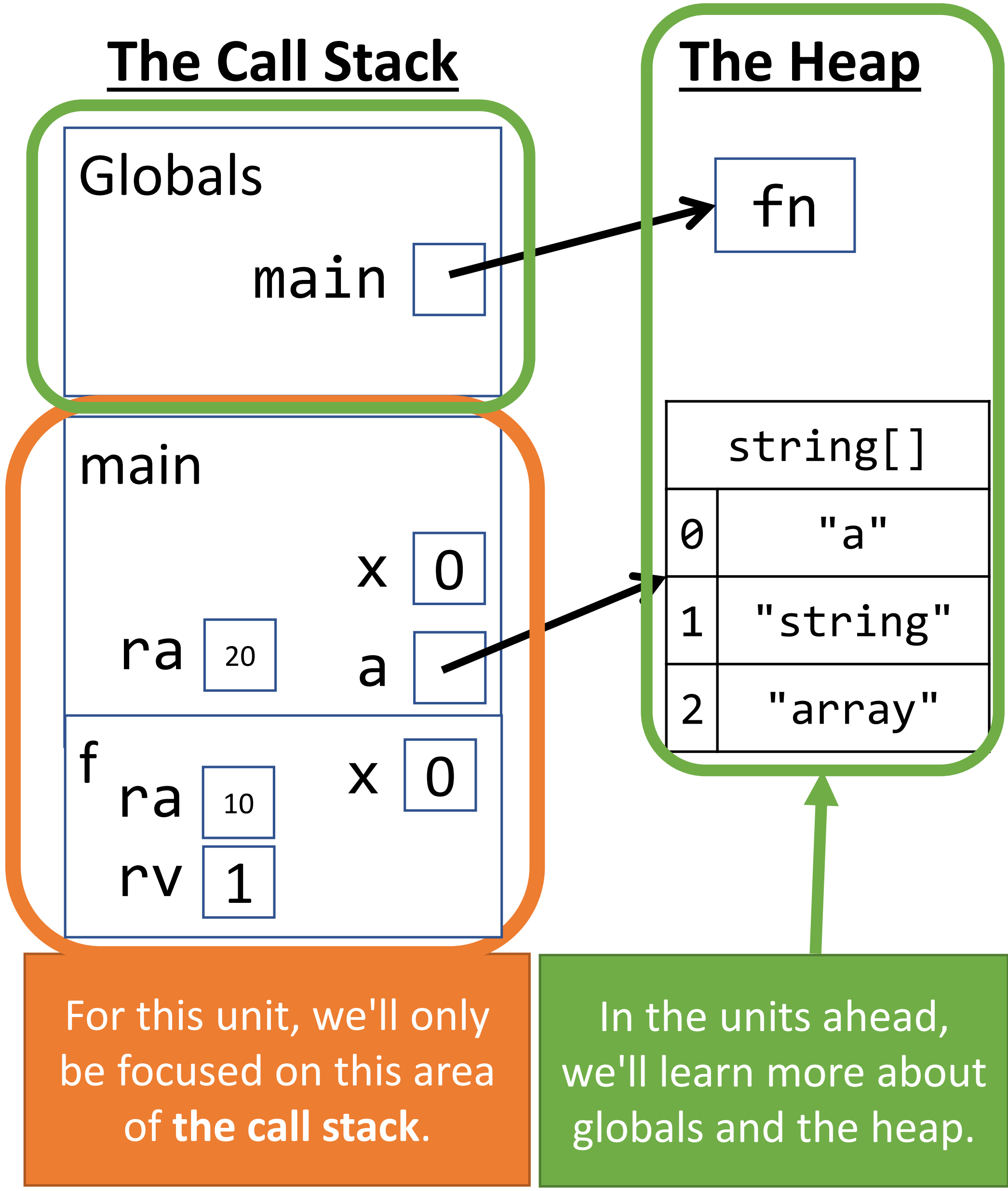
- There are two areas of an environment diagram:

1. Call Stack (or **"The Stack"**)
   - 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**
     - Variable values are stored in stack frames
     - A place to represent its return value (**rv**) when it returns.

2. Dynamic Memory Heap (or **"The Heap"**)
   - We'll come back to this in the next unit.

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

Example:



The Call Stack
The Heap
Globals
main
fn
main
x
0
ra
20
a
f
ra
10
x
0
rv
1
string[]
0
"a"
1
"string"
2
"array"
For this unit, we'll only be focused on this area of **the call stack**.
In the units ahead, we'll learn more about globals and the heap.

Environment Diagram Example

```
export let main = async () => {
  let x = 4;
  let y = f(x);
  print("y: " + y);
};

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

main();
```

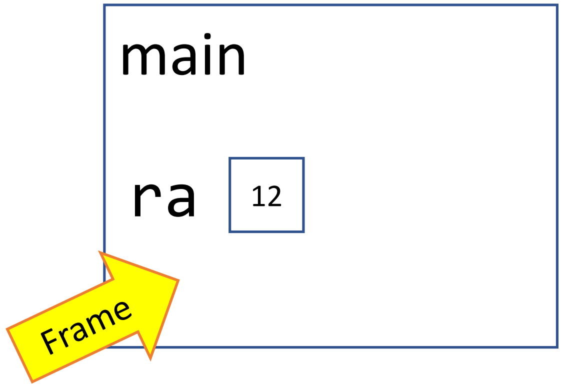
- Let's trace the example to the left using an environment diagram!
- In the process you will learn how to:
  - Establish a frame for **main**
  - Establish **local** variables (those declared *inside* of a function's body) in the frame
  - Call functions
    - Establish a **frame** for the function
    - Establish **parameters** as local variables, assigned their **argument's** values
    - Keep track of the value returned by a function call

Function Call - `main`

When a function call* is encountered, a new **frame** is added to your stack. Label it with the function's name. Add a "Return Address" entry with the line # of call.

```
01 export let main = async () => {
02   let x = 4;
03   let y = f(x);
04   print("y: " + y);
05 };
06
07 let f = (n: number): number => {
08   let x = n + 1;
09   return x;
10 };
11
12 main();
```

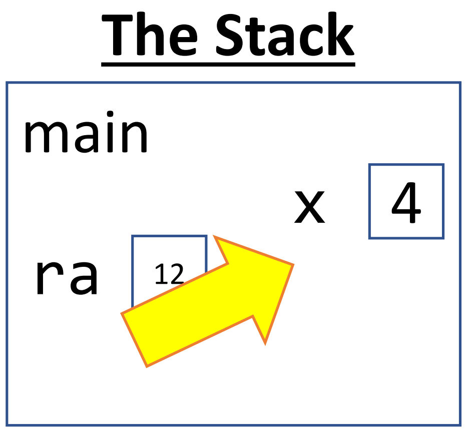
The Stack



Variable Declaration and Initialization

When a variable is declared and initialized *first* evaluate the value on the right. In this case it's the number literal 4, no more work is needed. Then, establish it in the current frame.

```
export let main = async () => {
  let x = 4;
  let y = f(x);
  print("y: " + y);
};

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

main();
```

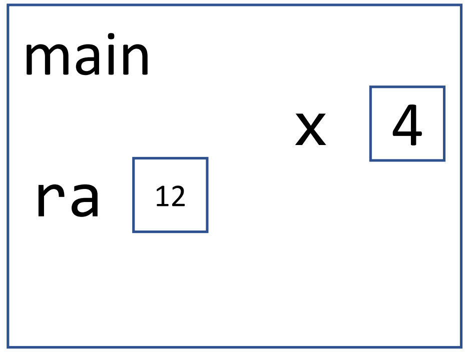


Variable Declaration and Initialization

When a variable is declared and initialized *first* evaluate the value on the right. In this case it's a function call, so let's evaluate what the function call will return first.

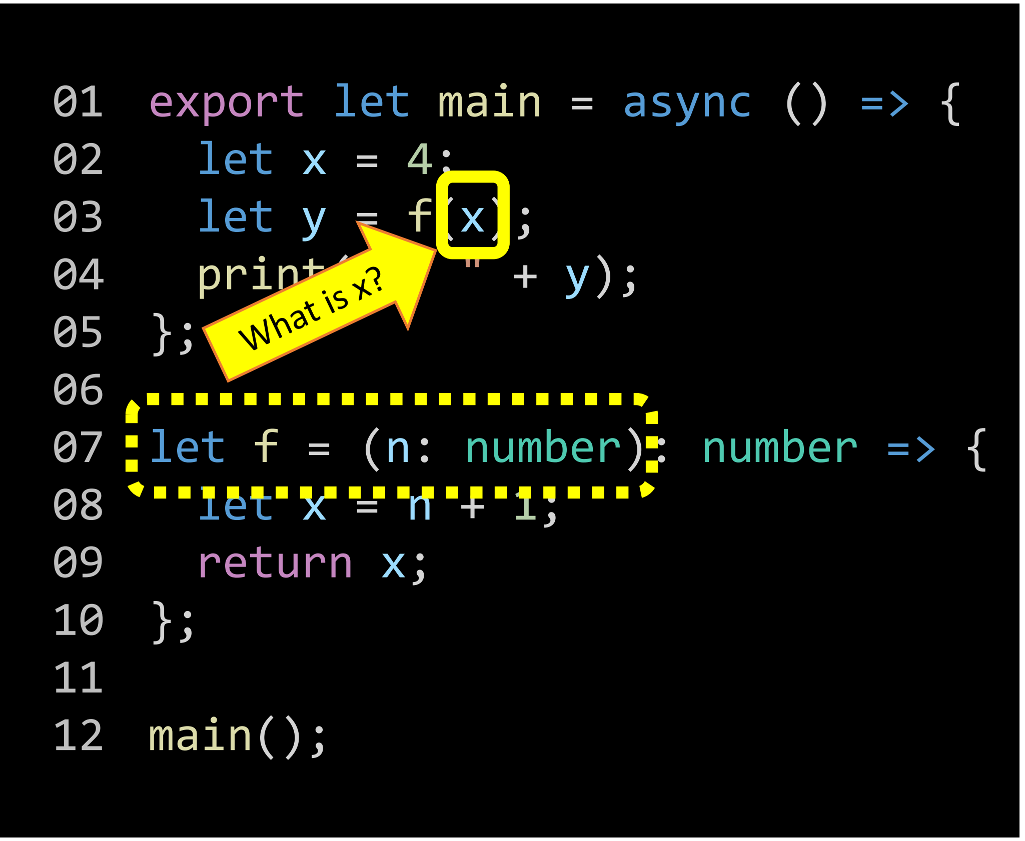
```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack



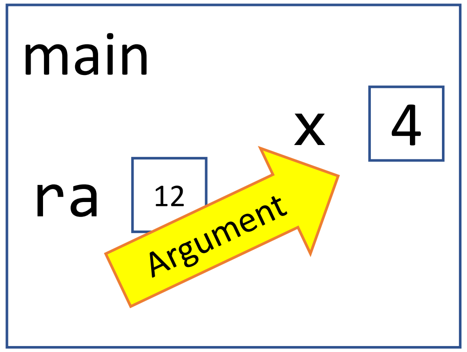
Function Call - Step 1) Evaluate Arguments

Before evaluating the function call to f, we must determine the values of each argument. What is the name **`x`** bound to in **`main`**'s frame? We look in our diagram to see its value is 4. Since it is an argument, we will copy this value to the corresponding parameter **n**.



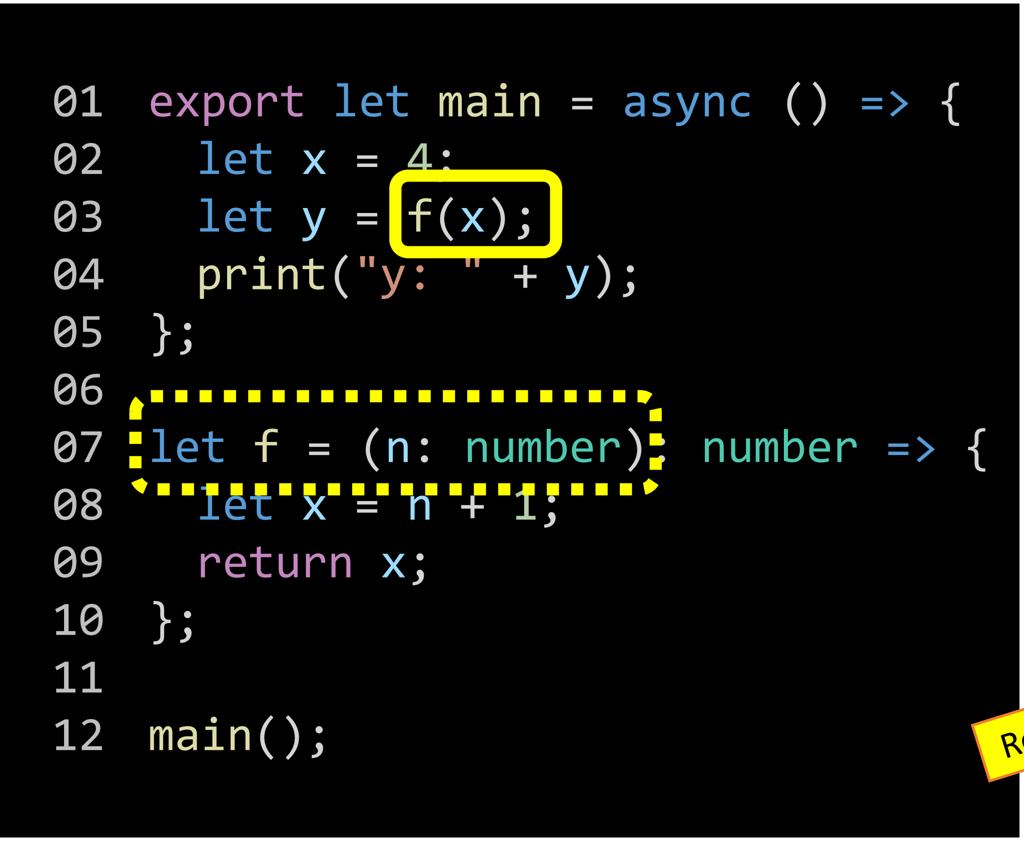
```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print([illegible]" + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack

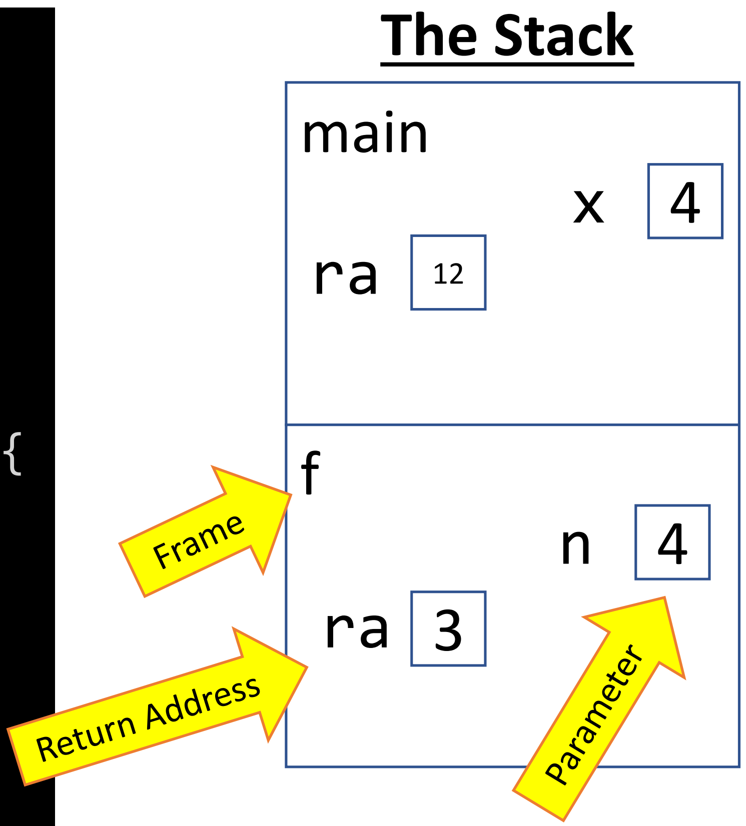


Function Call - Step 2) Establish a Frame

1. Give the frame the function's **name**. 2. Write down the line the function call occurred on as the frame's **Return Address (RA)**. 3. Assign argument values to the function's **parameters**.



```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

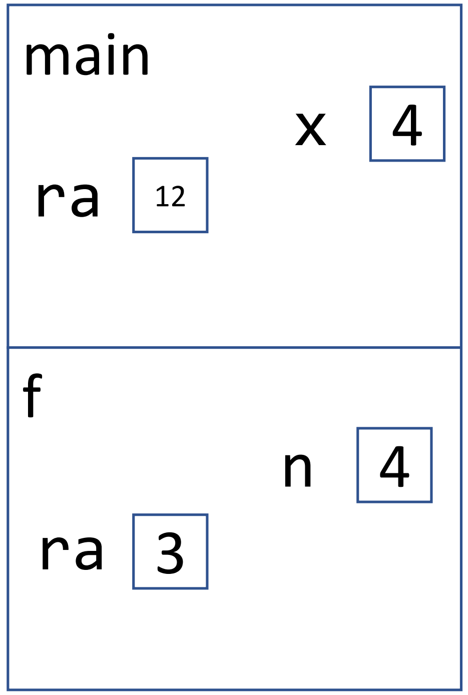


Function Call - Step 3) Jump to Function

The return address for frame f tells us we'll return back to line 3 once the call to f completes. Now we're ready to jump in to the first line of the function.

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack

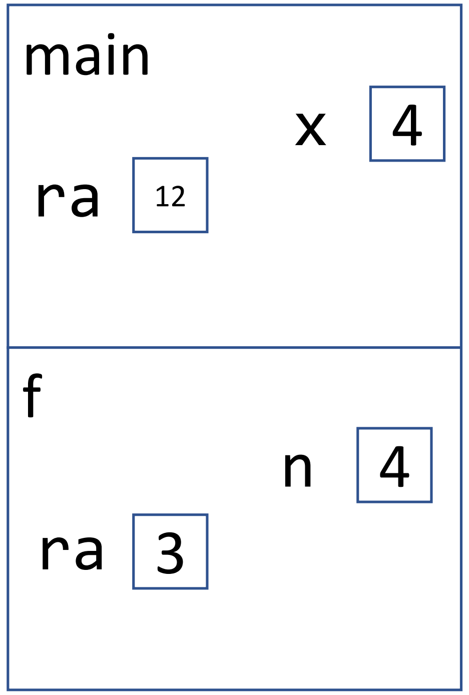


Variable Declaration and Initialization

When a variable is declared and initialized *first* evaluate the value on the right. In this case it's an arithmetic expression. Let's focus on it.

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack

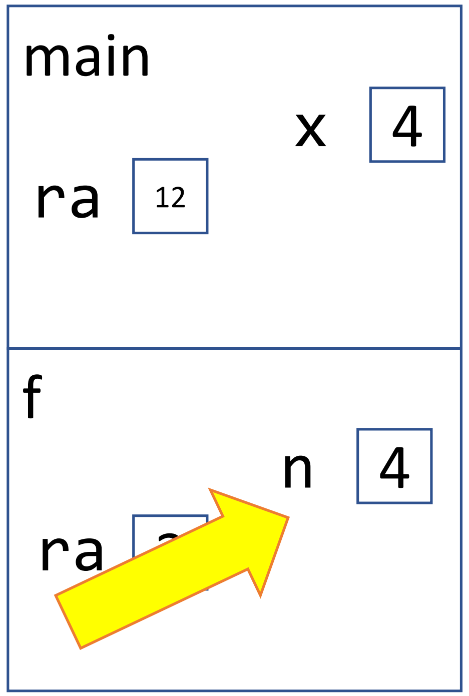


Name Resolution: What is *n*?

When a name is encountered in our program we look to the current frame of the stack for its value. In this case, **n**'s value in **f**'s frame is bound to **4**. The expression **4 + 1**, then, is **5**.

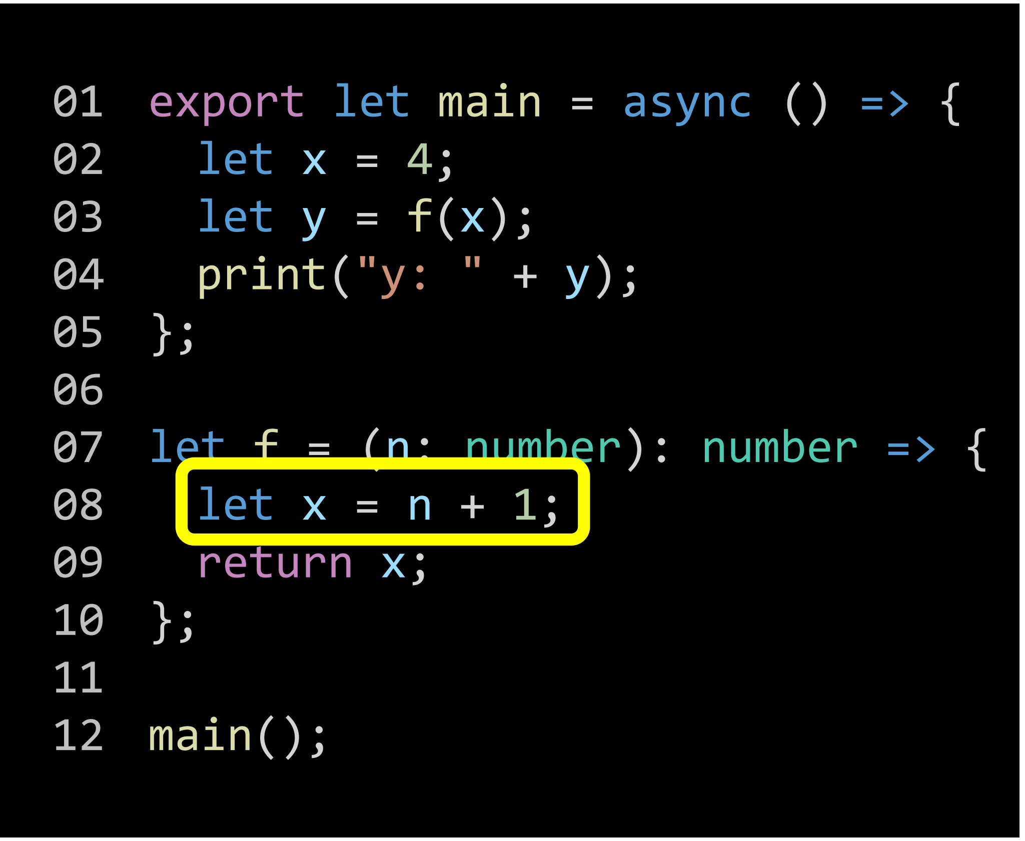
```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack

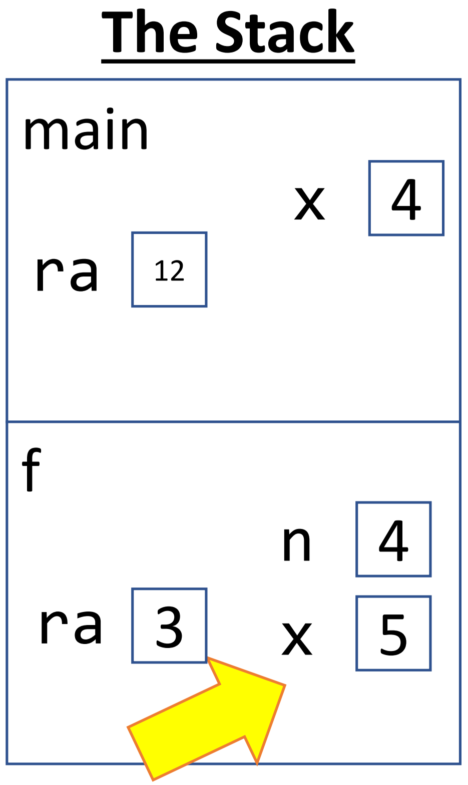


Variable Declaration and Initialization

Now that we've evaluated the right hand side, we add an entry for the newly declared variable **x** to the current frame for **f**. ***Notice, there are two separate values of x in our program!***



```
export let main = async () => {
  let x = 4;
  let y = f(x);
  print("y: " + y);
};

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

main();
```



Notice the frame for *main* has its own variable x with a value of 4.

The frame for *f* also has its own variable x with a different value.

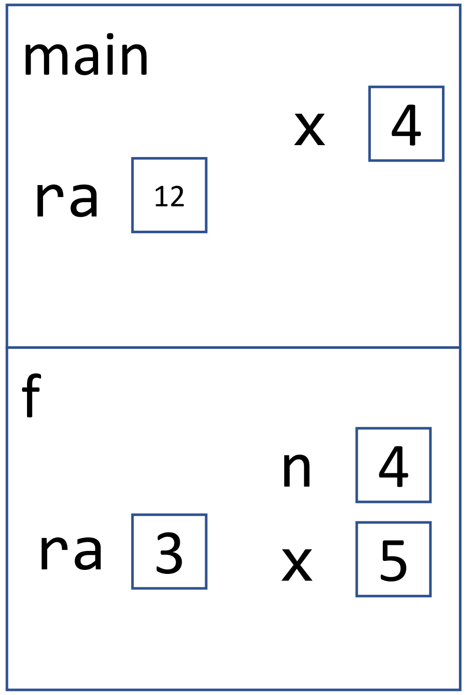
This is *entirely ok* and a *wonderful, powerful thing.* This means when you write functions you don't need to concern yourself with the variable names in other functions.

Return Statement - Step 1) Evaluate its Value

When a return statement is encountered, you must first evaluate the value it is returning. Let's focus on evaluating **x**.

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack

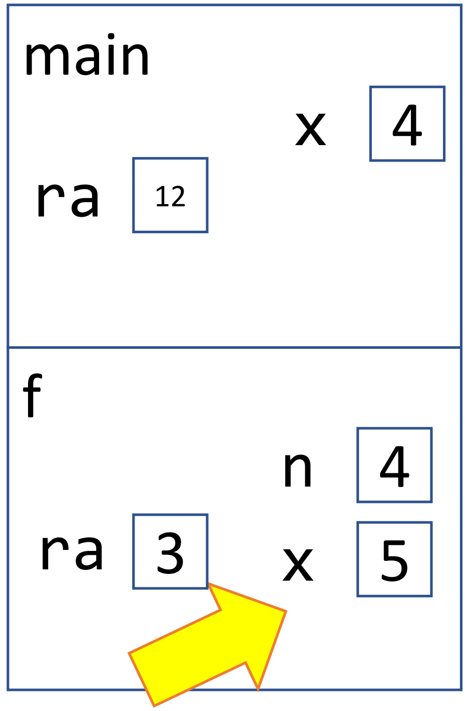


Name Resolution: What is *x*?

When a name is encountered in our program we look to the current frame of the stack for its value. In this case, **x**'s value in **f**'s frame is bound to **5.**

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack

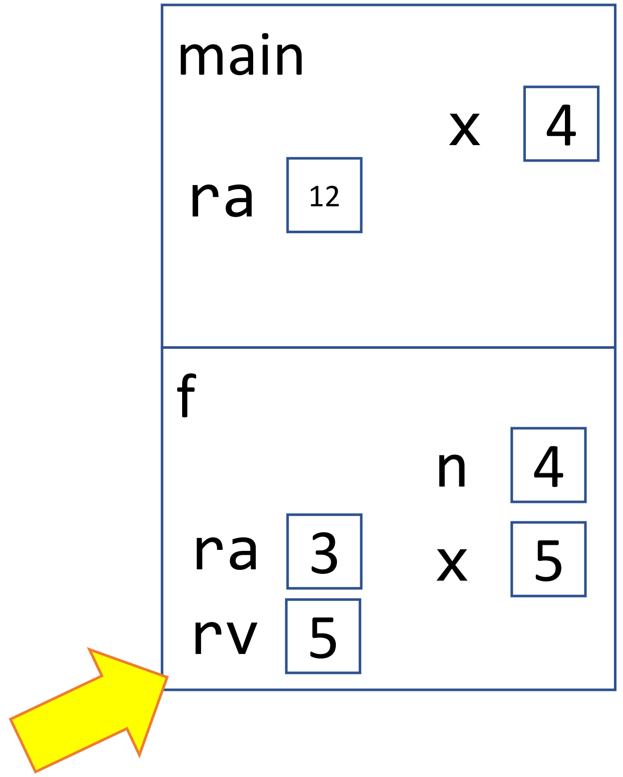


Return Statement - Step 2) Record its Value

When a return statement is encountered, once you know the value, enter the return value in a box named ***rv*** in the current frame.

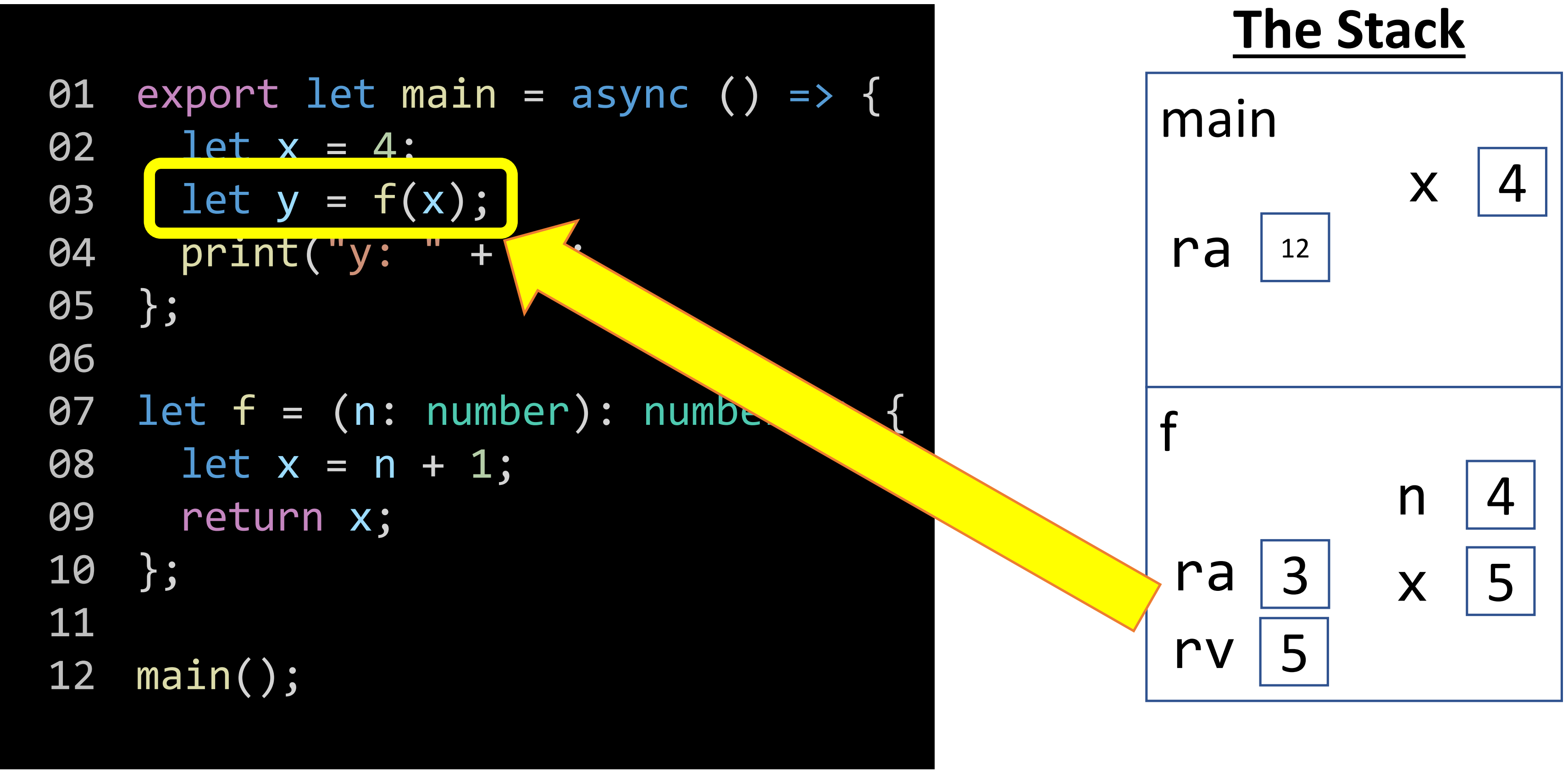
```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack



Return Statement - Step 3) Send value back to RA

The return value is then *returned* to the return address where the call originated. Its value will be substituted for the function call. So back in main, this line is evaluated as **`let y = 5;`**



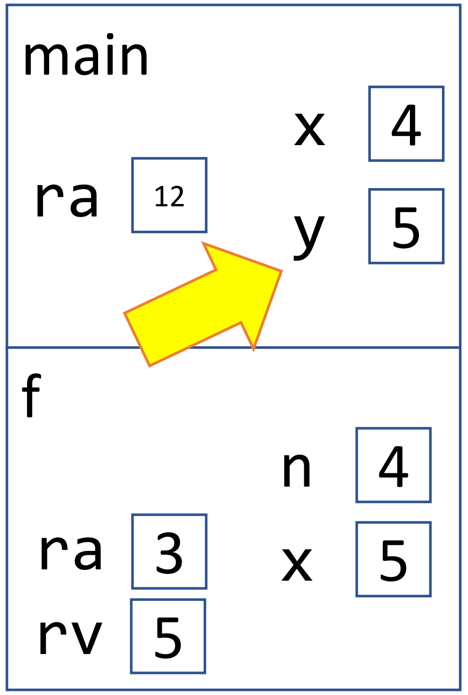
```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " +
05  };
06
07  let f = (n: number): numbe        {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

Variable Declaration and Initialization

Now that we've evaluated the right hand side, we add an entry for the newly declared variable **y** to the current frame **main**.

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack



How can you tell what the **current frame** of execution is?

The current frame is always the *lowest* frame that *has not returned*. So, if a frame has an *rv* entry, that frame is ignored.

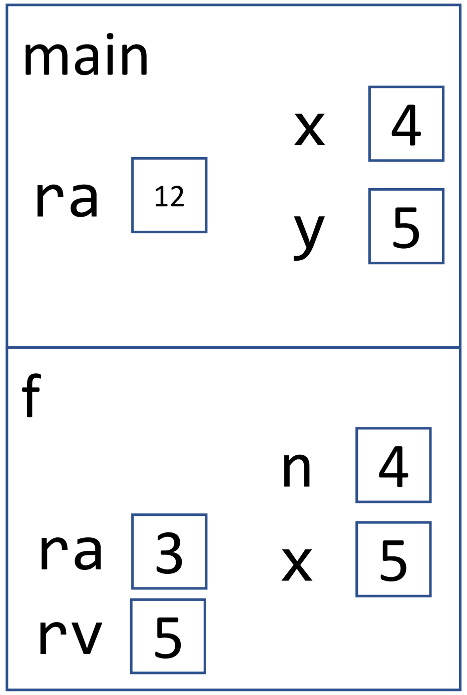
Behind the scenes in your computer, once a function call returns its *environment* and *state* are erased. When working on paper, though, it is helpful to keep track of all the work it took to arrive at a given position in our program.

Print Function Call

Technically a call to a function like print will *also* add a frame to the stack and go through the same series of steps. For functions defined outside of our code, though, we'll skip that.

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack

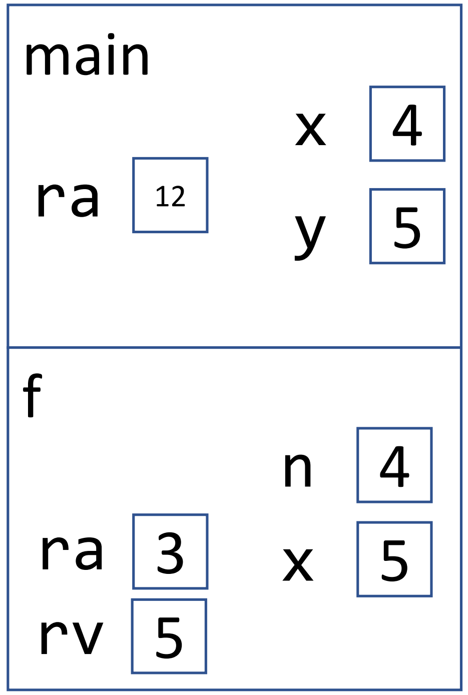


Name Resolution: What is *y*?

When a name is encountered in our program we look to the current frame of the stack for its value. In this case, **y**'s value in **main**'s frame is bound to **5.**

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack

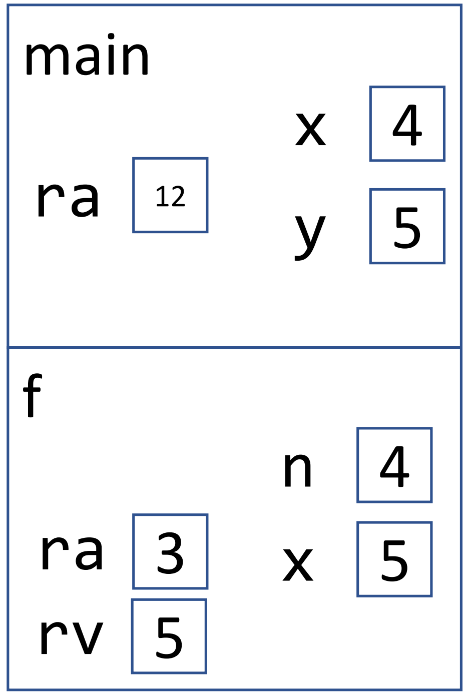


Printed Output

When a print statement appears, you'll evaluate its argument as we just did, and record its output. Remember, this is output for the person running the program to see. The rest of what happened in our environment diagram was for the computer's purposes only.

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack



Output

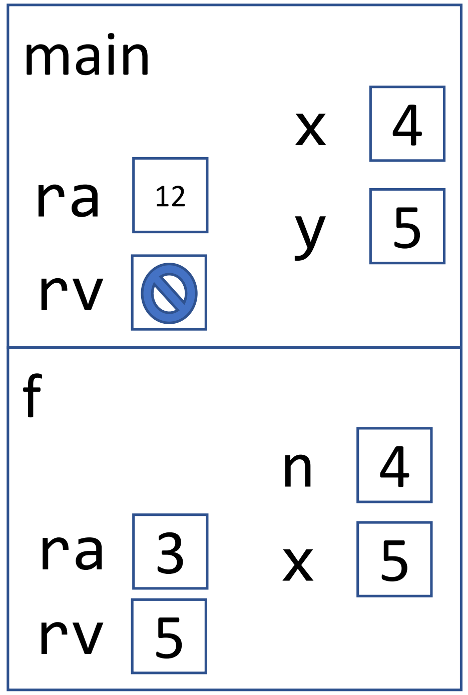
```
y: 5
```

End of `main`

When our program reaches the end of the *main* function you'll notice it has no *return statement.* We'll talk more about these kinds of functions next week. For now, expect its return value is *nothing.* The processor would jump back to the return address at line 12 and reach the end of the program.

```
01  export let main = async () => {
02    let x = 4;
03    let y = f(x);
04    print("y: " + y);
05  };
06
07  let f = (n: number): number => {
08    let x = n + 1;
09    return x;
10  };
11
12  main();
```

The Stack



Output

y: 5

CQ#2 - What is the printed output? Try diagramming!

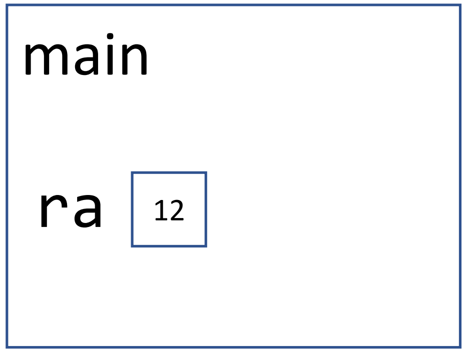
```
export let main = async () => {
    let x = 0;
    f(x + 1);
    print("x: " + x);
};

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

main();
```

Call to `main`

Your code begins when the last line of the file reaches the call to the **main** function. This establishes the frame for main.

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

The Stack



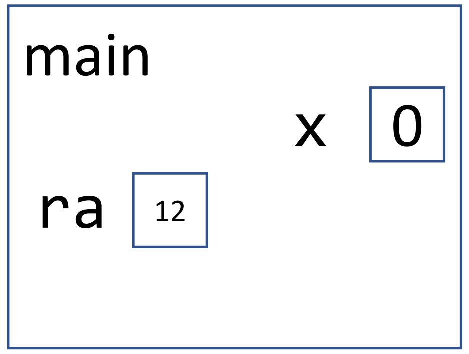
Output

Variable Declaration and Initialization

The right hand side of this initialization is a constant so we can establish the variable in the current frame of the stack directly.

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

The Stack



Output

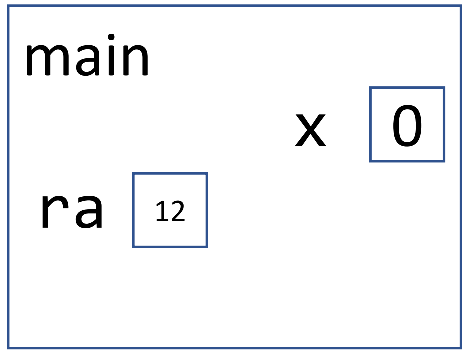
Function Call - Step 1) Evaluate Arguments

Using *name resolution* we lookup the name *x* in *main's* frame and see that its value is 0.
0 + 1 means the value for the **x** parameter of the call to **f** will be assigned the argument value **1**.
Important: The value of x in main's frame did not change because we *did not assign* a new value to x.

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

1

The Stack

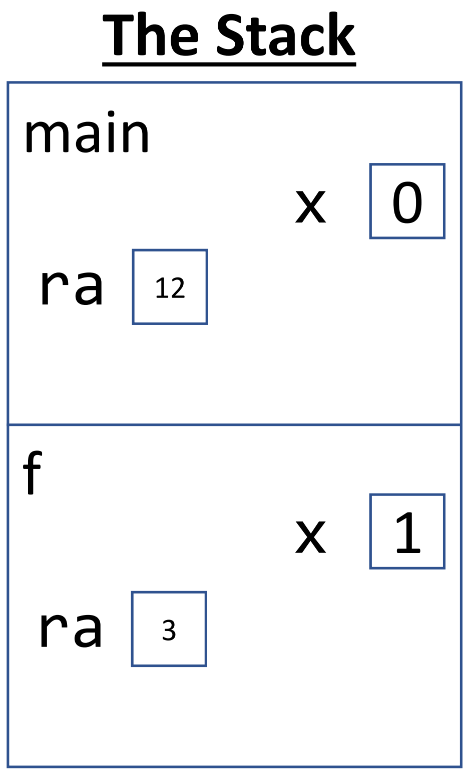


Output

Function Call - Step 2) Establish a Frame

1. Give the frame the function's **name**. 2. Write down the line the function call occurred on as the frame's **Return Address (RA)**. 3. Assign the evaluated argument values to the function's **parameters**.

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



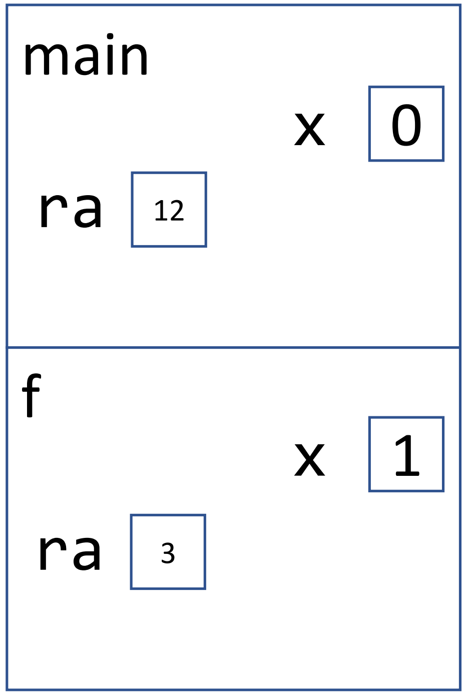
Output

Function Call - Step 3) Drop a Bookmark, Jump to Function

We know we'll have to return back to where the function call occurred, so leave a bookmark.
Then, jump in to the first line of the function.

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

The Stack



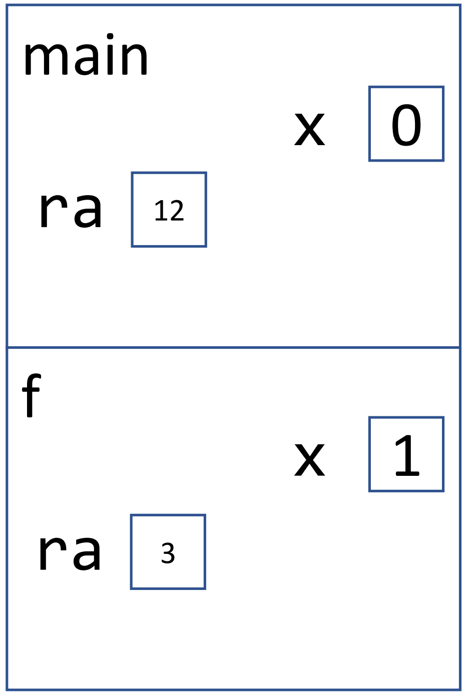
Output

Variable Assignment

When an assignment statement is encountered, we must evaluate its right-hand side first.

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

The Stack



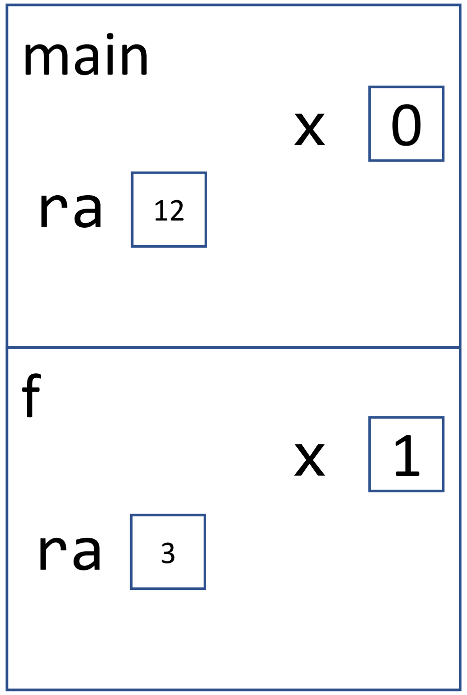
Output

Name Resolution

What is **x**'s value? We look in the *current stack frame* which is the lowest frame that hasn't returned and see that x's value is 1. So, the right-hand side evaluates to 2.

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

The Stack



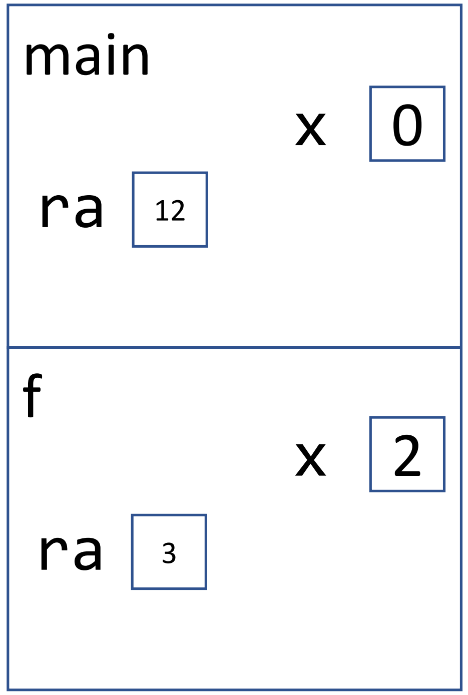
Output

Variable Assignment

When assigning to a variable, name resolution rules once again apply. Which **x** are we assigning to? *The one in the current stack frame!*

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

The Stack



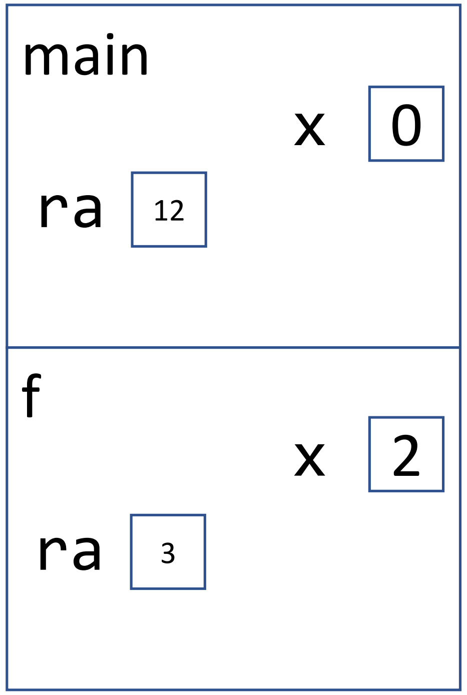
Output

Return Statement - Step 1) Evaluate its Value

When a return statement is encountered, you must first evaluate the value it is returning. Let's focus on evaluating **x**. We look in the current frame and see its value is 2.

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

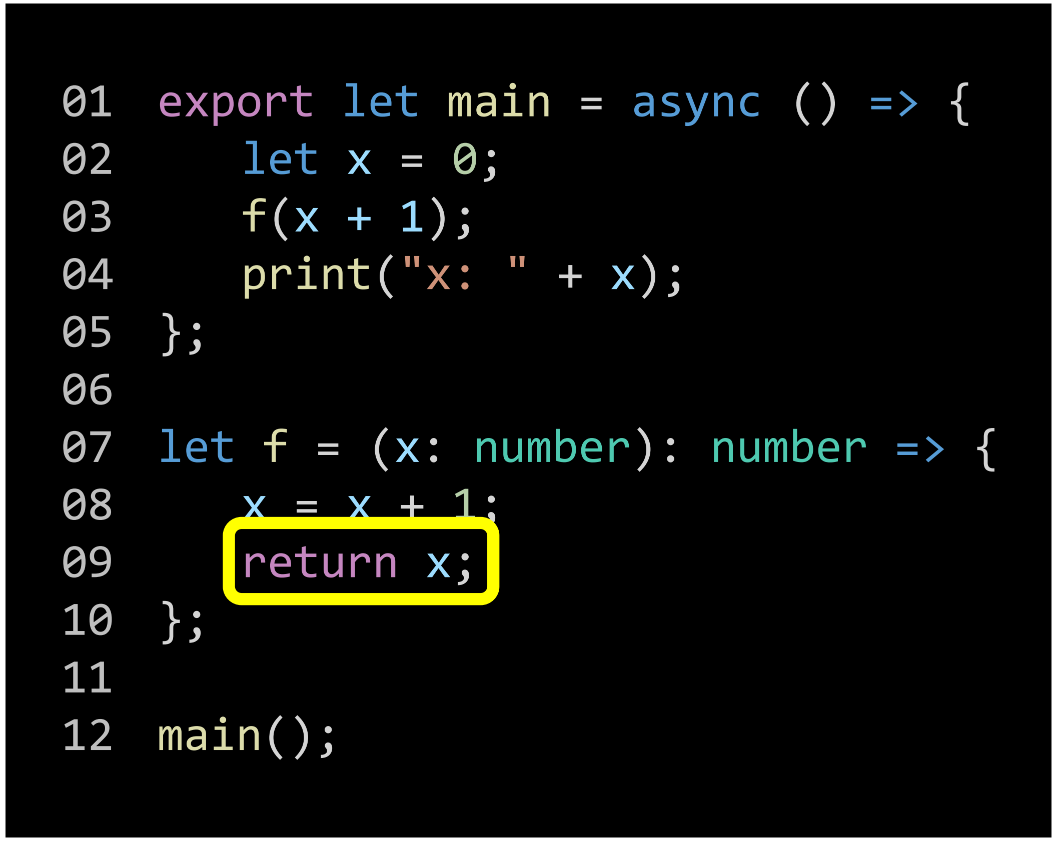
The Stack



Output

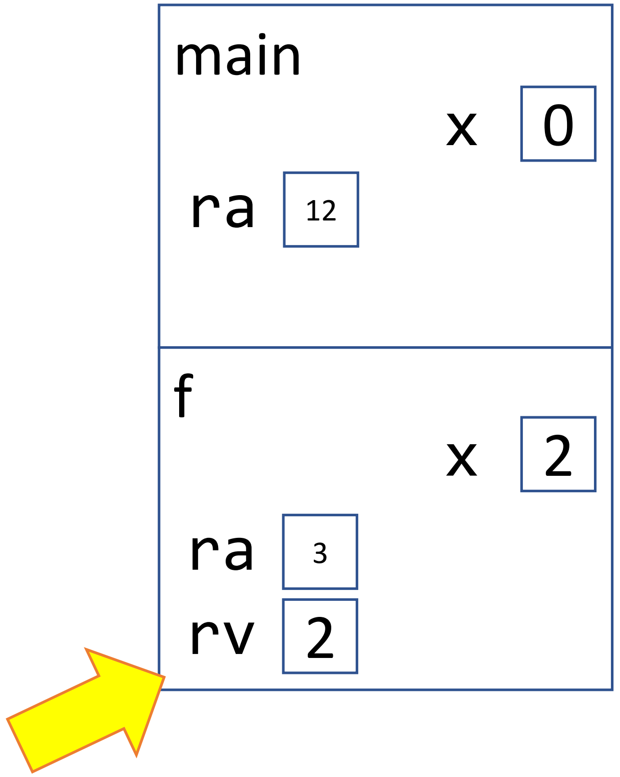
Return Statement - Step 2) Record its Value

When a return statement is encountered, once you know the value, enter the return value in a box named ***rv*** in the current frame.



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

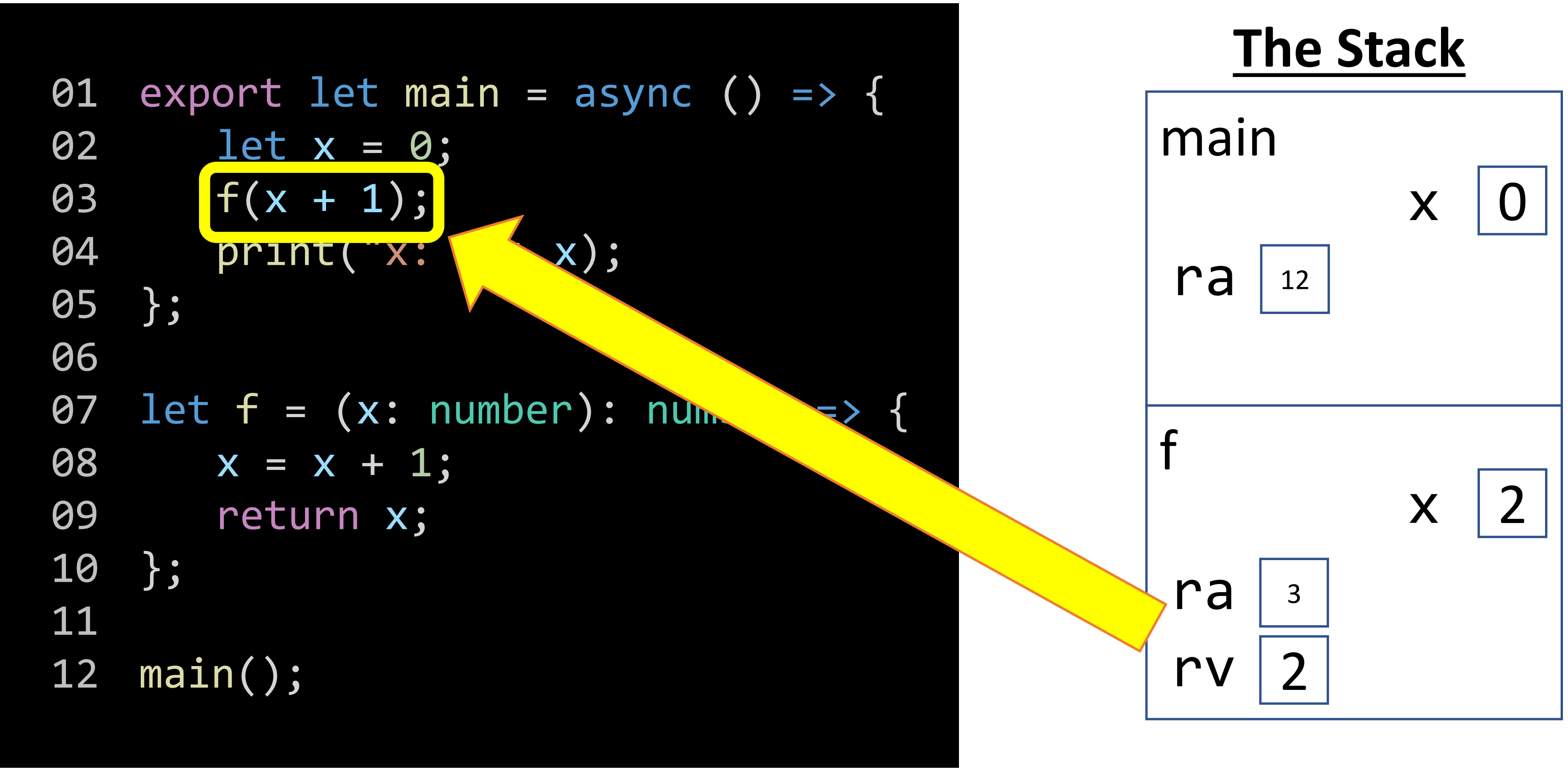
The Stack



Output

Return Statement - Step 3) Send value back to Bookmark

The return value is then *returned* to where the function call originated. Its value will be substituted for the function call. So back in main, this line is evaluated as **`2;`**



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

Notice if we simply write the line of code: **`2;`**

It would not *change the value* of anything else. We're also not printing it. We're not *doing anything* with the value returned by this function call.

"If a function call occurs in the woods and there's no assignment statement there to receive its return value, did it really get called?" *(Yes, actually.)*

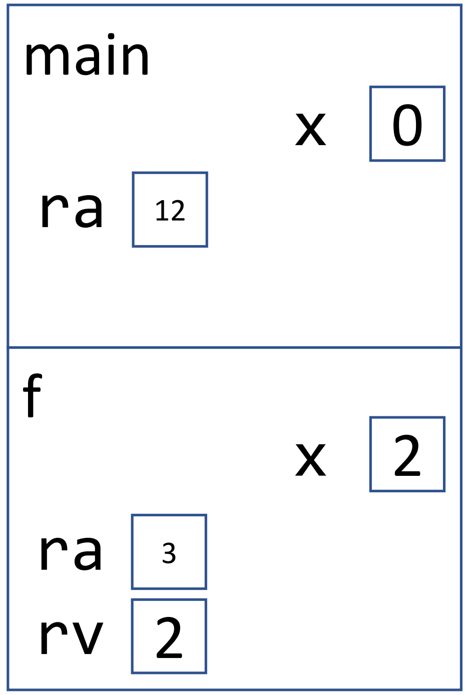
This example would be a *silly* thing to do, however, it illustrates a surprising, important concept.

Print Statement

We need to evaluate the argument sent to the print statement before we know what is output.

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

The Stack

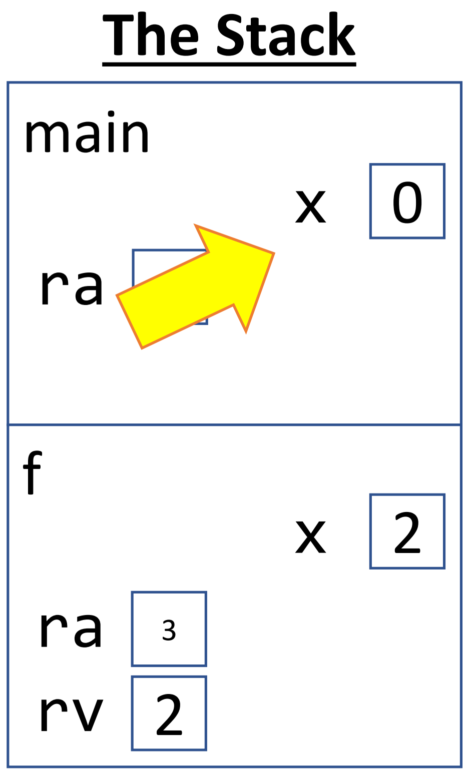


Output

Variable Access / Name Resolution

What is the value of the variable x? We look in the *current frame on the stack* which is main's. Its value is 0.

```
export let main = async () => {
    let x = 0;
    f(x + 1);
    print("x: " + x);
};

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

main();
```



Output

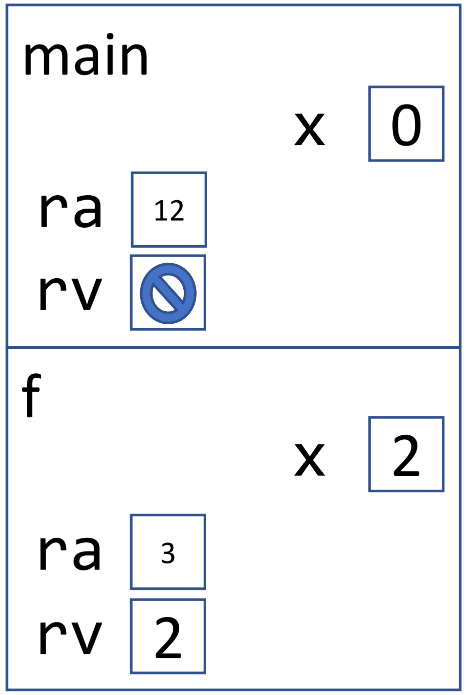
x: 0

End of Main

When our program reaches the end of the *main* function you'll notice it has no *return statement.* Its return value is *nothing.* The processor would jump back to `main`'s return address at line 12 and reach the end of program.

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

The Stack



Output

x: 0

Expressions

- **Expressions** are a fundamental building block in programs

- Expressions are analogous to the idea of clauses in English
  - Single clause sentence:
    "I am a student."
  - Multiple clause sentence:
    "I am a student and I am currently sitting in COMP110."
  - In English, *Sentences* are *more expressive* through the creative use of *clauses*

- In code, ***statements*** are *more expressive* through creative uses of ***expressions!***

How can we compute the volume of a cube using different expressions?

```
let answer: number;
answer = 3 * 3 * 3;
```

1. We can "hard-code" the expression with exact numbers.

How can we compute the volume of a cube using different expressions?

```
let answer: number;
let length = 3;
answer = length * length * length;
```

2. We can use a variable to hold the length of a side of the cube.

Notice, in doing so, our *expression* has more meaning:
`length * length * length` is more expressive than `3 * 3 * 3`

How can we compute the volume of a cube using different expressions?

```
let answer: number;
let length = await promptNumber("Length:");
answer = length * length * length;
```

3. We can use the **promptNumber** function to allow *any number*!

Our program is more generally useful.

How can we compute the volume of a cube using different expressions?

```
let cubeVolume = (side: number): number => {
   return side * side * side;
};
```

```
let answer: number;
let length = await promptNumber("Length:");
answer = cubeVolume(length);
```

4. We can write a *function* to compute the volume and *call the function*.

This has two benefits:

1. It reads more naturally: "answer is assigned the result of calculating `cubeVolume` using the given `length`"
2. We can *reuse* the `cubeVolume` function without rewriting the equation!

Expressions

There are two **big ideas** behind expressions:

1. *Every* expression *simplifies to a single value at runtime*
    - Thus, every expression has a *single type*.
    - This occurs *only* when the program runs (runtime) and when the processor reaches the expression in the program.

2. Anywhere you can write an expression you can substitute any other expression *of the same type*

Where have we *used* expressions?

- Assignment operator:

```
let <name>: <type> = <expression of same type>;
```

- We are able to assign *any* of the expressions below because each results in a single *number* value:

```
let x: number = 1;
let y: number = x + 1;
let cubeY: number = y * y * y;
```

- Notice that we are combining *multiple* expressions in the same line.
- After each line completes, the declared variable has a *single* value.

Where else have we *used* expressions?

- if-then statement

```
if (<boolean expression>) {
  // ... elided ...
}
```

- ***Any boolean expression*** can be used as the test expression in an if-then statement
  - `if (age >= 21) { // ...`
  - `let is21 = (await promptNumber("Age")) >= 21;`

    `if (is21) { // ...`
- When the computer reaches the boolean expression of an if-then statement, it evaluates the expression down to the single value of either **`true`** or **`false`**.

Expressions of Various Kinds

- Literal Values
  - `3.14`
  - `true`
  - `"hi"`
- Variable Access
  - `x`
  - `compCourseNumber`
- "Unary" operators
  - `-x` (number *negation*)
  - `!is21` (boolean negation)
- Function Calls
  - `cubeVolume(x)`

- "Binary" Operators
  - Arithmetic
    - `1 + 2`
  - Concatenation
    - `"Hello " + name`
  - Equality
    - `x === 1`
    - `x !== 1`
  - Relational
    - `age >= 21`
    - `age < 13`

Challenge Question #3 - What input at the prompt would cause "C" to print?

```
let x = await promptNumber("Enter a value for x");
if (x < 18) {
     print("A");
} else {
   if (x > 13) {
     print("B");
   } else {
     print("C");
   }
}
```

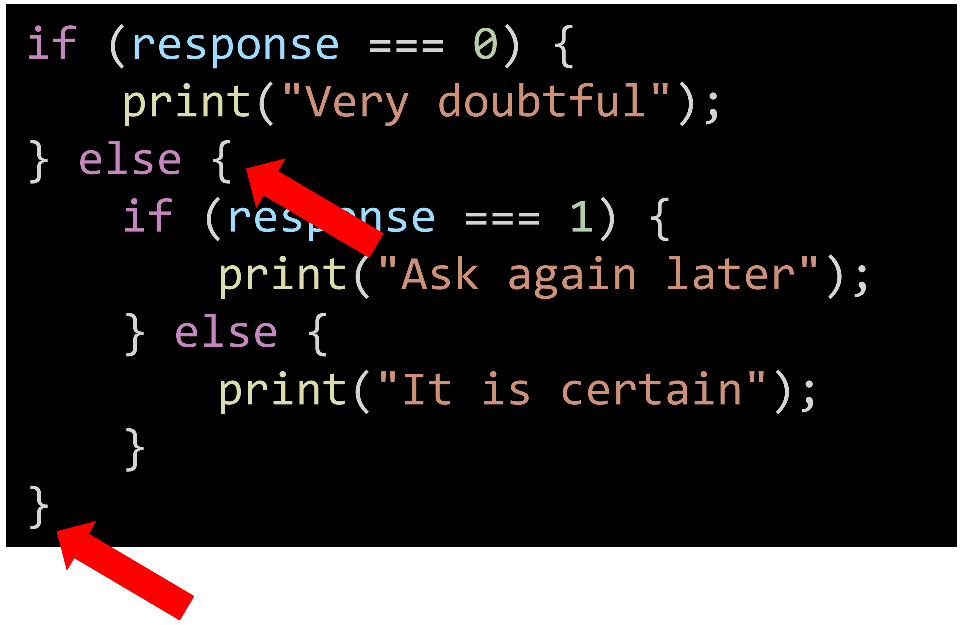
Pattern: Nesting **if-then** in an **else** Pattern

- It is commonly useful to nest additional if-then-else statements inside of subsequent else-blocks

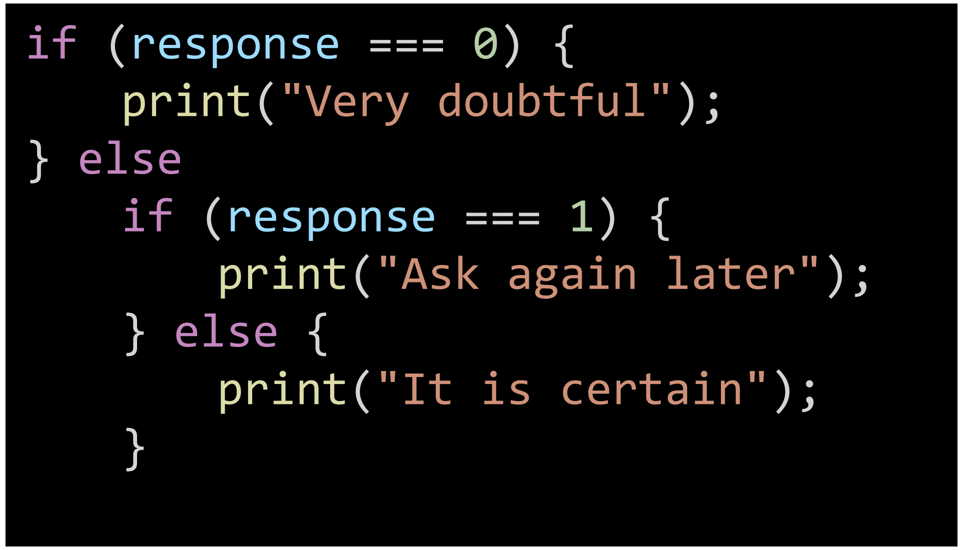
- Why? It allows us to choose one next step from many possible options.
  - "If *this* then do X, otherwise if *that* do Y, *otherwise* do Z."

```
if (response === 0) {
    print("Very doubtful");
} else {
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
}
```

This is so common and useful, we tend to use simpler syntax for it…



```
if (response === 0) {
    print("Very doubtful");
} else {
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
}
```



```
if (response === 0) {
    print("Very doubtful");
} else
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
```

1. First we remove the curly braces surrounding the if-then that is nested inside of the else-block.

This is so common and useful, we tend to use simpler syntax for it...

```
if (response === 0) {
    print("Very doubtful");
} else
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
```

→

```
if (response === 0) {
    print("Very doubtful");
} else if (response === 1) {
    print("Ask again later");
} else {
    print("It is certain");
}
```

2. Then we clean up the spacing.

Using the **else-if** pattern is a change of *style* only.
These two listings of code have the ***exact same logic***.

```
if (response === 0) {
    print("Very doubtful");
} else {
    if (response === 1) {
        print("Ask again later");
    } else {
        print("It is certain");
    }
}
```

```
if (response === 0) {
    print("Very doubtful");
} else if (response === 1) {
    print("Ask again later");
} else {
    print("It is certain");
}
```

Notice the code is visually simpler and cleaner by using else-if.

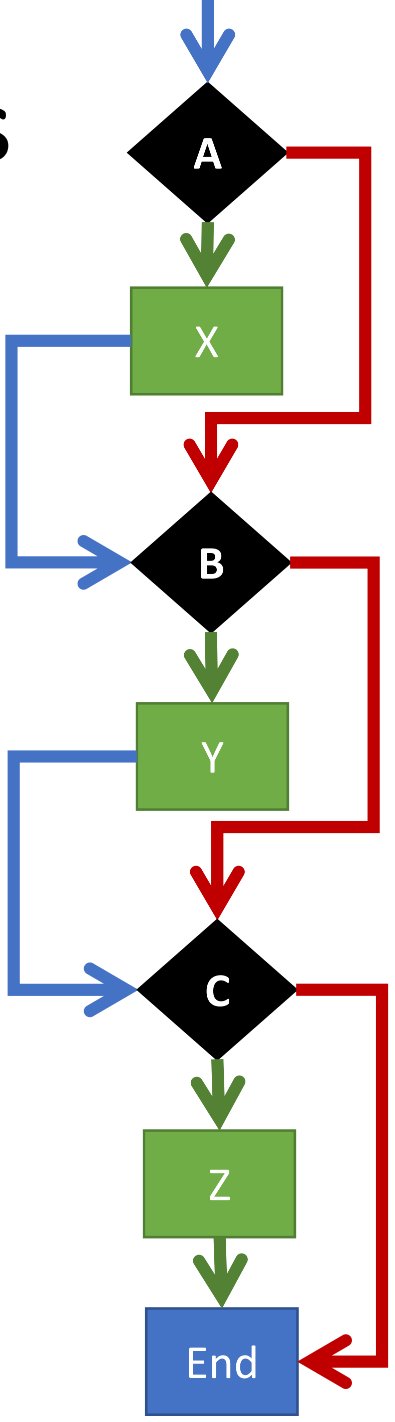
Follow-Along) Using the else-if Syntax Pattern

- Still in 8-ball-app.ts
- Reformat the conditional logic to use the else-if syntax pattern.
- Step 1) Remove the curly brace directly following the *first* else and its matching closing curly brace.
- Step 2) Clean up the spacing by bringing the nested if to directly follow else and unindenting.
- Check-in when complete! pollev.com/compunc

Many, independent `if-then-else` statements

- When two or more if-then-else statements are *not* nested, they are independent statements of one another.
- Each `boolean` test expression will be evaluated.
- Notice in the diagram that there is a path through *every* block X, Y, Z.

```
if (A) {
  print("X");
}

if (B) {
  print("Y");
}

if (C) {
  print("Z");
}

print("End");
```



Tracing through `else-if` statements

- The previous slide does not apply to else-if statements *because…*
  - An else-if is a nested if-then
  - It is nested in the else-block

- Each `boolean` test expression will be evaluated **<u>until one evaluates to true</u>**. The rest are then skipped.

- Notice in the diagram that there is a path through *only one* outcome X, Y, Z.

- Useful when there are many possible next steps but you only want to choose one.

```
if (A) {

    print("X");

} else if(B) {

    print("Y");

} else if(C) {

    print("Z");

}

print("End");
```